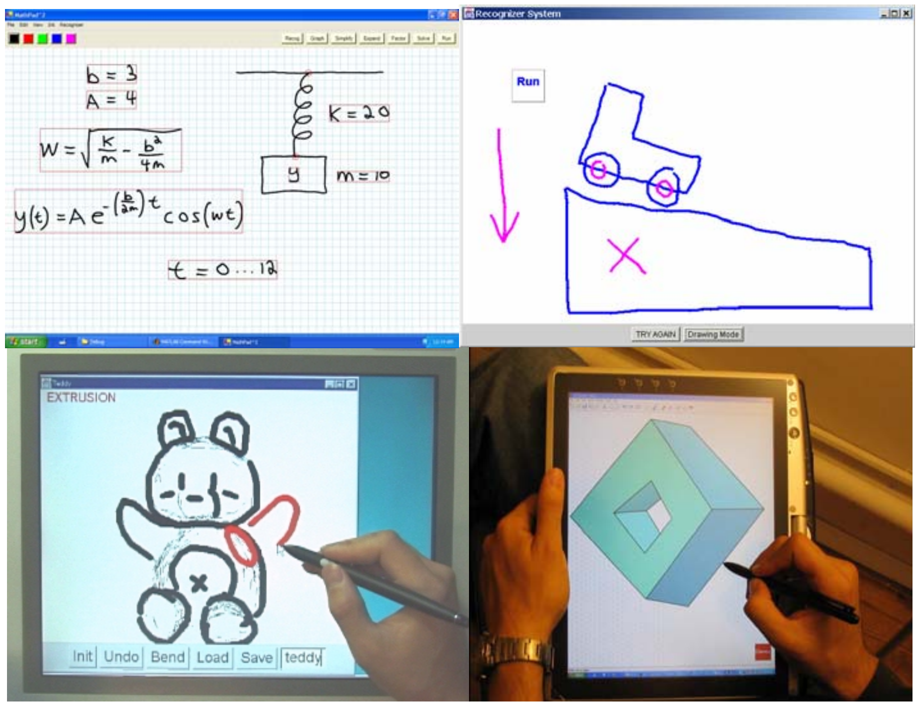
Sketch-Based Interfaces: Techniques and Applications

SIGGRAPH 2007
August 5, 2007
Course Notes



Joseph J. LaViola Jr.
University of Central Florida

Takeo Igarashi
University of Tokyo

Christine Alvarado
Harvey Mudd College

Hod Lipson
Cornell University

Course Overview

Sketch-based interfaces offer a natural method of interaction with computer applications because they mimic traditional pencil and paper. The desire for this form of interaction has been around since the 1960s, but has only recently begun to be possible. Today, sketch-based interfaces are starting to be used in many different applications such as modeling, animation, user interface prototyping, music composition, and object oriented design. To many, these interfaces often seem “magical” in nature. Thus is important for anyone interested in working with and developing sketch-based interfaces to have a firm understanding of the techniques and methodologies for creating them.

In this course, the presenters will demystify the workings of sketch-based interfaces by exploring how they are developed and examining their internal components. We will discuss a variety of different sketch-based interface styles along the sketch-input continuum ranging from gestural command systems for application control to sophisticated sketch understanding systems that rely on techniques from pattern classification, 2D parsing, and machine learning. Attendees will receive a thorough understanding of what types of sketch-based interfaces exist and what tools and strategies are needed to develop their own.

Topics will include:

- the sketch-input continuum
- sketch-based applications
- gestural command systems
- modeless gestural user interfaces
- gestures used in interactive computer graphics
- multi-domain sketch understanding
- sketching for mechanical design and CAD
- sketching freeform surfaces
- creating geometry from sketches
- mathematical sketching

Speaker Biographies

Joseph J. LaViola Jr. is an assistant professor in the School of Electrical Engineering and Computer Science at the University of Central Florida as well as an adjunct assistant research professor in the Computer Science Department at Brown University. His primary research interests include pen-based interactive computing, 3D interaction techniques, predictive motion tracking, multimodal interaction in virtual environments, and user interface evaluation. His work has appeared in journals such as IEEE PAMI, Presence, and IEEE Computer Graphics & Applications, and he has presented research at conferences including ACM SIGGRAPH, the ACM Symposium on Interactive 3D Graphics, IEEE Virtual Reality, and Eurographics Virtual Environments. He has also co-authored "3D User Interfaces: Theory and Practice," the first comprehensive book on 3D user interfaces. Joseph received a Sc.M. in Computer Science in 2000, a Sc.M. in Applied Mathematics in 2001, and a Ph.D. in Computer Science in 2005 from Brown University.

Takeo Igarashi is an associate professor at computer science department, the University of Tokyo. His research interest is in user interface in general and current focus is on interaction techniques for 3D graphics. His work has appeared in premier conferences and journals in computer graphics and human-computer interactions such as ACM Transactions on Computer Graphics, Computer Graphics Forum, ACM SIGCHI conference, and ACM symposium on User Interfaces Software and Technology. He received the ACM SIGGRAPH Significant New Researcher Award in 2006 and his paper on a sketch-based 3D freeform modeling system received an Impact Paper Award at SIGGRAPH 1999. He also received a number of other awards including the ACM UIST 2002 best paper award, the IEEE Visual Languages 1998 best paper award, and the IBM Science Award (2004). Takeo received his master's degree and Ph.D. in Information Engineering in 1997and 2000 respectively from the University of Tokyo. He worked as a postdoctoral research associate in Computer Science at Brown University from 2000 to 2002.

Christine Alvarado is an assistant professor of computer science at Harvey Mudd College. Her primary research interests lie in the intersection of artificial intelligence and human-computer interaction. She focuses on building robust, free-sketch recognition-based interfaces and exploring how to resolve the user interface challenges associated with these interfaces. She has presented her work at international conferences such as IJCAI and UIST. In 2006, she received an NSF Faculty Early Career Development (CAREER) Award to improve computer recognition of digital circuit designs in order to construct a sketch-based simulation tool that may be incorporated into Harvey Mudd's introductory digital design and computer architecture class. In addition to her sketch understanding research, Dr. Alvarado is actively involved in outreach efforts to increase the number of women in computer science. Dr. Alvarado received her undergraduate degree in computer science from Dartmouth in 1998, graduating summa cum laude. She received her S.M. and Ph.D. in computer science from MIT in 2000 and 2004, respectively.

Hod Lipson is an Assistant Professor of Mechanical & Aerospace Engineering and Computing & Information Science at Cornell University in Ithaca, NY. He directs the Computational Synthesis group, which focuses on novel ways for automatic design, fabrication and adaptation of virtual and physical machines. He has led work in areas such as evolutionary robotics, functional rapid prototyping, machine self-replication and self-assembly, and design automation interfaces. His work on evolutionary robotics made front-page news worldwide in 2000, and his lab's work on machine self-replication and self-modeling made headlines again in 2005 and 2006. Lipson has received the NSF CAREER award in 2006 and his work was covered by TIME magazine's survey of most important events in 2000. Lipson received his Ph.D. from the Technion - Israel Institute of Technology in 1998, and continued to a postdoc at Brandeis University and MIT. His research focuses primarily on biologically-inspired approaches, as they bring new ideas to engineering and new engineering insights into biology.

SIGGRAPH 2007 COURSE 3: SKETCH-BASED INTERFACES: TECHNIQUES AND APPLICATIONS

Morning

8:30	Welcome and Introduction – LaViola
8:50	Sketching and Gestures 101 – LaViola
9:15	Sketch Understanding Systems -- Alvarado
10:00	Break
10:15	Sketch-Based Interfaces for Interactive Computer Graphics – Igarashi
11:15	Sketching for Mechanical Design and CAD – Lipson
11:45	Sketching and Education -- LaViola
12:00	Questions – All
12:15	Lunch Break

Afternoon

1:45	Multi-Domain Sketch Understanding – Alvarado
2:35	Designing Freeform Surfaces by Sketching – Igarashi
3:20	Break
3:35	Creating Geometry from Sketch-Based Input – Lipson
4:35	Mathematical Sketching – LaViola
5:15	Questions – All

Table of Contents

“Resolving Ambiguities to Create a Natural Sketch Based Interface”, Christine Alvarado and Randall Davis

“Hierarchical Parsing and Recognition of Hand Sketched Diagrams”, Levent Burak Kara and Thomas Stahovich

Sketch-Based Interfaces for Interactive Computer Graphics

"Interactive Beautification: A Technique for Rapid Geometric Design", T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka

"SKETCH: An interface for sketching 3D scenes", R.C. Zeleznik, K.P. Herndon, and J.F. Hughes

"A Suggestive Interface for 3D Drawing", T. Igarashi andJ.F. Hughes

"Teddy: A Sketching Interface for 3D Freeform Design", T. Igarashi, S. Matsuoka, and H. Tanaka

"Smooth Meshes for Sketch-based Freeform Modeling", T. Igarashi and J.F. Hughes

"A Sketching Interface for Articulated Figure Animation", J. Davis, M. Agrawala, E. Chuang, Z. Popovic, and D. Salesin

“Motion Doodles: An Interface for Sketching Character Motion”, M. Thorne, D. Burke, and M. Panne

“Spatial Keyframing for Performance-driven Animation”, T. Igarashi, T. Moscovich, and J.F. Hughes

"Interactive Design of Botanical Trees Using Freehand Sketches and Example-based Editing", M. Okabe, S. Owada, and T. Igarashi

"Floral Diagrams and Inflorescences: Interactive Flower Modeling Using Botanical Structural Constraints", T. Ijiri, M. Okabe, S. Owada, and T. Igarashi

"Sketching Garments for Virtual Characters", E. Turquin, M-P. Cani, and J.F. Hughes

"Clothing Manipulation", T. Igarashi and J.F. Hughes

"ShapeShop: Sketch-Based Solid Modeling with BlobTrees", R. Schmidt, B. Wyvill, M. C. Sousa, and, J. A. Jorge

Creating Geometry from Sketch-Based Input

"Correlation-Based Reconstruction of a 3D Object From a Single Freehand Sketch", H. Lipson and M. Shpitalni

“Optimization-Based Reconstruction of a 3D Object From a Single Freehand Line Drawing”, H. Lipson and M. Shpitalni

Mathematical Sketching

“MathPad^2: A System for the Creation and Exploration of Mathematical Sketches”, Joseph LaViola and Robert Zeleznik

“An Initial Evaluation of a Pen-Based Tool for Creating Dynamic Mathematical Illustrations”, Joseph LaViola

PART III: Bibliography

Lecture Slides and Notes

Sketch-Based Interfaces: Techniques and Applications

Welcome and Introduction

SIGGRAPH2007

Joseph J. LaViola Jr.

School of EECS

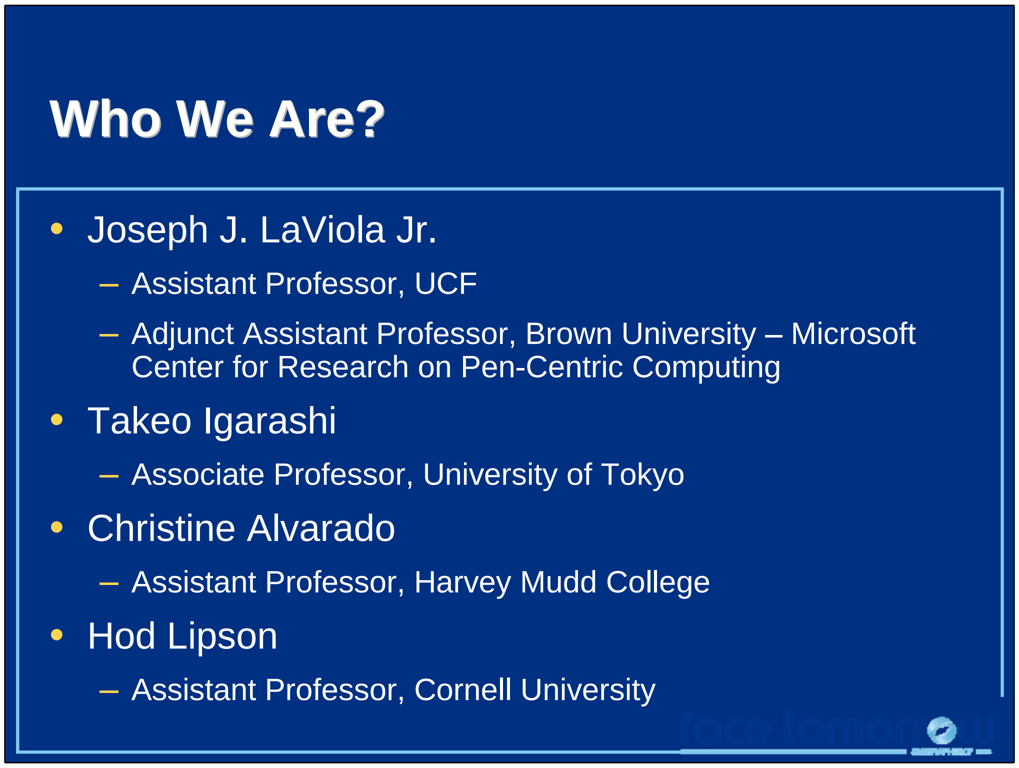
University of Central Florida

Welcome to the Sketch-Based Interfaces course at SIGGRAPH 2007. The organizer of this course is Joseph LaViola from the University of Central Florida. In this lecture, you will meet the other presenters, get an overview of the topics we will cover during the course, and see some examples of sketch-based interfaces.

My contact information:

Joseph J. LaViola Jr.
Assistant Professor
University of Central Florida
School of EECS
4000 Central Florida Blvd.
Orlando, FL 32816

Email: jjl@cs.ucf.edu
http://www.cs.ucf.edu/~jjl/



Who We Are?

- Joseph J. LaViola Jr.
  - Assistant Professor, UCF
  - Adjunct Assistant Professor, Brown University – Microsoft Center for Research on Pen-Centric Computing
- Takeo Igarashi
  - Associate Professor, University of Tokyo
- Christine Alvarado
  - Assistant Professor, Harvey Mudd College
- Hod Lipson
  - Assistant Professor, Cornell University

Joseph LaViola (University of Central Florida)

jjl@cs.ucf.edu

http://www.cs.ucf.edu/~jjl/

Takeo Igarashi (University of Tokyo)

takeo.igarashi@gmail.com

http://www-ui.is.s.u-tokyo.ac.jp/~takeo/

Christine Alvarado (Harvey Mudd College)

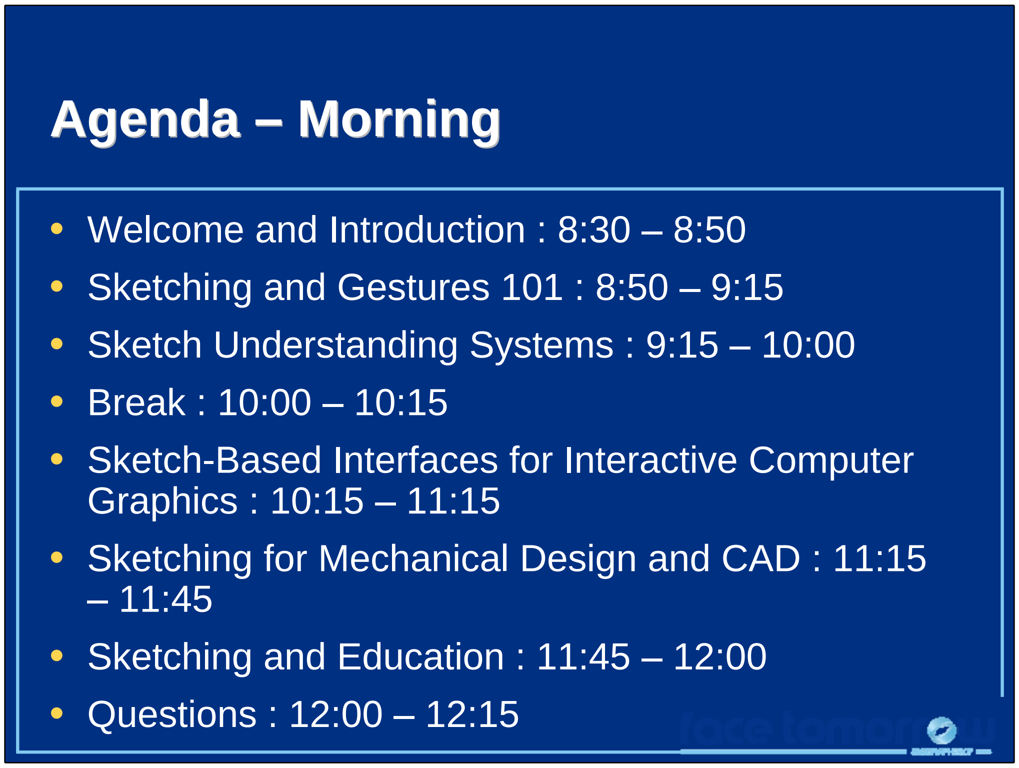
alvarado@cs.hmc.edu

http://www.cs.hmc.edu/~alvarado/

Hod Lipson (Cornell University)

hod.lipson@cornell.edu

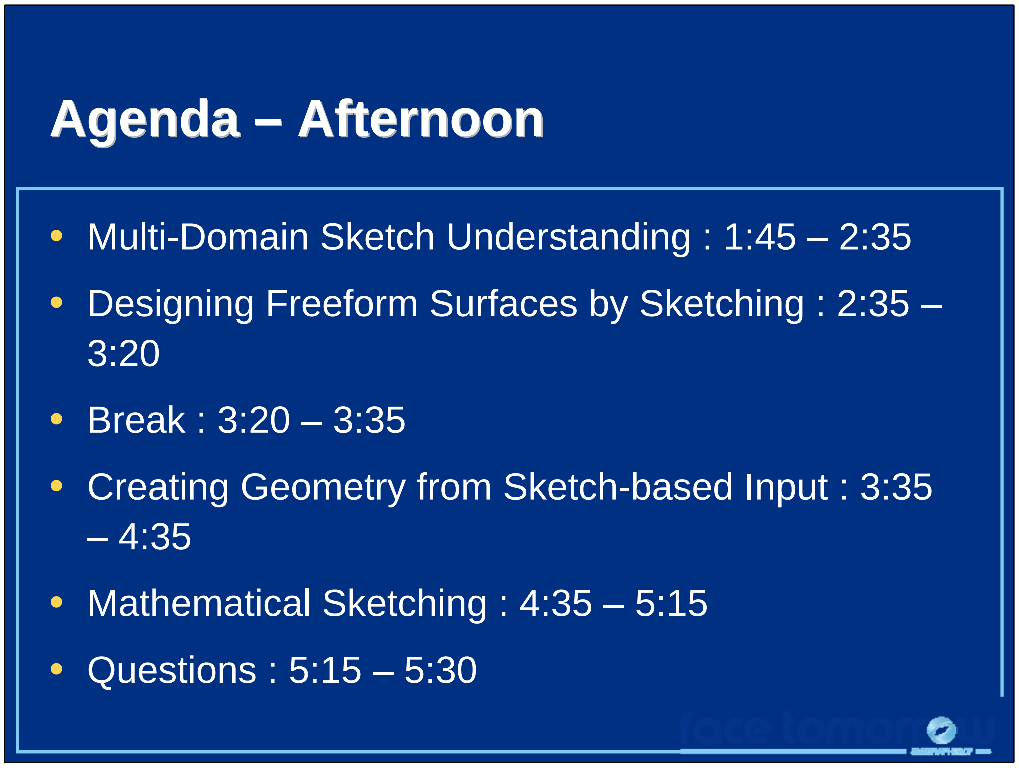
http://www.mae.cornell.edu/lipson/



Agenda – Morning

- Welcome and Introduction : 8:30 – 8:50
- Sketching and Gestures 101 : 8:50 – 9:15
- Sketch Understanding Systems : 9:15 – 10:00
- Break : 10:00 – 10:15
- Sketch-Based Interfaces for Interactive Computer Graphics : 10:15 – 11:15
- Sketching for Mechanical Design and CAD : 11:15 – 11:45
- Sketching and Education : 11:45 – 12:00
- Questions : 12:00 – 12:15

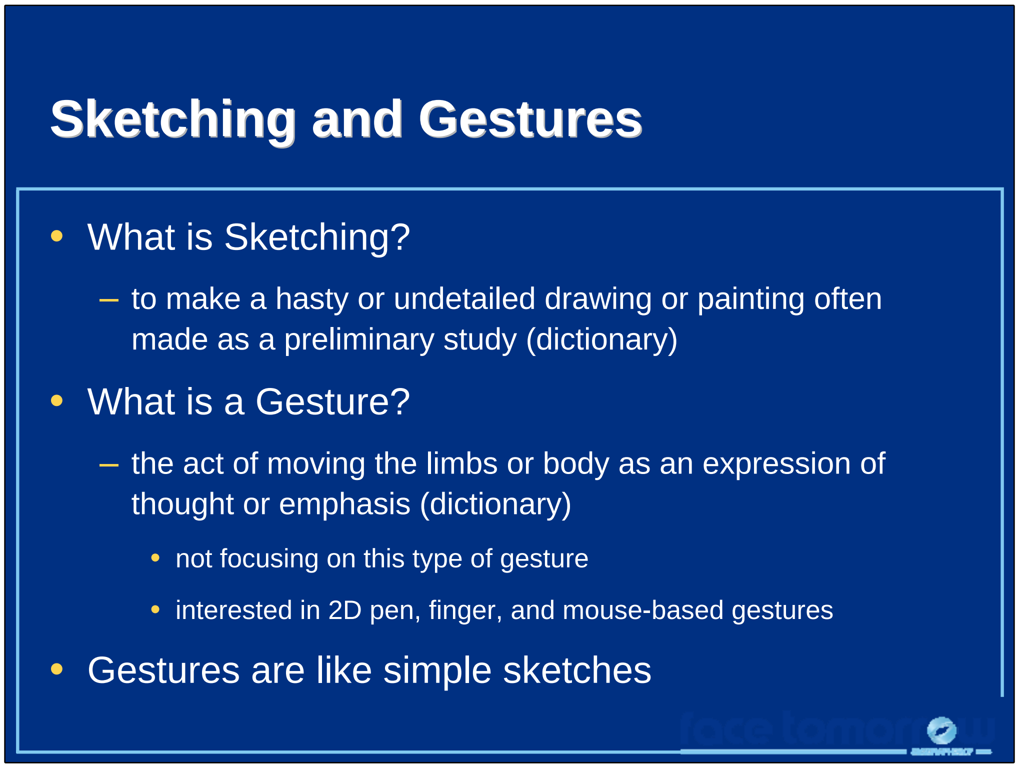
We will begin the course by introducing the idea of sketch-based interfaces and looking at some examples. Next, we will give a short tutorial on sketching and gestures and talk about some of the issues you need to think about when working with these types of interfaces. Next, we will discuss sketch understanding systems and present an architecture for building them. After the break, we will have two lectures on sketching and computer graphics. First, we will discuss how sketch-based interfaces are used for general interactive computer graphics. Second, we will discuss sketch-based interfaces directly related to mechanical design and CAD. The last lecture in the morning session will be on some examples of how sketch-based interfaces can be used in educational settings. To close out the morning, we will have a short question and answer period.



Agenda – Afternoon

- Multi-Domain Sketch Understanding : 1:45 – 2:35
- Designing Freeform Surfaces by Sketching : 2:35 – 3:20
- Break : 3:20 – 3:35
- Creating Geometry from Sketch-based Input : 3:35 – 4:35
- Mathematical Sketching : 4:35 – 5:15
- Questions : 5:15 – 5:30

In the second half of the course, we will go into more algorithmic detail on sketch-based interfaces. In the first lecture of the afternoon, we will discuss how to develop sketch understanding systems that can handle multiple domains. The second lecture will focus on algorithms for designing free-form surfaces by sketching. After the break, we will give a lecture on creating geometry, related to mechanical design and CAD, from sketch-based input. The final lecture in the course will discuss mathematical sketching, which is an approach for creating dynamic illustrations by combining handwritten mathematics and free-form drawings. We will conclude the afternoon with a short question and answer period.

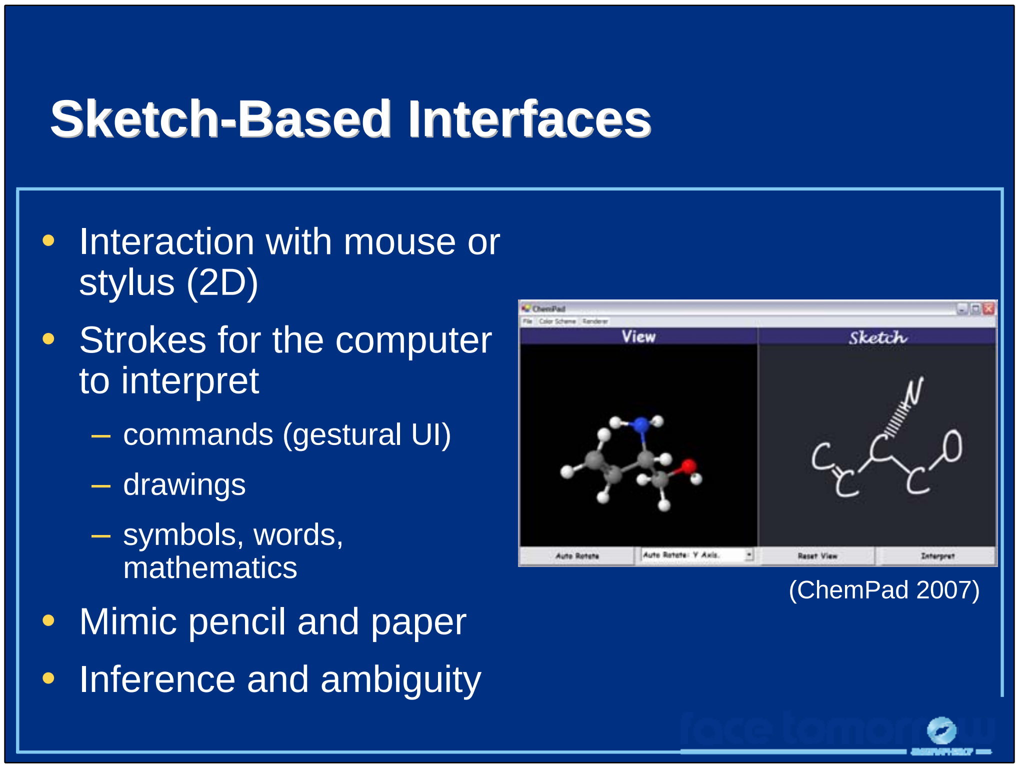


Sketching and Gestures

- What is Sketching?
  - to make a hasty or undetailed drawing or painting often made as a preliminary study (dictionary)
- What is a Gesture?
  - the act of moving the limbs or body as an expression of thought or emphasis (dictionary)
    - not focusing on this type of gesture
    - interested in 2D pen, finger, and mouse-based gestures
- Gestures are like simple sketches

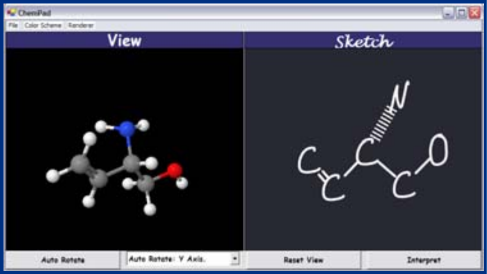
According to the dictionary, the idea of sketching is to make a hasty or undetailed drawing or painting as a preliminary study. For our purposes, as it pertains to interfaces, sketching is a process for entering information into the computer using a stylus or mouse with digital ink strokes. Typically in a sketch-based interface, the user enters a series of strokes and the computer interprets them to accomplish some task, be it creating a conceptual model, making an animation, or recognizing mathematics.

When we look up “gesture” in the dictionary, the definition states it is the act of moving the limbs or body as an expression of thought or emphasis. The dictionary’s definition is more suited to 3D gestures such as hand gestures used in virtual reality applications or in sign language. We will not focus on these types of gestures in this course. In our case, gestures, just like in sketching, are created using a stylus or mouse. The main difference between them is that gestures can be much shorter in terms of the number of strokes used. With a sketching-based interface, the user could make dozens of strokes before the computer interprets the sketch. Typically one or two strokes are used (sometimes as many as 3 or 4) as a gesture. Thus, we can consider gestures to be simple sketches.



Sketch-Based Interfaces

- Interaction with mouse or stylus (2D)
- Strokes for the computer to interpret
  - commands (gestural UI)
  - drawings
  - symbols, words, mathematics
- Mimic pencil and paper
- Inference and ambiguity

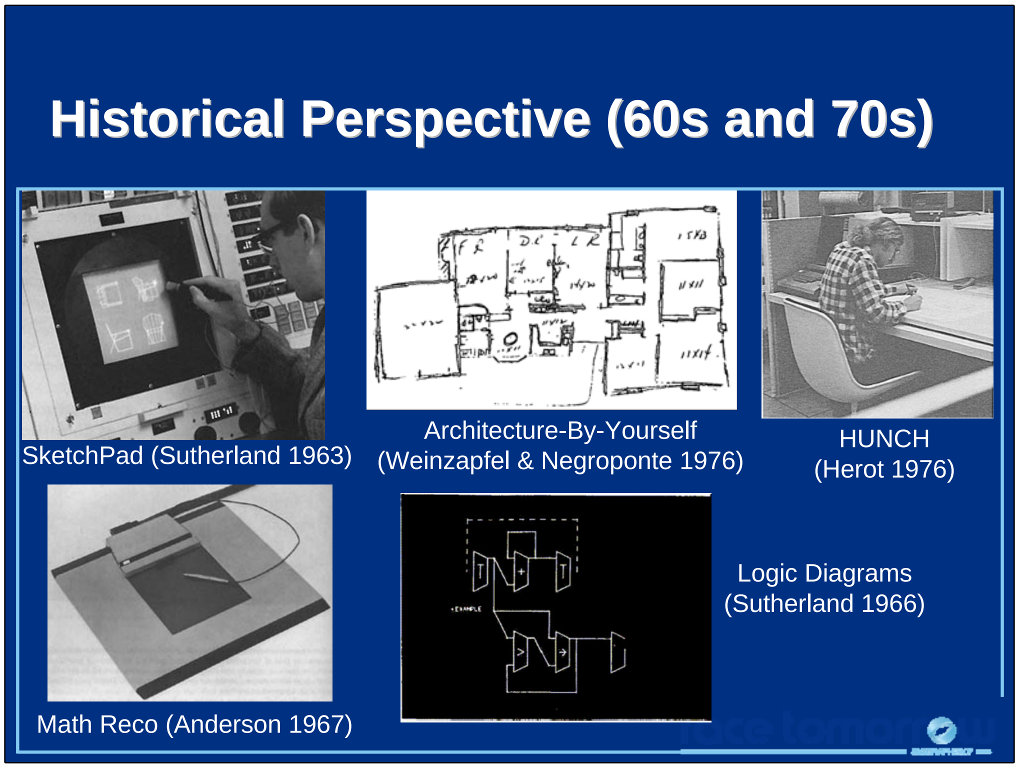


(ChemPad 2007)

Sketch-based interfaces typically use a mouse or stylus and are restricted to 2D input (there are some sketch-based interfaces that use 3D strokes but we will not discuss them here). Given a set of strokes, the computer interprets them, and, depending on the number of strokes and the application, they can be used to issue commands, take drawings and turn them into models or diagrams, or interpret them as symbols, words, or mathematical expressions. The main idea behind sketch-based interfaces is that we want them to mimic pencil and paper, a tool we have been using since the early days of our education that represents a natural way to think about ideas and communicate. By creating interfaces that mimic pencil and paper, we can, in many cases, enter information more easily keeping a natural style of interaction and leveraging the power of computation. For example, the figure in the slide shows a screenshot of the ChemPad application. It lets organic chemistry students sketch out molecules in 2D and then represents them in 3D so they can gain a better understanding of the 3D structure of certain molecular configurations. Sketch-based interface also have the property that they often contain ambiguities, since it can be difficult for the computer to completely and precisely understand the user's intentions when they make the sketch. Dealing with these ambiguities and making inferences about the users' intentions is one of the fundamental research areas in sketch-based interfaces.

References:

graphics.cs.brown.edu/research/chempad/home.html



Historical Perspective (60s and 70s)

SketchPad (Sutherland 1963)

Architecture-By-Yourself (Weinzapfel & Negroponte 1976)

HUNCH (Herot 1976)

Math Reco (Anderson 1967)

Logic Diagrams (Sutherland 1966)

The idea of using sketching and gestures for interacting with computers is not a new idea. In fact, researchers have been thinking about sketch-based interfaces since the early 1960s. The pictures in the slide show some of the earliest examples of work on sketch-based interfaces. Most notably, Ivan Sutherland's seminal work on SketchPad, where he used a light pen to make drawings and create geometric primitives. His brother, Bert, created a system for sketching out logic diagrams. Richard Anderson used the RAND tablet to recognize mathematical expressions. Other notable work came out of MIT in the late 1970s, such as Architecture-By-Yourself and HUNCH, which began to explore how computers could interpret hand-drawn diagrams and what inference mechanisms and domain knowledge were needed to do so.

References:

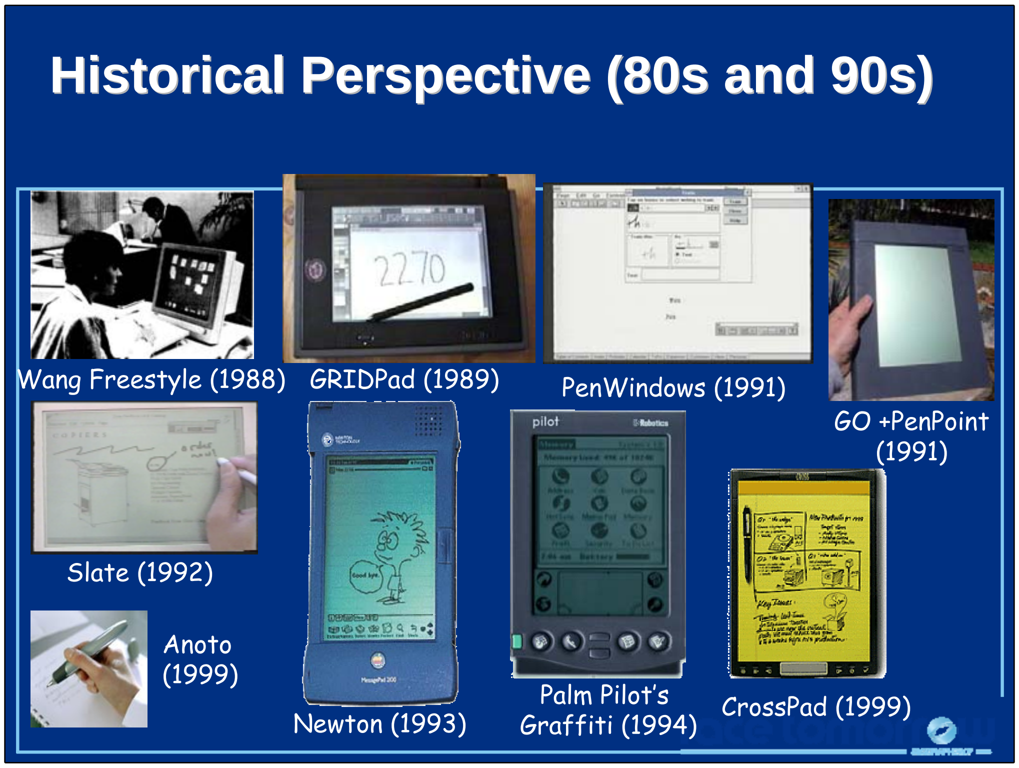
Sutherland, I. SketchPad: A Man-Machine Graphical Communication System, *Proceedings of AFIPS Spring Joint Computer Conference*, 329-346, 1963.

Sutherland, William Robert. On-Line Graphical Specification of Computer Procedures. PhD. Thesis, MIT, 1966

Anderson, Robert. Syntax-Directed Recognition of HandPrinted Two dimensional Equations. PhD. Thesis, Harvard University, 1968.

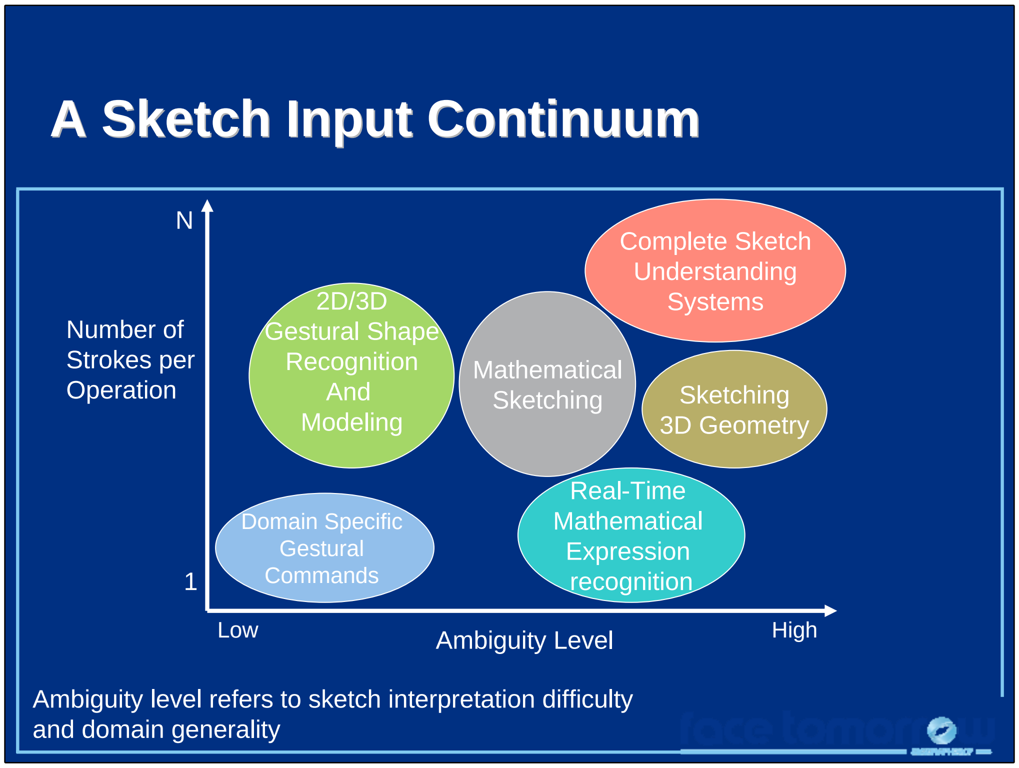
Weinzapfel, G. and N. Negroponte. Architeture-By-Yourself: An Experiment with Computer Graphics for House Design, *Proceedings of SIGGRAPH'76*, 74-78, 1976.

Herot, C. Graphical Input Through Machine Recognition of Sketches, *Proceedings of SIGGRAPH'76*, 97-102, 1976.



Historical Perspective (80s and 90s)

In the 80s and 90s, a number of pen-based devices began to appear. An example of some of them are shown in the slide. Unfortunately, most of these commercial efforts failed for various reasons such as slow computing speed, insufficient battery life, and lack of sophisticated recognition technology. Given our advances in software and hardware, we have seen an explosion of both sketch-based interfaces and pen-based computing devices in the last decade as they are becoming much more practical.



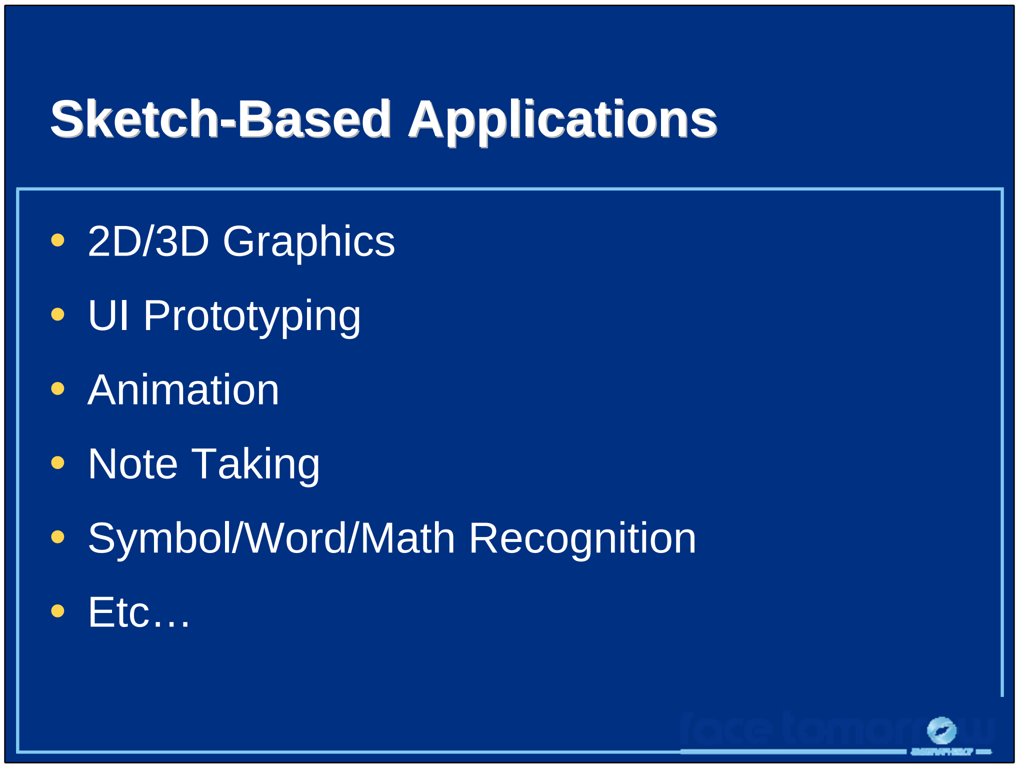
A Sketch Input Continuum

N
Number of Strokes per Operation
1
Low
Ambiguity Level
High
Complete Sketch Understanding Systems
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition

Ambiguity level refers to sketch interpretation difficulty and domain generality

There are many different sketch-based interfaces that exist today. Thus, it is important to be able to distinguish and categorize them in some way. Two of the most important characteristics of sketch-based interfaces are the number of strokes the computer looks at to make an interpretation and the underlying ambiguity level. The ambiguity level refers to how difficult it is to interpret a sketch given the generality or specificity of the domain. In other words, a sketch-based interface can have a high ambiguity level if there are many possible interpretations the computer could find for any one particular sketch, and this often occurs with very general domains. Restricting the domain to be very limited in scope (e.g., simple geometric shapes) can reduce the ambiguity level, but not always.

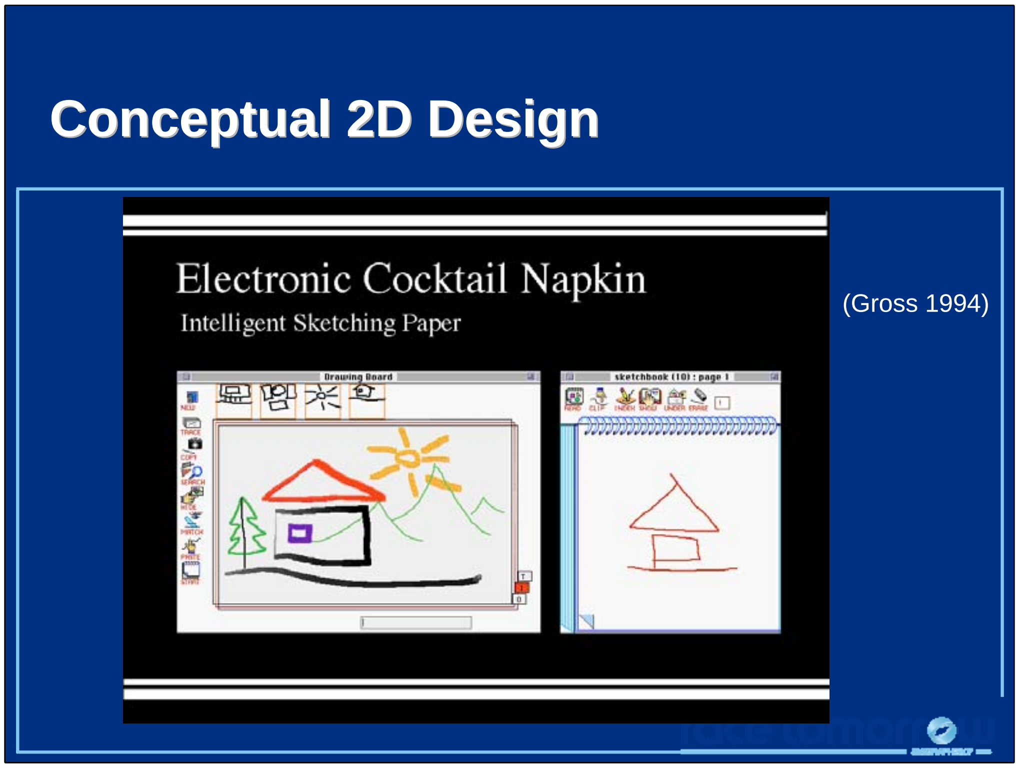
We refer to our classification scheme (there are, of course, other possibilities) as the sketch input continuum. The figure in the slide show some examples of where certain types of sketch-based interfaces fit into the continuum. For example, complete sketch understanding systems often must examine many strokes at a time, which can lead to high ambiguity levels. In another example, domain specific gestural commands (e.g., simple editing commands for note taking) often require only one or two strokes per command with low ambiguity levels, given they are often very application specific. Of course, keep in mind that this continuum is more of a guideline than a strict rule.



Sketch-Based Applications

- 2D/3D Graphics
- UI Prototyping
- Animation
- Note Taking
- Symbol/Word/Math Recognition
- Etc…

Sketch-based interfaces have been used in a variety of different application domains such as graphics, user interface prototyping, animation, character recognition, and others. In the next several slides, we will look at some specific examples.



Conceptual 2D Design

Electronic Cocktail Napkin

Intelligent Sketching Paper

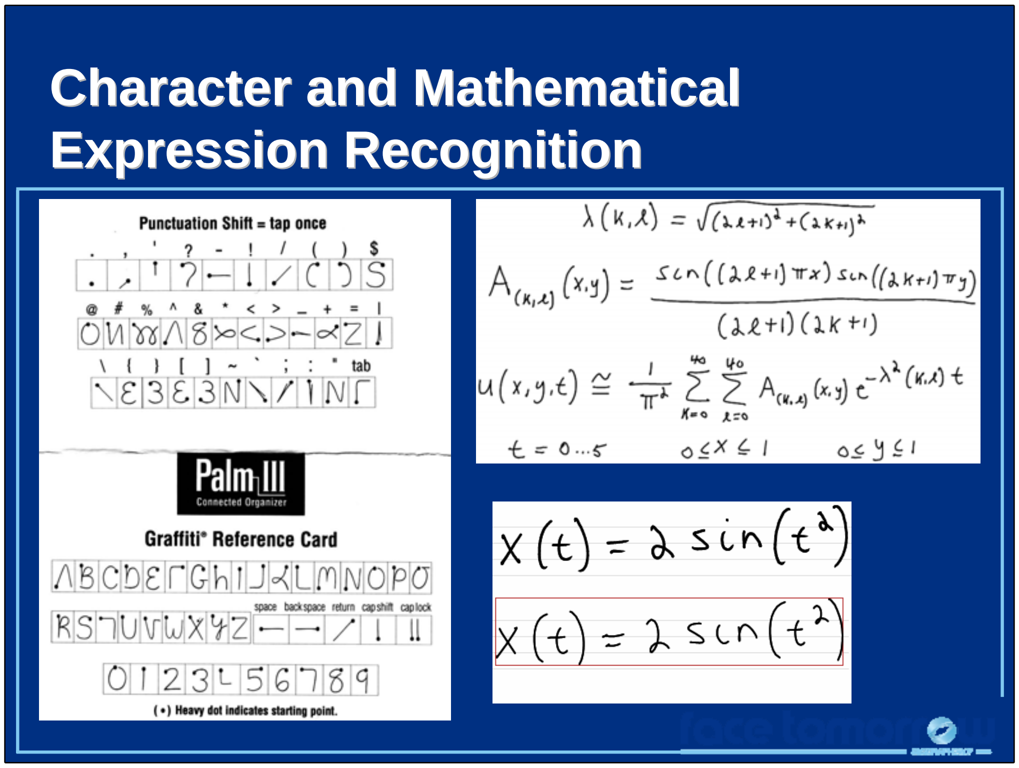
(Gross 1994)

A system for doing conceptual 2D design is Gross's Electronic Cocktail Napkin. It lets users make simple 2D drawings that are interpreted using low-level recognizers as part of a diagram interpretation system.

References:

Gross, Mark. Advanced Visual Interfaces in "Recognizing and Interpreting Diagrams in Design" Gross, M.D. In T. Catarci. M. Costabile, S. Levialdi, G. Santucci eds., *Advanced Visual Interfaces '94 (AVI '94)*, ACM Press, 1994.

Gross, Mark. Stretch-a-Sketch: A Dynamic Diagrammer. *IEEE Symposium on Visual Languages (VL '94)* 232-238, 1994.



Character and Mathematical Expression Recognition

Punctuation Shift = tap once
Palm III
Connected Organizer
Graffiti® Reference Card
space backspace return capshift caplock
(•) Heavy dot indicates starting point.

$$\lambda(k,l) = \sqrt{(2l+1)^2+(2k+1)^2}$$

$$A_{(k,l)}(x,y) = \frac{\sin((2l+1)\pi x)\sin((2k+1)\pi y)}{(2l+1)(2k+1)}$$

$$u(x,y,t) \cong \frac{1}{\pi^2}\sum_{k=0}^{40}\sum_{l=0}^{40} A_{(k,l)}(x,y)\, e^{-\lambda^2(k,l)\,t}$$

$$t = 0 \ldots 5 \qquad 0 \le x \le 1 \qquad 0 \le y \le 1$$

$$x(t) = 2\sin(t^2)$$

$$x(t) = 2\sin(t^2)$$

Sketch-based interfaces are used for character, word, and mathematical expression recognition. Although these tasks are not traditionally associated with sketch-based interfaces, alphanumeric symbols and languages such as Graffiti can be considered simple sketches or gestural commands, as they have many of the same features that more traditional sketch-based interfaces do. Mathematical expression recognition is actually a more complex recognition process than dealing with isolated symbols, like those used in Graffiti, because the spatial relationships between the symbols in a mathematical expression are critical to its interpretation. Work in the recognition of handwritten characters and mathematics has been going on since the 1960s. Only recently, have we seen these recognizers robust enough for use in commercial applications.

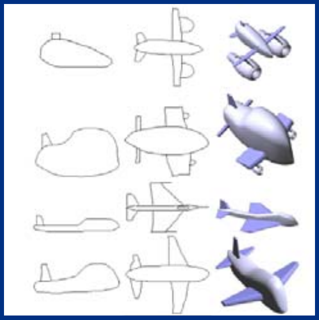
References:

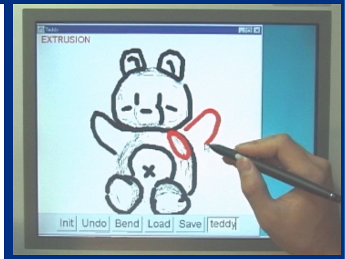
D. Blostein and A.Grbavec. Recognition of Mathematical Notation, *Handbook of Character Recognition and Document Image Analysis*, Eds. H. Bunke and P. Wang, World Scientific, 1997, pp. 557-582.

Tappert, C.C., C.Y. Suen, and T. Wakahara, The state-of-the-art in on-line handwriting recognition, *IEEE Transactions Pattern Analysis Machine Intelligence*, Vol. PAMI-12, pp. 787-808, August 1990.

3D Modeling

SKETCH
(Zeleznik et al. 1996)


Parameterized Object
Sketching
(Yang et al. 2005)



TEDDY
(Igarashi et al. 1999)

A common application for sketch-based interfaces is 3D modeling. Using a sketch-based interface for this type of task is a very natural one since users can make rough drawings of the models they are interested in and have the computer interpret them to generate the 3D geometry. The SKETCH (the figure on the left) system, developed by Zeleznik et al., uses a gestural interface to create standard 3D geometric primitives such as cubes, cylinders, and pyramids for conceptual 3D modeling. The figure on the right shows the Teddy system, developed by Igarashi et al., that lets users make more free-form, organic 3D models. Both of these systems try to reduce the number of strokes to create 3D geometry. Other approaches use more strokes, such as the sketch-based system for modeling of parameterized objects, developed by Yang et al.

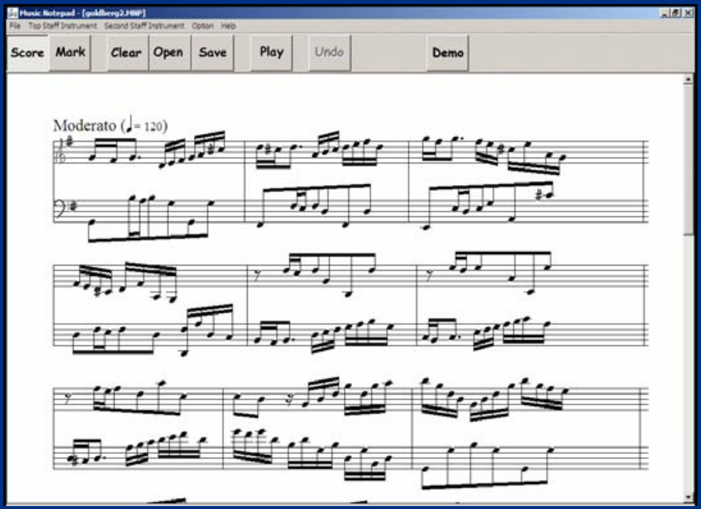
References:

Zeleznik, R., K. Herndon, and J. Hughes. SKETCH: An Interface for Sketching 3D Scenes. *Proceedings of SIGGRAPH'96*, ACM Press, 163-170, 1996.

Igarashi, T., S. Matsuoka, and H. Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *Proceedings of SIGGRAPH'99*, ACM Press, 409-416, 1999.

Yang C., D. Sharon, and Mi. van de Panne. Sketch-based Modeling of Parameterized Objects. *2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 63-72, 2005.

Musical Score Creation


Music Notepad - [goldberg2.MNP]
File Top Staff Instrument Second Staff Instrument Option Help
Score Mark Clear Open Save Play Undo Demo
Moderato (♩= 120)

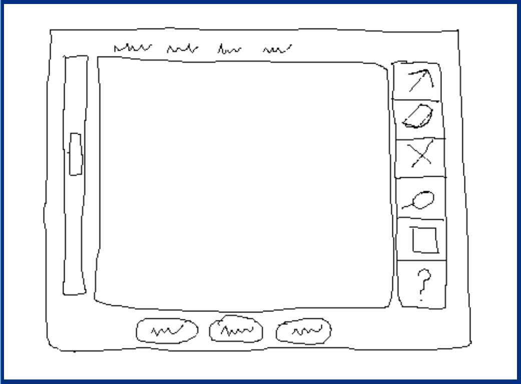
Music NotePad (Forsberg et al. 1998)

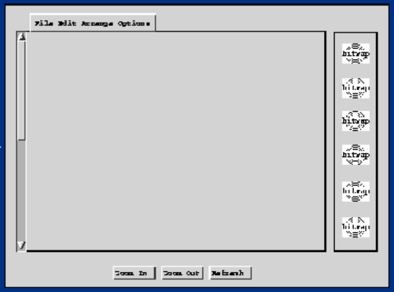
Sketch-based interfaces have also been used for entering music. The figure in the slide shows the Music Notepad, an application that lets users enter and edit musical notation with gestural commands. Once interpreted, the music can be played through the computer's sound card.

References:

Forsberg, A., M. Dieterich, and R. Zeleznik. The Music Notepad. *Proceedings of the ACM Symposium on User Interface and Software Technology (UIST)*. ACM Press, 203-210, 1998.

User Interface Prototyping





SILK (Landay and Myers 1995)

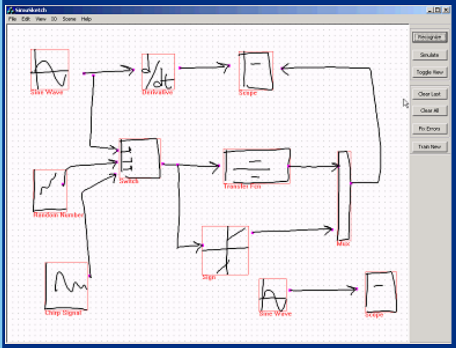
Sketch-based interfaces can be used to prototype other user interfaces. The SILK system let users sketch out the design of a user interface by drawing where the buttons, menus, sliders, and other widgets should be on the screen. The system then interprets the drawing and creates an interface skeleton that could be used in the early stages of UI design. In a similar vain, the DENIM system helps web site designers by letting them create web site prototypes using sketches.

References:

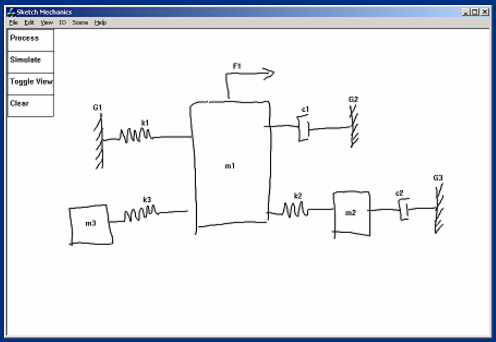
Lin J., M. Newman, J. I. Hong, and J. A. Landay, DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. *CHI Letters: Human Factors in Computing Systems, CHI 2000*, 2(1):510-517, 2000.

Landay J. and B. A. Myers, Interactive Sketching for the Early Stages of User Interface Design. Proceedings of *Human Factors in Computing Systems: CHI 95*, 43-50, 1995.

Simulation



Sim-U-Sketch
(Kara and Stahovich 2004)



VibroSketch
(Kara et al. 2004)

Sketch-based interfaces can be used in simulation. The figure on the left show an example of a sketch used as input to Simulink. The figure on the right shows a sketch that explores vibratory systems. In both cases, the users sketches out a system, the computer interprets the drawings and uses that interpretation as input to a backend system for running a simulation.

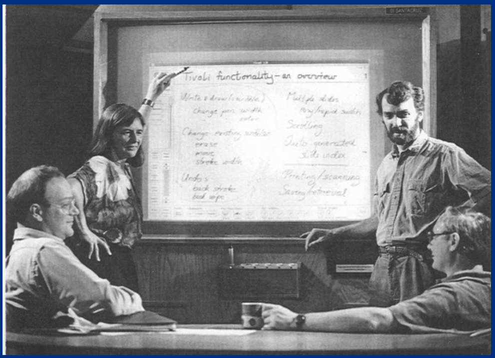
References:

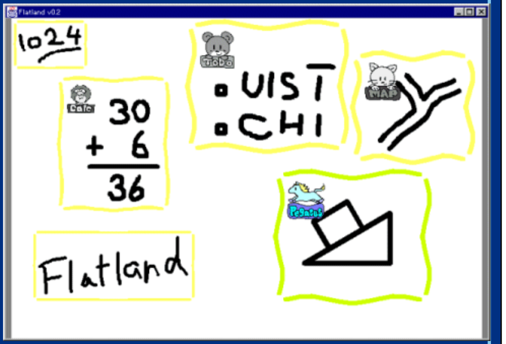
Kara, L. B. and T. F. Stahovich. Sim-U-Sketch: A Sketch-Based Interface for Simulink, *Proceedings of Advanced Visual Interfaces, 3*54-357, 2004.

Kara L. B., L. Gennari, and T. F. Stahovich. A Sketch-Based Interface for the Design and Analysis of Simple Vibratory Mechanical Systems*, 2004 ASME International Design Engineering Technical Conferences (ASME/ DETC 2004).*

Electronic Whiteboard Systems

Tivoli
(Pedersen et al. 1993)





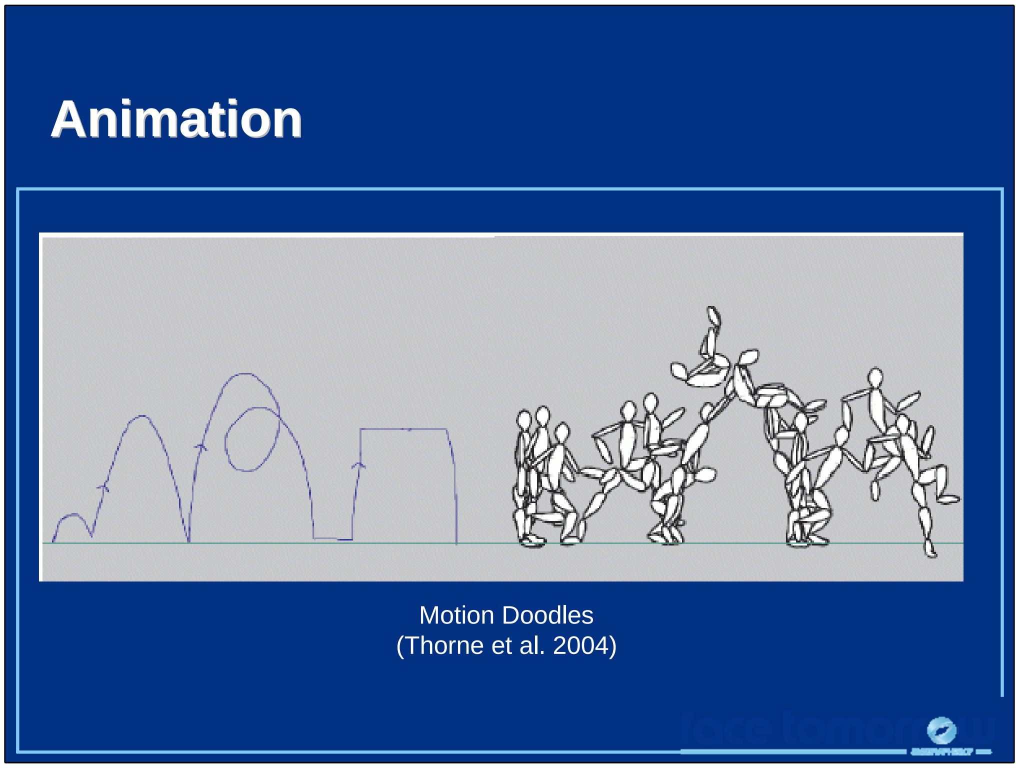
Flatland
(Mynatt et al. 1999)

Sketch-based interfaces have been developed for electronic whiteboard systems that are specifically targeted toward office meetings. The two examples shown in the slide, Tivoli and Flatland, use gestural commands rather than complex sketches for interaction. They lets users group objects together and edit and manipulate notes written down on the whiteboard.

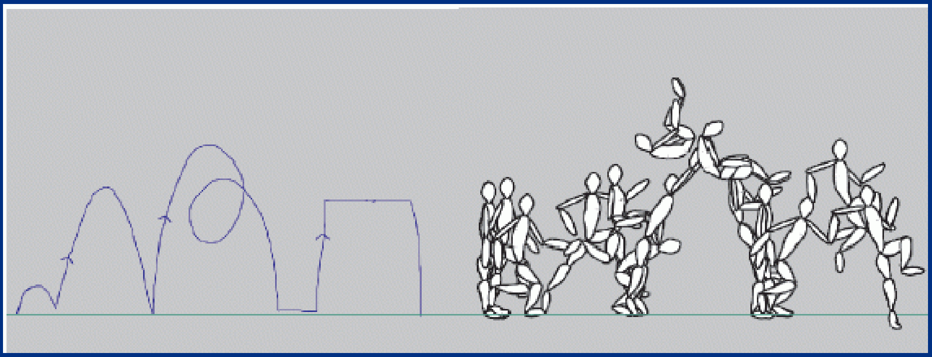
References:

Mynatt, E., T. Igarashi, W. K. Edwards, A. LaMarca, Flatland: New Dimensions in Office Whiteboards, *ACM SIGCHI Conference on Human Factors in Computing Systems CHI'99*, 346-353, 1999.

Pedersen, E., K. McCall, T. Moran, and F. Halasz, Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings, *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, 391-389, 1993.



Animation

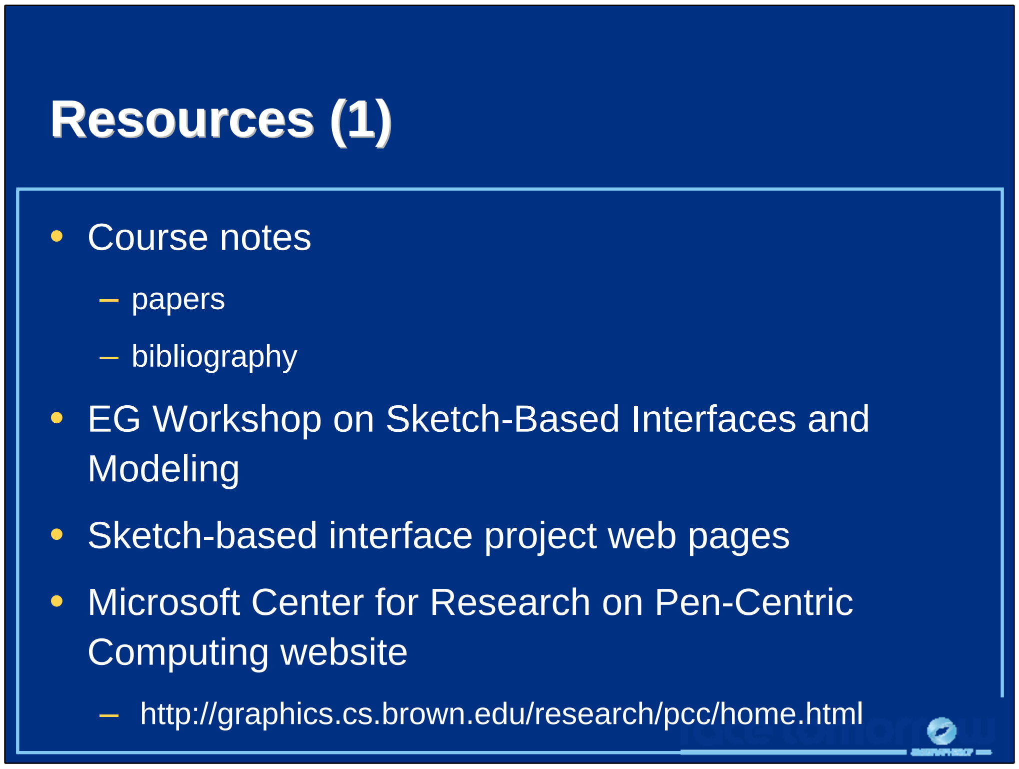


Motion Doodles
(Thorne et al. 2004)

Sketch-based interfaces can be used for making informal animations. The figure in the slide shows an example from the Motion Doodles system. Users can sketch out how a character is supposed to move in 2D and the computer interprets the sketch in order to make the animation. Other examples of sketch-based tools for animation will be given in Dr. Igarashi's lecture.

References:

Thorne, M., D. Burke, and M. van de Panne, Motion Doodles: An Interface for Sketching Character Motion. ACM Transactions on Graphics, 23(3):424-431, 2004.



Resources (1)

- Course notes
  - papers
  - bibliography
- EG Workshop on Sketch-Based Interfaces and Modeling
- Sketch-based interface project web pages
- Microsoft Center for Research on Pen-Centric Computing website
  - http://graphics.cs.brown.edu/research/pcc/home.html

There is a lot of information available on sketch-based and gestural interfaces. The notes for this course has valuable information in the lecture slides as well as some important papers about various aspects of sketch-based interface design and the types of applications that use sketch-based interfaces. Additionally, we are including a bibliography that can act as a good starting point for working in sketch-based interfaces. Another good resource is the Eurographics workshop on sketch-based interfaces and modeling. This workshop occurs yearly and has been going on since 2004. The Microsoft Center for Research on Pen-Centric Computing has a website and a wiki where researchers and practitioners can share ideas.

Another valuable resource are the various sketch-based interface research project pages. A nice sample of them is given below:

www.engr.ucr.edu/~stahov/

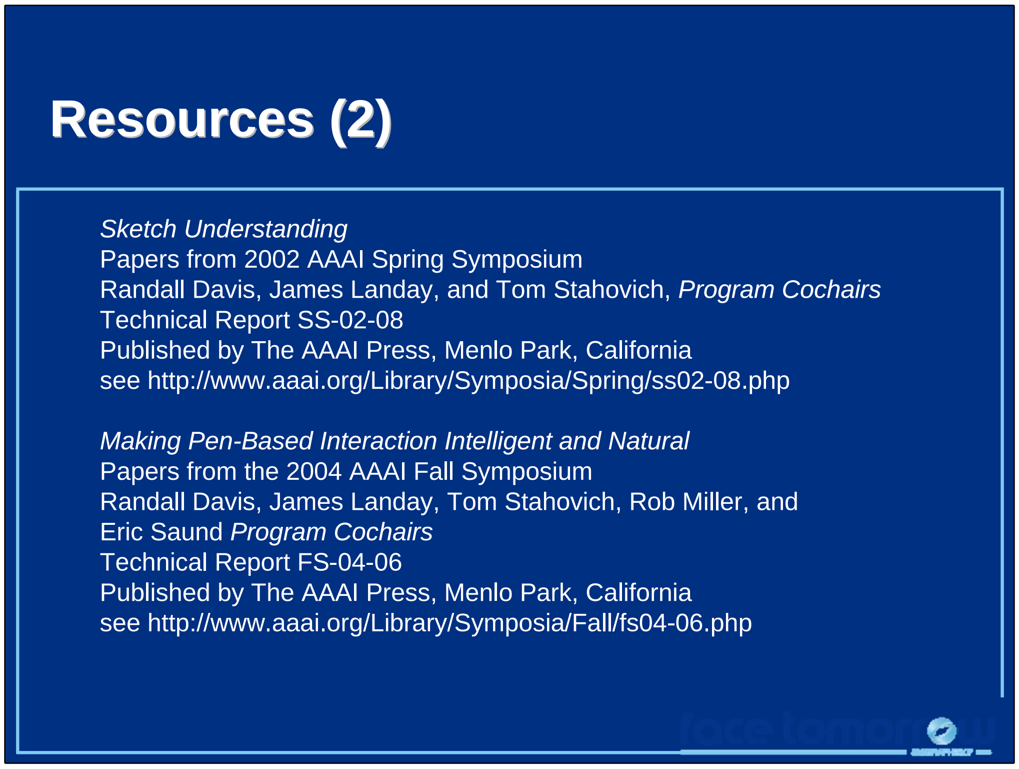
rationale.csail.mit.edu/

graphics.cs.brown.edu/research/#gesture

www-ui.is.s.u-tokyo.ac.jp/~takeo/research/Projects.html

dub.washington.edu/index.shtml

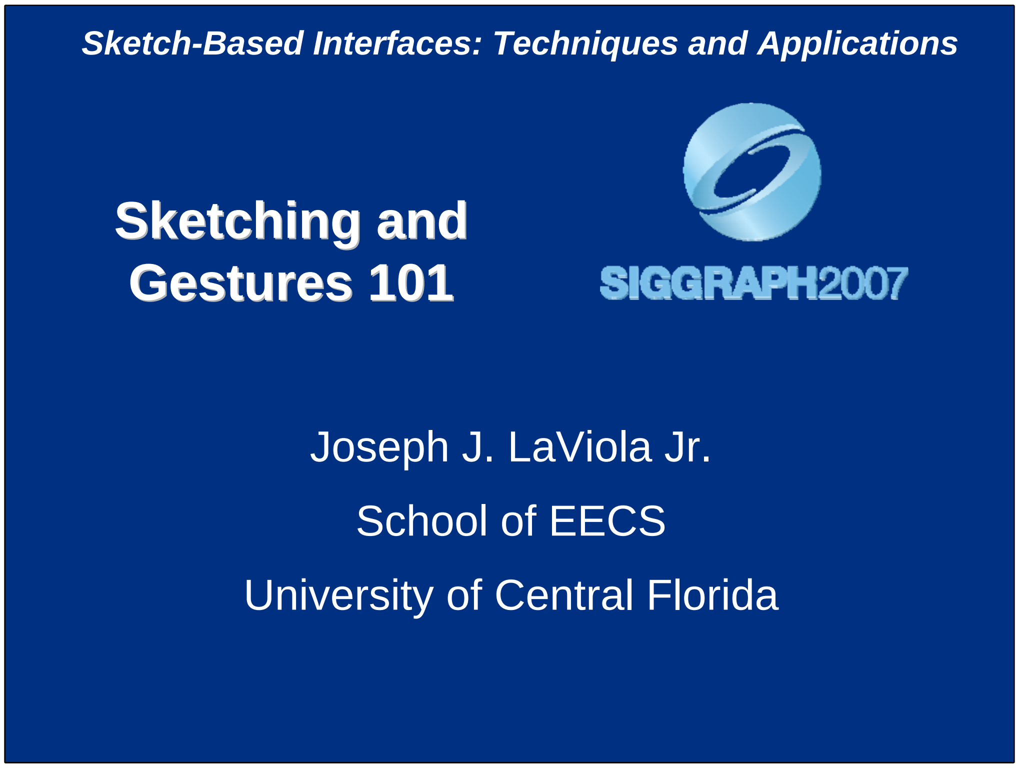
code.arc.cmu.edu/lab/html/projects.html



Resources (2)

Sketch Understanding
Papers from 2002 AAAI Spring Symposium
Randall Davis, James Landay, and Tom Stahovich, *Program Cochairs*
Technical Report SS-02-08
Published by The AAAI Press, Menlo Park, California
see http://www.aaai.org/Library/Symposia/Spring/ss02-08.php

Making Pen-Based Interaction Intelligent and Natural
Papers from the 2004 AAAI Fall Symposium
Randall Davis, James Landay, Tom Stahovich, Rob Miller, and
Eric Saund *Program Cochairs*
Technical Report FS-04-06
Published by The AAAI Press, Menlo Park, California
see http://www.aaai.org/Library/Symposia/Fall/fs04-06.php



Sketch-Based Interfaces: Techniques and Applications

Sketching and Gestures 101

SIGGRAPH2007

Joseph J. LaViola Jr.

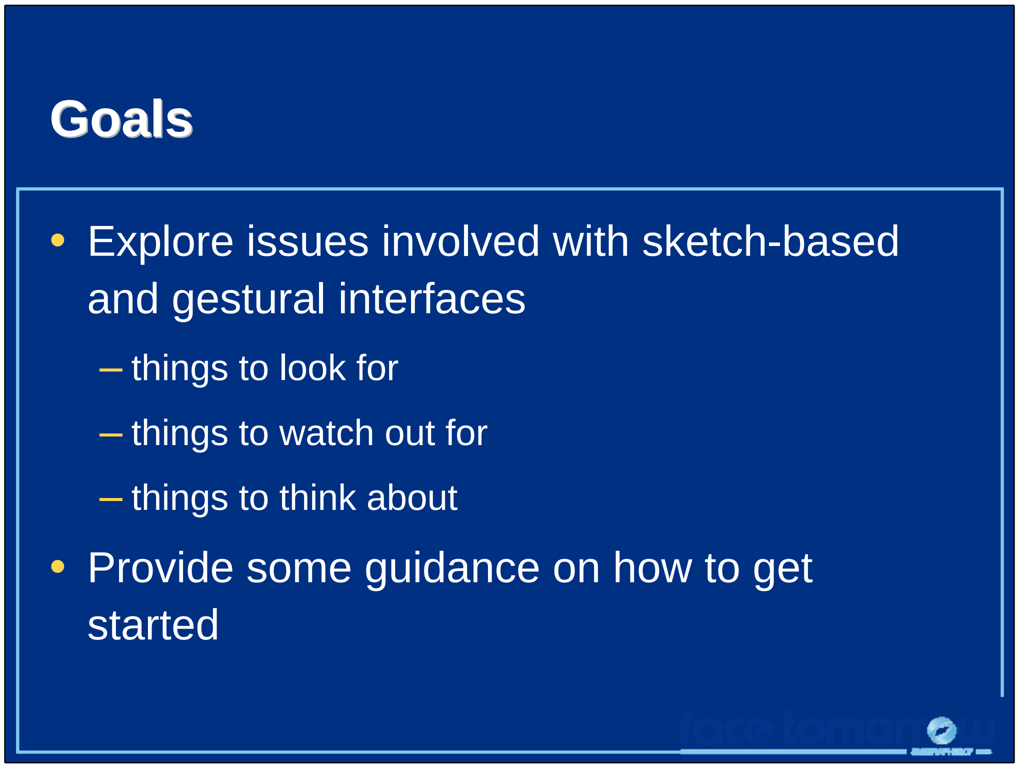
School of EECS

University of Central Florida

Sketching and Gestures 101

Joseph J. LaViola Jr.
Assistant Professor
University of Central Florida
School of EECS
4000 Central Florida Blvd.
Orlando, FL 32816

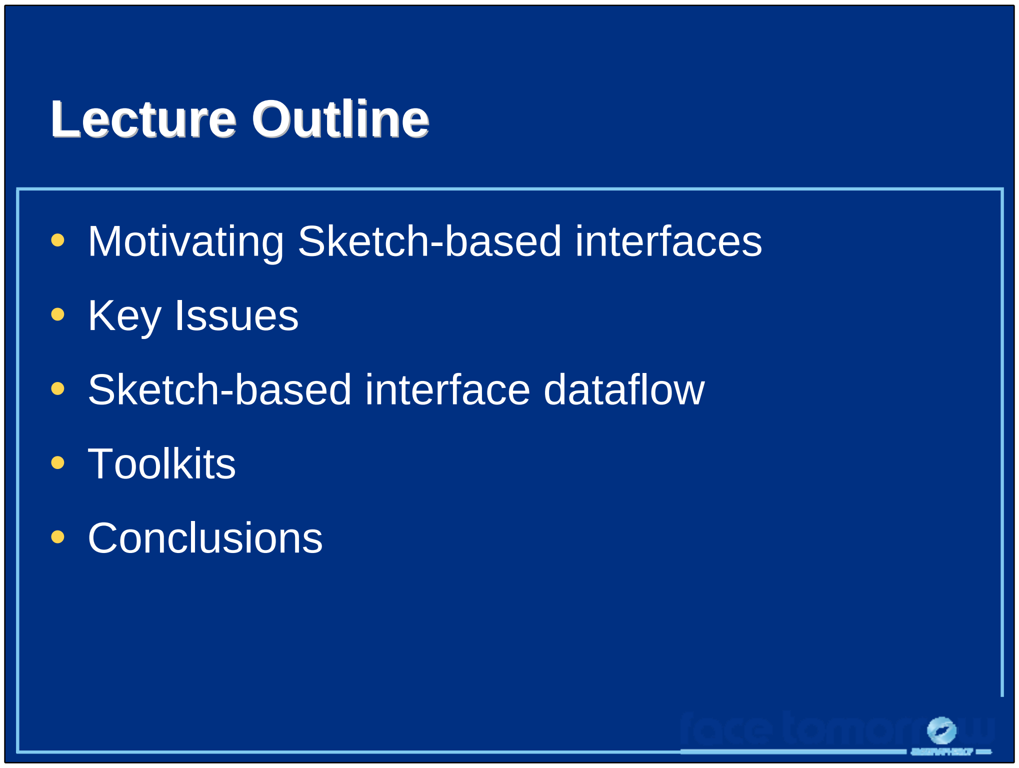
Email: jjl@cs.ucf.edu
http://www.cs.ucf.edu/~jjl/



Goals

- Explore issues involved with sketch-based and gestural interfaces
  - things to look for
  - things to watch out for
  - things to think about
- Provide some guidance on how to get started

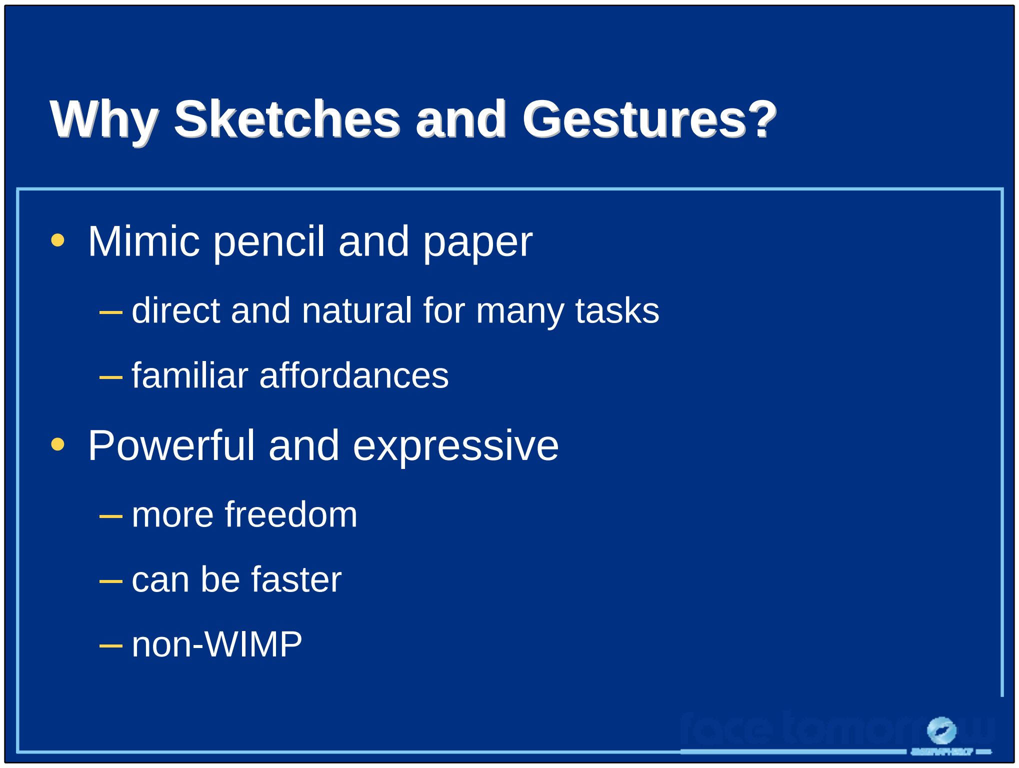
The goals for this lecture are to introduce you to some of the important issues involved with designing and implementing sketch and gesture-based interfaces. Additionally, this lecture will give you some guidance about how to get started with these interfaces.   For the remainder of this lecture, unless otherwise noted, we will include gestural command systems and interfaces within the general area of sketch-based interfaces.



Lecture Outline

- Motivating Sketch-based interfaces
- Key Issues
- Sketch-based interface dataflow
- Toolkits
- Conclusions

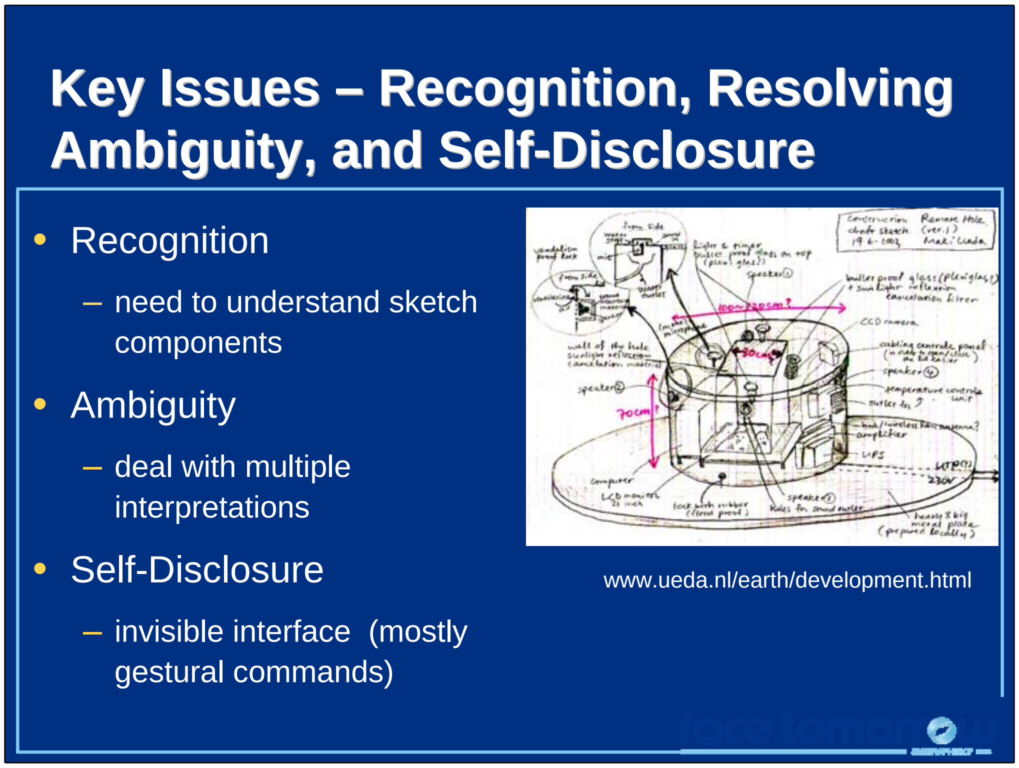
First, we will discuss why someone would want to use sketch-based interfaces and what they are good for. Second, we will discuss some important key issues that are fundamental to designing and developing sketch-based interfaces. Third, we will look at how data is typically processed moves from when users enter strokes with a stylus or mouse to the desired action, command, or interpretation. Fourth, we will briefly look at some toolkits that can assist the developer with building sketch-based interfaces. Finally, we will present some conclusions and general observations about sketching and gestures.



Why Sketches and Gestures?

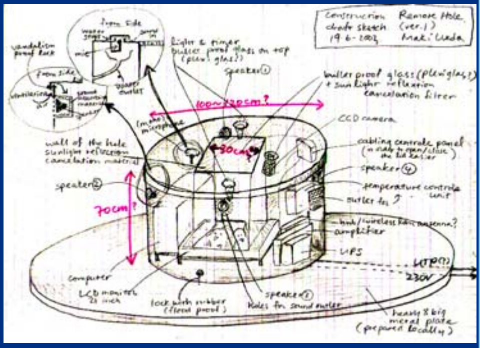
- Mimic pencil and paper
  - direct and natural for many tasks
  - familiar affordances
- Powerful and expressive
  - more freedom
  - can be faster
  - non-WIMP

One of the main reasons why using sketching and gestures for interacting with computer applications is that these types of interfaces can mimic pencil and paper (when using a stylus). They give users familiar affordances because typically most users will have had intimate knowledge of the pencil-and-paper medium. Additionally, sketch-based interfaces can be a direct and natural way to interact for many different tasks from entering notes and mathematics to drawing shapes and entering musical notation. Sketch-based interfaces can be more powerful and expressive than traditional WIMP (Windows, Icons, Menus, Point and Click) interfaces. Gestural interfaces can also be much faster than traditional interaction techniques because they often are performed "in band". In band means performing a gestural command where the action is supposed to take place. For example, a user who wants to delete an object can simply use a scribble erase gesture (moving a stroke back and forth over the object) over that object instead of having to first click on a button to get into delete mode or select a menu item in the upper left corner of the screen.



Key Issues – Recognition, Resolving Ambiguity, and Self-Disclosure

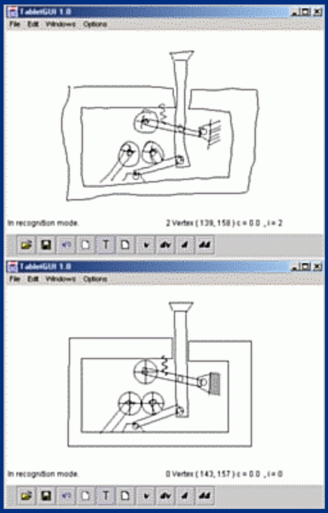
- Recognition
  - need to understand sketch components
- Ambiguity
  - deal with multiple interpretations
- Self-Disclosure
  - invisible interface (mostly gestural commands)

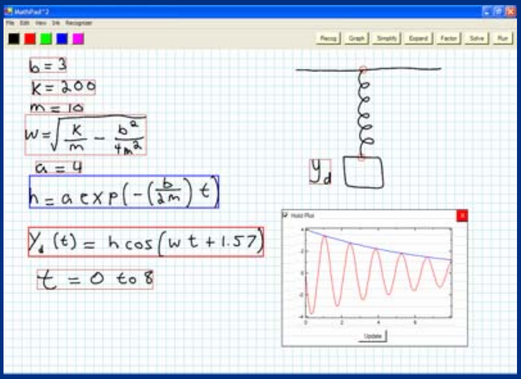


www.ueda.nl/earth/development.html

In terms of designing and developing sketch-based interfaces, there are three fundamental issues that must be addressed. First, some form of recognition will be needed. Second, there typically will be some form of ambiguity involved. In some cases, there may be multiple interpretations for a sketch or sketch-component. In other cases, there might not be enough information to make an interpretation. Third, because sketch-based interfaces often try to mimic pencil and paper, they are often non-disclosing. In other words, a novice user will not know how to use the interface by simply looking at it. For a sketch-based interface to be usable by both novices and experts, it is important to have a way for users to learn the system.

Recognition

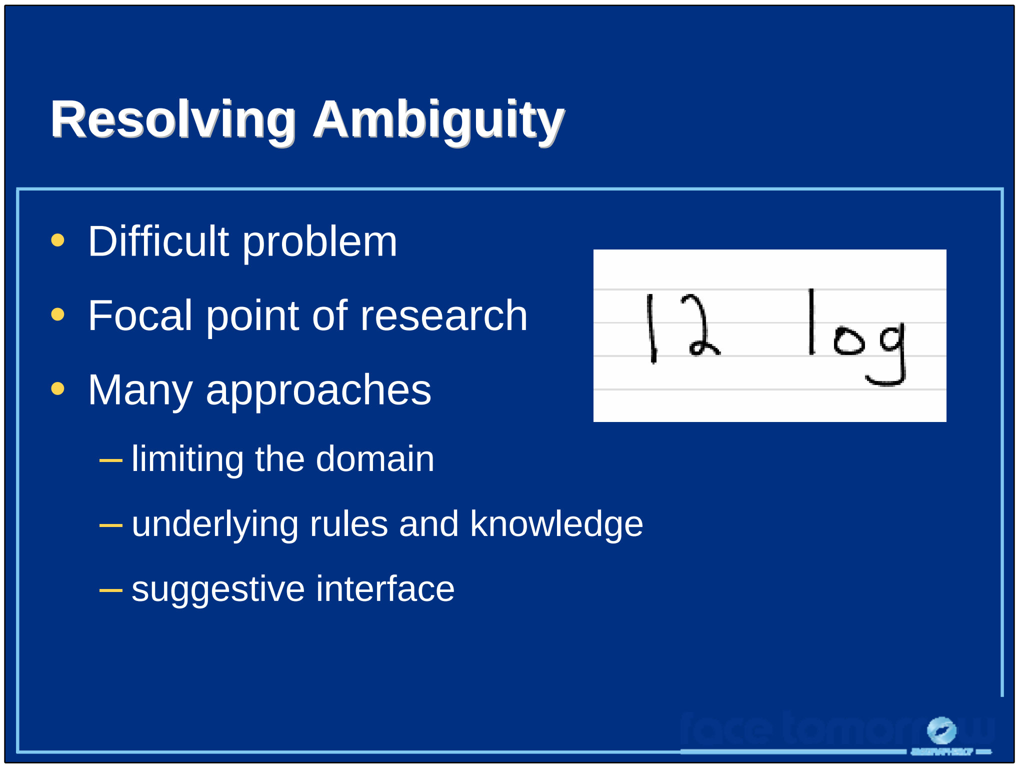




MathPad^2

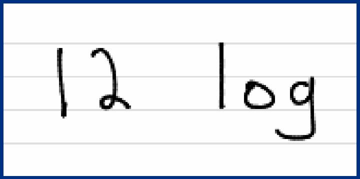
rationale.csail.mit.edu/project_earlyprocess.shtml

Regardless of what the sketch-based interface does, there will almost always be some form of recognition involved. The type and sophistication of the recognition algorithms used will vary depending on the application. Additionally, the way the recognizer is used will also vary depending on the application and tasks involved. The figure on the left shows the sketch and subsequent interpretation of the direction reversal mechanism for a walkman. In this case, the recognizer would take as input the complete drawing and output the interpreted result. The figure on the right shows several mathematical expressions. In this case, each expression is recognized individually using a lasso gesture. In other cases, real-time recognition can be performed where, as users enter strokes, the recognition algorithm is continuously running in the background trying to interpret each stroke as it is entered.

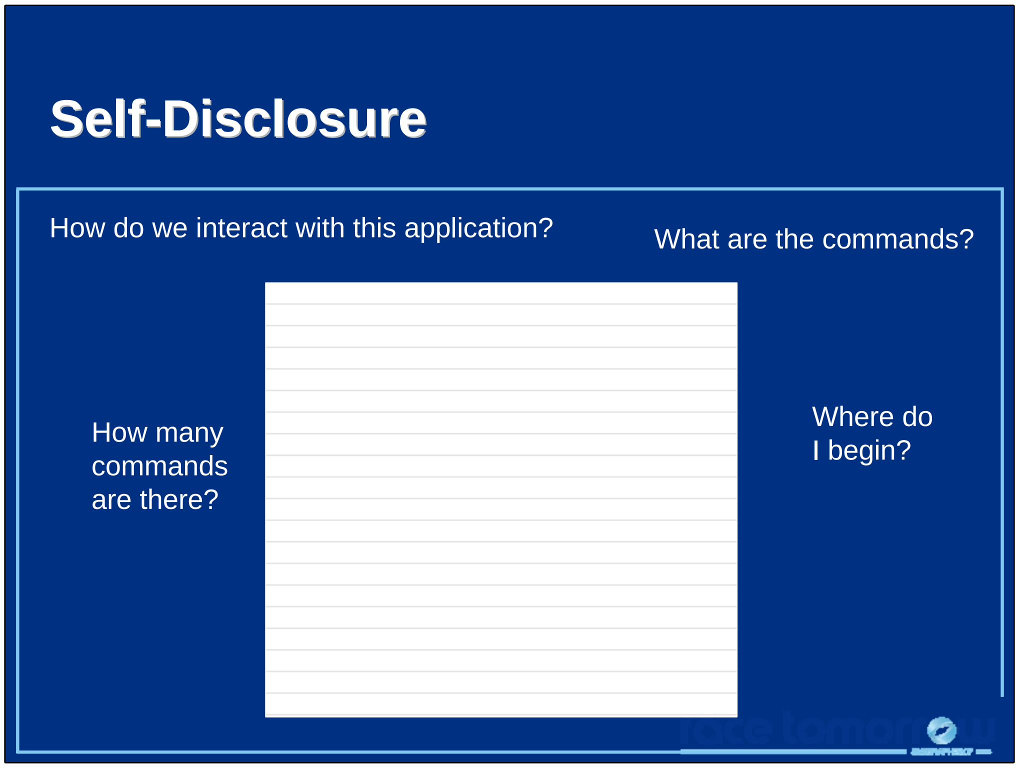


Resolving Ambiguity

- Difficult problem
- Focal point of research
- Many approaches
  - limiting the domain
  - underlying rules and knowledge
  - suggestive interface



Sketches are inherently ambiguous and resolving these ambiguities is both a difficult problem and a focal point of research in making sketch-based interfaces usable. For example, the figure in the slide shows the number *12* and the function *log.* Both the letter *l* and the number *1* can be written the same way making it very difficult to recognize them correctly. In this particular case, either the number *1* or the letter *l* is before the *og*. However, since the last two symbols are *og*, we are pretty confident that the first symbol in the function is an *l*. The first symbol in the number 12 is not as obvious. There are many different approaches to resolving ambiguities. One of the best ways to deal with this issue is to limit the sketch-based interface's domain. Limiting the domain can help to reduce the number of possible interpretations for a particular sketch. Another approach is to have additional knowledge about the domain and underlying rules embedded in the sketch interpretation scheme. In some cases, it might not be possible to resolve an ambiguity automatically. In these cases, asking the user for assistance can help by presenting several alternatives that the user can choose from. Dealing with ambiguities will be addressed further throughout the rest of the course and we also include the Mankoff, Hudson, and Abowd (2000) paper on techniques for ambiguity resolution in the course notes.



Self-Disclosure

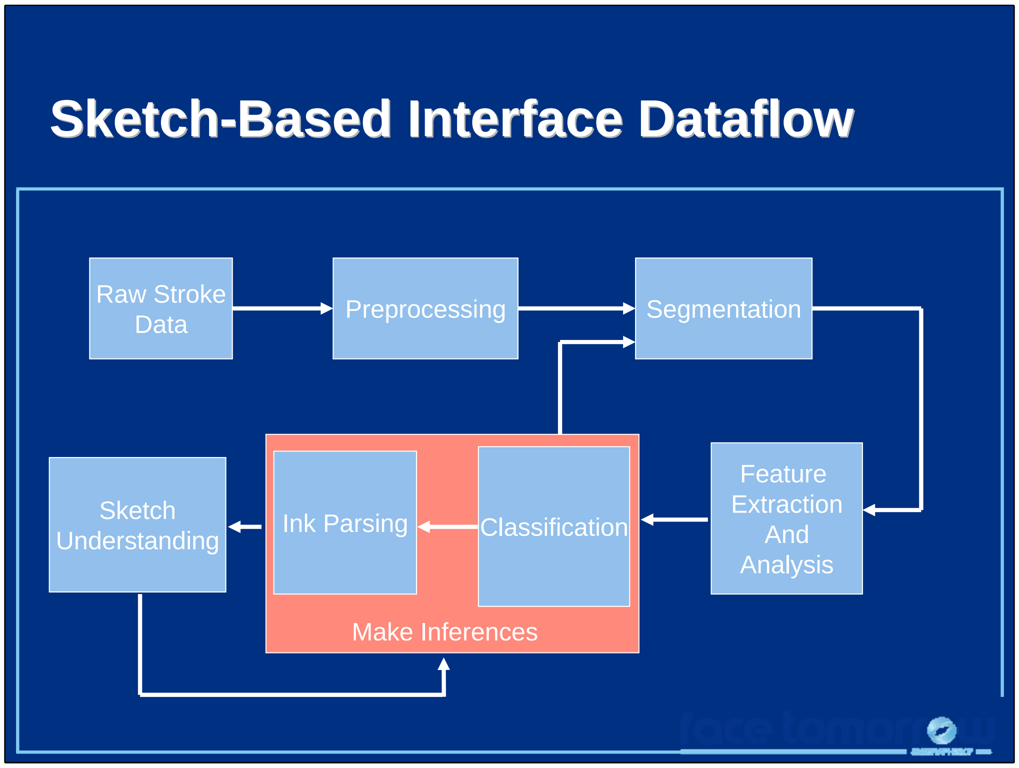
How do we interact with this application?

What are the commands?

How many commands are there?

Where do I begin?

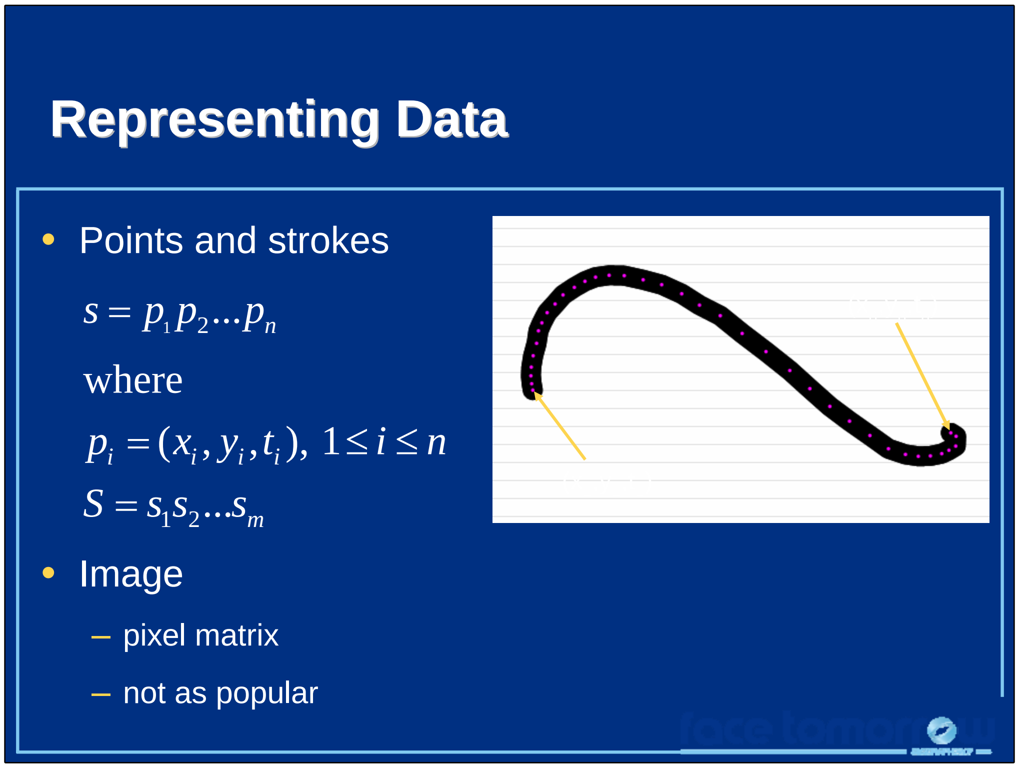
One area of research on sketch-based interfaces that has received little attention is how to teach users how to use a sketch-based or gestural interface. Sketch-based interfaces are typically not self-disclosing (more so for gestural interfaces). In other words, the application will start with just a blank screen or piece of virtual paper (like the figure in the slide). Although the application screen looks like a piece of paper (giving us our pencil-and-paper interaction metaphor) it does not tell us what and how many commands there are and where a user begins. Once the interface is learned (from a cheat sheet or tutor) self-disclosure is not an issue. One of the advantages of WIMP based interfaces is that a user can look at the application and have a pretty good idea on how to interact with the system. This is not the case with a sketch-based interface. Thus, presenting information on how to use the interface but still maintaining the pencil-and-paper look is an interesting challenge.



Sketch-Based Interface Dataflow

The stroke data that users enter in a sketch-based interface goes through a series of transformations on the way to being interpreted as a gesture or sketch. Typically, the raw data first undergoes a preprocessing step followed by a segmentation step that breaks up the stroke data into logical pieces. These segmented pieces are analyzed and important features are extracted from them. These features are then used in a classification and parsing (interpretation) step, ultimately leading to some form of sketch understanding. Note that in some cases, feedback loops occur. For example the results from the inference step (i.e., classification and ink parsing) can be fed back into the segmentation step to provide it with added knowledge to make better segmentation decisions. In another example, understanding part of a sketch can be used to help guide classification and ink parsing as new ink strokes are entered by the user.

In the next few slides we will examine each of these steps. Keep in mind that not all sketch-based interfaces use all of these transformations in the order presented in the slide. Some approaches, especially when dealing with gesture-based interfaces, may skip some of the steps, while others may perform the steps in a different order or may perform a particular step more than once. However, having said this, the dataflow model shown in the slide is a good way to think about the internal workings of a sketch-based interface.



Representing Data

- Points and strokes

  $s = p_1 p_2 \ldots p_n$

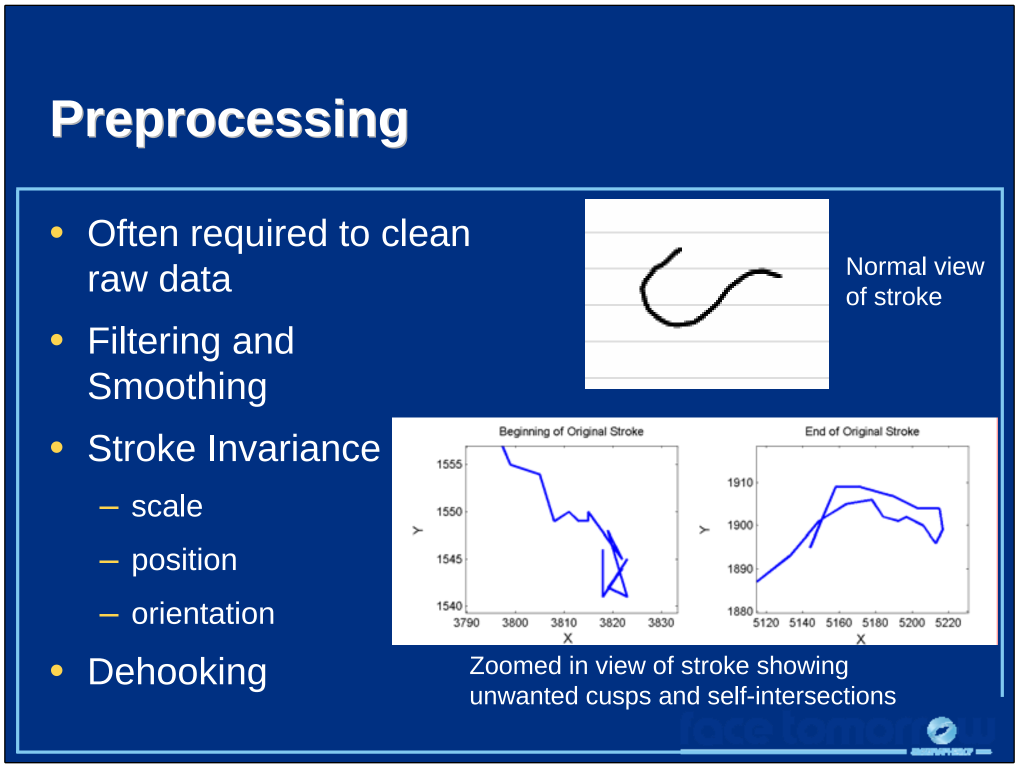
  where

  $p_i = (x_i, y_i, t_i),\ 1 \le i \le n$

  $S = s_1 s_2 \ldots s_m$

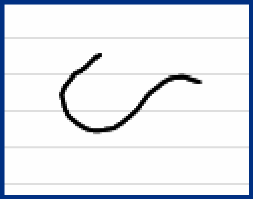
- Image
  - pixel matrix
  - not as popular

Raw stroke data is represented in one of two ways. The first is with strokes where each stroke is made up of a list of points. Each point has an x and y coordinate as well as a timestamp. Note that the timestamp is not always present. However, it can be very useful during feature extraction and sketch parsing. A sketch is then made up of a set of strokes. The second way to represent raw stoke data to treat them as images where strokes are just pixel data. This approach to representing the stroke data is not as popular as using points, but it does have some advantages. Specifically, computer vision algorithms can be used on images for segmentation and feature extraction. For our purposes, we will use the points and strokes approach.

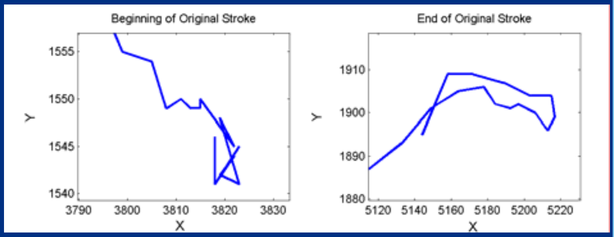


Preprocessing

- Often required to clean raw data
- Filtering and Smoothing
- Stroke Invariance
  - scale
  - position
  - orientation
- Dehooking



Normal view of stroke

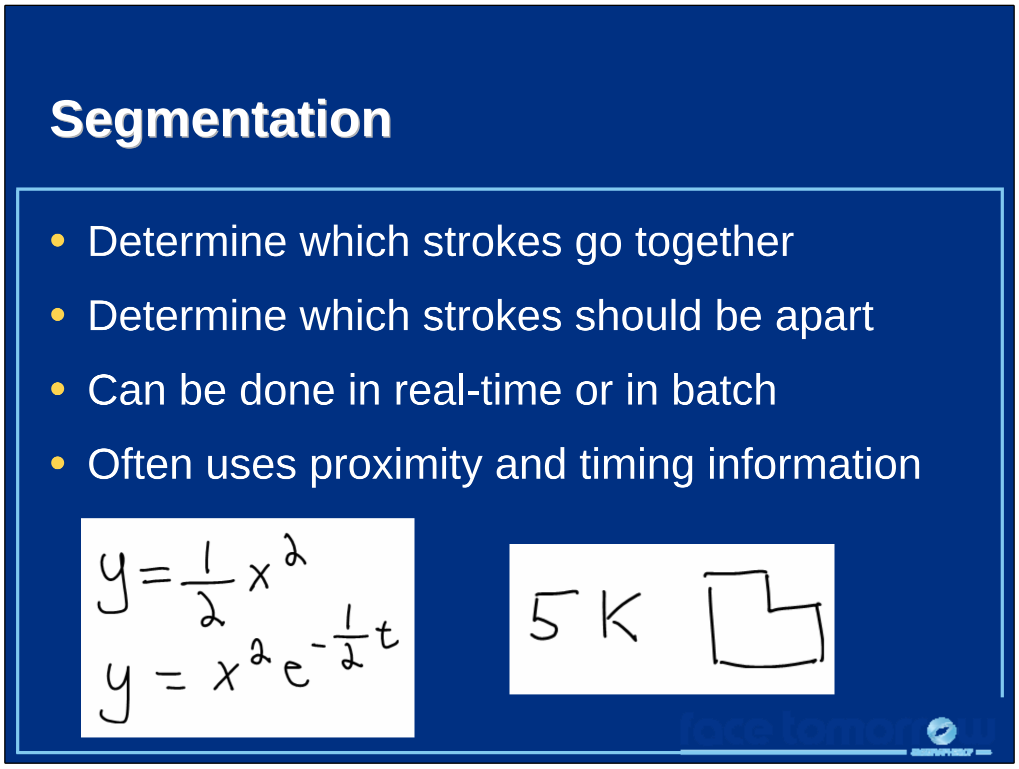


Zoomed in view of stroke showing unwanted cusps and self-intersections

In many cases, a preprocessing step is needed to get the raw stroke data ready for feature extraction and recognition. This step is very important when dealing with pen-based input devices, such as the stylus, especially when there is a high sampling rate. Preprocessing many not be required when dealing with mouse input. There are several types of data transformations that can be done during the preprocessing step. First, filtering and smoothing, using Gaussian or exponential filters for example, are used to get rid of subtle variations and noise in the strokes. Second, the strokes should be transformed to an invariant state. This involves moving strokes to a canonical position, size (while still maintaining aspect ratio), and, in some cases, orientation. Making the stroke invariant to scale, position and orientation, allows for feature extraction to remain consistent, regardless of the location, size and rotation of the ink strokes. Third, in many cases, strokes will have hooks at their endpoints that are caused when users lift the stylus off of the surface of the device. These hooks can cause problems in recognition because they may or may not be present depending on the particular user. Thus, getting rid of them is important for maintaining consistency in recognition. Other methods of preprocessing include maintaining stroke and direction ordering and slant invariance. Nice surveys on preprocessing techniques can be found in

Guerfali, W. and R. Plamondon, Normalizing and Restoring On-Line Handwriting, *Pattern Recognition*, 26(3):419-431, 1993.

Tappert, C.C., C.Y. Suen, and T. Wakahara, The state-of-the-art in on-line handwriting recognition, *IEEE Transactions Pattern Analysis Machine Intelligence*, Vol. PAMI-12, pp. 787-808, August 1990.



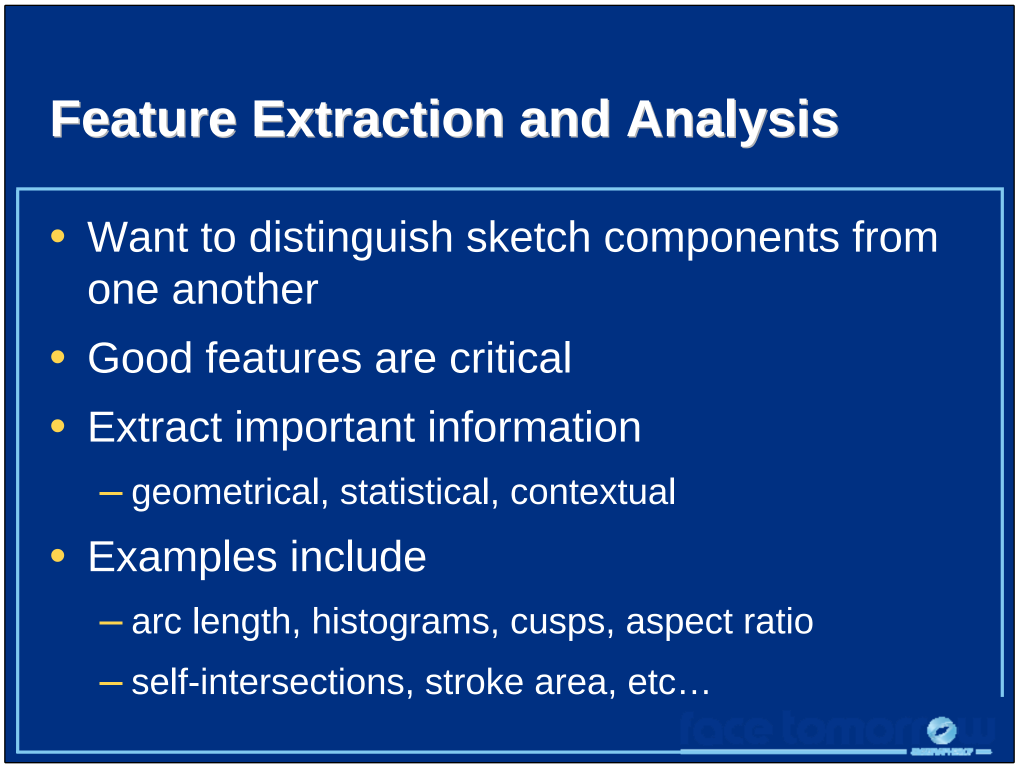
Segmentation

- Determine which strokes go together
- Determine which strokes should be apart
- Can be done in real-time or in batch
- Often uses proximity and timing information

$$y=\frac{1}{2}x^2$$
$$y=x^2e^{-\frac{1}{2}t}$$

5 K

After the preprocessing step, some form of segmentation is performed. Segmentation is one of those steps in the data flow diagram that has some freedom as to where it gets done It could be done before preprocessing, for example. Segmentation could be done in real-time as the user makes a sketch or all at once after the sketch is complete. Segmentation not only involves finding which strokes should go together, such as the 5, k, and the geometric object shown on the right of the slide, but also determining which strokes show be apart, like the separate mathematical expressions shown on the left of the slide. Basic techniques for performing segmentation use proximity or timing information or sometimes both at the same time.   The segmentation algorithm can check to see if strokes are touching or have any part of close to each other. Additionally, strokes that have been written, one after another, may be treated as one symbol given their proximity.

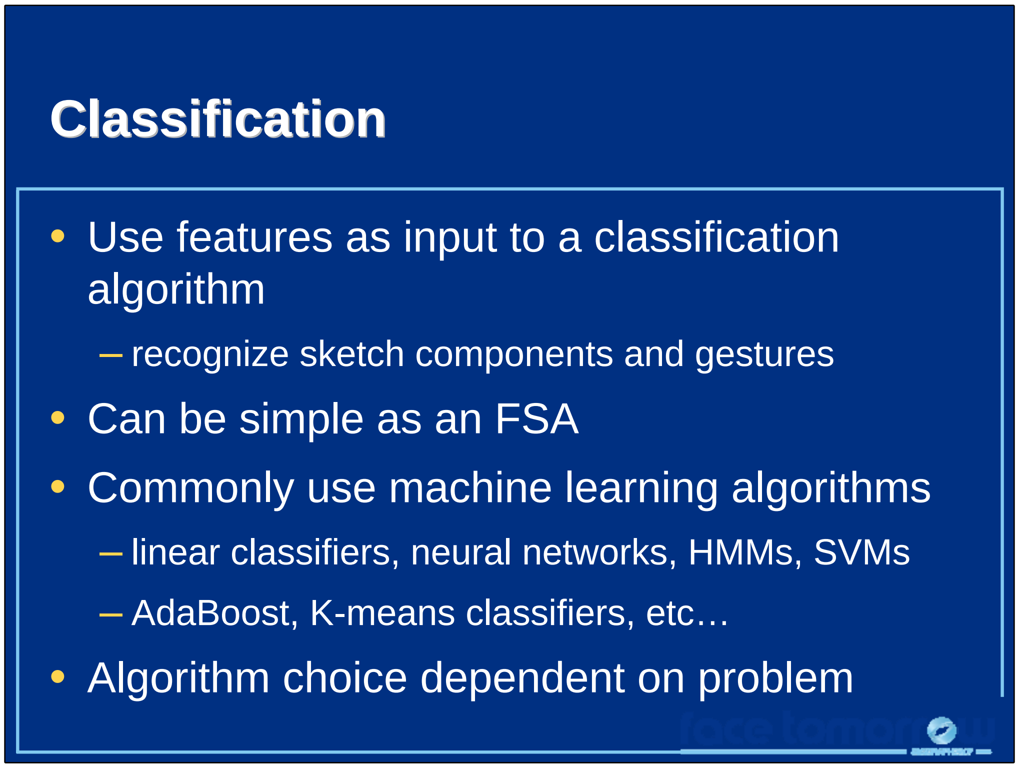


Feature Extraction and Analysis

- Want to distinguish sketch components from one another
- Good features are critical
- Extract important information
  - geometrical, statistical, contextual
- Examples include
  - arc length, histograms, cusps, aspect ratio
  - self-intersections, stroke area, etc…

Once the strokes have been preprocessed and properly segmented, the feature extraction step is performed. These features are used as input to a recognition algorithm since we want to distinguish the different pieces of the sketch from one another (or one gesture from another). Finding good features to use is critical in having good recognition, and there are many different possibilities. Features based on stroke geometry, statistics and context have all been used as input to recognizers. Features based on stroke geometry include arc length, cusps, aspect ratio, and self-intersections. Examples of statistically-based features include angle and point histograms. Contextual features include where a symbol or sketch component is in relation to other components such as two lines being parallel. Note that is possible that a good set of features can be used to perform recognition for a small set of symbols or components without the need for sophisticated machine classification algorithms. A nice list of features used for distinguishing strokes from one another can be found in chapter 5 of

LaViola, J. *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*, Ph.D. Dissertation, Brown University, Department of Computer Science, May 2005.



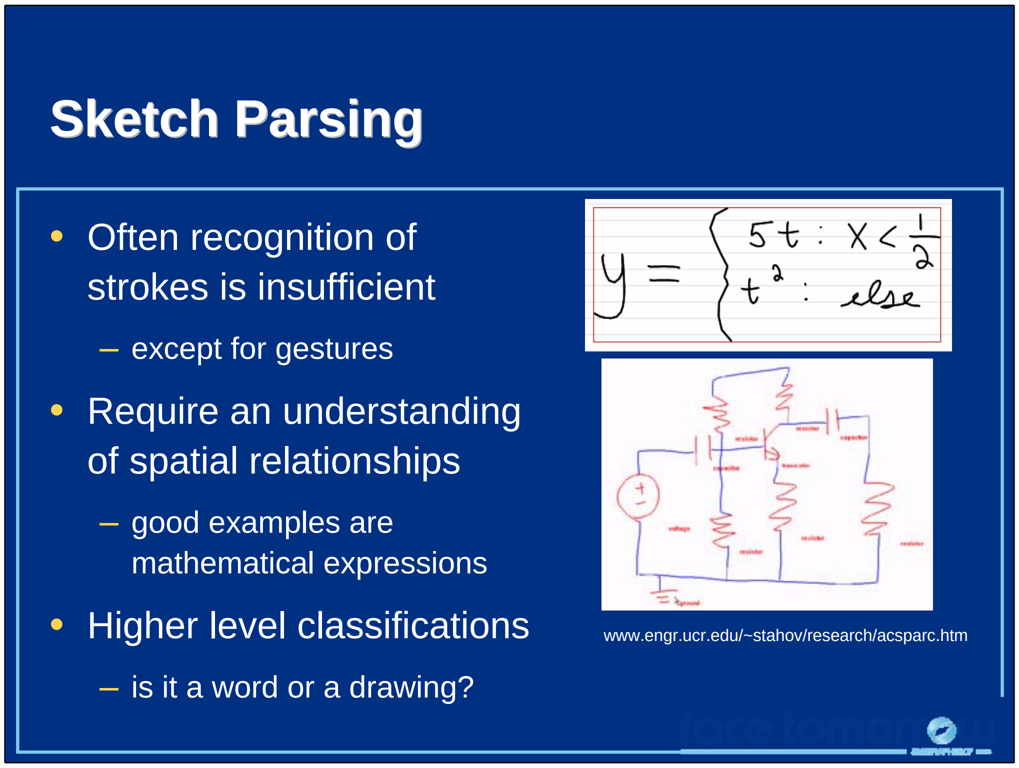
Classification

- Use features as input to a classification algorithm
  - recognize sketch components and gestures
- Can be simple as an FSA
- Commonly use machine learning algorithms
  - linear classifiers, neural networks, HMMs, SVMs
  - AdaBoost, K-means classifiers, etc…
- Algorithm choice dependent on problem

The features collected from the last step are used as input to a classification algorithm which will recognize sketch components and gestures. The choice of classification algorithm is really dependent on the problem. A classification can be as simple as a finite state machine (FSM). For example the SKETCH system used an FSM for gesture recognition. As the number and complexity of the sketch or gesture set increases, the more sophisticated the classification algorithm needs to be. There are many different classification algorithms that could be used, and they all have advantages and disadvantages. Going into detail on these algorithms is beyond the scope of this course, but Pattern Classification, by Duda et al., is a excellent resource.

References:

Duda, Richard O., Peter E. Hart, and David G. Stork. *Pattern Classification*, John Wiley and Sons, 2001.



Sketch Parsing

- Often recognition of strokes is insufficient
  - except for gestures
- Require an understanding of spatial relationships
  - good examples are mathematical expressions
- Higher level classifications
  - is it a word or a drawing?

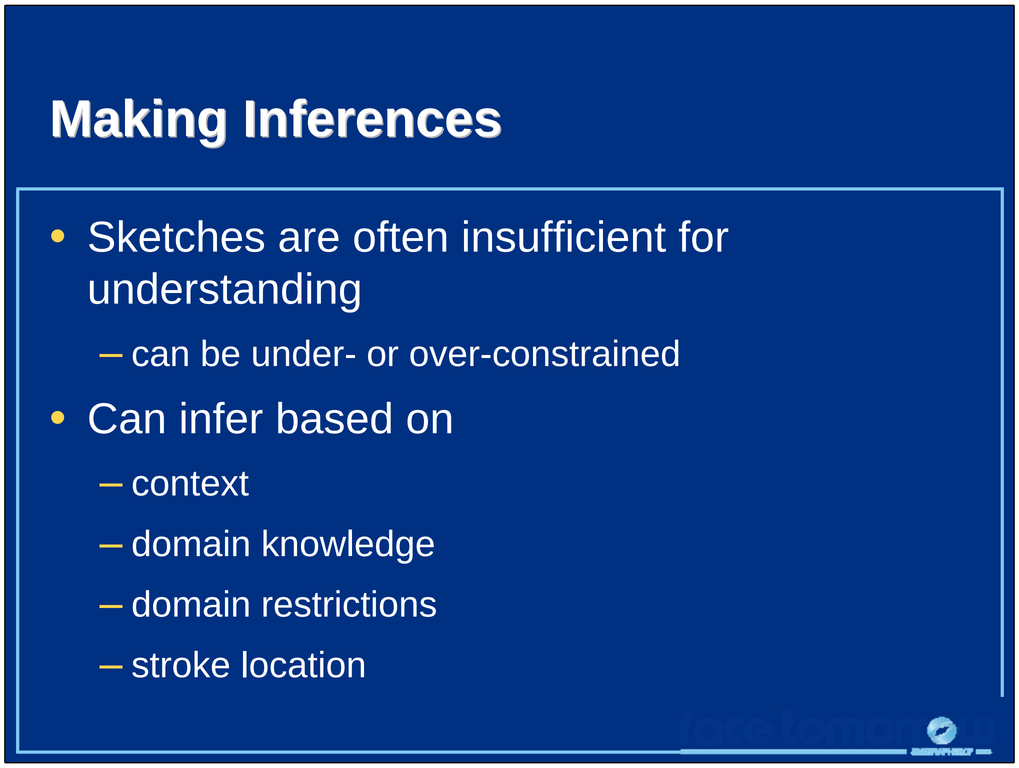
$$y = \begin{cases} 5t : x < \frac{1}{2} \\ t^2 : else \end{cases}$$

www.engr.ucr.edu/~stahov/research/acsparc.htm

Except for simple gestural command systems, recognizing symbols and sketch components is often insufficient. What is required is an understanding of the spatial relationships between sketch components so we can parse the sketch to obtain sketch understanding. Even some more complicated gesture-based interfaces require a form of sketch parsing. A great example of where sketch parsing is critical is when recognizing mathematical expressions. The expression shown in the top right requires more than just the recognition of the individual symbols. It also needs to be parsed so syntactic and semantic meanings can be extracted. In another example, the circuit diagram also needs to be parsed, since the interface needs to know how the individual components fit together to make a correctly formed diagram. Other, higher level classifications can also be done in the sketch parsing step. For example, the sketch parsing step could look to see whether a sketch is a group of words, mathematics, or a drawing. There are many different approaches to sketch parsing. One nice reference is the survey by Blostein and Grbavec. Although the techniques presented are focused on parsing mathematical expressions, they are applicable to more general parsing.

Reference:

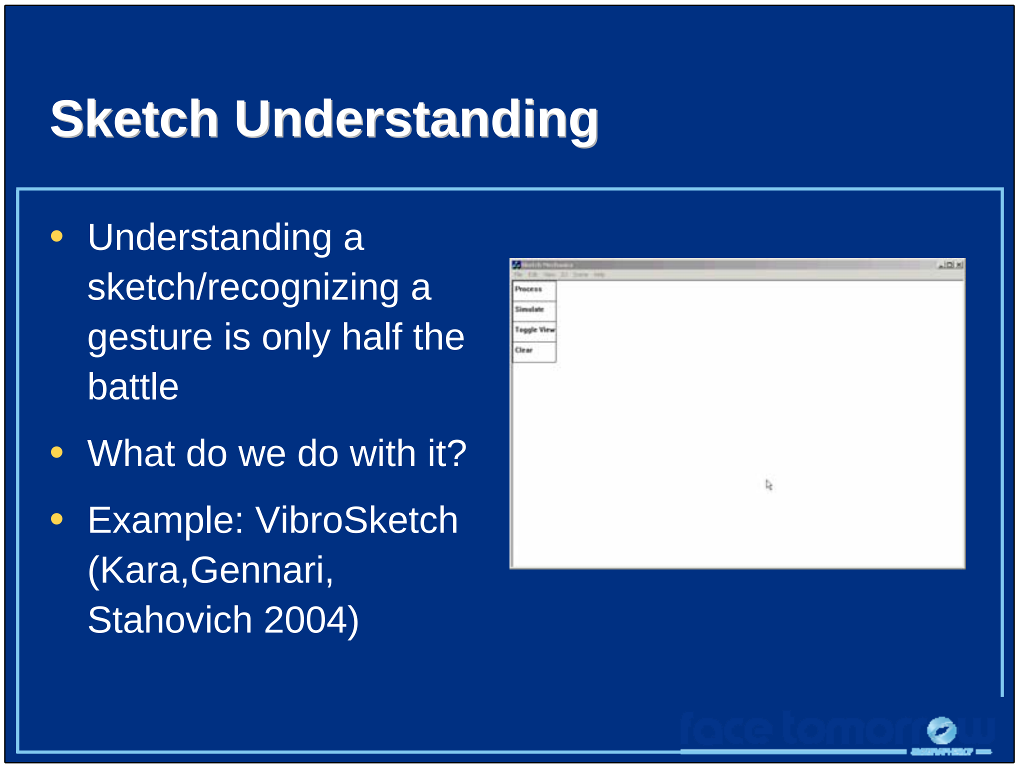
Blostein, D. and A. Grbavec. Recognition of Mathematical Notation. *Handbook on Optical Character Recognition and Document Image Analysis*, eds. P.S.P. Wang and H. Bunke, World Scientific Press, 557-582, 1997.



Making Inferences

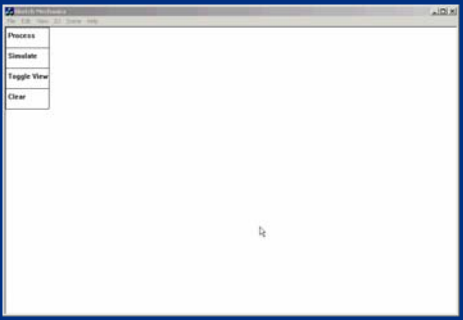
- Sketches are often insufficient for understanding
  - can be under- or over-constrained
- Can infer based on
  - context
  - domain knowledge
  - domain restrictions
  - stroke location

During the recognition and parsing stages, inferences are often required to deal with ambiguities or the lack of knowledge about the strokes in a sketch so it can be completely understood. In some cases, the sketch will be under-constrained (i.e., we do not have enough information to recognize and/or parse a sketch component) or over-constrained (i.e., there are multiple recognitions and parses). In either case, making inferences can help resolve these problems. A sketch interpretation system can infer information from a variety of different sources such as context, domain knowledge, and stroke location. For example, a gesture can be used for more than one command, given its location relative to other objects. In another example, we could restrict the domain we are interested in to just simple spring and pulley systems. This restriction would then help us to reduce the number of possible interpretations available during the recognition and parsing steps. More details on making inferences will be provided in Dr. Davis's lecture on sketch understanding.



Sketch Understanding

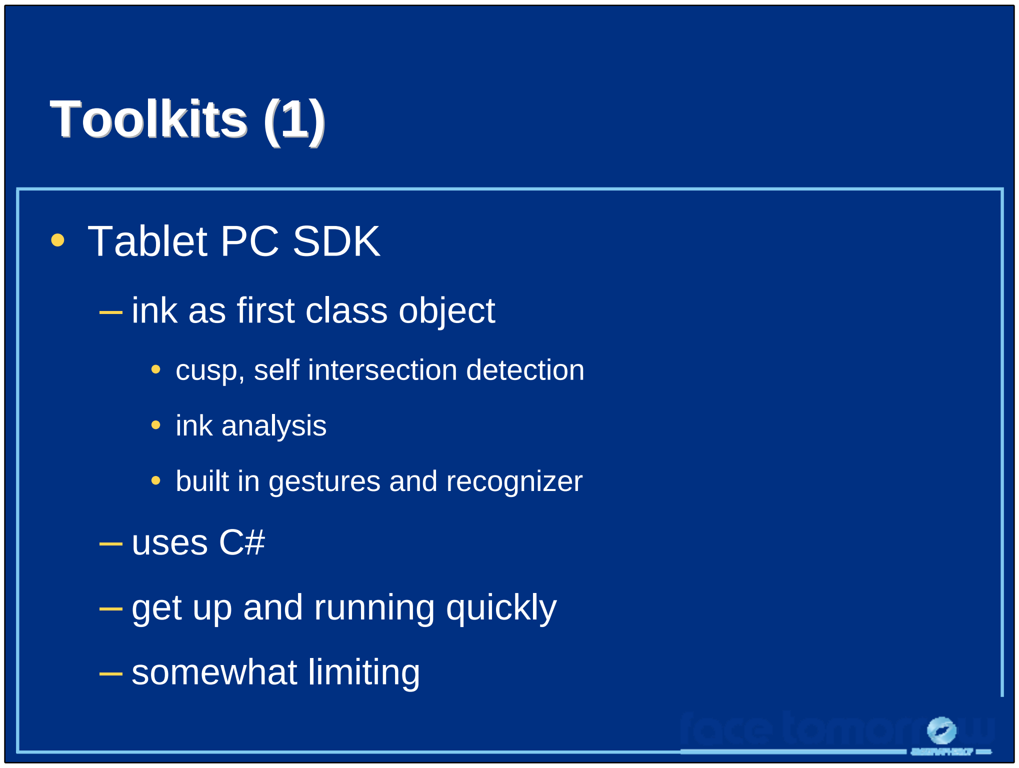
- Understanding a sketch/recognizing a gesture is only half the battle
- What do we do with it?
- Example: VibroSketch (Kara,Gennari, Stahovich 2004)



After we have gone through these sketch interpretation steps (and all has gone well), we will have some form of sketch understanding. At this point, the real challenge begins in what we to do next. We need to take our new knowledge and perform some action or task that will help the user in some way. A nice example showing a complete sketch-based interface, from beginning to end, is VibroSketch (Kara, Gennari, Stahovich 2004). You will see many more examples throughout the rest of the course and the videos we show are included in your course notes.

References:

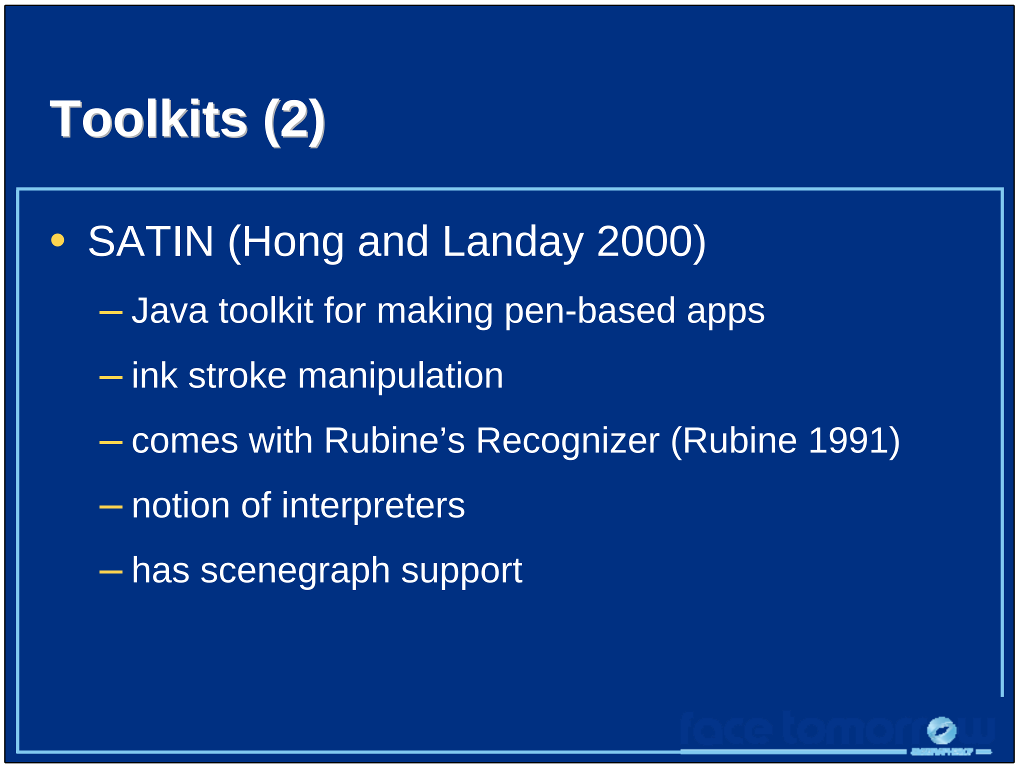
Kara L. B., L. Gennari, and T. F. Stahovich. A Sketch-Based Interface for the Design and Analysis of Simple Vibratory Mechanical Systems, *2004 ASME International Design Engineering Technical Conferences (ASME/ DETC 2004).*



Toolkits (1)

- Tablet PC SDK
  - ink as first class object
    - cusp, self intersection detection
    - ink analysis
    - built in gestures and recognizer
  - uses C#
  - get up and running quickly
  - somewhat limiting

Getting started in designing and developing sketch-based interfaces can seem like a daunting task. However, there are some toolkits that exist that can help and we will mention some of them here. The first one is the Tablet PC SDK. It is designed to run on Tablet PCs and uses C#. It treats ink as a first class object and has some nice routines for determining cusps, self-intersections, and whether strokes are contained within other strokes. It also uses the Microsoft handwriting recognizer and has a built in gesture recognizer. It also has rudimentary ink analysis, giving it the ability to tell whether a collection of strokes is a word or a drawing. The nice thing about the SDK is it is easy to use and you can get applications up and running very quickly because it has support for event handling and dealing with other MS Windows components. It is somewhat limiting in that it will only take you so far. The gesture recognizer uses a standard set of gestures and the recognizer only deals with characters and words. The Tablet PC SDK can be downloaded on Microsoft's web page.



Toolkits (2)

- SATIN (Hong and Landay 2000)
  - Java toolkit for making pen-based apps
  - ink stroke manipulation
  - comes with Rubine’s Recognizer (Rubine 1991)
  - notion of interpreters
  - has scenegraph support

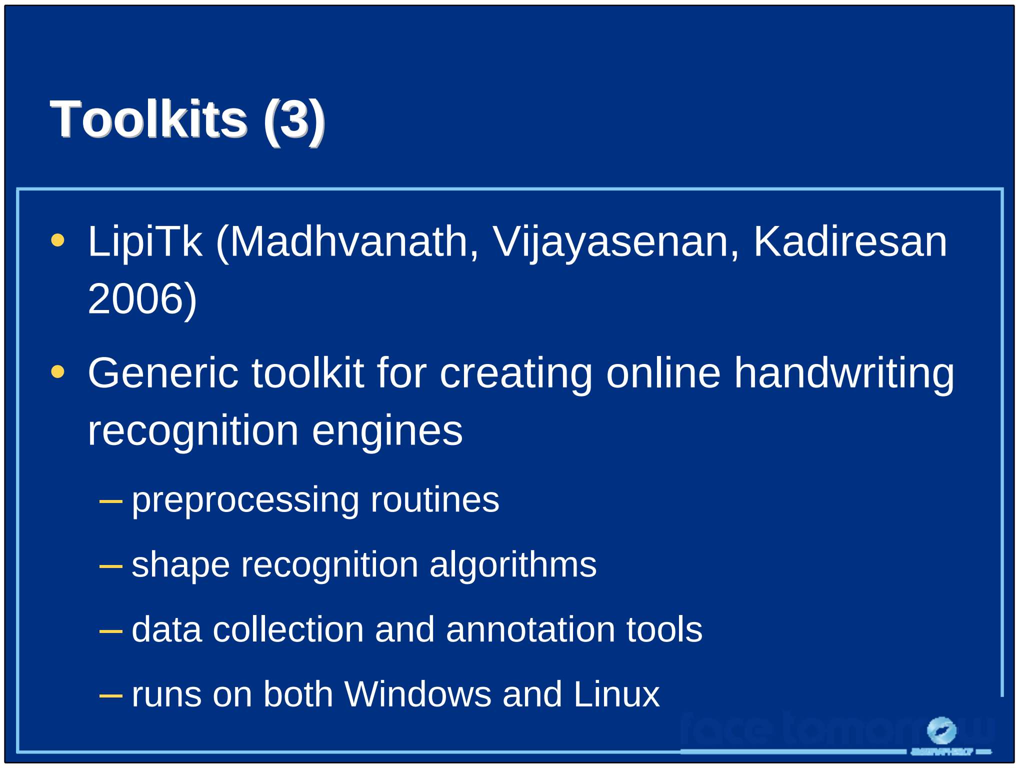
Another toolkit that can get designers and developers started is SATIN, developed by Jason Hong and James Landay. SATIN is a Java-based toolkit for making pen-based applications. It has support for ink stroke manipulation, scenegraphs, and uses the notion of interpreters for easy extensibility. It also comes with Rubine’s recognizer adapted from his 1991 SIGGRAPH paper. SATIN can be downloaded at

guir.berkeley.edu/projects/satin/

References:

Hong, J. and J. Landay, SATIN: A Toolkit for Informal Ink-based Applications, ACM Symposium on User Interface Software and Technology, *CHI Letters*, 2(2):63-72, 2000.

Rubine, Dean. Specifying Gestures by Example. *Proceedings of SIGGRAPH'91*, ACM Press, 329-337, 1991.



Toolkits (3)

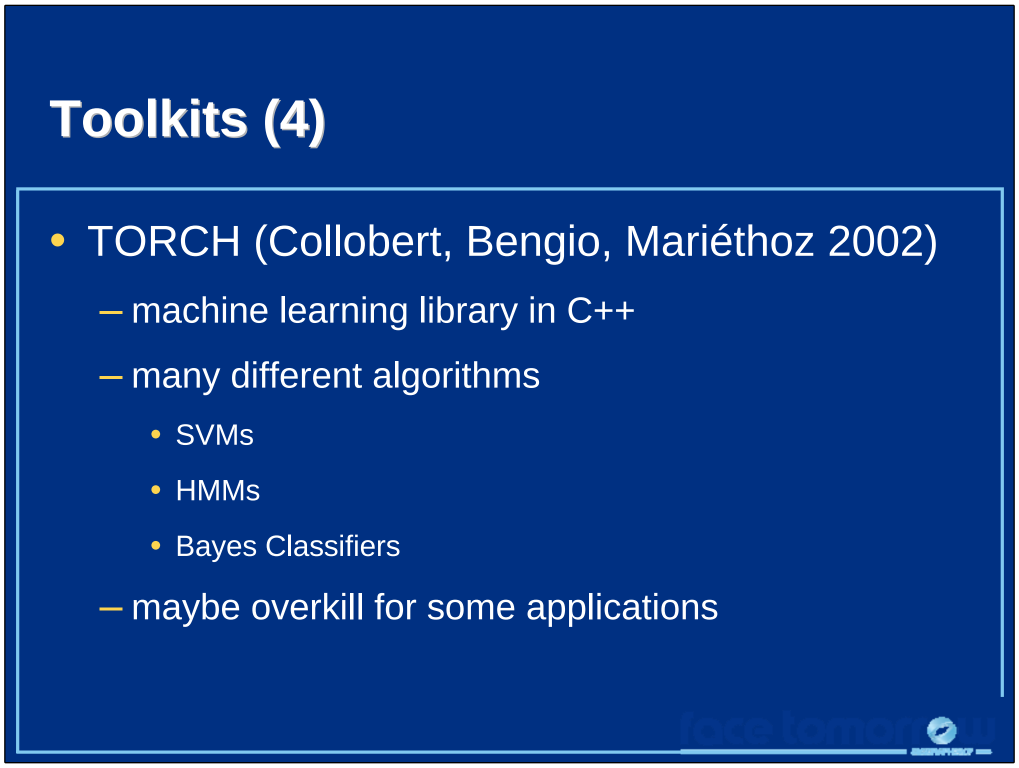
- LipiTk (Madhvanath, Vijayasenan, Kadiresan 2006)
- Generic toolkit for creating online handwriting recognition engines
  - preprocessing routines
  - shape recognition algorithms
  - data collection and annotation tools
  - runs on both Windows and Linux

LipiTk is a toolkit developed at HP Labs, India by Madhvanath, Vijayasenan, and Kadiresan, designed to assist in the development of new online handwriting recognizers. The toolkit provides implementations of tools, scripts and sample code required to support handwriting data collection and annotation, recognizer training and evaluation, and packaging of engines and their integration into pen-based applications. It includes preprocessing routines for smoothing, size normalization, dehooking, and equidistant resampling. It also includes two shape recognition algorithms using subspace-based classification and nearest-neighbor classification based on Dynamic Time Warping. Although the toolkit was developed primarily for handwriting recognition, it has enough generic functionality to be used for creating more general sketch and gesture-based interfaces.

It is freely available on SourceForge at http://lipitk.sourceforge.net/.

References:

Madhvanath, S., D. Vijayasenan, and T. M. Kadiresan, LipiTk: A Generic Toolkit for Online Handwriting Recognition, Tenth International Workshop on Frontiers in Handwriting Recognition, 2006.



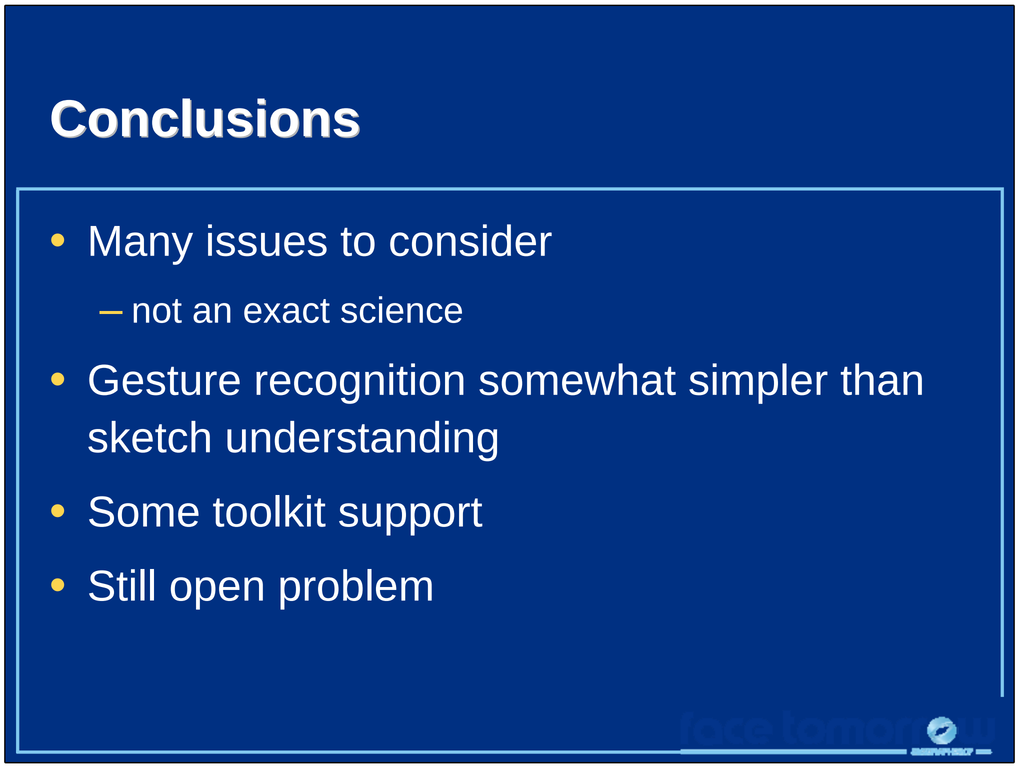
Toolkits (4)

- TORCH (Collobert, Bengio, Mariéthoz 2002)
  - machine learning library in C++
  - many different algorithms
    - SVMs
    - HMMs
    - Bayes Classifiers
  - maybe overkill for some applications

TORCH is a machine learning library written in C++. Although it has nothing to do with making sketch-based interfaces or pen-based applications, it provides many different machine learning algorithms that can be uses for recognition. Thus, using the library can help reduce the amount of development time because the recognition portion of the sketch-based interface will be already taken care of. TORCH can be downloaded at www.torch.ch/

References:

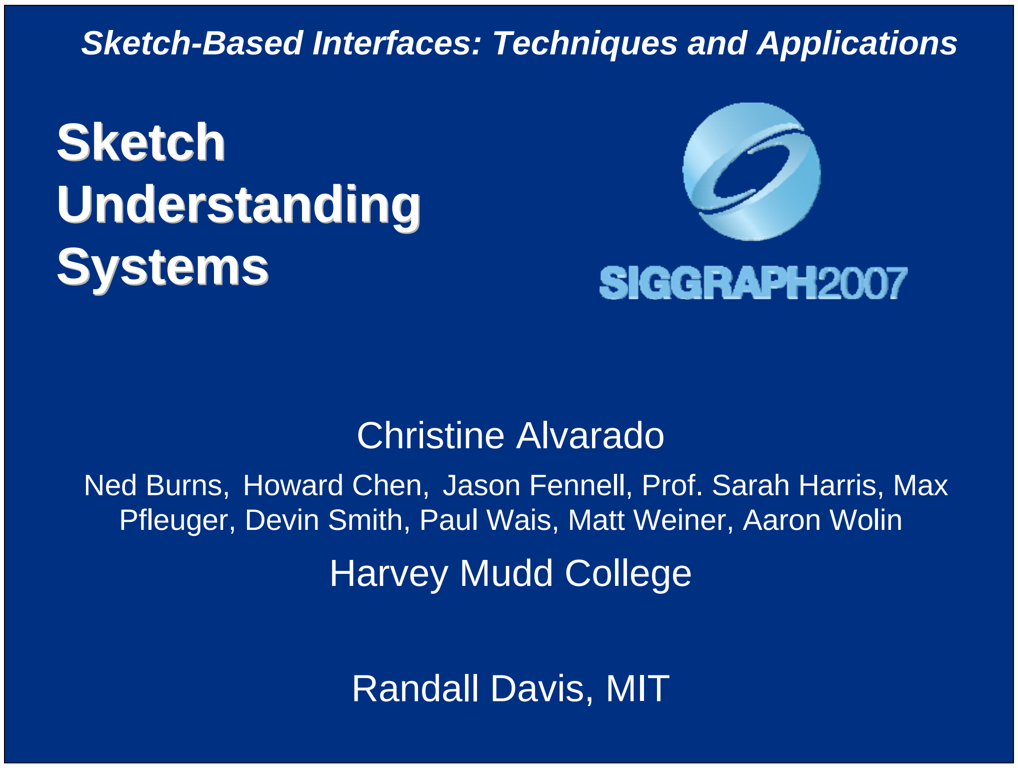
R. Collobert, S. Bengio, and J. Mariéthoz*. Torch: a modular machine learning software library.* Technical Report IDIAP-RR 02-46, IDIAP, 2002.



Conclusions

- Many issues to consider
  - not an exact science
- Gesture recognition somewhat simpler than sketch understanding
- Some toolkit support
- Still open problem

In conclusion, there are many issues to consider when building a sketch-based interface with the decisions for dealing with them dependent on the application and the interface's. Gesture recognition is somewhat simpler than sketch understanding but it still can be challenging when the gesture set is large. There is some toolkit support for developers and designers who want to get started in sketch-based interfaces. All in all, making sketch-based interfaces still has many open issues and remains an interesting research problem.



Sketch-Based Interfaces: Techniques and Applications

Sketch Understanding Systems

SIGGRAPH2007

Christine Alvarado

Ned Burns, Howard Chen, Jason Fennell, Prof. Sarah Harris, Max Pfleuger, Devin Smith, Paul Wais, Matt Weiner, Aaron Wolin

Harvey Mudd College

Randall Davis, MIT

Sketch Understanding Systems

I will be discussing the problem of *sketch understanding* which, as we shall see shortly, involves much more than just recognizing sketched symbols. Most of the work that I will present has been done by myself and the other members of my former research group back at MIT, lead by Professor Randy Davis. In addition, I would like to acknowledge my current team of undergraduate researchers who are helping me explore these problems further.

My Contact information:

Christine Alvarado

Assistant Professor

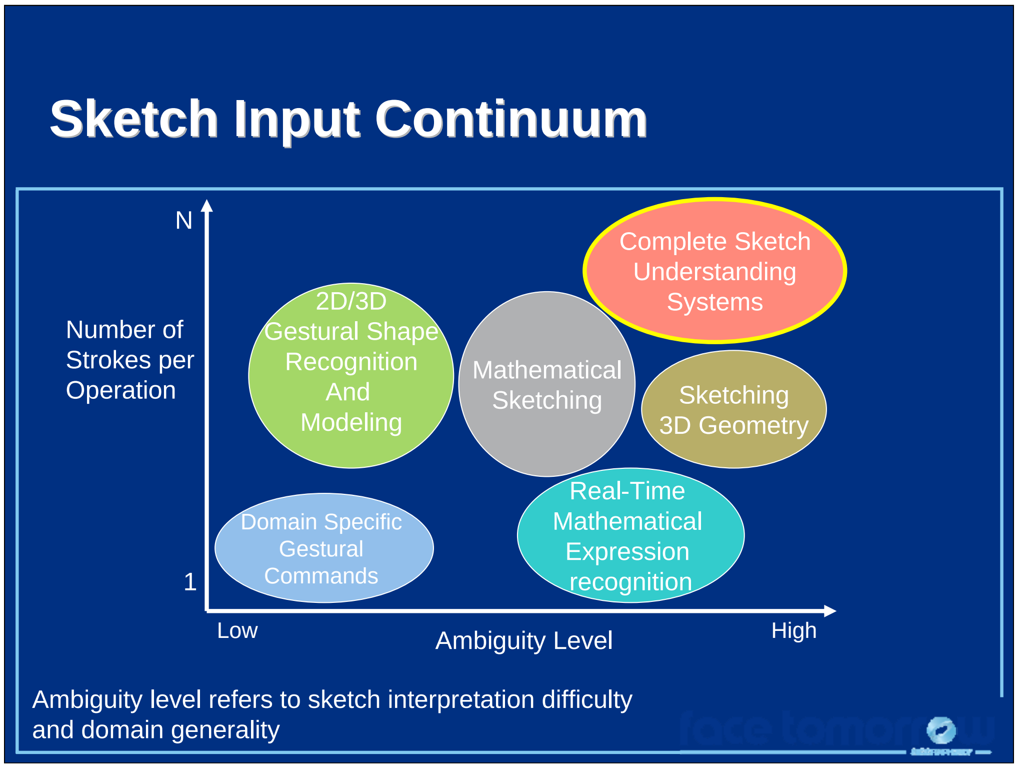
Harvey Mudd College, CS Department

1250 N. Dartmouth Ave.

Claremont, CA 91711

Email: alvarado@cs.hmc.edu

http://www.cs.hmc.edu/~alvarado/

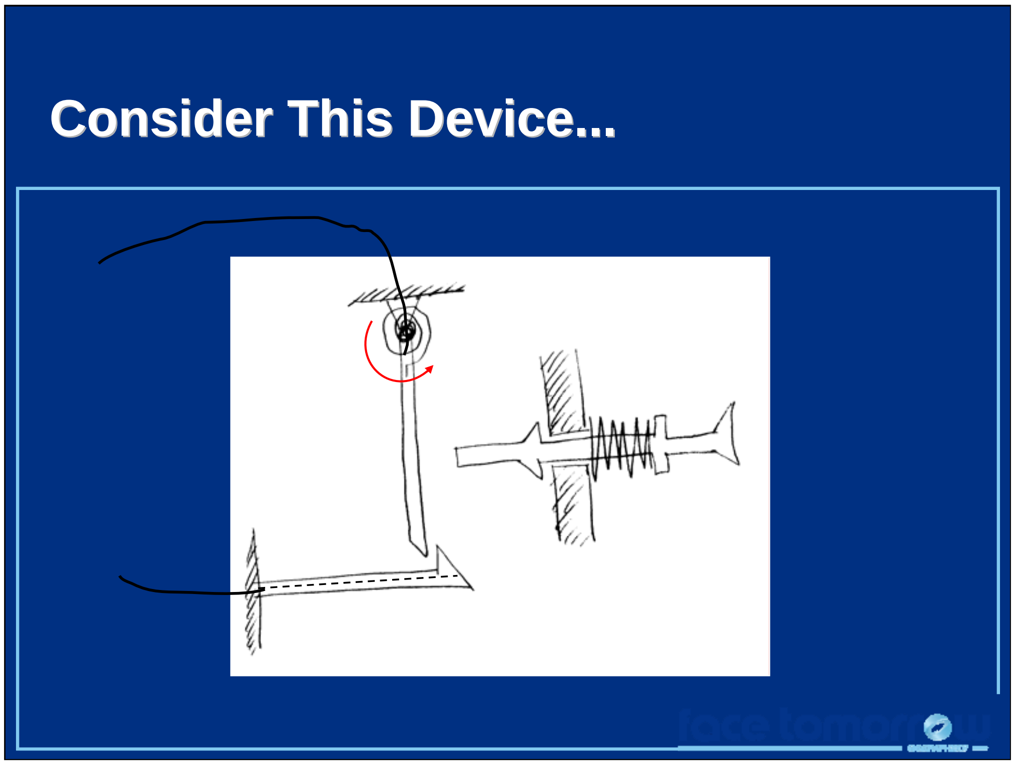


Sketch Input Continuum

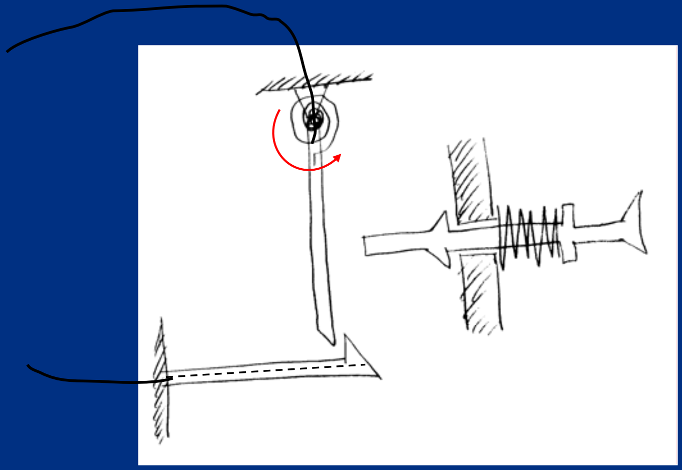
N
Number of Strokes per Operation
1
Low
Ambiguity Level
High
Complete Sketch Understanding Systems
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition

Ambiguity level refers to sketch interpretation difficulty and domain generality

Complete sketch understanding systems are the "holy grail" of this area of research. Unfortunately, they are also the most difficult to construct given the high ambiguity and the large (and variable) number of strokes per operation.

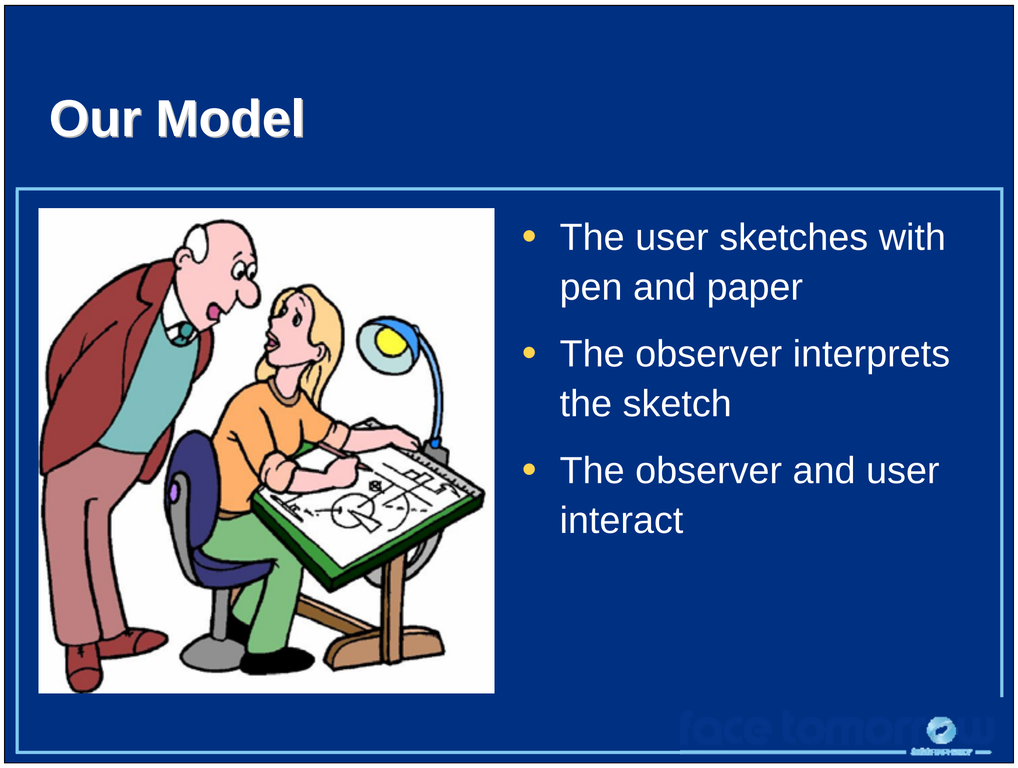


Consider This Device...



Sketches are powerful because of the amount of information they convey. Consider this sketch of a circuit breaker. How does it work? If you understand the diagrams you will see immediately how it functions. The bottom component is a bi-metallic strip. When it gets too hot, it bends down and the circular spring causes the upper strip to rotate out to break the circuit.

The diagram along with a few simple words helps you understand completely what is going on. You can now probably see how to reset the circuit. In fact, you probably have a little movie that plays in your head of what happens when the circuit is broken and reset. A diagram conveys this information in way no words could.

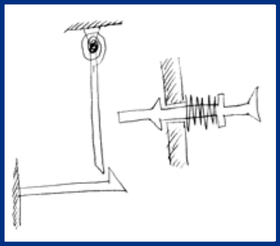


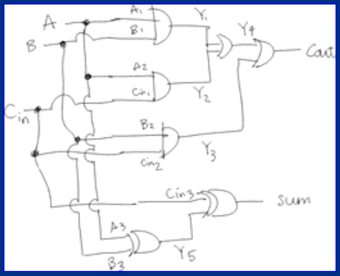
Our Model

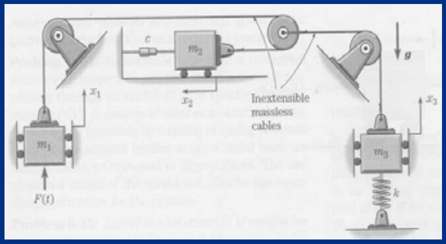
- The user sketches with pen and paper
- The observer interprets the sketch
- The observer and user interact

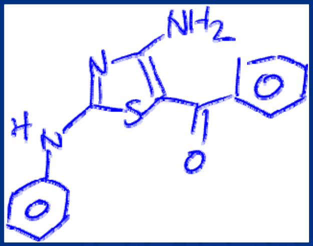
Our goal is to build a system that understands sketches the way that you and I do and can interact with the user in the same way a human observer might—to learn how a device works, to offer feedback on a design, etc. We would like to build a system that can literally "look over the user's shoulder" as she draws and interact with her seamlessly about her design.

Sketches are ***Dead***





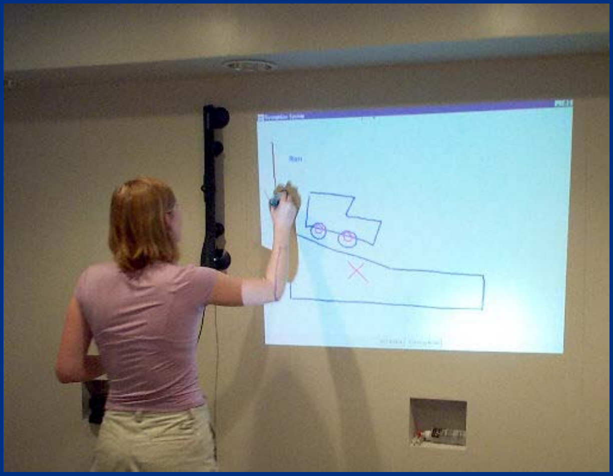




Because of their power, diagrams are used in many domains. Pen and paper provide a natural, seamless way for people to create diagrams, so users often sketch diagrams throughout the design process.

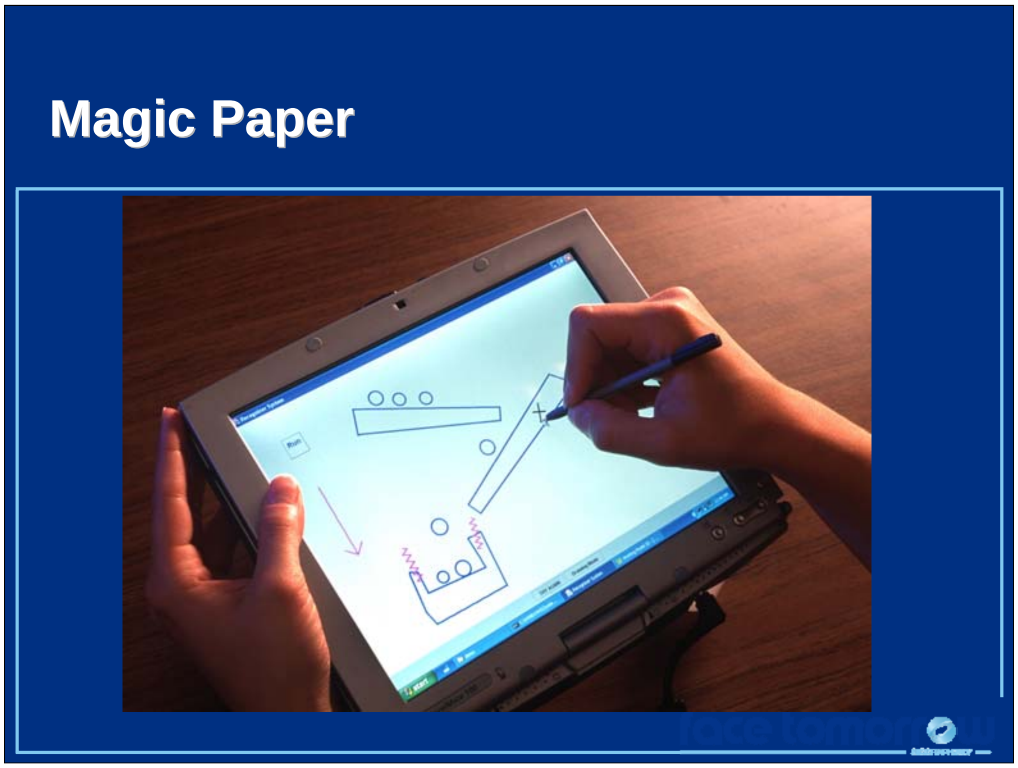
Unfortunately, diagrams drawn on paper (or drawn on the computer but not interpreted) are dead. If the computer does not understand them, they are nothing more than pixels on the screen.

Magic Paper

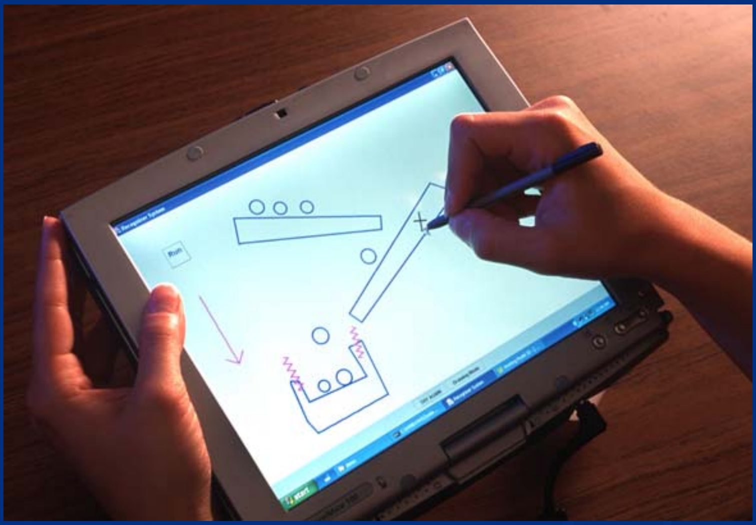


Our dream is to build a system that is as free and natural to use as paper, but that brings a diagram to life by understanding the domain-specific meaning behind the strokes.

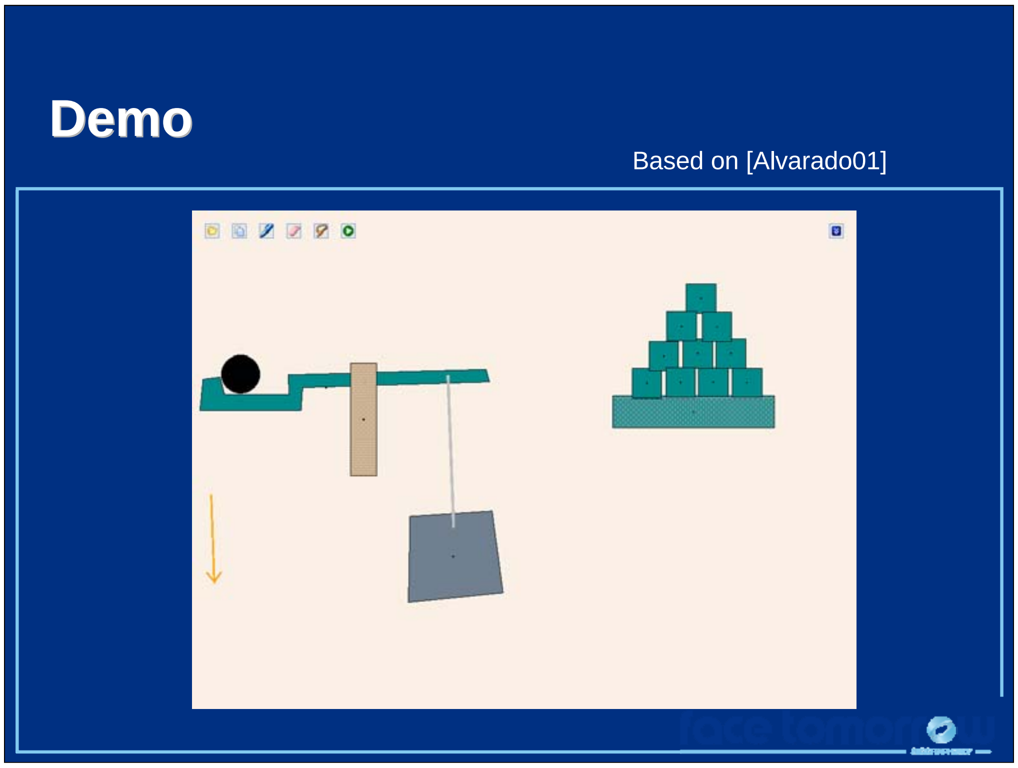
Of course, "paper" is a flexible concept in the electronic world. We would like the user to be able to sketch in the medium that makes most sense to her, whether on the whiteboard…



Magic Paper



… or on a tablet computer. The important piece is the software behind the interface that can understand the users strokes as she draws.


Demo

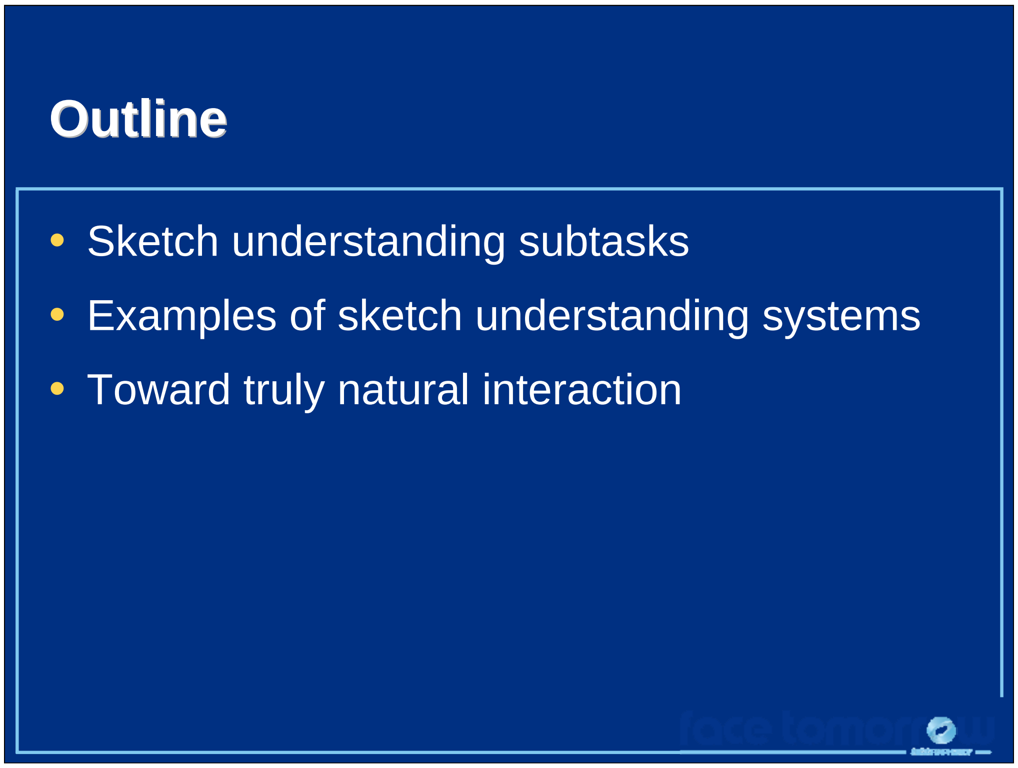
Based on [Alvarado01]

This is a demo of the Microsoft Physics Illustrator, a power toy for the Tablet PC. This application was developed based on the ASSIST sketch understanding system. It allows a user to sketch freely on the Tablet computer and then literally see their designs come to life when they press the play button.

References:

Microsoft Physics Illustrator for Tablet PC: http://www.microsoft.com/downloads/details.aspx?familyid=56347faf-a639-4f3b-9b87-1487fd4b5a53&displaylang=en

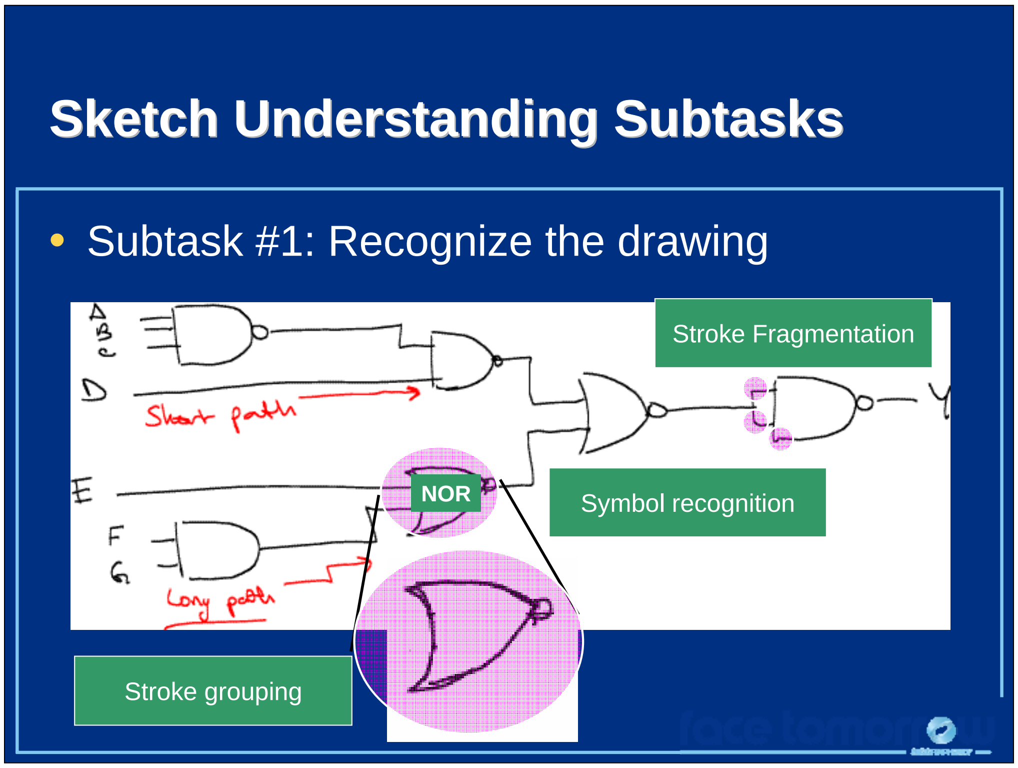
Christine Alvarado and Randall Davis. Resolving Ambiguities to Create a Natural Sketch Based Interface. In Proceedings. of IJCAI-2001. August 2001.



Outline

- Sketch understanding subtasks
- Examples of sketch understanding systems
- Toward truly natural interaction

Sketch understanding is more than just recognizing the symbols in a diagram. I will begin by talking about the many subtasks involved in complete sketch *understanding*. Then I will describe a number of systems that address one or more of the various sketch understanding subtasks. Finally, I will describe briefly the state of the art and where we are in our quest to build a system that understands freely hand-drawn diagrams.



Sketch Understanding Subtasks

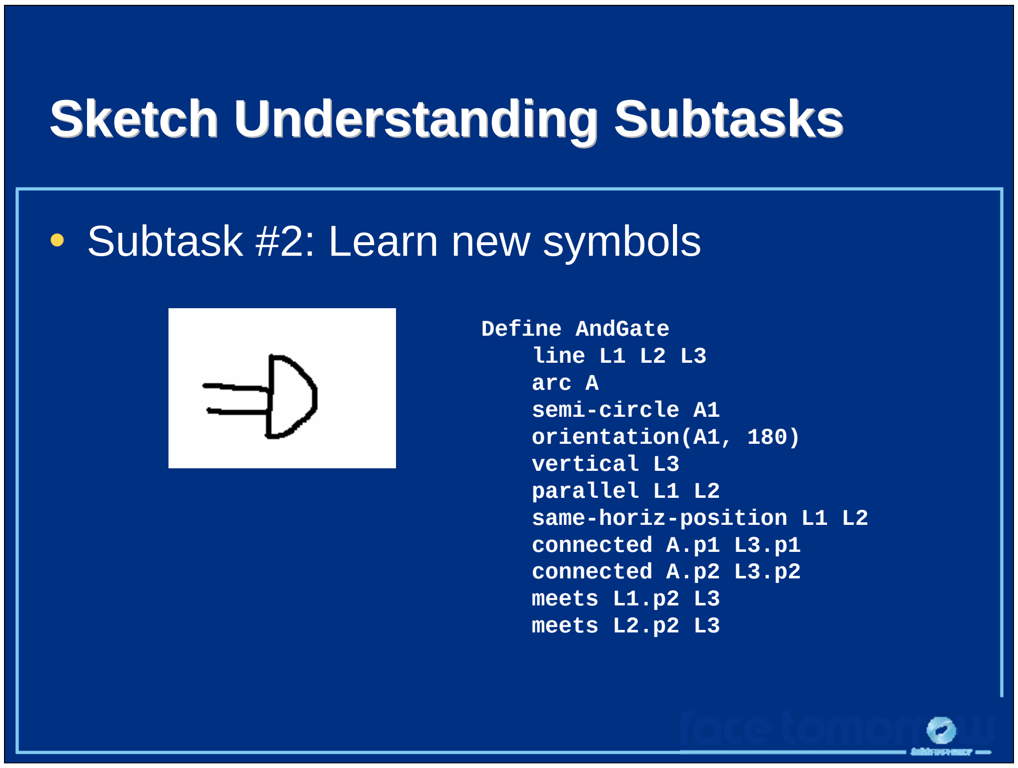
- Subtask #1: Recognize the drawing

There are many different aspects to complete sketch understanding. The first (and most obvious) is being able to recognize the symbols in the sketch. Even this recognition problem is not trivial, and it too involves a number of subtasks.

First, it is often useful to identify corners in the corners in the individual strokes. This process, called *Stroke Fragmentation*, is important for two reasons. First, users sometimes draw more than one symbol using a single stroke. In order to detect the boundary between the two symbols, we need to break the stroke apart into two (or more) pieces. Second, fragmenting strokes into primitive components (i.e., lines and arcs) allows the system to construct a single, consistent representation for each symbol in terms of these primitive components.

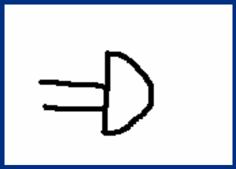
Second, a diagrams recognition system must identify groups of symbols that comprise single objects. This process is called *stroke grouping*.

Finally, the system must be able to recognize a given set of strokes (known to comprise a single object) as a symbol in the domain of interest. This process is called *symbol recognition* or *symbol identification.*



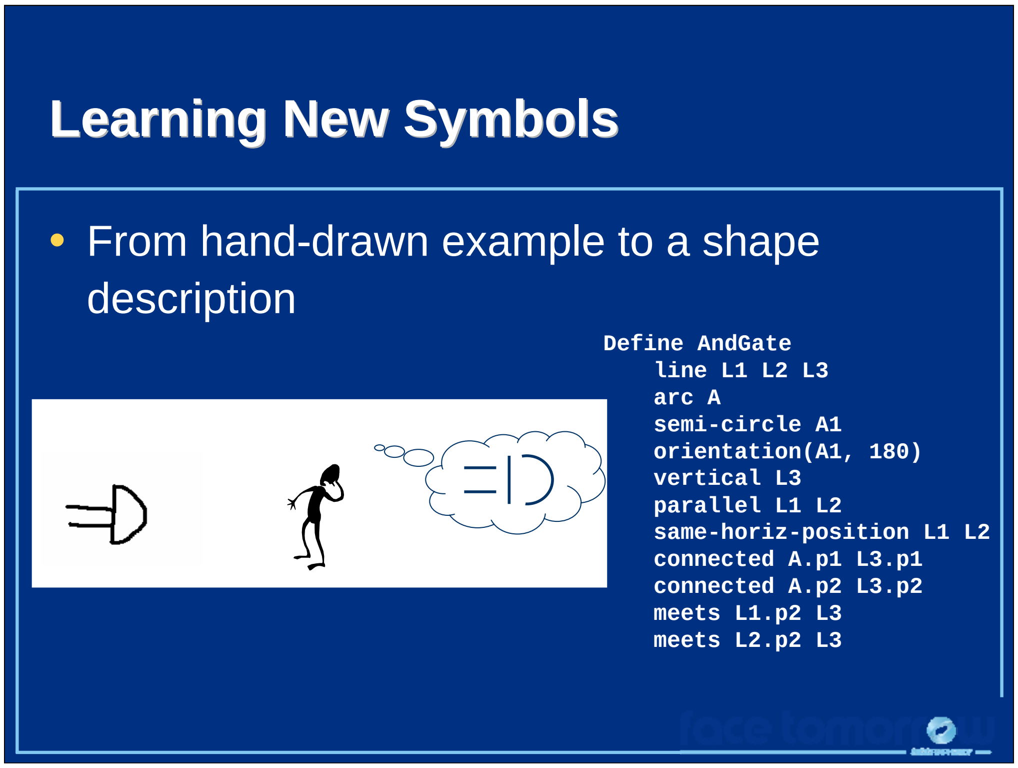
Sketch Understanding Subtasks

- Subtask #2: Learn new symbols



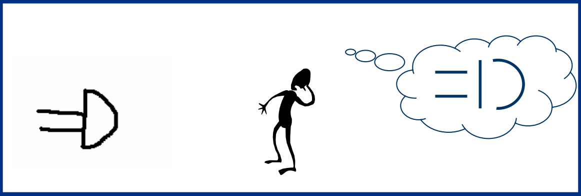
```
Define AndGate
    line L1 L2 L3
    arc A
    semi-circle A1
    orientation(A1, 180)
    vertical L3
    parallel L1 L2
    same-horiz-position L1 L2
    connected A.p1 L3.p1
    connected A.p2 L3.p2
    meets L1.p2 L3
    meets L2.p2 L3
```

The next sketch understanding subtask is to learn the symbols to be recognized. While learning symbol representations is not strictly necessary (the programmer can also hand-code descriptions or recognizers), hand-coding recognizers or descriptions of symbols is tedious, and often leads to poor recognition.



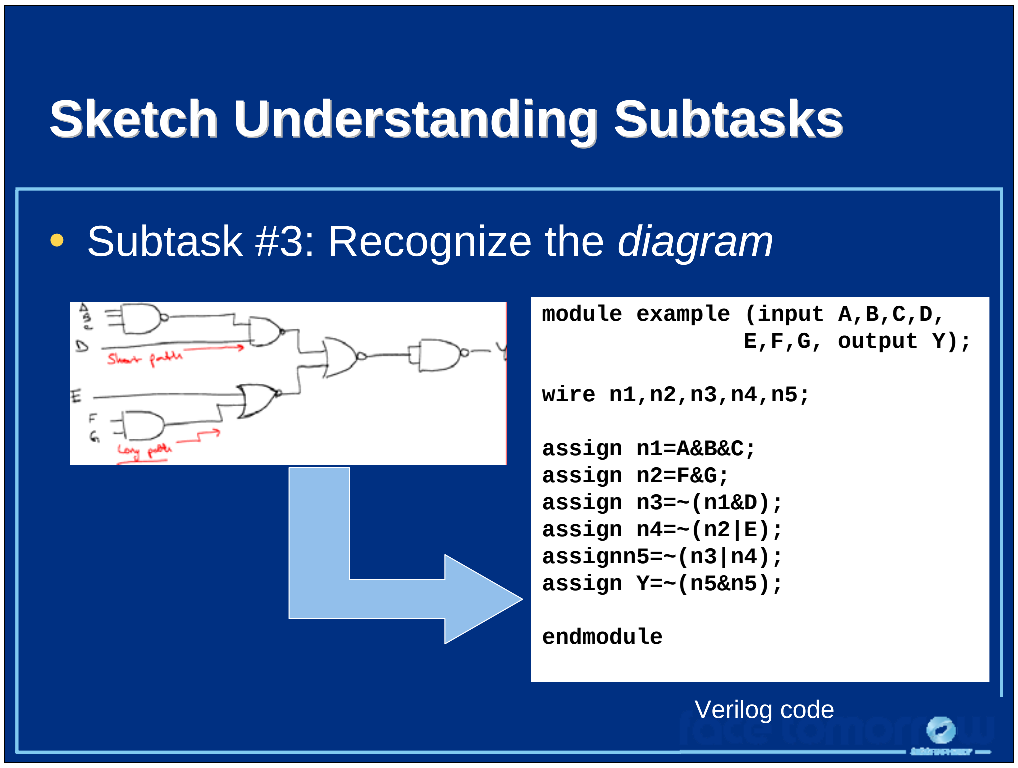
Learning New Symbols

- From hand-drawn example to a shape description



```
Define AndGate
    line L1 L2 L3
    arc A
    semi-circle A1
    orientation(A1, 180)
    vertical L3
    parallel L1 L2
    same-horiz-position L1 L2
    connected A.p1 L3.p1
    connected A.p2 L3.p2
    meets L1.p2 L3
    meets L2.p2 L3
```

There are many possible ways to "learn symbols". One way that ties in to what I will talk about this afternoon is to learn the symbolic description of a shape in terms of its primitive components and the necessary constraints between those components. I will talk more about how this process works later in this talk.



Sketch Understanding Subtasks

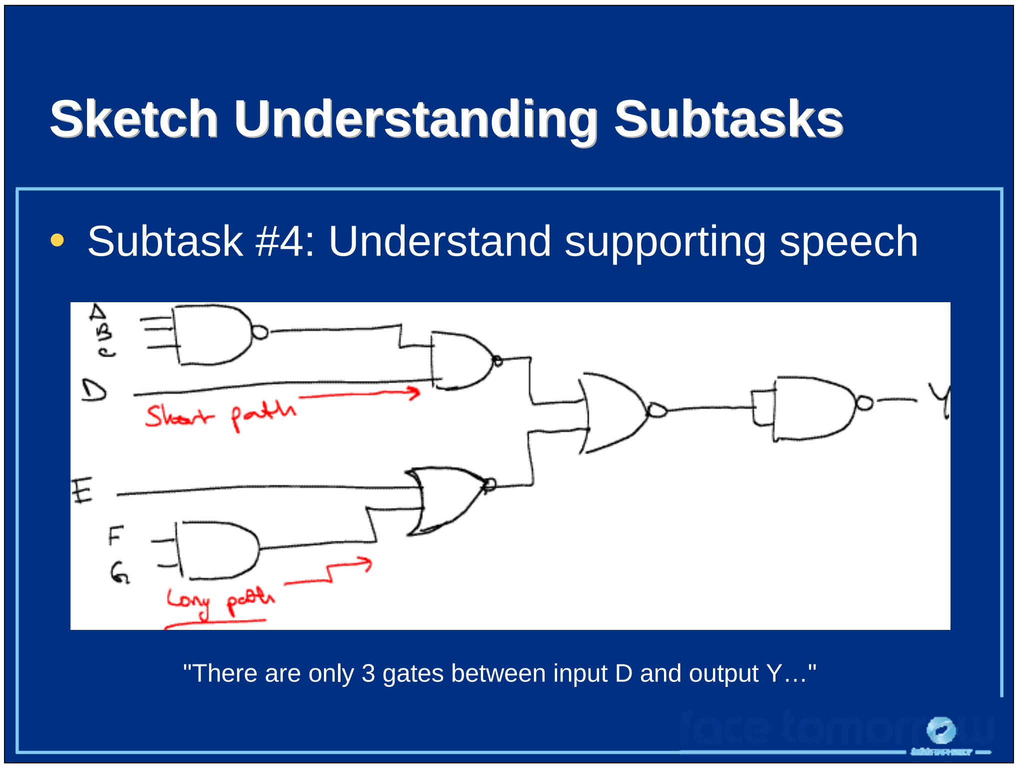
- Subtask #3: Recognize the *diagram*

A
B
C
D
Short path
E
F
G
Long path
Y

```
module example (input A,B,C,D,
               E,F,G, output Y);

wire n1,n2,n3,n4,n5;

assign n1=A&B&C;
assign n2=F&G;
assign n3=~(n1&D);
assign n4=~(n2|E);
assignn5=~(n3|n4);
assign Y=~(n5&n5);

endmodule
```

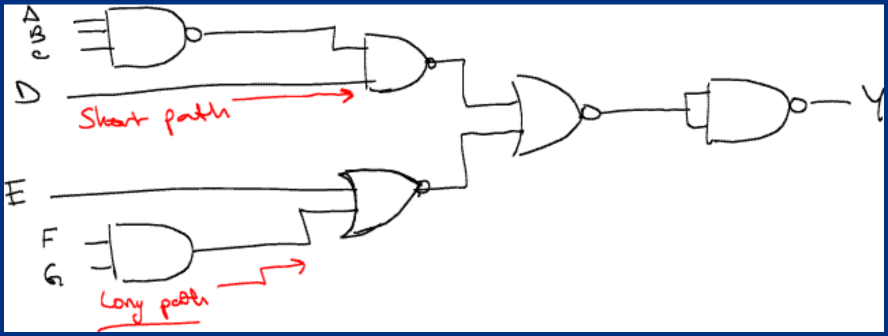
Verilog code

Third, it is not enough simply to recognize the symbols in the diagram. A sketch understanding system must also be able to compose those symbols into a meaningful *diagram*. For example, understanding a circuit diagram means understanding how many inputs and outputs the circuit has, and which wires are connected to which gates (and whether they are inputs to or outputs from the gate). Once the sketch understanding system recovers this information, it can translate the sketched diagram into a more precise form that can be simulated (i.e., the Verilog code that represents the circuit diagram on the left in the example above).



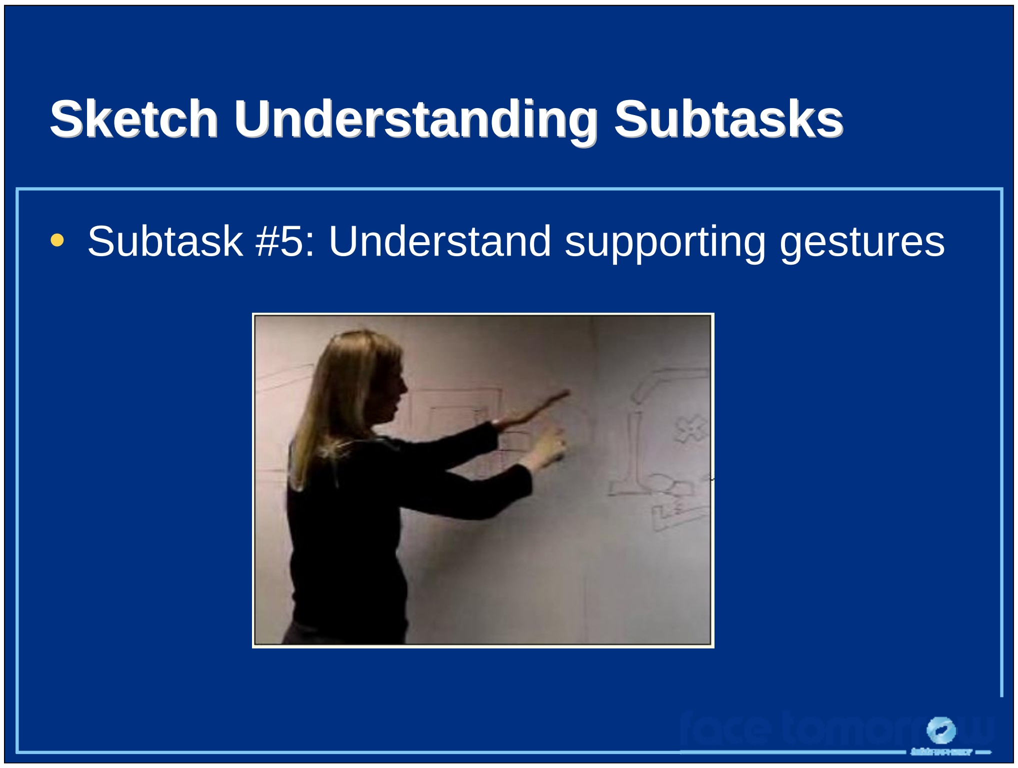
Sketch Understanding Subtasks

- Subtask #4: Understand supporting speech



"There are only 3 gates between input D and output Y…"

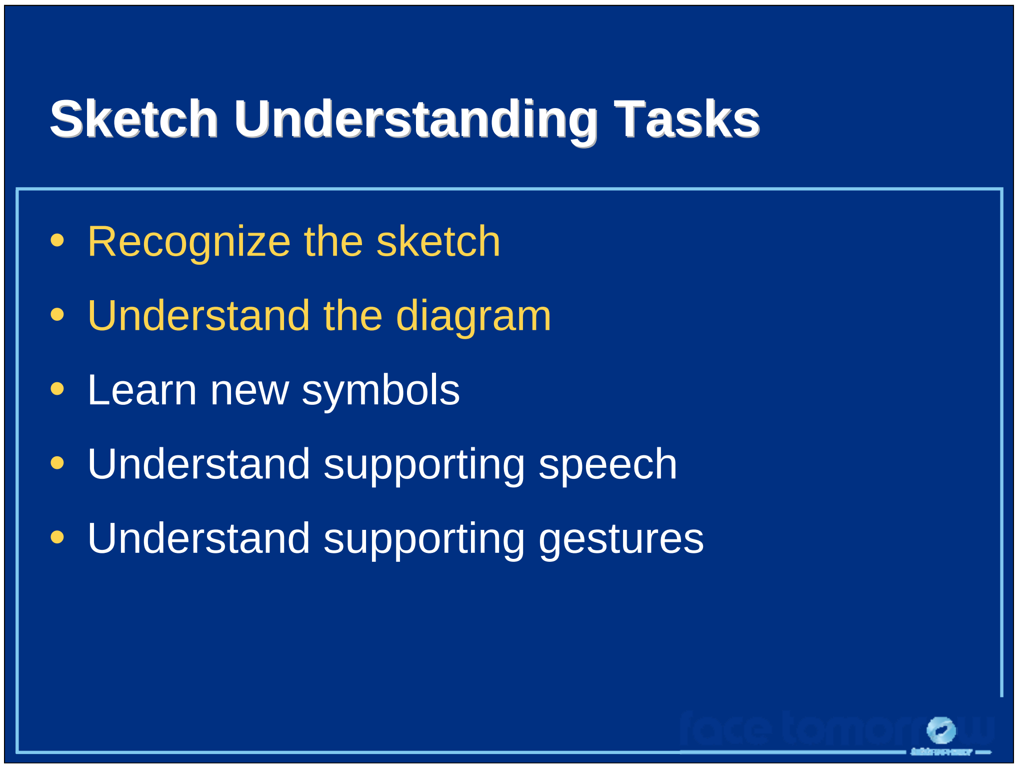
Often, understanding a sketch involves more than just understanding what the user draws. While sketching, designers often convey important information verbally, and this verbal information can help us make sense of the diagram. In the example above, the user talks about the shortest path through the circuit. This speech can potentially help us understand the annotation (in red) below the wire carrying input D.



Sketch Understanding Subtasks

- Subtask #5: Understand supporting gestures

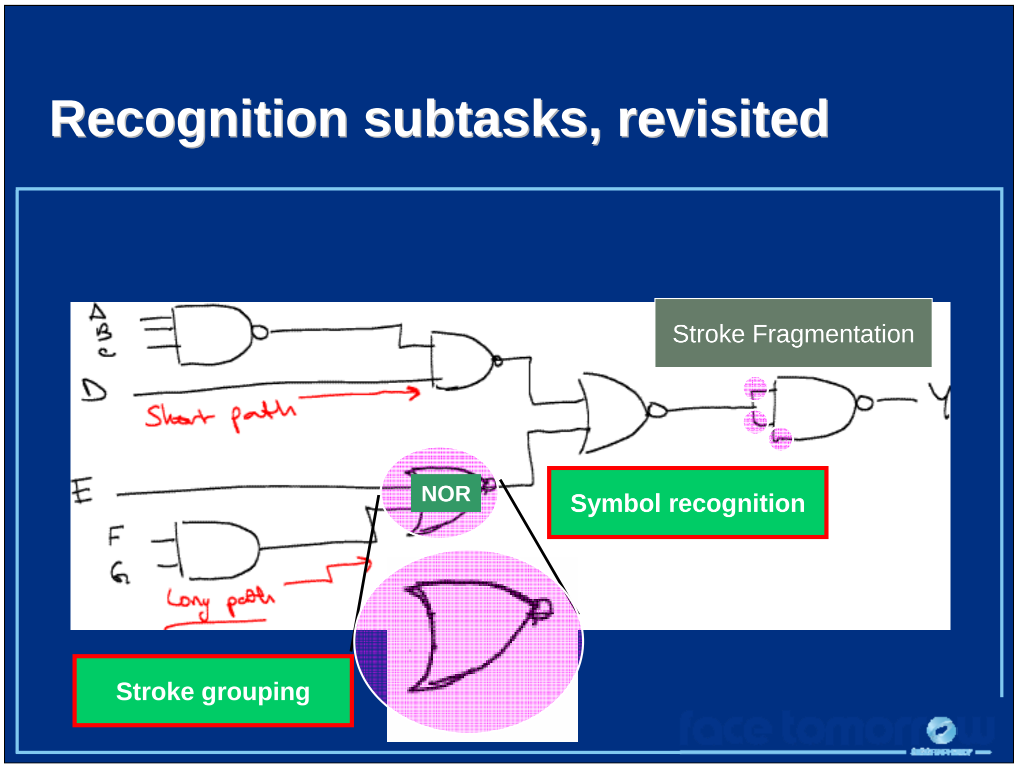
In addition to speaking, people often gesture when they sketch (or when they talk about a sketch). Understanding these gestures can also help us understand the behavior of the system they are sketching.



Sketch Understanding Tasks

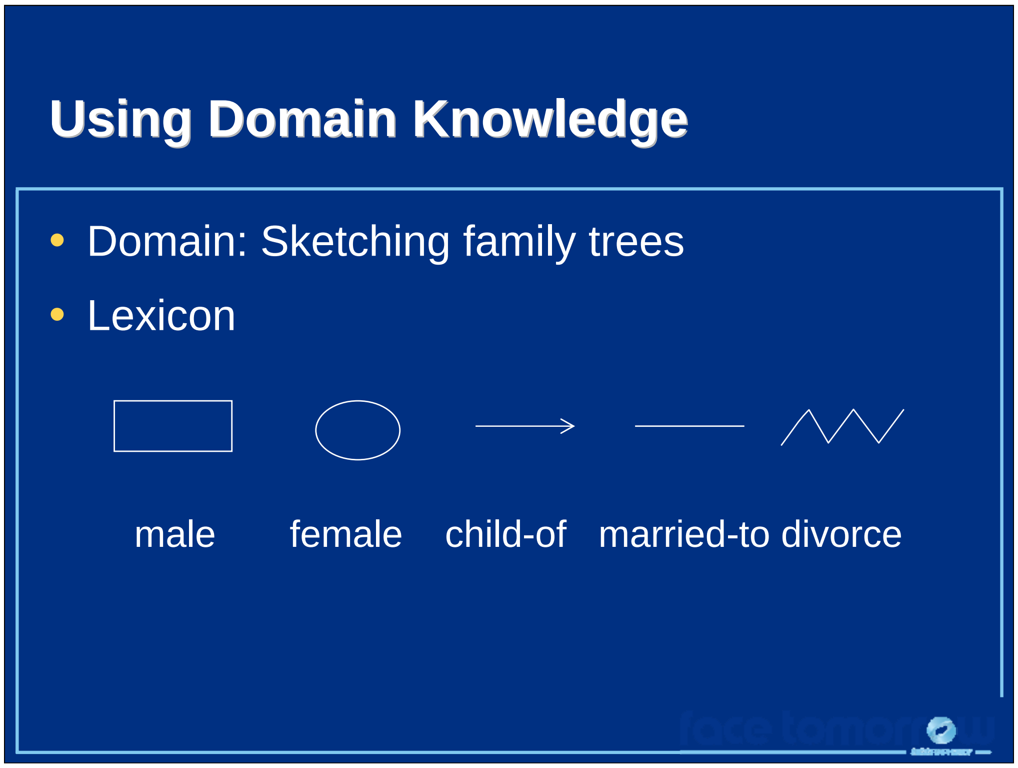
- Recognize the sketch
- Understand the diagram
- Learn new symbols
- Understand supporting speech
- Understand supporting gestures

In this part of the talk, I will introduce a number of implemented sketch understanding systems that focus on one (or more) of the various sketch understanding subtasks. All of these systems can be combined to form a complete sketch understanding system. I will start by talking about a system that understands chemical diagrams. In the process, I will discuss the difficulties inherent in the task of recognizing a sketch.



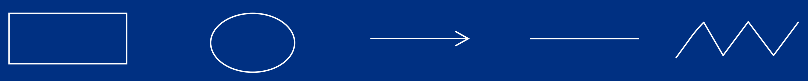
Recognition subtasks, revisited

As we saw previously, diagram recognition involves a number of subtasks. Here I will focus only on symbol recognition and stroke grouping. I will talk about stroke fragmentation in some detail this afternoon.

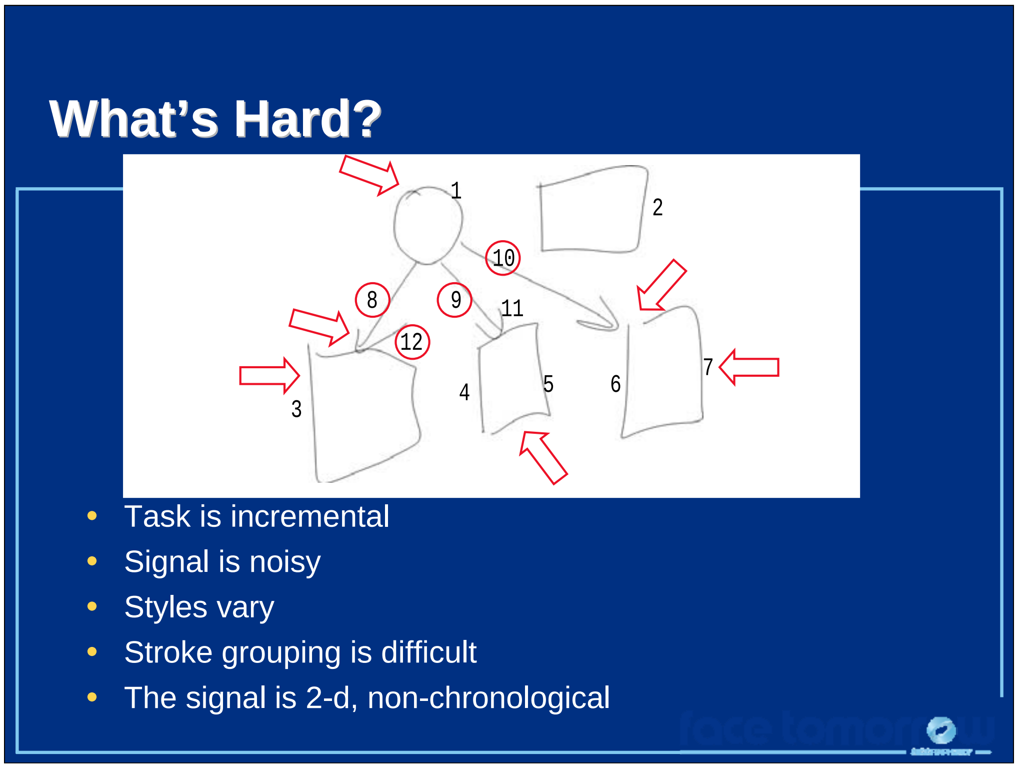


Using Domain Knowledge

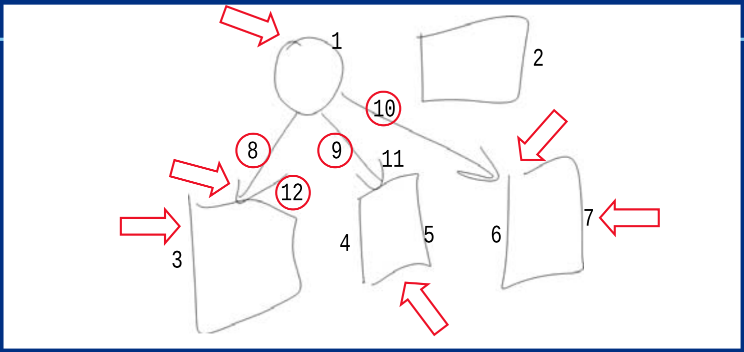
- Domain: Sketching family trees
- Lexicon



To illustrate the many of the challenges of sketch recognition, we will use a simple domain representing family trees. The symbols in this domain are shown here.



What's Hard?



- Task is incremental
- Signal is noisy
- Styles vary
- Stroke grouping is difficult
- The signal is 2-d, non-chronological

Imagine that the user draws the diagram above with the stroke order given by the numbers on the diagram. She first draws the mother, then the father, and then draws the three sons. She then links the mother to each of the sons by drawing the shaft of three arrows and then drawing the arrowheads.

This diagram illustrates many of the challenges a recognition system faces. First, the task is incremental. The system cannot be sure when the user has completed one symbol and moved on to the next. We would like to be able to recognize the diagram as the user sketches to enable seamless interaction with the diagram. For example, if the user wants to move the middle son, she should be able to click and drag it and have the whole box move, without explicitly telling the system that there are two strokes in that symbol.

Second, the signal is noisy. For example, the endpoints in the top circle do not meet exactly and there is a gap between strokes 6 and 7.

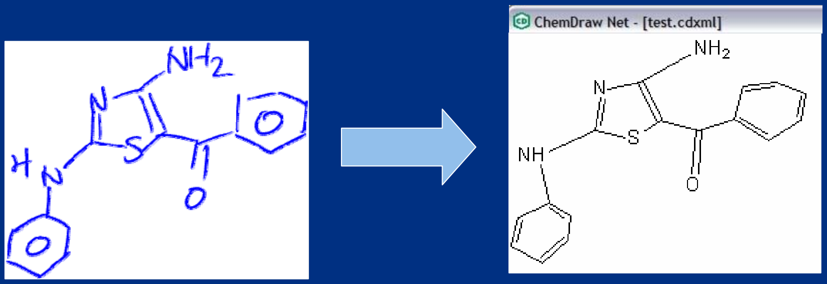
Third, drawing styles vary, even for a single user. Notice that the user drew the left box with only one stroke (Stroke 3) and the right box with two strokes. Style inconsistencies make it more difficult to build recognizers.

Fourth, stroke grouping is difficult. Stroke 12 touches both stroke 3 and stroke 8, so it is difficult to know which stroke it should be grouped with unless you recognize the arrow already.

Fifth, people may (and do) draw the strokes in each symbol in many orders, and they may even draw part of one symbol, move to draw another symbol, and then come back to finish the first symbol. We cannot rely strokes in a single symbol to be temporally contiguous.

The Importance of Domain Information

- Problem: Recognize hand-drawn chemical structures

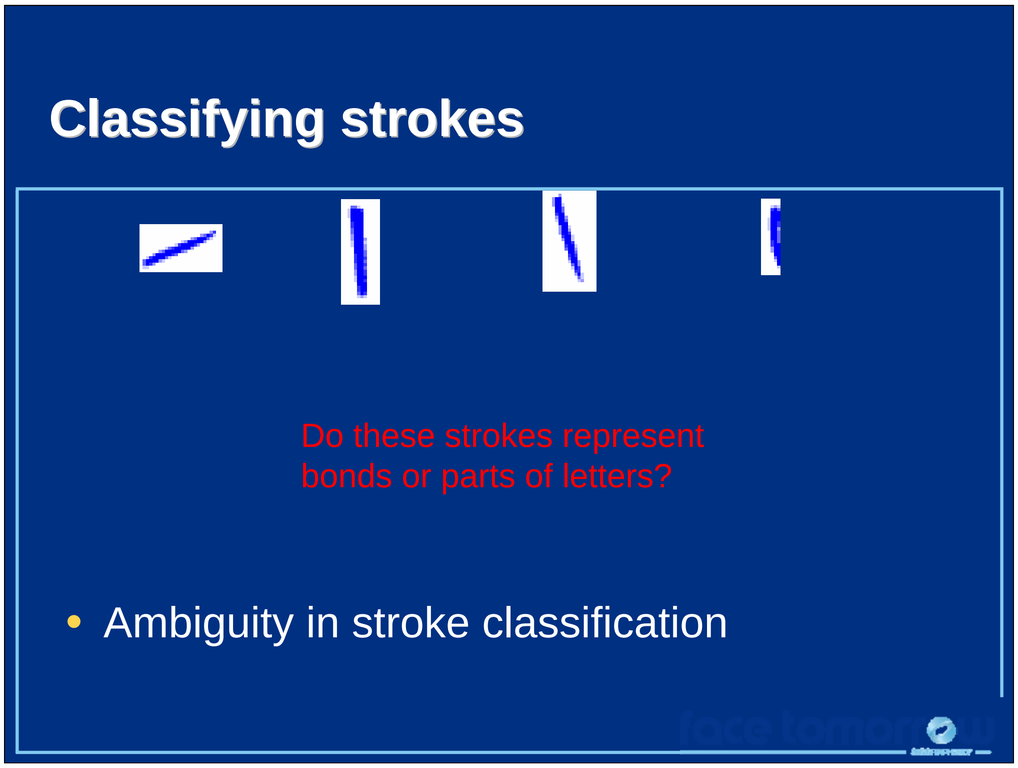


Tom Ouyang and Randall Davis [Ouyang06],
In collaboration with Pfizer Research Technology Center, Cambridge, MA

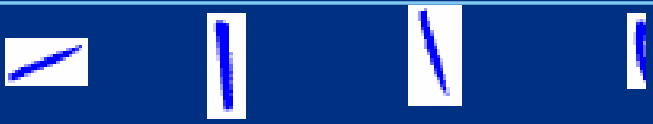
The way we overcome many of these challenges is to rely on domain-specific information for recognition. This application, developed by Davis and Ouyang, uses chemistry domain information to cope with inherent ambiguities in the diagram.

References:

Tom Ouyang and Randall Davis. Recognition of Hand Drawn Chemical Diagrams. In Second Annual CSAIL Student Workshop. 2006.

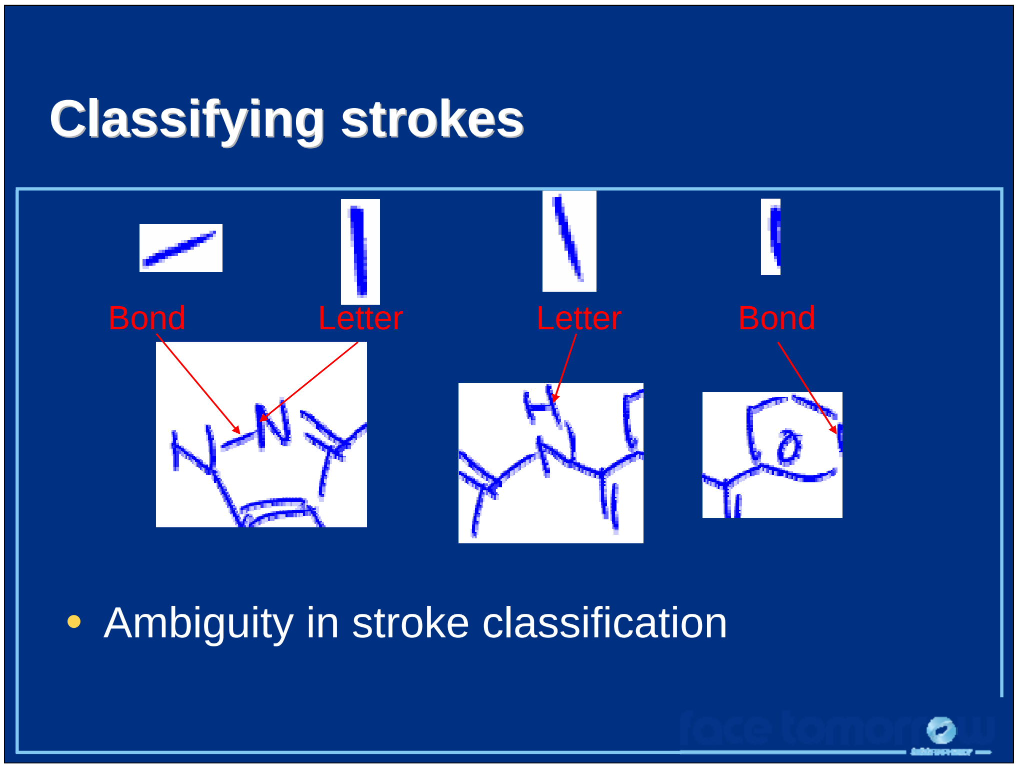


Classifying strokes

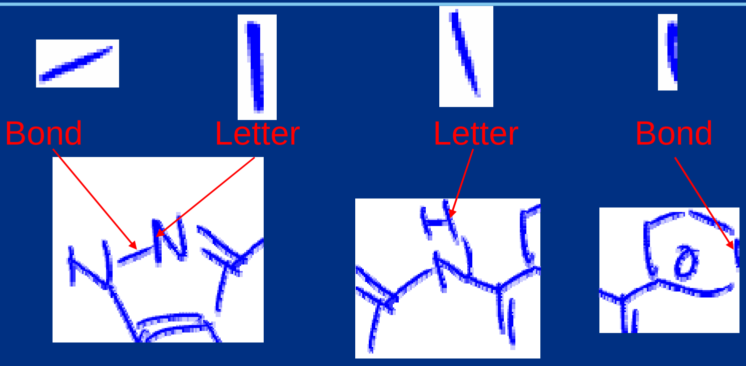


Do these strokes represent bonds or parts of letters?

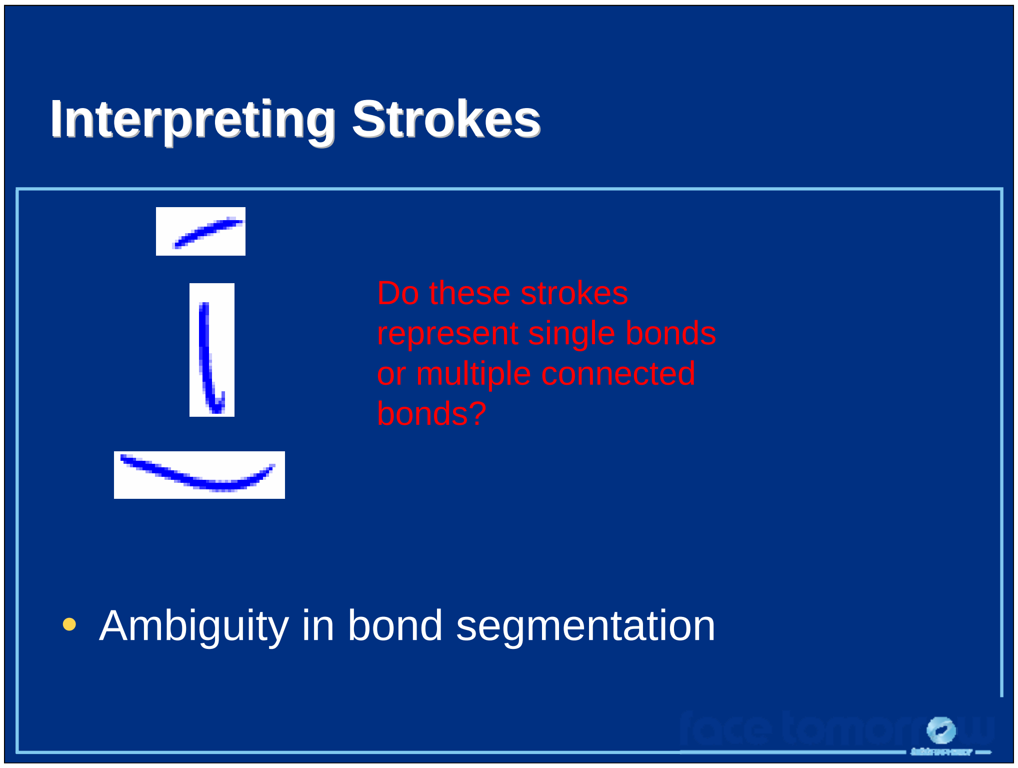
- Ambiguity in stroke classification



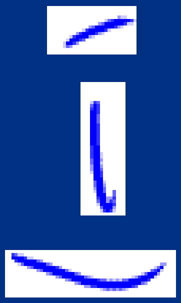
Classifying strokes



- Ambiguity in stroke classification

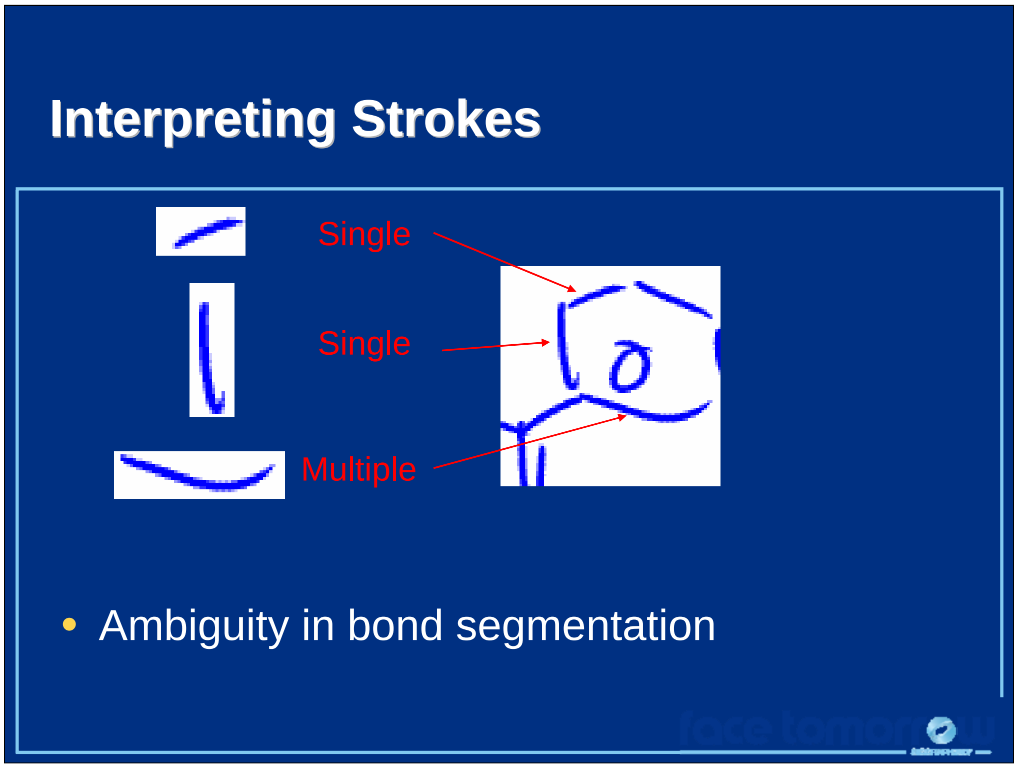


Interpreting Strokes

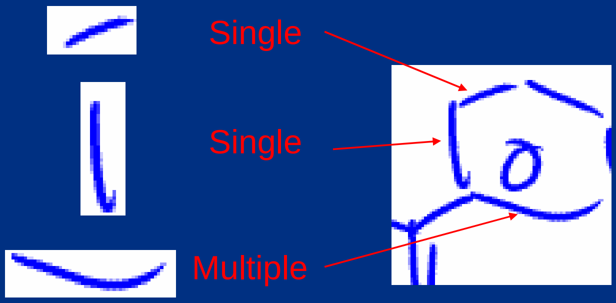


Do these strokes represent single bonds or multiple connected bonds?

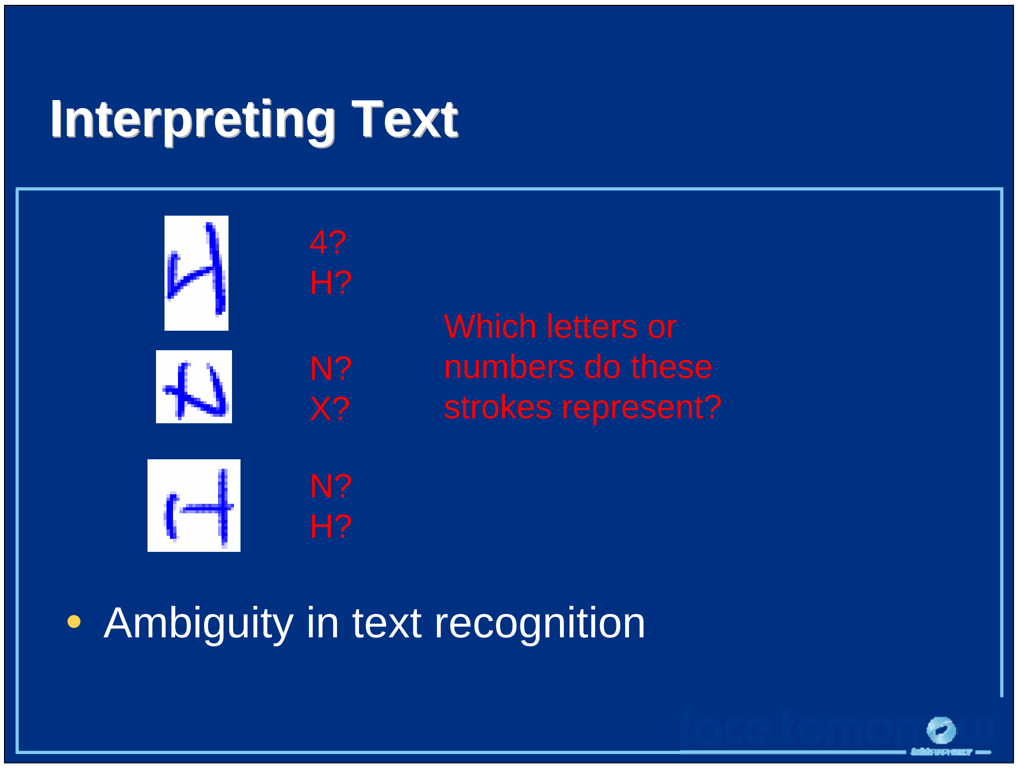
- Ambiguity in bond segmentation



Interpreting Strokes



- Ambiguity in bond segmentation



Interpreting Text

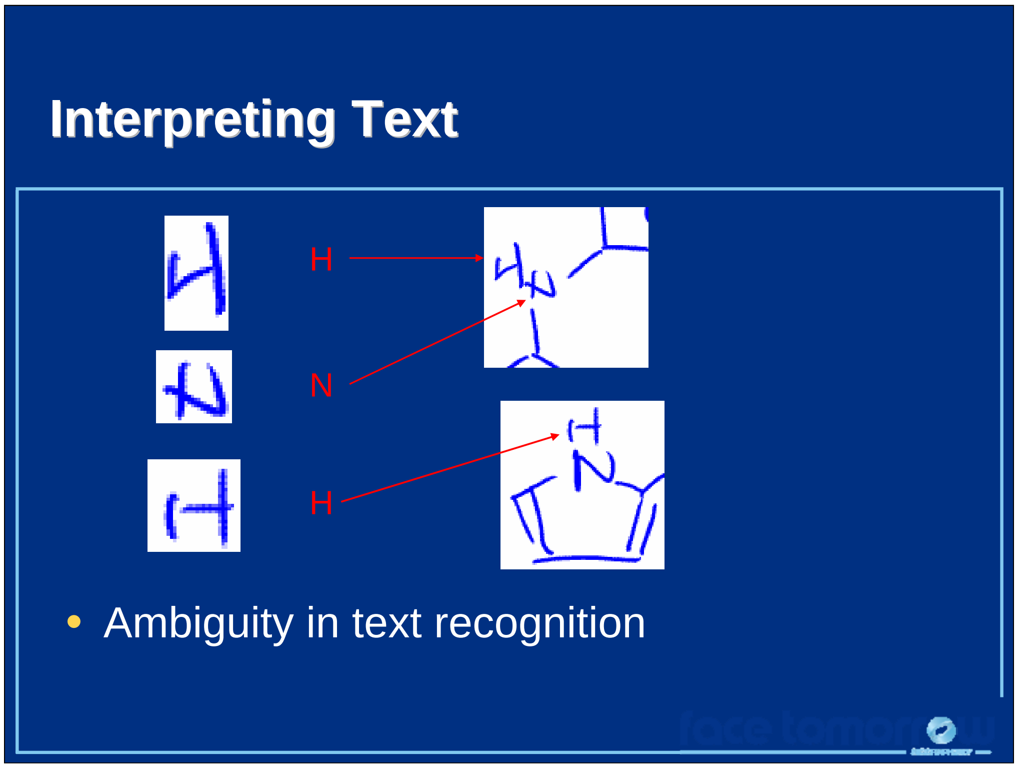
4?
H?

N?
X?

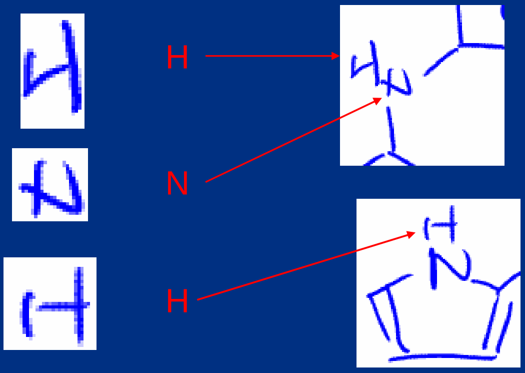
N?
H?

Which letters or numbers do these strokes represent?

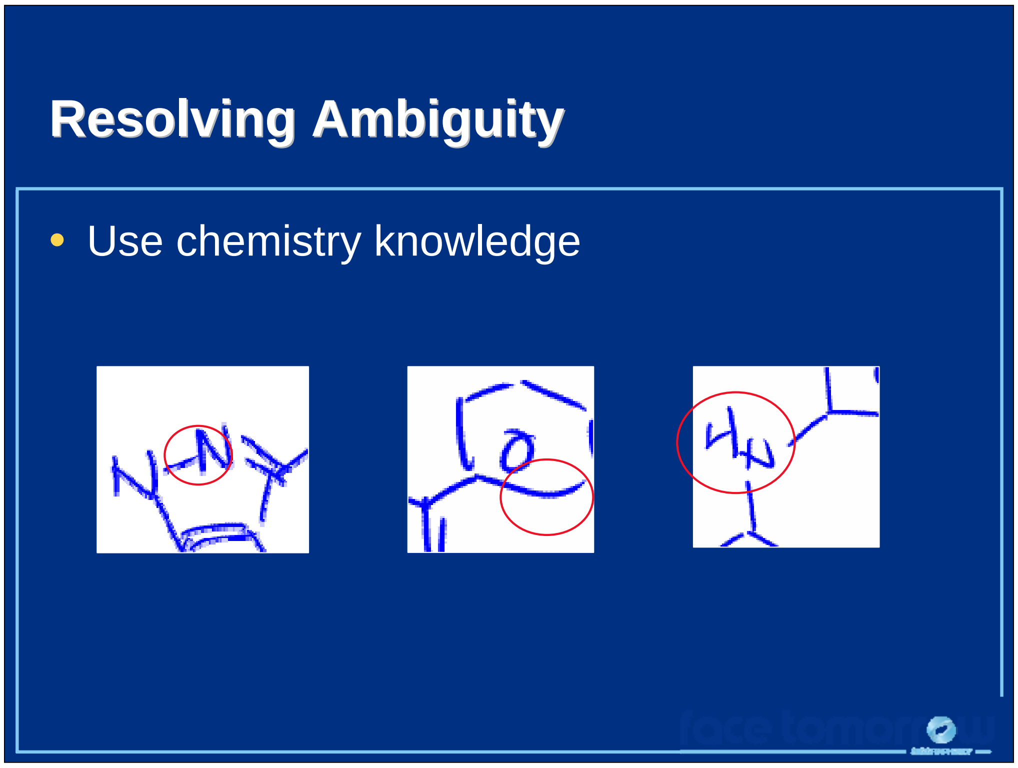
- Ambiguity in text recognition



Interpreting Text



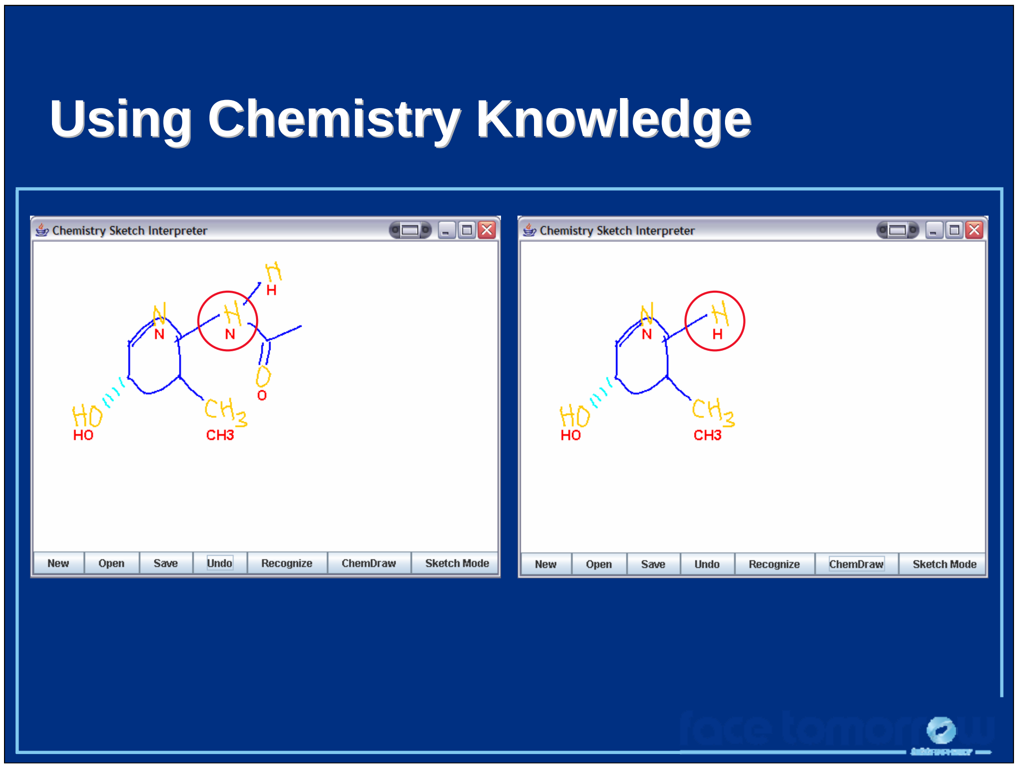
- Ambiguity in text recognition



Resolving Ambiguity

- Use chemistry knowledge

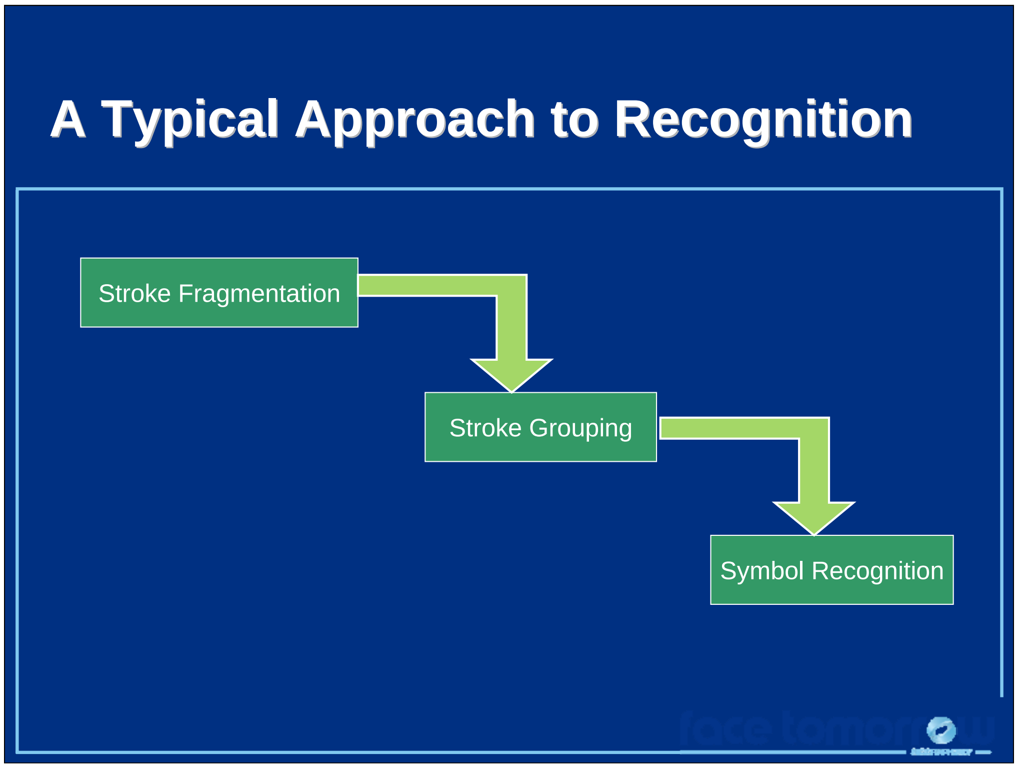
The context surrounding each individual stroke combined what we know about chemical structure diagrams makes clear the correct interpretation of each of these ambiguous strokes.



Here we can see how chemistry knowledge allows the system to interpret the same set of strokes as two different symbols (Nitrogen or Hydrogen) based on the constraints of the domain. In the structure on the left, although the symbol looks like an H, hydrogen, which contains room for only one electron, would not be able to bond with the three surrounding atoms, while nitrogen can. On the left the system interprets the stroke as hydrogen because the bond structure is consistent with the structure of the hydrogen atom.

Domain knowledge is used in a similar way in the following system

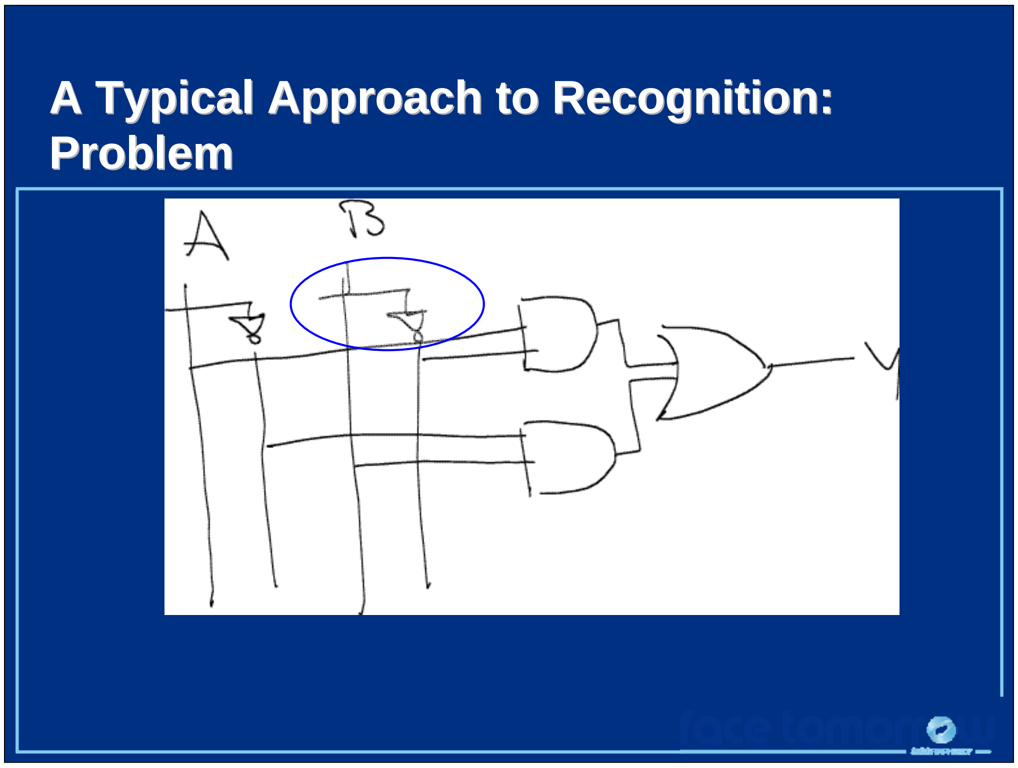
Leslie Gennari, Levent Burak Kara, Thomas F. Stahovich (2005) Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams. Computers & Graphics 29(4): 547-562 2005.



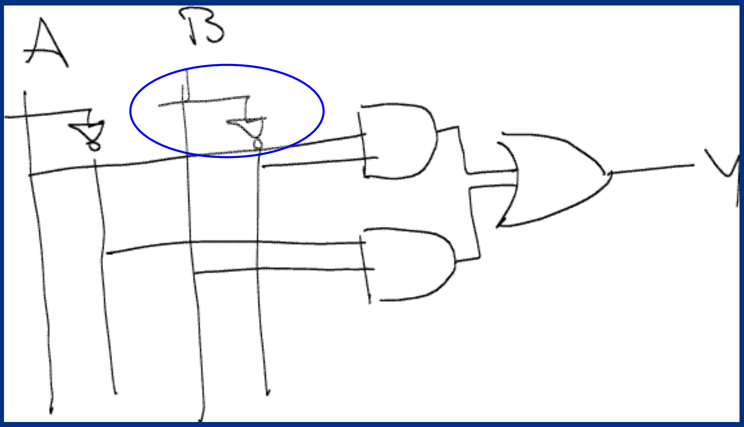
A Typical Approach to Recognition

$$\begin{tikzcd}
\text{Stroke Fragmentation} \arrow[r] & \text{Stroke Grouping} \arrow[r] & \text{Symbol Recognition}
\end{tikzcd}$$

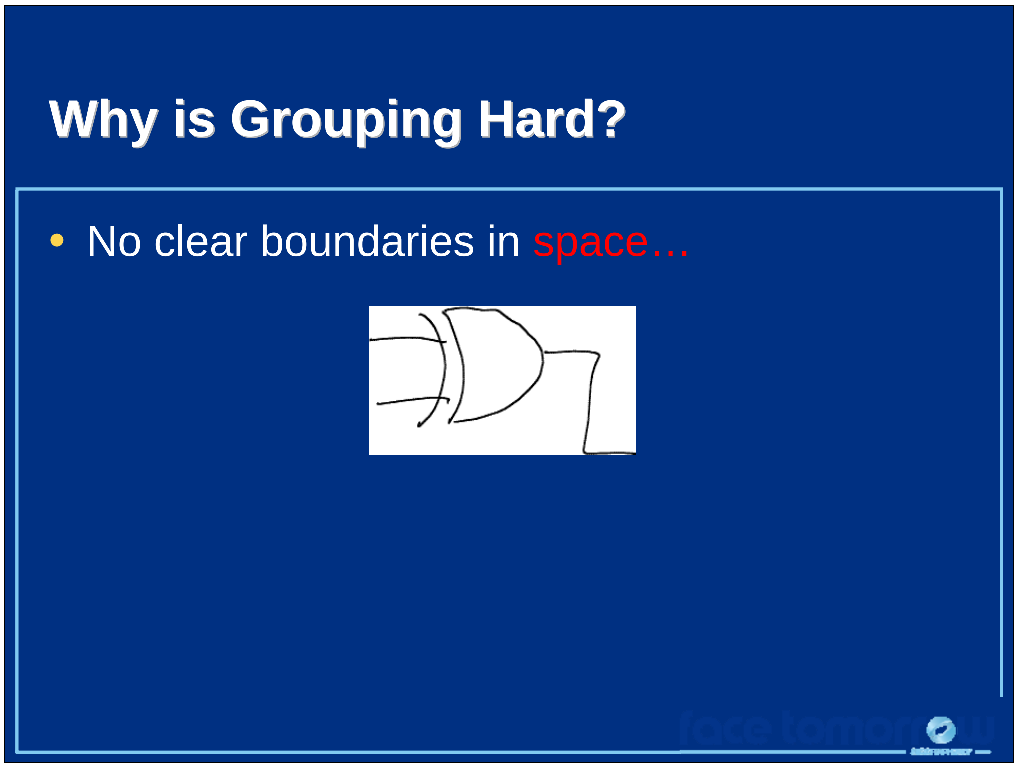
So far we have focused on symbol recognition, assuming the stroke grouping was done prior to this step. This example shows the typical model for many recognition systems: strokes are grouped, and then those groups are handed off to shape recognizers.



A Typical Approach to Recognition: Problem

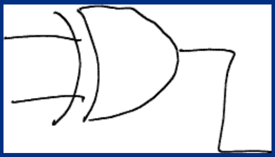


However, there is a problem with this waterfall model—errors cascade through the system. If the system groups strokes incorrectly, then it will also fail to recognize the symbol or symbols those strokes represent.

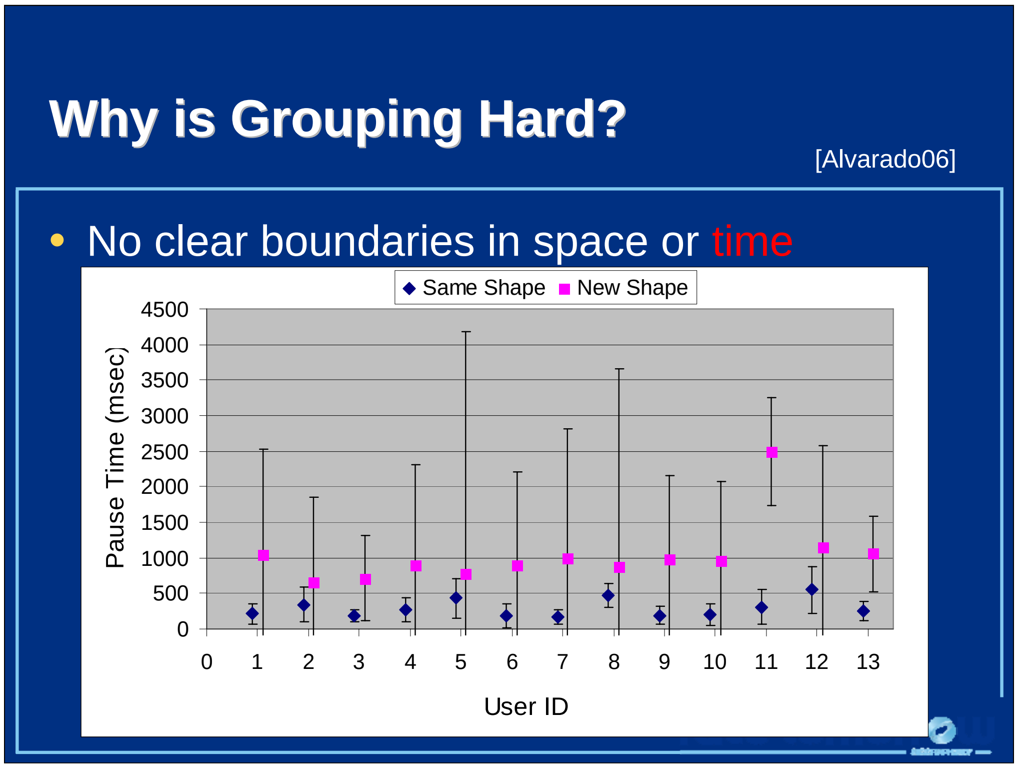


Why is Grouping Hard?

- No clear boundaries in space…



Stroke grouping is difficult, and errors are common. There are two main reasons why grouping is difficult. First, there are no clear spatial boundaries between symbols. In this example, you can see that the two input wires touch the left piece of the XOR gate, but the two pieces of the gate itself do not actually touch one another. A spatial grouper would not tend to group the strokes comprising the gate into a single symbol, but would prefer to group each have to the gate with the wires that connect to it.



Why is Grouping Hard?

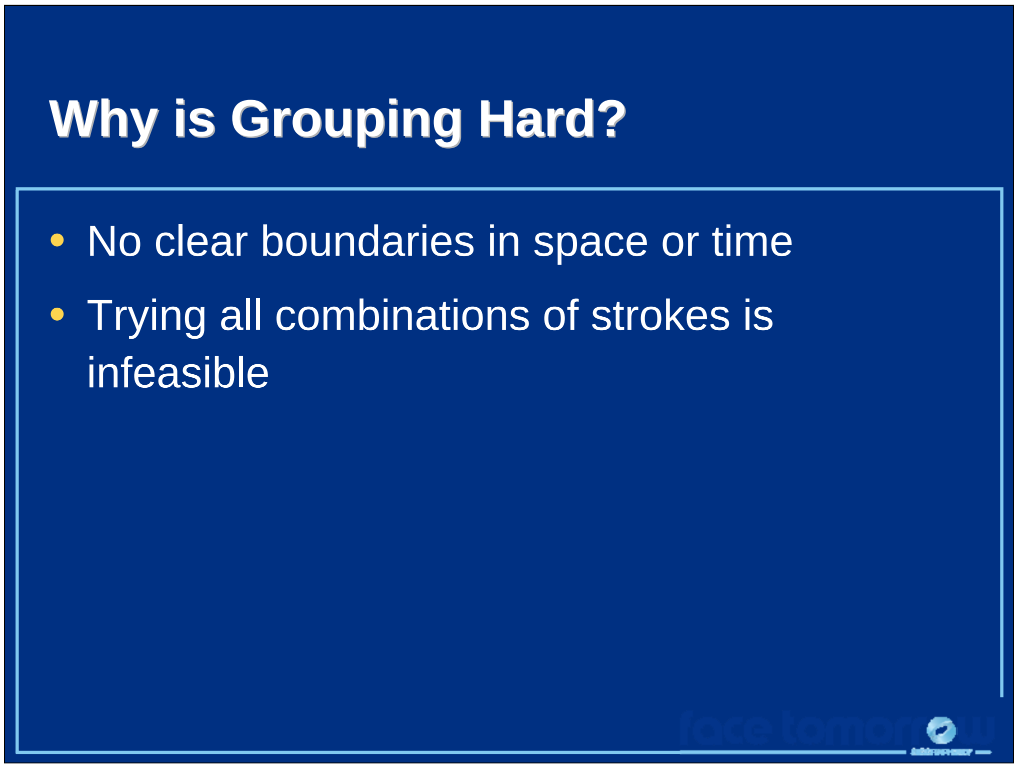
[Alvarado06]

- No clear boundaries in space or time

Stroke grouping is also difficult because there are no clear boundaries between symbols in time either. This graph shows the median pause time between consecutive strokes for strokes that are part of the same shape and strokes that are part of different shapes. The error bars show the interquartile range. It is clear that these distributions overlap and there no temporal boundary that will reliably indicate when the user has finished drawing one shape and moved on to the next.

References:

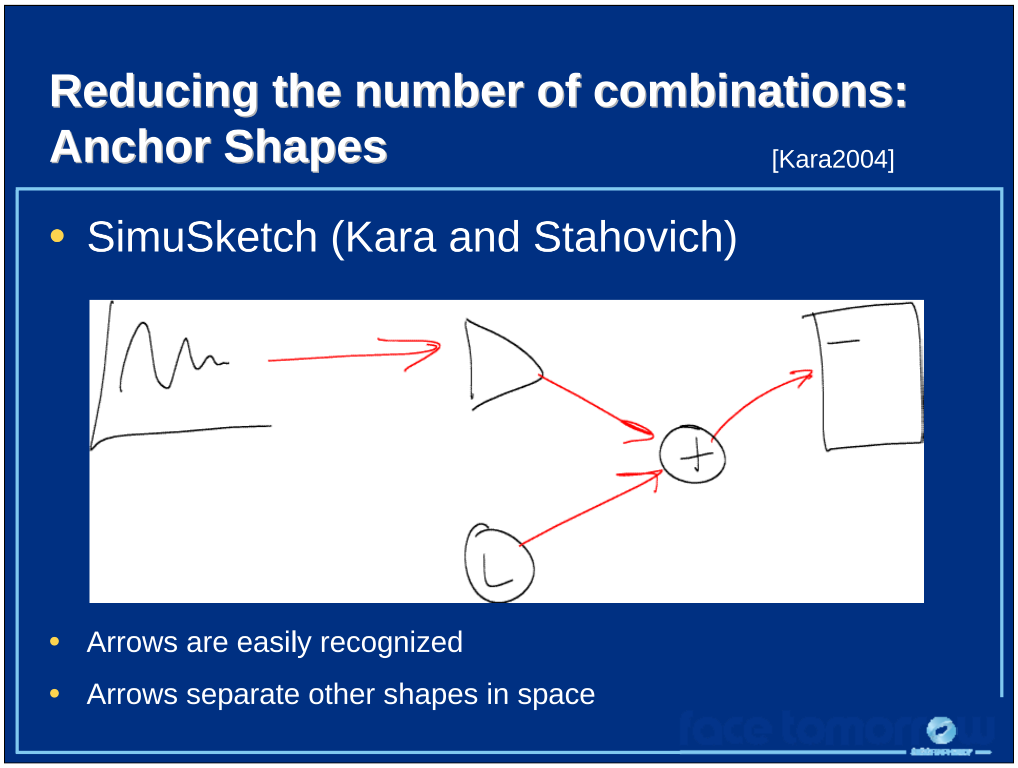
Christine Alvarado and Michael Lazzareschi. Properties of Real World Digital Logic Diagrams**.** *1st International Workshop on Pen-based Learning Technologies (PLT*) 2007.



Why is Grouping Hard?

- No clear boundaries in space or time
- Trying all combinations of strokes is infeasible

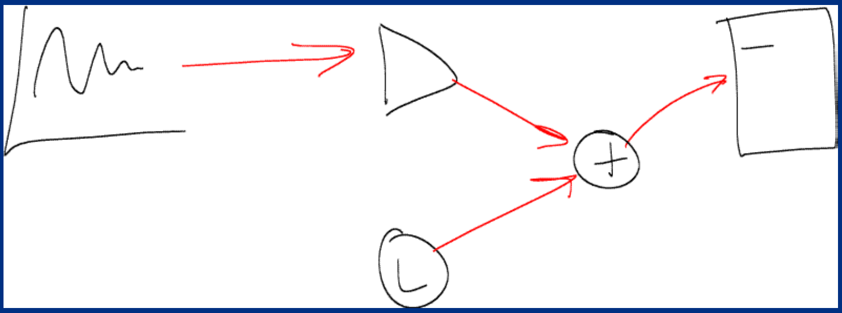
Furthermore, trying all combinations of strokes is computationally infeasible.



Reducing the number of combinations: Anchor Shapes

[Kara2004]

- SimuSketch (Kara and Stahovich)



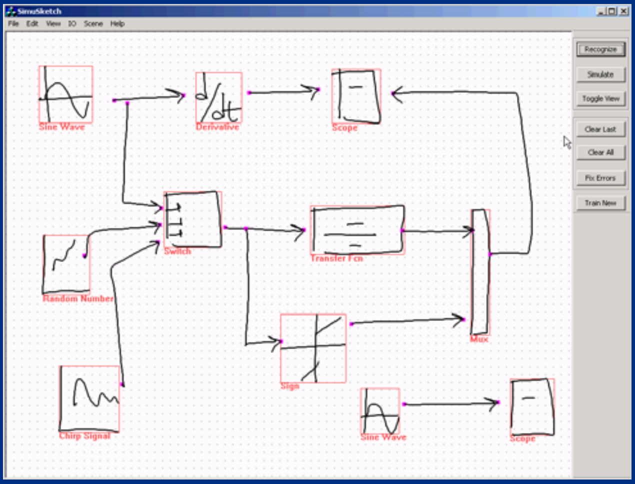
- Arrows are easily recognized
- Arrows separate other shapes in space

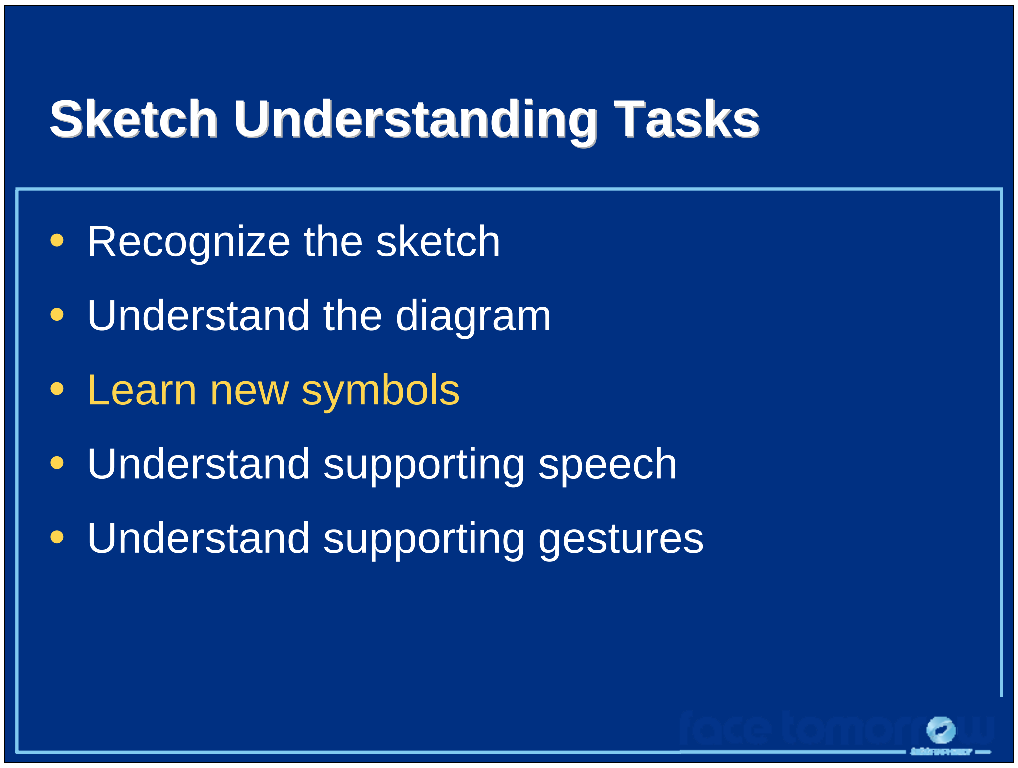
We must take steps to reduce the number of possible groupings the system must consider. One approach taken by Kara and Stahovich is to use shapes that are easily recognized as "anchor points" in the drawing. Their system reliably identifies arrows. Once these arrows are identified, they spatially separate the remaining pieces of the drawing so grouping becomes straightforward using simple distance metrics.

References:

Levent Burak Kara, Thomas F. Stahovich (2004) Hierarchical Parsing and Recognition of Hand-Sketched Diagrams. 17th ACM User Interface Software Technology (UIST) 2004.

SimuSketch Video

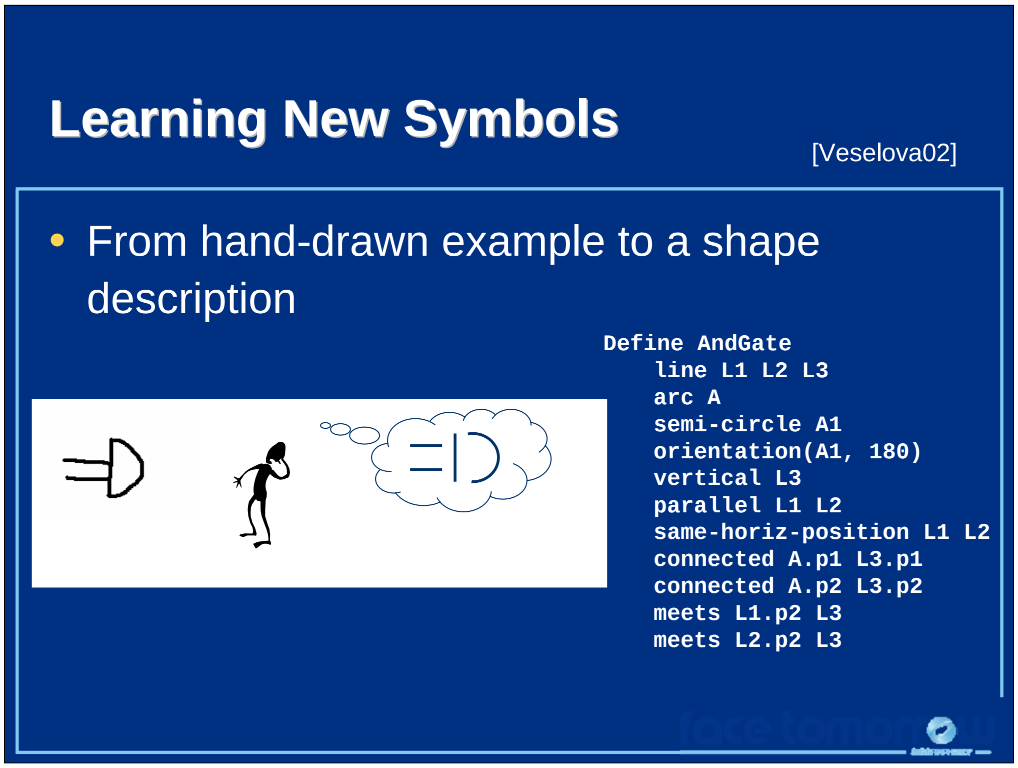




Sketch Understanding Tasks

- Recognize the sketch
- Understand the diagram
- Learn new symbols
- Understand supporting speech
- Understand supporting gestures

The next system I will discuss is a system for learning new symbols to be recognized.



Learning New Symbols

[Veselova02]

- From hand-drawn example to a shape description

```
Define AndGate
    line L1 L2 L3
    arc A
    semi-circle A1
    orientation(A1, 180)
    vertical L3
    parallel L1 L2
    same-horiz-position L1 L2
    connected A.p1 L3.p1
    connected A.p2 L3.p2
    meets L1.p2 L3
    meets L2.p2 L3
```

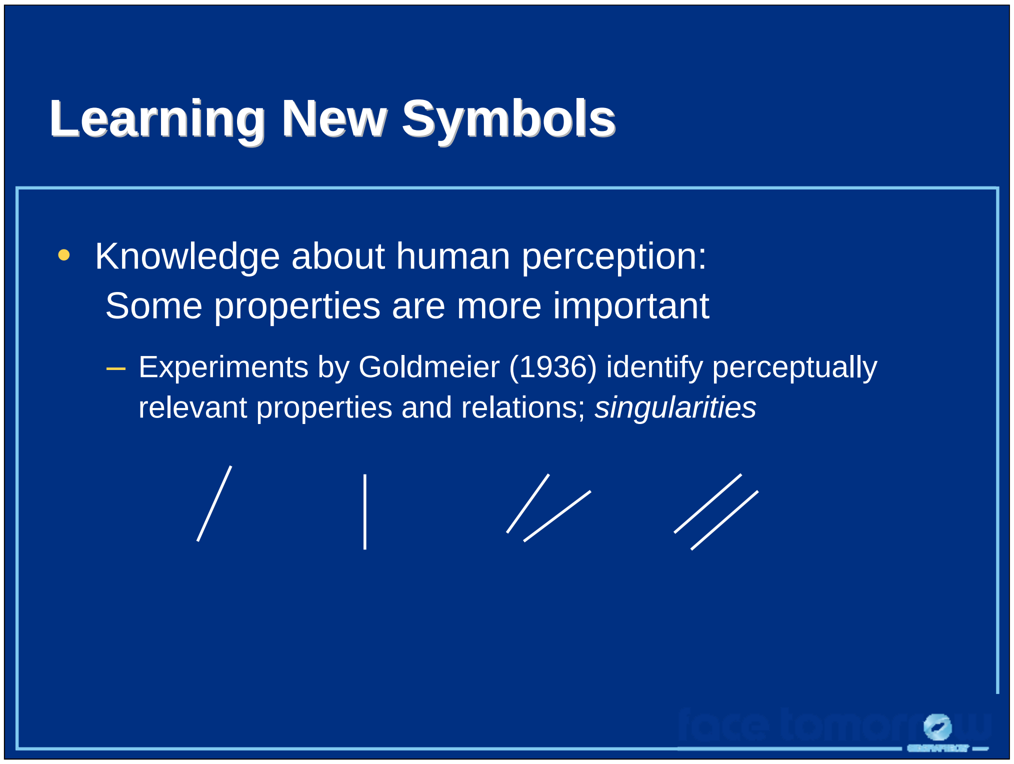
One way we can represent shapes to be recognized is through a structural description of the shape in terms of its simple subshapes and constraints between those subshapes. I will talk more about how we build a recognition system using this representation this afternoon.

Notice that these descriptions, while relatively straightforward, are somewhat tedious to write out. Futhermore, they are prone to errors—the human writing the description can easily forget to specify an important relationship between subcomponents.

Instead, we would like to build a system that can decompose a single hand-drawn example into its primitive components and automatically detect the constraints between them. Unfortunately, a naïve system will detect constraints very literally (e.g., the angle between line 1 and line 2 is 182.3432 degrees). Furthermore, it will likely detect constraints that don't really matter (i.e., the length of the arc is 1.54 time the length of the line at the base of the gate). How can we build a system that will learn *perceptually salient* constraints?

References:

Olya Veselova and Randall Davis. Perceptually Based Learning of Shape Descriptions. In Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), pp.482-487. San Jose, California, 2004.



Learning New Symbols

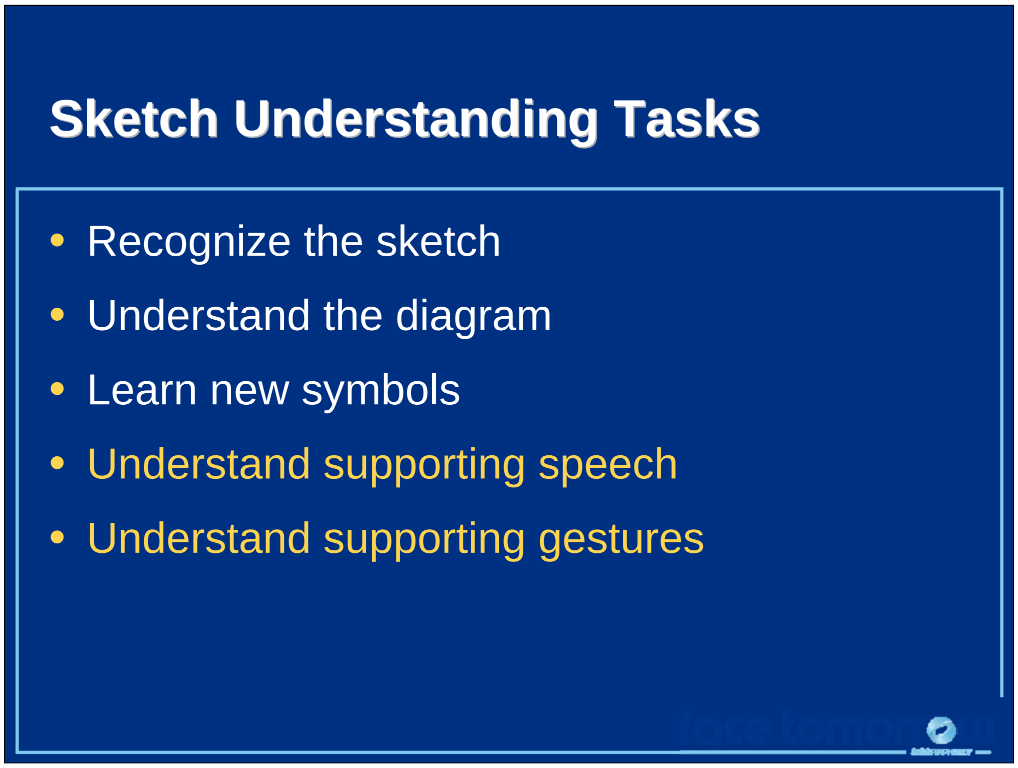
- Knowledge about human perception: Some properties are more important
  - Experiments by Goldmeier (1936) identify perceptually relevant properties and relations; *singularities*

Our goal is to build a system that will learn constraints the way we perceive them, so to figure out how we humans perceive constraints we must turn to the psychology literature. Through experiments, Goldmeier deteremined a number of perceptually salient constraints that people tended to easily identify. For example, people notice when a line is exactly vertical, or when two lines are exactly parallel. They do not believe these relationships happened by accident; they believe they were intentional.

Vesselova and Davis built a system capable learning shape descriptions from a single example by leveraging the notion of intentional constraints.

References:

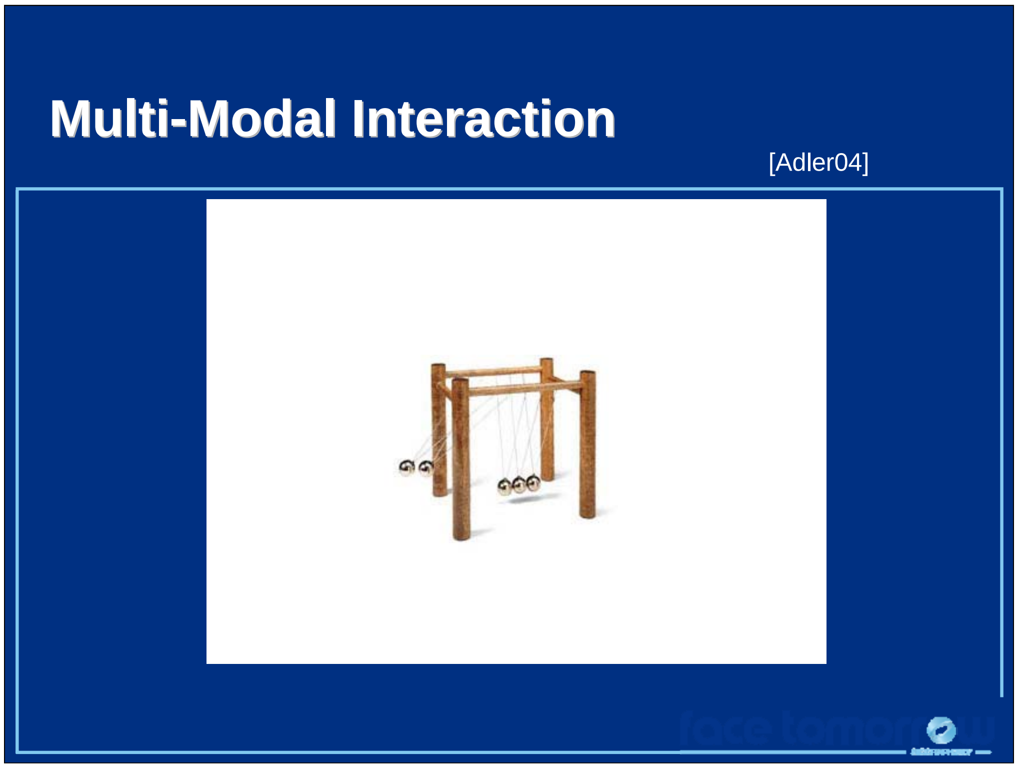
Olya Veselova and Randall Davis. Perceptually Based Learning of Shape Descriptions. In Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), pp.482-487. San Jose, California, 2004.



Sketch Understanding Tasks

- Recognize the sketch
- Understand the diagram
- Learn new symbols
- Understand supporting speech
- Understand supporting gestures

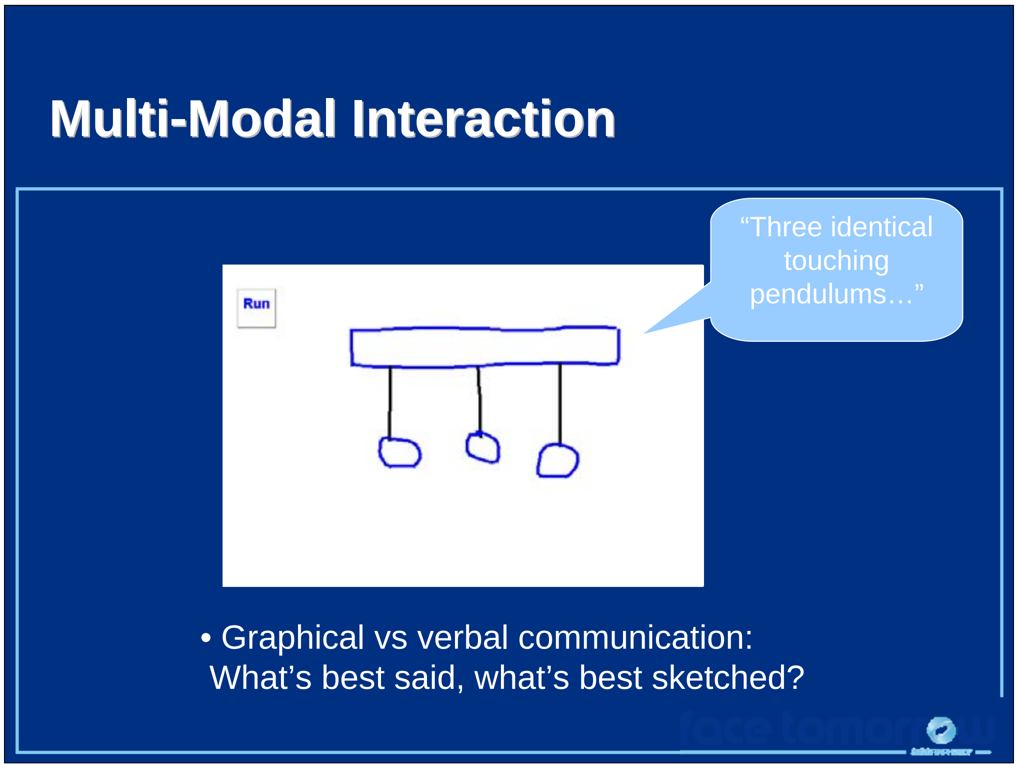
Finally, we examine two systems that support multi-modal interactions.



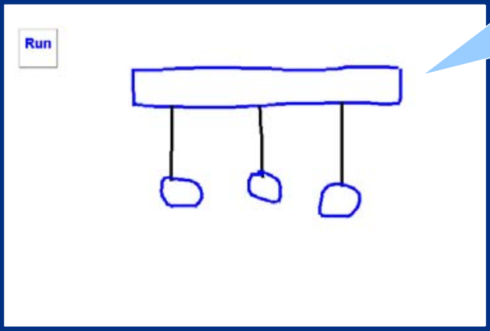
Multi-Modal Interaction

[Adler04]

In many cases, people don't just sketch, but they also speak about the device they are designing. Understanding this speech can help us understand what they are sketching.



Multi-Modal Interaction

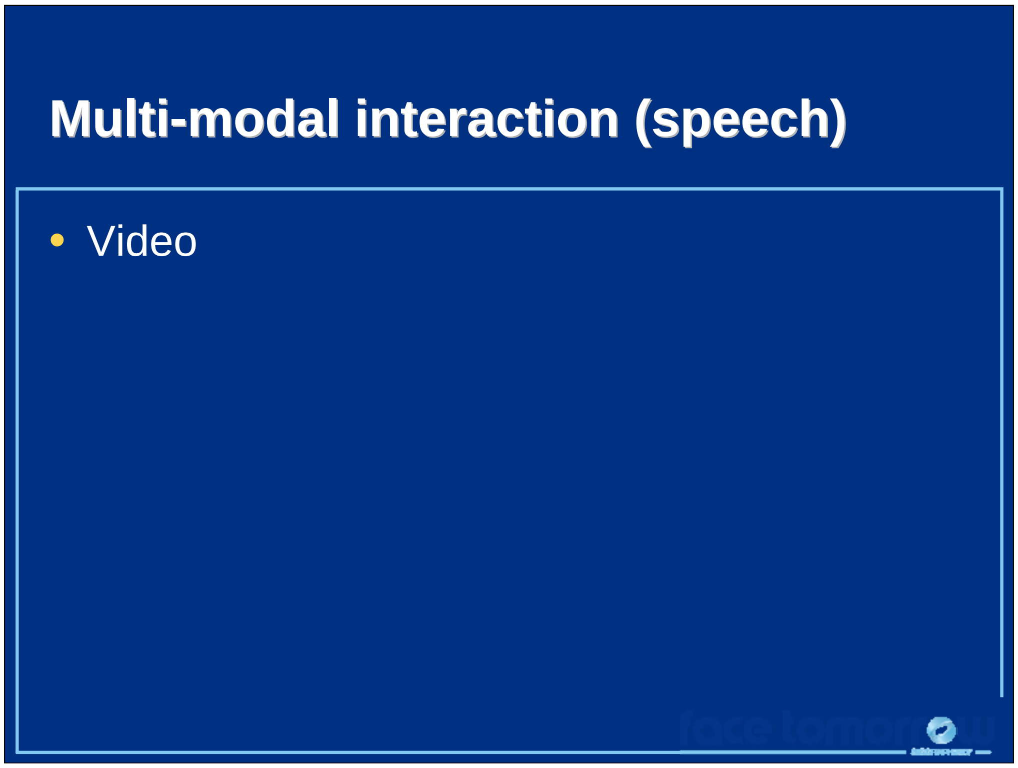


- Graphical vs verbal communication: What's best said, what's best sketched?

For example, in the Newton's cradle toy from the previous slide it is critical that all of the pendulums are identical in size and just barely touching each other at rest. It is incredibly difficult to draw three identical, exactly touching pendulums, but it is quite simple to say this. The tool designed by Adler and Davis, an extension of the ASSIST sketch understanding system, recognizes supporting speech and uses it to modify the diagram to achieve the designed behavior, as you see in the following video.

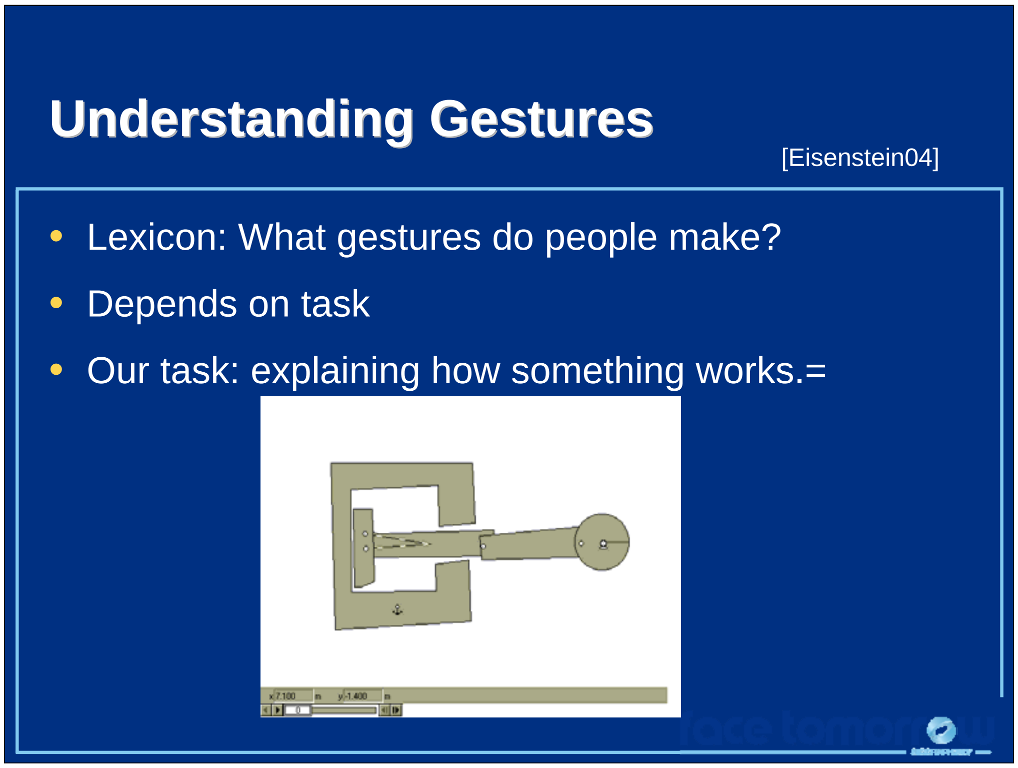
References:

Aaron Adler and Randall Davis. Speech and Sketching for Multimodal Design. In Proceedings of the 9th International Conference on Intelligent User Interfaces, pp.214-216. 2004.



Multi-modal interaction (speech)

- Video

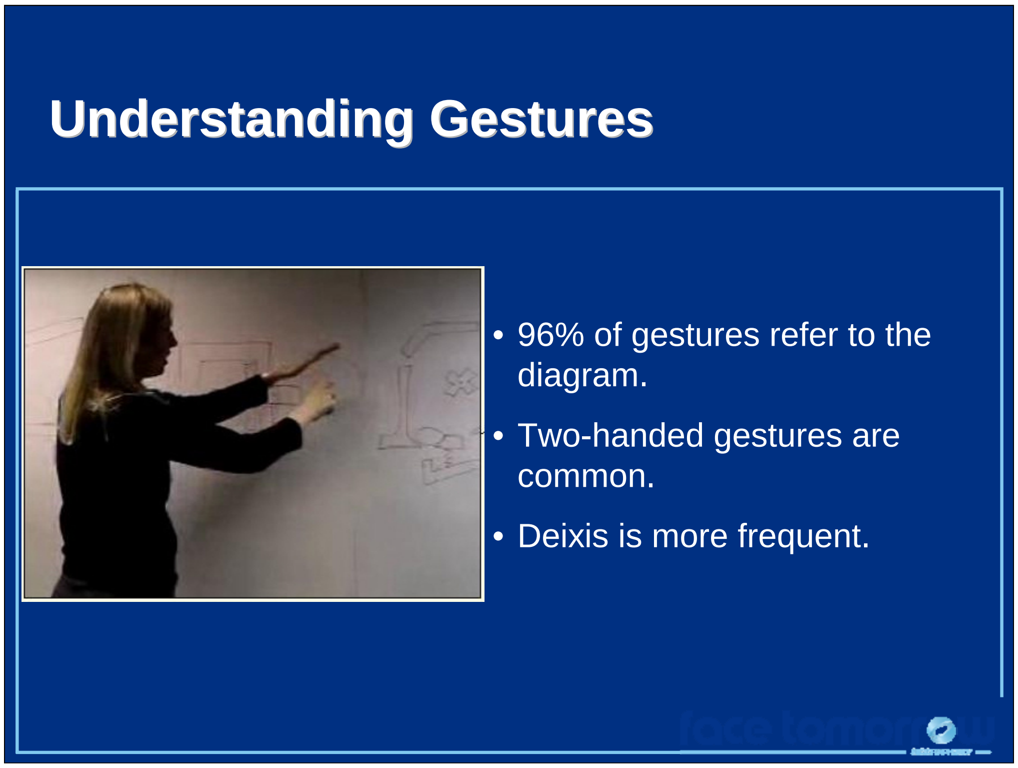


Understanding Gestures

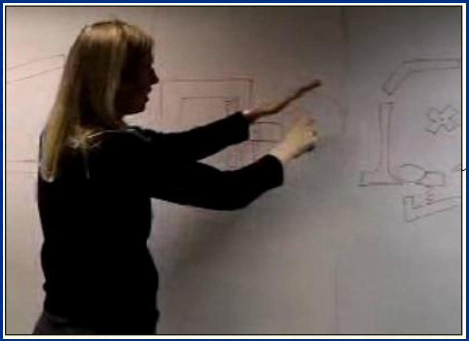
[Eisenstein04]

- Lexicon: What gestures do people make?
- Depends on task
- Our task: explaining how something works.=

Finally, physical gestures provide additional clues about the behavior of a system, and thus the correct interpretation of a sketch. Eisenstein and Davis studied what kinds of gestures people make when describing how a physical device functions.



Understanding Gestures



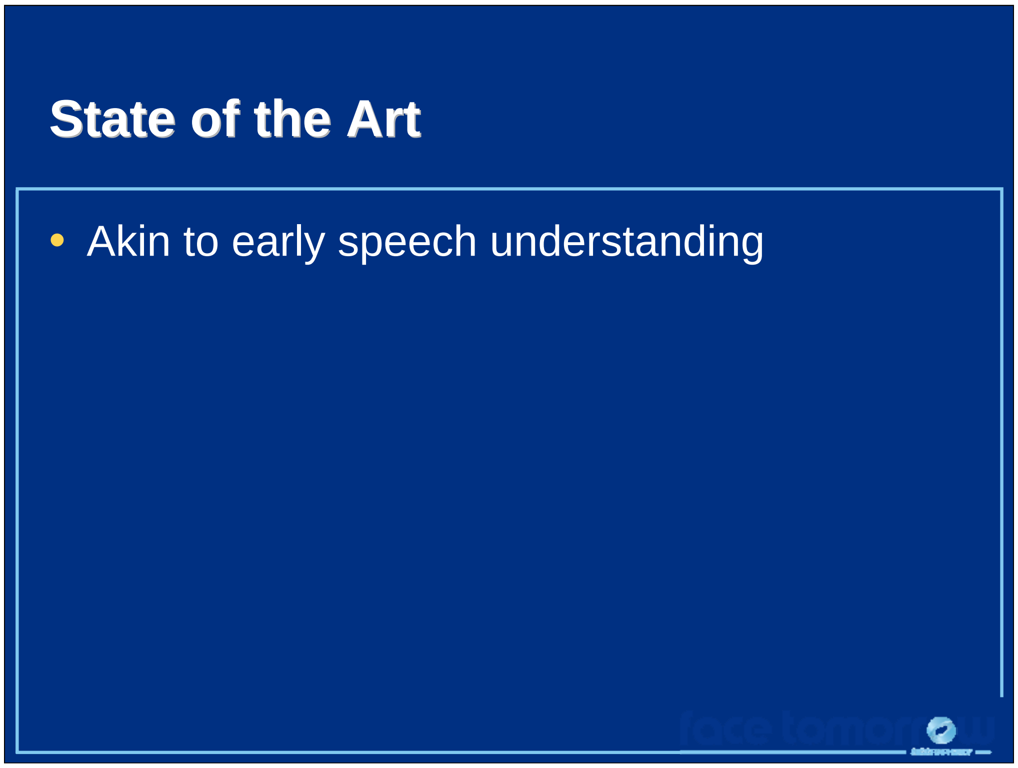
- 96% of gestures refer to the diagram.
- Two-handed gestures are common.
- Deixis is more frequent.

They found that the gestures people make when referring to diagrams are different from the gestures they make when interacting with other people. They also showed that gestures in reference to a diagram can be used to disambiguate associated speech.

References:

Jacob Eisenstein and Randall Davis. Visual and Linguistic Information in Gesture Classification. In International Conference on Multimodal Interfaces (ICMI'04), pp.113-120. New York, New York, October 14-15 2004.

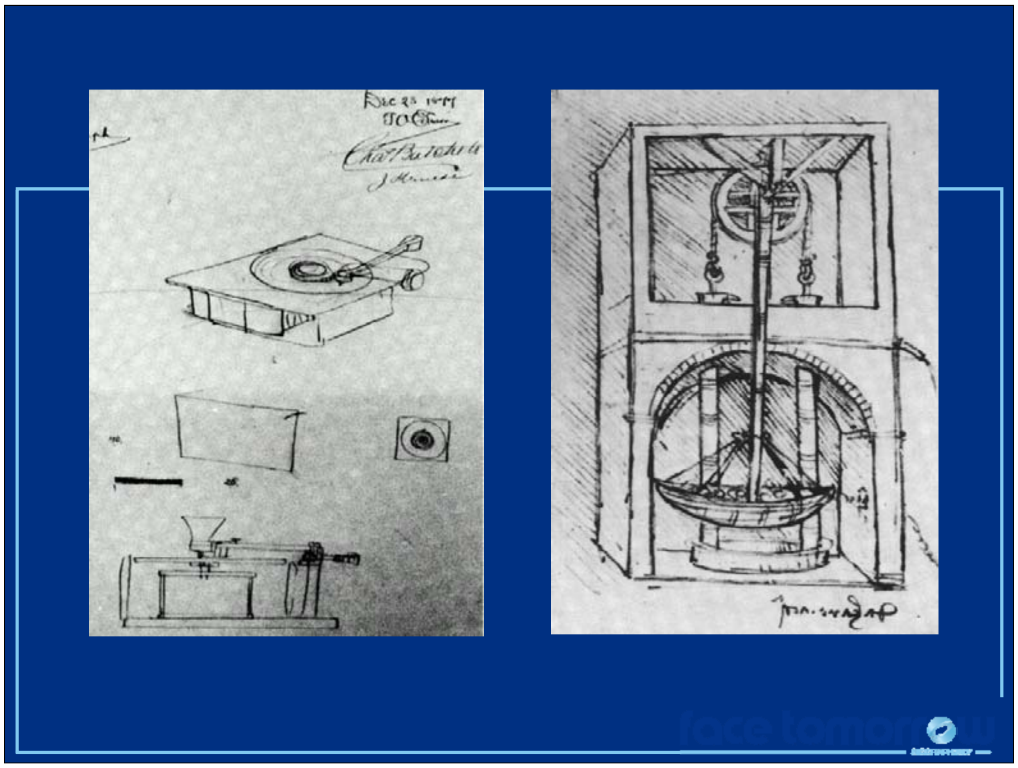
Jacob Eisenstein and Randall Davis. Gesture Improves Coreference Resolution. In Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers, pp.37-40. New York City, USA, June 2006.



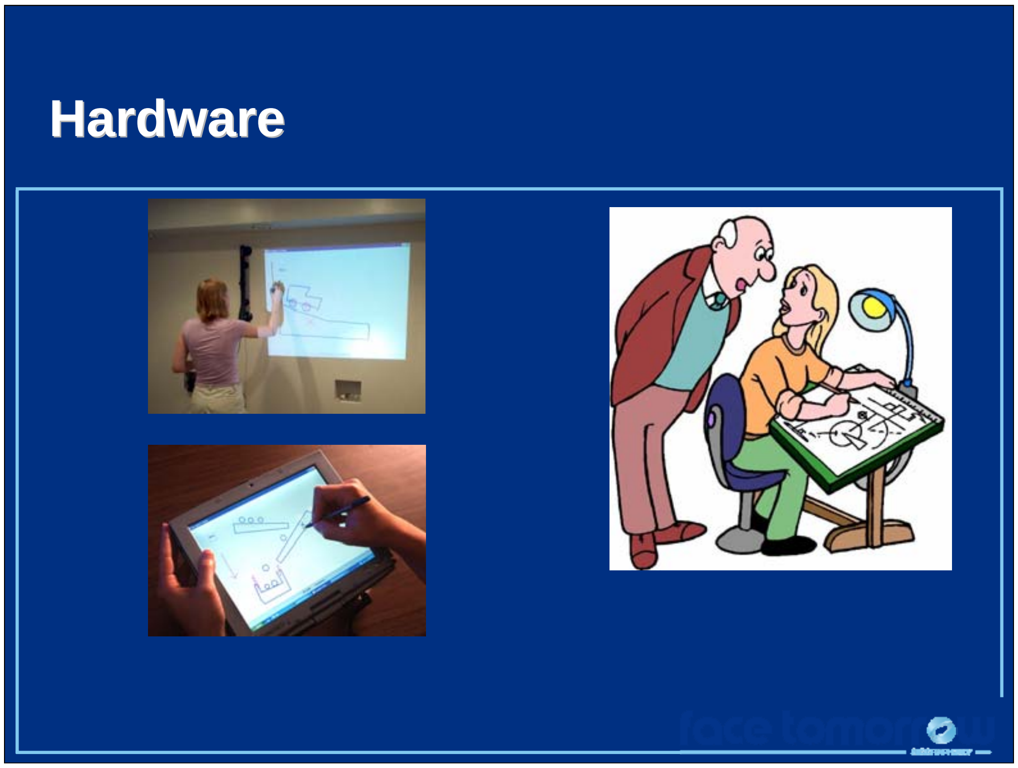
State of the Art

- Akin to early speech understanding

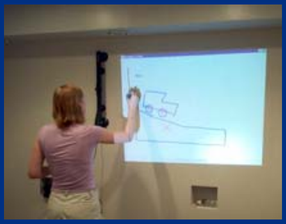
Although we have made great progress over the past decades, there is still a long way to go. In many ways, sketch recognition is at the level that speech recognition was at 15 to 20 years ago. Back then, isolated word recognition was feasible and actually robust enough to be useful, domain-specific continuous speech recognition systems were just beginning to prove useful, but domain-independent continuous speech recognition was far too error-prone to be useful. Today, isolated gesture recognition systems, such as Palm Graffiti, are commonplace. Some domain-specific recognition systems are emerging, but most of these systems require significant user assistance with the process of stroke grouping. Free-sketch recognition, even for restricted domains, remains an unsolved problem.

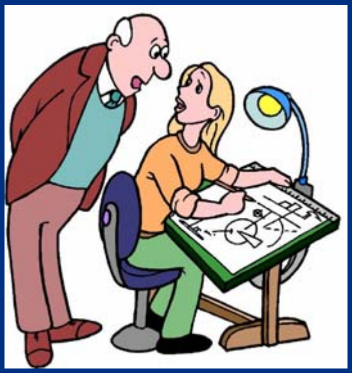


Furthermore, a lot of work on sketch understanding focuses on *symbolic diagram* understanding. Recognizing artistic sketches such as these is well beyond the abilities of any system today.

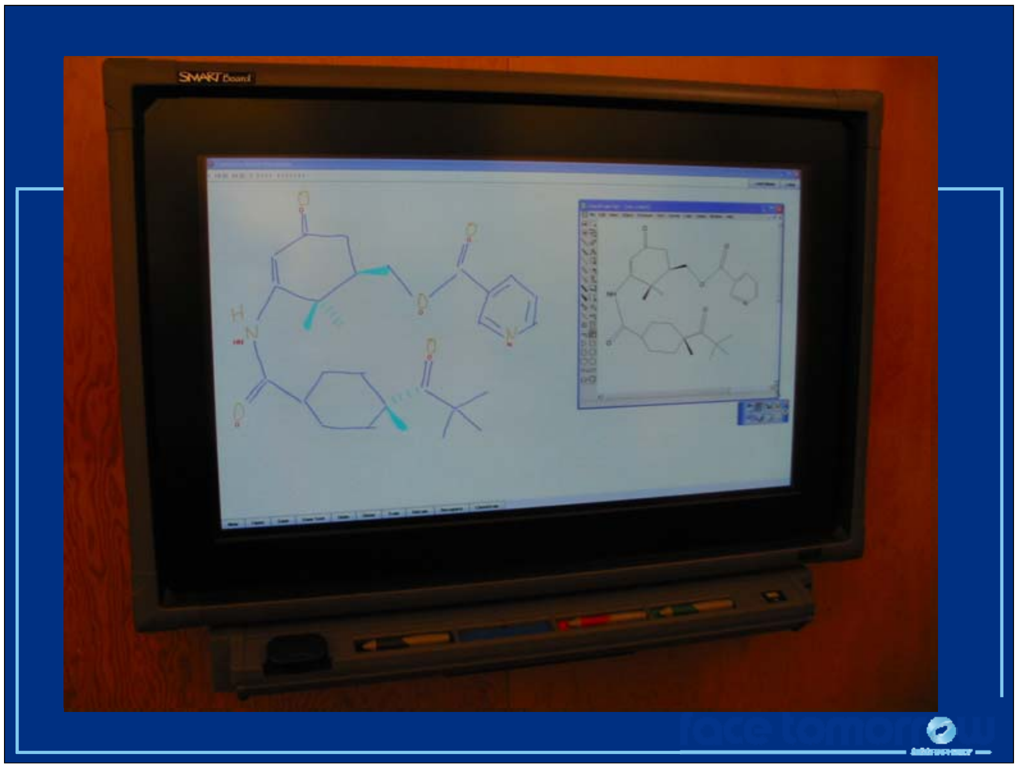


Hardware

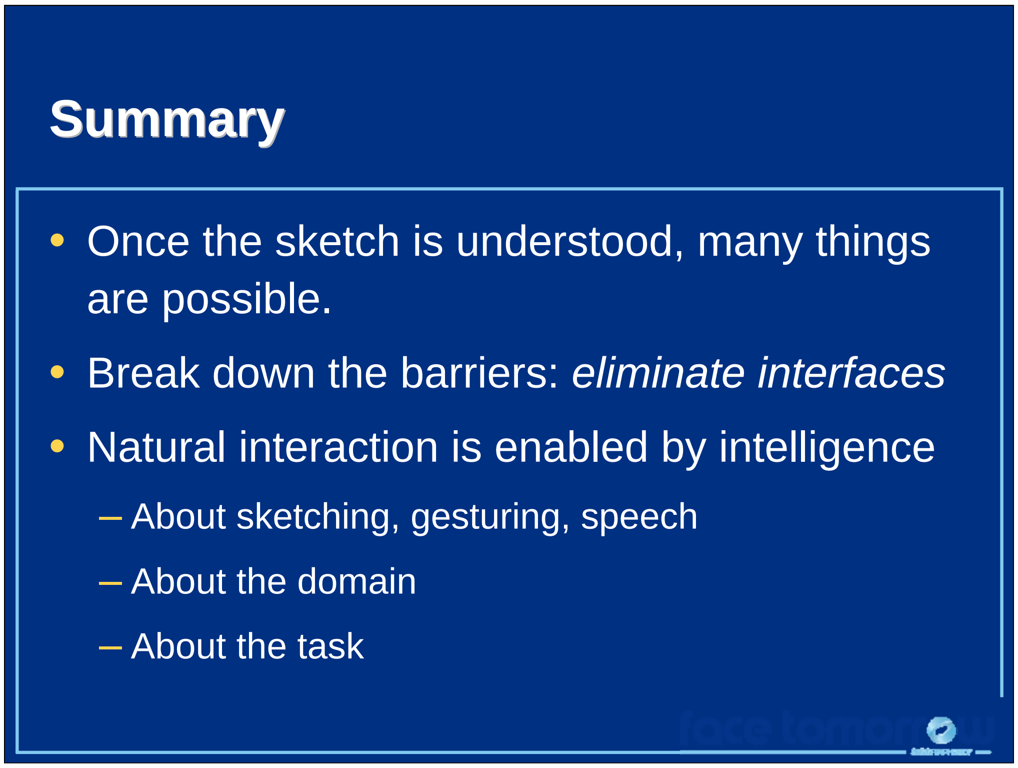




Hardware platforms continue to emerge, from electronic whiteboards, to electronic drafting tables, to the increasingly ubiquitous Tablet PC.



With the rise of digital pen-based hardware, sketch understanding is becoming more relevant than ever.

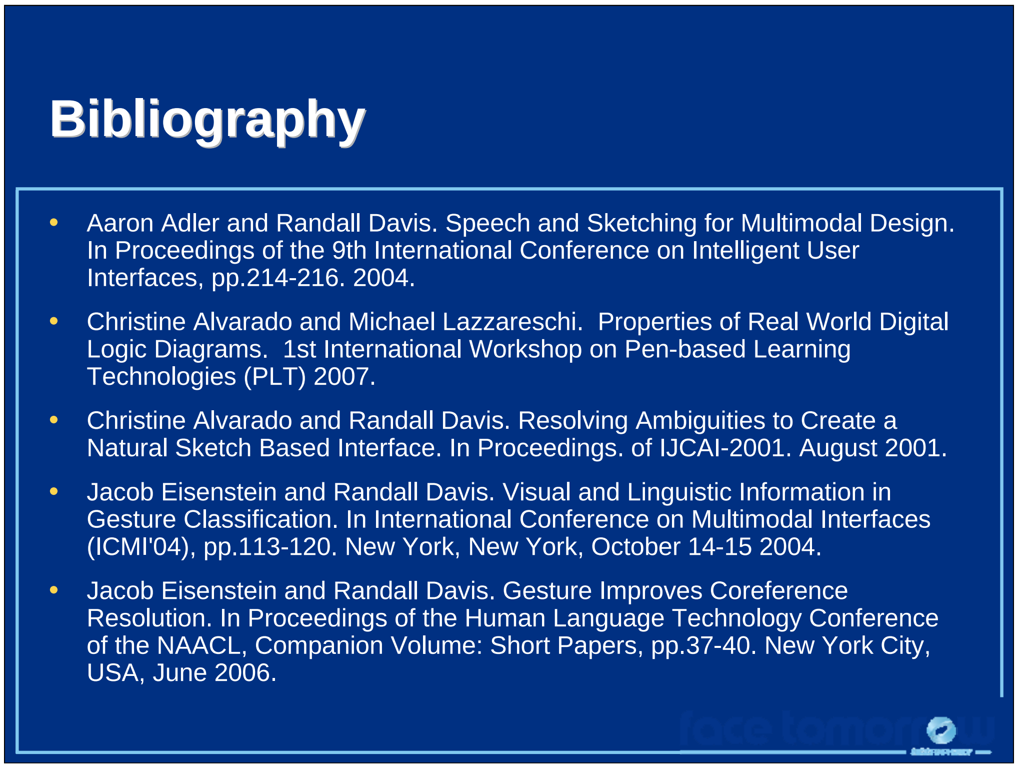


Summary

- Once the sketch is understood, many things are possible.
- Break down the barriers: *eliminate interfaces*
- Natural interaction is enabled by intelligence
  - About sketching, gesturing, speech
  - About the domain
  - About the task

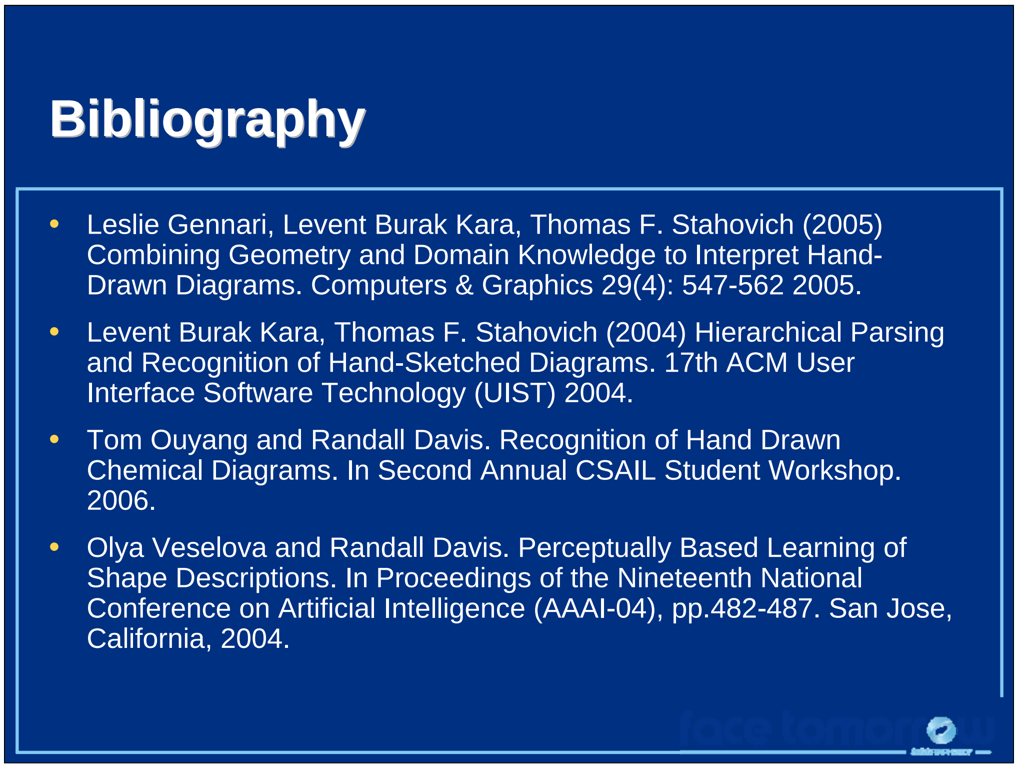
True sketch *understanding* will open the door to many new types of interactions with the computer. The better we can understand the user's sketch, the less she will have to think about the interface and the more she can focus on the more important task of design.

Intelligence is critical for natural interaction. To understand what the user is drawing, the computer must understand the symbols in her sketch, in her speech and in her gestures. This understanding requires detailed knowledge about the domain and the task. This afternoon we will explore this problem further, examining how to build a system that is capable taking this domain information as input, allowing us to build a single sketch understanding system that may be adapted easily to many different domains.



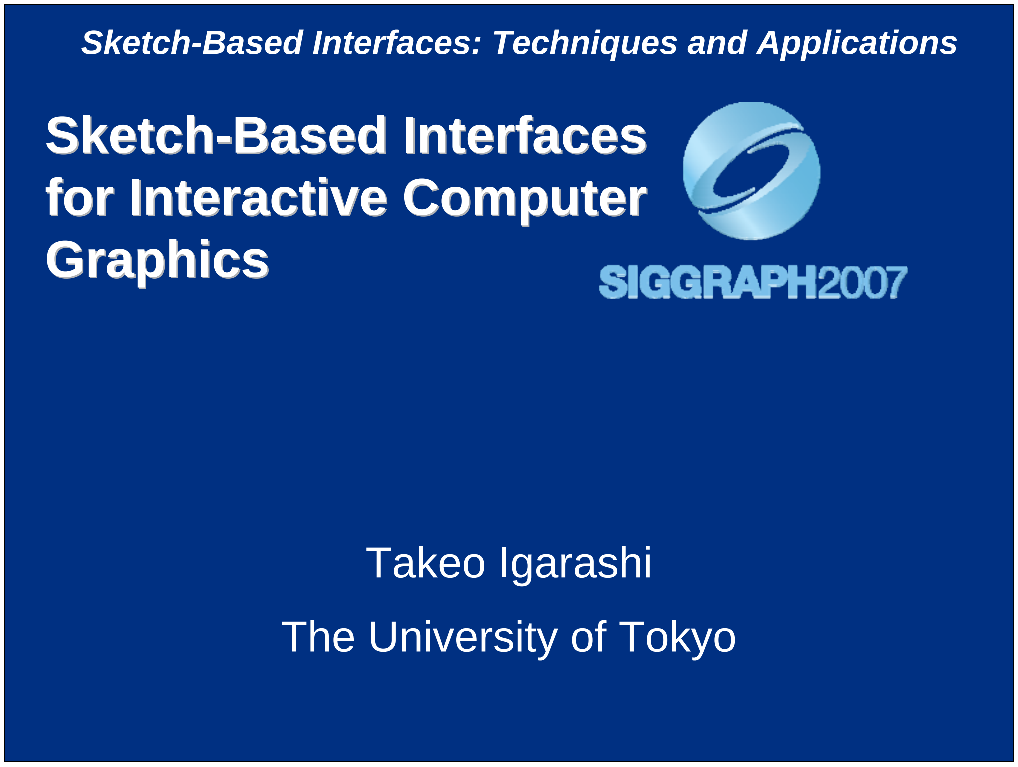
Bibliography

- Aaron Adler and Randall Davis. Speech and Sketching for Multimodal Design. In Proceedings of the 9th International Conference on Intelligent User Interfaces, pp.214-216. 2004.
- Christine Alvarado and Michael Lazzareschi. Properties of Real World Digital Logic Diagrams. 1st International Workshop on Pen-based Learning Technologies (PLT) 2007.
- Christine Alvarado and Randall Davis. Resolving Ambiguities to Create a Natural Sketch Based Interface. In Proceedings. of IJCAI-2001. August 2001.
- Jacob Eisenstein and Randall Davis. Visual and Linguistic Information in Gesture Classification. In International Conference on Multimodal Interfaces (ICMI'04), pp.113-120. New York, New York, October 14-15 2004.
- Jacob Eisenstein and Randall Davis. Gesture Improves Coreference Resolution. In Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers, pp.37-40. New York City, USA, June 2006.



Bibliography

- Leslie Gennari, Levent Burak Kara, Thomas F. Stahovich (2005) Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams. Computers & Graphics 29(4): 547-562 2005.
- Levent Burak Kara, Thomas F. Stahovich (2004) Hierarchical Parsing and Recognition of Hand-Sketched Diagrams. 17th ACM User Interface Software Technology (UIST) 2004.
- Tom Ouyang and Randall Davis. Recognition of Hand Drawn Chemical Diagrams. In Second Annual CSAIL Student Workshop. 2006.
- Olya Veselova and Randall Davis. Perceptually Based Learning of Shape Descriptions. In Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), pp.482-487. San Jose, California, 2004.



Sketch-Based Interfaces: Techniques and Applications

Sketch-Based Interfaces for Interactive Computer Graphics

SIGGRAPH2007

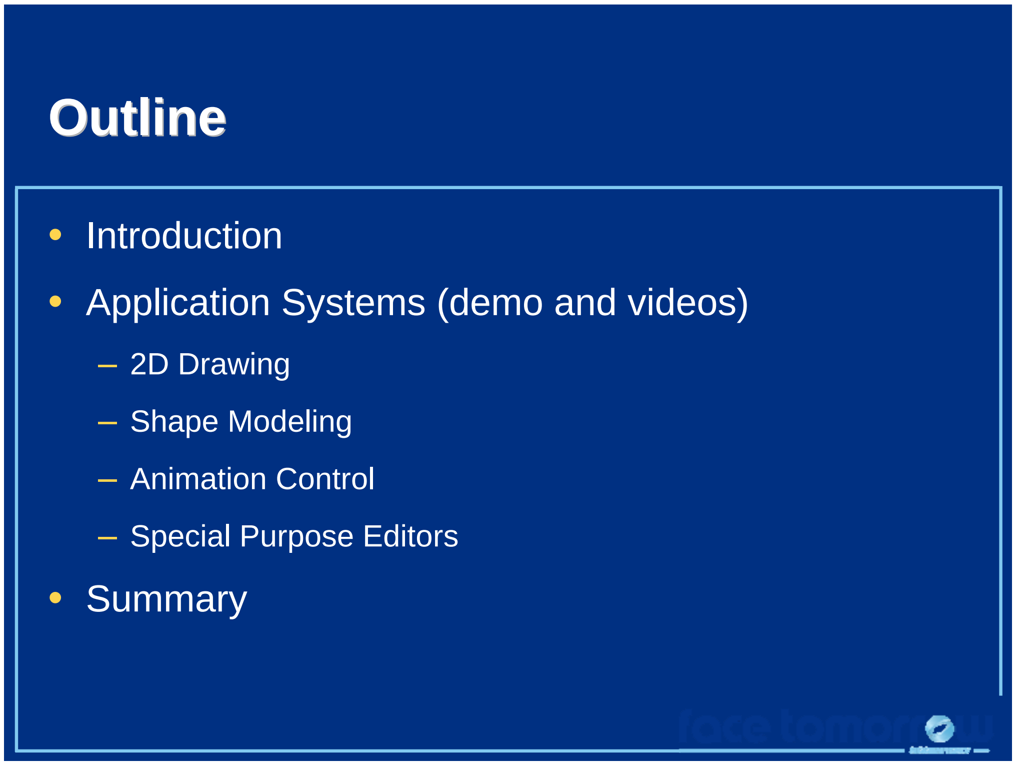
Takeo Igarashi

The University of Tokyo

Sketch-Based Interfaces for Interactive Computer Graphics

Takeo Igarashi
Associate Professor, University of Tokyo
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033
Tokyo, JAPAN

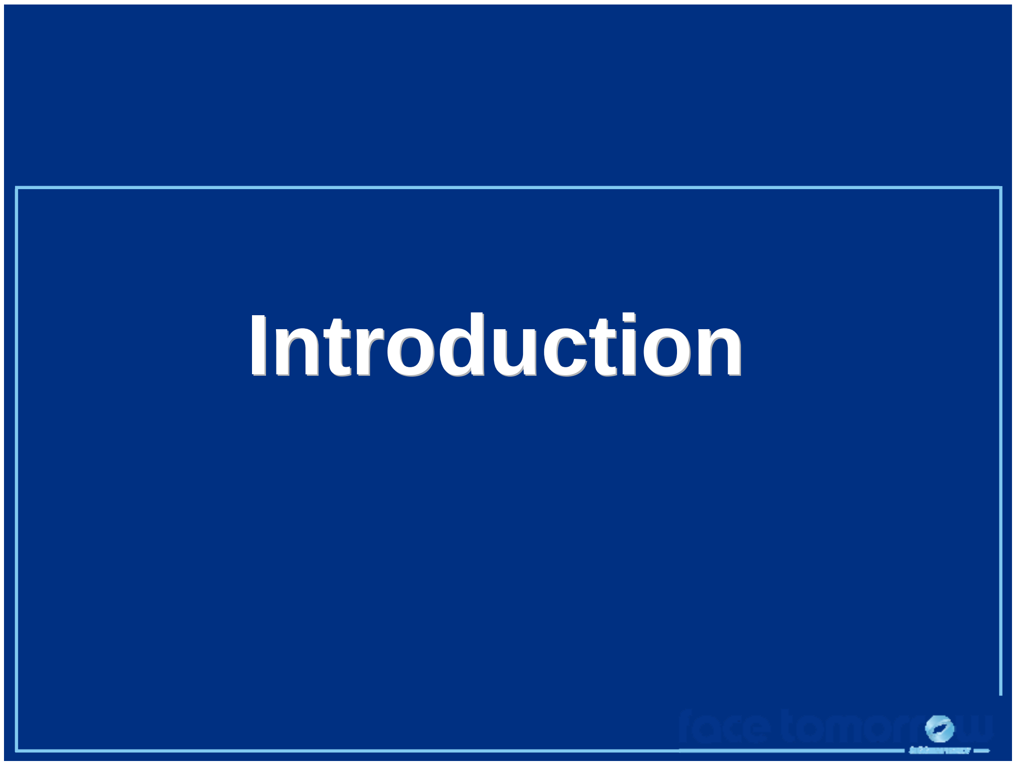
Email: takeo@acm.org
www-ui.is.s.u-tokyo.ac.jp/~takeo/



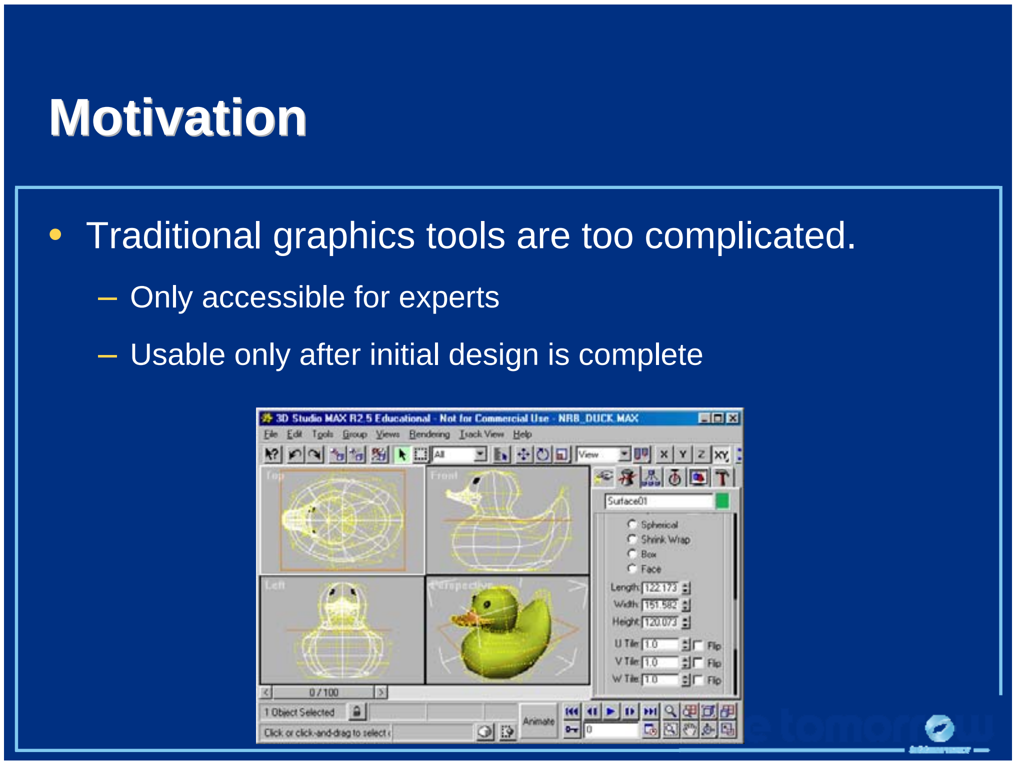
Outline

- Introduction
- Application Systems (demo and videos)
  - 2D Drawing
  - Shape Modeling
  - Animation Control
  - Special Purpose Editors
- Summary

In this lecture, we will introduce various sketching interfaces for computer graphics applications. We first give a short introduction and then discuss several sketch-based systems.

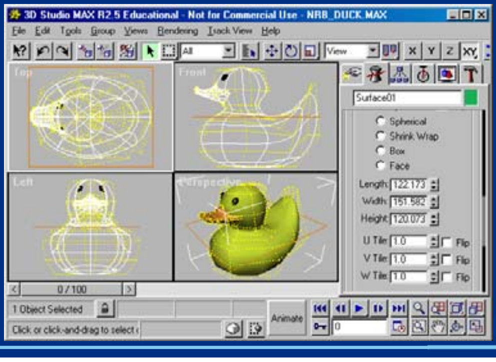


Introduction



Motivation

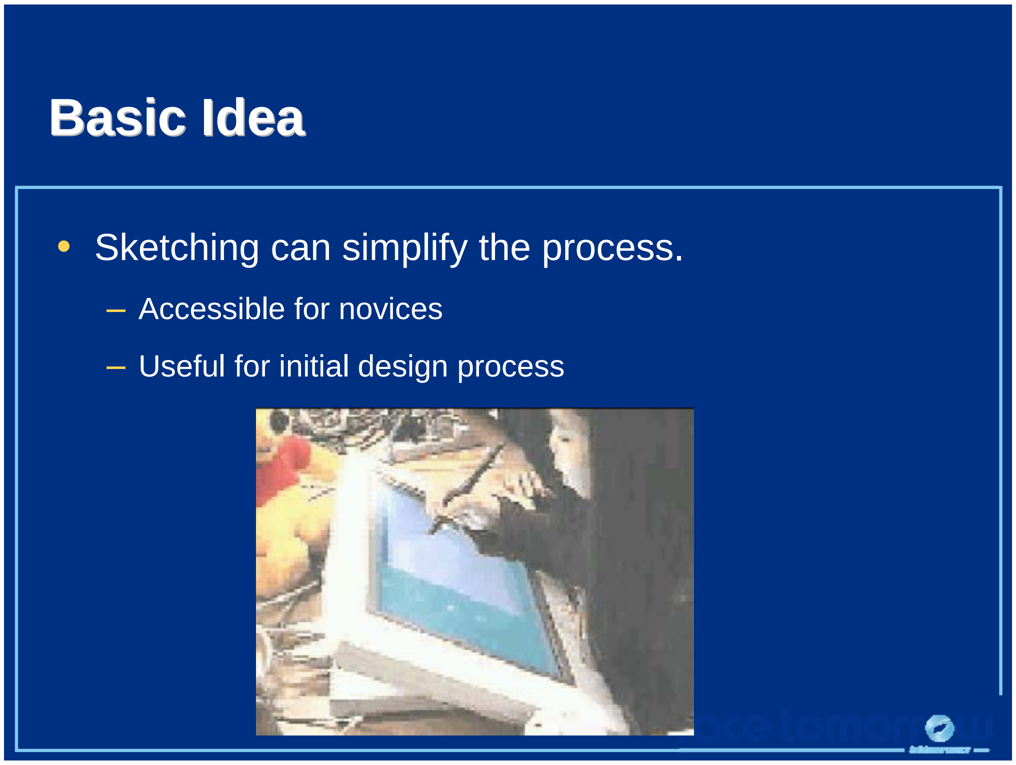
- Traditional graphics tools are too complicated.
  - Only accessible for experts
  - Usable only after initial design is complete



Traditional tools for authoring computer graphics are mainly designed for expert users and are very difficult to use. One must spend a long time learning the tool and is expected to work on a task for a extended period of time to obtain a result.

These tools are certainly suitable for professional artists who create production quality graphics (movies, games, etc.), but they are not accessible for novice users. Suppose a child wants to make a 3D duck. These traditional interfaces are too complicated for beginners.

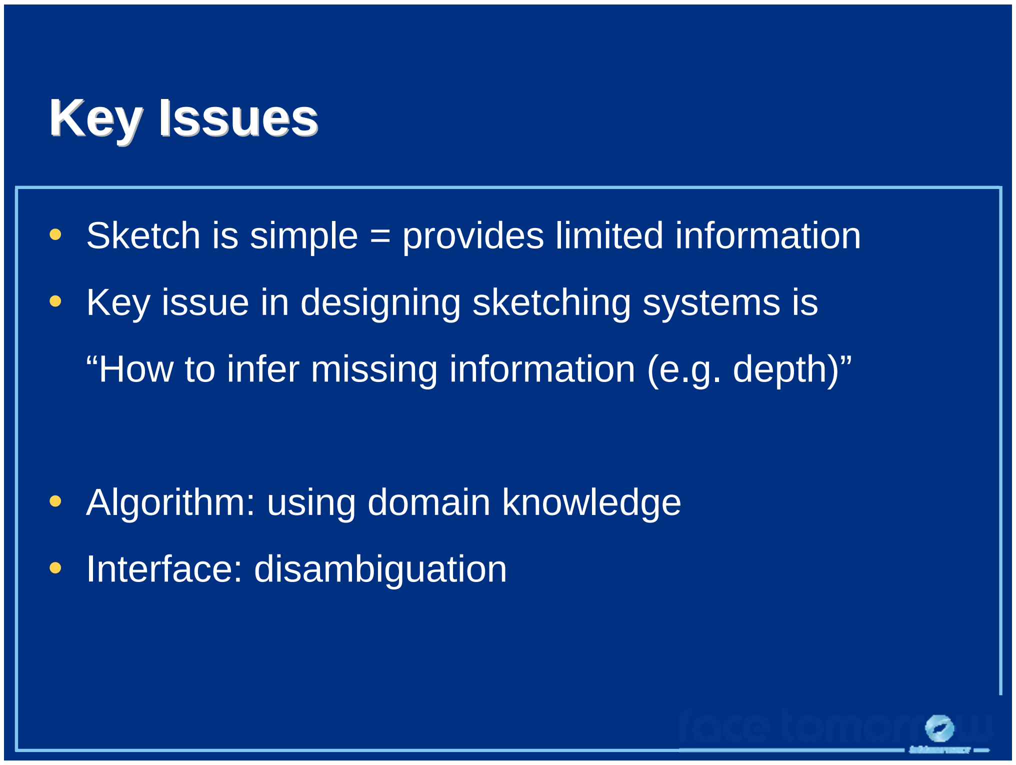
Complicated interfaces can also make it difficult for professional users to use these tools in the initial design phase. As a result, artists still first work on pen and paper in initial design phase, and then move on to computational tools when the design is fixed.



Basic Idea

- Sketching can simplify the process.
  - Accessible for novices
  - Useful for initial design process

The goal of sketching interfaces is to solve the problem of accessibility by dramatically simplifying the user interface. Instead of requiring the user to work with traditional buttons, menus, and dragging operations, sketching allows them to directly express their thoughts in the form of freehand sketching. It can make 3D graphics authoring accessible to novice users, even children. In addition, simple interfaces make it possible for experts to use them in the initial design process.

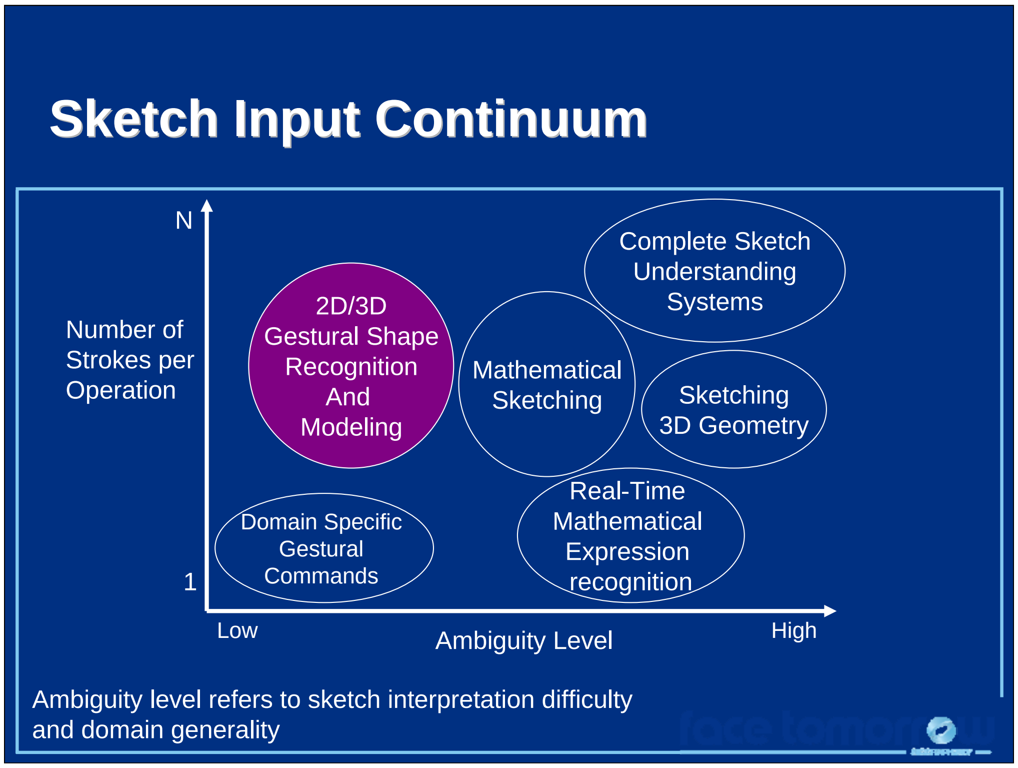


Key Issues

- Sketch is simple = provides limited information
- Key issue in designing sketching systems is
  “How to infer missing information (e.g. depth)”
- Algorithm: using domain knowledge
- Interface: disambiguation

The basic idea behind sketching systems is to simplify the interface by reducing the amount of explicit control by the user. As a result, the information provided by the user is not complete to perform a task, and it is necessary to complement missing information in various ways. Complementing missing information is the key issue in designing a sketching interface. For example, a typical problem in sketch-based 3D modeling is how to infer depth information. The user’s input is 2D, so the system needs to complement depth information in some way.

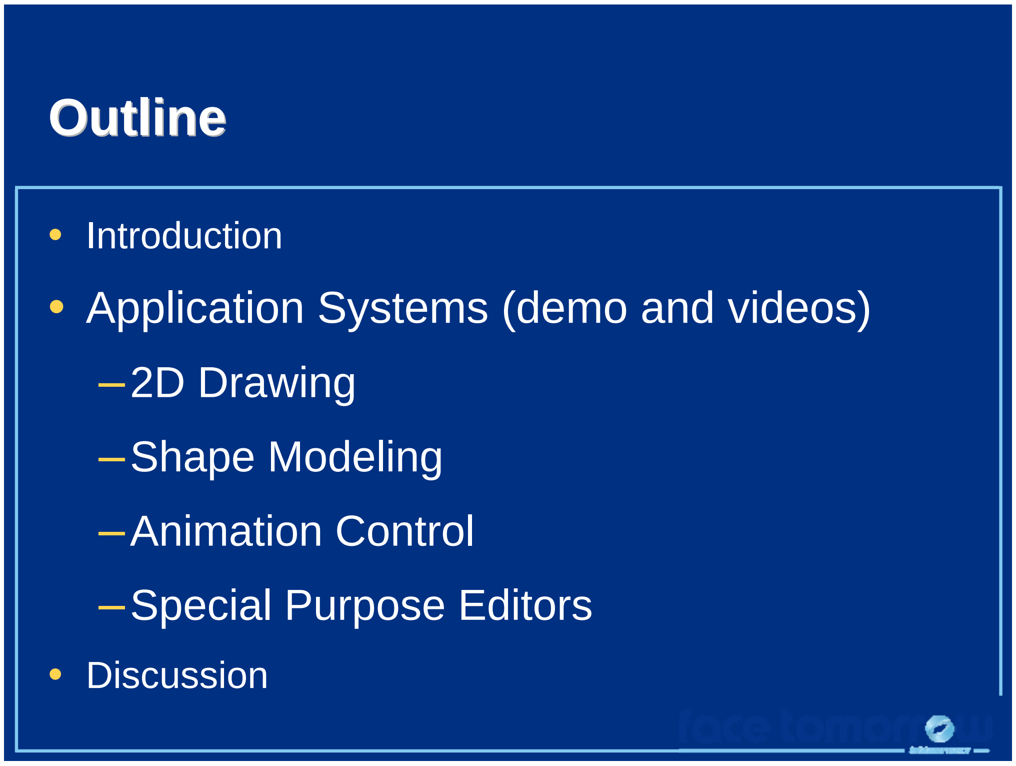
Sketching interfaces solve this problem by combining algorithm and interface tricks. With algorithms, sketching systems heavily exploit domain knowledge to infer appropriate information from limited input. In other words, the interface achieves simplicity by limiting the target domain. With the user interface, sketching systems need to handle ambiguity in the user input. Construction of multiple candidates are one of very useful method to do this.



Sketch Input Continuum

N
Number of Strokes per Operation
1
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Complete Sketch Understanding Systems
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition
Low
Ambiguity Level
High

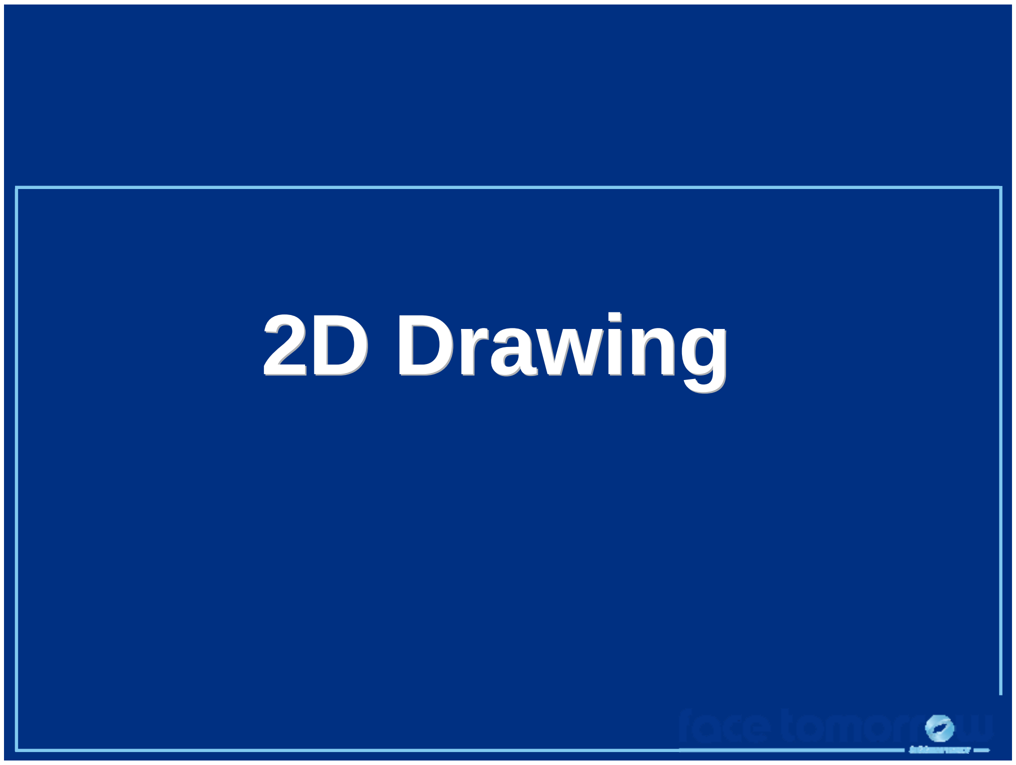
Ambiguity level refers to sketch interpretation difficulty and domain generality



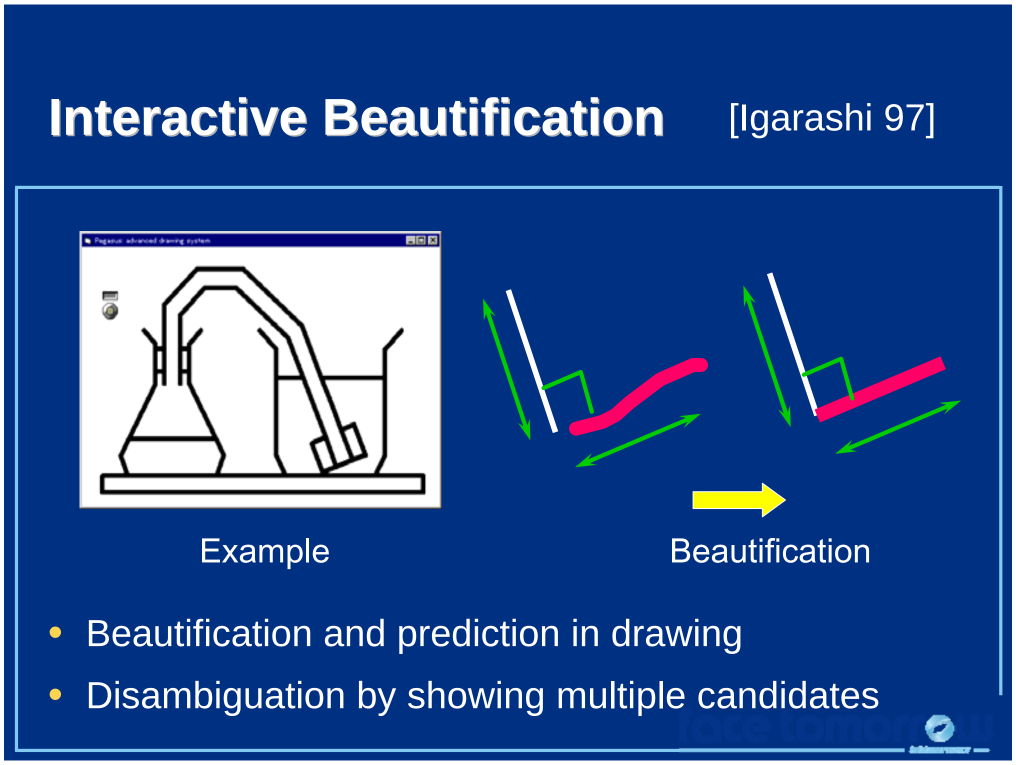
Outline

- Introduction
- Application Systems (demo and videos)
  - 2D Drawing
  - Shape Modeling
  - Animation Control
  - Special Purpose Editors
- Discussion

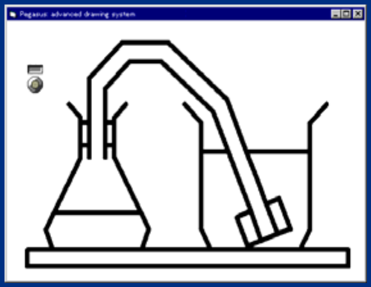
Now we discuss individual systems from different areas of interactive computer graphics.



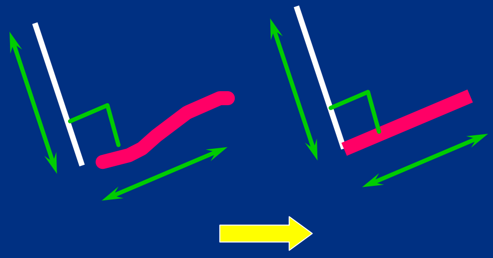
2D Drawing



Interactive Beautification [Igarashi 97]



Example

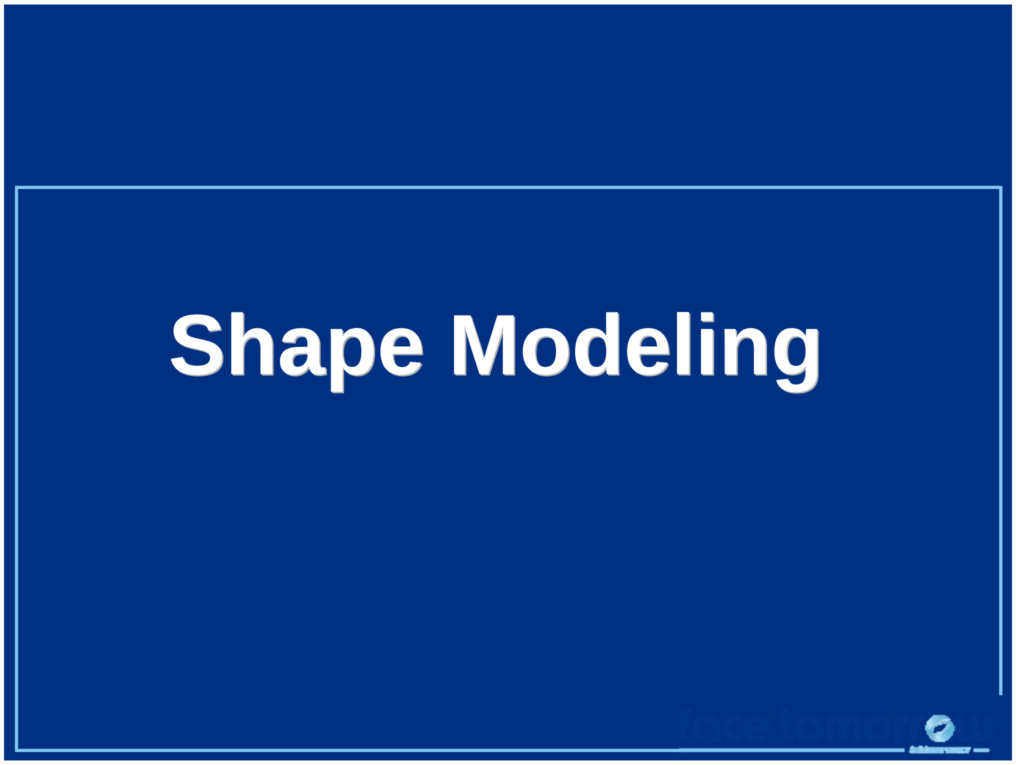


Beautification

- Beautification and prediction in drawing
- Disambiguation by showing multiple candidates

First, we introduce a sketching interface for drawing simple diagrams as shown in the slide. It is tedious to draw these diagrams using traditional drawing editors. The user needs to combine multiple commands such as copy, paste, flip, move, etc. The interactive beautification system simplifies the process by taking the user's freeform sketching and beautifying them considering geometric relationships between line segments. If the user's drawn stroke is almost perpendicular to an existing line segment, the system makes it perfectly perpendicular.

A user's sketches can be very ambiguous. To address this problem, the system shows multiple candidates and let the user choose one. The system also predicts the user's next drawing and shows the result to the user. If the user likes the predictions, he or she can continue drawing just by successive clicking.

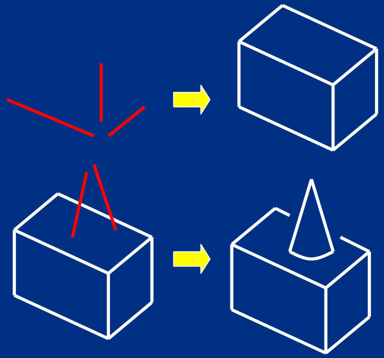


Shape Modeling

.sketch.avi

SKETCH

[Zeleznik 96]



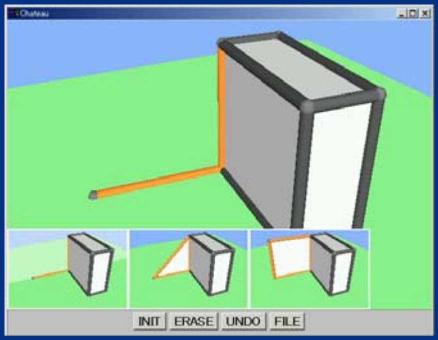
- 3D scene construction using gestures.
- “Every object is on top of another object”

SKETCH is a gestural interface for creating 3D scenes such as those shown here. The user draws a few 2D lines in the scene as a gesture, and the system creates a corresponding 3D primitive. For example, three lines create a box, two parallel lines create a cylinder, and two connected line creates a cone.

The question is how to determine the 3D position (depth) of each 3D primitive. The SKETCH system solved this ambiguity by assuming that every object is on top of another object in the scene. This strategy frees the user from manually specifying the depth of each primitive and it significantly simplifies the interface. The SKETCH system also introduces other interesting interaction techniques such as floating an object in the air by drawing its shadow.

Suggestive Interfaces

[Igarashi 01]

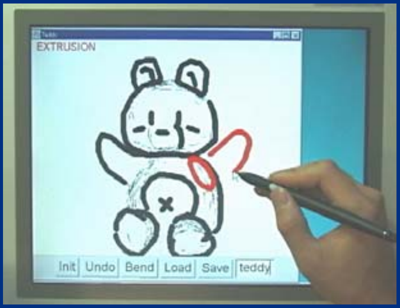


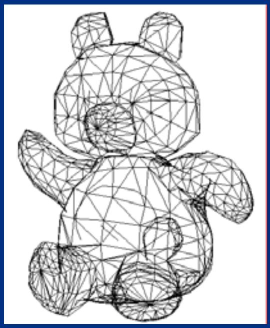
- User provides hints, system shows suggestions
- Disambiguation by showing multiple candidates

The figures in the slide show a system that is a combination of the Pegasus system and the SKETCH system. The user draws a line by dragging and highlights a line by clicking. The system then shows the result of possible operations related to the highlighted line segments. If two perpendicular lines are highlighted, the system suggests to create a new drawing canvas, a triangle, and a box. The user can simply click on the desired suggestion and the scene is updated accordingly.

Teddy

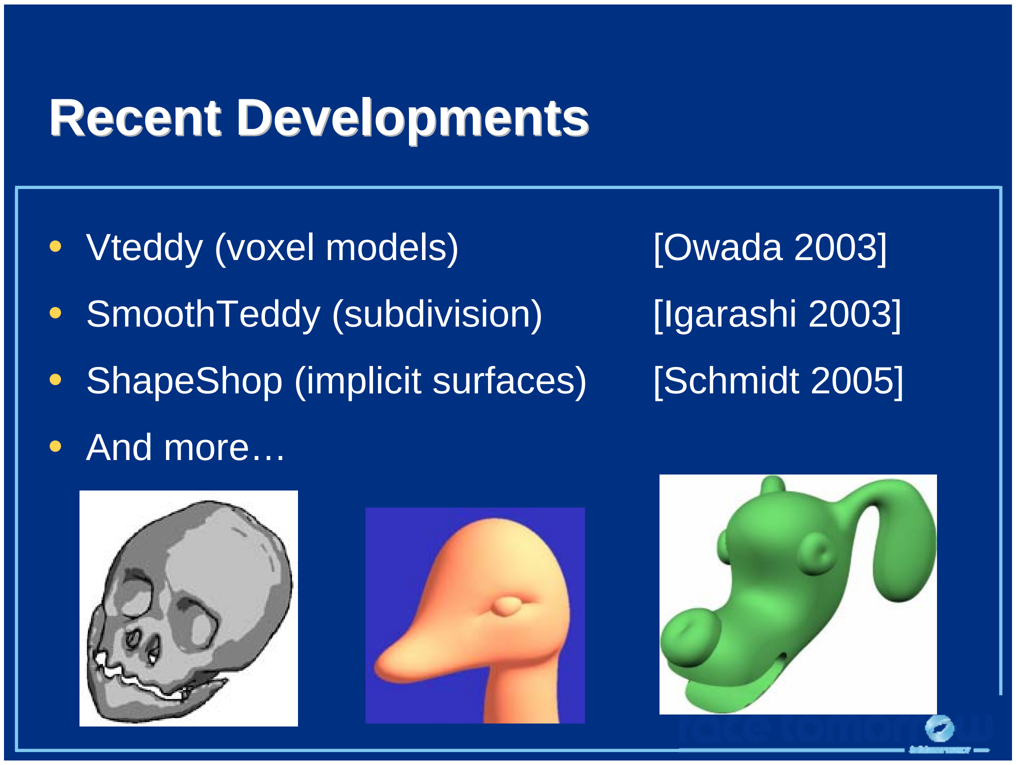
[Igarashi 99]





- Freeform models from sketching.
- “Sketches represent some rotund shapes”

Previous sketch-based modeling systems focus on rectiliniear models such as buildings. In contrast, Teddy is designed for freeform models such as animals. The user draws the silhouette of the desired geometry and the system automatically create a 3D model by inflating the region surrounded by the silhouette. Teddy also supports various editing operations such as cutting, extrusion, and transformation. All operations are performed by drawing freeform strokes on the screen and they and appropriately interpreted by the system considering the context information.



Recent Developments

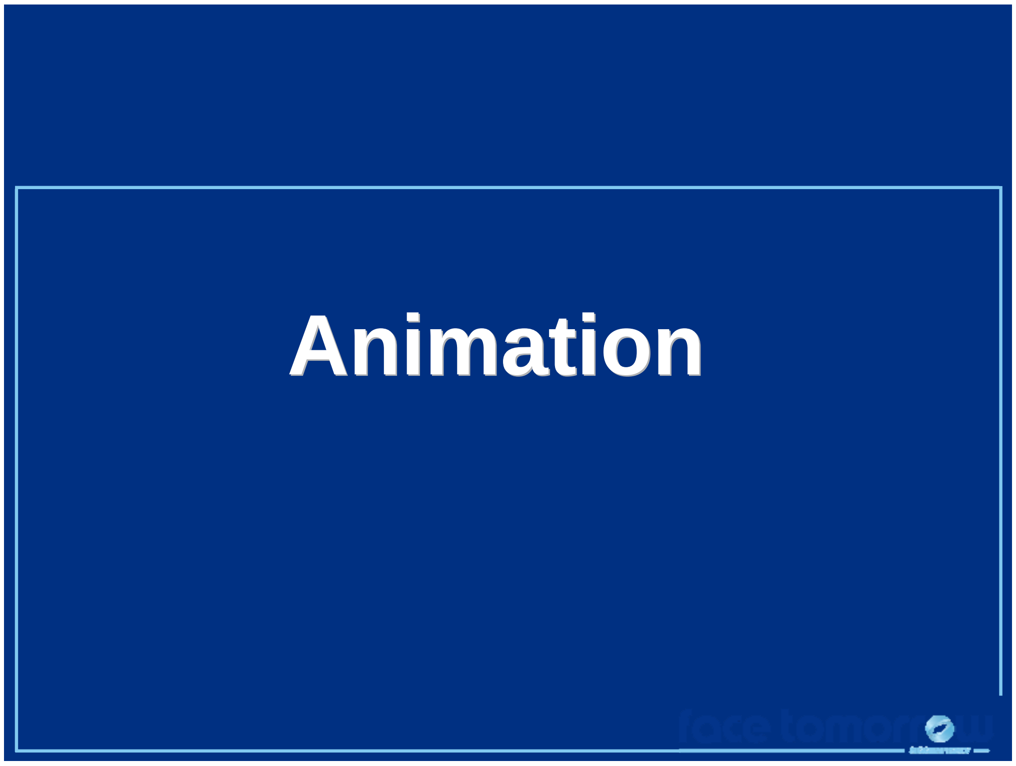
- Vteddy (voxel models) [Owada 2003]
- SmoothTeddy (subdivision) [Igarashi 2003]
- ShapeShop (implicit surfaces) [Schmidt 2005]
- And more…

There are several extensions to the original Teddy system. The original Teddy system used a very coarse mesh model and these extensions experiment with different representations.

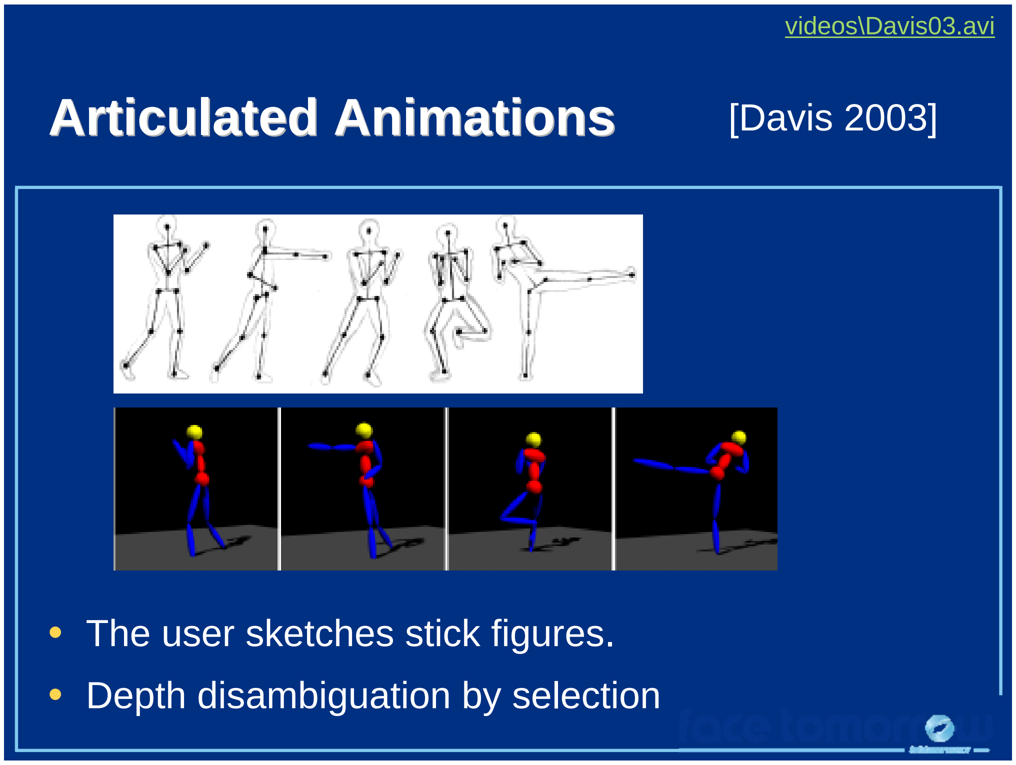
Vteddy uses a voxel representation for 3D models to handle complicated internal structures. The original Teddy system was limited to models with topology identical to a sphere, but Vteddy can handle models with arbitrary topology. It provides a “temporary cutting” operation for editing internal structures. Vteddy temporarily reveals the cross section of a model and allows the user to edit internal structures. When it is done, the system automatically closes the cross section.

Smoothteddy beautifies the mesh generated by the original Teddy algorithms. It iteratively updates the mesh connectivity and vertex positions to yield equal-sized, near-regular triangles. The resulting mesh is suitable for creating a refined mesh by applying subdivision.

ShapeShop uses a blob-tree representation. Blob-trees are hierarchically organized implicit surfaces and can naturally represent smooth surfaces. It is also possible to move added parts to different locations afterwards while preserving a smooth connection.



Animation

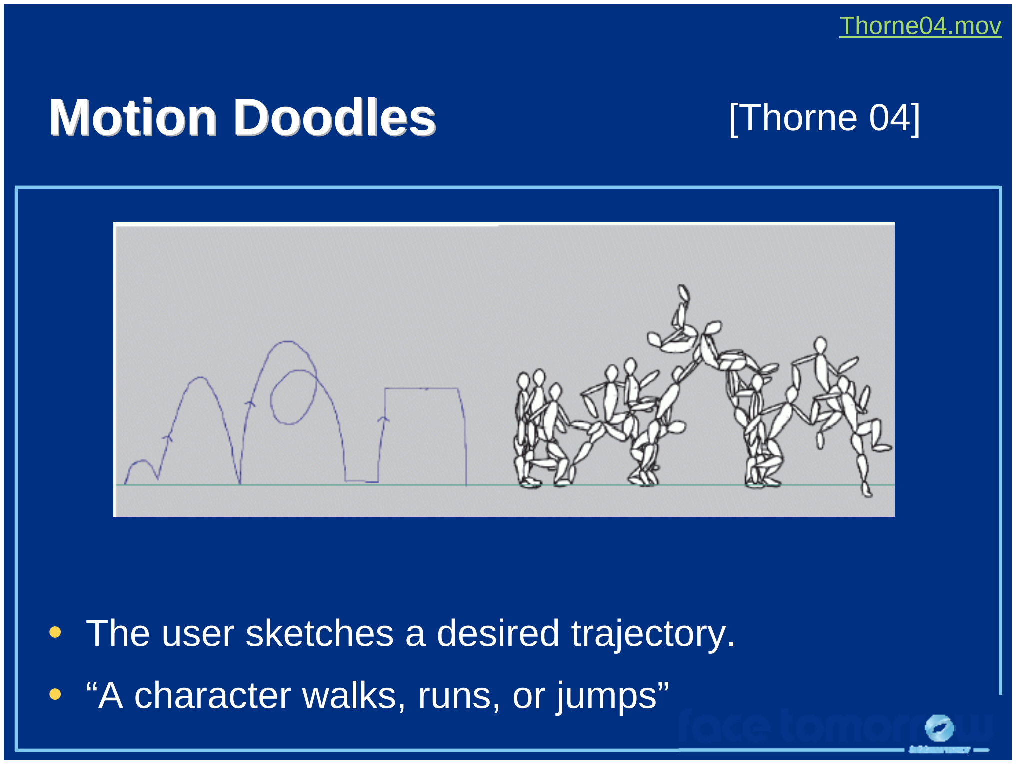


videos\Davis03.avi

Articulated Animations [Davis 2003]

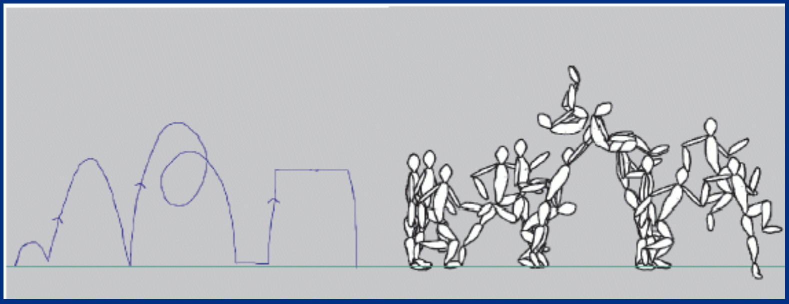
- The user sketches stick figures.
- Depth disambiguation by selection

The animation system shown in the slide takes a series of stick figure drawings as input and generates a character animation. The key issue is how to define the depth (direction) of the body parts. This system solves the problem by using pre-defined constraints and by asking the user to choose one from multiple candidates.



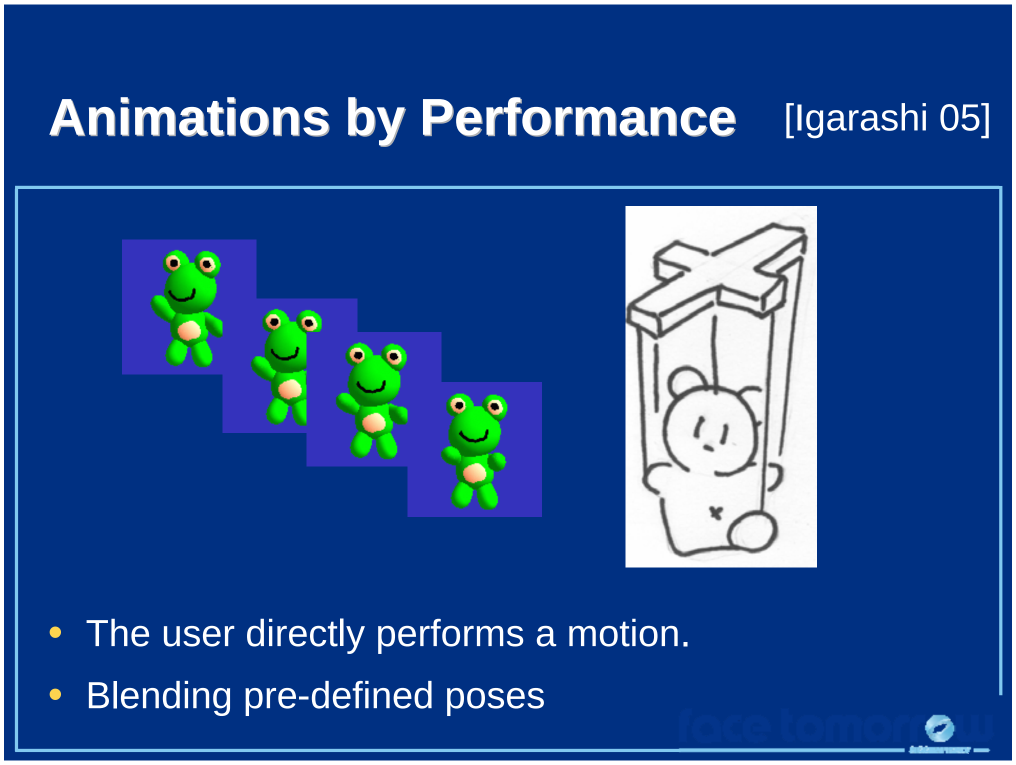
Thorne04.mov

Motion Doodles [Thorne 04]

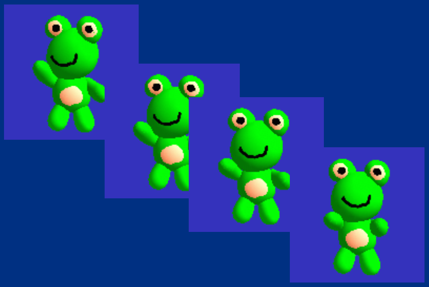


- The user sketches a desired trajectory.
- “A character walks, runs, or jumps”

In the previous system, the user draws each pose. With Motion Doodles, the user draws a desired trajectory of a character. The shape and timing of the sketching is used to control the motion. The system supports a repertoire of 18 different types of motions and combines them according to the sketch.



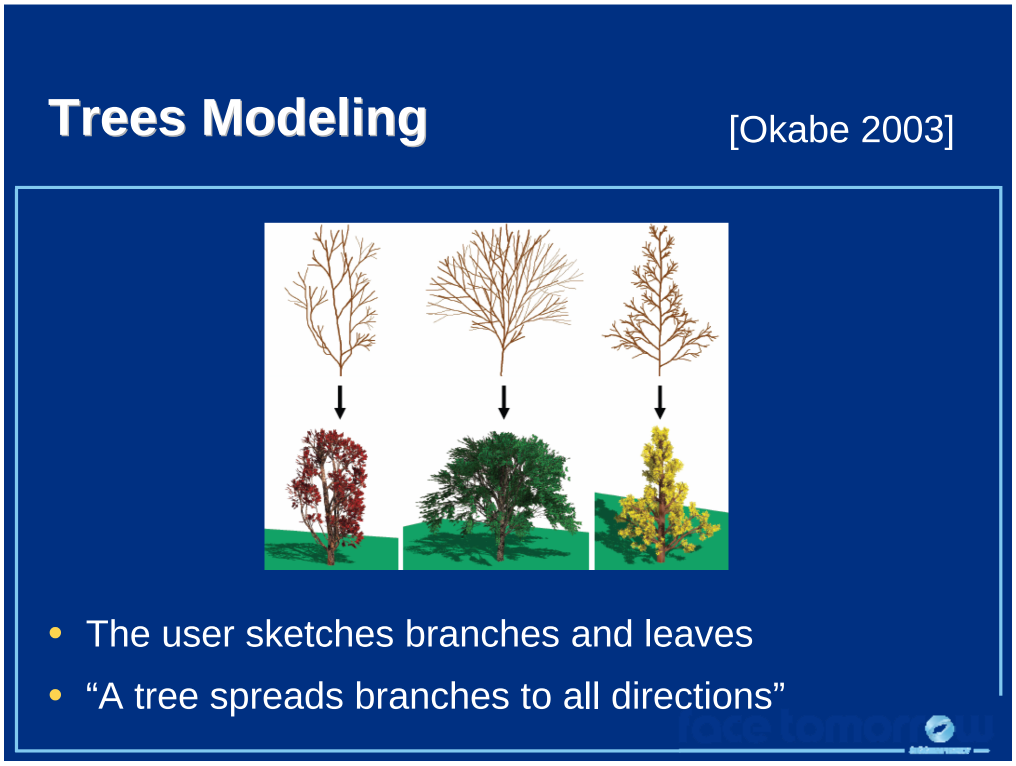
Animations by Performance [Igarashi 05]



- The user directly performs a motion.
- Blending pre-defined poses

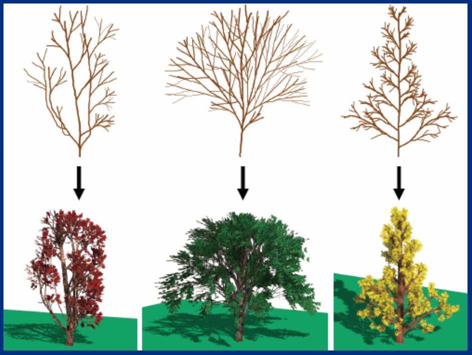
With Animations by Performance, the system records the user's performance (direct manipulation process) and plays back the recorded motion as an animation. To allow the user to simultaneously control multiple degrees of freedom, the user specifies a set of key poses beforehand and the system blends them according to the user's cursor movement during recording.

Special Purpose Editors



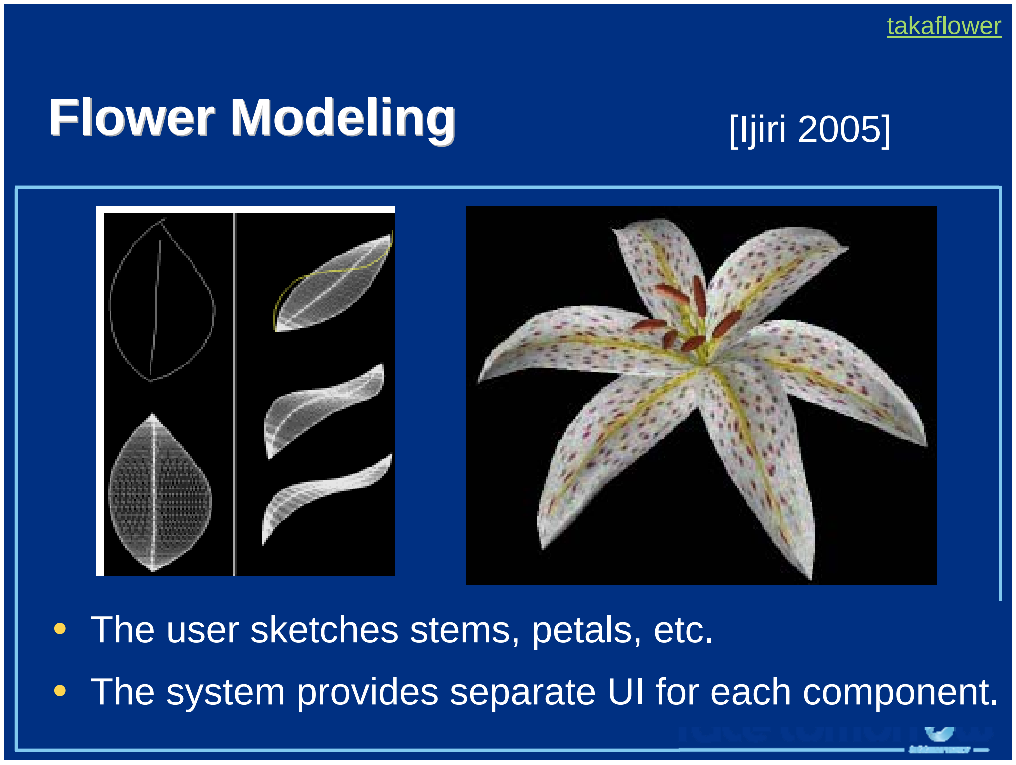
Trees Modeling

[Okabe 2003]



- The user sketches branches and leaves
- “A tree spreads branches to all directions”

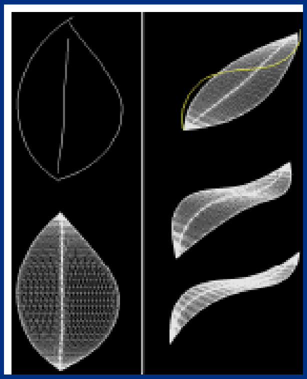
The system shown in the slide takes a 2D sketch of a tree drawn by the user and generates a 3D tree. It computes the depth of branches so that the distance between branches become as large as possible. Since it is tedious to draw all branches manually, the system provides interfaces to generate many similar branches by using a limited number of manually drawn branches as examples.



takaflower

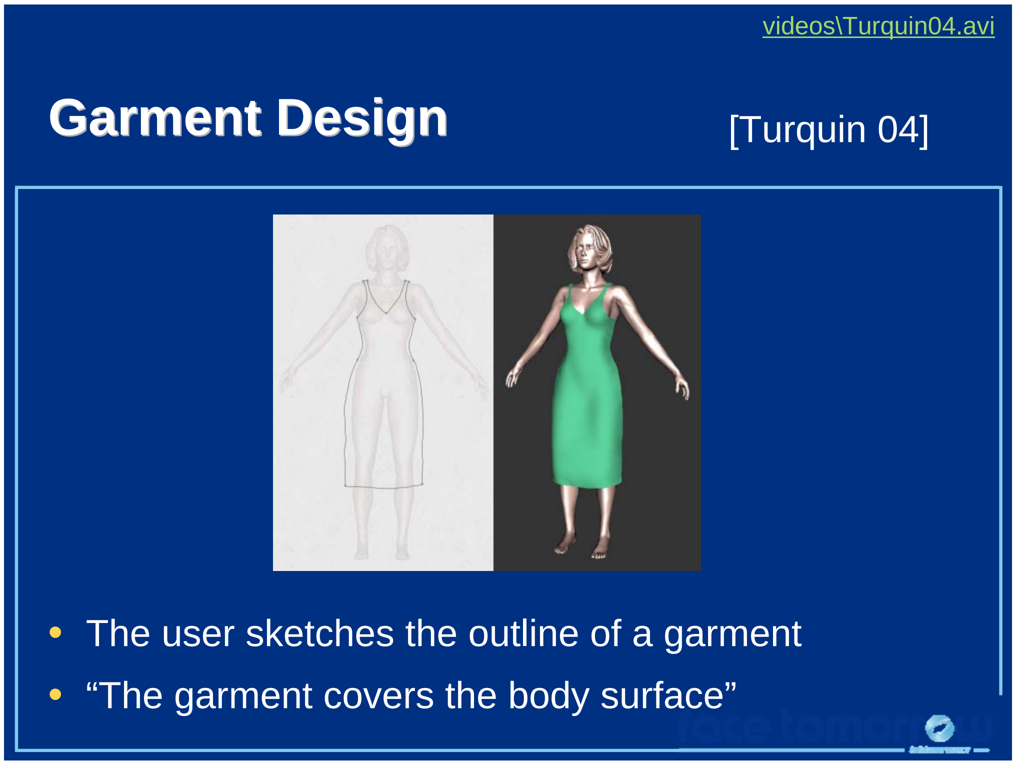
Flower Modeling

[Ijiri 2005]



- The user sketches stems, petals, etc.
- The system provides separate UI for each component.

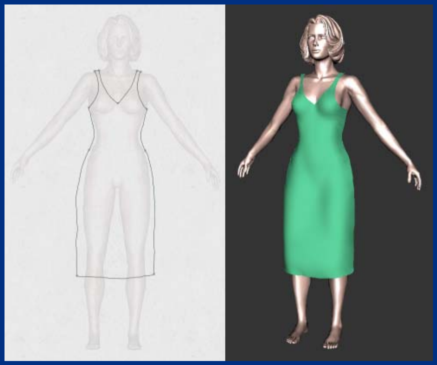
This system models flowers. It combines geometry editing and structural editing to make the construction of complicated models possible. The geometry modeling part is a collection of special purpose sketching interfaces for individual floral components, such as receptacle, anther, sepal, and petal. The structural editing part provides a simple diagrammatic interface for specifying the arrangement of the geometric components.



videos\Turquin04.avi

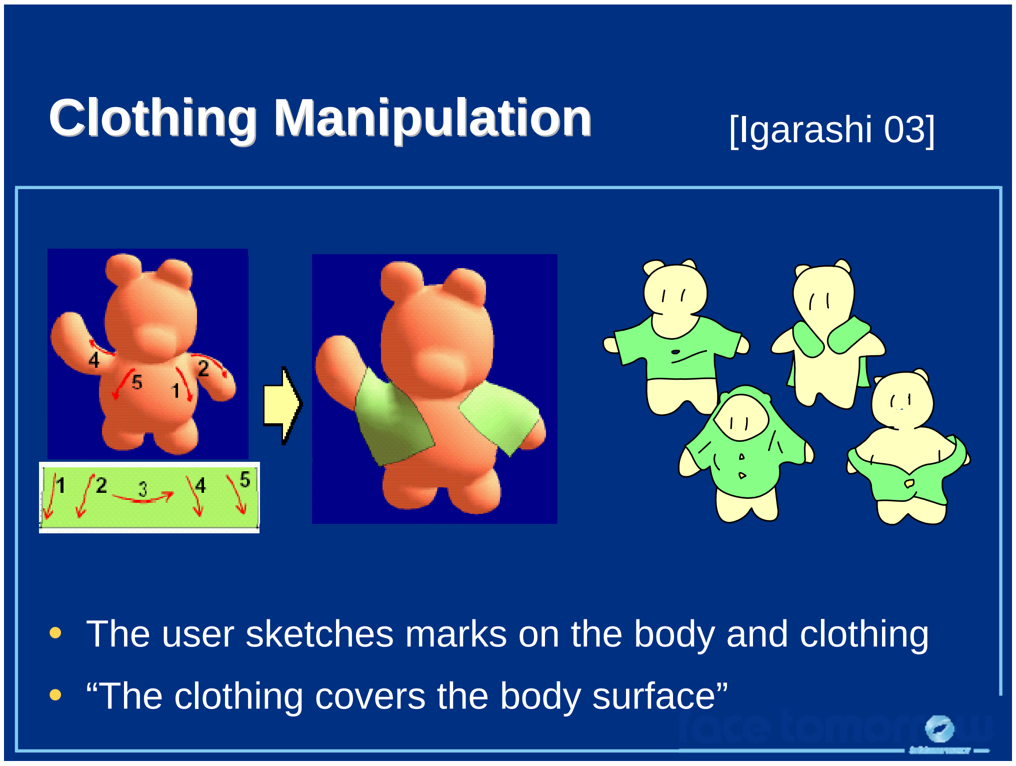
Garment Design

[Turquin 04]



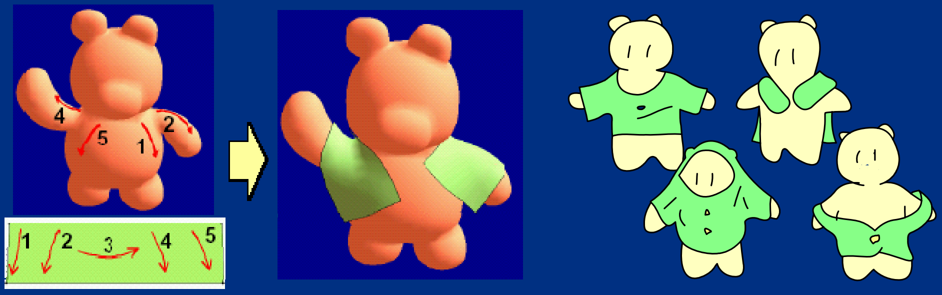
- The user sketches the outline of a garment
- “The garment covers the body surface”

The system shown in the slide is a sketch-based approach for garment design. The user draws a 2D sketch of the desired garment around the given 2D human body. The system then constructs a 3D garment so that the garment surrounds the 3D body so that the silhouette of the 3D garment matches the original 2D sketch. Using the system, a designer can see various garments directly on the character body.



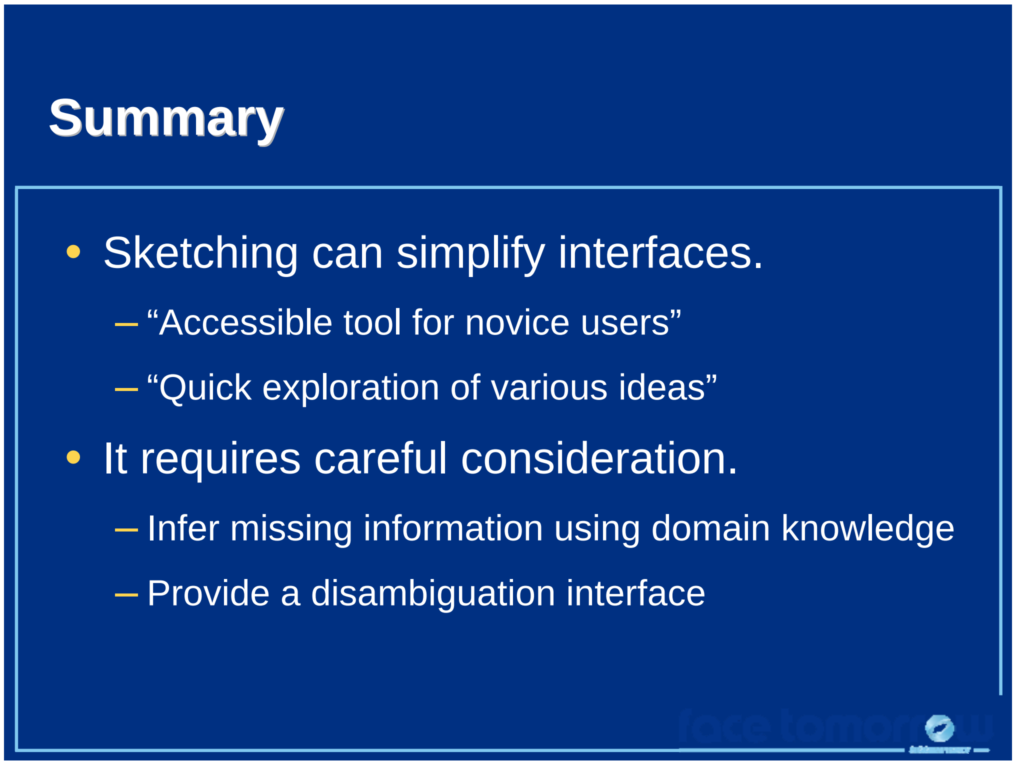
Clothing Manipulation

[Igarashi 03]



- The user sketches marks on the body and clothing
- “The clothing covers the body surface”

The system shown in the slide provides an easy-to-use interface for putting clothing on a character and manipulating it on the character. To put clothing represented as a 2D pattern on a 3D character, the user draws marks on both the 2D pattern and the 3D character. The system then places the clothing on the character so that corresponding marks match. After putting the clothing on the character, the user can drag the clothing on the character’s body surface to experiment with many different styles.



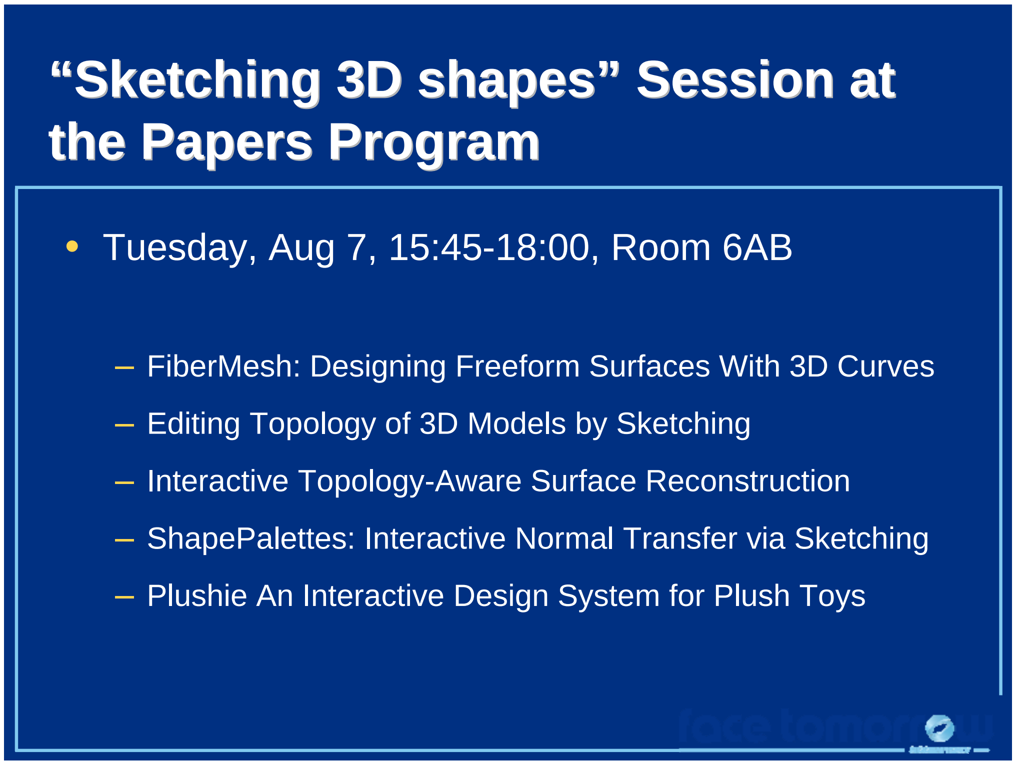
Summary

- Sketching can simplify interfaces.
  - “Accessible tool for novice users”
  - “Quick exploration of various ideas”
- It requires careful consideration.
  - Infer missing information using domain knowledge
  - Provide a disambiguation interface

As shown in examples from this lecture, sketching can simplify the interfaces for graphics applications. It makes graphics tools accessible for novice users and also enables expert users to use advanced computational tools in the initial design phase.

To design successful sketching interfaces, it is very important to understand the incomplete and ambiguous nature of sketching. First, sketching contains very limited information and so the system needs to infer implicit information using domain knowledge. To do so, it is crucial to carefully limit the target domain. For example, the SKETCH system assumes that a scene consists of stacked primitives and the Teddy system assumes that everything is rotund. Second, sketching is very ambiguous and it is important to provide a means for disambiguation. One way to deal with ambiguity is to generate multiple candidates and ask the user to choose one, as seen in Pegasus and Chateau.

I hope you to try sketch-based interfaces in your work to give a different feel to your tool. It is a lot of fun.



“Sketching 3D shapes” Session at the Papers Program

- Tuesday, Aug 7, 15:45-18:00, Room 6AB
  - FiberMesh: Designing Freeform Surfaces With 3D Curves
  - Editing Topology of 3D Models by Sketching
  - Interactive Topology-Aware Surface Reconstruction
  - ShapePalettes: Interactive Normal Transfer via Sketching
  - Plushie An Interactive Design System for Plush Toys

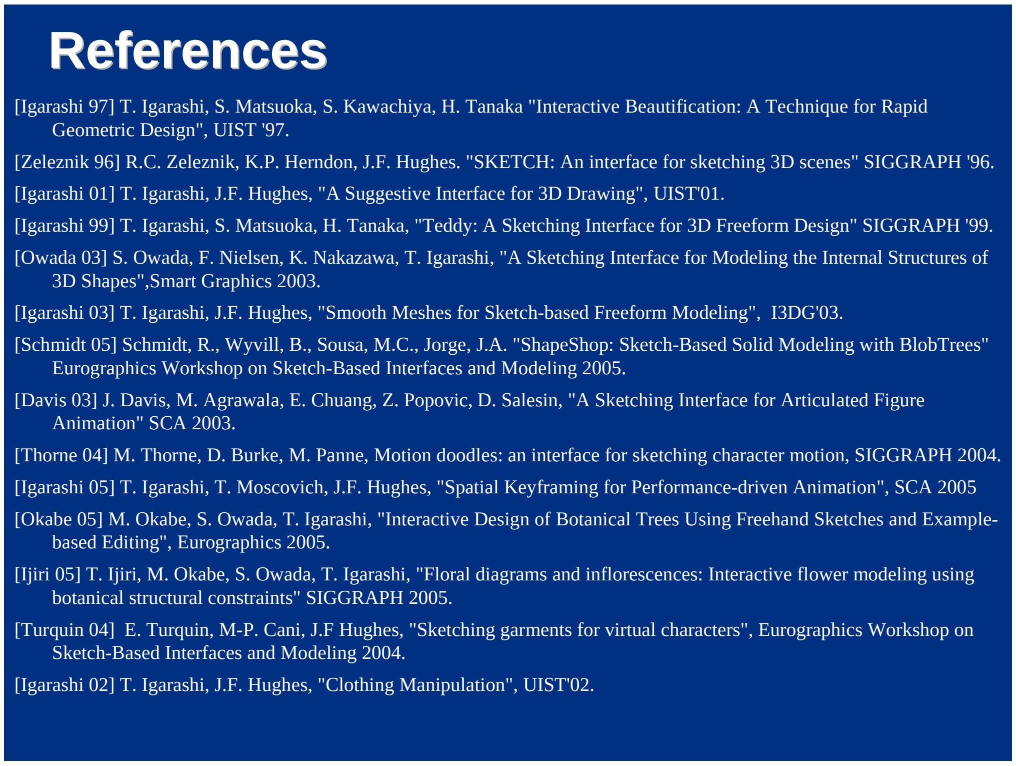
Here is a little bit of an advertisement.

This year, the papers program features a “Sketching 3D Shapes” session!

It will be in room 6AB on Tuesday evening.

It shows the latest work in sketching 3D shapes, including 3D modeling, topology editing, normal map design, and plush toy design.

Please come and see them!



References

[Igarashi 97] T. Igarashi, S. Matsuoka, S. Kawachiya, H. Tanaka "Interactive Beautification: A Technique for Rapid Geometric Design", UIST '97.

[Zeleznik 96] R.C. Zeleznik, K.P. Herndon, J.F. Hughes. "SKETCH: An interface for sketching 3D scenes" SIGGRAPH '96.

[Igarashi 01] T. Igarashi, J.F. Hughes, "A Suggestive Interface for 3D Drawing", UIST'01.

[Igarashi 99] T. Igarashi, S. Matsuoka, H. Tanaka, "Teddy: A Sketching Interface for 3D Freeform Design" SIGGRAPH '99.

[Owada 03] S. Owada, F. Nielsen, K. Nakazawa, T. Igarashi, "A Sketching Interface for Modeling the Internal Structures of 3D Shapes",Smart Graphics 2003.

[Igarashi 03] T. Igarashi, J.F. Hughes, "Smooth Meshes for Sketch-based Freeform Modeling", I3DG'03.

[Schmidt 05] Schmidt, R., Wyvill, B., Sousa, M.C., Jorge, J.A. "ShapeShop: Sketch-Based Solid Modeling with BlobTrees" Eurographics Workshop on Sketch-Based Interfaces and Modeling 2005.

[Davis 03] J. Davis, M. Agrawala, E. Chuang, Z. Popovic, D. Salesin, "A Sketching Interface for Articulated Figure Animation" SCA 2003.

[Thorne 04] M. Thorne, D. Burke, M. Panne, Motion doodles: an interface for sketching character motion, SIGGRAPH 2004.

[Igarashi 05] T. Igarashi, T. Moscovich, J.F. Hughes, "Spatial Keyframing for Performance-driven Animation", SCA 2005

[Okabe 05] M. Okabe, S. Owada, T. Igarashi, "Interactive Design of Botanical Trees Using Freehand Sketches and Example-based Editing", Eurographics 2005.

[Ijiri 05] T. Ijiri, M. Okabe, S. Owada, T. Igarashi, "Floral diagrams and inflorescences: Interactive flower modeling using botanical structural constraints" SIGGRAPH 2005.

[Turquin 04] E. Turquin, M-P. Cani, J.F Hughes, "Sketching garments for virtual characters", Eurographics Workshop on Sketch-Based Interfaces and Modeling 2004.

[Igarashi 02] T. Igarashi, J.F. Hughes, "Clothing Manipulation", UIST'02.

Sketch-Based Interfaces: Techniques and Applications

Sketching for Mechanical Design and CAD

SIGGRAPH2007

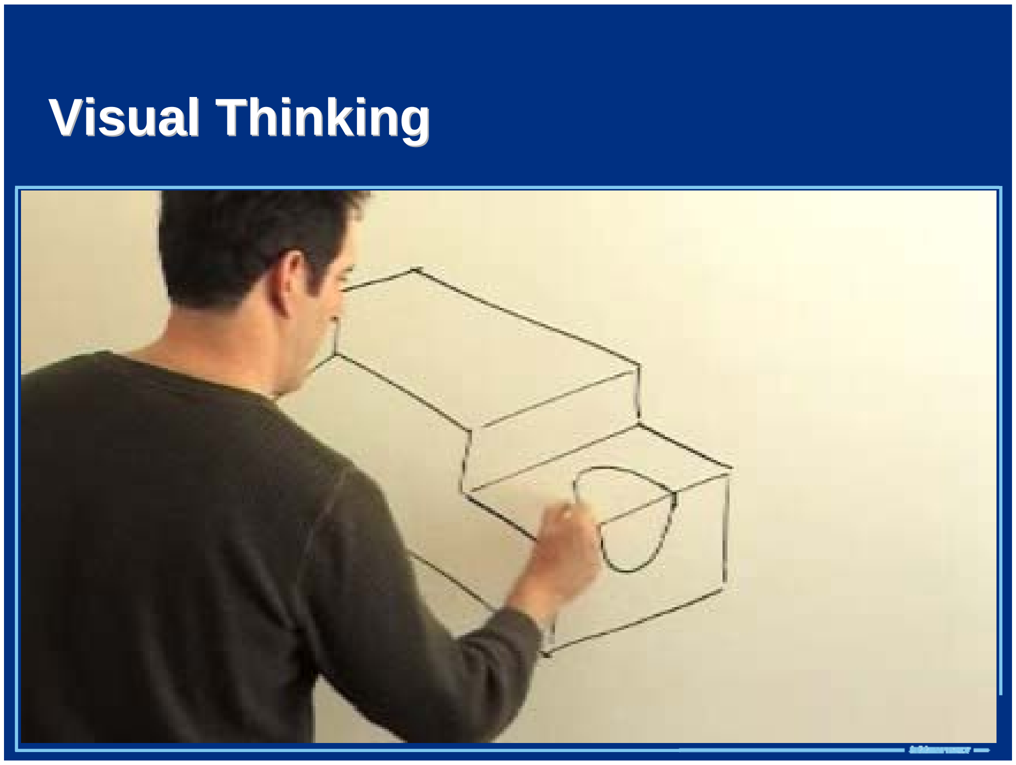
Hod Lipson

Cornell University

Sketching for Mechanical Design and CAD (30 min)

Hod Lipson, Assistant Professor
Department of Mechanical & Aerospace Engineering
216 Upson Hall, Cornell University
Ithaca NY 14853, USA

Email: hod.lipson@cornell.edu
http://www.mae.cornell.edu/lipson

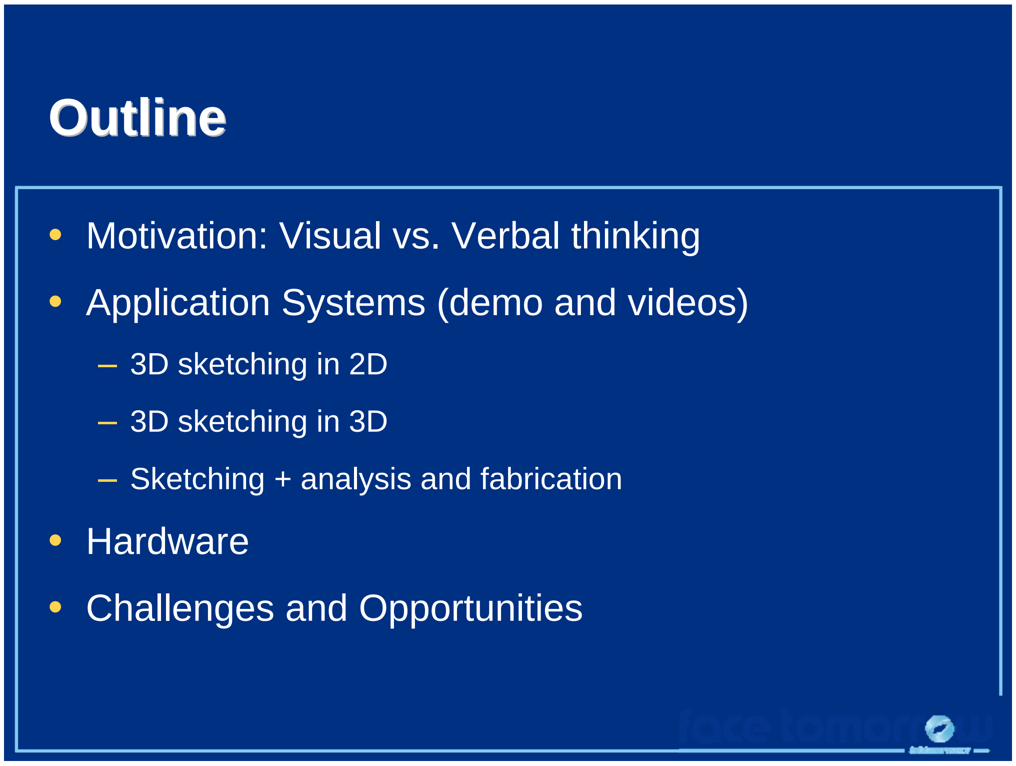


Visual Thinking

“It is fascinating to watch how a designer, when given a design problem, instinctively reaches for a pencil and paper…”

So wrote Eugene Ferguson, a science historian, in his book “The Mind’s Eye”

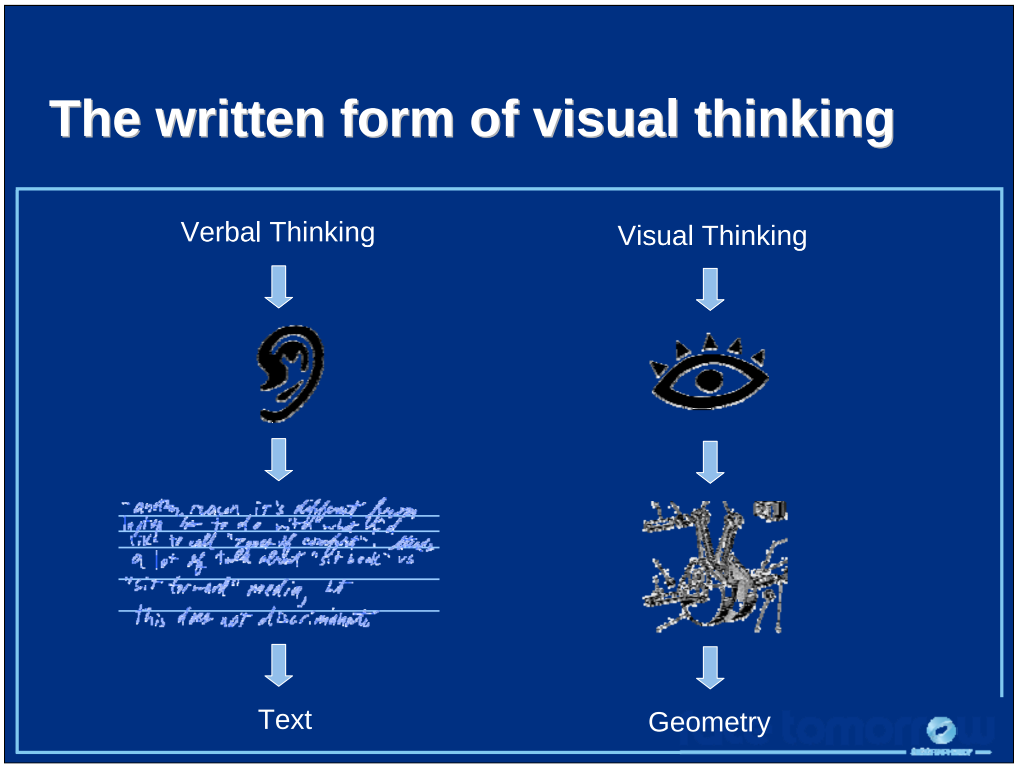
Visual thinking is an integral part of any design process, and sketching is the informal hand written form of this thinking process.



Outline

- Motivation: Visual vs. Verbal thinking
- Application Systems (demo and videos)
  - 3D sketching in 2D
  - 3D sketching in 3D
  - Sketching + analysis and fabrication
- Hardware
- Challenges and Opportunities

In this lecture, we will introduce various sketching interfaces for computer graphics applications. We first give a short introduction and then discuss several sketch-based systems.



The written form of visual thinking

Verbal thinking manifests itself in verbal language and speech, with handwriting being the informal written form of this language.

In contrast, visual thinking – the other half of the brain – has its own way of representing information, and sketching is the informal written form of this language.

While extensive effort has been spent on trying to get machines to understand speech and handwriting, relatively little has been spent trying to get machines to understand visual communication.

This is our goal here today.

The importance of visual thinking is demonstrated well by a piece written by Richard Feynman's memoir:

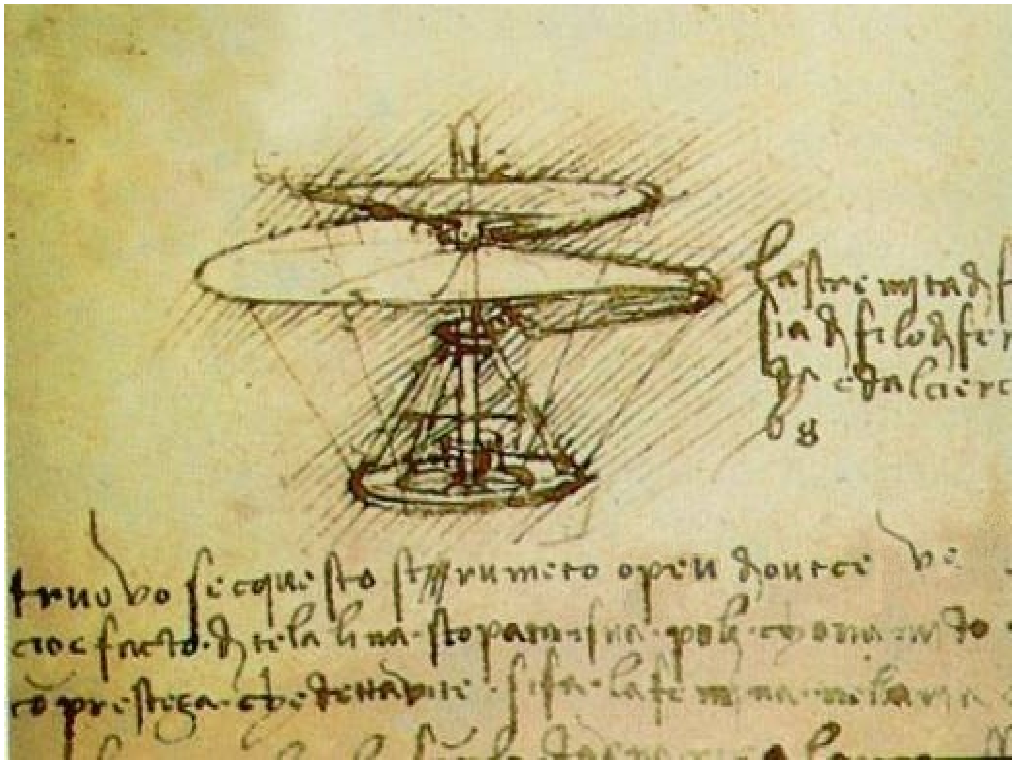
One time, we were discussing something – we must have been eleven or twelve at the time – and I said, "But thinking is nothing but talking to yourself."

– "Oh, yeah?", Bennie said, "Do you know the crazy shape of the crankshaft in a car?"

– "Yeah, what of it?"

– "Good. Now tell me: how did you describe it when you were talking to yourself?"

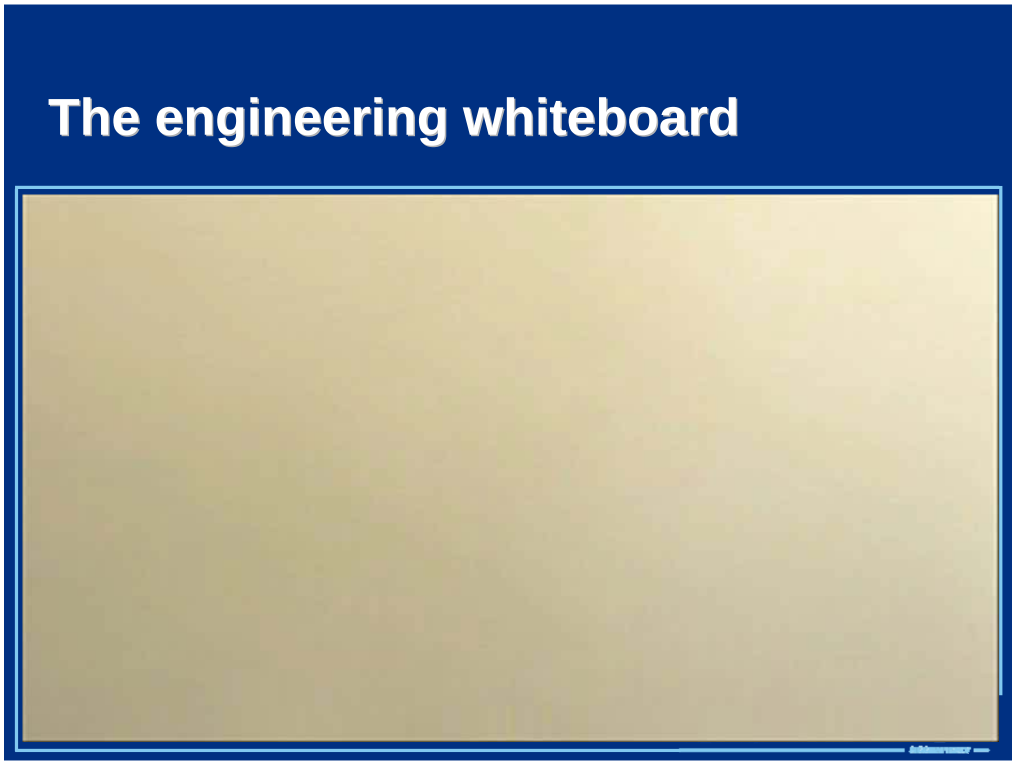
(Feynman, 1988)



While handwriting has many forms, sketches transcend space and time. You can understand sketches made long ago by people from other cultures and who speak different languages.

While you probably have no clue as to what Leonardo Da Vinci wrote here, you can certainly understand the concept he drew out.

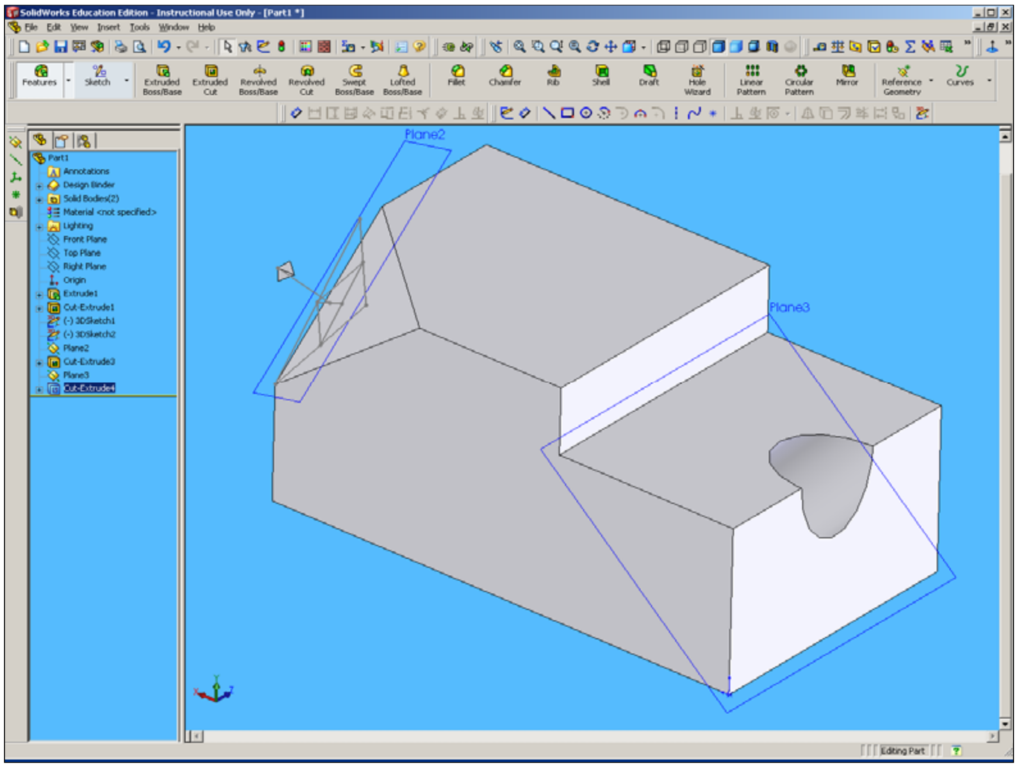
We'd like a computer to be able to understand us in the same way.



The engineering whiteboard

Lets say I wanted to describe a particular shape –

A rectangle, with a notch, a chamfered corner, and a diagonal hole.

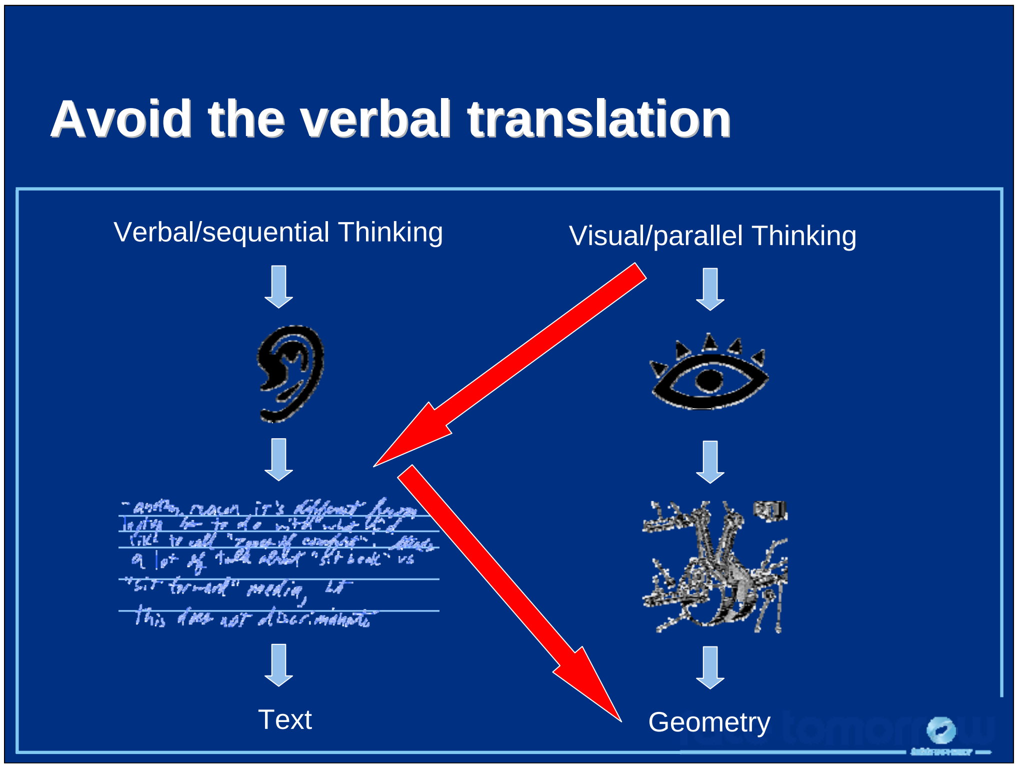


I could do it in CAD, but it would take me some time. Why?

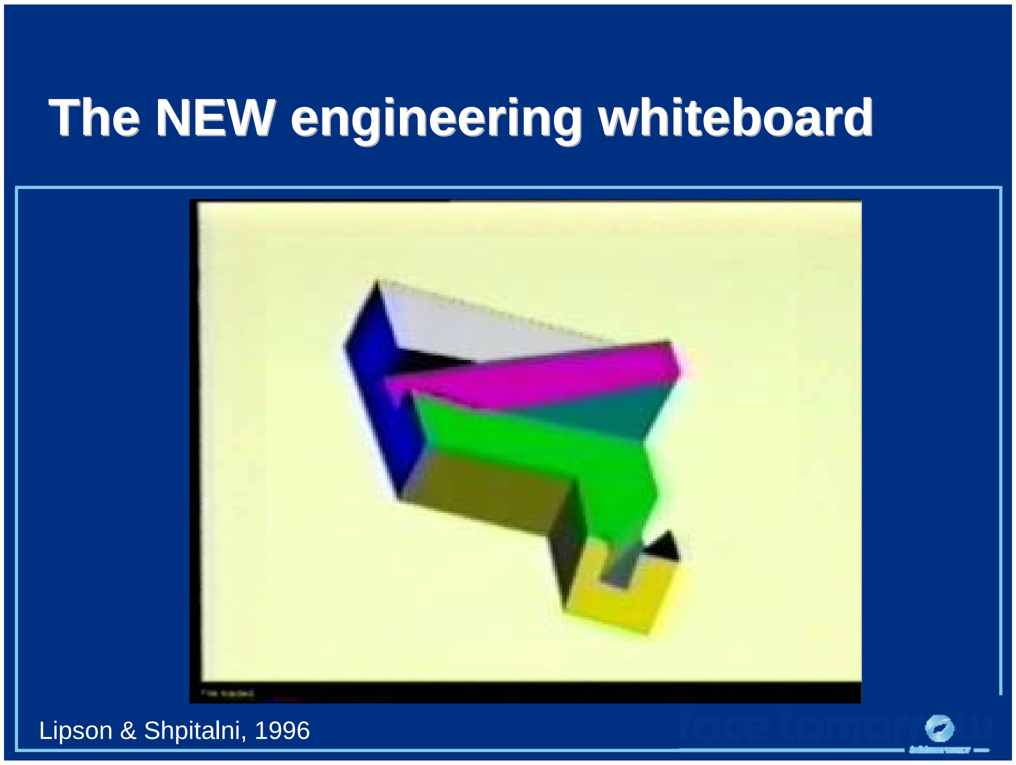
Because I'd have to familiarize myself with the interface.

I'd have to provide accurate information, even though its not important.

But most importantly, it will force me to have broken my visual thinking into a sequential, verbal thinking – into a set of commands.


Avoid the verbal translation

Sketch understanding is really about skipping the forced translation from the parallel, visual thinking mode which characterizes design, into the verbal, sequential process that is typically needed to enter it into a computer.

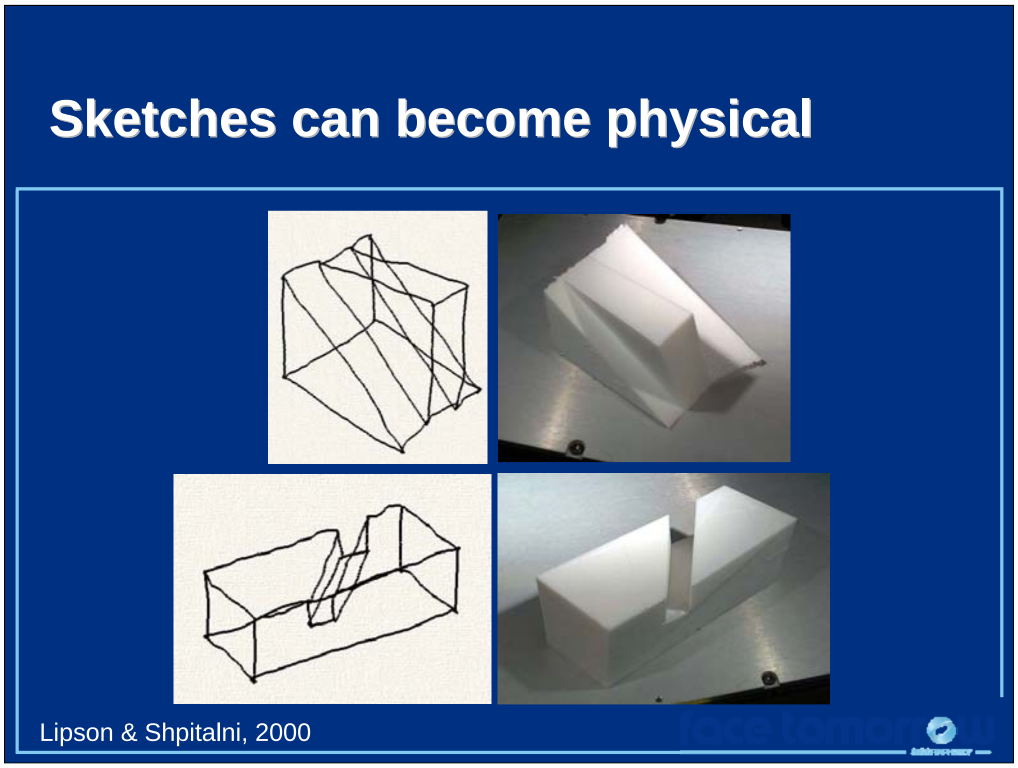


The NEW engineering whiteboard

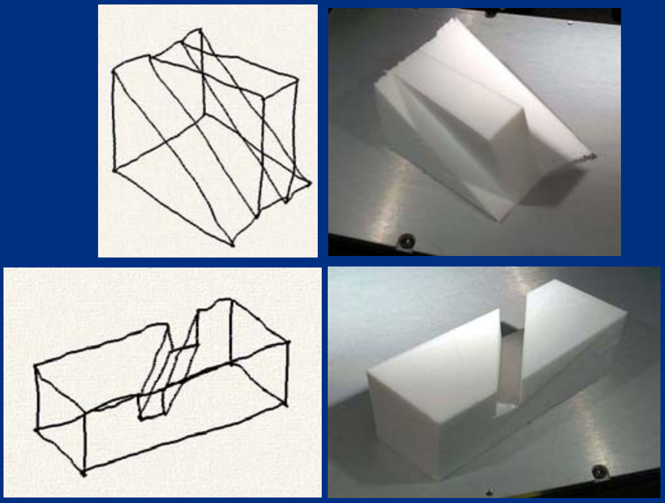
Lipson & Shpitalni, 1996

A better way to describe the shape would combine the freehand sketching, with a 3D interpreter that would transform the 2D lines into a 3D shape.

The shape is not necessarily very accurate, but it conveys the concept. It is sufficient to allow for discussing the concept.

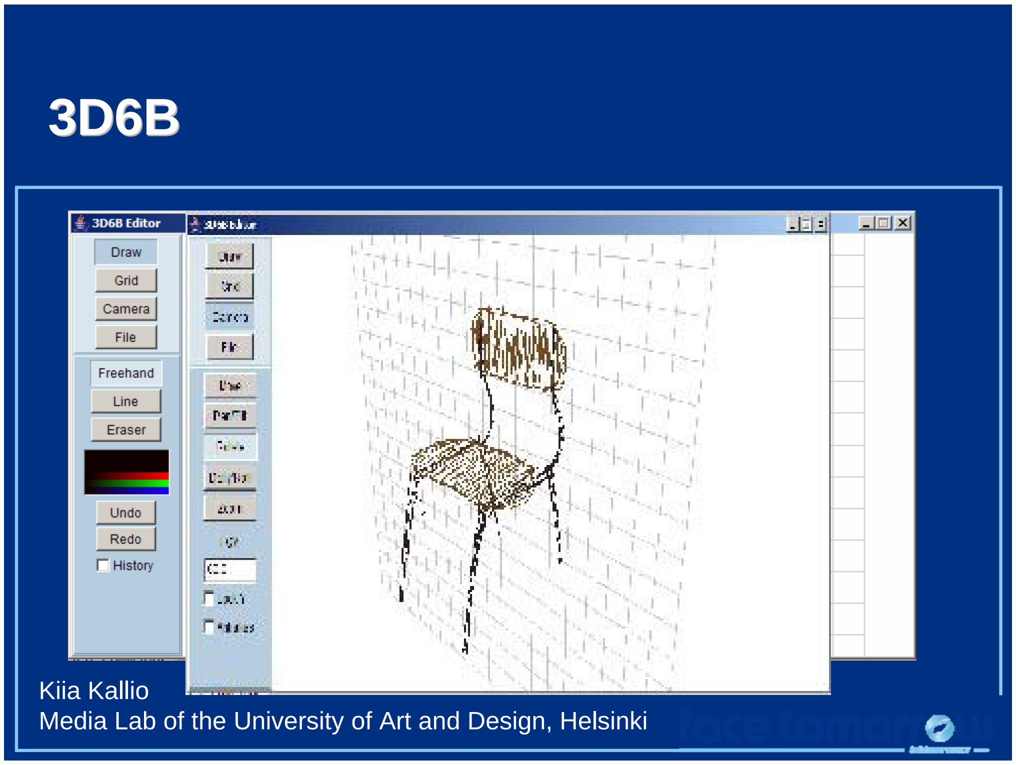


Sketches can become physical



Lipson & Shpitalni, 2000

Rapid prototyping technology provides means for physically realizing the sketch mode – again, bypassing a sequential construction sequence.



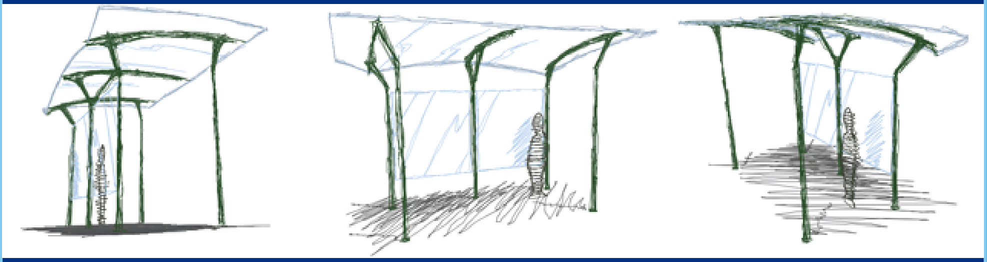
3D6B

Kiia Kallio
Media Lab of the University of Art and Design, Helsinki

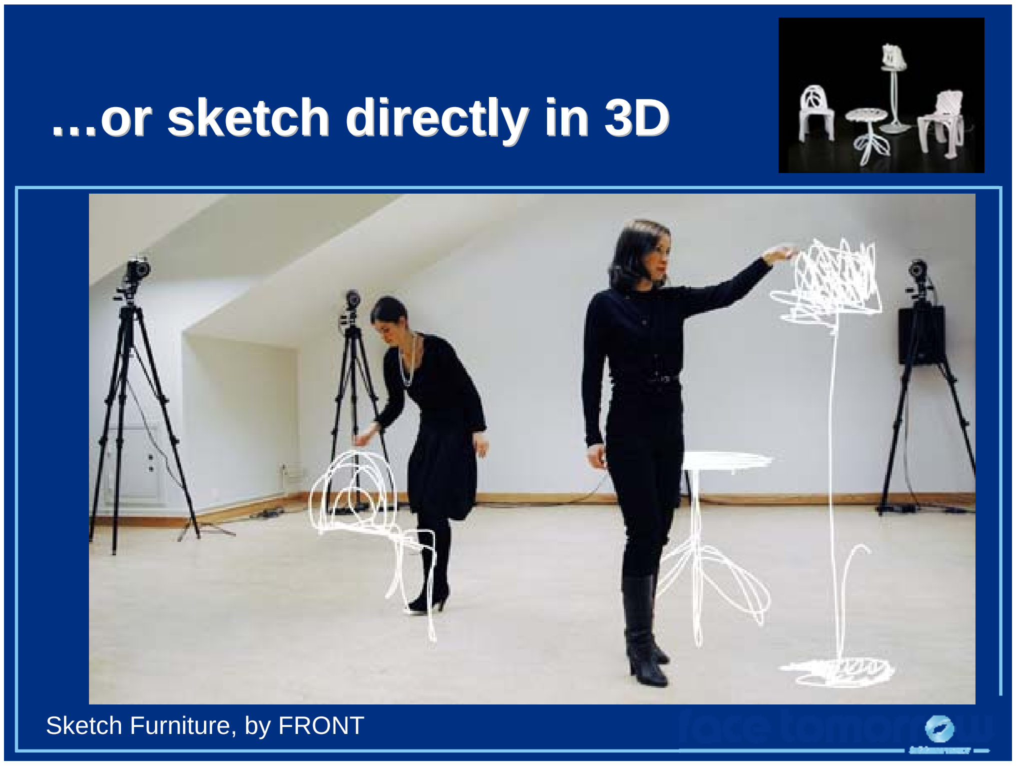
But why try to interpret the sketch?

Perhaps a better way is just to keep the stroked in 3D, without trying to make "geometric sense" out of them

3D6B



Kiia Kallio
Media Lab of the University of Art and Design, Helsinki



…or sketch directly in 3D

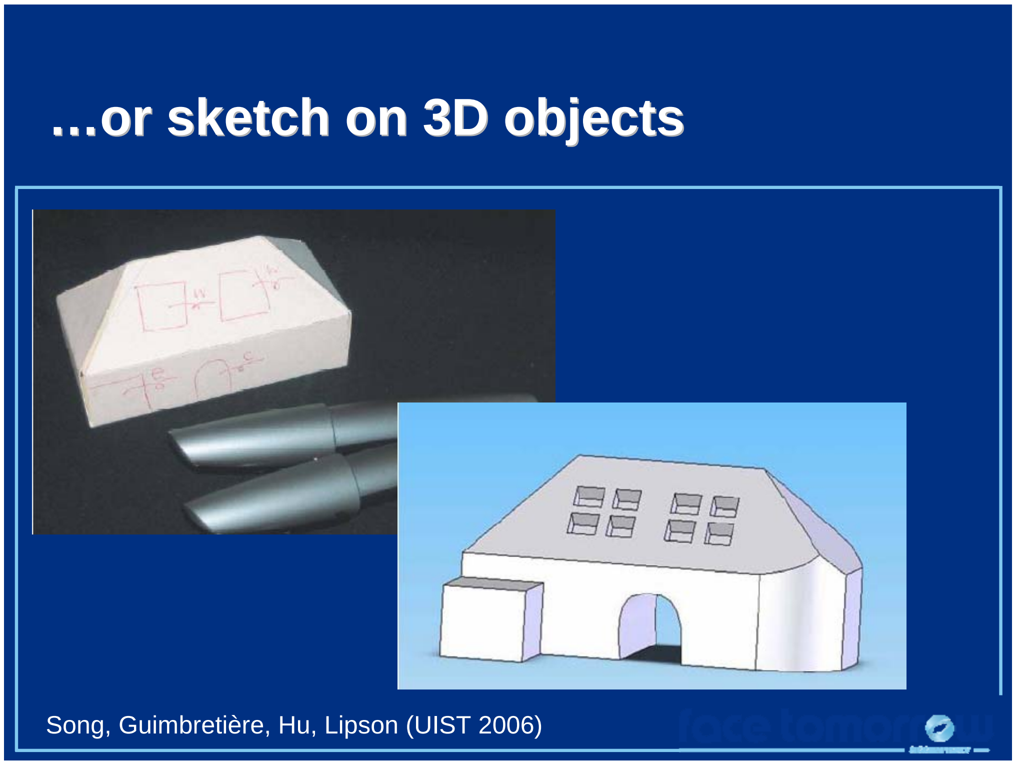
Sketch Furniture, by FRONT

Ultimately, we can think of sketching directly in 3D.

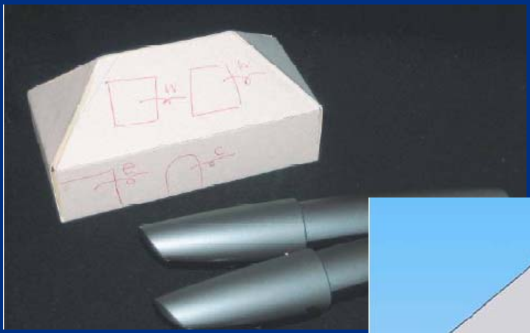
Note how the creativity of the designers here is completely unhampered by menus and commands – the ideas just flow into space.

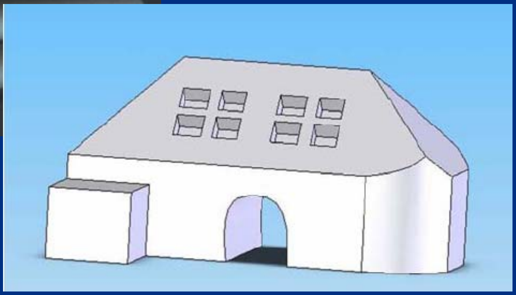
If course there is some acting in this movie – the designers can't actually see what they are creating (which is a serious problem!), but one can see that should some technological hurdles be solved (e.g. using augmented reality). Interesting creative freedom can be unleashed.

Again, notice how rapid prototyping technology provides means for physically realizing the sketch model.



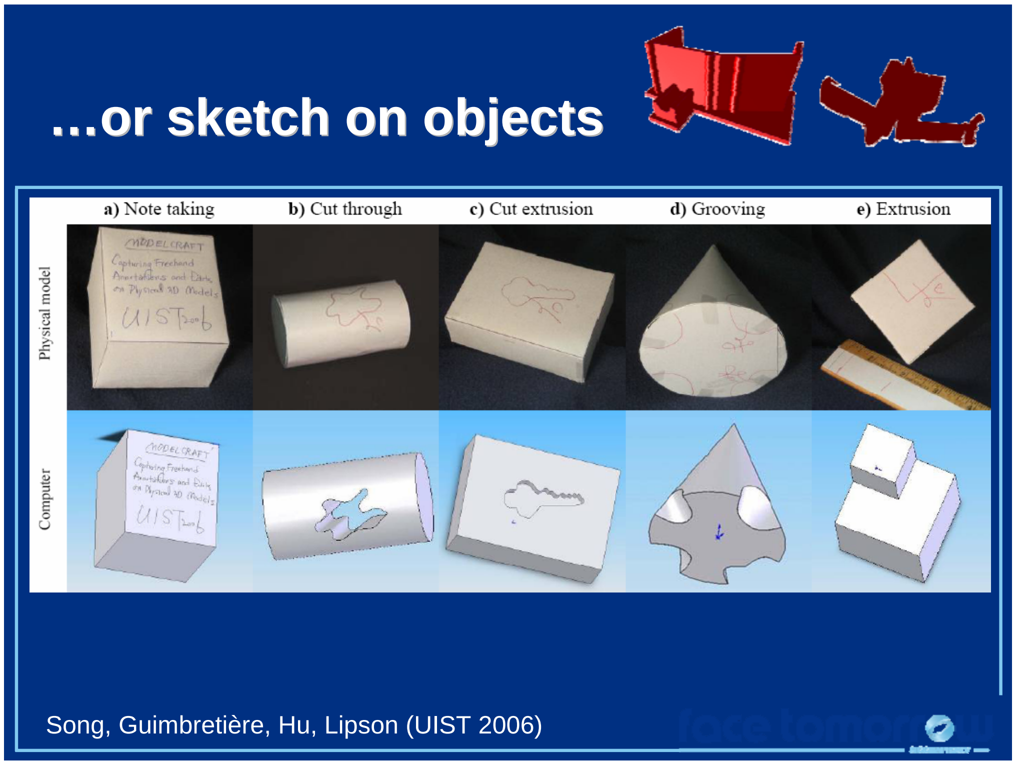
…or sketch on 3D objects



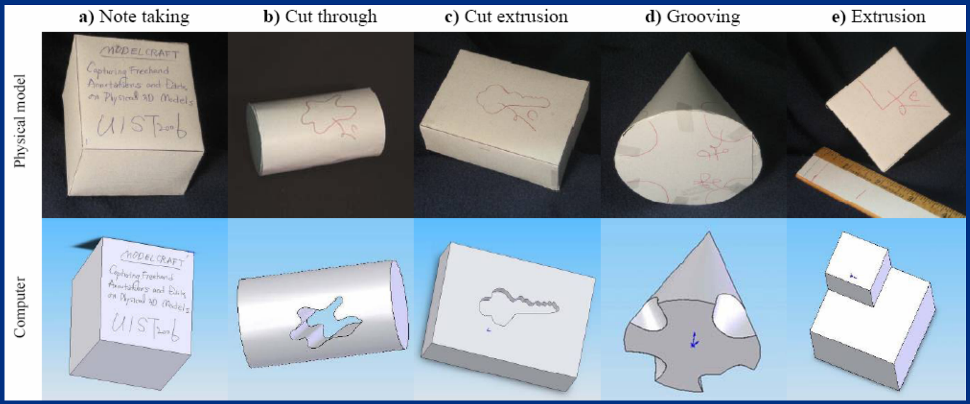


Song, Guimbretière, Hu, Lipson (UIST 2006)

We can even sketch onto 3D models.

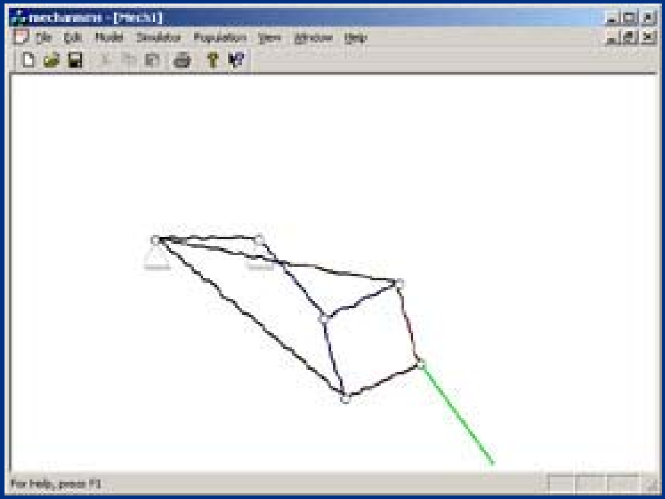


…or sketch on objects



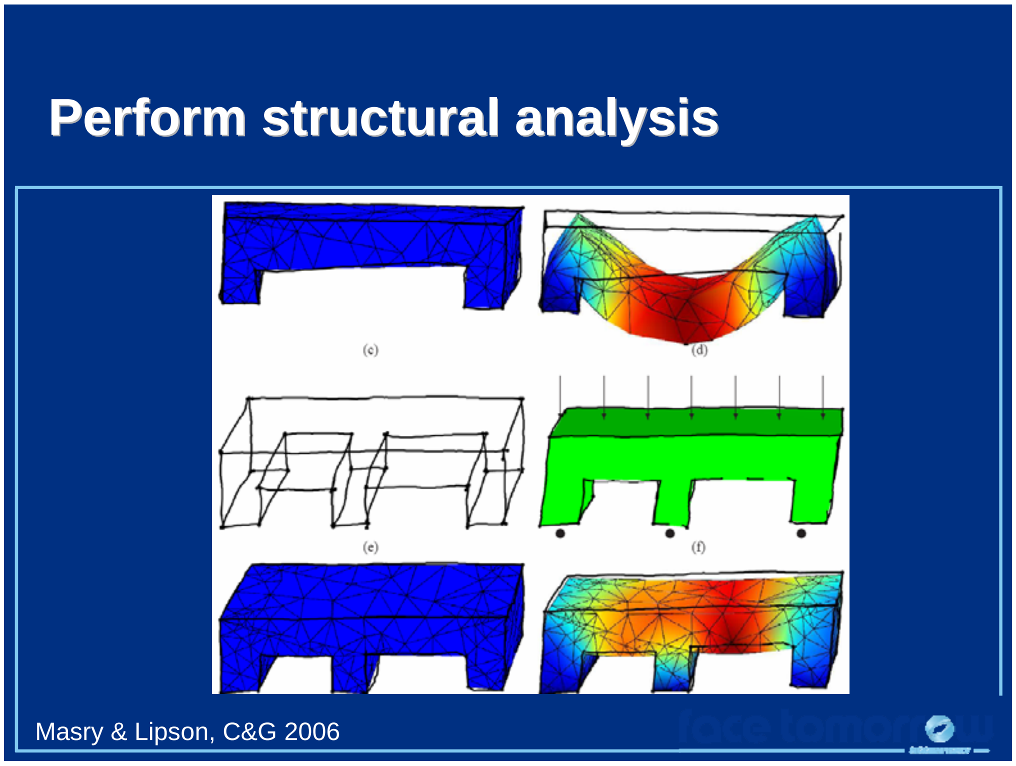
Song, Guimbretière, Hu, Lipson (UIST 2006)

Perform kinematic analysis

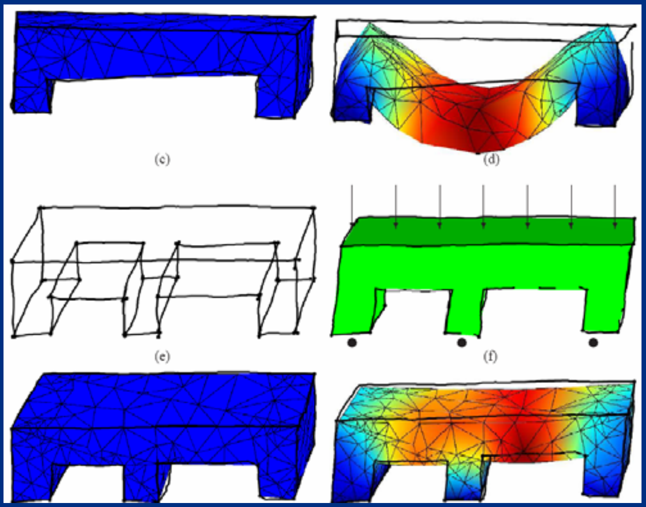


Sketch kinematics, Lipson 2005

The sketches do not need to be static. They can be live sketches – which can be analyzed in various ways.

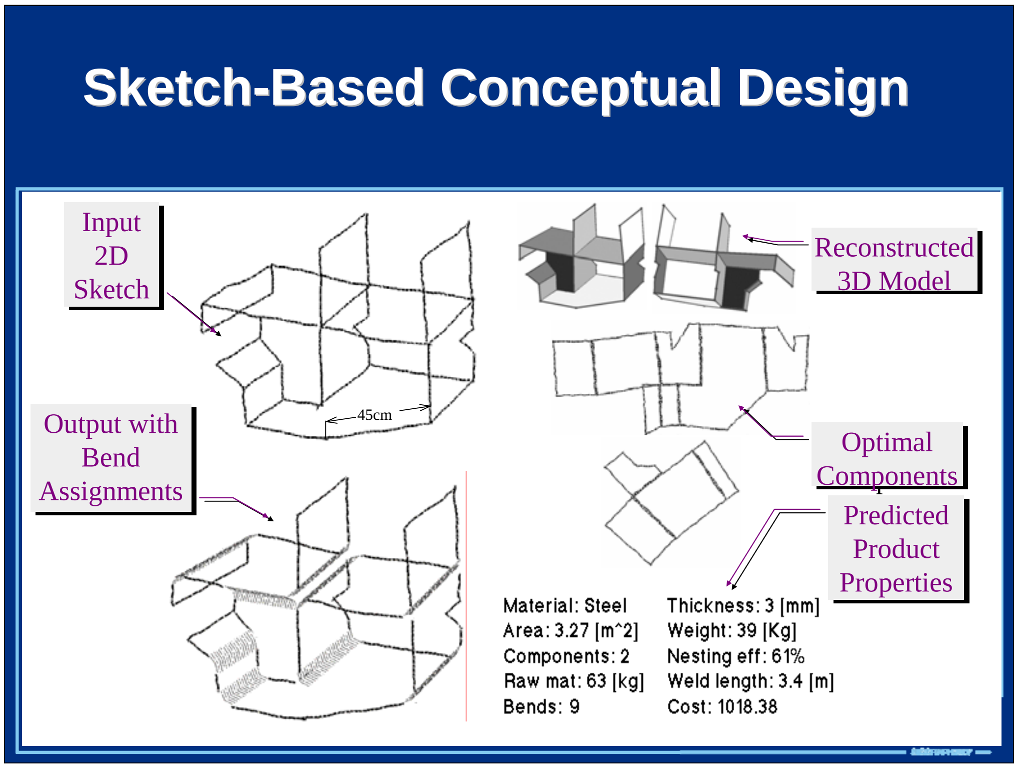


Perform structural analysis



Masry & Lipson, C&G 2006

The sketches do not need to be static. They can be live sketches – which can be analyzed in various ways.



Sketch-Based Conceptual Design

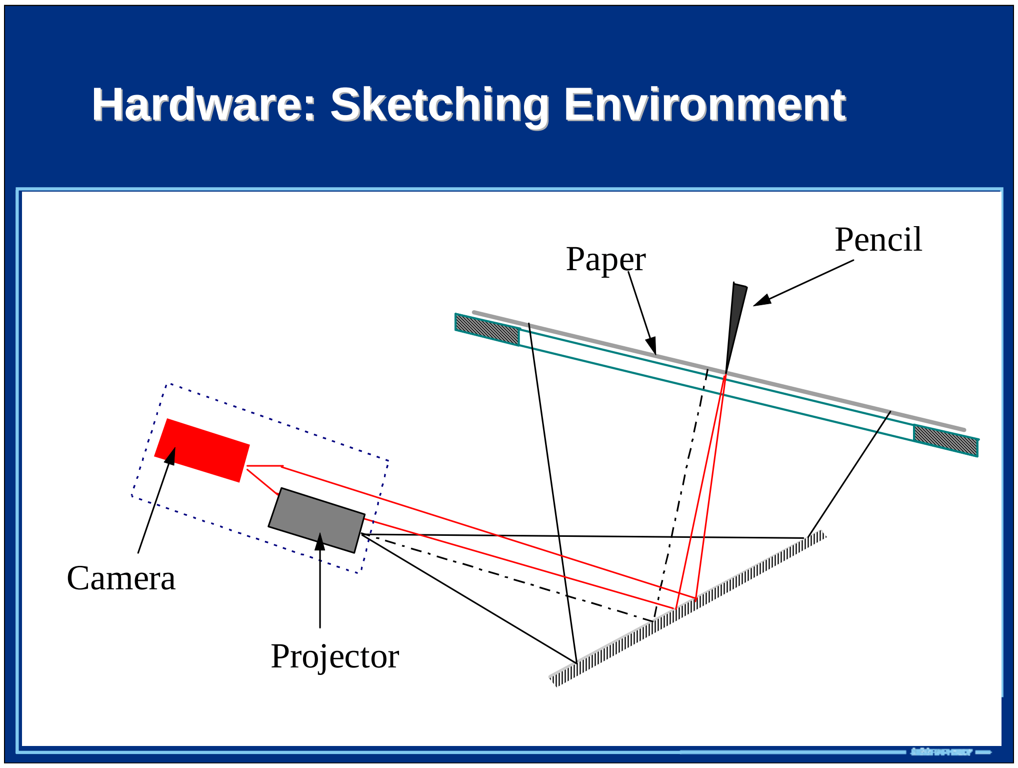
Material: Steel	Thickness: 3 [mm]
Area: 3.27 [m^2]	Weight: 39 [Kg]
Components: 2	Nesting eff: 61%
Raw mat: 63 [kg]	Weld length: 3.4 [m]
Bends: 9	Cost: 1018.38

Combining sketching and analysis, we can close the loop between design and analysis, bringing CAE tool much sooner into the design cycle.

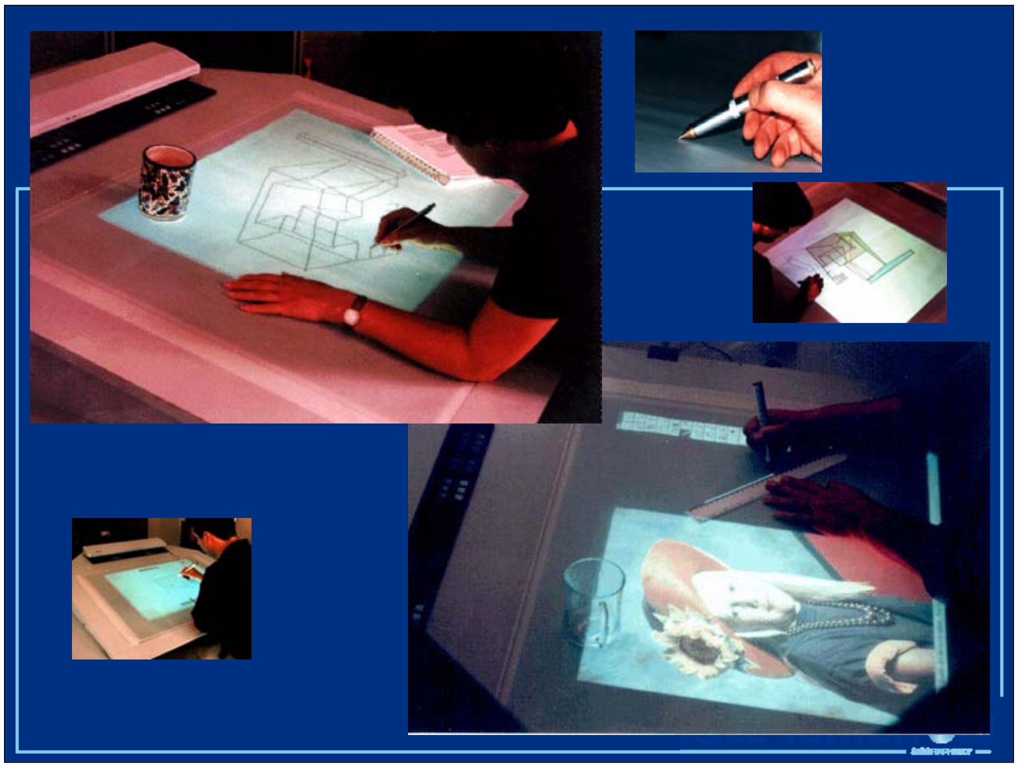
Hardware: Sketching Environment

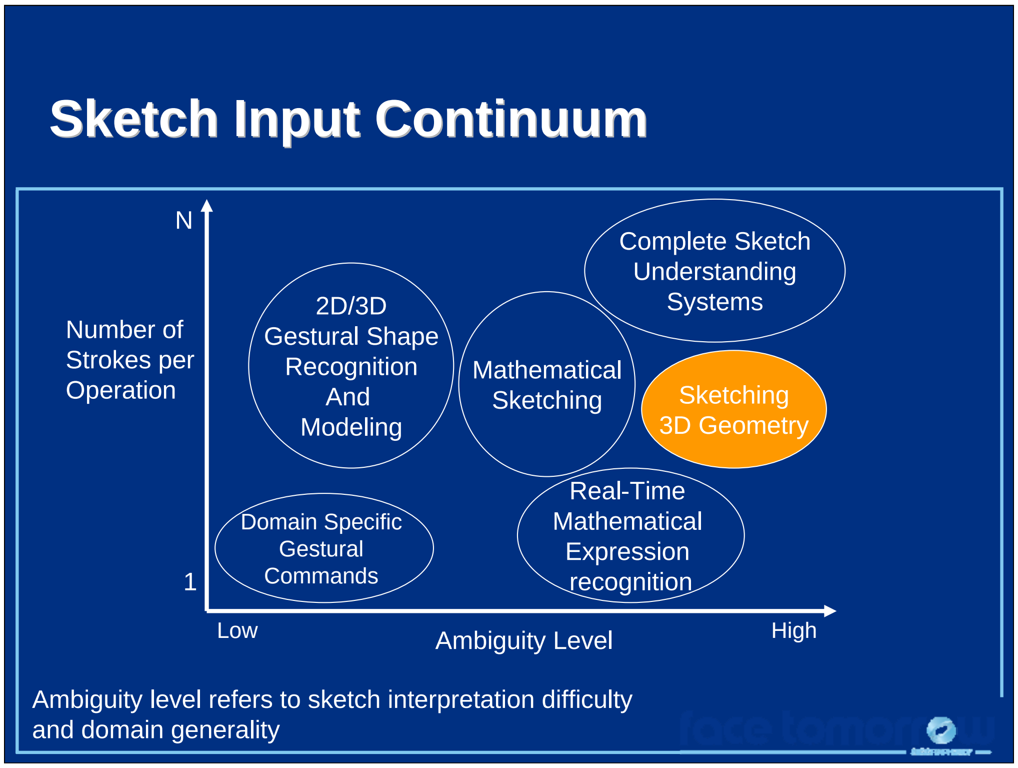
Hardware platforms have evolved, but many challenges remain:

Moving from light-pens to real natural paper texture; real-time sketching, and 3D?



Hardware: Sketching Environment

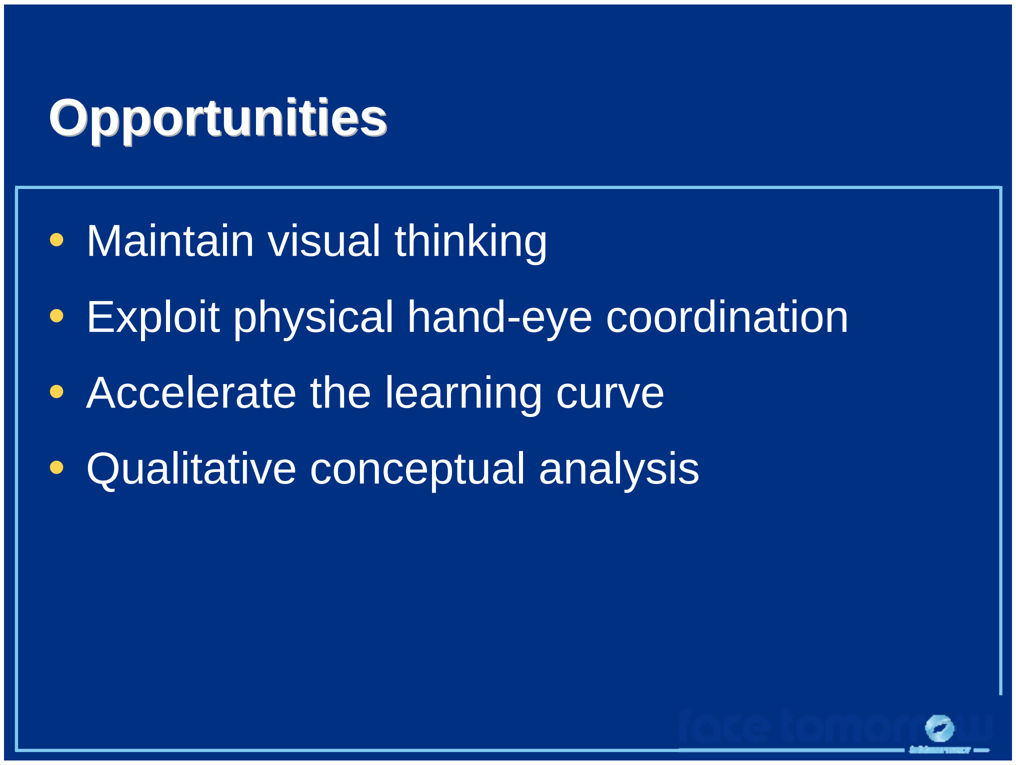




Sketch Input Continuum

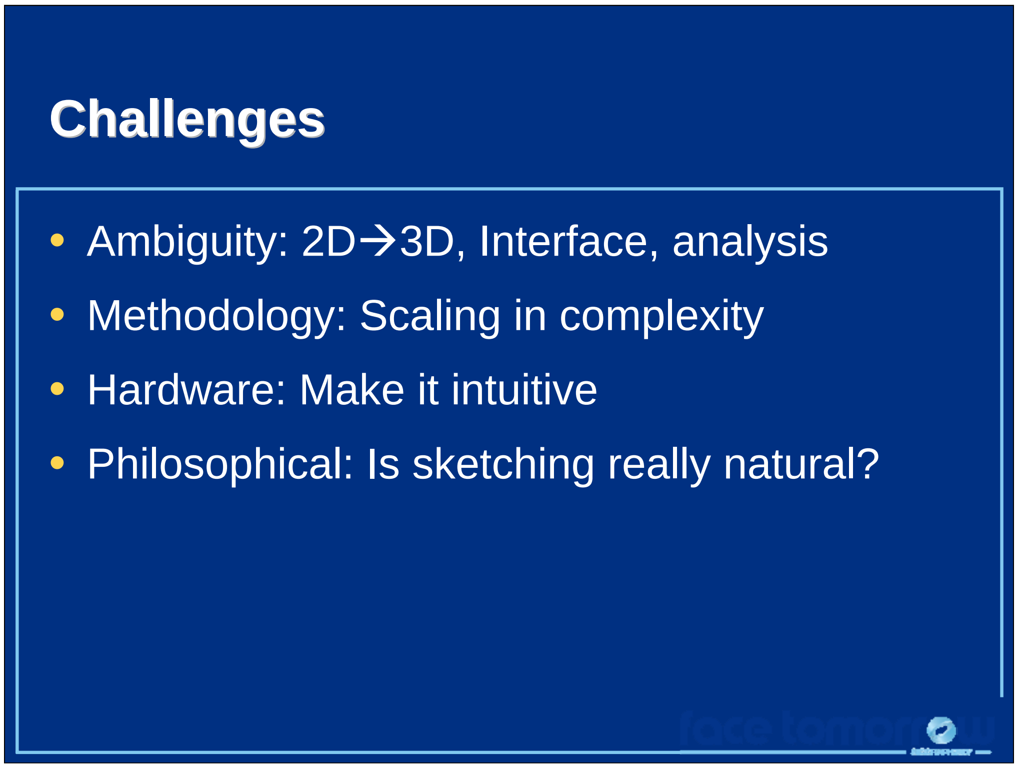
N
Number of Strokes per Operation
1
Low
Ambiguity Level
High
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Complete Sketch Understanding Systems
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition

Ambiguity level refers to sketch interpretation difficulty and domain generality



Opportunities

- Maintain visual thinking
- Exploit physical hand-eye coordination
- Accelerate the learning curve
- Qualitative conceptual analysis



Challenges

- Ambiguity: 2D→3D, Interface, analysis
- Methodology: Scaling in complexity
- Hardware: Make it intuitive
- Philosophical: Is sketching really natural?

Sketch-Based Interfaces: Techniques and Applications

Sketching and Education

SIGGRAPH2007

Joseph J. LaViola Jr.

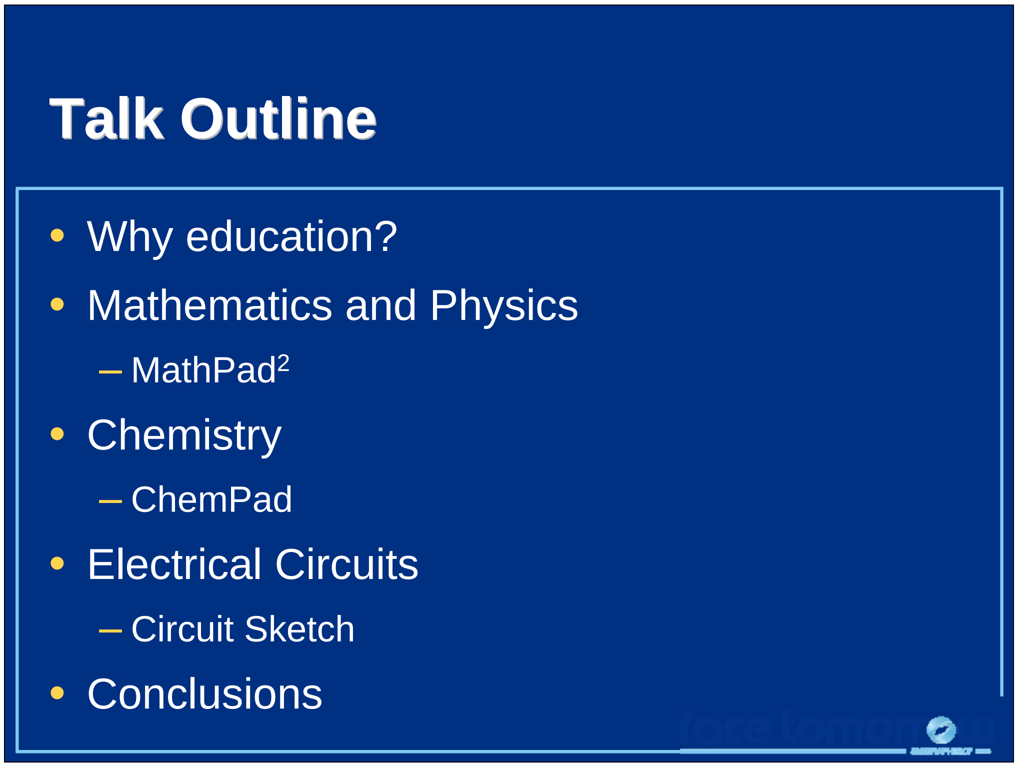
School of EECS

University of Central Florida

Sketching and Education

Joseph J. LaViola Jr.
Assistant Professor
University of Central Florida
School of EECS
4000 Central Florida Blvd.
Orlando, FL 32816

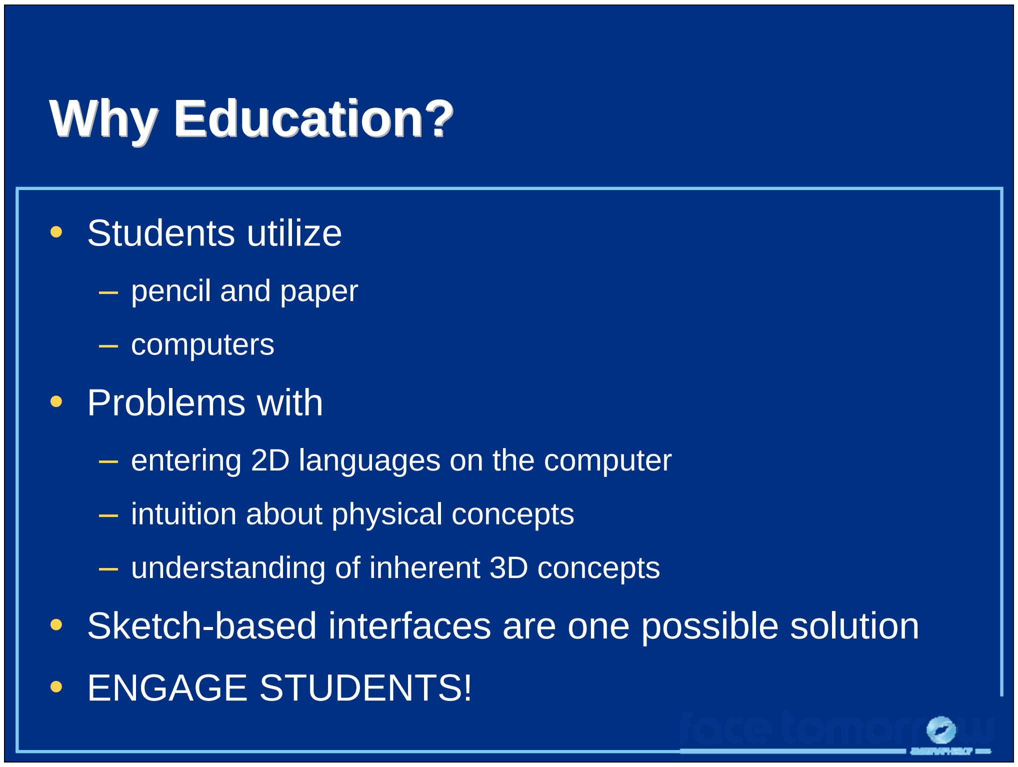
Email: jjl@cs.ucf.edu
http://www.cs.ucf.edu/~jjl/



Talk Outline

- Why education?
- Mathematics and Physics
  - MathPad^2
- Chemistry
  - ChemPad
- Electrical Circuits
  - Circuit Sketch
- Conclusions

In this lecture, we will discuss how sketch-based interfaces and sketch-based applications can be used in an educational setting. First, we will discuss why sketch-based interfaces are a good fit for several different application areas in education such as mathematics and physics, chemistry, and electrical circuit analysis. Second, we will discuss three prototype applications, MathPad^2, ChemPad, and Circuit Sketch, that are targeted toward helping teachers explain science and engineering concepts, and assisting students with learning these concepts. Finally, we will provide some general guidelines for when sketch-based interfaces should be considered when building educational applications and learning tools.

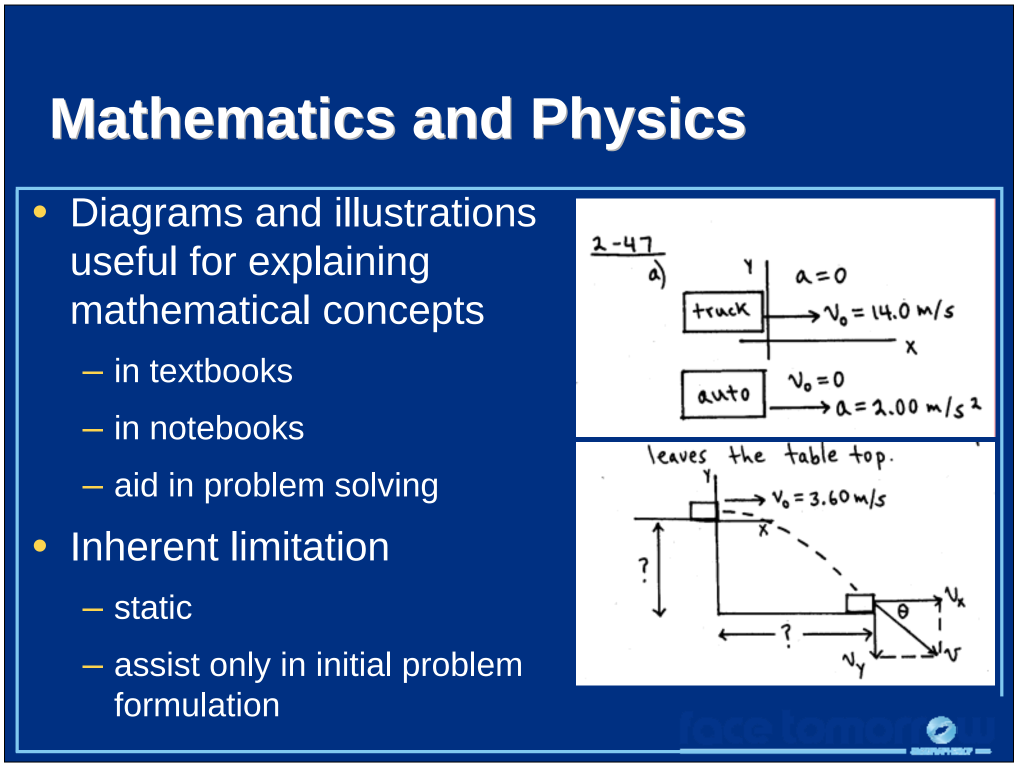


Why Education?

- Students utilize
  - pencil and paper
  - computers
- Problems with
  - entering 2D languages on the computer
  - intuition about physical concepts
  - understanding of inherent 3D concepts
- Sketch-based interfaces are one possible solution
- ENGAGE STUDENTS!

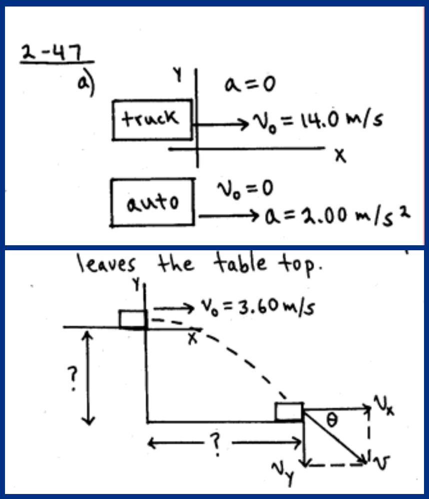
In learning environments, students typically used pencil and paper to take notes and do homework problems. They also use computers to write papers, explore the web for research, and use a variety of educational software tools to assist in their instruction and learning. However, there is a disconnect between these two mediums. There is the notion of pencil and paper being a static medium but providing an easy to use tool for expressing ideas. In contrast, the computer provides a dynamic medium for student learning but makes expressing ideas less fluid and expressive; constrained by an application input model usually resorting to a keyboard and mouse. Thus, it makes sense to combine the best qualities from these two mediums; the power of the computer with the expressive power of pencil and paper.

More specifically, there are certain disciplines that require 2D notations for representing concepts in areas a such mathematics, chemical bonds, and circuit diagrams. Entering these notations into a computer is much easier if users can simply write then down as if using pencil and paper. After these notations are written with traditional the traditional pencil-and-paper medium, they are static and only assist in initial concept and problem formulations. Thus, going from static representations to dynamic visualizations is another reason to go utilize computing power in conjunction with a natural pencil -and-paper style interface. Sketch-based interfaces are one possible approach to dealing with these issues.

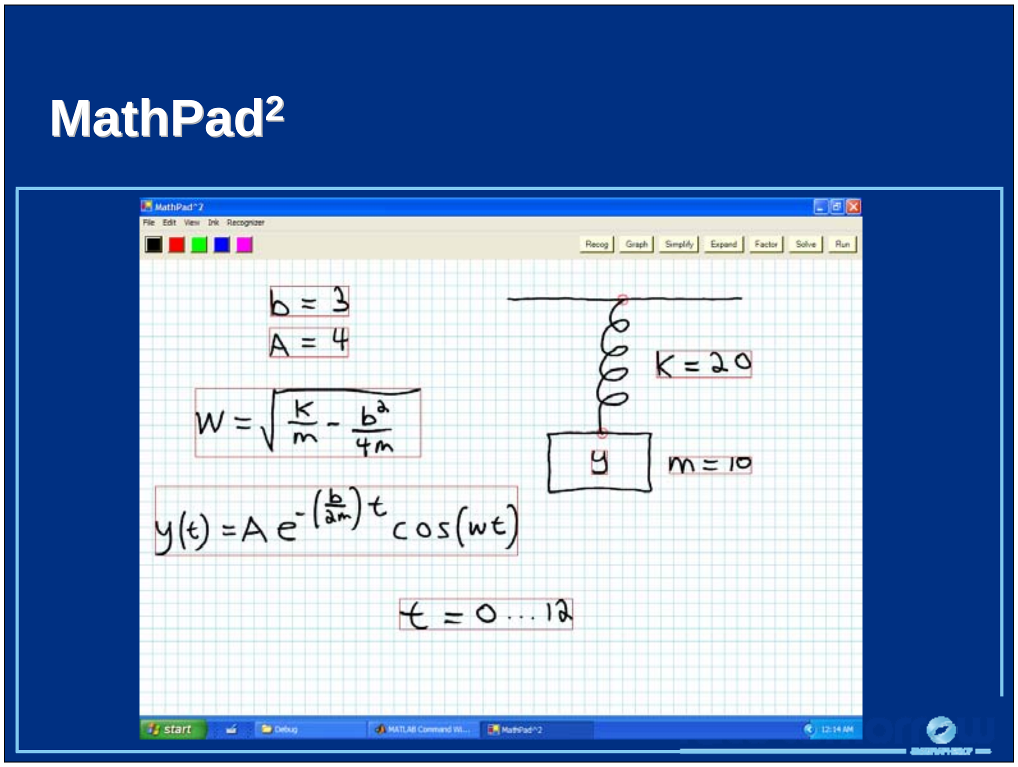


Mathematics and Physics

- Diagrams and illustrations useful for explaining mathematical concepts
  - in textbooks
  - in notebooks
  - aid in problem solving
- Inherent limitation
  - static
  - assist only in initial problem formulation



Diagrams and illustrations are very useful for explaining mathematics and physics concepts and are found in textbooks and notebooks. They are used as an aid in problem solving to give some intuition about the problem. For example, consider the figures in the slide. These illustrations would be used to help someone get started with solving a problem. The one on the top shows an illustration depicting two cars, one with constant velocity and the other with constant acceleration. The figure on the bottom shows an illustration of a block flying off of a table. The inherent limitation with these illustrations is that they are static and assist only in the initial problem formulation. Thus, it seems logical that if these illustrations could animate, they would help provide more physical intuition about a particular problem and act as an even greater guide in physics and mathematics problem solving.



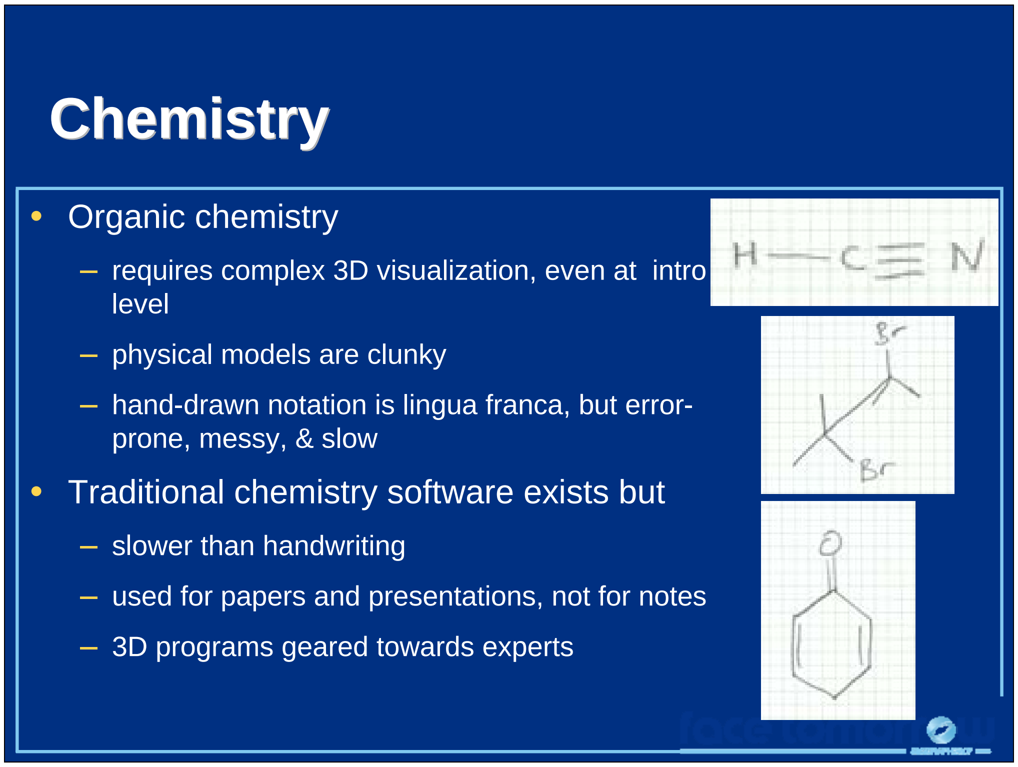
MathPad^2

MathPad^2
File Edit View Ink Recognizer
Recog Graph Simplify Expand Factor Solve Run
$b = 3$
$A = 4$
$W = \sqrt{\frac{k}{m} - \frac{b^2}{4m}}$
$y(t) = A e^{-\left(\frac{b}{2m}\right)t} \cos(wt)$
$t = 0 \ldots 12$
$k = 20$
y
$m = 10$
start

MathPad^2 is a Tablet PC-based application prototype that lets users create dynamic illustrations of mathematics and physics concepts using mathematical sketching. Users write down mathematics, make drawings, and combine the two together using associations. The system then takes that information and processes it to create an animation. The picture in the slide shows a mathematical sketch illustrating damped harmonic motion. MathPad^2 also lets users graph functions, evaluate expressions and solve equations. From an educational perspective, both students and teacher can use this tool to aid in the learning process by providing a dynamic context to abstract mathematical notation. The details behind MathPad^2 and mathematical sketching will be discussed in the Mathematical Sketching lecture in the course's afternoon session.

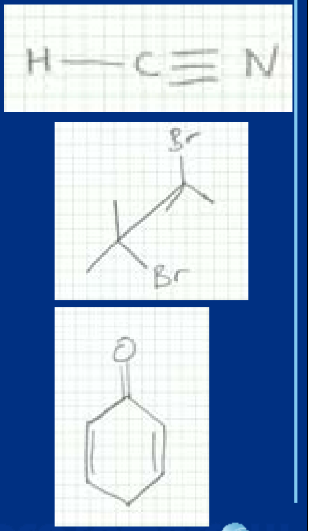
References:

LaViola, J. and Zeleznik, R. MathPad^2: A System for the Creation and Exploration of Mathematical Sketches, *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, 23(3):432-440, August 2004.

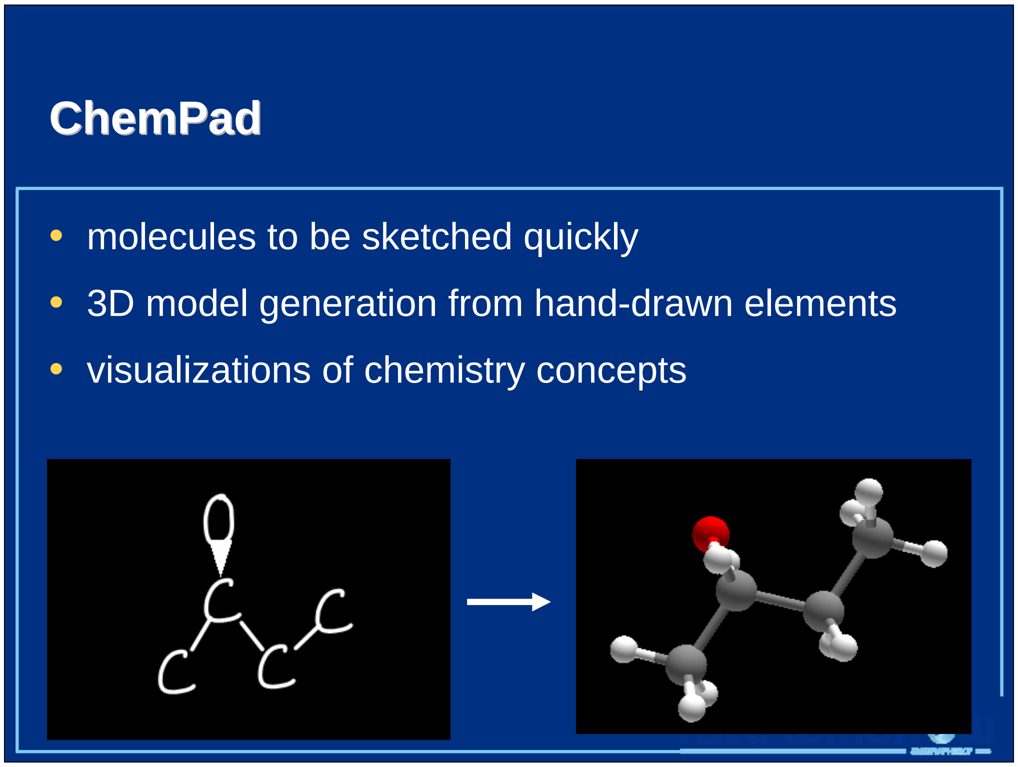


Chemistry

- Organic chemistry
  - requires complex 3D visualization, even at intro level
  - physical models are clunky
  - hand-drawn notation is lingua franca, but error-prone, messy, & slow
- Traditional chemistry software exists but
  - slower than handwriting
  - used for papers and presentations, not for notes
  - 3D programs geared towards experts

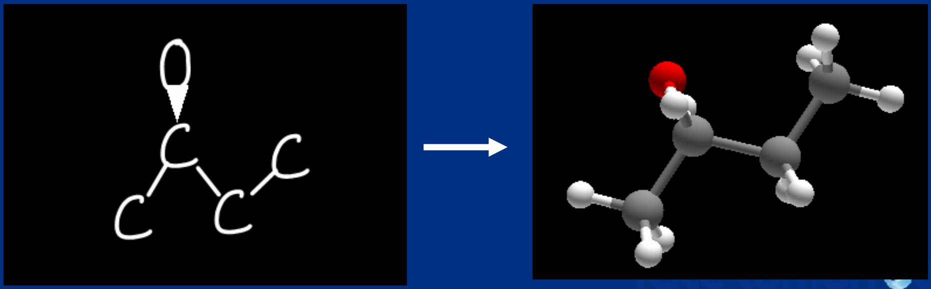


In chemistry, chemical bonds and molecules are often represented using a 2D notation that indicates connections between elements and molecules. In organic chemistry, 3D visualization is also important to understanding the nature of molecules since they may have different 3D representations given very similar 2D representations. Although, there are traditional chemistry software applications that provide 3D molecule visualization, their interfaces use cumbersome methods of input for molecule creation (e.g., selecting atoms and bonds from toolbars and menus). Therefore, a system that lets students enter a handwritten 2D molecular structure with the computer interpreting it and subsequently creating a 3D visualization of the molecule has the potential to be a powerful tool for learning concepts in chemistry.



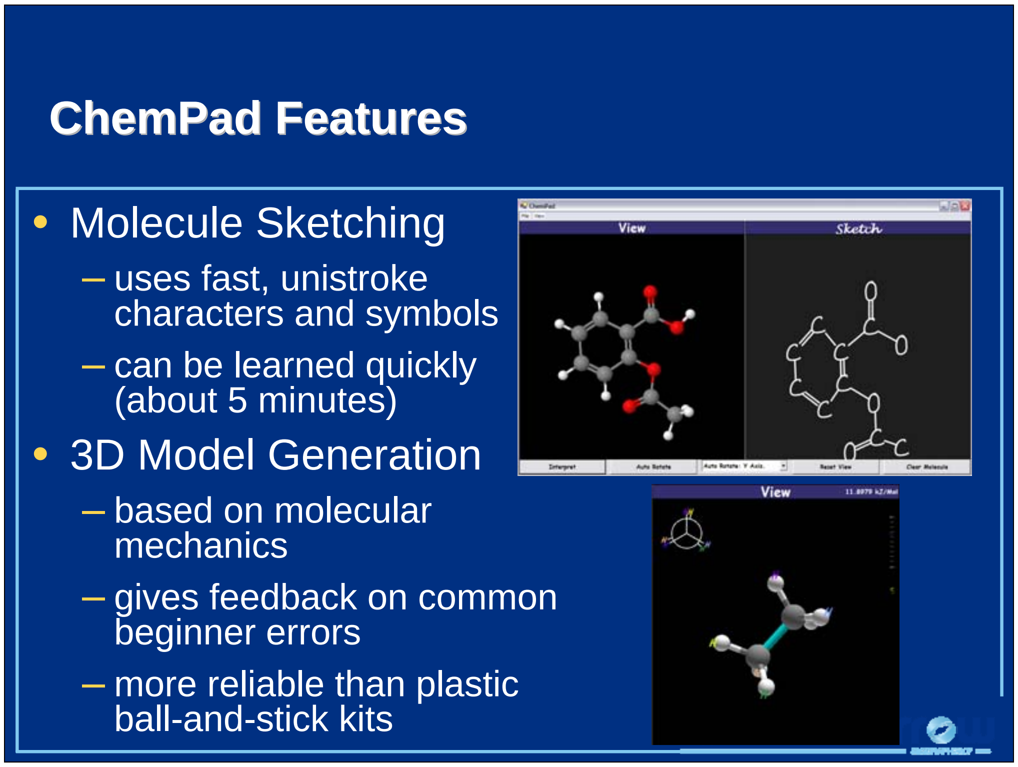
ChemPad

- molecules to be sketched quickly
- 3D model generation from hand-drawn elements
- visualizations of chemistry concepts



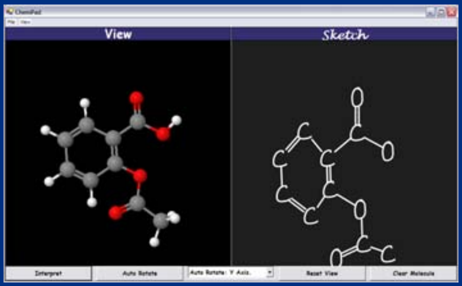
ChemPad is a pen-based application developed by Dana Tenneson at Brown University, where users can draw molecules on the computer (left figure) and observe its 3D structure (right figure). Using a Tablet PC or interactive whiteboard, users can sketch these molecules quickly so they can be converted to 3D models that will help them visualize chemistry concepts. The sketches look very similar to the drawing they would make in their notebooks or on exams or homework problems.

The ChemPad application is freely downloadable at www.chempad.org.



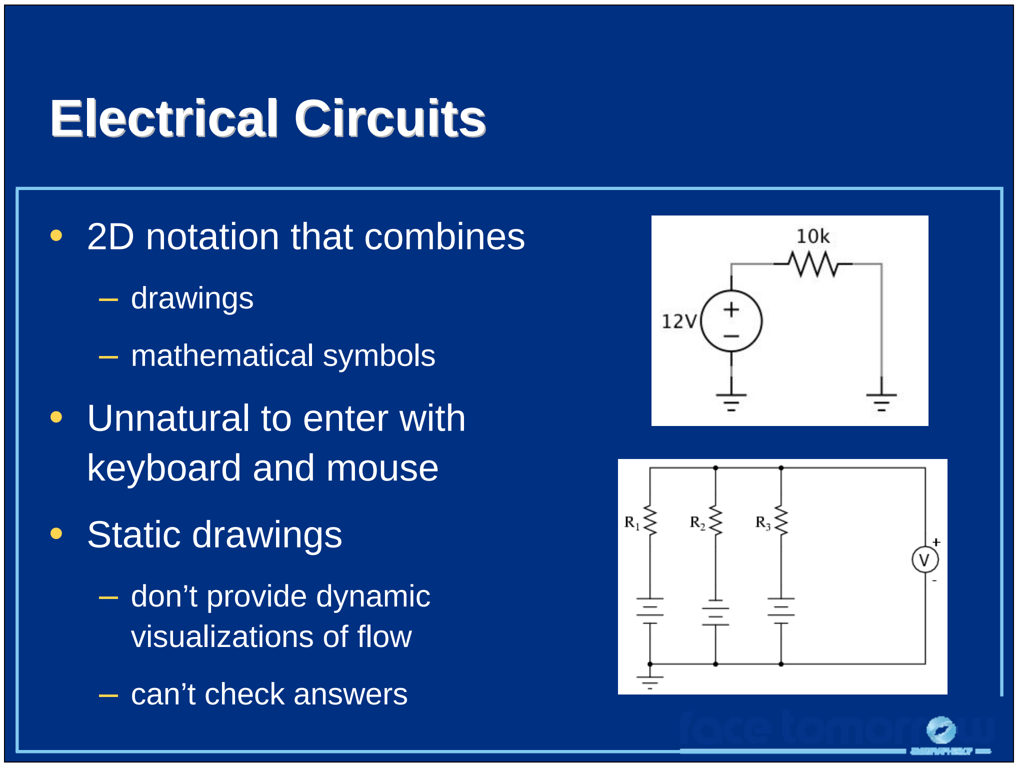
ChemPad Features

- Molecule Sketching
  - uses fast, unistroke characters and symbols
  - can be learned quickly (about 5 minutes)
- 3D Model Generation
  - based on molecular mechanics
  - gives feedback on common beginner errors
  - more reliable than plastic ball-and-stick kits



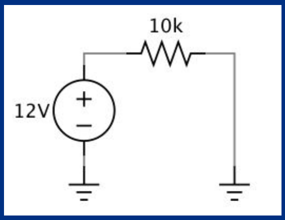
ChemPad provides molecule sketching using fast, unistroke characters and symbols that can be learned fairly quickly. For example, a student drawing Ethanol would draw two C's and an O to represent two carbon atoms and one oxygen atom in the molecule. The atoms are connected using covalent bonds by drawing straight lines between the letters. As the atoms and bonds are being drawn, the balls and sticks of the atoms and bonds are being recognized, with the student's handwriting incrementally prettified. The student then clicks on the "Interpret" button and ChemPad presents the student with a 3D scene showing the Ethanol molecule with implicit hydrogens attached and the molecule oriented in a configuration approximated by the student's drawing. The application also provides tools for stereochemistry and different 3D rendering styles.

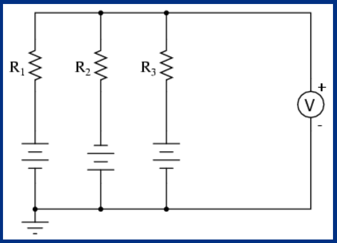
A video of ChemPad is included in the videos directory in the course notes.



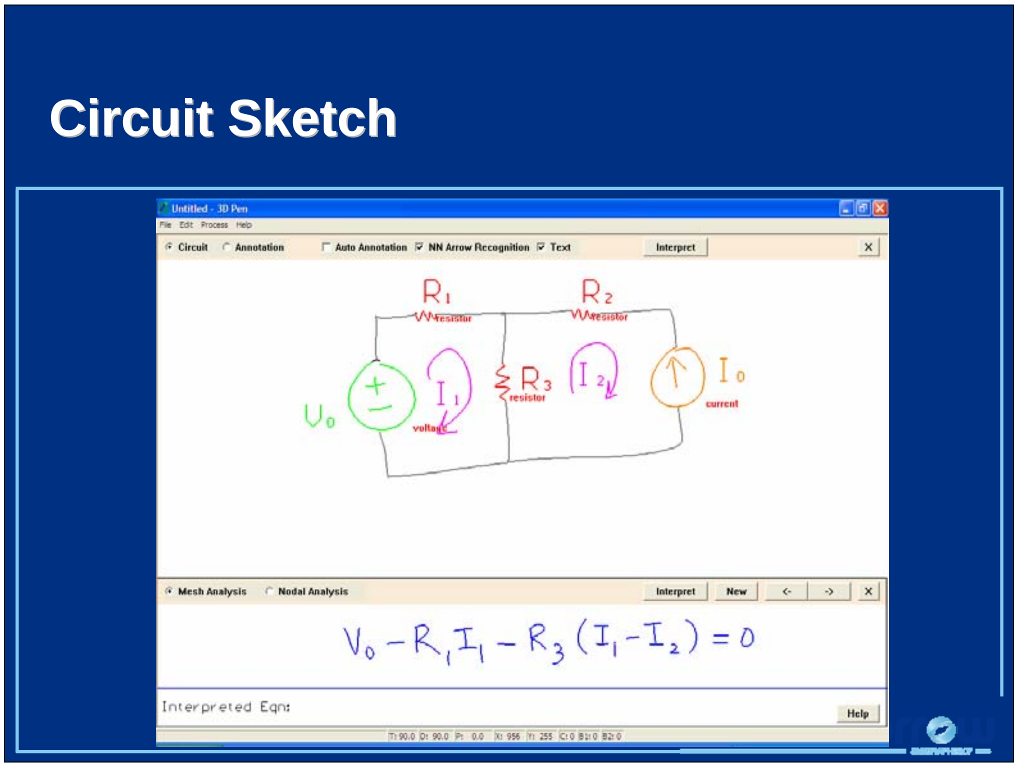
Electrical Circuits

- 2D notation that combines
  - drawings
  - mathematical symbols
- Unnatural to enter with keyboard and mouse
- Static drawings
  - don't provide dynamic visualizations of flow
  - can't check answers





There is a common notation for representing electrical circuits that is used in a variety of engineering courses. This notation is 2D in nature and combines both drawings and mathematical symbols to label different circuit components. The inherent 2D structure of these circuit diagrams makes entering them into a computer a very unnatural task when using the keyboard and mouse. Additionally, when students work with these diagrams using pencil and paper, their static nature makes it difficult to visualize current and voltage flow through the circuit and they only assist in the initial formulation of the problem (much like physics diagrams). Therefore, letting users enter electrical circuit diagrams using the standard 2D notation will provide a number of benefits since they will not have to learn a new language and, once the computer interprets the sketch, they can ask “what if” questions about the diagram, visualize flow, and check their answers to certain questions.



Circuit Sketch

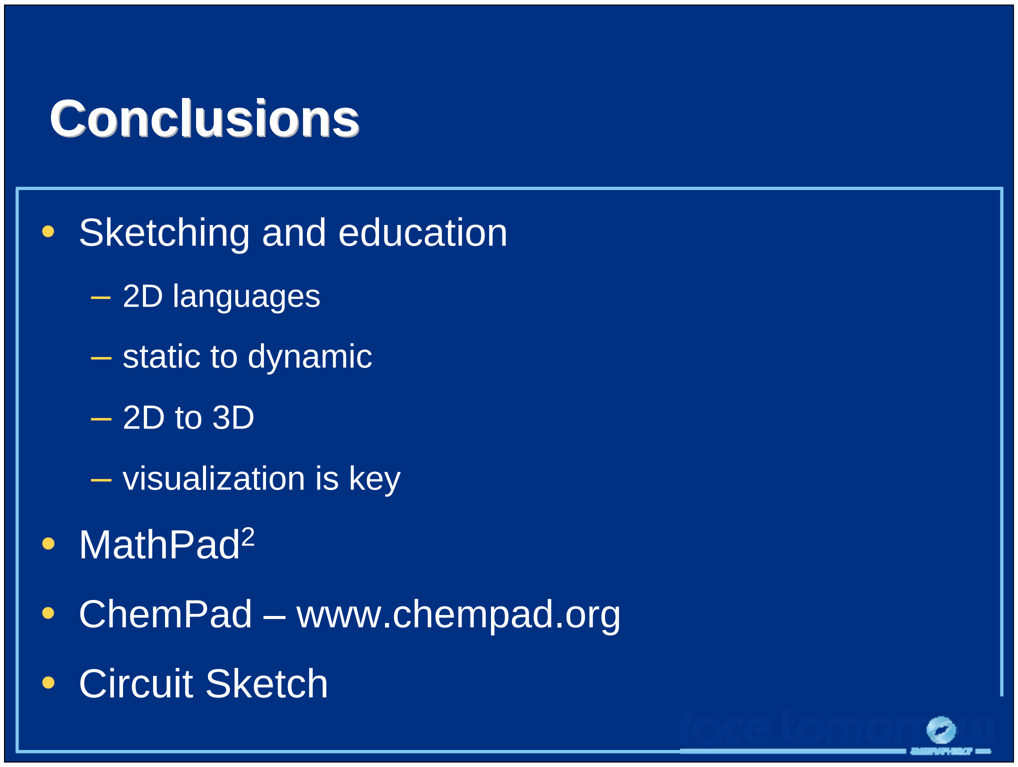
$$V_0 - R_1 I_1 - R_3 (I_1 - I_2) = 0$$

Circuit Sketch, developed by Tom Stahovich and his colleagues at UC Riverside, is a prototype pen-based tutoring system used to help students learn to analyze electrical circuits. Circuit Sketch is build upon the AC-SPARC system, a tool that transforms hand drawn circuits into analysis models for the SPICE circuit simulator. With Circuit Sketch, students can learn the fundamental principles of circuit analysis by helping them learn to apply Kirchhoff's current and voltage laws. To use the system, the student sketches a circuit and annotates it to indicate the component labels, mesh currents, and nodal voltages. The student then selects either mesh analysis (Kirchhoff's voltage law) or nodal analysis (Kirchhoff's current law) and writes the appropriate equation in the equation window as shown in the figure in the slide. The system then interprets the equation, compares them to the circuit, and provides feedback if there are errors. Future plans for Circuit Sketch include circuit simplification by identifying parallel and series components and by transforming sources.

A video of Circuit Sketch is included in the videos directory in the course notes.

References:

Gennari, L., L. B. Kara, and T. F. Stahovich, Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams. *Computers & Graphics* 29(4): 547-562 2005.



Conclusions

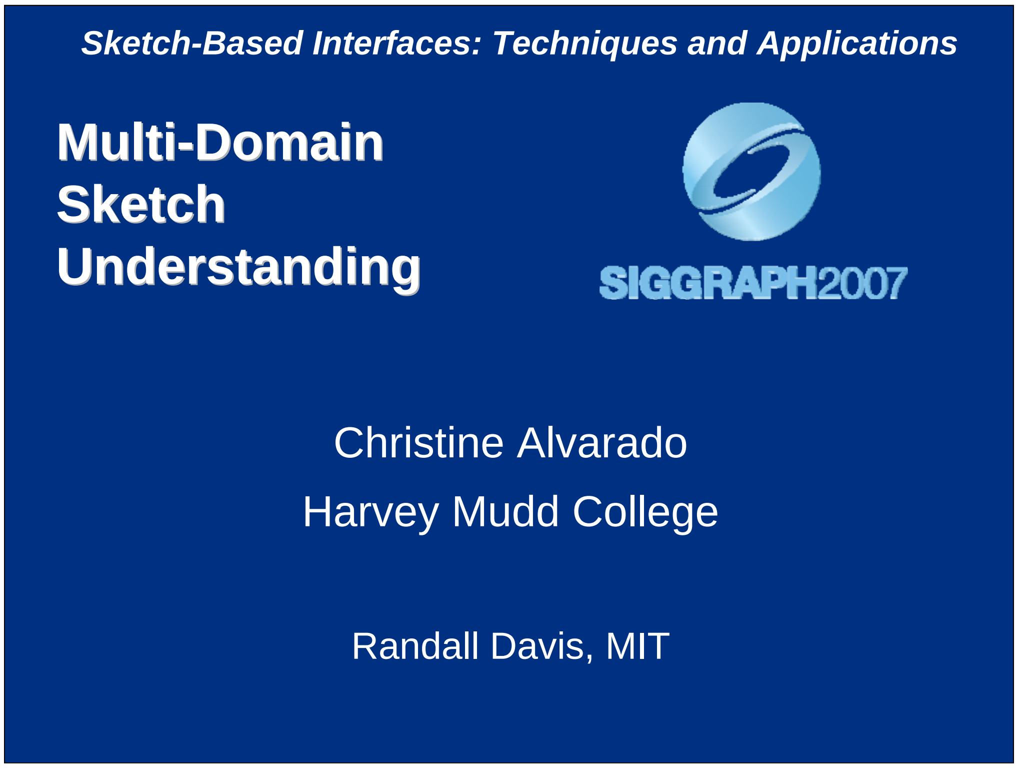
- Sketching and education
  - 2D languages
  - static to dynamic
  - 2D to 3D
  - visualization is key
- MathPad^2
- ChemPad – www.chempad.org
- Circuit Sketch

In this lecture, we have seen three examples, MathPad^2, ChemPad, and Circuit Sketch, of sketch-based interface that are targeted toward education. The common theme with all of these applications is that each application identified a 2D notational language used to convey information in a particular domain. In each case, entering this 2D language requires an unnatural mapping to a equivalent but cumbersome 1D language when using a keyboard and mouse-based interface. However, with a sketch-based interface, users can utilize these 2D notations and enter them directly into a computer with a pen or stylus as if they where writing with pencil and paper. Each of these applications also go beyond just entering the notations. Once the computer interprets them, the diagrams can be visualized in a variety of ways including 3D renderings, dynamic illustrations, and answer checking.

More details on these types of interfaces can be found in

van Dam, A. S. Becker, and R. M. Simpson, Next-Generation Educational Software: Why We Need It and a Research Agenda for Getting It, *Educause Review*, 40(2):26-43, March/April 2005.

which is included in the course notes.



Sketch-Based Interfaces: Techniques and Applications

Multi-Domain Sketch Understanding

SIGGRAPH2007

Christine Alvarado

Harvey Mudd College

Randall Davis, MIT

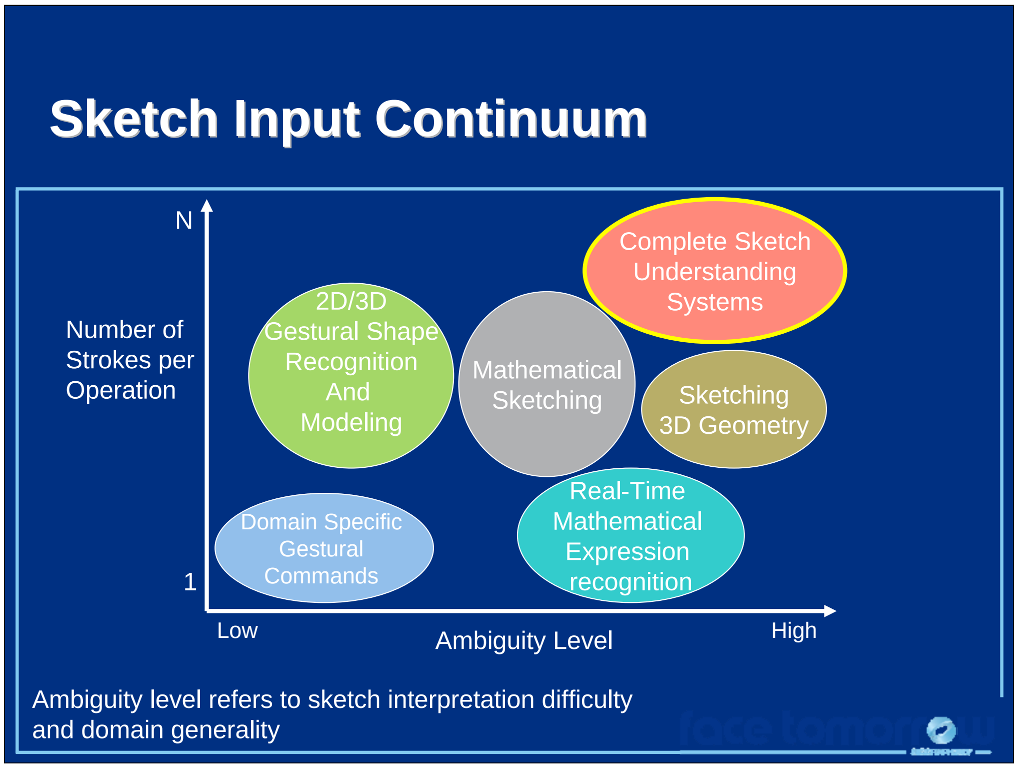
Multi-Domain Sketch Understanding

The work I will describe in this session was again done in conjunction with Randy Davis and other members of my former group at MIT.

My Contact information:

Christine Alvarado
Assistant Professor
Harvey Mudd College, CS Department
1250 N. Dartmouth Ave.
Claremont, CA 91711

Email: alvarado@cs.hmc.edu
http://www.cs.hmc.edu/~alvarado/

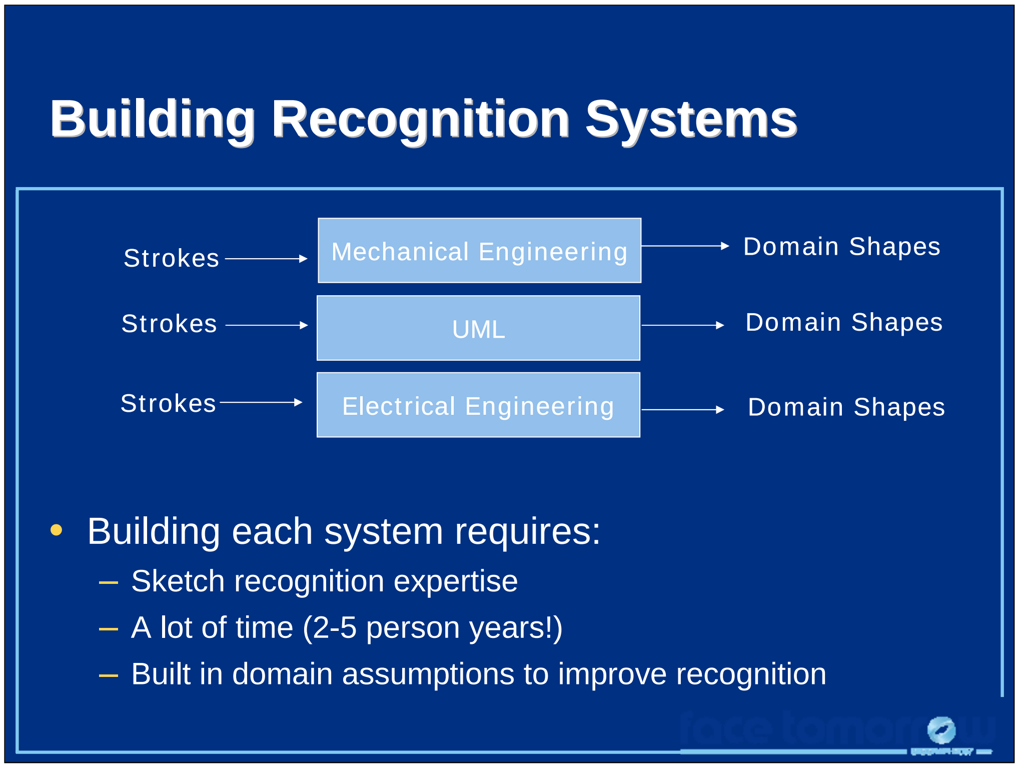


Sketch Input Continuum

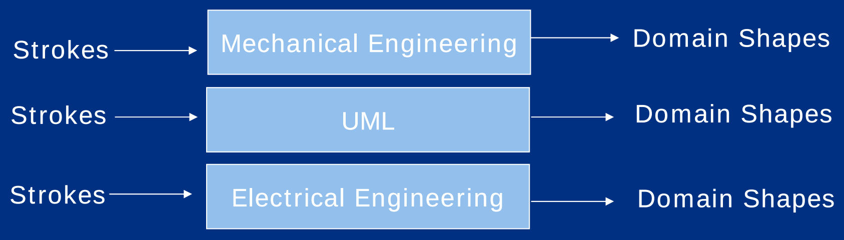
N
1
Number of Strokes per Operation
Complete Sketch Understanding Systems
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition
Low
Ambiguity Level
High

Ambiguity level refers to sketch interpretation difficulty and domain generality

In this session we will again discuss the task of complete sketch understanding, focusing in particular on diagram recognition. This time, however, we will discuss how to build *domain-flexible* recognition systems.



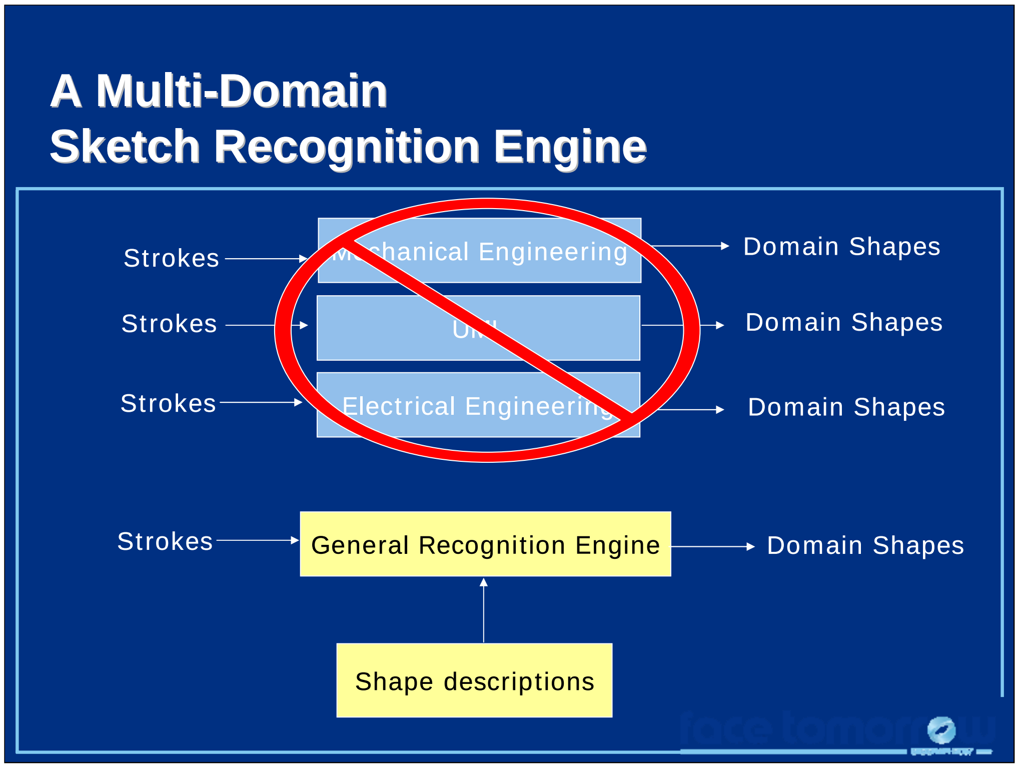
Building Recognition Systems



- Building each system requires:
  - Sketch recognition expertise
  - A lot of time (2-5 person years!)
  - Built in domain assumptions to improve recognition

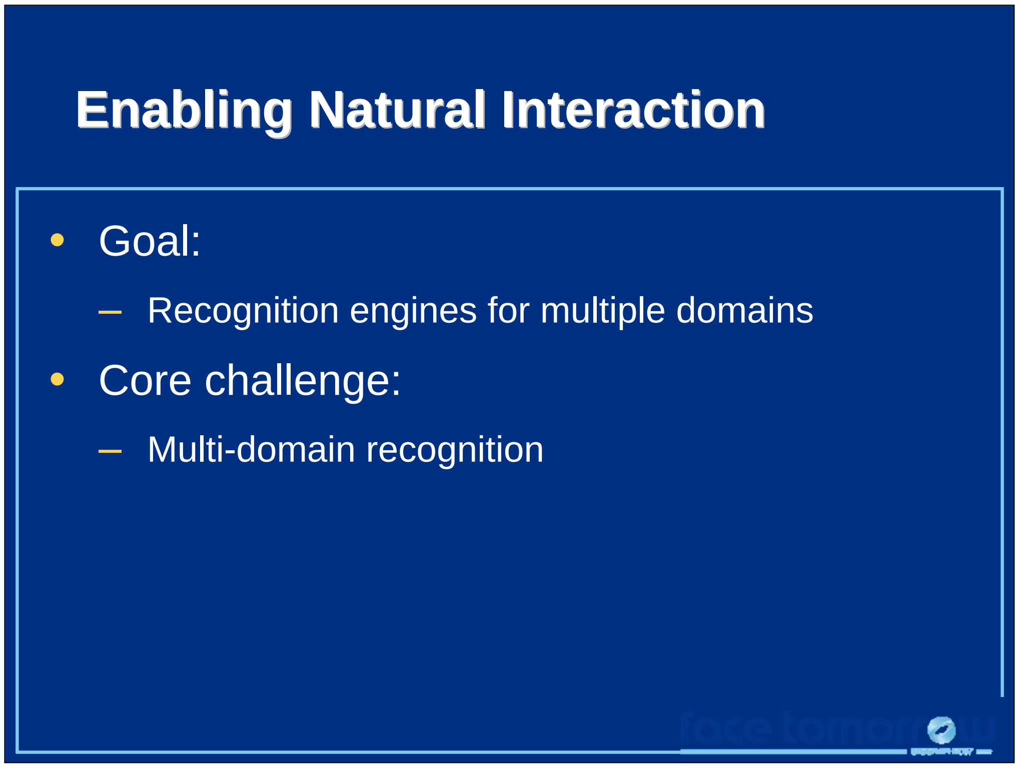
Sketch recognition systems will be extremely useful in many domains. As you have seen, researchers are actively working on building recognition systems in domains such as electrical engineering, software design, chemistry, mechanical engineering, among others.

Unfortunately, building a recognition system is a huge undertaking requiring sketch recognition expertise, a lot of time and a lot of hand-coded recognition routines to encode specific domain information.



A Multi-Domain Sketch Recognition Engine

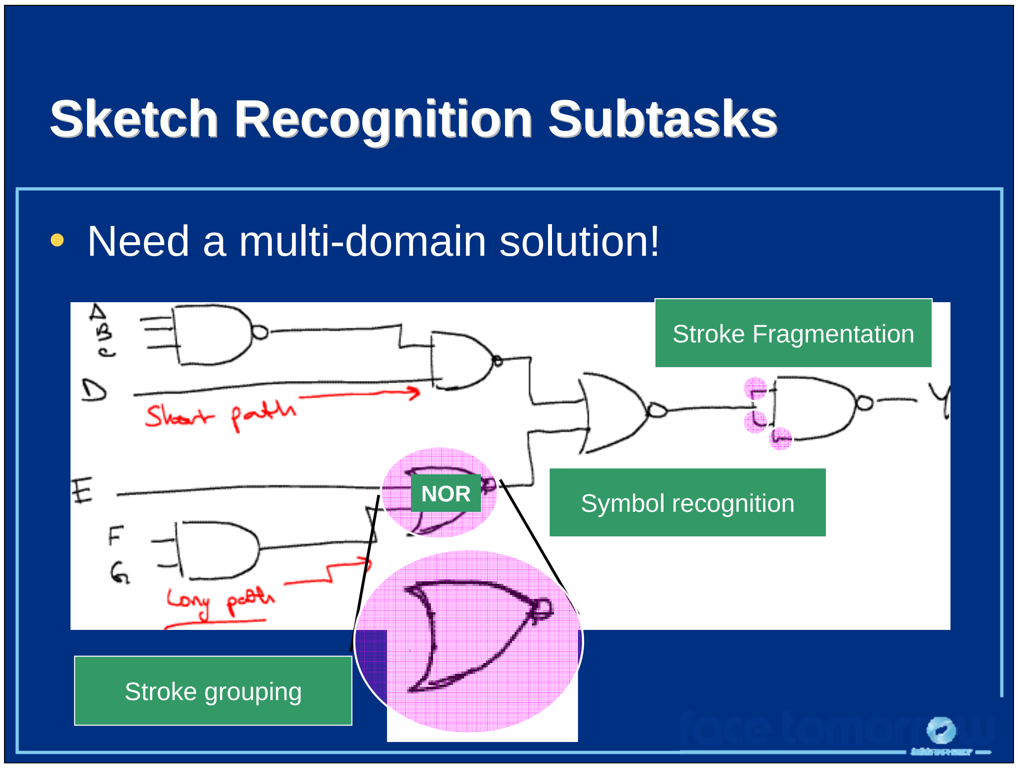
Instead, I will present a *general recognition engine* that may be adapted to recognize diagrams in a number of domains. This engine takes as input shape descriptions for a particular domain and can then recognize hand-drawn diagrams in that domain without any hand-tuning.



Enabling Natural Interaction

- Goal:
  - Recognition engines for multiple domains
- Core challenge:
  - Multi-domain recognition

The core challenge in constructing a multi-domain recognition system is to develop recognition techniques that work across multiple domains.

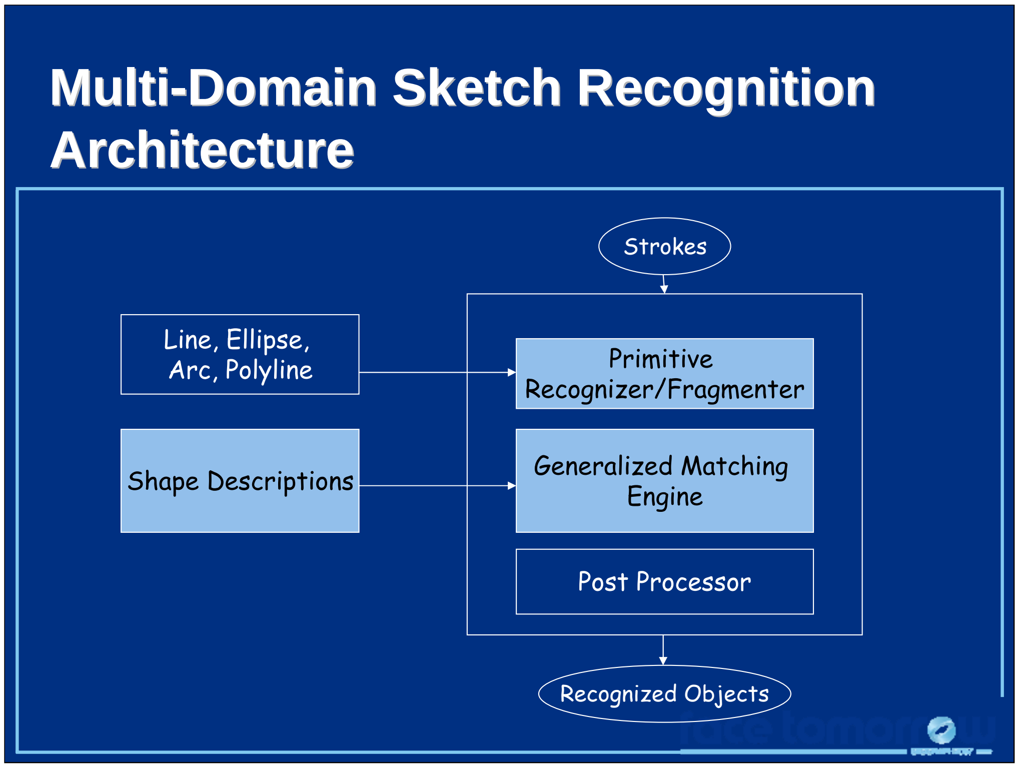


Sketch Recognition Subtasks

- Need a multi-domain solution!

Stroke Fragmentation
Symbol recognition
Stroke grouping
NOR
Short path
Long path
A
B
C
D
E
F
G
Y

Recall that we identified three sketch recognition subtasks: stroke fragmentation, stroke grouping and symbol recognition. Our challenge now is to develop a general technique for each of these tasks that does not depend directly on any particular domain. Of course, we said this morning that domain information is essential to recognition. The approach I describe disaggregates domain information from the recognition algorithm, allowing the core recognition system to be adapted to many domains simply by inputting a small amount of domain-specific information.

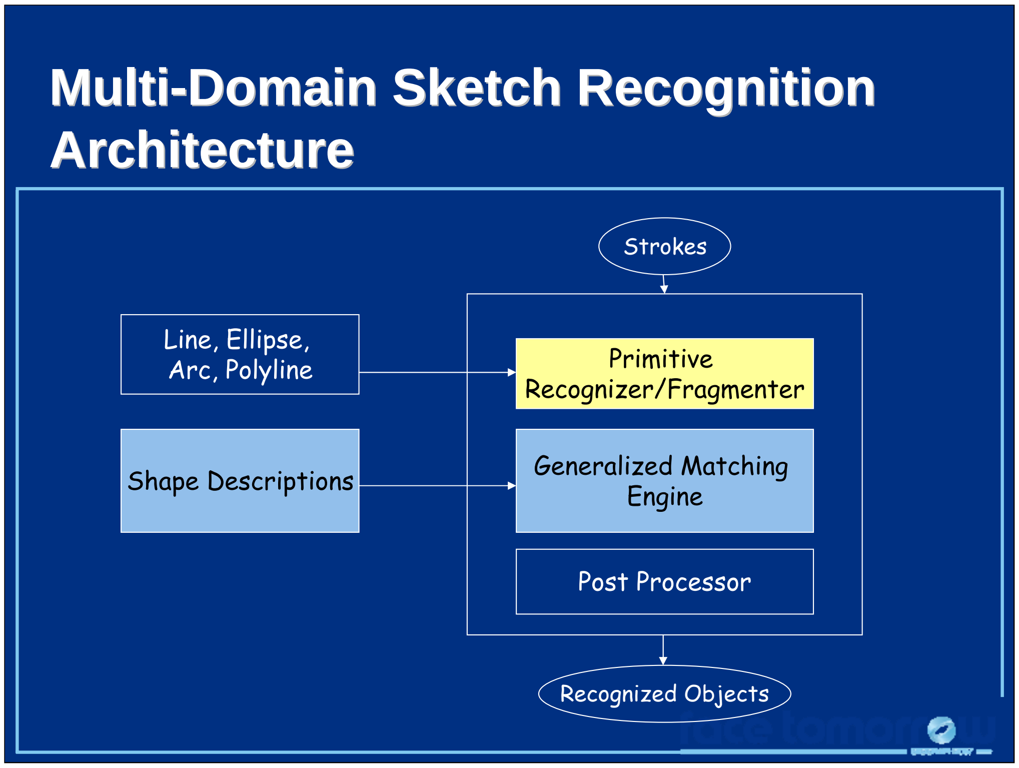


Multi-Domain Sketch Recognition Architecture

Strokes
Line, Ellipse, Arc, Polyline
Primitive Recognizer/Fragmenter
Shape Descriptions
Generalized Matching Engine
Post Processor
Recognized Objects

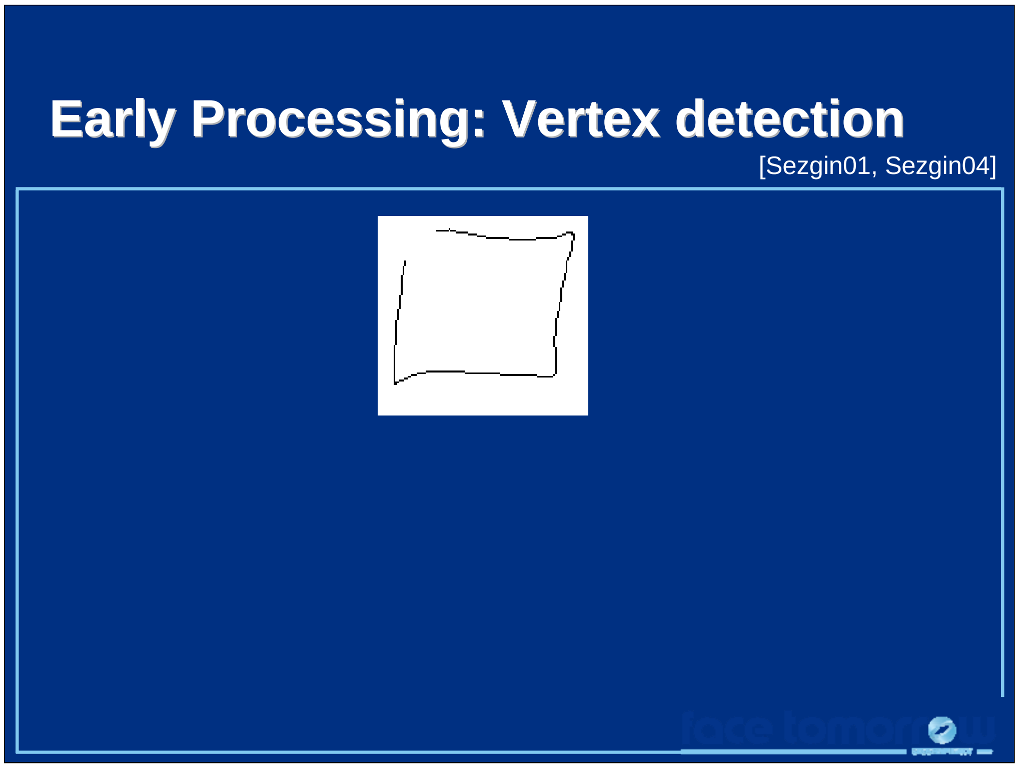
This slide shows an overview of the multi-domain recognition architecture I will discuss. As the user draws, her strokes are first sent to a primitive recognizer that breaks the strokes at the corners and then recognizes generic low level shapes such as lines and arcs. Next, the primitive shapes are passed to a generic shape matching engine that groups the primitives into individual symbols and then recognizes them by matching them against domain-specific shape descriptions, input to the engine. Finally the recognized symbols are passed to a post-processor and the sketch is output as a recognized diagram.

This session will focus on the stroke fragmentation algorithms, the shape descriptions and the generalized matching algorithms.



Multi-Domain Sketch Recognition Architecture

Strokes
Line, Ellipse, Arc, Polyline
Primitive Recognizer/Fragmenter
Shape Descriptions
Generalized Matching Engine
Post Processor
Recognized Objects



Early Processing: Vertex detection

[Sezgin01, Sezgin04]

The first stage in our multi-domain recognition process is to break the user's stroke into primitive shapes such as lines and arcs. We use a domain-independent approach, although we will see later how domain-specific information can help correct errors in stroke fragmentation and primitive recognition.

Breaking a user's stroke into primitive shapes involves detecting the corners, or vertices, in the stroke. In this section I will give an overview of the algorithms presented in

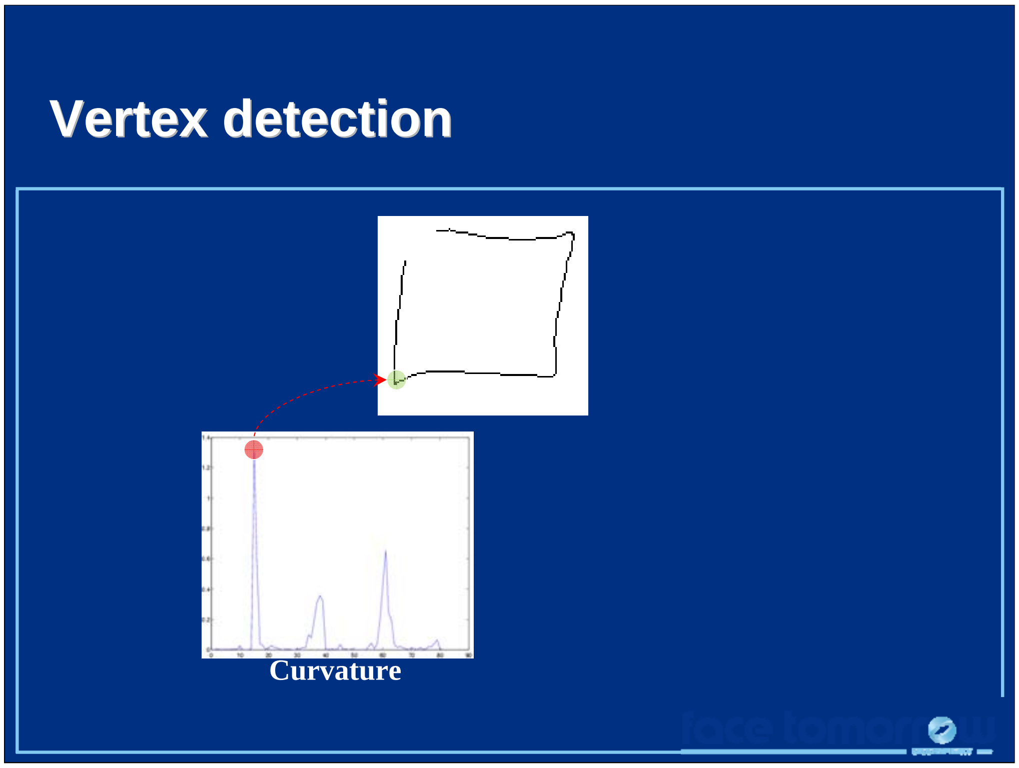
Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. **Sketch Based Interfaces: Early Processing for Sketch Understanding.** *Workshop on Perceptive User Interfaces*, Orlando FL . 2001.

Tevfik Metin Sezgin and Randall Davis. **Scale-space Based Feature Point Detection for Digital Ink.** In *Making Pen-Based Interaction Intelligent and Natural* . 2004.

A similar approach is described in

Thomas F. Stahovich "Segmentation of Pen Strokes Using Pen Speed. "
AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural.

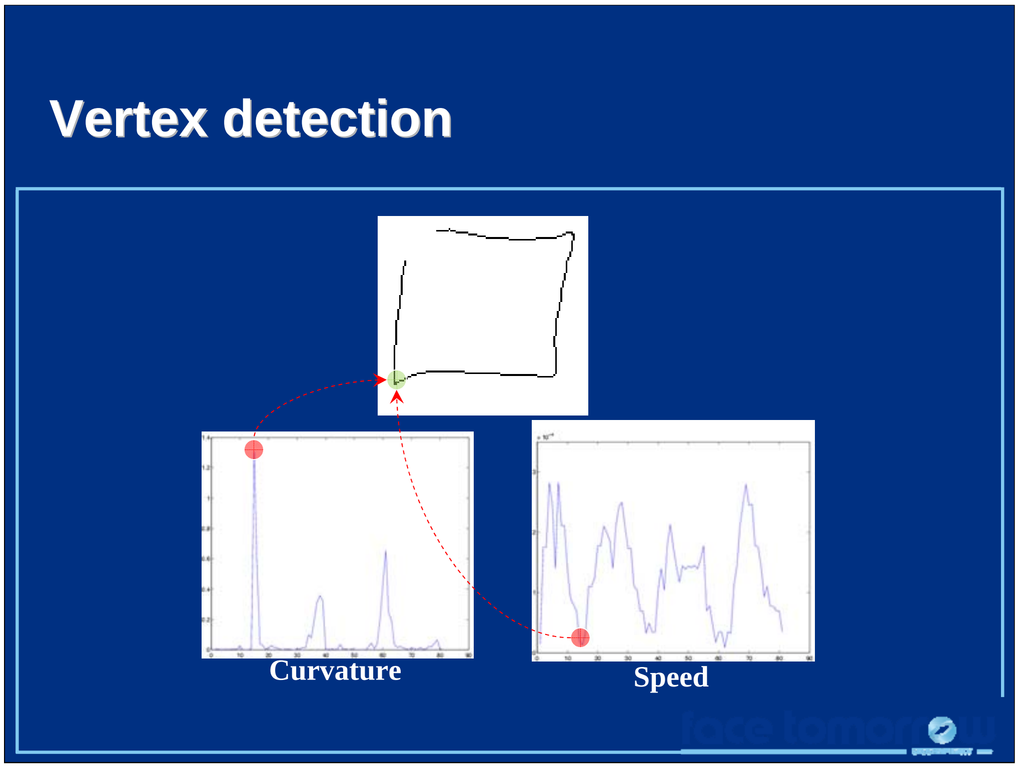
The key to all of these approaches is to combine curvature and stroke speed information to detect corners in the stroke.



Vertex detection

Curvature

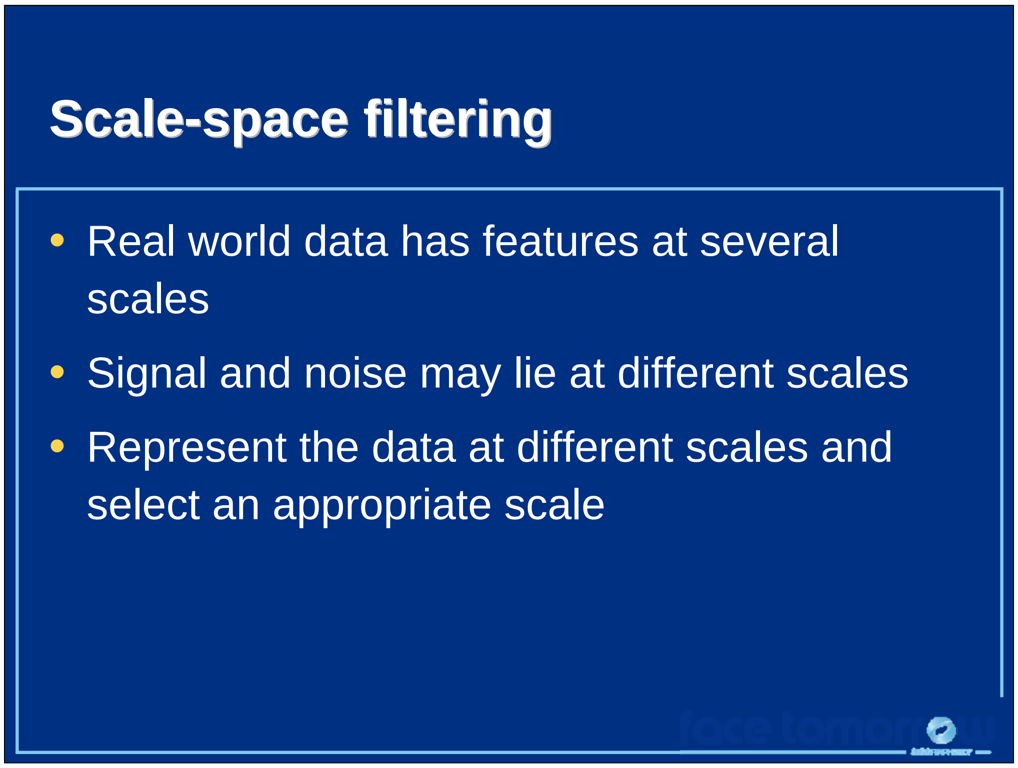
Of course, curvature is high at the corners…



… but people also tend to slow down when they change direction, so pen speed is also low at the corners.

Thus, corner detection is simply a matter of detecting points that are both local maxima on the curvature graph and local minima on the speed graph.

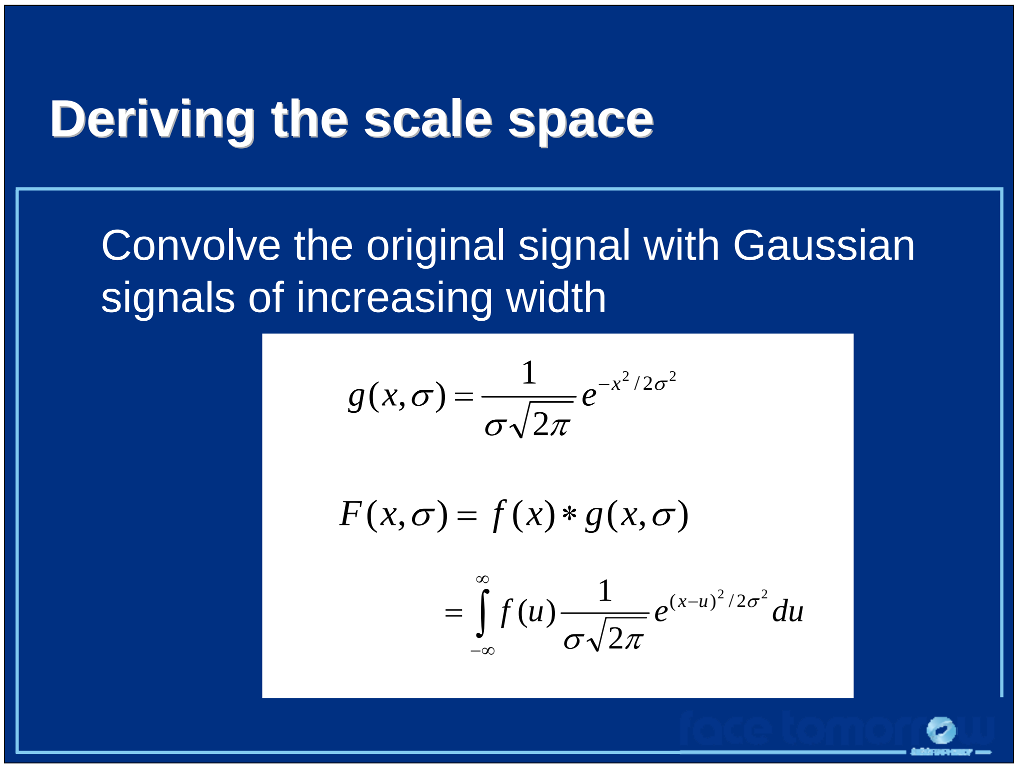
Unfortunately, you can see that these graphs are quite noisy. How exactly do we detect local maxima/minima that we care about? Clearly, we must set a threshold, but then the question becomes, what threshold should we choose? How do we know which peaks are noise and which are real corners?



Scale-space filtering

- Real world data has features at several scales
- Signal and noise may lie at different scales
- Represent the data at different scales and select an appropriate scale

Scale-space filtering is a technique that helps us separate the noise from the true corners. The fundamental idea is that noise and signal (corners) lie at different scales in our input. Basically, our plan is to smooth our data more and more. As we do this, more and more of the local minima, the minima that correspond to the noise, will drop out. If we smooth too much, we will lose the corners as well. The goal is to find the scale at which we filter all of the noise but none of the signal.



Deriving the scale space

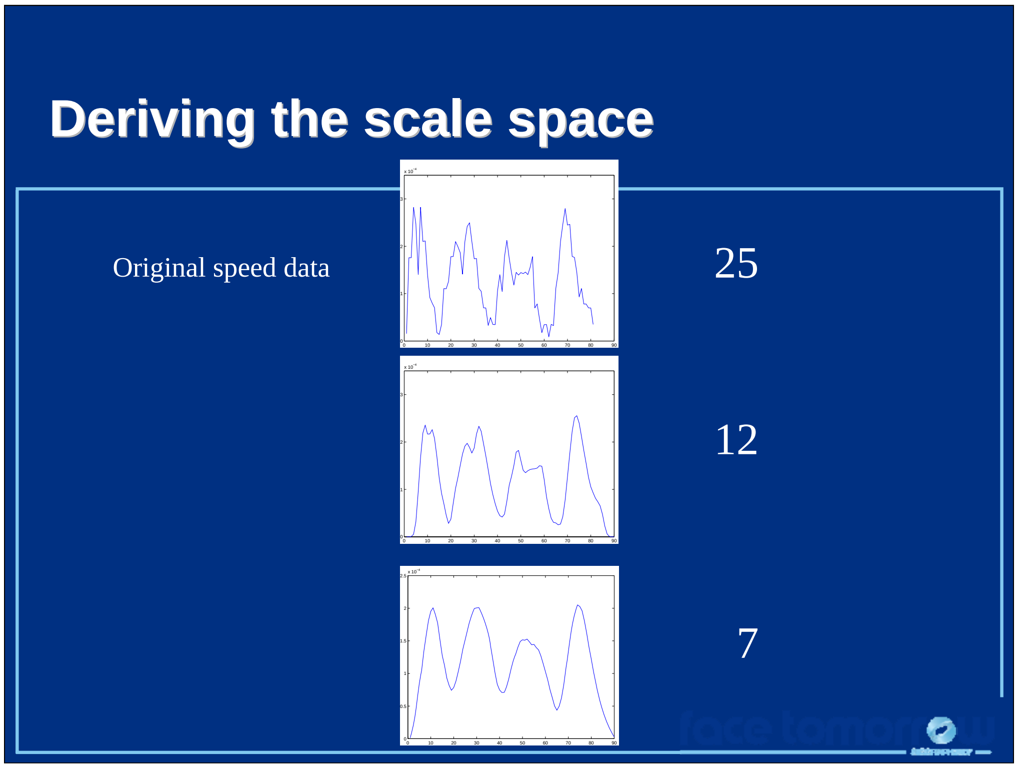
Convolve the original signal with Gaussian signals of increasing width

$$g(x,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2}$$

$$F(x,\sigma) = f(x) * g(x,\sigma)$$

$$= \int_{-\infty}^{\infty} f(u) \frac{1}{\sigma\sqrt{2\pi}} e^{(x-u)^2/2\sigma^2} du$$

We smooth our data by convolving it with Gaussian signals of increasing width.



Deriving the scale space

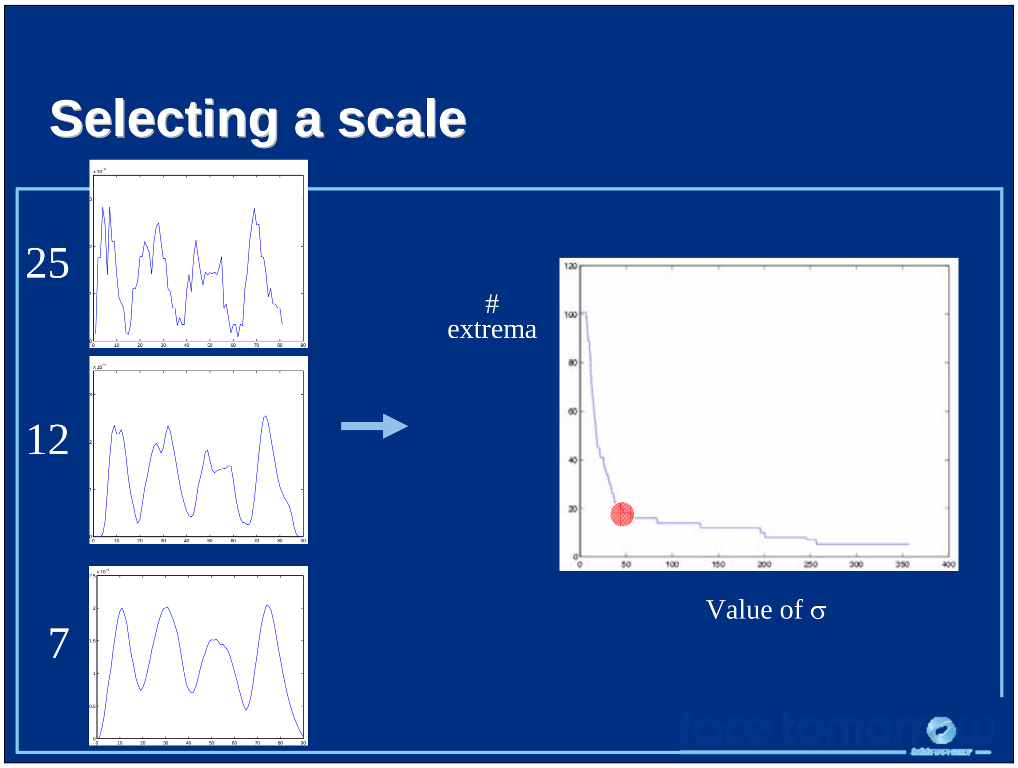
Original speed data

25

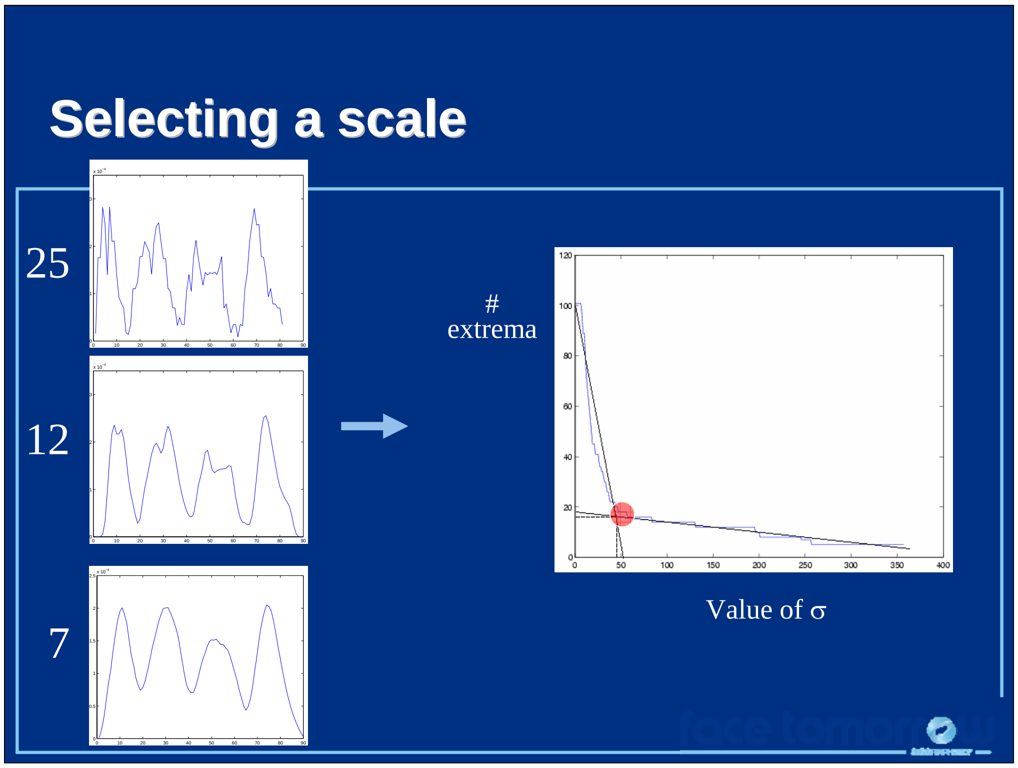
12

7

As we smooth the data, more and more local minima drop out.

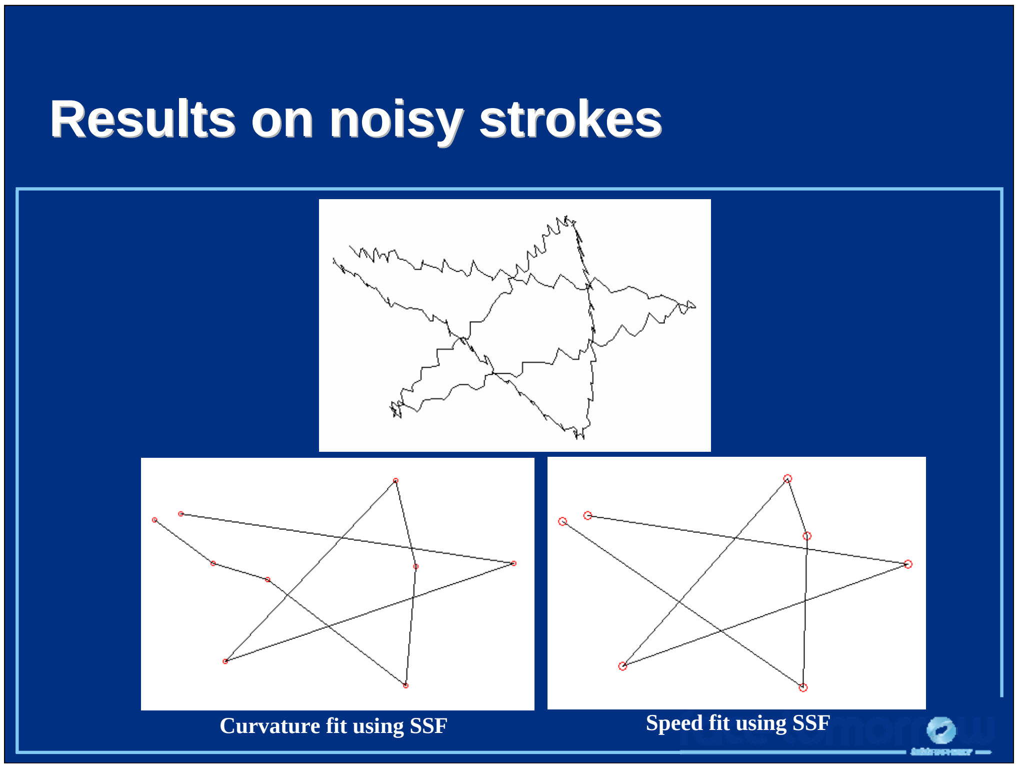


But how do we know when we are no longer filtering noise and have started to filter corners instead? Corners, fortunately, tend to be much more "robust" to smoothing than noise. Therefore as we increase the smoothing, the noise drops out quickly, but the corners do not drop out until the width of the Gaussian is relatively large. When we plot the standard deviation against the number of local maxima in the curvature graph (or minima in the speed graph) we see that the slope of the curve is very steep at first, but flattens out relatively quickly. The point at which the curve flattens is (roughly) the point at which we start filtering corners instead of noise.

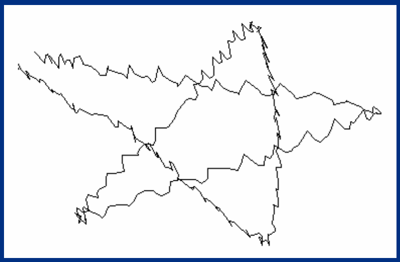


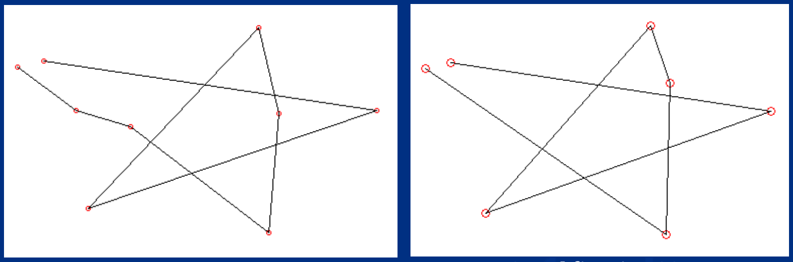
Selecting a scale

To find this point, we fit one line to the first part of the curve and one line to the last part of the curve. The point at which these lines intersect is the optimal scale at which to filter.



Results on noisy strokes

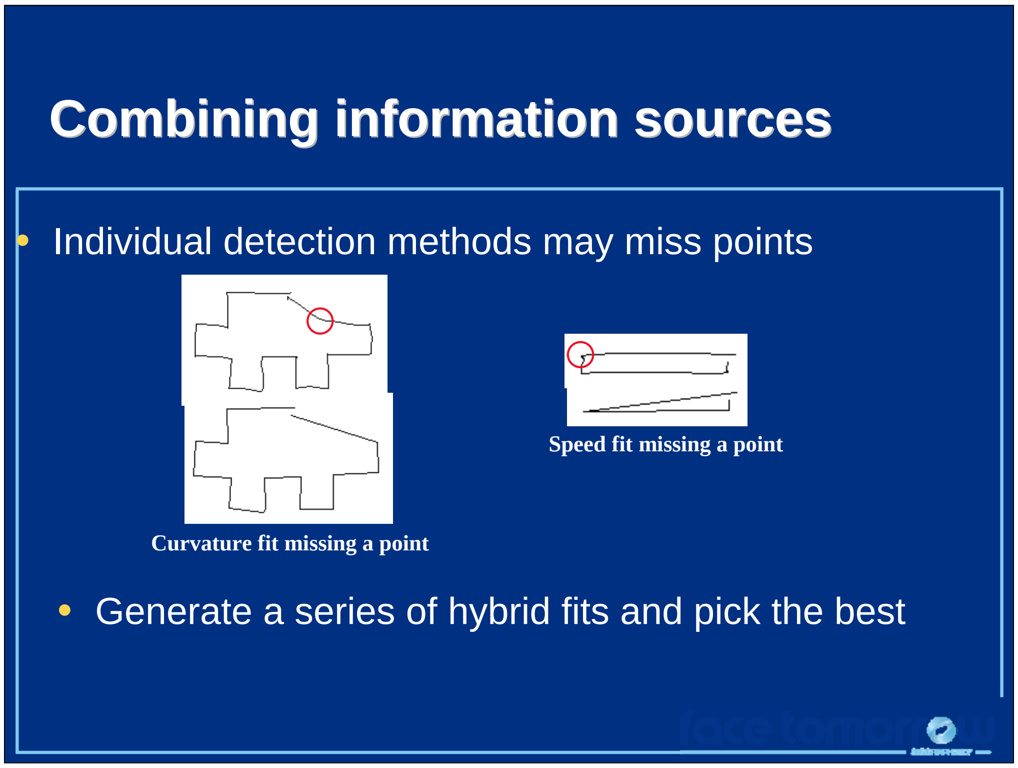




Curvature fit using SSF

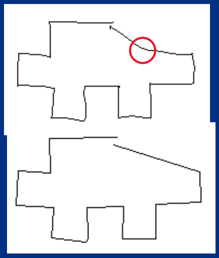
Speed fit using SSF

Scale space filtering works well on extremely noisy data. Here we see the corners detected using scale space filtering on the speed graph and the curvature graph.

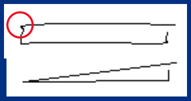


Combining information sources

- Individual detection methods may miss points



Curvature fit missing a point

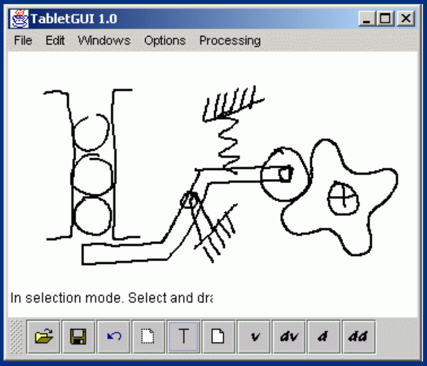


Speed fit missing a point

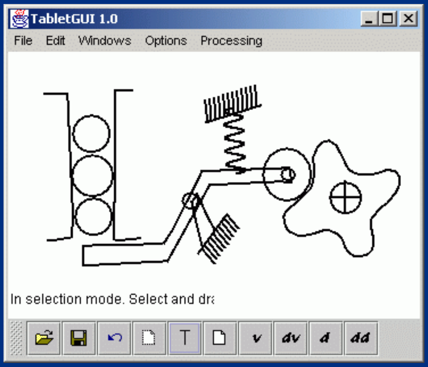
- Generate a series of hybrid fits and pick the best

Speed and curvature each provide an important piece of information that helps us locate corners, but either one alone is not sufficient. To find the best set of corners we must combine the two sources of information. We generate a set of corners from both the curvature and the speed data and then measure the quality of each corner by fitting primitive shapes (lines and spline curves) to the portions of the strokes between the proposed corners. If the primitive shape fit is unaffected when a corner is removed, then the system removes that corner. If the fit gets much worse, then the corner is kept.

Results

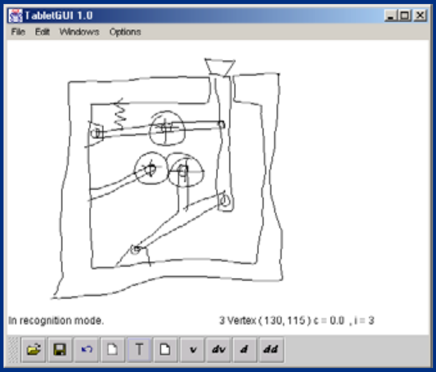


In selection mode. Select and dr

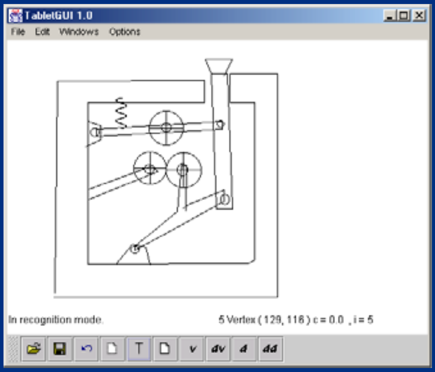


In selection mode. Select and dr

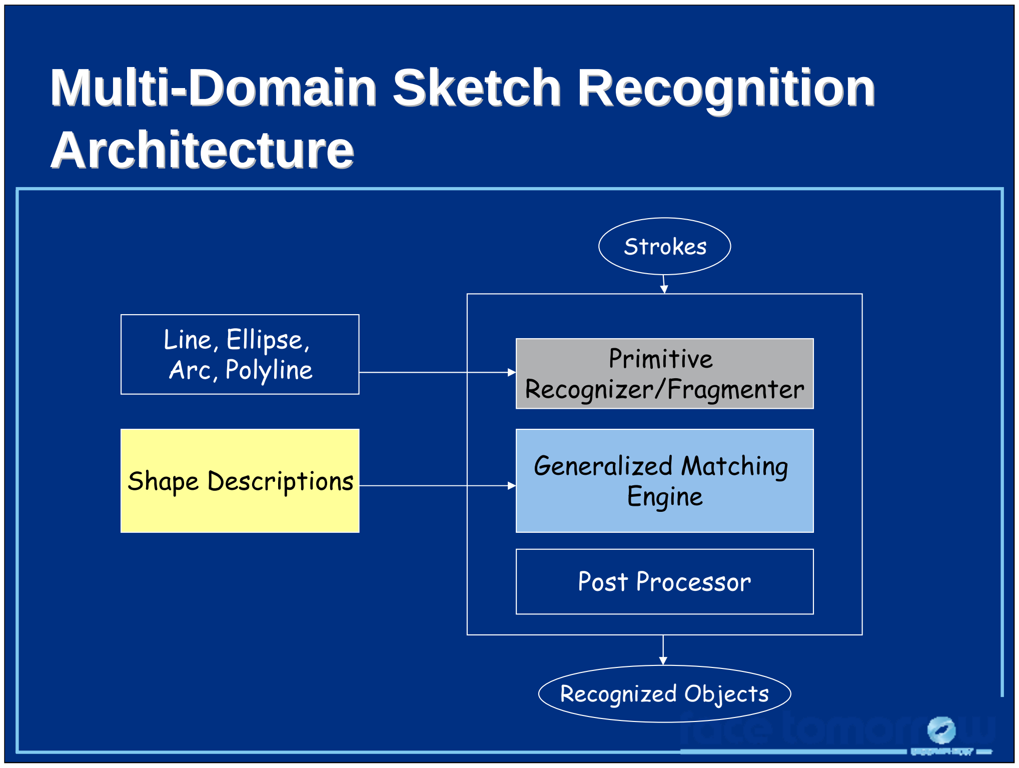
Results



```
In recognition mode.        3 Vertex ( 130, 115 ) c = 0.0  , i = 3
```



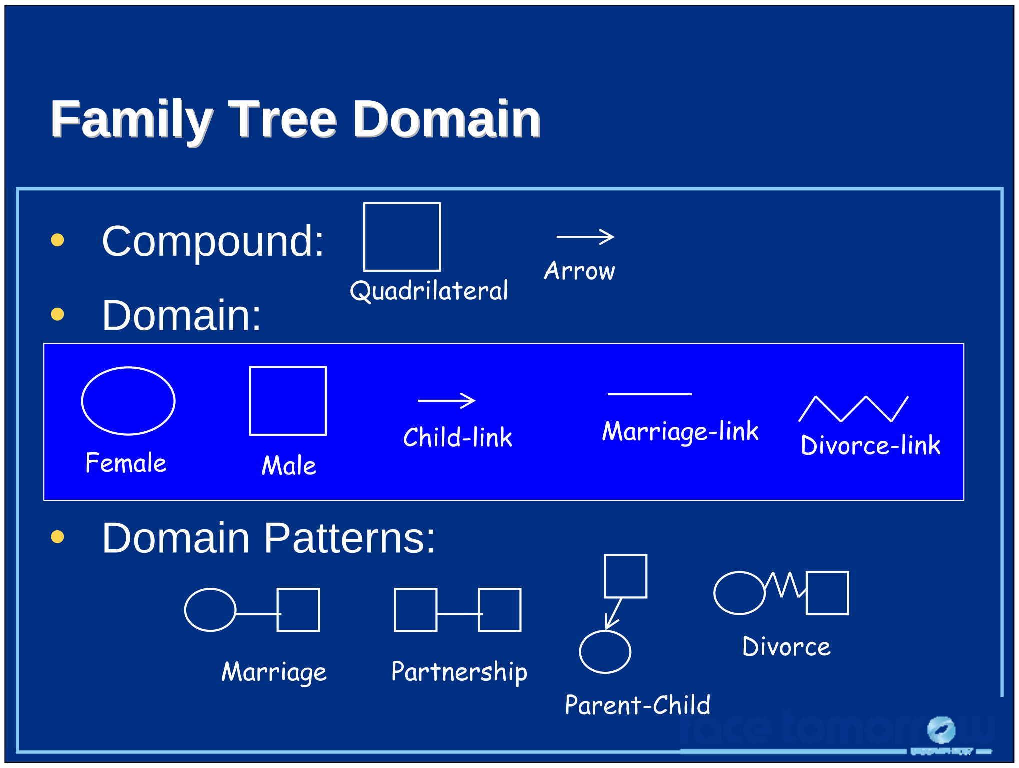
```
In recognition mode.        5 Vertex ( 129, 118 ) c = 0.0  , i = 5
```



Multi-Domain Sketch Recognition Architecture

Strokes
Line, Ellipse, Arc, Polyline
Primitive Recognizer/Fragmenter
Shape Descriptions
Generalized Matching Engine
Post Processor
Recognized Objects

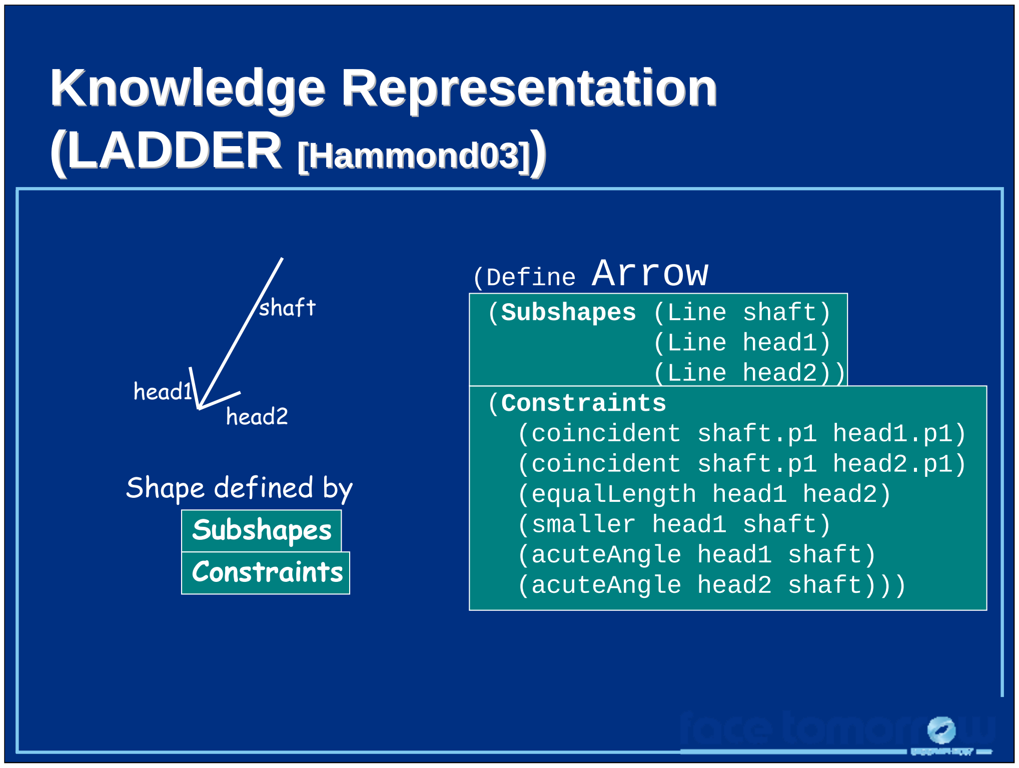
The next piece of our multi-domain system is the shape descriptions that will be fed into the generalized matching engine. Our goals for shape description are (1) that they are easy for the human to specify (or for the computer to learn) and (2) that they are useful for recognition. To be useful for recognition, a description must be a general representation of a symbol and it must be easy to match the description to the user's strokes to determine if the strokes match the description.



Family Tree Domain

- Compound:
- Domain:
- Domain Patterns:

We use the family tree domain to illustrate our shape representation. Our shape descriptions are hierarchical. At the lowest level we have general compound shapes (built from lines) including quadrilaterals and arrows. The domain specific shapes include ellipses to represent females, quadrilaterals to represent males, arrows to represent links between a parent and a child, lines to represent marriages marriage, and jagged lines to represent divorces. Finally, as we saw this morning, a robust recognition system must not only know about the symbols in the domain but also the context in which those symbols are likely to appear. The contexts in the family tree domain are given here. Although these contexts are input to the recognition system, recognition takes place at the domain shape level (i.e., the output of the system is a set of domain shapes).



Knowledge Representation (LADDER [Hammond03])

shaft
head1
head2

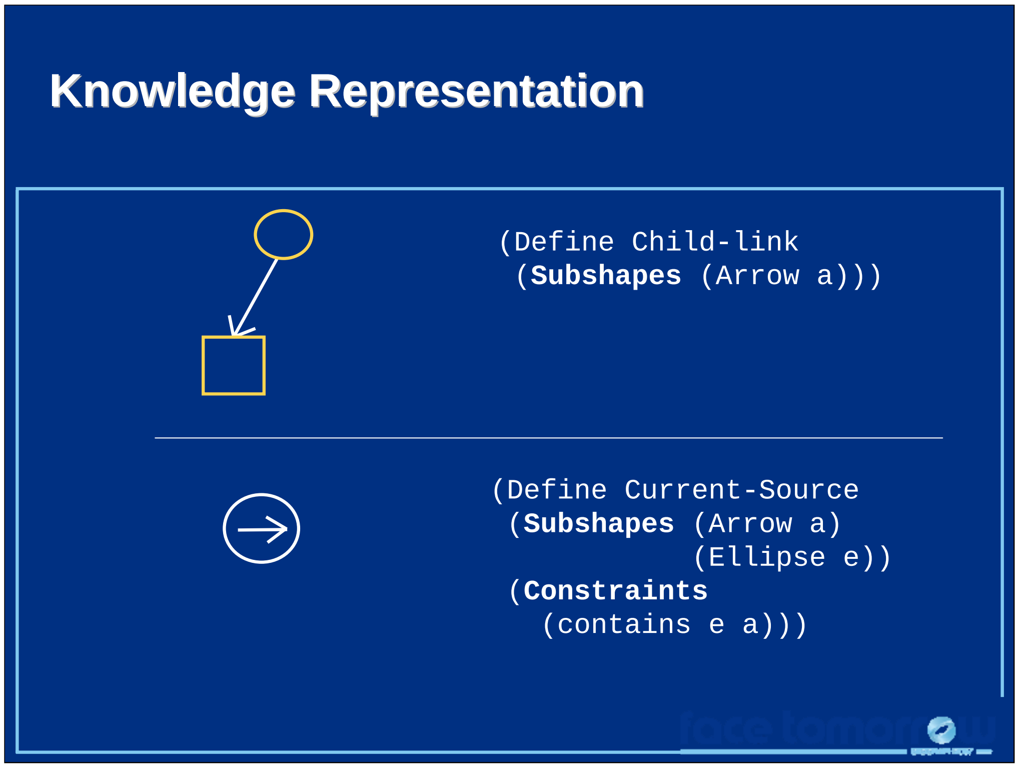
Shape defined by
Subshapes
Constraints

```
(Define Arrow
 (Subshapes (Line shaft)
            (Line head1)
            (Line head2))
 (Constraints
   (coincident shaft.p1 head1.p1)
   (coincident shaft.p1 head2.p1)
   (equalLength head1 head2)
   (smaller head1 shaft)
   (acuteAngle head1 shaft)
   (acuteAngle head2 shaft)))
```

We use a hierarchical shape description language called LADDER to represent compound shapes, domain shapes and domain contexts. In LADDER, a shape is defined by a set of subshapes (e.g., the three lines in an arrow) and a set of constraints that must hold between those subshapes (e.g., the two lines in the head of the arrow must be the same length and shorter than the shaft of the arrow).

References:

Tracy Hammond and Randall Davis. LADDER, a sketching language for user interface developers. Elsevier, Computers and Graphics 28, pp.518-532. 2005.

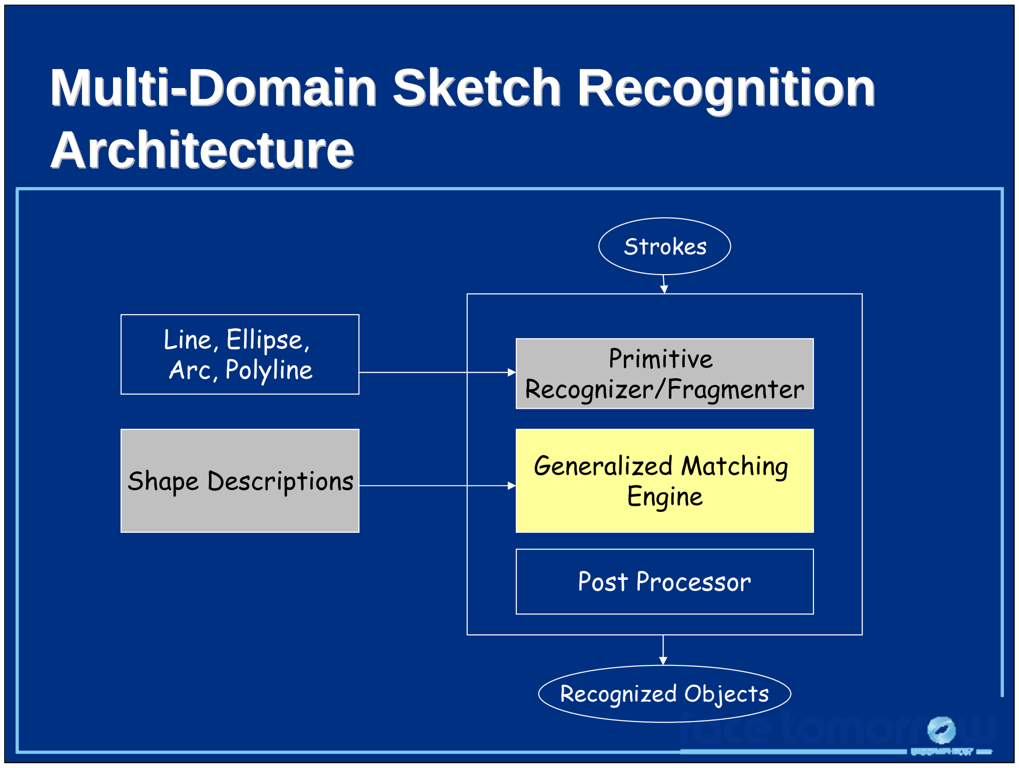


Knowledge Representation

```
(Define Child-link
 (Subshapes (Arrow a)))
```

```
(Define Current-Source
 (Subshapes (Arrow a)
            (Ellipse e))
 (Constraints
   (contains e a)))
```

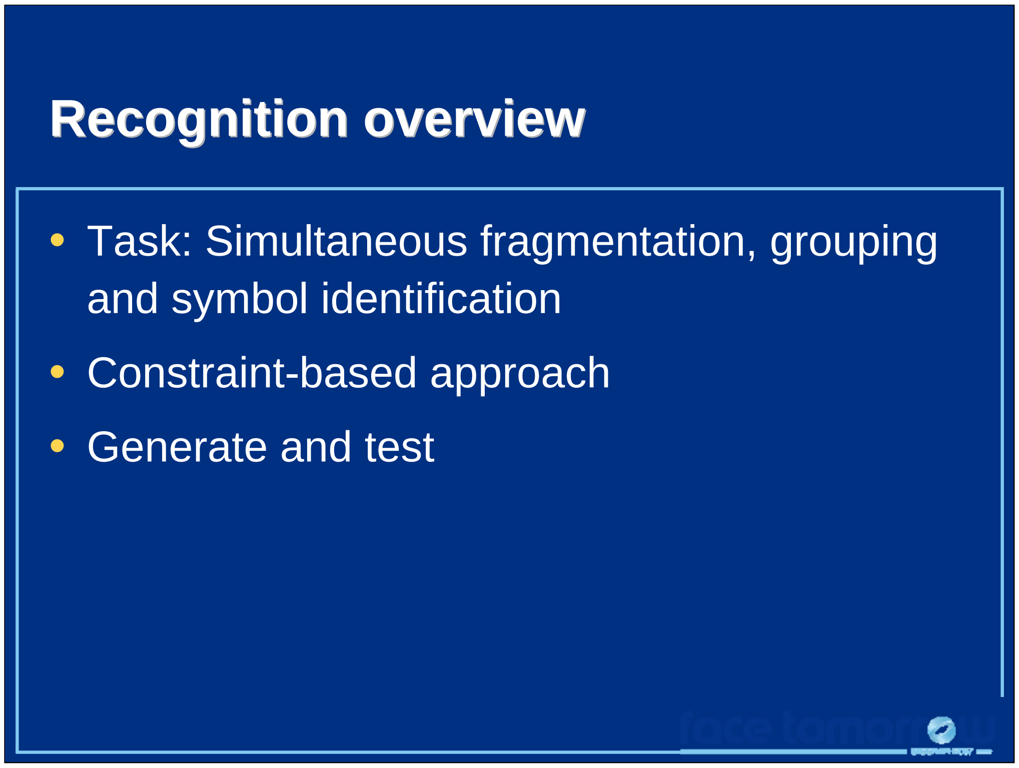
Once we have defined a shape, we can then use it in one or more higher level shape descriptions. In this example, we define a child-link, a domain shape (in the family tree domain), as simply an arrow. However, because we defined the arrow separately, we can also use it in the definition of a current source (in the analog circuit domain).



Multi-Domain Sketch Recognition Architecture

Strokes
Line, Ellipse, Arc, Polyline
Primitive Recognizer/Fragmenter
Shape Descriptions
Generalized Matching Engine
Post Processor
Recognized Objects

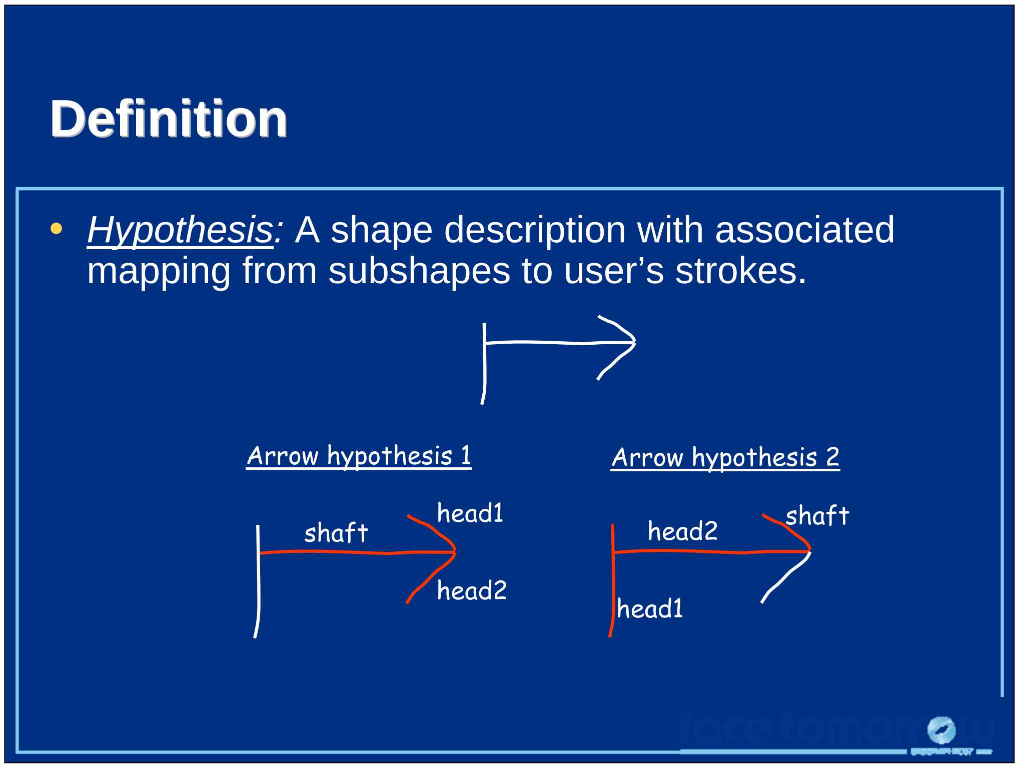
Our goal now is to develop a domain-independent algorithm to match these shape descriptions against the users strokes in order to recognize the sketch.



Recognition overview

- Task: Simultaneous fragmentation, grouping and symbol identification
- Constraint-based approach
- Generate and test

Here we give an overview of our approach to recognition. Again, "recognition" in this case means more than isolated shape recognition. We must simultaneously group the strokes into individual symbols and then recognize those symbols using the shape descriptions we just described. Building on our shape descriptions, we use a constraint-based approach to symbol identification where we first identify a symbol's subshapes and then verify that the necessary constraints between the subshapes hold. We deal with stroke grouping using a generate and test approach—we generate a number of plausible groups and test each by matching it against a particular shape description.

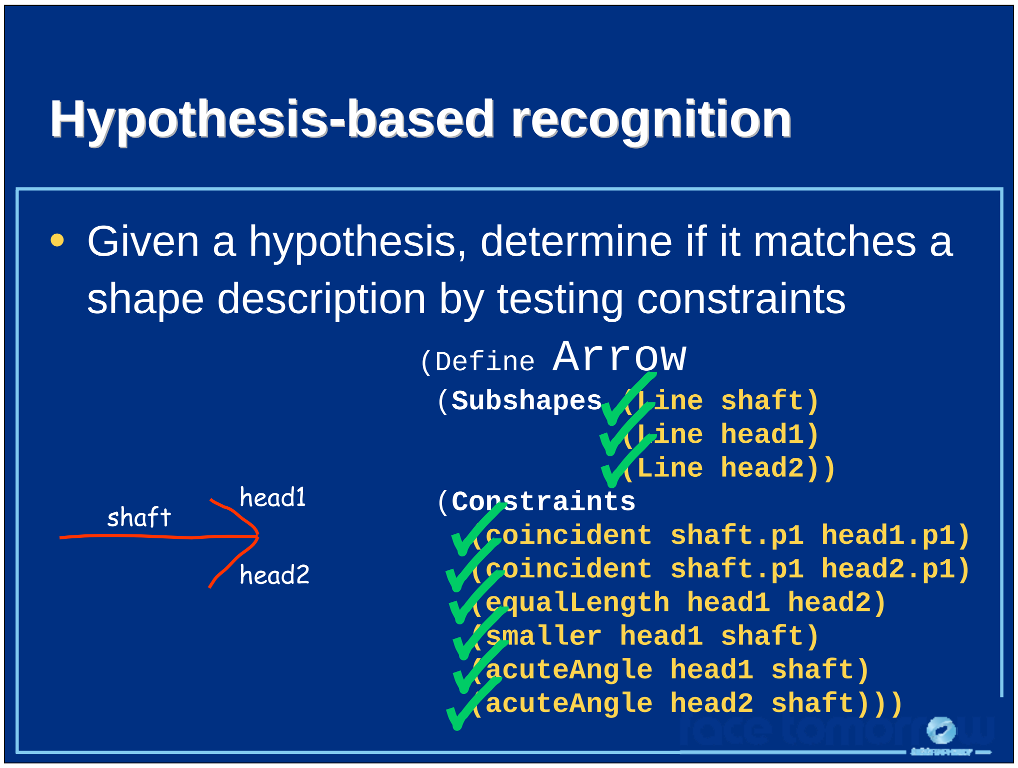


Definition

- *Hypothesis:* A shape description with associated mapping from subshapes to user's strokes.

Arrow hypothesis 1
shaft
head1
head2
Arrow hypothesis 2
head2
shaft
head1

The core piece of our generate-and-test recognition algorithm is the *hypothesis*. A hypothesis is a group of strokes with an associated mapping from subshapes to strokes. This slide shows two arrow hypotheses. Note that the process of generating hypotheses is a more constrained version of stroke grouping in that we must not only group the strokes but we must also assign each stroke a subshape in a shape description.



Hypothesis-based recognition

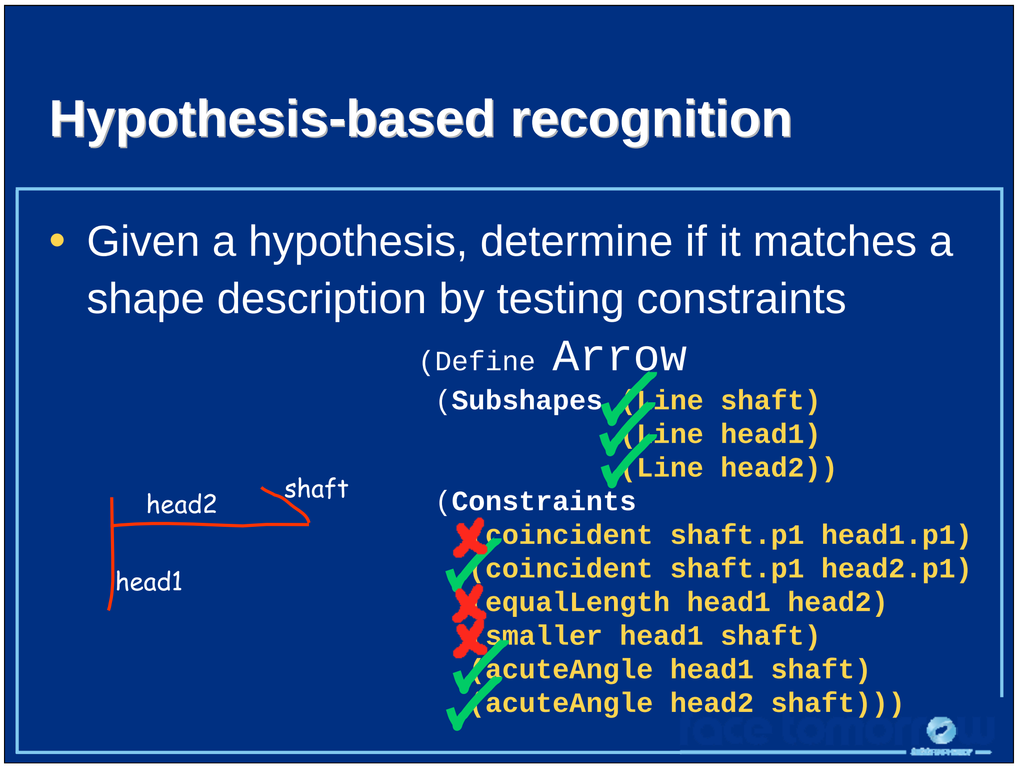
- Given a hypothesis, determine if it matches a shape description by testing constraints

```
(Define Arrow
 (Subshapes ✓(Line shaft)
            ✓(Line head1)
            ✓(Line head2))
 (Constraints
  ✓(coincident shaft.p1 head1.p1)
  ✓(coincident shaft.p1 head2.p1)
  ✓(equalLength head1 head2)
  ✓(smaller head1 shaft)
  ✓(acuteAngle head1 shaft)
  ✓(acuteAngle head2 shaft)))
```

shaft
head1
head2

Given a hypothesis, our goal is to determine how well it matches the shape description. We do this by verifying that the strokes match the subshape they have been assigned and that all of the constraints hold between the subshapes. In the example above we see that all three strokes look like lines and all of the constraints appear to hold.

I am being intentionally vague about what we mean by "a stroke matches a subshape" and "a constraint appears to hold". There are many possible ways we can verify that the shapes match and the constraints hold. LADDER's associated recognition system uses the least squared error to determine whether or not a stroke matches a primitive shape. If the error is below a threshold, the stroke matches the shape. Constraints are also measured using threshold values for corresponding features. For example, two points are considered coincident if they are below some distances away from one another, otherwise they are not. Shortly we will describe a problem with this approach and describe a way to address it.



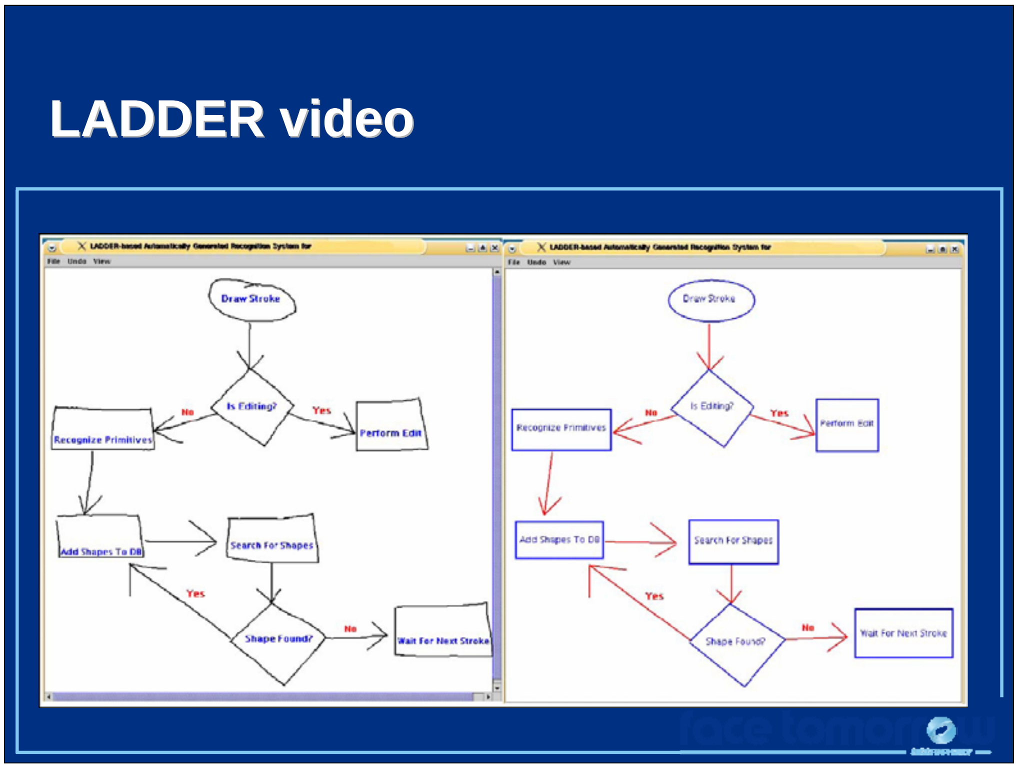
Hypothesis-based recognition

- Given a hypothesis, determine if it matches a shape description by testing constraints

```
(Define Arrow
 (Subshapes ✓(Line shaft)
            ✓(Line head1)
            ✓(Line head2))
 (Constraints
  ✗(coincident shaft.p1 head1.p1)
  ✓(coincident shaft.p1 head2.p1)
  ✗(equalLength head1 head2)
  ✗(smaller head1 shaft)
  ✓(acuteAngle head1 shaft)
  ✓(acuteAngle head2 shaft)))
```

head2
shaft
head1

This hypothesis does not match because a number of the necessary constraints do not hold. For example, head1 is not smaller than the shaft.



LADDER video

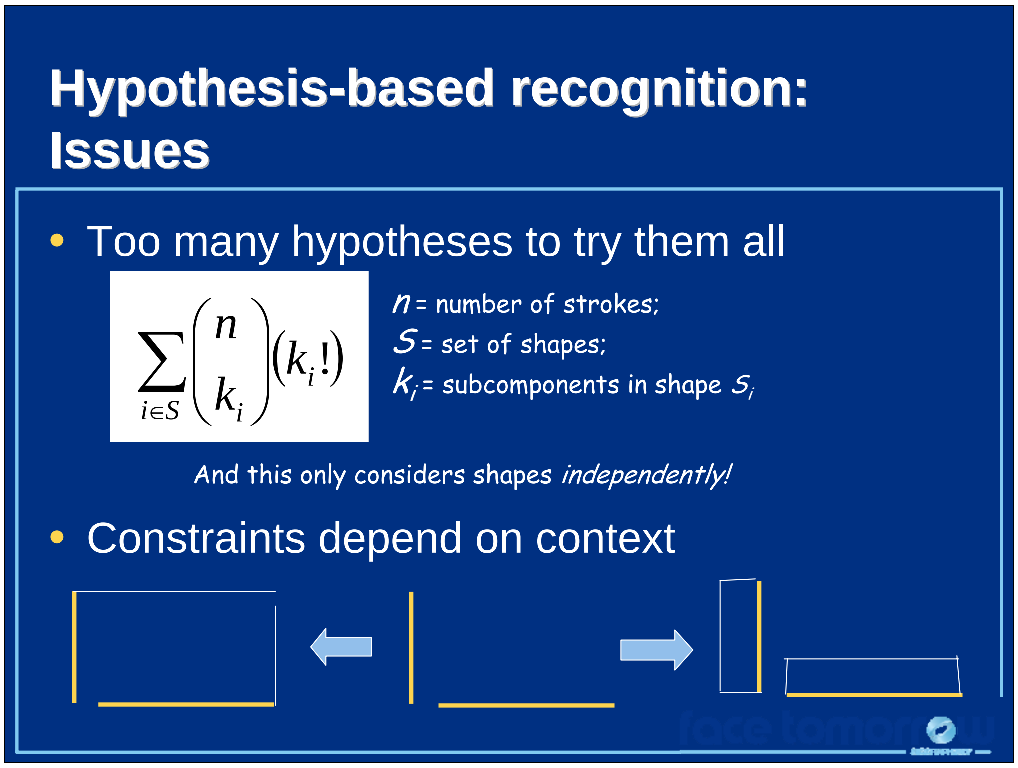
Hammond has developed a suite of tools to support the LADDER sketch description language, including a constraint-based recognition engine of the type I just described. LADDER is a complete multi-domain sketch recognition system, although it places some restrictions on the way users can drawn shapes. LADDER uses stroke order to assist in stroke grouping and hypothesis generation (i.e., shapes must consist of consecutive strokes).

The following papers provide more information on all of the components associated with LADDER:

Hammond, Tracy and Davis, Randall (2005). **LADDER, a sketching language for user interface developers** Elsevier, Computers and Graphics 29 (2005) 518-532,

Hammond, Tracy and Davis, Randall (2004). **Automatically Transforming Symbolic Shape Descriptions for Use in Sketch Recognition** The Nineteenth National Conference on Artifical Intelligence (AAAI-04), July 2004

Hammond, Tracy and Davis, Randall (2004). **Shady: A Shape Description Debugger for Use in Sketch Recognition** Proceedings of the 2004 AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural, October, 2004.



Hypothesis-based recognition: Issues

- Too many hypotheses to try them all

$$\sum_{i \in S} \binom{n}{k_i} (k_i!)$$

n = number of strokes;
S = set of shapes;
k_i = subcomponents in shape S_i

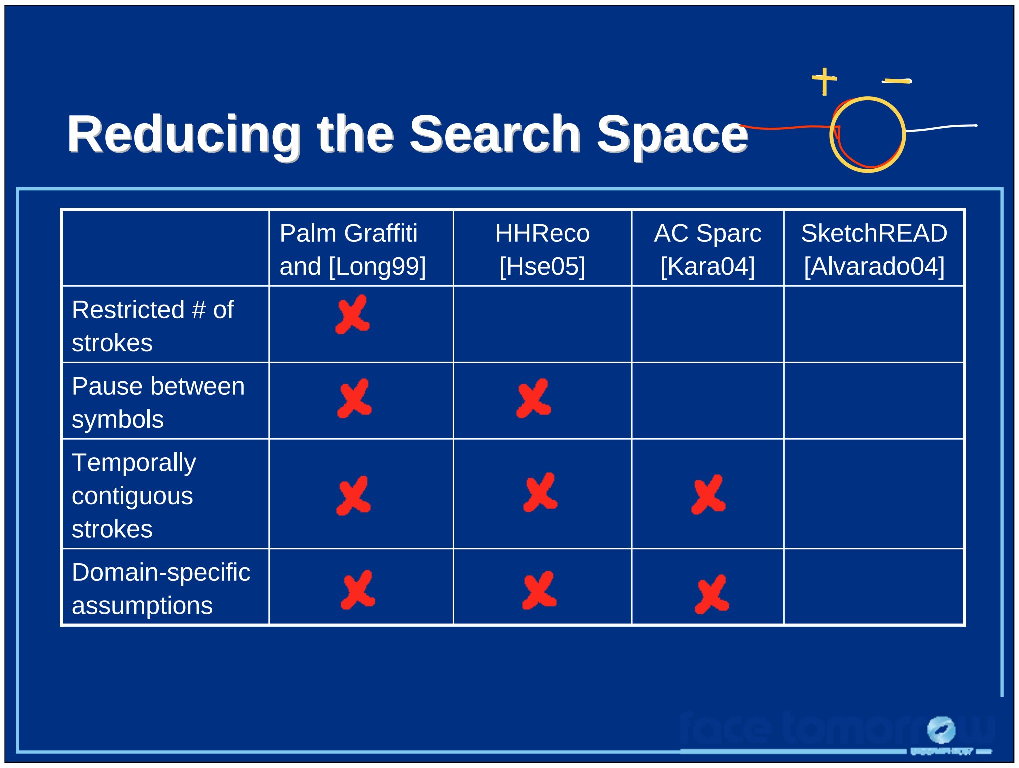
And this only considers shapes *independently!*

- Constraints depend on context

One of the challenges in hypothesis-based recognition is that there are simply too many hypotheses to try them all, so we risk missing the correct hypothesis because we did not have time to generate it. Considering each shape independently, we need to try all combinations of strokes (n choose k), and then for each combination, we need to consider all possible mapping of strokes to subcomponents (k!). Many systems, including LADDER place restrictions on the way the user may draw (e.g., strokes in a single shape must be temporally contiguous), but we have found that these constraints often do not match the way users actually draw.

A second problem with the approach as we have described it so far is that often we cannot tell whether or not a constraint holds until we see the context in which it appears. For example, the user may have intended for the two lines at the bottom of the slide to touch (as when they are part of a single box, on the left) or not to touch (as when they are part of two separate boxes, on the right).

The approach I will now present addresses both of these problems.



Reducing the Search Space

	Palm Graffiti and [Long99]	HHReco [Hse05]	AC Sparc [Kara04]	SketchREAD [Alvarado04]
Restricted # of strokes	✗			
Pause between symbols	✗	✗		
Temporally contiguous strokes	✗	✗	✗	
Domain-specific assumptions	✗	✗	✗	

Many systems, place restrictions on the way the user may draw in order to reduce the number of stroke groups a the recognition system must consider. Four common assumptions are:

1. A shape can be drawn only with a fixed number of strokes (often one). This restriction is made by many graffiti-like systems.
2. The user must draw shapes with temporally contiguous strokes, and she must pause between symbols. With this restriction, the system can reliably detect groups using a simple temporal threshold.
3. The user must draw shapes with temporally contiguous strokes, but need not pause between shapes. This restriction reduces the grouping task to a one-dimensional search problem (forward and backward in time).
4. The system relies on domain specific assumptions. For example, the SimuSketch system for recognizing network flow diagrams we looked at this morning exploited the fact that arrows separated other components in space.

While these restrictions aid recognition, we find that they often do not match the way users draw in practice. The approach I will discuss, used in the SketchREAD system, places none of the standard constraints on the user's drawing style.

References:

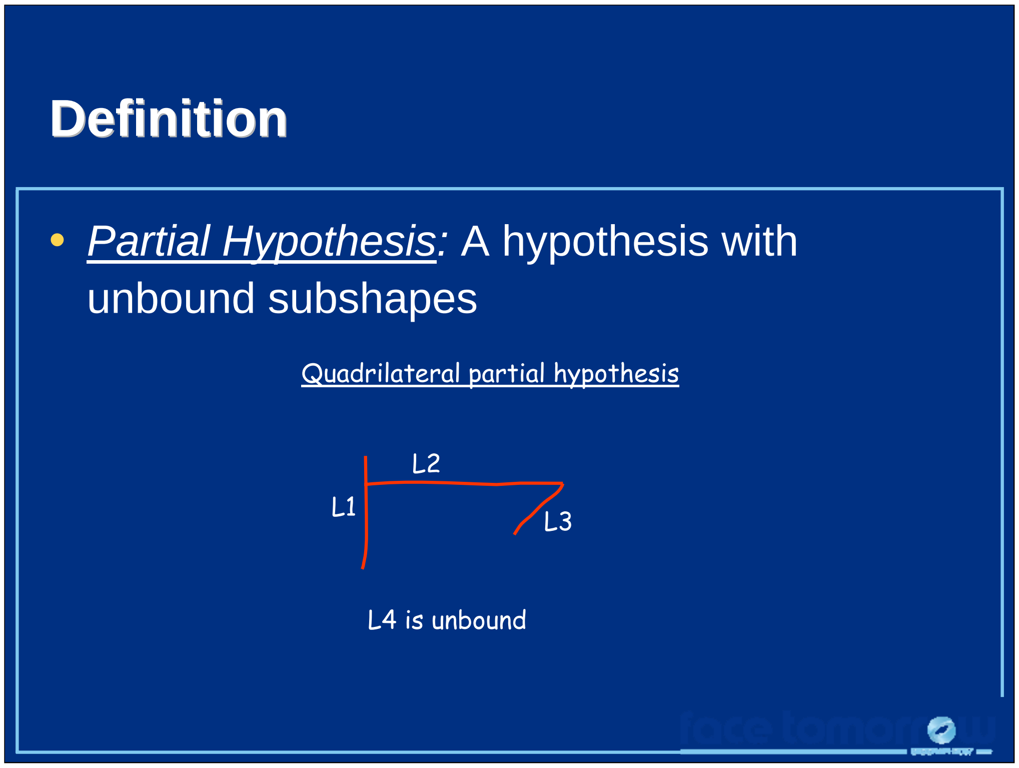
Christine Alvarado and Michael Lazzareschi. **Properties of Real World Digital Logic Diagrams.** *1st International Workshop on Pen-based Learning Technologies (PLT*) 2007.

Alvarado, Christine; and Davis, Randall. **SketchREAD: A Multi-domain Sketch Recognition Engine.** Proceedings of UIST 2004.

Leslie Gennari, Levent Burak Kara, Thomas F. Stahovich (2005) Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams.
Computers & Graphics 29(4): 547-562 2005.

H. Hse and A. R. Newton. Recognition and beautification of multi-stroke symbols in digital ink. Computers and Graphics, 2005.

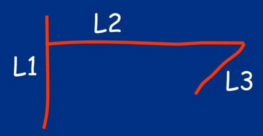
Allan Christian Long, Jr., James A. Landay, Lawrence A. Rowe. "Implications for a Gesture Design Tool." *CHI 99 Proceedings*, May 15-20, 1999.



Definition

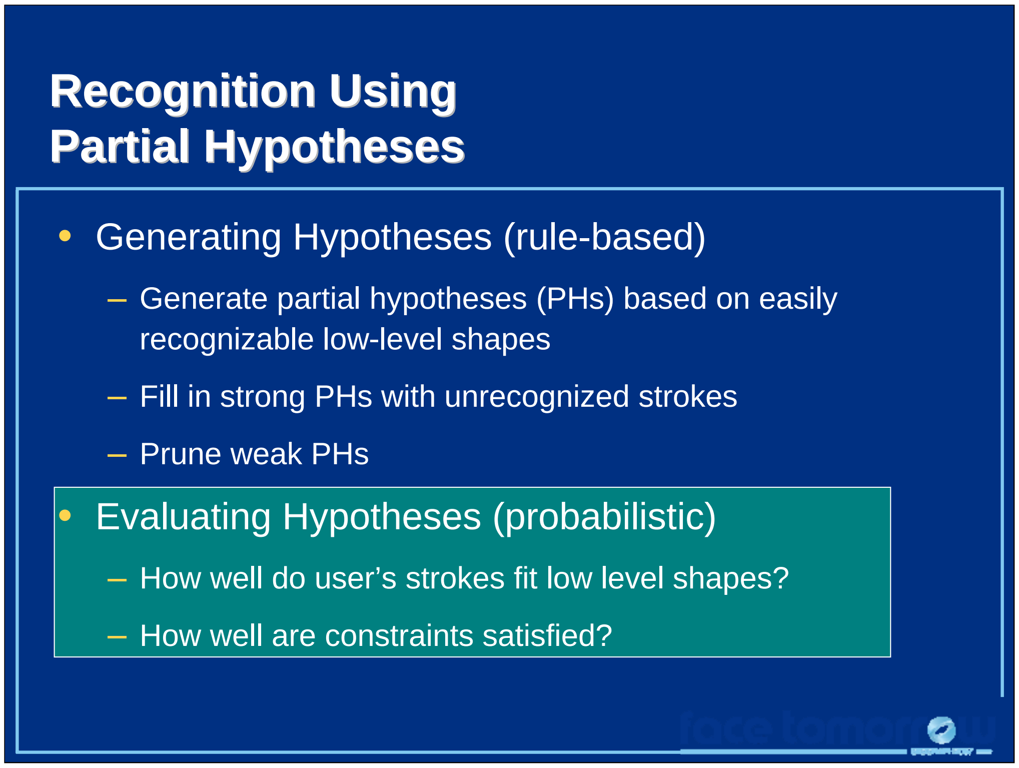
- *Partial Hypothesis:* A hypothesis with unbound subshapes

Quadrilateral partial hypothesis



L4 is unbound

The key to our approach is that we will use promising partially recognized shapes to guide the search for correct stroke groupings. A *partial hypothesis* is simply a hypothesis with unbound subshapes.



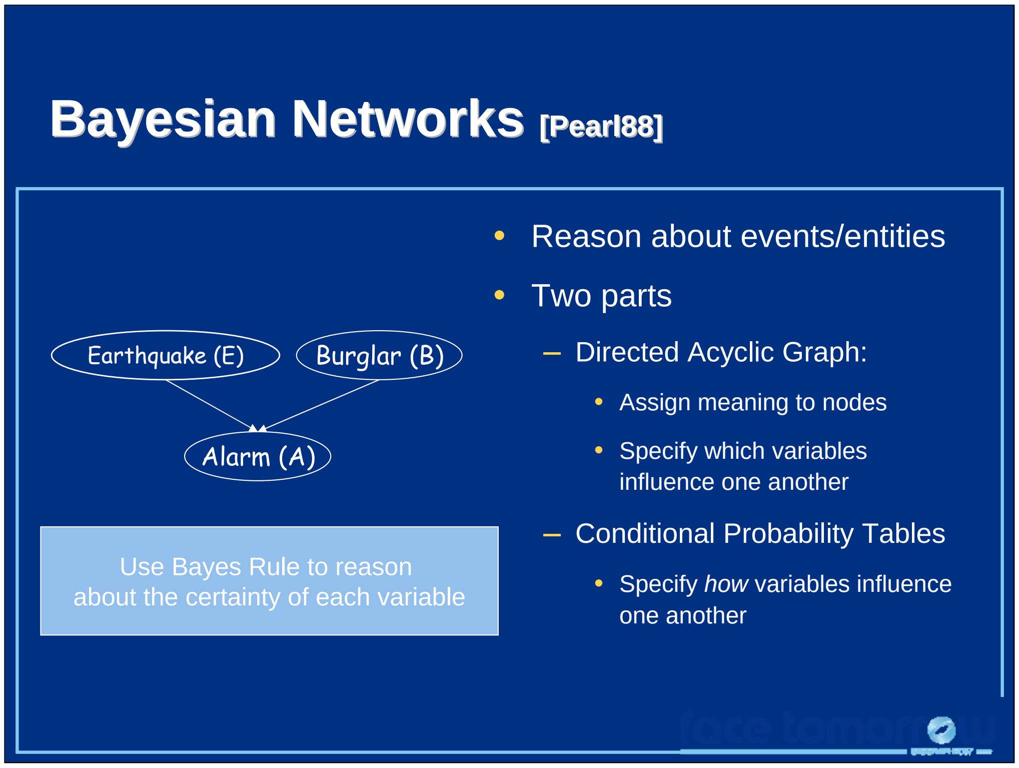
Recognition Using Partial Hypotheses

- Generating Hypotheses (rule-based)
  - Generate partial hypotheses (PHs) based on easily recognizable low-level shapes
  - Fill in strong PHs with unrecognized strokes
  - Prune weak PHs
- Evaluating Hypotheses (probabilistic)
  - How well do user’s strokes fit low level shapes?
  - How well are constraints satisfied?

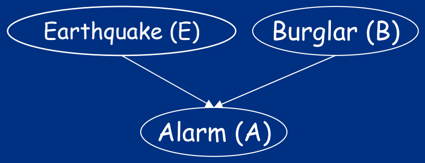
The problem with stroke grouping is the there are too many possible groupings to try them all, but most symbol recognition algorithms are not capable of matching part of a shape. If symbol recognition fails, the system has gained no information about how it should modify the set of strokes—was the symbol almost recognized? Did any part of the symbol look about right? The stroke grouping process has no choice but to blindly try another group of strokes. If instead we could recognize part of a symbol, we could use that information to narrow the set of stroke groups to try. If symbol recognition fails, instead of throwing away the whole group of strokes we could swap out only those strokes that we knew did not match the symbol and keep those strokes that did.

We call this process Recognition Using Partial Hypotheses. It is a two-stage generate and test approach that uses the probabilistic evaluation of partial hypotheses (PHs) to guide the search for correct hypotheses. In the hypothesis generation step, the system generates a number of PHs based on easily recognizable low-level shapes. It fills in strokes PHs with unrecognized strokes and discards weak PHs.

Of course, to determine which hypotheses to keep and which to discard we need to be able to evaluate them. We used a probabilistic model that measures how well a user's strokes fit low level shapes and how well constraints are satisfied *in the context of the shape being considered*. I will begin by discussing hypothesis evaluation and then discuss hypothesis generation.



Bayesian Networks [Pearl88]



Use Bayes Rule to reason about the certainty of each variable

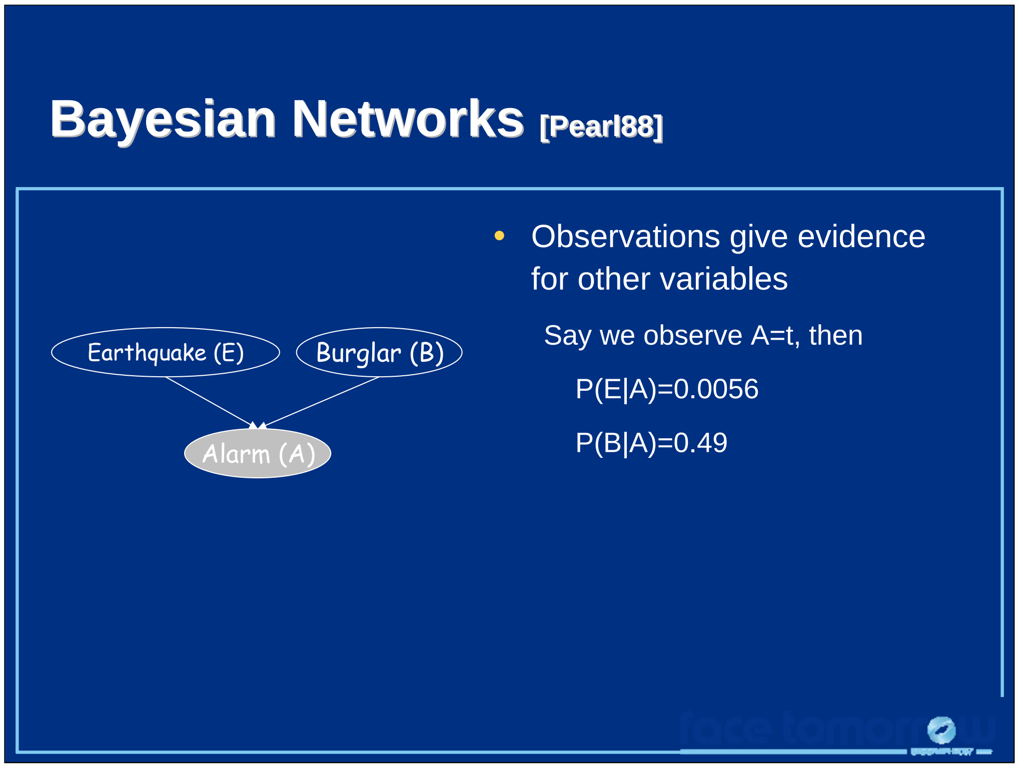
- Reason about events/entities
- Two parts
  - Directed Acyclic Graph:
    - Assign meaning to nodes
    - Specify which variables influence one another
  - Conditional Probability Tables
    - Specify *how* variables influence one another

We use a Bayesian network framework to evaluate partial hypotheses. Briefly, a Bayesian network (or Bayes net) is a probabilistic model for reasoning about events or entities in the world. This slide shows a Bayesian network for reasoning about whether or not a burglar broke into your home from [Pearl88]. Imagine that you live in Southern California, where earthquakes and crime both occur on occasion. Hoping to avoid the latter, you install an alarm system in your home. Unfortunately, while a burglar is likely to trigger the alarm, so is an earthquake. You receive a phone call telling you with absolute certainty that your alarm is going off. You want to know how likely it is that you have been robbed.

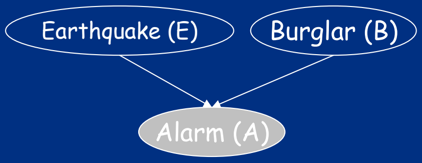
We can model this problem using a Bayesian network. A Bayes net consists of two parts: a directed acyclic graph (DAG) and a set of conditional probability tables (CPTs). In the DAG, nodes represent random variables corresponding to entities in the world, and edges represent direct (causal) relationships between the variables. In this example, because both the earthquake and the burglar might set off the alarm, both of these nodes are directly connected to the Alarm node in the network. There is one CPT for each node, specifying how it is influenced (probabilistically) by its parents. I have not given the CPTs for this example.

References:

Pearl, J. 1988. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. San Mateo, CA: Morgan Kaufman Publishers.



Bayesian Networks [Pearl88]



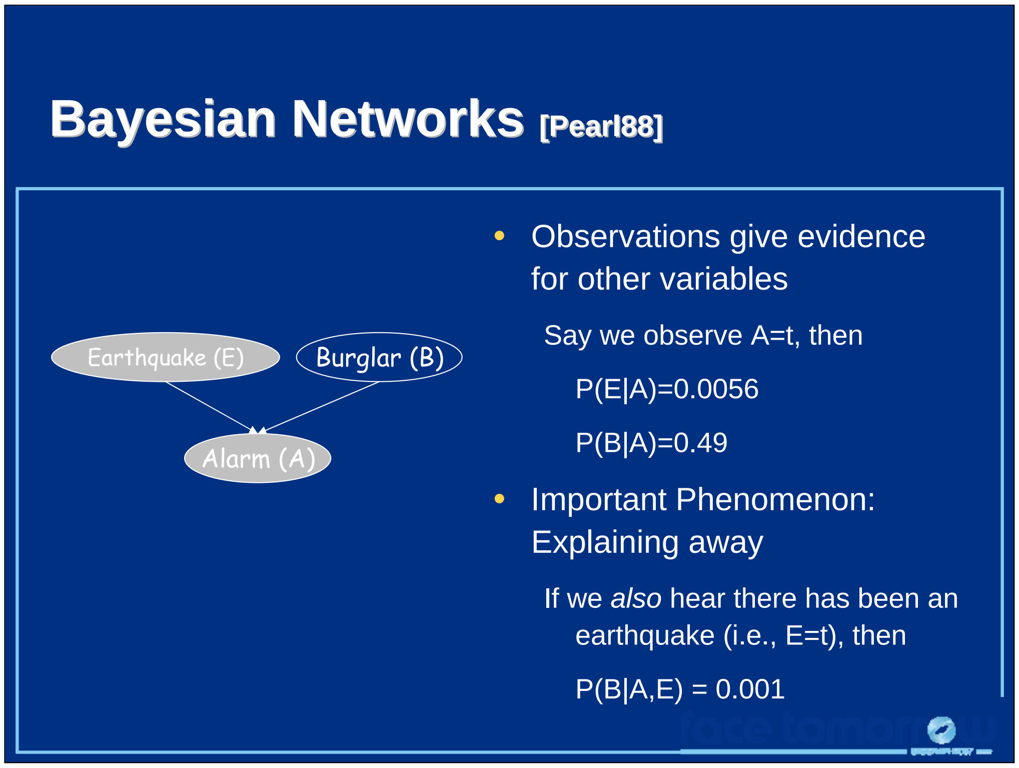
- Observations give evidence for other variables

  Say we observe A=t, then

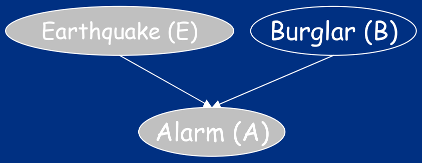
  P(E|A)=0.0056

  P(B|A)=0.49

When we observe the values for some of the variables in the network, we can use Bayes rule to reason about other variables. For example, in this case, when observe that the alarm is sounding (or receive the phone call assuring us that it is), our belief that there was a burglary becomes 0.49, while our belief that there was an earthquake becomes 0.0056.



Bayesian Networks [Pearl88]



- Observations give evidence for other variables

  Say we observe A=t, then

  P(E|A)=0.0056

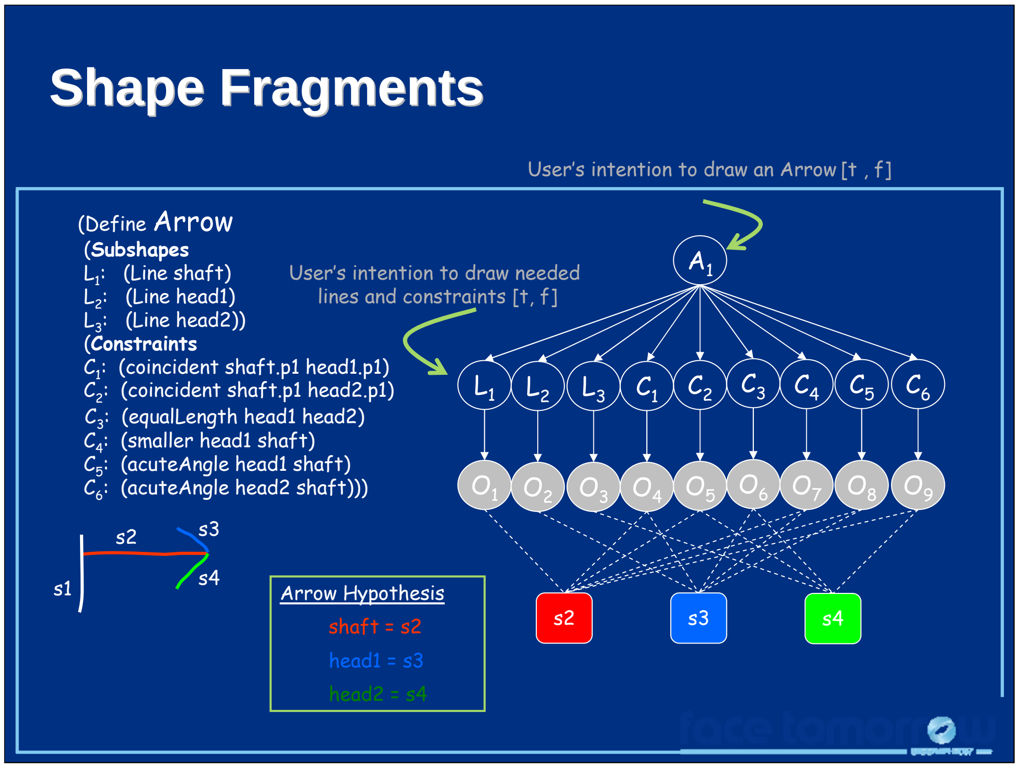
  P(B|A)=0.49

- Important Phenomenon: Explaining away

  If we *also* hear there has been an earthquake (i.e., E=t), then

  P(B|A,E) = 0.001

One important phenomenon that occurs in Bayes nets is that certain information *explains away* other information. For example, if we now hear on the radio that there was a very small earthquake near our home, our belief that we were robbed drops to 0.001. The earthquake provides a plausible explanation for why the alarm was sounding and there is no need for us to believe that a burglar broke in.



Shape Fragments

(Define Arrow
(Subshapes
L_1: (Line shaft)
L_2: (Line head1)
L_3: (Line head2))
(Constraints
C_1: (coincident shaft.p1 head1.p1)
C_2: (coincident shaft.p1 head2.p1)
C_3: (equalLength head1 head2)
C_4: (smaller head1 shaft)
C_5: (acuteAngle head1 shaft)
C_6: (acuteAngle head2 shaft)))

Arrow Hypothesis
shaft = s2
head1 = s3
head2 = s4

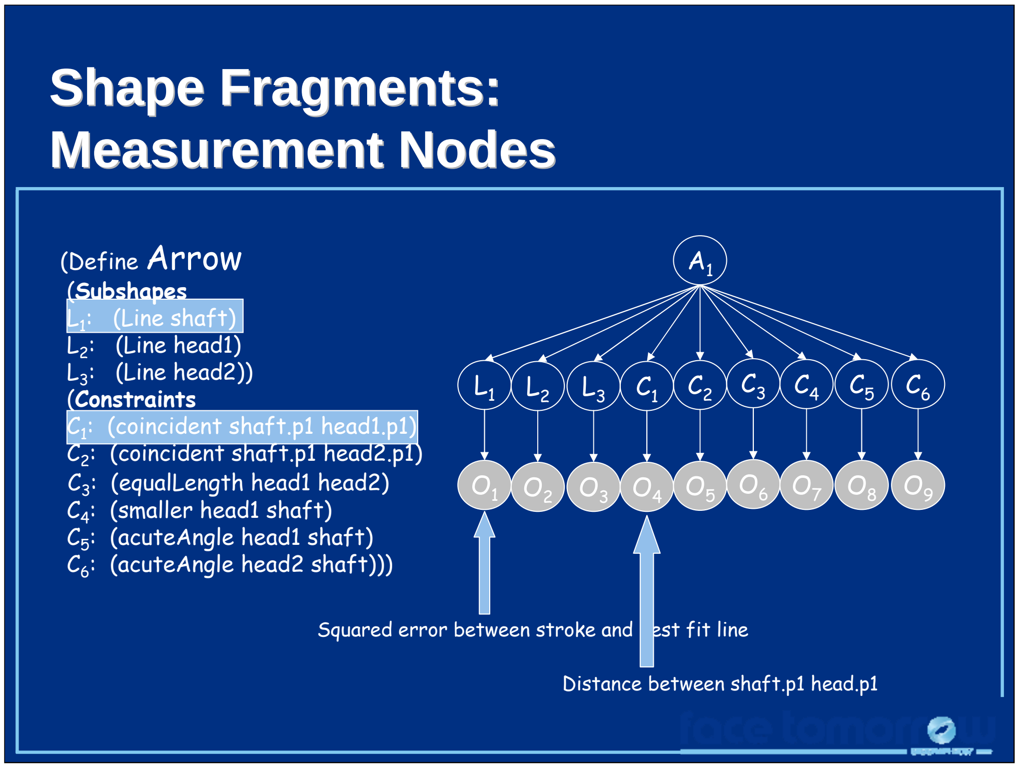
User's intention to draw an Arrow [t , f]
User's intention to draw needed lines and constraints [t, f]
A_1
L_1 L_2 L_3 C_1 C_2 C_3 C_4 C_5 C_6
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9
s2 s3 s4
s1 s2 s3 s4

With this background, we can now explain how we use Bayes nets to evaluate shape hypotheses and partial hypotheses. For each hypothesis, we automatically construct a Bayes net the encodes a the structural description of the symbol. The network has one node for the symbol to be recognized and one node for each of the subshapes and constraints in that symbol. These nodes represent Boolean variables reflecting the user's *intention* to draw a particular shape on constraint. For example, the value A1=true means that the user was trying to draw an arrow with strokes s2, s3, and s4.

There are directed edges between the node representing the high-level symbol and the nodes for each subshape and constraint that comprise that symbol. The edges represent the causal connection between the user's intention to draw the high level shape and her intention to draw the subshapes and constraints: if the user intends to draw an arrow, then she also likely intends to draw three lines, two of which are about the same length and one that is longer, etc, i.e. the lines that make up the arrow.

Note that given the user's intention to draw the high-level shape, her intentions to draw the low-level shapes and constraints are conditionally independent. For example, if we *know* the user is drawing an arrow, then being told that she intends for s3 and s4 to be the same length does not change our belief that she intends for s2 to be longer than s3.

As we noted earlier, we cannot observe even the low-level shapes and constraints directly. Stroke data is noisy; the low-level interpretation of a stroke and whether or not a constraint holds between two strokes depend on the context in which those strokes appear. What we can observe are stroke features that provide strong clues about the low-level interpretation or whether or not a constraint holds.



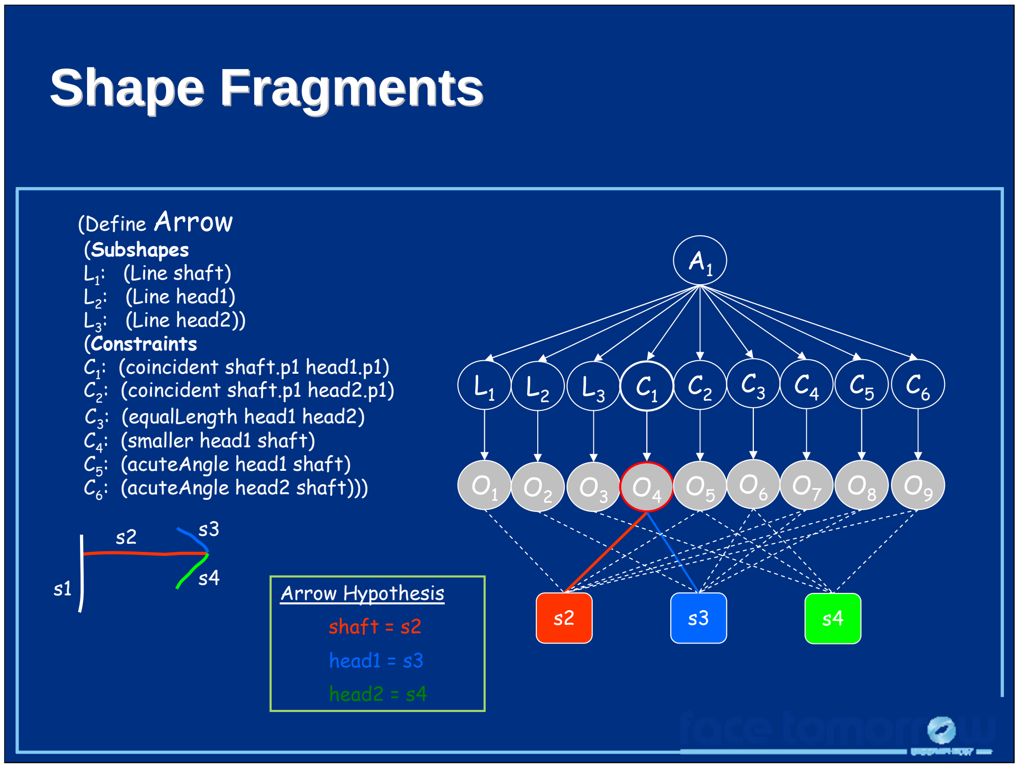
Shape Fragments: Measurement Nodes

(Define Arrow
(Subshapes
L_1: (Line shaft)
L_2: (Line head1)
L_3: (Line head2))
(Constraints
C_1: (coincident shaft.p1 head1.p1)
C_2: (coincident shaft.p1 head2.p1)
C_3: (equalLength head1 head2)
C_4: (smaller head1 shaft)
C_5: (acuteAngle head1 shaft)
C_6: (acuteAngle head2 shaft)))

A_1
L_1 L_2 L_3 C_1 C_2 C_3 C_4 C_5 C_6
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9
Squared error between stroke and best fit line
Distance between shaft.p1 head.p1

Each node in the bottom layer of the network (called an *observation* or *feature* node) represents a continuous-valued variable (which we discretize in practice) representing a specific stroke feature (or set of features) that provides information about its parent shape or constraint. For example, node O_1 is a variable whose value is the squared error between the stroke corresponding to the shaft of the arrow (stroke s2 in this hypothesis) and the best fit line to this stroke. Note that the value of this variable is strongly influenced by whether or not the user actually intended for s2 to be a line: if she did, we expect this error to be low (but probably not zero), if she did not, we expect the error to be higher. Similarly, O_4 is a variable whose value is the distance between point p1 in the stroke mapped to the shaft of the arrow (stroke s2) and point p1 in the stroke mapped to head1 (stroke s3). The value of this variable is influenced by whether or not the user intended for these two strokes to connect.

We can now see how this network allows us to evaluate primitive shapes and constraints using the surrounding context. For example, the value of C_1 (the hypothesis that the user intended for strokes s2 and s3 to connect) depends both on how close together the strokes' endpoints are (the data) and one how strongly we believe that the user intends to draw an arrow (the context), which in turn is influenced by how well the primitive shapes fit and the other constraints hold. There is no need to define a threshold for each constraint. The network seamlessly blends data and context information.



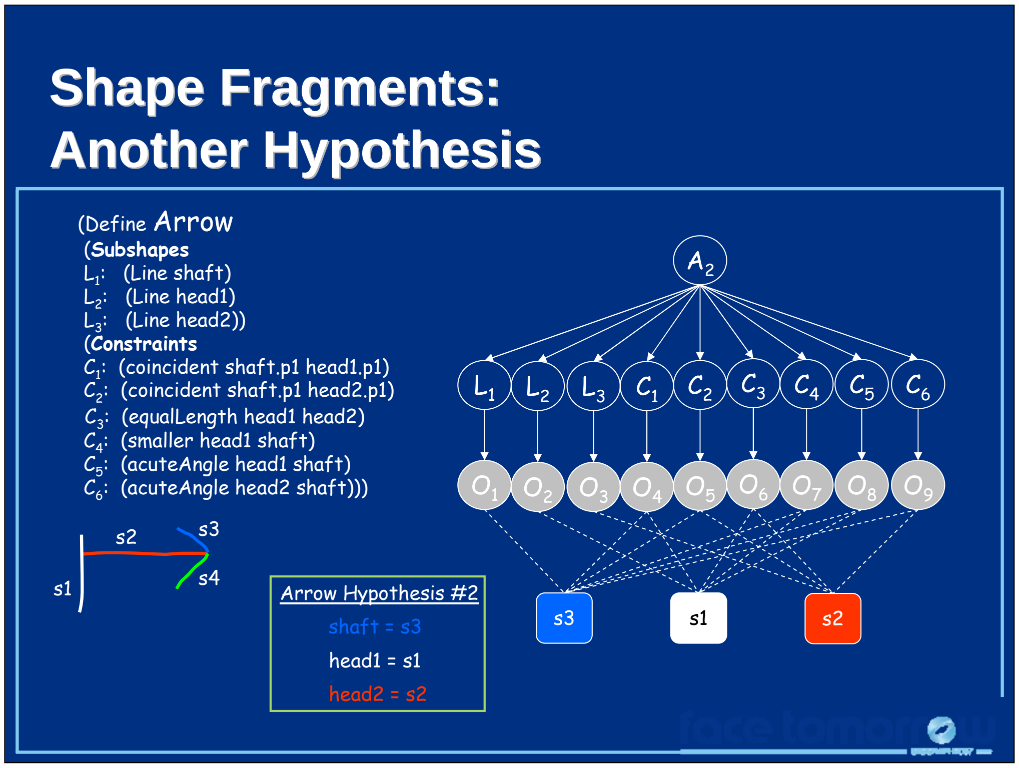
Shape Fragments

```
(Define Arrow
 (Subshapes
 L1:  (Line shaft)
 L2:  (Line head1)
 L3:  (Line head2))
 (Constraints
 C1: (coincident shaft.p1 head1.p1)
 C2: (coincident shaft.p1 head2.p1)
 C3: (equalLength head1 head2)
 C4: (smaller head1 shaft)
 C5: (acuteAngle head1 shaft)
 C6: (acuteAngle head2 shaft)))
```

Arrow Hypothesis
shaft = s2
head1 = s3
head2 = s4

s1
s2
s3
s4
A_1
L_1 L_2 L_3 C_1 C_2 C_3 C_4 C_5 C_6
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9
s2
s3
s4

It is important to realize that this network evaluates exactly one hypothesis because each observation node corresponds to exactly one stroke or set of strokes. The colored boxes at the bottom of the slide are not part of the network, but visually indicate which strokes correspond to which observation nodes. For example, O_1 represents a feature of stroke s2, while O_4 represents a feature of strokes s2 and s3.



Shape Fragments: Another Hypothesis

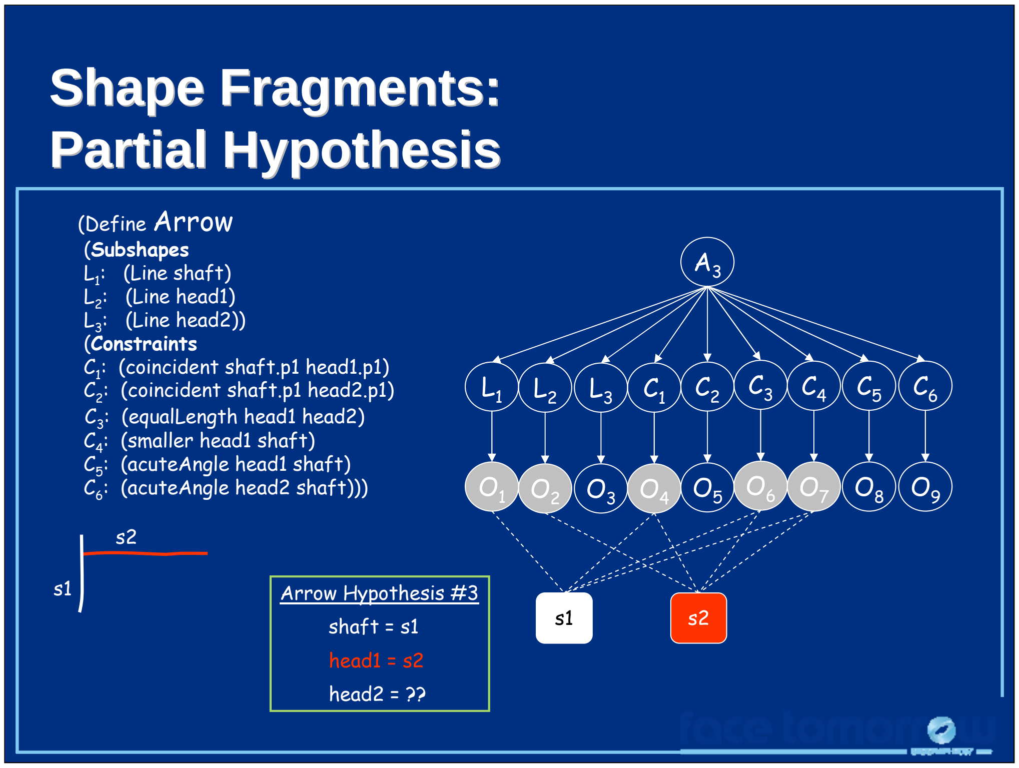
(Define Arrow
(Subshapes
L_1: (Line shaft)
L_2: (Line head1)
L_3: (Line head2))
(Constraints
C_1: (coincident shaft.p1 head1.p1)
C_2: (coincident shaft.p1 head2.p1)
C_3: (equalLength head1 head2)
C_4: (smaller head1 shaft)
C_5: (acuteAngle head1 shaft)
C_6: (acuteAngle head2 shaft)))

Arrow Hypothesis #2
shaft = s3
head1 = s1
head2 = s2

A_2
L_1 L_2 L_3 C_1 C_2 C_3 C_4 C_5 C_6
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9
s3
s1
s2
s1
s2
s3
s4

With this framework, it is simple to evaluate other arrow hypotheses. We simply create an identical network with a different mapping between observation nodes and strokes.

Note, too, that we can automatically construct networks for any shape described using the LADDER description language, so this framework is entirely domain-flexible.



Shape Fragments: Partial Hypothesis

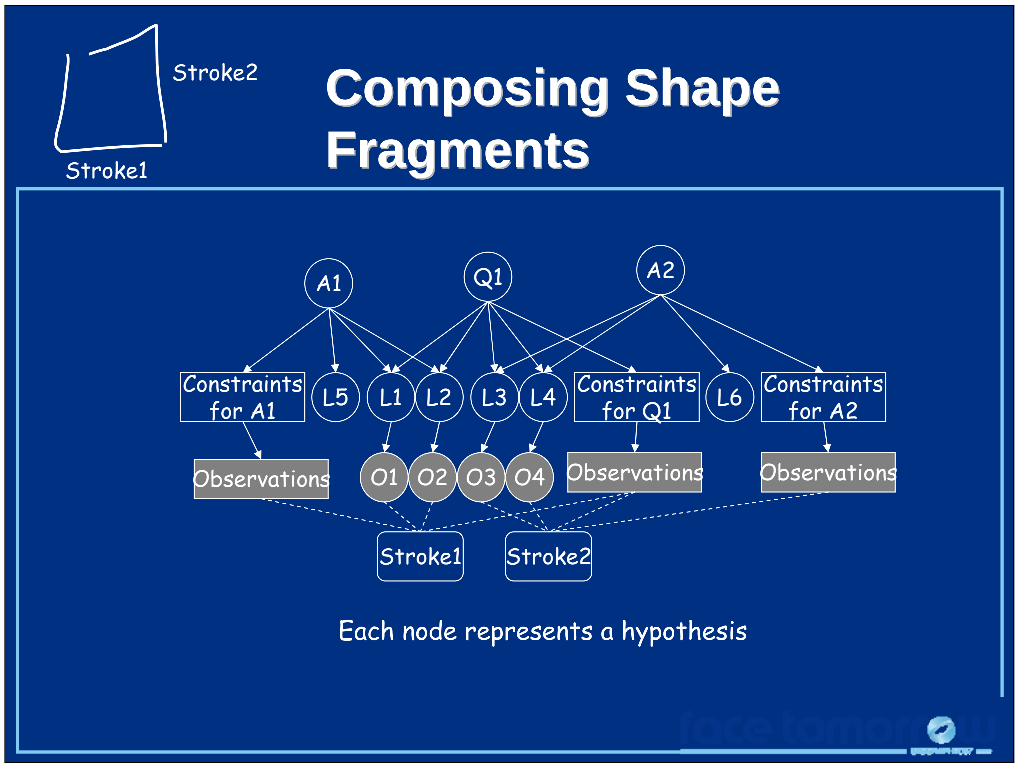
(Define Arrow
(Subshapes
L_1: (Line shaft)
L_2: (Line head1)
L_3: (Line head2))
(Constraints
C_1: (coincident shaft.p1 head1.p1)
C_2: (coincident shaft.p1 head2.p1)
C_3: (equalLength head1 head2)
C_4: (smaller head1 shaft)
C_5: (acuteAngle head1 shaft)
C_6: (acuteAngle head2 shaft)))

Arrow Hypothesis #3
shaft = s1
head1 = s2
head2 = ??

A_3
L_1 L_2 L_3 C_1 C_2 C_3 C_4 C_5 C_6
O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9
s1
s2
s1
s2

Evaluating partial hypotheses is also straightforward using this framework. To evaluate a partial hypothesis we create the same network, but we will be unable to observe values for one or more of the observation nodes.

All this means is that we will have less data to support the hypothesis that the user intended to draw a particular arrow, but the missing data will not significantly detract from our belief if all of the other data supports the hypothesis. In essence, this network encodes how likely it is that the user *intends* to draw the arrow (and all of the pieces involved) but just hasn't gotten around to drawing the last piece. We still may believe, based on the context of the other strokes, that the user intends for that final piece of the arrow to exist, even though we don't see it. This idea will be critical in helping us guide our search through the hypothesis space.



Composing Shape Fragments

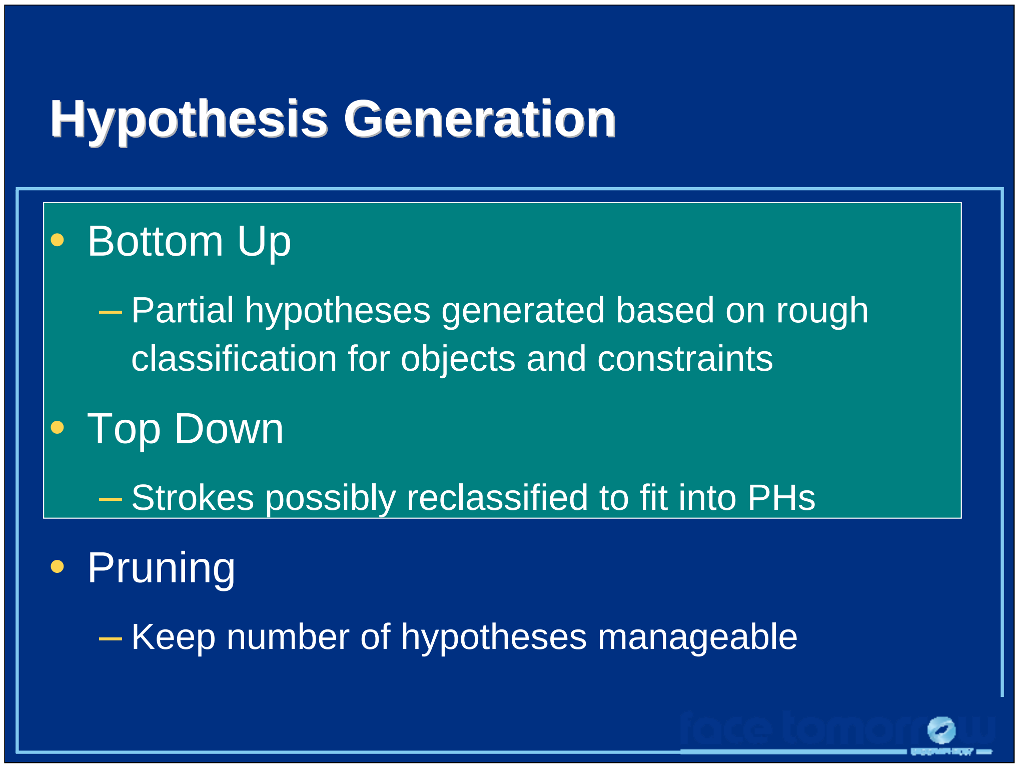
Each node represents a hypothesis

So far, we have looked at how to evaluate a single hypothesis, but our goal during sketch recognition is to evaluate multiple, competing hypotheses. In this very simple example we examine how to automatically construct a network to evaluate multiple hypotheses. Throughout this example we assume the existence of a separate hypothesis generation process (which we will examine shortly). Here we are concerned only with how to evaluate these hypotheses.

The user begins by drawing stroke 1 (in the upper left corner of the slide). The low-level recognizer recognizes this stroke as two lines and generates two line hypotheses. The system detects that these lines might be part of an arrow or they might be part of a quadrilateral, and it generates two competing partial hypotheses. These hypotheses share the same primitive shape nodes in the network; i.e., they are competing explanations for why the user is drawing two connected lines.

When the user draws stroke 2, the low level recognizer again detects two possible lines, and the system adds these two new line hypotheses to the network. The hypothesis generator notes that these new strokes appear to fit into the existing quadrilateral hypothesis, and the updated hypothesis is reflected by connecting stroke 2 to observation nodes for L3 and L4 in the Bayes net. However, L3 and L4 do not immediately appear to fit into the existing arrow hypothesis, so the system creates a new arrow hypothesis for these two strokes (A2).

Now, determining the strength of each hypothesis is a simple matter of Bayesian inference. Note, too, that the arrow and quadrilateral hypotheses are not independent—the more we believe the in Q1, the less we will believe in A1 and A2 (and vice versa) because of the explaining away phenomenon we observed earlier. This behavior is desirable for a sketch recognition system (because, usually, strokes do not represent two separate shapes simultaneously) and is a natural consequence of the Bayesian network model.

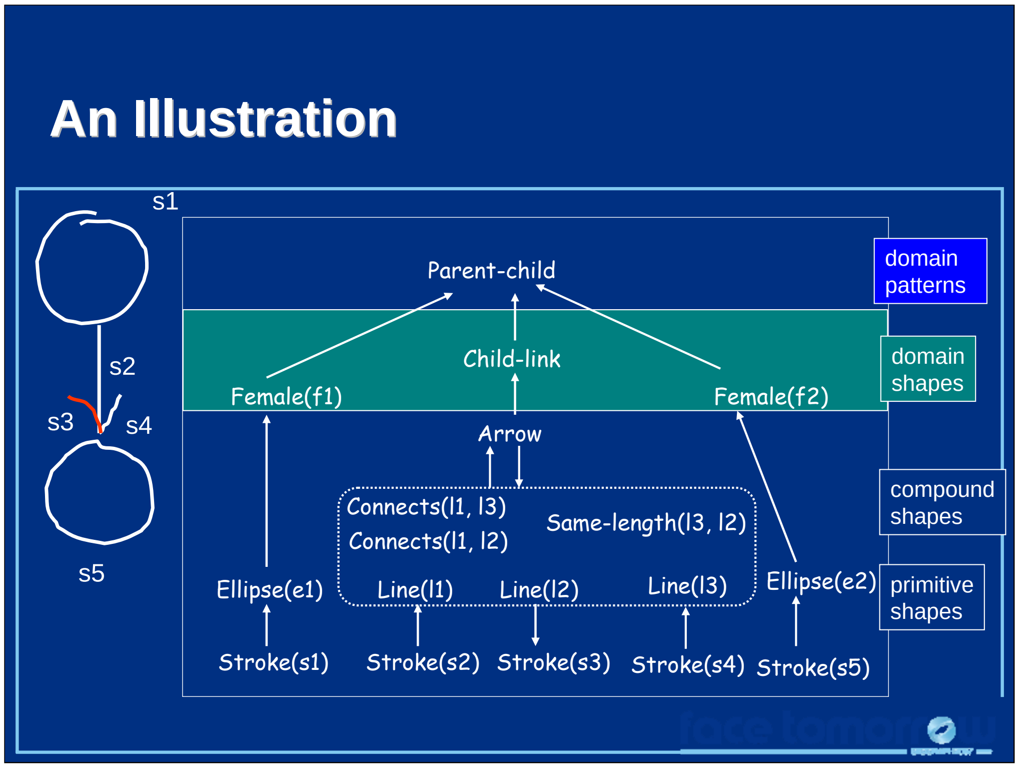


Hypothesis Generation

- Bottom Up
  - Partial hypotheses generated based on rough classification for objects and constraints
- Top Down
  - Strokes possibly reclassified to fit into PHs
- Pruning
  - Keep number of hypotheses manageable

Now that we have a method for evaluating hypotheses, let's look at how to generate them. After the user draws each stroke, we generate and prune hypotheses using a three stage iterative process. Each stage is guided by the relative strength of the partial hypotheses generated in previous stages. First, we use rough classification and constraint thresholds to generate partial hypotheses. Then we refine these hypotheses by searching for strokes that may have been misrecognized that could complete probable partial hypotheses. Finally, we prune weak hypotheses to keep the search space manageable. We repeat these steps until no new hypotheses are generated (taking care not to regenerate pruned hypotheses).

In this talk I will focus only on hypothesis generation.



An Illustration

I will illustrate how the bottom-up and top-down phases of hypotheses generation work by considering a simple example, again from the family tree domain. Additionally, to simplify the example, we will ignore marriages and divorces and consider only parent-child relationships.

First, the user draws stroke s1, which the primitive recognizer identifies as an ellipse. The bottom up step matches the ellipse against the shape descriptions it knows about and it sees that females are represented by ellipses so it generates a *female* hypothesis (f1). The bottom-up process stops at this point because, although the system could fit the female into a parent-child partial hypothesis, the system will not generate partial hypotheses containing only a single subshape if the shape consists of more than two subshapes. Because no partial hypotheses exist yet, there is nothing for the top-down step to do at this point.

Next, the user draws s2, which is recognized as a line. The bottom-up processing stops at this point and again there is nothing for the top-down step to do. Then, the user draws s3, which is too messy, and the primitive recognizer fails to recognize it as a line so nothing more happens. Next the user draws stroke s4, which is recognized as a line. The system then detects that s2 and s4 match many of the constraints in the arrow shape description and it generates an arrow partial hypothesis. Again, to control the search space, the system will not generate higher-level partial hypotheses using other partial hypotheses, so the bottom-up step finishes.

At this point, the top-down process sees that there is a partial arrow hypothesis (that looks quite likely) and it attempts to find a stroke to fill the missing subshape. It detects stroke s3 and adds it to the existing arrow hypothesis. Even though s3 does not look perfectly like a line, the other constraints between s3, s2 and s4 strongly support the arrow hypothesis and the system accepts this modification, reinterpreting s3 as a line.

Now that the arrow hypothesis is complete, the system proposes a Child-link hypothesis for the arrow and then combines that hypothesis with f1 to create a parent-child relationship hypothesis. Finally, the user draws s5, which the system recognizes as an ellipse and proposes a female hypothesis which is added to the existing parent-child hypothesis. s5 strongly supports the parent-child hypothesis, which gains strength, in turn strengthening the system's belief that s3 is indeed a line.

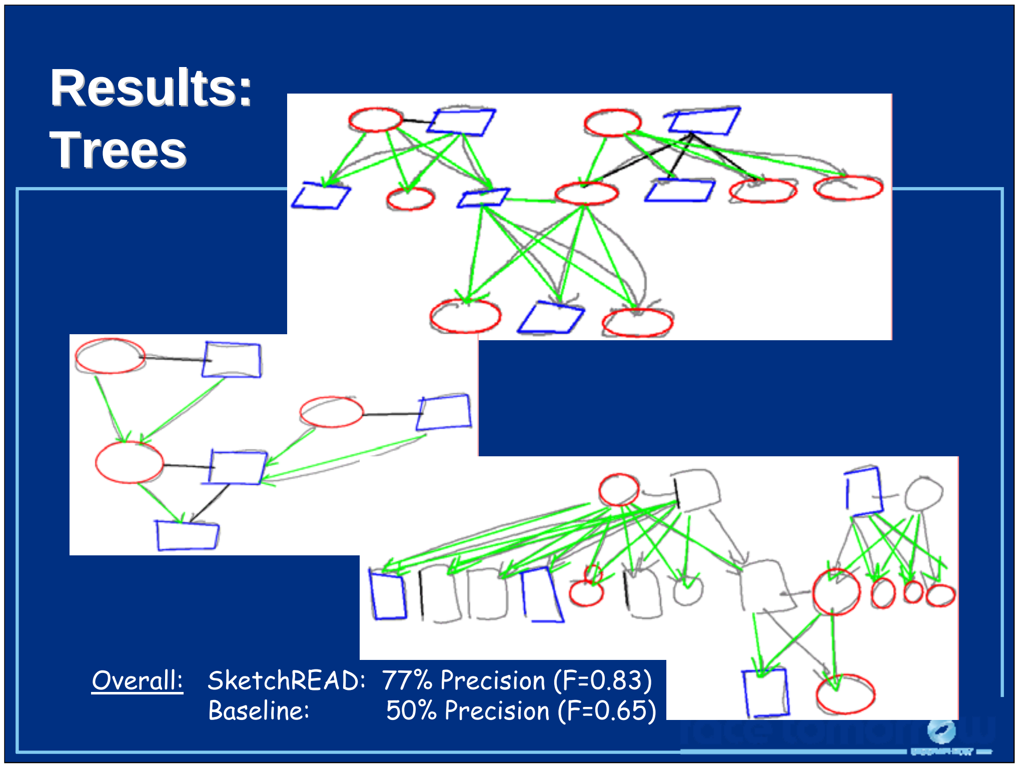
I have given a very brief introduction to both hypothesis management and hypothesis generation. More information on each is provided in the following sources:

Christine Alvarado and Randall Davis, Dynamically Constructed Bayes Nets for Multi-Domain Sketch Understanding. Proceedings of the International Joint Conference on Artificial Intelligence, 2005.

Christine Alvarado and Randall Davis. SketchREAD: A Multi-domain Sketch Recognition Engine**.** Proceedings of UIST 2004.

Christine Alvarado. Multi-domain Sketch Understanding. PhD Thesis. Massachusetts Institute of Technology. September, 2004.

All are available online here: http://www.cs.hmc.edu/~alvarado/research/publications.html

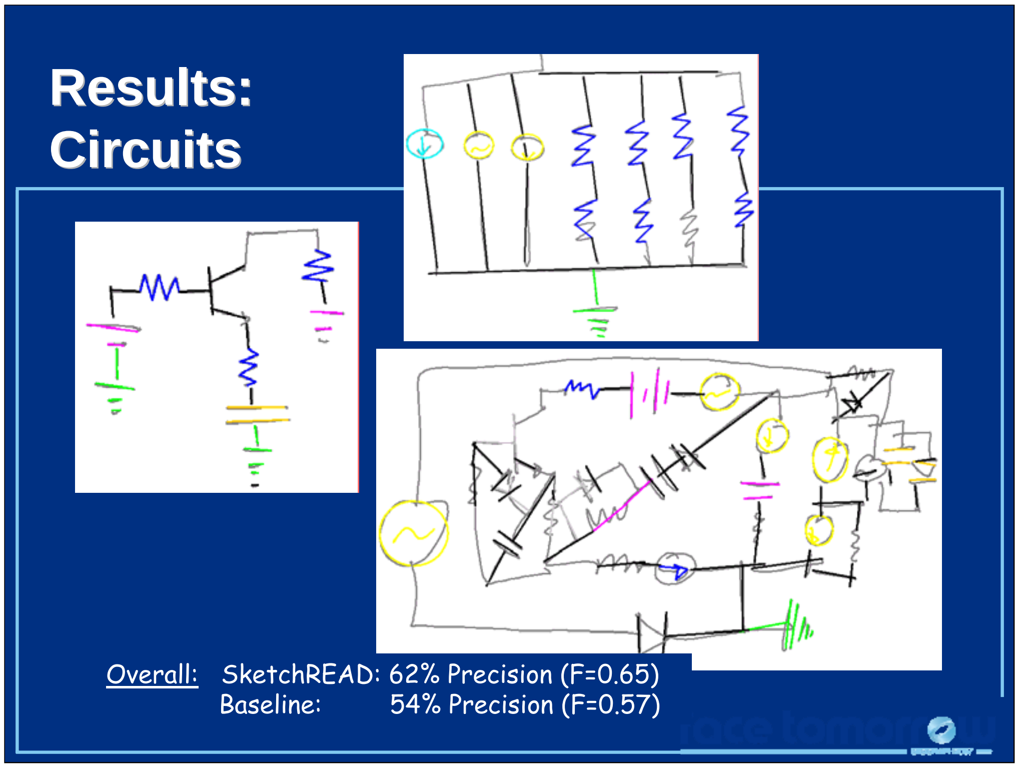


Results: Trees

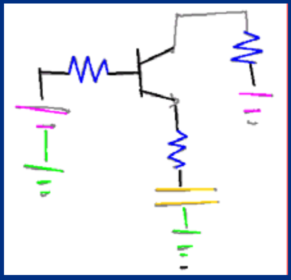
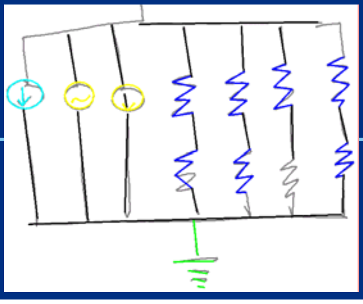
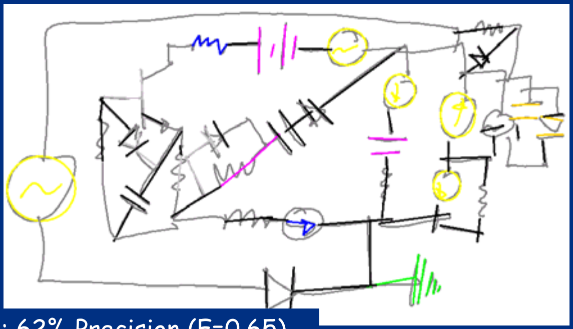
Overall: SketchREAD: 77% Precision (F=0.83)
Baseline: 50% Precision (F=0.65)

Now that we have examined how this approach works, let's look at how it performs. We collected data using a tablet PC using a data collection program that allows the user to sketch freely and displays the user’s strokes exactly as she draws them, without performing any type of recognition. We asked users to draw their family tree and to design simple analog circuits. We compared our system's performance with a constraint-based approach that used thresholds and only bottom up processing.

We found that our system performed reasonably well on even complex sketches, outperforming the baseline system significantly. It was able to recover from low-level recognition errors using the context provided by the surrounding strokes.

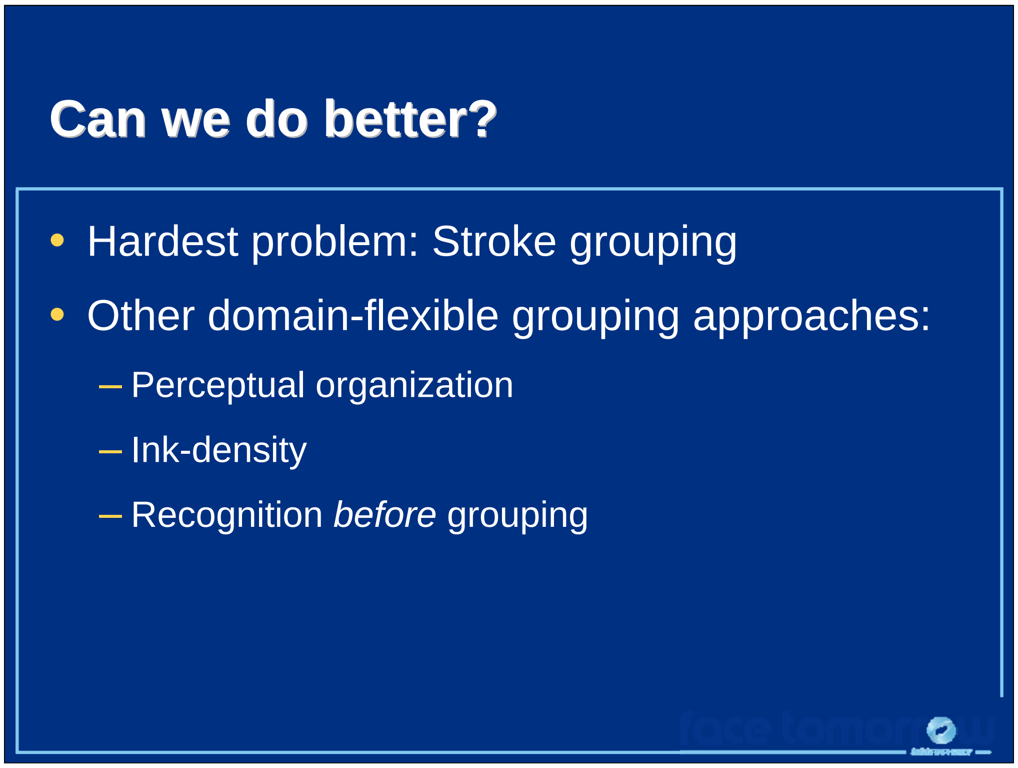


Results: Circuits

Overall: SketchREAD: 62% Precision (F=0.65)
Baseline: 54% Precision (F=0.57)

Performance on the circuit diagrams was slightly lower, but our system still outperformed the baseline system. One reason for the lower performance is that the circuit diagrams were generally quite a bit messier than the family tree diagrams.

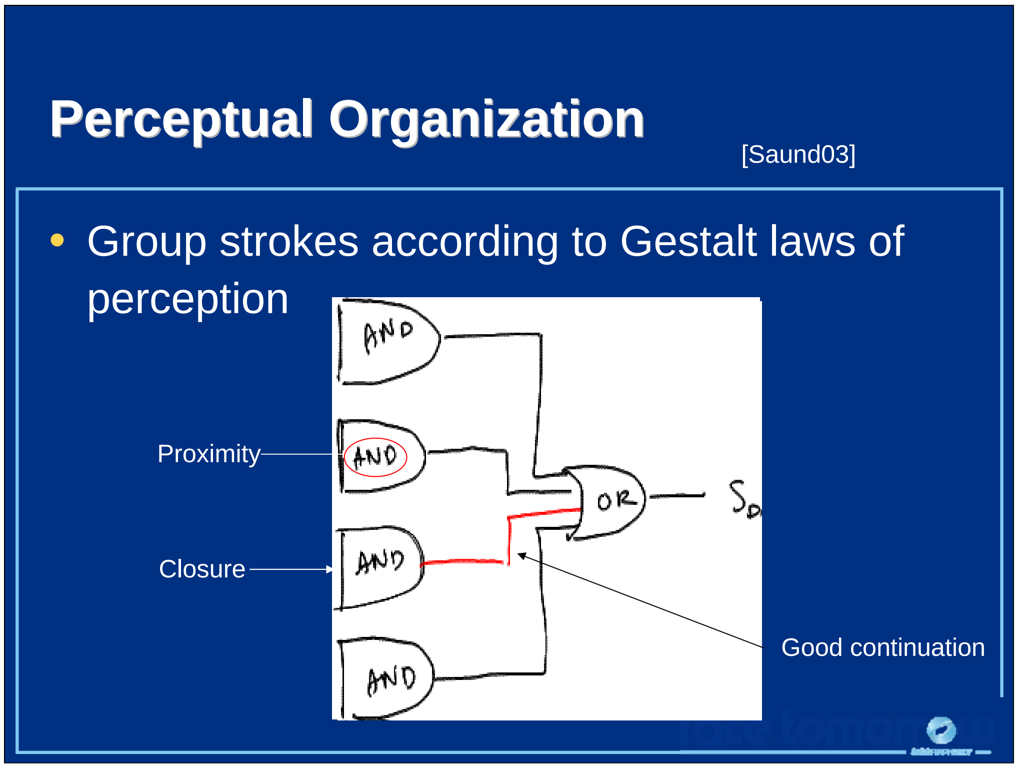


Can we do better?

- Hardest problem: Stroke grouping
- Other domain-flexible grouping approaches:
  - Perceptual organization
  - Ink-density
  - Recognition *before* grouping

While this approach outperforms the baseline system, 38% error is far from acceptable in a real application. Despite our context-driven search for possible interpretation, the biggest problem this approach faces is that it sometimes fails to generate the correct hypothesis. In particular, the initial set of partial hypotheses still relies on a blind search through combinations of strokes close together in space and time. Often this search was not able to explore a large enough set of combinations and failed to generate a partial hypothesis corresponding to the correct interpretation.

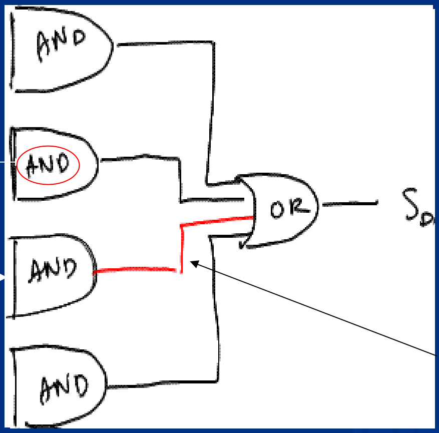
I will briefly discuss three domain independent or domain-flexible, approaches that can help us generate a better initial set of partial hypotheses: an approach based on perceptual organization, and approach based on "ink density" and an approach that (roughly) recognizes what objects individual strokes are part of before trying to group them.



Perceptual Organization

[Saund03]

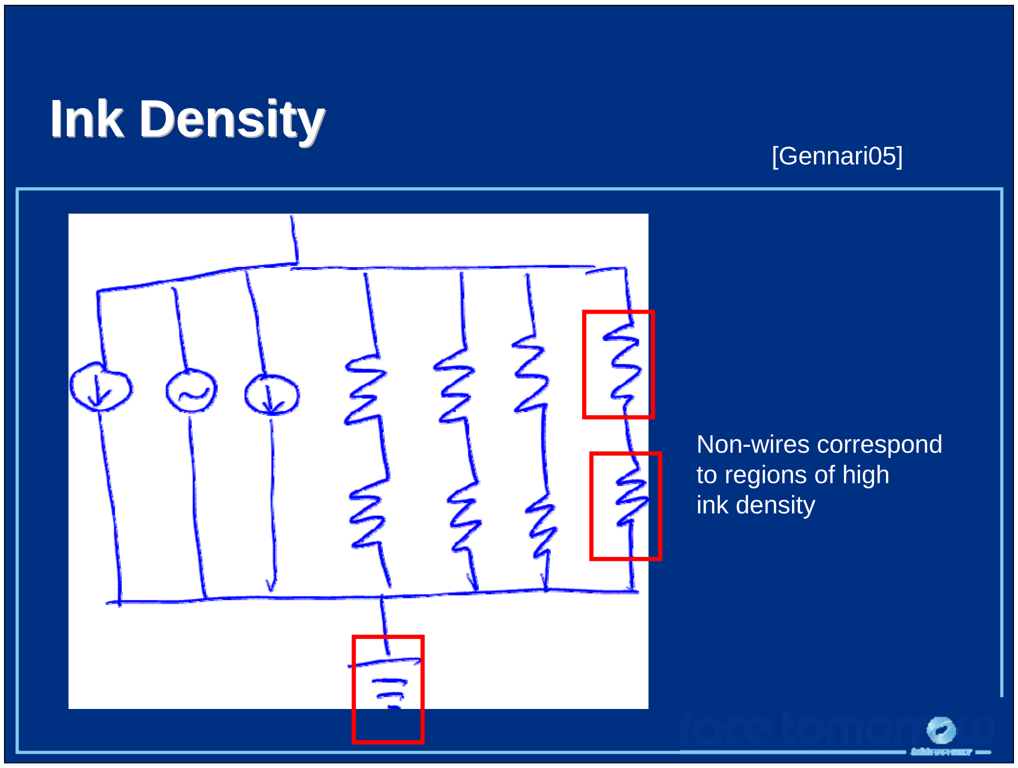
- Group strokes according to Gestalt laws of perception



Saund has developed a domain-independent grouping algorithm based on the Gestalt laws of perception. A key insight to this approach is that it distinguishes between two different types of primitive type of ink—"strokes" (essentially, long line or arc segments) and "blobs" (relatively dense ink, usually corresponding to text)—and then uses different Gestalt laws to group each type. After the algorithm identifies the strokes and blobs in a sketch, groups strokes using the laws of closure and good continuation and groups blobs using the law of proximity.

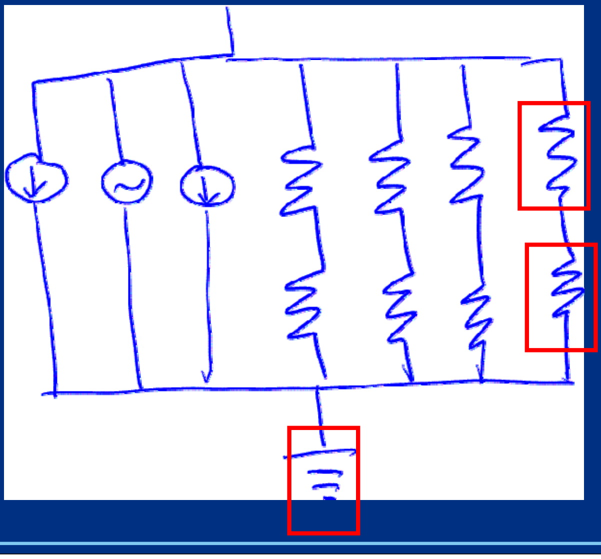
More information about this approach can be found in:

Saund, E., Fleet, D., Larner, D., and Mahoney, J.; "Perceptually-Supported Image Editing of Text and Graphics," *Proc. UIST '03 (ACM Symposium on User Interface Software and Technology)*, pp. 183-192.



Ink Density

[Gennari05]



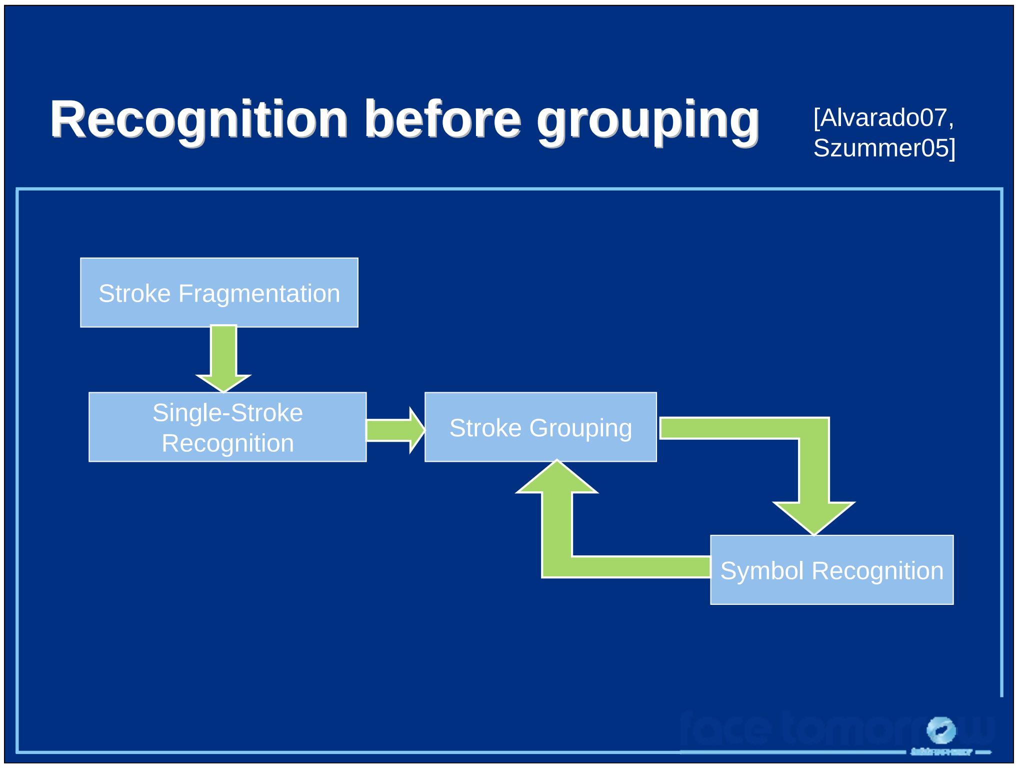
Non-wires correspond to regions of high ink density

Gennari *et al.* developed an approach to stroke grouping that relies on ink density of find stroke groups. Their insight was that in some domains (such as analog circuit design) diagrams consist of shapes with relatively high density (such as resistors and ground symbols) separated by shapes with relatively low density (such as wires). They detect shape groups by measuring the amount of ink relative to the square bounding box of a stroke.

Unfortunately, this approach requires that shapes be drawn with consecutive strokes as the stroke density search algorithm only considers groups of temporally contiguous strokes. Still, the algorithm may provide a good starting point for a more general recognition engine.

More details about this approach can be found in

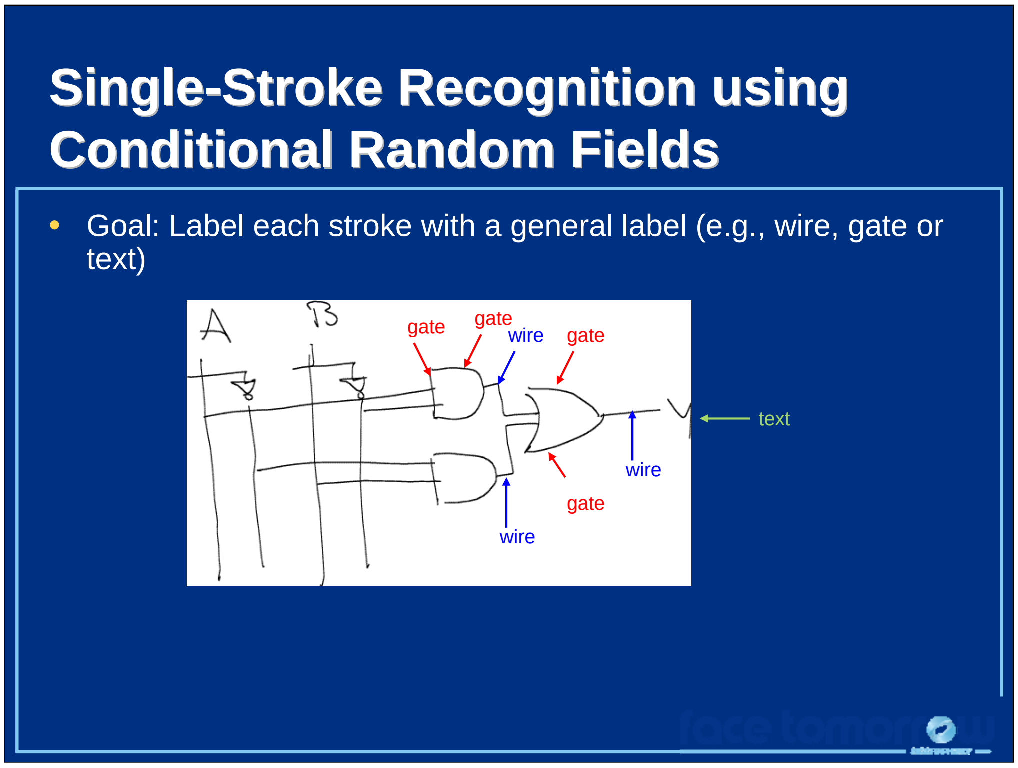
Leslie Gennari, Levent Burak Kara, Thomas F. Stahovich (2005) Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams.
Computers & Graphics 29(4): 547-562 2005.



Recognition before grouping

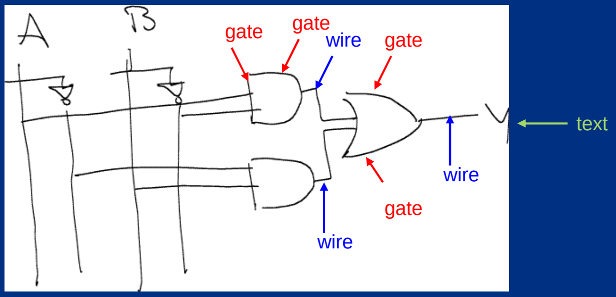
[Alvarado07, Szummer05]

A final approach to improving stroke grouping is to attempt to recognize the domain level shape each stroke is part of *before* attempting to group the strokes. The idea is once strokes are labeled as according to their high-level symbols, strokes with the same label will be far away in time and space and it will be easy to detect the boundaries between strokes.



Single-Stroke Recognition using Conditional Random Fields

- Goal: Label each stroke with a general label (e.g., wire, gate or text)



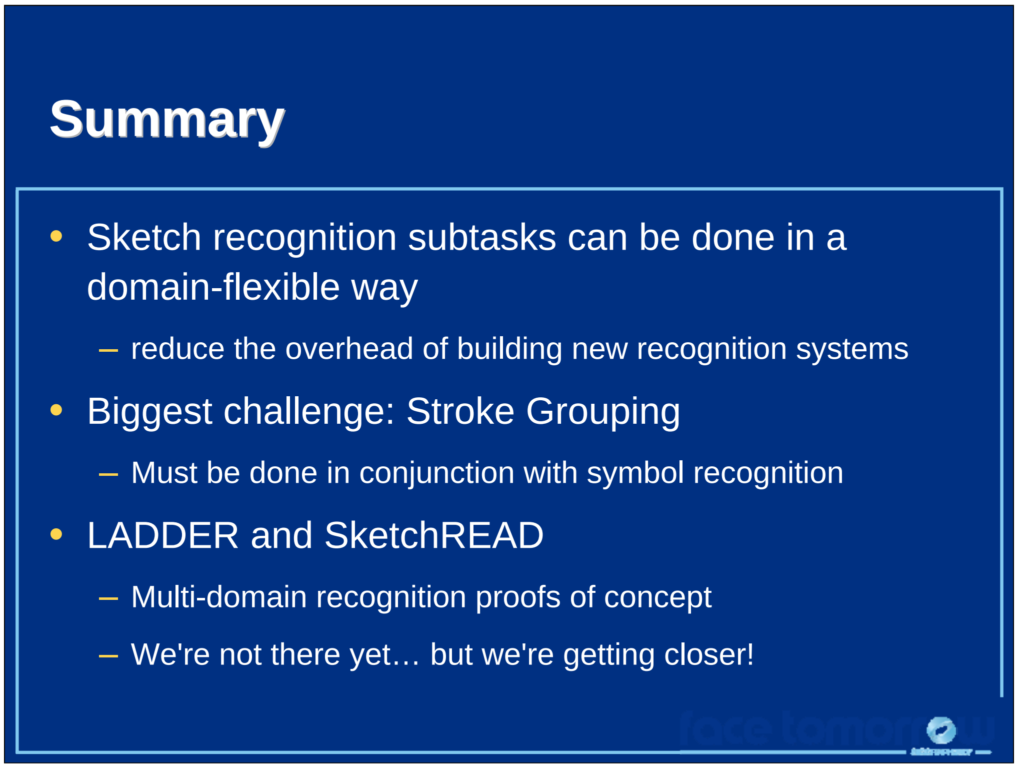
For example, in this example the goal is to label each individual stroke as part of a wire, a gate or text. While this task seems virtually impossible (how can you tell if a stroke is part of a gate unless you see the whole gate?), a technique that uses conditional random fields is surprisingly effective. Conditional random fields allow the system to model the its belief in the interpretation of a single stroke in terms of the stroke data and the context of its neighbors (i.e., the strokes close to it in time and space).

Unfortunately, conditional random fields must be trained for a particular domain. Fortunately, they require relatively few training sketches (10-20) to be quite effective at recognizing single strokes.

More information on this approach can be found in:

Yuan Qi, Martin Szummer, Thomas P. Minka. Diagram Structure Recognition by Bayesian Conditional Random Fields 2005 Proc Comp. Vision Pattern Recogn. (CVPR) C. Schmid and S. Soatto and C. Tomasi 191--196

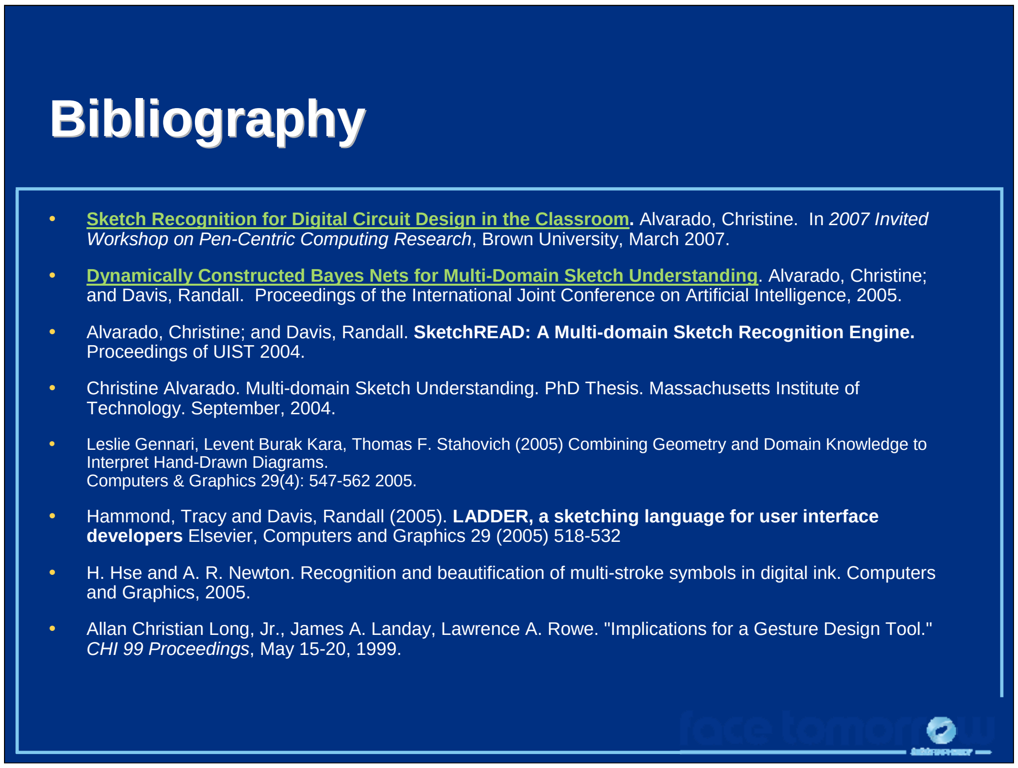
Alvarado, Christine. Sketch Recognition for Digital Circuit Design in the Classroom. In *2007 Invited Workshop on Pen-Centric Computing Research*, Brown University, March 2007.



Summary

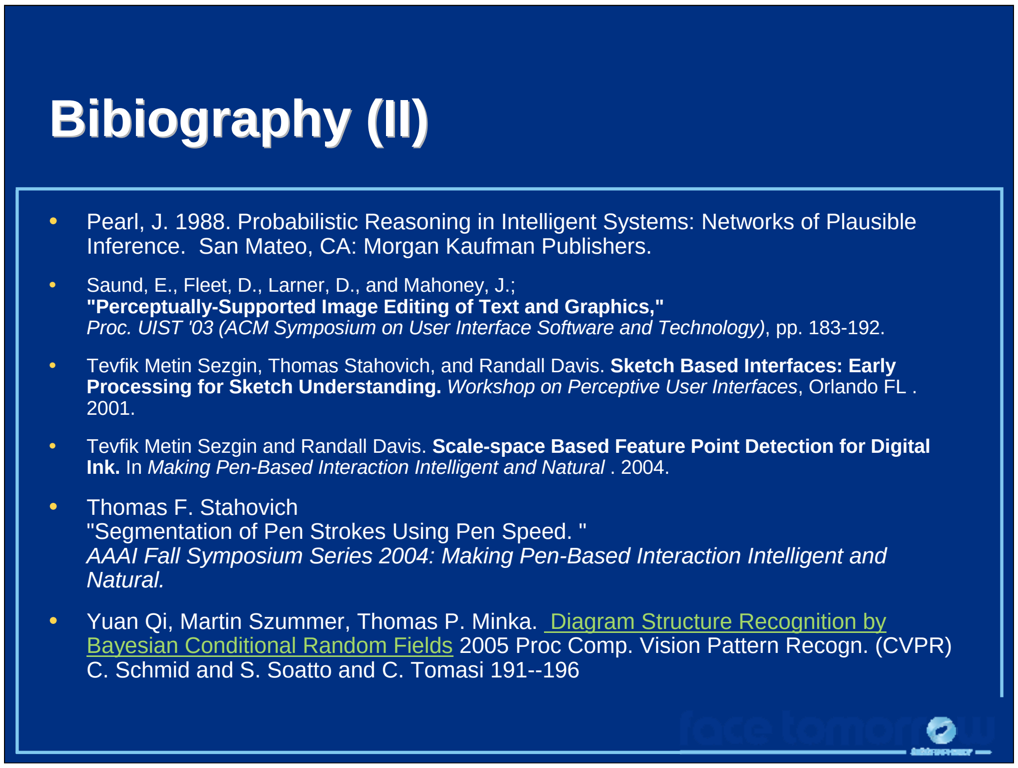
- Sketch recognition subtasks can be done in a domain-flexible way
  - reduce the overhead of building new recognition systems
- Biggest challenge: Stroke Grouping
  - Must be done in conjunction with symbol recognition
- LADDER and SketchREAD
  - Multi-domain recognition proofs of concept
  - We're not there yet… but we're getting closer!

Today we have examined a number of techniques for building multi-domain sketch recognition systems that allow the user to sketch freely and do not place unnatural restrictions on her drawing style. We have seen two proof of concept systems that accomplish this goal, and there certainly have been and will be other systems designed for the same task. Stroke grouping remains the biggest challenge in this process, but new methods reveal promising directions for exploration. By continuing to explore these areas we will move closer to the dream of natural, seamless sketch-based interaction with the computer.



Bibliography

- Sketch Recognition for Digital Circuit Design in the Classroom**.** Alvarado, Christine. In *2007 Invited Workshop on Pen-Centric Computing Research*, Brown University, March 2007.
- Dynamically Constructed Bayes Nets for Multi-Domain Sketch Understanding. Alvarado, Christine; and Davis, Randall. Proceedings of the International Joint Conference on Artificial Intelligence, 2005.
- Alvarado, Christine; and Davis, Randall. **SketchREAD: A Multi-domain Sketch Recognition Engine.** Proceedings of UIST 2004.
- Christine Alvarado. Multi-domain Sketch Understanding. PhD Thesis. Massachusetts Institute of Technology. September, 2004.
- Leslie Gennari, Levent Burak Kara, Thomas F. Stahovich (2005) Combining Geometry and Domain Knowledge to Interpret Hand-Drawn Diagrams.
  Computers & Graphics 29(4): 547-562 2005.
- Hammond, Tracy and Davis, Randall (2005). **LADDER, a sketching language for user interface developers** Elsevier, Computers and Graphics 29 (2005) 518-532
- H. Hse and A. R. Newton. Recognition and beautification of multi-stroke symbols in digital ink. Computers and Graphics, 2005.
- Allan Christian Long, Jr., James A. Landay, Lawrence A. Rowe. "Implications for a Gesture Design Tool." *CHI 99 Proceedings*, May 15-20, 1999.



Bibiography (II)

- Pearl, J. 1988. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. San Mateo, CA: Morgan Kaufman Publishers.
- Saund, E., Fleet, D., Larner, D., and Mahoney, J.;
  "Perceptually-Supported Image Editing of Text and Graphics,"
  Proc. UIST '03 (ACM Symposium on User Interface Software and Technology), pp. 183-192.
- Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. **Sketch Based Interfaces: Early Processing for Sketch Understanding.** *Workshop on Perceptive User Interfaces*, Orlando FL . 2001.
- Tevfik Metin Sezgin and Randall Davis. **Scale-space Based Feature Point Detection for Digital Ink.** In *Making Pen-Based Interaction Intelligent and Natural* . 2004.
- Thomas F. Stahovich
  "Segmentation of Pen Strokes Using Pen Speed. "
  AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural.
- Yuan Qi, Martin Szummer, Thomas P. Minka. Diagram Structure Recognition by Bayesian Conditional Random Fields 2005 Proc Comp. Vision Pattern Recogn. (CVPR) C. Schmid and S. Soatto and C. Tomasi 191--196

Sketch-Based Interfaces: Techniques and Applications

Designing Freeform Surfaces by Sketching

SIGGRAPH2007

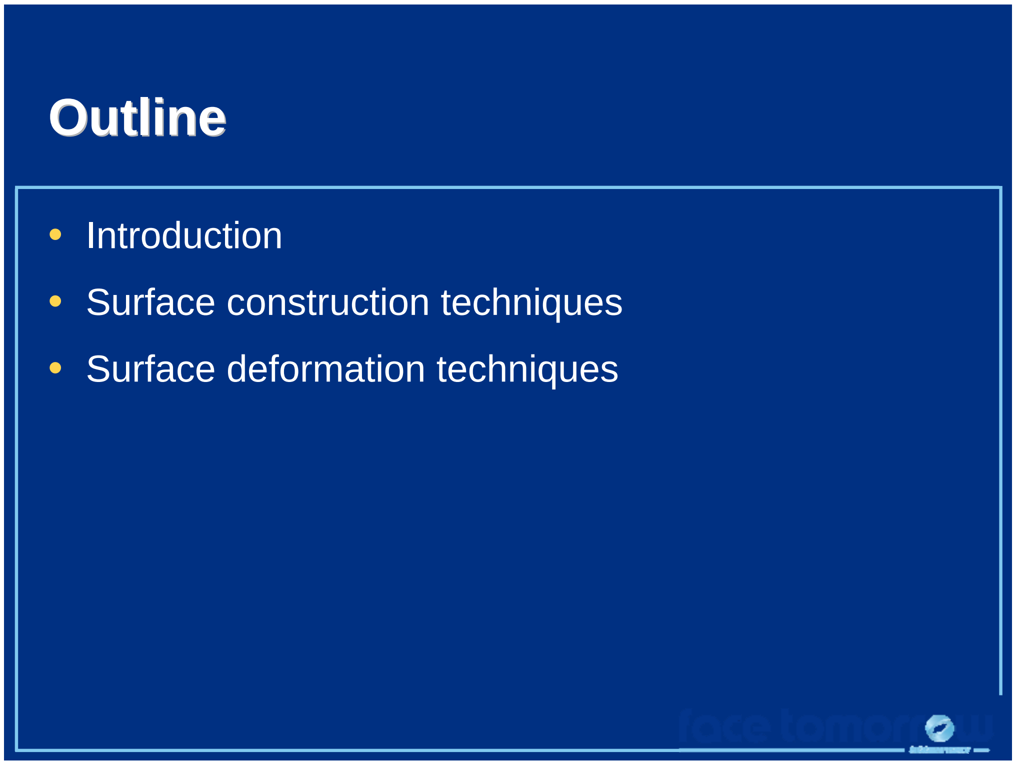
Takeo Igarashi

The University of Tokyo

Designing Freeform Surfaces by Sketching

Takeo Igarashi
Associate Professor, University of Tokyo
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033
Tokyo, JAPAN

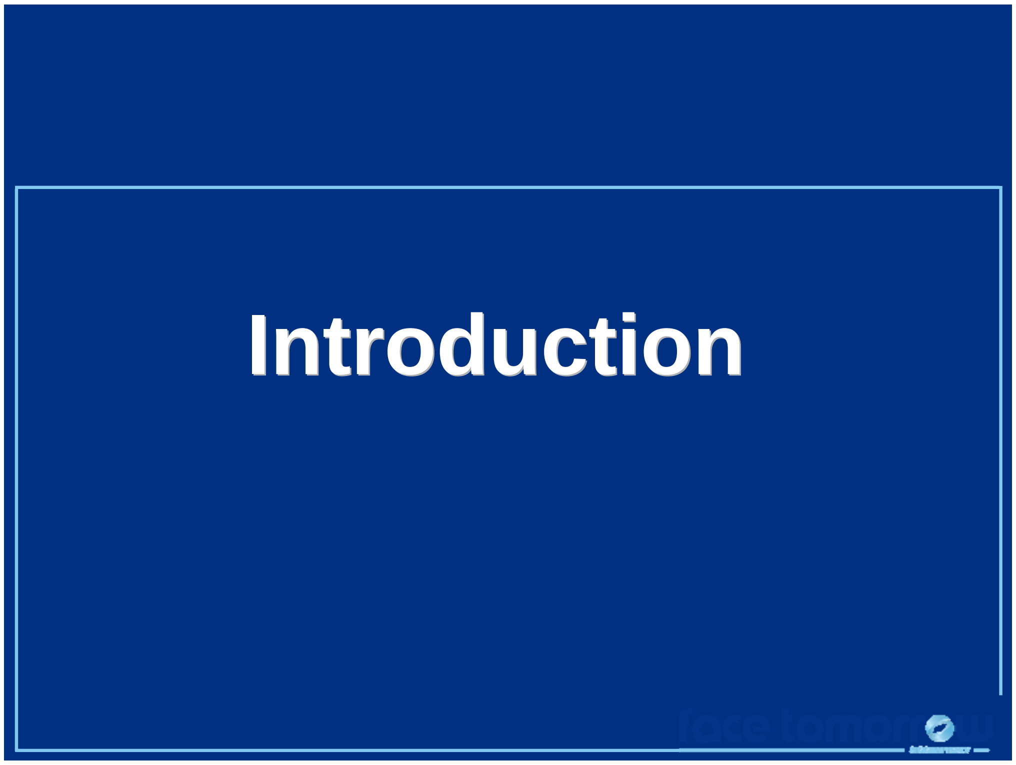
Email: takeo@acm.org
www-ui.is.s.u-tokyo.ac.jp/~takeo/



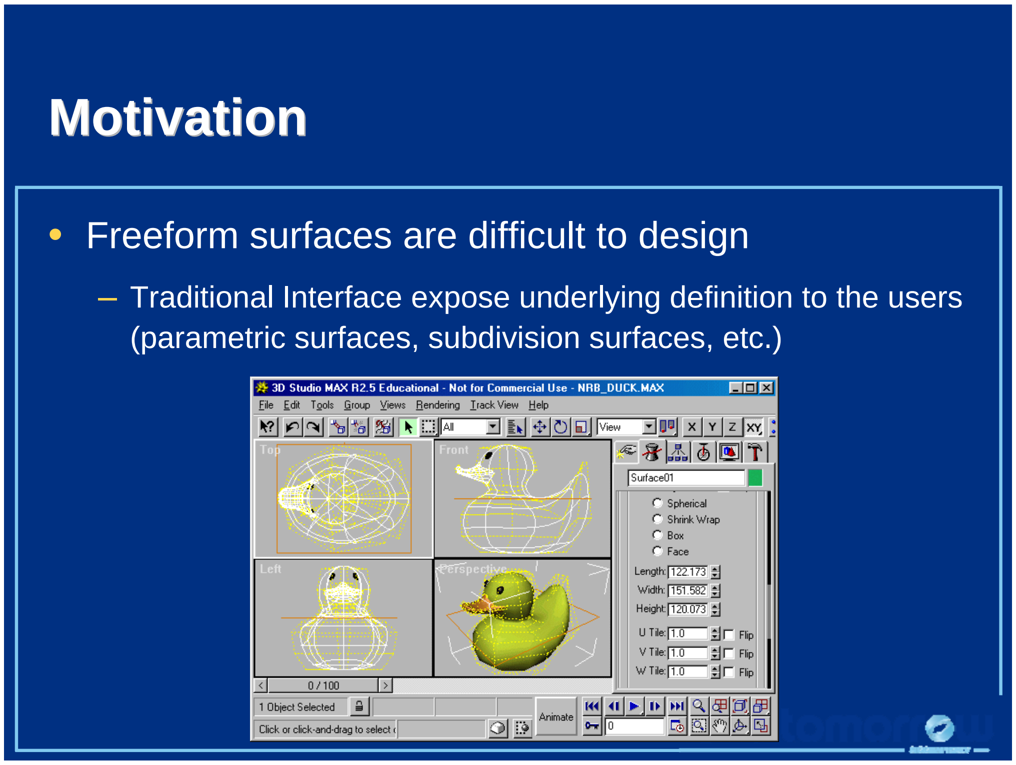
Outline

- Introduction
- Surface construction techniques
- Surface deformation techniques

In the morning session, we reviewed sketching systems in general including freeform modeling, animation, and other special purpose modeling systems. This session focuses on freeform modeling and discusses their implementation details.

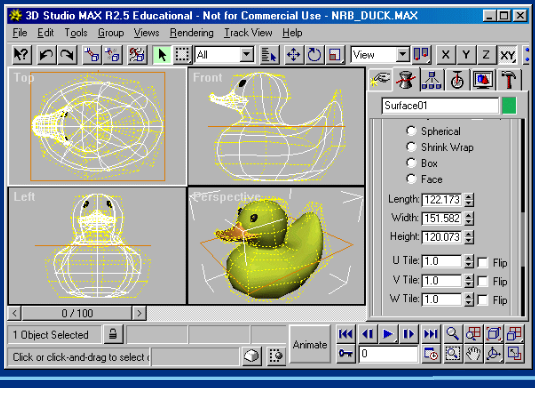


Introduction



Motivation

- Freeform surfaces are difficult to design
  - Traditional Interface expose underlying definition to the users (parametric surfaces, subdivision surfaces, etc.)



3D Studio MAX R2.5 Educational - Not for Commercial Use - NRB_DUCK.MAX

Surface01

Spherical
Shrink Wrap
Box
Face

Length: 122.173
Width: 151.582
Height: 120.073

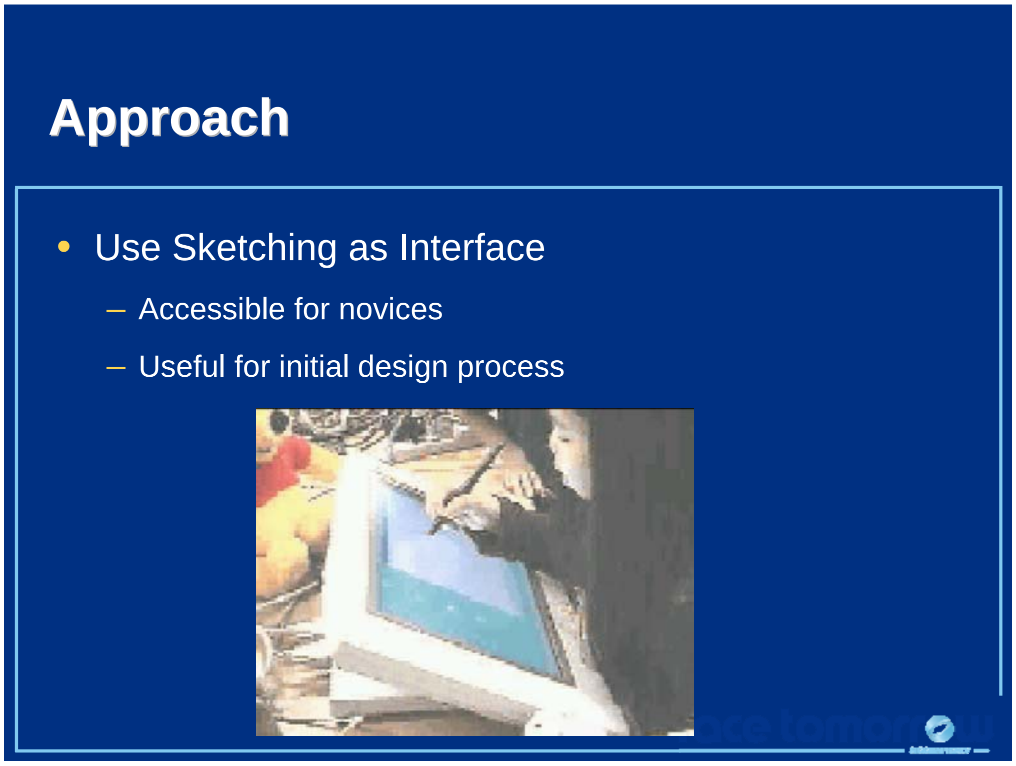
U Tile: 1.0 Flip
V Tile: 1.0 Flip
W Tile: 1.0 Flip

0 / 100

1 Object Selected

Click or click-and-drag to select

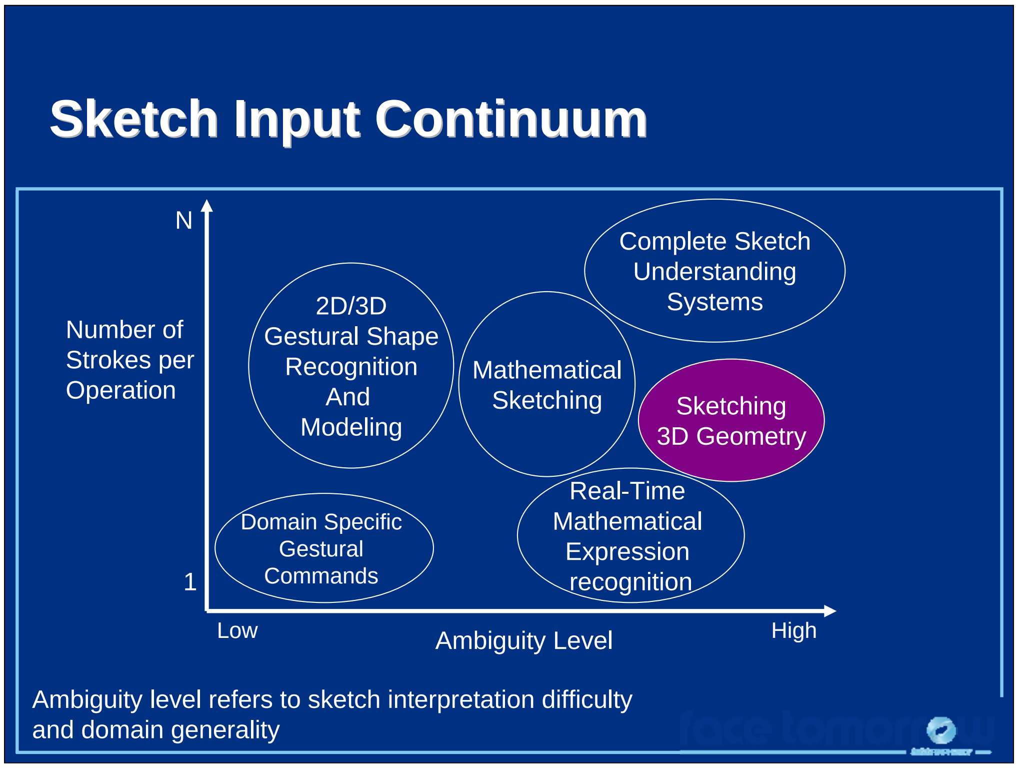
Traditional modeling interfaces expose the underlying definition of the surface to the user. This is good for detailed design, but not accessible for novice users.



Approach

- Use Sketching as Interface
  - Accessible for novices
  - Useful for initial design process

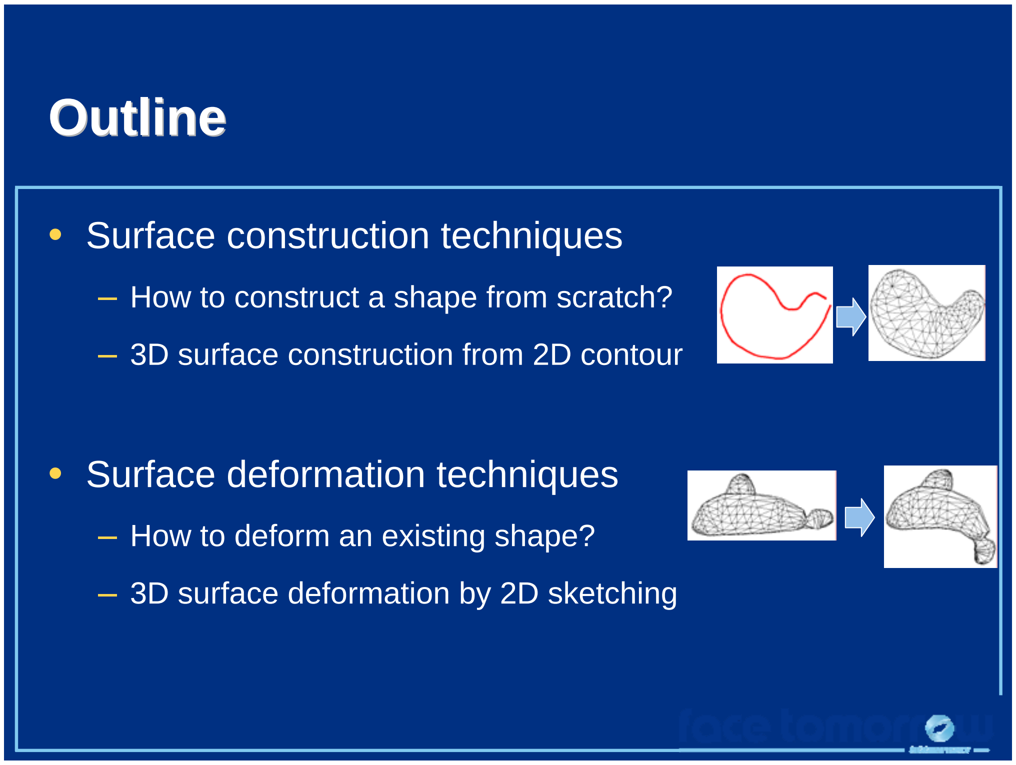
The goal of sketching interfaces is to solve the problem of accessibility by dramatically simplifying the user interface. Instead of requiring the user to work with traditional buttons, menus, and dragging operations, sketching allows them to directly express their thoughts in the form of freehand sketching. It can make 3D graphics authoring accessible to novice users, even children. In addition, simple interfaces make it possible for experts to use them in the initial design process.



Sketch Input Continuum

N
Number of Strokes per Operation
1
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Complete Sketch Understanding Systems
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition
Low
Ambiguity Level
High

Ambiguity level refers to sketch interpretation difficulty and domain generality

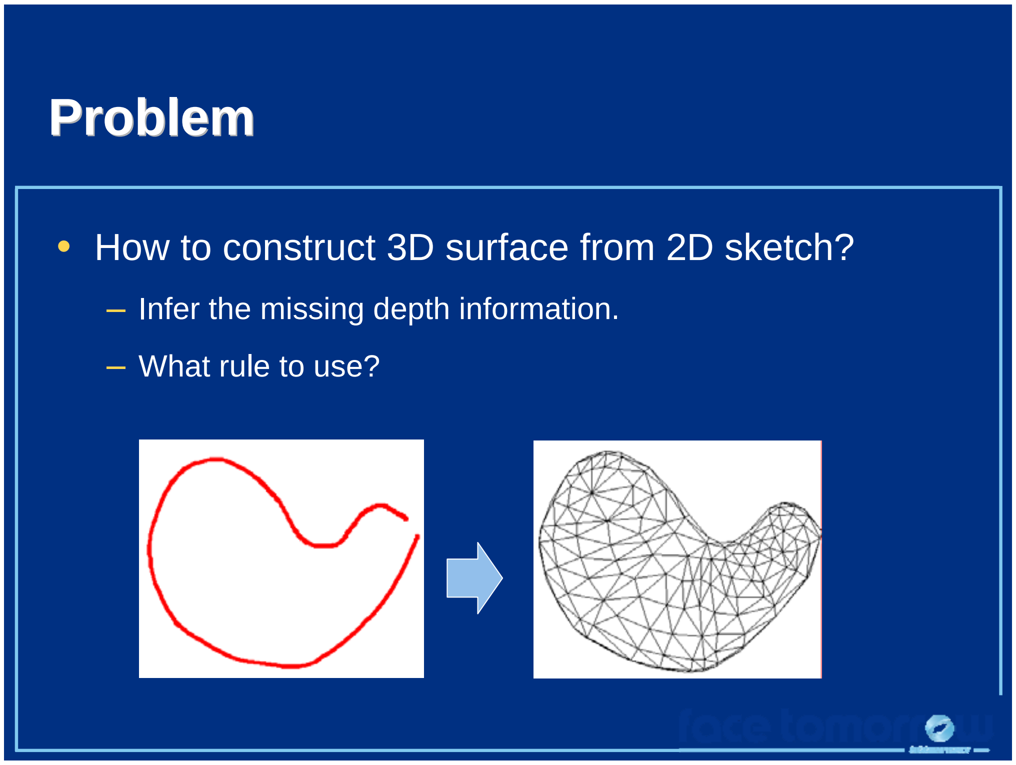


Outline

- Surface construction techniques
  - How to construct a shape from scratch?
  - 3D surface construction from 2D contour
- Surface deformation techniques
  - How to deform an existing shape?
  - 3D surface deformation by 2D sketching

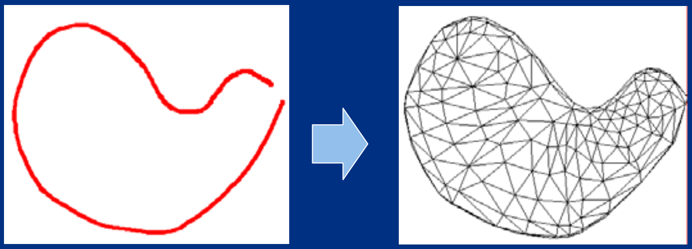
I will describe surface construction techniques and surface deformation techniques. Surface construction methods generate 3D surfaces from user's 2D sketch. Surface deformation methods deform an existing 3D shape according to the user's sketch.

Surface Construction Techniques

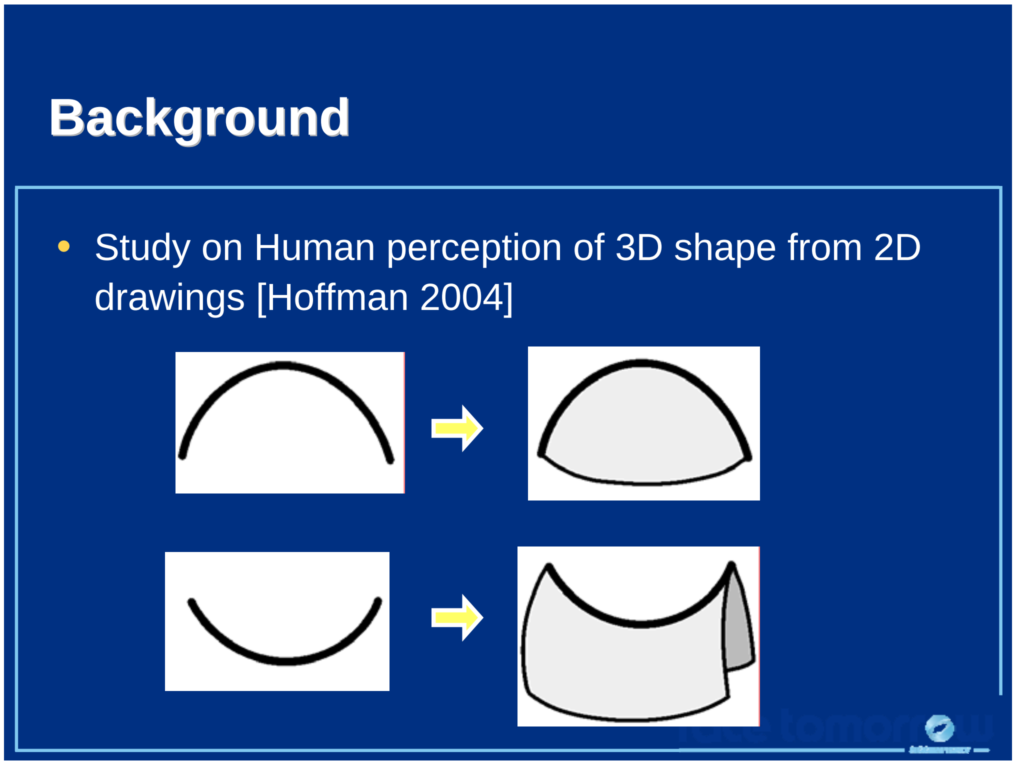


Problem

- How to construct 3D surface from 2D sketch?
  - Infer the missing depth information.
  - What rule to use?



The problem here is how to construct a 3D surface from a 2D sketch. Depth information is missing in a 2D sketch, so the system must infer the depth of the surface using some rule.



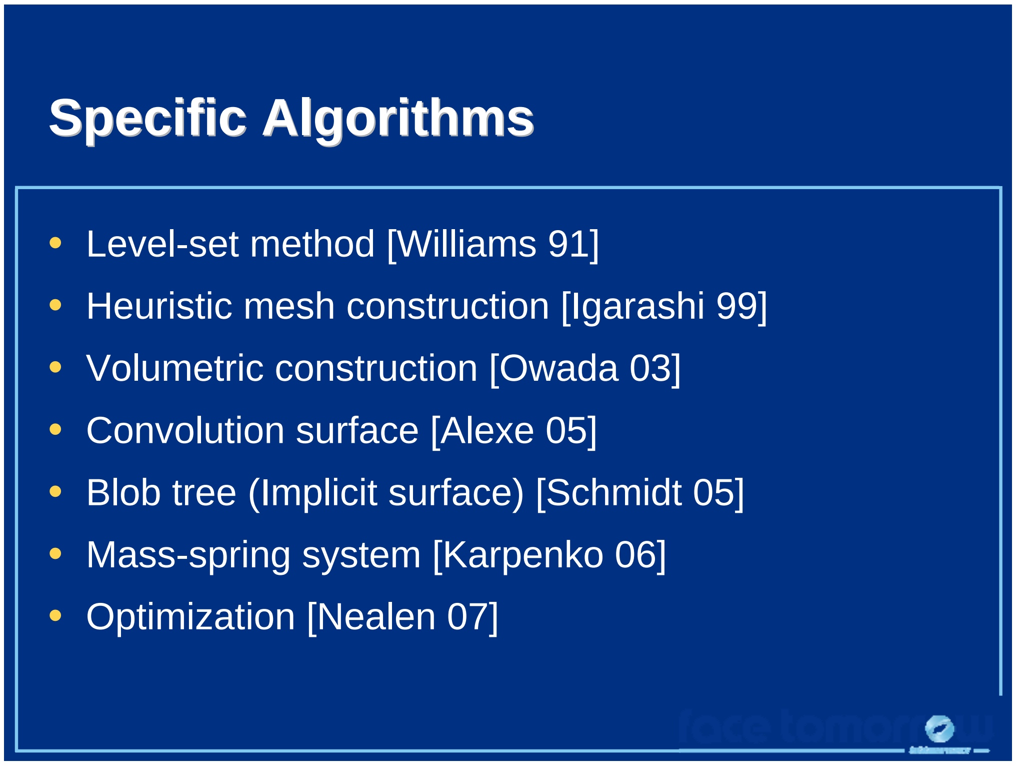
Background

- Study on Human perception of 3D shape from 2D drawings [Hoffman 2004]

The basis of sketch-based 3D surface construction is human perception of 3D shape from 2D drawings.

According to a book titled “Visual intelligence”,

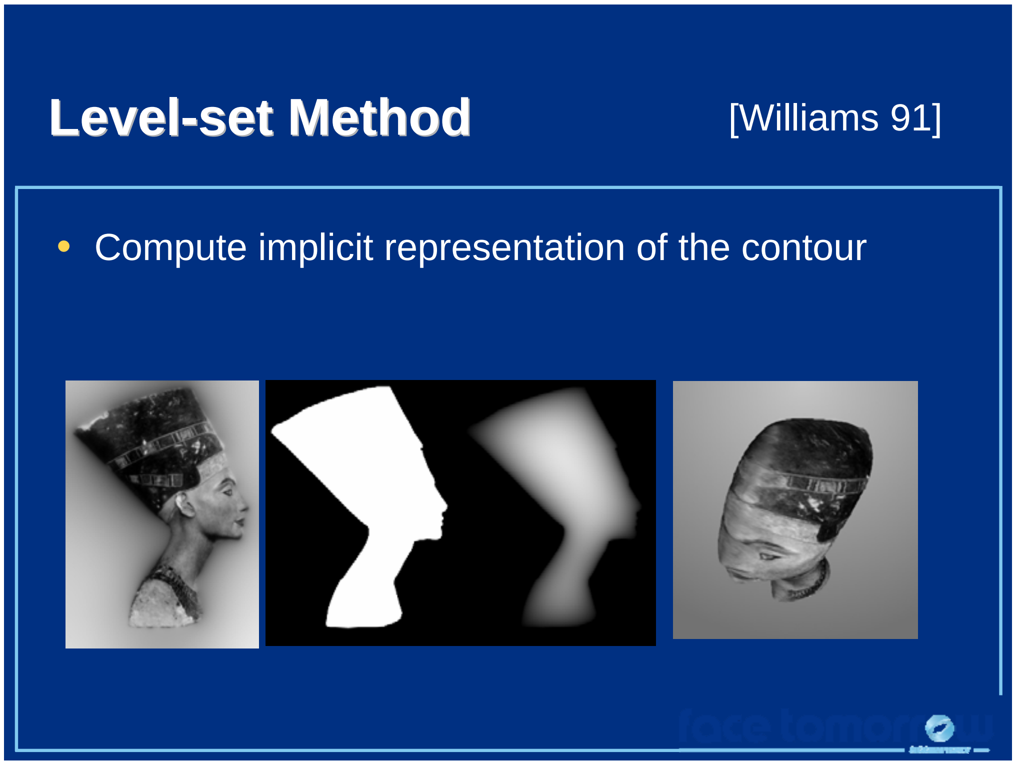
“human vision assumes a convex surface when seeing a convex silhouette and assumes a saddle-shaped surface when seeing a concave silhouette. These kinds of rules drive the design of surface construction algorithms.”



Specific Algorithms

- Level-set method [Williams 91]
- Heuristic mesh construction [Igarashi 99]
- Volumetric construction [Owada 03]
- Convolution surface [Alexe 05]
- Blob tree (Implicit surface) [Schmidt 05]
- Mass-spring system [Karpenko 06]
- Optimization [Nealen 07]

I will introduce several different approaches.

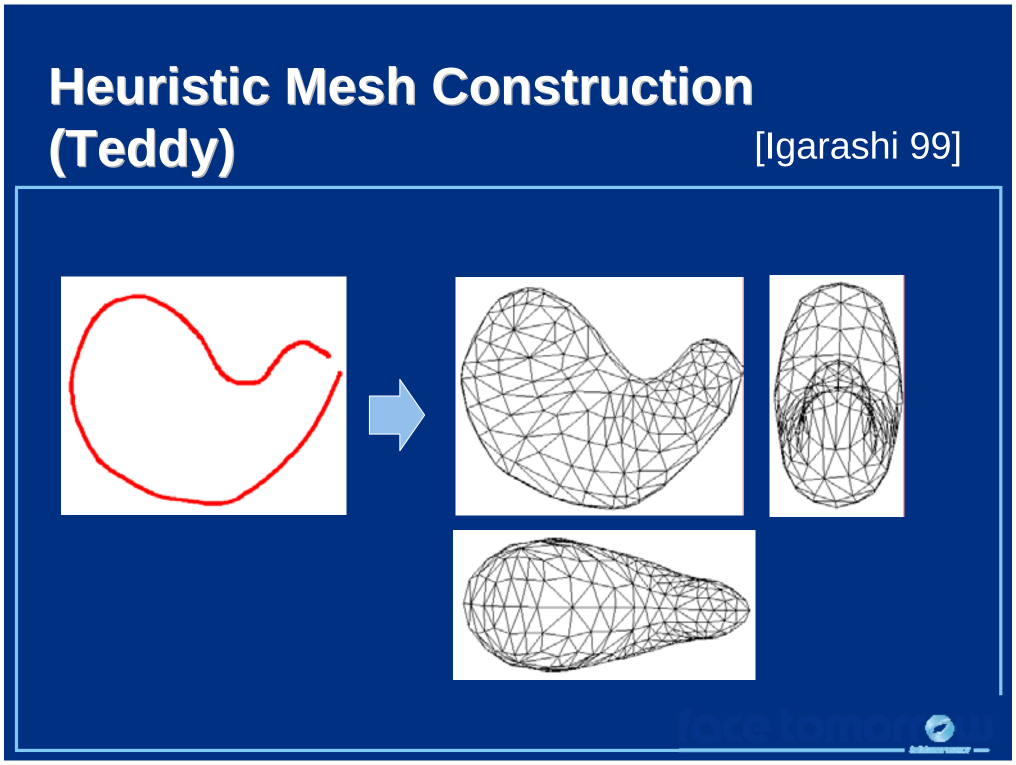


Level-set Method

[Williams 91]

- Compute implicit representation of the contour

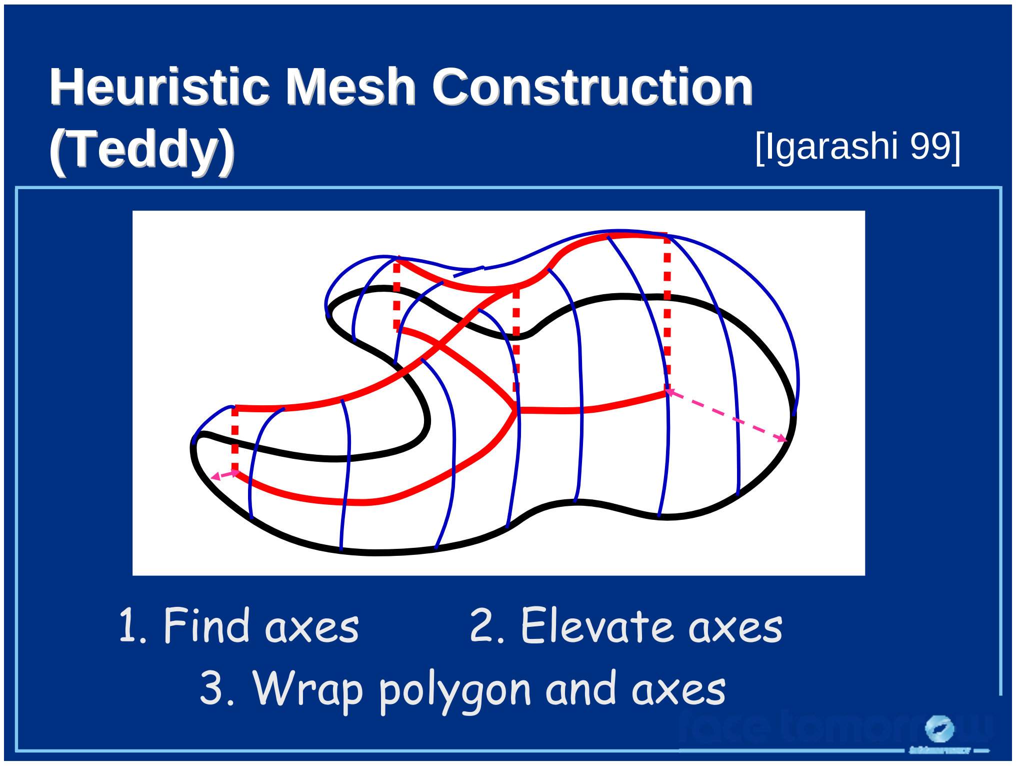
Williams inflated a 2D image by using a level-set method. The shape is first represented implicitly as signed pseudometric functions of the surface boundaries and inflated according to the result.



Heuristic Mesh Construction (Teddy)

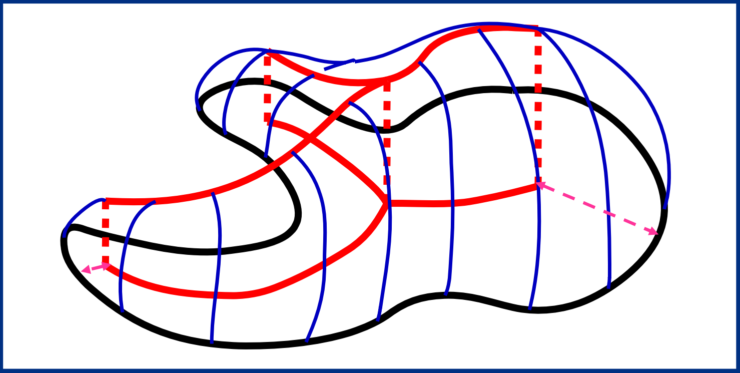
[Igarashi 99]

Teddy generates a mesh directly from a stroke (2D polyline).



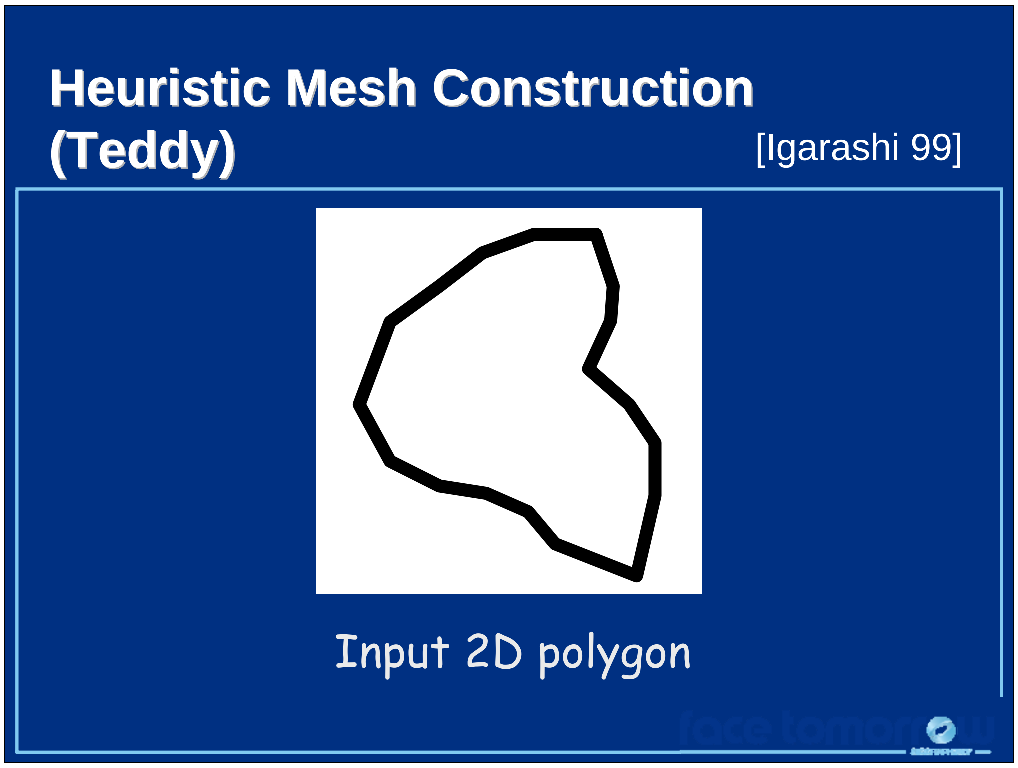
Heuristic Mesh Construction (Teddy)

[Igarashi 99]



1. Find axes
2. Elevate axes
3. Wrap polygon and axes

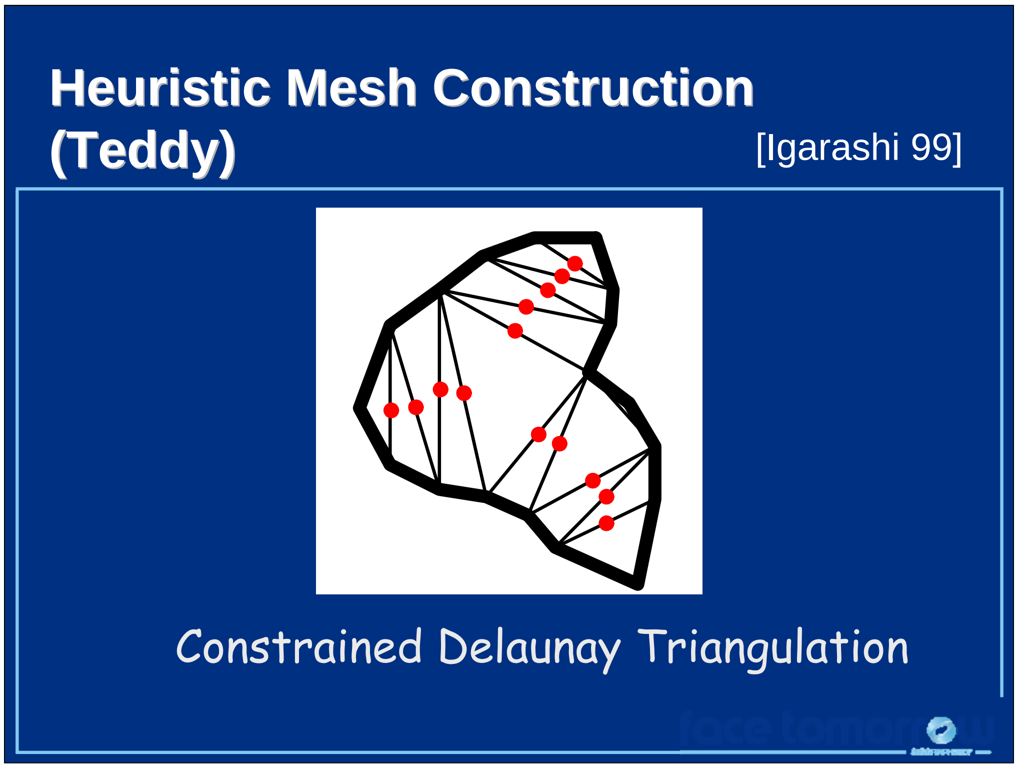
The input is a closed 2D polygonal region. The system first computes the skeleton or axis of the polygon. The system then elevates the skeleton. The amount of elevation is proportional to the distance between the skeleton and boundary. The system generates a surface by wrapping the elevated skeleton and the boundary.



Heuristic Mesh Construction (Teddy)

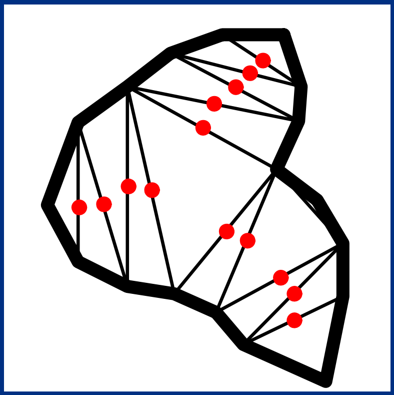
[Igarashi 99]

Input 2D polygon

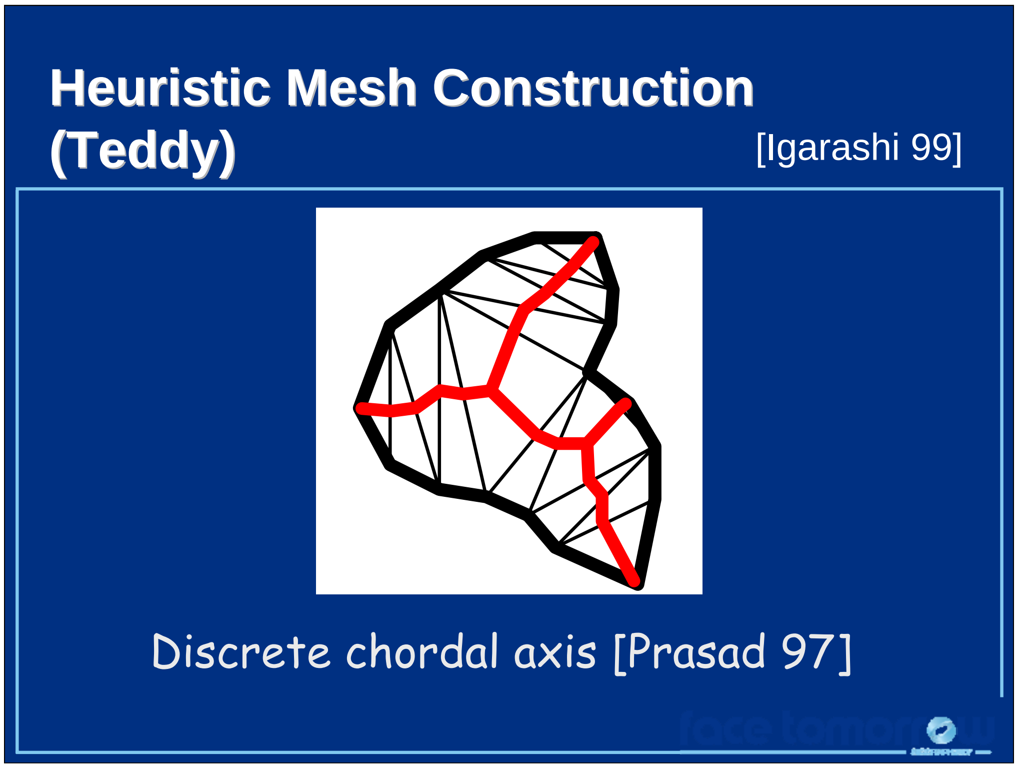


Heuristic Mesh Construction (Teddy)

[Igarashi 99]

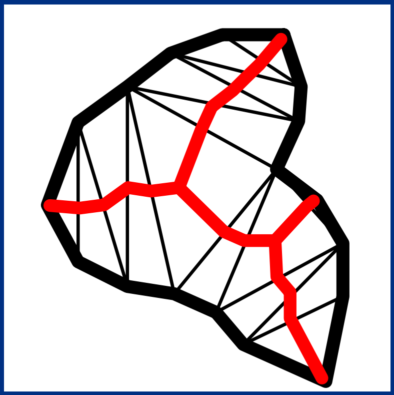


Constrained Delaunay Triangulation



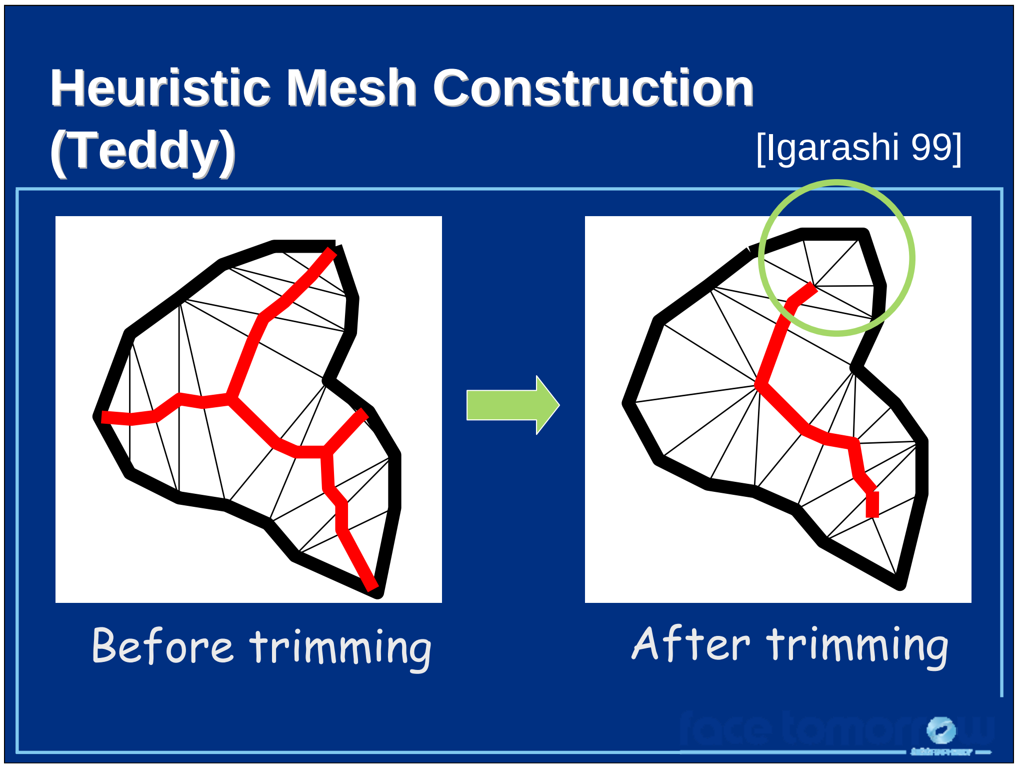
Heuristic Mesh Construction (Teddy)

[Igarashi 99]



Discrete chordal axis [Prasad 97]

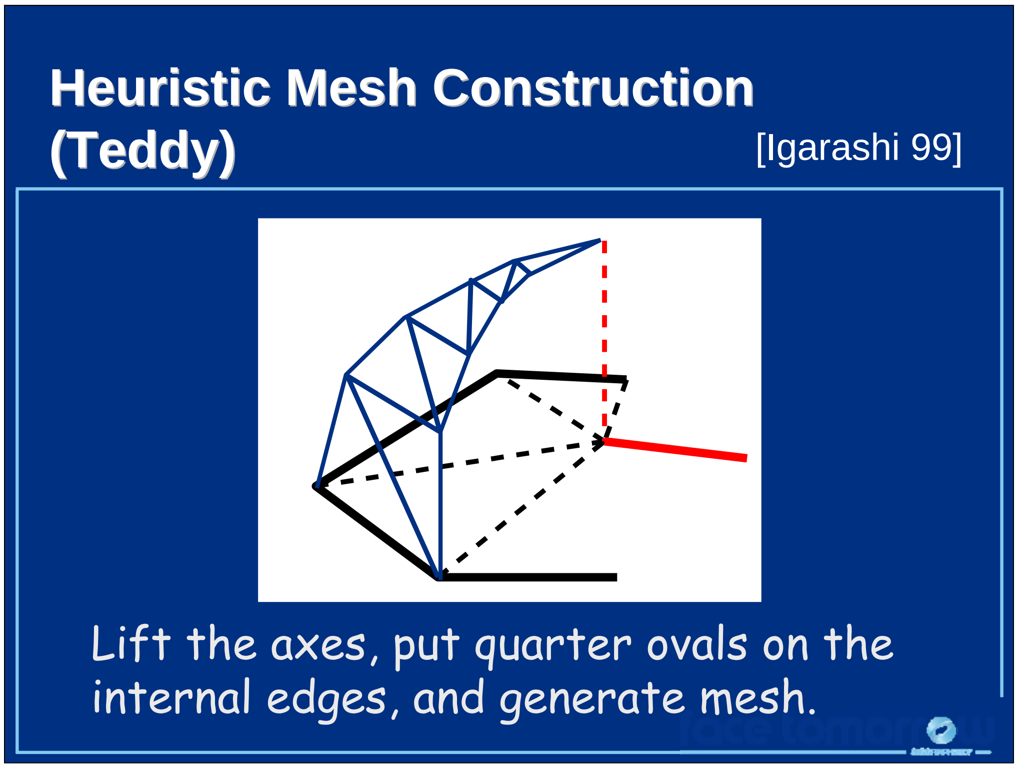
You can obtain the axis by connecting the mid points of internal triangles.



Heuristic Mesh Construction (Teddy)

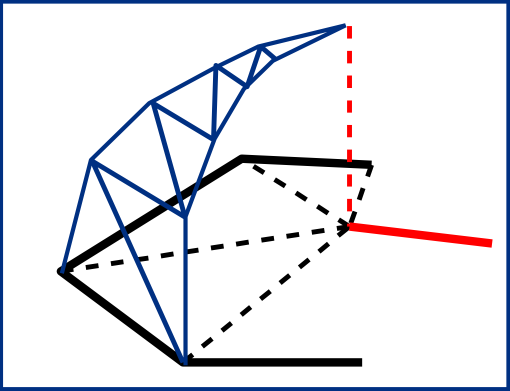
[Igarashi 99]

The system needs to trim insignificant branches.



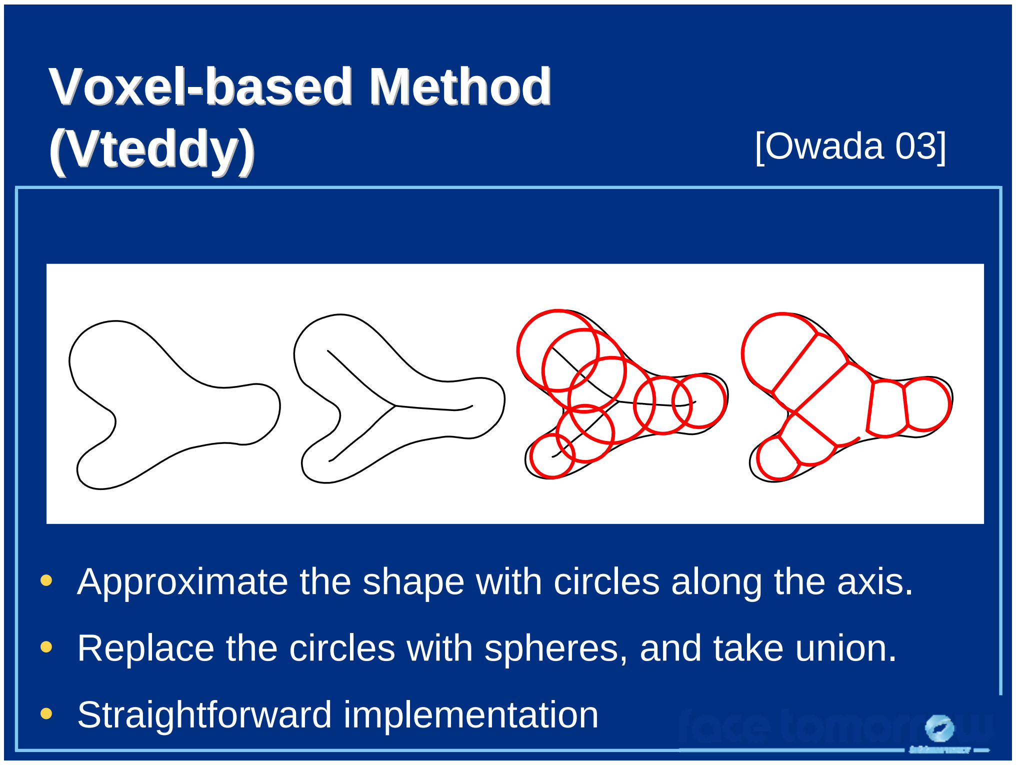
Heuristic Mesh Construction (Teddy)

[Igarashi 99]



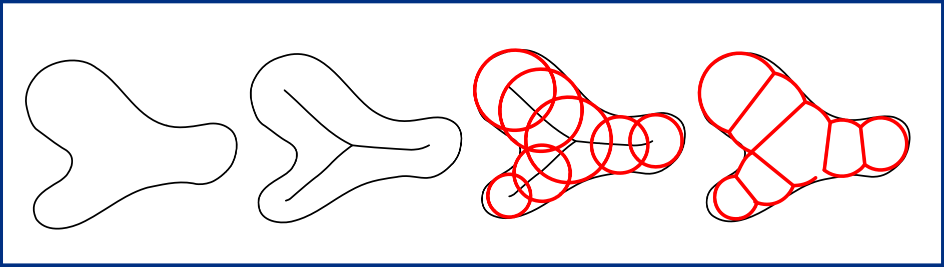
Lift the axes, put quarter ovals on the internal edges, and generate mesh.

The system elevates the axes, puts quarter ovals on the internal edges, and generates a mesh.



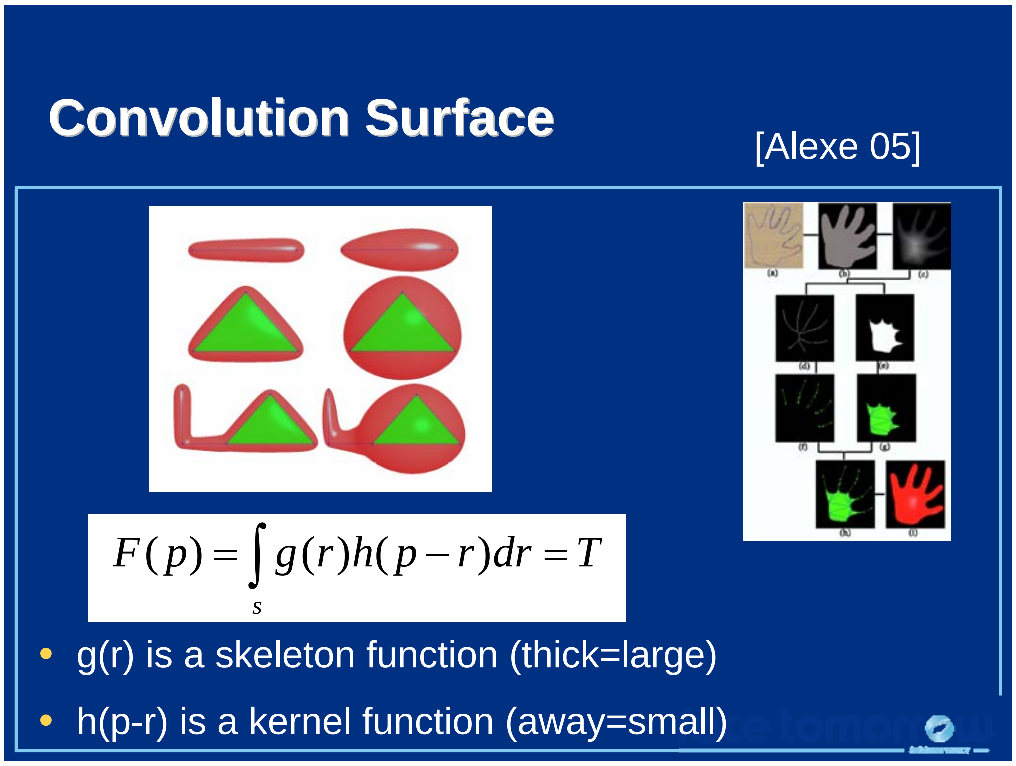
Voxel-based Method (Vteddy)

[Owada 03]



- Approximate the shape with circles along the axis.
- Replace the circles with spheres, and take union.
- Straightforward implementation

Owada proposed a voxel based method to generate 3D model from 2D contour. The system first obtains a medial axis by computing the distance field and tracing the local maximum. The system then places circles along the medial axis, the radius of each circle is the distance between the axis and the contour. The system then replaces the circles with spheres with the same radius and takes their union.



Convolution Surface

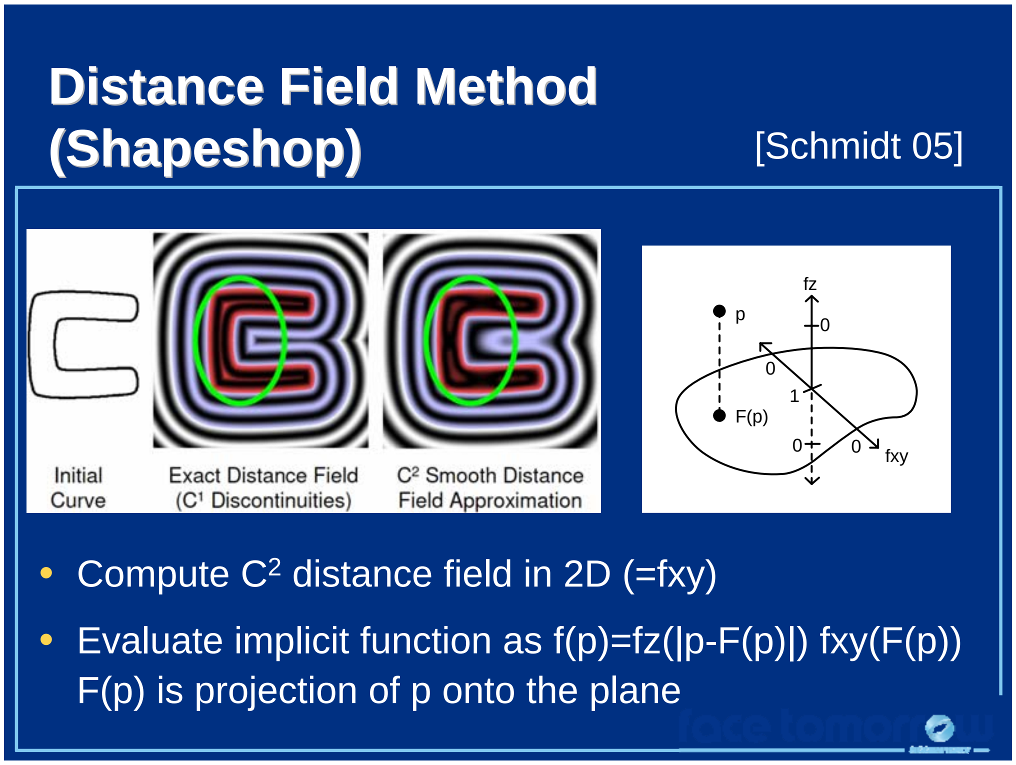
[Alexe 05]

$$F(p) = \int_S g(r)h(p-r)dr = T$$

- g(r) is a skeleton function (thick=large)
- h(p-r) is a kernel function (away=small)

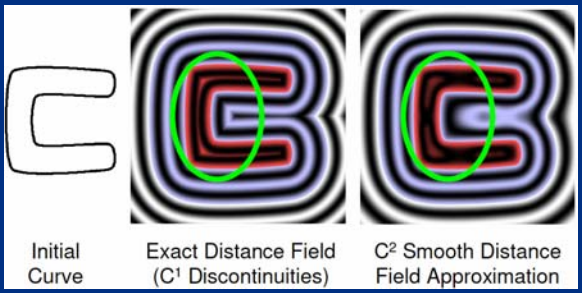
Alexe used convolution to construct implicit surfaces.

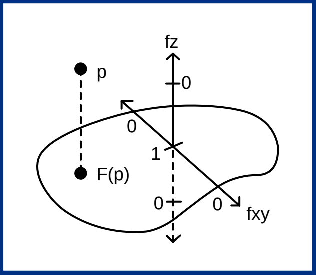
g(r) is a skeleton function. It become large at thick areas and small at thin areas. *h(p-r)* is a kernel function, which decays as it goes farther away from the center. This can generate a smooth surface.



Distance Field Method (Shapeshop)

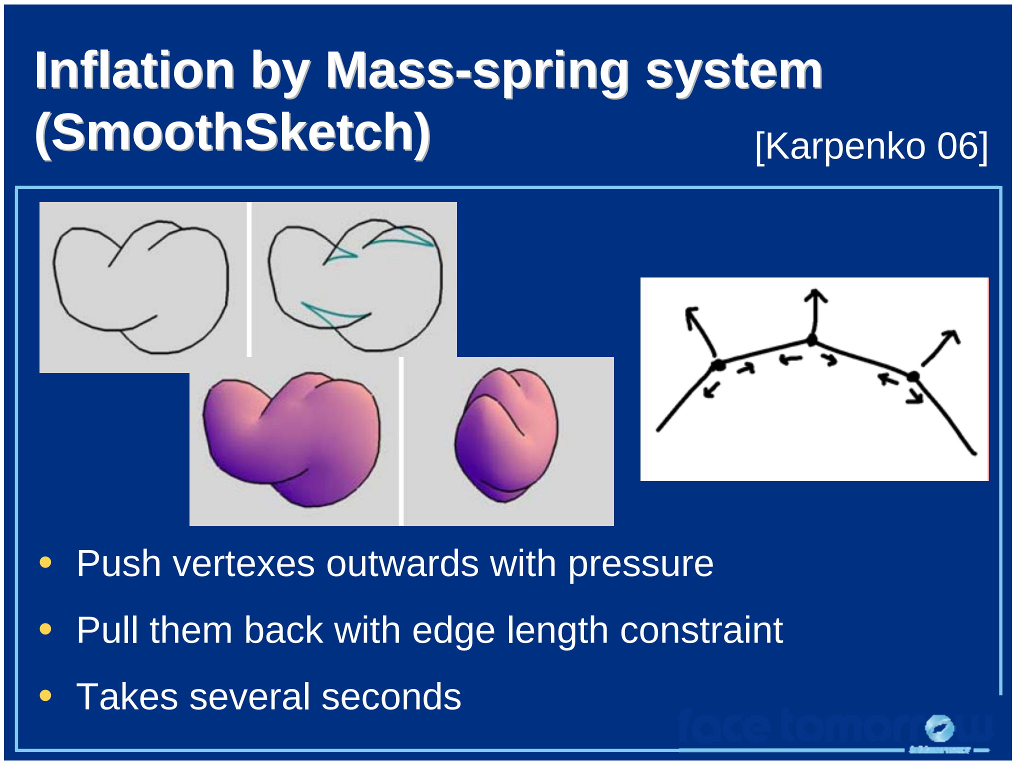
[Schmidt 05]


Initial Curve
Exact Distance Field (C^1 Discontinuities)
C^2 Smooth Distance Field Approximation



- Compute C^2 distance field in 2D (=fxy)
- Evaluate implicit function as f(p)=fz(|p-F(p)|) fxy(F(p)) F(p) is projection of p onto the plane

ShapeShop uses a distance field method. It first construct a C2 smooth distance field approximation on the input image plane. For any given point in the 3D space, the system evaluates the implicit function by multiplying the component perpendicular and parallel to the plane. The parallel component is the distance field approximation and, the perpendicular component is a decay function that gradually drops to 0 as it moves away from the plane.

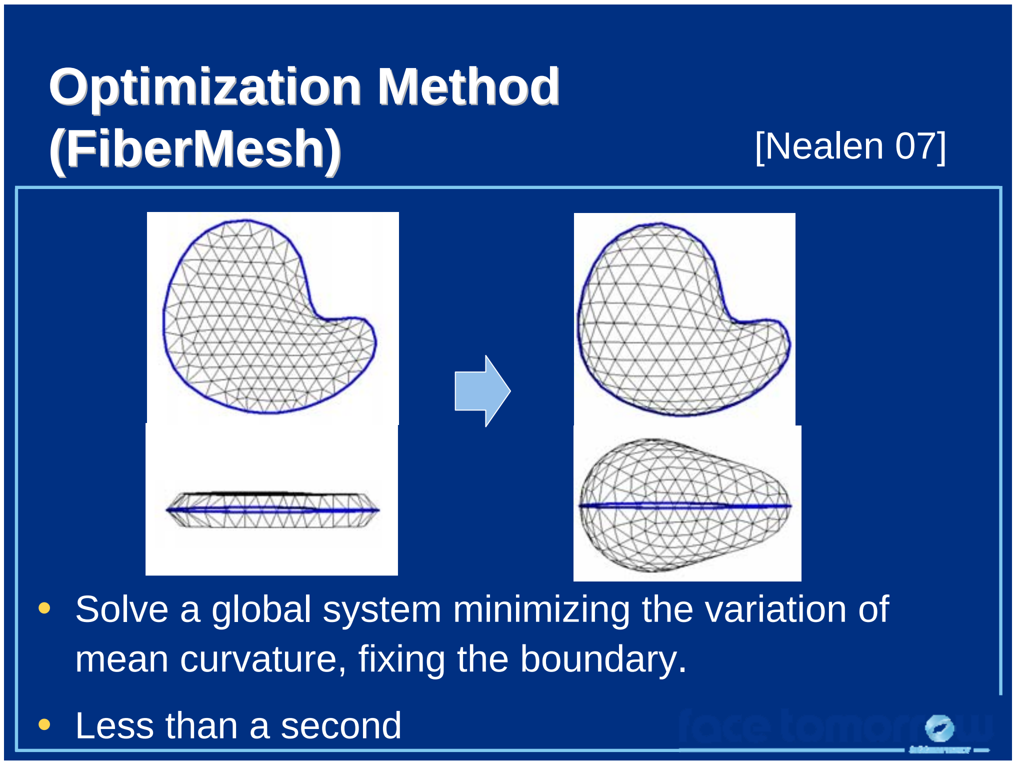


Inflation by Mass-spring system (SmoothSketch)

[Karpenko 06]

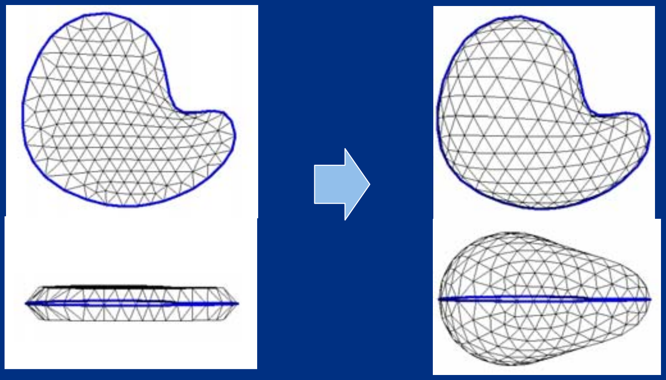
- Push vertexes outwards with pressure
- Pull them back with edge length constraint
- Takes several seconds

The SmootSketch system by Karpenko et. al generates a surface by using an ad-hoc mass-spring method. It pushes each vertex outwards mimicking pressure and pulls it back to keep the edge length, constraining the vertices along the input stroke. Their technical contribution is in the treatment of self-occluding contours, but we do not discuss it here.



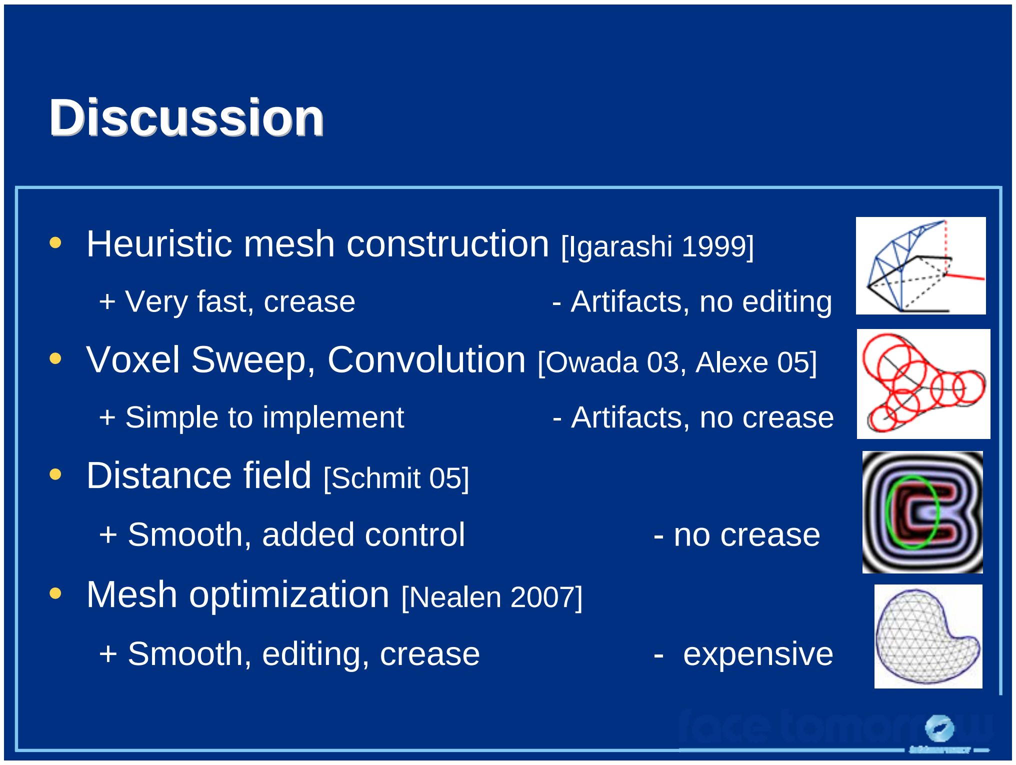
Optimization Method (FiberMesh)

[Nealen 07]



- Solve a global system minimizing the variation of mean curvature, fixing the boundary.
- Less than a second

The latest system, FiberMesh, generates smooth surfaces via optimization. The system solves a global system that minimizes the variation of mean curvature at all free vertices.

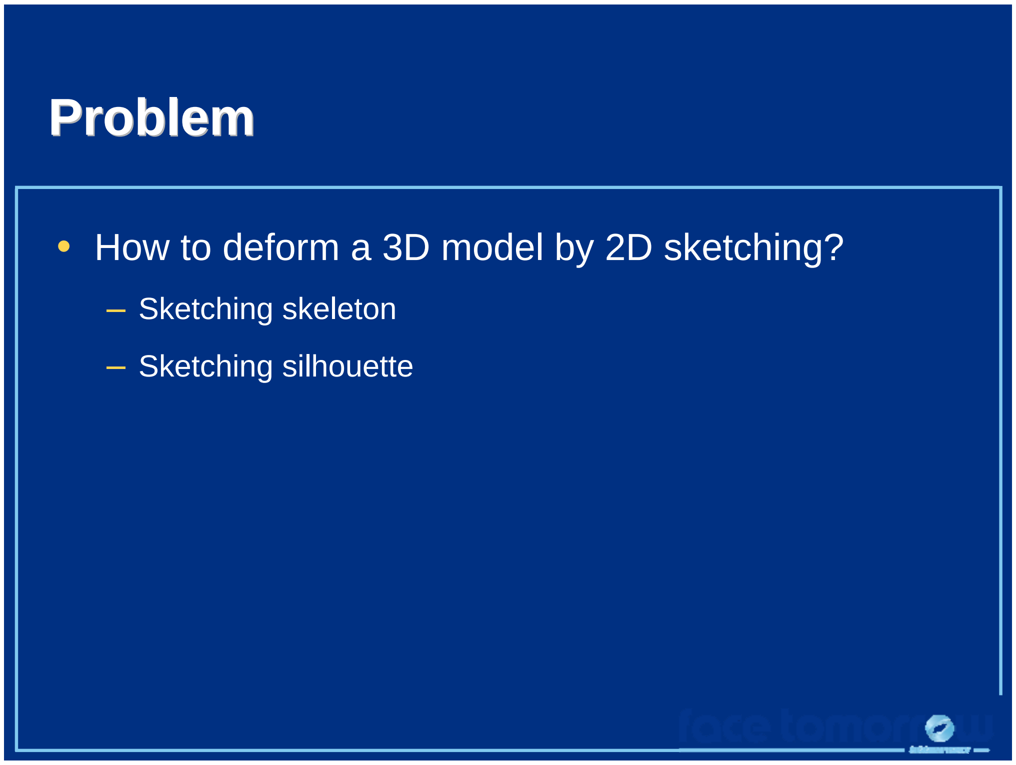


Discussion

- Heuristic mesh construction [Igarashi 1999]
  - + Very fast, crease - Artifacts, no editing
- Voxel Sweep, Convolution [Owada 03, Alexe 05]
  - + Simple to implement - Artifacts, no crease
- Distance field [Schmit 05]
  - + Smooth, added control - no crease
- Mesh optimization [Nealen 2007]
  - + Smooth, editing, crease - expensive

Each method has its own strength and weakness. Teddy’s method is very fast, but shows visible artifacts and does not support subsequent editing. Voxel sweep is easy to implement, but smoothness is limited by the voxel resolution. Implicit surfaces obtained via convolution or distance field computation are globally smooth, but it is difficult to represent sharp features such as crease. Finally, the surface optimization method supports both smooth and sharp features, but it is relatively expensive to compute.

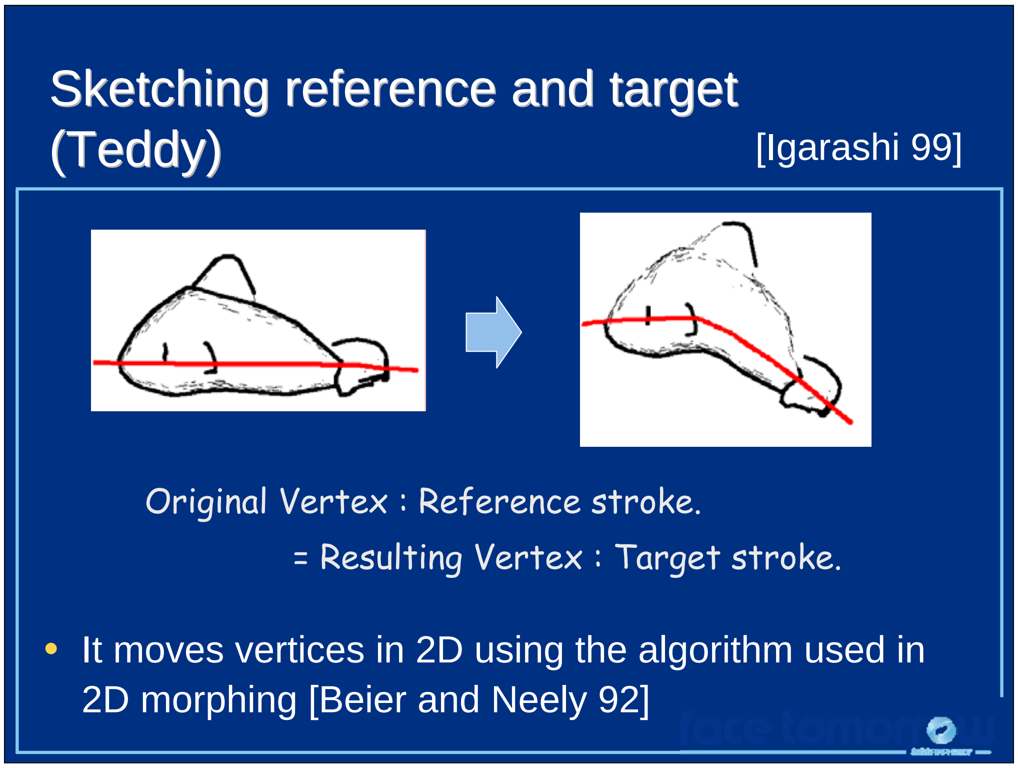
Deformation Techniques



Problem

- How to deform a 3D model by 2D sketching?
  - Sketching skeleton
  - Sketching silhouette

Deformation techniques modify the existing geometry according to the user's sketch.



Sketching reference and target (Teddy)

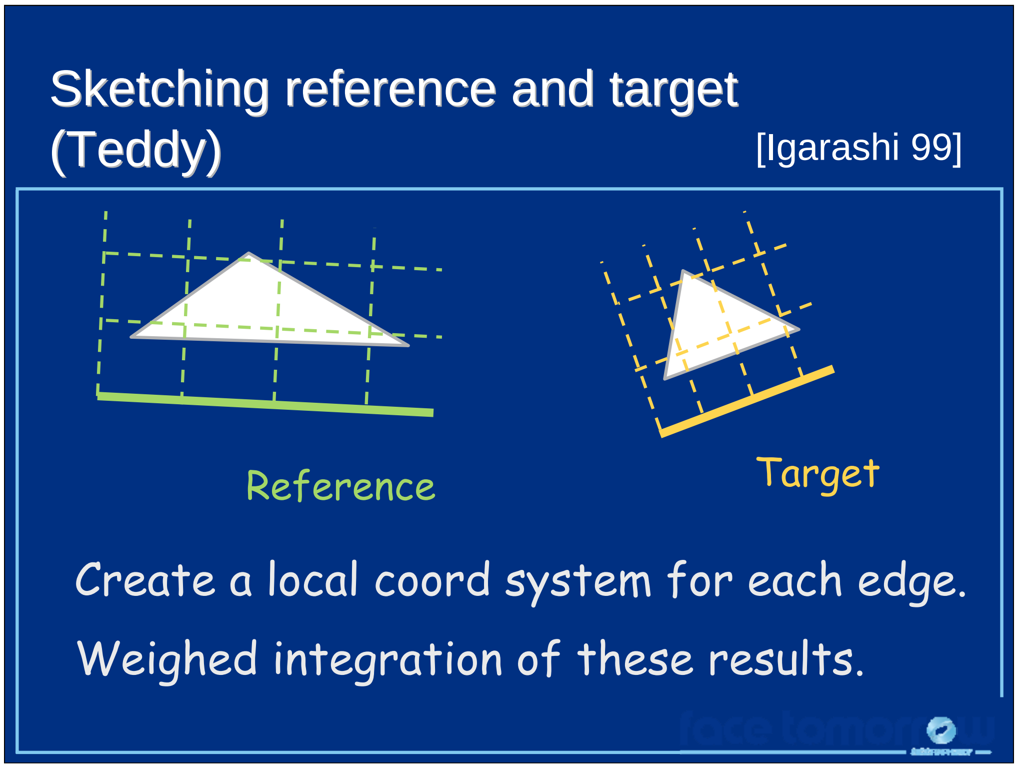
[Igarashi 99]

Original Vertex : Reference stroke.

= Resulting Vertex : Target stroke.

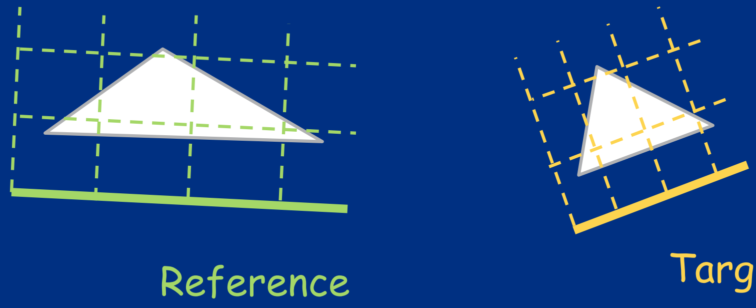
- It moves vertices in 2D using the algorithm used in 2D morphing [Beier and Neely 92]

The original Teddy system supported “bending”, taking a reference stroke and a target stroke. It deforms the shape so that the relation between the original shape and reference stroke is similar to the relation between the resulting shape and target stroke. It uses the algorithm used in 2D morphing.



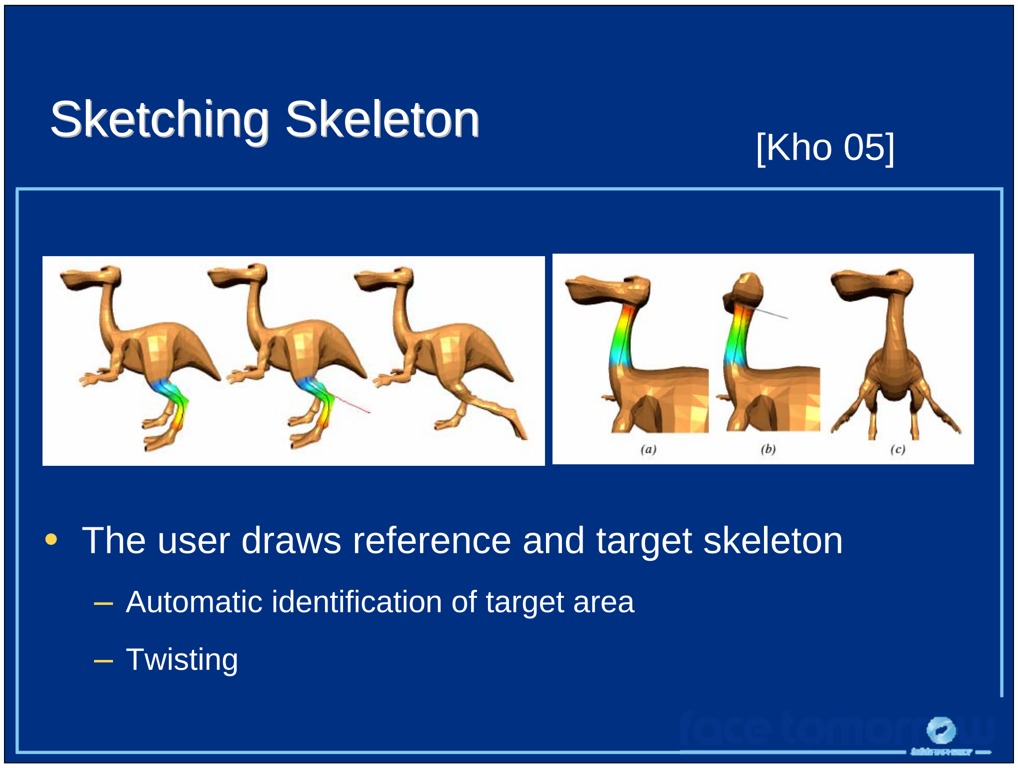
Sketching reference and target (Teddy)

[Igarashi 99]



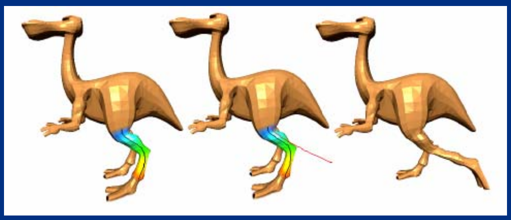
Create a local coord system for each edge.

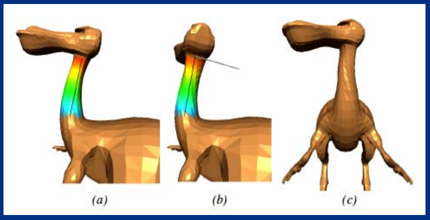
Weighed integration of these results.



Sketching Skeleton

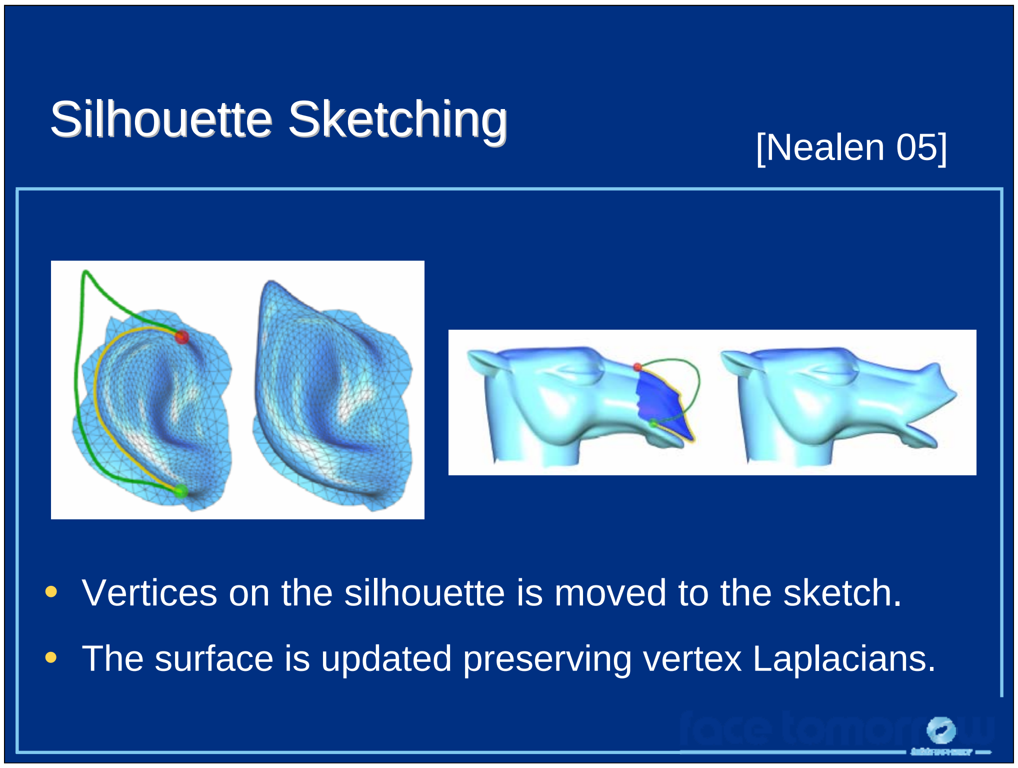
[Kho 05]





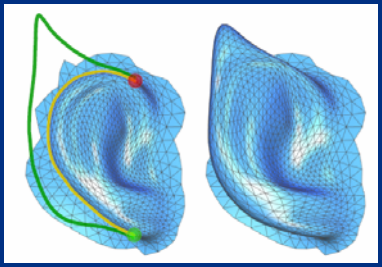
- The user draws reference and target skeleton
  - Automatic identification of target area
  - Twisting

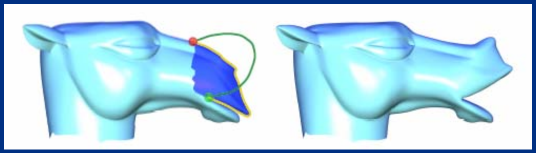
Kho’s system focuses on skeleton-based deformation.



Silhouette Sketching

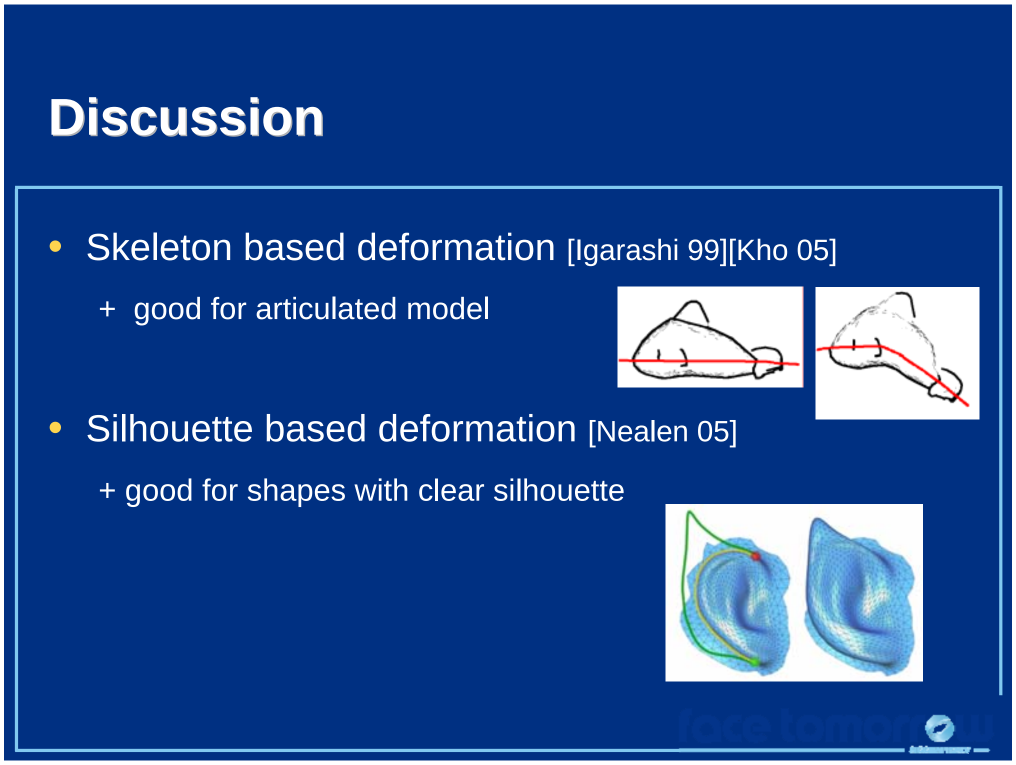
[Nealen 05]





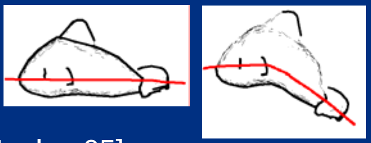
- Vertices on the silhouette is moved to the sketch.
- The surface is updated preserving vertex Laplacians.

Nealen's system allow the user to directly specify the desired shape of the silhouette. The system deforms the surface by minimizing the distance between the deformed silhouette and the user drawn stroke, while preserving the surface details in the form of laplacians.

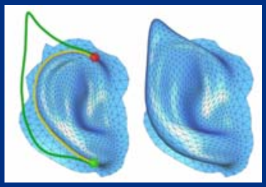


Discussion

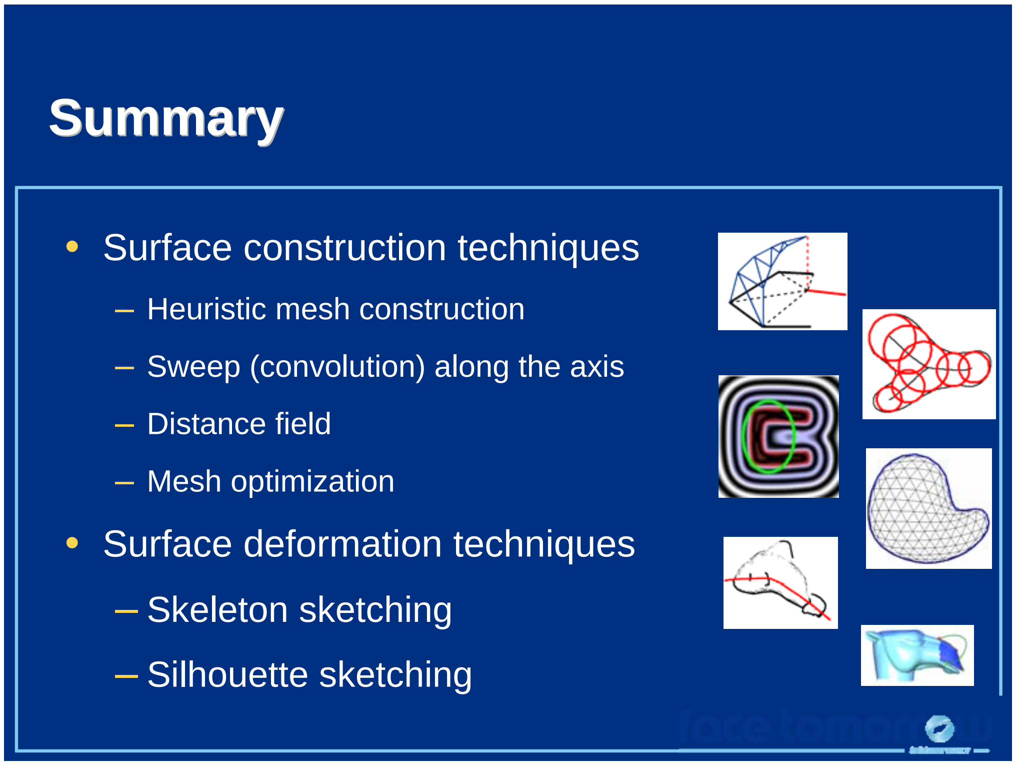
- Skeleton based deformation [Igarashi 99][Kho 05]
  - + good for articulated model



- Silhouette based deformation [Nealen 05]
  - + good for shapes with clear silhouette

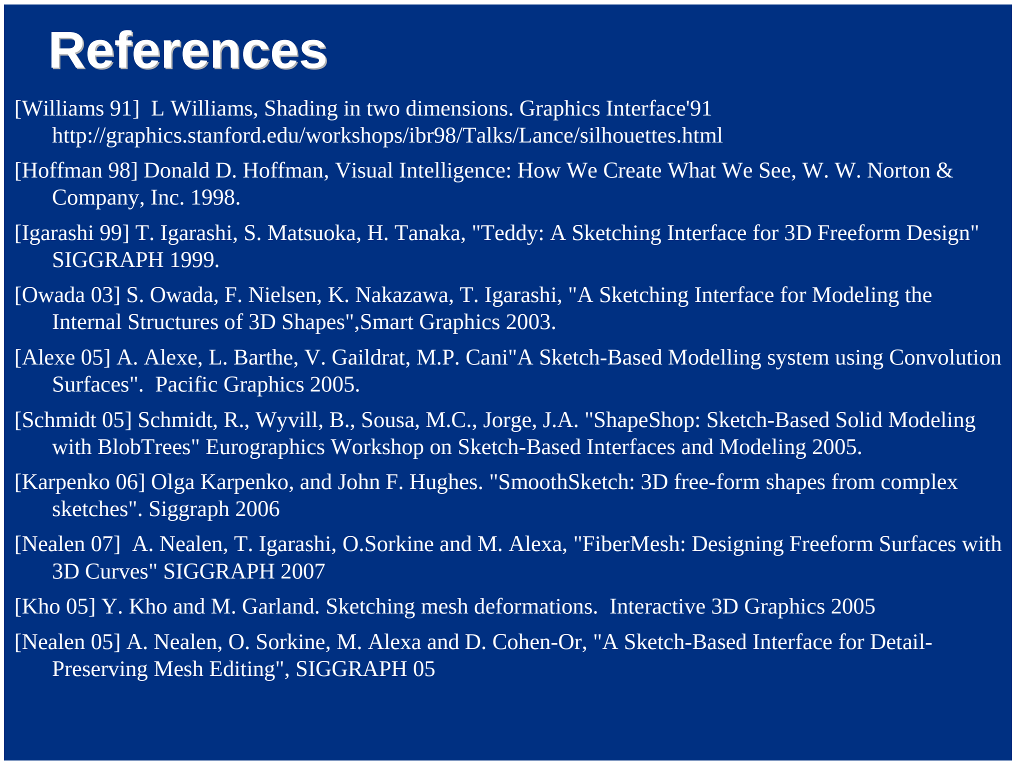


Skeleton-based methods are good for articulate models while silhouette-based methods are good for shapes with clear, smooth silhouettes.



Summary

- Surface construction techniques
  - Heuristic mesh construction
  - Sweep (convolution) along the axis
  - Distance field
  - Mesh optimization
- Surface deformation techniques
  - Skeleton sketching
  - Silhouette sketching



References

[Williams 91] L Williams, Shading in two dimensions. Graphics Interface'91 http://graphics.stanford.edu/workshops/ibr98/Talks/Lance/silhouettes.html

[Hoffman 98] Donald D. Hoffman, Visual Intelligence: How We Create What We See, W. W. Norton & Company, Inc. 1998.

[Igarashi 99] T. Igarashi, S. Matsuoka, H. Tanaka, "Teddy: A Sketching Interface for 3D Freeform Design" SIGGRAPH 1999.

[Owada 03] S. Owada, F. Nielsen, K. Nakazawa, T. Igarashi, "A Sketching Interface for Modeling the Internal Structures of 3D Shapes",Smart Graphics 2003.

[Alexe 05] A. Alexe, L. Barthe, V. Gaildrat, M.P. Cani"A Sketch-Based Modelling system using Convolution Surfaces". Pacific Graphics 2005.

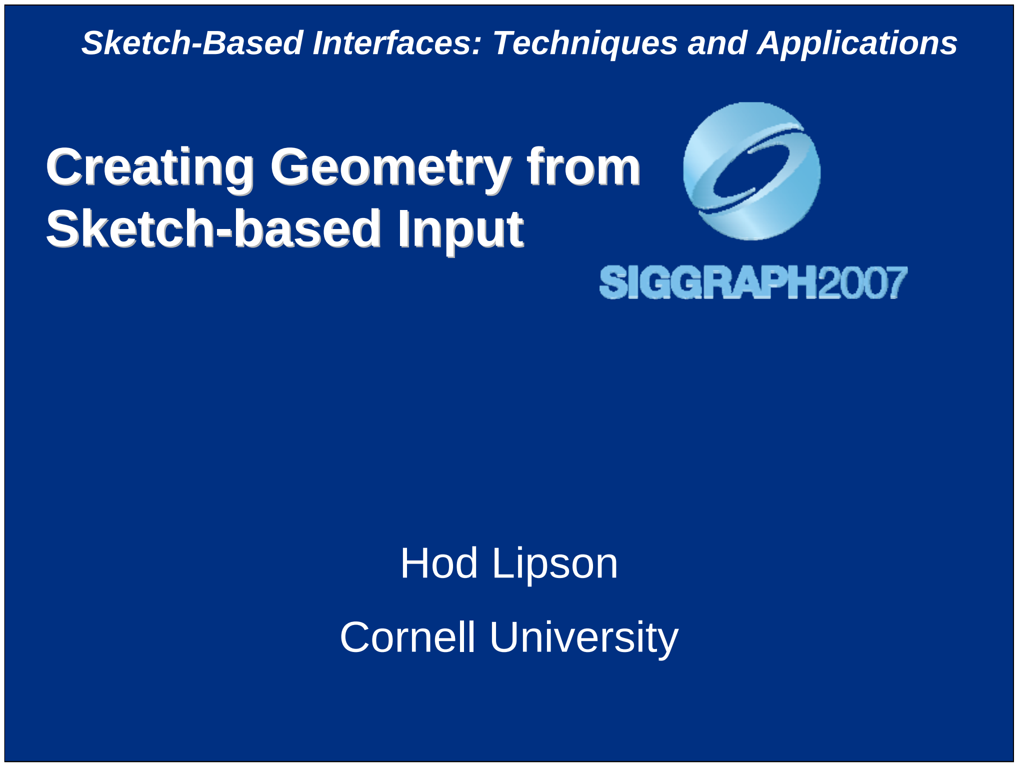
[Schmidt 05] Schmidt, R., Wyvill, B., Sousa, M.C., Jorge, J.A. "ShapeShop: Sketch-Based Solid Modeling with BlobTrees" Eurographics Workshop on Sketch-Based Interfaces and Modeling 2005.

[Karpenko 06] Olga Karpenko, and John F. Hughes. "SmoothSketch: 3D free-form shapes from complex sketches". Siggraph 2006

[Nealen 07] A. Nealen, T. Igarashi, O.Sorkine and M. Alexa, "FiberMesh: Designing Freeform Surfaces with 3D Curves" SIGGRAPH 2007

[Kho 05] Y. Kho and M. Garland. Sketching mesh deformations. Interactive 3D Graphics 2005

[Nealen 05] A. Nealen, O. Sorkine, M. Alexa and D. Cohen-Or, "A Sketch-Based Interface for Detail-Preserving Mesh Editing", SIGGRAPH 05



Sketch-Based Interfaces: Techniques and Applications

Creating Geometry from Sketch-based Input

SIGGRAPH2007

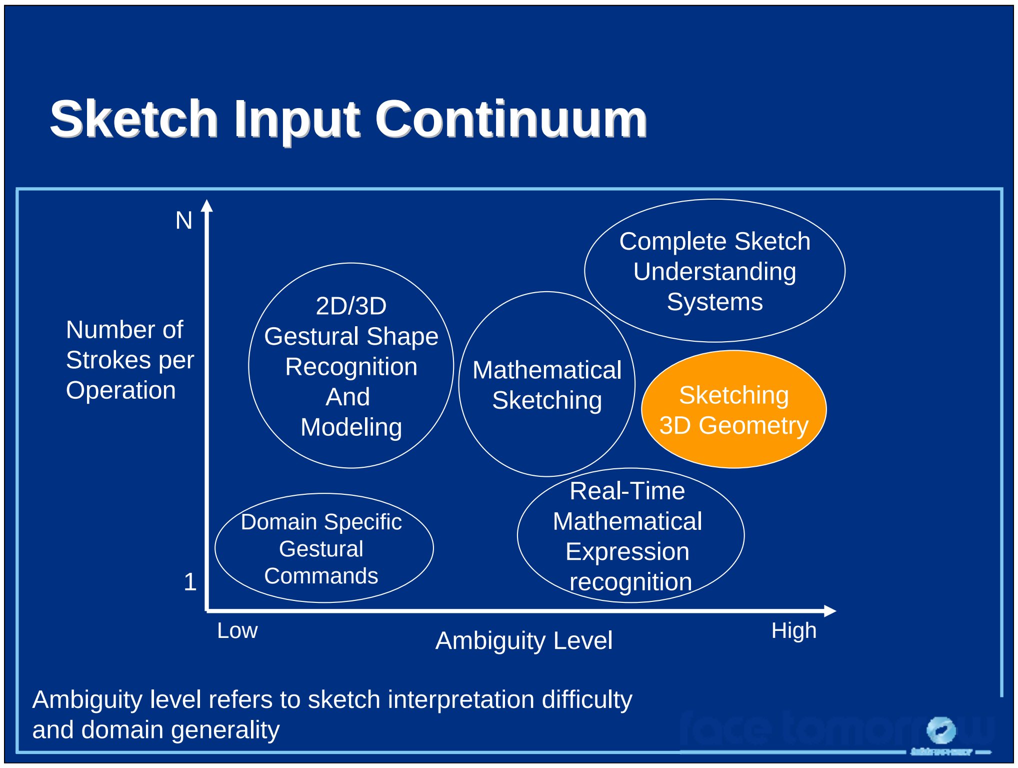
Hod Lipson

Cornell University

Creating Geometry from Sketch-based Input (60 min)

Hod Lipson, Assistant Professor
Department of Mechanical & Aerospace Engineering
216 Upson Hall, Cornell University
Ithaca NY 14853, USA

Email: hod.lipson@cornell.edu
http://www.mae.cornell.edu/lipson



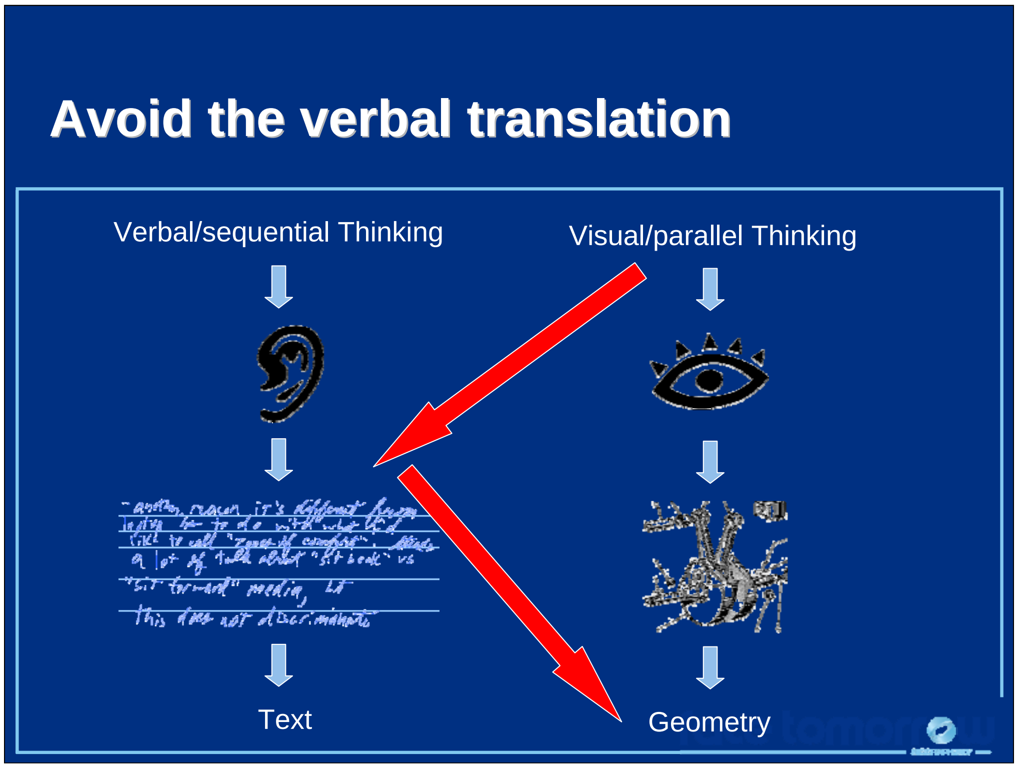
Sketch Input Continuum

N
1
Number of Strokes per Operation
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Complete Sketch Understanding Systems
Sketching 3D Geometry
Domain Specific Gestural Commands
Real-Time Mathematical Expression recognition
Low
Ambiguity Level
High

Ambiguity level refers to sketch interpretation difficulty and domain generality

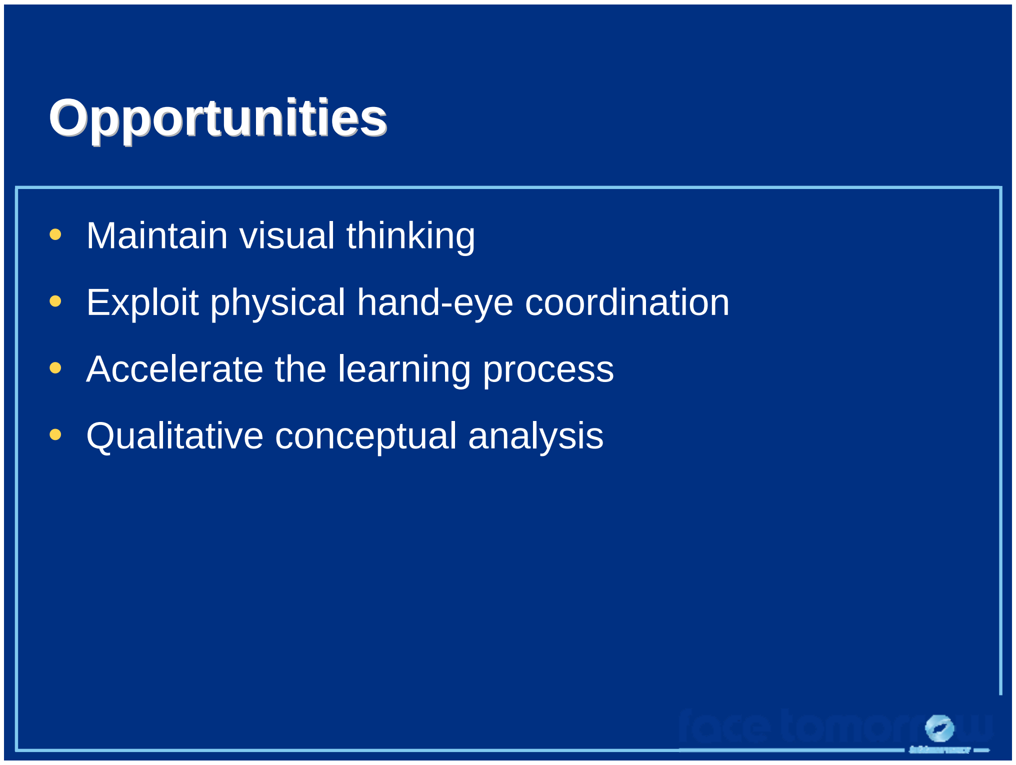
Earlier today we outlined a number of concepts for using sketch-based interfaces for modeling and design. I'd like to return to talk about one particular application and the challenges and opportunities involved in this application area.

Sketching to produce 3D geometry is particularly challenging because it involves several levels of ambiguity: Ambiguity inherent in any sketch based application due to inaccuracy of the sketch strokes and lack of a direct interface, but then also ambiguity related to the 2D to 3D mapping. Compounded to this, we have a scaling problem due to the relatively complexity of objects to be inferred. And so we end up being at the upper-right corner of the complexity-ambiguity space.



Avoid the verbal translation

Let’s recall that sketch understanding is really about skipping the “forced” translation from the parallel, visual thinking mode which characterizes design, into the verbal, sequential process that is typically needed to enter it into a computer.

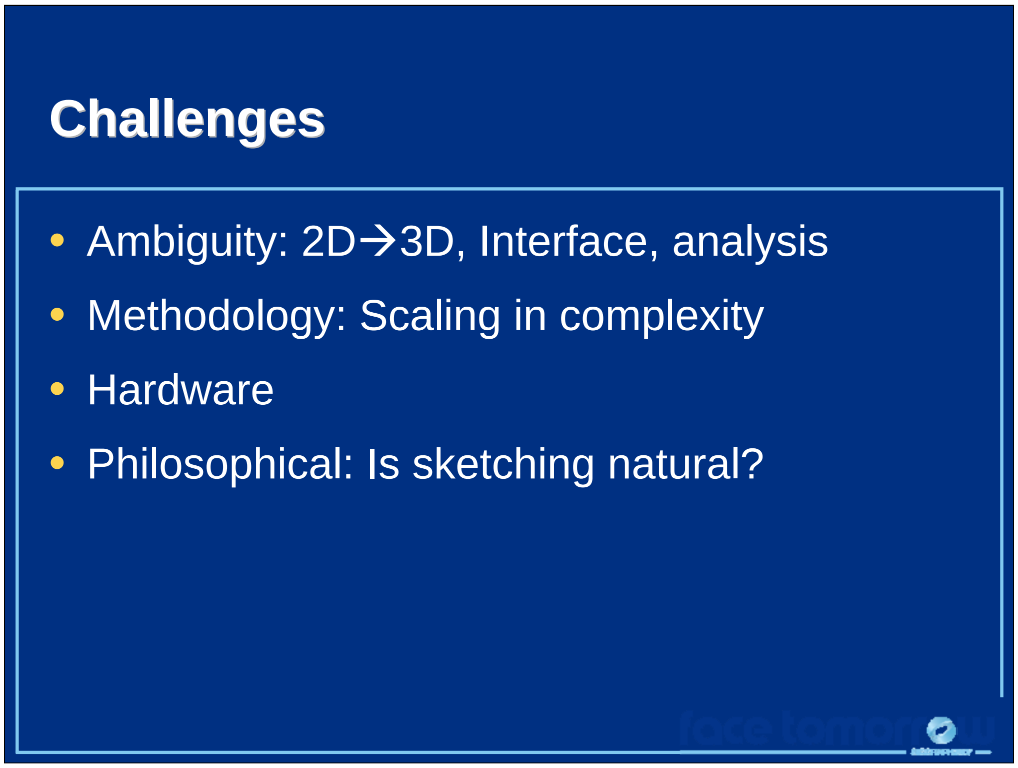


Opportunities

- Maintain visual thinking
- Exploit physical hand-eye coordination
- Accelerate the learning process
- Qualitative conceptual analysis

Done well, staying at the “visual side” of the brain could have several profound advantages, such as exploiting the rich-hand-eye coordination, thereby flattening the learning curve. Takeo Igarashi showed how even untrained children can design relatively complex 3D shapes.

In engineering, this has an additional advantage: We can bring computer aided engineering (CAE) tools to bear at the initial stages of design – where most of the critical decisions are made. Today most, of these types of analyses are done at a fairly late stage of the design when a considerable amount of effort has already been invested in creating detailed models.

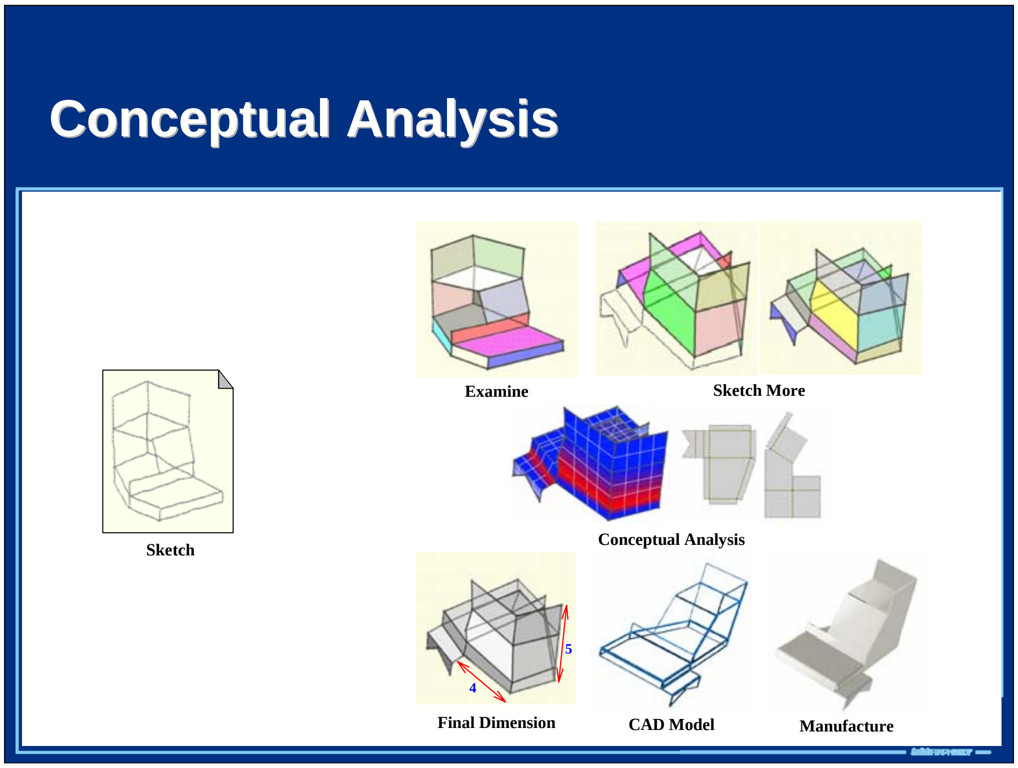


Challenges

- Ambiguity: 2D→3D, Interface, analysis
- Methodology: Scaling in complexity
- Hardware
- Philosophical: Is sketching natural?

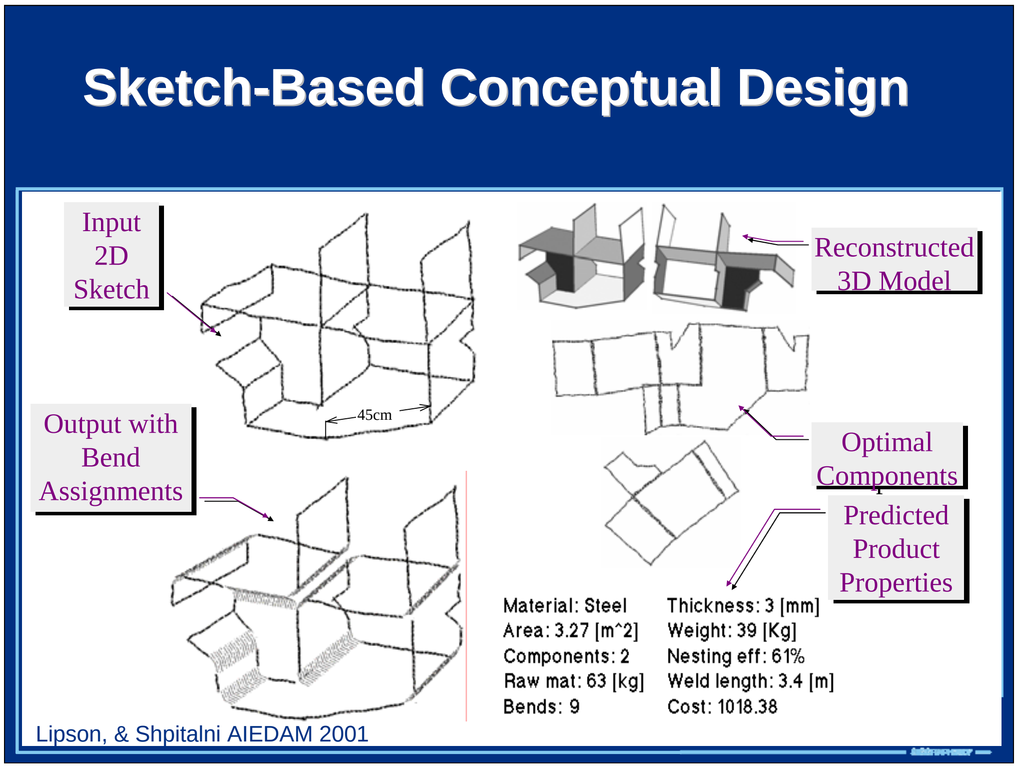
The two main challenges are those outlined on the first slide: Dealing with ambiguity and dealing with complexity.

These are compounded by fundamental hardware issues and deeper questions about the naturalness of sketching – is it really natural, or is it an artifact of the way technology evolved.



Conceptual Analysis

So lets go back to the ideal sketch-based design. We'd like a CAD system to act like an "expert colleague" at a whiteboard. We'd like to get feedback on our design before investing much into it. For example, imagine you can sketch an object, and you asked an expert to look at your sketch and provide feedback on potential manufacturing and stress issues before you invested in details. If a colleague expert could do it, why can't a computer do it? That's not the case now for a number of issues: Sketch interpretation is one, and another is how to analyze "rough" models: How to provide reliable analysis from an unreliable model.

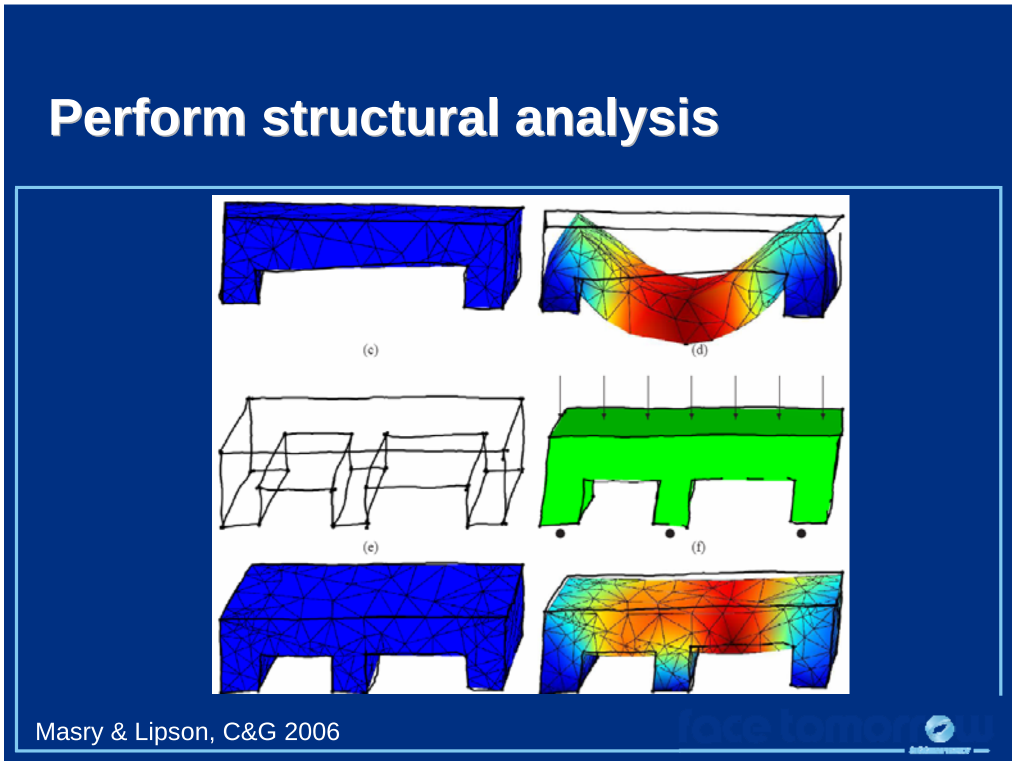


Sketch-Based Conceptual Design

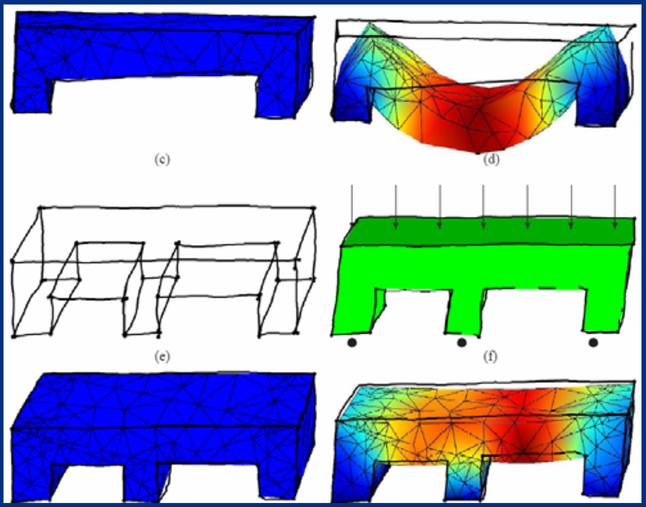
Material: Steel	Thickness: 3 [mm]
Area: 3.27 [m^2]	Weight: 39 [Kg]
Components: 2	Nesting eff: 61%
Raw mat: 63 [kg]	Weld length: 3.4 [m]
Bends: 9	Cost: 1018.38

Lipson, & Shpitalni AIEDAM 2001

Here is an example…

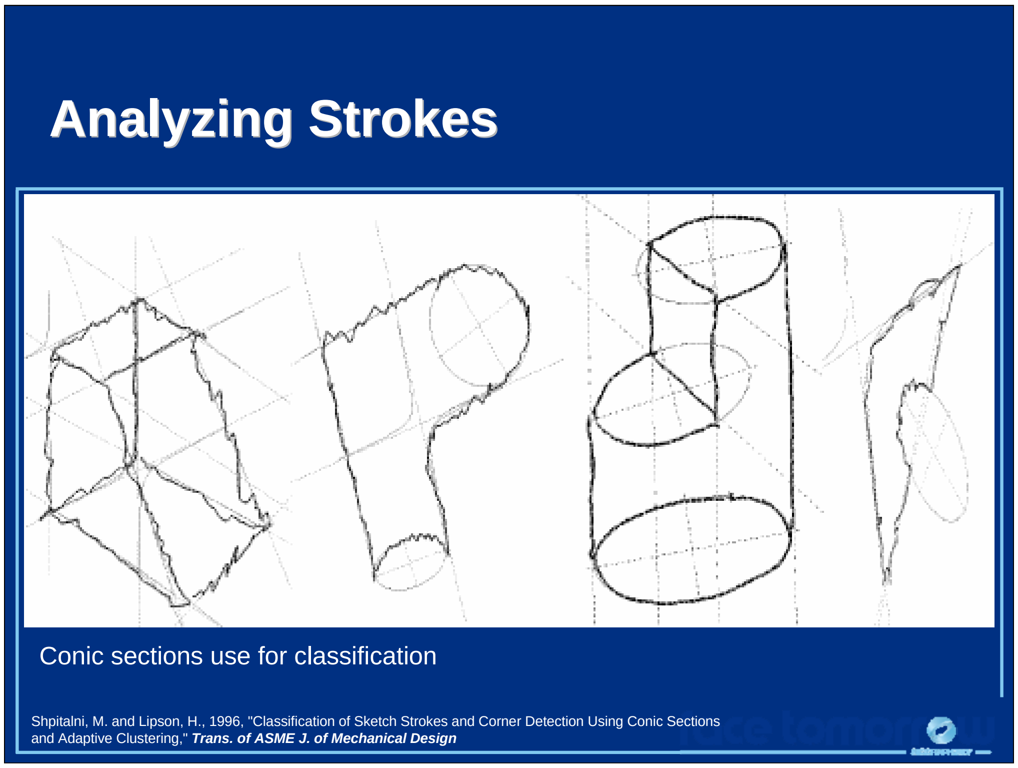


Perform structural analysis

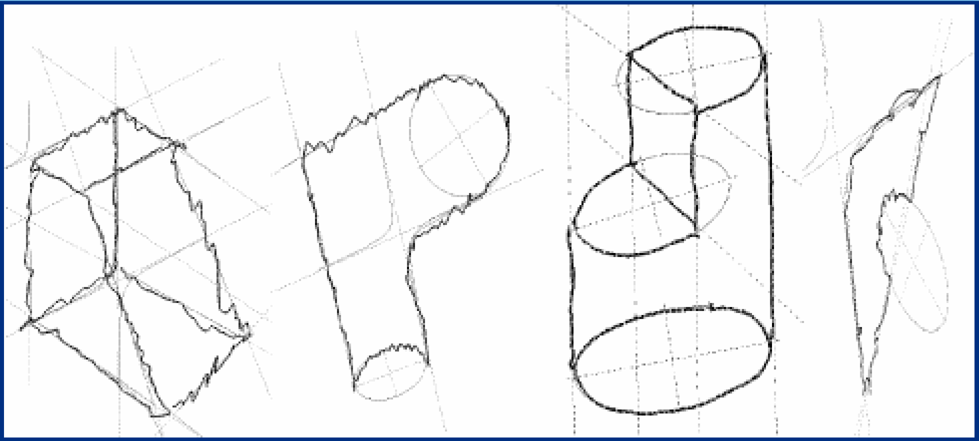


Masry & Lipson, C&G 2006

And another one…



Analyzing Strokes

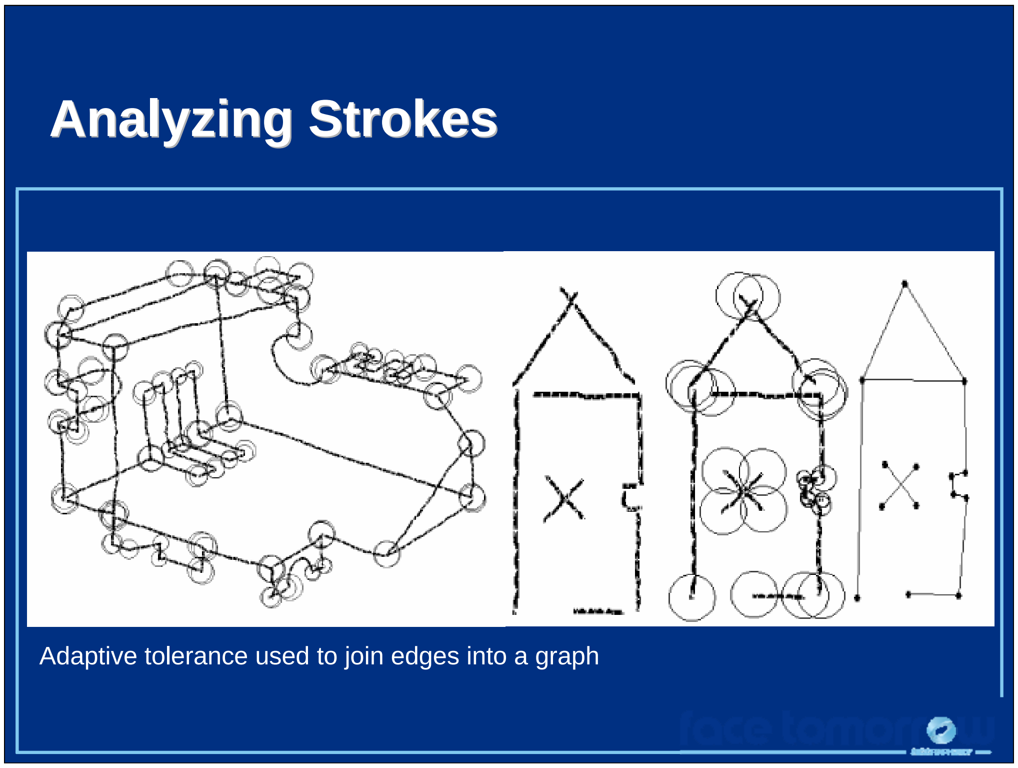


Conic sections use for classification

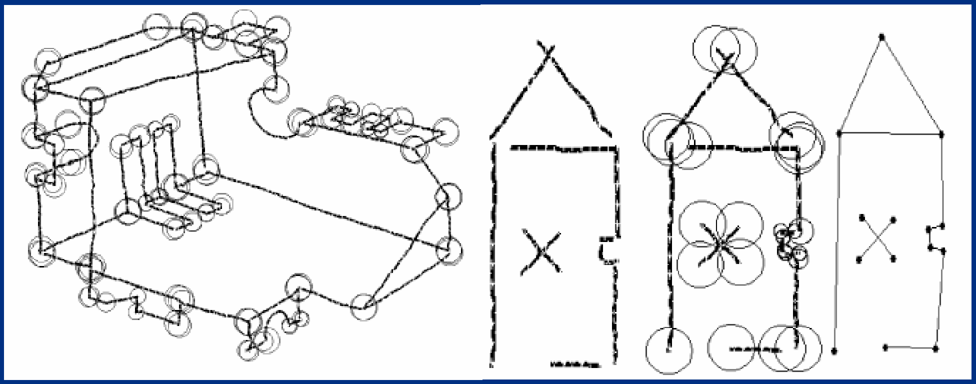
Shpitalni, M. and Lipson, H., 1996, "Classification of Sketch Strokes and Corner Detection Using Conic Sections and Adaptive Clustering," ***Trans. of ASME J. of Mechanical Design***

Lets look at the way it works.

First, there is an issue of stroke recording and classification. These issues are addressed by many systems, but here we have a particular interest in straight lines and conic sections.

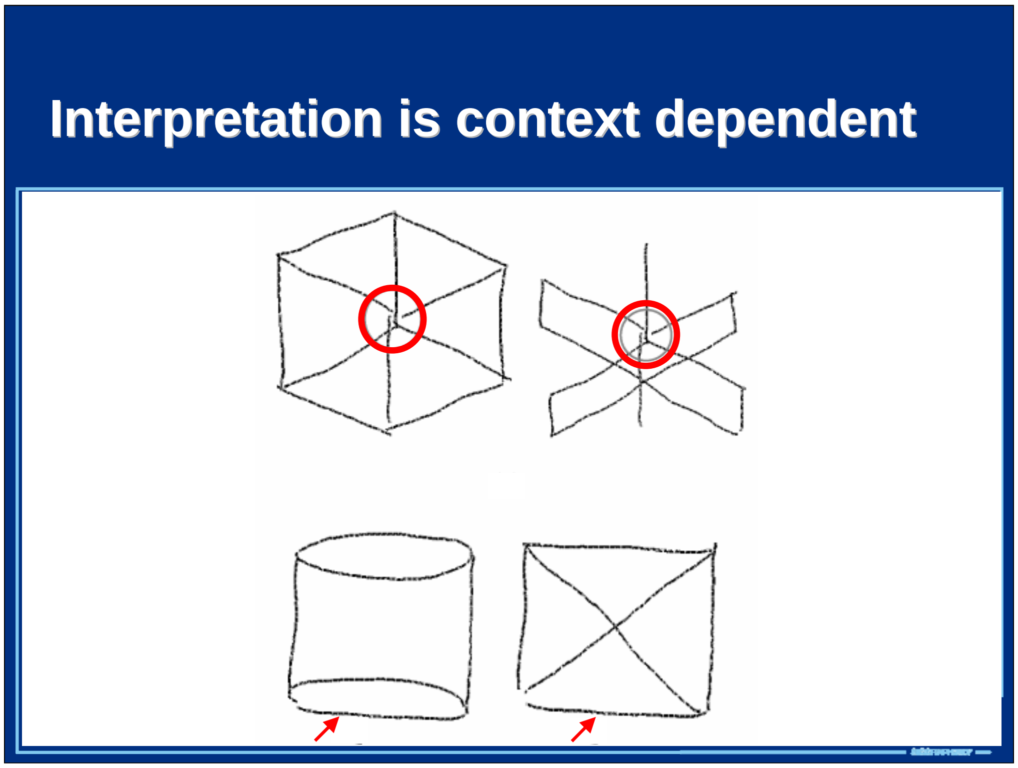


Analyzing Strokes

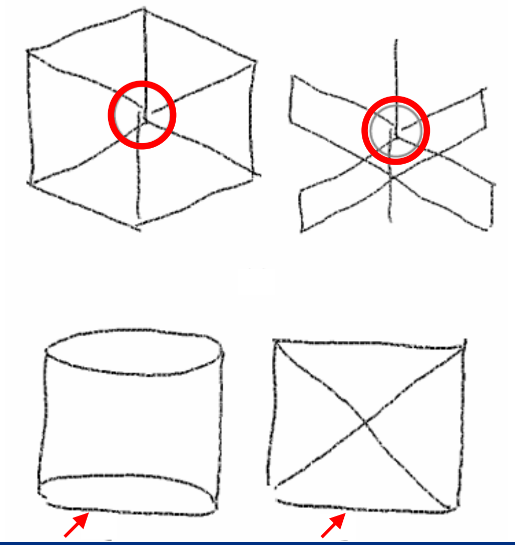


Adaptive tolerance used to join edges into a graph

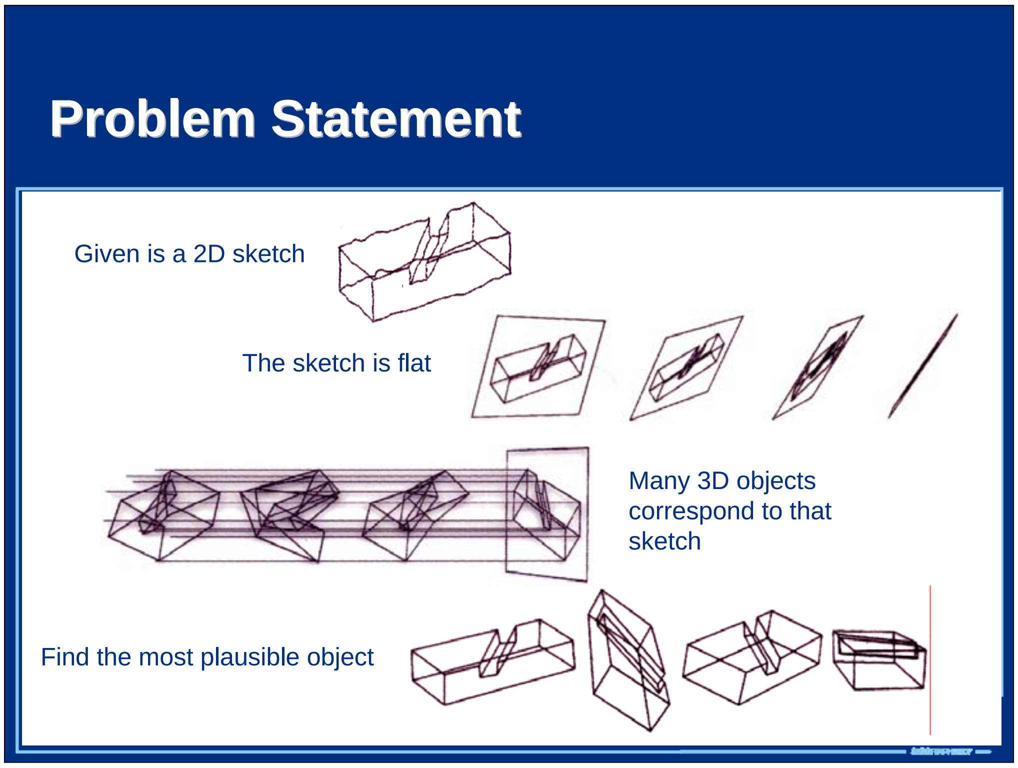
Evan linking the strokes at their ends is a challenge, as several other speakers have alluded to. There are multi-resolution decisions that need to be made adaptively…



Interpretation is context dependent



And possibly based on context.



Problem Statement

Given is a 2D sketch

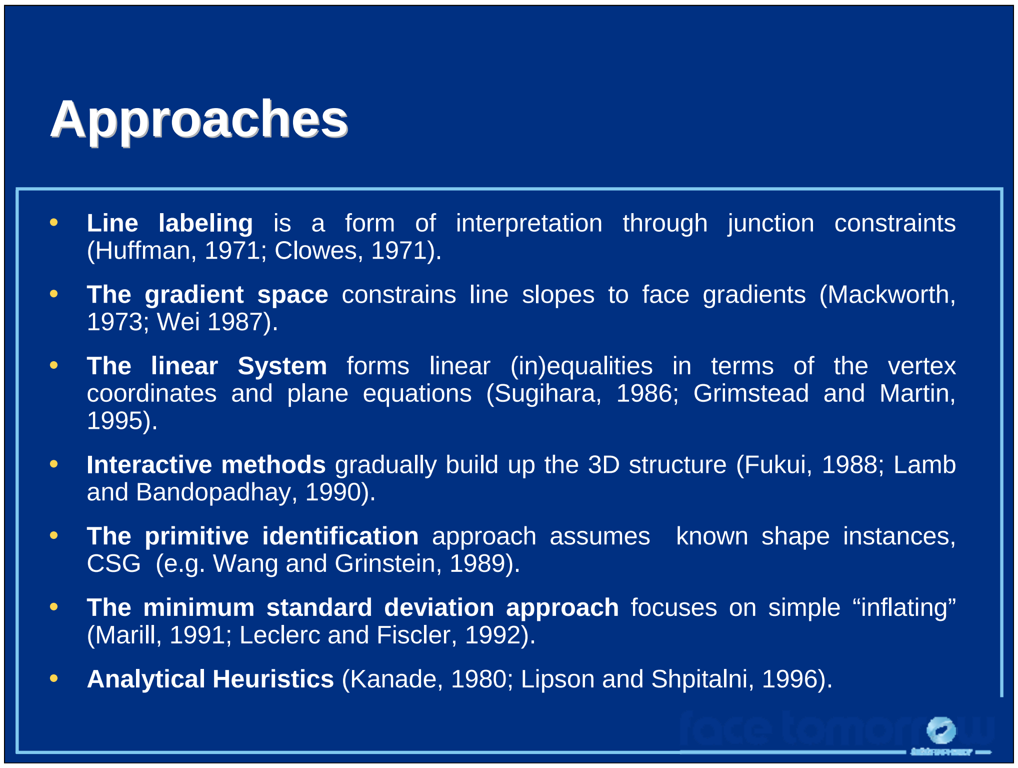
The sketch is flat

Many 3D objects correspond to that sketch

Find the most plausible object

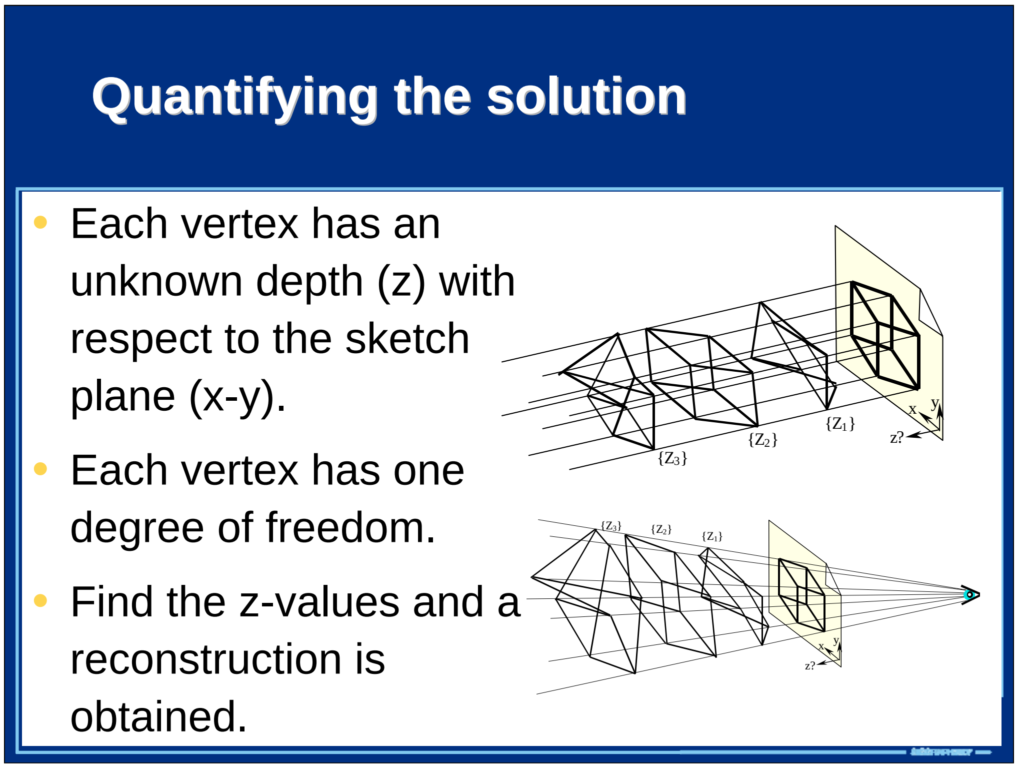
Once the basic classification and joining of strokes is complete, we end up having a graph embedded in 2D, the real problem begins:

The sketch is flat, and it represents, in theory, and infinite set of 3D objects. Yet we can easily reach a consensus about the most plausible object it represents. How do we do this? There are clearly som interesting issues here that have to do with the human visual system. We make some assumptions.



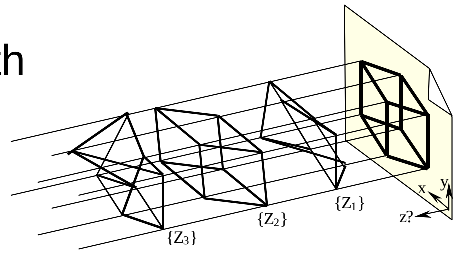
Approaches

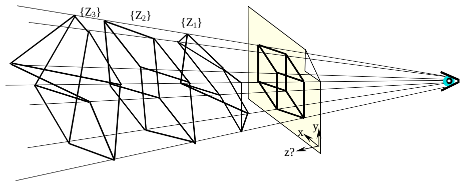
- **Line labeling** is a form of interpretation through junction constraints (Huffman, 1971; Clowes, 1971).
- **The gradient space** constrains line slopes to face gradients (Mackworth, 1973; Wei 1987).
- **The linear System** forms linear (in)equalities in terms of the vertex coordinates and plane equations (Sugihara, 1986; Grimstead and Martin, 1995).
- **Interactive methods** gradually build up the 3D structure (Fukui, 1988; Lamb and Bandopadhay, 1990).
- **The primitive identification** approach assumes known shape instances, CSG (e.g. Wang and Grinstein, 1989).
- **The minimum standard deviation approach** focuses on simple “inflating” (Marill, 1991; Leclerc and Fiscler, 1992).
- **Analytical Heuristics** (Kanade, 1980; Lipson and Shpitalni, 1996).



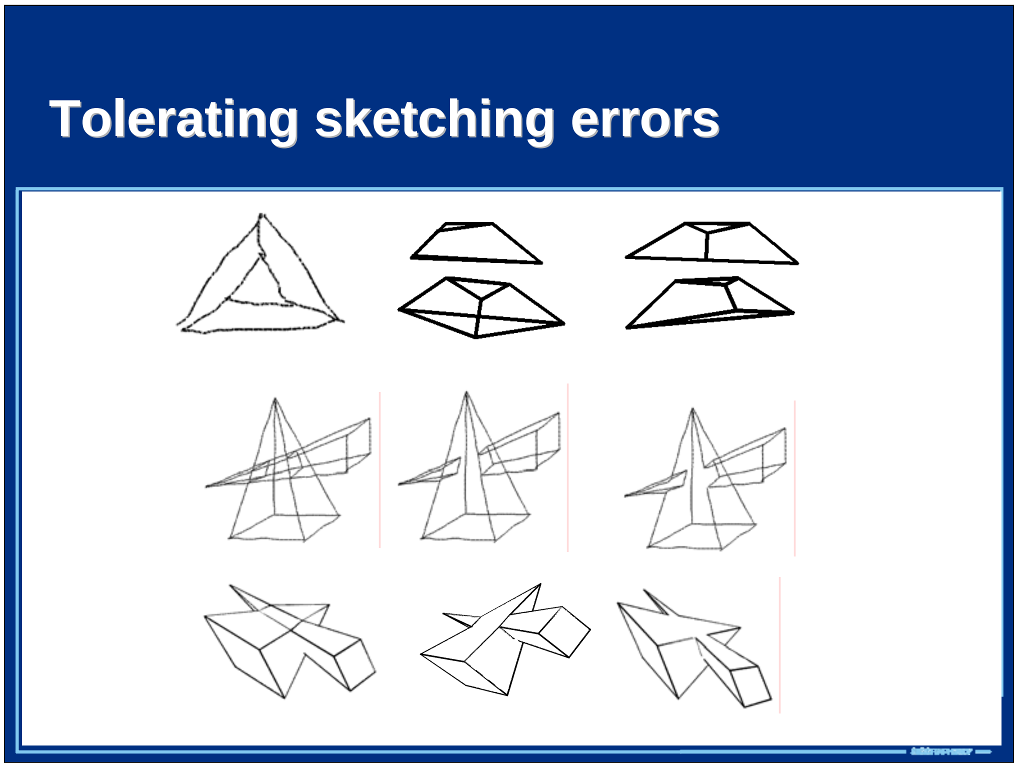
Quantifying the solution

- Each vertex has an unknown depth (z) with respect to the sketch plane (x-y).
- Each vertex has one degree of freedom.
- Find the z-values and a reconstruction is obtained.

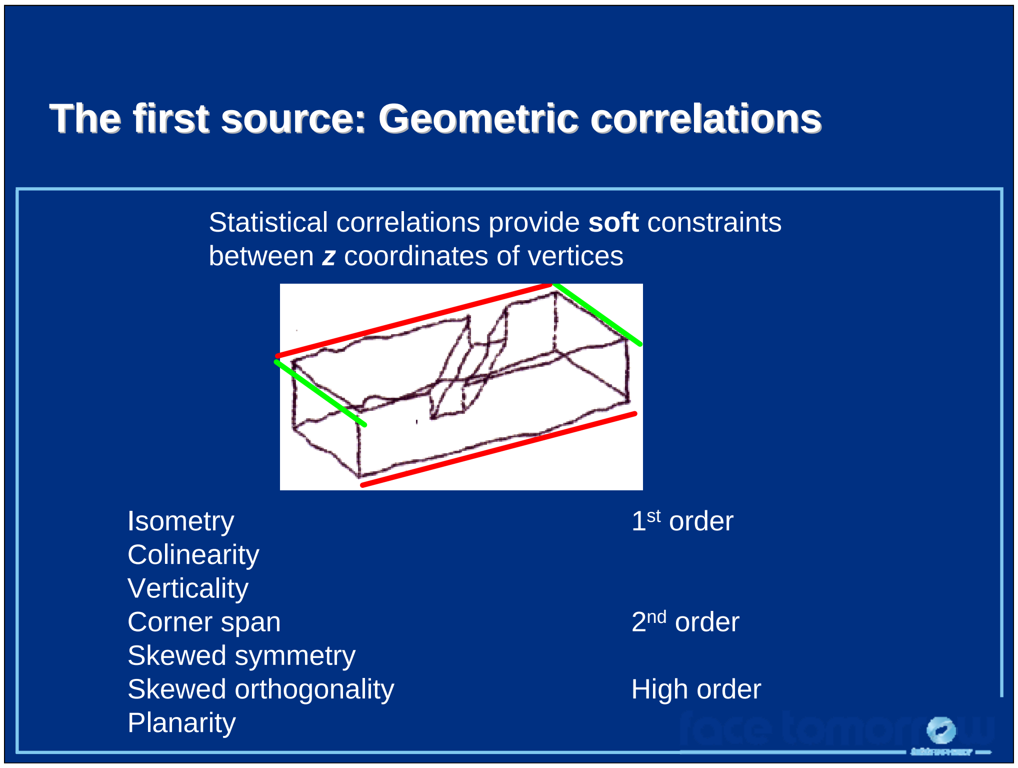




We can quantify the question as an optimization/search problem.

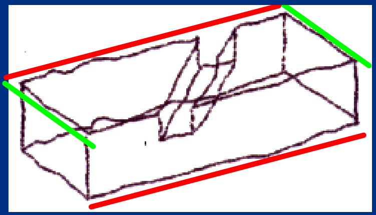


Tolerating sketching errors

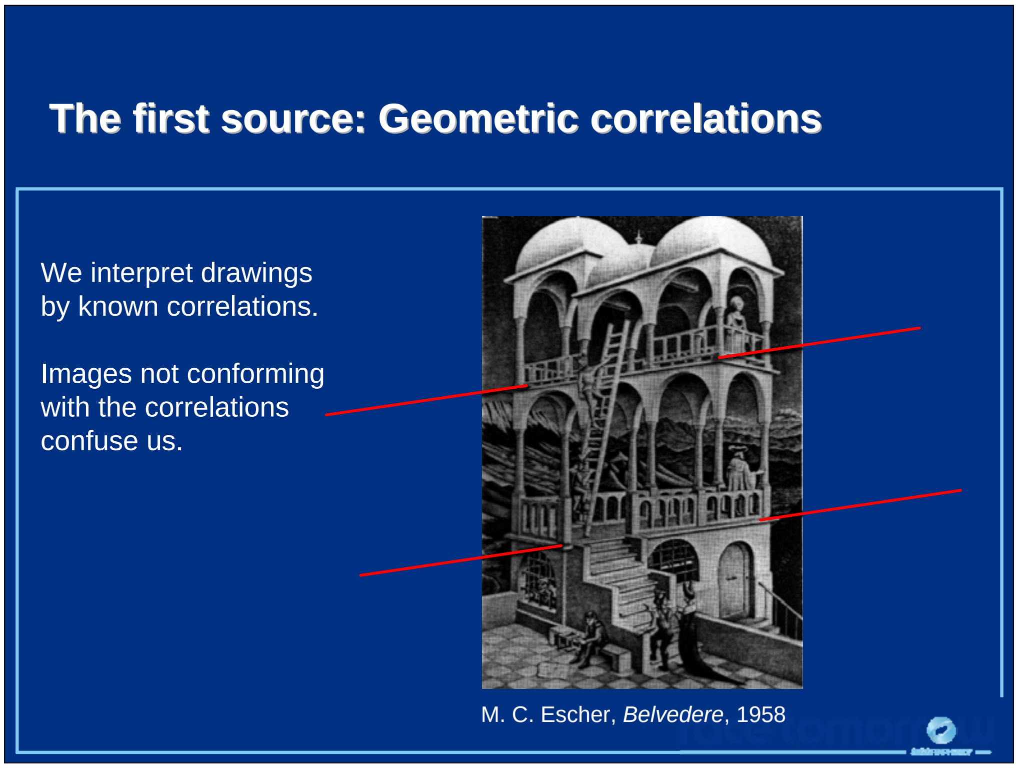


The first source: Geometric correlations

Statistical correlations provide **soft** constraints between ***z*** coordinates of vertices



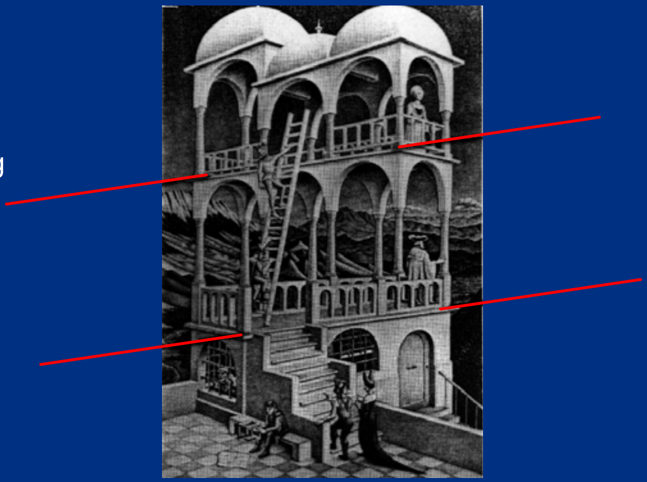
Isometry	1st order
Colinearity	
Verticality	
Corner span	2nd order
Skewed symmetry	
Skewed orthogonality	High order
Planarity	



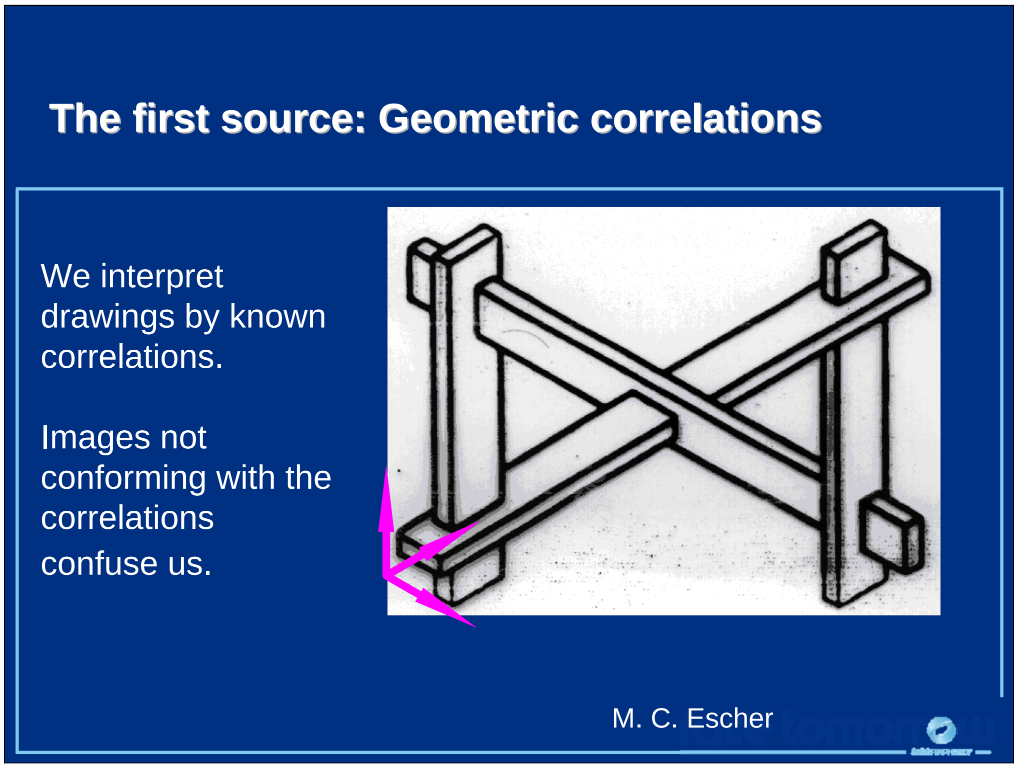
The first source: Geometric correlations

We interpret drawings by known correlations.

Images not conforming with the correlations confuse us.



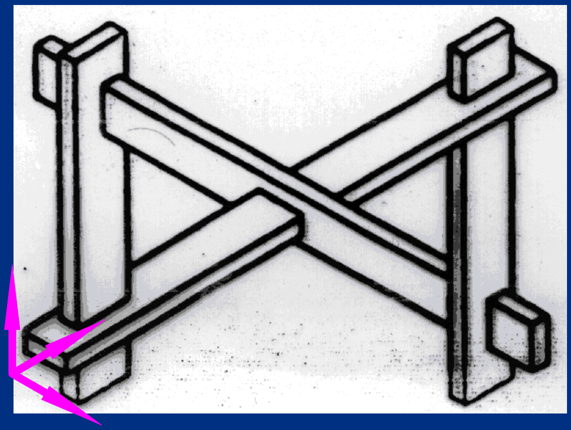
M. C. Escher, *Belvedere*, 1958



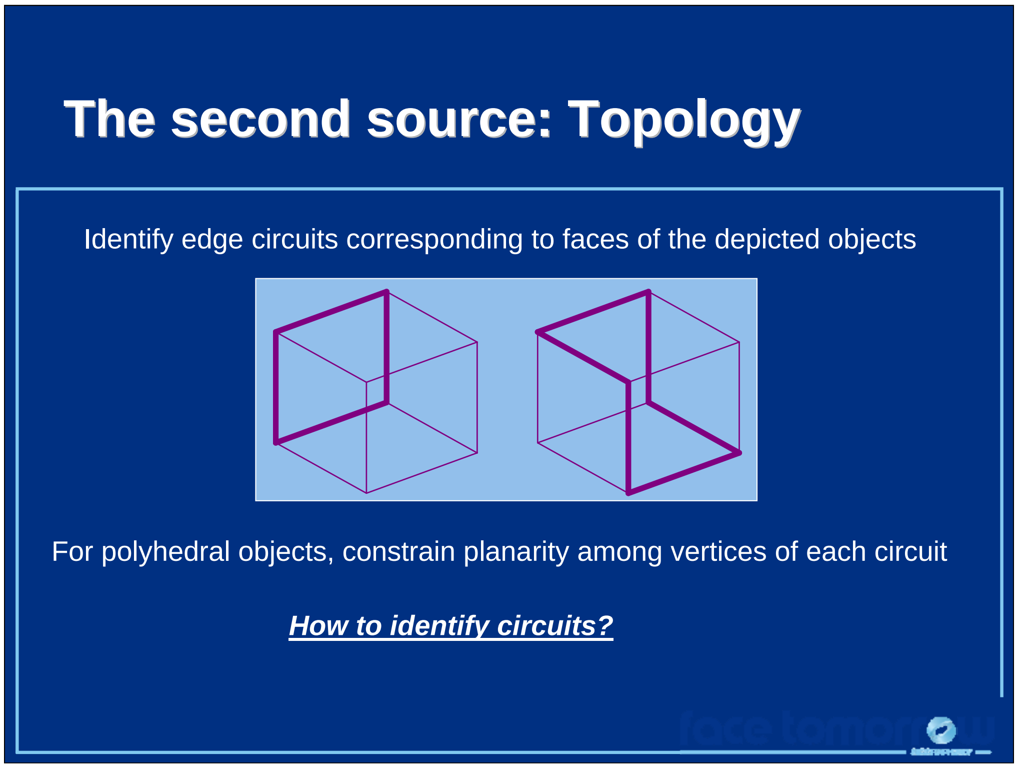
The first source: Geometric correlations

We interpret drawings by known correlations.

Images not conforming with the correlations confuse us.

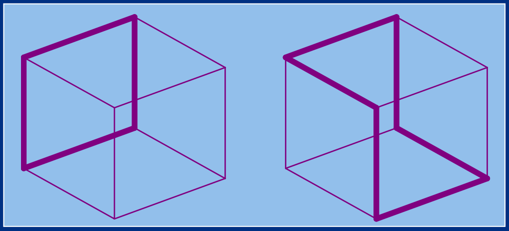


M. C. Escher



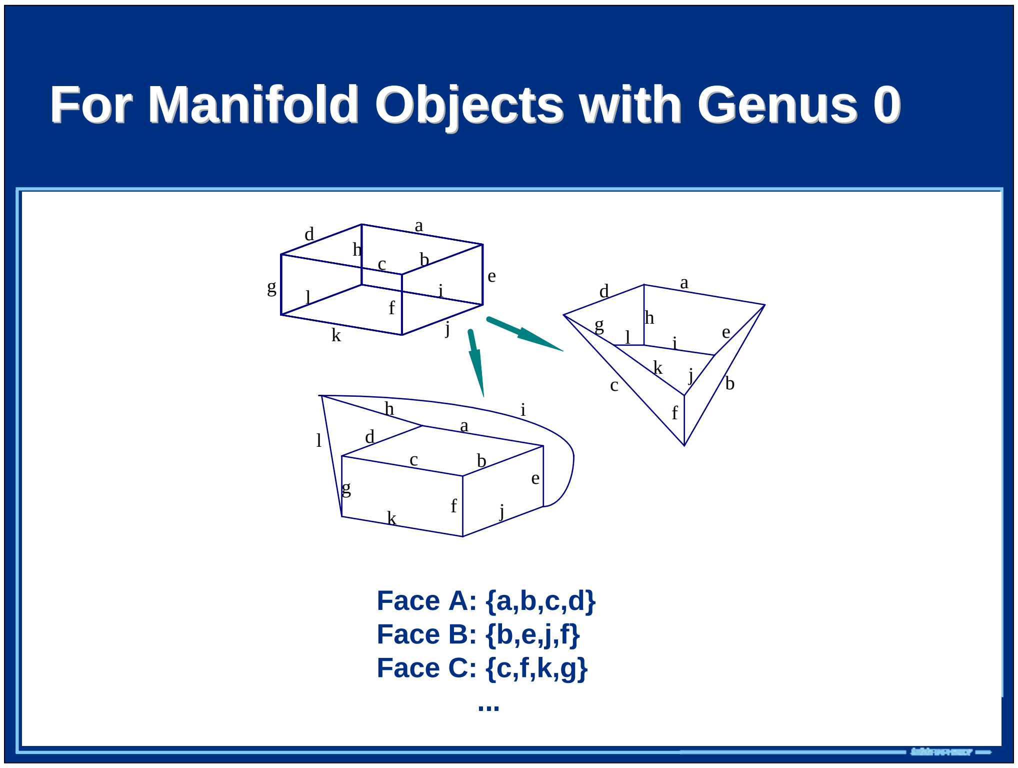
The second source: Topology

Identify edge circuits corresponding to faces of the depicted objects

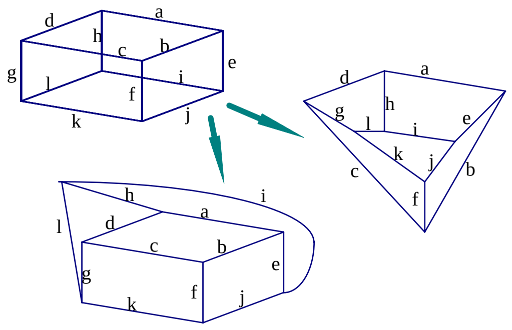


For polyhedral objects, constrain planarity among vertices of each circuit

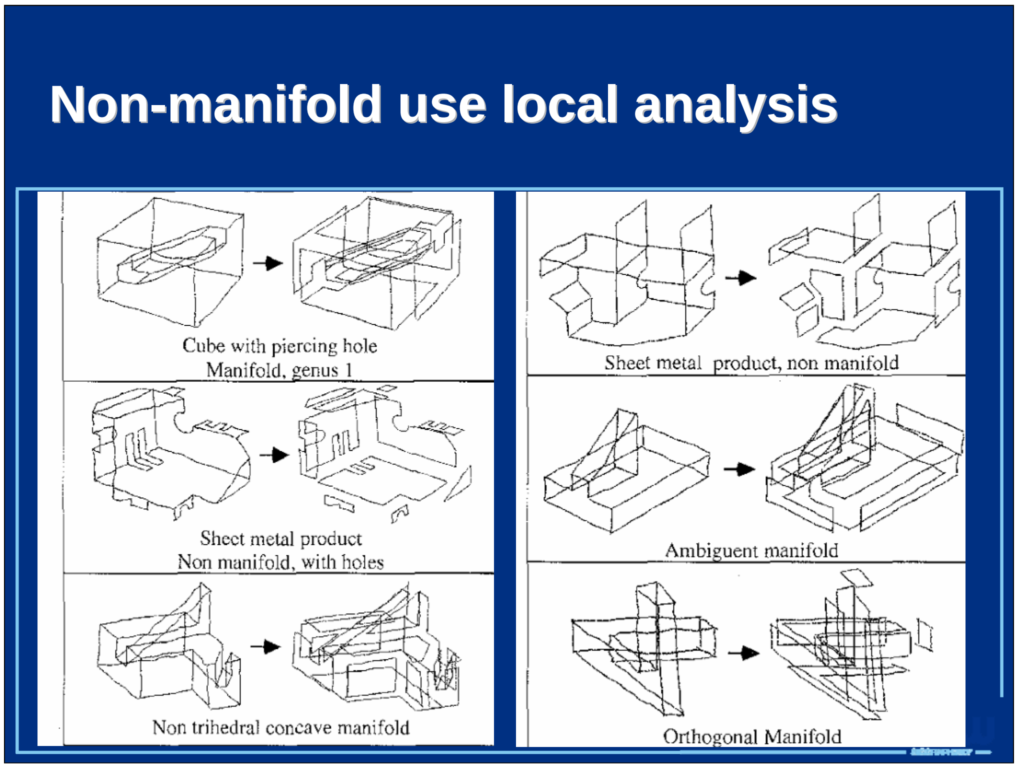
How to identify circuits?



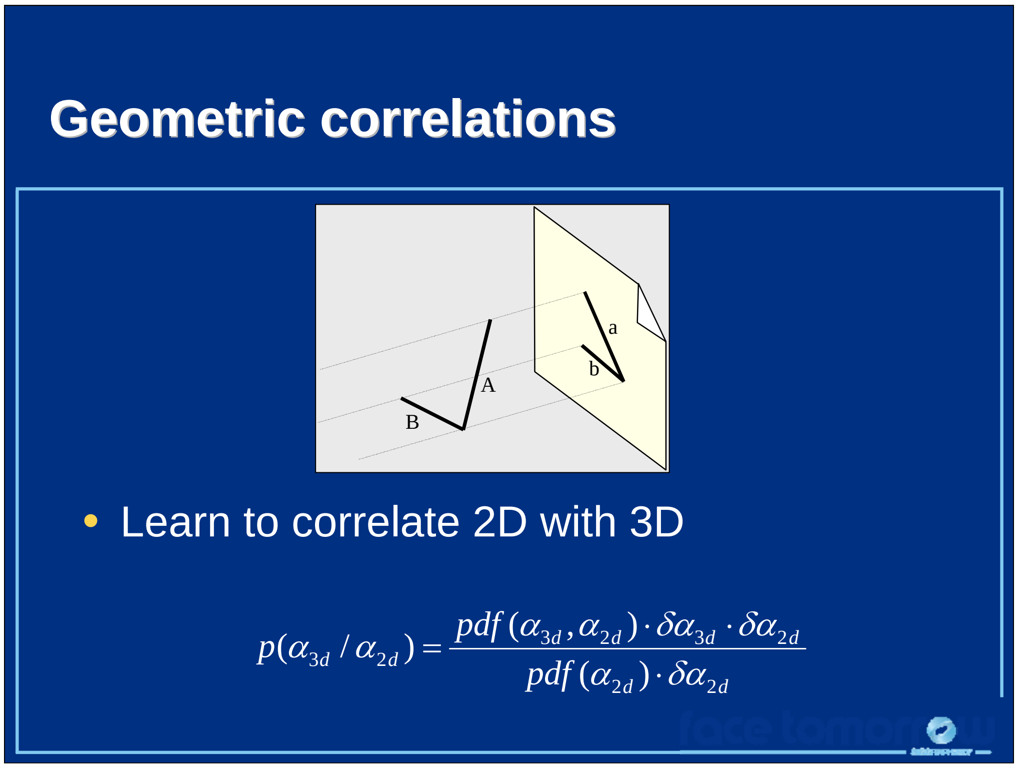
For Manifold Objects with Genus 0



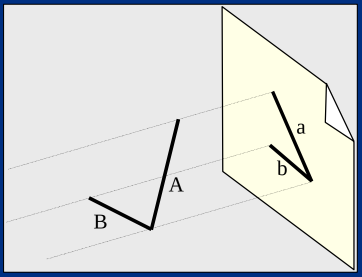
Face A: {a,b,c,d}
Face B: {b,e,j,f}
Face C: {c,f,k,g}
...



Non-manifold use local analysis

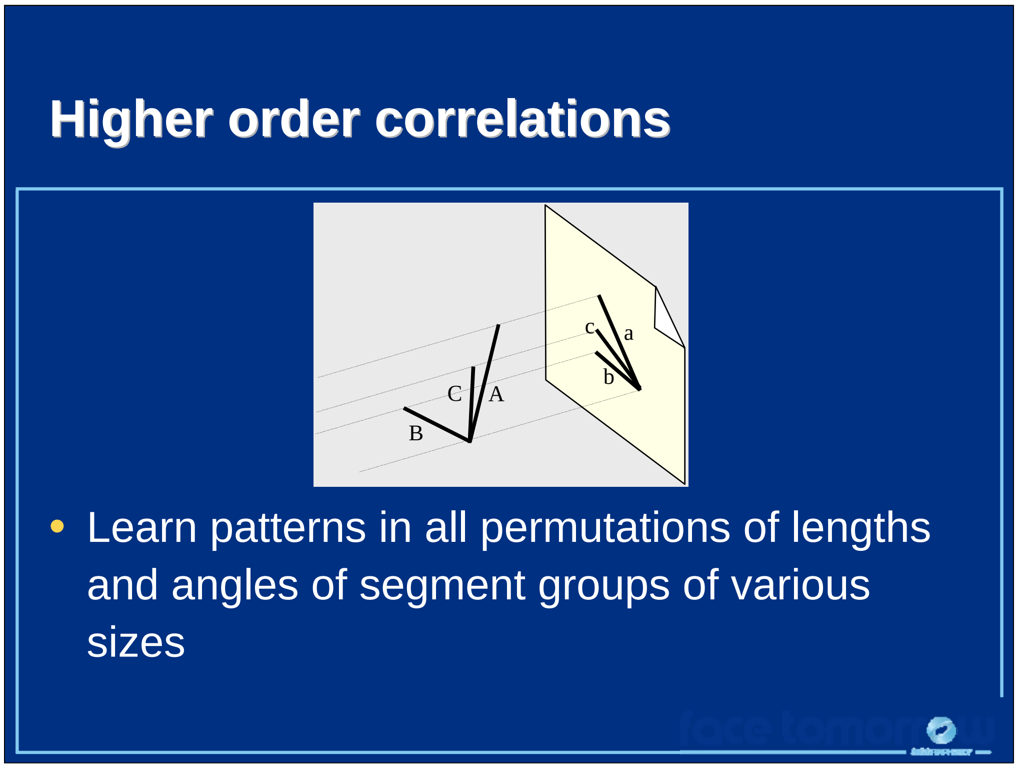


Geometric correlations

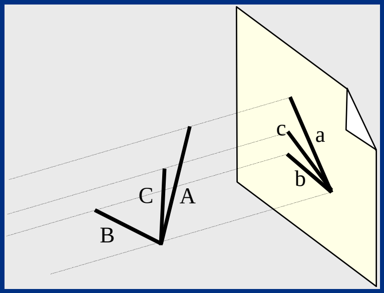


- Learn to correlate 2D with 3D

$$p(\alpha_{3d} / \alpha_{2d}) = \frac{pdf(\alpha_{3d}, \alpha_{2d}) \cdot \delta\alpha_{3d} \cdot \delta\alpha_{2d}}{pdf(\alpha_{2d}) \cdot \delta\alpha_{2d}}$$

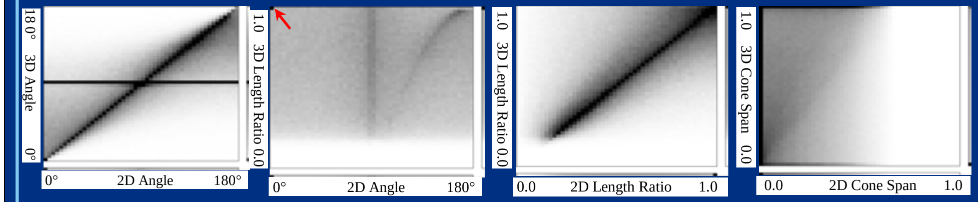


Higher order correlations

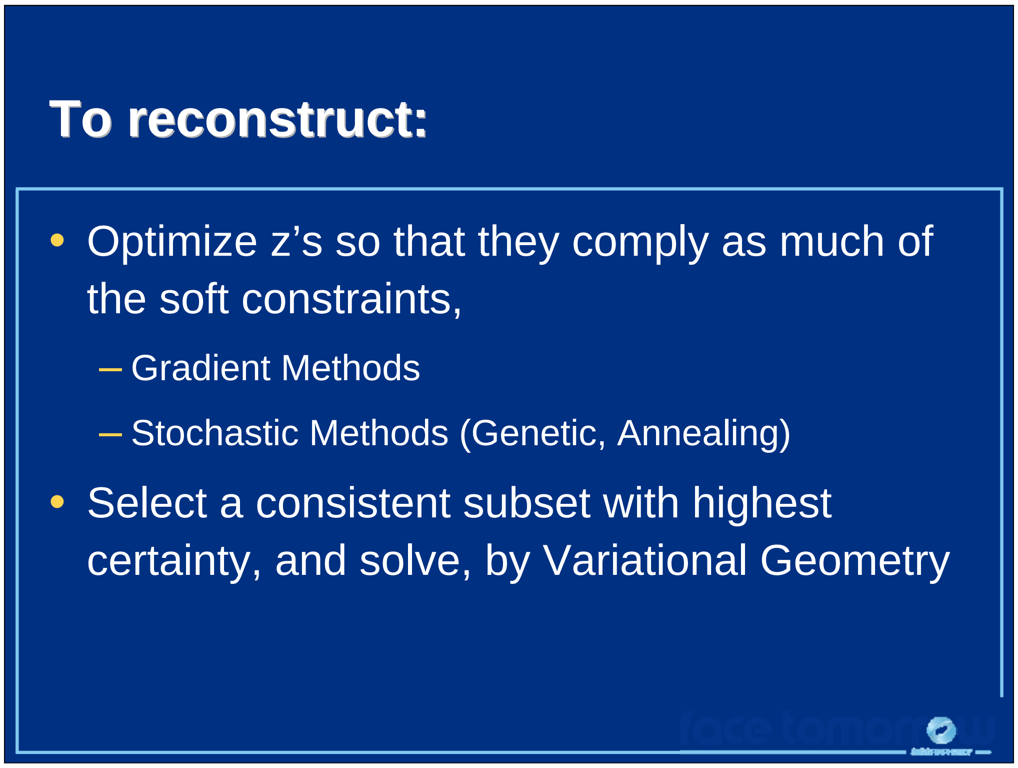


- Learn patterns in all permutations of lengths and angles of segment groups of various sizes

Measuring 2D-3D second order correlations

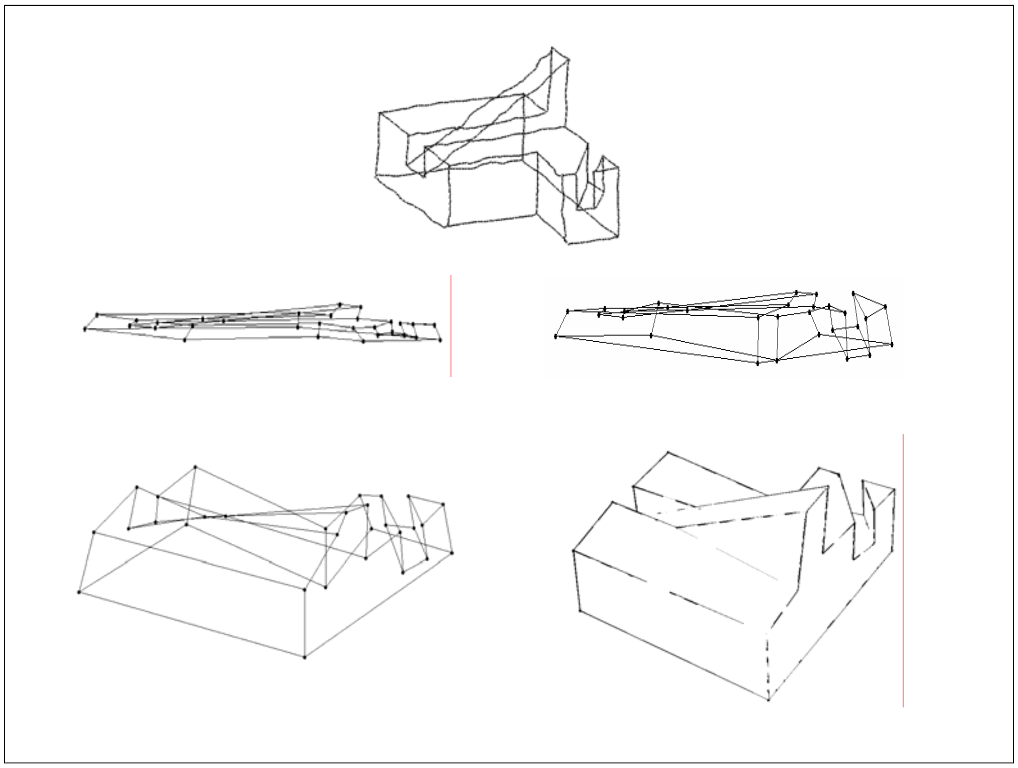


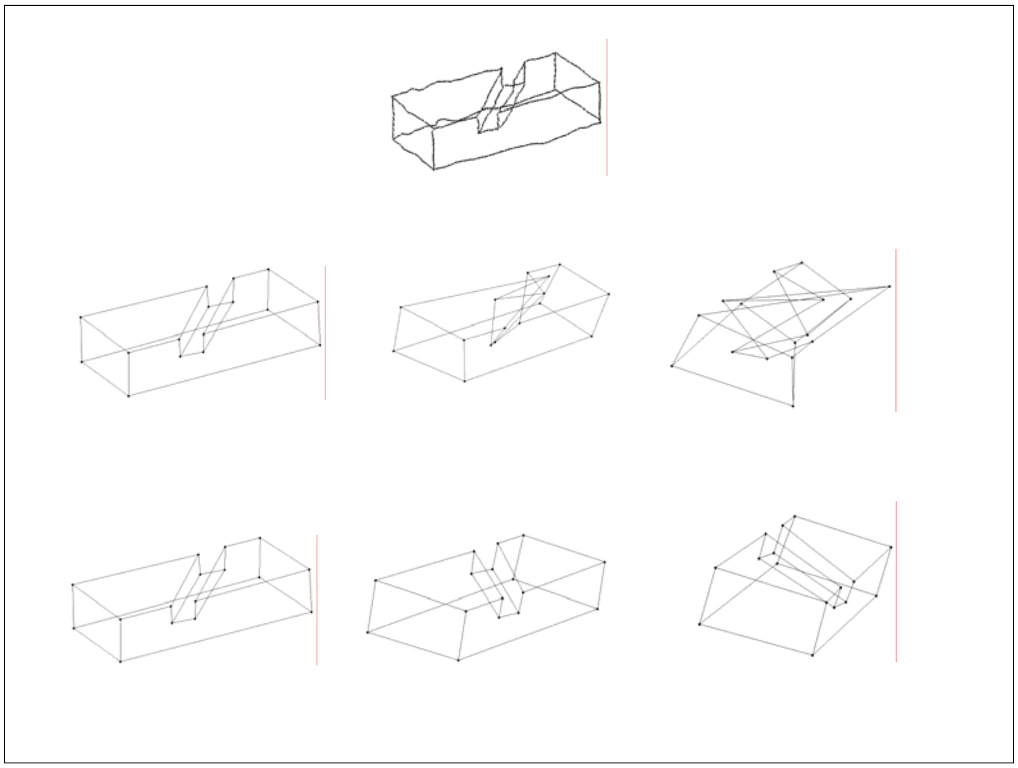
Dark areas show high correlation. Strips on right and bottom of each table show marginal probabilities. Note dark top-left corner of bottom-left plot.

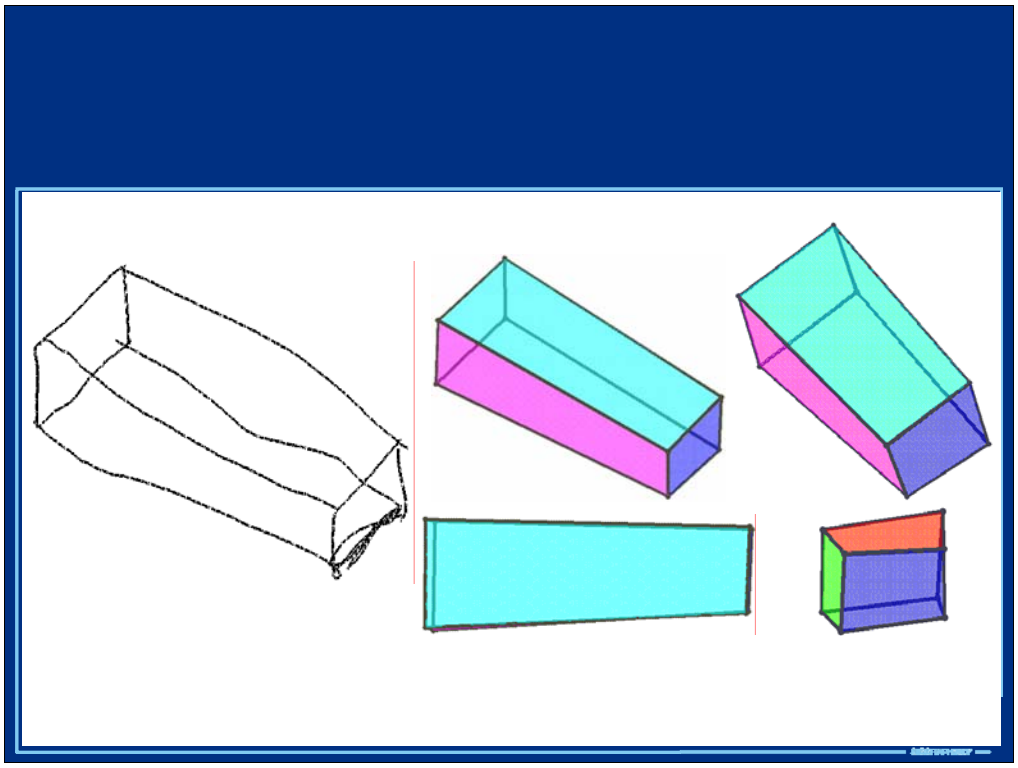


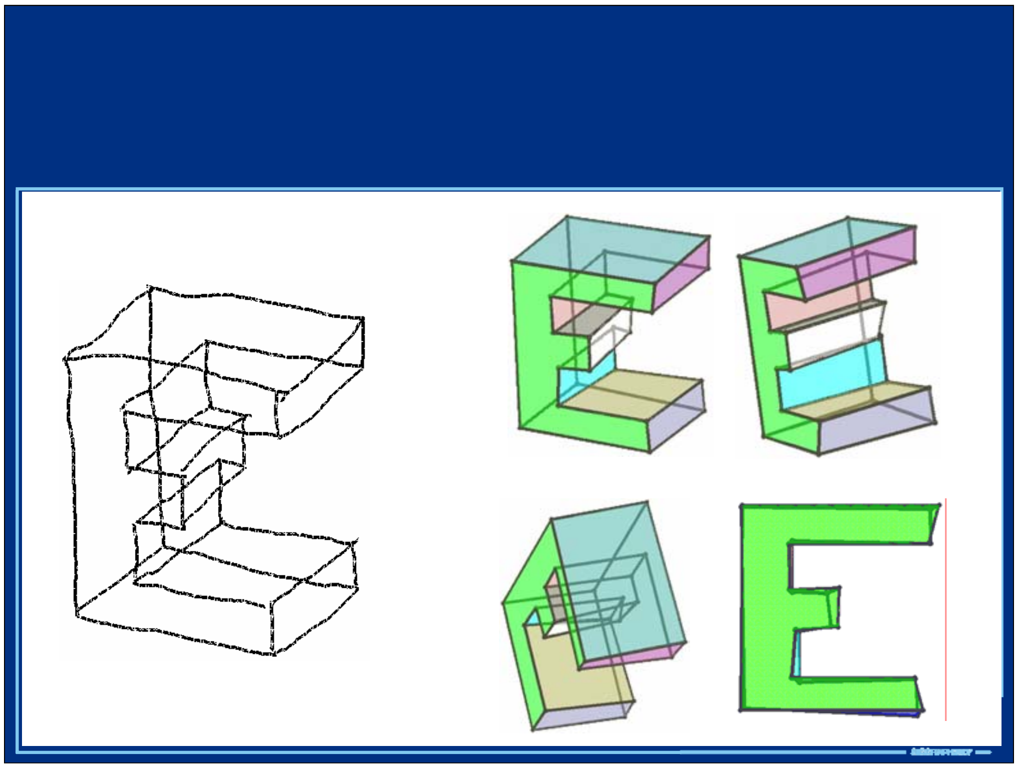
To reconstruct:

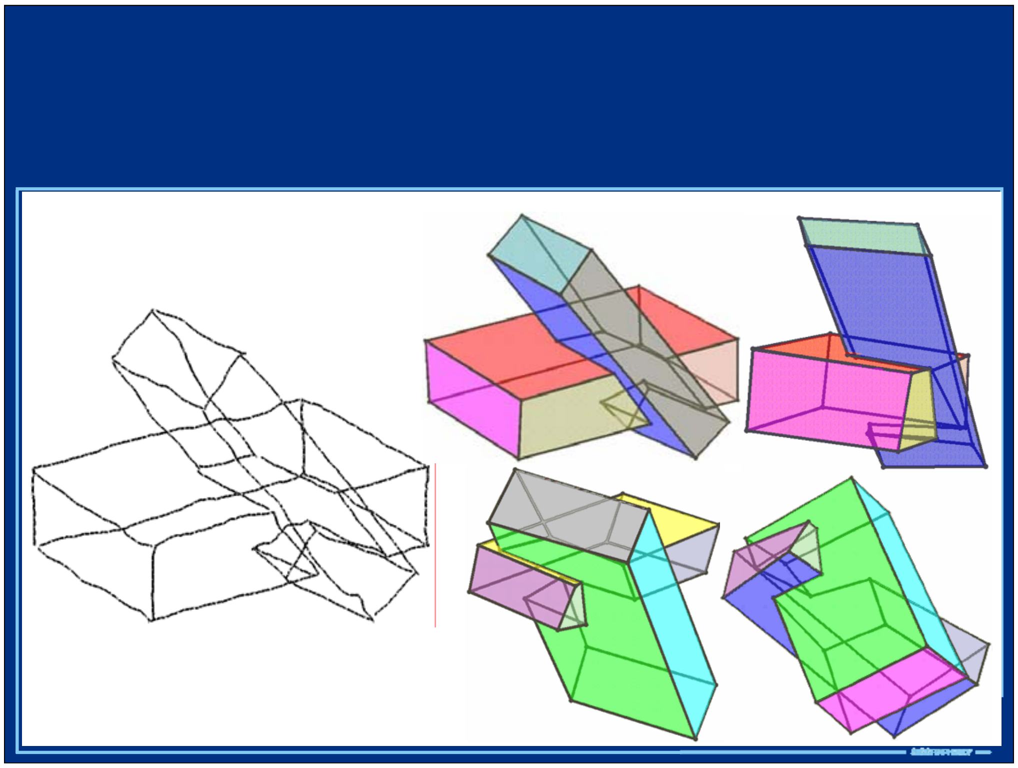
- Optimize z’s so that they comply as much of the soft constraints,
  - Gradient Methods
  - Stochastic Methods (Genetic, Annealing)
- Select a consistent subset with highest certainty, and solve, by Variational Geometry

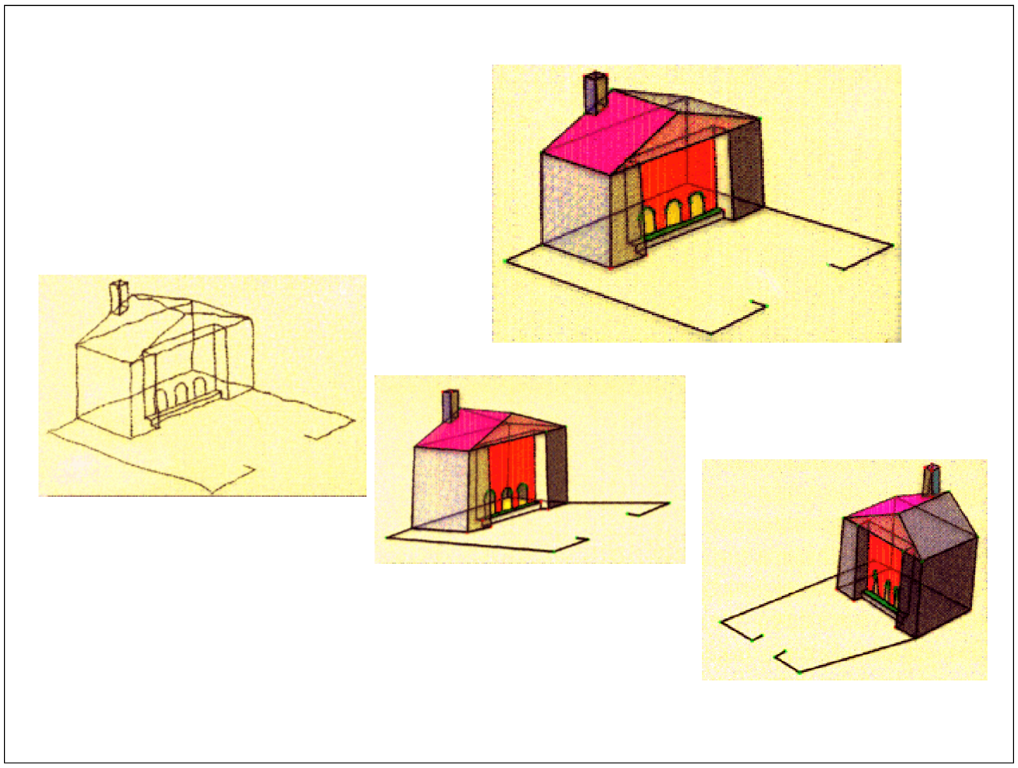


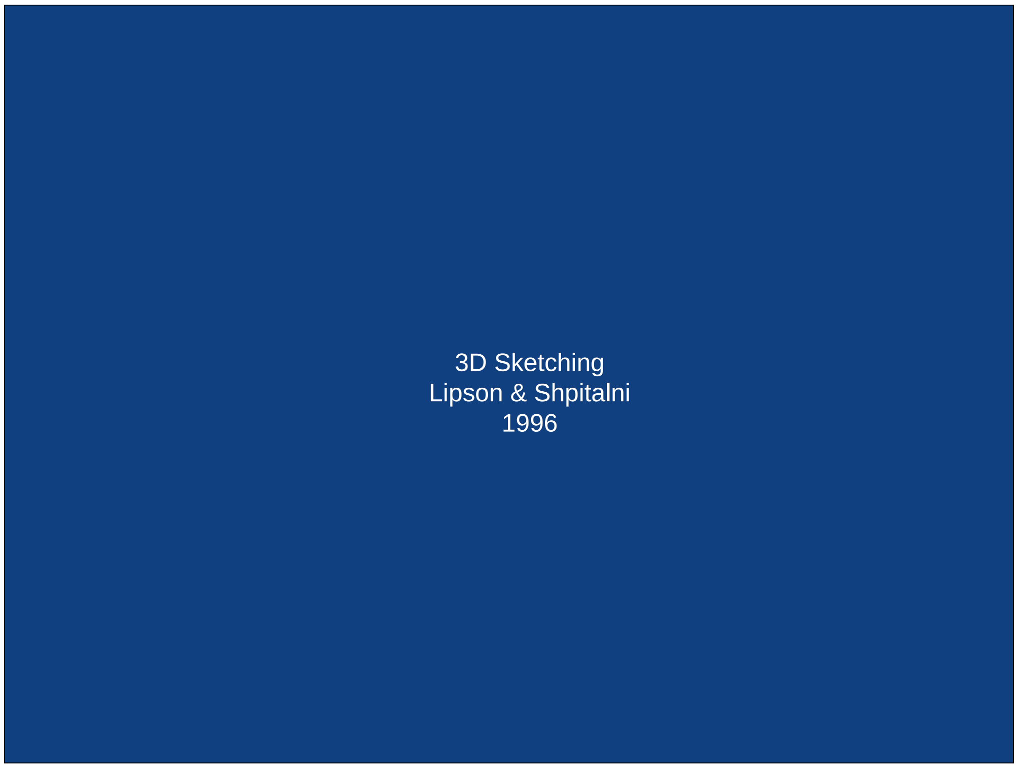




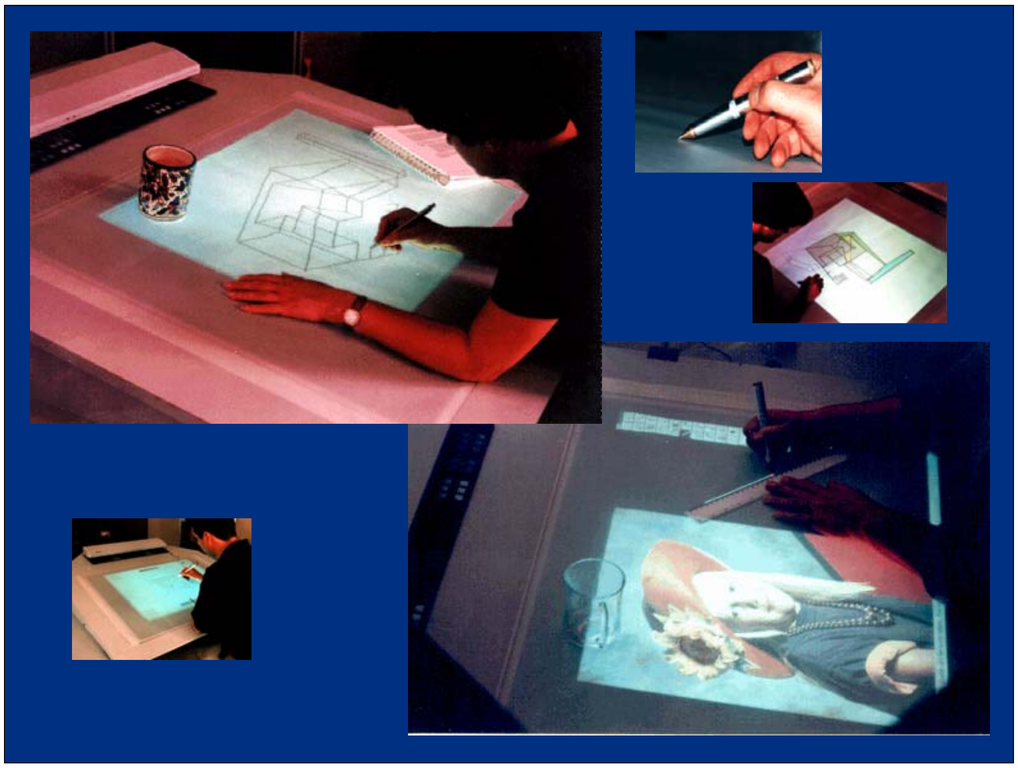


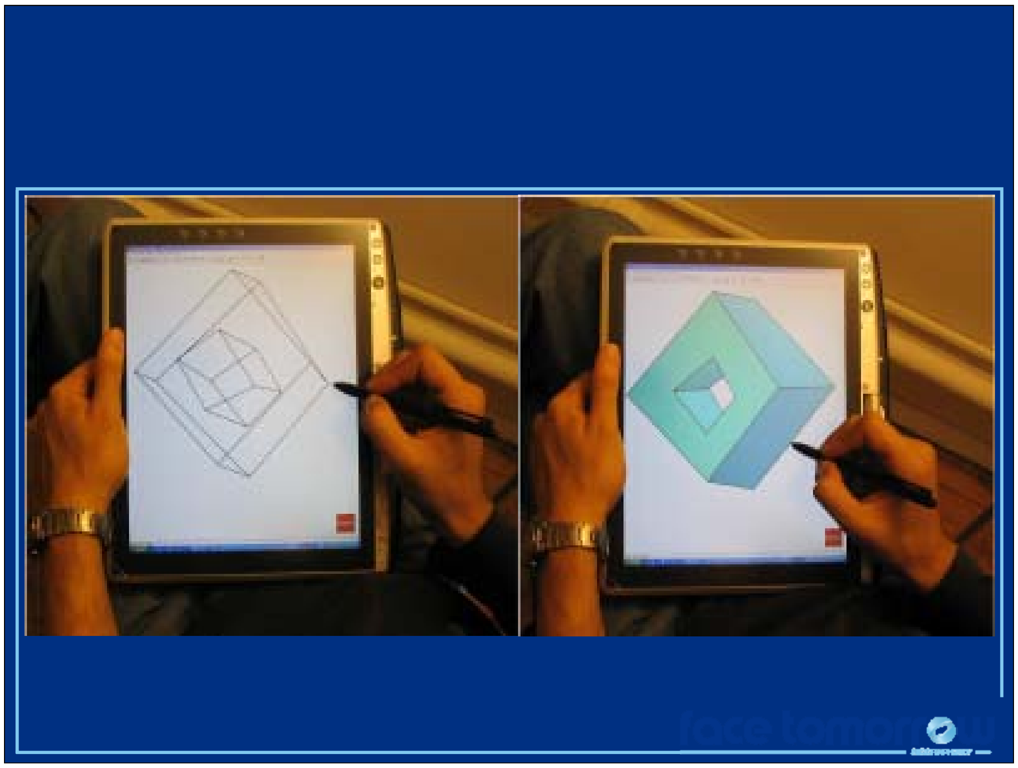


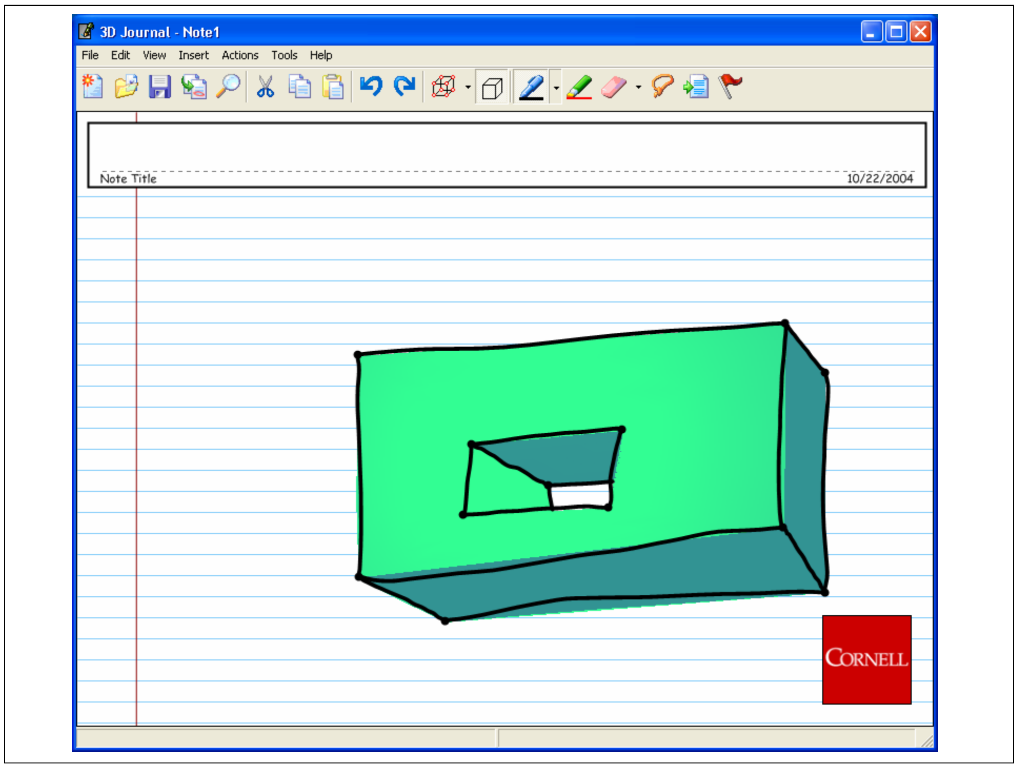


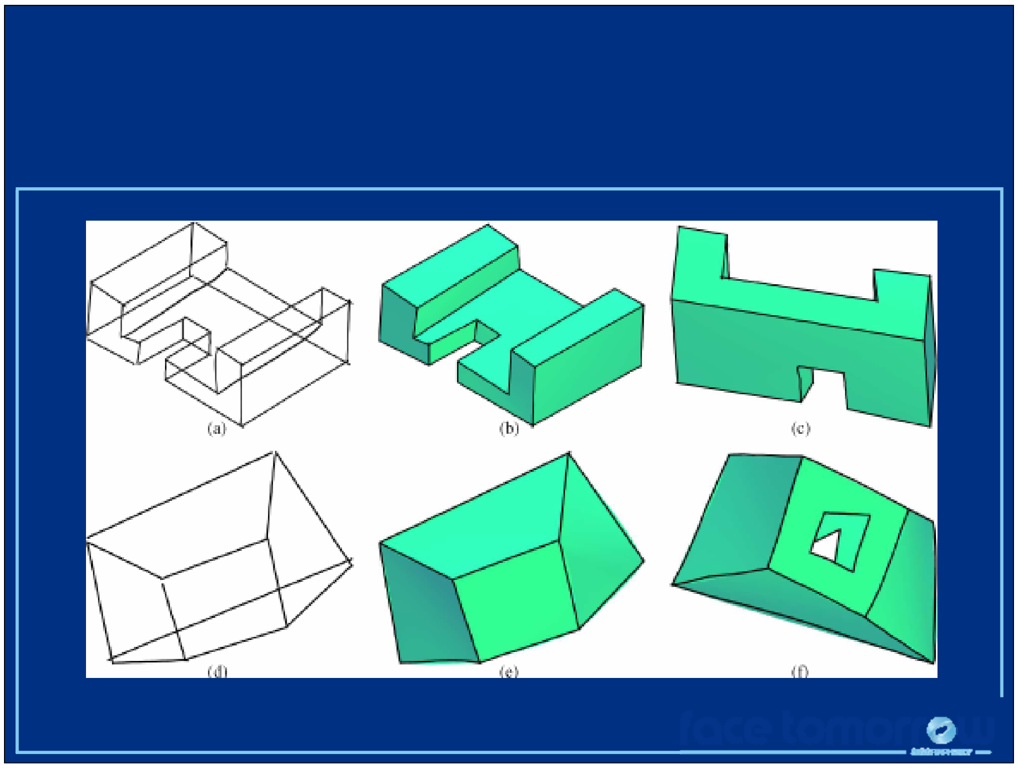


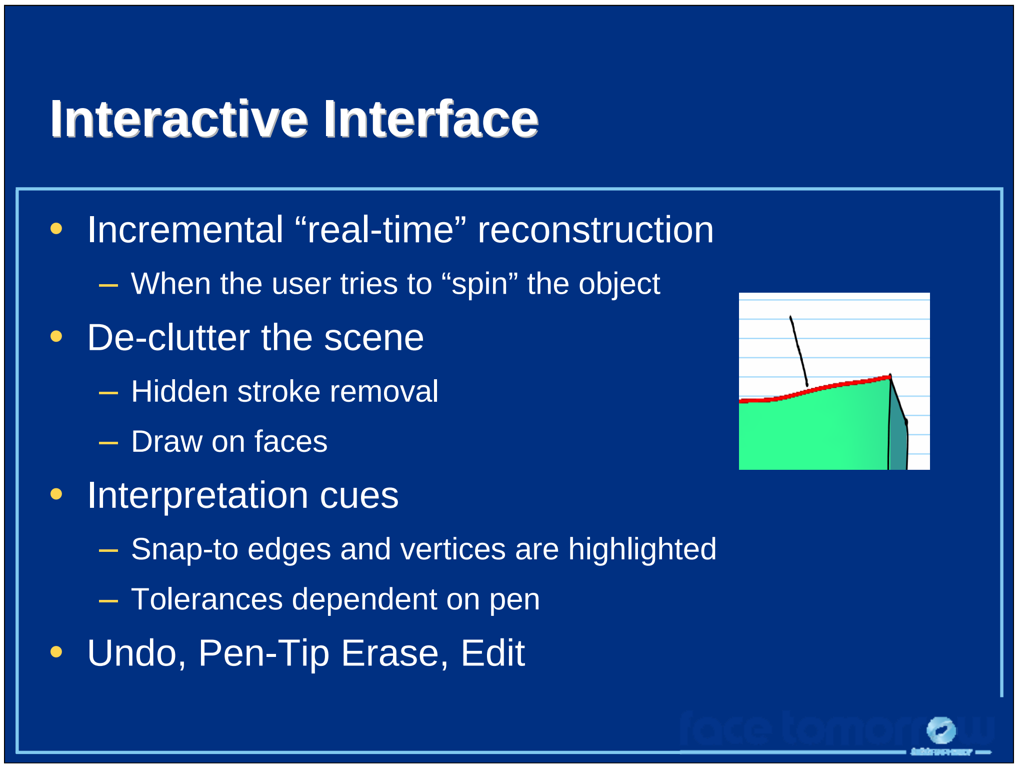
3D Sketching
Lipson & Shpitalni
1996





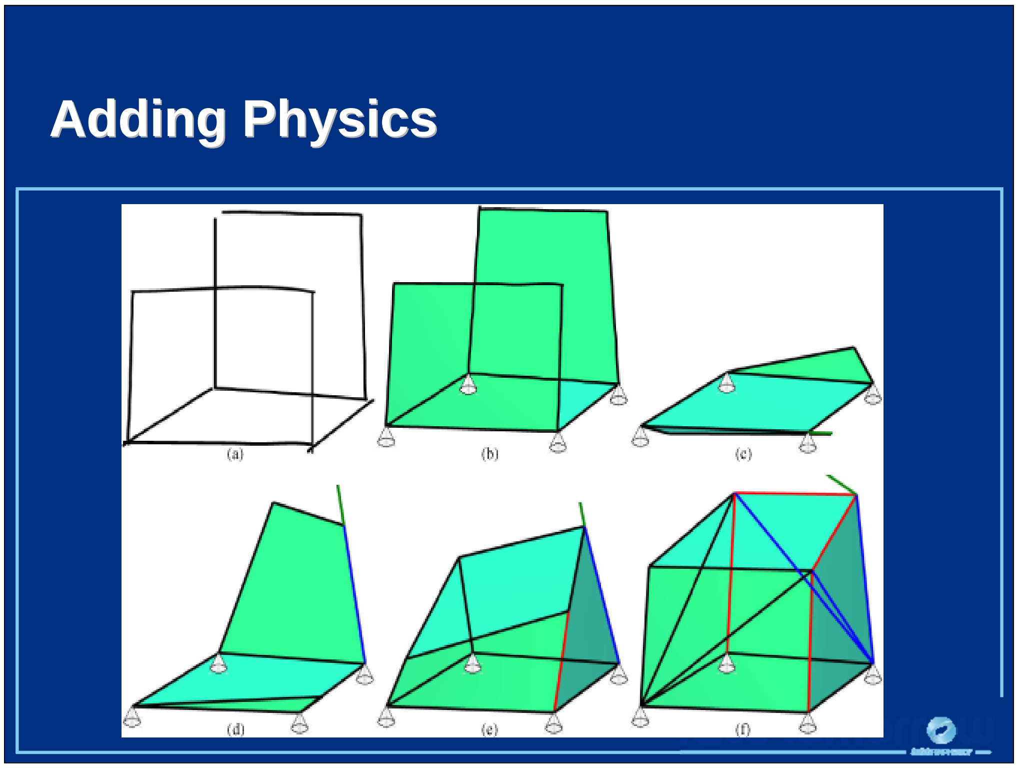

3D Journal - Note1
File Edit View Insert Actions Tools Help
Note Title
10/22/2004
CORNELL


(a)
(b)
(c)
(d)
(e)
(f)

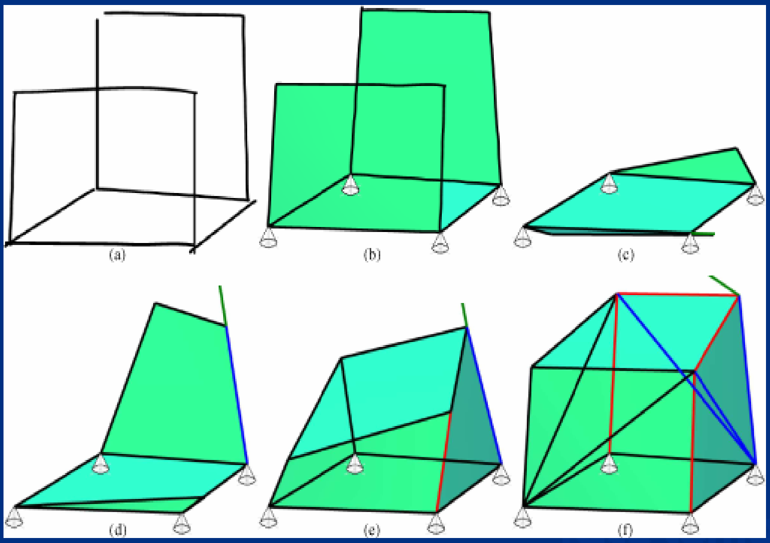


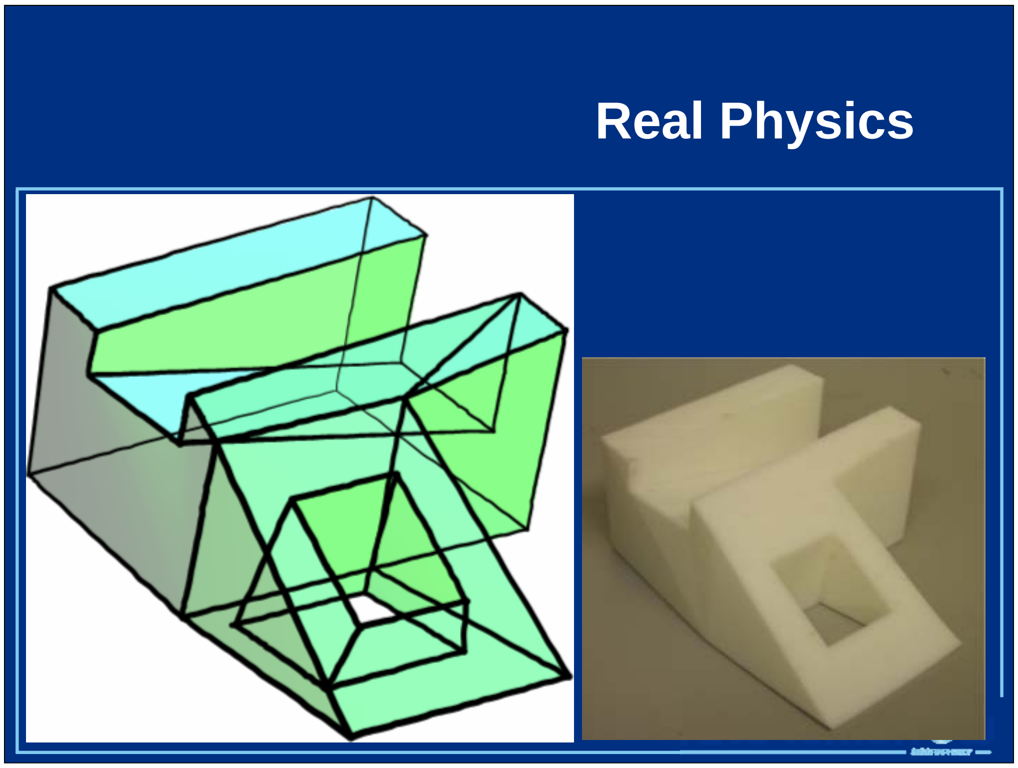
Interactive Interface

- Incremental “real-time” reconstruction
  - When the user tries to “spin” the object
- De-clutter the scene
  - Hidden stroke removal
  - Draw on faces
- Interpretation cues
  - Snap-to edges and vertices are highlighted
  - Tolerances dependent on pen
- Undo, Pen-Tip Erase, Edit

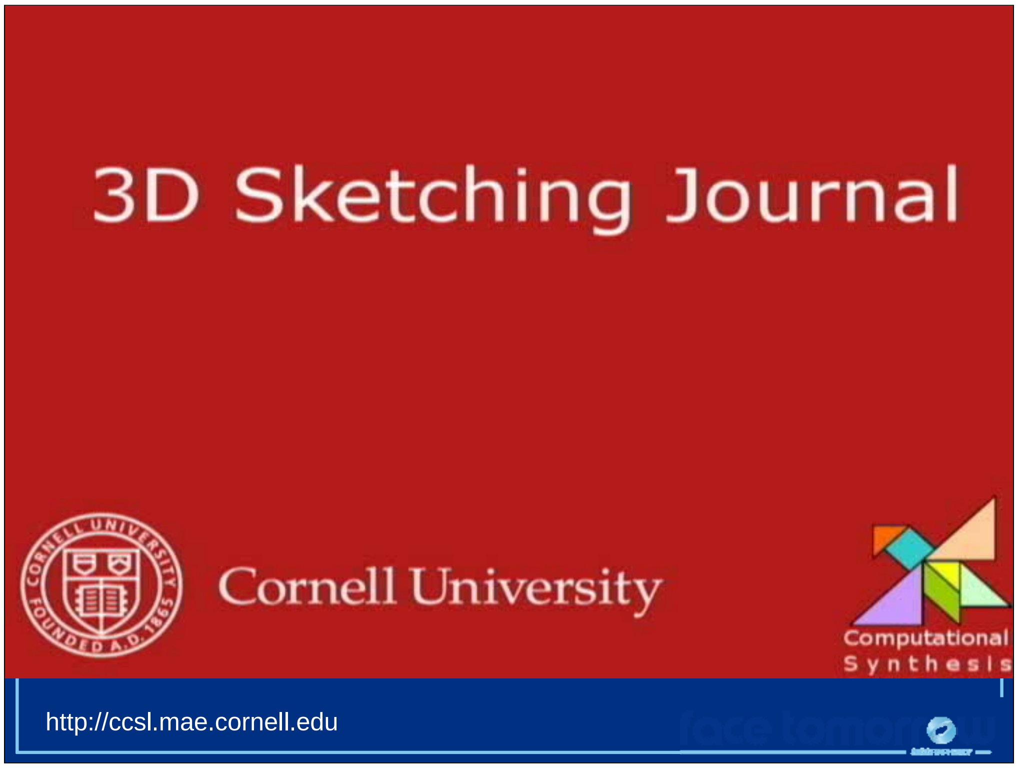


Adding Physics

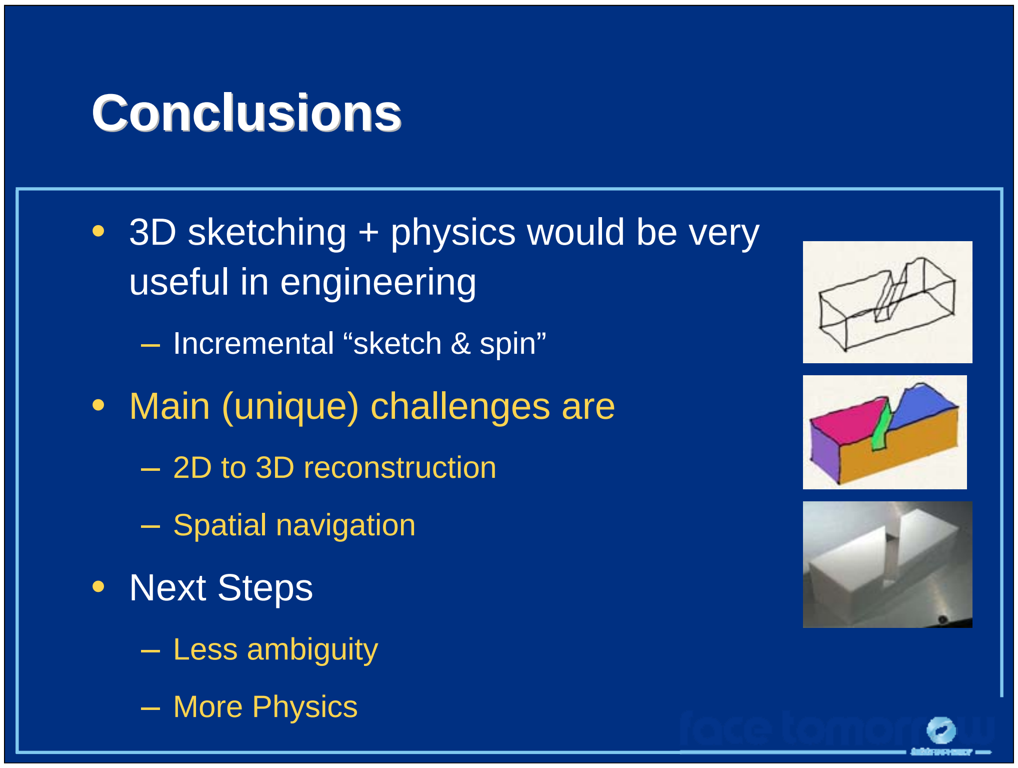




Real Physics

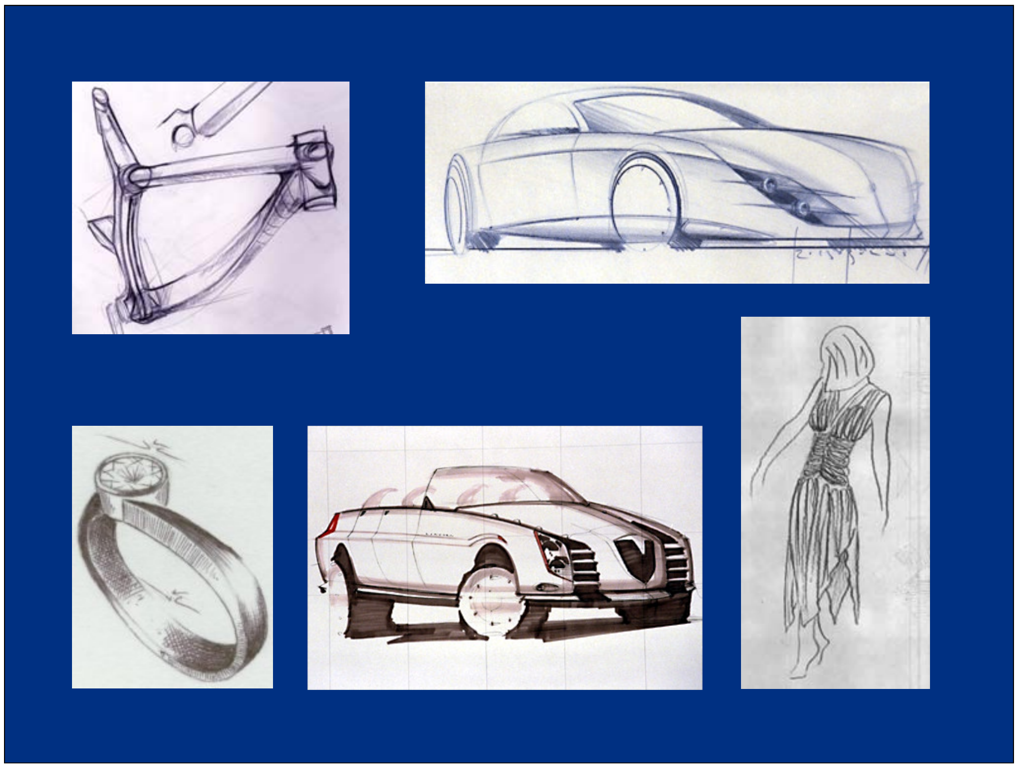


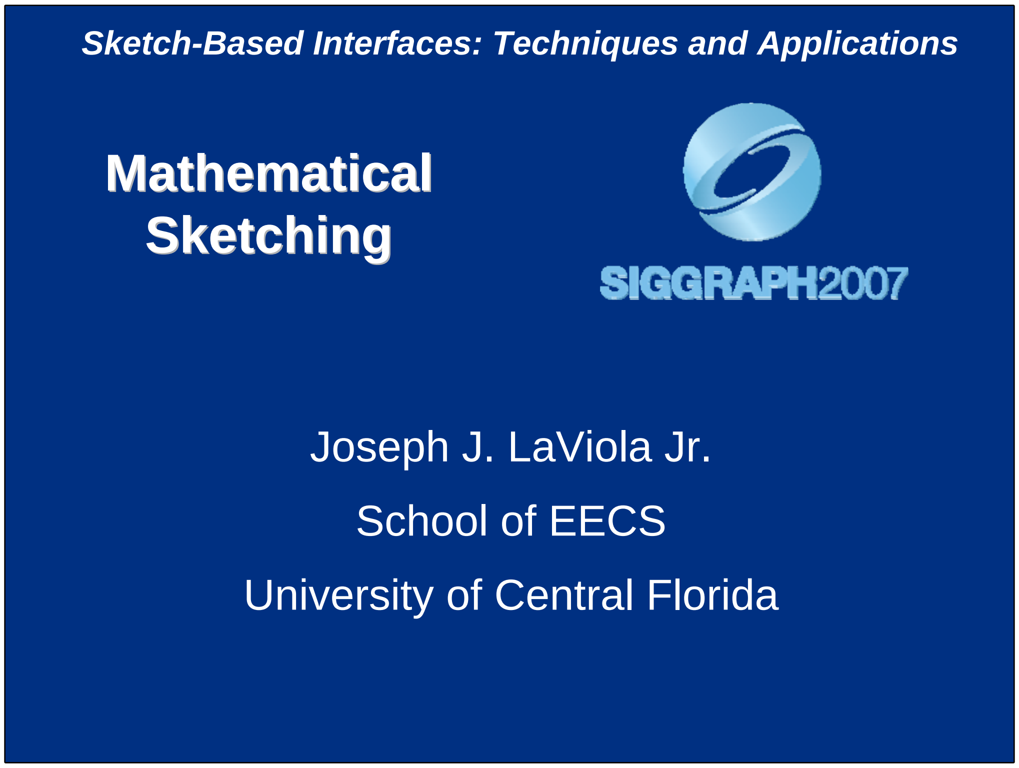
http://ccsl.mae.cornell.edu



Conclusions

- 3D sketching + physics would be very useful in engineering
  - Incremental “sketch & spin”
- Main (unique) challenges are
  - 2D to 3D reconstruction
  - Spatial navigation
- Next Steps
  - Less ambiguity
  - More Physics





Sketch-Based Interfaces: Techniques and Applications

Mathematical Sketching

SIGGRAPH2007

Joseph J. LaViola Jr.

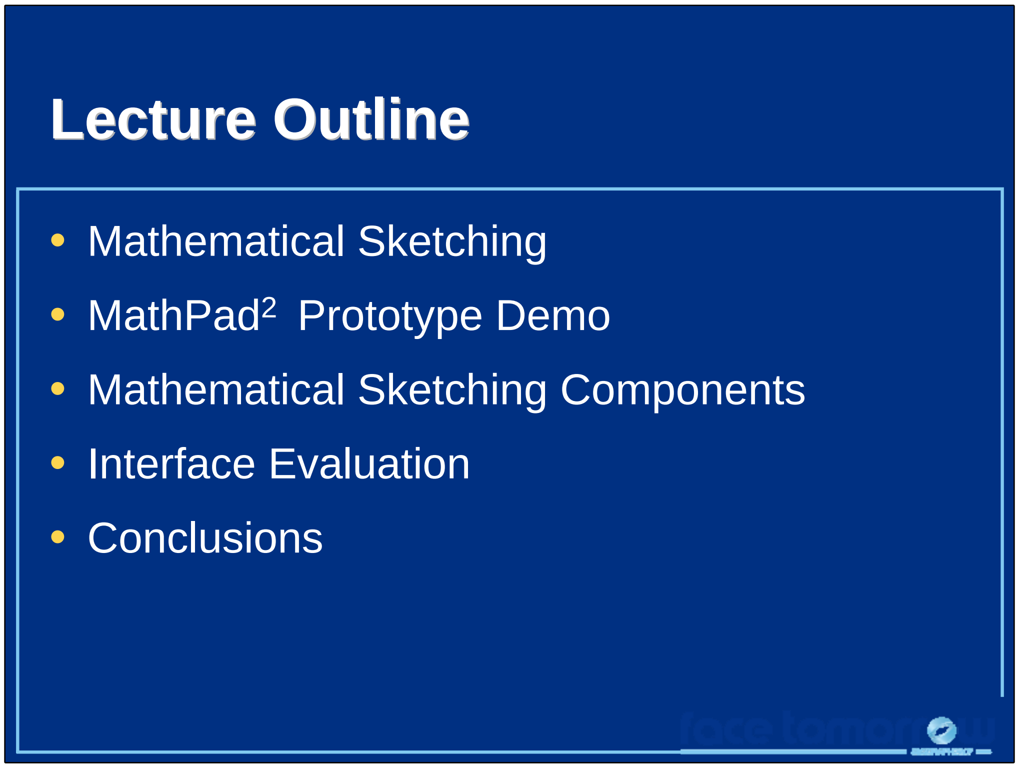
School of EECS

University of Central Florida

Mathematical Sketching

Joseph J. LaViola Jr.
Assistant Professor
University of Central Florida
School of EECS
4000 Central Florida Blvd.
Orlando, FL 32816

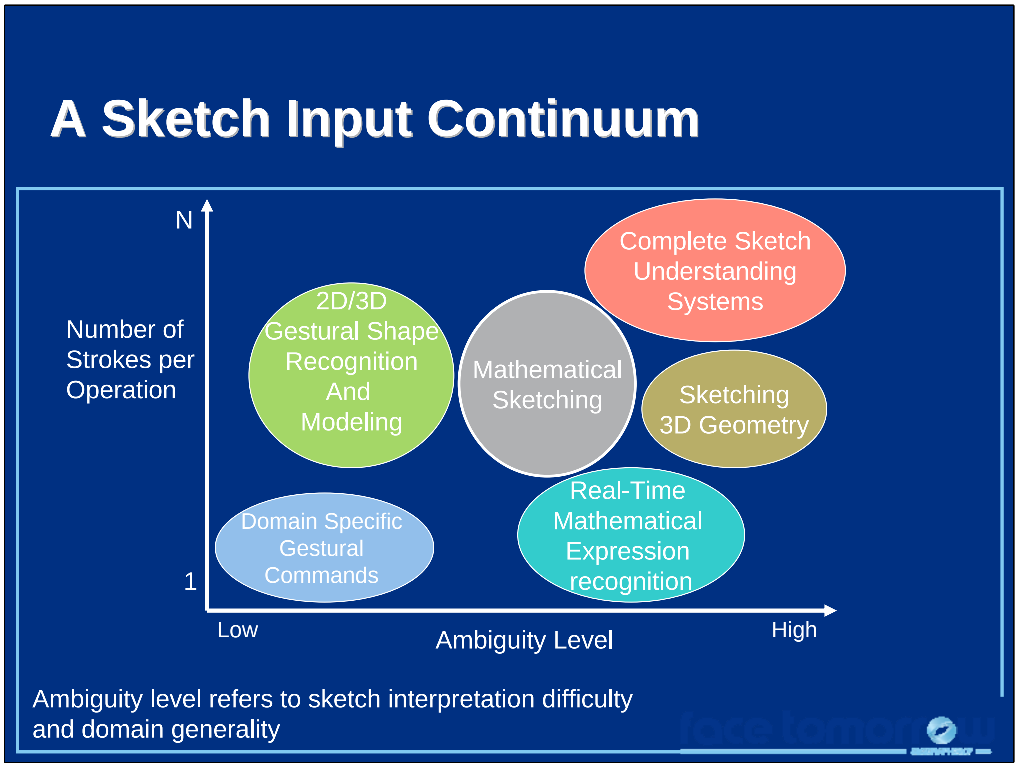
Email: jjl@cs.ucf.edu
http://www.cs.ucf.edu/~jjl/



Lecture Outline

- Mathematical Sketching
- MathPad^2 Prototype Demo
- Mathematical Sketching Components
- Interface Evaluation
- Conclusions

In this lecture, we will discuss mathematical sketching and its components. First, we will provide some motivation for why mathematical sketching is important. Second, we will explore the definition of mathematical sketching. Third, we will give a demo of MathPad^2, an application prototype that implements mathematical sketching. Fourth, we will look at the various components that make up mathematical sketching. Finally, we will present some conclusions.

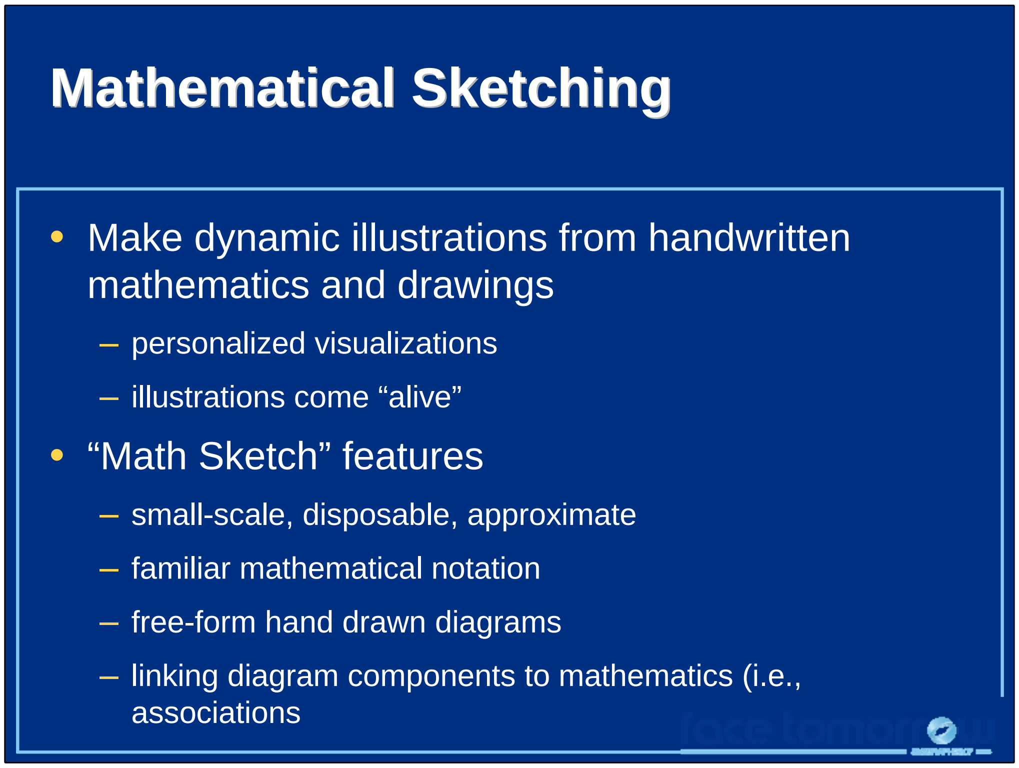


A Sketch Input Continuum

N
Number of Strokes per Operation
1
Low
Ambiguity Level
High
Domain Specific Gestural Commands
2D/3D Gestural Shape Recognition And Modeling
Mathematical Sketching
Real-Time Mathematical Expression recognition
Complete Sketch Understanding Systems
Sketching 3D Geometry

Ambiguity level refers to sketch interpretation difficulty and domain generality

Mathematical sketching as an interaction paradigm has many different parts. As such, it falls somewhere in the middle of the sketch-input continuum.



Mathematical Sketching

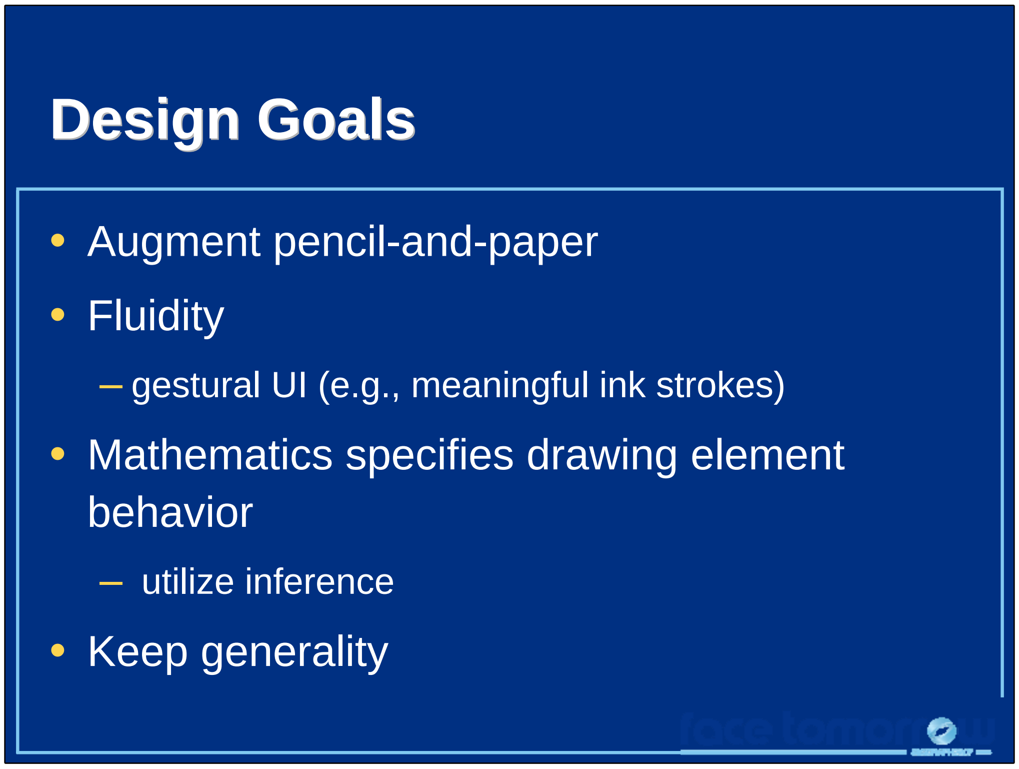
- Make dynamic illustrations from handwritten mathematics and drawings
  - personalized visualizations
  - illustrations come “alive”
- “Math Sketch” features
  - small-scale, disposable, approximate
  - familiar mathematical notation
  - free-form hand drawn diagrams
  - linking diagram components to mathematics (i.e., associations

We want to be able to create these illustrations and make them dynamic with users constructing them as if they were using pencil and paper. Mathematical sketching is an approach to solving this problem. It is a sketch-based interaction paradigm that lets users make dynamic illustrations by associating handwritten mathematics and drawings together. These dynamic illustrations become personalized visualizations of the concept in question since the user creates them with a sketch-based interface.

What makes a mathematical sketch? Mathematical sketches are small-scale, disposable, and approximate. They are not meant to solve complex problems but rather to help illustrate fundamental concepts in physics and mathematics. They utilize familiar mathematical notation and free-form hand drawn diagrams where diagram components are linked to the mathematics with associations.

References:

LaViola, J. Advances in Mathematical Sketching: Moving Toward the Paradigm's Full Potential, *IEEE Computer Graphics and Applications*, 27(1):38-48, January/February 2007.

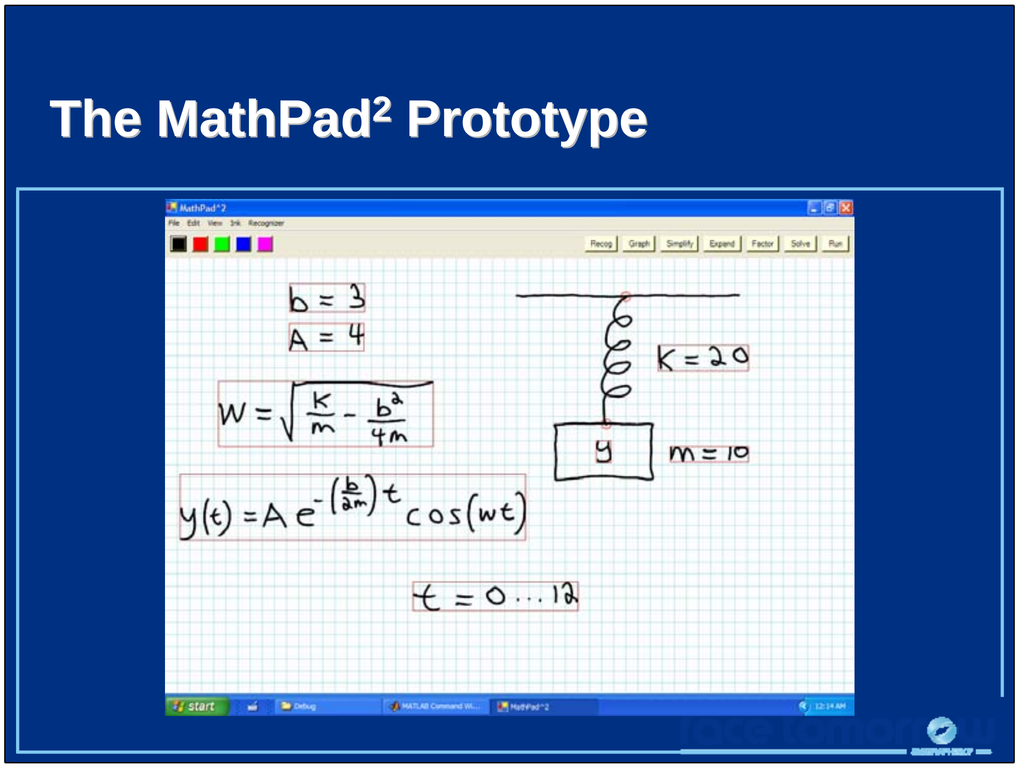


Design Goals

- Augment pencil-and-paper
- Fluidity
  - gestural UI (e.g., meaningful ink strokes)
- Mathematics specifies drawing element behavior
  - utilize inference
- Keep generality

One of the major principles behind mathematical sketching is that it should augment pencil and paper. Thus, users should be able to create mathematical sketches using a pencil-and-paper style interface and leverage the power of a computer to take their input and produce visualizations and dynamic illustrations. As part of this design approach, the interface must be fluid and a gestural user interface that combine handwriting, drawing, and issuing commands provides mathematical sketching with a natural interaction mechanism.

The other major principle behind mathematical sketching is that the recognized handwritten mathematics be the driving force behind any drawing element behavior. This approach exposes the user to the mathematics behind an illustration so connections between the two can be observed and understood. Making the recognized handwritten mathematics the driving force behind mathematical sketching also provides a level of generality since no specific domain knowledge is required.



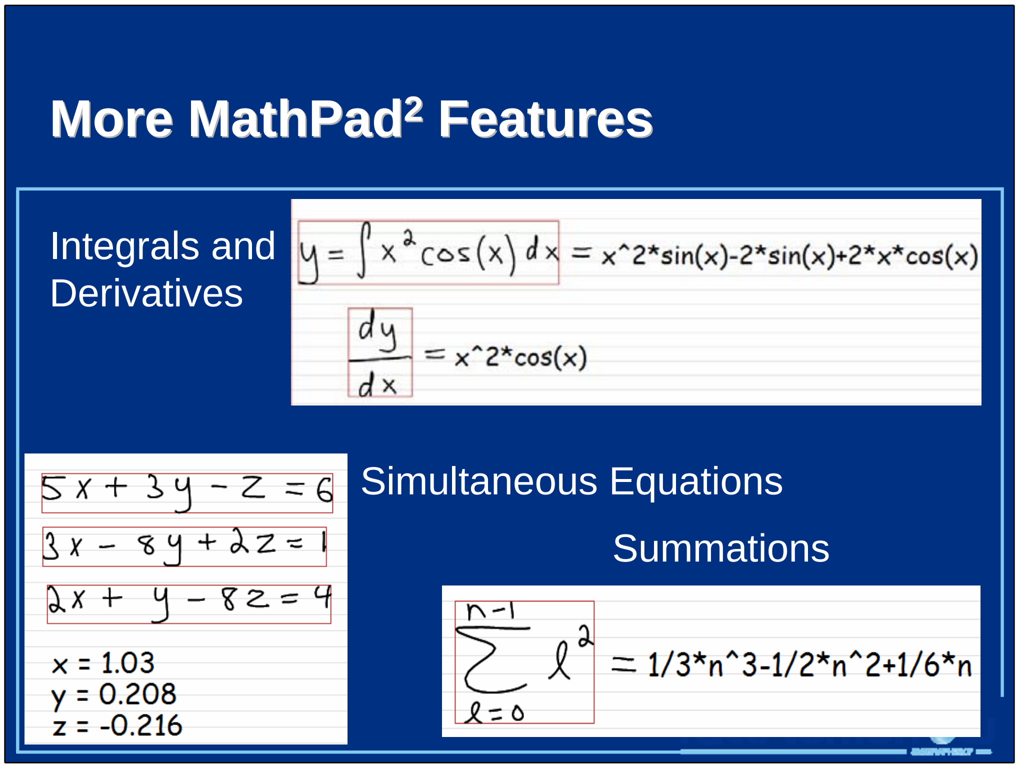
The MathPad^2 Prototype

MathPad^2
File Edit View Ink Recognizer
Recog Graph Simplify Expand Factor Solve Run
$b = 3$
$A = 4$
$W = \sqrt{\frac{k}{m} - \frac{b^2}{4m}}$
$y(t) = A e^{-\left(\frac{b}{2m}\right)t} \cos(wt)$
$t = 0 \ldots 12$
$k = 20$
y
$m = 10$
start
Debug
MATLAB Command Wi...
MathPad^2

The idea of mathematical sketching is very broad and should support many different dynamic illustrations. To evaluate some of these illustration possibilities, MathPad^2, a Tablet PC application prototype, was developed. A video of the MathPad^2 prototype is included in the videos directory.

References:

LaViola, J. and Zeleznik, R. MathPad^2: A System for the Creation and Exploration of Mathematical Sketches, *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, 23(3):432-440, August 2004.



More MathPad² Features

Integrals and Derivatives

$y = \int x^2 \cos(x)\,dx$ = x^2*sin(x)-2*sin(x)+2*x*cos(x)

$\frac{dy}{dx}$ = x^2*cos(x)

Simultaneous Equations

$5x + 3y - z = 6$

$3x - 8y + 2z = 1$

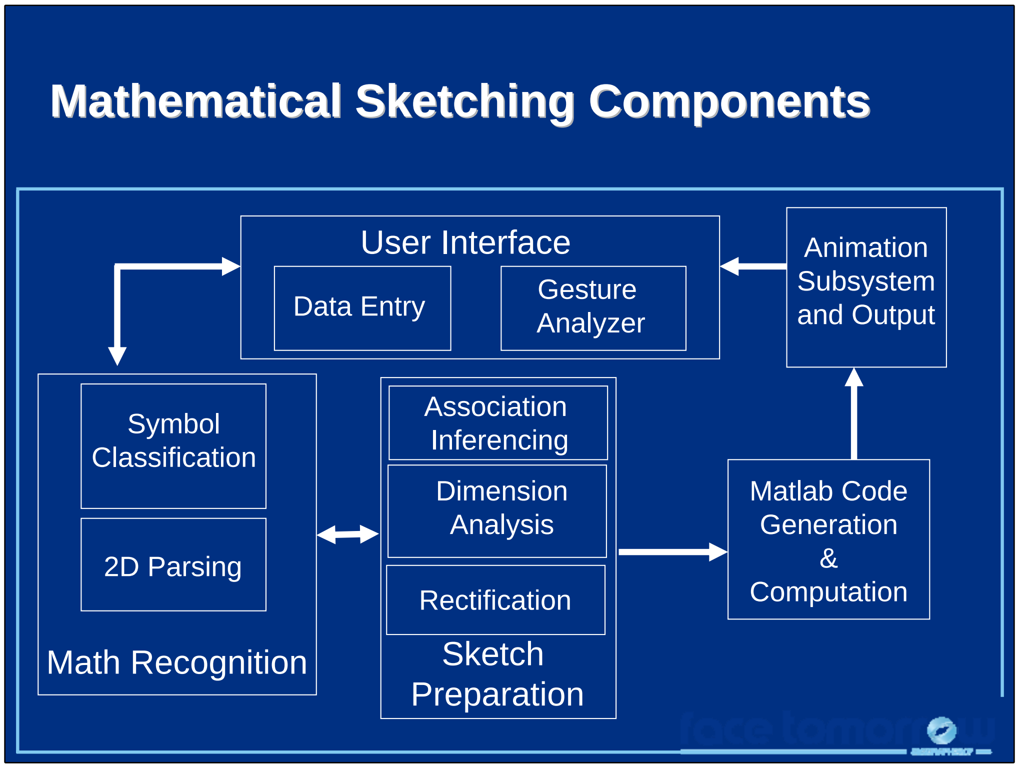
$2x + y - 8z = 4$

x = 1.03
y = 0.208
z = -0.216

Summations

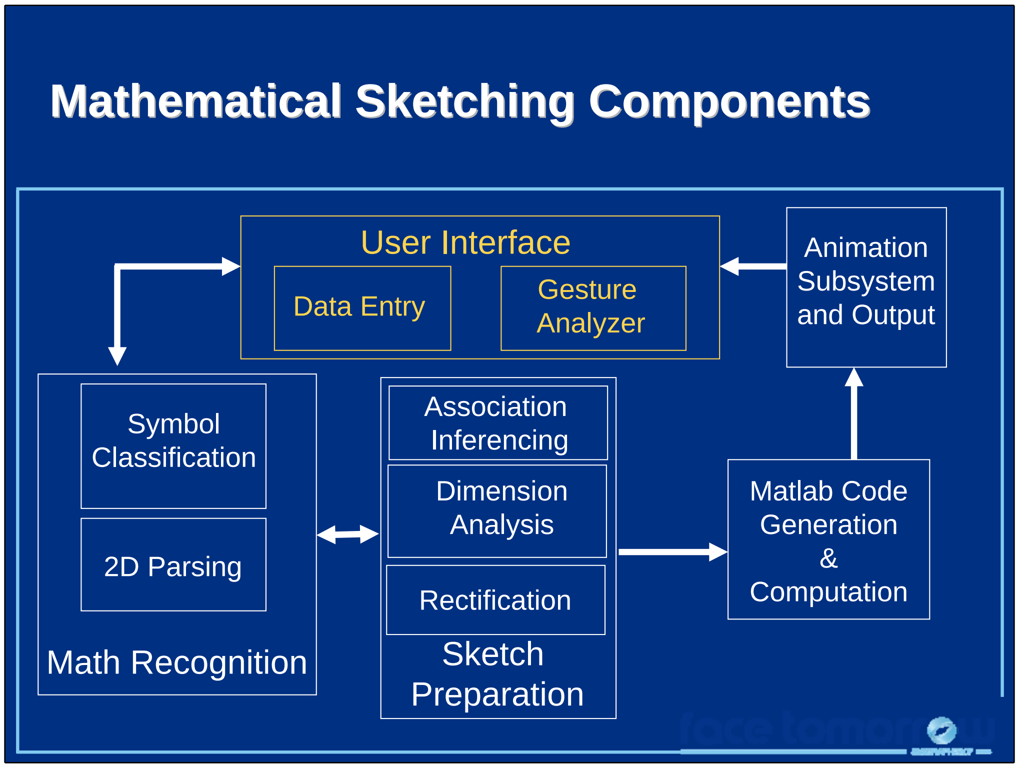
$\sum_{l=0}^{n-1} l^2$ = 1/3*n^3-1/2*n^2+1/6*n

In addition to making mathematical sketches, MathPad^2 also supports the evaluation of integrals, derivatives, and summations. It also can solve simple and simultaneous equations and graph functions.



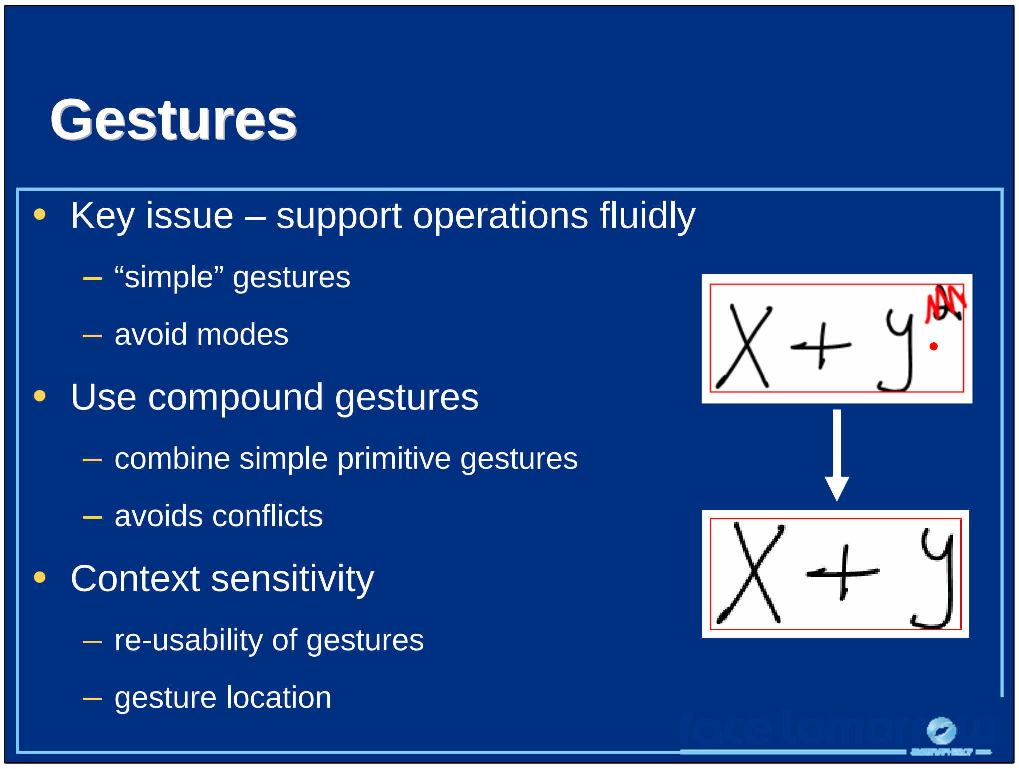
Mathematical Sketching Components

Mathematical sketching, as implemented in MathPad[2], has several different components, as shown in the slide. We will go though some of the interesting aspects of these components as they pertain to sketch-based interfaces.



Mathematical Sketching Components

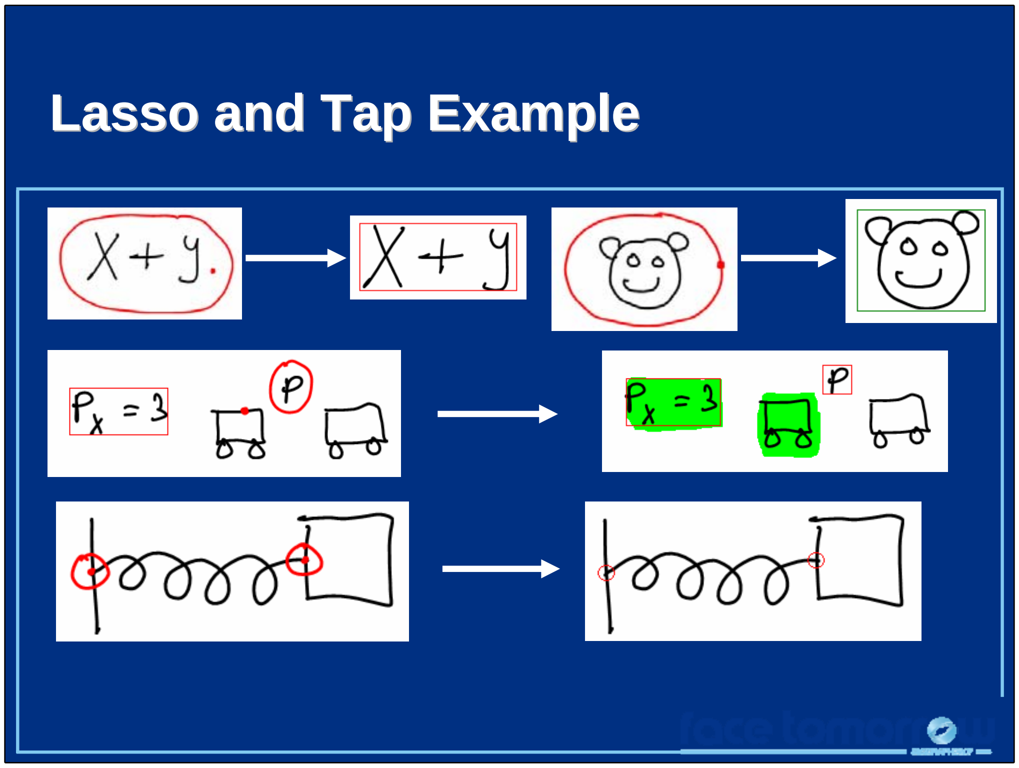
First, we will examine some of the issues with the gestural command system.



Gestures

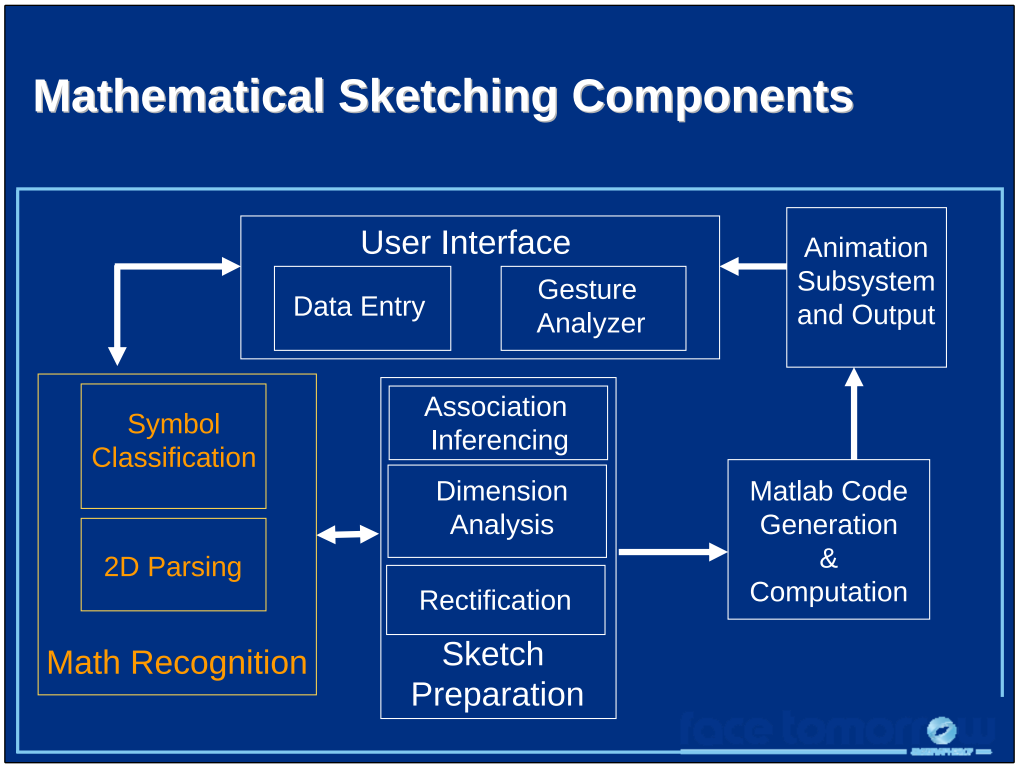
- Key issue – support operations fluidly
  - “simple” gestures
  - avoid modes
- Use compound gestures
  - combine simple primitive gestures
  - avoids conflicts
- Context sensitivity
  - re-usability of gestures
  - gesture location

One of the key issues with mathematical sketching is that users must be able to write down mathematics, make drawings and issue commands without conflict. Thus, a key issue is to support operations fluidly. Since one of the goals of mathematical sketching is to have a sketch-based interface as fluid as pencil and paper, gestural commands were needed, making mathematical sketching require both gesture recognition and sketch understanding. To support fluid operations, these gestures needed to be simple and modeless. Two strategies for dealing with this requirement are to use compound gestures and context sensitivity. Compound gestures combine simple primitive gestures together as a way to avoid conflicts between making gestural commands, making drawings, and writing mathematics. For example, consider the figure in the slide. To erase a stroke, the user scribbles over it and makes a tap. This compound gesture helps to delineate scribble erase from simply making a scribble as part of a drawing. Context sensitivity allows for the reusability of gestures for different operations. Making a gesture and looking at its location to determine the appropriate action to take is one way to take advantage of context sensitivity. An example of how context sensitivity is used is shown in the next slide.



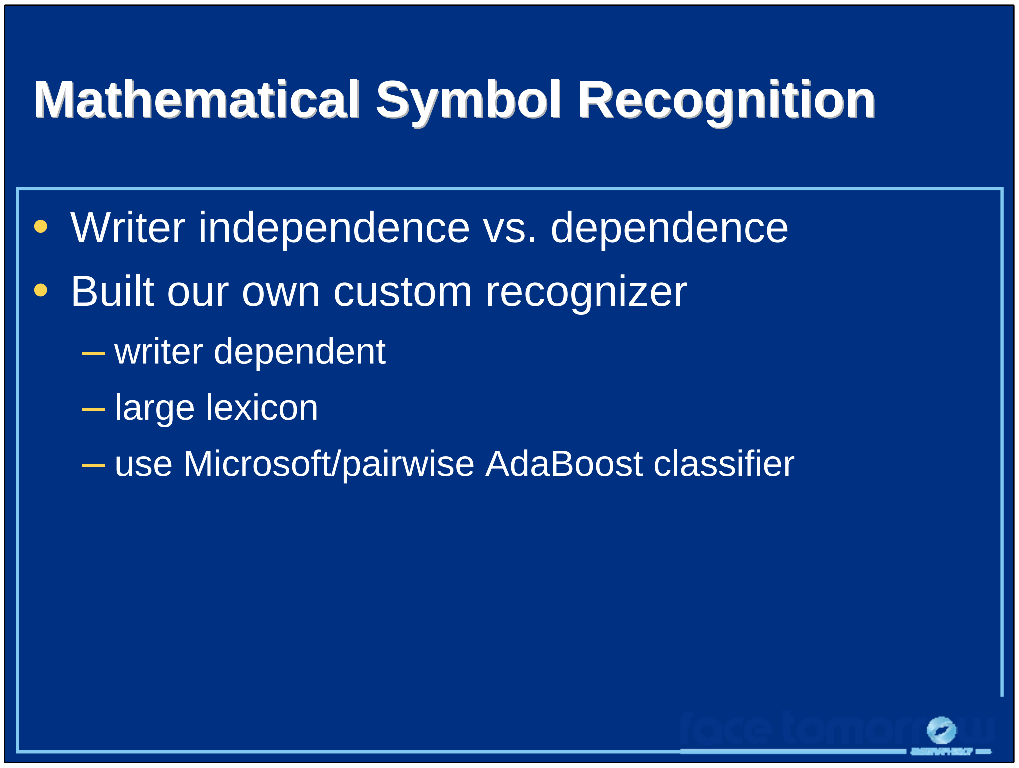
Lasso and Tap Example

The lasso and tap gesture provides a nice example of how one gesture can be used for several different commands using context sensitivity. The figures in the slide show that lasso and tap can be used for recognizing a mathematical expression, making a composite ink object, making an association, and nailing two strokes together. The gesture's location in each case is the distinguishing characteristic.



Mathematical Sketching Components

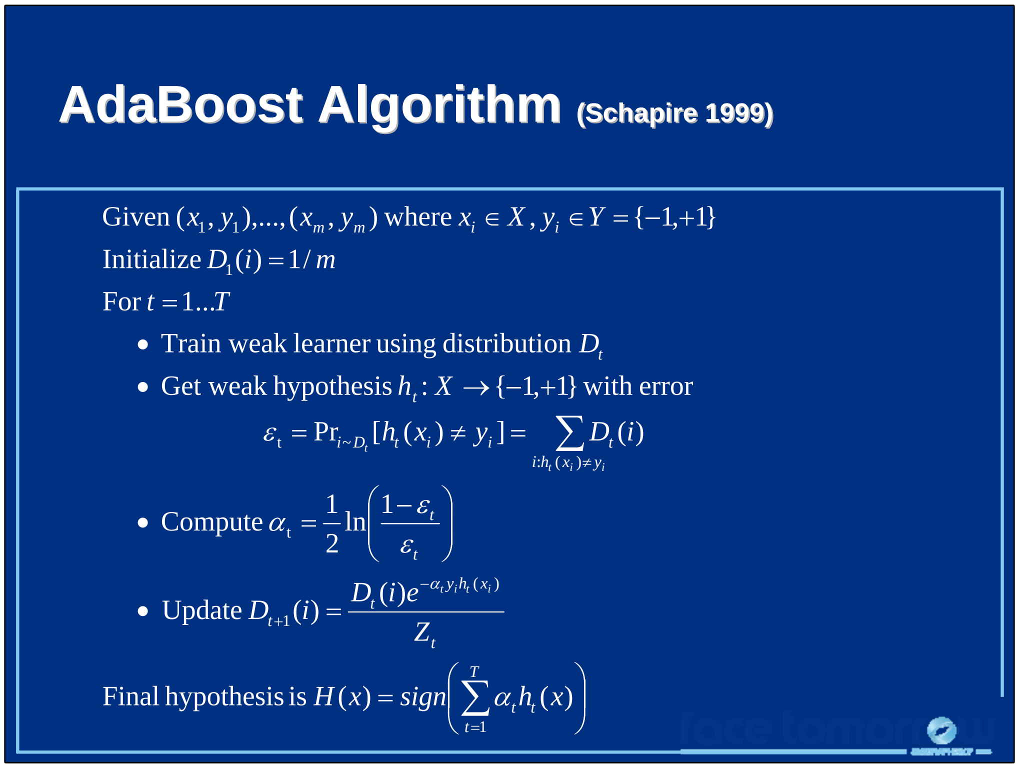
A critical part of mathematical sketching is writing mathematical expressions. Thus, a mathematical expression recognizer is needed.



Mathematical Symbol Recognition

- Writer independence vs. dependence
- Built our own custom recognizer
  - writer dependent
  - large lexicon
  - use Microsoft/pairwise AdaBoost classifier

Mathematical expression recognition consists of two parts. First, the individual symbols must be recognized. Second, the spatial relationships between these recognized symbols must be understood to give syntactic and semantic meaning to the expressions. For mathematical symbol recognition, a choice of a writer-independent or writer-dependent system must be made. Writer independent systems have the advantage that a user can start using the system with little, if any, training but its recognition accuracy is not optimized to any one user. In contrast, a writer-dependent system requires training for each user, but typically has much higher accuracy than a writer-independent approach. To achieve the highest possible accuracy, we chose a writer-dependent solution using a combination of the Microsoft handwriting recognizer and a pairwise AdaBoost classifier.



AdaBoost Algorithm (Schapire 1999)

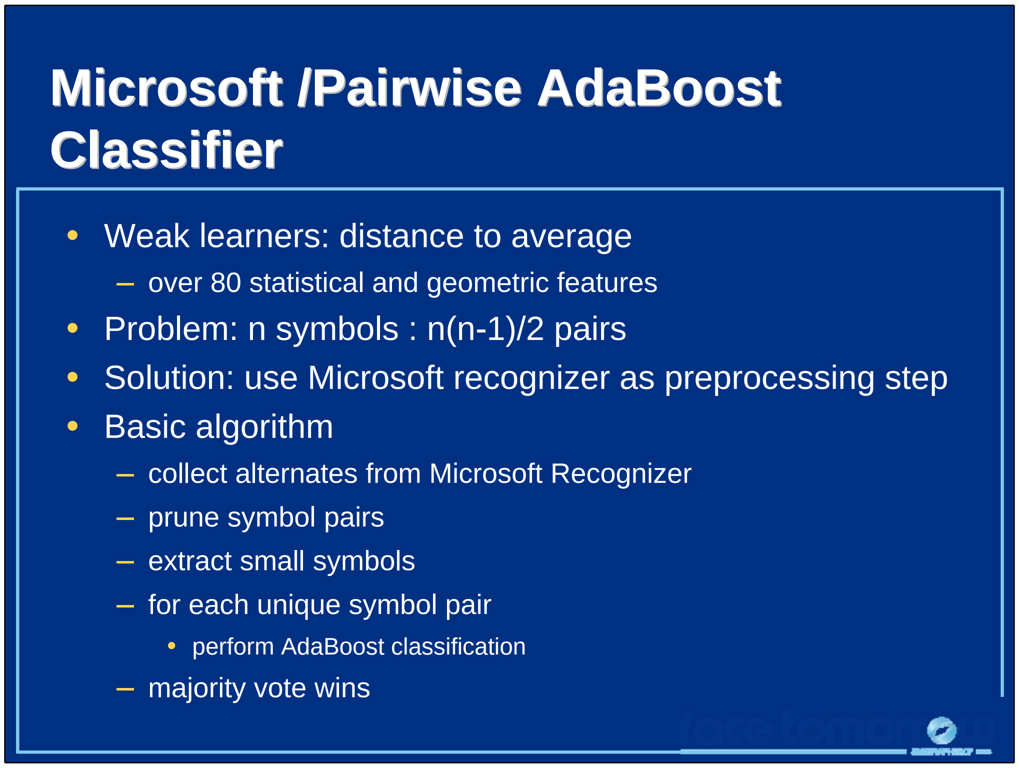
```
Given $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X, y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$
For $t = 1 \ldots T$
    • Train weak learner using distribution $D_t$
    • Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error
          $\varepsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i] = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$
    • Compute $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right)$
    • Update $D_{t+1}(i) = \frac{D_t(i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$
Final hypothesis is $H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
```

The AdaBoost algorithm was invented by Robert Schapire in mid 1990s. It is designed to take a group of weak learning algorithms (accuracy not equal to 50%) and combine them using weights, based on the training data. A pseudocode description of the general AdaBoost algorithm is shown in the slide.

References:

www.cs.princeton.edu/~schapire/boost.html

Schapire, R.. A Brief Introduction to Boosting, Proceedings of the 16th International Joint Conference on Artificial Intelligence, 1401-1406, 1999.



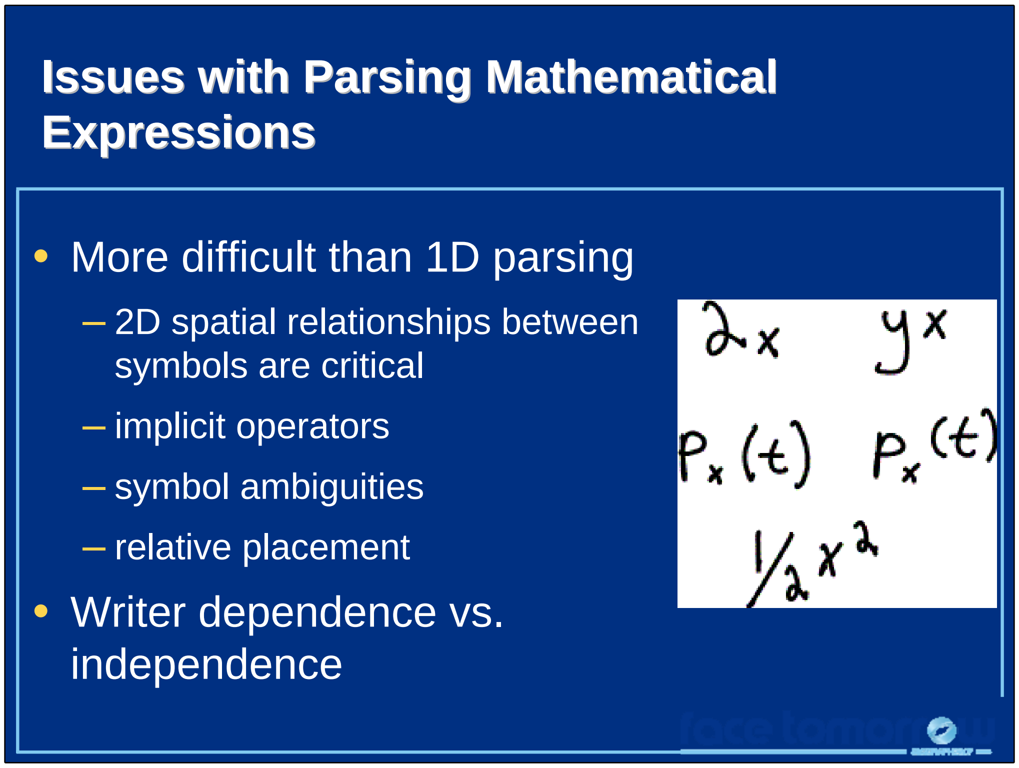
Microsoft /Pairwise AdaBoost Classifier

- Weak learners: distance to average
  - over 80 statistical and geometric features
- Problem: n symbols : n(n-1)/2 pairs
- Solution: use Microsoft recognizer as preprocessing step
- Basic algorithm
  - collect alternates from Microsoft Recognizer
  - prune symbol pairs
  - extract small symbols
  - for each unique symbol pair
    - perform AdaBoost classification
  - majority vote wins

We made the observation that it seems simpler for a recognizer to only have to make a binary decision instead of choosing from many different possibilities. Thus, we chose to use a pairwise approach to AdaBoost classification where our weak learners are based on a distance to average metric for each feature (over 80 in all). One of the issues with constructing a classifier in this way is with the number of pairwise classifiers that are needed. If we have 40 symbols in our lexicon, a total of 780 unique pairs must be checked. We found empirically that the Microsoft handwriting recognizer recognizes the correct character or has the character in its n-best list over 99% of the time. Thus, we use the Microsoft recognizer as a preprocessing step to help prune down the number of possible pairs that need to be examined. This is a very effective technique for speeding up the algorithm and increasing accuracy. The basic algorithm is shown in the slide.

References:

LaViola, J., and Zeleznik, R. A Practical Approach to Writer-Dependent Symbol Recognition Using a Writer-Independent Recognizer, To appear *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 2007.



Issues with Parsing Mathematical Expressions

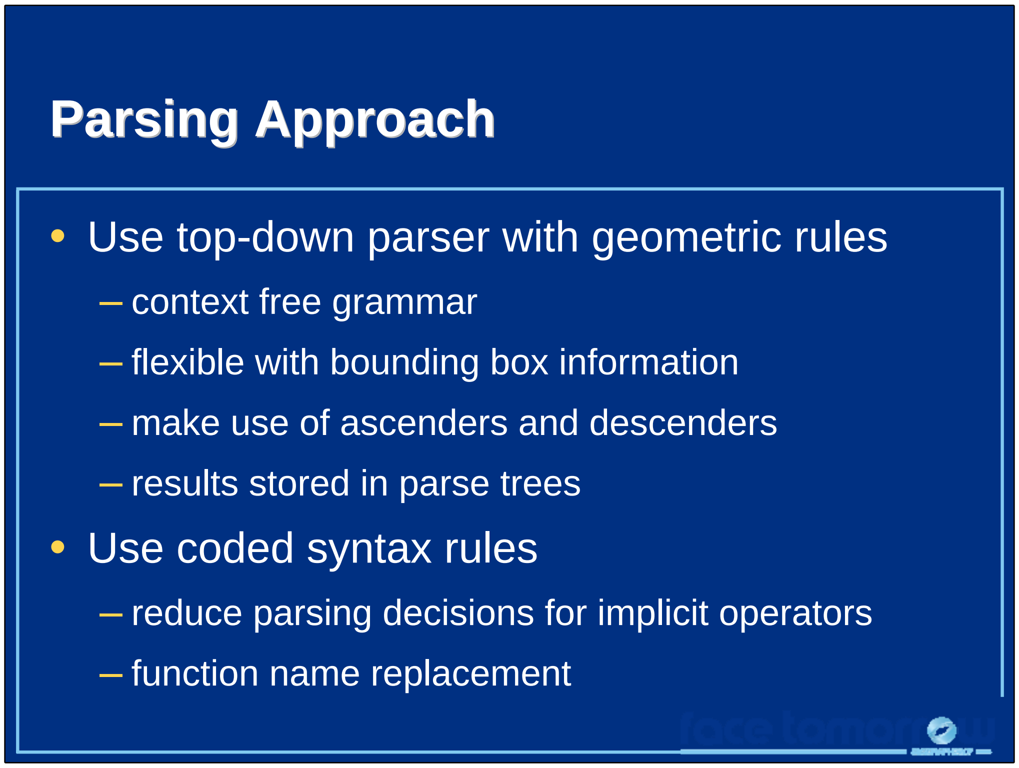
- More difficult than 1D parsing
  - 2D spatial relationships between symbols are critical
  - implicit operators
  - symbol ambiguities
  - relative placement
- Writer dependence vs. independence

$2x \quad y^x$

$P_x(t) \quad P_x^{(t)}$

$\frac{1}{2}x^2$

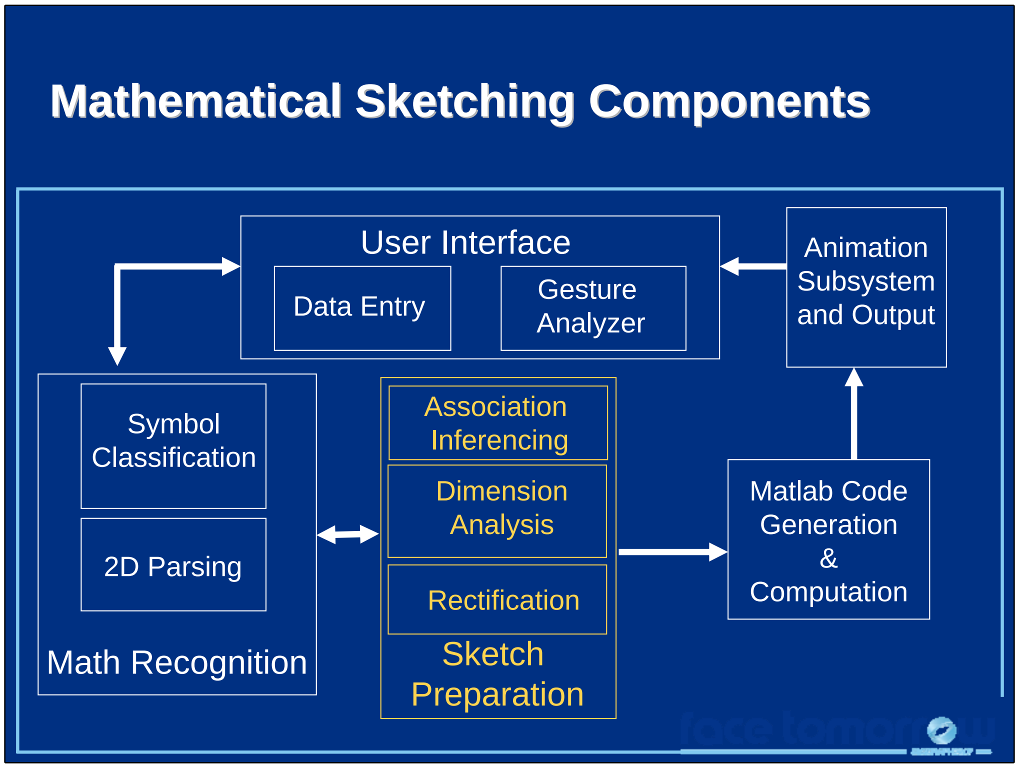
Once the mathematical symbols have been recognized, they need to be parsed. Parsing mathematical expressions is more difficult than traditional 1D parsing because the 2D spatial relationships between symbols are critical. Additionally, dealing with implicit operators (subscripts, superscripts and implicit multiplication), symbol ambiguities and the variability of symbol placement cause a multitude of problems. As with symbol recognition, expression parsing systems can be writer-dependent or writer-independent. Making a mathematical expression parser writer-dependent is more difficult than having writer-dependent symbol recognition because of the many more possible parses and spatial relationships that can occur with mathematical expressions. The alternative to this is to use a writer-independent approach, making spatial relationship rules flexible enough for the general user. In this case, we chose a writer-independent parsing system.



Parsing Approach

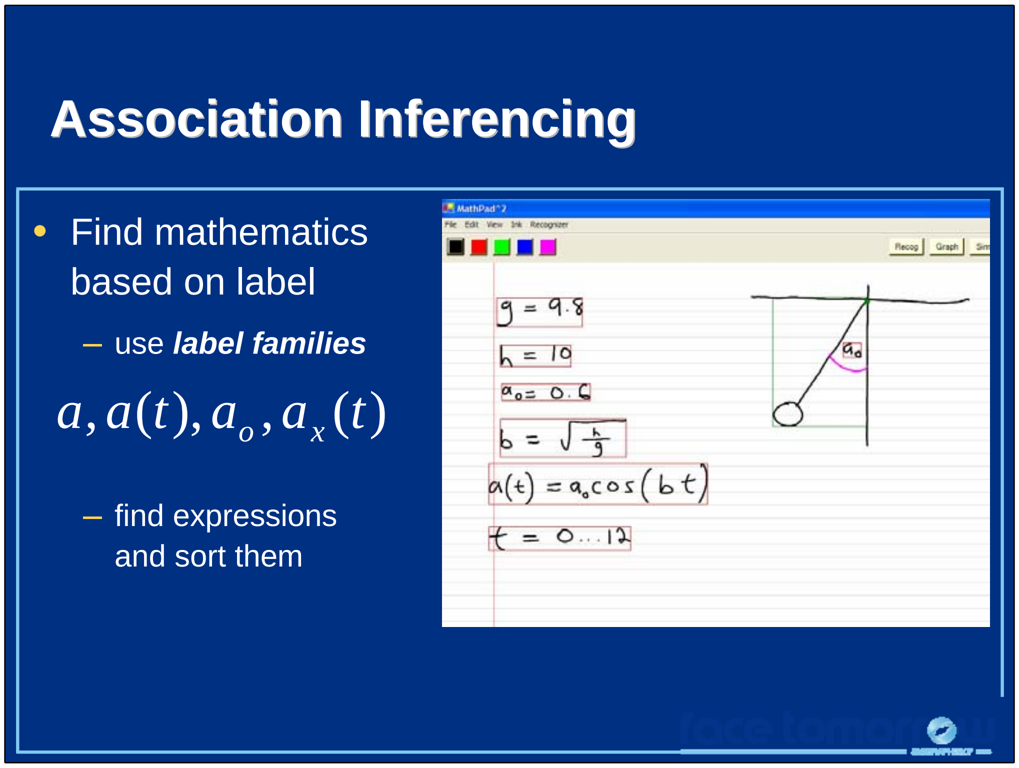
- Use top-down parser with geometric rules
  - context free grammar
  - flexible with bounding box information
  - make use of ascenders and descenders
  - results stored in parse trees
- Use coded syntax rules
  - reduce parsing decisions for implicit operators
  - function name replacement

We use a top-down parsing strategy with embedded geometric rules. The parser uses a context-free grammar and takes the bounding boxes for each recognized symbol to determine the relationships between them. The parser also makes use of some underlying knowledge about how letters are written, whether they are ascenders (e.g., b,f,t) or descenders (e.g., p,y,g), for dealing with implicit operators. The results of the parse are stored in parse trees for further processing. The parser also makes use of procedurally coded syntax rules. These rules are used to help reduce the number of parsing decisions that need to be made. For example, with mathematical sketching, a number with a subscript has no meaning. Thus it is not a valid parse and can be ignored. Other procedurally coded rules include function name replacement. For example, if the user is writing *cos* and the recognizer comes back with *co5*, then the parser will assume the user meant *cos*.



Mathematical Sketching Components

With the mathematical expressions recognized, the mathematical sketch has to be prepared so it can be executed.

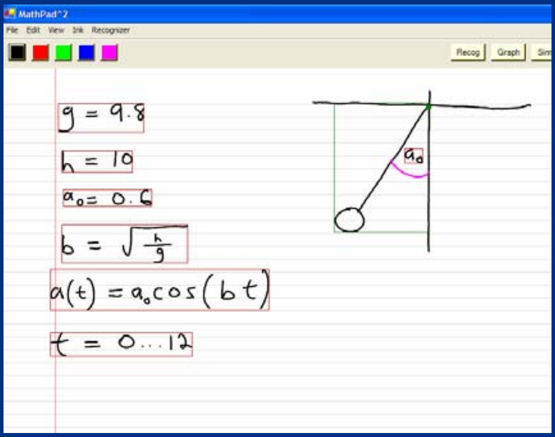


Association Inferencing

- Find mathematics based on label
  - use ***label families***

$$a, a(t), a_o, a_x(t)$$

  - find expressions and sort them



$g = 9.8$

$h = 10$

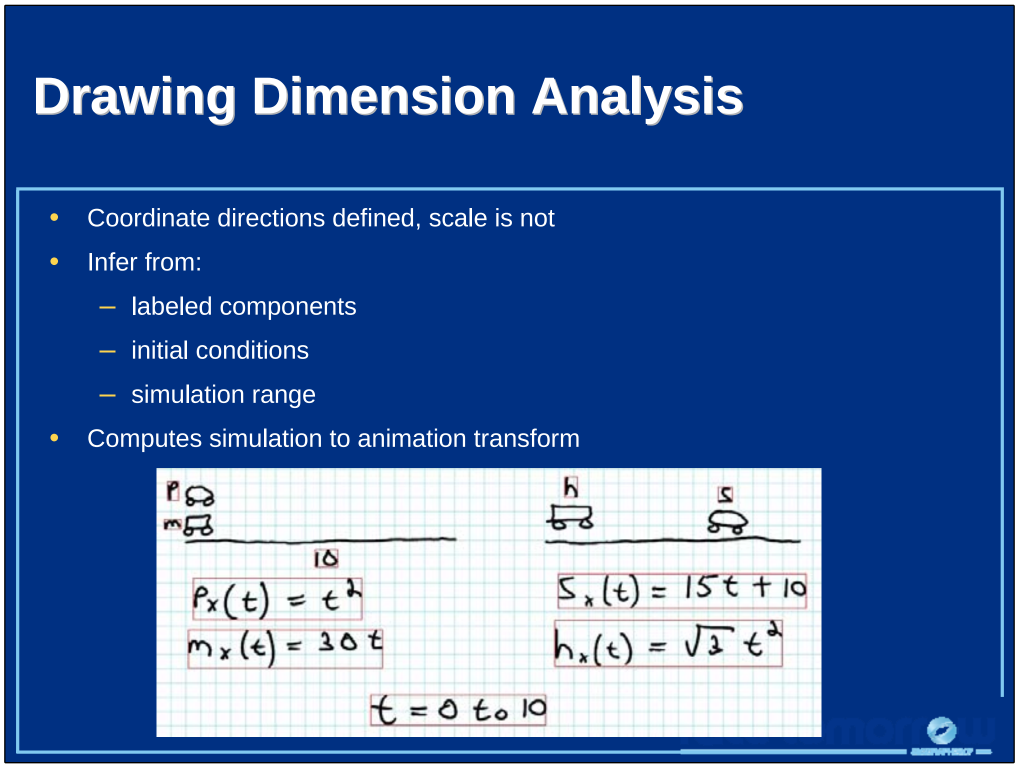
$a_0 = 0.6$

$b = \sqrt{\frac{h}{g}}$

$a(t) = a_0 \cos(bt)$

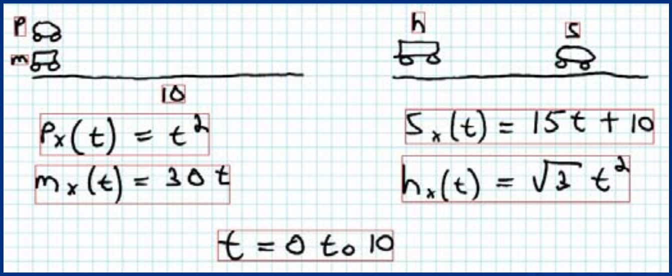
$t = 0 \ldots 12$

When users make illustrations to help them understand mathematics and physics problems, they often label individual parts of the illustration to give them names. Mathematical sketching uses this strategy for making associations. Users can label a drawing element and that label will act as a guide to finding all of the necessary mathematics that is needed to animate that element. We use the notion of label families to infer which mathematical expressions should be included as part a drawing element's behavior. In the figure in the slide, a user labels the angle between the pendulum and vertical line with a_0. MathPad^2 uses this label to find all of the mathematical expressions which contain an *a* on their left hand sides. Then, using these expressions, the right hand sides are analyzed for variables, function names, and constants, which are then used to find other mathematical expressions, and so on until all of the required mathematics has been found. Once all of the expressions have been found, they are sorted, constants first, then variables defined by other variables, and finally functions.

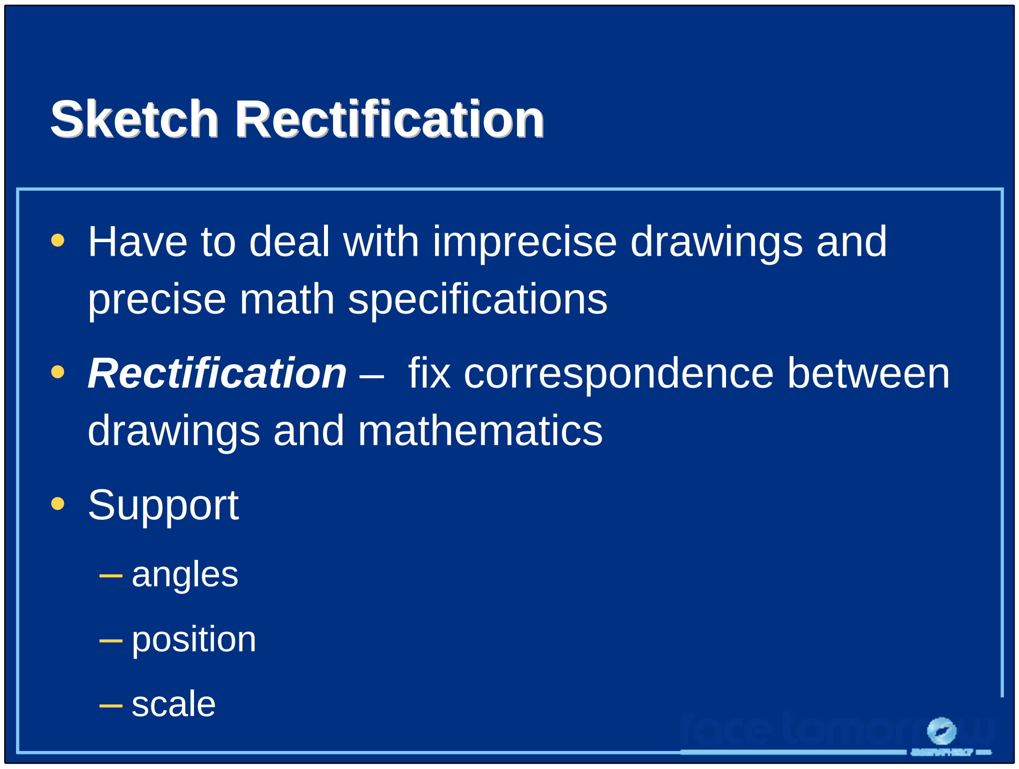


Drawing Dimension Analysis

- Coordinate directions defined, scale is not
- Infer from:
  - labeled components
  - initial conditions
  - simulation range
- Computes simulation to animation transform



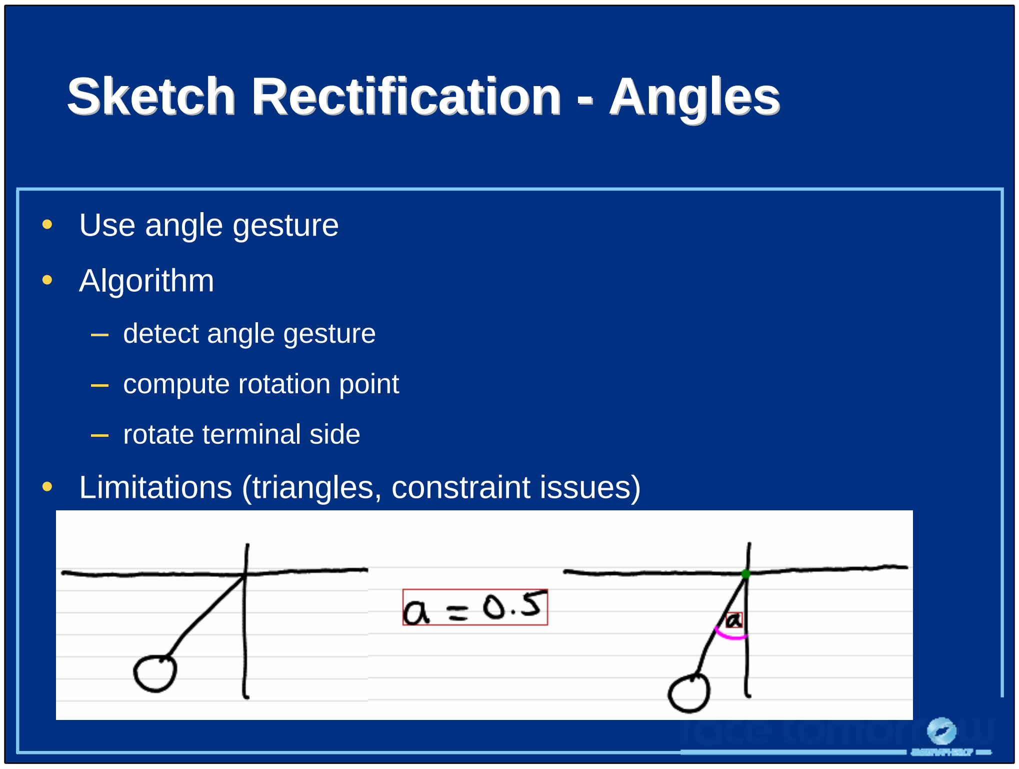
The coordinate system in MathPad[2] is predefined with the +*x* axis pointing to the right and the +*y* axis pointing up. However, what is not defined is the scale of the coordinate system (i.e., how much one unit is in either the *x* or *y* direction in screen space). What is needed is a drawing dimension analysis to give us a mapping between simulation space and animation space. The system attempts to infer what this mapping is by looking at the sketch. There are several ways of finding this mapping, including examining labeled components, initial conditions, and the simulation range. The figure in the slide shows two example sketches. In the first one, on the left side of the figure, the road is labeled with a 10. This label tells us that the value 10 in simulation space is equal to the length of the line in screen space. This provides a mapping for the *x* direction (it could also be used for the *y* direction as well). The sketch on the right of the figure does not have the road labeled. However, if we look at the initial values of the functions $s_x(t)$ and $h_x(t)$, we see that they are 10 and 0 respectively. This gives us a length to use between the two cars that will provide an appropriate mapping. If no mapping can be inferred, then a default one is used.



Sketch Rectification

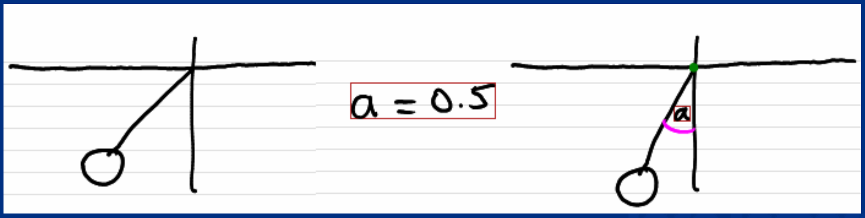
- Have to deal with imprecise drawings and precise math specifications
- ***Rectification*** – fix correspondence between drawings and mathematics
- Support
  - angles
  - position
  - scale

The nature of mathematical sketching implies that precise mathematical specifications must interact with approximate, free-form drawings. Thus, there will be mismatches between the two that need to be rectified. Rectification is the process of fixing the correspondence between precise mathematical specifications and imprecise drawings. Currently MathPad^2 supports rudimentary rectification for angles, position and scale.

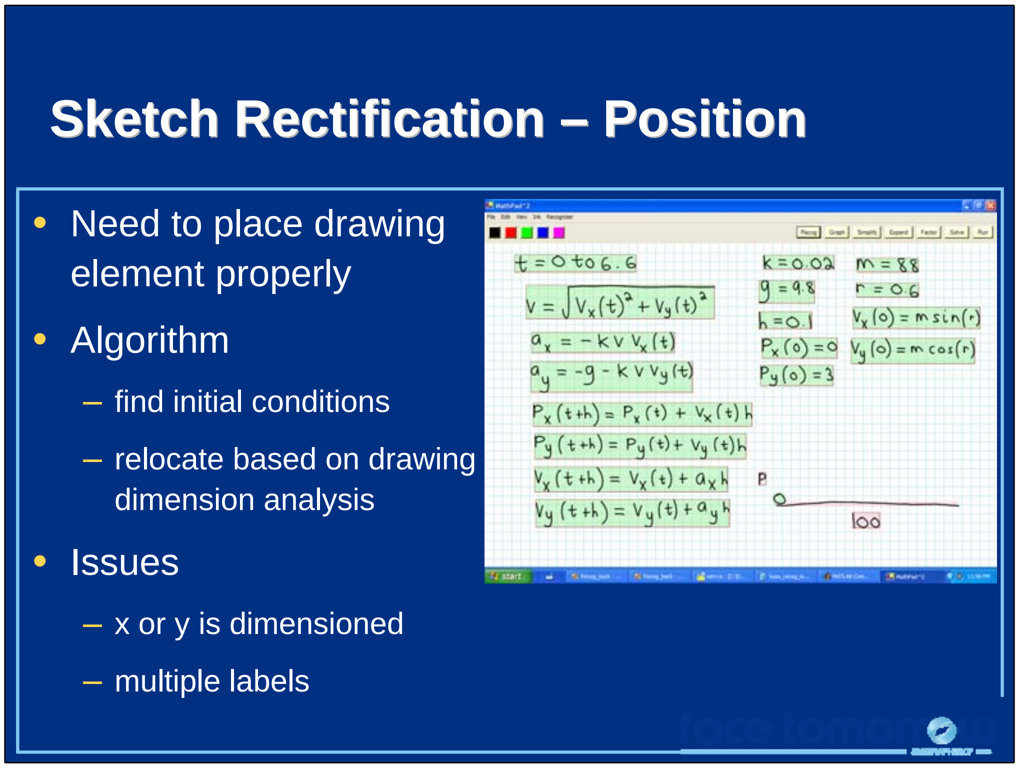


Sketch Rectification - Angles

- Use angle gesture
- Algorithm
  - detect angle gesture
  - compute rotation point
  - rotate terminal side
- Limitations (triangles, constraint issues)

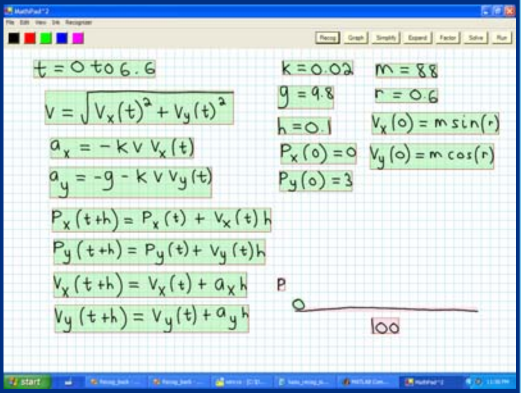


When a users makes a drawing that has an angle that they know needs to be a certain value. They make the drawing but the angle value may not be correct. In the figure, the user wants to make an angle between a pendulum and a vertical line. The value of this angle should be 0.5 radians. Angle rectification occurs when the user labels the angle. The system detects an angle gesture and finds the enclosed symbol (in this case the letter a). Using the angle gesture, a point of rotation is computed (the green dot on the right side of the figure). This rotation point is used to rotate the angle's terminal side (the pendulum) so that it is 0.5 radians from the vertical line. This approach breaks down when dealing with over-constrained problems, such as dealing with triangles. Rectifying one of the three angle's that make up a the triangle will potentially affect not only the other two angles but also the lengths of the sides of the triangle. This type of problem is typical in constraint satisfaction systems.



Sketch Rectification – Position

- Need to place drawing element properly
- Algorithm
  - find initial conditions
  - relocate based on drawing dimension analysis
- Issues
  - x or y is dimensioned
  - multiple labels



MathPad^2
File Edit View Ink Recognizer
Recog Graph Simplify Expand Factor Solve Run
P
100
start

$t = 0$ to 6.6

$V = \sqrt{V_x(t)^2 + V_y(t)^2}$

$a_x = -k\,V\,V_x(t)$

$a_y = -g - k\,V\,V_y(t)$

$P_x(t+h) = P_x(t) + V_x(t)h$

$P_y(t+h) = P_y(t) + V_y(t)h$

$V_x(t+h) = V_x(t) + a_x h$

$V_y(t+h) = V_y(t) + a_y h$

$k = 0.02$

$g = 9.8$

$h = 0.1$

$P_x(0) = 0$

$P_y(0) = 3$

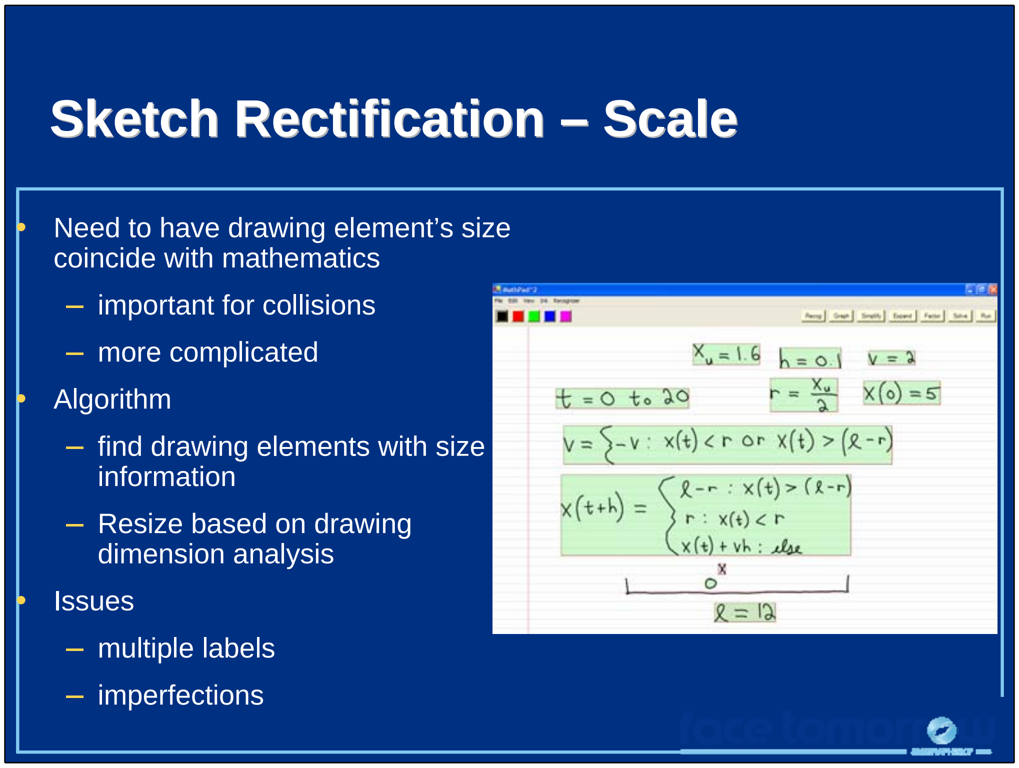
$m = 88$

$r = 0.6$

$V_x(0) = m\sin(r)$

$V_y(0) = m\cos(r)$

Fixing the position of a drawing element is also an important rectification problem, since they need to be placed properly give the origin and dimensions of the coordinate system. MathPad[2] finds any initial conditions in the mathematical specifications for drawing elements and then relocates them based on the information in the drawing analysis step. The origin of the coordinate system is also found during the drawing dimension analysis step and is located at the left most point of the drawing element used in the analysis step. In the figure in the slide, the origin is the leftmost point of the line labeled as 100 units long. Ambiguities can occur when there is more than one numerically labeled drawing element. In these cases, there is no clear cut answer to the problem. However, making suggestions to the user is a viable approach.



Sketch Rectification – Scale

- Need to have drawing element's size coincide with mathematics
  - important for collisions
  - more complicated
- Algorithm
  - find drawing elements with size information
  - Resize based on drawing dimension analysis
- Issues
  - multiple labels
  - imperfections

$X_u = 1.6$ $h = 0.1$ $v = 2$

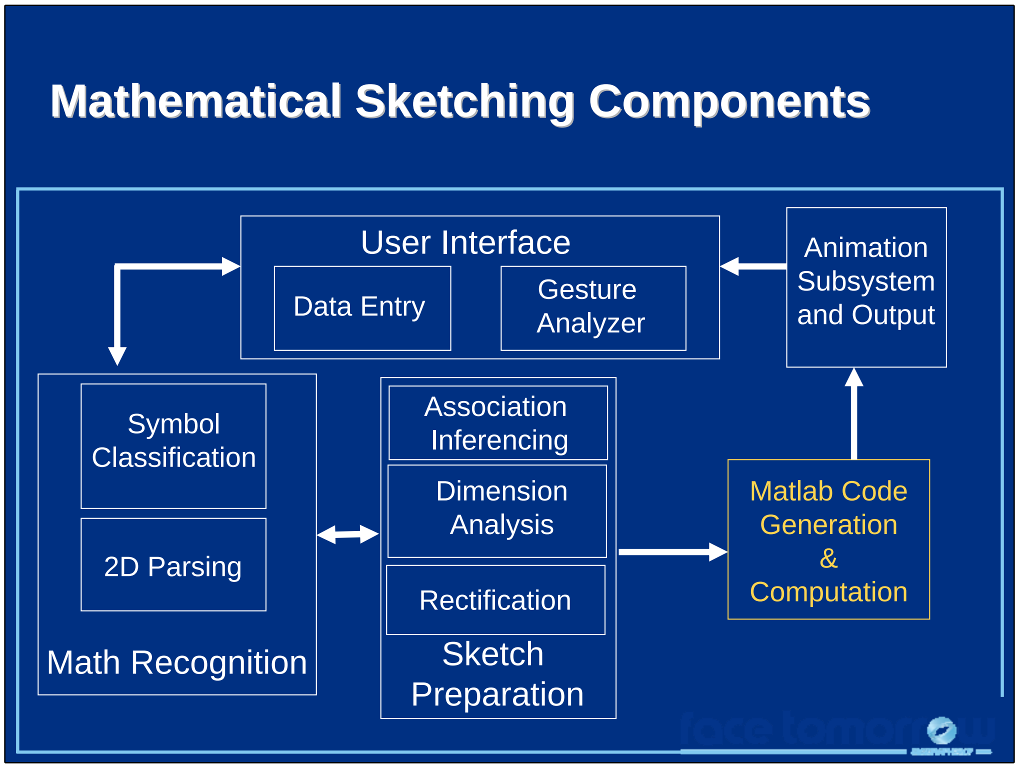
$t = 0$ to 20 $r = \frac{X_u}{2}$ $X(0) = 5$

$v = \{ -v : x(t) < r$ or $x(t) > (\ell - r)$

$$x(t+h) = \begin{cases} \ell - r : x(t) > (\ell - r) \\ r : x(t) < r \\ x(t) + vh : else \end{cases}$$

$\ell = 12$

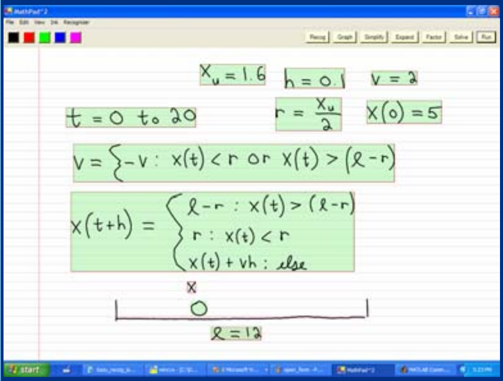
As with position, scale rectification is important when the size of a drawing element has relevance to the mathematical sketch. One simple way to rectify scale is to use mathematics to specify the size of the drawing element. In the figure in the slide, the ball, labeled as x, has a size value of 16, which is denoted with x_u. The subscript *u* stands for uniform scaling. If *w* or *h* was used, it would indicate the width or height of the drawing element was 16. With this information, resizing of the drawing element can be done using the mapping from the dimension analysis step. As with position rectification, multiple numerically labeled drawing elements can be problematic. Additionally, imperfections in how the drawing element was made can also cause problems. In these cases, further rectification of the stroke geometry is required.


Mathematical Sketching Components

Given recognized mathematical expressions and a prepared mathematical sketch, executable code needs to be generated for the purposes of getting animation data.

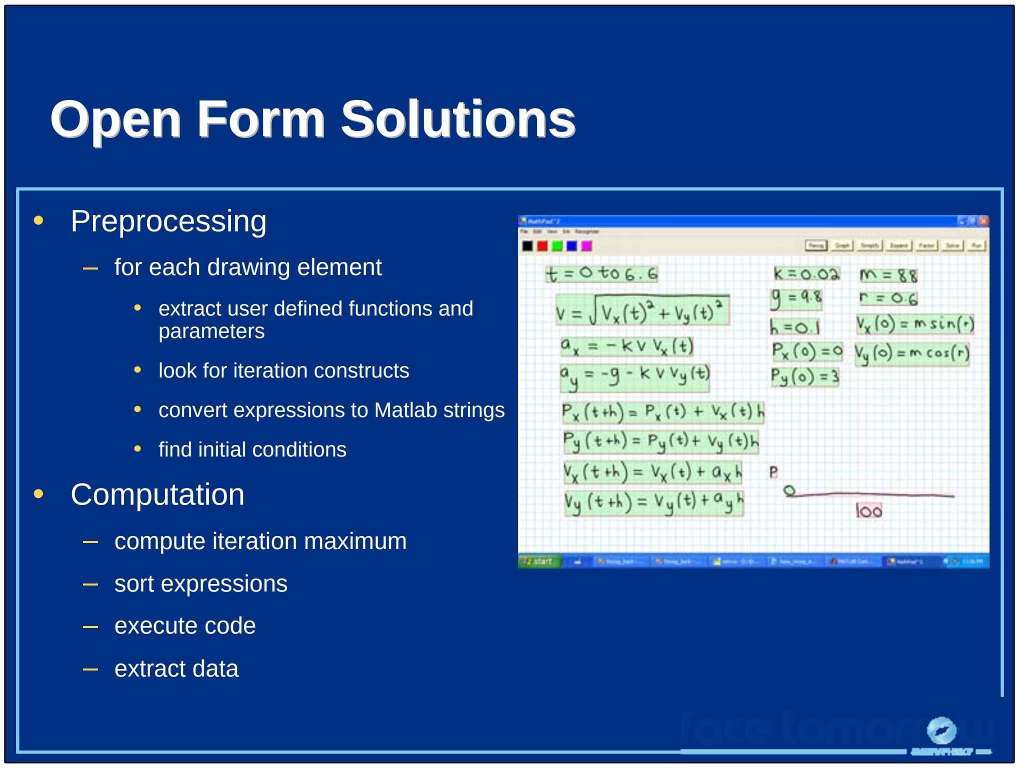
Generating Executable Code

- Support closed and open form solutions
- Use Matlab syntax


$X_u = 1.6$
$h = 0.1$
$v = 2$
$t = 0$ to 20
$r = \frac{X_u}{2}$
$x(0) = 5$
$v = \{-v : x(t) < r$ or $x(t) > (l - r)$
$x(t+h) = \begin{cases} l-r : x(t) > (l-r) \\ r : x(t) < r \\ x(t) + vh : else \end{cases}$
$l = 12$

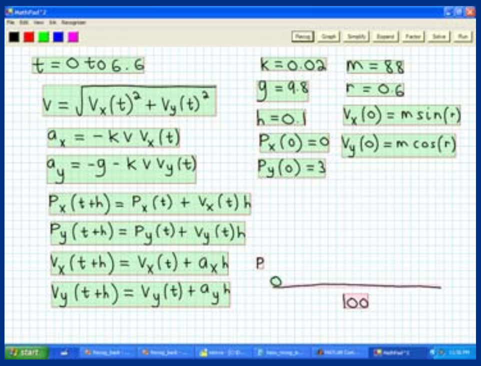
```
h = 0.1
x_u = 1.6
v = 2
r = x_u/2
l = 12
x(1) = 5
for i = 2:200
  if (x(i-1)<r | x(i-1)>(l-r))
    v = -v;
  end
  if (x(i-1)>(l-r))
    x(i) = l-r;
  elseif (x(i-1)<r)
    x(i) = r;
  else
    x(i) = x(i-1)+v*h;
  end
end
```

MathPad^2 currently supports open and closed form solutions and requires executable code to be in Matlab syntax, given we use Matlab as our computational engine. Our goal is then to transform the mathematics in the figure on the left of the slide to the code on the right.



Open Form Solutions

- Preprocessing
  - for each drawing element
    - extract user defined functions and parameters
    - look for iteration constructs
    - convert expressions to Matlab strings
    - find initial conditions
- Computation
  - compute iteration maximum
  - sort expressions
  - execute code
  - extract data



$t = 0 \text{ to } 6.6$

$V = \sqrt{V_x(t)^2 + V_y(t)^2}$

$a_x = -k V V_x(t)$

$a_y = -g - k V V_y(t)$

$P_x(t+h) = P_x(t) + V_x(t) h$

$P_y(t+h) = P_y(t) + V_y(t) h$

$V_x(t+h) = V_x(t) + a_x h$

$V_y(t+h) = V_y(t) + a_y h$

$k = 0.02$ $m = 88$

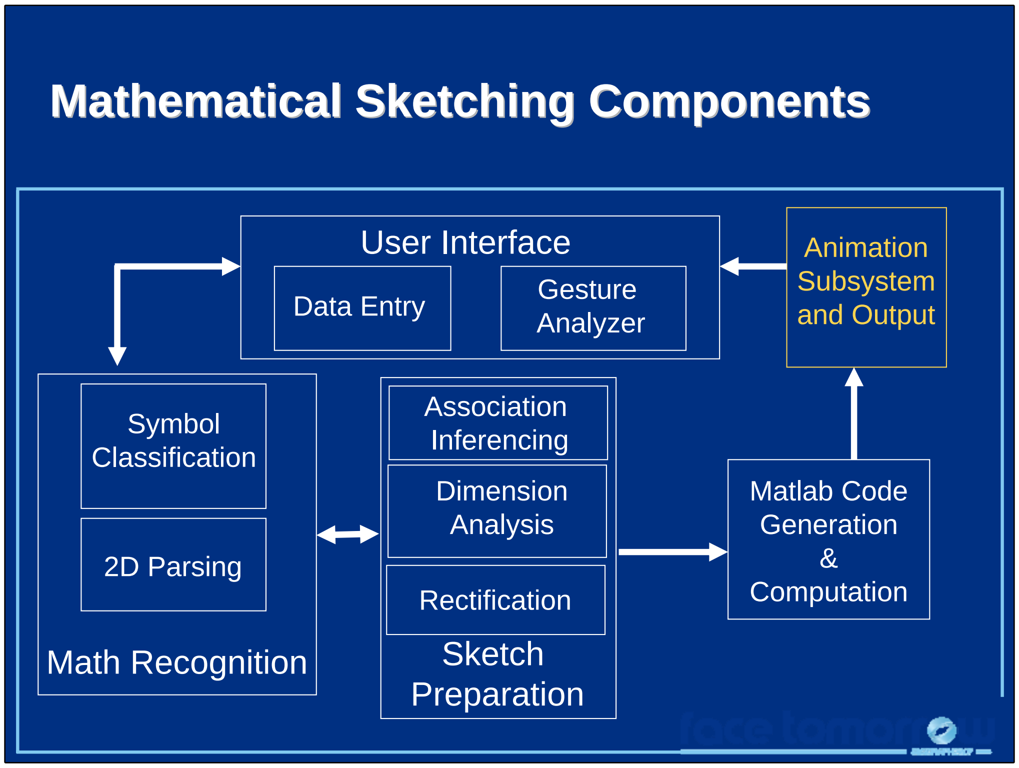
$g = 9.8$ $r = 0.6$

$h = 0.1$ $V_x(0) = m \sin(r)$

$P_x(0) = 0$ $V_y(0) = m \cos(r)$

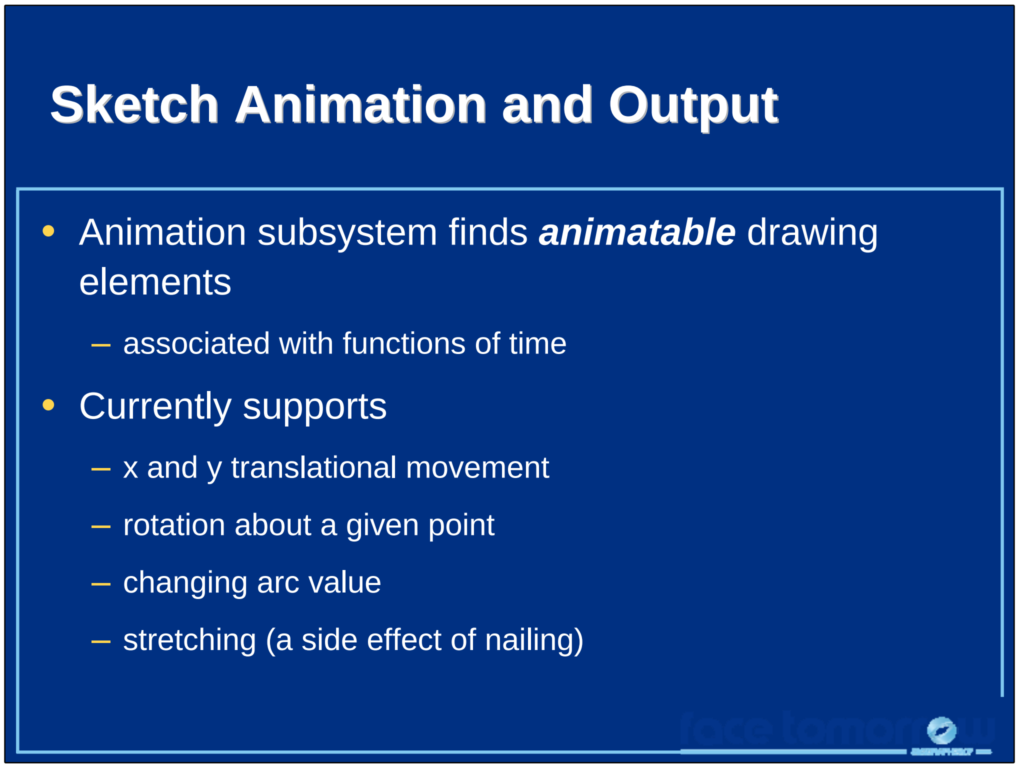
$P_y(0) = 3$

Closed-form solutions are fairly straightforward. However, open-form solutions are slightly more complicated. There are two important steps to converting a mathematical sketch into executable code, a preprocessing step and the computation step. In the preprocessing step, for each drawing element, user defined functions and parameters need to be extracted. Next, any iteration constructs are found. Third, all the necessary mathematical expressions are converted to Matlab strings. Finally, initial conditions need to be found. With this information, the computation step can proceed by first finding the iteration maximum based on the iteration construct and the delta value (in the figure, the delta value is *h*). The mathematical expressions need to be sorted so they will execute in a logical order. The expressions are then executed and the appropriate data is extracted from Matlab.



Mathematical Sketching Components

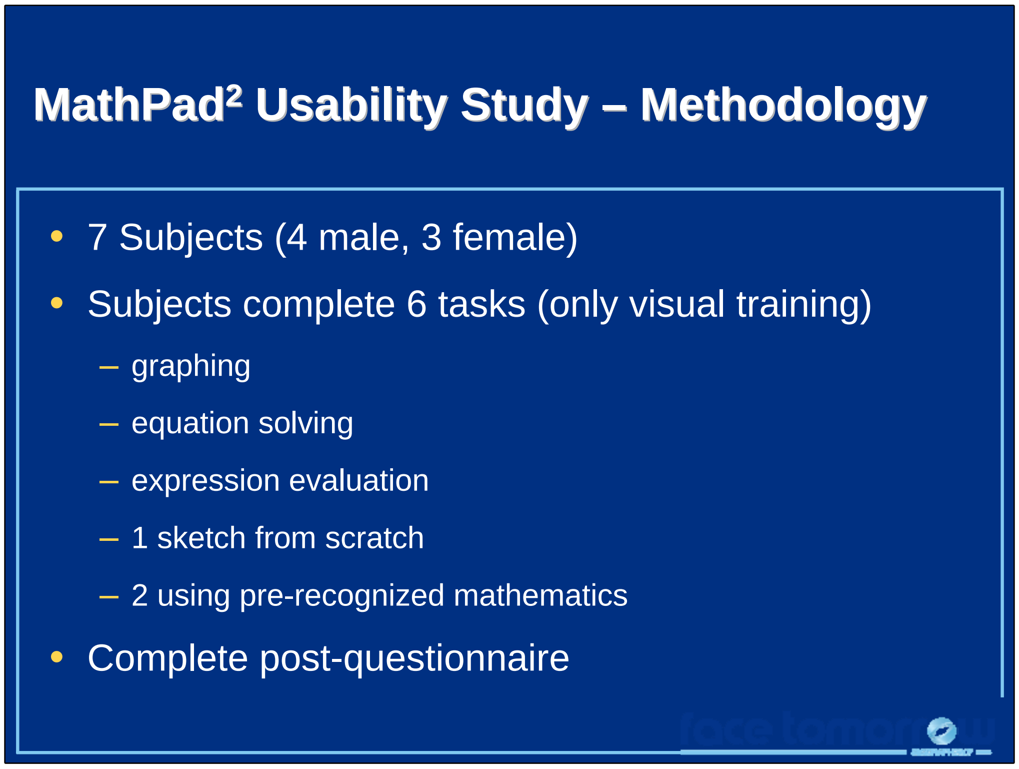
Once we have the data from Matlab, the sketch can be animated.



Sketch Animation and Output

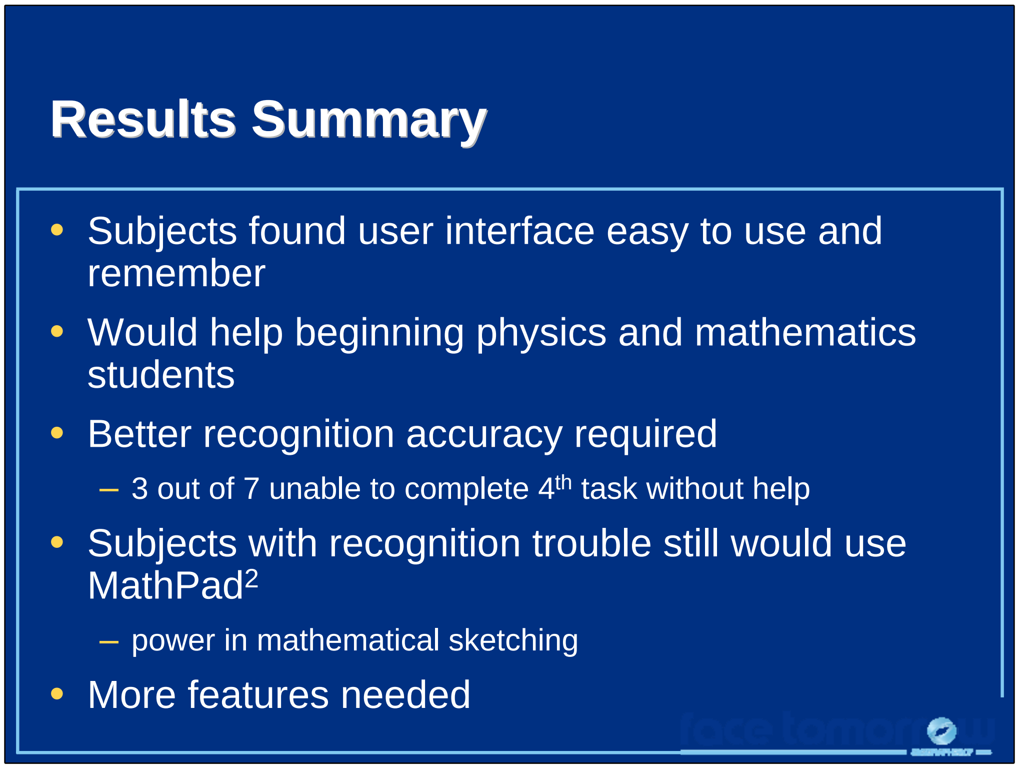
- Animation subsystem finds ***animatable*** drawing elements
  - associated with functions of time
- Currently supports
  - x and y translational movement
  - rotation about a given point
  - changing arc value
  - stretching (a side effect of nailing)

The animation system finds all drawing elements that are *animatable* (i.e., associated with functions of time). The animation system then takes the data and animates the sketch appropriately. Animations include *x* and *y* translational motion, rotation about a point, changing of an arc value, and the stretching of objects.



MathPad^2 Usability Study – Methodology

- 7 Subjects (4 male, 3 female)
- Subjects complete 6 tasks (only visual training)
  - graphing
  - equation solving
  - expression evaluation
  - 1 sketch from scratch
  - 2 using pre-recognized mathematics
- Complete post-questionnaire

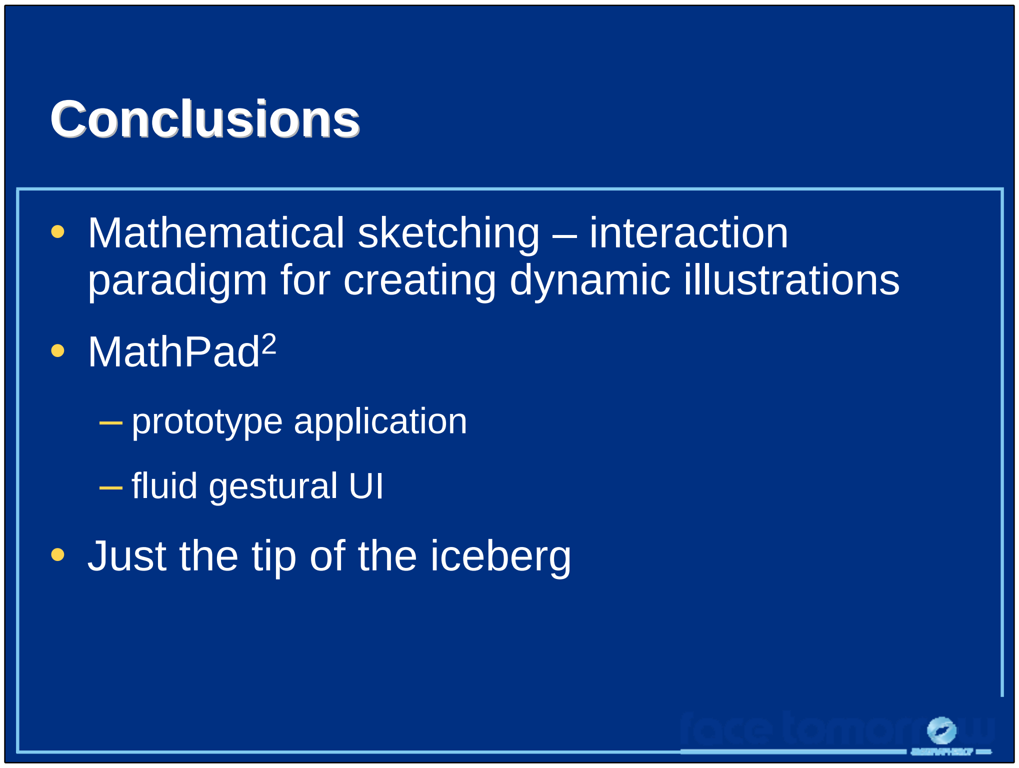


Results Summary

- Subjects found user interface easy to use and remember
- Would help beginning physics and mathematics students
- Better recognition accuracy required
  - 3 out of 7 unable to complete 4th task without help
- Subjects with recognition trouble still would use MathPad^2
  - power in mathematical sketching
- More features needed

References:

LaViola, J. An Initial Evaluation of a Pen-Based Tool for Creating Dynamic Mathematical Illustrations, Proceedings of the *Eurographics Workshop on Sketch-Based Interfaces and Modeling 2006*, 157-164, September 2006.



Conclusions

- Mathematical sketching – interaction paradigm for creating dynamic illustrations
- MathPad^2
  - prototype application
  - fluid gestural UI
- Just the tip of the iceberg

Mathematical sketching is an interaction paradigm that uses a sketch-based interface for creating dynamic illustrations with the combination of handwritten mathematics and free-form drawings. MathPad^2 is a prototype application for exploring ideas in mathematical sketching. Mathematical sketching is in its very early stages and there is a significant amount of work to do before the ultimate goal of mathematical sketching is realized. However, MathPad^2 shows that mathematical sketching is viable and is an good example of a sketch-based interface.

Note that this lecture only gave a cursory overview of the ideas and details behind mathematical sketching. A more thorough discussion can be found in

LaViola, J. *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*, Ph.D. Dissertation, Brown University, Department of Computer Science, May 2005.

Selected Papers

Sketching and Gestures 101

Interaction techniques for ambiguity resolution in recognition-based interfaces

Jennifer Mankoff[1], Scott E. Hudson[2], and Gregory D. Abowd[1]

[1] College of Computing & GVU Center
Georgia Institute of Technology
Atlanta, GA 30332-0280
{mankoff,abowd}@cc.gatech.edu

[2] HCI Institute
Carnegie Mellon University
Pittsburgh, PA 15213
hudson@cs.cmu.edu

ABSTRACT

Because of its promise of natural interaction, recognition is coming into its own as a mainstream technology for use with computers. Both commercial and research applications are beginning to use it extensively. However the errors made by recognizers can be quite costly, and this is increasingly becoming a focus for researchers. We present a survey of existing error correction techniques in the user interface. These *mediation* techniques most commonly fall into one of two strategies, repetition and choice. Based on the needs uncovered by this survey, we have developed OOPS, a toolkit that supports resolution of input ambiguity through mediation. This paper describes four new interaction techniques built using OOPS, and the toolkit mechanisms required to build them. These interaction techniques each address problems not directly handled by standard approaches to mediation, and can all be re-used in a variety of settings.

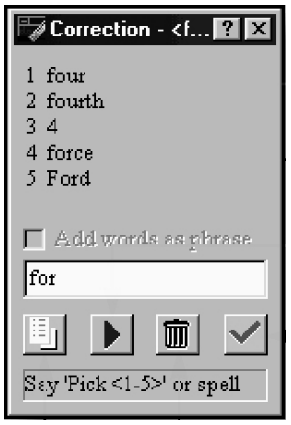
INTRODUCTION

Because of its promise of natural interaction, recognition is coming into its own as a mainstream technology for use with computers. Recognition is being used in personal assistants such as the PalmPilot™, as well as on the desktop (*e.g.* IBM's ViaVoice™). Research initiatives in areas such as multimodal computing are investigating how to build effective, usable applications involving recognizers.

However, the errors made by recognizers can be quite costly and annoying for users to correct, and this is increasingly becoming a focus of research [7,11,23,28,30] For example, when studying a speech dictation system, Halverson *et al*. found that input speeds decrease from the 120 words per minute (wpm) of conversational speech to 25 wpm due in large part to time spent correcting recognition errors [11]. These errors are corrected through explicit user interaction. For example, a user can delete misrecognized words and then repeat them.

This *repetition* of input is one of the two common classes of interaction techniques for correcting recognition errors. The other common class gives the user a *choice* of different possible interpretations of her input. We call repetition and choice *mediation techniques* because they are mediating between the user and the computer to specify the correct interpretation of the user's input. Tables 1 and 2 show some of the variations of both repetition and choice techniques that we found in the literature. Figure 1 shows an example of a hybrid mediator from IBM's ViaVoice™ system. It provides a choice technique (an *n*-best list) with an escape path to a repetition technique. Other commercial systems such as the Apple MessagePad™ and DragonDictate™ provide similar hybrids. Choice and repetition strategies have a fairly wide range of possible instantiations, making mediation techniques ripe for reusable toolkit-level support.

In general, the goal of perfect recognition is difficult to achieve because correct recognition is best defined as what the user intends. Since a system cannot know this *a priori*, we model possible interpretations of user input internally as *ambiguous* input. Mediation techniques then serve to resolve this ambiguity, helping to determine which of those potential interpretations is the correct one, the one the user intended. In order to do this properly, integrated architectural support for ambiguity at the input-handling level is required. This makes it possible to track which interactors use ambiguous information and will need to be


Correction - <f...

1 four
2 fourth
3 4
4 force
5 Ford

Add words as phrase

for

Say 'Pick <1-5>' or spell

Figure 1: An n-best list from the ViaVoice™ speech system [3]. Note that it provides a text entry area for mediation by repetition. Illustration reprinted with permission from IBM Corporation.

notified when interpretations are accepted or rejected. With such integrated support, we can treat all events the same way whether they are generated by a recognizer, sensor, mouse, or keyboard. We have developed a toolkit, called the Organized Option Pruning System (OOPS) that provides these features, introduced in [19,20].

The important contribution of OOPS is the separation of recognition and mediation from application development. The separation of recognition leads to the ability to adapt mediation to situations which do not seem to be recognition-based, but where some problem in interaction causes the system to do something other than what the user intended (our ultimate definition of error). The separation of mediation allows us to develop complex mediation interactions independent of both the source of ambiguity and the application. It also allows us to defer mediation for arbitrary periods when appropriate. OOPS also provides hooks that facilitate application-specific mediation in situations that benefit from specific knowledge to do the right thing, such as dealing with errors of omission.

In addition to an architecture, OOPS includes a library of mediators that fill out the design space illustrated in Tables 1 and 2, described in the next section. In particular, we have built re-usable, generic choice and repetition mediators that can be easily modified along the dimensions shown.

The focus of the work presented here is on expanding the repertoire of mediation techniques into settings where standard techniques, such as the *n*-best list in Figure 1, face problems. The work described in this paper involves the design and implementation of re-usable mediation techniques that address some of the deficiencies or problems not handled well by the standard set of techniques found in the literature. We begin by describing the design space in more detail in the next section. After introducing the toolkit concepts necessary to understand our solutions to these problems in the following section, we will discuss each problem in detail. The first problem involves *adding alternatives* to choice mediators, as the correct answer (as defined by the user) may not appear in the list of choices. The second problem, *occlusion*, occurs because choice mediators may cover important information when they appear. The third problem we address is mediation for *target ambiguity*, which can arise when there are multiple possible targets of a user action (such as selecting part of a drawing). Finally, our fourth mediator illustrates one way to deal with *errors of omission*, where the user's input is completely missed by the recognizer.

DESIGN SPACE

Mediation is the process of selecting the correct interpretation of the user's input (as defined by the user). Because of this, mediation often involves the user in determining the correct interpretation of her input. Automatic mediation, which does not involve the user, is also fully supported by OOPS, although not a focus of this paper. Good *mediators* (components or interactors representing specific mediation strategies) minimize the effort required of the user to correct recognition errors or select interpretations. This section presents a survey of existing interfaces to recognition systems, including speech recognition, handwriting recognition, gesture recognition, word prediction, and others (expanded from [19]).

The survey led us to identify two basic categories of interactive mediation techniques. The first, and most common mediation strategy, is repetition. In this mediation strategy, the user repeats her input until the system correctly interprets it. In the second strategy, choice, the system displays several alternatives and the user selects the correct answer from among them.

Repetition

When the user specifies the correct interpretation of her input by explicitly repeating some or all of it, we refer to

I/O	System	Repair Modality	Undo	Repair Granularity
Handwriting / Words	MessagePad™[1], Microsoft Pen for Windows™	Soft keyboard, or individual letter writing	Automatic	Letters
Speech / Words, Phrases	ViaVoice™[3]	Speech, letter spelling, typing	Automatic	Letters or words
	Suhm speech dictation [30]	Voice, pen	User must select area to replace	Letters or words
Speech / Names (non GUI)	Chatter [22]	Speech, letter spelling, military spelling, with escape to choice	Automatic	Letters
Typing / Words	Word Prediction [2,10,25]	Letters (as user enters additional characters, new choices are generated)	Unnecessary (user has to explicitly accept a choice)	Letters
	POBox [21]			

Table 1: A representative set of systems (as defined by their input and output modalities) that vary along the dimensions of repetition mediators. All of these systems provide additionally unmediated repetition, in which the user deletes an entry and repeats it using the original system modality. In contrast, a system which does not provide mediated repetition is the PalmPilot™. I/O gives input/output of recognizer. Systems are representative examples. Military spelling uses "alpha" for 'a', "bravo" for 'b', *etc.*

this as repetition. For example, when a recognizer makes an error of omission (and thus generates no alternatives at all), this option is available to the user. Specific examples are given in Table 1. See [28] for a discussion of how such variations may affect input speeds. Below are the three dimensions of repetition:

Modality: The user often has the option of repeating her input in a different (perhaps less error-prone) modality. However, research in speech systems shows that users may choose the same modality for at least one repair before switching [11], despite the fact that the repair will have lower recognition accuracy [7].

Undo: In order to repeat her input, the user may first have to undo some or all of it. This is most often required of systems without explicit support for mediation (*e.g.* the PalmPilot™).

Repair granularity: Repair granularity may differ from input granularity. For example, the user may speak words or phrases, yet repair individual letters [30].

Choice

When the user selects the correct interpretation of her input from a set of choices presented by the system, we refer to this as choice mediation. The *n*-best list in Figure 1 is an example of this. Like repetition, choice mediators vary along a set of common dimensions. We illustrate the dimensions below by comparing two examples, the *n*-best list used in ViaVoice™ (Figure 1) and the Pegasus drawing beautification system (Figure 2). Pegasus recognizes user input as lines and supports rapid sketching of geometric designs [14]. Additional examples are given in Table 2.

Layout: The *n*-best list uses a menu layout. In contrast, Pegasus does layout "in place". Possible lines are simply displayed in the location they will eventually appear if selected (Figure 2(c&e)).

Instantiation time: The *n*-best list can be instantiated by a speech command, or can be always visible (even when no ambiguity is present). Pegasus shows the alternative lines as soon as they are generated.

Contextual information: Pegasus also shows contextual information about the lines by indicating the constraints that were used to generate them (Figure 2(c&e)). The *n*-best list, which is more generic, shows no additional information.

I/O	System	Layout	Instantiation	Context	Interaction	Feedback
Handwriting / Words	MessagePad™ [1]	Linear menu	Double click	Original ink	Click on choice	ASCII words
Speech / Words	ViaVoice™ [3]	Linear menu	Speech command / Continuous	None	Speech command	ASCII words
Speech/Commands (non GUI)	Brennan and Hulteen [4]	Spoken phrases	On completion	System state (audio icons)	Natural language	Pos.&neg. evid.-nat. lang.
Handwriting / Characters	Goldberg *et Al.* [9]	Below top choice	On completion	None	Click on choice	ASCII letters
Typing / Words (Word prediction)	Assistive Tech. [2, 10]	Bottom of screen (grid)	Continuously	None	Click on choice	ASCII words
	Netscape™ [25]	In place	Continuously	None	Returns to select, arrow for more	ASCII words
Gesture / Commands	Marking Menu [15]	Pie menu	On pause	None	Flick at choice	Commands, ASCII letters
Gesture / Lines	Beautification [14]	In place	On prediction / completion	Constraints	Click on choice	Lines
Context / Text	Remembrance Agent [27]	Bottom of screen, linear menu	Continuously	Certainty, result excerpts	Keystroke command	ASCII sentences
UI description / Interface spec.	UIDE [29]	Grid	On command	None	Click on choice	Thumbnails of results
Multimodal / Commands	Quickset [23]	Linear menu	On completion	Output from multiple recognizers	Click on choice	ASCII words
Email / Appointment	Lookout [12]	Pop up agent, speech, dialogue box	On completion	None	Click OK	ASCII words

Table 2: The layout, instantiation mode, context, selection, and representation used by commercial and research choice mediators. Note that feedback in the mediator may differ from the final output result of recognition. I/O gives input/output of recognizer. Systems are representative examples.

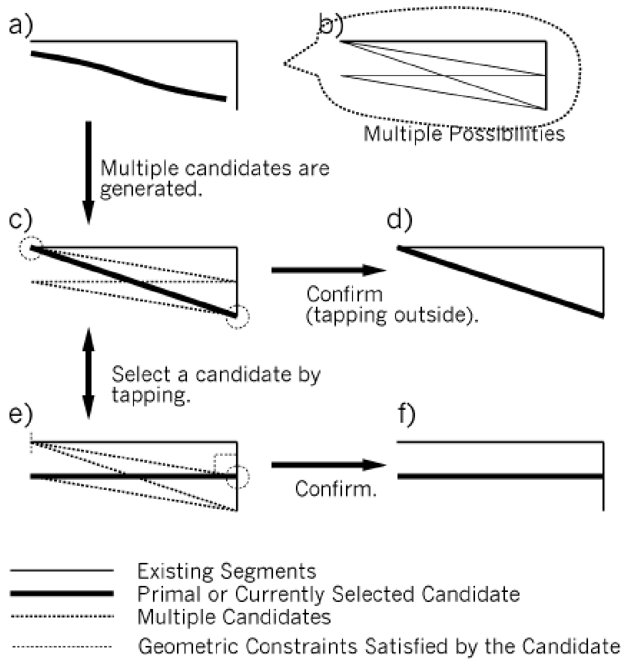

a)
b)
Multiple Possibilities
Multiple candidates are generated.
c)
d)
Confirm (tapping outside).
Select a candidate by tapping.
e)
f)
Confirm.
Existing Segments
Primal or Currently Selected Candidate
Multiple Candidates
Geometric Constraints Satisfied by the Candidate

Figure 2: A choice mediator in the Pegasus drawing beautification system (Figure 7 in [14]). The user can click on a line to select it (e). © ACM (reprinted with permission).

Interaction: In both examples, interaction is quite straightforward. In ViaVoice™, the user says "Pick [#]." In Pegasus, the user can select a choice by clicking on it (Figure 2(e)). Drawing a new line will automatically accept the currently selected choice in Pegasus (Figure 2(d&f)). The *n*-best list is only used to correct errors, the top choice is always sent to the current document as soon as recognition is complete.

Feedback: As stated earlier, feedback in Pegasus is in the form of lines on screen. Contrast this to the ASCII words used in the *n*-best list.

In summary

We have identified a rich design space of mediators which fall into two major classes of techniques. Each system we reference implemented their solutions from scratch, but as Tables 1 and 2 make clear, the same design decisions show up again and again. The space of mediation techniques is, therefore, amenable to toolkit-level support, and that is why we built OOPS. OOPS includes both an architecture and a library of mediators, including a generic repetition and a generic choice mediator that can be varied along the dimensions described above in a pluggable fashion [19,20]. We expand upon that work in this paper by identifying some significant gaps in the design space. We were able to use OOPS to create new mediators which address the problems responsible for those gaps.

TOOLKIT-LEVEL SUPPORT FOR AMBIGUITY

OOPS is an extension of the subArctic toolkit [6]. Here we will review the basic features of OOPS discussed in [20], and describe additional features that facilitated the development of the mediators described in this paper.

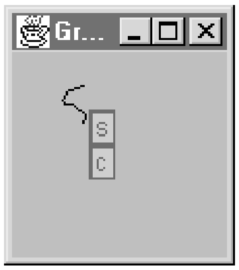
In the past, GUI toolkits have separated recognized input from mouse and keyboard input. Even when a recognizer generates the same data type as a device (such as text), the application writer has to take responsibility for informing interface widgets about information received from the recognizer. Both the Amulet [17] and the Artkit [13] toolkits go beyond this model for pen gesture recognition by allowing interactors to receive gesture results through the same API as mouse and keyboard events.

OOPS takes a step further by allowing recognizers to produce arbitrary input events that are dispatched through the same input handling system as any raw events produced by mouse or keyboard. Thus, they may be consumed by the same things that consume raw events including, possibly, other recognizers. Here we use the term "event" in the traditional GUI toolkit sense, to mean a single discrete piece of input (*e.g.* "mouse down" or "key press (a)").

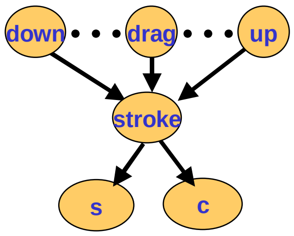
A recognizer produces events that are *interpretations* of other events or raw data (such as audio received from a microphone). This is a very broad definition of recognition. Essentially, a recognizer is a function that takes events or raw data as input and produces interpretations (also events) as output. For example, a recognizer might produce text from mouse events (which, as described above, is dispatched and might then be consumed by a standard text entry widget such as the one in Figure 4). It could start with text and produce more text. Or it could start with audio and produce mouse events (which might cause a button to depress). It might also produce a new event type such as a "command" or "interaction" event.

As mentioned in the previous section, a recognition error is defined by the user's intent and neither the recognizer nor OOPS necessarily knows what the correct interpretation is. It is through automatic or interactive mediation that this is determined. Until mediation is completed, OOPS stores information about all known possible interpretations. We refer to the input as ambiguous at this point. Information about ambiguity is kept in a hierarchical ambiguous event graph in OOPS (which can be seen as an extension of the command objects described in [24]). Raw input such as mouse down, drag, and up events make up the root nodes of that graph. Whenever input is interpreted, a node representing the new interpretation is added to the graph. For example, the graph shown in Figure 3(b) represents a series of mouse events that have been interpreted as a stroke, and then recognized as either a '**c**' or an '**s**'. The '**c**' and '**s**' are ambiguous (only one of them is correct). A graph node is considered ambiguous when it is one of multiple interpretations.

OOPS provides infrastructure for tracking ambiguity and for resolving ambiguity (mediation). By providing a consistent, recognizer-independent internal model of ambigu-



(a)



(b)

Figure 3: (a) A sketched letter and associated mediator. **(b)** Our internal representation of the events making up the sketched, recognized stroke.

ity, OOPS is able to provide re-usable support for mediation. For example, when the stroke in Figure 3(a) is interpreted as a '`c`' or an '`s`', OOPS automatically sends the event hierarchy to the mediation subsystem because part of it is ambiguous. The default choice mediator (Figure 3(a)) simply displays the leaf nodes of the hierarchy. When the user selects his intended input, it is accepted, and the other interpretation is rejected, resolving the ambiguity.

Most interpretations in OOPS are generated and dispatched during an initial input cycle before the start of mediation. When a new interpretation is created after mediation has begun, it is dispatched as well, and the event hierarchy being mediated is updated, along with any current mediators. Any events that have already been accepted or rejected remain that way.

OOPS supports both automatic mediation and a variety of interactive choice and repetition techniques. Mediation in OOPS may occur immediately or at any later time determined by the current mediator. Thus, a mediator may choose to wait for further input, or simply defer mediation until an appropriate time in the interaction.

When separation is too separate

As stated, we provide a recognizer-independent internal model of ambiguous input in OOPS which allows the separation of recognition, mediation, and application development. However, there are times when two or more of these pieces may need to communicate. For example, recognizers may wish to know which interpretations are accepted or rejected by mediators in order to facilitate learning. OOPS stores information about who created each event in order to inform those recognizers about which of their interpretations are accepted or rejected by the user.

In addition to creating events and receiving accept/reject messages, recognizers in OOPS may support guided re-recognition. Guided re-recognition allows a recognizer to receive more detailed information than a simple reject. This information may be domain specific, and includes an event that should be re-recognized. The intent is to allow a recognizer to make a better guess as to how to interpret the user's input. Recognizers supporting guided re-recognition must implement the **rerecognize(event, Object)** method, where *event* is an event that the recognizer interpreted at some time in the past and *Object* may contain additional domain-specific information.

For example, a choice mediator could have a "none of the above" option. If the user selects it, that mediator could ask the recognizer(s) that generated the current set of choices to **rerecognize()** each of their source events. If there is more than one source event for a given interpretation, the mediator may call the **resegment(Set, Object)** method instead. This tells the recognizer that a mediator has determined that the events in *Set* should be treated as one segment and re-interpreted.

Thus far, we have described the minimal architectural support required by all of our example mediators. In addition to this architecture, we provide a library of standard mediators in OOPS. Since OOPS allows separation of mediation from recognition and from the application, it is possible to swap between different mediation strategies without redesigning any recognizers or the application.

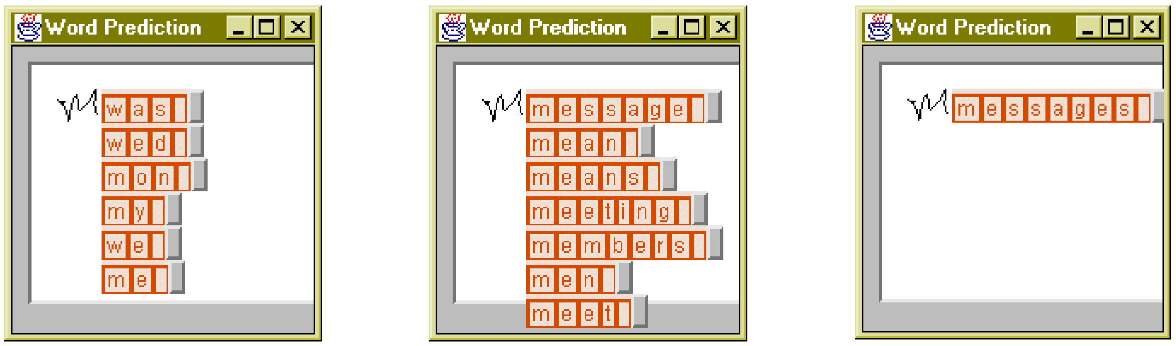
The next four sections consider four new mediation techniques. Each of these techniques is designed to illustrate a method for overcoming a problem with existing mediation techniques. In each section, after discussing the identified problem area (adding alternatives, occlusion, target ambiguity, and omission), and a new interaction technique that addresses it, specific toolkit mechanisms necessary to support these solutions will be considered.

ADDING ALTERNATIVES TO CHOICE MEDIATORS

One problem with choice mediators is that they only let the user select from a fixed set of choices. If none of those choices is right, the choice mediator is effectively useless. For example, as Figure 1 illustrates, if the user intended to say '`for`', the choice mediator cannot help her—she must escape to a repetition technique (spelling the word).

Our goal is to support a smooth transition from selection of an existing choice to specification of a new one. Our solution is to extend an *n*-best list to include some support for repetition. We allow the user to specify new interpretations as well as to select from existing ones using the same mediator.

For example, in the application shown in Figure 4, the user can sketch Graffiti™ letters, which are recognized as characters (by [18]). A word predictor then recognizes the characters as words, generating many more choices than can be displayed by the mediator. When the graffiti letters are ambiguous, the word-predictor returns words starting with each possible letter. Once mediation is completed, the text edit window updates to show the correct choice. Our goals in this mediator are:



(a) (b) (c)

Figure 4: A choice mediator which supports specification. The user is writing 'messages'. **(a)** The user sketches a letter which is interpreted as either a '**m**' or a '**w**' by a character recognizer. A word predictor then generates options for both letters. The user clicks on the '**e**' in '**me**.' **(b)** This causes the mediator to filter out all words that do not begin with '**me**.' The word '**message**' is now the top choice, but it needs to be pluralized. The user clicks on the space after '**message**' indicating that the mediator should generate a word beginning with '**message**' but one character longer. **(c)** The resulting word.

Provide choice at the word level: The user can select from among choices as he did in a standard choice mediator, by clicking on the gray button to the right of a word.

Allow users to control filtering: By clicking on a character, the user specifies a prefix. The mediator reflects this by displaying only words starting with the same prefix. Although word-predictors support dynamic filtering, in most cases, a prefix can only be specified by entering each letter in the prefix in turn. If the user filters repeatedly on the same prefix, the mediator will display a new set of words each time.

Allow users to specify length: by clicking on the space at the end of a word, the user causes the mediator to add a character to that word.

Allow users to specify individual characters: The user can right-click on a character to cycle through other possible characters. This can be used to generate words not returned by the word-predictor.

Allow users an escape: if the user sketches a new letter, only the current prefix will be accepted.

Suppose the user enters an (for which the graffiti recognizer returns '**m**' and '**w**'). The word predictor returns words starting with '**m**' and '**w**' (derived from a frequency analysis of a corpus of email messages), of which the mediator displays the top choices: **was**, **wed**, **mon**, … (Figure 4(a)). The user, who intended '**messages**', filters for words starting with '**me**' by clicking on the '**e**' in '**me**.' The resulting list (Figure 4(b)) includes '**message**', but not '**messages**.' The user indicates that a word one character longer than '**messages**' is needed by clicking on the space at the end of the word, and '**messages**' appears as the top choice (Figure 4(c)). The user selects this choice by clicking on the gray button to its right.

Was this mediator really useful? Without word prediction, the user would have had to sketch 8 characters. Given an 85% accuracy rate (typical for many recognizers), she would have to correct at least one letter (with a total of at least 9 strokes). Here, the user has sketched one letter and clicked 3 times. While this is only one data point, it should be noted that the mediator is built with appropriate defaults so that if the user simply ignores it and sketches the 8 characters, the result will be identical to a situation without word-prediction.

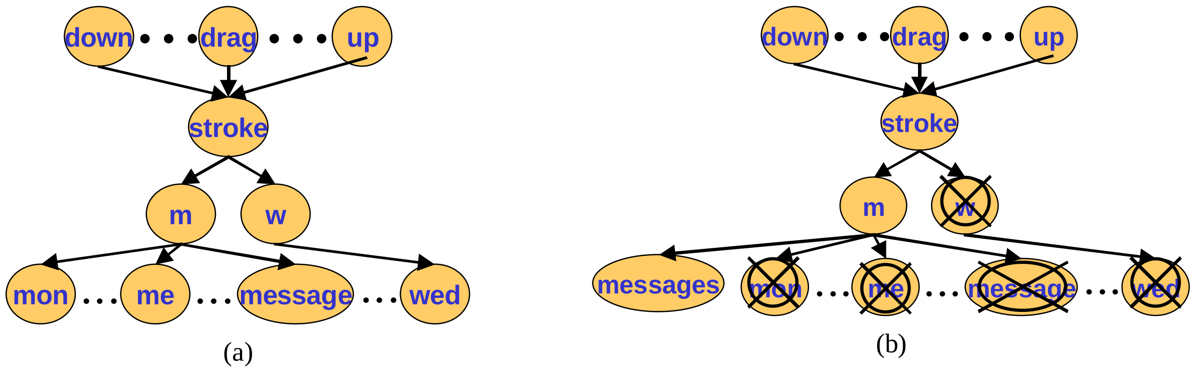


Figure 5: (a) The original event hierarchy in Figure 4 (a-c) and **(b)** the final hierarchy after mediation.

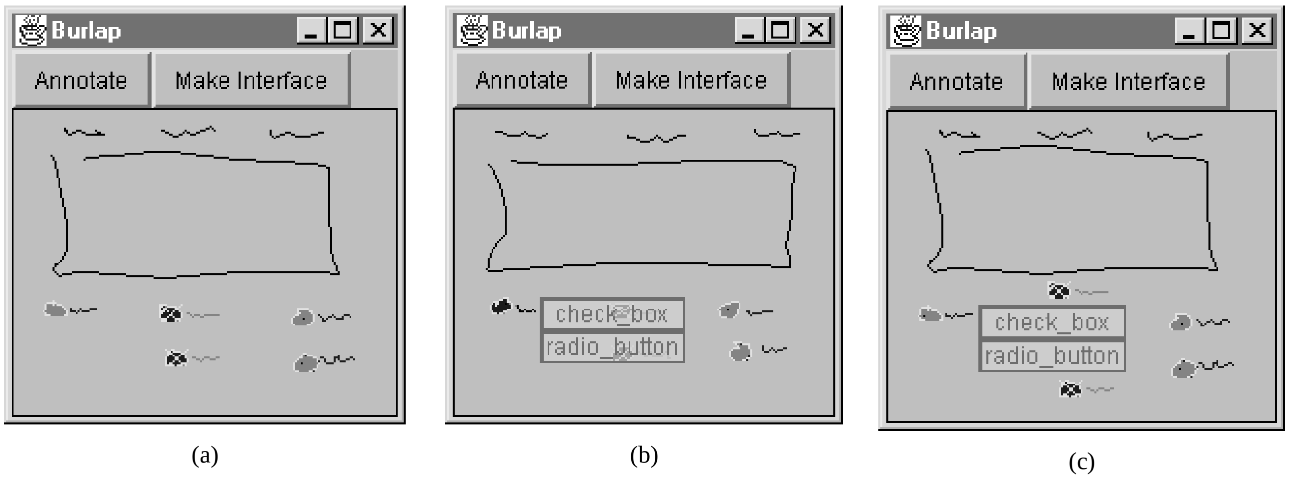


Figure 6: An example of fluid negotiation to position a mediator in the Burlap application. **(a)** The user needs to mediate whether the object on the bottom left edge is a checkbox or radiobutton. **(b)** This mediator (an *n*-best list) occludes some of the sketched interactors. **(c)** This mediator repositions all interactors that intersect with it so as to remove any occlusion.

Toolkit support for adding alternatives

This example dynamically generates new interpretations during mediation. In the example of Figure 4, a new interpretation ('`messages`') is created. All other interpretations are rejected. Figure 5 shows the original and changed hierarchy. The new event is accepted immediately since the user just specified that it was correct using the mediator. It is then dispatched, with the result that it is consumed by the text edit window. No further mediation is necessary since neither the new event nor the event hierarchy it is added to is ambiguous.

Reusability

This mediator can be used with any recognizer that returns text, including speech recognition or handwriting recognition, since the alternatives generated by these recognizers tend to have some similarities to the intended word. For example, some of the characters in a written word may be recognized correctly while others are not. The general idea, to add interactivity supporting repetition to a selection task, can be applied to other domains as well. For example, the automatic beautifier, Pegasus, uses a choice mediator to display multiple lines [14]. Repetition could be added to this by allowing the user to drag the end point of a line around. This would also allow Pegasus to display fewer alternatives in cases where too many are generated, in which case the line could snap to hidden interpretations.

OCCLUSION IN CHOICE MEDIATORS

Choice mediators generally display several possible interpretations on the screen for the user to select from. They are fairly large, and may obscure important information needed by the user to select the correct choice. Since they are also temporary, it doesn't make sense to leave screen space open just for them. An alternative is to dynamically make space for them.

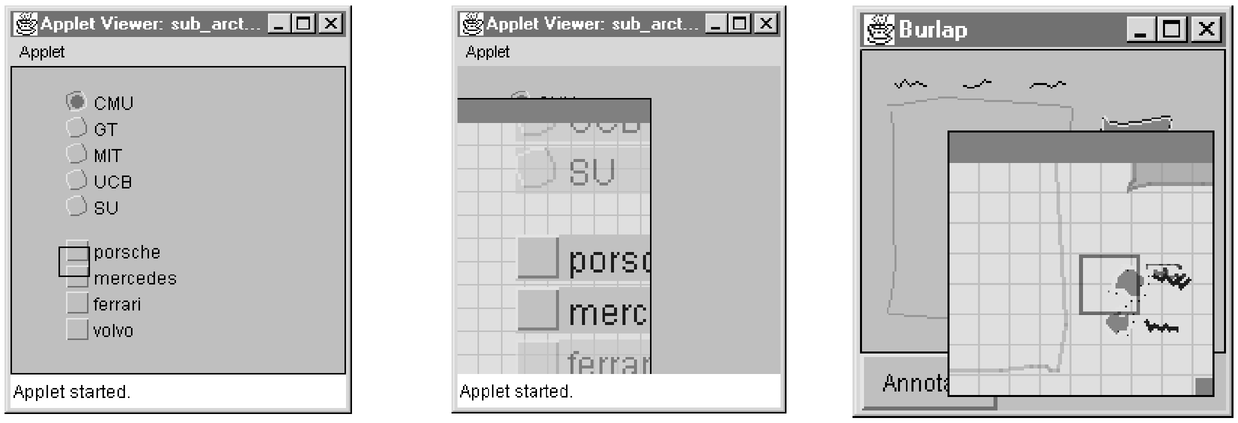
For example, consider Burlap, the application shown in Figure 6 [20]. Burlap is a drawing program for sketching user interface elements, based on SILK [16]. The user can sketch buttons, scrollbars, and so-on. These interactors are recognized and become interactive. However, recognition is error-prone. For example, checkboxes are easily confused with radiobuttons.

The *n*-best list in Figure 6(b) is obscuring two buttons. Is the leftmost sketch a checkbox or a radiobutton? This type of ambiguity is not mediated in Burlap until the user tries to interact with a button. So he may have drawn the hidden buttons some time ago. In order to be consistent, the user may need to see the buttons now in order to determine their status.

Our solution moves the sketches occluded by the mediator into a more visible location (Figure 6(c)). This approach is based on one of the interface techniques used in *fluid negotiation*, a concept that Chang *et al.* developed for handling temporary displays of information [5]. Some of the possible approaches they suggested include shrinking, fading, and call-outs. In [5], the temporary display negotiated the best approach with the underlying document. Because our mediator is used for input as well as output, and is the focus of the interaction, we have chosen a technique that only changes the underlying document (the sketched interface), not the size or shape of the mediator.

Toolkit support for dealing with occlusion

This is accomplished in a way that requires no changes to the underlying application. The only difference between the application shown in Figure 6(b) and (c) is which mediator is installed. The new mediator is based on a lens that uses the subArctic interactor tree to pick out all of the interactors that intersect its bounding box. It then uses the techniques described in [6] to modify the way they are drawn (without changing the interactors themselves). This



(a) (b) (c)

Figure 7: (a) The user is trying to click on a button. Which one? **(b)** A mediator magnifies the area in order to let the user specify this (The ambiguous choices are displayed in darker colors). **(c)** An example of the same mediator being used in Burlap.

mediator is modal, the user is required to resolve the ambiguity before continuing his interaction.

Reusability

This mediator can be used anywhere an *n*-best list might be used. For example, considering the mediators described in Table 1, this includes word-prediction, speech and handwriting recognition in GUI settings, and the Remembrance Agent [27]. More generally, similarly to the fluid negotiation work, the lens responsible for moving screen elements out from under the mediator could be combined with any mediator that is displayed on screen in a GUI. Of course, there are some situations where occlusion is appropriate, such as in the next example.

TARGET AMBIGUITY

We have encountered three major classes of ambiguity in our work. These are recognition ambiguity (which word did she write?); segmentation ambiguity (was that "sew age" or "sewage"?) and target ambiguity. In [20], we described these classes of ambiguity in more detail, and introduced a mediator of segmentation ambiguity. In almost all previous systems, mediation has only addressed recognition ambiguity. Here we demonstrate mediation of target ambiguity.

Target ambiguity arises when the target of the user's input is uncertain. For example, it is unclear if the circle around the word circle is intended to include "is" or not.

We are interested in using target ambiguity to model situations that, although they may seem ambiguous to the user, are commonly treated as straightforward by the computer. In particular, we are interested in situations where mouse motion becomes difficult. For example, people use the term “fat finger syndrome” to refer to situations in which the user's finger is larger than the button they want to press (very small cell phones, touch screens). In addition, misalignment on something like a digital white board can cause a mouse click to go to the wrong interactor. Also, certain disabilities may make it difficult to control a mouse, as can age. For example, research shows that older users have trouble selecting common GUI targets [31] as do people with disabilities such as cerebral palsy.

These problems can be addressed by treating the mouse as an area instead of a point [31]. However, the resulting area may overlap more than one interactor (an example of target ambiguity). We mediate this target ambiguity using a magnifier (Figure 7(b&c)). This magnifier only appears when there is a need due to ambiguity. For context, we include an area four times the size of the area mouse. The magnified area is interactive and users can click on interactors inside it just as they would on an unmagnified portion of the screen. As soon as the user completes his action, or the mouse leaves the magnifier, it goes away.

Toolkit support for target ambiguity

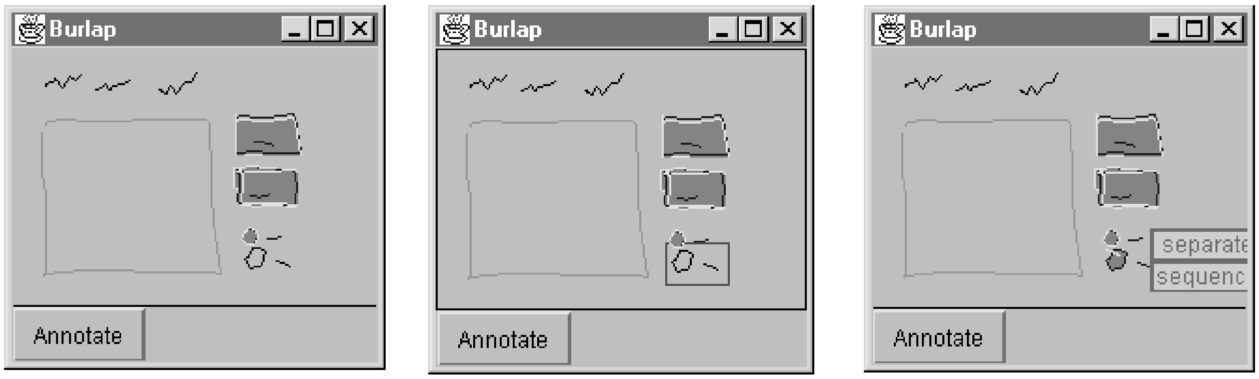
First, target ambiguity is generated by a recognizer that checks for multiple mouse targets. If only one target exists, the input is processed normally. If several targets exist, the results are passed to the mediator.

It is because of our extremely general support of recognition that this is possible. For example, when the extended mouse area (but not the mouse itself) intersects a single interactor, this recognizer creates a new mouse event over that interactor as an interpretation of the raw mouse event it gets as input. This interpretation is dispatched and consumed by the interactor, which does not even know that a recognizer was involved. As far as the interactor is concerned, the user clicked on it.

Our mediator makes use of a lens that magnifies the area under the input [6]. In addition, the lens is responsible for adjusting any positional input it gets based on the new size and position of the pixels it is magnifying.

Reusability

The magnification mediator works with any interface built in OOPS. This includes animated (moving) interactors;



(a) (b) (c)

Figure 8: (a) An interface sketched in Burlap. In the lower right is an unrecognized radiobutton. **(b)** The user has selected the radiobutton in order to indicate that re-recognition should occur. **(c)** Re-recognition is completed, and the result is further ambiguity. One of the possible interpretations generated was a radiobutton. If that is correct, should the new button be in sequence with the old one, or should it be separate?

Burlap; and any other interface which uses mouse clicks. Although it is currently limited to mouse input, in theory it could be generalized to any positional input.

ERRORS OF OMISSION

The last problem area addressed in this paper is errors of omission. An error of omission occurs when some or all of the user's input is not interpreted at all by the recognizer. For example, in Figure 8, the user has sketched a radio button in the lower right, but those strokes were not recognized. An error of omission has occurred.

The rules used for recognition in Burlap are taken from [16] and are based upon the shape and size of individual strokes drawn by the user and the graphical relationships between sets of strokes (such as distance, orientation, *etc*.). When the user draws something too big, too far apart, or so on, it is not recognized. In general, the causes of errors of omission are very recognizer-dependent. For example, an error of omission may occur in speech recognition because the user speaks too quietly.

One solution to this problem is repetition. The user can simply try the sketch again. However, research in both pen [8], and speech [7], recognition has found that re-recognition accuracy rates are no better than recognition rates

We address this with guided re-recognition. The user can initiate re-recognition by selecting the strokes that should have been recognized using the right mouse button. By doing this, the user is giving the system important new information. In Burlap's case, the new information is that the selected strokes should be interpreted as an interactor. We can use this information to eliminate options, such as interactors that have a different number of strokes.

Toolkit support for guided re-recognition

Essentially what the user is doing in this example is providing segmentation information to the recognizer. Although the unistroke gesture recognizer used in Burlap (a third party recognizer [18]) does not support guided re-recognition, the interactor recognizer does. We pass the selected strokes to the interactor recognizer using **resegment(Set, Object)** method described in the toolkit section of this paper. The recognizer generates a new set of interpretations based on those strokes. Because the recognizer now knows the number of strokes, it can quickly narrow the possible interactors and generate alternatives.

Since the recognizer generates new events during mediation, those events must be dispatched, potentially resulting in further interpretations. The new events are ambiguous and OOPS will mediate them (Figure 8(c)), and tell any currently visible mediators to update themselves to show the new interpretations. Any events that were already accepted or rejected in the hierarchy remain that way.

Reusability

The same mediator could be applied in other graphical settings with segmentation issues such as handwriting recognition. This mediator must be told which recognizer it should work with (in the case of Figure 8, the interactor recognizer). It will automatically cache any events generated by that recognizer. Alternatively, it may be given a filter that knows enough about the domain to cache the correct events. In either case, once the user specifies an area, any cached events inside that area are sent to appropriate recognizer to be re-recognized.

CONCLUSIONS AND FUTURE WORK

Recognition today is used in many applications and these applications make use of some common user-interface techniques, called mediators, for handling the recognition errors and ambiguity. However, there are problems with existing techniques. Some recognition errors, such as those caused by target ambiguity and errors of omissions, are harder to deal with. Also, there are limitations to how choice interfaces are commonly handled.

The work described in this paper addresses these problems. All of the mediators presented in this paper were enabled by the OOPS toolkit. They were built with the intent to be re-used in many situations.

We have shown that it is possible to build a variety of techniques that go beyond the current state of the art in correction strategies. Beyond our exploration of the previously known classes of mediation techniques, we have shown how principled handling of ambiguity at the input level allows for mediation in other important settings.

In the future, we wish to explore mediation strategies applicable to segmentation errors, non-GUI applications, and command recognition. All are difficult to mediate because they have no obvious representation (unlike, for example, the text generated in handwriting recognition).

We also plan to use the OOPS toolkit to support empirical work comparing the effectiveness of different mediation techniques. OOPS can allow us to build a framework for evaluating existing and new mediation technologies in a more controlled setting. Because we can build a variety of mediators and easily apply them to the same application, controlled studies to compare the effectiveness of the mediation strategies is now made much simpler.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants IRI-9703384, EIA-9806822, IRI-9500942 and IIS-9800597. Many thanks to Chris Long for the use of his recognizer [18], and to the reviewers.

REFERENCES

1. Apple Computer, Inc. (1997) *MessagePad 2000 User's Manual*.
2. Alm, N. *et al.* (1992) Prediction and conversational momentum in an augmentative communication system. *Communications of the ACM,* **35**(5), pp. 46–57.
3. Baumgarten *et al*. (2000) *IBM® ViaVoice™ QuickTutorial®*. South-Western Educational Publishing.
4. Brennan, S. and E. A. Hulteen. (1995) Interaction and feedback in a spoken language system: A theoretical framework. *Knowledge-Based Systems*, **8**(2–3):143–151.
5. Chang, B. *et al.* (1998) A negotiation architecture for fluid documents. In *Proc. of UIST'98.* pp.123–132.
6. Edwards, K. *et al.* (1997) Systematic output modification in a 2D UI toolkit. In *Proc. of UIST'97*. pp. 151–158.
7. Frankish, C. *et al.* (1992) Decline in accuracy of automatic speech recognition as function of time on task: Fatigue of voice drift? *International Journal of Man-Machine Studies* **36**(6):797–816.
8. Frankish, C. *et al*. (1995) Recognition Accuracy and User Acceptance of Pen Interfaces. In *Proc. of CHI'95.* pp. 503-510.
9. Goldberg, D. and A. Goodisman (1991) Stylus user interfaces for manipulating text. In *Proc. of UIST'91,* pp.127–135.
10. Greenberg, S. *et al.* (1995) Predictive interfaces: What will they think of next? In *Extra-ordinary human-computer interaction: Interfaces for users with disabilities,* A.D.N. Edwards, editor, pp. 103–139.
11. Halverson, C. *et al.* (1999) The beauty of errors: Patterns of error correction in desktop speech systems. In *Proc. of INTERACT'99.*
12. Horvitz, E. (1999) Principles of mixed-initiative user interfaces. In *Proc. of CHI'99*, pp. 159–166.
13. Henry, T. *et al.* (1990) Integrating Gesture and Snapping into a User Interface Toolkit Interaction Techniques. In *Proc. of UIST'90*, pp.112–122.
14. Igarashi, T. *et al.* (1997) Interactive beautification: A technique for rapid geometric design. In *Proc. of UIST'97,* pp. 105–114.
15. Kurtenbach, G. *et al.* (1994) User learning and performance with marking menus. In *Proc. of CHI'94*, pp. 258–264.
16. Landay, J. A. *et al.* (1995) Interactive sketching for the early stages of user interface design. In *Proc. of CHI'95.* pp.43–50.
17. Landay, J.A. and B.A. Myers. (1993) Extending an existing user interface toolkit to support gesture recognition. In *Proc. INTERCHI'93*. pp. 91-92.
18. Long, A.C. *et al.* (1999) Implications for a gesture design tool. In *Proc. of CHI'99.* pp. 40–47.
19. Mankoff, J. *et al*. (2000) Techniques for handling ambiguity in recognition-based input. *Computes and Graphics,* special issue on calligraphic interfaces. Elsevier. To appear.
20. Mankoff, J. *et al.* (2000) Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proc. of CHI'2000,* pp. 368–375.
21. Masui, T. An efficient text input method for pen-based computers. In *Proc. of CHI''98,* pp.328–335
22. Marx, M. *et al*. (1994) Putting people first: Specifying proper names in speech interfaces. In *Proc. of UIST'94*. pp. 29-37.
23. McGee, D. R. *et al*. (1998) Confirmation in multimodal systems. In *Proc. of COLING-ACL '98*, Montreal, Canada.
24. Myers, B.A. and D.S. Kosbie. (1996) Reusable hierarchical command objects. In *Proc. of CHI'96,* pp. 260–267.
25. Netscape Communications, Corp. Netscape Navigator. http://www.netscape.com.
26. Nigay, l. and Coutaz, J. A Generic platform addressing the multimodal challenge. In *Proc. of CHI'95*. pp.98–105.
27. Rhodes, B.J. and T. Starner. (1996) Remembrance Agent. In *Proc. of PAAM'96,* pp. 487–495.
28. Rudnicky, A.I. and A.G. Hauptmann (1991). Models for evaluating interaction protocols in speech recognition. In *Proc. of CHI'91*, pp. 285-291.
29. Sukaviriya, P. *et al*. (1993) A second-generation user interface design environment: The model and the runtime architecture. In *Proc. of INTERCHI'93*, pp. 375–382.
30. Suhm, B. *et al*. (1999) Model-based and empirical evaluation of multimodal interactive error correction. In *Proc. of CHI'99,* May, 1999, pp. 584-591.
31. Worden, A. *et al*. (1997) Making Computers Easier for Older Adults to Use: Area Cursors and Sticky Icons. In *Proc. of CHI'97*, pp. 266–271.

SATIN: A Toolkit for Informal Ink-based Applications

Jason I. Hong and James A. Landay
Group for User Interface Research, Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776 USA
+1 510 643 7354
{jasonh, landay}@cs.berkeley.edu

ABSTRACT
Software support for making effective pen-based applications is currently rudimentary. To facilitate the creation of such applications, we have developed SATIN, a Java-based toolkit designed to support the creation of applications that leverage the informal nature of pens. This support includes a scenegraph for manipulating and rendering objects; support for zooming and rotating objects, switching between multiple views of an object, integration of pen input with interpreters, libraries for manipulating ink strokes, widgets optimized for pens, and compatibility with Java's Swing toolkit. SATIN includes a generalized architecture for handling pen input, consisting of recognizers, interpreters, and multi-interpreters. In this paper, we describe the functionality and architecture of SATIN, using two applications built with SATIN as examples.

Keywords
toolkits, pen, ink, informal, sketching, gesture, recognition, interpreter, recognizer, SATIN

INTRODUCTION

Sketching and writing are natural activities in many settings. Using pen and paper, a person can quickly write down ideas, as well as draw rough pictures and diagrams, deferring details until later. The informal nature of pens allows people to focus on their task without having to worry about precision.

However, although more and more computing devices are coming equipped with pens, there are few useful pen-based applications out there that take advantage of the fact that pens are good for sketching[1]. Most applications use pens only for selecting, tapping, and dragging. These applications simply treat the pen as another pointing device, ignoring its unique affordances.

Furthermore, the few compelling applications that do exist are built from scratch, despite the fact that many of them share the same kinds of functionality. This is because of the rudimentary software support for creating pen-based applications. Despite the fact that many new and useful pen-based interaction techniques have been developed, such as gesturing[1] and pie menus [5], these techniques have not yet been widely adopted because they are difficult and time-consuming to implement.

With respect to input and output for pens, we are at a stage similar to that of windowing toolkits in the early 1980s. Many example applications and many novel techniques exist, but there are no cohesive frameworks to support the creation of effective pen-based applications. As a first step towards such a framework, we have developed SATIN[2], a toolkit for supporting the creation of informal ink-based applications [15]. From a high-level perspective, there were three research goals for SATIN:

- Design a generalized software architecture for informal pen-based applications, focusing on how to handle sketching and gesturing in a reusable manner
- Develop an extensible toolkit that simplifies the creation of such informal pen-based apps
- Distribute this toolkit for general use by researchers

As a first step, we surveyed existing pen-based applications (both commercial and research) in order to determine what shared functionality would be most useful. Afterwards, we implemented the first iteration of the toolkit in Java, and built our first significant application with it, DENIM [26] (see Fig. 1). From the lessons learned, we developed the second iteration of SATIN, and built another application, SketchySPICE.

In this paper, we first outline functionality common in existing pen-based applications, and take a look at current software support for pen-based interfaces. We continue by describing the high-level and then detailed design of the SATIN toolkit. Specifically, we focus on a generalized architecture for handling pen input, consisting of three components: recognizers, interpreters, and multi-interpreters. We describe how pen input is handled in terms of the two applications, DENIM and SketchySPICE. We conclude with an evaluation of the toolkit, as well as our plans for future work and a discussion of lessons learned.

[1] By *sketching*, we mean the process of drawing roughly and quickly. We use the term *ink* for the strokes that appear. By *gesturing*, we mean a pen-drawn stroke that issues a command

[2] The SATIN project page and software download is at: `http://guir.berkeley.edu/projects/satin`

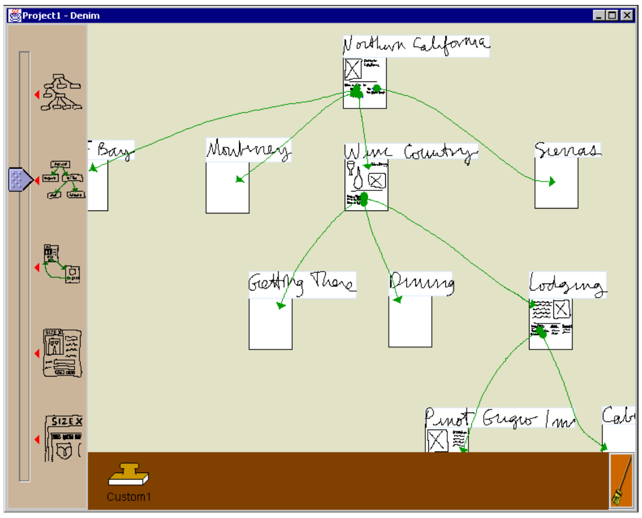


Figure 1 – A screenshot of DENIM, a sketch-based web site design tool created on top of SATIN

PEN APPLICATION SPACE

Recently, there have been many applications developed that use sketching and gesturing. We performed a survey of these applications, looking specifically for examples of informal ink-based interaction, ones that step away from rigid structure and precise computation, instead supporting ambiguity, creativity, and communication [15]. Many pen-based research systems have headed in the direction of informal interfaces in recent years, either by not processing the ink [11, 41, 43] or by processing the ink internally while displaying the unprocessed ink [14, 24, 32, 40].

The applications we examined include design tools [9, 12, 14, 20-22, 24, 43, 47]; whiteboard applications [1, 32, 33, 37]; annotation tools [41, 44-46]; note-taking applications [10, 11, 42]; and applications demonstrating new interaction techniques [19, 28, 40]. These applications share much functionality with each other, including:

- Pen input as ink
- Pen input as gestures
- Pen input for selecting and moving
- Interpreters that act on ink input
- Manipulation of other kinds of objects besides ink
- Grouping of objects
- Layering of objects
- Time indexing of ink input
- Transformation of ink to other cleaned-up objects
- Immediate and deferred processing of ink

Later, in the process of developing DENIM, our first application, we discovered we needed techniques for managing information, and turned to using zooming and semantic zooming, as demonstrated in Pad++ [3] and Jazz [4]. We decided that this functionality was useful enough to developers that it should be included in the toolkit.

EXISTING PEN FRAMEWORKS

In this section, we outline existing frameworks for developing pen-based applications, and describe where SATIN builds on their ideas.

Commercial Software Support for Pens

PalmOS [8] offers some very simple pen input processing. The default behavior is to process strokes and taps in the silk screen area as key events, with all other strokes passed on to the application for processing. PalmOS also provides some APIs for getting individual stroke points, enabling and disabling the Graffiti shorthand recognizer, and for getting the last known location of the pen.

Microsoft Windows for Pen Computing [29] provides minimal support for pens. Text entry areas were replaced either by handwriting edit controls (`hedit`) or by boxed edit controls (`bedit`), in which individual characters can be written. Simple gesture recognition was also supported. These extensions give the developer very little support for building informal ink-based applications.

In Windows CE [30], pen input is treated as a subset of mouse input. Applications can receive messages when the pen is moved, goes down, comes up, and is double-tapped. Windows CE also provides simple handwriting recognition.

NewtonOS [2] uses sheets of paper as its input metaphor. Users can write on these sheets without having to explicitly save. Furthermore, users can specify several ink modes in which strokes are processed as text, as shapes, or left unprocessed as raw ink. Recognition errors can be corrected by choosing from an n-best list. Gestures are also integrated into the system. Drawing a zig-zag shape over a word or shape, known as scrubbing, deletes that object. Holding down the pen for a second activates select mode. After select is enabled, the user can drag the pen and either highlight or circle the objects to select. Lastly, NewtonOS provides an extensive widget set for pens, designed to minimize the amount of end-user writing necessary.

Perhaps the most sophisticated commercial support for pens was in GO Corporation's PenPoint [6]. PenPoint is an operating system built from the ground up to support pens. Besides providing many of the services described above, such as gestures and pen widgets, PenPoint also has such features as live embedding of documents within documents, and extensive integration of gesture recognition and handwriting recognition.

There are two main differences between SATIN and the systems described above. First, all of the systems listed above are designed to build formal user interfaces, and are thus focused on handwriting recognition and form entry tasks. In contrast, SATIN is targeted towards the development of informal ink-based applications. The second difference is extensibility. Aside from handwriting recognition, the systems listed above provide minimal support for manipulating and processing ink. In contrast, one of our primary goals with SATIN was to give developers flexibility in how ink is processed and to make it simple to do so. For example, new gestures cannot be added in the systems described above.

Research Software Support for Pens

Simple ink and gesture support is provided in Artkit [16]. Artkit uses the notion of *sensitive regions*, invisible rectangles that can be placed on top of screen objects. The sensitive region intercepts stroke input, and processes the input in a recognition object, which possibly forwards a higher-level event to the screen object underneath.

Mankoff et al., extended the subArctic toolkit [17] to support inking, gesturing, and recognition, specifically for exploring techniques in resolving ambiguity [28].

Garnet [23, 34] and Amulet [36] also have support for gestures. A gesture interactor was added to these toolkits to support recognizing pen gestures using Rubine's algorithm [39]. The recognizer simply calls the registered callback procedure with the result as a parameter. No other pen and ink-based support is provided.

Flatland [18, 37] is a lightweight electronic whiteboard system that has much in common with SATIN. Flatland uses the notion of *segments* to divide up screen space, and uses *strokes* both as input and as output. Furthermore, *behaviors* can be dynamically plugged into segments, changing how stroke input is processed and displayed. This architecture is very similar to SATIN.

One clear difference between Flatland and SATIN is Flatland combines mechanism and policy in several cases, mixing *how* something is done with *when* it is done. For example, in Flatland, all strokes belong to a segment, and new segments are automatically created if a stroke is not drawn in an existing segment, whether or not an application designer wants a new segment. Our goal with SATIN was to focus on fine-grained mechanisms that can be used for a range of ink-based applications. Another difference is that Flatland only allows one application behavior to be active in a segment at any time. We introduce the notion of multi-interpreters to manage multiple interpreters.

Kramer's work in translucent patches and dynamic interpretations [21, 22] significantly influenced the design and implementation of SATIN. We use Kramer's notions of patches and dynamic interpretation, but again, our focus is at the toolkit level.

The chief characteristics that differentiate SATIN from all of the work above are flexibility and fine granularity. We are focused on developing an extensible toolkit. We provide a set of mechanisms for manipulating, handling, and interpreting strokes, as well as a library of simple manipulations on strokes, with which developers can build a variety of informal pen-based applications.

HIGH LEVEL DESIGN OF SATIN

SATIN is intended to support the development of 2D pen-based applications. We chose to support 2D instead of 3D since most of the applications surveyed utilize two dimensions only. The current implementation of SATIN does not support multiple users, as that introduces another level of complexity beyond the scope of this project.

SATIN is built in Java, using JDK1.3[3]. SATIN uses Java2D for rendering, and makes extensive use of the Java core classes as well as the Swing windowing toolkit [31].

Fig. 2 shows how a pen-based application would be built using SATIN, Swing, and Java. Roughly speaking, SATIN can be partitioned into twelve interrelated concepts (See Table 1). Each of these concepts is briefly summarized in the next section. Some of these concepts are very loosely coupled to one another, and can be used independently of the rest of the toolkit. In other words, a developer can use some portions of the SATIN toolkit without a complete buy-in of the entire system.

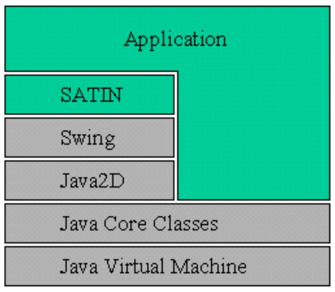


Figure 2 – This diagram shows the relationship between Java, Swing, SATIN, and pen-based applications.

Concept	Can use outside SATIN?	For pens only?
Scenegraph	No	No
Rendering	No	No
Views	No	No
Transitions	No	No
Strokes	Some portions	Yes
Events	No	Yes
Recognizers	Some portions	Yes
Interpreters	No	Yes
Clipboard	No	No
Notifications	Yes	No
Commands	Yes	No
Widgets	Yes	Yes

Table 1 – The twelve major components in SATIN. Some portions of SATIN have been designed to be independent of the rest of the system and can be used outside of SATIN.

Design Overview

We call objects that can be displayed and manipulated *graphical objects*. Like most 3D modeling systems (such as Java3D and OpenGL) we use the notion of a *scenegraph*, a tree-like data structure that holds graphical objects and groups of graphical objects. The simplest graphical object that the user can create is a *stroke*, which is automatically created in SATIN by the path drawn by a pen or mouse. Another primitive graphical object is a *patch*, an arbitrarily shaped region of space that can contain other graphical

[3] We began SATIN in JDK1.2, and transitioned to each early access version of the JDK as they were released.

objects. Patches *interpret* strokes either as gestures or as ink. Our notion of patches is derived from the work by Kramer [21, 22]. SATIN also provides a *sheet*, which is a Java Swing component as well as a graphical object. A Sheet serves as the root of a scenegraph, and is essentially a drawing canvas that can contain SATIN objects.

Graphical objects have x-, y-, and layer-coordinates. The x-axis and y-axis coordinates are Cartesian coordinates. The layer-coordinate is used to denote the relative position of one graphical object to another along the z-axis. That is, SATIN simply keeps track of which objects are on top of others, but does not store exact z-axis coordinates.

Graphical objects also have *styles*. Styles take many of the graphics concepts in Java, such as line style, color, and font, and translucency, abstracting them out into a single object. Styles are automatically applied by the rendering subsystem when rendering.

When *rendering*, SATIN uses the same damage-redraw cycle that is standard in windowing systems. The system never repaints a region unless it is marked damaged. If an area is damaged, then only the graphical objects in the damaged area are traversed. For common operations, such as translation and rotation, graphical objects automatically damage the region they are in. For application-specific operations, however, the developer may need to explicitly call the damage method.

SATIN also automatically changes the rendering quality depending on the current context. For example, when the user is drawing strokes, the damaged areas are rendered in low quality in order to speed up performance. However, when the stroke is completed, SATIN reverts to the highest quality rendering level.

Graphical objects have one or more *view* objects, which dictate how a graphical object is drawn. If a graphical object has more than one view, then it must also have a *MultiView*, an object that specifies the policy of which view objects are rendered and when. An example multi-view we have included is a Semantic Zoom Multi View, which uses the current zoom scale to choose the view to be displayed, as in Pad++ [3] and Jazz [4].

SATIN provides support for simple *transitions* on graphical objects, such as zooming and rotation. Given a graphical object and a transform, the system can automatically generate and render the intermediate steps, providing a smooth animation. The default transition type is Slow-In / Slow-Out [7, 25], a transition that spends the majority of time in the beginning and in the end of the animation.

There are also several classes for manipulating strokes. The *stroke assembler* aggregates user input into *strokes* and dispatches them as *events* to graphical objects. Each graphical object knows how to handle stroke events, and can choose how the stroke events are handled. This process is described in more detail in the Detailed Design section. There are also utility classes for manipulating strokes, such as splitting strokes, merging strokes, turning strokes into straight lines, and for simplifying strokes.

We use the term *recognizers* to mean subsystems used to classify ambiguous input, such as ink strokes. In SATIN, we have defined recognizers as objects that take some kind of ambiguous input and return a well-defined n-best list of classifications and probabilities ordered by probability. This definition allows us to plug in other stroke recognizers into the system. Examples of stroke recognizers include Rubine's recognizer [38, 39] and neural net recognizers. Currently, SATIN only contains the gdt [27] implementation of Rubine's recognizer. Recognizers may or may not retain state across classifications. However, recognizers do not take any kind of action based on the act of classification. Instead, this is left to interpreters.

Interpreters take action based on user-generated strokes. For example, one interpreter could take a stroke and transform it into a straight line. A different interpreter could issue a command if the stroke resembled a gesture in the system. Interpreters can use *recognizers* to classify strokes, but are not required to do so.

We distinguish between *gesture interpreters* and *ink interpreters*. A gesture interpreter tries to process a stroke as a command (e.g., cut), while an ink interpreter processes a stroke and displays the result as ink (e.g., straightens it out). We also make the distinction between *progressive-stroke interpreters* and *single-stroke interpreters*. A progressive-stroke interpreter tries to perform actions as a stroke is being drawn, while a single-stroke interpreter only takes action after a stroke is completed. SATIN currently does not support multi-stroke interpreters.

A graphical object can have one or more gesture interpreters, as well as one or more ink interpreters. Like views, a *MultiInterpreter* specifies the policy for which interpreters are used when more than one is present. Multi-interpreters are a new concept introduced in SATIN, and are discussed in the Detailed Design section.

The *clipboard* acts the same as in modern GUIs, supporting cut, copy, and paste for graphical objects.

Notifications are messages generated and sent internally within the system in order to maintain consistency. These messages are often used to maintain constraints between graphical objects or to notify objects that a graphical object has been deleted.

Commands are a common design pattern used for supporting macros, as well as undo and redo [13, 35]. Commands reify operations by encapsulating a transaction into an object that knows how to do, undo, and redo itself. SATIN's command subsystem extends the one provided in Java Swing (`javax.swing.undo`), by adding in the notion of executing a command (instead of simply undoing an operation). The command subsystem also has a notion of time, tracking when commands were executed, as well as allowing classes of commands to be enabled and disabled.

Application developers are not required to use the command subsystem in order to use SATIN. The Command subsystem can also be used outside of SATIN.

SATIN also provides some *widgets* optimized for pens. Currently, the only new widget we provide is a pie menu [5] that can be used as a normal Java Swing widget. The pie menu implements `javax.swing.MenuElement`, Swing's menu interface, and in many cases can be used in lieu of normal pop up menus with few changes to the code.

We also provide a Pen Pluggable Look and Feel (PenPLAF). The PenPLAF uses Java Swing's pluggable look and feel [31] to modify the standard file opener and slider widgets to make them easier to use for pens. The file opener was modified to accept single mouse clicks to open folders (instead of double clicks). The slider was modified to have a larger elevator, as well as the ability to have the slider value changed by tapping anywhere on the slider. The pie menu and the PenPLAF are not tied to SATIN, and can be used in Java applications outside of the toolkit.

Bridging the Gap between Java Swing and SATIN

We also provide some classes to help bridge the gap between SATIN and Java Swing (See Fig. 3). Currently, SATIN support for Swing consists of two classes. The first, `GObJComponent`[4], wraps up Swing widgets in a SATIN graphical object. Thus, Swing widgets can be displayed in SATIN, though full interaction (e.g., keyboard input), has not yet been completed. The second, `GObImage`, allows Java `Image` objects to be displayed in SATIN. This enables SATIN to be able to display any image file format that Java understands.

Conversely, SATIN can be used in Swing applications. As stated before, the Sheet is both the root of a scenegraph in SATIN and is a fully compatible Swing widget. A `JSatinComponent` is a Swing widget that wraps around a SATIN graphical object, letting SATIN graphical objects be displayed in Swing applications. Lastly, `SatinImageLib` provides some utilities for turning SATIN graphical objects into Java `Image` objects. This enables SATIN to be able to write out to any image file format that Java understands.

DETAILED DESIGN OF SATIN INK HANDLING

In this section, we describe strokes, recognizers, and interpreters in more detail, as well as how they interact with each other at runtime.

Strokes

In SATIN, strokes are simply a list of (`x`, `y`, `t`) tuples, where `x` is the x-coordinate, `y` is the y-coordinate, and `t` is the time the point was generated (since the Unix epoch).

SATIN also provides some utilities and interpreters for manipulating strokes, including splitting a stroke into

[4] `JComponent` is the parent class of all Swing widgets.

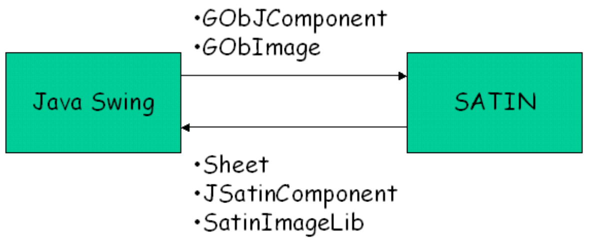


Figure 3 – Classes bridging the gap between SATIN and Java Swing. Swing widgets can be displayed in SATIN, and SATIN graphical objects can be embedded in Swing applications.

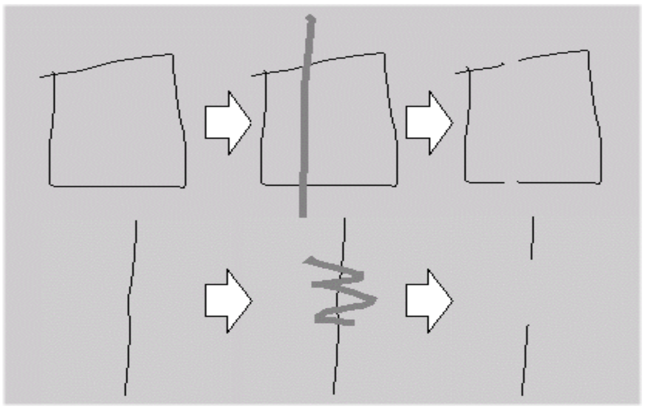


Figure 4 – Two example policies of splitting strokes. The thicker line is a gesture created by pressing the right button.

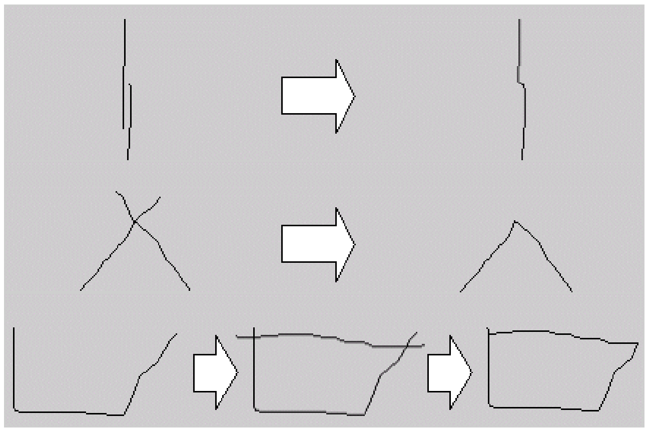


Figure 5 – At the top, two separate strokes near each other are combined into a single stroke. In the middle, two separate strokes that intersect near their endpoints are merged into a single stroke. At the bottom, two separate strokes that intersect near both of their endpoints are merged into a closed shape.

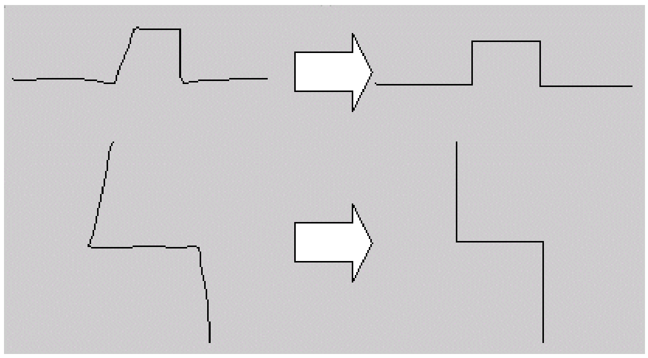


Figure 6 – Two examples of straightening strokes.

smaller substrokes, merging strokes together, straightening strokes into straight lines, and simplifying strokes.

Strokes can be split by specifying a rectangle in which all substrokes will be removed. Fig. 4 shows a sample interpreter that removes substrokes that lie in the bounding box of the gesture stroke.

Fig. 5 shows some examples of merging strokes. To see if two strokes can be merged, the algorithm first checks if the two strokes are near each other. If they are, then the algorithm checks if either extremity of one stroke is near an extremity of the other. If a successful match is made, then the two extremities are joined together in a new stroke, with short trailing ends discarded.

SATIN straightens strokes by changing strokes to lines that go up, down, left, or right (See Fig. 6). To straighten a stroke, we first examine each pair of adjacent points and classify each pair as going up, down, left, or right. For each subsequence of points that is going the same direction, we create a line that goes through the average value of that subsequence. After this is done, all of the lines created are joined together and returned as a new stroke.

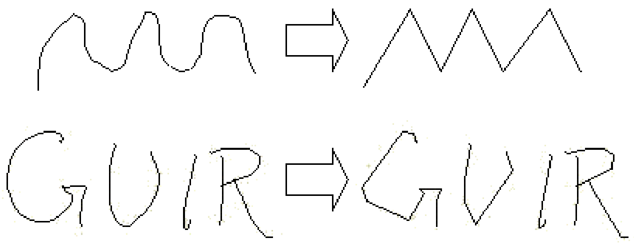


Figure 7 – Two examples of stroke simplification. The algorithm generates a stroke similar to the original stroke, but has fewer points and can thus be rendered faster.

SATIN also provides utilities for simplifying strokes (See Fig. 7). This technique is automatically used to help speed up animated transitions. The following approach is used to simplify a stroke:

- For each point, calculate the absolute angle relative to the stroke's top-left corner using `atan2()`
- Calculate the angle delta between each adjacent pair of points
- Add the starting and ending point of the original stroke to the simplified stroke
- Go through the deltas and add each local minima to the simplified stroke

Once a stroke is simplified, it is cached in the system. On a sample set of fifty strokes, the number of points reduced ranged from 20% to 50%, averaging a 32% reduction. Using a battery of performance regression tests using 100 to 1000 strokes, the performance speedup[5] for animating the simplified strokes ranged from 1.02 to 1.34, with an average speedup 1.11. Speedup improves somewhat linearly as the number of strokes is increased, as expected.

Recognizers

In SATIN, a recognizer is a subsystem that classifies ambiguous input, which in our case are strokes. SATIN defines a standard interface for two types of recognizers: *progressive stroke* and *single stroke* recognizers. These definitions are not mutually exclusive, so a recognizer could be both a progressive and a single stroke recognizer. SATIN also defines a `Classification` object, which recognizers are defined to return when passed a stroke to classify. The classification is simply an n-best list of beliefs, ordered by probability. This definition for recognizers means that new recognizers can be plugged into the system simply by implementing the defined interface.

Interpreters

The class diagram in Fig. 8 illustrates the relationship between the classes used for interpretation, and shows some of the interpreters built in SATIN.

Besides processing strokes, interpreters are also stroke event filters, meaning they can specify what kinds of strokes they will accept. The simplest filter accepts or rejects strokes depending on which pen button was held when creating the stroke. Another kind of filter rejects strokes that are too long. In addition to filtering, individual interpreters can also be disabled, meaning that they will not process any strokes at all.

Some of the interpreters, on the right side of Fig. 8, have already been discussed (see above), or will be discussed with DENIM and SketchySPICE (next section). The more interesting part is the left portion of Fig. 8, which shows the multi-interpreters. Multi-interpreters are collections of interpreters combined with a policy that controls which interpreters are used and when they are used.

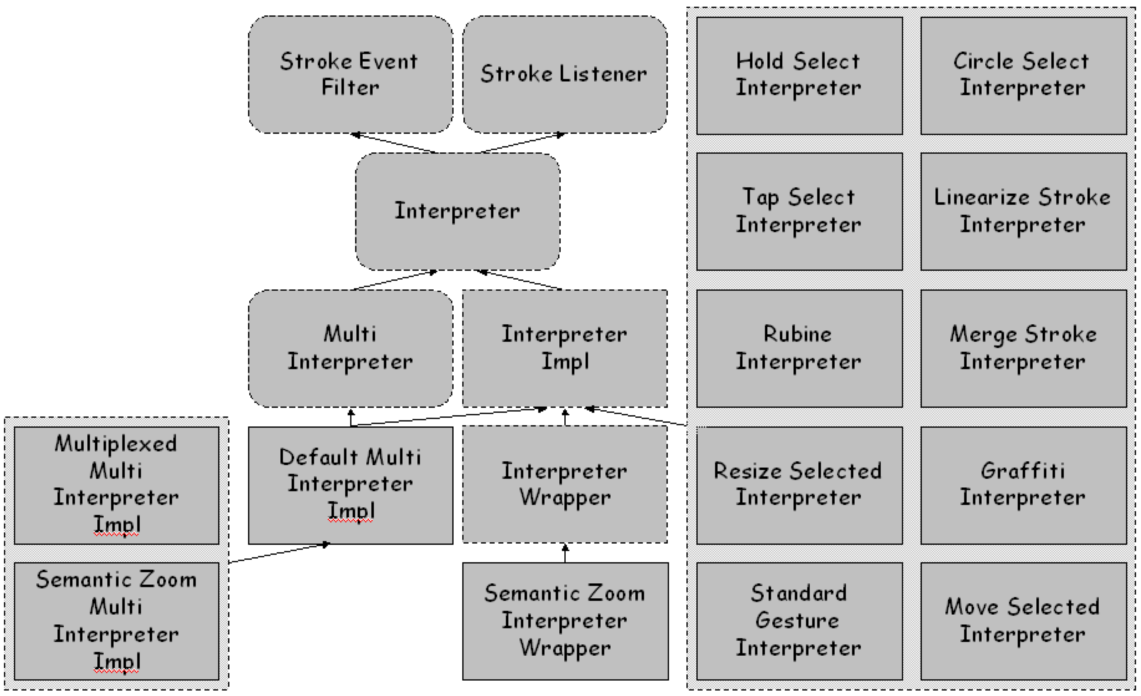
The default multi-interpreter is the Default Multi Interpreter, which simply calls all of the interpreters it contains, stopping when one of the interpreters says that it has successfully handled the stroke. The Multiplexed Multi Interpreter lets the developer specify one interpreter as active, which can be changed at runtime. The Semantic Zoom Multi Interpreter enables and disables interpreters depending on the current zoom level.

Runtime Handling of Strokes

Strokes are dispatched to graphical objects in a top-down manner: strokes are sent first to the parent before being re-dispatched to any of the parent's children. A stroke is re-dispatched to a child only if the child contains the stroke entirely (within a certain tolerance). By default, graphical objects handle strokes in a four-step process, as follows:

- Process the stroke with the gesture interpreters
- Re-dispatch the stroke to the appropriate children
- Process the stroke with the ink interpreters
- Handle the stroke in the graphical object

[5] Speedup $_{\text{overall}}$ = Execution time $_{\text{old}}$ / Execution time $_{\text{new}}$



Stroke Event Filter
Stroke Listener
Interpreter
Multi Interpreter
Interpreter Impl
Multiplexed Multi Interpreter Impl
Default Multi Interpreter Impl
Interpreter Wrapper
Semantic Zoom Multi Interpreter Impl
Semantic Zoom Interpreter Wrapper
Hold Select Interpreter
Circle Select Interpreter
Tap Select Interpreter
Linearize Stroke Interpreter
Rubine Interpreter
Merge Stroke Interpreter
Resize Selected Interpreter
Graffiti Interpreter
Standard Gesture Interpreter
Move Selected Interpreter

Figure 8 – Class diagram for Interpreters and Recognizers. Arrows point up towards parent classes. Rounded rectangles are interfaces; dashed square rectangles are abstract classes, and solid square rectangles are concrete classes.

At any point in this process, an interpreter or a graphical object can mark the stroke as being handled, which immediately stops the dispatching process. We give some examples of how strokes are handled in the DENIM and SketchySPICE sections below.
We chose this four-step approach as the default in order to separate handling of gestures from handling of ink. Processing gestures first lets gestures be global on the Sheet, or within a patch. This default approach can also be overridden in user code.

APPLICATIONS BUILT WITH SATIN

In this section, we describe two applications built using the SATIN toolkit, their high-level architectures, as well as how strokes are processed and interpreted in each.

First Application – DENIM

DENIM [26] is a web site design tool aimed at the early stages of information, navigation, and interaction design (See Figs. 1 and 9). An informal pen-based system [15], it allows designers to quickly sketch web pages, create links among them, and interact with them in a run mode. Zooming is used to integrate the different ways of viewing a web site, from site map to storyboard to individual page.

Although there are many gesture and ink interpreters in DENIM, from a user perspective, DENIM seems to use a minimal amount of recognition. Gestures are differentiated from ink by using the “right” pen button, while ink is created using the “left” button. This is the behavior we selected in DENIM, but can be modified in SATIN.

The scenegraph is comprised of five objects: the sheet, labels, panels, ink strokes, phrases, and arrows. The *sheet* is the root of the scenegraph. *Labels* are titles of web pages, for example “Lodging” and “Cabernet Lodge.” Labels are sticky, meaning that they are always displayed the same size, to ensure that they can always be read at the same size they were created. *Panels* are located beneath labels, and represent the content in a web page. *Ink strokes* are what are drawn in a panel. *Phrases* are collections of nearby strokes automatically aggregated together. *Arrows* connect ink and phrases from one page to another page.

Currently, DENIM only uses single stroke interpreters. All strokes are first passed through the Sheet’s gesture interpreters, and then, if rejected by all of the gesture interpreters, are passed to the ink interpreters[6]. The gesture interpreters used in DENIM are all provided by SATIN, and include (in the order called):

- *hold select*, which processes a tap and hold to select shallowly if zoomed out (i.e. selects top-level scenegraph objects such as panels), or deeply if zoomed in (i.e. deeper level scenegraph objects, such as individual ink and phrases)
- *circle select*, which processes a circle-like gesture to select everything contained in the gesture (again shallowly or deeply depending on zoom level)
- *move*, in which all selected objects are moved the same distance the pen is moved
- *standard gesture*, which uses Rubine’s recognizer [39] to recognize simple gestures like cut, copy, paste, undo, redo, and pan. Some gestures work shallowly if zoomed out, deeply if zoomed in.

[6] This is where the right and left button distinction is made. All gesture interpreters in DENIM only accept “right” button, and all ink interpreters only accept “left” button.

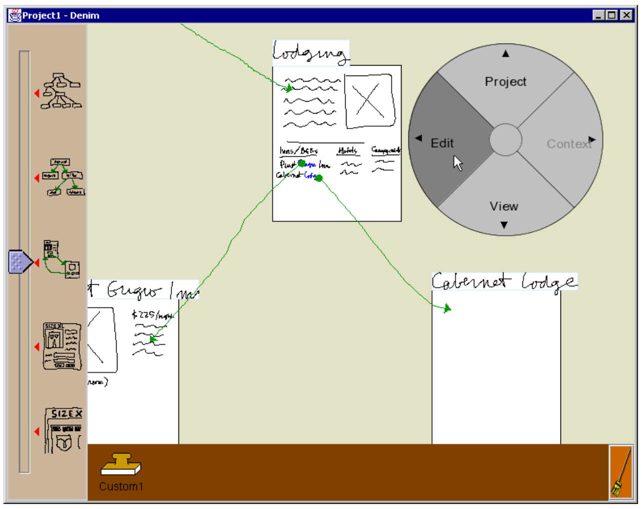


Figure 9 – A screenshot from DENIM, an application built on top of SATIN. This picture shows some ink, as well as the pie menu provided by SATIN. The Swing slider on the left is used to zoom in and out, and was modified by the PenPLAF to have a larger elevator, as well as the ability to have its value changed by taps anywhere on the slider.

If a stroke is not a gesture, then we check if the stroke should be re-dispatched to any of the Sheet's children, which in this case are labels and panels. A stroke is re-dispatched only if the label or panel bounds contain the stroke. If the stroke is re-dispatched to the label, then it is added to the label. If the stroke is re-dispatched to a panel, it is first processed by a *phrase* interpreter, which tries to group nearby ink strokes together in a single phrase object. Otherwise, it is just added to the panel as ink.

If the stroke is not re-dispatched, then the stroke is processed by the Sheet's ink interpreters. The ink interpreters are part of DENIM's code base, and include (in the order they are called):

- *arrow*, which processes lines drawn from one page to another, replacing the line by an arrow
- *label*, which processes ink that might be handwritten text, creating a new label & web page
- *panel*, which processes ink that resembles large rectangles, creating a new label and web page

If the stroke is not handled by any of the Sheet's ink interpreters, then it is just added as ink to the Sheet.

The pie menu is attached to the Sheet, and is activated by clicking the right button and not moving too far. We assigned this behavior so as not to interfere with gestures.

Second Application – SketchySPICE

SketchySPICE[7] is a simple circuit CAD tool intended as a demonstration of some features in SATIN (Figs. 10 and 11). Users can sketch AND, OR, and NOT gates, as well as wires connecting these gates. As proof-of-concept, AND and OR gates can be drawn in two separate strokes instead of just one, but this feature uses specific domain knowledge

[7] SPICE is a circuit CAD tool developed at UC Berkeley.

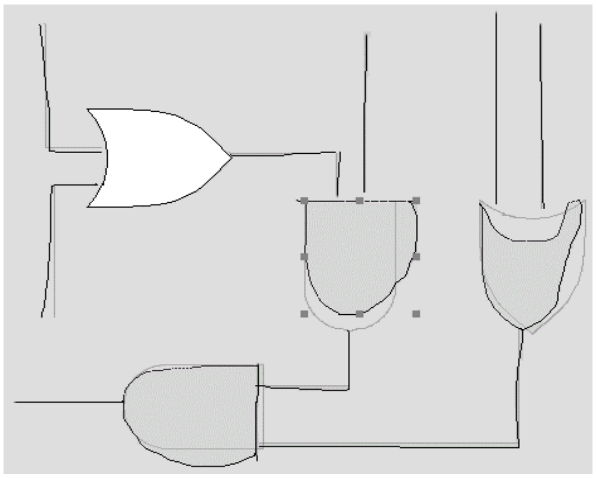


Figure 10 – A screenshot from SketchySPICE.

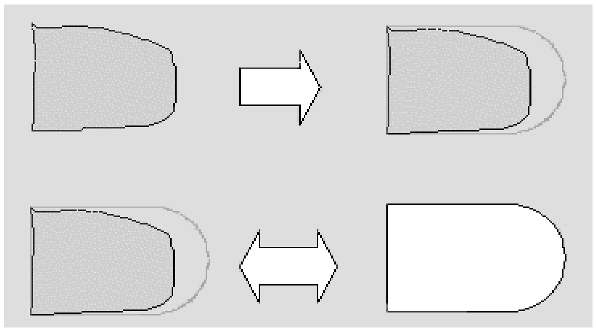


Figure 11 – SketchySPICE gives feedback by rendering the formal representation of the object translucently (top). An object can be displayed either in its original sketchy format, or in a cleaned-up format (bottom).

and is not part of SATIN. Once an object is recognized, SketchySPICE will take one of two actions, depending on the current mode. In *immediate* mode, recognized sketches are replaced immediately by a cleaned up version. In *deferred* mode, recognized objects are left sketchy, but feedback is provided to let users know that the object was recognized. This feedback consists of drawing the recognized object translucently behind the sketched object.

Individual gates can be selected and "cleaned up" to be displayed as formal looking gates, or can be "sketchified" and returned to their roughly drawn origins. In addition, the entire diagram can be cleaned up or sketchified.

The only new interpreter is the Gate interpreter. When a new stroke is added, the Gate interpreter looks at that stroke and the last stroke that was added. The two strokes are classified by Rubine's recognizer [39]. If the two separate classifications combined have a high probability of being a gate, then an AND Gate or an OR Gate object is added.

EVALUATION OF SATIN

SATIN has been in development for about two years, and is currently in its second iteration. There are about 20,000 source lines of code, and 13,000 comment lines of code, distributed in 2192 methods in 180 source code files. SATIN also uses debugging, collection, and string manipulation libraries developed by our research group, consisting of about 8000 source lines of code.

In contrast, DENIM, a fairly mature and large app, is only about 9000 source lines of code in 642 methods. The four interpreters in DENIM (arrow, label, panel and phrase) are only 1000 lines of code. Overall, it took three people three months to implement DENIM as described in [26].

SketchySPICE, a small proof-of-concept application, took about three days to implement. It is only 1000 lines of code in 32 methods. Half of the code is devoted to the pie menu, and 350 lines to the Gate interpreter.

	SATIN	DENIM	SketchySPICE
#source files	180	76	7
size of source files (kbytes)	1900	865	63
#methods	2192	642	63
#comments lines of code	13000	4500	400
#source lines of code	20000	9000	1000
#class files	220	131	32

Table 2 – Code size of SATIN and applications

Performance

We have used performance regression tests throughout the development of SATIN. The regression test suite is a repeated battery of operations, comprised of adding randomly generated graphical objects (always using the same seed value), zooming both in and out, and rotating. The regression tests were all run on the same computer, a Pentium II 300MHz running Windows NT 4.0 with a Matrox Millennium II AGP video card.

The overall performance speedup, from when the first regression test was run to when this paper was written, is 1.87. Approximately 54% of the speedup is due to code optimizations in SATIN, with the rest due to performance enhancements in the Java Virtual Machine. The two most significant gains came from polygon simplification and reduction of temporary objects generated.

FUTURE WORK

We are currently implementing a more extensive PenPLAF, which would make existing Java Swing applications more usable with pens. Besides eliminating the need for double-taps and making some widgets larger, we are also looking at integrating handwriting recognition and other interpreters with the existing Swing widgets.

Furthermore, we are working on making interpreters more sophisticated. For example, we are looking at mechanisms for adding in notions of time, to make it easy for developers to specify operations in which the pen must be held down for a period of time. We are also examining techniques to make it easier for developers to manage ambiguity. This ranges from implementing reusable, generic probabilistic data structures and algorithms, to interaction techniques, such as the mediators suggested by Mankoff [28].

SUMMARY

We introduced SATIN, a Java-based toolkit for developing informal pen-based user interfaces. By informal interfaces, we mean user interfaces that step away from the rigidity of traditional user interfaces, supporting instead the flexibility and ambiguity inherent in natural modes of communication. As a reusable toolkit, SATIN provides features common to many informal pen-based prototypes, including scenegraph support, zooming, multiple views, and stroke manipulation.

We have also described a generalized software architecture for informal pen-based applications that can handle sketching and gesturing in an extensible manner. This architecture consists of separating recognizers, which are components that classify strokes, from interpreters, which are components that process and manipulate strokes. Furthermore, multi-interpreters allow developers to specify policies of which interpreters are used and when they are used. Combined together, these features in the SATIN toolkit simplify application implementation.

With respect to input and output for pens, we are at a stage similar to that of windowing toolkits in the early 1980s. There are many bits and pieces here and there, but no cohesive frameworks to support the creation of effective informal pen-based applications. We hope that SATIN will be a significant step towards creating such a framework.

SATIN has been publicly released and can be found at: `http://guir.berkeley.edu/projects/satin`

ACKNOWLEDGEMENTS

We would like to thank Raecine Sapien, Ben Schleimer Mark Newman, James Lin, Will Lee, Benson Limketkai, Carol Hu, and Juan Valencia for their feedback and for improving the system. Lastly, we would like to thank Ben Bederson for giving us ideas and directions to explore early in the development of SATIN.

REFERENCES

1. Abowd, G., et al. Investigating the Capture, Integration and Access Problem of Ubiquitous Computing in an Educational Setting. In Proceedings of *CHI '98*. Los Angeles, CA. pp. 440-447, April 18-23 1998.
2. Apple, Newton Toolkit User's Guide. 1996.
3. Bederson, B.B. and J.D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternative Interface Physics. In Proceedings of *the ACM Symposium on User Interface Software and Technology: UIST '94*. Marina del Rey, CA. pp. 17-26, November 2–4 1994.
4. Bederson, B.B. and B. McAlister, *Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java*. Tech Report HCIL-99-07, CS-TR-4015, UMIACS-TR-99-24, University of Maryland, Computer Science Dept, College Park, MD 1999.
5. Callahan, J., et al. An Empirical Comparison of Pie vs. Linear Menus. In Proceedings of *Human Factors in Computing Systems*. pp. 95-100 1988.
6. Carr, R. and D. Shafer, *The Power of PenPoint*: Addison-Wesley, 1991.
7. Chang, B. and D. Ungar. Animation: From Cartoons to the User Interface. In Proceedings of *UIST'93*. Atlanta, GA: ACM Press. pp. 45-55 1993.

8. Palm Computing., Developing Palm OS 2.0 Applications.
9. Damm, C.H., K.M. Hansen, and M. Thomsen. Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 518-525.
10. Davis, R.C. and J.A. Landay, Making sharing pervasive: Ubiquitous computing for shared note taking. *IBM Systems Journal*, 1999. **38**(4): p. 531-550.
11. Davis, R.C., *et al.* NotePals: Lightweight Note Sharing by the Group, for the Group. In Proceedings of *CHI '99*. Pittsburgh, PA. pp. 338-345, May 15-20 1999.
12. Forsberg, A., M. Dieterich, and R. Zeleznik. The Music Notepad. In Proceedings of *UIST98*. San Francisco: ACM Press 1998.
13. Gamma, E., et al, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
14. Gross, M.D. and E.Y. Do. Ambiguous Intentions: A Paper-like Interface for Creative Design. In Proceedings of *ACM Symposium on User Interface Software and Technology*. Seattle, WA. pp. 183-192, November 6–8 1996.
15. Hearst, M.A., M.D. Gross, J.A. Landay, and T.E. Stahovich. Sketching Intelligent Systems. *IEEE Intelligent Systems*, 1998. **13**(3): p. 10-19.
16. Henry, T.R., S.E. Hudson, and G.L. Newell. Integrating Gesture and Snapping into a User Interface Toolkit. In Proceedings of *UIST90*: ACM Press 1990.
17. Hudson, S.E. and I. Smith. Ultra-Lightweight Constraints. In Proceedings of *UIST96*: ACM Press 1996.
18. Igarashi, T., et al. An Architecture for Pen-based Interaction on Electronic Whiteboards. To Appear In Proceedings of *Advanced Visual Interfaces*. Palermo, Italy May 2000.
19. Igarashi, T., S. Matsuoka, S. Kawachiya, and H. Tanaka. Pegasus: A Drawing System for Rapid Geometric Design. In Proceedings of *CHI98*. Los Angeles: ACM Press. 1998.
20. Igarashi, T., S. Matsuoka, and H. Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. In Proceedings of *ACM SIGGRAPH99*. Los Angeles: ACM Press. pp. 409-416 1999.
21. Kramer, A. Dynamic Interpretations in Translucent Patches. In Proceedings of *Advanced Visual Interfaces*. Gubbio, Italy, 1996.
22. Kramer, A. Translucent Patches – Dissolving Windows. In Proceedings of *ACM Symposium on User Interface Software and Technology*. Marina del Rey, CA. November 2–4 1994.
23. Landay, J.A. and B.A. Myers. Extending an Existing User Interface Toolkit to Support Gesture Recognition. In Proceedings of *INTERCHI '93*. Amsterdam, The Netherlands. pp. 91-92, April 24–29 1993.
24. Landay, J.A. and B.A. Myers. Interactive Sketching for the Early Stages of User Interface Design. In Proceedings of *CHI '95*. Denver, CO. pp. 43-50, May 7–11 1995.
25. Lassiter, J. Principles of Traditional Animation Applied to 3D Computer Animation. In Proceedings of *ACM SIGGRAPH '87*: ACM Press. pp. 35-44 1987.
26. Lin, J., M. Newman, J. Hong, and J. Landay. DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 510-517.
27. Long, A.C., J.A. Landay, and L.A. Rowe. Implications For a Gesture Design Tool. Proceedings of *CHI '99*. Pittsburgh, PA. pp. 40-47, May 15-20 1999.
28. Mankoff, J., S.E. Hudson, and G.D. Abowd. Providing Integrated Toolkit-Level Support for Ambiguity in Recognition-Based Interfaces. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 368-375.
29. Microsoft, *Microsoft Windows for Pen Computing - Programmer's Reference Version 1*: Microsoft Press, 1992.
30. Microsoft, MSDN Library: Windows CE Documentation. http://msdn.microsoft.com/library/default.asp
31. Sun Microsystems. Java Foundation Classes. http://java.sun.com/products/jfc
32. Moran, T.P., P. Chiu, and W. van Melle. Pen-Based Interaction Techniques For Organizing Material on an Electronic Whiteboard. In Proceedings of *UIST '97*. Banff, Alberta, Canada. pp. 45-54, October 14-17 1997.
33. Moran, T.P., P. Chiu, W. van Melle, and G. Kurtenbach. Implicit Structures for Pen-Based Systems Within a Freeform Interaction Paradigm. In Proceedings of *CHI'95*. Denver, CO. May 7–11 1995.
34. Myers, B. *et. al.*, Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 1990. **23**(11): p. 289-320.
35. Myers, B. and D. Kosbie. Reusable Hierarchical Command Objects. In Proceedings of *CHI'96*. Vancouver, BC, Canada: ACM Press 1006.
36. Myers, B., *et al.*, The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering*, 1996. **23**(6): p. 347-365.
37. Mynatt, E.D., T. Igarashi, W.K. Edwards, and A. LaMarca. Flatland: New Dimensions in Office Whiteboards. In Proceedings of *CHI99*. Pittsburgh, PA: ACM Press 1999.
38. Rubine, D., *The Automatic Recognition of Gestures*, Unpublished Ph.D. Carnegie Mellon, Pittsburgh, PA, 1991.
39. Rubine, D., Specifying Gestures by Example. *Computer Graphics*, 1991. **25**(3): p. 329-337.
40. Saund, E. and T.P. Moran. A Perceptually-Supported Sketch Editor. In Proceedings of *the ACM Symposium on User Interface Software and Technology: UIST '94*. Marina del Rey, CA. pp. 175-184, November 2–4 1994.
41. Schilit, B.N., G. Golovchinksy, and M.N. Price. Beyond Paper: Supporting Active Reading with Free Form Digital Ink Annotations. In Proceedings of *CHI '98*. Los Angeles, CA. April 18-23 1998.
42. Truong, K.N., G.D. Abowd, and J.A. Brotherton. Personalizing the Capture of Public Experiences. In Proceedings of *UIST'99*. Asheville, NC 1999.
43. van de Kant, M., et al. PatchWork: A Software Tool for Early Design. In Extended Abstracts of *CHI '98*. Los Angeles, CA. pp. 221-222, April 18-23 1998.
44. Weber, K. and A. Poon. Marquee: A Tool for Real-Time Video Logging. In Proceedings of *CHI '94*. Boston, MA. pp. 58-64, April 24-28 1994.
45. Whittaker, S., P. Hyland, and M. Wiley. Filochat: Handwritten Notes Provide Access to Recorded Conversations. In Proceedings of *CHI '94*. Boston, MA. April 24-28 1994.
46. Wilcox, L.D., B.N. Schilit, and N.N. Sawhney. Dynomite: A Dynamically Organized Ink and Audio Notebook. In Proceedings of *CHI'97*. Atlanta, GA. pp. 186-193, March 22-27 1997.
47. Zeleznik, R.C., K.P. Herndon, and J.F. Hughes, SKETCH: An Interface for Sketching 3D Scenes. *Computer Graphics (Proceedings of SIGGRAPH '96)*, 1996.

LipiTk: A Generic Toolkit for Online Handwriting Recognition

Sriganesh Madhvanath *Deepu Vijayasenan* *Thanigai Murugan Kadiresan*
Hewlett-Packard Laboratories
Bangalore, India
{srig, deepuv}@hp.com

Abstract

This paper describes Lipi Toolkit (LipiTk) - a generic toolkit whose aim is to facilitate development of online handwriting recognition engines for new scripts, and simplify integration of the resulting engines into real-world application contexts. The toolkit provides robust implementations of tools, algorithms, scripts and sample code necessary to support the activities of handwriting data collection and annotation, training and evaluation of recognizers, packaging of engines and their integration into pen-based applications. The toolkit is designed to be extended with new tools and algorithms to meet the requirements of specific scripts and applications. The toolkit attempts to satisfy the requirements of a diverse set of users, such as researchers, commercial technology providers, do-it-yourself enthusiasts and application developers. In this paper we describe the first version of the toolkit which focuses on isolated online handwritten shape and character recognition.

Keywords: Online Handwriting Recognition, Shape Recognition, Toolkit, Linguistic Resources, API

1. Introduction

There are still large parts of the world characterized by the extensive use of paper and handwriting in all facets of society, and poor penetration of traditional PCs and keyboards. In India for instance, only 12 in 1000 persons have PCs, as compared to 785 in the US, and over 500 in most Western European countries[1]. In this setting, products and solutions with pen and/or paper-based interfaces may play an important role in making the benefits of Information Technology more pervasive. HWR is an important technology in order to research appropriate user interfaces, and create innovative products, solutions and services for these markets. For example, the Gesture Keyboard [14] is an experimental desktop peripheral that uses recognition of handwritten gestures for Indic text input, and form-filling solutions in local languages also appear to have potential [15].

Unfortunately for many of the languages in these parts of the world - such as the Indic languages - no commercial handwriting recognition technology exists. The central problem being addressed in this paper is: how can we simplify the creation of HWR technology for a new script, and how can we simplify its integration into real-world applications? This problem has been addressed by sister language technology communities working on speech recognition, speech synthesis, and machine translation through the creation of toolkits comprised of tools and algorithms that can be used to create language technology for a new language [1,2]. The issue of integration has been addressed by the creation of standard interfaces/protocols such as MRCP for speech recognition engines [3]. However, to the best of our knowledge, no generic toolkit or standards exist for online handwriting recognition.

1.1. The Lipi Toolkit

There are many different challenges involved in developing a "generic" toolkit for online handwriting recognition. The first is that the toolkit should provide good enough generic components to perform reasonably well on simple as well as complex scripts, while providing the flexibility to tune, extend or even replace them with more suitable components, to meet the challenge of a particular script or application. A second challenge is to balance the needs of different classes of potential users (Figure 1). For researchers in handwriting recognition, the toolkit should serve as a

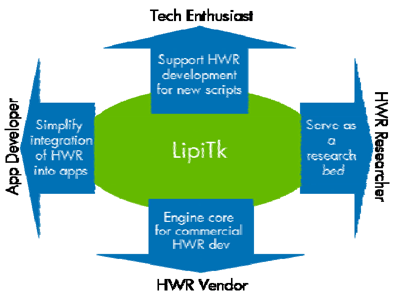

Tech Enthusiast
Support HWR development for new scripts
App Developer
Simplify integration of HWR into apps
LipiTk
Serve as a research bed
HWR Researcher
Engine core for commercial HWR dev
HWR Vendor

Figure 1. Design goals of Lipi Toolkit

research bed to experiment with new algorithms. For a certain class of do-it-yourself enthusiasts, it should allow the creation of engines for new shapes and

[1] 2004 figures, Source: Gartner Report

scripts out of the box, without requiring much knowledge of the algorithms. For a potential vendor interested in building commercial HWR engines, it should support the building of robust engines for new scripts. Finally, for the application developer, it should allow easy integration of engines built using the toolkit into any pen-based application.

The Lipi Toolkit ("lipi" being Sanskrit for "script") is our effort to create a generic toolkit whose components can be used to build an online HWR engine for a new script, while addressing the challenges described. There are three distinct stages in the lifecycle of the toolkit. In the first stage, standard tools and algorithms are packaged into a "downloadable toolkit". In the second stage, an intermediate user (such as a researcher or vendor) downloads the toolkit source and binaries, experiments with and potentially modifies the algorithms or adds new ones in order to build a custom engine for a new script and/or application context. In the third stage, an end-user (such as the application developer) integrates the built engine into a pen-based application. The three stages can clearly be separated temporally and spatially and involve very different sets of users. While designing the toolkit, our emphasis is primarily in the first stage, while ensuring that the toolkit supports the second and third stages in the hands of the respective user groups.

The first version of the toolkit (LipiTk 1.0) supports the recognition of isolated handwritten shapes and characters, as well as boxed words, captured as digital ink. The components of the toolkit make it possible to carry out all the steps involved in building a character recognition engine. These components are represented in Figure 2 and described in Section 3. But first we take a look at some of the salient features of the toolkit in the light of previous efforts.

2. LipiTk: Salient Features

While toolkits such as Sphinx from Carnegie Mellon University [1] and Festival from University of Edinburgh [2] exist for problems such as Automatic Speech Recognition and Speech Synthesis respectively, we believe LipiTk to be one of the first to address the problem of online HWR. Open source implementations of online gesture and character recognition such as Rosetta [4], XStroke [5] and WayV [6] are not primarily intended for experimentation with HWR algorithms, which is one of the (many) core goals of LipiTk.

LipiTk is designed to support a data-driven methodology for the creation of recognition engines. This implies tools for handwriting data collection and annotation, standard script-independent algorithms for preprocessing and feature extraction, algorithms for training and pattern classification, and tools for subsequent evaluation and error analysis.

While many handwriting recognition algorithms are script specific, LipiTk is intended to provide robust implementations of generic features and classifiers that are expected to perform reasonably well on any given set of symbols by learning the statistical shape properties of that set. This allows a reasonably performing recognition engine to be built with a minimum of effort.

One of the characteristics of handwriting recognition is that no single approach or set of features/classification algorithms is known to work optimally for all scripts. Also, the nature and quality of the input (digital ink) tends to vary widely with the capture device. LipiTk has been designed to accommodate different tools and algorithms specific to the device, script and/or application.

LipiTk 1.0 uses open standards such as UNIPEN [17] for the representation of digital ink and its annotation, facilitating the creation of shareable linguistic resources within the community. Future versions may use W3C InkML [18] and UPX [19] for digital ink and annotation respectively.

There is a focus on robust and efficient implementation of algorithms in LipiTk, in order to facilitate the integration of engines created using the toolkit into real-world applications.

LipiTk also specifies standard shape and word recognition interfaces for recognition engines. All engines built using the toolkit hence expose a standard interface, simplifying their integration into pen-based applications. The shape recognition API is less ambitious in some aspects than previous efforts [7], but is distinct from them in that it includes training and online adaptation of shape recognizers.

3. LipiTk Architectural Design

The Lipi Toolkit was intended to support both Windows and Linux platforms, hence its design and implementation considers portability related issues. Most of the algorithms and tools are implemented using C++ & STL. Only ANSI functions are used for portability. Some of the utilities and scripts are written in Perl. The major components of the toolkit are described below.

3.1. Generic Class and Utilities Library

The generic class library includes classes to store and manipulate ink traces, such as *Trace* and *TraceGroup*, and classes to store device and screen

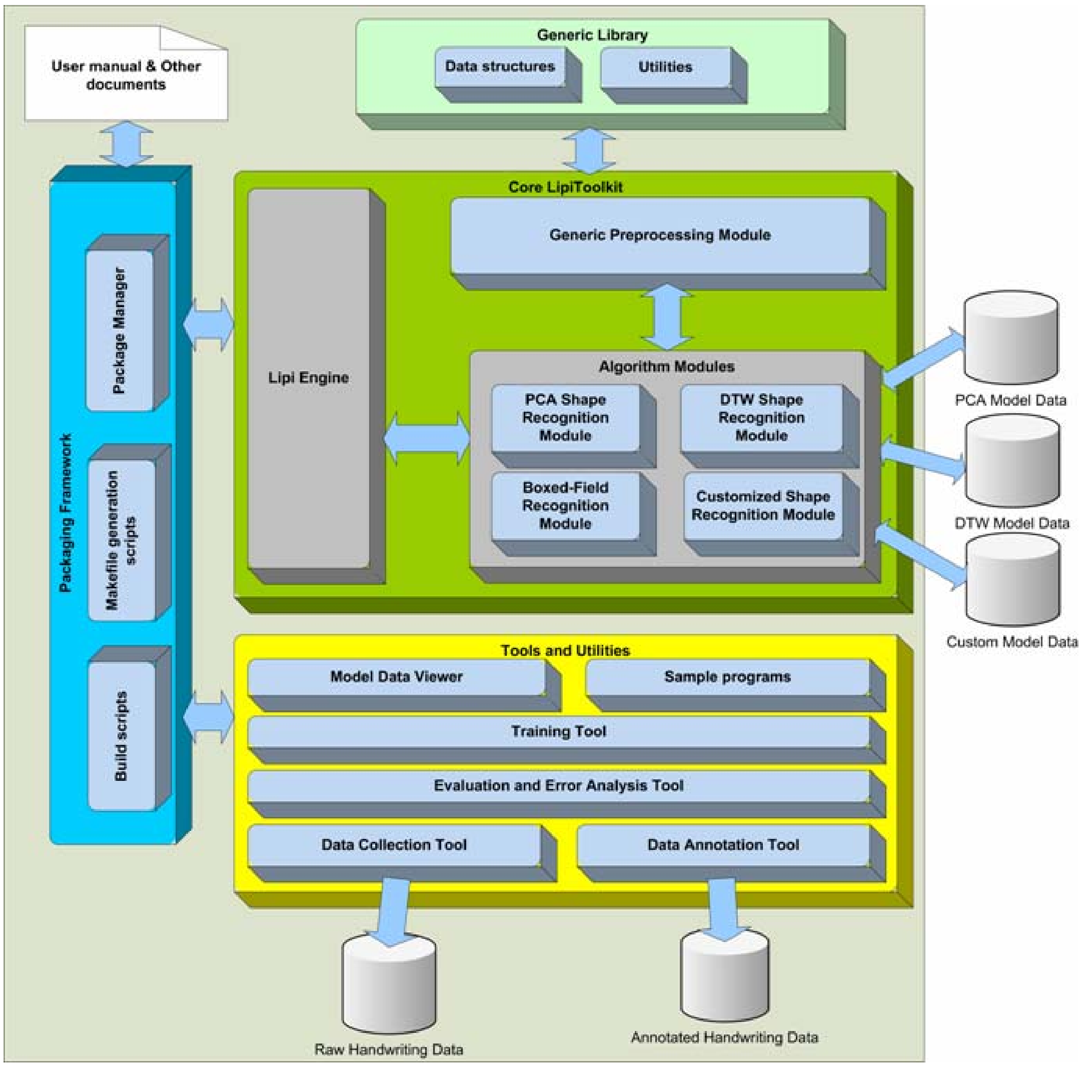


Figure 2. Lipi Toolkit Architecture and Components

context. These classes are shared by different algorithm and tool implementations. The design of these classes reflects a tradeoff between a conceptually intuitive and object-oriented data model, and efficient access to frequently accessed attributes, such as X and Y channels in the case of ink traces.

The utilities library provides utility functions to read and write LipiTk configuration files, read and write UNIPEN data files, and so on.

3.2. HWR algorithms

LipiTk 1.0 provides implementations of common preprocessing operations, common shape recognition algorithms, as well as a boxed field recognizer which iteratively calls one of the shape recognizers to interpret a boxed field of ink input. The preprocessing module, as well as the shape recognition and boxed field recognition modules are implemented as separate dynamic link libraries that can be loaded at runtime.

3.2.1. Generic Preprocessing

The generic preprocessing module provides implementations of commonly used shape/character preprocessing operations such as moving-average smoothing, size normalization, dehooking, and equidistant resampling. All of the operations have configuration options that can be varied using corresponding properties captured in a configuration file.

3.2.2. Shape Recognition

The two shape recognition algorithms bundled with LipiTk 1.0 are Subspace-based classification (PCA), and Nearest-Neighbor classification based on Dynamic Time Warping (DTW).

In subspace classification, each shape class is represented by a set of Principal Components computed from a fixed length representation of the online shape, obtained after size normalization and equidistant resampling [12]. The training method provided computes the Principal Components from the training data, and stores them in a standard binary format.

The DTW implementation uses the same fixed length representation as the subspace classifier, together with a Nearest Neighbor classifier [10,11]. The training method exposed by the shape recognizer provides a choice of different prototype selection algorithms for prototype reduction.

Both shape recognizers expose a standard shape recognition API which allows the recognizer to be loaded, trained, and invoked on a *TraceGroup* (group of traces) corresponding to a single or multi-stroke shape or character.

In either case, important parameters (such as the number of Principal Components) as well as the sequence of preprocessing operations are externally configurable using configuration files.

3.2.3. Boxed field recognition

As mentioned earlier, the boxed field recognizer is useful for recognizing a boxed field of shapes, and in turn invokes a trained shape recognizer on each of the boxes, and uses a simple trellis for decoding the best strings based on the cumulative shape recognition confidences.

Significantly, the boxed field recognizer exposes a generic word recognition API, allowing the possibility of plugging in a connected word recognizer in the future in a backward-compatible manner.

3.3. Tools and Utilities

LipiTk 1.0 provides a number of tools and utilities to support the tasks of handwriting data collection, data annotation, and the training and evaluation of shape recognizers.

3.3.1. Data Collection and Annotation Tools

Collection and annotation of handwriting data is an important activity in the data-driven methodology for creating recognition engines. LipiTk 1.0 includes a generic TabletPC-based data collection tool capable of collecting isolated symbols, characters or words from writers [13]. We hope to include in subsequent releases, tools based on Digital Pen and Paper or PDAs, as well as tools running on Linux.

The annotation tool supports the tagging of digital ink with labels corresponding to ground truth, writing style etc. at different levels of an appropriate hierarchy of annotation. The tool included with LipiTk 1.0 is a generic tool written in C++/Qt with the ability to annotate entire documents of handwritten text, and supports plug-ins for segmentation and recognition to partially automate the annotation process [22].

3.3.2. Evaluation Tool

The Evaluation Tool computes statistics related to classification accuracy and performance of the built engine on the test data, and allows visualization of the results in ways that facilitate analysis of the errors. LipiTk 1.0 includes a basic evaluation tool written in Perl which renders top N accuracies and confusion matrices in the form of HTML pages.

3.3.3. Utilities

In addition to the above tools, LipiTk 1.0 also includes a number of scripts to facilitate tasks such as extraction of isolated character data from the annotated data (which may be words), and splitting the annotated data randomly into training and test sets.

3.4. Packaging framework

LipiTk provides build scripts to support the creation of specific engines from the source code. These scripts interpret project configuration files and build the necessary source code into libraries and binaries, using a hierarchy of static module-specific *Makefiles*.

LipiTk also provides scripts for packaging the built engine(s) for deployment, and integration into a pen-based application. The components of the package are fully user-configurable, and the packaging script creates a self-extract package file (or gzipped tar file in the case of Linux) that contains all the components selected for packaging by the user.

Finally, LipiTk provides sample code to assist the application developer in integrating an engine created using LipiTk into his or her application.

3.5. Lipi Engine

The Lipi Engine is the run-time component of engines created using the Lipi Toolkit. It is responsible for loading one or more shape/word recognition modules as specified in its configuration file, routing requests for recognition from the user application to the appropriate modules, and returning recognition results to the application.

4. Working with the Toolkit

In this section, we will take a brief look at the use of the toolkit to develop a recognizer for a set of shapes, for example, the 10 Indo-Arabic numerals. The first step in the process is the creation of a new shape recognition project, which we will call `numrec`. The associated project configuration file identifies the project as a shape recognition project (as opposed to a word recognition project), and the number of shape classes to be recognized as 10. Within the `numrec` project, one or more profiles may be defined. Each profile contains configuration settings for the recognizer corresponding to a particular user-defined mode of operation or “flavor” of the recognizer. Profiles may be used to distinguish settings for writer-independent recognition vs. specific writers, specific training data (e.g. US versus UK writers), specific recognition algorithms (PCA vs. DTW vs. Custom), specific preprocessing settings, and so on – at the discretion of the researcher/developer. The profile also stores the results of training the shape recognizer (recognizer-specific model data) using the specific set of configuration settings.

The build scripts provided with the toolkit allow the building of a specific project and profile, and use the associated settings to determine which recognizers to build. For instance, if the `numrec` project's `default` profile specified PCA as the recognizer, and PCA's configuration file in turn specified a sequence of preprocessing operations, building the project and profile would cause the generic preprocessing as well as the PCA code to be built into dynamic link libraries. Building the project also builds a command line utility `runshaperec` that links with these libraries and supports training and testing of the shape recognizer on labeled numeral data.

The data collection tool provided with the toolkit may be used to collect samples of numerals from different writers, and organize them in the form of an annotated UNIPEN dataset. Alternatively an existing dataset may be used. A utility script supports the creation of training and test lists from the dataset using regular expressions to match file names.

The `runshaperec` utility may now be used to train the shape recognizer by providing the training list of samples as input. This causes subspaces for the 10 classes to be computed as stored as binary model data as part the `default` profile.

The same `runshaperec` utility may be used in evaluation mode to classify the test samples in the test list. The result file produced may then be provided as input to the evaluation tool to compute recognition accuracy and the confusion matrix.

Once the cycle of training and testing is completed, the packaging script may be used to package the entire `numrec` project and `default` profile into a self-extracting archive file or gzipped tar file for deployment on the target machine where the pen-based application is going to be deployed.

On the target machine, the numeral recognizer may be extracted and integrated into a pen-based application as per the sample code provided.

The above describes the simplest of scenarios wherein an existing recognition algorithm (PCA) is used for creating a shape recognizer more or less out of the box. Other scenarios address the extraction of isolated shape data for training from words or larger units of writing, adding and using new preprocessing methods, experimenting with different configuration parameters, writing a new shape recognizer, and using the Boxed Field recognizer to recognize a boxed field of characters.

5. Status

The first version of LipiTk has been implemented at HP Labs, Bangalore, India, by a team composed of GDIC engineers and HP Labs researchers, and was released internally in Dec 2005. The toolkit was subsequently released into the Open Source in April 2006 under a BSD-like license, and is now hosted on SourceForge at http://lipitk.sourceforge.net. As mentioned earlier, this first version is aimed at isolated shape and character recognition, and includes generic tools for data collection, annotation and evaluation, source code for common preprocessing and classification, build and packaging script, miscellaneous scripts, and sample data and code.

The included simple model-based shape classification algorithms based on Dynamic Time Warping and Principal Component Analysis have been shown to perform in the range of 80% accuracy on Indic scripts such as Tamil whose character set has many similar looking characters with complex shapes [11,12]. These algorithms have also been benchmarked on standard datasets such as Unipen [20] and IRONOFF [21]. In all these cases, it is clear that while these model-based methods offer a reasonable first level of classification, discriminative methods together with other features are necessary to achieve high accuracy. The toolkit is designed to support the addition of such methods.

Internally, the toolkit has been used to create the gesture-stroke recognition component of the Gesture Keyboard for Devnagari [23]. Specifically, this uses the PCA recognizer in a rank-based combination with a global gesture shape recognizer, and yields 97% accuracy on the constrained gestures used by the Devanagari Gesture Keyboard. The toolkit is also being used internally to explore other concepts in text input of Indic scripts, and solutions such as form filling in local languages [15,16].

6. Summary and Next Steps

In summary, the Lipi toolkit effort aims to facilitate development of online handwriting recognition engines for new scripts, and simplify integration of the resulting engines into real-world application contexts. The first version of the toolkit provides robust implementations of tools, algorithms, scripts and sample code necessary to support the entire process starting from the collection of handwritten data, to the deployment and integration of a robust engine, for a particular set of shapes or characters.

The design of the toolkit makes it possible to integrate new tools and algorithms (such as a data collection tool specifically for gestures, a different type of preprocessing, or classification algorithm) into the toolkit. For instance, to facilitate training of a classifier, LipiTk provides a common interface that calls the training module of a specific classifier, while hiding the specific details of its implementation from the user.

Given the need to support the potential LipiTk user community (whether researchers or application developers) across multiple operating systems and computing platforms, the toolkit is also designed to simplify creation of versions for different platforms using a common code base.

As already indicated, there are several important research directions for the toolkit, including inclusion of discriminative classification algorithms, native support for emerging standards such as W3C Ink Markup Language, improved tools for data collection, annotation and error analysis, and even potential extensions to Offline HWR. However, our major focus at present is to validate the design and utility of the toolkit with different sets of users. Projects within HP Labs such as the Gesture Keyboard are the first “internal” users of the toolkit, and provide an opportunity to study the usability and cross-platform robustness of the toolkit at close quarters. We are also interested in collaborative projects with university research groups using the toolkit. We hope that some of these users can contribute by trying to use the toolkit and providing feedback, while others may contribute to the toolkit by way of new tools and algorithms.

Acknowledgements

The authors would like to acknowledge Sudhakaran K., Rajesh Pandry and Vijay Kumar of the LipiTk Development team, GDIC, and M. Dinesh, Sridhar Muralikrishna, V. Jagannadan and A. Bharath from the Pen-based Solutions and Handwriting Recognition team at HP Labs India, for their contributions towards the toolkit.

References

[1] The Carnegie Mellon Sphinx Project, http://cmusphinx.sourceforge.net.

[2] The Festival Speech Synthesis System, http://www.cstr.ed.ac.uk/projects/festival/.

[3] Shanmugham, S., Monaco, P. and B. Eberman, "MRCP: Media Resource Control Protocol", Internet Draft draft-shanmugham- mrcp-05, January 2004

[4] Rosetta - Multistroke / Full Word Handwriting Recognition for X, http://www.handhelds.org/project/rosetta/

[5] XStroke: Full-screen Gesture Recognition for X,

[6] WayV Project, http://www.stressbunny.com/wayv/

[7] HRE API: A Portable Handwriting Recognition Engine Interface, http://playground.sun.com/pub/multimedia/handwriting/hre.html

[8] C.C Tappert, C.Y Suen, and T.Wakahara, “The State of the Art in On-Line Handwriting Recognition,” IEEE PAMI, vol.12, No.8, pp. 179-190, 1990.

[9] Rejean Plamondon and Sargur N. Srihari, “On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey,” IEEE, vol. 22, No.1, pp. 63-84, 2000.

[10] S.D Connell, R.M.K Sinha, and A.K.Jain, “Recognition of Unconstrained On-Line Devanagari Characters”, in Proc. 15th ICPR, pp. 368-371, 2000.

[11] Niranjan Joshi, G. Sita, and A. G. Ramakrishnan and Sriganesh Madhvanath, “Comparison of Elastic Matching Algorithms for Online Tamil Handwritten Character Recognition”, Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition (IWFHR-9), Tokyo, Oct 2004

[12] Deepu Vijayasenan, A.G. Ramakrishnan and Sriganesh Madhvanath, “Principal Component Analysis for Online Handwritten Character Recognition”, 17th Intl Conf. Pattern Recognition (ICPR 2004), Cambridge, United Kingdom, August 23-26, 2004.

[13] Experiences in Collection of Handwriting Data for Online Handwriting Recognition in Indic Scripts, Ajay S Bhaskarabhatla and Sriganesh Madhvanath, 4th Intl Conf. Linguistic Resources and Evaluation (LREC 2004), Lisbon, Portugal, May 26-28, 2004

[14] Gesture Keyboard - User Centered Design of a Unique Input Device for Indic Scripts , Ashish Krishna, Rahul Ajmera, Sandesh Halarnkar and Prashant Pandit, HCI International-2005, Las Vegas, Nevada, USA, July 22-27 2005.

[15] Indic scripts based online form filling - A usability exploration, Ashish Krishna, Girish Prabhu, Kalika Bali, Sriganesh Madhvanath, 11th International Conference on Human-Computer Interaction(HCI 2005), Las Vegas, July 22-27, 2005

[16] Coffei: Common Forms Framework for Electronic Ink, Sriganesh Madhvanath et al., HP Tech Con ’05.

[17] UNIPEN 1.0 Format Definition, http://www.unipen.org/pages/5/index.htm

[18] InkML – The Digital Ink Markup Language, www.w3.org/2002/mmi/ink

[19] UPX: A New XML Representation for Annotated Datasets of Online Handwriting Data, Mudit Agrawal, Kalika Bali, Sriganesh Madhvanath, Louis Vuurpijl, 8th International Conference on Document Analysis and Recognition (ICDAR 2005), Seoul, Korea, Aug 29 - Sept 1, 2005.

[20] The UNIPEN collection release #1: train_r01_v07, http://unipen.nici.ru.nl/cdroms/

[21] C. Viard-Gaudin, P.M. Lallican, S. Knerr, P. Binter, “The IRESTE On/Off (IRONOFF) Dual Handwriting Database”, ICDAR’99.

[22] Representation and Annotation of Online Handwritten Data, Ajay S. Bhaskarabhatla, Sriganesh Madhvanath, M. N. S. S. K. Pavan Kumar, A. Balasubramanian and C. V. Jawahar, Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition (IWFHR-9), Tokyo, Oct 2004.

[23] Handwritten Gesture Recognition for Gesture Keyboard, R. Balaji, V. Deepu, Sriganesh Madhvanath and Jayasree Prabhakaran. Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition (IWFHR-10), La Baule, France, Oct 2006.

Sketch Understanding Systems

Speech and Sketching for Multimodal Design

Aaron Adler and Randall Davis
MIT Computer Science and Artificial Intelligence Laboratory
77 Massachusetts Avenue, 32-239
Cambridge, MA 02139 USA
{cadlerun, davis}@csail.mit.edu

ABSTRACT

While sketches are commonly and effectively used in the early stages of design, some information is far more easily conveyed verbally than by sketching. In response, we have combined sketching with speech, enabling a more natural form of communication. We studied the behavior of people sketching and speaking, and from this derived a set of rules for segmenting and aligning the signals from both modalities. Once the inputs are aligned, we use both modalities in interpretation. The result is a more natural interface to our system.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces—*Natural language, Graphical user interfaces (GUI), Evaluation/methodology, Input devices and strategies (e.g., mouse, touchscreen), Interaction styles (e.g., commands, menus, forms, direct manipulation), User-centered design, Voice I/O*

General Terms

Performance, Design, Experimentation, Human Factors

Keywords

speech, sketch, multimodal interaction

1. INTRODUCTION

Sketches are commonly used in the early stages of design. Our previous system, ASSIST[2], lets users sketch in a natural fashion and recognizes mechanical components (e.g., springs, pulleys, axles, etc.). Sketches can be drawn with any variety of pen-based input (e.g., tablet PC). ASSIST (see Figure 1) displays a "cleaned up" version of the user's sketch and interfaces with a simulation tool to show users their sketch in action.

Some parts of a mechanical system might be too difficult to express by sketching alone, but might be easy to describe verbally. In that case, adding speech recognition creates a more natural user interface. Our goal is to create a multimodal system where the user

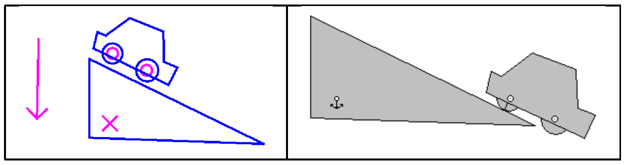


Figure 1: The left image shows the sketch in ASSIST. The right image shows the simulation.

can have a natural conversation with the computer, of the sort a user might have with another person. We do not want the speech to be limited to simple, single word commands, like uttering "spring" while pointing. Rather, we want to allow the user to say whatever comes to mind and have the system gather as much as possible from the speech input [1].

We begin with an example that motivates our work, then describe how we collected data and created the set of rules for our system. Next, we describe how the speech and sketching components of the system are combined and conclude with related and future work.

2. MOTIVATING EXAMPLE

Newton's Cradle (see Figure 2) is a system of pendulums that consists of a row of metal balls on strings. When you pull back a number of balls on one end, after a nearly elastic collision, the same number of balls will move outward from the other end of the system. Although this system seems simple enough to sketch, it is in fact nearly impossible to draw so that it operates properly. The system works because the metal balls at the end of the pendulums just touch each other, and because each pendulum is identical to the others. In the sketching system, you would have to draw identical pendulums, and align them perfectly. If the user could simply say that "there are five identical, evenly spaced and touching pendulums," the device would be easy to create.

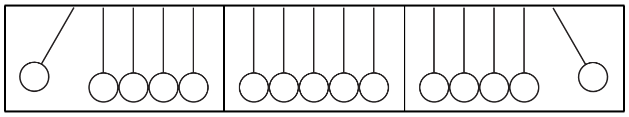


Figure 2: A sequence of images showing Newton's Cradle when one of the pendulums is pulled back and released.

3. OBTAINING SAMPLE DATA

To support natural speech, we conducted an empirical investigation of spoken descriptions of mechanical devices while the participant was drawing. We videotaped six outside participants while they sketched six mechanical systems at a whiteboard. They were given small hardcopy drawings of the systems and were told to draw them on the whiteboard, describing them as they did so. They were told to describe them as if they were talking to a small group of people, such as in a physics tutorial. The figures had marks to indicate identical components and identical distances. These graphical marks were provided to get an idea of how the participants would describe identical or equally spaced objects without inadvertently biasing their language by using words we had chosen. The recordings from the participants were transcribed, and each speech and sketching action was time-stamped. This provided a basis for developing a set of approximately 50 rules that could segment and align the speech and sketching events.

4. SEGMENTING DATA

The data from the videos were analyzed by hand, segmenting it into individual speech events (roughly, phrases) and sketching events (drawing part of an object), and aligning corresponding events. From this analysis, we manually derived a set of rules that encapsulated the knowledge gathered. Some rules group objects that are the same shape (e.g., grouping consecutively drawn triangles), others use the timing between the speech and sketching events to identify overlapping events and pauses between events (e.g., pauses are gaps of at least 0.8 seconds where there is no sketching or speech event), while others look for key words in the speech events. For example, words such as “and,” “then,” or “there are” were good indicators that the user started a new topic. In our analysis we noted that users never talked about one thing while sketching another.

The rules determine a set of times, or break points, that group together speech and sketching events that refer to the same objects. One rule indicates a possible break point when a speech utterance starts with a key word which is preceded by a pause. This might produce a group that included the speech phrase “that’s suspended by springs on the bottom” and the three sketching events in which a spring is sketched.

The rules were created using 18 data sets. The rules were kept general and do not use specific features or vocabulary of the mechanical engineering domain.

This process of segmenting and aligning the data also allows us, in a limited way, to use both modalities in interpretation. For example, if the user draws three pendulums and says there are two, the system will ignore the speech. However, if the user says that there are four pendulums, then the system will wait for another pendulum to be drawn.

There are three stages to the processing of the speech and sketching. The initial partitioning of both is done by the rule system. In the second phase, a search is conducted within a group found in the first phase to align the speech and sketching events (e.g., match the speech event containing the word “pendulums” with any sketched pendulums). In the third phase, the search is widened to adjacent groups in the event that the correspondence can’t be found in the original group alone. The third phase relaxes the constraints determined by the rules to provide more flexibility in the grouping.

4.1 Results

To determine how well the rules work, the transcript files from the videos were parsed and run though the rule system, with each speech and sketching action presented sequentially as if arriving from a user. The data used to test the system was separate from the data used to create the rule system.

The results of running the rules on the video transcripts were compared in detail to hand-generated results for 4 data sets that comprised the test set. There were 29 break points in the hand-generated segmentations. The computer-generated segmentation matched on 24 of these, and found 18 additional break points. The 18 additional break points were analyzed by hand and further classified as “incorrect,” “inconsequential,” or as resulting from “shallow knowledge.” The “inconsequential” category includes break points that were immaterial to parsing, such as break points added at the beginning, prior to any speech or sketching events, and extra break points between some speech events at the end of the interaction (see Table 1). The “shallow knowledge” category contains additional break points that were placed between sketching events (see Table 2).

1a	“I’m puzzled as to how to indicate that”
2a	“equal size of”
2b	“the suspended balls”
3a	“and that it is not the same as”
3b	“the falling balls”

Table 1: Data from one of the participants exhibits how the speech we are working with is not grammatical. The hand segmentation placed all 5 events into the same group, however, the software placed the events into three groups by placing “inconsequential” break points between speech events 1a and 2a and between speech events 2b and 3a.

1a	“The slopes are fixed in position”
1b	[draws middle ramp]
1c	[draws middle ramp anchor]
2a	[draws bottom ramp]
2b	“slope”

Table 2: Example of a “shallow knowledge” break point. The hand segmentation placed all 5 events into the same group, however the software placed the events into two groups by placing an extra break point between sketching events 1c and 2a. The rules do not have any knowledge of the meaning of the anchor or the spatial relationship between the ramps. As a result, the rules did not place these events into the same group, as the hand segmentation did.

The hand segmentation had the advantage of having all the sketching and speech events to examine at once, as well as the spatial relationships between sketched components. The software segmentation processed speech and sketching events sequentially and did not have access to any spatial relationship information.

5. SYSTEM OVERVIEW

Figure 3 shows screen shots of the working system.

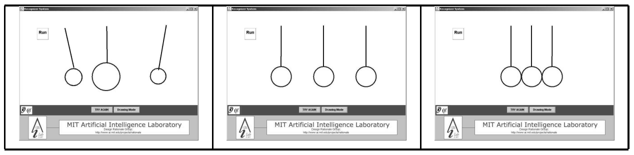


Figure 3: Three successive steps in our multimodal system. The first image shows the sketch before the user says anything. The second image shows the sketch after the user says “there are three identical equally spaced pendulums.” The third image shows the sketch after the user says that the pendulums are touching.

The vocabulary and sentences from the transcribed videos, augmented with a few additional words (e.g., plurals and numbers), were used to create a speech recognizer for the system. The speech understanding is provided by part of Galaxy[4], a speaker-independent speech understanding system that functions in a continuous recognition mode. The system allows users to talk without prior calibration of the system and without having to warn the system before each utterance. Both factors help create a natural user interface.

ASSIST was modified so that the sketch interpretations were combined with the speech recognition data, possibly resulting in a modified sketch. For example, for Newton's Cradle, functions were needed to space the pendulums equally and to make them identical. Changing the sketch required performing a simple translation from the descriptions, such as "equally spaced," to a set of manipulation commands that were implemented in ASSIST.

The system has a grammar framework that recognizes certain nouns and adjectives and thereby produces a modest level of generality. For instance, one noun it can recognize is "pendulum." The system needs to be told what a pendulum looks like, i.e., a rod connected to a circular body, so that it can link the user's intentions (e.g., drawing three identical pendulums) to a modification of the sketch. Adjectives it can recognize include numbers and words like "identical" and "touching." Adjectives are modifications to be made to the sketch (e.g., "touching"). The framework is general enough to allow the system to be extended to work with more examples.

6. RELATED WORK

ASSISTANCE[6] was a previous effort in our group to combine speech and sketching. It built on ASSIST by letting the user describe the behavior of the mechanical device with additional sketching and voice input. Our new system lets the users simultaneously talk in an unconstrained manner and sketch, which produces a more natural interaction.

QuickSet[7] is a collaborative multimodal system built on an agent-based architecture. The user can create and position items on a map using voice and pen-based gestures. For example, a user could say "medical company facing this way <draws arrow>." QuickSet is more command-based, targeted toward improving efficiency in a military environment. This differs from our goal of creating the most natural user interface possible. In contrast to our system where the user starts with a blank screen, QuickSet is a map-based system and the user starts with a map to refer to. Like our system, QuickSet uses a continuous speaker-independent speech recognition system.

AT&T Labs has developed MATCH[5], which provides a speech and pen interface to restaurant and subway information for New York City. This program uses a finite-state device and lets users make simple queries. This tool provides some multimodal dialogue capabilities, but it is not a sketching system and has only text recognition and basic circling and pointing gestures for the graphical input modality.

There are several other related projects[3, 7] that involve sketching and speech, but they are focused more on a command-based interaction with the user. In our system, speech augments the sketching; in other systems, the speech is necessary to the interaction.

7. FUTURE WORK

Speech will allow the system to capture information that is not currently available with only the sketching interface. Speech is a rich input modality and more information, such as numerical references, can be extracted from it to aid in the disambiguation of the inputs. Future work will attempt to make it easier to add new objects and commands to the system. We also want to evaluate how people actually talk when presented with a working system of this type. Other input modalities, such as gesture, could also help disambiguate the sketches and correctly simulate the user's designs.

8. ACKNOWLEDGMENTS

This project was funded by MIT's Project Oxygen, iCampus, and Intel Corporation.

9. REFERENCES

[1] A. Adler. Segmentation and Alignment of Speech and Sketching in a Design Environment. Master's Thesis, Massachusetts Institute of Technology, 2003.

[2] C. Alvarado and R. Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1365–1374, 2001.

[3] K. D. Forbus, R. W. Ferguson, and J. M. Usher. Towards a computational model of sketching. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, pages 77–83. ACM Press, 2001.

[4] T. J. Hazen, S. Seneff, and J. Polifroni. Recognition confidence scoring and its use in speech understanding systems. *Computer Speech and Language*, 16:49–67, 2002.

[5] M. Johnston, S. Bangalore, G. Vasireddy, A. Stent, P. Ehlen, M. Walker, S. Whittaker, and P. Maloor. MATCH: An architecture for multimodal dialogue systems. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 376–383, 2002.

[6] M. Oltmans. Understanding Naturally Conveyed Explanations of Device Behavior. Master's Thesis, Massachusetts Institute of Technology, 2001.

[7] S. Oviatt, P. Cohen, L. Wu, J. Vergo, L. Duncan, B. Suhm, J. Bers, T. Holzman, T. Winograd, J. Landay, J. Larson, and D. Ferro. Designing the user interface for multimodal speech and pen-based gesture applications: State-of-the-art systems and future research directions. *Human Computer Interaction*, 15(4):263–322, 2000.

Visual and Linguistic Information in Gesture Classification

Jacob Eisenstein and Randall Davis
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
32 Vassar St, Cambridge, MA 02139 USA
jacobe@csail.mit.edu
davis@csail.mit.edu

ABSTRACT

Classification of natural hand gestures is usually approached by applying pattern recognition to the movements of the hand. However, the gesture categories most frequently cited in the psychology literature are fundamentally multimodal; the definitions make reference to the surrounding linguistic context. We address the question of whether gestures are naturally multimodal, or whether they can be classified from hand-movement data alone. First, we describe an empirical study showing that the removal of auditory information significantly impairs the ability of human raters to classify gestures. Then we present an automatic gesture classification system based solely on an n-gram model of linguistic context; the system is intended to supplement a visual classifier, but achieves 66% accuracy on a three-class classification problem on its own. This represents higher accuracy than human raters achieve when presented with the same information.

Categories and Subject Descriptors

H.1.2 [**User-Machine Systems**]: Human information processing; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Artifical, augmented, and virtual realities*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation/methodology, Natural language, Theory and methods, Voice I/O*

General Terms

Human Factors, Reliability, Experimentation

Keywords

Gesture Recognition, Gesture Taxonomies, Multimodal Disambiguation, Validity

1. INTRODUCTION

A number of multimodal user interfaces afford interaction through the use of communicative free hand gestures [6, 7, 9, 11]. Since hand gestures can be used for a number of different communicative purposes–e.g., pointing at an object to indicate reference, or tracing a path of motion–*classification* of hand gestures is an important problem.

One class of systems focuses on artificial gestures, such as waving, closed fist, or "thumbs up" (e.g., [6]). These are not intended to correspond to the natural gestures that spontaneously arise during speech. For such systems, the goal is to maximize ease and speed of recognition, rather than the naturalness of the user interface. With such artificial gestures, gesture classes are distinguished purely on the basis of the dynamics of hand motion; however, mutual disambiguation with speech [16] could be used to improve recognition.

There is, however, a growing set of user interfaces that attempt to allow users to communicate using more natural gestures [7, 9, 11]. Here too, gesture classification has been taken to be primarily a problem for computer vision [9] or pattern recognition applied to glove input devices [7, 11]. Mutual disambiguation has been applied to improve recognition by constraining the gesture recognition candidates based on a set of possible semantic frames [7]. But the idea that gesture classes themselves are fundamentally multimodal entities – *defined* not only by the hand motion but also by the role of gesture within the linguistic context – has not yet been given full consideration.

We begin with a brief summary of the most frequently cited gesture taxonomy from the psychology literature; there has been some work on automatic classification for subsets of this taxonomy. Next, we present an empirical study of the ability of naïve raters to classify gestures according to this taxonomy, evaluating the effect of removing either the visual or auditory modalities. Then we present a gesture classification system that uses only the linguistic context; no hand-movement information is used.

2. TYPES OF GESTURES

Kendon describes a spectrum of gesturing behavior [8]. On one end are artificial and highly structured gestural languages, such as American Sign Language. In the middle, there are artificial but culturally shared *emblems*, such as the "thumbs-up" sign. At the far end is *gesticulation*, gestures that naturally and unconsciously co-occur with speech. Gesticulation is of particular interest for HCI since it is completely natural; speakers do not need to be taught how to do it. However, gesticulation is challenging because of the potential for variety in gesturing behavior across speakers, particularly across cultures.

Linguists have created a taxonomy of gesticulation, and gestures that naturally co-occur with speech are now typically divided into several classes: deictic, iconic, metaphoric, beat [13]. McNeill notes that these types should not be thought of as discrete, mutually exclusive bins, but rather, as features that may be present in varying degrees, possibly in combination. Thus, identification of the extent to which each feature is present would be the ultimate goal, rather than gesture classification. For the moment, however, implemented systems have focused on classification [3, 7, 9, 11].

The following definitions are quoted and summarized from Cassell [2].

- "**Deictics** spatialize, or locate in physical space..." Deictics can refer to actual physical entities and locations, or to spaces that have previously been marked as relating to some idea or concept.

- "**Iconic** gestures depict by the form of the gesture some features of the action or event being described." For example, a speaker might say "we were speeding all over town," while tracing an erratic path of motion with one hand.

- "**Metaphoric** gestures are also representational, but the concept they represent has no physical form; instead the form of the gesture comes from a common metaphor." For example, a speaker might say, "it happened over and over again," while repeatedly tracing a circle.

- "**Beat** gestures are small baton-like movements that do not change in form with the content of the accompanying speech. They serve a pragmatic function, occurring with comments on one's own linguistic contribution, speech repairs and reported speech." Speakers that emphasize important points with a downward motion of the hand are utilizing beat gestures.

2.1 Vision and Speech

One thing to notice about the definitions of the gesture types is that they are *linguistic* in nature. That is, gesture types are defined in terms of the role they play in the discourse, rather than in terms of a specific hand trajectory or class of trajectories. Indeed, researchers have found that there is no canonical set of hand trajectories that define each gesture class. For example, Cassell states, "Deictics do not have to be pointing index fingers." [2] For non-deictic gestures, it is even harder to characterize a "typical" set of hand shapes or trajectories; there are perhaps an infinite variety of possible iconic and metaphoric gestures [18]. Clearly, some amount of linguistic evidence – prosodic, lexical, or semantic – is necessary to classify gestures.

The remainder of this paper will seek to answer two questions.

1. To what extent does our perception of gesture types depend on a visual analysis of the hand motion, and to what extent does linguistic evidence come into play?

2. Can we build an accurate gesture classification system using linguistic data? What linguistic features are most informative for this purpose?

This paper describes two experiments aimed at answering these questions. In the first, naïve participants were trained to classify gestures according to the taxonomy described above. We assessed the level of interrater agreement to show that the taxonomy presents meaningful categories. We then removed the auditory and visual modalities separately, and found that participants make significantly different ratings in the absence of either modality. In other words, neither modality alone is sufficient to classify gestures.

Next we describe a gesture classification system that considers only the text surrounding the gesture. This system is trained using the majority classifications from the human raters as ground truth. Our classifier achieves a 66% agreement on a cross-validated evaluation; this is higher than the human coders achieved when they were denied access to the visual modality.

3. CLASSIFICATION BY HUMAN RATERS

In a previous study, nine speakers were videotaped while describing the behavior of three different mechanical devices [5]. These monologues were transcribed and the gesture phrases were segmented by the experimenter. Speakers ranged in age from 22 to 28; eight were native English speakers; four were women. The devices they described were: a latchbox, a piston, and a pinball machine. None of the participants had any special expertise in physics or mechanical engineering.

A second group of participants was then asked to classify the gestures from this corpus of videos, using the categorization scheme described above. There were four types of conditions: both video and audio (VA) were available, video only (V), audio only (A), and a textual transcription of the audio with no video (T). The VA condition was presented twice, and the ordering of conditions was identical for all participants: VA, V, A, T, VA.

A permutation of the videos was used so that no participant saw the same video in more than one condition, and so that each video was used in each condition nearly an equal number of times. The ninth video, of a male native English speaker, was used for training examples, as discussed below. Only videos of the explanations of the piston device were used. Overall, each video was annotated by eight or nine different participants in the VA condition, and by four or five participants in every other condition.

In each condition, participants were required to classify every gesture in the video. The videos ranged in length from 10 to 90 seconds, and included as few as four and as many as 53 distinct gesture phrases.

The entire study was performed using automated software that required no intervention from the experimenter. Participants were able to play each gesture segment from the video whenever and as frequently as they desired. Radio buttons were used to indicate the gesture classes in a fixed order, and were not preset to any value; participants were required to classify each gesture before moving on to the next condition. The video was presented in a separate window, 300 by 400 pixels in size. Each video segment ran from the beginning to the end of the gesture phrase, as segmented by the experimenter. In the audio-only condition, a beep was used

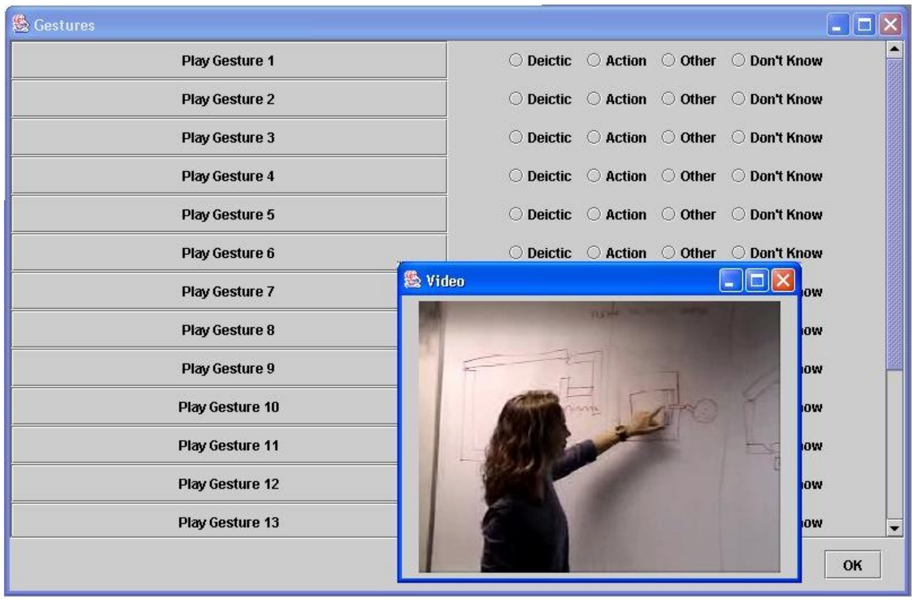


Figure 1: The experimental user interface for the VA, V, and A conditions

to indicate the onset of the stroke phase of the gesture. The user interface for the experimental tool is shown in Figure 1.

A different user interface was used for the text-only condition (Figure 2). Participants were presented with a list of the gestures (at left), while the center of the screen presented a transcript of all of the text used in a 4 second interval surrounding the onset of the stroke phase of the gesture. The location of the onset of the stroke phase was indicated in the transcript as "[GESTURE]". Radio buttons were once again used for the gesture classification.

3.1 Participants

There were 36 participants in this study; 22 men and 14 women. They ranged in age from 18 to 57, with a median of 26 and a mean of 29.3. Ten of the participants self-reported their English as being worse than that of a native speaker. Participants were recruited using posters placed around a university campus, and were compensated with free movie passes for completing the study. None of the participants had any prior experience with gestural or linguistic analysis, and all can be considered "novice" annotators. One participant was excluded because the experimental software crashed.

3.2 Instructions

Text and video examples were used to instruct participants about the gesture classification scheme. The instructions described both the kinetic and verbal components of each gesture class. The label "Action" was used in place of "Iconic", since pilot participants found the latter term to be confusing. Similarly, the label "Other" was used to capture "Beat" gestures, as well as any additional gestures that the listener felt did not belong to either of the other two categories. As reported in [5], metaphoric gestures are extremely infrequent in this corpus. A subset of participants were also allowed to classify gestures as "Unknown."

The written instructions given to participants can be found in the appendix.

4. RESULTS

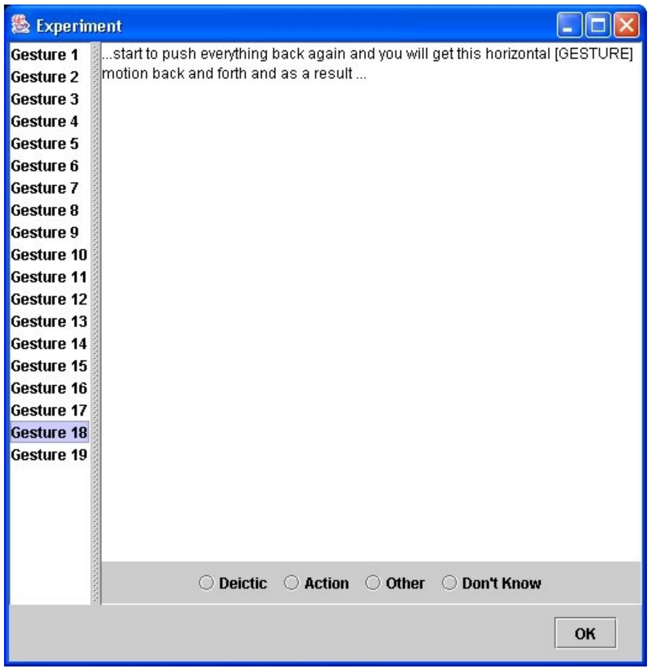
The standard Kappa (κ) metric was used to assess interrater reliability [1]. In the Kappa statistic, a value of zero indicates chance agreement, and a value of one indicates perfect agreement.

A confusion matrix for the second iteration of the video-audio (VA) condition is shown in Table 1. For each condition, a confusion matrix is generated for every pair of raters, and these confusion matrices are then averaged together. Given two raters r_1 and r_2, both pairs $\langle r_1, r_2\rangle$, and $\langle r_2, r_1\rangle$ will be included in the average, so the resulting matrix is necessarily symmetric.

The table indicates reasonable agreement for the deictic and action categories: $\kappa = .581$ when isolating the submatrix containing only these categories. However, the labeling of the "other" category is essentially random, lowering the overall Kappa to .449 when this category is included. It is possible to compute the variance of the Kappa statistic; in this case, $\sigma = .033$, yielding better than chance agreement at $p < .01$.

The relatively low Kappa here may reflect McNeill's contention that the gesture types are not truly mutually exclusive. Another possible factor is the limited training for these participants, which typically lasted less than five minutes (see the Appendix for the raters' instructions). Interrater agreement was significantly higher in the second iteration of the VA condition than in the first iteration, where $\kappa = .273$. This suggests that the raters' assessments of the meaning of the gesture categories converged as they gained experience with the rating task. For expert raters, Nakano reports Kappa agreement of .81 using similar categories [15].

The extremely low agreement on the "other" category suggests that some raters may have used "other" whenever they were unable to classify the gesture as either "deictic" or



...start to push everything back again and you will get this horizontal [GESTURE] motion back and forth and as a result ...

Figure 2: The experimental user interface for the text-only condition

	deictic	action	other	unknown
deictic	.270	.069	.060	.017
action	.069	.249	.032	.009
other	.060	.032	.079	.015
unknown	.017	.014	.015	.004

Table 1: Confusion matrix for the second VA condition

"action." Note that this confusion matrix includes results from the sixteen participants who did not have access to the "don't know" option, as well as those who did have access to this option. The "don't know" option increased the Kappa marginally, to 0.451, but this difference is not significant ($p > .05$).

4.1 Conditions

The agreement for the audio-only (A) condition was significantly lower than the VA condition, $\kappa = .337, p < .01$. The same is true of the video-only (V) condition, $\kappa = .276$, $p < .01$, and the text condition (T), $\kappa = .315, p < .01$. However, in all cases, the Kappa value was better than chance, $p < .01$.

It may be somewhat surprising that interrater agreement was lower in the impaired conditions. One conceivable source of disagreement in the VA condition is the choice of which modality to favor when each suggests a different classification. In the impaired conditions, no such choice need be made, so one might predict that agreement within the impaired conditions would be higher. But in fact, the opposite is the case – intra-condition agreement increases when both modalities are available. This suggests that the modalities usually provide complementary cues, and that in many cases, neither modality provides enough information on its own.

Condition	Intra-condition agreement (κ)	Agreement with VA majority
VA	.451	78%
V	.276	59%
A	.335	45%
T	.315	41%

Table 2: Agreement results for each condition

We computed the majority vote classifications for each video in the second VA condition, and took this as ground truth. Then for each condition, we computed the average percentage agreement between ground truth and each rater's annotations. As an upper bound, in the VA condition, the average rater agreed with the majority annotations at a level of 78%, $\sigma = 0.018$. In the audio-only condition (A), the average agreement with the modal classifications from the VA condition is 45%, $\sigma = 0.017$. In the video-only condition (V), the average agreement is better, at 59%, $\sigma = 0.021$. In the text-only condition (T), the average agreement is 41%, $\sigma = 0.016$.

Since the video-only condition had the highest level of agreement with the VA condition, this would suggest that visual information is the primary cue for gesture classification. However, there is a statistically significant drop-off from the VA condition to the video-only condition ($p < .01$), suggesting that audio cues do play a necessary supplementary role.

5. AUTOMATIC CLASSIFICATION FROM TEXT

The previous section shows that human listeners use both vision and audition when recognizing gestures, and that two modalities contain complementary information. In this section, we explore the idea of classifying gestures using only linguistic information. The goal here is to determine what type of linguistic cues are most useful for gesture classification, to get a sense for the classification performance these cues can provide, and to develop a system that could be combined with a vision-based approach in an integrated multimodal gesture classifier. We use the majority classifications from the previous study as ground truth, and evaluate our system's ability to replicate these classifications using only textual information.

5.1 Features

For each gesture, a feature vector was constructed using the words that appear within a series of windows surrounding the onset of the stroke phase of the gesture. According to the psychology literature, the stroke phase usually overlaps the most prosodically prominent part of the associated speech [13]. We used two windows to differentiate words that appear during the stroke phase from words that appear at any point during the whole gesture phrase (see Figure 3). The windows were buffered by 133 milliseconds at the front and 83 milliseconds at the back. Ideally, these parameters should be estimated by cross-validation, but the results are not overly sensitive to their settings.

Since strokes are a component of gesture phrases, the

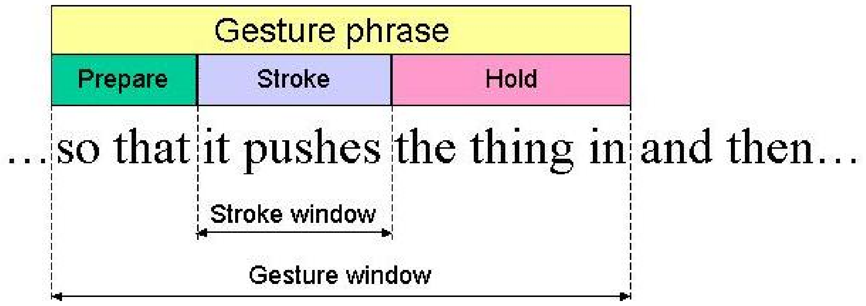


Figure 3: Separate windows are used to capture stroke and gesture phrase features

stroke window is a subset of the gesture phrase window. By including the stroke window, we are heeding McNeill's advice that the words overlapping the stroke phase are the most important for determining the semantic content of the multimodal utterance [13]. This did in fact improve performance; from 61.5% using the only the gesture phrase window, to 65.9% when using both windows. Using the stroke phase window alone produced performance of 58.7%; the multiple-window technique was significantly better than both alternatives.

The stroke window contained n-grams that were highly informative but sparse. For example, consider the part-of-speech unigram "VBZ", indicating a verb in the 3rd person singular, present tense. This feature is somewhat informative when appears in a gesture phrase window:

$$p(\text{VBZ} \in \text{GP window} \mid \text{Deictic}) = .38 \quad (1)$$
$$p(\text{VBZ} \in \text{GP window} \mid \text{Iconic}) = .52 \quad (2)$$

This feature is more informative if it appears in the stroke window:

$$p(\text{VBZ} \in \text{Stroke window} \mid \text{Deictic}) = .21 \quad (3)$$
$$p(\text{VBZ} \in \text{Stroke window} \mid \text{Iconic}) = .44 \quad (4)$$

Put another way, if the VBZ feature appears during the gesture phrase window of an iconic gesture, it is almost always during the stroke phase. For deictic gestures, it could appear with equal likelihood anywhere throughout the gesture phrase.

5.1.1 Linguistic Analysis

Each word was stemmed, using a lexically-based stemmer, and tagged, using a Java implementation of the Brill tagger [12]. Stemming had no appreciable affect on performance. Each word stem was included as a feature. We also tried some coarse word-sense disambiguation by appending the part-of-speech tag to each word, and including each type of usage as an independent feature (e.g., "fish/NN" and "fish/VB") – this decreased performance from 65.9% to 64.2%. POS tags were used as features on their own; without them, performance decreased to 58.6%. Both differences were significant.

For both words and POS tags, n-grams of size 1 to 3 were used. All n-grams were simply thrown into the feature vector together; in the future we may use backoff models to combine the different size n-grams in a more intelligent way. Unigrams alone provided a performance of 55.1%; adding bigrams improved performance to 60.0%; adding trigrams improved performance to 65.9%; adding 4-grams decreased performance to 65.7%, an insignificant change (all other changes were significant).The mean number of words in each gesture phrase window was 5.0 (median = 4, σ = 3.7), and the mean for the stroke window was 2.8 (median = 2, σ = 2.0). Thus it is unsurprising that larger n-grams afforded no improvement. In total, when using unigrams, bigrams, and trigrams, there were 2746 features.

5.2 Classifier Performance

Table 3 compares the performance of various classifiers on this task. For all classifiers except TWCNB, the Weka [19] implementation was used.

HyperPipes is a simple, fast classifier for situations with a large number of attributes (there are 2746 in this case). HyperPipes records the attribute bounds for each category, and then classifies each test instance according to the category that most contains the instance. As shown in the table, HyperPipes significantly outperforms all other classifiers on this task.

TWCNB is a modification of the Naive Bayes classifier designed by Rennie et. al [17] to better suit text-classification problems. It includes a complement-class formulation which is useful when the number of examples is poorly balanced across classes, as is the case here. It also implements term-frequency transformations, addressing the fact that the multinomial distribution is a poor model of text. Our own implementation of this classifier is used in these experiments.

The NaiveBayes, SVM, and C4.5 classifiers are used "as is" from the Weka library; default settings are used for all parameters. While any one of these classifiers might perform substantially better given an optimal choice of parameters, our purpose is to show the range of performance on this task achieved by some commonly-used techniques, rather than to offer a comprehensive comparison of classifiers.

Table 3 compares the performance of each classifier on the gesture classification task. The results were the average of one hundred experiments, each of which involved randomizing the dataset and then performing a stratified ten-fold cross-validation. All classification accuracy differences were significant, except for SVM versus C4.5, where the difference was not significant.

The "always deictic" classifier chooses the "deictic" class every time. All classifiers significantly outperformed this

	Accuracy	σ
HyperPipes	65.9%	1.47
TWCNB	63.5%	1.66
Naive Bayes	58.9%	1.10
C4.5	56.0%	2.17
SVM	55.9%	2.17
Always deictic	48.7%	N/A
Humans: audio-only	45%	2.7
Humans: audio-video	78%	2.8

Table 3: Comparison of classifier performance, averaged over 100 stratified, ten-fold cross-validation experiments

baseline. Another baseline is the performance of human raters who had access to the same information, the audio surrounding the gesture. The performance of human raters in the audio-only condition was actually worse than the "always deictic" baseline. This suggests that while the linguistic context surrounding the gesture clearly does provide cues for classification, human raters were unable to use these cues in any meaningful way when the video was not also present.

As an upper bound, we consider the performance of the human raters who had access to both the audio and video; the majority opinion of these raters forms the ground truth for this experiment. As shown in the table, the average rater agreed with the majority 78% of the time. This appears to be a reasonable upper bound for a multimodal gesture classification system; it seems unlikely that using the text only, we could achieve higher performance than human raters who had access to visual and prosodic information.

5.3 Discussion

Table 4 lists the ten features that were found to be carry the highest information gain. Capital letters indicate part-of-speech tags, which are defined according to the Penn Treebank set. "UH" indicates an interjection, e.g., "um", "ah", "uh"; "VB*" is a verb, with the last character indicating case and tense; "PRP" is a personal pronoun.

The features correlate with gesture categories in a way that accords well with linguistic theory about the role of speech and gesture as part of an integrated communicative system [13]. For deictics, the word "here" is a good predictor, since it is typically accompanied by a gestural reference to a location in space. The class of "other" gestures is primarily composed of beats, which serve the same turn-keeping function as interjections such as "uh." The "VBZ" tag – indicating a verb in the third-person singular – is a good predictor of iconic gestures, as are the more domain-specific cue words, "back" and "push." These words were used by several speakers to describe the motion of the piston, and were typically accompanied by an iconic gesture describing that motion.

6. RELATED WORK

For a more detailed discussion of the gesture classes described in this paper, see [13]; for an analysis specifically geared towards multimodal user interfaces, see [2].

Computational analysis of unconstrained, natural gesture is relatively unexplored territory, but one exception is the research of Quek and Xiong et al. They have applied McNeill's catchment model [14] to completely unconstrained dialogues, extracting discourse structure information from a number of different hand movement cues, such as gestural oscillations [20].

Pattern-recognition approaches to recognizing some of these gesture classes have been reported in a few publications. Kaiser et al. [7] describe a system that recognizes deictic pointing gestures and a set of manipulative gestures: point, push, and twist. Kettebekov and Sharma [9] present a map-control user interface that distinguishes between deixis and "motion" gestures that are a subset of the class of iconic gestures in the taxonomy that we have used. Kettebekov, Yeasin, and Sharma also applied prosodic information to improve gesture segmentation and the recognition of movement phrases and various types of deictic gestures [10].

Perhaps the most closely related research topic is mutual disambiguation [16], which views speech and gesture as co-expressive streams of evidence for the underlying semantics. If the speech modality suggests a given semantic frame with very high probability, then the probabilities on gestures that are appropriate to that frame are increased; the converse is also possible, with gesture disambiguating speech. While most of the work on mutual disambiguation involves pen/speech interfaces [4], it has more recently been applied to free hand gestures as well [7].

Mutual disambiguation relies on having a constrained domain in which the semantics for every utterance can be understood within the context of a formal model of the topic of discourse. Our approach gives up some of the power of mutual disambiguation, in that semantic information may provide tighter constraints on gesture than the linguistic cues that we use. Our approach is more appropriate to situations in which a formal model of the domain is not available.

7. FUTURE WORK

The ultimate goal of this research is multimodal gesture recognition: a combination of linguistic priors of gesture classes with vision-based recognition. Consequently, the most pressing future work is to combine the textual classifier developed here with traditional pattern-recognition techniques. Hopefully this will show that linguistic context does indeed improve classification performance, as it does for humans.

In addition, there are a number of other ways in which both the empirical study and the automatic classifier can be extended.

7.1 Prosodic versus lexical cues

The experiment involving human raters showed that auditory cues significantly improve visual classification of gestures. However, this experiment does not disambiguate the role of prosody versus lexical and higher-order linguistic features. We can remove prosody by transcribing the speech and feeding it to a text-to-speech engine. If the results using this audio and the original video are indistinguishable from the video-audio condition with human speech, then we could conclude that prosody plays no role in gesture classification. Alternatively, we can remove lexical and higher-order linguistic cues by having speakers communicate in a language unknown to the listeners, but with similar prosodic conventions. If the results prove to be indistinguishable from the video-audio condition in which the listener understands the

Feature	**Window**	**Information**	$p(w\|$ Deictic)	$p(w\|$ Iconic)	$p(w\|$ Other)
back	phrase	0.088	.013	.17	.04
UH	stroke	0.064	.051	.017	.24
push	stroke	0.058	.013	.12	.04
VBZ	stroke	0.056	.21	.44	.16
back	stroke	0.055	.026	.15	.04
here	phrase	0.054	.23	.051	.24
as	phrase	0.053	.064	.20	.04
uh	stroke	0.051	.039	.017	.20
as	stroke	0.044	.039	.15	.04
PRP-VBP	phrase	0.044	.12	.017	.08

Table 4: The top ten features by information gain

speaker, then lexical and higher-order linguistic cues are irrelevant to gesture classification.

7.2 Domain generality

All of the test and training data in this corpus is drawn from an experiment within a single domain: engineering mechanical devices. Another experiment could help determine whether the language model learned here is general beyond that domain. The absence of obviously domain-specific terms in the set of more informative features described in the previous section is encouraging.

7.3 Recognized speech and gesture boundaries

The current evaluation is performed using transcriptions, rather than automatically recognized speech. Thus, this system does not have to deal with word errors. In the future, we hope to demonstrate that this classifier is still accurate, even when presented with errorful speech. In addition, we would like to segment gestures automatically, possibly with the aid of prosodic cues as in [10].

7.4 Feature fusion

The classification system as implemented uses classes of features varying on several dimensions: gesture phrase window versus stroke window; word versus part of speech tag; n-gram size. Currently, all features are combined into a single vector and sent to a classifier. A more sophisticated approach might be to interpolate between multiple classifiers and use backoff models to combine the different size n-grams.

8. CONCLUSIONS

Natural, communicative gesture is well described by gesture classes that are fundamentally multimodal in nature, pertaining to both the hand motion and the role played by the gesture in the surrounding linguistic context. Humans rely on auditory as well as visual cues to classify gestures; without auditory cues, performance decreases significantly. This suggests that automatic classification of gestures should make use of both hand movement trajectories and linguistic cues. We have developed a gesture classifier that uses only linguistic features and achieves 66% accuracy on a corpus of unconstrained, communicative gestures.

Acknowledgements

We thank Aaron Adler, Christine Alvarado, Sonya Cates, Tracy Hammond, Michael Oltmans, Sharon Oviatt, Metin Sezgin, Vineet Sinha, and Özlem Uzuner for their helpful suggestions about this work. We also thank Jason Rennie and Kai Shih for their help with the TWCNB classifier. This research is supported by the Intel Corporation, the Microsoft iCampus project and the sponsors of MIT Project Oxygen.

9. REFERENCES

[1] Carletta, J. Assessing agreement on classification tasks: the kappa statistic. *Computational Linguistics 22*, 2 (1996), 249–254.

[2] Cassell, J. A framework for gesture generation and interpretation. *Computer Vision in Human-Machine Interaction.* Cambridge University Press (1998), 191–215.

[3] Cassell, J., Vilhjalmsson, H., and Bickmore, T. Beat: the behavior expression animation toolkit. *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques.* ACM Press (2001), 477–486.

[4] Cohen, P. R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., Chen, L., and Clow, J. Quickset: Multimodal interaction for distributed applications. *ACM Multimedia'97.* ACM Press (1997), 31–40.

[5] Eisenstein, J., and Davis, R. Natural gesture in descriptive monologues. *UIST'03 Supplemental Proceedings.* ACM Press (2003), 69–70.

[6] Freeman, W. T., and Weissman, C. Television control by hand gestures. *International Workshop on Automatic Face- and Gesture- Recognition.* IEEE Press (1995), M. Bichsel, Ed., 179–183.

[7] Kaiser, E., Olwal, A., McGee, D., Benko, H., Corradini, A., Li, X., Cohen, P., and Feiner, S. Mutual disambiguation of 3d multimodal interaction in augmented and virtual reality. *Proceedings of the Fifth International Conference on Multimodal Interfaces.* ACM Press (2003), 12–19.

[8] Kendon, A. *Conducting Interaction.* Cambridge University Press, 1990.

[9] Kettebekov, S., and Sharma, R. Toward natural gesture/speech control of a large display. *Engineering for Human-Computer Interaction (EHCI'01). Lecture Notes in Computer Science.* Springer Verlag (2001).

[10] Kettebekov, S., Yeasin, M., and Sharma, R. Prosody based co-analysis for continuous recognition of coverbal gestures. *Proceedings of the Fourth International Conference on Multimodal Interfaces (ICMI'02).* IEEE Press (Pittsburgh, USA, 2002), 161–166.

[11] Koons, D. B., Sparrell, C. J., and Thorisson, K. R. Integrating simultaneous input from speech, gaze, and hand gestures. *Intelligent Multimedia Interfaces.* AAAI Press (1993), 257–276.

[12] Liu, H. Montylingua v1.3.1: An end-to-end natural language processor of english for python/java, 2003.

[13] McNeill, D. *Hand and Mind*. The University of Chicago Press, 1992.

[14] McNeill, D., Quek, F., McCullough, K.-E., Duncan, S., Furuyama, N., Bryll, R., Ma, X.-F., and Ansari, R. Catchments, prosody, and discourse. *Gesture 1* (2001), 9–33.

[15] Nakano, Y. I., Okamoto, M., Kawahara, D., Li, Q., and Nishida, T. Converting a text into agent animations: Assigning gestures to a text. *HLT-NAACL 2004: Companion Volume.* ACL Press (2004), 153–156.

[16] Oviatt, S. L. Mutual disambiguation of recognition errors in a multimodel architecture. *Human Factors in Computing Systems (CHI'99).* ACM Press (1999), 576–583.

[17] Rennie, J. D. M., Shih, L., Teevan, J., and Karger, D. R. Tackling the poor assumptions of naive bayes text classifiers. *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003).* AAAI Press (2003).

[18] Sparrell, C. Coverbal iconic gesture in human-computer interaction. Master's thesis, Massachusetts Institute of Technology, 1993.

[19] Witten, I. H., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[20] Xiong, Y., Quek, F., and McNeill, D. Hand motion gestural oscillations and multimodal discourse. *Fifth International Conference on Multimodal Interfaces (ICMI'03).* IEEE Press (2003), 132–139.

Appendix: Instructions for Raters

In this study you will be asked to identify gestures as belonging to one of three classes: deictic, action, or other.

DEICTIC gestures involve pointing at, tracing the outline of, or otherwise indicating a specific object or region of space. For example, a speaker might point at a book and say, "this is the book I read last week." Drag the mouse over the squares below to see short video clips of deictic gestures.

ACTION gestures reenact a physical interaction, trajectory of motion, or some other event. For example, a speaker might describe a bouncing pinball by tracing a path of motion with the hand while saying, "the ball bounces all over the place." Drag the mouse over the square below to see a short video clip of an action gesture.

OTHER gestures include the gesticulation that typically accompanies speech (e.g., creating visual "beats" to emphasize important speaking points) as well as any other gesture that cannot easily be classified in either of the above two categories. Drag the mouse over the square below to see a short video clip of an "other" gesture.

First, you will be presented with a video, and a user-interface window that allows you to play specific clips from the video. Each clip includes a single gesture, which you will be asked to classify using the above framework. You will be presented with four such videos; at times, the audio may be muted, or the video itself may be hidden. Based on whatever information is available, please make your best effort to correctly classify each gesture. Even if you feel that you do not have enough information to correctly classify a gesture, please make your best guess.

Next, you will be presented with a set of textual transcriptions of the speech surrounding each gesture. Based on this text, please make your best effort to correctly classify each gesture.

In Proceedings of IJCAI-01

Resolving Ambiguities to Create a Natural Computer-Based Sketching Environment

Christine Alvarado, Randall Davis
MIT Artificial Intelligence Laboratory

Abstract

Current computer-based design tools for mechanical engineers are not tailored to the early stages of design. Most designs start as pencil and paper sketches, and are entered into CAD systems only when nearly complete. Our goal is to create a kind of "magic paper" capable of bridging the gap between these two stages. We want to create a computer-based sketching environment that feels as natural as sketching on paper, but unlike paper, understands a mechanical engineer's sketch as it is drawn. One important step toward realizing this goal is resolving ambiguities in the sketch— determining, for example, whether a circle is intended to indicate a wheel or a pin joint—and doing this as the user draws, so that it doesn't interfere with the design process. We present a method and an implemented program that does this for freehand sketches of simple 2-D mechanical devices.

1 Sketching Conceptual Designs

Engineers typically make several drawings in the course of a design, ranging from informal sketches to the formal manufacturing drawings created with drafting tools. Drawing is far more than an artifact of the design process; it has been shown to be essential at all stages of the design process [Ullman *et al.*, 1990]. Yet almost all early drawings are still done using pencil and paper. Only after a design is relatively stable do engineers take the time to use computer aided design or drafting tools, typically because existing tools are too difficult to use for the meager payoff they provide at this early stage.

Our aim is to allow designers to sketch just as they would on paper, e.g., without specifying in advance what component they are drawing, yet have the system understand what has been sketched. We want to have the input be as unconstrained as possible, in order to make interaction easy and natural; our route to accomplishing this is to build a sufficiently powerful sketch recognizer.

It is not yet obvious that a freehand sketching interface will be more effective in real use than a carefully designed menu-based system. In order to do the comparison experiments, however, we must first build powerful sketch-based systems. It is the construction of such a system that is the focus of this paper.

The value of sketching as an interface and the utility of intelligent sketch understanding has gained increasing attention in recent years (e.g., [Hearst, 1998]). Some early research was concerned with single stroke classification ([Rubine, 1991]), while more recent work ([Gross, 1995; Landay and Myers, 2001]) puts groups of strokes together to form larger components. A number of efforts (e.g., [Gross and Do, 1996], [Mankoff *et al.*, 2000]) have acknowledged the necessity of representing ambiguities that arise in interpreting strokes, but have not substantially addressed how to resolve those ambiguities.

Given the frequency of ambiguities in a sketch, a tool that constantly interrupts the designer to ask for a choice between multiple alternatives would be cumbersome. Our work is thus focused, in part, on creating a framework in which to both represent and use contextual (top-down) knowledge to resolve the ambiguities. We built a program called ASSIST (A Shrewd Sketch Interpretation and Simulation Tool) that interprets and understands a user's sketch as it is being drawn, providing a natural-feeling environment for mechanical engineering sketches.

The program has a number of interesting capabilities.

- The basic input to the program is a sketch, i.e., a sequence of strokes drawn "while the system watches," not a finished drawing to be interpreted only after it is complete.
- Sketch interpretation happens in real time, as the sketch is being created.
- The program allows the user to draw mechanical components just as on paper, i.e., as informal sketches, without having to pre-select icons or explicitly identify the components.
- The program uses a general architecture for both representing ambiguities and adding contextual knowledge to resolve the ambiguities.
- The program employs a variety of knowledge sources to resolve ambiguity, including knowledge of drawing style and of mechanical engineering design.
- The program understands the sketch, in the sense that it recognizes patterns of strokes as depicting particular

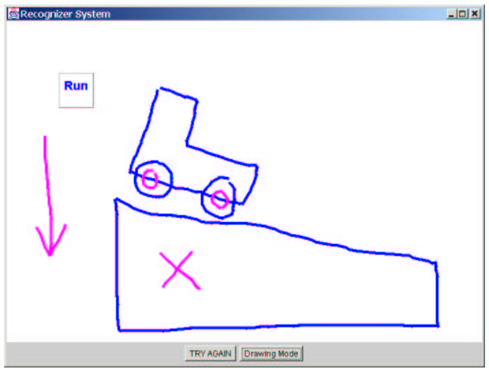


Figure 1: A car on a hill, as drawn by the user in ASSIST.

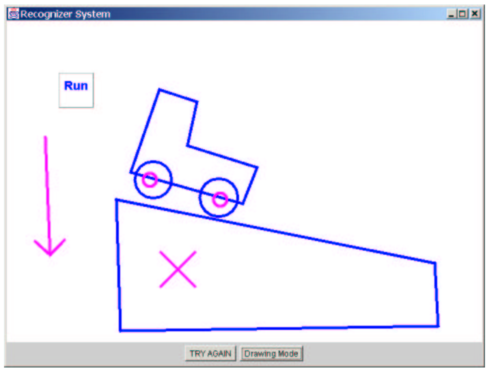


Figure 2: The sketch as displayed by ASSIST.

components, and illustrates its understanding by running a simulation of the device, giving designers a way to simulate their designs as they sketch them.

We describe the system and report on a pilot user study evaluating the naturalness of the program's interface and the effectiveness of its interpretations.

2 Designing with ASSIST

Figure 1 shows a session in which the user has drawn a simple car on a hill. The user might begin by drawing the body of the car, a free-form closed polygon. As the user completes the polygon, the system displays its interpretation by replacing the hand-drawn lines (shown in Figure 1) with straight blue lines. Next the user might add the wheels of the car, which also turn blue as they are recognized as circular bodies. The user can then "attach" the wheels with pin joints that connect wheels to the car body and allow them to rotate. The user might then draw a surface for the car to roll down, and anchor it to the background (the "x" indicates anchoring; anything not anchored can fall). Finally, the user can add gravity by drawing a downward pointing arrow not attached to any object. The user's drawing as re-displayed by ASSIST is shown in Figure 2.

The system recognizes the various components in the drawing by their form and context; when the "Run" button is tapped, it transfers the design to a two-dimensional mechanical simulator which shows what will happen (Figure 3).[1]

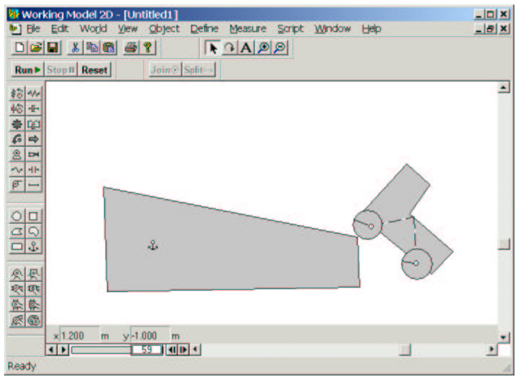

Working Model 2D - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run Stop Reset
Join Split
x 1.200 m y -1.000 m
Ready

Figure 3: The sketch simulated, showing the consequences.

Note that the user drew the device without using icons, menu commands, or other means of pre-specifying the components being drawn. Note, too, that there are ambiguities in the sketch, e.g., both the wheels of the car and pin joints are drawn using circles, yet the system was able to select the correct interpretation despite these ambiguities, by using the knowledge and techniques discussed below. The automatic disambiguation allowed the user to sketch without interruption.

Figure 4 shows a session in which the user has drawn a more interesting device, a circuit breaker, and run a simulation of its behavior.

Note that ASSIST deals only with recognizing the mechanical components in the drawing and is, purposely, literal-minded in doing so. Components are assembled just as the user drew them, and component parameters (e.g. spring constants, magnitudes of forces, etc) are set to default values. The car in Figures 1–3, for example, wobbles as it runs down the hill because the axles were not drawn in the center of the wheels. The combination of literal-minded interpretation and default parameter values can produce device behavior other than what the user had in mind. Other work in our group has explored the interesting and difficult problem of communicating and understanding the *intended* behavior of a device once it has been drawn using ASSIST [Oltmans, 2000].

3 Embedding Intelligent Assistance

We created a model for sketch understanding and ambiguity resolution inspired by the behavior of an informed human observer, one that recognizes the sketch by relying on both low-level (i.e., purely geometric) routines and domain specific knowledge.

One interesting behavior of an informed observer is that interpretation begins as soon as the designer begins sketching. While not a required strategy—people can obviously interpret a finished sketch—there are advantages in ease of use and in speed from having the program do its interpretation in parallel with drawing. Ease of use arises because the program

[1]We use Working Model 2D from Knowledge Revolution, a commercial mechanical simulator; any simulator with similar capabilities would do as well.

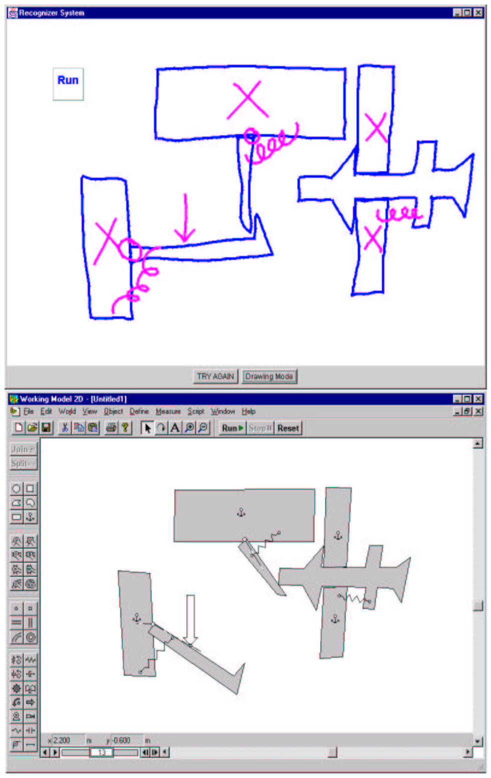

Recognizer System
Run
TRY AGAIN
Drawing Mode
Working Model 2D - [Untitled1]
File Edit World View Object Define Measure Script Window Help
Run
Stop
Reset

Figure 4: A sketch of a circuit breaker (left) and its simulation (right).

can provide an indication of its interpretation of parts of the sketch as soon as they are drawn, making it easier for the user to correct a misinterpretation. Interpretation is faster because incremental interpretation effects a divide and conquer strategy: parts of the drawing interpreted correctly can provide useful context when interpreting parts drawn subsequently.[2]

A second interesting behavior of an informed observer is the ability to accumulate multiple interpretations and defer commitment. Consider for example the objects in Figure 5. Are the strokes in 5a going to become part of a ball and socket mechanism (5b), or are they the beginning of a gear (5c)? Committing too soon to one interpretation precludes the other. Hence interpretation must be capable of revision in the face of new information.

There is clearly a need to balance out the desire for interpretation occurring in parallel with drawing, and the need to avoid premature commitment. We discuss below how our system accomplishes this.

Third, while commitment should be deferred, it must of course be made eventually, and determining when to make that commitment is not easy. Timing information can assist. Consider the case of circles: Because circles are low-level structures, it is likely that they will be used in higher-level structures, as for example when a circle turns out to be part of a pulley system. One way of dealing with this is to use timing

[2]The program also seems faster because it is working while the user is drawing, reducing the user's wait.

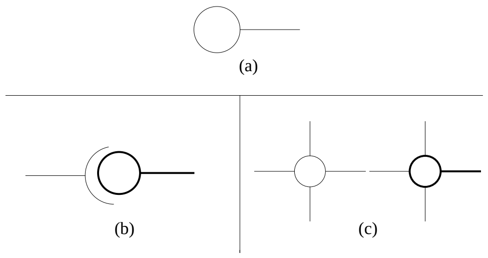


Figure 5: An example of ambiguity: The bold strokes in (b) and (c) are identical to the strokes in (a).

data: the system gets to "watch" the sketch being drawn and knows when each stroke was made. If, some time after the circle has been drawn, it has still not been used in any other structure, the observer can plausibly guess that it will not be incorporated into another piece and should be interpreted as an independent circular body.[3]

Finally, parts may remain ambiguous even when a piece of the drawing is finished. To resolve these residual ambiguities, the observer uses his knowledge of mechanical engineering components and how they combine. Consider, for example, the small circles inside the larger circles in Figure 2; ASSIST determines that these are more likely to be pivot joints than additional circular bodies, both because small circles typically indicate pin joints and because bodies do not typically overlap without some means of interconnection (i.e., the pin joint).

Our system incorporates each of these observations: it begins interpreting the sketch as soon as the user starts drawing; it accumulates multiple interpretations, deferring commitment until sufficient evidence (e.g., stroke timing) accumulates to suggest a component has been finished, and it resolves ambiguities by relying on knowledge from the domain about how components combine.

4 ASSIST's Interpretation and Disambiguation Process

ASSIST's overall control structure is a hierarchical template-matching process, implemented in a way that produces continual, incremental interpretation and re-evaluation as each new stroke is added to the sketch. Each new stroke triggers a three stage process of recognition, reasoning and resolution. Recognition generates all possible interpretations of the sketch in its current state, reasoning scores each interpretation, and resolution selects the current best consistent interpretation. After each pass through the three stages the system displays its current best interpretation by redrawing the sketch.

4.1 Recognition

In the recognition stage, ASSIST uses a body of recognizers, small routines that parse the sketch, accumulating all possible

[3]A *body* is any hunk of material not otherwise interpreted as a more specialized component (like a spring, pin joint, etc.). The car body is a polygonal body; its wheels are circular bodies.

interpretations as the user draws each stroke. A recognizer takes as input raw strokes and previously recognized objects, and if the input fits its template, produces a new object. For example, the circle recognizer reports a circle if all the points on a stroke lie at roughly the same distance from the average X and Y coordinate of the stroke.[4] The circle is then available to other recognizers, e.g., the pulley recognizer.

4.2 Reasoning

In the second stage the system scores each interpretation using a variety of different sources of knowledge that embody heuristics about how people draw and how mechanical parts combine.

Temporal Evidence

People tend to draw all of one object before moving to a new one. Our system considers interpretations that were drawn with consecutive strokes to be more likely than those drawn with non-consecutive strokes.

Additional evidence comes from "longevity:" the longer a figure stays unchanged, the stronger its interpretation becomes, because as time passes it becomes more likely that the figure is not going to be turned into anything else by additional strokes.

Simpler Is Better

We apply Occam's razor and prefer to fit the fewest parts possible to a given set of strokes. For example, any polygonal body (e.g., the car body in Figure 2) could have been interpreted as a set of (connected) individual rods, but the system prefers the interpretation "body" because it fits many strokes into a single interpretation.

More Specific is Better

Our system favors the most specific interpretation. Circles, for example, (currently) have three interpretations: circular bodies, pin joints, and the "select" editing gesture. The selection gesture is the most specific interpretation, in the sense that every circle can be a circular body or pin joint, but not every circle can be a selection gesture (e.g., if it does not encircle any objects). Hence when a circle contains objects inside of it, the system prefers to interpret it as a selection gesture.

Domain Knowledge

ASSIST uses basic knowledge about how mechanical components combine. For example, a small circle drawn on top of a body is more likely to be a pin joint than a circular body.

User Feedback

User feedback also supplies guidance. The "Try Again" button (see the bottom of Figure 1) permits the user to indicate that something was recognized incorrectly, at which point the system discards that interpretation and offers the user an ordered list of alternative interpretations. Conversely the system can be relatively sure an interpretation is correct if the user implicitly accepts it by continuing to draw.

[4]In other work we describe recognizers that use more sophisticated early processing of basic geometry [Sezgin,].

Combining Evidence

The heuristics described above all independently provide evidence concerning which interpretation is likely to be correct. Our method of combining these independent sources involves distinguishing between two categories of evidence: categorical and situational.

Categorical evidence ranks interpretations relative to one another based on the first four knowledge sources described above. Each source is implemented in the system as a set of rules that takes two interpretations as input, and outputs an ordering between them. In processing Figure 1, for example, the interpretation "body" is ranked higher than the interpretation "connected rods," based on the "Simpler is Better" heuristic.

Situational evidence comes from implicit and explicit feedback from the user. Explicit feedback is provided by use of the "Try Again" button; implicit feedback arises when the user keeps drawing after the system displays an interpretation, suggesting that the user is satisfied that the system has understood what has been drawn so far.

The system gives each interpretation two numeric scores, one from each category of evidence. The categorical score is an integer from 0 to 10; the situational score is an integer from -11 to 11. These values are chosen so that the situational dominates the categorical, because we want user feedback to dominate general ranking rules. An interpretation's total score is simply the sum of its two scores.

To convert categorical evidence to a numerical score (so it can be combined it with the situational score), we generate a total ordering of all the interpretations consistent with the partial orders imposed by the categorical evidence. We do a topological sort of the graph of partial orders produced by the evidence and distribute scores evenly, from 0 to 10, over all the interpretations in the sorted graph.[5]

Situational scores start out at 0 and are strengthened or weakened by evidence that can raise of lower the current value by 1 or by 11. Situational evidence thus either modifies an interpretation's value by a small amount (1 unit) or declares it to be certainly correct or certainly incorrect. The system declares an interpretation to be certainly correct or certainly incorrect when the user explicitly accepts or rejects the interpretation using the "Try Again" dialog box. The system strengthens an interpretation by a small amount each time strokes added by the user are consistent with that interpretation.[6]

We developed this approach to accumulating and combining evidence, and implemented our knowledge sources as a rule based system, in order to provide a degree of modularity

[5]The system first removes cycles in the graph by collapsing strongly connected components. Conceptually, this step indicates that the system will give an equal score to all interpretations that have inconsistent ordering given the evidence (i.e., one rule says A is more likely than B, while another says B is more likely than A). In addition, if there are more than 11 interpretations, the top ten are assigned scores of 10 through 1; the remaining interpretations all receive a score of 0.

[6]The system does not yet weaken an interpretation by a small amount; we have included this possibility for symmetry and possible future use.

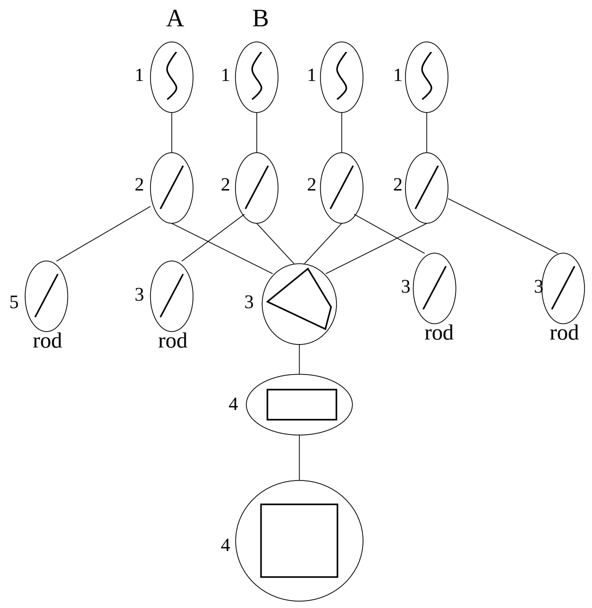


Figure 6: A recognition graph for four strokes; scores are shown at the left of each interpretation.

to the system. Our overall approach to the problem is to take into account as many sources of knowledge as prove useful in interpreting the sketch. We knew that it would be impossible to identify and implement them all at the outset, hence our design put a high premium on the ability to add and remove sources of evidence easily.

4.3 Resolution

The third stage in the interpretation process involves deciding which interpretation is currently the most likely. Our system uses a greedy algorithm, choosing the interpretation with the highest total score, eliminating all interpretations inconsistent with that choice, and repeating these two steps until no more interpretations remain to be selected.

The process is illustrated by the interpretation graph in Figure 6, which shows in graphical form all of the possible interpretations of four strokes (the top row of ovals): 4 separate lines, 4 rods, a quadrilateral, rectangle, or square. The rod on the left has the highest score, so it is chosen as a correct interpretation for stroke A. Choosing that interpretation eliminates the interpretations of quadrilateral, rectangle or square, because stroke A is needed in any of these interpretations. In this context the other strokes are interpreted as rods because that interpretation has the highest score of any remaining interpretation.

Recall that our interpretation process is continuous: all three stages of processing occur after every new stroke is added to the sketch, and the current best interpretation as selected by the greedy algorithm is presented to the user. The process tends to settle down reasonably quickly, in part because, as noted, we reward longevity. Hence once an interpretation has been presented to the user and unchanged for some period of time, it becomes increasingly unlikely to change.

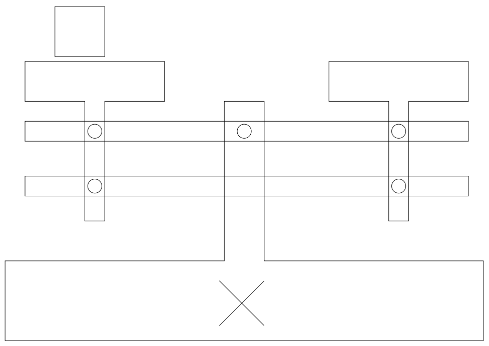


Figure 7: A scale.

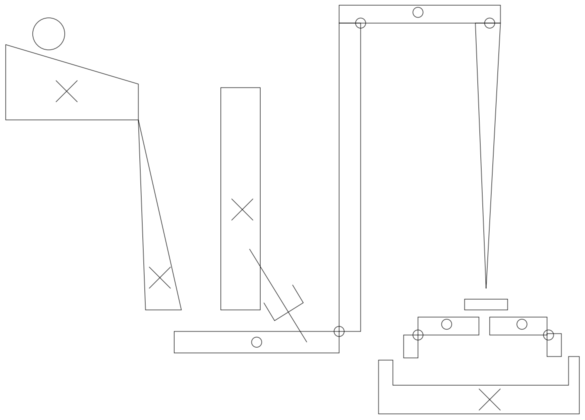


Figure 8: A Rube-Goldberg machine. The ball rolling down the incline sets in motion a sequence of events that eventually pushes the block at the right into the receptacle at bottom right. The device is an adaptation of the one found in [Narayanan, 1995].

5 Evaluation and Results

Our initial evaluation of ASSIST has focused on its naturalness and effectiveness. We asked subjects to sketch both on paper and using ASSIST. We observed their behavior and asked them to describe how ASSIST felt natural and what was awkward about using it.

We tested the system on eleven people from our the laboratory, two of whom had mechanical engineering design experience. All were asked first to draw a number of devices on paper (Figures 7, 8, 9), to give them a point of comparison and to allow use to observe differences in using the two media.

They were then asked to draw the same systems using ASSIST (they drew with a Wacom PL-400 tablet, an active matrix LCD display that allows users to sketch and see their strokes appear directly under the stylus). We asked them how often they felt the system got the correct interpretation and how reasonable the misinterpretations were, and asked them to compare using our system to drawing on paper and to using a menu-based interface.

The system was successful at interpreting the drawings despite substantial degrees of ambiguity, largely eliminating the

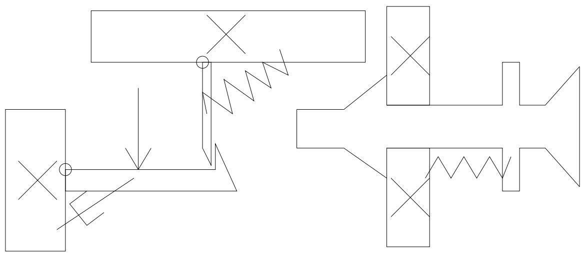


Figure 9: A circuit breaker.

need for the user to specify what he was drawing. As a consequence, a user's drawing style appeared to be only mildly more constrained than when drawing on paper.

People reported that the system usually got the correct interpretation of their sketch. Where the system did err, examination of its performance indicated that in many cases the correct interpretation had never been generated at the recognition step, suggesting that our reasoning heuristics are sound, but we must improve the low-level recognizers. This work is currently under way.

Users tended to draw more slowly and more precisely with ASSIST than they did on paper. The most common complaint was that it was difficult to do an accurate drawing because the system changed the input strokes slightly when it re-drew them (to indicate its interpretations). Users felt that the feedback given by ASSIST was effective but at times intrusive. Our next generation of the system leaves the path of the strokes unchanged, changing only their color to indicate the interpretation.

For a more complete discussion responses to the system from a user interface perspective, see [Alvarado and Davis, 2001].

6 Related Work

The Electronic Cocktail Napkin (ECN) project [Do and Gross, 1996; Gross, 1996] attacks a similar problem of sketch understanding and has a method for representing ambiguity. Our system takes a more aggressive approach to ambiguity resolution and as a result can interpret more complicated interactions between parts. In order for ECN to to resolve ambiguity, the user must either inform the system explicitly of the correct interpretation, or the system must find a specific higher-level pattern that would provide the context to disambiguate the interpretation of the stroke. Our system, in contrast, takes into account both drawing patterns and knowledge of drawing style.

[Mankoff *et al.*, 2000] presents a general framework for representing ambiguity in recognition-based interfaces. This work is similar in using a tree-like structure for representing ambiguity, but touches only briefly on ambiguity resolution. Our work pushes these ideas one step further within the domain of mechanical engineering by providing a framework and set of heuristics for ambiguity resolution.

SILK [Landay and Myers, 2001] allows a user to sketch out rough graphical user interface designs, then transform them into more polished versions. SILK addresses the notion of ambiguity, but limits its handling of it to single parts, e.g., is this group of strokes a radio button or a check box? This does not in general affect the interpretation of the other strokes in the sketch. In contrast, our system can resolve ambiguities that affect the interpretation of many pieces of the sketch.

A theoretical motivation to our work was provided by work in [Saund and Moran, 1995], which outlines several goals in interpreting ambiguous sketches. Our work implements many of the multiple representation and disambiguation techniques suggested in their work.

We have also been motivated by work in mechanical system behavior analysis, especially in the field of qualitative behavior extraction and representation [Sacks, 1993; Stahovich *et al.*, 1998]. The work by Stahovich aims to extract the important design constraints from the designer's rough sketch and is less focused on the interface or sketch recognition process. It was nevertheless the inspiration for our work in this area.

7 Future Work

The work presented in this paper is a first step toward creating a natural interface. It can usefully be expanded in several areas.

First, our current formulation of recognition and evidential reasoning is of course quite informal. This is a consequence of our focus at this stage on the knowledge level, i.e., trying to determine what the program should know and use to evaluate interpretations. Once the content has become more stable and better understood, a more formal process of evaluation and control (e.g., Bayes' nets) may prove useful both for speed and scaling.

Second, in our efforts to combine the best properties of paper and the digital medium we have yet to find many of the appropriate trade-off points. How aggressive should the system be in its interpretations? Forcing the user to correct the system immediately when it makes a mistake greatly aids recognition, but may distract the designer by forcing her to focus on the system's recognition process rather than on the design. In addition, some ambiguities are resolved as more of the sketch is drawn, yet if the system waits for the sketch to be finished, unraveling an incorrect interpretations can be a great deal of work.

In the same vein, it will be important to calibrate how important true freehand sketching is to designers. The obvious alternative is a icon-based system with graphical editing capabilities (e.g., moving and resizing the standard components). Freehand drawing can be powerful, but alternative interface styles need to be considered as well.

The system should also adapt to new users and their sketching style. For example, one of our heuristics was that people draw all of one object before moving onto the next, but there are of course exceptions. The system should be able to adjust to this type of behavior and learn to override its default heuristic.

8 Conclusion

CAD systems are rarely used in early design because they do not allow for quick and natural sketching of ideas. To be useful here, computers must allow the designer to sketch as on paper, yet provide benefits not available with paper, such as the ability to simulate the system.

To provide an interface that feels natural yet interprets sketches as the user draws, the system must be able to resolve ambiguities without interrupting the user. This work provides one solution to problem of ambiguity resolution in a framework of reasonable generality.

Acknowledgments

Luke Weisman and Mike Oltmans helped extensively in the development of these ideas and this system. The work was supported in part by the National Science Foundation, the MIT/Ford Motor Company Collaboration, and MIT's Project Oxygen.

References

[Alvarado and Davis, 2001] Christine Alvarado and Randall Davis. Preserving the freedom of sketching to create a natural computer-based sketch tool. In *Human Computer Interaction International Proceedings*, 2001.

[Do and Gross, 1996] Ellen Yi-Luen Do and Mark D. Gross. Drawing as a means to design reasoning. *AI and Design*, 1996.

[Gross and Do, 1996] Mark Gross and Ellen Yi-Luen Do. Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of UIST 96*, pages 183–192, 1996.

[Gross, 1995] Mark D. Gross. Recognizing and interpreting diagrams in design. In *2nd Annual International Conference on Image Processing*, pages 308–311, 1995.

[Gross, 1996] Mark D. Gross. The electronic cocktail napkin - a computational environment for working with design diagrams. *Design Studies*, 17:53–69, 1996.

[Hearst, 1998] Marti Hearst. Sketching intelligent systems. *IEEE Intelligent Systems*, pages 10–18, May/June 1998.

[Landay and Myers, 2001] James A. Landay and Brad A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer*, 34(3):56–64, March 2001.

[Mankoff *et al.*, 2000] Jennifer Mankoff, Scott E Hudson, and Grefory D. Abowd. Providing intergrated toolkit-level support for ambiguity in recogntiion-based interfaces. In *Proceedings of the CHI 2000 conference on Human factors in computing systems*, pages 368–375, 2000.

[Narayanan *et al.*, 1995] N. Hari Narayanan, Masaki Suwa, and Hiroshi Motoda. *Behavior Hypothesis from Schematic Diagrams*, chapter 15, pages 501–534. The MIT Press, Cambridge, Massachusetts, 1995.

[Oltmans, 2000] Michael Oltmans. Understanding natually conveyed explanations of device behavior. Master's thesis, Massachusetts Institute of Technology, 2000.

[Rubine, 1991] Dean Rubine. Specifying gestures by example. *Computer Graphics*, pages 329–337, July 1991.

[Sacks, 1993] Elisha Sacks. Automated modeling and kinematic simulation of mechanisms. *Computer-Aided Design*, 25(2):107–118, 1993.

[Saund and Moran, 1995] Eric Saund and Thomas P. Moran. Perceptual organization in an interactive sketch editing application. In *ICCV 1995*, 1995.

[Sezgin,] Metin Sezgin. Early processing in sketch understanding. Unpublished Master's Thesis, Massachusetts Institute of Technology.

[Stahovich *et al.*, 1998] T. Stahovich, R. Davis, and H.Shrobe. Generalting multiple new designs from a sketch. *Artificial Intelligence*, 104(1-2):211–264, 1998.

[Ullman *et al.*, 1990] David G. Ullman, Stephan Wood, and David Craig. The importance of drawing in mechanical design process. *Computer & Graphics*, 14(2):263–274, 1990.

Hierarchical Parsing and Recognition of Hand-Sketched Diagrams

Levent Burak Kara
Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213
lkara@andrew.cmu.edu

Thomas F. Stahovich
Mechanical Engineering Department
University of California, Riverside
Riverside, CA 92521
stahov@engr.ucr.edu

ABSTRACT
A long standing challenge in pen-based computer interaction is the ability to make sense of informal sketches. A main difficulty lies in reliably extracting and recognizing the intended set of visual objects from a continuous stream of pen strokes. Existing pen-based systems either avoid these issues altogether, thus resulting in the equivalent of a drawing program, or rely on algorithms that place unnatural constraints on the way the user draws. As one step toward alleviating these difficulties, we present an integrated sketch parsing and recognition approach designed to enable natural, fluid, sketch-based computer interaction. The techniques presented in this paper are oriented toward the domain of network diagrams. In the first step of our approach, the stream of pen strokes is examined to identify the arrows in the sketch. The identified arrows then anchor a spatial analysis which groups the uninterpreted strokes into distinct clusters, each representing a single object. Finally, a trainable shape recognizer, which is informed by the spatial analysis, is used to find the best interpretations of the clusters. Based on these concepts, we have built SimuSketch, a sketch-based interface for Matlab's Simulink software package. An evaluation of SimuSketch has indicated that even novice users can effectively utilize our system to solve real engineering problems without having to know much about the underlying recognition techniques.

Categories and Subject Descriptors: H.5.2 [**User Interfaces**]: Graphical User Interfaces (GUI), Interaction Styles; I.5.5 [**Pattern Recognition**]: Implementation, Interactive systems

Additional Keywords and Phrases: Sketch understanding, pen computing, symbol recognition, visual parsing, sketch understanding, SimuSketch, Simulink

INTRODUCTION
Pen-based computer interaction is becoming increasingly ubiquitous as evidenced by the growing interest in Tablet PC's, electronic whiteboards and PDA's. Many of these devices now come equipped with robust handwriting recognition, making them an attractive alternative to the keyboard and mouse for text entry. However, when it comes to *graphical input*, such as sketches and diagrams, such devices either leave the pen strokes uninterpreted, or offer only limited support in the form of stroke beautification or simple shape recognition.

We believe that among the many issues that remain to be solved, there are two particular challenges that hinder the development of robust sketch understanding systems. The first is *ink parsing*, the task of grouping a user's pen strokes into clusters representing intended symbols, without requiring the user to indicate when one symbol ends and the next one begins. This is a difficult problem as the strokes can be grouped in many different ways, and moreover, the number of stroke groups to consider increases exponentially with the number of strokes. The combinatorics thus clearly render approaches based on exhaustive search infeasible. To alleviate this difficulty, many of the current systems require the user to explicitly indicate the intended partitioning of the ink. This is often done by pressing a button on the stylus, or more commonly, by pausing between symbols [11, 25]. Alternatively, some systems avoid parsing by requiring each object to be drawn in a single pen stroke [20, 27, 17]. However, such constraints usually result in a less than natural drawing environment.

The second issue is *symbol recognition*, the task of recognizing individual hand drawn figures such as geometric shapes, glyphs and symbols. While there has been significant recent progress in symbol recognition [27, 11, 6, 24], many recognizers are either hand-coded or require large sets of training data to reliably learn new symbol definitions. Such issues make it difficult to extend these systems to new domains with novel shapes and symbols. Additionally most symbol recognizers have been built as stand alone applications without addressing the issue of integration into high-level sketch understanding systems.

In this paper, we address the issue of parsing and recognition of hand-drawn sketches in the domain of network diagrams. The types of sketches we consider can be broadly characterized as a set of symbols (nodes) connected by a set of arrows. The techniques we present are thus well-suited to a variety of diagrams such as signal flow diagrams, organizational charts and algorithmic flowcharts, and to various graphical models

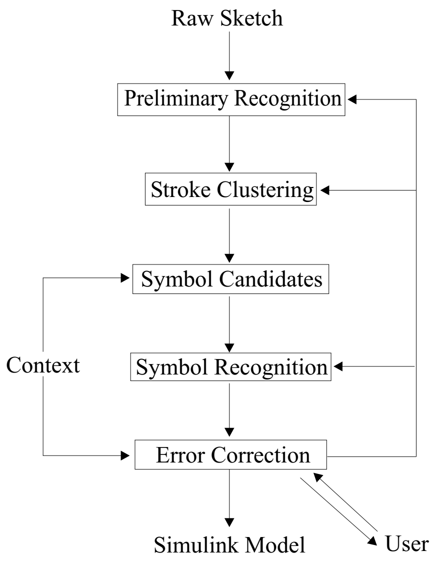


Figure 1: Recognition Architecture.

such as finite state machines, Markov models and Petri nets.

Our approach is based on the hierarchical mark-group-recognize architecture shown in Figure 1. The first step focuses on identifying the arrows in the sketch. We refer to these arrows as “markers” because of two important properties: First, their geometric and kinematic characteristics enable them to be easily extracted from a continuous stream of strokes, and second, they serve as delimiters, which allow the remaining strokes to be efficiently clustered into distinct groups corresponding to individual symbols. The key here is that stroke clustering is driven exclusively by the arrows identified in the first step, without need for search. Next, informed by the result of the clustering algorithm, our approach employs contextual knowledge to generate a set of candidate interpretations for each of the stroke groups. The groups are then evaluated using a symbol recognizer to determine which of these interpretations is correct. The key advantage of our recognizer is that it can learn new symbol definitions from single prototype examples, thus allowing users to easily customize the system to their unique styles. The underlying image-based pattern recognition techniques allow our recognizer to be applicable to multiple-stroke symbols without restricting the order in which the strokes are drawn. In cases of misrecognitions, the last step involves error correction where the user rectifies the mistakes.

OVERVIEW

To provide a test bed for our work, we have created SimuSketch; a prototype sketch-based front-end to Matlab’s Simulink package (Figure 2). Simulink is used for analyzing feedback control systems and other similar dynamic systems. It has a typical drag and drop interface in which the user navigates through a nested symbol palette to find, select and drag the components, one at a time, onto an empty canvas. With SimuSketch, on the other hand, the user can construct functional Simulink models by simply sketching them on a computer screen. The sketch interface does not restrict the order in which the symbols must be drawn nor the number of strokes used to draw them. Furthermore, it does not require the user to indicate when one symbol ends and the next begins. Likewise, the user need not complete one symbol before moving onto another, and thus the user may come back to a previous location to add more strokes at any time.

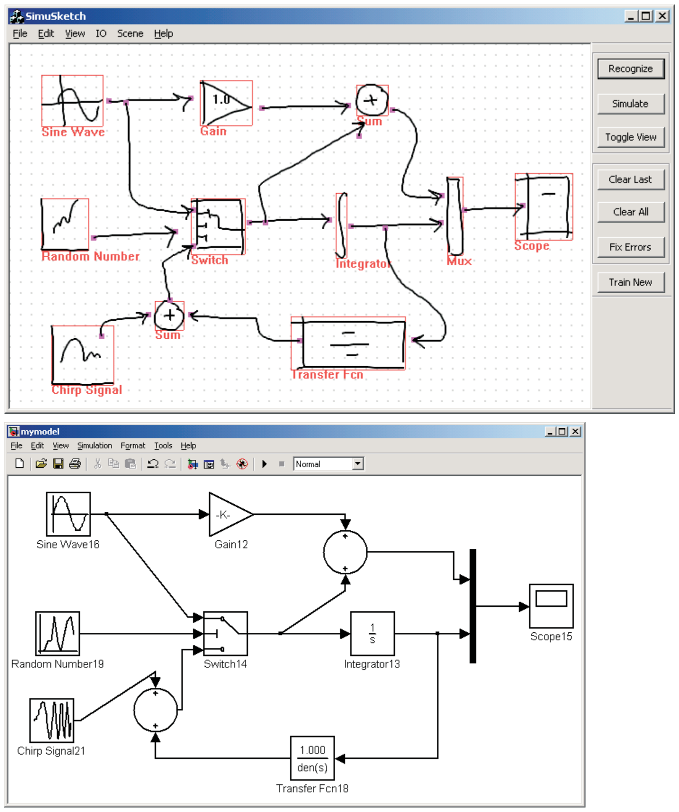


Figure 2: (Top) SimuSketch, (Bottom) Automatically derived Simulink model.

The objects interpreted by SimuSketch are live from the moment they are recognized, thus enabling users to interact with them. For example users can edit the objects through dialog boxes or alter their sketch using traditional means such as selection and deletion. Once the user’s model is recognized, a simulation can be run and viewed directly in SimuSketch. At the end, users can save their work either in their original sketchy form or in a format compatible with Matlab, thus allowing users to resume their work either in the SimuSketch or the conventional Matlab environments.

In the next section, we present a survey of previous research on sketch-based systems with an emphasis on parsing and recognition approaches. Further detail about interaction with SimuSketch and the underlying parsing and recognition techniques are detailed in the subsequent sections.

RELATED WORK

Inspired by the advances in speech recognition, some systems facilitate parsing by requiring visual objects to be drawn with a predefined sequence of pen strokes [30, 33]. While useful at reducing computational complexity, the strong temporal dependency in these methods forces the user to remember the correct order in which the strokes must be drawn. The nature of these approaches thus makes them more suitable to handwriting recognition rather than sketch recognition. Other approaches employ constrained search methods, where the idea is to generate a multitude of partial interpretations from the strokes, and later support or refute these interpretations based on new evidence [13]. Such approaches are often faced with the difficulty of non-optimal thresholds that either prematurely terminate a promising path, or retain a futile one for too long. Alvarado [3], on the other hand, proposed an extension to this idea in the form of Probabilistic Relational Models but has not yet presented formal evaluations.

A number of techniques have been devised for parsing and recognition in visual scenes. Shilman et. al. [31] present a statistical visual language model for ink parsing. During training, a number of spatial relationships between objects are used to construct the object models. During recognition, the models are matched against the users' strokes using a Bayesian framework. Their approach requires a description of a visual grammar, which is currently encoded manually. The trainable parser, on the other hand, requires a large number of training examples. Costagliola and Deufemia [7] present an approach based on LR parsing for the construction of visual language editors. They employ "extended positional grammars" to encode the attributes of the graphical objects and present a set of production/reduction rules for the grammar. Saund et. al. [29] present a system that uses Gestalt principles to determine the salient objects represented in a line drawing. Their work only concerns the grouping of the strokes and does not employ recognition to verify whether the identified groups are in fact the intended ones. Jacobs [16] describes a system to recognize objects with straight-line perimeter representations. The system uses a number of heuristic rules to group edges that likely come from a single object, and then uses simple recognizers to identify the objects represented by the edges. However the rules rely on the presence of straight line segments and sharp corners, and thus are not well-suited to less structured patterns such as sketches.

A number of systems that support sketch-based interaction have been developed in recent years. For user interface design, Landay and Myers [20] present an interactive sketching tool called SILK that allows designers to quickly sketch out a user interface and transform it into a fully operational system. Hong and Landay [14] describe a program called SATIN designed to support the creation of pen-based applications. Lin et al [22] describe a program called DENIM that helps web site designers in the early stages of the design process. All three programs use Rubine's single-stroke gesture recognizer [27] as their main recognition tool and are thus not concerned with parsing. Alvarado and Davis [4] describe a system that can interpret and simulate a variety of simple, hand drawn mechanical systems. The system uses a number of heuristics to construct a recognition graph containing the likely interpretations of the sketch. The best interpretation is chosen using a scoring scheme that uses both contextual information and user feedback. In their approach, each time a new stroke is entered, the entire recognition tree is updated. By contrast, we allow recognition to be controlled by the user. Also, their shape recognizers are sensitive to the results of segmentation (*i.e.*, fitting line and arc segments to the raw ink) forcing the user to be cautious during sketching. Our approach does not rely on segmentation, thus allowing for more casual drawing styles.

Matsakis [24] describes a system for recognizing handwritten mathematical expressions. The work presents an interesting idea based on minimum-spanning trees used for uncovering the spatial structure of the expressions. However the approach requires a large amount of training samples to learn new symbols, and each training sample needs to be drawn using the same number of strokes in the same direction and order. Similarly, recognition is sensitive to the number of strokes and order. Kurtoglu and Stahovich [18] describe a program that augments sketch understanding with qualitative physical reasoning to understand schematic sketches of physical devices. One key feature of their system is that it allows users to incorporate shapes from several different domains, instead of limiting them to one particular domain.

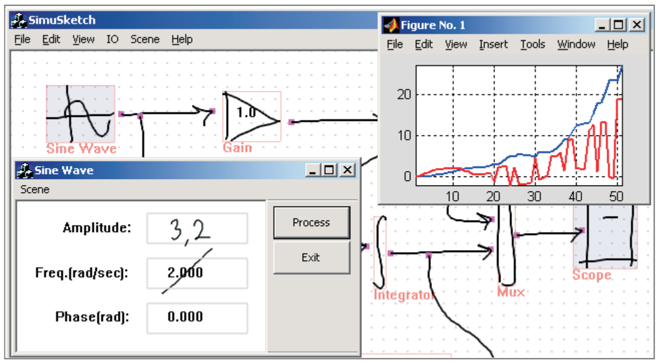
In the field of shape recognition, some methods either rely on single stroke methods in which an entire symbol must be drawn in a single stroke [27, 17], or constant drawing order methods in which two similarly shaped patterns are considered different unless the pen strokes leading to those shapes follow the same sequence [26, 33]. Systems such as [5, 12] allow for multiple stroke symbols, however the recognizers are manually coded. While trainable, systems such as [11, 6, 24, 15] typically require a multitude of training examples. By contrast, we present a multiple stroke symbol recognizer that can learn definitions from single prototype examples.

INTERACTION WITH SIMUSKETCH

SimuSketch is deployed on a 9 in x 12 in Wacom Cintiq digitizing tablet with a cordless stylus. The drawing surface of the tablet is an LCD display, which enables users to see virtual ink directly under the stylus. Data points are collected as time sequenced (x,y) coordinates sampled along the stylus' trajectory. As shown in Figure 2-top, SimuSketch's interface consists of a drawing region and a toolbar that contains buttons for commonly used commands.

The user draws as he or she ordinarily would on paper. As the user is drawing, SimuSketch does not attempt to interpret the scene. Instead, it employs a *recognize on demand* (ROD) strategy in which the user taps the "Recognize" button in the toolbar whenever he wants the scene to be interpreted. This command invokes the sketch recognition engine which then parses the current sketch, recognizes the objects, and produces a Simulink model. As shown Figure 2-top, the program demonstrates its understanding by displaying a faint bounding box around each object, along with a text label indicating what the object is. Recognized arrows are delineated with small colored points at each of their two ends.

The ROD strategy has a number of advantages over the systems that try to interpret the scene after each input stroke. First, as the users are not distracted by display of potentially premature interpretation results, they can focus exclusively on sketching. Second, as very little internal processing takes



Amplitude: 3,2
Freq.(rad/sec): ~~2.000~~
Phase(rad): 0.000

Figure 3: The user can interact with the system through sketch-based dialog boxes. The simulation results are displayed through conventional Simulink graphs.

place after each stroke, the program is better able to keep up with the user's pace[1]. Third, by delaying recognition in a user controlled manner, it allows the system to acquire more context that would help improve the recognition accuracy of earlier strokes. Note that ROD does not require the model to be entirely completed before it can be used. In fact, it encourages an iterative construction process in which the user draws a portion of the final model, asks SimuSketch to recognize it, tests the model, and continues with the rest of the model.

Once the sketch is recognized, the user can run a simulation of it by pressing the "Simulate" button. This command simply hands the model over to Simulink (which runs in the background) for processing. The results of the simulation can be viewed directly in the sketch interface by double tapping on the Scope blocks. As shown in the right part of Figure 3, this brings up a window showing the simulation results. At any time, the user can add new objects to the model by simply sketching them.

Object Manipulation: SimuSketch offers a number of gestures for different tasks. To select an object or an arrow, the user either taps on it or circles it with the stylus; the selected item is highlighted in a translucent blue color indicating its selection. The circular selection gesture is differentiated from a drawing stroke based on its end points and the region it encircles. If the distance between the stroke's first and last points is less than 10% of the total stroke length (*i.e.*, the stroke forms a nearly closed contour) and the stroke encircles one or more objects or arrows, the stroke is taken as a selection gesture. Once an object is selected, one of four things can happen depending on the subsequent input stroke. First, if the stroke is simply a quick tap in the blue region, a *pop dialog* message is dispatched, which brings up a dialog box pertinent to the selected object. Second, if the stroke is not a tap but its initial contact point is still within the blue region, a *move* message is dispatched and the selected object(s) is moved to the lift point of the stroke. Third, if both the contact and lift points of the stroke are outside the blue region but the midpoint is in the blue region, a *delete* message is dispatched and the object is removed from the visual scene. A typical manifestation of this gesture is a stroke through the selected object. Finally, if the entirety of the stroke is outside the blue region, all selected objects are *de-selected* and the stroke is added to the raw sketch. An alternative to de-selection is a tap in the white space.

Object Dialogs: For objects with variable parameters, selecting and tapping on the object brings up a dialog box for editing its parameters. The left part of Figure 3 shows an example. Interaction in these dialog boxes is also sketch-based in that users can cross out the old value with a delete gesture (a stroke through the number) and simply write in the new value. The program can recognize negative and/or decimal numbers using a digit recognizer we have developed.

Views: Once the user's sketch has been interpreted, the user has the option of viewing the model in its sketchy or cleaned up form. In the cleaned up view, the sketchy symbols are replaced by their iconic forms and the arrows are straightened out into line segments. Users can toggle between these two views by tapping the "Toggle view" button. Subjects in our user studies have indicated that the informality of the sketchy view gave a sense of freedom and creativity, while the cleaned up view gave a sense of completeness and definiteness. Despite these perceived differences, the cleaned up view is just as functional as the sketchy view in that it supports the same interaction mechanisms, including sketching, object selection, object manipulation and editing.

SYSTEM DETAILS

In the following sections, we detail each of the steps of our multi-level parsing and recognition approach outlined in Figure 1.

Preliminary Recognition

One key to successful sketch understanding lies in the ability to establish the ground truths about the sketch early on, before costly mistakes take place. Based on this idea, we introduce the concept of "marker symbols," symbols that are easy to recognize and that can guide the interpretation of the remainder of the sketch. In the domain of network diagrams, arrows fulfill this purpose. This approach is similar in spirit to the construction of "islands of certainty" in the Hearsay-II speech understanding system [9].

There are several reasons why arrows are useful marker symbols. First, they occur relatively frequently in network diagrams, thus providing good resolution for separating the other symbols. Second, arrows have unique geometric and kinematic (*e.g.*, pen speed) features that allow them to be reliably extracted from the input stream. Third, as explained later, arrows help guide the interpretation of the other symbols in the sketch by narrowing down the set of possible interpretations. SimuSketch thus begins by recognizing the arrows.

Our observational tests on a small set of users during the design stages of our system indicated that, despite some exceptions, arrows were usually drawn as either a single pen stroke or two consecutive strokes, one for the shaft and one for the head. We thus developed two types of arrow recognizers to account for these two styles. To simplify our analysis, we require that both types of arrows be drawn from tail to head.

[1]Systems that interpret the sketch after each stroke, such as [2], often force the user to pause for a short duration between the strokes.

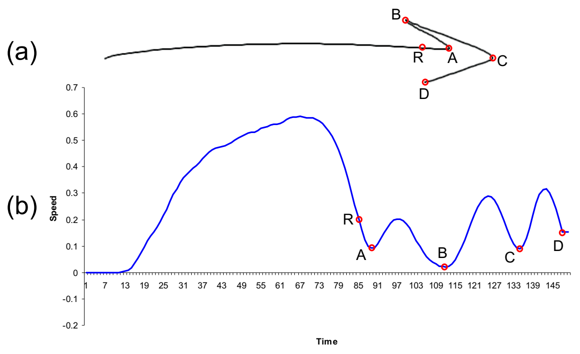


Figure 4: Arrow recognition. (a) A one-stroke arrow with the key points labeled. (b) Speed profile. Key points are speed minima.

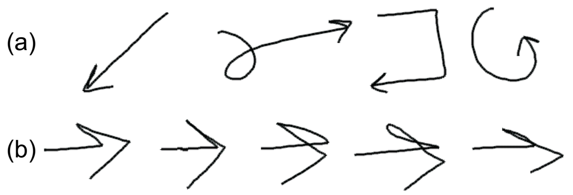


Figure 5: Examples of (a) arrows and (b) arrow heads, that are correctly recognized.

Here we describe only the single-stroke arrow recognizer, as the two-stroke recognizer is a minor extension of it.

Arrow recognition is based on the identification of five key points, labeled A, B, C, D and R in Figure 4a. Points A, B and C correspond to the sharp corners on the arrowhead. The distinguishing characteristic of these points is that they all correspond to pen speed minima, as can be seen in the pen speed profile in Figure 4b. These points are thus identified by locating the last three global minima in the speed profile, excluding the end point, which is labeled point D. Finally, R is a "reference" point on the arrow shaft and is obtained by moving a small distance backwards from point A.

Once these points are determined, a series of geometric tests is performed to determine whether or not the stroke really is an arrow. We require the four angles $\widehat{ABC}$, $\widehat{BCD}$, $\widehat{RAB}$ and $\widehat{RAD}$ to all be less than $90°$, and the length of line segments BC and DC to be less than 20% of the total stroke length. These geometric tests were designed empirically by collecting a corpus of positive and negative examples of arrows from several users, and experimenting with different levels of specificity and thresholds until the best classification performance was obtained. With the resulting recognizer, a variety of arrow shapes with different arrowhead styles can be recognized as shown in Figure 5.

Stroke Clustering

Once the arrows have been recognized, the next step is to group the remaining strokes into different clusters, representing different symbols. The key idea behind stroke clustering is that strokes are deemed to belong to the same symbol only when they are spatially proximate. The challenge is reliably determining when two pen strokes should be considered close together. Here, we rely on the arrows to help make this determination. In network diagrams, each arrow typically connects a source object at its tail to a target object at its head. Hence, different clusters can be identified by grouping together all the strokes that are near the end of a given arrow. In effect, two strokes are considered spatially proximate if the nearest arrow is the same for each. Based on this observation, we developed the following procedure for identifying symbol clusters:

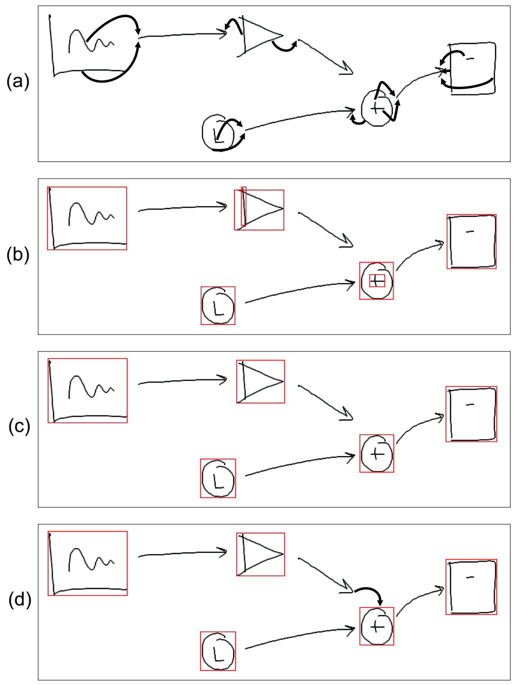


Figure 6: Illustration of the cluster analysis. (a) Each stroke is assigned to the nearest arrowhead or tail. (b) Strokes assigned to the same arrow are grouped into clusters. (c) Clusters with overlapping bounding boxes are merged. (d) Arrows that did not receive any strokes are attached to the nearest cluster.

Step-1 Assign each non-arrow stroke to the nearest arrow: Stroke clustering begins by assigning each non-arrow stroke to the nearest arrow (Figure 6a). The distance between a stroke and an arrow is defined to be the Euclidean distance between the median point of the stroke and either the head or tail of the arrow, whichever is closer. The head is taken to be the apex, which is shown as point C in Figure 4.

Step-2 Combine strokes into clusters: Strokes assigned to the same arrow end in Step-1 are grouped to form a stroke cluster. These clusters will form the basis of the symbols. Figure 6b shows the results of this step.

Step-3 Merge overlapping clusters: Next, clusters with partially or fully overlapping bounding boxes are merged. The bounding box of a cluster is the minimum sized rectangle, aligned with the coordinate axes, that fully encloses the constituent strokes. As shown in Figure 6c, this process combines strokes that are part of the same symbol but which were initially assigned to different arrows in Step-1. If bounding

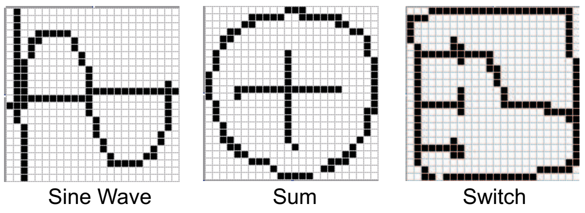


Figure 7: Examples of symbol templates.

boxes of different symbols overlap, this process could erroneously merge the symbols. However, in our experience, we have found that users rarely draw in such a way that this happens. Thus, at the completion of this step, each cluster is assumed to be a distinct symbol.

Step-4 Connect empty arrowhead/tails to the nearest cluster: Step-1 guarantees that each non-arrow stroke is attached to the nearest arrow end. However, some of the arrow ends might remain devoid of any strokes. In this step, empty arrow ends are linked to the nearest stroke cluster (Figure 6d). This step helps to ensure the intended connectivity of the diagram by ensuring that each arrow has a cluster at its tail and head.

Generating Symbol Candidates

After identifying the stroke clusters, the next step is to recognize the symbols they represent. Our approach combines contextual knowledge with shape recognition to achieve accuracy and efficiency. In particular, we examine the number of input and output arrows associated with each stroke cluster to help constrain its possible interpretations. For example, function generators such as the Sine Wave can have only output terminals, and therefore, must have only outgoing arrows. Likewise, certain symbols can have only input terminals, such as the Scope block, or may have an arbitrary number of input and output terminals such as the Sum block.

By examining the number of input and output arrows for a given cluster, SimuSketch identifies a set of *candidate* symbols for the cluster. This reduces the amount of work the subsequent shape recognizer must do and additionally helps increase accuracy by reducing the possibilities for confusion. For example, while the Sum block and the Clock look quite similar (the two circular symbols in Figure 6), context dictates that a Sum block must have at least two incoming arrows while the Clock must have none. With this additional knowledge, the shape recognizer would never consider the Sum block and the Clock as two competing candidates during shape recognition.

Symbol Recognition

We have developed a novel image-based symbol recognizer that can recognize shapes independent of their position, size and orientation.[2] However, it is sensitive to non-uniform scaling, and thus we can distinguish between, say, a square and a rectangle. A distinguishing feature of this recognizer is that it is used for recognizing both the Simulink objects, and the digits in the objects' dialog boxes.

[2]Our recognizer uses a polar coordinate representation to efficiently account for changes in orientation, but that is beyond the current scope.

Input symbols are internally described as 24x24 quantized bitmap images which we call "templates". Figure 7 shows example symbol templates. This representation has a number of desirable characteristics. First, segmentation – the process of decomposing the symbol into constituent primitives such as lines and curves – is eliminated entirely. Second, the representation is well suited for recognizing "sketchy" symbols such as those with heavy overtracing, missing or extra segments, and different line styles (solid, dashed, *etc.*). Lastly, this recognizer puts no restrictions on the number of strokes, or the order in which the strokes are drawn.

Unlike many traditional methods, our shape recognizer requires only a *single* prototype example to learn a new symbol definition. Using the "Train New" button in the interface, the user can create a new symbol definition by simply drawing a shape and assigning a name to it. With this approach, users can seamlessly train new symbols or overwrite existing ones on the fly, without having to depart the main application. This feature makes it easy for users to extend and customize their symbol libraries.[3]

Our recognizer uses an ensemble of four different classifiers to evaluate the match between an unknown symbol and a candidate definition symbol. The classifiers we use are extensions of the following methods: (1) Hausdorff distance [28], (2) Modified Hausdorff distance [8], (3) Tanimoto coefficient [10] and (4) Yule coefficient [32]. The Hausdorff methods reveal the *dissimilarity* between two templates by measuring the distance between the maximally distant pixels in the two point sets. The Tanimoto coefficient on the other hand reveals the *similarity* between two templates by measuring the amount of overlapping black pixels. The Yule coefficient is also a similarity measure except it considers the matching white pixels in addition to the matching black pixels. The motivation for using a multiple classifier scheme lies in the pragmatic evidence that, although individual classifiers may not perform perfectly, they usually rank the true definition highly, and tend to misclassify differently [1]. Hence, by advocating definitions ranked highly by all four classifiers, while suppressing those that are not, we can determine the true class more reliably.

During recognition, each classifier outputs a list of symbol definitions ranked according to their similarity to the unknown. Results of the individual classifiers are then synthesized by first transforming the similarity measures into dissimilarity measures, then normalizing the classifiers' output into a unified scale (to make them compatible), and finally combining the modified outputs of the classifiers. The definition symbol with the best combined score is chosen as the symbol's interpretation.

Error Correction

Our system provides several means to correct recognition errors when they occur. Our techniques have strong parallels with the mediation techniques presented in [23]. When an

[3]Currently SimuSketch has the operational knowledge of 16 Simulink objects. However, the extension to other Simulink objects is straightforward, requiring only code for linking the objects in SimuSketch to those in Simulink.

object is misrecognized, the user can *repeat* the object by selecting, deleting and redrawing it. A more direct way is by choosing the correct interpretation from a *choice list*, which is revealed by bringing the stylus near the misrecognized object and pressing one of the buttons on its side. This list contains only the candidate symbols previously determined using contextual information, and is ranked according to the results of the shape recognizer. Hence, the list is typically short with the correct interpretation usually occurring near the top. Finally, if an arrow goes undetected, and hence becomes part of an object, the user can *dictate* the correct interpretation by drawing a small circle on or near the stroke. This gesture, which we call the ‘o’ gesture, explicitly forces the stroke in question to be an arrow. The ‘o’ gesture is distinguished from a regular drawing stroke based on its absolute size and its two end points. If the gesture fits in a 30 x 30 square on a 1024 x 768 screen, and the stroke forms a closed contour (similar to a selection gesture) without encircling any object, the stroke is interpreted as an ‘o’ gesture. Once a misrecognized arrow is corrected, SimuSketch automatically rectifies the portion of the sketch that was affected by the missed arrow.

USER STUDIES

We conducted two user studies to evaluate our system. The first study focused on the performance of our symbol recognizer and was conducted with a simple interface designed for this study. The second investigated users’ reactions to SimuSketch as a pen-based interaction system, and the evaluation was more observational compared to the first study.

Evaluation of the Symbol Recognizer: Our evaluation of the symbol recognizer consisted of two experiments. In the first experiment, we used a set of 20 graphic symbols shown in Figure 8. Five users participated in this experiment, each of whom was asked to provide three sets of the symbols using the digitizing tablet. In the second experiment, we used digit recognition as our test bed. Nine users participated in the second study and each was asked to provide six sets of digits from “0” to “9”. Both experiments were conducted in a user-dependent setting in which the recognizer was evaluated using the user’s own training symbols. The last set from each user was used for training while the previous ones were used for testing. Each session involved only data collection; the data was processed at a later time. This approach was chosen to prevent users from adjusting their writing style based on our program’s output.

When the top-one classification performance is considered, the recognition rate from the graphic symbol study was 87%. However when top-two classification performance is considered, *i.e.*, the rate at which the correct class is either the highest or second highest ranked class, the accuracy was 97.5%. We consider the top-two classification performance to be of considerable importance, as it provides a measure of how frequently the correct class will appear in the list of alternatives suggested by our program during error correction.

For the digit recognition study, the top-one accuracy was 93.8% and the top-two accuracy was 98.0%. State-of-the-art hand-drawn digit recognition systems achieve recognition rates above 96-97% in user-independent settings [19, 21], however, these systems usually work from scanned images

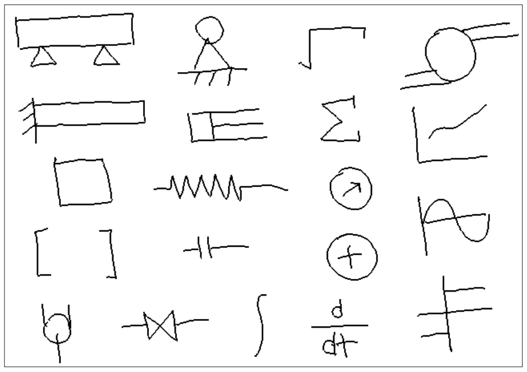


Figure 8: Symbols used in the graphic symbol recognition experiment.

which adds another level of complexity to their task. We achieve about 94% accuracy in a setting where the recognition is user-dependent and the input data is not affected by poor image quality. Nevertheless, we consider our approach to be quite attractive given that it works from a *single* training example. To have a point of comparison, LeCun’s neural network recognizer [21] for handwritten digits, one of the best in its class, uses a total of 60,000 digits for training purposes. As one would expect, if the problem is to recognize digits only, it is better to use a dedicated digit recognizer. However, if the problem involves user defined symbols, such as those shown in Figure 8, our approach has distinct advantages.

Evaluation of SimuSketch: The focus of this study was to assess the performance of SimuSketch. Among the various aspects that we investigated, we were particularly interested in SimuSketch’s ease of use, its parsing and recognition accuracy, users’ adaptability to the system, their success at recovering from recognition errors, and their short and long term view of SimuSketch as a practical front-end to Simulink.

A total of 14 graduate and undergraduate students – 12 engineering and 2 computer science majors – participated in the studies. Nearly half of the users either regularly used Simulink or had previously used it once or twice, while the other half had never used Simulink before. 10 users had no prior experience with the digitizing tablet or the stylus, while 4 users had once used the hardware in a previous study. However, none of the users had previously used SimuSketch, nor had seen it in use by others.

Each session lasted approximately 30 to 40 minutes. For those who were not familiar with Simulink, we first described what Simulink is and gave a brief demonstration on its interface. Next, we introduced SimuSketch. Using simple examples, we demonstrated the means for creating a sketch, selecting, deleting and moving objects, editing object properties, correcting recognition errors, running simulations, training new symbols and switching between views. During this period, we elaborated on SimuSketch’s arrow recognizer as our experience with the first few users had indicated the arrow recognition to be fragile at times. Particularly, we told the users that only single or two stroke arrows were permitted and both types had to be drawn from a source object toward a target object. Other than the recognition of arrows, no fur-

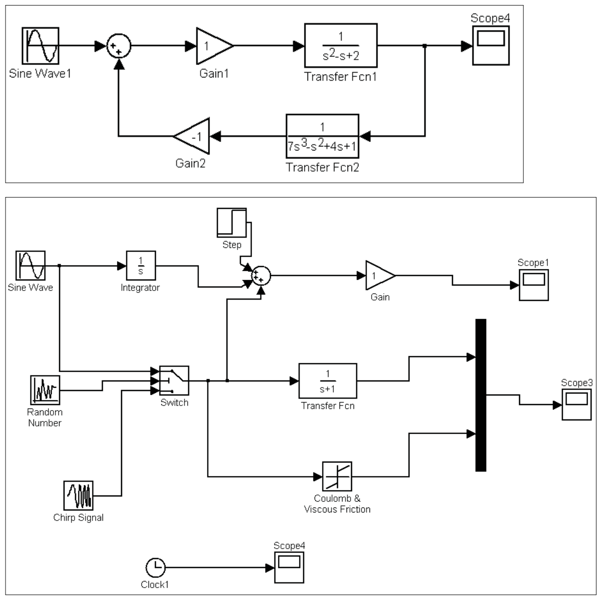


Figure 9: Test problems employed in the user studies.

ther explanation was given regarding the underlying parsing and recognition algorithms. At the end of this introduction, a brief warm up period of approximately 5 minutes was given to let the users become familiar with the hardware and SimuSketch's interface.

The main test involved the two Simulink models shown in Figure 9. Users were asked to use SimuSketch to construct these models, run a simulation of each, and view the results with minimal help from us. The first model involved changing the parameters of several objects through their dialog boxes while in the second model the default values were accepted. Because the users were not involved in the training of the object shapes, none of them knew what the trained shapes looked like. Although users were given the option to train their own set of symbols before starting, none of them chose to do so. Hence, we provided a sketched version of each of the two models as a quick reference. Both the original models and the sample sketches were presented on paper. Similarly, all users decided to use the pre-trained digit recognizer rather than training their own set of digits. However, in this case we did not provide sample figures of the trained digits. Although no time constraints were set, we encouraged users to complete their tasks in a total of 20 minutes.

Observations, Evaluations and Discussions

One consistent pattern among the users was that their encounter with SimuSketch began with great excitement as observed from their reactions during the demo session. This was followed by a period of frustration at the beginning of the warm up period, and finally reached a favorable equilibrium toward the end of the warm up period and during the actual testing. At the end, all users completed the first task successfully, while all but four users completed the second task. In the case of the four users, either the program crashed unexpectedly and they did not have time to redo it, or it was taking too long for them to finish the task.

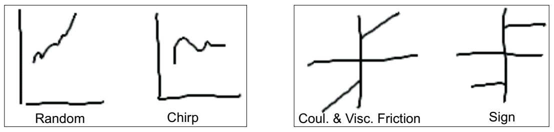
The users' main remark about SimuSketch was that it was intuitive and fast to use, and easy to learn. They particularly liked the idea of simply drawing the objects without having to navigate through an object library to find them. Most users found the interaction mechanisms to be “natural” and “familiar.” Many highlighted the ability to quickly train a custom set of symbols as an outstanding attribute, although they did not make use of it.

The user studies enabled us to evaluate the individual accuracies of our arrow recognizer, parsing algorithm, and symbol recognizer. In its current implementation, our program saves only the user's final sketch, and any objects that are deleted during a drawing session are lost. Our initial accuracy calculations thus do not reflect errors that users repaired by deleting and redrawing objects. This does not produce a significant error in our accuracy calculations, however, because users in the study rarely repaired interpretation errors in this way. In the results presented below, we include estimates of the accuracy that would have been obtained if all interpretation errors had been considered.

The study has shown the main strength of SimuSketch to be its parsing algorithm. In cases where the arrows were all correctly recognized, or the misrecognized ones were corrected by the user, the parsing algorithm had an accuracy above 95%. In the few cases it failed, two distinct symbols were drawn too close to each other and thus their strokes were grouped into a single cluster.

In cases where all stroke clusters were correctly identified, the symbol recognition accuracy was between 85 and 90%. Note that while this result is obtained in a user-independent setting (*i.e.*, the training and test symbols belong to different individuals), it is similar to the result of the user-dependent study explained in the previous section. We believe that SimuSketch's ability to maintain the same level of accuracy in a more difficult setting can be attributed to its use of contextual knowledge for narrowing down the set of interpretations of a symbol prior to recognition. Nevertheless, when errors occurred, they were mostly due to: (1) the confusion between similarly shaped objects, or (2) the recognizer's sensitivity to non-uniform scaling. Figure 10 shows examples of these issues. However, contrary to our expectation, users did not seem to mind such occasional errors, mainly because they found the means for recovery – either by deleting and redrawing, or by selecting the right interpretation from the list of alternatives – to be intuitive and undemanding. In the latter case, the correct interpretation was always in the list of alternatives suggested by SimuSketch.

The main complaint about SimuSketch centered around the arrow recognizer being too restrictive. Although several users quickly became adept at drawing arrows during the warm up period, most users continued having difficulty during the main test session. As we expected, the majority of the errors thus occurred due to the misrecognized arrows. For the most successful users, the arrow recognition accuracy was above 90%. However, when considering all users, the average accuracy for arrow recognition was between 65 and 70%. These results indicate that our arrow recognizer must be further improved to accommodate a wider variety of

(a)


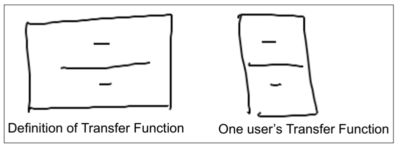
(b)


Figure 10: (a) Pairs of most frequently confused objects. (b) A misrecognition due to non-uniform scaling. (Left) Definition symbol, (Right) One user's misrecognized symbol.

styles. One approach in this direction would be to replace the hard-coded thresholds of the geometric constraints with thresholds that are tunable to individual users.

Besides the issue with arrows, some users had difficulty tapping the stylus to select an object or to bring up a dialog box. Usually, faulty taps were either too gentle, in which case the program did not receive a tap message, or persisted too long on the tablet, in which case the tap was interpreted as a drawing stroke. Another observed difficulty was with the digit recognition in the dialog boxes. While our pre-trained digit recognizer had acceptable performance for certain users, it could not accommodate the vastly dissimilar digit styles that it was not trained for. In cases where the numbers were misrecognized, we asked the users to re-enter them until they got it right. If each user had trained his or her set of digits, we expect that the accuracy would have been similar to the results presented in the previous section.

To obtain the users' evaluation of SimuSketch's performance, we asked each user to complete a questionnaire at the end of the session. The results shown in Table 1 indicate that, while there are a number of usability issues that must be addressed, most users viewed SimuSketch as a promising alternative to Simulink.

Because SimuSketch is still at an early stage, we have deliberately avoided a head-to-head comparison between SimuSketch and Simulink in our user studies. Nevertheless, as a subjective test of how an individual who is proficient in both environments would perform, one of the authors used the two programs to construct and simulate a variation of the second model shown in Figure 9. The test involved creating the model, changing the default properties of several objects, and viewing the simulation results. While the task took 241 seconds to complete in Simulink, it took only 183 seconds in SimuSketch. Although simplistic, we believe this experiment helps reveal the latent value of SimuSketch as a practical tool.

CONCLUSIONS

We have presented a multi-level parsing and recognition approach designed to enable natural sketch-based computer interaction. This approach allows users to continuously sketch without indicating when one symbol ends and a new one begins. Additionally, it does not restrict the number of strokes, or the order in which they must be drawn.

	Score
As I was using SimuSketch , I was able to adapt to it easily	8.2
The software was fast enough to keep up with my pace	7.8
Most of the time, SimuSketch interpreted my sketch the way I intended	7.4
Most of the time, SimuSketch behaved expectedly and when it did not, I felt I was in control to fix it	8.2
The visual feedback on the interpretation results was adequate and unobtrusive	9.1
The editing operations (select, move, delete deselect) were intuitive and easy to use	8.3
I was comfortable using objects' dialog boxes to enter numeric values	7.7
Currently, the overall performance of SimuSketch is	7.6
Assuming that SimuSketch was significantly more robust I would use it in my projects	9.4
Overall, my rating of SimuSketch is	8.7

Table 1: Average scores obtained from user questionnaire. Scale: 1-10, 10 being excellent.

Our approach is based on a mark-group-recognize architecture. In the first step, our program identifies the arrows in the sketch, which serve as useful markers that separate the uninterpreted strokes into distinct clusters. After the symbol clusters are identified, an image-based symbol recognizer, which is informed by clustering and domain specific knowledge, is used to find the best interpretations of the strokes. One advantage of this recognizer over traditional ones is that it can learn new definitions from single prototype examples. The recognizer is versatile in that we use it both for graphical symbol recognition and digit recognition.

We have demonstrated our approach with SimuSketch, a sketch-based interface for Simulink. User studies have indicated that we have sound algorithms for parsing and symbol recognition, and useful means for error recovery. However, our current arrow recognizer should be improved to enhance the user's experience with SimuSketch. Overall, most users had highly favorable opinions of our prototype system, and found it easy and straightforward to use.

While useful for the practicing engineer, SimuSketch is likely to have distinct advantages in engineering education. By its nature, SimuSketch is ideally suited for electronic whiteboard applications and thus can be readily integrated into the classroom environment. In the near future, we plan to explore this possibility with pilot studies.

Finally, although the techniques presented in this paper are tailored toward the domain of network diagrams, our preliminary studies suggest that our mark-group-recognize approach may be applicable to other domains as well. We are currently working to apply this approach to several other domains including electrical circuits and mechanical systems.

References

1. Fevzi Alimoglu and Ethem Alpaydin. Combining multiple representations for pen-based handwritten digit recognition. *ELEKTRIK: Turkish Journal of Electrical Engineering and Computer Sciences*, 9(1):1–12, 2001.
2. Christine Alvarado. *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Master thesis, MIT, 2000.
3. Christine Alvarado. Dynamically constructed bayesian networks for sketch understanding. Technical report, MIT Project Oxygen Student Workshop Abstracts, 2003.
4. Christine Alvarado and Randall Davis. Resolving ambiguities to create a natural sketch based interface. In *IJCAI-2001*, 2001.
5. Ajay Apte, Van Vo, and Takayuki Dan Kimura. Recognizing multistroke geometric shapes: An experimental evaluation. In *UIST 93*, pages 121–128, 1993.
6. Chris Calhoun, Thomas F Stahovich, Tolga Kurtoglu, and Levent Burak Kara. Recognizing multi-stroke symbols. In *AAAI Spring Symposium on Sketch Understanding*, pages 15–23, 2002.
7. Gennaro Costagliola and Vincenzo Deufemia. Visual language editors based on lr parsing techniques. In *8th International Workshop on Parsing Technologies (IWPT'03)*, Nancy, France, 2003.
8. Marie-Pierre Dubuisson and Anil K Jain. A modified hausdorff distance for object matching. In *12th International Conference on Pattern Recognition*, pages 566–568, Jerusalem, Israel, 1994.
9. Lee D Erman, Frederick Hayes-Roth, Victor R Lesser, and D Raj Reddy. The hearsay-ii speech understanding system: Integrating knowldge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, 1980.
10. Michael Fligner, Joseph Verducci, Jeff Bjoraker, and Paul Blower. A new association coefficient for molecular dissimilarity. In *The Second Joint Sheffield Conference on Chemoinformatics*, Sheffield, England, 2001.
11. Manueal J Fonseca, Cesar Pimentel, and Jaoquim A Jorge. Cali-an online scribble recognizer for calligraphic interfaces. In *AAAI Spring Symposium on Sketch Understanding*, pages 51–58, 2002.
12. Manuel J Fonseca and Joaquim A Jorge. Using fuzzy logic to recognize geometric shapes interactively. In *Proceedings of the 9th Int. Conference on Fuzzy Systems (FUZZ-IEEE 2000)*. San Antonio, USA, 2000.
13. W Eric L Grimson. The combinatorics of heuristic search termination for object recognition in cluttered environments. *IEEE PAMI*, 13(9):920–935, 1991.
14. Jason I Hong and James A Landay. Satin: A toolkit for informal ink-based applications. In *ACM UIST 2000 User Interfaces and Software Technology*, pages 63–72, San Diego, CA, 2000.
15. Heloise Hse and A. Richard Newton. Sketched symbol recognition using zernike moments. Technical report, EECS, University of California, 2003.
16. David W Jacobs. The use of grouping in visual object recognition. Technical Report Technical Report 1023, MIT AI Lab, 1988.
17. T D Kimura, A Apte, and S Sengupta. A graphic diagram editor for pen computers. *Software Concepts and Tools*, pages 82–95, 1994.
18. Tolga Kurtoglu and Thomas F Stahovich. Interpreting schematic sketches using physical reasoning. In *AAAI Spring Symposium on Sketch Understanding*, pages 78–85, 2002.
19. Ernst Kussul and Tatyana Baidyk. Improved method of handwritten digit recognition tested on mnist database. In *15th International Conference on Vision Interface*, Calgary, Canada, 2002.
20. James A Landay and Brad A Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer*, 34(3):56–64, 2001.
21. Y LeCun, L D Jackel, L Bottou, A Brunot, C Cortes, J S Denker, H Drucker, I Guyon, U A Muller, E Sackinger, P Simard, and V Vapnik. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks*, pages 53–60, Paris, 1995.
22. James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. Denim: Finding a tighter fit between tools and practice for web site design. In *CHI Letters: Human Factors in Computing Systems*, pages 510–517. ACM Press, 2000.
23. Jennifer Mankoff, Gregory D. Abowd, and Scott E Hudson. Oops: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers and Graphics*, 24(6):819–834, 2000.
24. Nicholas E Matsakis. *Recognition of Handwritten Mathematical Expressions*. Master thesis, MIT, 1999.
25. Shankar Narayanaswamy. *Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals*. Ph.d. thesis, University of California at Berkeley, 1996.
26. Omer Faruk Ozer, Oguz Ozun, C Oncel Tuzel, Volkan Atalay, and A Enis Cetin. Vision-based single-stroke character recognition for wearable computing. *IEEE Intelligent Systems and Applications*, 16(3):33–37, 2001.
27. Dean Rubine. Specifying gestures by example. *Computer Graphics*, 25:329–337, 1991.
28. W J Rucklidge. *Efficient Visual Recognition Using the Hausdorff Distance*. Number 1173 Lecture Notes in computer Science,. Springer-Verlag, Berlin, 1996.
29. Eric Saund, James Mahoney, David Fleet, Dan Larner, and Edward Lank. Perceptual organisation as a foundation for intelligent sketch editing. In *AAAI Spring Symposium on Sketch Understanding*, pages 118–125, 2002.
30. Tevfik Metin Sezgin. Generic and HMM based approaches to freehand sketch recognition. Technical report, MIT Project Oxygen Student Workshop Abstracts, 2003.
31. Michael Shilman, Hanna Pasula, Stuart Russell, and Richard Newton. Statistical visual language models for ink parsing. In *AAAI Spring Symposium on Sketch Understanding*, pages 126–132, 2002.
32. Jack D Tubbs. A note on binary template matching. *Pattern Recognition*, 22(4):359–365, 1989.
33. H Yasuda, K Takahashi, and T Matsumoto. A discrete HMM for online handwriting recognition. *International Journal of Pattern Recognition and Articial Intelligence*, 14(5):675–688, 2000.

Sketch-Based Interfaces for Interactive Computer Graphics

Interactive Beautification: A Technique for Rapid Geometric Design

†*Takeo Igarashi*, †*Satoshi Matsuoka*, ‡*Sachiko Kawachiya*, †*Hidehiko Tanaka*
†Dept. of Information Engineering, ‡Dept. of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyoku, Tokyo, Japan
+81-3-3812-2111 ext. 7413
{takeo, tanaka}@mtl.t.u-tokyo.ac.jp, {matsu, sachiko}@is.s.u-tokyo.ac.jp

ABSTRACT

We propose *interactive beautification*, a technique for rapid geometric design, and introduce the technique and its algorithm with a prototype system Pegasus. The motivation is to solve the problems with current drawing systems: too many complex commands and unintuitive procedures to satisfy geometric constraints. Interactive beautification system receives the user's *freestroke* and beautifies it by considering *geometric constraints* among segments. A single stroke is beautified one after another, preventing accumulation of recognition errors or catastrophic deformation. Supported geometric constraints includes perpendicularity, congruence, symmetry, etc., which were not seen in existing freestroke recognition systems. In addition, the system generates *multiple candidates* as a result of beautification to solve the problem of ambiguity. Using the technique, the user can draw precise diagrams rapidly satisfying geometric relations without using any editing commands.

Interactive beautification is achieved by three sequential processes; 1) inferring underlining geometric constraints based on the spatial relationships among the input stroke and the existing segments, 2) generating multiple candidates combining inferred constraints appropriately, and 3) evaluating the candidates to find the most plausible candidate and to remove the inappropriate candidates. An user study was performed using the prototype system, a commercial CAD, and an OO-based drawing system. The result showed that the users can draw required diagrams more *rapidly* and more *precisely* using the prototype system.

KEYWORDS: Drawing programs, sketching, pen-based computing, constraints, beautification.

INTRODUCTION

Commercial Object-Oriented(OO) drawing editors such as MacDraw and CAD systems have various editing commands and special interaction modes. An user can construct a diagram with geometric constraints by combining these commands appropriately. For example, symmetry can be achieved by the combination of duplication, flipping, and location adjustment, while perpendicularity can be achieved by duplication and 90 degree rotation. In addition, CAD systems often have special interaction modes such as a mode for drawing perpendicular lines. However, invoking these commands or switching to the special editing modes requires additional overhead, and selection of appropriate commands or interaction modes is difficult, especially for novice users[12].

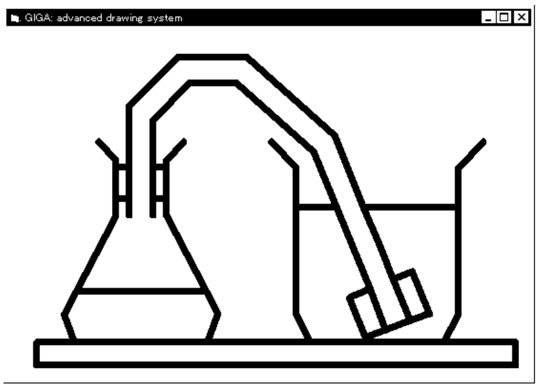


Figure 1: A diagram drawn on the prototype system Pegasus: this diagram is drawn *without any editing commands* such as rotation, copy, or gridding.

To solve these problems, we propose a new interaction technique for drawing, *interactive beautification*. Interactive beautification is a technique for rapid construction of geometric diagrams (an example is shown in Figure 1) without using any editing commands or special interaction modes. Interactive beautification can be seen as an extension of free stroke vectorization [7] and diagram beautification [18]. It receives a user's free stroke and beautifies the stroke considering various geometric constraints among segments. The intuitiveness of the technique allows novice users to draw such precise diagrams rapidly without any training.

Interactive beautification is characterized by the following three features; 1) stroke by stroke beautification, 2) automatic inference and satisfaction of higher level geometric constraints, and 3) generation and selection of multiple candidates as a result of beautification. These three features work together to achieve rapid and intuitive drawing, avoiding the problem of ambiguity.

Interactive beautification is currently implemented on a prototype system Pegasus (an acronym for "Perceptually Enhanced Geometric Assistance Satisfies US!"), and user evaluations using it showed promising results. This paper introduces interactive beautification, and describes the implementation of prototype the system Pegasus in detail.

The remainder of this paper is organized as follows: the next section describes related work in diagram drawing on computers. Then, we describe the technique as seen by the user using several examples. We describe the algorithm of the technique in detail, and introduce the prototype system Pegasus. An user study performed to confirm the effectiveness of the technique is described. Finally, we consider the limitation of our current implementation and conclude the paper.

RELATED WORK

Much work has been done to facilitate the diagram drawing on computers for these 35 years. We will overview several important techniques that have been developed, which affected the design of interactive beautifiacation.

At a glance, the system may seem similar to existing sketch-based interfaces including commercial products such as Apple's Newton, GO's Penpoint, and freestroke drawing mode in typical drawing editors (SmartSketch, Corel Draw, etc.). These systems convert freestrokes into vector segments, and satisfy primitive geometric constraints such as connection. The difference is that interactive beautification considers complex, global constraints such as parallelism, symmetry, or congruence, which enhances the range of geometric models. In addition, the generation and selection of multiple candidates is unseen in the existing systems.

Gesture based systems [1][23][19][16] also employ freestroke input, but they convert input strokes into independent primitives, while interactive beautification converts them into simple line segments satisfying geometric relations. Gross pointed out the importance of context in solving the problem of ambiguity[9], which has influenced our idea.

Beautification systems [18][22][14] are basically batch-based, which can lead to unwanted results because of ambiguity in users input. Interactive beautification prevents such results by interactively presenting multiple candidates and requesting user's confirmation.

While interactive beautification systems control the placement of two vertices (start and end) simultaneously, many existing drawing systems assist the placement of a vertex by controlling the movement of the mouse cursor. Grid restricts cursor placement to some specific geometry and gravity function snaps the cursor to some meaningful places [3]. For example, the Adobe Intellidraw editor[17] automatically aligns the cursor to existing edges. In comparison, the advantages of interactive beautification are as follows: 1) Freestroke drawing is more intuitive and less cumbersome than careful manipulation of the cursor, especially for pen-based interface. 2) The system can attain more information from freestroke trace than cursor placement. For example, equality of interval between parallel lines cannot be detected from the placement of a single vertex.

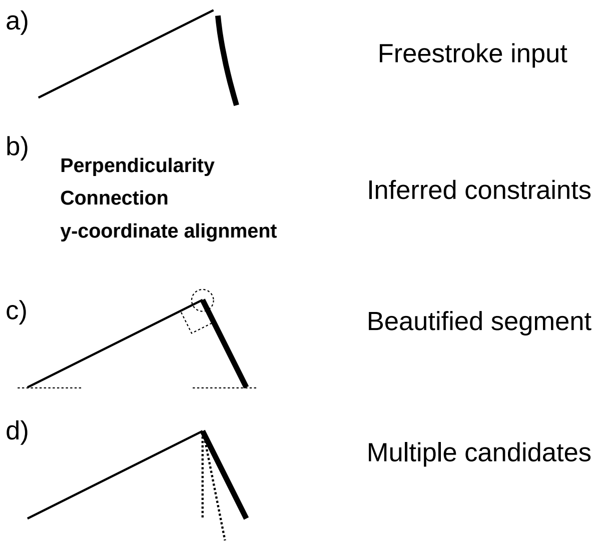


Figure 2: Basic operation of interactive beautification

Bier's Snap Dragging[2], an extension of gravity-active grids, has the same motivation as ours; to make construction of geometric design easier. However, interactive beautification requires much simpler and fewer operations to construct precise diagrams. Moran et al.'s work [20] shares our aims, but does not support the construction of precise diagrams.

Constraint based systems [10][5][21][6] facilitate the construction of complex diagrams with many constraints, but require considerable amount of effort to specify the constraints. Interactive beautification aims at an opposite goal: to reduce the effort by focusing on relatively simple diagrams.

INTERACTIVE BEAUTIFICATION

Basically, interactive beautification is a freestroke vectorization system; it receives a freestroke and converts it into a vector segment, inferring and satisfying geometric constraints.

First, the user draws an approximate shape of his desired segment with a freestroke using a pen or a mouse (Figure 2a). Then, the system infers geometric constraints the input stroke should satisfy by checking the geometric relationship among the input stroke and existing segments(Figure 2b). Finally, the system calculates

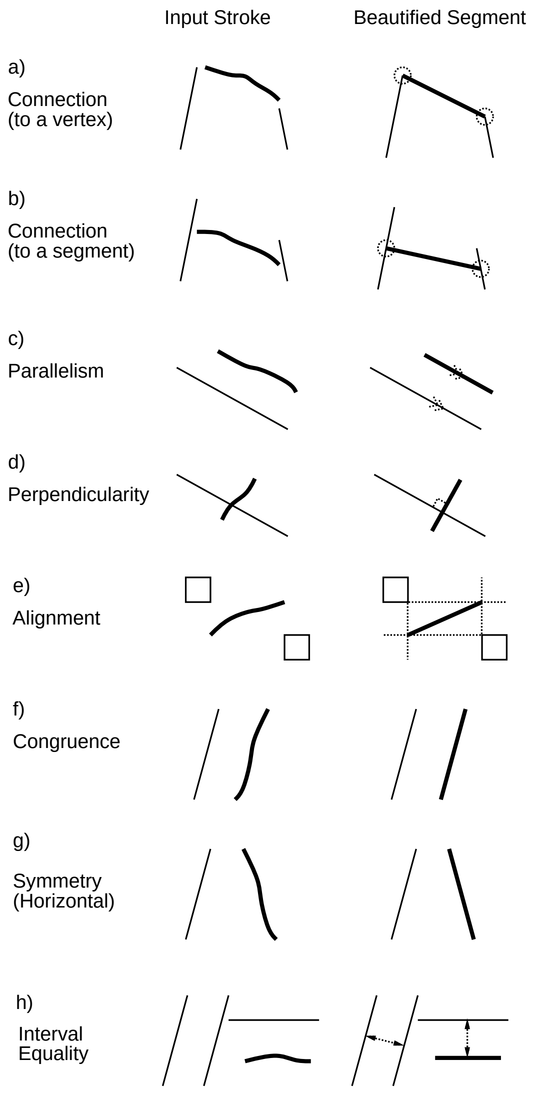


Figure 3: Supported geometric relations

the placement of the beautified segment by solving the simultaneous equations of inferred constraints, and display the result to the user(Figure 2c). In addition, the system generates multiple candidates to deal with the ambiguity of the freestroke (Figure 2d).

The characteristics of interactive beautification are 1) stroke by stroke beautification, satisfying higher level constraints such as congruence, perpendicularity, or symmetry, and 2) generation and selection of multiple candidates. We describe the detail of the interaction in the following subsections.

Stroke by Stroke Beautification Satisfying Geometric Constraints

This subsection describes how diagrams are constructed using stroke by stroke free stroke beautification, satisfying various geometric constraints. To make it simple, we assume that the system generates only one candidate as a result of beautification in this subsection. Next subsection describes the generation of multiple candidates in detail.

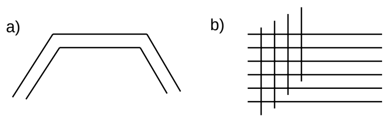


Figure 4: Example use of interval equality among segments

Figure 3 shows some examples of supported constraints, input strokes, and beautified segments. Figures 3a,b describe the connection constraint. If the user draws a free stroke whose start or end point is located near a vertex of an existing segment, the system automatically detects the adjacency and connects the point to the vertex or the body of a segment.

Figures 3c,d illustrate parallelism and perpendicularity constraints. The system compares the slope of the input stroke and those of existing segments, and if it finds an existing segment with approximately the same slope, it makes the slope of the beautified segment identical to the detected slope. Similarly, if the system finds an existing segment approximately perpendicular to the input stroke, it converts the stroke into a precisely perpendicular segment.

Figure 3e shows vertical and horizontal alignment constraints. When a free stroke is drawn, the system individually checks the x and y coordinates of the vertices of the input stroke, and makes the coordinates precisely identical to the existing ones if they are near.

Figures 3f,g show congruence and symmetry constraints. When a new input stroke is drawn, the system searches for a segment almost congruent to the stroke among the existing segments. If such a segment is found, the system makes the input stroke exactly congruent to the segment (Figure 3f). Similarly, the system searches for a segment that is similar to the vertically or horizontally flipped input stroke. If such a segment is found, the system makes the input stroke exactly congruent to the flipped one (Figure 3g).

Figure 3h describes interval equality. This relation is detected by comparing the interval between the input stroke and an existing line segment parallel to the stroke, and intervals between existing parallel segments. This mechanism can be used to draw a pipe with a constant width or to draw cross stripes or grids (Figure 5). Construction of these diagrams is particularly difficult with menu-based systems, where the user must copy, rotate, and move the segment.

In actual drawing, the geometric constraints described above are combined and work together to produce a precise diagram. In Figure 3a, relations such as connection, perpendicularity, and y-coordinate alignment are simultaneously satisfied. In Figure 3b, interval equality, y-coordinate alignment and flipped congruence (symme-

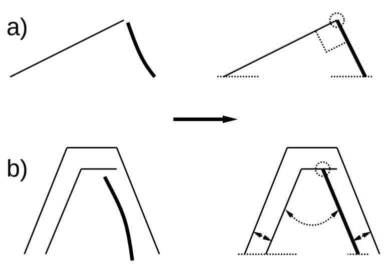


Figure 5: Construction of a diagram with many constraints

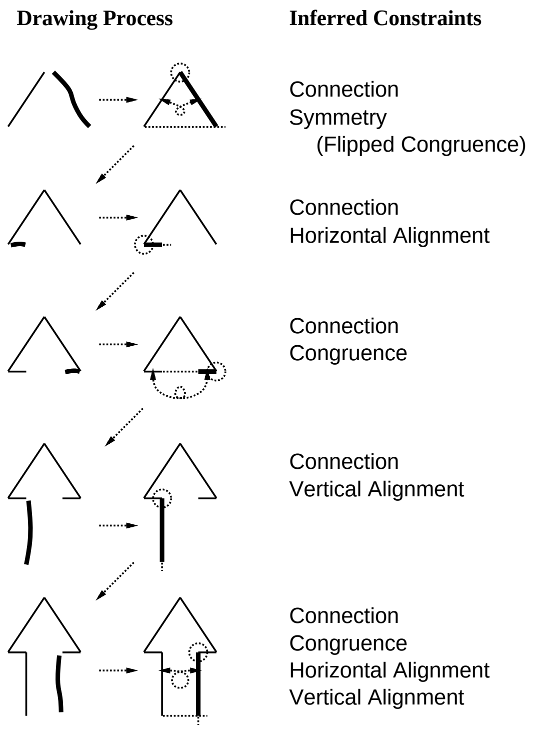


Figure 6: Construction of a symmetric diagram

try) work together to generate the arch (The unnecessary line fragments can be removed easily by 'erasing' interaction, which is explained later).

Figure 6 illustrates how a symmetric diagram is constructed using interactive beautification. For each input stroke, the system infers appropriate constraints and returns a beautified segment. Notice that, except for the slope sides which constitute the arrowhead, the symmetry for the rest of the arrow shape is achieved solely by locally defined relationships (alignment, congruence and connection constraints) without resorting to some special constraints to achieve global symmetry.

Generation and Selection of Multiple Candidates

The inherent difficulty with any freestroke recognition systems is that a freestroke is ambiguous in nature. The user draws an input stroke with an intended image in mind, and the system must infer the intended image

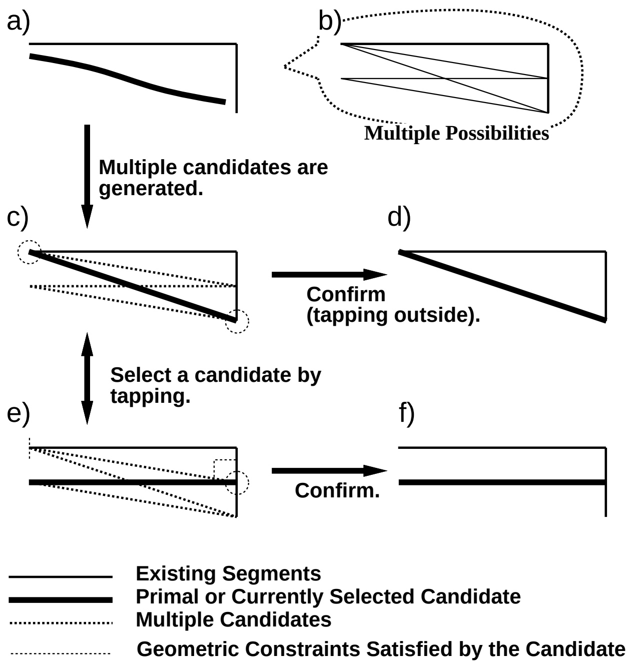

a)
b)
Multiple Possibilities
Multiple candidates are generated.
c)
d)
Confirm (tapping outside).
Select a candidate by tapping.
e)
f)
Confirm.
Existing Segments
Primal or Currently Selected Candidate
Multiple Candidates
Geometric Constraints Satisfied by the Candidate

Figure 7: Interaction with multiple candidates: the user can select a candidate by tapping on it, and satisfied constraints are visually indicated.

based on the shape of the freestroke. However, it is not an easy problem to reconstruct the intended image from the ambiguous input stroke. For example, when the system observes an input stroke shown in Figure 7a, it is difficult to guess which segment in Figure 7b is the one the user intended. Existing systems do not consider these multiple possibilities, and just return a single segment as a result. If the user is not satisfied with the result, he must draw the stroke again, but the revised stroke may also fail.

To solve the problem, interactive beautification infers all possible candidates and allows the user to select one among them (Figure 7c). If the user is not satisfied with the primary candidate, he can select other candidates by tapping on them directly (Figure 7e). During the selection, the system visually indicates what kinds of constraints are satisfied by the currently selected candidate. Visualized constraints ensure that the desired constraints are precisely satisfied. In addition, they assist the selection of a candidate in a cluttered region, where it is difficult to find the desired one. The selection completes when the user taps on outside the candidates or draws the next stroke (Figure 7d,f).

Generation of multiple candidates, together with visualization of the satisfied constraints, greatly reduces the failure in recognition, and makes it possible to construct complex diagrams such as Figure 1 using freestroke only. Additional overhead caused by candidate selection is minimum because the user can directly go to the next stroke without any operation when the primary candidate is satisfactory.

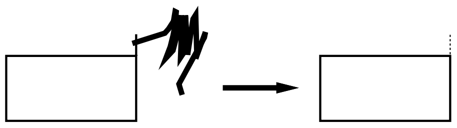


Figure 8: Erasing gesture and trimming operation

Auxiliary Interfaces
In addition to free stroke drawing and selection by tapping, the current system supports a floating menu and an erasing gesture. The floating menu is a button on the screen, and the user can place the button anywhere by dragging it. Menu commands appear when the user taps on the button, similar to a pie menu[11]. Currently, 'clear screen' and 'undo' commands are implemented in the menu.

The erasing gesture is scribbling. If the system detects the gesture, it deletes the nearest line segment to the start point of the scribbling gesture. As the system partitions the line segments at every cross point and contact point beforehand, the user can easily *trim* the unnecessary fragments (Figure 8). Trimming is a frequently used operation on any drawing system, and this easily accessible trimming operation greatly contributes to the efficient construction of complex geometric diagrams.

ALGORITHM
This section describes the algorithm of interactive beautification in detail. From a programmer's point of view, the interactive beautification system works as follows (Figure 9); 1)When the user finishes drawing and lifts the pen from the tablet, the system first checks whether the stroke is an erasing gesture or not. 2)If the input stroke is not an erasing gesture, the beautification routine is called. It receives the stroke and the scene description as input and returns multiple candidates as output. Then, the generated candidates are indicated to the user, allowing him to select one. 3)*The settlement routine* is called when the user finishes selection, that is, starts to draw the next stroke or taps on outside the candidates. The settlement routine adds the selected candidate to the scene description and discards all other candidates. 4)If an erasing gesture is recognized, the erasing routine detects the segment to be erased and removes the segment from the scene. The settlement routine is called after the erasing routine to refresh the scene description. Settlement routine also performs some preliminary calculations to accelerate the beautification process (sorting the vertex coordinates, for example).

We now describe the algorithm of beautification routine in detail. The beautification routine consists of three separate modules (Figure 10). First, a constraint inference module infers the underlining constraints the input stroke should satisfy. Next, a constraint solver generates multiple candidates based on the set of inferred constraints. Finally, an evaluation module evaluates the certainty of generated candidates and selects a primary candidate. The separation of the constraint inference

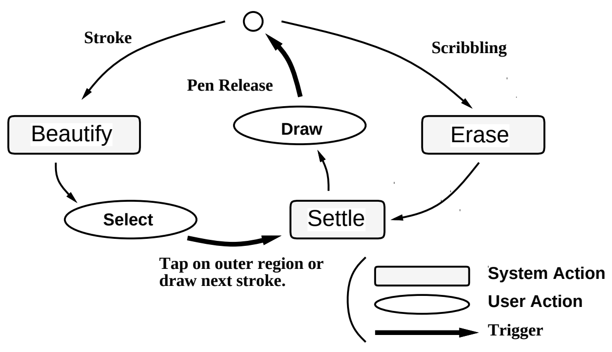


Figure 9: Operational model of interactive beautification

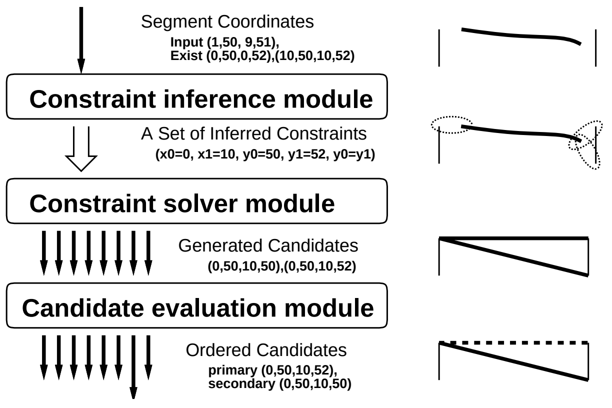


Figure 10: Sturcture of the beautification routine

and the constraint solving remarkably improves the efficiency of multiple candidates generation, because the system performs the most time-consuming task of checking all combinations of segments only once, instead of performing the task for each candidates.

The evaluation process must follow to the solver because it is necessary to consider the resulting coordinates as well as the satisfied constraints to calculate the certainty of a candidate. That is, the candidate located close to the input stroke should be evaluated highly, but the location is unkown until the constraint is solved.

Constraints are represented as numerical equalities binding four variables (coordinates of the new segment). The constraint inference module communicates the inferred geometric relations in a form of numerical equalities, and the constraint solver solves the simultaneous equations. Figure 11 shows the currently supported geometric relations and the corresponding numerical equalities.

Constraint Inference module
First, the system searches the table of parameters of all the existing segments, in order to find values that are 'adjacent' to those of the input stroke and generates constraints that would constraint the parameters of the input stroke as variables. To be specific, the system examines and compares the 5 parameters of the input stroke (x, y coordinates of start/end vertex, and

Geometric Relations	Corresponding Equalities

Connection (start point on a vertex)	x0 = const
	y0 = const

Connection (end point on a vertex)	x1 = const
	y1 = const

Connection (start point on a line)	y0 = const * x0 + const
Connection (end point on a line)	y1 = const * x1 + const
Alignment (start -x)	x0 = const
Alignment (start -y)	y0 = const
Alignment (end -x)	x1 = const
Alignment (end -y)	y1 = const
Vertical line	x0 = x1
Horizontal line	y0 = y1
Congruence (Symmetry)	x1 - x0 = cosnt y1 - y0 = const
Parallelism (Perpendicularity)	y1 - y0 = const * (x1 - x0)

Interval equality	y0 = const * x0 + const
	y1 = const * x1 + const

Figure 11: Relation between geometric relations and equalities

the slope of the stroke). As a result, constraints to represent geometric relations such as x and y coordinate alignment, parallelism, and perpendicularity, are generated. As the parameters of all segments in the scene are sorted in the settlement routine, the computational complexity of this routine is $O(\log n)$ while n is the number of existing segments. Perpendicularity is achieved by storing 90 degrees rotation of the existing slopes.

Next, all the segments in the scene are examined to find various geometric relations between the existing segments and the input stroke, such as congruence, connection and symmetry. In addition, to find the equality of intervals among segments, this routine calculates the interval between the input stroke and each approximately parallel segment in the scene, and searches for the stored interval that are adjacent. The computational complexity of this routine is $O(n \log n)$.

This two-phased constraint inference process generates a set of constraints to be satisfied. To reduce unnecessary overhead in constraint solving, the system checks the duplication whenever a new one is created during the constraint inference.

Constraint Solver

Subsequently to the constraint inference, the system calculates the coordinates of the beautified segment based on the inferred constraints. As the inferred constraints are usually over-constrained (they can not be under-constrained because all variables are automatically bounded to the original coordinates of the input stroke), the system searches for all the possible combinations of inferred constraints to generate multiple candidates.

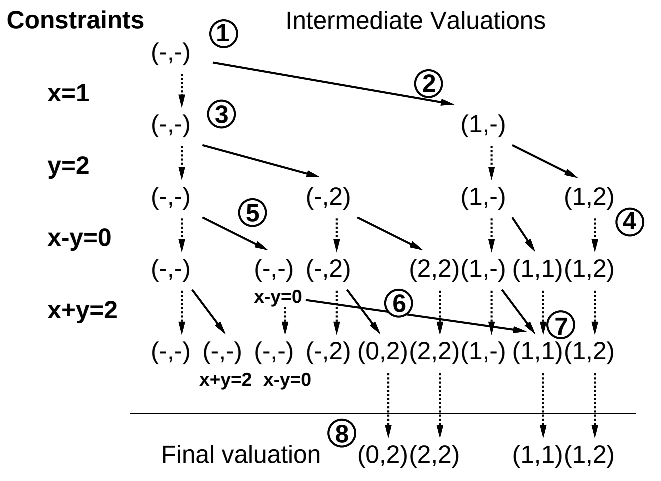


Figure 12: Algorithm for constraint solving

The constraint solver is a modification of the equality solver of CLP($\Re$)[13] with an extension to generate multiple candidates from over-constrained equalities. Similar to the equality solver of CLP($\Re$), the initial state consists of an empty valuation, and the system tries to apply the constraint one by one to the intermediate valuation. The difference is that the system maintains a set of valuations instead of a single valuation, and the new valuation is *added* to the valuation set without *discarding* the previous valuation when a constraint is successfully applied.

Figure 12 shows how the solver works using a simplified example with two variables and four constraints. First, the solver creates an empty valuation (1), and then, applies the first constraint `(x=1)` to the valuation. Naturally, the constraint is successfully applied and a new valuation is created `(1,-)`(2). Note that the initial valuation `(-,-)` is preserved instead of being replaced by the new valuation (3). When the solver tries to apply the constraint `(x-y=0)` to the valuation `(1,2)`, the application fails and no new valuation is created (4). On the other hand, the constraint can be successfully applied to the empty valuation `(-,-)`, creating a new valuation with a suspended (delayed) constraint (5). The suspended constraints are solved when enough variables are ground or enough equalities are given(6). Identical valuations are detected and unified by the solver to prevent redundant calculations (7). Finally, the system returns the fully grounded valuations as multiple candidates (8).

To improve efficiency, intermediate valuations are stored in a tree structure whose root node is the initial empty valuation. This representation is natural because every valuation is created as a child of another valuation with additional grounded variables or additional suspended constraints. If a constraint fails to be applied to a valu-

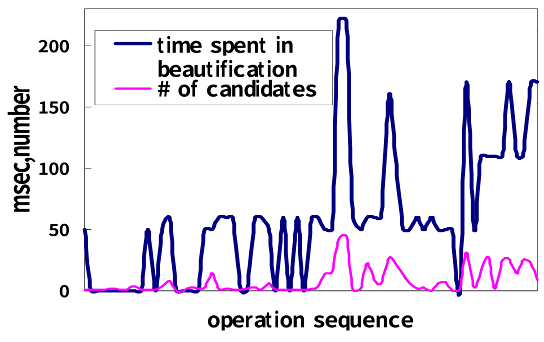


Figure 13: Time spent in the beautification routine and the number of generated candidates during the construction of Figure 1.

ation, it means that the constraint cannot be applied to all its descendants, and the system can avoid wasteful calculations.

The basic method to solve simultaneous equations is Gaussian elimination, because current implementation supports only liner equations. Other algorithms, such as Newton's method[8][10] would be required to support non-liner constraints, such as line length equality or tangency of curved segments. Pair equalities for such constraints as connection to a vertex, congruence, and interval equality (see Figure 11) are bound by `and` condition; both equalities fail if one of them is not satisfied.

In summary, our constraint solver is a multi-way numerical equality solver with an extension to generate multiple solutions efficiently from over-constrained constraints. The complexity of computation is $O(2^n)$, but is substantially reduced by proning wasteful calculations using a tree structure and unifying identical intermediate valuations, and has not caused problems in interaction so far in our prototype system.

PROTOTYPE SYSTEM PEGASUS

The prototype system, Pegasus, is being developed with Microsoft Visual Basic and Visual C++ on Windows 95. The user interface part of the code that manages the input operations and visual feedbacks is written in Visual Basic for ease of implementation and frequent revision. The beautification routine is written in Visual C++ to accelerate the most time consuming process.

Pegasus can work on any PC where Windows 3.1/95 is in operation. However, as Pegasus is basically designed for pen based input, it is developed and tested mainly on portable pen computers (Mitsubishi AMiTY SP) and a pen-based electronic blackboard system (Xerox Liveboard). As pen based freestroke input and mouse based freestroke input have considerably different characteristics, the preprocessor of the recognition algorithm needs to be *tuned* differently depending on the input device (pen or mouse).

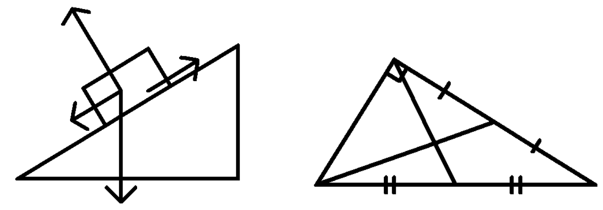


Figure 14: Diagrams for physics and mathematics

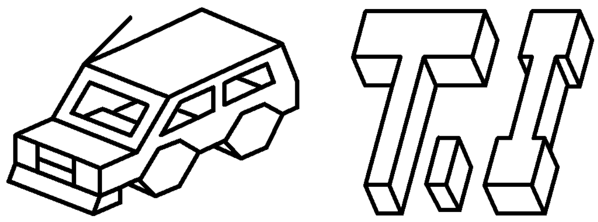


Figure 15: Three-dimensional illustrations

Figure 13 briefly illustrates the processing performance of the current beautification routine. This data was recorded during the construction process of Figure 1 on a PC/AT machine (Pentium 75MHz). Recorded time is not accurate because of the coarse sampling rate of the system call, but approximately 80% of the beautifications finished within 100 msec, sufficient for interactive drawing. The number of generated candidates are usually small, where 62% of beautifications generated less than 5 candidates. However, in some cases (17%) the system generated more than 20 candidates, which made candidate selection difficult.

We show some of the pictures that have been produced on Pegasus. Figure 14 illustrates the usage of the technique in classrooms. Menu-based operations have deterred the use of precise diagrams on electronic whiteboards during oral communications. but the simplicity of interactive beautification may encourage the use of more precise diagrams. Figure 15 shows 3D illustrations. The construction of these diagrams is achieved using parallelism and congruence among segments. It is notable that these diagrams are easily constructed using rather simple constraints, instead of some special techniques for 3D models. Figure 16(left) shows an example

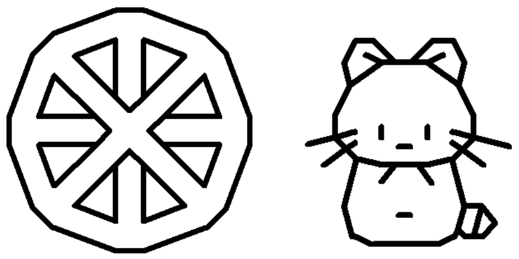


Figure 16: Geometric illustrations

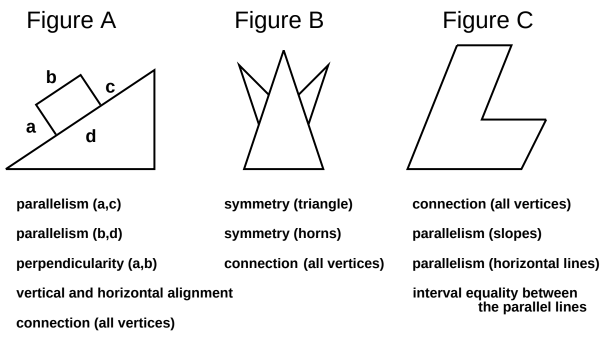


Figure A
Figure B
Figure C
a
b
c
d
parallelism (a,c)
parallelism (b,d)
perpendicularity (a,b)
vertical and horizontal alignment
connection (all vertices)
symmetry (triangle)
symmetry (horns)
connection (all vertices)
connection (all vertices)
parallelism (slopes)
parallelism (horizontal lines)
interval equality between the parallel lines

Figure 17: The diagrams used in the experiment, and required geometric relations

of a geometric design. The widths of the ring and spokes are all identical, which may be difficult for conventional editors. Figure 16(right) gives an example of symmetric illustration. As horizontal symmetry is achieved without any additional operation, a designer can concentrate on *design* itself, instead of struggling with complex operations.

EXPERIMENT

This section describes an experiment performed to evaluate the interactive beautification using the prototype system compared to existing drawing systems in some diagram drawing tasks. We were particularly interested in whether or not interactive beautification would improve the task performance time (*rapidness*) and the completeness of the geometric constraint satisfaction in the diagrams (*precision*). Similar experiment is presented in [12], but this experiment is focused on *evaluation* of the technique, while previous paper intended to clarify the problems of existing drawing editors.

Method

Systems The experiment was conducted on a Mitsubishi pen computer AMiTY SP (i486DX4 75MHz, Win95). Along with our prototype system Pegasus, we used a CAD system (Auto Sketch by AutoDesk Inc.) and an OO-based drawing system (Smart Sketch by Future Software Inc.) The CAD system is used as a representative for precise geometric design systems, and the OO-based editor is selected as a representative for easy-to-use rapid drawing editors.

Task Subjects were required to draw three diagrams shown in Figure 17 using the editors. They were instructed to 1) draw as rapidly as possible, satisfying the required geometric relations as much as possible, 2) to quit drawing when drawing time exceeds the limit of 5 minutes, and 4) give the completion of drawing priority over the complete constraint satisfaction, if it appears to be too difficult.

Subjects 18 student volunteers served as subjects in the experiment. They vary in their proficiency in using computers and each software. 8 subjects were accustomed to typical window-based GUI, but other subjects had little experience with computers.

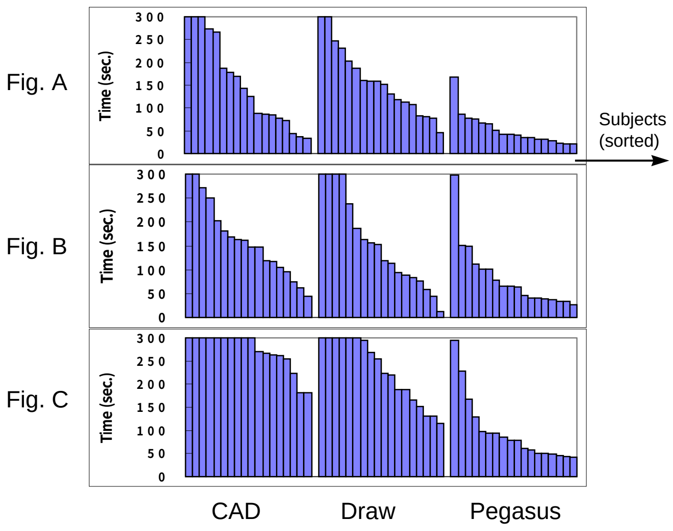


Figure 18: Drawing time required for each task: Each column corresponds to a drawing session of a subject. The order of subjects is sorted by the time required.

Procedure To avoid the effect of learning, the order of editor usage was changed for each subject in a balanced way. The experiment consisted of 18 (subjects) $\times$ 3 (systems) $\times$ 3 (diagrams) = 162 diagram drawing sessions in total. Each session lasted less than 5 minutes and they were video-recorded and examined later.

Prior to performing the experiment with each system, each subject was given a brief explanation of each system and a practice trial. This tutorial session lasted 5 - 10 minutes varying among systems and subjects. CAD system generally required more tutorial time than others.

Result and discussion

Rapidness Figure 18 shows the time required for each subject to complete each task. Each column corresponds to a drawing session of a subject. The order of subjects is sorted by the drawing time. As the drawing time was limited to 300sec., drawing sessions which exceeded the limit are indicated as 300sec. The time required with the prototype system was clearly shorter than with other systems, and all sessions finished within the limit, while many sessions exceeded the limit with the CAD system and the OO-based drawing editor.

Figure 19 shows how many sessions are finished within the limit. Many subjects failed to finish drawing tasks within the limit using the CAD system and the OO-based editor, while all subjects finished drawing using our prototype. Whether the required constraints are precisely satisfied or not is not considered in this graph.

It is impossible to calculate the exact mean drawing time and the mean variance because the recorded drawing time was limited to 300sec., but Figure 20 gives an approximation of the mean drawing time. Drawing time

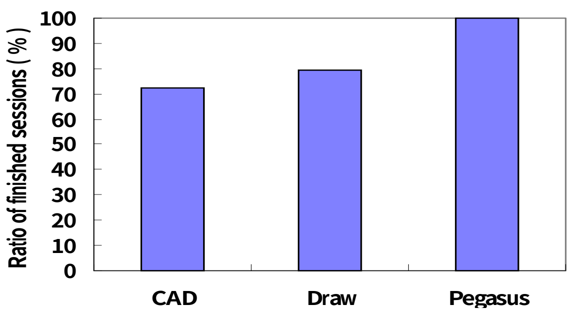


Figure 19: The ratio of finished sessions: this figure shows in how many sessions subjects finished drawing within 300sec. among each $3 \times 18 = 54$ sessions.

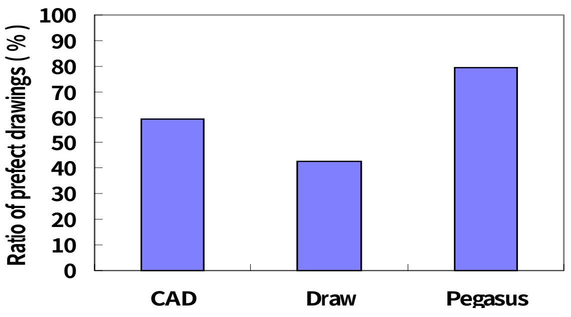


Figure 21: The ratio of diagrams where required constraints are perfectly satisfied: this graph shows in how many sessions subjects successfully satisfied all the required geometric constraints among each $3 \times 18 = 54$ sessions.

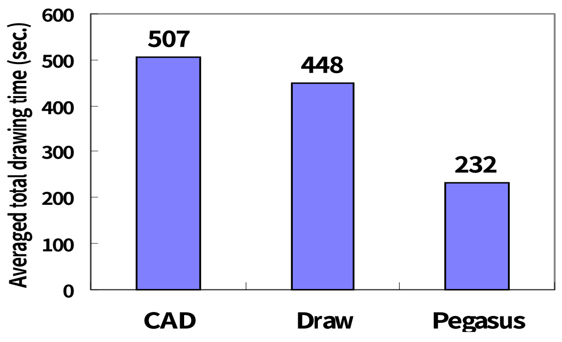


Figure 20: Estimation for time required for a subject to draw the three diagrams: the prototype system exhibits considerable advantage.

is averaged for each diagram-editor combination over those sessions that finished within the limit, and the averaged time for each editor is summed to estimate "total drawing time for a subject to draw three diagrams on each editor." According to the calculations, subjects were able to draw the three diagrams at least 48 % faster than the OO-based editor and 54 % faster than the CAD system. As the averages do not include sessions exceeding 300sec., actual differences are greater.

Precision Even if task performance time might be improved, the benefit could be nullified if the precision of the resulting diagrams is considerably lost. Figure 20 shows how many sessions finished satisfying *all* the required geometric relations shown in Figure 17. The sessions where the subjects finished drawing within 300sec. but failed to satisfy the required geometric relations completely are not counted. It is interesting to see that the OO-based system is superior to the CAD system in time performance, but the opposite holds true concerning the precision, which is in accordance with the natural expectation. Our prototype system showed better performance in both criteria than either systems.

We must note, however, that this experiment is still a preliminary evaluation. Many important aspects of diagram drawing are not accounted for, such as line pattern variation, scaling, rotation, etc. Curves, circles, and text did not appear in the diagrams. Also various kinds of diagrams must be considered, such as node-link diagrams, informal illustrations, complex mechanical diagrams, etc. In spite of these limitations, this preliminary experiment clearly shows a promising potential of interactive beautification system, particularly its significant advantage in rapid and precise construction of simple geometric diagrams. Time performance and constraint satisfaction rate were considerably improved, even though interactive beautification is rather new for the subjects compared with other systems.

LIMITATIONS AND FUTURE WORK

Unsolved problem with interactive beautification is that it is difficult to select the intended candidate among many overlapping candidates. This problem becomes serious when one draws complex diagrams. Possible solutions are to reduce the number of generated candidates and to improve the user interface for candidate selection.

The number of candidates can be reduced by restricting the number of inferred constraints in the constraint inference module and the number of valuations in the constraint solving module, and removing the unwanted candidates in the evaluation module. Various heuristics and user adaptation may be required to find intended constraints and candidates.

Improvement of user interface is also required. One solution is to magnify the cluttered region to help the user to distinguish the desired one from others. Another technique is to let the user specify the reference segment and display those candidates that satisfy constraints related to the specified reference segment.

We plan to implement curves, texts, and line pattern variations to see whether interactive beautification can work as an established interaction technique. Implementation of arcs and curves give rise to various difficulties, but is strongly desirable because satisfaction of curve-related constraints is especially difficult with conventional menu based editors.

We would like to perform more user studies to answer various questions: what kinds of constraints are required for rapid geometric design, how fast user can master the effective use of the technique, and to what extent the generation of multiple candidates facilitates the interaction, etc.

Integration of interactive beautification into 3D scene construction systems such as [24] is also being considered. The most challenging issue may be how to *display* half-constructed 3D models and multiple candidates without confusing the user.

SUMMARY

We have proposed *interactive beautification*, a technique for rapid geometric design. The beautification system receives a freestroke and converts it into a precise segment. The technique is characterized by stroke-by-stroke beautification, recognition of global geometric constraints, and generation and selection of multiple candidates, which make the technique suitable for *precise* geometric design preserving considerable *dexterity.* Our prototype system, *Pegasus*, is implemented on pen computers, and user evaluations showed promising results. The beautification process consists of three independent modules, constraint inference, constraint solving, and candidate evaluation, which achieves efficient generation of multiple candidates.

This technique can be used for geometric modeling on traditional CAD systems, but more informal pen-based rapid drawing of simple diagrams seems to be the most promising target. To be specific, interactive beautification appears to be an ideal technique for note-taking on pen-based PDA systems and graphical explanation on electronic whiteboards during meeting or in classrooms. Finally, this technique can be used for creative design process[15], which has been done with traditional pen and paper rather than on computers because of complex operations.

REFERENCES

1. Apte,A., Vo,V., Kimura,T.D., "Recognizing Multistroke Geometric Shapes: An Experimental Evaluation," *Proc. of UIST'93*, pp. 121-128, 1993.
2. Bier,E.A., Stone,M.C., "Snap Dragging", *Proc. of SIGGRAPH '86*, pp. 233-240, 1986.
3. Bier,E.A., "Snap Dragging: Interactive Geometric Design in Two and Three Dimensions", Ph.D thesis, U.C. Berkley EECS Department, April, 1988.
4. Bolz, D., "Some Aspects of the User Interface of a Knowledge Based Beautifier for Drawings", *Proc. of 1993 Int'l Workshop on Intelligent User Interfaces*, ACM Press, New York, 1993.
5. Borning,A., "The Programming Language Aspects of ThingLab, A constraint-Oriented Simulation Laboratory", *ACM Trans. on Program. Lang. Syst.*, Vol.3, No.4, pp.353-387. 1981.
6. Bouma,W., Fudos,I., Hoffman.D., Cai,J., Paige,R., "Geometric constraint solver", *Computer Aided Design*, Vol.27, No.6, pp. 487-501, 1995.
7. Chen,C.L.P., Xie,S., "Freehand drawing system using a fuzzy logic concept", *Computer Aided Design*, Vol.28, No.2, pp.77-89, 1996.
8. Conte,S.D., Boor,d.C., "Elementary Numerical Analysis", McGraw-Hill, 1972.
9. Gross,M.D., Do,E.Y., "Ambiguous Intentions: A Paper-like Interface for Creative Design", *Proc. of UIST'96*, pp. 183-192, 1996.
10. Heydon,A., Nelson,G., "The Juno-2 Constraint-Based Drawing Editor", SRC Research Report 131a, System Research Center, Digital Equipment Corporation, Palo Alto, California, USA, December, 1994.
11. Hopkins,D., "The design and implemetation of pie menus", *Dr.Dobb's Journal* 1, Vol.6, No.12, pp.16-26, 1991.
12. Igarashi,T., Kawachiya,S., Matsuoka,S., Tanaka,H., "In Search for an Ideal Computer-Assisted Drawing System" *Proc. of INTERACT'97*, 1997, (in press).
13. Jaffar,J., Michaylov,S., Stuckey,P.J., Yap,R.H.C., "The CLP($\Re$) Language and System", *ACM Trans. on Program. Lang. Syst.*, Vol.14, No.3, pp. 339-395, 1992.
14. Kurlander,D., Feiner,S., "Interactive Constraint-Based Serach and Replace", *Proc. of CHI'92*, pp.609-618, 1992.
15. Lakin,F., Wambaugh,J., Leifer,S., Cannon,D., Steward,C., "The electronic notebook: performing medium and processing medium", *Visual Computer*, Vol.5, pp.214-226, 1989.
16. Landay,J.A., Myers,B.A., "Interactive Sketching for Early Stages of User Interface Design", *Proc. of CHI'95* , pp. 43-50, 1995
17. Myers,B.A., Wolf,R., Potosnak,K., Graham,C., "Huristics in Real User Interfaces", INTERCHI'93 Panel, *Proc. of InterCHI'93*, pp.304-307, 1993.
18. Pavlidis,T., VanWyk,C.J., "An Automatic Beautifier for Drawings and Illustrations", *Proc. of SIGGRAPH '85* , pp. 225-234, 1985.
19. Rubine,D., "Combining Gestures and Direct Manipulation", *Proc. of CHI'92*, pp.659-660, 1992.
20. Saund,E., Moran,T.P., "A Perceptually Supported Sketch Editor", *Proc. of UIST'94*, pp. 175-184, 1994.
21. Sutherland,I.E., "Sketchpad: A Man-Machine Graphical Communication System", *Proc. of Spring Jint Computer Conf.*, No.23, pp.329-346, 1963.
22. Weitzman,L., "Designer: A Knowledge-Based Graphic Design Assistant", ICS Report 8609, University of California, San Diego, 1986.
23. Zao,R., "Incremental Recognition in Gesture-Based and Syntax-Directed Diagram Editors", *Proc. of InterCHI'93*, pp. 95-100, 1993.
24. Zeleznik,R.C., Herndon,K.P., Hughes,J.F., "SKETCH: An Interface for Sketching 3D Scenes", *Proc. of SIGGRAPH '96* , pp. 163-170, 1996.

SKETCH: An Interface for Sketching 3D Scenes

Robert C. Zeleznik
Kenneth P. Herndon John F. Hughes
{bcz,kph,jfh}@cs.brown.edu

Brown University site of
the NSF Science and Technology Center
for Computer Graphics and Scientific Visualization
PO Box 1910, Providence, RI 02912

Abstract

Sketching communicates ideas rapidly through approximate visual images with low overhead (pencil and paper), no need for precision or specialized knowledge, and ease of low-level correction and revision. In contrast, most 3D computer modeling systems are good at generating arbitrary views of precise 3D models and support high-level editing and revision. The SKETCH application described in this paper attempts to combine the advantages of each in order to create an environment for *rapidly* conceptualizing and editing *approximate* 3D scenes. To achieve this, SKETCH uses simple non-photorealistic rendering and a purely gestural interface based on simplified line drawings of primitives that allows all operations to be specified *within* the 3D world.

Keywords: Interaction Techniques, 3D Modeling, Gestural Interface, Direct Manipulation, Sketching, Nonphotorealistic Rendering

CR Categories: I.3.8. [Computer Graphics]: Applications; I.3.6. [Computer Graphics]: Methodology and Techniques — Interaction Techniques

1 Introduction

SKETCH targets the exploration and communication of 3D geometric ideas. Traditionally, people have attacked conceptual design with paper and pencil, not with computers, even though computer models offer numerous advantages. The reasons for this include the low overhead of a single-tool interface (pencil), the lack of special knowledge needed to draw, the ease with which many kinds of changes can be made, and the fact that precision is not required to express an idea. Consider Ann sketching a table with an oval top for Joe. Joe gets an immediate sense of the object, without Ann having to indicate the precise locations of the legs, nor the exact shape of the top. By scribbling over what she has sketched, Ann can make the top round or square or freeform without affecting Joe's perception that the legs are attached to the top. (Imagine doing this in a typical CAD or drawing program.) Nevertheless, pencil and paper are still imperfect. After many changes, the paper can become cluttered. Drastic alterations such as showing the model from different viewpoints require new drawings, and collections of drawn objects cannot be transformed as a unit. While computer models do not have these disadvantages, they are typically considerably more difficult to create.

SKETCH is designed to bridge the gap between hand sketches and computer-based modeling programs, combining some of the features of pencil-and-paper sketching and some of the features of CAD systems to provide a lightweight, gesture-based interface to "approximate" 3D polyhedral modeling. Conceptually, our approach is very similar to Landay and Myers' use of sketching to support the early stages of conventional 2D interface design [16]. SKETCH uses a gestural mode of input in which all operations are available directly in the 3D scene through a three-button mouse. The user sketches the salient features of any of a variety of 3D primitives and, following four simple placement rules, SKETCH instantiates the corresponding 3D primitive in the 3D scene. SKETCH allows both geometry and the camera to be gesturally manipulated, and uses an automatic grouping mechanism, similar to that described by Bukowski and Sequin [6], to make it easier to transform aggregates of geometry. Since the set of geometric primitives is more restricted than those in most CAD systems, the user *approximates* complex shapes with aggregates of simpler primitives. Since we know these conceptual models are approximations (often to only partially formed mental images) SKETCH renders them with *non-photorealistic* rendering techniques designed to help viewers see what they want to see.

We also imagine that SKETCH might be used as part of a storyboarding system, for generating a series of scenes and camera views in planning a 3D animation.

The accompanying videotape[1] shows the features of SKETCH and indicates the utility of its simple approach in creating and editing 3D models.

2 Related work

A variety of efforts have been made to simplify the process of generating 3D models, including the "idea sketching" described by Akeo et al. [1]. Akeo allows users to scan real sketches into the computer where they are "marked-up" with perspective vanishing lines and 3D cross sections. The scanned data is then projected onto the 3D mark-up to complete the process.

Nearly all CAD applications employ some form of 2D sketching, although sketching is rarely used in 3D views. A notable exception is Artifice's Design Workshop [2], which allows cubes, walls, and constructive solid geometry (CSG) operations to be constructed directly in the 3D view. However, the overall style of interaction is still menu-oriented and the set of primitives is small.

The considerable work done in the area of drawing interpretation, surveyed by Wang and Grinstein [28], focuses solely on interpreting an entire line drawing at once. In contrast, we attempt to provide a complete interface for progressively conceptualizing 3D scenes using aspects of drawing interpretation to recognize primitives from

[1]The videotape can be obtained upon request from the authors.

a gesture stream. Viking [20] uses a constraint based approach to derive 3D geometry from 2D sketches. In Viking, the user draws line segments, and the system automatically generates a number of constraints which then must be satisfied in order to re-create a 3D shape. The difficulty with these approaches is that even though they are generally restricted to polygonal objects, they are often slow and difficult to implement. In addition, they are often intolerant of noisy input and may either be unable to find a reasonable 3D solution, or may find an unexpected solution. Branco et al. [5] combine drawing interpretation with more traditional 3D modeling tools, like CSG operators in order to simplify the interpretation process; however, their system is limited by a menu-oriented interaction style and does not consider constructing and editing full 3D scenes.

Deering [10], Sachs et al. [22], Galyean and Hughes [11], and Butterworth et al. [7] take a very different approach to constructing 3D models that requires 3D input devices as the primary input mechanism. A variety of systems have incorporated gesture recognition into their user interfaces, including Rubine [21], who uses gesture recognition in a 2D drawing program, but we know of no systems that have extended the use of gesture recognition for 3D modeling.

We also use a variety of direct-manipulation interaction techniques for transforming 3D objects that are related to the work of Snibbe et al. [25], and Strauss and Cary [27]. In addition, we also exploit some very simple flexible constrained manipulation techniques that are similar to those described by Bukowski and Sequin [6]. The latter automatically generates motion constraints for an object directly from that object's semantics. Therefore, for example, when a picture frame is dragged around a room, the frame's back always remains flush with some wall in the room to avoid unnatural situations in which the picture frame might float in mid-air. Also, when a table is manipulated, all of the objects that are on top of the table are automatically moved as well.

In our system, since we have less semantic information than Bukowski, we have less opportunity to automatically generate appropriate constraints, and therefore we occasionally require the user to explicitly sketch constraints in addition to geometry. Our constraint techniques are fast, flexible and almost trivial to implement, but they are not as powerful as the constrained manipulation described by Gleicher [12] or Sistare [24]. Although Gleicher exploits the fact that constraints always start off satisfied, thereby reducing constraint satisfaction to constraint maintenance, he still must solve systems of equations during each manipulation which are often slow and subject to numerical instability. Other approaches like Bier's snap-dragging [4] are also related to our constrained manipulation, although we never present the user with a set of constraint choices from which to select.

Lansdown and Schofield [17] and Salisbury et al. [23] provide interesting techniques for non-photorealistic rendering, although none of these systems specifically targets interactive rendering.

3 The interface

All interaction with SKETCH is via a three-button mouse[2] with occasional use of one modifier key on the keyboard, and a single orthographic window onto the 3D scene. The mouse is used to generate gestures rather than to select operations from menus. Choosing an operation like object creation, transformation or grouping is seamlessly integrated with the natural expression of intent. SKETCH infers intended tools by recognizing gestures — sequences of two types of gestural elements — in its input stream.

Strokes, the first type of gestural element, are pixel-tracks on the film plane[3], made with the first mouse button. There are five classes of strokes shown in Table 1.

Each axis-aligned stroke is aligned with the projection of one the three principal axes of the world. We have also tried aligning strokes with the three principal axes of the surface over which the gesture is drawn. In general, this latter approach seems more effective, although it also presents some difficulties, especially for curved surfaces and for gestures which span over different surfaces. Since we have not yet adequately handled these concerns in our implementation, we will assume for the rest of the paper that all lines are aligned with the world's principal axes except those that are drawn with the "tearing" or freehand strokes.

mouse action	**stroke**
click and release	dot
click and drag without delaying	axis-aligned line: line follows axis whose screen projection is most nearly parallel to dragged-out segment
click and drag, then "tearing" motion to "rip" line from axis	non-axis-aligned line
click, pause, draw	freehand curve
click with Shift key pressed, draw	freehand curve drawn on *surface* of objects in scene

Table 1: The five stroke classes.

Interactors, the second type of gestural element, are made with the second mouse button. The two classes of interactors, a "click and drag" and a "click," have no visual representation.

In addition to gestural elements, SKETCH supports direct-manipulation of camera parameters with the third mouse button, as outlined in Table 2. Third-button manipulations are not discussed further in this paper.

mouse action	**camera manipulation**
click and drag	*pan*: point on film plane beneath mouse remains beneath mouse
click, pause, drag	*zoom/vertical pan*: dragging horizontally zoom in/out towards clicked-on point, dragging vertically pan up/down
click near window boundary, drag	*rotate*: performs continuous XY controller rotation about center of screen [8]
click on object	*"focus"*: camera moves so that object is in center of view [18]
shift-click	*change rendering*: cycles through available rendering styles (see Section 5)

Table 2: Gestures for camera manipulation.

4 The implementation

SKETCH processes sequences of *strokes* and *interactors* to perform various modeling functions with a finite-state machine. The mapping between gestural input and modeling functions is easy to remember and gives the user a clear and direct correspondence. However, one of the principal difficulties in developing a good gesture-based interface is managing the delicate tradeoff among gestures that are natural, gestures that are effective, and gestures that are effective within a system that may already use similar gestures for other functions. For superior gestures to evolve, this tradeoff should continue to be explored especially with user studies.

4.1 Creating geometry

We believe gestures can be a natural interface for the fundamentally visual task of creating 3D geometry. The difficulty is choosing the "right" gesture for each geometric primitive. In SKETCH, we define "primary" gestures for instantiating primitives as sequences

[2]We think that a tablet/pen and an active LCD screen implementation might be even better. See Section 6.

[3]A plane perpendicular to the view direction and close enough to the eyepoint not to interfere with the objects in the scene.

of strokes that correspond to important visual features — generally edges — in partial drawings of the primitives. (see Figure 2 for an overview of all such gestures.) For instance, a drawing of three non-collinear line segments which meet at a point imply a corner, based on our visual understanding of drawings [19]; consequently, we interpret similar gestures composed of three line strokes as a cuboid construction operation.

We also provide alternate construction gestures using non-edge strokes. For example, an object of revolution is sketched via its profile and axis, and cuboids can be created by sketching a single edge and two "dimensioning segments" (perpendicular to the edge) that meet at a vertex lying anywhere along this edge. These alternative gestures take their structure from the notions of generative modeling [26].

SKETCH's other primitives — cones, cylinders, spheres, objects of revolution, prisms, extrusions, ducts and superquadrics — have their own gestures. For most, SKETCH forces some aspect of the shapes to be axis-aligned, so that the gestures are easier to both draw and recognize. For example, to create a duct, the user strokes a closed freehand curve for its cross section, and another freehand curve for its path of extrusion. However, an arbitrary 3D curve is not uniquely determined by a single 2D projection, so SKETCH's ducts must have extrusion paths that lie on an axis-aligned plane, specified by a third gesture — an axis-aligned line stroke normal to the plane on which the path of extrusion should be projected.

The small number of primitive objects sometimes requires the user to build up geometry from simpler pieces, and precludes some complex objects — freeform surfaces and true 3D ducts, for example — from being made at all. But in exchange for this, we believe that our small set of primitives minimizes cognitive load on the user and makes gesture recognition and disambiguation easier. Future work, including user studies, should explore this tradeoff.

4.2 Placing geometry

Object creation requires *placement* of the object in the scene. We base object placement on four rules: first, geometry is placed so that its salient features project onto their corresponding gesture strokes in the film plane; second, new objects are instantiated in contact with an existing object when possible; third, certain invariants of junctions in line drawings [9] that indicate the placement or dimension of geometry are exploited; and fourth, CSG subtraction is inferred automatically from the direction of gesture strokes (Figure 2).

These easy-to-understand rules often generate good placement choices; when they do not, users can edit the results. Furthermore, the few users that the system has had so far have rapidly learned to use the simple rules to their advantage, "tricking" the algorithm into doing what they want. (This may be a consequence of their programming background.)

Figure 1: A series of strokes is drawn in the film plane in red (left). The salient vertex is projected into the scene thus defining the placement of new geometry (green). Though this figure suggests a perspective camera, we use a parallel projection in our application.

The first rule determines object placement except for translation along the view direction. This ambiguity is generally resolved by the second rule, implemented as follows: each gesture has a "most salient" vertex (the trivalent vertex for a cuboid, for example, or the first vertex of the two parallel strokes that indicate a cylinder); a ray is traced through this vertex to hit a surface at some point in the scene. The object is then instantiated so that the salient vertex is placed at the intersected surface point (Figure 1).[4]

The third placement rule exploits invariants of vertex junctions in line drawings, as described by Clowes [9]. However, our use of T junctions is related to the treatment given by Lamb and Bandopadhay [15]. In particular, T-shaped junctions arise in line drawings when a line indicating the edge of one surface, *Estem*, ends somewhere along a line segment indicating the edge of another surface, *Ebar*. These junctions generally signify that the surface associated with *Estem* is occluded by the surface associated with *Ebar*, although it does not necessarily indicate that the two surfaces meet. In SKETCH, a similar notion exists when a straight line segment (except for connected polyline segments) of a gesture ends along an edge of an object already in the scene (Figure 3). To uphold the intuition that such T junctions indicate the occlusion of one surface by another, SKETCH first places the gesture line into the 3D scene according to the previous two placement rules. Then, SKETCH sends a ray out along the gesture line (toward the T junction). If this ray intersects the object that defined the bar of the T junction and the normal at the point of intersection is pointed in approximately the opposite direction of the ray, then the gesture edge is extended so that it will meet that surface.

If the ray does not intersect the surface, then the object defined by the surface is translated along the viewing vector toward the viewer until its edge exactly meets the end of the gesture edge. If the end of the gesture edge is never met (because it was farther away from the viewer), then neither the gesture, nor the existing objects are modified. We never translate objects away from the viewer as a result of T junctions; tests of this behavior on a variety of users indicated that it was both unintuitive and undesirable.

The final rule determines whether the new geometry should be CSG-subtracted from the scene when added to it. If one or more of the gesture strokes are drawn *into* an existing surface (i.e., the dot product of a stroke and the normal to the existing surface on which it is drawn is negative), then the new piece of geometry is placed in the scene and subtracted from the existing object (Figure 2). CSG subtraction is recomputed each time the new geometry is manipulated. If the new geometry is moved out of the surface from which it was subtracted, CSG subtraction is no longer recomputed. This makes possible such constructions as the desk drawer in the Editing-Grouping-Copying section of the videotape.

4.3 Editing

SKETCH supports multiple techniques for editing geometry. Some exploit paper and pencil editing techniques by recognizing editing gestures composed of strokes (e.g., oversketching and drawing shadows). Others use gestures that contain an interactor to transform shapes as a whole by translation or rotation.

Resizing. A common way to "resize" a surface with pencil and paper is to sketch back and forth over its bounding lines until they are of the right size. SKETCH recognizes a similar "oversketching" gesture to reshape objects. If two approximately coincident lines are drawn in opposite directions nearly parallel to an existing edge, SKETCH infers a resizing operation (Figure 2). This sketching operation works for all primitives constructed of straight line segments, including cubes, cylinders and extrusions. Additionally, the two endpoints of an extrusion path can be attached to two objects

[4]If the ray intersects no surface (possible because we use a finite ground rectangle instead of an infinite ground plane), the object is placed in the plane perpendicular to the view direction that passes through the origin; this turns out in practice to be a reasonable compromise.

in the scene; whenever either object moves, the extrusion will resize to maintain the span. However, general reshaping of objects defined by freehand curves is more difficult and not yet fully implemented. We are currently adapting Baudel's mark-based interaction paradigm [3] for use in reshaping 3D objects.

Sketching shadows. Shadows are an important cue for determining the depth of an object in a scene [29]. In SKETCH, we exploit this relationship by allowing users to edit an object's position by drawing its shadow. The gesture for this is first to stroke a dot over an object, and then to stroke its approximate shadow — a set of impressionistic line strokes — on another surface using the Shift modifier key.[5] The dot indicates which object is being shadowed, and the displacement of the shadow from the object determines the new position for the object (as if there were a directional light source directed opposite to the normal of the surface on which the shadow is drawn). The resulting shadow is also "interactive" and can be manipulated as described by Herndon et al. [13].

Transforming. Objects can be transformed as a unit by using a "click-and-drag" interactor (with the second mouse button): the click determines the object to manipulate, and the drag determines the amount and direction of the manipulation. By default, objects are constrained to translate, while following the cursor, along the locally planar surface on which they were created. However, this motion can be further constrained or can be converted to a rotational motion.

It is important to keep in mind that our interaction constraints are all very simple. In SKETCH, instead of using a constraint solver capable of handling a wide variety of constraints, we associate an interaction handler with each geometric object. This handler contains constraint information including which plane or axis an object is constrained to translate along, or which axis an object is constrained to rotate about. Then when the user manipulates an object, all of the mouse motion data is channeled through that object's handler which converts the 2D mouse data into constrained 3D transformations. To define which of our simple constraints is active, we require that the user explicitly specify the constraint with a gesture. Whenever a new constraint is specified for an object it will persist until another constraint is specified for that object. Each new constraint for an object overwrites any previous constraint on the object.

The advantages of such a simple constraint system are that it is robust, fast, and easy to understand. A more sophisticated constraint engine would allow a greater variety of constrained manipulation, but it would also require that the user be aware of which constraints were active and how each constraint worked. It would also require additional gestures so that the user could specify these other constraints.

Systems such as Kurlander and Feiner's [14] attempt to infer constraints from multiple drawings, but this approach has the drawback that multiple valid configurations of the system need to be made in order to define a constraint. Such approaches may also infer constraints that the user never intended, or may be too limited to be able to infer constraints that a user wants.

Constrained transformation. The gestures for constraining object transformations to single-axis translation or rotation, or to plane-aligned translation are composed of a series of strokes that define the constraint, followed by a "click-and-drag" interactor to perform the actual manipulation (Figure 2). To constrain the motion of an object to an axis-aligned axis, the user first strokes a constraint axis, then translates the object with an interactor by clicking and dragging parallel to the constraint axis. The constraint axis is stroked just as if a new piece of geometry were being constructed; however, since this stroke is followed by an interactor, a translation gesture is recognized and no geometry is created. Similarly, if the user drags perpendicular to the constraint axis instead of parallel to it, the gesture is interpreted as a single axis rotation. (This gesture roughly corresponds to the motion one would use in the real world to rotate an object about an axis.)

To translate in one of the three axis-aligned planes, two perpendicular lines must be stroked on an object. The directions of these two lines determine the plane in which the object is constrained to translate. If the two perpendicular lines are drawn over a different object from the one manipulated, they are interpreted as a *contact* constraint (although non-intuitive, this gesture is effective in practice). This forces the manipulated object to move so that it is always in contact with *some* surface in the scene (but not necessarily the object over which the gesture was drawn) while tracking the cursor. Finally, a dot stroke drawn on an object before using an interactor is interpreted as the viewing vector; the object will be constrained to translate along this vector. This constraint is particularly useful for fine-tuning the placement of an object if SKETCH has placed it at the "wrong" depth; however, since we use an orthographic view that does not automatically generate shadows, feedback for this motion is limited to seeing the object penetrate other objects in the scene. We believe that a rendering mode in which shadows were automatically generated for all objects would be beneficial, although we have not implemented such a mode because of the expected computational overhead. We did, however, mock up a rendering mode in which just the manipulated object automatically cast its shadow on the rest of the scene. People in our group generally found the shadow helpful, but were slightly disturbed that none of the other objects cast shadows.

In each case, the manipulation constraint, once established, is maintained during subsequent interactions until a new constraint is drawn for that object. The only exception is that single axis rotation and single axis translation constraints can both be active at the same time; depending on how the user gestures — either mostly parallel to the translation axis or mostly perpendicular to the rotation axis — a translation or rotation operation, respectively, is chosen.

Finally, objects are removed from the scene by clicking on them with an interactor gesture. In early versions of SKETCH we used an apparently more natural gesture to remove objects: the user "tossed" them away by translating them with a quick throwing motion, as one might brush crumbs from a table. We found, however, that this gesture presented a complication: it was too easy to toss out the wrong object, especially if its screen size were small.

4.4 Grouping and copying

By default, objects are automatically unidirectionally grouped with the surface on which they were created, an idea borrowed from Bukowski and Sequin [6], generally resulting in hierarchical scenes. Each geometric object in SKETCH contains a list of objects that are grouped to it. Whenever an object is transformed, that object will also apply the same transformation to all other objects that are grouped to it; each grouped object will in turn transform all objects grouped to itself. Cycles can occur and are handled by allowing each object to move only once for each mouse motion event.

This kind of hierarchical scene is generally easier to manipulate than a scene without groupings since the grouping behavior typically corresponds to both expected and useful relationships among objects. For example, objects drawn on top of a table move whenever the table is manipulated, but when the objects are manipulated, the table does not follow. Grouping also applies to non-vertical relationships, so a picture frame drawn on a wall is grouped with the wall.

In some cases, grouping is bidirectional. The choice of bidirectional and uni-directional grouping is guided by what we be-

[5]Recall from Table 1 that Shift-modified strokes normally produce lines drawn on the surface of objects without special interpretation.

lieve is an inherent difference in the way people interpret relationships between certain horizontal versus vertical drawing elements. When an object is drawn that extends horizontally between two surfaces, like a rung on a ladder, the two surfaces that are spanned are grouped bidirectionally, so that if one rail of the ladder moves so does the other. Although the rung moves whenever either rail is manipulated, the rails do not move when the rung is manipulated. The grouping relationship for objects that span vertically, however, establishes only one-way relationships: the topmost object is unidirectionally grouped to the bottommost object and the spanning object is similarly grouped to the topmost object. Thus, a table leg that spans between a floor and a table top causes the top to be grouped to the floor and the leg to be grouped to the top, but the floor is *not* grouped to the top. We only exploit the difference between horizontal and vertical elements to distinguish these two grouping relationships. However we believe it is important to study with user tests how effective this automatic grouping approach actually is, and perhaps to determine as well if there are other ways that we might be able to exploit the differences between vertical and horizontal elements.

Unlike Bukowski, object grouping is not automatically recomputed as objects are moved around the scene. Therefore, if an object is moved away from a surface, it will still be grouped with the surface. Grouping relationships are recomputed only when objects are moved using the contact constraint mentioned in Section 4.3 — the moved object is grouped to the surface it now contacts and ungrouped from any surface it no longer contacts. We have found this approach to automatic grouping to be simple and effective, although in some environments, Bukowski's approach may be more appropriate.

Lassoing groups. SKETCH also allows the user to explicitly establish groups by stroking a *lasso* around them (Figure 2).[6] Deciding which objects are considered *inside* the lasso is based on the heuristic that the geometric center and all of the visible corners of the object must be inside the lasso; shapes like spheres must be completely contained in the lasso. SKETCH currently approximates this heuristic by first projecting an object's geometric center and all of its crease vertices (where there is a discontinuity in the derivative of the surface) and silhouette vertices into the film plane, then testing whether all these projected points are contained within the lasso. Currently, no test is made for whether objects are occluded or not; future work should address the ambiguities that arise in using the lasso operation.

All lassoed objects are copied if they are manipulated while the Shift modifier is used. Lassoed objects can be scaled by dragging the lasso itself.

Repeating gestures. A different form of copying is used when a user wants to repeat the last geometry-creation operation. After SKETCH recognizes gesture strokes corresponding to a geometric construction, it creates and places the new geometry in the scene, but *does not erase* the gesture strokes. Thus, the user can click on any of these strokes (using button 1) in order to "drag and drop" (re-execute) them elsewhere. Gesture strokes are erased when a new gesture is started or when any object is manipulated. These techniques are shown in the videotape.

5 Rendering

SKETCH renders orthographic views of 3D scenes using a conventional z-buffer. Color Plates I-VI show some of the rendering techniques that SKETCH supports.

[6]The lasso gesture is similar to the sphere gesture. We differentiate between them by requiring that the sphere gesture be followed by a dot stroke, whereas the lasso is simply a free-hand closed curve stroke followed by a manipulation gesture.

"Sketchy" rendering styles are essential because they often enable users to focus on the essence of a problem rather than unimportant details. Non-photorealistic rendering draws a user's attention away from imperfections in the *approximate* scenes she creates while also increasing the scene's apparent complexity and ambiguity. By making scenes more ambiguous, users can get beyond SKETCH's approximate polygonal models to see what they want to see. This is an important concept: we do not believe that sketchy rendering adds noise to a signal; rather we believe that it conveys the very wide tolerance in the user's initial estimates of shape. The user is saying "I want a box *about* this long by *about* that high and *about* that deep." Showing a picture of a box with *exactly* those dimensions is misleading, because it hides the important information that the dimensions are not yet completely determined.

A line drawing effect is achieved by rendering all polygonal objects completely white, and then rendering the outlines and prominent edges of the scene geometry with multiple deliberately jittered lines; the z-buffer therefore handles hidden-line removal. A charcoal effect is created by mapping colors to grayscale and increasing the ambient light in the scene; a watercolor effect that washes out colors is created by increasing the scene's ambient light. There are a number of other techniques that we would like to explore, including pen and ink style textures, and drawing hidden edges with dashed lines.

Objects are assigned a default random color when they are created to help differentiate them from the scenery. We can also copy colors from one object to another. By just placing the cursor on top of one object and pressing the Shift modifier, we can "pick up" that object's color. Then, we can "drop" this color on another object by placing the cursor over it and releasing the modifier. We can also explicitly specify colors or textures for objects. In our present implementation, we do this by placing the cursor over the object and typing the name of the color or texture. Although this interface requires the keyboard, it is consistent with SKETCH's interface philosophy of not making users search through a 2D interface for tools to create particular effects. In the future, we expect that voice recognition, perhaps in conjunction with gesturing, will be a more effective way to establish surface properties for objects (and perhaps other operations as well).

6 Future Work

We regard SKETCH as a proof-of-concept application, but it has many flaws. Many of the gestures were based on an ad hoc trial and error approach, and some of the gestures still do not satisfy us. For example, the pause in the freehand curve gesture rapidly becomes annoying in practice — the user wants to do something, and is forced to wait. Possible solutions of course include using more modifier keys, although we would rather find a solution that preserves the simplicity of the interface.

SKETCH is based on an interface that is stretched to its limits. We expect that adding just a few more gestures will make the system hard to learn and hard to use. We'd like to perform user studies on ease of use, ease of learning, and expressive power for novice users as a function of the number of gestures. We're also interested in trying to determine to what extent artistic and spatial abilities influence users' preference for sketching over other modeling interfaces.

We have begun to implement a tablet-based version of SKETCH. The current generation of tablet pens include pressure sensitivity in addition to a single finger-controlled button, and one "eraser-like" button. In order to develop an equivalent interface for the tablet, we simply need to treat a specific pressure level as a button click to achieve the equivalent of three buttons. Therefore, the button 1 drawing interactions described for the mouse are done by simply pressing hard enough with the penpoint of the tablet pen. To achieve the button 2 operations of the mouse, the user simply presses the finger controlled button on the tablet pen. Finally, to effect camera motion, the user turns the pen over and uses its "eraser" to manipulate the camera. Our initial efforts with a tablet based interface lead us to believe that a tablet based system could be far more effective than a mouse based system, especially if pressure sensitivity is cleverly exploited.

SKETCH is a tool for initial design — the "doodling" stage, where things are deliberately imprecise. But initial design work should not be cast away, and we are examining ways to export models from SKETCH to modelers that support more precise editing, so that the sketch can be moved towards a final design. Since subsequent analysis and design often requires re-thinking of some initial choices, we are also interested in the far more difficult task of re-importing refined models into SKETCH and then re-editing them, without losing the high-precision information in the models except in the newly-sketched areas.

The scenes shown here and in the video are relatively simple. Will sketching still work in a complex or cluttered environments? We do not yet have enough experience to know. Perhaps gestures to indicate an "area of interest," which cause the remainder of the scene to become muted and un-touchable might help.

The tradeoffs in gesture design described in Section 4 must be further explored, especially with user-studies.

7 Acknowledgments

Thanks to Dan Robbins, Tim Miller, and Lee Markosian for many helpful discussions. Thanks also to Andries van Dam and the Graphics Group, as well as our sponsors: grants from NASA, NSF, Microsoft, Sun and Taco; hardware from SGI, HP and Sun.

References

[1] M. Akeo, H. Hashimoto, T. Kobayashi, and T. Shibusawa. Computer graphics system for reproducing three-dimensional shape from idea sketch. *Eurographics '94 Proceedings*, 13(3):477–488, 1994.

[2] Artifice, Inc. Design Workshop. Macintosh application.

[3] T. Baudel. A mark-based interaction paradigm for free-hand drawing. *UIST '94 Proceedings*, pages 185–192, Nov. 1994.

[4] E.A. Bier. Snap-dragging in three dimensions. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):193–204, Mar. 1990.

[5] V. Branco, A. Costa, and F.N. Ferriera. Sketching 3D models with 2D interaction devices. *Eurographics '94 Proceedings*, 13(3):489–502, 1994.

[6] R. Bukowski and C. Séquin. Object associations: A simple and practical approach to virtual 3D manipulation. *Computer Graphics (1995 Symposium on Interactive 3D Graphics)*, pages 131–138, Apr. 1995.

[7] J. Butterworth, A. Davidson, S. Hench, and T.M. Olano. 3DM: A three dimensional modeler using a head-mounted display. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):135–138, Mar. 1992.

[8] M. Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-D rotation using 2-D control devices. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):121–129, August 1988.

[9] M. Clowes. On seeing things. *Artificial Intelligence*, (2):79–116, 1971. North-Holland.

[10] M. Deering. Holosketch: A virtual reality sketching/animation tool. *ACM Transactions on Computer-Human Interaction*, 2(3):220–238, 1995.

[11] T. Galyean and J. Hughes. Sculpting: An interactive volumetric modeling technique. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):267–274, July 1991.

[12] M. Gleicher. Integrating constraints and direct manipulation. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):171–174, March 1992.

[13] K.P. Herndon, R.C. Zeleznik, D.C. Robbins, D.B. Conner, S.S. Snibbe, and A.van Dam. Interactive shadows. *UIST '92 Proceedings*, pages 1–6, Nov. 1992.

[14] D. Kurlander and S. Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, Oct. 1993.

[15] D. Lamb and A. Bandopadhay. Interpreting a 3D object from a rough 2D line drawing. *Visualization '90 Proceedings*, pages 59–66, 1990.

[16] J.A. Landay and B.A. Myers. Interactive sketching for the early stages of user interface design. *Proceedings of CHI'95*, pages 43–50, 1995.

[17] J. Lansdown and S. Schofield. Expressive rendering: A review of nonphotorealistic techniques. *IEEE Computer Graphics & Applications*, pages 29–37, May 1995.

[18] J.D. Mackinlay, S.K. Card, and G.G. Robertson. Rapid controlled movement through a virtual 3d workspace. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 171–176, October 1986.

[19] G. Magnan. *Using technical art: An industry guide*. John Wiley and Sons, Inc., 1970.

[20] D. Pugh. Designing solid objects using interactive sketch interpretation. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):117–126, Mar. 1992.

[21] D. Rubine. Specifying gestures by example. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):329–337, July 1991.

[22] E. Sachs, A. Roberts, and D. Stoops. 3-draw: A tool for designing 3D shapes. *IEEE Computer Graphics & Applications*, pages 18–25, Nov. 1991.

[23] M. Salisbury, S. Anderson, R. Barzel, and D. Salesin. Interactive pen–and–ink illustration. *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 101–108, July 1994.

[24] S. Sistare. Graphical interaction techniques in constraint-based geometric modeling. *Proceedings of Graphics Interface '91*, pages 85–92, June 1991.

[25] S.S. Snibbe, K.P. Herndon, D.C. Robbins, D.B. Conner, and A. van Dam. Using deformations to explore 3D widget design. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):351–352, July 1992.

[26] J.M. Snyder and J.T. Kajiya. Generative modeling: A symbolic system for geometric modeling. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):369–378, July 1992.

[27] P. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):341–349, July 1992.

[28] W. Wang and G. Grinstein. A survey of 3D solid reconstruction from 2D projection line drawings. *Computer Graphics Forum*, 12(2):137–158, June 1993.

[29] L.R. Wanger, J.A. Ferwerda, and D.P. Greenberg. Perceiving spatial relationships in computer-generated images. *IEEE Computer Graphics and Applications*, 12(3):44–58, May 1992.

A Suggestive Interface for 3D Drawing

Takeo Igarashi *John F. Hughes*
Computer Science Department, Brown University
115 Waterman Street, Providence, RI 02912, USA
takeo@acm.org, jfh@cs.brown.edu

ABSTRACT

This paper introduces a new type of interface for 3D drawings that improves the usability of gestural interfaces and augments typical command-based modeling systems. In our suggestive interface, the user gives hints about a desired operation to the system by highlighting related geometric components in the scene. The system then infers possible operations based on the hints and presents the results of these operations as small thumbnails. The user completes the editing operation simply by clicking on the desired thumbnail. The hinting mechanism lets the user specify geometric relations among graphical components in the scene, and the multiple thumbnail suggestions make it possible to define many operations with relatively few distinct hint patterns. The suggestive interface system is implemented as a set of suggestion engines working in parallel, and is easily extended by adding customized engines. Our prototype 3D drawing system, Chateau, shows that a suggestive interface can effectively support construction of various 3D drawings.

KEYWORDS: interaction technique, user interface design, 3D drawing, prediction, gestural interface.

INTRODUCTION

Typical 3D modeling tools are designed for precise control of complicated shapes, and the interfaces are generally hard for casual users to learn. To provide simplified interfaces for sketching 3D structures quickly, various gestural interfaces have been explored [12,25]. These let the user interact directly with 3D scenes without using buttons and menus and reduce the explicit control required by implementing various context-dependent rules.

Although gestural interfaces for 3D modeling have been successful as an experimental effort, they still have several limitations. First, they have been designed primarily for building approximate models rather than for the precise

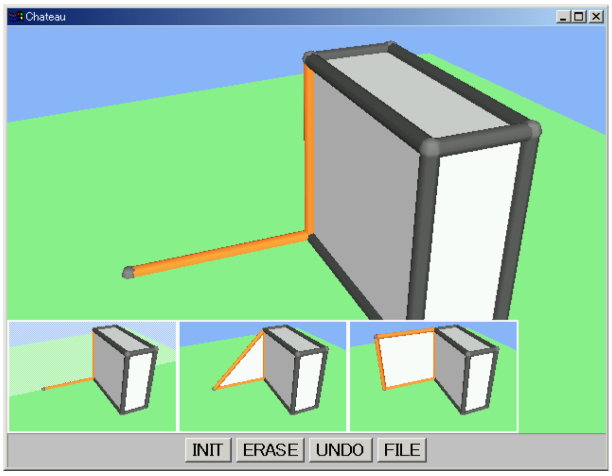


Figure 1: A screen snapshot of our prototype system Chateau. The user gives the hints to the system by highlighting related lines (red lines), and the system suggests possible operations (thumbnails at bottom) based on the hints.

control used in traditional 3D modeling tools, and the realm between these, i.e., approximate modeling of objects with important symmetries and repeated substructures, has been largely unexplored. This makes it hard to sketch many interesting architectural forms, for example.[1] Second, they do not scale well because a system designer cannot define many gestures with limited combinations of gestural elements (stroke, click, modifier key, etc.). Third, it is difficult for novice users to learn a set of gestures because the user must complete a gesture to see the result, and must start over if it fails.

This paper introduces a new type of interface that extends gestural interfaces to address these limitations. In the proposed *suggestive interface,* the user gives the system *hints* about the desired operation by highlighting related components in the scene, and the system *suggests* subsequent operations in an array of small thumbnails derived from the hints and the overall scene configuration (Figure 1). The user can complete an operation by choosing

[1] It was an interest in sketching French chateaus that originally motivated this work.

one of these suggestions, or can ignore them and continue constructing and/or hinting. Suggestions are generated by *suggestion engines,* each of which constantly observes the scene and generates a suggestion when the current hint configuration matches its *input pattern*.

A suggestive interface can be viewed as a mediated version of a gestural interface. Instead of responding to the user's input by updating the scene immediately, the system asks for the user's confirmation after showing multiple suggestions. This approach has several advantages over earlier gestural interfaces. First, the hinting mechanism lets us use existing components as input. This naturally helps in the specification of geometric relations among components in the scene. Second, because suggestions are merely *offered,* a single collection of hints can serve both as a gesture and as a *subset* of a more complex gesture; e.g., the new-drawing-plane engine responds to a single selected line, while the rectangle-creation engine responds to two connected perpendicular selected lines. Third, the suggestive interface helps the learning process because users can progressively refine their hints until the desired result appears among the suggestions. The suggestions themselves, even if not taken, may be helpful in the creative process.

In this paper we describe *Chateau*, a simple proof-of-concept 3D modeling tool we developed to explore the suggestive interface idea. Our experience shows that it is quite promising. Our initial concern was that too many suggestions might be generated, confusing the user. However, if the system is carefully designed so that most suggestion engines have mutually exclusive input patterns, users see only a few suggestions at a time and can control the system fluently. Our informal test users understood the interface quickly and created various 3D drawings successfully.

Although our original goal was to improve gestural interfaces, we believe suggestive interfaces can also be useful in augmenting traditional command-based interfaces for 3D modeling. Hinting and suggestions encourage novice users to explore a new system and find unknown operations, and some operations that require combinations of commands can be specified naturally by only a few hints.

A demonstration video and the prototype program are available at www.cs.brown.edu/research/chateau.html or www.mtl.t.u-tokyo.ac.jp/~takeo/.

RELATED WORK

Many researchers are exploring possible next-generation user interfaces beyond current WIMP-style GUIs [23]. A common observation is that next-generation user interfaces should be context-aware in order to reduce the number of explicit command operations required [21]. This paper reports our experimental effort to implement context-awareness in the domain of 3D modeling.

Our interface is similar to predictive interfaces [5][19] in that the system (or an agent) suggests possible subsequent operations, but prior efforts have focused on operation histories to facilitate repetitive operations, while our system suggests various predefined operations based on the static configuration of the user-provided hints.

Multiple candidates are commonly used in recognition-based systems such as handwriting or speech recognition to solve the inherent ambiguity problem [16]. Japanese text-entry interfaces rely on multiple candidates to input thousands of characters using a limited number of keys [18]. In computer graphics, multiple candidates have been used to find desired parameter settings in a large parameter space [22], most recently in the Design Galleries work [17]. Typically, however, galleries represent samples of a large continuous space of possibilities, while our suggestions work with a small discrete space of possibilities.

Some constraint-based drawing systems infer geometric constraints from the user’s operations. Briar [7] and ROCKIT [13] infer graphical constraints based on a user’s dragging operation, and allow the user to select from several candidate constraints. Hudson and Hsi presented a system that infers layout algorithms by generalizing examples provided by the user [9]. The system presents multiple candidates for the generalization and lets the user select the desired one. While these systems infer hidden relationships or rules in a programming-by-example manner [6], our system constructs static scenes using a simple pattern-matching method [14].

Suggestive user interfaces extend the notions of beautification and prediction introduced in the Pegasus system [10,11]. Pegasus beautifies hand-drawn strokes by inferring desired geometric relationships, and predicts the next operation based on the surrounding context. It also generates multiple candidates to facilitate these processes. One problem is that too many candidates are offered as the scene becomes complicated. We address this “candidate explosion” by introducing an explicit hinting mechanism. To prevent clutter, we also primarily use visual thumbnails instead of presenting candidates in the scene.

Gesture-based interfaces, frequently used in 2D pen-based applications [8,15,20], recognize specific stroke shapes as gestures and replace them with predefined primitives or invoke editing operations such as undo. The SKETCH system [25] introduced a gesture-based interface for making 3D scenes consisting of stacked geometric primitives. Teddy [12] used a gesture-based interface for freeform 3D modeling. Our goal is to extend these systems to increase scalability and to support geometric relations such as symmetry and congruence.

THE USER INTERFACE

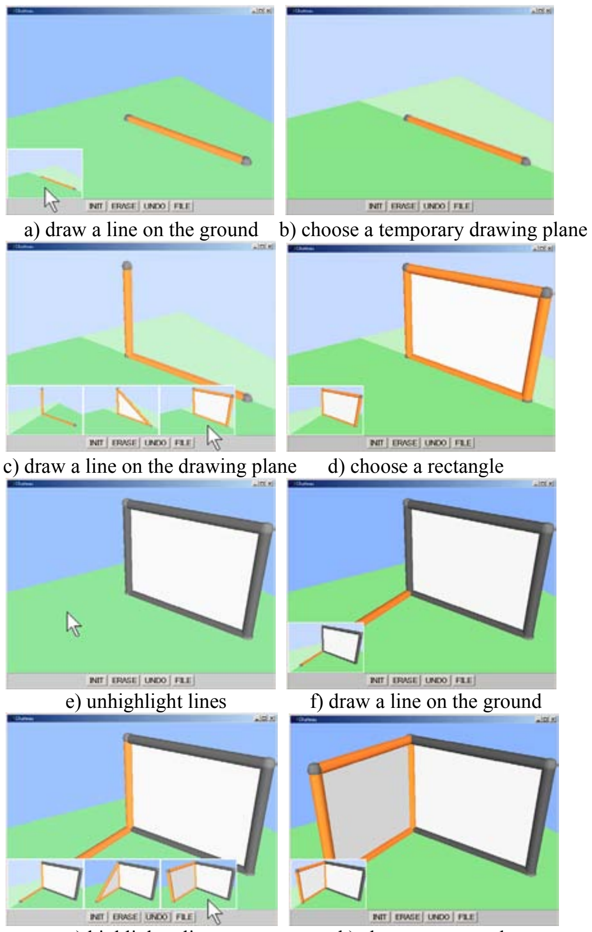
The user constructs 3D scenes by drawing 2D lines on the screen. The system converts 2D lines on the screen into 3D lines by projecting them onto 3D elements already in the scene. Prediction and suggestion mechanisms facilitate this drawing process by inferring possible subsequent operations. Highlighting plays an essential role throughout; highlighted lines guide the snapping mechanism for drawing lines and provide hints for prediction and suggestions. This section introduces the basics of the modeling system and then describes the prediction and suggestion mechanisms in detail.

A First Example

Suppose that a user wants to create two adjoining walls of a room, i.e., the model shown in Figure 2h. We'll briefly describe WHAT she does and her intention at each stage (i.e., WHY), and then, in the following sections, give further details and examples.

At the start of a modeling session, the user sees a ground plane. She wants to create a wall that meets this plane, so she draws a line segment on the plane to begin with: she clicks at some point, drags to the right, and releases. This creates a segment on the ground and automatically highlights it. The single highlighted line causes a candidate operation to be offered: the system offers to create a drawing-plane that's perpendicular to the ground and passes through the line (Figure 2a). Because the user wants to draw a wall in just such a plane, she clicks on the candidate and the transparent drawing plane appears (Figure 2b). Now she again clicks on the same starting point, drags a line upwards on the screen and releases, which creates a second line perpendicular to the first and highlights it as well; because both segments are highlighted, the system offers a candidate operation — the creation of a rectangle in the drawing plane (Figure 2c). This candidate is ideal, so she clicks on the thumbnail to make it happen (Figure 2d). She now wants to draw a new "aseline" on the ground plane, so she first clicks on the background to unhighlight all lines (Figure 2e). She then clicks on the ground plane some distance in front of the first click point and drags back towards it and releases the mouse over it. A new line appears and is highlighted (Figure 2f). Finally, highlighting (by clicking) the first vertical line she drew makes the system offer a rectangle in the new drawing plane as a candidate (Figure 2g), which she selects by clicking on the thumbnail, resulting in the model shown in Figure 2h.

Thus the basic operations are "dragging out lines segments," clicking on things to highlight/unhighlight them, and clicking on thumbnail "candidates" to choose them.



a) draw a line on the ground b) choose a temporary drawing plane
c) draw a line on the drawing plane d) choose a rectangle
e) unhighlight lines f) draw a line on the ground
g) highlight a line h) choose a rectangle

Figure 2: A first example.

Basics

Chateau currently supports the construction of 3D scenes consisting of straight line segments and planar polygons (curves and circles are not yet supported). Each line segment (called a *line*) is defined by two terminal vertices (called *joints*). Polygons (called *plates*) are always surrounded by lines. The ground plane is always visible and the user begins construction of every model by drawing a line on the ground.

All modeling operations are effected by left-mouse-button clicks and drags in the main screen. The right mouse button is reserved for camera control, for which we use the UniCam interface [24]. Only a few GUI buttons (clear, erase, undo) are provided on the screen. Our system requires no keyboard operation, and hence supports one-handed operation on devices like hand-held notepads.

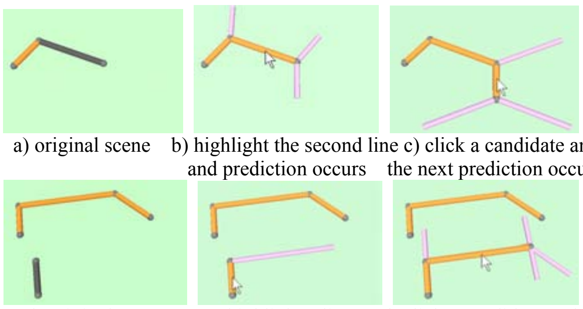
Highlighting plays an essential role in our system: the user controls snapping, prediction, and suggestion by highlighting appropriate lines as hints. The user highlights a line on the screen by clicking on it. If the user clicks an already highlighted line, it is unhighlighted. When the user double-clicks a line, the system highlights all lines connected to it. Any newly drawn lines are automatically highlighted. The user can unhighlight all lines by clicking on the ground or the background. When the user clicks on a plate, the system highlights all its edges.

The user draws a new line on a plate or the ground plane in the 3D scene by a dragging operation. To be precise, the system first finds the foremost plate or plane under the mouse cursor at the beginning of dragging, and projects the line on the plate or plane. The end points snap to existing lines and their end points [2] on the plate or plane. We also implemented a "drafting assistant" mechanism [1] whereby the user can activate additional snapping constraints by touching a line during the dragging operation. For example, if the user touches the midpoint of a line, the mouse cursor starts to snap to the perpendicular bisector of the line. Furthermore, snapping is affected by the highlighted lines; it guides the user to draw lines that are parallel or congruent[2] to the highlighted lines. In addition to the visible plates and the ground plane, the user can draw a line on a *temporary drawing plane*, so that lines can be drawn floating in the air [3]. A temporary drawing plane is activated by the suggestive interface mechanism described later, and appears as an translucent plane in the display. The user erases a line or plate by a scribbling gesture (moving the mouse cursor back and forth while dragging). The "erase" button on the screen erases all highlighted lines at once.

Predictions

A prediction mechanism like that in the Pegasus system [11] predicts the next lines to be drawn around the most recently highlighted line and presents multiple candidates as purple lines in the 3D scene. (This can be seen as a very specialized version of suggestion; its rules are so simple and it's so often applicable that its candidates are shown in the 3D scene rather than as thumbnails.) While Pegasus uses all lines in the scene as the context information for prediction, Chateau uses only the highlighted lines, which significantly reduces the number of candidates generated. Specifically, Chateau generates the flipped duplications of the highlighted lines connected to the most recently highlighted line (Figure 3a-c). It also searches for a reference line that is congruent to the most recently highlighted line, and copies the lines connected to the reference line around the most recently highlighted line (Figure 3d-f). The user can click a candidate to adopt it or simply proceed to the next operation to ignore the prediction. This prediction mechanism helps users draw locally symmetric or congruent structures. Prediction and suggestion are always active, but for clarity we suppress prediction in the remaining figures.

[2] Here, congruence means translational congruence and does not include rotational congruence.

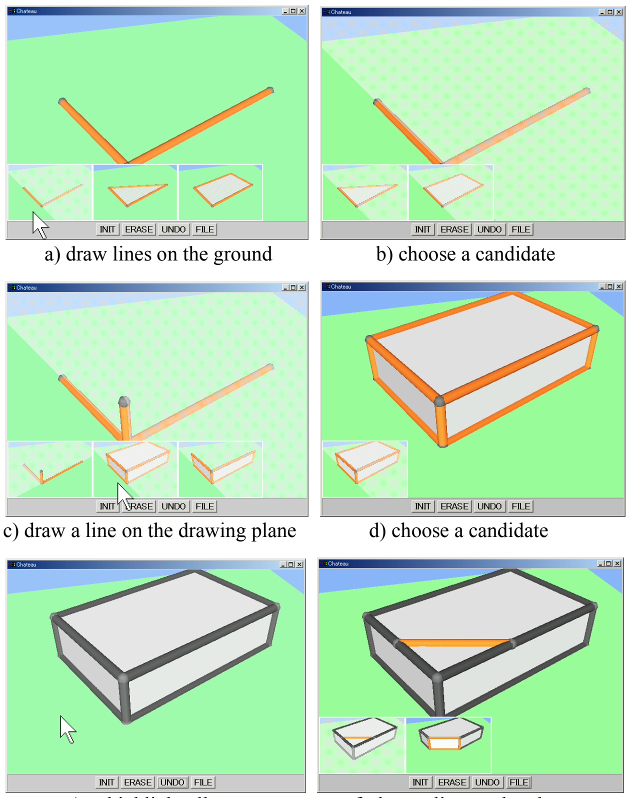

a) original scene b) highlight the second line and prediction occurs c) click a candidate and the next prediction occurs

d) original scene e) highlight a line and prediction occurs f) click a candidate and the next prediction occurs

Figure 3: The prediction mechanism.

Suggestions

Chateau generates suggestions whenever the user adds, erases, highlights, or unhighlights a line. The system automatically infers possible next operations based on the configuration of the highlighted lines, and presents the results of the operations as an array of thumbnails (Figure 1). The user can either ignore these or adopt one by clicking the thumbnail. The user can also "preview" the result as a large image in the main screen by dragging the mouse cursor across the thumbnails. The operation is finalized when the user releases the mouse button over the desired thumbnail.



a) draw lines on the ground b) choose a candidate

c) draw a line on the drawing plane d) choose a candidate

e) unhighlight all f) draw a line on the plate

Figure 4: Example operation sequence.

Figure 4 shows an example operation sequence. The user first draws two lines on the ground and the system presents three suggestions (a). Then she chooses the leftmost suggestion, which creates a new drawing plane (b). She draws the third line on the drawing plane and the system presents three new suggestions (c). She chooses to make a box (d). She unhighlights everything by clicking on the ground (e). She draws a line on the box, and the system shows two candidates (f), including one that suggests chamfering.

Candidates are generated by a set of *suggestion engines*. Each engine observes the scene, and when the current scene configuration matches its input test pattern it returns the updated scene as a candidate (Figure 5). The current implementation duplicates the entire scene for each candidate instead of maintaining a progressive data structure. The behavior of an individual suggestion engine can be seen as a variation of the constraint-based search-and-replace operation in the Chimera system [14], but our engines focus only on the highlighted lines instead of pattern-matching against the entire scene. When a suggestion is created, a thumbnail is rendered as an offscreen image, using the same camera parameters (i.e., view) as in the main window. For efficiency, we use fixed bitmaps for the thumbnails, which therefore do not update as the main-window view is changed.

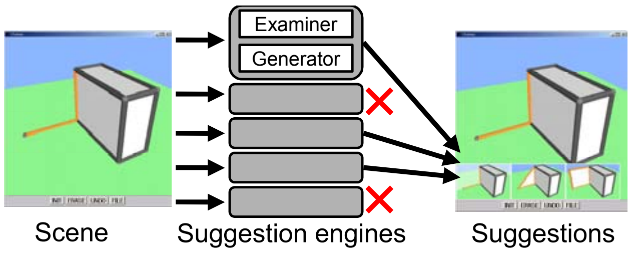


Figure 5: Suggestion engines observe the scene and return candidates when the scene matches their input patterns.

Figure 6 shows our current list of engines, S1 to S20. The first two suggestion engines support fundamental operations. S1 creates a temporary drawing plane to let the user draw lines in the air. If the most recently highlighted two lines are on a single plane, the system offers it as the next drawing plane. If not, the system offers a plane that contains the last-highlighted line and is perpendicular to the current drawing plane. S1 always returns a suggestion unless the resulting plane is identical to the ground plane. S2 creates a plate in a planar loop of highlighted lines.

All modeling operations can be achieved using just the basic drawing operations and the two engines just described. All the other suggestions can be seen as "assistants" that facilitate typical modeling tasks. For example, instead of using S4, the user *could* draw a box by drawing 12 lines and making 6 plates manually. We briefly describe the behavior of the suggestion engines to supplement the visual description in Figure 6.

S3 and S4 respond to two/three highlighted lines connected perpendicularly to one another. S5 and S6 respond to a highlighted line that is perpendicular to the plane containing all other highlighted lines; S6 responds only when the remaining lines form a loop, in which case the top vertex is positioned over the loop's center. S7 responds when the last-highlighted line overlaps a line in the highlighted group. (A group is a set of highlighted lines and plates connected to one another.) S8 responds to two sets of highlighted lines when each set lies on a plane and each line in a set has a parallel partner in the other set. S9 responds when the extrusions from the planar highlighted lines hit an existing plate (this is useful, for example, in making the legs of a table). S10 and S11 respond to highlighted lines that touch the edges of a polyhedron. Specifically, S10 requires that the two edges touched by the highlighted line share a vertex and that the vertex be shared by three plates. There must also be another plate at the opposite side. S11 requires that the highlighted lines form a planar loop and that all highlighted lines be on plates surrounding a corner. S12 responds to two parallel highlighted lines, of which one is an edge of a plate and the other touches the edges next to it. S13 responds to two intersecting lines (this is useful for trimming operations). S14 responds when the last-highlighted line is isolated from the highlighted group and is congruent with a line in the group. S15 is similar, but responds when the highlighted line is the mirror copy of the corresponding line. S16 responds when two congruent groups are highlighted, and therefore appears whenever the user has adopted an S14 suggestion. S17 responds to sequences of parallel lines such that the gaps between corresponding segments are nearly equal. S18 responds when three congruent groups or lines are linearly aligned. It generates equally spaced copies of the group between the external two as hinted by the middle one. S19 responds to irregular "stairs" (a repeated sequence of mutually perpendicular lines). S20 responds to three lines of equal length sharing a vertex when two of the joint angles are equal. This engine is useful in drawing regular polygons.

The particular choice of engines was determined by our needs as we experimented with the system and is clearly application-dependent. In a plumbing application, for example, it would be natural to have engines that created standard junctions (tees, unions, couplers, elbows, etc.).

In the current implementation, engines require exact matching in the examination phase. For example, S4 requires that all three edges to be exactly perpendicular and S17 requires that the pairs be exactly parallel. Alternatively, one can allow small deviations and *beautify* them after the operation [14]. We did not adopt this scheme in order to

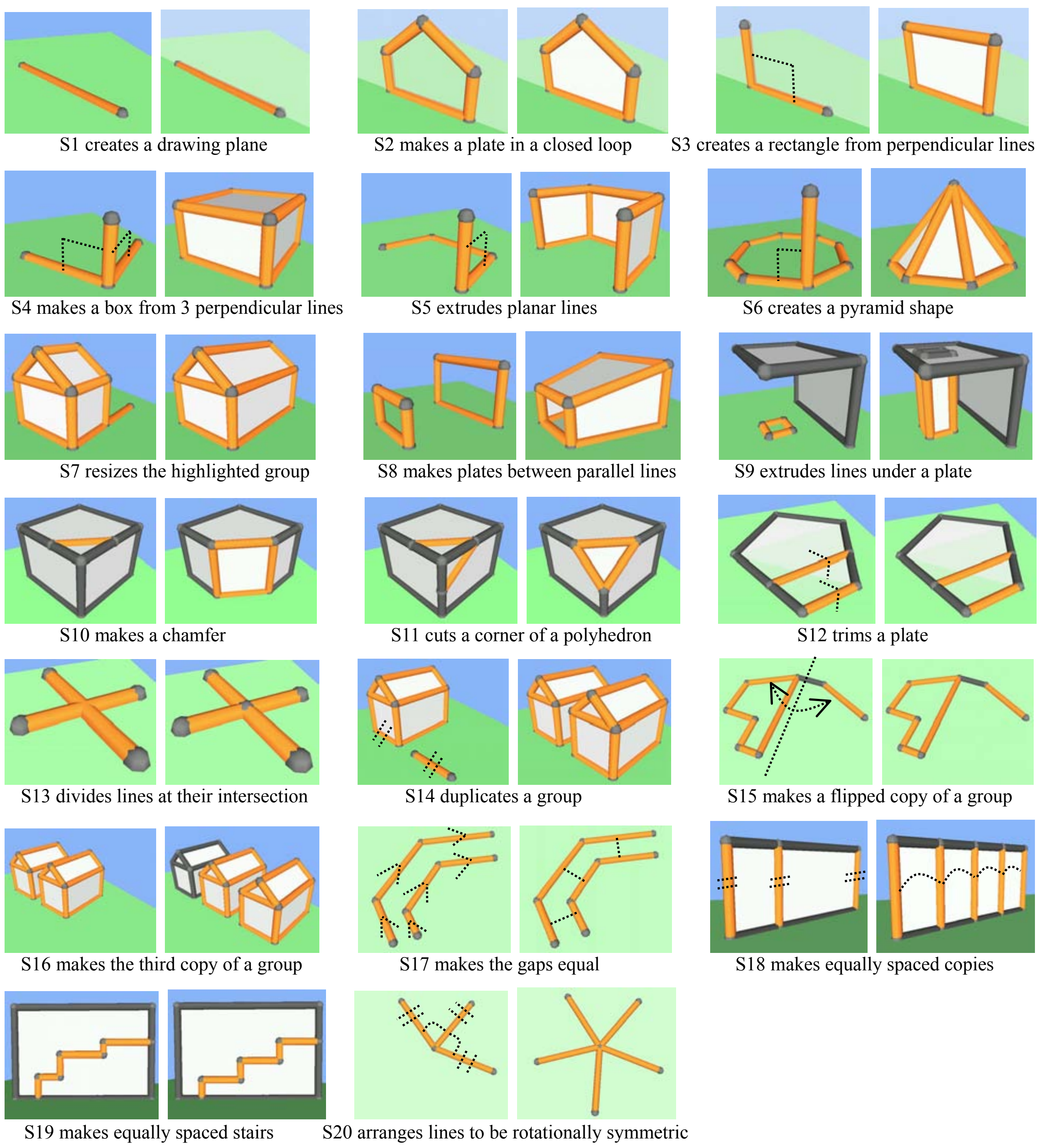

S1 creates a drawing plane
S2 makes a plate in a closed loop
S3 creates a rectangle from perpendicular lines
S4 makes a box from 3 perpendicular lines
S5 extrudes planar lines
S6 creates a pyramid shape
S7 resizes the highlighted group
S8 makes plates between parallel lines
S9 extrudes lines under a plate
S10 makes a chamfer
S11 cuts a corner of a polyhedron
S12 trims a plate
S13 divides lines at their intersection
S14 duplicates a group
S15 makes a flipped copy of a group
S16 makes the third copy of a group
S17 makes the gaps equal
S18 makes equally spaced copies
S19 makes equally spaced stairs
S20 arranges lines to be rotationally symmetric

Figure 6: Complete list of suggestions implemented in the current prototype (left: hints, right: result). (The dotted lines are added for clarity; they do not appear in the actual system.)

clearly distinguish the role of snapping/prediction and suggestions. Our design principle is to use snapping and prediction for satisfying basic relations such as congruence and parallelism, and to use suggestions for completing construction tasks. Another reason is that small deviations in the hints can make the result of suggestions ambiguous. For example, in the case of S4, the system has three options for positioning the resulting box if the three lines are not exactly perpendicular each other, and thus must ask the user to choose one among them.

To investigate the capability of a *pure* suggestive interface, we intentionally excluded traditional editing operations such as translation, rotation, and duplication. However, it is natural and useful to provide both command-based and suggestion-based operations in a single system. We envision that in practical applications, suggestive user interfaces will *augment* command-based interfaces.

IMPLEMENTATION

The Chateau system is implemented in Java (JDK1.1.5), and uses directX3 for 3D rendering. Suggestion engines (Java class files) are implemented as listeners that respond to changes in the scene configuration. An engine has an input examination part that determines whether it responds to the scene, and a suggestion generator that edits a copy of the scene to construct an updated scene. When the current scene matches an engine's input pattern, the engine returns the updated scene object and a thumbnail image (Figure 5). The implementation of suggestion engines is relatively simple because standard routines are provided by the base system. In the examination part, an engine checks the scene based using such criteria as the number of highlighted lines, connectivity, and spatial interrelationships. A typical suggestion engine's source code is between 100 and 200 lines.

It is essential to design suggestion engines carefully so that their input conditions are as mutually exclusive as possible. If many suggestion engines match a single scene configuration, they will generate many suggestions, confusing the user and cluttering the screen. With our current choice of engines, the system generates only a few suggestions at a time, showing that careful design can help prevent suggestion explosion. In the future, we will investigate the feasibility of the interface with many more suggestion engines.

USER EXPERIENCES

We have started an informal user study using the prototype system. Figure 7 shows examples of 3D models created by our test users, all of whom are graduate students in computer science. They learned the behavior of the system in approximately 30 minutes of tutorial and practice and created various models, including those shown in Figure 7, within a few hours. Test users were generally satisfied with the interface, but they wanted simple direct manipulation functions such as "move" and "rotate." Because of the limitations of the current implementation, the system gets too slow when the model becomes more complicated than these examples.

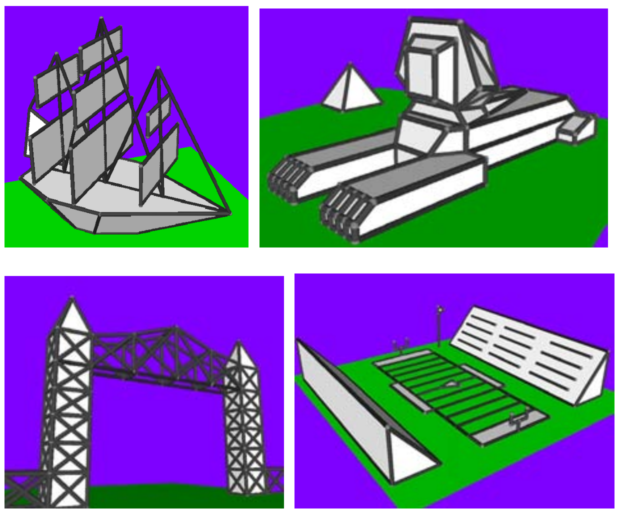


Figure 7: 3D drawings created by test users using Chateau.

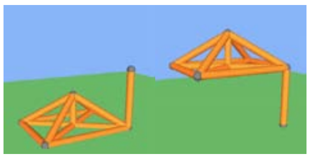
We also asked students in an advanced computer graphics class (including both graduate and undergraduate students) to test the prototype system and to implement their own suggestion engines as a part of an assignment. In general, they found the idea of a suggestive interface attractive and useful, but also felt that the current implementation requires substantial improvements. They wanted a better interface for controlling temporary drawing planes, appropriate feedback for camera control and snapping, the ability to turn off/on each feature, keyboard shortcuts for frequent operations, and command-based direct manipulation or 3D widgets for translation and rotation. This result suggests that a *pure* suggestive interface is not very practical, and that suggestion may be most effective when combined with traditional interfaces. We also asked them to list suggestion engines that they evaluated positively (useful) and negatively (useless or difficult to use). Table 1 summarizes the results. The basic engines (S1-S6) were popular, but other engines received mixed reactions reflecting large diversity in personal preferences.

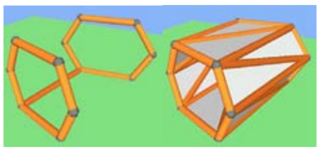
Table 1: Subjective evaluation of suggestion engines. The table shows the number of subjects who evaluated each suggestion positively or negatively. Six subjects provided answers.

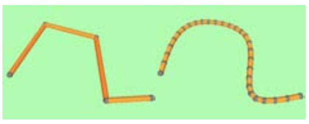
S	1	2	3	4	5	6	7	8	9
positive	3	5	6	5	5	3	1	1	0
negative	0	0	0	0	0	1	2	3	5

10	11	12	13	14	15	16	17	18	19	20
3	3	3	3	2	1	2	0	4	0	1
3	2	1	1	2	3	2	3	1	2	4

Figure 8 shows some suggestion engines implemented by the students. S21 takes a structure on the ground plane and a vertical line, and hoists the structure. S22 takes two closed loops that are not parallel, and makes a *tube* between the loops. S23 takes connected lines and returns a spline curve. S24 takes three lines in Y shape and fractalizes the Y. Overall, students implemented their own suggestion engines successfully, showing that one can extend the system as desired with reasonable effort.


S21 hoists structure.


S22 makes a tube.


S23 makes spline curve.

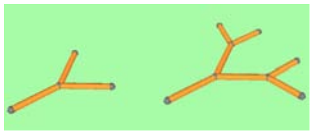

S24 makes fractal Y shapes

Figure 8: Examples of suggestion engines implemented by test users.

LIMITATIONS AND FUTURE WORK

Suggestive interfaces have some drawbacks: they can help promote serendipitous discovery of available operations, but they give a user no way to discover all operations directly, as "browsing the menus" can in a WIMP interface. If the hints given are inadequate, the system never responds and it is unclear to the user why the system is failing. A visual summary of suggestions, such as shown in Figure 6, is necessary for learning and reference. For operations with continuous parameters (e.g., shearing), there is no opportunity for partial feedback (such as a highlighted bounding box or parallelogram) during parameter adjustment. These operations may be best supported by a traditional direct-manipulation approach such as 3D widgets [4].

As with any experimental interaction technique, scalability is a major concern with the suggestive interface. One scalability problem is the complexity of the 3D scene. Although the hinting mechanism effectively limits the number of candidate suggestions compared with the simple search-entire-scene approach [11][14], complicated 3D scenes can make it difficult to specify hints and to find the desired one among small thumbnails. We need some advanced mechanisms such as grouping and locking to deal with complicated scenes.

Another scalability issue is related to the number of engines. The current suggestive interface system may not work well when hundreds of engines are implemented since the system may generate too many suggestions and confuse the user. We need refined mechanisms that automatically suppress inadequate engines based on the user's preferences, or let the user manually activate/inactivate specific engines. We also need to provide traditional command-based interfaces to perform complicated tasks.

The order of suggestion presentation is fixed in the current implementation: it is determined by the order in which the suggestion engines are implemented in the system, so S1 always appears first, S2 (if appropriate) second, and so on. We could instead first display recently used suggestions, or sort the suggestions based on the current context, or organize suggestions into a hierarchy. The value of such approaches will have to be determined through careful user studies.

In the near future, we will extend the current interface to support circles and curves. We plan to implement suggestion engines that construct cylinders, revolved surfaces, and rounded corners. In the longer term, we hope to use a suggestive user interface in a sketch-based freeform modeling system [12].

One advantage of the suggestive interface is extensibility. Users can customize the interface by adding their own special-purpose engines to the system. In the current implementation the user must write Java code, but we hope to provide an end-user programming environment, possibly an example-based framework [6].

Suggestive interfaces can be useful in various other graphical applications such as 2D bitmap editors and graph drawing programs. For example, if the user highlights almost-aligned objects in a 2D drawing program, the system might suggest an aligning operation, and it would be natural in a graph-drawing program to support even spacing of nodes or replication of selected subgraphs. Indeed, we believe that the ease of describing useful suggestions for a variety of applications indicates the promise of suggestive interfaces.

ACKNOWLEDGMENTS

We would like to thank the Brown University computer graphics group, especially Bob Zeleznik and Andy Forsberg, for thoughtful discussions, and the CMU stage3 research group, especially Dennis Cosgrove, for allowing us to use their Jalice scenegraph. We also thank Brown CS224 students for testing the system and providing valuable feedback.

REFERENCES

1. Ashlar Vellum Products, Ashlar Inc., http://www.ashlar.com/
2. E.A. Bier and M.C. Stone. Snap Dragging. *Computer Graphics*, Vol. 20, No. 4, pp. 233-240, 1986.
3. J.M. Cohen, L. Markosian, R.C. Zeleznik, J.F. Hughes, and R. Barzel. An Interface for Sketching 3D Curves.

1999 Symposium on Interactive 3D Graphics, pp. 17-21, 1999.

4. D.B. Conner, S.S. Snibbe, K.P. Herndon, D.C. Robbins, R.C. Zeleznik, and A. van Dam. Three-Dimensional Widgets. *1992 Symposium on Interactive 3D Graphics*, pp. 183-188, 1992.
5. A. Cypher. Eager: Programming Repetitive Tasks by Example. *Proceedings of CHI'91,* pp.33-39, 1991.
6. A. Cypher. *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press. 1993.
7. M. Gleicher and A. Witkin. Drawing with constraints. *The Visual Computer*, Vol. 11, No. 1, pp. 39-51, 1994.
8. M.D. Gross and E.Y.L. Do. Ambiguous Intentions: A Paper-like Interface for Creative Design. *Proceedings of UIST'96*, pp. 183-192, 1996.
9. S. Hudson and C. Hsi. A Synergistic Approach to Specifying Simple Number Independent Layouts by Example, *Proceedings of INTERCHI'93*, pp. 285-292, 1993.
10. T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Interactive Beautification: A Technique for Rapid Geometric Design. *Proceedings of UIST'97*, pp. 105-114, 1997.
11. T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Pegasus: A Drawing System for Rapid Geometric Design. *CHI'98 Summary*, pp. 24-25, 1998.
12. T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *SIGGRAPH 99 Conference Proceedings*, pp. 409-416, 1999.
13. S. Karsenty, J.A. Landay, and C. Weikart. Inferring Graphical Constraints with Rockit, *Proceedings of HCI'92*, 1992.
14. D. Kurlander and S. Feiner. Interactive Constraint-Based Search and Replace. *Proceedings of CHI'92*, pp. 609-618, 1992.
15. J.A. Landay and B.A. Myers. Interactive Sketching for the Early Stages of User Interface Design. *Proceedings of CHI'95*, pp. 43-50, 1995.
16. J. Mankoff, S.E. Hudson and G.D. Abowd. Interaction Techniques for Ambiguity Resolution in Recognition-based Interfaces. *Proceedings of UIST'00*, pp. 11-20, 2000.
17. J. Marks, B. Andalman, P. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. *SIGGRAPH 97 Conference Proceedings*, pp. 389-400, 1997.
18. T. Masui. An Efficient Text Input Method for Pen-based Computers. *Proceedings of CHI'98*, pp. 328-335, 1998.
19. D. Maulsby, I.H. Witten and K.A. Kittlitz. Metamouse: Specifying Graphical Procedures by Example. *Proceedings SIGGRAPH'89*, pp. 127-136, 1989.
20. T.P. Moran, P. Chiu, W. van Melle, and G. Kurtenbach. Pen-based Interaction Techniques for Organizing Material on an Electronic Whiteboard. *Proceedings of UIST'97*, pp. 45-54, 1997.
21. J. Nielsen. Noncommand User Interfaces. *Communications of the ACM*, Vol. 36, No. 4, pp. 83-99, 1993.
22. K. Sims. Artificial Evolution for Computer Graphics. *SIGGRAPH 91 Conference Proceedings*, pp. 319-328, 1991.
23. A. van Dam. Post-WIMP User Interfaces, *Communications of the ACM*, Vol. 40, No. 2, pp. 63-67, 1997.
24. R.C. Zeleznik and A. Forsberg. UniCam — 2D Gestural Camera Controls for 3D Environments. *Proceedings of 1999 Symposium on Interactive 3D Graphics*, 1999.
25. R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. *SIGGRAPH 96 Conference Proceedings*, pp. 163-170, 1996.

Teddy: A Sketching Interface for 3D Freeform Design

Takeo Igarashi[†], Satoshi Matsuoka[‡], Hidehiko Tanaka[†]

[†] University of Tokyo, [‡] Tokyo Institute of Technology

Abstract

We present a sketching interface for quickly and easily designing freeform models such as stuffed animals and other rotund objects. The user draws several 2D freeform strokes interactively on the screen and the system automatically constructs plausible 3D polygonal surfaces. Our system supports several modeling operations, including the operation to construct a 3D polygonal surface from a 2D silhouette drawn by the user: it inflates the region surrounded by the silhouette making wide areas fat, and narrow areas thin. Teddy, our prototype system, is implemented as a Java™ program, and the mesh construction is done in real-time on a standard PC. Our informal user study showed that a first-time user typically masters the operations within 10 minutes, and can construct interesting 3D models within minutes.

CR Categories and Subject Descriptions: I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction Techniques; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms

Additional Keywords: 3D modeling, sketching, pen-based systems, gestures, design, chordal axes, inflation

1 INTRODUCTION

Although much progress has been made over the years on 3D modeling systems, they are still difficult and tedious to use when creating freeform surfaces. Their emphasis has been the precise modeling of objects motivated by CAD and similar domains. Recently SKETCH [29] introduced a gesture-based interface for the rapid modeling of CSG-like models consisting of simple primitives.

This paper extends these ideas to create a sketching interface for designing 3D freeform objects. The essential idea is the use of freeform strokes as an expressive design tool. The user draws 2D freeform strokes *interactively* specifying the silhouette of an object, and the system automatically constructs a 3D polygonal surface model based on the strokes. The user does not have to manipulate control points or combine complicated editing operations. Using our technique, even first-time users can create simple, yet expressive 3D models within minutes. In addition, the resulting models have a hand-crafted feel (such as sculptures and stuffed

{takeo, tanaka}@mtl.t.u-tokyo.ac.jp, matsu@is.titech.ac.jp

http://mtl.t.u-tokyo.ac.jp/~takeo

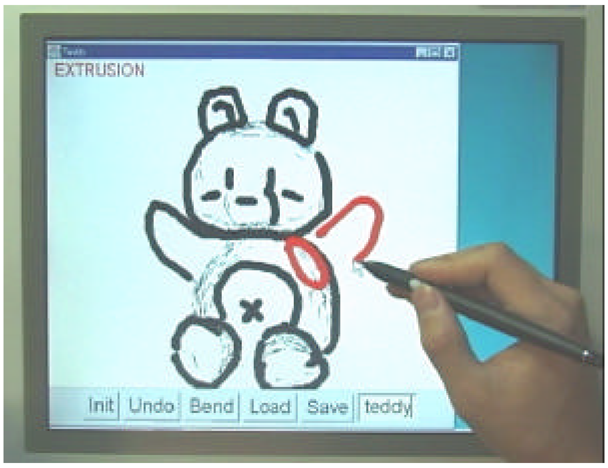


Figure1: Teddy in use on a display-integrated tablet.

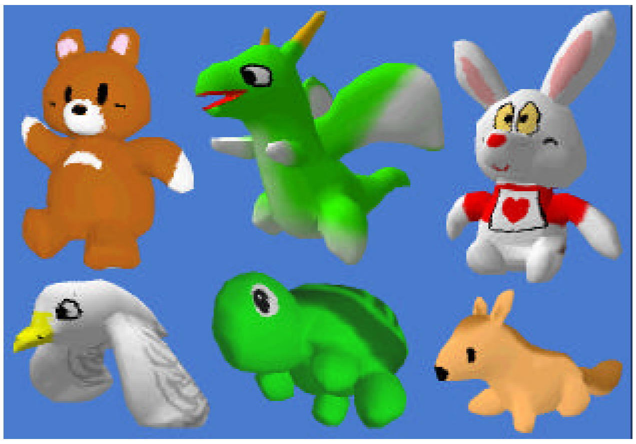


Figure 2: Painted models created using Teddy and painted using a commercial texture-map editor.

animals) which is difficult to accomplish with most conventional modelers. Examples are shown in Figure 2.

We describe here the sketching interface and the algorithms for constructing 3D shapes from 2D strokes. We also discuss the implementation of our prototype system, Teddy. The geometric representation we use is a standard polygonal mesh to allow the use of numerous software resources for post-manipulation and rendering. However, the interface itself can be used to create other representations such as volumes [25] or metaballs [17].

Like SKETCH [29], Teddy is designed for the rapid construction of approximate models, not for the careful editing of precise models. To emphasize this design goal and encourage creative exploration, we use the real-time pen-and-ink rendering described in [16], as shown in Figure 1. This also allows real-time interactive rendering using Java on mid-range PCs without

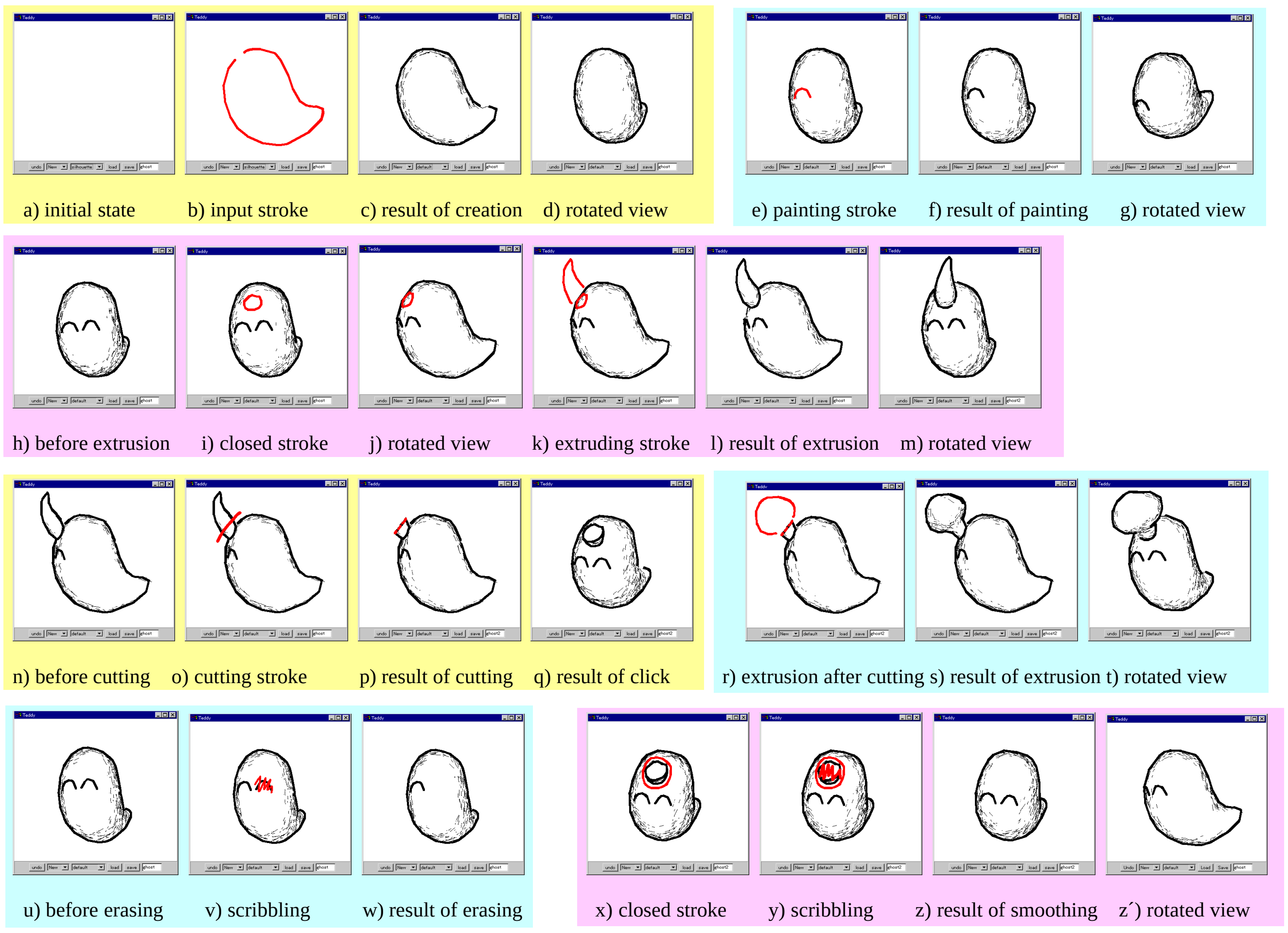


Figure 3: Overview of the modeling operations.

dedicated 3D rendering hardware.

An obvious application of Teddy is the design of 3D models for character animation. However, in addition to augmenting traditional 3D modelers, Teddy's ease of use has the potential to open up new application areas for 3D modeling. Possibilities include rapid prototyping in the early stages of design, educational/recreational use for non-professionals and children, and real-time communication assistance on pen-based systems.

The accompanying videotape demonstrates Teddy's user interface. Teddy is available as a Java applet at the following web site. http://www.mtl.t.u-tokyo.ac.jp/~takeo/teddy/teddy.htm

2 RELATED WORK

A typical procedure for geometric modeling is to start with a simple primitive such as a cube or a sphere, and gradually construct a more complex model through successive transformations or a combination of multiple primitives. Various deformation techniques [15,23] and other shape-manipulation tools [8] are examples of transformation techniques that let the user create a wide variety of precise, smooth shapes by interactively manipulating control points or 3D widgets.

Another approach to geometric modeling is the use of implicit surfaces [3,18]. The user specifies the skeleton of the intended model and the system constructs smooth, natural-looking surfaces around it. The surface inflation technique [17] extrudes the polygonal mesh from the skeleton outwards. In contrast, our approach lets the user specify the silhouette of the intended shape directly instead of by specifying its skeleton.

Some modeling systems achieve intuitive, efficient operation using 3D input/output devices [6]. 3D devices can simplify the operations that require multiple operations when using 2D devices.

Our sketching interface is inspired by previous sketch-based modeling systems [7,29] that interpret the user's freeform strokes and interactively construct 3D rectilinear models. Our goal is to develop a similar interface for designing rounded freeform models.

Inflation of a 2D drawing is introduced in [27], and 3D surface editing based on a 2D painting technique is discussed in [28]. Their target is basically a 2D array with associated height values, rather than a 3D polygonal model.

The use of freeform strokes for 2D applications has recently become popular. Some systems [10,14] use strokes to specify gestural commands and others [2] use freeform strokes for specifying 2D curves. These systems find the best matching arcs or splines automatically, freeing the users from explicit control of underlying parameters.

We use a polygonal mesh representation, but some systems use a volumetric representation [9,25], which is useful for designing topologically complicated shapes. Our mesh-construction algorithm is based on a variety of work on polygonal mesh manipulation, such as mesh optimization [12], shape design [26], and surface fairing [24], which allows polygonal meshes to be widely used as a fundamental representation for geometric

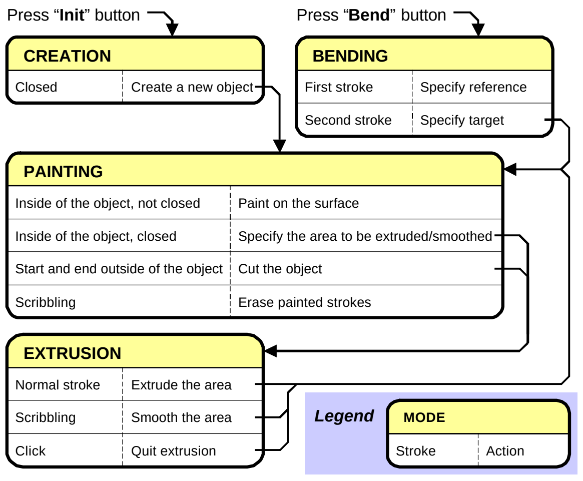


Figure 4: Summary of the gestural operations.

modeling and computer graphics in general.

3 USER INTERFACE

Teddy's physical user interface is based upon traditional 2D input devices such as a standard mouse or tablet. We use a two-button mouse with no modifier keys. Unlike traditional modeling systems, Teddy does not use WIMP-style direct manipulation techniques or standard interface widgets such as buttons and menus for modeling operations. Instead, the user specifies his or her desired operation using freeform strokes on the screen, and the system infers the user's intent and executes the appropriate editing operations. Our videotape shows how a small number of simple operations let the users create very rich models.

In addition to gestures, Teddy supports direct camera manipulation using the secondary mouse button based on a virtual trackball model [13]. We also use a few button widgets for auxiliary operations, such as save and load, and for initiating bending operations.

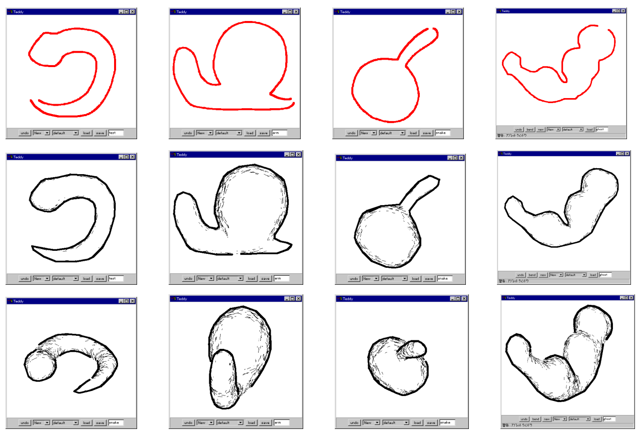
4 MODELING OPERATIONS

This section describes Teddy's modeling operations from the user's point of view; details of the algorithms are left to the next section. Some operations are executed immediately after the user completes a stroke, while some require multiple strokes. The current system supports neither the creation of multiple objects at once, nor operations to combine single objects. Additionally, models must have a spherical topology; e.g., the user cannot create a torus. An overview of the model construction process is given first, and then each operation is described in detail.

The modeling operations are carefully designed to allow incremental learning by novice users. Users can create a variety of models by learning only the first operation (creation), and can incrementally expand their vocabulary by learning other operations as necessary. We have found it helpful to restrict first-time users to the first three basic operations (creation, painting, and extrusion), and then to introduce other advanced operations after these basic operations are mastered.

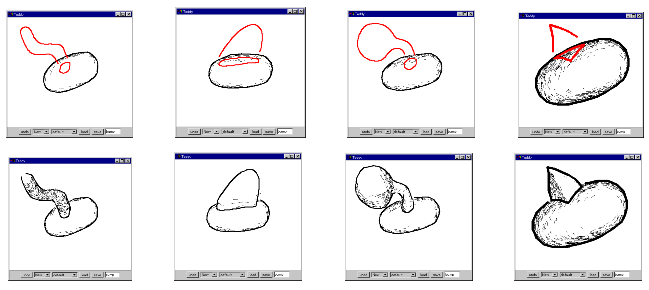
4.1 Overview

Figure 3 introduces Teddy's general model construction process. The user begins by drawing a single freeform stroke on a blank canvas (Figures 3a-b). As soon as the user finishes drawing the stroke, the system automatically constructs a corresponding 3D



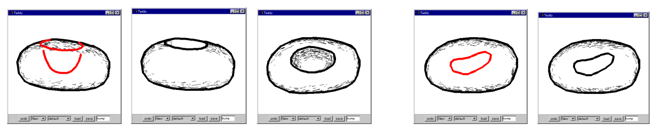
a) snake b) snail c) cherry d) muscular arm

Figure 5: Examples of creation operation (top: input stroke, middle: result of creation, bottom: rotated view).



a) long b) thin c) fat d) sharp

Figure 6: Examples of extrusion (top: extruding stroke, bottom: result of extrusion).

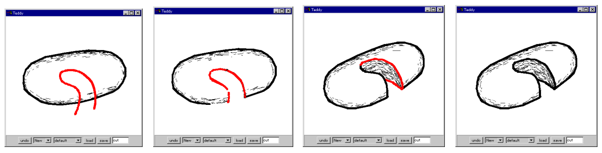


a) digging stroke b) result c) rotated d) closed stroke e) after click

Figure 7: More extrusion operations: digging a cavity (a-c) and turning the closed stroke into a surface drawing (d-e).

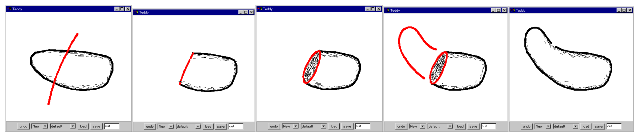
shape (c). The user can now view the model from a different direction (d). Once a model is created, it may be modified using various operations. The user can draw a line on the surface (e-g) by drawing a stroke within the model silhouette. If the stroke is closed, the resulting surface line turns red and the system enters "extrusion mode" (h-i). Then the user rotates the model (j) and draws the second stroke specifying the silhouette of the extruded surface (k-m). A stroke that crosses the silhouette cuts the model (n-o) and turns the cut section red (p). The user either clicks to complete the operation (q) or draws a silhouette to extrude the section (r-t). Scribbling on the surface erases the line segments on the surface (u-w). If the user scribbles during the extrusion mode (x-y), the system smoothes the area surrounded by the closed red line (z-z´).

Figure 4 summarizes the modeling operations available on the current implementation. Note that the appropriate action is chosen based on the stroke's position and shape, as well as the current



a) biting stroke b) result c) rotated view d) after click

Figure 8: Cutting operation.



a) cutting stroke b) result c) rotated d) extruding stroke e) result

Figure 9: Extrusion after cutting.

mode of the system.

4.2 Creating a New Object

Starting with a blank canvas, the user creates a new object by drawing its silhouette as a closed freeform stroke. The system automatically constructs a 3D shape based on the 2D silhouette. Figure 5 shows examples of input strokes and the corresponding 3D models. The start point and end point of the stroke are automatically connected, and the operation fails if the stroke is self-intersecting. The algorithm to calculate the 3D shape is described in detail in section 5. Briefly, the system inflates the closed region in both directions with the amount depending on the width of the region: that is, wide areas become fat, and narrow areas become thin. Our experience so far shows that this algorithm generates a reasonable-looking freeform shape. In addition to the creation operation, the user can begin model construction by loading a simple primitive. The current implementation provides a cube and a sphere, but adding more shapes is straightforward.

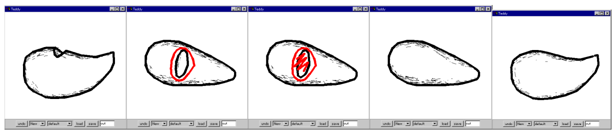
4.3 Painting and Erasing on the Surface

The object surface is painted by drawing a freeform stroke within the object's silhouette on the canvas (the stroke must not cross the silhouette) [11]. The 2D stroke is projected onto the object surface as 3D line segments, called surface lines (Figure 3e-g). The user can erase these surface lines by drawing a scribbling stroke[1] (Figure 3u-w). This painting operation does not modify the 3D geometry of the model, but lets the user express ideas quickly and conveniently when using Teddy as a communication medium or design tool.

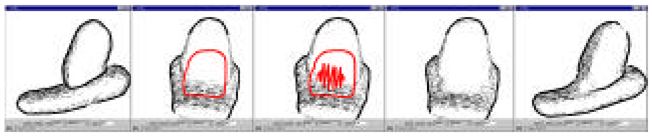
4.4 Extrusion

Extrusion is a two-stroke operation: a closed stroke on the surface and a stroke depicting the silhouette of the extruded surface. When the user draws a closed stroke on the object surface, the system highlights the corresponding surface line in red, indicating the initiation of "extrusion mode" (Figure 3i). The user then rotates the model to bring the red surface line sideways (Figure 3j) and draws a silhouette line to extrude the surface (Figure 3k). This is basically a sweep operation that constructs the 3D shape by moving the closed surface line along the skeleton of the silhouette

[1] A stroke is recognized as scribbling when `sl/pl > 1.5`, where `sl` is the length of the stroke and `pl` is the perimeter of its convex hull.

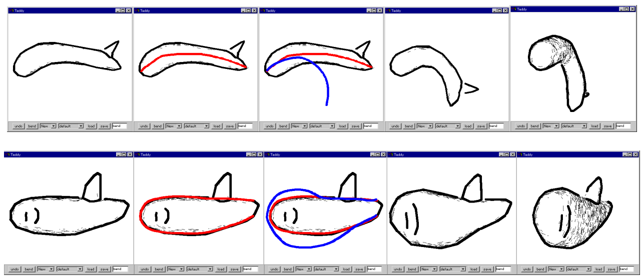


a) cleaning a cavity



b) smoothing a sharp edge

Figure 10: Smoothing operation.



a) original b) reference stroke c) target stroke d) result e) rotated

Figure 11: Examples of transformation (top: bending, bottom: distortion).

(Figure 3l-m). The direction of extrusion is always perpendicular to the object surface, not parallel to the screen. Users can create a wide variety of shapes using this operation, as shown in Figure 6. They can also make a cavity on the surface by drawing an inward silhouette (Figure 7a-c). The current implementation does not support holes that completely extend to the other side of the object. If the user decides not to extrude, a single click turns the red stroke into an ordinary painted stroke (Figure 7d-e).

4.5 Cutting

A cutting operation starts when the user draws a stroke that runs across the object, starting and terminating outside its silhouette (Figure 3o). The stroke divides the object into two pieces at the plane defined by the camera position and the stroke. What is on the screen to the left of the stroke is then removed entirely (Figure 3p) (as when a carpenter saws off a piece of wood). The cutting operation finishes with a click of the mouse (Figure 3q). The user can also `bite' the object using the same operation (Figure 8).

The cutting stroke turns the section edges red, indicating that the system is in "extrusion mode". The user can draw a stroke to extrude the section instead of a click (Figure3r-t, Figure 9). This "extrusion after cutting" operation is useful to modify the shape without causing creases at the root of the extrusion.

4.6 Smoothing

One often *smoothes* the surface of clay models to eliminate bumps and creases. Teddy lets the user smooth the surface by drawing a scribble during "extrusion mode." Unlike erasing, this operation modifies the actual geometry: it first removes all the polygons surrounded by the closed red surface line and then creates an entirely new surface that covers the region smoothly. This operation is useful to remove unwanted bumps and cavities (Figure 3x-z', Figure 10a), or to smooth the creases caused by earlier extrusion operations (Figure 10b).

4.7 Transformation

We are currently experimenting with an additional "transformation" editing operation that distorts the model while preserving the polygonal mesh topology. Although it functions properly, the interface itself is not fully gestural because the modal transition into the bending mode requires a button push.

This operation starts when the user presses the "bend" button and uses two freeform strokes called the *reference stroke* and the *target stroke* to modify the model. The system moves vertices of the polygonal model so that the spatial relation between the original position and the target stroke is identical to the relation between the resulting position and the reference stroke. This movement is parallel to the screen, and the vertices do not move perpendicular to the screen. This operation is described in [5] as *warp*; we do not discuss the algorithm further.

Transformation can be used to bend, elongate, and distort the shape (Figure 11). We plan to make the system infer the reference stroke automatically from the object's structure in order to simplify the operation, in a manner similar to the mark-based interaction technique of [2].

5 ALGORITHM

We next describe how the system constructs a 3D polygonal mesh from the user's freeform strokes. Internally, a model is represented as a polygonal mesh. Each editing operation modifies the mesh to conform to the shape specified by the user's input strokes (Figure 12). The resulting model is always topologically equivalent to a sphere. We developed the current implementation as a prototype for designing the interface; the algorithms are subject to further refinement and they fail for some illegal strokes (in that case, the system indicates the problem and requests an alternative stroke). However, these exceptional cases are fairly rare, and the algorithm works well for a wide variety of shapes.

Our algorithms for creation and extrusion are closely related to those for freeform surface construction based on skeletons [3,18], which create a surface around user-defined skeletons using implicit surface techniques. While our current implementation does not use implicit surfaces, they could be used in an alternative implementation.

In order to remove noise in the handwriting input stroke and to construct a regular polygonal mesh, every input stroke is re-sampled to form a smooth polyline with uniform edge length before further processing [4].

5.1 Creating a New Object

Our algorithm creates a new closed polygonal mesh model from the initial stroke. The overall procedure is this: we first create a closed planar polygon by connecting the start-point and end-point of the stroke, and determine the spine or axes of the polygon using

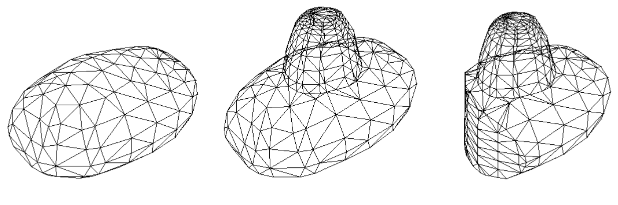


Figure 12: Internal representation.

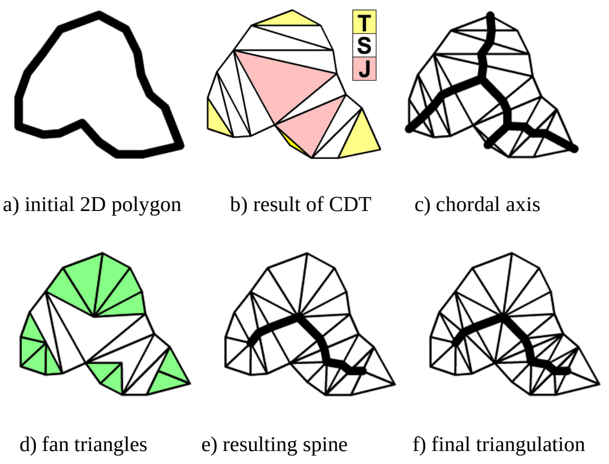


Figure 13: Finding the spine.

the *chordal axis* introduced in [21]. We then elevate the vertices of the spine by an amount proportional to their distance from the polygon. Finally, we construct a polygonal mesh wrapping the spine and the polygon in such a way that sections form ovals.

When constructing the initial closed planar polygon, the system makes all edges a predefined unit length (see Figure 13a). If the polygon is self-intersecting, the algorithm stops and the system requests an alternative stroke. The edges of this initial polygon are called *external edges*, while edges added in the following triangulation are called *internal edges*.

The system then performs constrained Delaunay triangulation of the polygon (Figure 13b). We then divide the triangles into three categories: triangles with two external edges (terminal triangle), triangles with one external edge (sleeve triangle), and triangles without external edges (junction triangle). The chordal axis is obtained by connecting the midpoints of the internal edges (Figure 13c), but our inflation algorithm first requires the *pruning* of insignificant branches and the retriangulation of the mesh. This pruning algorithm is also introduced in [21].

To prune insignificant branches, we examine each terminal triangle in turn, expanding it into progressively larger regions by merging it with adjacent triangles (Figure 14a-b). Let X be a terminal triangle; then X has two exterior edges and one interior edge. We erect a semicircle whose diameter is the interior edge, and which lies on the same side of that edge as does X. If all three vertices of X lie on or within this semicircle, we remove the interior edge and merge X with the triangle that lies on the other side of the edge.

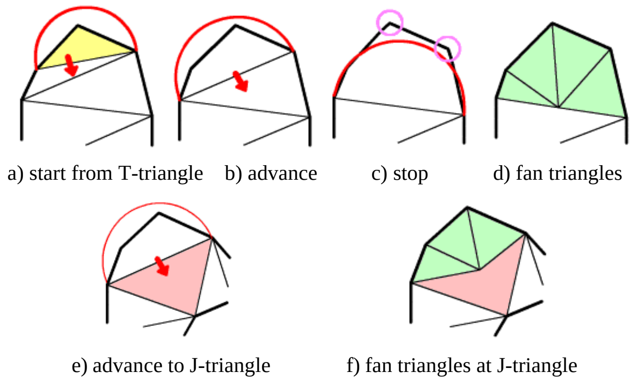


Figure 14: Pruning.

If the newly merged triangle is a sleeve triangle, then X now has three exterior edges and a new interior edge. Again we erect a semicircle on the interior edge and check that all vertices are within it. We continue until some vertex lies outside the semicircle (Figure 14c), or until the newly merged triangle is a junction triangle. In the first case, we triangulate X with a "fan" of triangles radiating from the midpoint of the interior edge (Figure 14d). In the second case, we triangulate with a fan from the midpoint of the junction triangle (Figure 14e-f). The resulting fan triangles are shown in Figure 13d. The pruned spine is obtained by connecting the midpoints of remaining sleeve and junction triangles' internal edges (Figure 13e).

The next step is to subdivide the sleeve triangles and junction triangles to make them ready for elevation. These triangles are divided at the spine and the resulting polygons are triangulated, so that we now have a complete 2D triangular mesh between the spine and the perimeter of the initial polygon (Figure 13f).

Next, each vertex of the spine is elevated proportionally to the average distance between the vertex and the external vertices that are directly connected to the vertex (Figure 15a,b). Each internal edge of each fan triangle, excluding spine edges, is converted to a quarter oval (Figure 15c), and the system constructs an appropriate polygonal mesh by sewing together the neighboring elevated edges, as shown in Figure 15d. The elevated mesh is copied to the other side to make the mesh closed and symmetric. Finally, the system applies mesh refinement algorithms to remove short edges and small triangles [12].

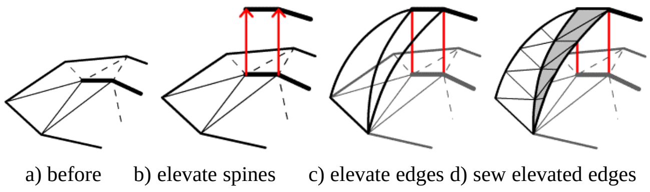


Figure 15: Polygonal mesh construction.

5.2 Painting on the Surface

The system creates surface lines by sequentially projecting each line segment of the input stroke onto the object's surface polygons. For each line segment, the system first calculates a bounded plane consisting of all rays shot from the camera through the segment on the screen. Then the system finds all intersections between the plane and each polygon of the object, and splices the resulting 3D line segments together (Figure 16). The actual implementation searches for the intersections efficiently using polygon connectivity information. If a ray from the camera crosses multiple polygons, only the polygon nearest to the camera position is used. If the resulting 3D segments cannot be spliced together (e.g., if the stroke crosses a "fold" of the object), the algorithm fails.

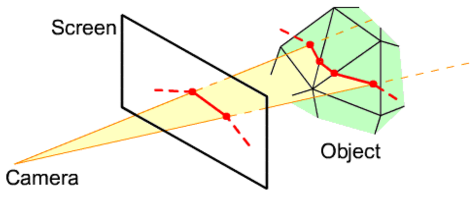


Figure 16: Construction of surface lines.

5.3 Extrusion

The extrusion algorithm creates new polygonal meshes based on a closed base surface line (called the base ring) and an extruding stroke. Briefly, the 2D extruding stroke is projected onto a plane perpendicular to the object surface (Figure 17a), and the base ring is swept along the projected extruding stroke (Figure 17b). The base ring is defined as a closed 3D polyline that lies on the surface of the polygonal mesh, and the normal of the ring is defined as that of the best matching plane of the ring.

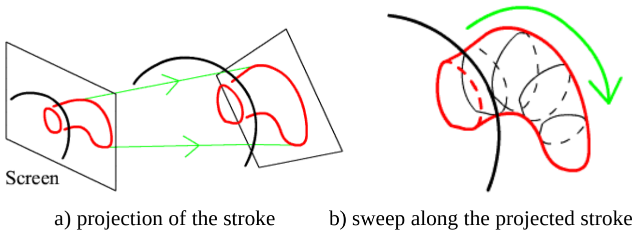

Screen
a) projection of the stroke
b) sweep along the projected stroke

Figure 17: Extrusion algorithm.

First, the system finds the plane for projection: the plane passing through the base ring's center of gravity and lying parallel to the normal of the base ring[2]. Under the above constraints, the plane faces towards the camera as much as possible (Figure 17a).

Then the algorithm projects the 2D extruding stroke onto the plane, producing a 3D extruding stroke. Copies of the base ring are created along the extruding stroke in such a way as to be almost perpendicular to the direction of the extrusion, and are resized to fit within the stroke. This is done by advancing two pointers (left and right) along the extruding stroke starting from both ends. In each step, the system chooses the best of the following three possibilities: advance the left pointer, the right pointer, or both. The *goodness* value increases when the angle between the line connecting the pointers and the direction of the stroke at each pointer is close to 90 degrees (Figure 18a). This process completes when the two pointers meet.

Finally, the original polygons surrounded by the base ring are deleted, and new polygons are created by sewing the neighboring copies of the base ring together [1] (Figure 18b). The system uses the same algorithm to dig a cavity on the surface.

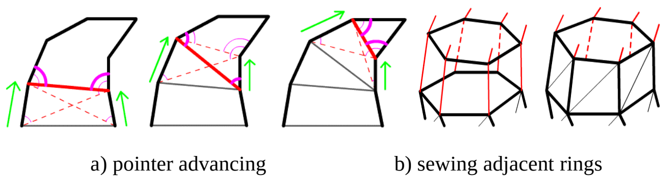


Figure 18: Sweeping the base ring.

This simple algorithm works well for a wide variety of extrusions but creates unintuitive shapes when the user draws unexpected extruding strokes or when the base surface is not sufficiently planar (Figure 19).

[2] The normal of the ring is calculated as follows: Project the points of the ring to the original XY-plane. Then compute the enclosed "signed area" by the formula:
`Axy = 0.5*sum(i=0, i=n-1, x[i]*y[i+1]-x[i+1]*y[i])`
(indices are wrapped around so that `x[n]` means `x[0]`).
Calculate `Ayx` and `Azx` similarly, and the vector `v=(Ayz,Azx,Axy)` is defined as the normal of the ring.

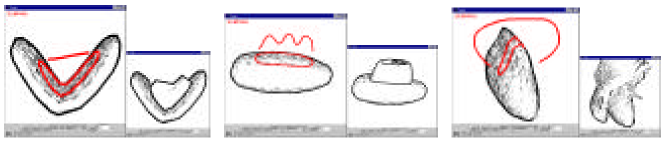

a) flat extrusion b) wavy extrusion c) wrapping extrusion

Figure 19: Unintuitive extrusions.

5.4 Cutting

The cutting algorithm is based on the painting algorithm. Each line segment of the cutting stroke is projected onto the front and back facing polygons. The system connects the corresponding end points of the projected edges to construct a planer polygon (Figure 20). This operation is performed for every line segment, and the system constructs the complete section by splicing these planer polygons together. Finally, the system triangulates each planer polygon [22], and removes all polygons to the left of the cutting stroke.

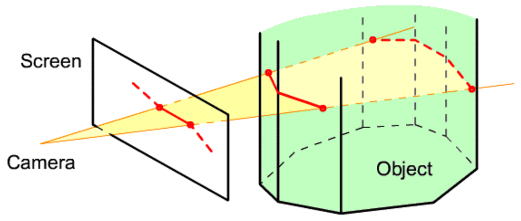


Figure 20: Cutting.

5.5 Smoothing

The smoothing operation deletes the polygons surrounded by the closed surface line (called a ring) and creates new polygons to cover the hole smoothly. First, the system translates the objects into a coordinate system whose Z-axis is parallel to the normal of the ring. Next, the system creates a 2D polygon by projecting the ring onto the XY-plane in the newly created coordinate system, and triangulates the polygon (Figure 21b). (The current implementation fails if the area surrounded by the ring contains creases and is folded when projected on the XY-plane.) The triangulation is designed to create a good triangular mesh based on [22]: it first creates a constrained Delaunay triangulation and gradually refines the mesh by edge splitting and flipping; then each vertex is elevated along the Z-axis to create a smooth 3D surface (Figure 21d).

The algorithm for determining the Z-value of a vertex is as follows: For each edge of the ring, consider a plane that passes through the vertex and the midpoint of the edge and is parallel to the Z-axis. Then calculate the z-value of the vertex so that it lies on the 2D Bezier curve that smoothly interpolates both ends of the ring on the plane (Figure 21c). The final z-value of the vertex is the average of these z-values.

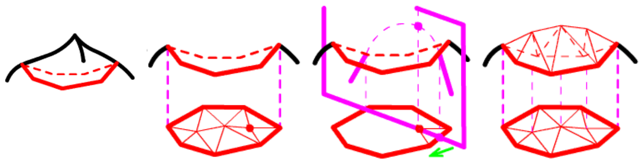

a) before b) triangulation c) calculating Z-value d) result

Figure 21: Smoothing algorithm.

Finally, we apply a surface-fairing algorithm [24] to the newly created polygons to enhance smoothness.

6 IMPLEMENTATION

Our prototype is implemented as a 13,000 line Java program. We tested a display-integrated tablet (Mutoh MVT-14, see Figure 1) and an electric whiteboard (Xerox Liveboard) in addition to a standard mouse. The mesh construction process is completely real-time, but causes a short pause (a few seconds) when the model becomes complicated. Teddy can export models in OBJ file format. Figure 2 shows some 3D models created with Teddy by an expert user and painted using a commercial texture-map editor. Note that these models look quite different from 3D models created in other modeling systems, reflecting the hand-drawn nature of the shape.

7 USER EXPERIENCE

The applet version of Teddy has undergone limited distribution, and has been used (mainly by computer graphics researchers and students) to create different 3D models. Feedback from these users indicates that Teddy is quite intuitive and encourages them to explore various 3D designs. In addition, we have started close observation of how first-time users (mainly graduate students in computer science) learn Teddy. We start with a detailed tutorial and then show some stuffed animals, asking the users to create them using Teddy. Generally, the users begin to create their own models fluently within 10 minutes: five minutes of tutorial and five minutes of guided practice. After that, it takes a few minutes for them to create a stuffed animal such as those in Figure 2 (excluding the texture).

8 FUTURE WORK

Our current algorithms and implementation are robust and efficient enough for experimental use. However, they can fail or generate unintuitive results when the user draws unexpected strokes. We must devise more robust and flexible algorithms to handle a variety of user inputs. In particular, we plan to enhance the extrusion algorithm to allow more detailed control of surfaces. We are also considering using implicit surface construction techniques.

Another important research direction is to develop additional modeling operations to support a wider variety of shapes with arbitrary topology, and to allow more precise control of the shape. Possible operations are creating creases, twisting the model, and specifying the constraints between the separate parts for animation systems [20]. While we are satisfied with the simplicity of the current set of gestural operations, these extended operations will inevitably complicate the interface, and careful interface design will be required.

9 ACKNOWLEDGEMENTS

This project was not possible without the support and encouragement of people. We would especially like to thank Marshall Bern, Lee Markosian, Robert C. Zeleznik, and Hiromasa Suzuki for their invaluable advice in the development of the system, and we thank John F. Hughes, Randy Pausch, Jeffrey S. Pierce, Joe Marks, Jock D. Mackinlay, Andrew Glassner, and Brygg Ullmer for their help in the preparation of this paper. We also thank the SIGGRAPH reviewers for their thoughtful comments. This work is supported in part by JSPS Research Fellowship.

References

1. G. Barequet and M Sharir. Piecewise-linear interpolation between polygonal slices. *ACM 10th Computational Geometry Proceedings*, pages 93-102, 1994.
2. T. Baudel. A mark-based interaction paradigm for free-hand drawing. *UIST'94 Conference Proceedings*, pages 185-192, 1994.
3. J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. *1990 Symposium on Interactive 3D Graphics*, pages 109-116, 1990.
4. J.M. Cohen, L. Markosian, R.C. Zeleznik, J.F. Hughes, and R. Barzel. An Interface for Sketching 3D Curves. *1999 Symposium on Interactive 3D Graphics*, pages 17-21, 1999.
5. W.T. Correa, R.J. Jensen, C.E. Thayer, and A. Finkelstein. Texture mapping for cel animation. *SIGGRAPH 98 Conference Proceedings*, pages 435-456, 1998.
6. M. Deering. The Holosketch VR sketching system. *Communications of the ACM*, 39(5):54-61, May 1996.
7. L. Eggli, C. Hsu, G. Elber, and B. Bruderlin, Inferring 3D models from freehand sketches and constraints. *Computer-Aided Design*, 29(2): 101-112, Feb.1997.
8. C.Grimm, D. Pugmire, M. Bloomental, J. F. Hughes, and E. Cohen. Visual interfaces for solids modeling. *UIST '95 Conference Proceedings*, pages 51-60, 1995.
9. T. Galyean and J.F. Hughes. Sculpting: an interactive volumetric modeling technique. *SIGGRAPH '91 Conference Proceedings*, pages 267-274, 1991.
10. M.D. Gross and E.Y.L. Do. Ambiguous intentions: A paper-like interface for creative design. *UIST'96 Conference Proceedings*, pages 183-192, 1996.
11. P. Hanrahan, P. Haeberli, Direct WYSIWYG Painting and Texturing on 3D Shapes, *SIGGRAPH 90 Conference Proceedings*, pages 215-224, 1990.
12. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. *SIGGRAPH 93 Conference Proceedings*, pages 19-26, 1993.
13. J. Hultquist. A virtual trackball. *Graphics Gems* (ed. A. Glassner). Academic Press, pages 462-463, 1990.
14. J.A. Landay and B.A. Myers. Interactive sketching for the early stages of user interface design. *CHI'95 Conference Proceedings*, pages 43-50, 1995.
15. R. MacCracken and K.I. Joy. Free-form deformations with lattices of arbitrary topology. *SIGGRAPH 96 Conference Proceedings*, pages 181-188, 1996.
16. L. Markosian, M.A. Kowalski, S.J. Trychin, L.D. Bourdev, D. Goldstein, and J.F. Hughes. Real-time nonphotorealistic rendering. *SIGGRAPH 97 Conference Proceedings*, pages 415-420, 1997.
17. H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, K. Omura. Object modeling by distribution function and a method of image generation. Transactions of the Institute of Electronics and Communication Engineers of Japan, J68-D(4):718-725, 1985
18. L. Markosian, J.M. Cohen, T. Crulli and J.F. Hughes. Skin: A Constructive Approach to Modeling Free-form Shapes. *SIGGRAPH 99*, to appear, 1999.
19. K. van Overveld and B. Wyvill. Polygon inflation for animated models: a method for the extrusion of arbitrary polygon meshes. *Journal of Visualization and Computer Animation*, 18: 3-16, 1997.
20. R. Pausch, T. Burnette, A.C. Capeheart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. Alice: Rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications*, 15(3): 8-11, May 1995.
21. L. Prasad. Morphological analysis of shapes. *CNLS Newsletter*, 139: 1-18, July 1997.
22. J.R. Shewchuk. Triangle: engineering a 2D quality mesh generator and Delauny triangulator. *First Workshop on Applied Computational Geometry Proceedings*, pages 124-133, 1996.
23. K. Singh and E. Fiume. Wires: a geometric deformation technique. *SIGGRAPH 98 Conference Proceedings*, pages 405-414, 1998.
24. G. Taubin. A signal processing approach to fair surface design. *SIGGRAPH 95 Conference Proceedings*, pages 351-358, 1995.
25. S.W. Wang and A.E. Kaufman, Volume sculpting. *1995 Symposium on Interactive 3D Graphics*, pages 109-116, 1995.
26. W. Welch and A. Witkin. Free-form shape design using triangulated surfaces. *SIGGRAPH 94 Conference Proceedings*, pages 247-256, 1994.
27. L. Williams. Shading in Two Dimensions. Graphics Interface '91, pages 143-151, 1991.
28. L. Williams. 3D Paint. *1990 Symposium on Interactive 3D Graphics*, pages 225-233, 1990.
29. R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. SKETCH: An interface for sketching 3D scenes. *SIGGRAPH 96 Conference Proceedings*, pages 163-170, 1996.

Smooth Meshes for Sketch-based Freeform Modeling

Takeo Igarashi
Computer Science Department, The University of Tokyo
takeo@is.s.u-tokyo.ac.jp

John F. Hughes
Computer Science Department, Brown University
jfh@cs.brown.edu

Abstract

This paper describes a framework for introducing visually smooth surfaces into sketch-based freeform modeling systems. An existing sketch-based freeform modeling system generates rough polygonal meshes with uneven triangulations after each operation. Our approach generates a dense, visually smooth polygonal mesh by beautifying and refining the original rough mesh. A *beautification* process generates near-equilateral triangles with a near-uniform distribution of vertices to mask the noise and bad sampling of the uneven mesh. The vertices are distributed on a smoothed surface that approximately interpolates the original mesh. *Refinement* generates a smooth, dense mesh by subdividing the beautified mesh and moving the vertices to the interpolative surface. The smooth interpolative surface is computed via implicit quadratic surfaces that best fit the mesh locally in a least-squares sense.

Keywords: Polygonal Meshes, Subdivision, Beautification, Skin, Implicit Surfaces, Sketch-based Modeling.

1 INTRODUCTION

Teddy [5] introduced a nice sketch-based modeling interface, but the resulting models were rough polygonal meshes. Their triangulations were uneven and the models had many undesirable small bumps and dents; such artifacts were introduced by almost all operations in the system. One could subdivide the mesh [9,20], but the resulting shape was not visually smooth because of the uneven triangulations. Our goal here is to introduce visually smooth surfaces like those seen in parametric and implicit models [17] to sketch-based modeling systems for free-form objects.

Our approach is to beautify and refine the irregular polygonal meshes resulting from the original Teddy algorithms (Figure 1). A *beautification* process, based on the Skin algorithm [11], generates near-equilateral triangles with a near-uniform distribution of vertices on the surface to hide irregularities in the original polygonal model; then *refinement* generates a dense polygonal mesh that smoothly interpolates the beautified mesh.

Beautification and refinement are guided by an implicit smooth surface that approximately interpolates the polygonal mesh. We compute implicit quadratic surfaces that best fit the mesh locally in a least-squares sense, and move the vertices to the surface during beautification and refinement. The implicit surfaces only *approximately* interpolate the mesh, and C^1 continuity among adjacent surface pieces is not guaranteed. This is not acceptable if one wants to use the implicit surface as final output, but works well for guiding the beautification and refinement of polygonal meshes. In addition, our framework is intended to apply to simple rotund objects without small details, such as those in Teddy.

One can smooth meshes with geometric fairing [1,6,14], but these methods are designed to remove high-frequency noise from dense polygonal meshes with fairly uniform vertex distributions, such as those arising from 3D scans; they do not work well for the

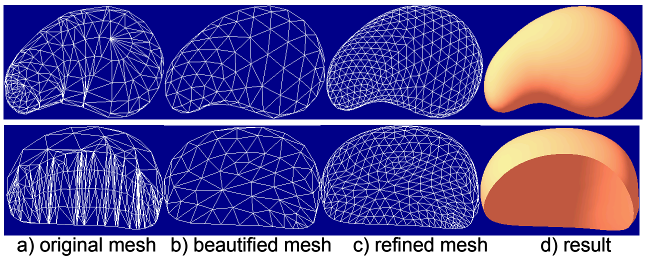


Figure 1: Overview of the algorithm. The system (a) constructs an uneven polygonal mesh from freeform strokes, (b) beautifies the mesh, (c) refines it, and (d) displays the refined mesh using smooth shading.

uneven, coarse meshes seen in Teddy. They also tend to make the surface drift away from the original mesh. We avoid this problem by fitting smooth surfaces to the mesh in a least-squares sense.

Our framework gives a basis for exploring various modeling operations with smooth surfaces. Given the built-in beautification mechanism, one can focus on the design of algorithms that construct arbitrary polygonal meshes without worrying about mesh quality or noise.

2 ALGORITHMS

Our basic representation for 3D geometry is a polygonal mesh. In response to editing operations, our system first generates an irregular polygonal mesh based on the algorithms introduced in Teddy [5]. Then we beautify the mesh internally and show the smoothly shaded refined mesh to the user. The algorithms have parameters that depend on the size of the models. The models are scaled to have their largest extent be 1.0.

2.1 Overview

The system maintains three polygonal mesh representations for each 3D model (Figure 2). The first is the *skin mesh*, which is the primary mesh for representing the target 3D shape. It adjusts itself over time through *beautification*. The second is the *skeleton mesh*, which is the irregular polygonal mesh created directly from the input strokes and serves as the reference for guiding the skin mesh during beautification. The third is the *visible mesh*, which is a dense, smooth polygonal mesh displayed on the screen as feedback to the user. The visible mesh is created from the skin mesh by *refinement* and is rendered using smooth shading. It is important to separate the visible mesh and skin mesh for efficient computation of the geometry. We describe beautification and refinement in detail in the following sections.

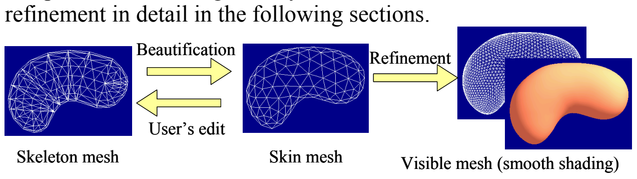


Figure 2: Three mesh representations.

When the user performs an editing operation, a copy of the skin mesh is modified to reflect the new geometry (Figure 3). This new geometry (whose triangulation is uneven and contains bumps and dents) is used as a *new* skeleton mesh; a new skin (which starts from this new skeleton mesh and gradually beautifies itself) is created from it. The user always sees the smooth visible mesh obtained through refinement.

When a modeling task is finished, the system stores the skin mesh as output. The user can use the mesh as a lightweight polygonal model or as a control mesh for subdivision[1], and can also store the visible mesh if a dense polygonal mesh is desired.

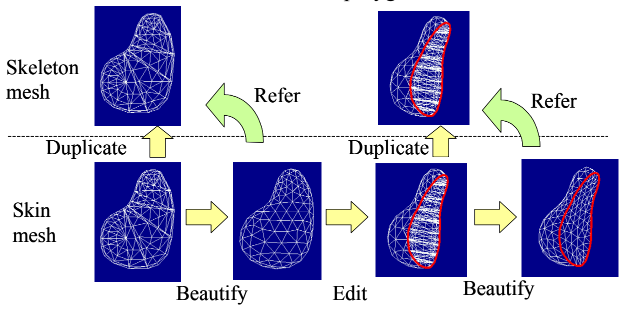


Figure 3: An editing sequence.

Edges along curves representing sharp ridges and creases are labeled as *sharp*. For example, the edges along the intersection loop resulting from a cut are labeled as sharp. We avoid blending surface normals of surrounding polygons at sharp edges so that smooth shading does not mask the sharp features. The Skin algorithm maintains the constraint that the sharp edges remain aligned along the curve [11].

2.2 Mesh Beautification

Mesh beautification aims to generate a mesh with near-equilateral triangles and a near-uniform vertex distribution while preserving some original overall shape, including sharp edges. Our algorithm is based on the Skin algorithm [11]. The vertices of our skin mesh move as particles around the skeleton, repeatedly adjusting their position and connectivity. Each skin vertex is associated with the nearest point on the skeleton mesh (called the *tracking point*). The main difference between our representation and that of Skin is that while Skin generates a *distance surface* around the skeleton with a certain offset, our beautification process tries to generate a surface that approximately *interpolates* the original skeleton mesh. One can obtain similar results simply by setting the offset to zero, but in the original Skin algorithm this actually shrinks the mesh (Figure 4 top). The amount of shrinkage is small if the skeleton mesh is dense, but is still problematic because the shrinkage accumulates through repeated edit-beautification cycles. This also occurs in other topological fairing techniques [7,16] because they insert new vertices on the existing polygonal surface.

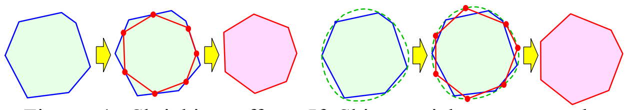


Figure 4: Shrinking effect. If Skin particles stay on the skeleton mesh, the resulting mesh gets smaller than the original (left). To prevent shrinking, the particles must move along an interpolative smooth surface (right).

To address this issue, we move the Skin particles along a smooth surface that approximately interpolates the skeleton mesh (Figure 4 bottom); we describe this surface in the next section.

2.2.1. Implicit quadratic surfaces

The many algorithms for creating interpolative parametric surfaces generally exhibit some artifacts due to the lack of global continuity [10]. Global optimization techniques can generate beautiful surfaces, but they are generally very slow [12]. Variational surfaces, represented by radial basis functions, are also globally (generically) smooth surfaces [17], but it is difficult to maintain a particular topology with them, and they sometimes exhibit unintuitive oscillations. Our approach is to compute implicit quadratic surfaces that best fit the mesh locally in a least-squares sense. This quadratic representation effectively eliminates small bumps and dents because of its limited degrees of freedom, and the least-squares fitting to neighboring vertices generates an aesthetically pleasing smooth surface from a coarse polyhedron.

Levin's approach [8] also uses least-squares fitting, but it locally computes a parametric surface while we locally fit implicit surfaces in 3D space (which makes it possible to fit shapes like ellipsoids perfectly). His approach also requires repeatedly solving a minimization problem when computing multiple positions on a surface. This would be prohibitively expensive when moving the skin vertices on the surface. On the other hand, the approach avoids the shrinkage problem mentioned above.

The implicit quadratic surface is computed for each skeleton vertex using nearby vertices as fitting targets. The quadratic function is formulated as

$$f(p) = f(x, y, z) = Ax^2+By^2+Cz^2+Dxy+Eyz+Fzx+Gx+Hy+Iz+J,$$

and the surface is implicitly defined as $f(p)=0$. We use the nearest 13 vertices around the vertex (including the vertex itself) as targets for fitting.[2] These are collected by a local search around the target vertex, which stops at edges labeled as sharp (Figure 5 left). To establish an orientation and to increase robustness, we also include extra low-weight constraints in the computation. These are obtained by moving each vertex in the direction of its temporary normal (the average of the surrounding polygon normals) with predefined offsets (±0.05 units). The system tries to fit the surface so that $f(p)$ becomes 0 at the target vertices, 1 at outside constraints, and –1 at inside constraints (Figure 5 right). Constraints are given smaller weights (0.01).

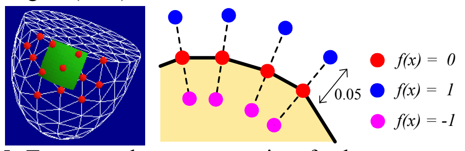


Figure 5: Targets and extra constraints for least-squares fitting. Red points indicate target vertices and the green surface represents the resulting implicit quadratic surface (left). We use 13 target vertices and additional in and out constraints (right).

The objective function for the least-square fitting is formulated as

$$E(f) = \sum_{p\in \text{targets}} (f(p))^2 + 0.01\{\sum_{p\in \text{outsideconstraints}} (f(p)-1)^2 + \sum_{p\in \text{insideconstraints}} (f(p)+1)^2\}$$

In matrix form,

$$\left(\begin{array}{ccc} \overbrace{\begin{matrix} x_0^2 & x_1^2 \\ y_0^2 & y_1^2 \\ z_0^2 & z_1^2 \\ x_0y_0 & x_1y_1 \\ y_0z_0 & y_1z_1 \\ z_0x_0 & z_1x_1 \\ x_0 & x_1 \\ y_0 & y_1 \\ z_0 & z_1 \\ 1 & 1 \end{matrix} \quad \cdots}^{\substack{\text{Target vertices} \\ v_i}} & \overbrace{\qquad\qquad}^{\substack{\text{Inside constraints} \\ v_i + 0.05n_i}} & \overbrace{\qquad\qquad}^{\substack{\text{Outside constraints} \\ v_i - 0.05n_i}} \end{array}\right) = \underset{(10\times 3n)}{L} \qquad \left(\begin{array}{ccc} \overbrace{\begin{matrix} 1 & & \\ & \ddots & \\ & & 1 \end{matrix}}^{n} & \overbrace{\qquad}^{n} & \overbrace{\quad 0 \quad}^{n} \\ & \begin{matrix} 0.01 & & \\ & \ddots & \\ & & 0.01 \end{matrix} & \\ 0 & & \begin{matrix} 0.01 & & \\ & \ddots & \\ & & 0.01 \end{matrix} \end{array}\right) = \underset{(3n\times 3n)}{W}$$

$$\left[\underbrace{0 \cdots 0}_{n} \quad \underbrace{1 \cdots 1}_{n} \quad \underbrace{-1 \cdots -1}_{n}\right]^T = \underset{(3n\times 1)}{B} \qquad X = (LWL^T)^{-1}LWB$$

The actual least-square fitting is done by solving the matrix system above, where X denotes the unknown vector of coefficients ($X^T=\{A,B,C,D,E,F,G,H,I,J\}$). The weighted overconstrained fitting

[1] The result of subdivision is slightly smaller than the visible mesh. For more accurate results, one can optimize the control mesh so that the result of subdivision faithfully matches the visible mesh [3].

[2] In a near-equilateral mesh, they are the vertices of the six triangles around the center and those of the six triangles around them.

problem is $WL^t X = WB$; multiplying by L on both sides leads to a solvable system. Such a system is solved once per vertex of the mesh.

Once we have the quadratic function for each skeleton vertex, we compute the target position for each skin vertex based on its tracking point on the skeleton mesh. If the tracking point is at a vertex, we simply use the quadratic function associated with the vertex. If the tracking point is at an edge, we compute the target position using each of the two quadratic functions associated with the edge's end points and linearly interpolate them according to the position on the edge. Similarly, the system uses quadratic functions associated with the three corners when the tracking point is on a triangle. To compute the position on the implicit quadratic surface, we apply a simple Newton's method three times, using the tracking point as initial value. This works reasonably well because the initial value is already close to the solution. For vertices lying on a sharp edge, we compute two implicit quadratic surfaces, and then move the vertex to one surface and then to the other in sequence using Newton's method.

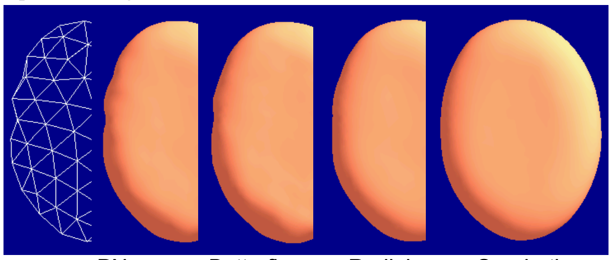


Figure 6: Comparison of various interpolations. The original mesh is subdivided twice and the vertices are moved to the surface defined by each interpolation scheme. Figures are rendered using smooth shading.

Figure 6 demonstrates the advantages of our approach. Local parametric interpolation (PN triangles [18]) and interpolative subdivision (butterfly subdivision [20]) exhibit small dents near the ridge that topological fairing techniques [7,16] cannot hide. Interpolation using radial basis functions [17] and our quadratic fitting both efficiently recover the smooth surface. An alternative solution to the problems arising in this example is to control the meshing so that edges are aligned to ridges; then silhouette problems are not so evident [19]. But this is difficult to do in general, and is in conflict with the behavior of skin.

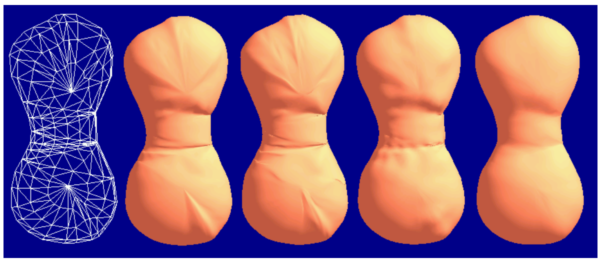


Figure 7: Comparison of various interpolations for the original mesh. This figure is generated in the same way as Figure 6. Existing schemes interpolate the original mesh *exactly*, which inevitably amplifies the small noise in the original mesh.

The piecewise quadratic surface approach may be unsuitable for some applications because of its approximating nature, but it works well for our purpose for several reasons. First, it generates a smooth surface from an uneven mesh with small bumps and dents; other methods are deliberately sensitive to these irregularities (Figure 7). Second, it is reasonably fast for interactive operation. Third, the implicit representation lets us move particles to the desired surface quickly, which can be difficult when using an interpolative subdivision scheme [6,9,20].

2.2.2. Computation of target edge length

The skin algorithm requires a target edge length for guiding the remeshing process; edges should be shorter at high-curvature regions and longer at low-curvature regions. A typical approach to computing surface curvature is to use the immediate neighbors of each vertex [13,15,16], but this can be unstable when applied to uneven meshes. We therefore use the implicit quadratic surface described in the previous section to compute the local curvature. The curvature for a skeleton vertex p is computed as follows. We compute the Hessian matrix $Hf(p)$ – the array of all second partial derivatives of f – and then the eigenvalues of

$$A = \begin{pmatrix} b_1^t Hf(p) b_1 & b_1^t Hf(p) b_2 \\ b_2^t Hf(p) b_1 & b_2^t Hf(p) b_2 \end{pmatrix}$$

where $\{b_1, b_2\}$ is an arbitrary orthonormal basis for the tangent plane at p. The principal curvature k_m is then $e_1 \,/\, ||\nabla f(p)||$ where e_1 denotes the larger eigenvalue of A [2, 4]. For vertices along a sharp curve, we use the curvature of the curve. Given k_m, we set the target edge length to $0.8/\,k_m$. To prevent excessively long or short edges, we clamp to a minimum and maximum edge length[3].

This procedure determines the desired target edge length for each vertex, but these values may not be appropriate from a more global point of view. Figure 8 illustrates the problem. The low-curvature point v suggests a long edge length, but the long edges at v fail to represent the high curvature region *near* v. To prevent this, we impose the following constraint to the target edge length, using $L(p)$ to denote the target edge length at vertex p: "For every vertex u whose distance to a vertex v is smaller than $L(v)$, $L(u)$ must be equal to or larger than $L(v)$." To satisfy the constraint, the system searches the neigbors U of each vertex v and sets $L(v)$ to max $(L(u), |v\text{-}u|)$ if $L(u) < L(v)$ and $u \in U$. We use mesh distance as the measure of distance between vertices.

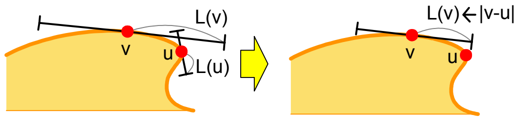


Figure 8: Postprocessing for target edge length.

2.3 Mesh Refinement

Mesh refinement generates a dense, smooth polygonal mesh from the skin mesh as feedback for the user. To do this, we subdivide the skin mesh, and then move the vertices to the quadratic surfaces fitted to the skin mesh. One can obtain smoother surfaces by applying the refinement process repeatedly, but we found that a single refinement generates visually satisfying results as feedback during editing operations.

3 IMPLEMENTATION AND RESULTS

We are developing a prototype modeling system based on our surface representation. The system uses a sketching interface like Teddy's, with some experimental smooth-surface editing operations such as filleting, creasing, and smoothly merging separate meshes (Figure 9). Filleting smooths the sharp corners

[3] The minimum edge length is set to 0.03 and the maximum to 0.3 in the current implementation. In the future we hope to find a way to compute these lengths from properties of the overall shape.

resulting from cutting or extrusion. We apply a geometric fairing algorithm [13] to the skeleton mesh for smoothing. Creasing puts a sharp crease where the user draws a stroke on the object surface. This is done by pushing the stroke edges inwards and labeling them as sharp [11]. For smooth merging we compute the union of the two meshes and put a fillet at the intersection. As in the original Teddy system, these operations simply edit the polygonal mesh; this is significantly easier to implement than it would be with parametric surfaces or implicit surfaces. The accompanying video demonstrates the behavior of the system from the user's point of view.

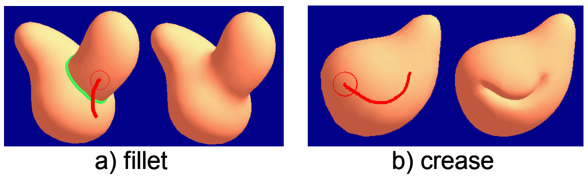


Figure 9: Experimental editing operations.

The system is implemented in Java™ (JDK1.4) and uses directX7 for 3D rendering. It takes a few seconds for skin algorithms to converge to a reasonably beautiful mesh after each editing operation on a high-end PC (AMD Athlon™ 1.54GHz). Figure 10 shows some example 3D models designed using the system (they show the visible mesh in our system). The duck's neck is smoothly merged with the head, and the four legs are smoothly merged to the octopus body. The palm and the bottom of the foot were made by putting fillets at intersections after cutting.

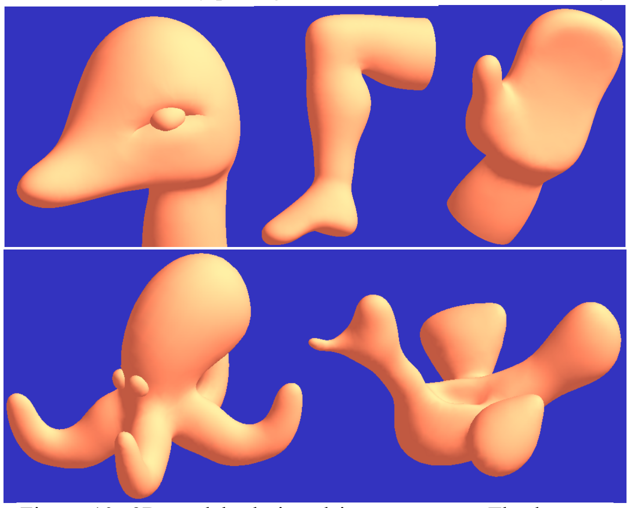


Figure 10: 3D models designed in our system. The last one was designed by a test user and the others by the author.

4 LIMITATIONS AND FUTURE WORK

There are some fundamental limitations in our technique. First, it works only for smooth, rounded surfaces. Second, it requires several empirically set constants. Third, there is as yet no theoretical guarantee of smoothness and robustness.

Our least-squares fitting finds good quadratic functions in most cases, but the resulting surface sometimes has a "discontinuity" in the middle of the target fitting area (Figure 11). This is a fundamental problem of implicit quadratics and our only solution so far is to have more vertices as fitting targets and to use "in" and "out" hints. This prevents the problem in almost all cases in our experience, but we clearly need a more complete solution.

The current implementation can represent sharp edges but not the tip of a cone, i.e., we handle one-dimensional singularities but not zero-dimensional ones. The system automatically rounds off sharp tips in our current implementation. Although this might be acceptable in most cases, we plan to search for an appropriate representation of such points.

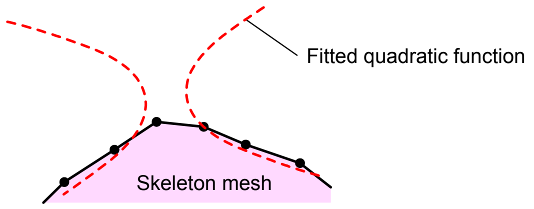


Figure 11: A limitation of quadratic fitting.

References

1. M. Desbrun, M. Mayer, P Schröder, and A.H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. *SIGGRAPH 99 Conference Proceedings*, pages 317-324, 1999.
2. P. Dombrowski. Krümmungsgrößen Gleichungsdefinierter Untermannigfaltigkeiten Riemannscher Mannigfaltigkeiten. *Mathematische Nachrichten*, vol. 38, pages 133-190. Berlin: Akademie Verlag, 1968.
3. H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. *SIGGRAPH 1994 Conference Proceedings*, pages 295-302, 1994.
4. J. Hughes, Differential Geometry of Implicit Surfaces in 3-Space – a Primer. *Technical Report CS-03-05*, Computer Science Dept., Brown University, 2003.
5. T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: a sketching interface for 3D freeform design. *SIGGRAPH 99 Conference Proceedings*, pages 409-416, 1999.
6. L. Kobbelt. Discrete fairing and variational subdivision for freeform surface design. *The Visual Computer, Vol. 16, Issue 3/4,* pages 142-158, 2000.
7. L. Kobbelt, T. Bareuther, H.P Seidel. Multiresolution shape deformations for meshes with dynamic vertex connectivity, *Computer Graphics Forum, Vol 19, No 3*, pages 249-260, 2000.
8. D. Levin. Mesh-independent surface interpolation. To appear in *Advances in Comp. Math*.
9. J. Maillot and J. Stam. A unified subdivision scheme for polygonal modeling. *Eurographics '01 proceedings*, 2001.
10. S. Mann, C. Loop, M. Lounsbery, D. Meyers, J. Painter, T. DeRose, and K. Sloan. A survey of parametric scattered data fitting using triangular interpolants. In Hans Hagen, editor, *Curve and Surface Design*, pages 145-172. SIAM, 1992.
11. L. Markosian, J.M. Cohen, T. Crulli, and J.F. Hughes. Skin: a constructive approach to modeling free-form shapes. *SIGGRAPH 99 Conference Proceedings*, pages 393-400, 1999.
12. H.P. Moreton and C.H. Sequin. Functional optimization for fair surface design. *SIGGRAPH 92 Conference Proceedings*, pages 167-176, 1992.
13. R. Schneider and L. Kobbelt. Geometric fairing of irregular meshes for free-form surface design. To appear in *Computer Aided Geometric Design*.
14. G. Taubin. A signal processing approach to fair surface design. *SIGGRAPH 95 Conference Proceedings*, pages 351-358, 1995.
15. G. Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. *Fifth International Conference on Computer Vision,* pages 902-907, 1995.
16. G. Turk. Re-tiling polygonal surfaces. *Computer Graphics, Vol. 26, No. 2, (SIGGRAPH 92)*, pages 55-64, 1992.
17. G. Turk and J. F. O'Brien. Variational implicit surfaces. *Technical Report GITGVU 9915*, Georgia Institute of Technology, May 1999.
18. A. Vlachos, J. Peters, C. Boyd, and J.L. Mitchell. Curved PN triangles. *Proc. of Interactive 3D Graphics,* pages 159-166, 2001.
19. J. Vorsatz, C. Rossl, L. Kobbelt, and H. Seidel. Feature sensitive remeshing. *Eurographics '01 proceedings*, pages 393-401, 2001.
20. D. Zorin, W. Sweldens, and P. Schröder. Interpolating subdivision for meshes of arbitrary topology. *SIGGRAPH 96 Conference Proceedings*, pages 189-192, 1996.

Eurographics/SIGGRAPH Symposium on Computer Animation (2003)
D. Breen, M. Lin (Editors)

A Sketching Interface for Articulated Figure Animation

James Davis[1], Maneesh Agrawala[2], Erika Chuang[3], Zoran Popović[4] and David Salesin[5]

[1] Honda Research Institute USA – jedavis@ieee.org
[2] Microsoft Research – maneesh@microsoft.com
[3] Stanford University – echuang@graphics.stanford.edu
[4] University of Washington – zoran@cs.washington.edu
[5] University of Washington and Microsoft Research – salesin@cs.washington.edu

Abstract

We introduce a new interface for rapidly creating 3D articulated figure animation, from 2D sketches of the character in the desired key frame poses. Since the exact 3D animation corresponding to a set of 2D drawings is ambiguous we first reconstruct the possible 3D configurations and then apply a set of constraints and assumptions to present the user with the most likely 3D pose. The user can refine this candidate pose by choosing among alternate poses proposed by the system. This interface is supported by pose reconstruction and optimization methods specifically designed to work with imprecise hand drawn figures. Our system provides a simple, intuitive and fast interface for creating rough animations that leverages our users' existing ability to draw. The resulting key framed sequence can be exported to commercial animation packages for interpolation and additional refinement.

1. Introduction

Traditional animators often begin work by quickly sketching thumbnails of a character in key poses to capture the character's overall motion.[1] The characters are drawn as stick figures or as simple rectangular and ellipsoidal volumes. Once a coarse version of the motion is on paper, they rework and refine the key poses, and fill in the in-between poses to eventually produce the final animation. While this coarse-to-fine motion refinement strategy is also used in 3D computer animation,[14] the initial step of generating a coarse set of key poses is far more difficult on a computer.

While existing 3D animation systems provide powerful tools, appropriate for precise 3D positioning, they are not well suited for rapidly posing articulated figures. In contrast, artists can quickly and easily sketch 2D figures and professional computer animators often draw key poses on paper before building them in the computer.[14]

In this paper we present an interface for using these sketches to directly infer the 3D pose of an articulated figure. Since sketches of arbitrary style would be very difficult to automatically parse, our interface requires the user to annotate or overlay their initial sketches with stick figures. These stick figures require only a few seconds to draw, much less time than the initial sketch itself. From the simple stick figures, our system automatically extracts the 2D location of joints and bones and then reconstructs 3D poses. These poses are then interpolated to quickly create a coarse animated motion that provides a good starting point for producing the refined final motion. In addition to allowing experienced 3D animators to quickly create rough motions, our interface provides a bridge to the world of 3D animation for the millions of artists who are skilled with pencil and paper, but lack experience with 3D tools.

The primary challenge in creating a 3D animation from 2D images is that many 3D poses may be consistent with a given 2D stick figure. As shown in Figure 1, multiple poses match the drawing exactly. The imprecise nature of hand drawings compounds this difficulty since poses that approximately match the drawing should be considered as well. Since our goal is to aid animators as they initially design an animation, a completely automated pose reconstruction system is not appropriate. However, manually posing an articulated figure by specifying the location of each joint is tedious. Instead, we desire a semi-automated method that allows the artist to influence and control the resulting animation.

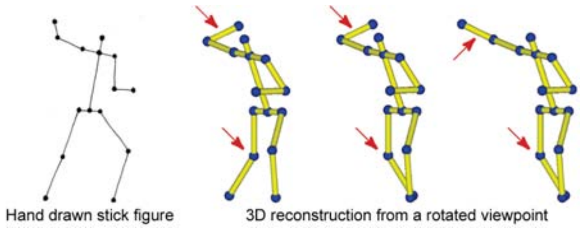


Figure 1 *Multiple 3D poses can be consistent with a single 2D stick figure. Each foreshortened bone can be pointed either towards the viewer or away from the viewer. Here we see three possible reconstructions of the hand drawn keyframe on the left. (The viewpoint has been rotated by 90 degrees about the vertical axis to expose the ambiguity.) The arrows indicate the joints or bones that have changed. In the leftmost reconstruction the knee bends inwards and looks unnatural. We eliminate such reconstructions using joint-angle constraints. In both the middle and rightmost reconstructions the raised forearm is within a natural range, and either pose is equally plausible.*

Our approach is to build an interface that constructs the set of poses that exactly match the drawing, automatically selects the best guess, and then allows the user to guide the system to the desired character pose. Precise reconstruction of pose is limited by the imprecision of hand drawings. Even skilled artists do not always draw bones with geometrically precise foreshortening. Our interface handles such imprecision through a process of automated refinement and optimization.

2. Related Work

The most common interfaces for posing 3D articulated figures allow users to interactively position the extreme joints of a character and use inverse kinematics (IK) to update the positions of interior joints.[6,26] Yet the power of such interfaces can also be a weakness. Novice users can find it particularly difficult to use such interfaces because every parameter is available for continuous manipulation. The freedom of motion can overwhelm the ability of users to obtain the desired pose. Rather than requiring continuous manipulation of 3D widgets as with IK systems, our interface asks users to choose the intended pose from a discrete set of possible choices.

The functionality of our interface complements IK systems. It acts as an alternate that is appropriate for novice users, and which may provide a way for skilled animators to quickly rough out motions before refining them with the full power of existing IK tools.

Hecker and Perlin[8] developed a sketch based animation system using a touch sensitive tablet that is similar in spirit to ours. However their system relies completely on the artist to resolve ambiguity, and no provisions for regularizing the resulting animation are explored.

Bregler et al.[2] propose a method for capturing the expressive motion of cartoons and retargeting it onto articulated figures. They require that 3D keyframes corresponding to the cartoon motion be manually constructed using a traditional animation package. Our method complements their work in that we focus on reconstructing keyframes, while they provide a method for interpolating between them.

In the domain of static 3D modeling, SKETCH,[25] Teddy,[12] and Chateau[11] all provide the casual sketch style interface we seek. In examining these and other systems we have extracted two high-level principles that can be applied to many such interfaces: The system should use a set of *default assumptions* to automatically resolve ambiguities. These assumptions should essentially guess what the user desires, without having the user specify every detail precisely. In addition, the system should provide an interface allowing *user guidance* when the default assumptions are wrong. The additional information about the user's intent should be used to refine the assumptions and produce a new guess from among the possible solutions.

At the core of an automated solution is some method of reconstructing pose from the drawn 2D structure. The computer vision community has explored the related problem of reconstructing 3D poses from a monocular video sequence. Several recent surveys provide an introduction to the range of methods that have been explored,[5,18] and an explanation of why this problem is particularly challenging for the task of animation is given by Gleicher and Ferrier.[7] Rather than attempting a comprehensive treatment here, we discuss broad categories of approaches with a few representative samples.

Model-based tracking and reconstruction methods[3,4] assume that a 3D skeleton is known a priori and that the initial 3D pose of this skeleton has been hand-specified so that the 3D joints match corresponding 2D image features in the first frame. These methods then use fully automatic optimization techniques to both track the 2D image features and find a set of 3D skeletal joint angles that match the 2D image features in the subsequent frames. However, these methods often rely on video frame rates and require that the user re-initialize the system if large frame-to-frame motions cause the tracking to fail. When the frame rate is high, these systems provide a useful automation. However, when the frame rate is low, reinitialization is common, and the problem becomes one of finding a method for quickly initializing pose. Since animators often choose to draw widely spaced keyframes, our problem is closer to that of initializing pose than to that of tracking closely spaced frames.

Another approach to the pose reconstruction problem is to use probabilistic techniques[10,20] to automatically learn the mapping between 2D image features and 3D poses. The main drawback of these techniques is that they require large sets of training data in which the correspondence between the 2D image and 3D pose is already known. In our case an artist would have to draw each of the training images and hand-specify the corresponding skeletons before applying the method to a new set of stick figures.

Completely automated solutions, such as those in the previous two categories, are attractive to computer scientists. Indeed, they are appropriate and useful in many contexts. However, they have an additional limitation: They would defeat the artistic intent of our tool. Automated solutions cannot ensure that the correct pose is chosen from among the many ambiguous solutions, since the correct pose is a matter of artistic intent. A tool designed for artists, such as the one described in this paper, must explicitly expose this ambiguity to the artist rather than hide it, allowing the user to guide the system interactively to the correct solution. Too much control, as in the case of IK interfaces, can also be difficult to use. We believe our solution provides a good balance between these two extremes.

A final approach for pose reconstruction explicitly acknowledges the existence of multiple solutions and creates a large set of all possible poses. This set is then pruned to find the desired pose. Lee and Chen[16] prune the set using joint angle constraints and a strong prior model of walking humans. In contrast, Taylor[22] relies on the user to select the correct pose. Neither the assumption of walking nor completely manual specification is desirable for our interface. However, because this class of methods allows for both automation and user guidance it provides one of the critical components of our interface.

Contributions. The primary contribution of this work is a method that allows an animator to create rough 3D articulated figure animation almost entirely from 2D sketches, with little additional effort. Our approach relies upon a user interface that follows the principles of default assumptions and user guidance derived from other sketch-based systems. In addition, we present a novel reconstruction method that both allows user guidance and can robustly reconstruct 3D pose from imprecise hand-drawn figures.

3. User Interface

An artist creates animations using our system in two stages. The artist first annotates a sequence of drawn keyframes that represent the desired motion. Since the exact 3D pose matching each annotated drawing is ambiguous, the artist next guides a semi-automated process to the correct reconstruction. The details of these interface procedures are given in this section. The implementation of supporting algorithms will be described in section 4.

Draw and annotate keyframes. Many artists prefer to create images using pen, paper, and light box, while others prefer to create images directly on a digital canvas in a computer. We support both styles of work. The artist simply sketches a sequence of keyframes in any style, and then annotates these sketches with the skeletal bone structure of the drawing.

On paper, the stick figures are drawn with thick circular dots at the joints, and thin lines connecting them as shown in Figure 1. These drawings are scanned and then automatically parsed by the system to locate joint positions and connectivity. When working from a digital canvas, we provide a stroke-based interface that allows artists to quickly draw the skeletal stick figure directly. New strokes automatically snap to previous strokes making it easy for the user to ensure that segments properly connect to one another.

After joint positions and connectivity are specified, the system automatically labels the stick figure, putting it in correspondence with a pre-defined template skeleton.

A template skeleton is required for 3D reconstruction and specifies both connectivity and bone lengths. The artist specifies bone lengths for a given character by drawing a sketch parallel to the image plane, with no foreshortening. For example, humanoid skeletons are typically drawn standing straight up with arms fully extended out to the sides. As an alternative we have found that bone lengths can often be adequately estimated by using the longest apparent length across all the keyframes. The assumption in this case is that the bone is fully extended when it is longest and therefore parallel to the image plane.

Although connectivity of the template could theoretically be extracted from the same sketch that provides bone lengths, we have not yet implemented this feature. Instead we ask the user to specify this information in a text configuration file.

Indicate desired 3D pose. The 3D pose of a character is not uniquely defined by the annotated keyframe. Given a labeled stick figure and the corresponding template skeleton we reconstruct all possible 3D poses that match the

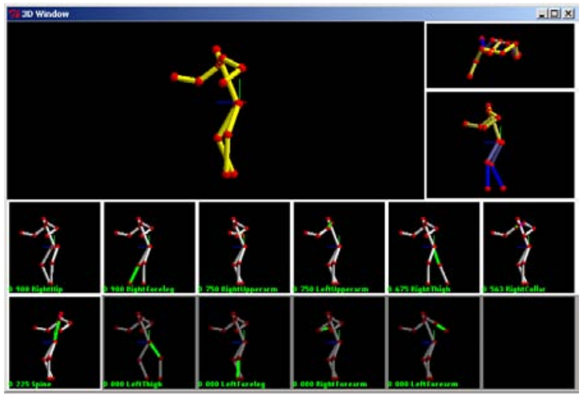


Figure 2 *Our system provides a suggestive interface that allows the user to quickly guide reconstruction of the character's 3D pose. The currently estimated best pose is shown above and thumbnails of alternate poses are shown below. Clicking on a thumbnail flips the towards/away direction (with respect to the viewer) of a single bone in the 3D reconstruction.*

drawing. The set of poses is then culled using joint angle constraints. The remaining 3D poses are ranked according to a set of heuristics, and the highest ranking pose is set as the default.

Since the default pose may not match the pose intended by the animator, our system also suggests a number of alternative poses and allows the animator to pick among them. Given a figure with n bones, there are in general 2^n possible 3D configurations for the figure, as each foreshortened bone can point either towards or away from the viewer with respect to the image plane. To keep the choices manageable, our system suggests just n alternative poses to the user, as shown in Figure 2. Each alternative pose is chosen so that the direction of a single bone is changed with respect to the default pose. The alternatives are displayed as thumbnails below the default, and the bone that has changed in each thumbnail is drawn in bright green with its name underneath. If the change would create a pose violating joint constraints, the thumbnail is drawn in dark gray. This approach is based on Igarashi and Hughes's suggestive interface 3D modeling system.[11]

To change the direction of a bone the user simply clicks on the appropriate thumbnail. The pose in that thumbnail then becomes the new default pose, and the

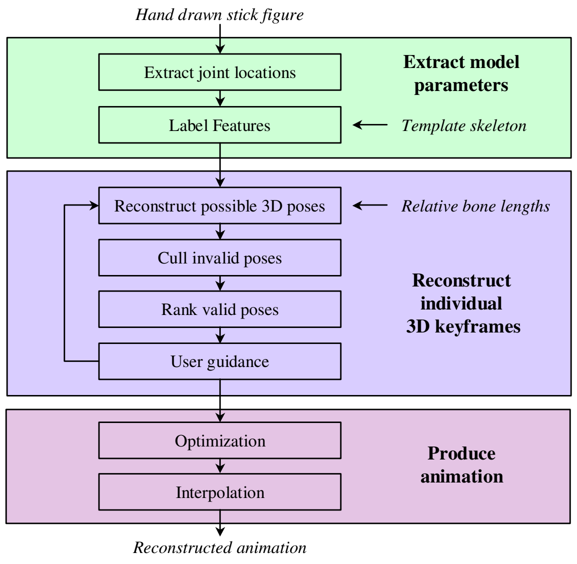


Figure 3 *Overview of system pipeline. Hand-drawn stick figures are processed by a sequence of stages to produce the final reconstructed animation. First, 2D model parameters, joint locations, and connectivity are extracted from drawings. This information is matched against a known template skeleton. Then, all possible 3D character poses are reconstructed from the labeled features and skeletal bone lengths. A semi-automated user-guided iterative process specifies the desired pose. The resulting key poses are optimized and exported for further interpolation and refinement.*

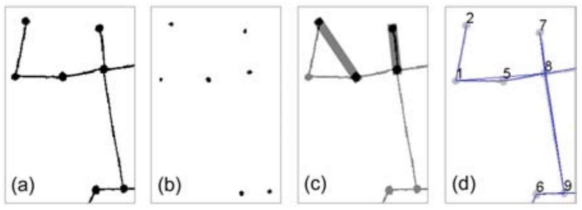


Figure 4 **(a)** *A drawn stick figure before automatic location of joints and bones.* **(b)** *Image erosion is iteratively applied to find the location of joints.* **(c)** *Bones are located by examining a linear region connecting all possible pairs of joints.* **(d)** *Regions found to have a single connected component are identified as bones. The final joint and bone structure is recovered after removing cycles from the graph of connected bones.*

thumbnails are redrawn to reflect all the single-bone changes with respect to that new pose.

A single change may be insufficient to select the desired pose. If additional changes are required the user merely continues to click on thumbnails until the correct pose is obtained. Although up to n choices could theoretically be required, we have found that the initial selection is often correct and that fewer than two bone direction choices are required on average.

After selecting the intended 3D pose for each keyframe, the animation can be easily exported to a commercial animation tool for interpolation and further refinement.

4. Implementation

The interface presented to the user employs a number of behind-the-scenes automations and assumptions. The box diagram in Figure 3 presents an overview of the computational tasks required to implement our interface.

Extract joint locations. The image-plane locations of joints and bones that define a keyframe must be determined before 3D reconstruction can take place. Given a scanned stick figure representation of the character, joints can be located through a sequence of image processing operations.[24] Figure 4(a) shows a stick figure drawing. By iteratively applying an image erosion operator to the keyframe bones and joints are gradually eliminated. Since joints are drawn more thickly, they will remain for a greater number of iterations. Figure 4(b) shows the result after several iterations of erosion. The process is halted when the number of connected regions in the image matches the number of joints in the template skeleton. The centroid of each remaining connected component is taken as the location of a joint.

In order to determine which joints are connected by bones, a linear region connecting each pair of joints in the original image is examined. Two examples of this region are shown as grey bars in Figure 4(c). If this region contains a single connected component then the joints are connected, if two or more components are present then the joints are separated by white space, and are therefore not connected. This process results in a graph of joints and their associated connectivity, as shown in Figure 4(d). When three or more joints are collinear, a cycle will form in the graph, e.g., joints 1, 5, and 8. Since the longest connection is a concatenation of the shorter connections in this collinear cycle, we remove the longest component of any cycle discovered in the graph.

Although failure cases exist, such as when joints lie atop one another, we have found this procedure to work in every instance in which it intuitively seems that it should, providing a reliable efficient automation for the process of specifying joint locations.

Label features. The joints in the extracted graph structure must be correctly associated with the template skeleton for reconstruction to take place. Since we have already determined the graph structure of our drawn stick figure, the joints can be labeled by computing an isomorphic mapping between the drawn skeleton and the template skeleton. Given two graphs G_1 and G_2 an isomorphism is a one-to-one mapping of the vertices that maintains adjacency and non-adjacency of the vertices. We use the graph matching algorithm of Schmidt and Druffel[21] to compute all valid isomorphisms between the two skeletons.

Unfortunately, if the connectivity structure of the graphs contains symmetries there will be more than one isomorphic mapping between the drawn skeleton and the template skeleton. To resolve such ambiguities the system chooses the labeling that would result in joint locations that most closely match the previous frame. If no previous frame is available we label the joints assuming the skeleton is facing forward. If the assumptions are incorrect the user can quickly cycle through the valid labelings for the skeleton by right-clicking near an incorrectly labeled joint. We have found that this combination of automation and user guidance allows a correctly labeled skeleton to be specified quickly.

Reconstruct possible 3D poses. A set of all possible 3D poses can be constructed given the 2D image location of each skeletal joint and template bone lengths. We follow the reconstruction approach described by both Taylor[22] and Lee and Chen.[16]

Assume a scaled orthographic camera model, which relates image coordinates $\mathbf{q}=(u,v)$ to world coordinates $\mathbf{p}=(X,Y,Z)$ through the following equation:

$$\mathbf{q} = s \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \mathbf{p} \qquad (1)$$

Since the X and Y world coordinates can be determined directly from the image plane observations, all that remains is to determine the Z coordinate of each joint.

Suppose that a bone segment is defined by two image points $\mathbf{q}_1$ and $\mathbf{q}_2$. We can compute the relative distance in Z (*dZ)* between $\mathbf{p}_1$ and $\mathbf{p}_2$ using the following equation:

$$dZ = \pm\sqrt{l^2 - (\mathbf{q}_1 - \mathbf{q}_2)^2 / s^2} \qquad (2)$$

where the length of the bone is given by *l*, and *s* is the scale parameter relating image and world coordinates. If all keyframes, and the figure specifying bone length, were drawn at the same scale, then the value of *s* will be 1.0. If keyframes have been drawn at different scales then the correct value of *s* changes to reflect the nature of the drawings. We allow *s* to either be set by the user or determined automatically using the heuristic given by Taylor.[22]

Equation (2) provides two possible answers for *dZ*, representing the pose ambiguity that has been previously discussed. We retain both answers, allowing all possible poses to be computed.

The above process is repeated, following the skeletal graph structure, until the possible coordinate values of all joints have been enumerated.

Intuitively, if a bone is drawn short in a particular keyframe, the bone is foreshortened; thus, the value of *dZ* will be relatively large. If a bone is drawn long, then the bone is relatively parallel to the image plane, and the value of *dZ* will be small. Hand-drawn animations present an interesting challenge to this intuition. Since *dZ* should never be complex, equation (2) provides an upper bound for the drawn length of a bone:

$$\|\mathbf{q}_1 - \mathbf{q}_2\| \le s \cdot l \qquad (3)$$

That is, a bone segment that is drawn too long has no physical meaning. However, cartoon figures are imprecisely drawn at best, and often actively subjected to squash and stretch. The previous algorithms did not deal with such imprecision, often adjusting *s* to force a physically valid interpretation. We take an alternate approach more in line with the intent of the animator. If a particular bone is illustrated stretched beyond meaning, we simply allow the length of the bone, *l*, to change in the corresponding frame of the 3D reconstruction.

Cull invalid poses. The reconstruction method described in the previous section produces the set of all possible 3D poses that match the input drawing with the template skeleton. It is critical that this set be pruned to the smaller set of poses that might reasonably match the artist's intention. As shown in the leftmost reconstruction of Figure 1 where the knee bends inwards, some of these poses are impossible. We use default assumptions in the form of joint angle constraints to identify and cull such invalid poses.

A number of methods for applying joint angle constraints have been proposed.[13,23] We choose to follow the method of Lee and Chen[16] and derive our angle limits from the biomechanical measurements of Houy.[9]

The simple template used in this work has 11 bones whose orientation are unknown, which amounts to 2^{11}, or over 2000, possible poses. After joint angle culling, we find that approximately 5% remain, equivalent to about 6 bones whose orientations remain ambiguous.

Rank valid poses. After culling we rank the remaining poses using a set of *preferences.* These preferences are prior assumptions about the naturalness of a given pose. We currently use three types of preferences: preferred joint angles, balance, and frame-to-frame coherence. Preference values are normalized to lie between 0.0 and 1.0. The rank of a given pose is computed as the product of individual preference values, aggregated over all joints and preference types.

Even within the range of valid joint angles, some angles are more natural than others. Based on this idea, we weight joint angles that fall within the valid range so that more natural poses are given a higher preference value. Each joint angle constraint is augmented so that it also specifies a preferred angle. In practice, we simply set this angle to the midpoint of the valid range. We compute the preference as inversely related to the angular distance between the projected bone and the preferred angle.

For the human skeleton we also compute a balance preference. When humans are upright, the spine is usually oriented so that the head is in front of the pelvis. When the head is behind the pelvis the spine looks hyper-extended and the body seems unbalanced. Therefore, we compute the angle between the spine and the world-space y-axis and if the head is behind the pelvis we reduce its preference value based on the angular distance from vertical.

Since the drawn stick figures represent key poses of a figure moving over time, it is expected that some coherence exists between neighboring frames. Assuming that the user has chosen the desired pose for the figure in frame *t*, the angular difference between bone directions in frame *t* and bone directions for each candidate pose in frame *t*+1 is computed. Candidate poses that are the most similar to the previous frame's reconstruction will receive the highest preference from this metric.

Given the ranked poses, the best one is presented to the user, who then guides the system towards the correct pose using the interface presented in the previous section. We have found our relatively simple preferences sufficient to rank the poses, resulting in an average of fewer than two user-specified bone reorientations to obtain the desired pose. Although it may be possible to further improve the quality of our pose ranking, we believe that automated ranking will never completely remove the fundamental necessity of user guidance, since the correct pose is a matter of artistic intent.

Optimization. Hand-drawn figures often exhibit distortions that create difficulties for reconstruction methods that rely on fixed bone lengths. Such imprecision appears as undesirable sliding and wobble in the reconstructed animations. Using only the reconstruction method presented earlier, the resulting animations are of relatively poor quality. Thus, after the user has specified the desired pose for each keyframe, an optimization process is invoked to remove these undesirable effects. By allowing for small variations in the user-specified joint positions and bone lengths, a smoother, more natural looking animation can be created.

Our notation is as follows. The final 3D location of a joint is $\mathbf{p_{jf}}=(X_{jf}, Y_{jf}, Z_{jf})$, where j indexes joints, and f indexes frame number. The drawn 2D location of a joint is $\mathbf{q_{jf}}=(u_{jf}, v_{jf})$. The length of a bone is given by l_b. The optimization vector is given as $\mathbf{p}=[\mathbf{p_{11}}\ \mathbf{p_{12}} \dots \mathbf{p_{jf}}]$. Our optimization objective is posed as a weighted sum of the terms described below.

Since we would like to maintain fidelity to the original drawings, our first objective term penalizes joints that move away from their drawn location on the XY image plane:

$$\left\| \mathbf{q_{jf}} - s \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \mathbf{p_{jf}} \right\|^2 \qquad (4)$$

It is important to note that joints are not constrained to lie exactly at the location in which they were drawn, as this would unnecessarily restrict the final animation.

The goal of optimization is to smooth out undesirable motions caused by imprecision. In order to achieve this goal, a second regularization objective is included to enforce temporal coherence between neighboring keyframes:

$$\left| Z_{jf} - Z_{j(f+1)} \right|^2 \qquad (5)$$

Undesirable motions are manifested primarily on the Z-axis, perpendicular to the 2D drawing. For this reason, we chose to penalize motion along this axis, so that each joint is encouraged to have more similar values across time.

The 3D reconstruction process does not necessarily maintain adjacencies that were intended by the artist. For example, joints that were drawn in nearly the same 2D position in neighboring keyframes were probably intended to remain static along the Z-axis as well. This commonly occurs with feet, which should remain stationary on the floor. Similarly, distinct joints, j and k, that are drawn as exactly coincident in an individual frame were probably also intended to be coincident in 3D. We add constraint terms to enforce both of these conditions:

$$\begin{array}{lcl} \left(Z_{jf} - Z_{kf} \right)^2 & \text{when} & \left\| \mathbf{q_{jf}} - \mathbf{q_{kf}} \right\| < \varepsilon \\ \left(Z_{jf} - Z_{j(f+1)} \right)^2 & & \left\| \mathbf{q_{jf}} - \mathbf{q_{j(f+1)}} \right\| < \varepsilon \end{array} \qquad (6)$$

The user-selected key poses can be thought of as a local minimum for the optimization function. Each of the many ambiguous poses that were rejected by the artist represents another minimum within the functional space. We would like to ensure that our optimization procedure maintains the user's intention while improving the smoothness of the animation. We therefore penalize joint positions

	Weight
Eqn 4	200
Eqn 5	1
Eqn 6	25
Eqn 7	200
Eqn 8	slider

Table 1 *Weights for each optimization objective.*

that would reverse the desired orientation of bones—i.e., the sign of dZ from equation (2) should not be changed. Letting $\mathbf{p'_{jf}} = (X'_{jf}, Y'_{jf}, Z'_{jf})$ indicate the initial joint position, we have:

$$\max\left(0, (Z_{jf} - Z_{kf})\frac{(Z'_{kf} - Z'_{jf})}{\left|Z'_{kf} - Z'_{jf}\right|}\right)^2 \qquad (7)$$

An artistic drawing could either be very realistic and precise, or contain unintentional distortions, such as squash and stretch. In the former case the optimization should preserve the length of bones, interpreting any changes in apparent bone length as foreshortening effects. In the latter case, the artistic intent is that bone length should be only loosely preserved. We provide a slider with which the artist can indicate his or her intent. This in turn specifies the weight by which changes in bone length are penalized by the following term:

$$\left(l_b - \left\|\mathbf{p_{jf}} - \mathbf{p_{kf}}\right\|\right)^2 \qquad (8)$$

The solution to the above objectives is given by equation 9, where E_i is an individual objective, and w_i is the weight of that objective. We use a publicly available package to perform this optimization, using finite differences to supply gradients.[15]

$$\arg\min_{\mathbf{p}} \sum_i w_i \, E_i \qquad (9)$$

While several terms contribute to the optimization objective, we have found that it is not necessary to provide user control over all weights. Our interface contains a single slider, to control "squashiness," which indicates the artistic precision with which bone lengths were illustrated. We find that this slider provides the necessary level of artistic control, while not overwhelming the user with the algorithm's full complexity. The values of other weights are given in Table 1.

Following optimization, the animation can be easily exported to a commercial animation package for interpolation and further refinement.

5. Results

We have created a number of animations using our system. Figure 6 shows drawn keyframes as well as the interpolated 3D motion for a few of these. The included video also shows all of the examples. The keyframes were drawn by several artists who ranged in experience from novice to professional. The relative timing between keyframes was adjusted as a post-process in Maya, since keyframes were not drawn with uniform time steps. Note that keyframes are shown side by side in this figure for clarity. In practice, the artist draws these frames atop one another using a lightbox, so that the image plane spatial relationship of the figures is preserved.

Table 2 gives statistics on each animation. Note in particular that it requires an average of fewer than 2 choices per keyframe for the artist to specify the desired 3D pose. Although we did not explicitly record the user time required to create each animation, it ranged from 5-15 minutes. Of this, a few seconds per keyframe was required to annotate each drawing with a stick figure, and 1-2 minutes per keyframe was required to browse through alternatives and select the intended pose. On average we found that it took longer to draw the initial keyframe sketches on paper than to reconstruct 3D poses using our interface.

	No. of frames	User choices
Box	4	3
Throw	7	8
Karate	5	4
Shotput	10	3
Golf	6	14
Skip	7	9
Run	6	6

Table 2 *User interaction statistics for each sequence.*

Animators typically draw such that all intended motions are visible, and false attachments are avoided. This fact was encoded as the optimization constraint described in equation 6. Figure 5 shows the visual effects of assuming that joints that *appear* to be coincident in either space or time actually are coincident. The golfer's feet stay planted on the ground, and the hands come together to grip the club.

The included video shows examples of animations created both using reconstruction alone and with our optimization stage. Note that optimization dramatically improves the results. The video examples of reduced wobble, as well as the improved adjacency of Figure 5, are intended to show the success of our interface; however, they point towards its limitations as well. A relatively small amount of imprecision will result in a small amount of wobble, or a small deviation in the adjacency of the golfer's hands. This is corrected in our optimization stage by regularizing or smoothing the motion using constraints. As the imprecision in drawing grows, so will these distortions. The user will eventually be left to choose between too much wobble, and too much smoothing.

The 'skip' example in the last row of Figure 6 shows the importance of our squashiness slider. Although at first glance this example looks precise, the bone lengths are in fact subjected to a great deal of squash and stretch. For example, the length of the upper leg shortens by nearly half

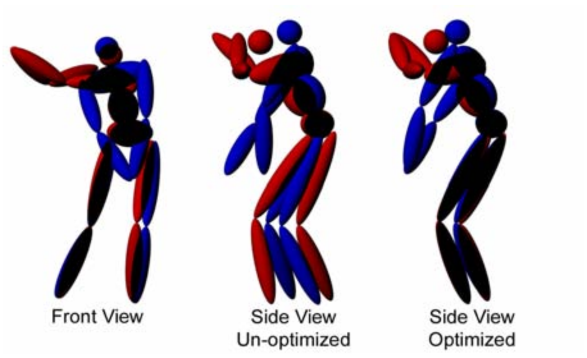


Figure 5 *(Left) Two keyframes from a golfing sequence are shown from the original drawn viewpoint. Note that the feet remain fixed and the hands come together. (Middle) The reconstructed 3D poses are shown from a perpendicular view, looking down the x-axis. Note that the feet do not remain fixed, and the hands are not together. (Right) After optimization, both of the coincidence objectives have been satisfied.*

during the ground impact, although no foreshortening effect is intended. Since the bone lengths have been drawn more imprecisely than in the other examples, a higher squashiness setting is chosen by the artist. This setting allows for greater variation in the 3D bone length, and thus greater regularization, during optimization.

6. Conclusions and Future Work

We have presented a novel interface that leverages the existing drawing skill of artists to construct rough animated sequences. By coupling a user-guided pose reconstruction algorithm with optimization, we are able to create animations despite the fact that the source drawings may have unintentional imprecision and distortion. Although the individual algorithms that make this possible are interesting and useful in and of themselves, the primary contribution of this work is that it allows an animator to create rough 3D animation almost entirely from 2D sketches, with little additional effort.

Despite the success of this interface, we feel that future enhancement would be beneficial. The method relies on a calculation of bone foreshortening to produce 3D pose. Inherent in this is an assumption that bones remain rigid and of approximately constant length. Consequently we ask artists to try to draw realistically. Alternate methods will be required to create animations from drawings that contain the truly extreme twisting, bending, and stretching that artists sometimes prefer.

This work originated due to frustration with existing IK posing interfaces, and we believe it represents a substantial improvement in ease of use for some users. Nevertheless it would be desirable to more carefully and objectively compare user performance on posing tasks.

Finally, we note that the system presented here is designed to construct rough animated sequences. Several recent animation systems including those by Liu and Popović[17] and Pullen and Bregler[19] were designed to start with rough animations as input, in order to derive more detailed or expressive animations. It would be interesting to join these methods, producing a complete path from sketch interface to final detailed animation.

7. References

1. Blair, P., *Cartoon Animation*. 1994, Laguna Hills, California: Walter Foster Publishing.
2. Bregler, C., et al., Turning to the masters: motion capturing cartoons. *ACM Transactions on Graphics*, 2002. 21(3): p. 399-407.
3. Bregler, C. and J. Malik. Tracking people with twists and exponential maps. in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition,*. 1998.
4. DiFranco, D., T.-J. Cham, and J. Rehg, Recovery of 3D Articulated Motion from 2D Correspondences. 1999, Compaq Cambridge Research Laboratory.
5. Gavrila, D.M., The visual analysis of human movement: a survey. *Computer Vision and Image Understanding*, 1999. 73(1): p. 82-98.
6. Girard, M. and A.A. Maciejewski. Computational modeling for the computer animation of legged figures. in *Proceedings of ACM SIGGRAPH 85*. 1985: ACM Press.
7. Gleicher, M. and N. Ferrier. Evaluating Video-Based Motion Capture. in *Computer Animation 2002*. 2002.
8. Hecker, R. and K. Perlin, Controlling 3D Objects by Sketching 2D Views. *SPIE - Sensor Fusion V*, 1992. 1828: p. 46-48.
9. Houy, D.R. Range of Joint Motion in College Males. in *Proceedings of the Human Factors Society*. 1983. Norfolk, Virginia.
10. Howe, N., M. Leventon, and W. Freeman, Bayesian Reconstruction of 3D Human Motion from Single-Camera Video. 1999, MERL - Mitsubishi Electric Research Laboratory.
11. Igarashi, T. and J.F. Hughes, A Suggestive Interface for 3D Drawing, in *Proceedings of the ACM Symposium on User Interface Software and Technology, UIST 01*. 2001, ACM: Orlando, Florida.
12. Igarashi, T., S. Matsuoka, and H. Tanaka, Teddy: A Sketching Interface for 3D Freeform Design, in *Proceedings of SIGGRAPH 99*. 1999, ACM SIGGRAPH / Addison Wesley Longman: Los Angeles, California. p. 409--416.
13. Korein, J.U., *A geometric investigation of reach*. 1985, Cambridge, MA, USA: MIT Press.
14. Lasseter, J., Tricks to Animating Characters with a Computer, in *SIGGRAPH 94 Course Notes No 1*. 1994: Orlando, Florida.
15. Lawrence, C.T., J.L. Zhou, and A.L. Tits, *User's Guide for CFSQP Version 2.5d*. 1993, Atlanta: AEM Design, Inc.
16. Lee, H.-J. and Z. Chen, Determination of 3D Human Body Postures from a Single View. *Computer Vision, Graphics, and Image Processing*, 1985. 30: p. 148--168.
17. Liu, C.K. and Z. Popovic, Synthesis of Complex Dynamic Character Motion from Simple Animations. *SIGGRAPH*

2002 : ACM Transactions on Graphics, 2002. 21(3): p. 408-416.

18. Moeslund, T.B. and E. Granum, A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding*, 2001. 81(3): p. 231-68.
19. Pullen, K. and C. Bregler, Motion Capture Assisted Animation: Texturing and Synthesis. *SIGGRAPH 2002 : ACM Transactions on Graphics*, 2002. 21(3): p. 501-508.
20. Rosales, R., et al., Estimating 3D Body Pose Using Uncalibrated Cameras, in *In IEEE Computer Vision and Pattern Recognition, CVPR 01*. 2001, IEEE.
21. Schmidt, D.C. and L.E. Druffel, A fast backtracking algorithm to test directed graphs for isomorphism under distance matrices. *Journal of the Association for Computing Machinery*, 1976. 23(3): p. 433-445.
22. Taylor, C.J., Reconstruction of Articulated Objects from Point Correspondences in a Single Uncalibrated Image. *Computer Vision and Image Understanding*, 2000. 80(3): p. 349-363.
23. Wilhelms, J. and A. Van Gelder, Fast and easy reach-cone joint limits. *Journal of Graphics Tools*, 2001. 6(no.2): p. 27-41.
24. Young, I.T., J.J. Gerbrands, and L.J.v. Vliet, *Fundamentals of Image Processing*. 1998: Delft University of Technology.
25. Zeleznik, R.C., K.P. Herndon, and J.F. Hughes, SKETCH: An Interface for Sketching 3D Scenes, in *Proceedings of SIGGRAPH 96*. 1996, ACM SIGGRAPH / Addison Wesley: New Orleans, Louisiana. p. 163--170.
26. Zhao, J. and N.I. Badler, Inverse Kinematics Positioning Using Nonlinear Porgramming for Highly Articulated Figures. *ACM Transactions on Graphics*, 1994. 13(4): p. 313-336.

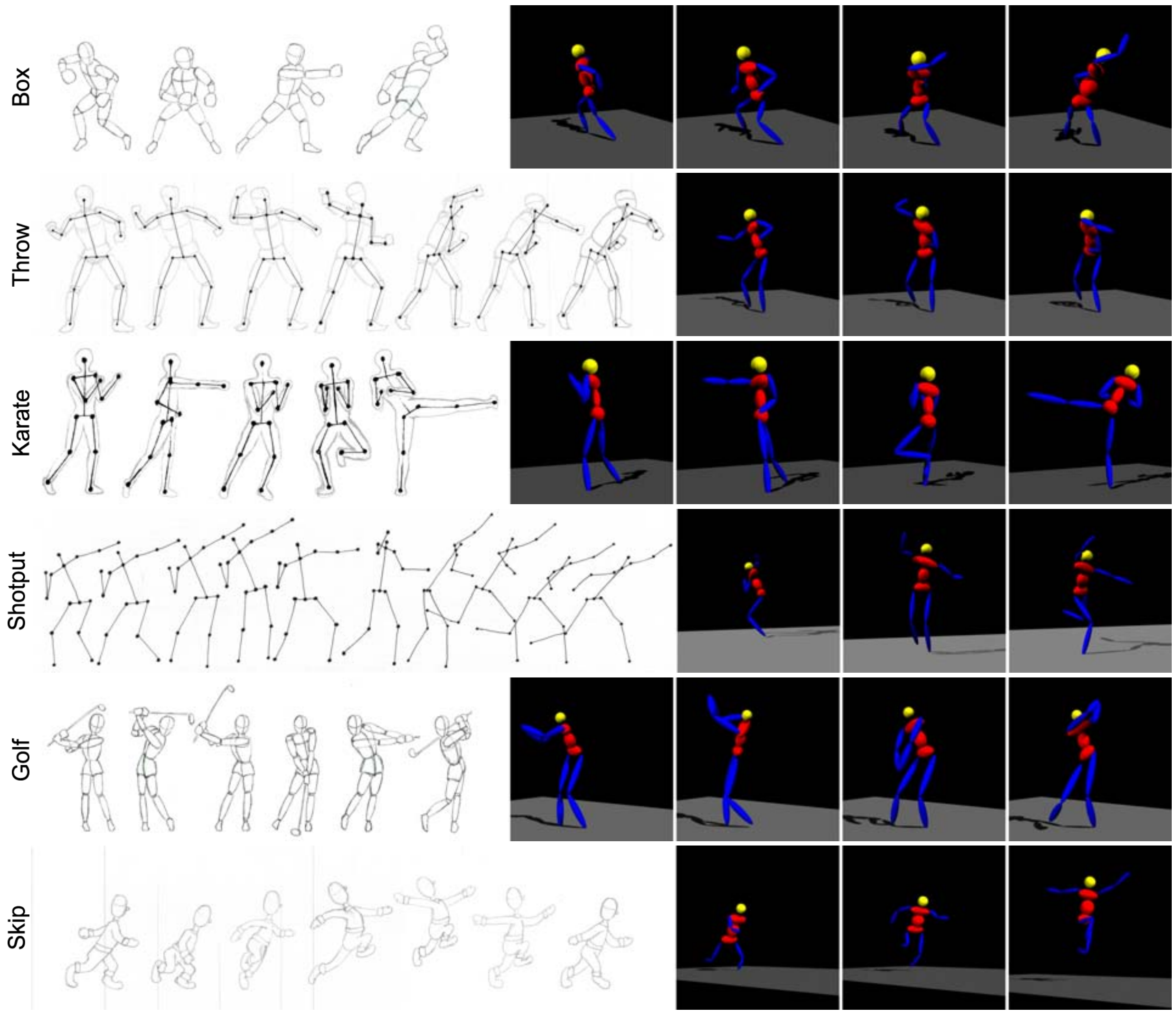


Figure 6 *Drawn keyframes are shown together with a representation of the final 3D animation. Several rows also show skeletal annotation.*

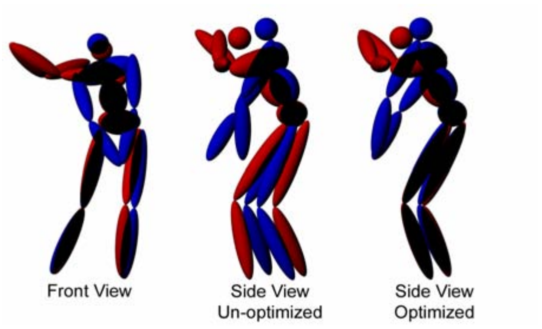


Figure 5 *(Left) Two keyframes from a golfing sequence are shown from the original drawn viewpoint. Note that the feet remain fixed and the hands come together. (Middle) The reconstructed 3D poses are shown from a perpendicular view, looking down the x-axis. Note that the feet do not remain fixed and the hands are not together. (Right) After optimization, both of the coincidence objectives have been satisfied.*

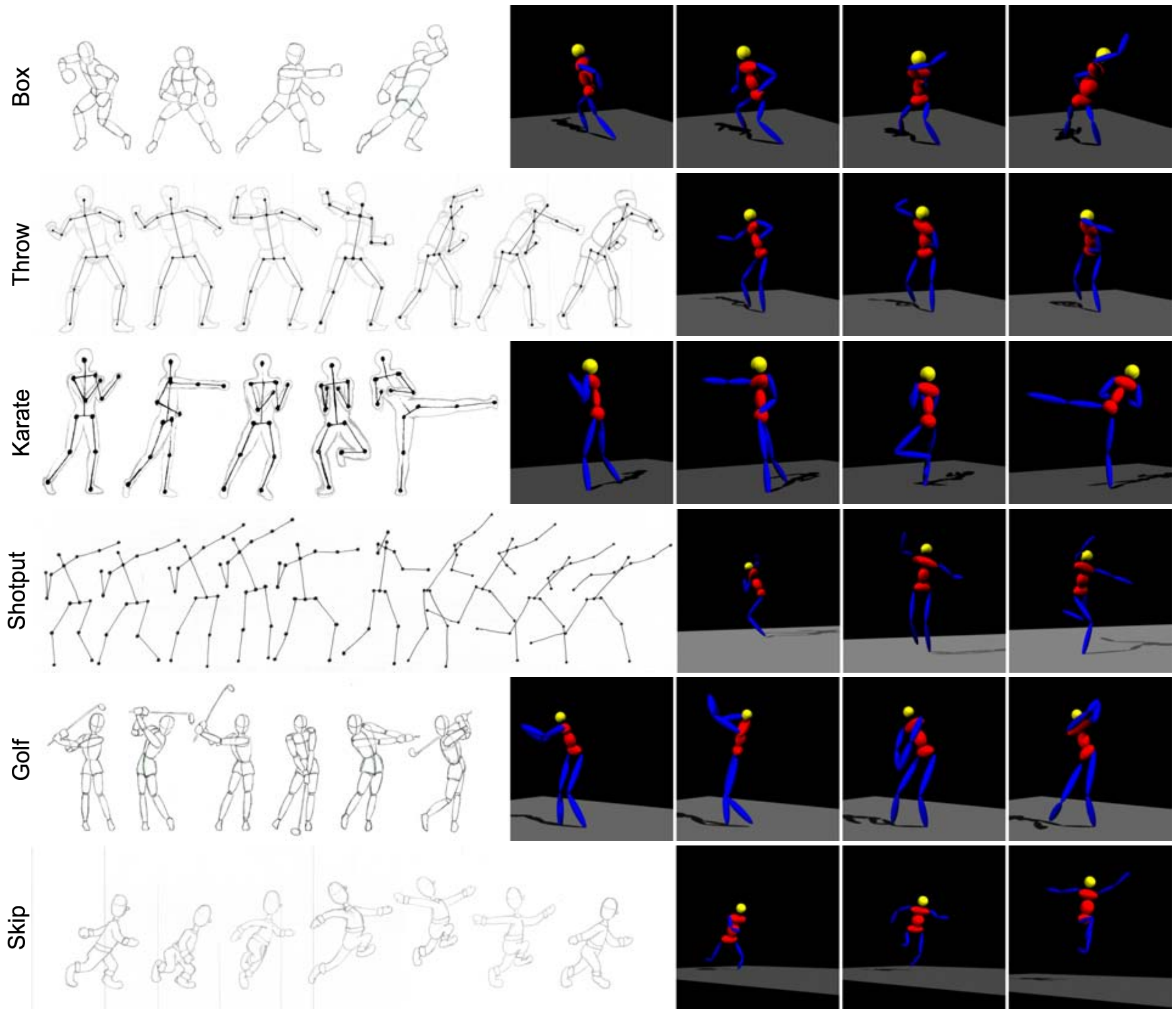


Figure 6 *Drawn keyframes are shown with a representation of the final 3D animation. Several rows also show skeletal annotation.*

Motion Doodles: An Interface for Sketching Character Motion

Matthew Thorne David Burke Michiel van de Panne

University of British Columbia[*]

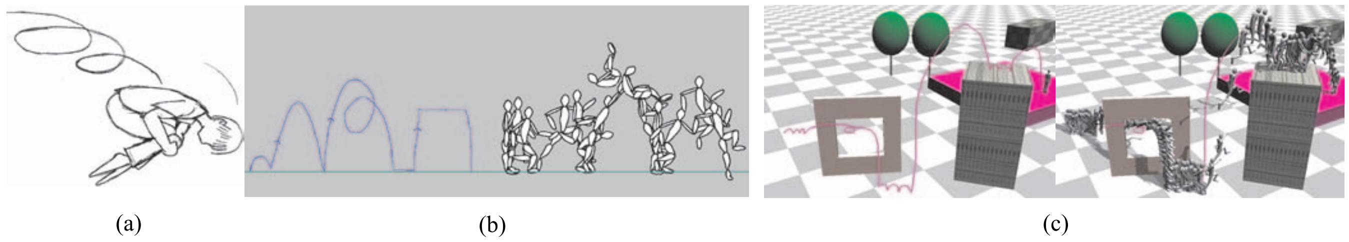


Figure 1: (a) Motion lines as used in a drawing (b) 2D motion sketch and the resulting animation (step, leap, front-flip, shuffle, hop) (c) 3D motion sketch and resulting animation (walk, flip through window, walk, leap onto building, walk, leap off building)

Abstract

In this paper we present a novel system for sketching the motion of a character. The process begins by sketching a character to be animated. An animated motion is then created for the character by drawing a continuous sequence of lines, arcs, and loops. These are parsed and mapped to a parameterized set of output motions that further reflect the location and timing of the input sketch. The current system supports a repertoire of 18 different types of motions in 2D and a subset of these in 3D. The system is unique in its use of a cursive motion specification, its ability to allow for fast experimentation, and its ease of use for non-experts.

Keywords: Animation, Sketching, Gestural Interfaces, Computer Puppetry

1 Introduction

Animation has existed as an artform for approximately 80 years and the technology used to create animations has evolved tremendously. Unfortunately, animation tools usable by non-experts remain in short supply. In this paper we develop a cursive motion notation that can be used to "draw" motions. We present an interactive animation system that interprets the notation as it is drawn. The system is simple enough to be usable by novice animators, including children. With an appropriate tailoring of the motion vocabulary, we expect that motion sketching systems may find applications in film storyboarding, theatre staging, choreography for dance and sports, and interactive games.

[*]email: mthorne,dburke,van@cs.ubc.ca

1.1 Overview

What does it mean to "sketch a motion" for a character? We propose one possible answer to this question, inspired in part by motion illustration techniques, such as the use of loops to indicate a tumbling motion as shown in Figure 1(a).

To begin using our system, the user draws the character they wish to animate. This is accomplished by drawing the body, head, arms, legs, and feet, from which a character armature is inferred. Animations can then be created by drawing motion sketches, which are interpreted on the fly to yield interactive animated motions. Figures 1(b) shows an example 2D motion sketch and the resulting animation. A subset of the motion sketch gestures can also be drawn on top of a 3D image to obtain a 3D character animation, as illustrated in Figure 1(c). Significantly, our gestures are highly visual in nature and thus serve both as a means of motion control and as a meaningful visual record (notation) of the motion.

A block diagram of the system is given in Figure 2. The path from sketching a motion to producing the motion itself is implemented as a pipeline in order to allow for animated motion to be produced while the motion sketch is still ongoing. The drawn sketch undergoes a multi-stage segmentation process to extract recognizable motion primitives from the input motion sketch. Having identified one or more motions, multiple parameters are extracted from the corresponding portion of sketch, including the start and end points, timing information, and possibly several other features. These parameters are passed on to an animation back-end, which in the current system is based on parameterized keyframe motions.

1.2 Contributions

The primary contributions of this paper are as follows:

- We present the design of a set of continuous (cursive) gestures for sketching a significant variety of motions, their locations, and their timing. These gestures are implemented in a sketch-based animation system and are demonstrated on a variety of display devices.
- We present a system that allows novices to sketch a 2D character and then draw a variety of animated motion for it, all within tens of seconds.

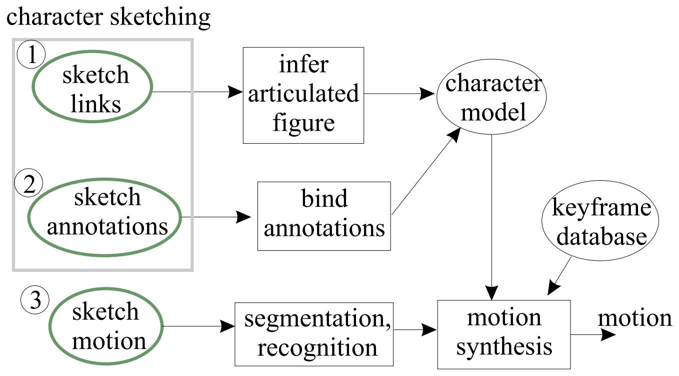


Figure 2: The sketching system.

2 Previous Work

The use of sketching in computer graphics dates back to the seminal SketchPad system[Sutherland 1963]. More recently, the sketch-based modeling systems SKETCH[Zeleznik et al. 1996] and Teddy[Igarashi et al. 1999] are inspired examples of how sketches or drawn gestures can provide a powerful interface for fast geometric modeling. The notion of "sketching a motion" is less well-defined than that of sketching an object. Nevertheless, a number of approaches have been explored. Early work explores an animation-by-example approach for a single point and fits splines to an input stream of 3D points in order to produce a smoothed version of the acted trajectory[Balaguer and Gobbetti 1995]. More recently, a considerably more sophisticated type of animation-by-example approach has been proposed[Popovic' et al. 2003], wherein the trajectory (position and orientation over time) of a rigid body can be specified by example using a 3D tracker and then "cleaned up" automatically to synthesize the physically-based motion that best fits the sketched motion. The system we propose is significantly different in that we focus on how 2D stylus input can be used to drive stylized 2D and 3D character motion.

Walking motions can be easily created by drawing a desired path on the ground plane for the character to follow, with the drawing also possibly governing the walking or running speed. Given the path and path timing, the character motion can be implemented in many different ways[Arikan and Forsyth 2002; Girard 1987; Kovar et al. 2002; Park et al. 2002; van de Panne 1997]. Our system supports many types of motion other than walking and allows for control over additional motion parameters.

Video game interfaces are perhaps one of the most readily-accessible forms of animation; a joystick and buttons provide the ability to control the direction and speed of the character as well as other sets of context-dependent predefined motions. However, game interfaces cannot fully replicate the control offered by a cursive device such as a stylus – consider for example the difficult task of cursively writing one's name with a joystick. Our system exploits a user's skills at cursive drawing and offers a greater degree of control over motions than many game interfaces. In the case of walking, for example, we offer simultaneous interactive control over step length, step height, step style, and step time.

A number of character animation systems use some form of acting as their interface. The spectrum of techniques here includes full-body acting, as used in motion capture or performance animation; the use of body silhouettes[Lee et al. 2002], various forms of computer puppetry[Sturman 1998; Oore et al. 2002; Laszlo et al. 2000], and systems that can infer a desired mapping from an actor or animator to a character using a two-pass imitate-then-modify process[Dontcheva et al. 2003]. The 2D stylus-based input of our system differs significantly from the above systems in that it aims to exploit drawing skills and not acting skills. The drawn input of our system also serves as a static visual record of the motion, something that is not available for acting-based interfaces. Labanotation[Hutchinson and Balanchine 1987] is an example of a written motion notation system for dance choreography, which can be automatically translated into 3D human figure animations[Wilke et al. 2003]. We aim for a written notation system that is much easier to learn and use, foregoing much of the detailed control that a general motion notation system can provide.

More distantly-related previous work looks at creating animations from sequentially-drawn sketches of a character, somewhat like traditionally-drawn keyframes. With appropriate constraints, a 3D character pose can be inferred for each hand-drawn frame[Davis et al. 2003].

3 Character Sketching

The two core components of our animation system are a character sketching tool and a motion sketching tool, as shown in Figure 2. We first discuss the character sketching tool, which consists of sketching the skeletal links comprising the basic character shape, followed by the optional addition of drawn annotations.

3.1 Sketching the Skeleton

A character sketch begins with drawing the links that will represent the character's articulations and basic shape. The system assumes that this is sketched in a side view using a total of 7 links, one for each of the head, torso, upper arm, lower arm, upper leg, lower leg, and foot. Each link is drawn using one continuous stroke, and the links can be drawn in any order. Links may or may not intersect when they are drawn and they may or may not contain some surface detail, such as adding in a sketched thumb, pot-belly, or nose. Figure 3(a) shows an example sketch.

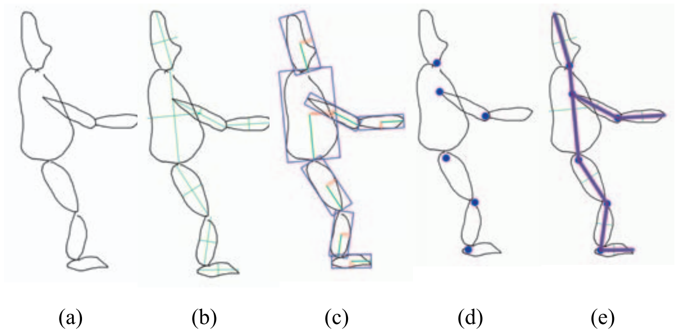


Figure 3: The process of inferring the skeleton from the sketch. (a) The seven sketched links. (b) Computed major and minor axes. (c) Oriented bounding boxes. (d) Computed joint locations. (e) Computed skeleton.

Once skeletal links have been drawn, the system automatically infers the locations of the joints, labels the links, and creates the second arm and leg. Recognizing the human form is addressed in numerous ways in the computer vision literature, but we are solving a simpler problem, one that benefits from additional constraints. Individual links do not need to be identified – each recorded stroke is already known to be a link. Also, the expected connectivity of the links is known in advance. Users can sketch the character in a wide range of initial configurations, as shown in Figure 5.

The pseudocode for inferring the skeletal structure from the sketched links is given in Figure 4. Once all seven links have been sketched, the principal axes of each link are computed as shown in Figure 3(b). Each sketched link outline is treated as a series of n points P_i, and the principal axes are computed by fitting the points to an anisotropic Gaussian distribution. If M is the mean of the points, the major and minor axes of the box are chosen as the unit-length eigenvectors U_j of the covariance matrix $\Sigma = \frac{1}{n}\sum_i (P_i - M)(P_i - M)'$. Σ is tridiagonal in 2D, so the QL algorithm with implicit shifting can be used to solve the eigenproblem[Press et al. 1992]. Next, the points are projected onto the axes to find the intervals of projection $[a_j, b_j]$ along those axes, in other words $a_j = min_i |U_j \cdot (P_i - M)|$ and $b_j = max_i |U_j \cdot (P_i - M)|$. Finally, an oriented bounding box is computed from the intervals of projection, centered at $C_{box} = M + \sum_j \frac{a_j + b_j}{2} U_j$, where $\frac{a_j + b_j}{2}$ are the extents along each axis.

```
1.  Wait for seven links to be sketched
2.  Fit oriented bounding boxes to all links
3.  For each link $i$
4.      For each major-axis end-point on link $i$, $P_i^1$ and $P_i^2$:
5.          Search all links $j \neq i$, for the closest point, $P_j$
6.          If links $i$ and $j$ are not aligned
7.              create joint $J_n$ at intersection of major axes of $i$ and $j$
8.          else
9.              create joint $J_n$ at midpoint of $P_iP_j$
10. Identify and remove all duplicate joints
11. Identify links based on connectivity
12. Create duplicate arm and leg segments.
```

Figure 4: Algorithm for inferring the skeleton from the sketch.

The next step is to determine the connectivity of the links and the locations of the resulting joints. For this, a closest link is defined for each major-axis endpoint, measured in terms of the minimal Euclidean distance from the major-axis endpoint to any point on another link. Once link j has been identified as being the closest link for a major-axis endpoint on link i, a joint is created at the geometric intersection of the extensions of the major axes of links i and j. However, this will not produce a sensible joint location if links i and j are nearly parallel. If the major axes are within 20° of being parallel, the mid-point of the line segment connecting the major axis end-points of i and j is used. This joint-creation process will result the creation of duplicate joints, such as a second 'ankle' joint being created when processing the major-axis endpoint at the toe of the foot. These duplicates are trivially removed.

Once the joints and their associated links are known, we resort to the expected topology of a human figure in order to label all the links as being the head, the torso, etc. The torso is identified as being the only link having 3 associated joints. The head link is identified as being attached to the torso and having no further joints. The arms and the legs are similarly identified by their unique connectivity. If the identification process fails, this is reported to the user. The bones for the underlying skeleton are finally constructed by connecting the appropriate joints with straight line segments. The sketched links are then redefined in the local coordinate frame of the resulting bones. The default reference pose used to start all animations is given by a standing posture that has all bones being vertical and the feet being horizontal. Figure 5 is illustrative of the variety of skeleton sketches that the system can recognize.

There are two additional joints internal to the torso that are not shown in the figures. They represent bending at the waist and the upper back, and are added to facilitate tucking during the forward and backwards somersault motions. The joints are located at fixed fractions along the torso bone. A joint is also automatically added at the ball of the foot in a similar fashion.

The current algorithm used for inferring the skeleton will fail if

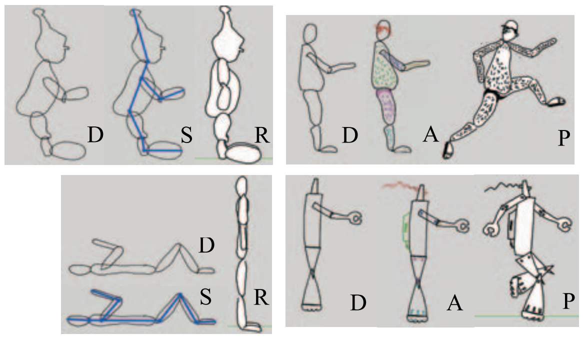


Figure 5: A variety of skeleton sketches and their inferred skeleton. *D*: the original drawing; *S*: inferred joints and the fitted skeleton; *R*: character in the reference pose; *A*: drawn annotations; *P*: an animated pose.

the arms are sketched in a downwards pose parallel to the torso, or if the character is sketched in a pose such that the hands are located close to the head, knees, or feet. These types of malformed sketches or erroneous link assignments could be addressed with some additional sophistication in the skeleton recognition algorithm. In practise the algorithm is quite robust in its current form. The system does not currently allow for the user to refine the skeleton after it is constructed by the system.

3.2 Adding Annotations

The system allows the user to add annotations which serve to decorate the links of the character. Thus, one can sketch additional features such as eyes, ears, hands, hair, a hat, a nose, and shoes. All annotations automatically become associated with the closest link. In our current version, this will result in annotations that break, such as for a sleeve that crosses multiple links. There are a number of known skinning techniques for addressing this problem, although these have not yet been implemented in the current system.

4 Motion Sketching

Sketching a motion for a character requires a degree of abstraction not present when sketching geometry. The motion sketch needs to convey a significant amount of information: (1) the type of motion; (2) the spatial location and extent of the motion; and (3) the timing of the motion. In this section we describe the design of the gestures for the 2D system, how the gestures are segmented and recognized, how the output animation is generated, and, lastly, how the 3D system works.

4.1 A Cursive Alphabet for Character Motions

Our gesture vocabulary was designed with the following principles in mind: (1) The motion gestures should be cursive, thus allowing the specification of one motion to smoothly flow into the specification of the next motion; (2) Given the limited number of very-easy-to-draw gestures that are available, the effort to draw the gesture should reflect the effort required to produce the corresponding motion. Thus, a regular walk should be easier to draw than a stiff-legged walk, which itself is easier to draw than a one-legged hop. (3) The gesture should be reminiscent of the corresponding motion, to the extent this is possible; (4) Gestures related to locomotion should allow for forwards and backwards motions; (5) Similar motions should have similar gestures; (6) Gestures should allow for the generation of stylistic variations where possible.

Guided by these principles, we developed the gesture vocabulary shown in Figure 6. These 18 motions gestures (31 when allowing for backwards-traveling variants) all allow for control over the timing of the motion by having the animated motions directly reflect the time taken to draw the gestures. They also allow for control over the start and end points of the motions, and all but two of the motions provide control over a height parameter, either the height of the swing foot during a step or the height of the body center of mass during a jump.

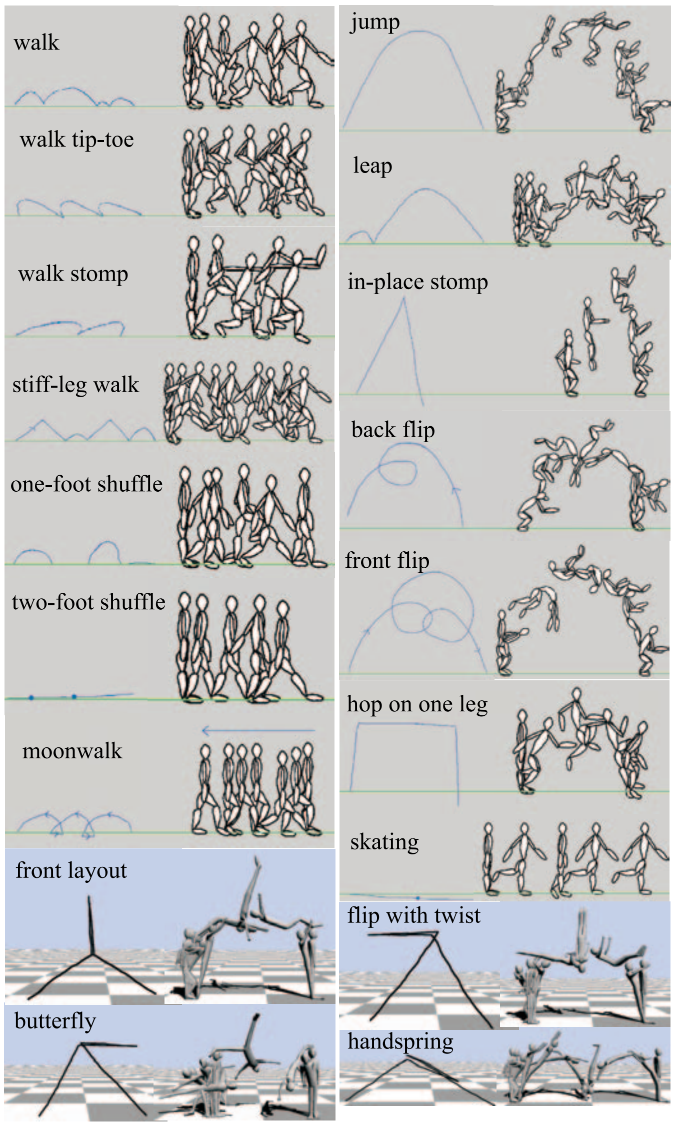


Figure 6: Gesture vocabulary.

The drawn gestures do not typically act as a direct representation of the motion of any particular part of the body. For example, the drawn arcs used for walking and its variations are evocative of the path taken by the swing foot, but are not an accurate representation of this motion. The path of the real swing foot begins from the previous foot-fall location, not the current one. Similarly, the arc drawn to represent a jump represents the location of the feet at the start and end of the arc, while the middle of the arc represents an approximate trajectory for the center of mass.

Because walks and jumps are both specified using arcs, they are distinguished by the height of the sketched arc. A "jump line" is overlaid onto the scene during motion sketching and represents the maximum arc height that is treated as a walking step. Arcs that pass over this line are treated as jumps or leaps. Additional details regarding the parsing and recognition of the input gestures are provided in the next section.

For motions such as a jump with a twist, it is difficult to find a 2D drawing gesture that is evocative of what is fundamentally a 3D motion. Our system recognizes a class of more abstract gestures in order to support such motions. The four motions appearing at the bottom of Figure 6 shows this class of gestures being employed to control various gymnastic motions.

4.2 Sketch Segmentation

The input sketch is processed in multiple stages. The first *tokenization* stage consists of taking a stream of input points from the stylus and producing a corresponding list of tokens. Figure 7(top) shows the six types of tokens that are output by this first stage and Figure 7(middle) shows an example input sketch which has been labelled using these tokens. Once the sketch input has been tokenized, a *parsing* stage is used to identify the set of admissable gestures, as shown in Figure 7(bottom). Lastly, the *motion identification* stage identifies the specific motions to be generated. Segmenting and recognizing gestures based on a regular expression grammar has a number of precedents, a good example being the framework set out in [Hammond and Davis 2003]. We now describe the segmentation steps in additional detail.

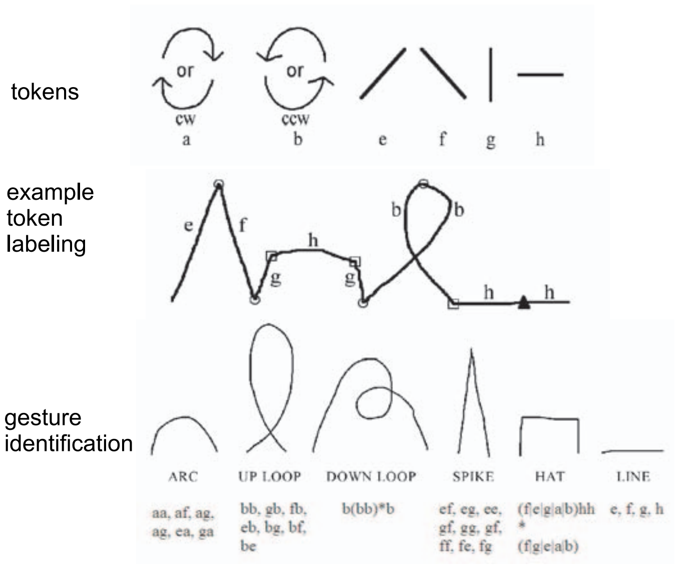


Figure 7: Segmenting the motion sketch input.

The tokenization stage processes a sequence of time-stamped input points in six steps, as shown in Figure 8. In step one, the input points are segmented based upon changes in the vertical direction of motion (thus discerning between rising and descending strokes). Step two applies a simple corner detection algorithm[Chetverikov and Szab 1999]. However, corners may be falsely identified for quickly drawn loops and arcs on slow input devices, resulting in curves being represented by only a few sparsely-spaced points. If the stylus velocity as measured by finite differences at a corner point exceeds 35% of the maximum stylus velocity, then the point is no longer regarded as a corner point. Step four classifies segments as being either straight or curved. A straight segment is defined as having $r < 1.2$, where r is the ratio of the arc length to the geometric distance between segment endpoints. Also in this step, colinear neighboring straight segments are merged. Step five adds segmentation points at locations where the stylus has paused, which are important for motions such as shuffles and skates. Lastly, step six assigns one of six tokens to each resulting segment based upon whether the segment is straight or curved, and, if straight, based upon the absolute angle of the line segments. Straight lines that make an angle of less than 30 degrees with the vertical or horizontal are assigned tokens g and h, respectively; otherwise they are assigned the token e or f, whichever is closer in terms of angle.

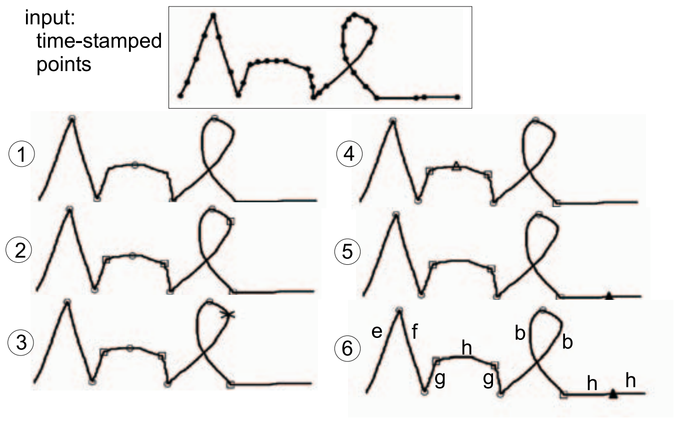


Figure 8: Tokenization consists of six steps: (1) Rough segmentation based on vertical direction of motion; (2) Corner detection; (3) Corner post-processing; (4) Merge colinear segments; (5) Identify pauses; (6) Token assignment.

Given an input sketch that has been labelled with tokens, the parsing stage groups tokens together into the set of admissable gestures shown in Figure 7(bottom). This is accomplished by matching the regular expressions corresponding to each gesture type. Lastly, the motion identification stage determines the specific motion to be executed. In some cases, the identified gestures will map directly to particular motions. For example, the gesture for a one-legged hop has a unique interpretation. In other cases, additional criteria are examined in order to disambiguate the desired motion.

In order to further distinguish between a walking step and a jump, the maximum height of the arc is used. An arc of height $h > h_{walk}$ is determined to be a jump or leap; all others are interpreted as some type of walking step. In our current implementation, a horizontal line is drawn at $y = h_{walk}$, thereby providing the animator with an easy point of reference while sketching.

An additional criteria is employed to avoid impossibly-large walking steps. Any arc of length greater than a maximum step length $d_{maxstep}$ is interpreted as a jump rather than a step. This avoids the ill-posed inverse-kinematics problems that would otherwise arise in performing such steps. Similarly, if an arc passes above $y = h_{walk}$ but does not allow sufficient ground clearance for the jump to be completed due to the character's geometry, a vertical offset is computed for the apex, allowing for a feasible jump.

Tip-toe and stomp walking steps are distinguished from regular walking steps by examining the relative position of the apex of the sketched arc with respect to its endpoints, as measured by $\alpha = (x_{apex} - x_{start})/(x_{end} - x_{start})$. A tip-toe step is identified for $\alpha < 0.35$ and stomp-step for $\alpha > 0.65$. At present, the system performs a discrete classification of each step as being a regular, tip-toe, or stomp step. A final ambiguity exists between shuffle steps and a slides, which are both represented by horizontal line segments. A shuffle step is assumed if the length of the step is less than $d_{maxstep}$ and otherwise becomes a slide.

If during a sketch the pen remains stationary for more than $0.5s$, the character is brought to a standing posture. A standing long jump occurs as a result of a sketched jump arc whenever both feet are together. If this is not the case, such as when a walking step is followed by a jump arc, the jump is classified as a leap.

Because the user's sketch is always ahead of the motion segmentation and synthesis, successfully segmented motion actions are stored in a queue for processing. It is possible that the motion queue is exhausted while the next gesture is still in the process of being drawn, in which case a pause is introduced into the output motion. This pause exists only as an artifact during the original sketching process and disappears during a motion replay.

Animators can also sketch motions directly in an environment with other animated objects. This allows for motions that need to be coordinated with existing animated objects, such as executing a leap over a falling character or jumping out of the way of a car. Because we use the time at which a motion sketch was drawn as the reference timing for the motion, coordination with existing animated motion is easily accomodated. The character animation produced during an initial 'live' sketch may not be properly synchronized with the action because of the necessary delay in recognizing gestures. However, a motion replay produces the correctly timed result.

4.3 Output Motion Synthesis

Once an input gesture has been appropriately identified and mapped to a particular motion, e.g., "single back flip", the key parameters for that motion segment are extracted, and the output motion synthesis can begin. Common parameters to all motions include the start and end positions, as well as the motion duration. Most other motions also extract a parameter relating to the location and timing of the apex of the sketched input gesture. For example, jumps and leaps are parameterized in terms of duration of ascent, duration of descent, the maximum height of the jump, and the start and end locations. Front and back flips have additional parameters describing the number and direction of rotations. The walk, walk-stomp, tiptoe, stiff-leg walk, hop, and one-foot shuffle all use the arc height parameter to control the height of the swing leg during stepping.

Once the type of motion and the related parameters are known, the desired motion could be synthesized in one of several ways. We choose to employ a parameterized keyframe-based motion synthesis technique. Motion-capture-retargeting techniques or space-time optimization techniques could also be considered, but the kind of physical realism obtained with these techniques is not one of our goals, nor are they necessarily appropriate for a system which encourages experimentation with cartoon-like super-human motions.

Each type of motion is implemented by breaking it into a fixed number of stages and then applying a number of tools: a keyframe database, a keyframe interpolator, an inverse-kinematics solver, and a means to position the center-of-mass at a specified point. As an example, the stages used to implement jumps are shown in Figure 9. A detailed description of the stages used for all motions may be found in [Thorne 2003].

The duration of each stage is determined as a fixed fraction of the input-sketch times associated with the motion. For example, a jump motion has both ascent and descent times, which are determined directly from the input sketch. The first three stages of a jump motion all determine their duration as a fixed fraction of the input-sketch ascent time. The durations of the remaining three stages correspond to fixed fractions of the input-sketch descent time.

All stages of any particular motion have an associated keyframe that defines target joint angles to be reached by the character at the end of the stage. A Catmull-Rom interpolant is used between successive keyframes. The global position of the character is controlled separately from the keyframes. For steps, the root of the character (located at the hip) is placed halfway between the known positions of the stance and swing feet. For the airborne phases of

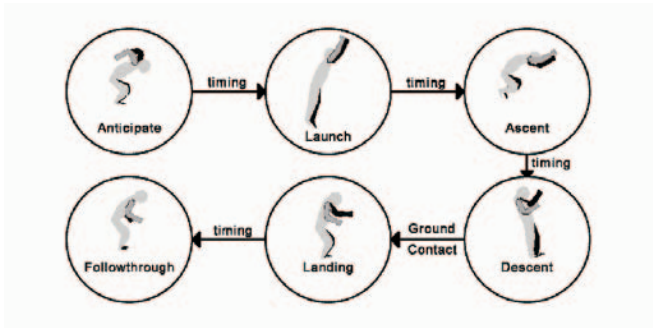


Figure 9: States and keyframes used for jumps.

jumps, flips, and leaps, the center-of-mass is directly placed at an appropriate position as determined by two parabolic center-of-mass curves, one for ascent and one for descent. These curves are fitted based upon the start, apex, and end locations of these motions.

Once the keyframes have been interpolated to obtain the pose and the skeleton has been positioned, some stage-specific modifications are implemented. These can perform a number of functions, one example being the modifications required to preserve continuity of the center-of-mass velocity upon jump landings, as required when the center-of-mass transitions from being controlled by the descending parabolic arc to being controlled by the landing-stage keyframes. Inverse kinematics is applied to the legs for all animation frames involving ground contact, such as landing and follow-through for the jump, or stance during walking.

A key issue in designing motions is making them robust to variations in the proportions of the character being animated. Thus, a character with a short torso and long legs will potentially move very differently than a character with a long torso and short legs. At present we deal with this issue primarily through the appropriate use of inverse kinematics during all ground contact phases. Also, the length of the largest possible step, $d_{maxstep}$, is dependent on the character's leg length. The remaining aspects of the motion in our current system are independent of the character proportions, being driven purely by joint angles. While generally robust, characters with extreme proportions will occasionally exhibit problems with body parts passing through the ground.

4.4 Sketching in 3D environments

The basic mechanisms used in the 2D system can be extended to work in a 3D setting, albeit with some caveats. The addition of an extra dimension introduces ambiguities into the interpretation of the 2D input sketch. A motion sketch for a 3D environment begins by positioning the camera such that it covers the desired workspace in its field of view. The sketch is then drawn directly overtop of the image produced by this fixed point of view, as shown in Figure 13. One can also sketch motion on top of a photograph by creating proxy geometry for the objects in the photo, as seen in Figure 14. The 3D character for our system is modelled in advance and not produced from a 2D sketch.

The sketching of walks, jumps, leaps, and flips can be mapped to a 3D environment in a relatively straightforward manner. The sketch is processed to find the start and end points of each gesture, as indicated by a change in the vertical direction of motion and the satisfaction of the corner metric. The identified 2D sketch segmentation points are then back-projected into the 3D scene in order to locate them in 3D. Given known 3D locations of the start and end points, the remaining 2D sketch points are back-projected onto the vertical plane V that embeds the 3D start and end points. The sketch points can now be processed in the 2D coordinate frame of V as with the 2D system. In this way, a proper apex for the motions can be extracted for arcs that are drawn "in perspective".

The above mapping process still results in a number of limitations. Motions moving directly towards the camera or away from the camera remain difficult to sketch. Additionally, some gestures become ambiguous when mapped to 3D, as shown in Figure 10. In-place stomps and stiff-legged walks become confused with the shuffle and skating motions, given that all these gestures are drawn with straight lines. There is a further ambiguity involving the direction the character is facing; the gestures for a forward-step and backwards-step cannot be distinguished in the 3D environment. Our system makes the particular assumptions shown in Figure 10 in order to resolve these ambiguities. Other assumptions, modes, use of context, or additional gestures could equivalently be used to help with disambiguation.

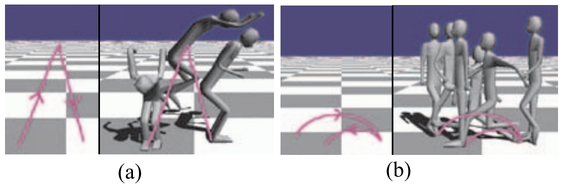


Figure 10: (a) A gesture which can be interpreted as two slides or an in-place stomp, resolved in favor of the stomp. (b) Ambiguity regarding the facing direction of the character is resolved by assuming that the character walks forwards.

Synthesizing motion in 3D occurs much as it does in the 2D case – the same features are extracted from the sketch as with the 2D system. A keyframe database serves as a back-end to fill in details of the motion that are not provided directly from the sketch or through inverse kinematics. Our prototype 3D system largely uses the same set of keyframes as in the 2D system.

There are two aspects in which 3D motion synthesis differs from its 2D counterpart: foot placement and direction smoothing. In the 2D system, the start and end point of each gesture marks where the foot (or feet) strikes the ground. In 3D, the feet are offset from the vertical plane embedding the sketch in order to allow for distinct left and right foot placements.

The facing direction of a character does not change in the 2D system but it may do so in the 3D system. A vector from the start point to the end point of a gesture is used to define the character's direction of travel. In order to have this direction change smoothly, some form of smoothing is required. Our system interpolates the heading direction over specific stages of each motion. For walking motions, this gives the character the appearance of rotating on its heel. For brevity, we refer the reader to [Thorne 2003] for a detailed explanation of this process.

5 Results and Discussion

Numerous interactive demonstrations of the system are shown in the video that accompanies this paper. Sketching both a character and a motion of the type shown in Figure 1(b) is easily done in under thirty seconds. The system can also be used to draw and animate *mech-bots*, as shown in Figure 11. These are drawn using 5 links instead of 7 and are animated using the same motion data base, except for the reverse knee bend direction and the lack of arms.

More abstract gestures allow for the system to animate wider classes of motions. Figure 12 shows the sketch of a gymnastics tumbling sequence. As with all motion gestures, the motion type, height, distance, and timing of the individual motions are derived from the motion sketch.

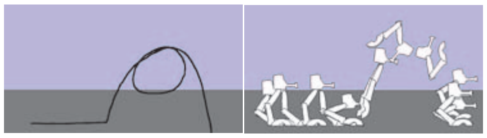


Figure 11: Animating a mech-walker: Shuffle walk and front flip.

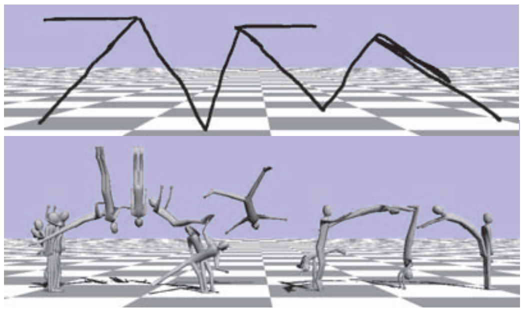


Figure 12: Sketching a gymnastics tumbling sequence: flip-with-twist, butterfly, and front-handspring.

The system is well suited for use with a variety of input devices. We have most commonly used it with a mouse during development, but other devices are more suited to producing fast and accurate gestures. Figure 15 shows the system in use on an electronic whiteboard and a Tablet PC. The SMARTboard allows the user to directly draw the desired motion sketch on top of the scene using a finger, and similarly for the Tablet PC with a stylus. Children found the system significantly easier to use with these direct input devices.

5.1 Uses of the system

The animation system presented here is not a substitute for the array of professional animation tools and techniques that are commonly used in film and games. Instead, the motion sketching system presents an alternate and highly-accesible means for users to create a certain class of character animations. The target class of motions should be tailored with the application in mind. The motions and gestures implemented in our prototype have been chosen to illustrate the motions that one might want for a storyboarding (variety of locomotion and jumping), for gymnastics choreography, and for motions that are illustrative of what can be done (moonwalk, flips).

There exist a number of commercial and research animation systems that are capable of synthesizing motions from a variety constraints. The goal of a cursive motion specification language is to make the specification of the constraints and timing a more transparent process – the users need not be fully aware of the specific parameters that drive the motion synthesis process. Unlike acting-based interfaces, the motion gestures provide a meaningful visual record of the motion, as well as allowing for "superhuman" unphysical and exaggerated motions.

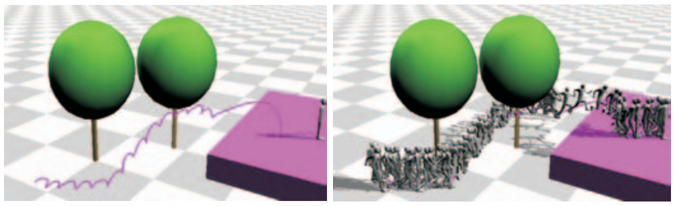


Figure 13: Walking and leaping around a set of trees.

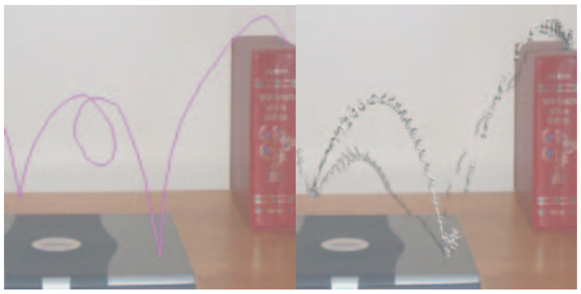


Figure 14: Stunts for a miniature character that have been sketched on top of an image with modelled 3D proxy geometry.

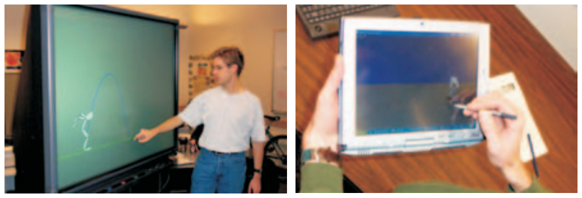


Figure 15: Sketching motions on a SMARTboard and a Tablet PC.

Over fifty people of all ages have used the system, including children as young as three. A number of anecdotal observations were made. Users rapidly learned the gesture vocabulary and enjoyed experimenting with the system in many ways. The gesture identification was occasionally problematic, with some gestures being interpreted in a fashion that did not reflect the user's intentions. We intend to further improve the robustness of the gesture recognition in order to address this issue.

Young children greatly enjoyed experimenting with the motion sketching, but found it difficult to understand the restrictions imposed on the character sketching, namely the use of seven links representing a side view of the human figure and the fact that the principal seven links had to be drawn before annotations could be added. Perhaps unsurprisingly, children would also put the system fully to the test by inevitably drawing motion gestures which had no meaningful interpretation. "Garbage in, garbage out" describes the behavior of the system in such circumstances.

Adults enjoyed the ability to sketch a character and then be immediately able to animate it. While we have conceived of the system as principally targeting novice users, an accomplished animator remarked that the system provided almost instantaneous satisfaction because of the immediacy of the animated results, something he felt was missing from present-day animation tools.

5.2 Scalability

Adding a new motion to the system requires the creation of a new gesture that can be identified by a novel sequence of tokens, as well as the implementation of an appropriate sequence of states and keyframes that is capable of generating parameterized versions of the desired motion. In the future we wish to add gestures for a variety of falling motions as well as interaction with the environment. The system can potentially be made more scalable through the use of 3D input devices and the improved use of context in specifying motions. Such additional controllability would come at the expense of increased complexity of the interface.

5.3 Limitations

The current system has a number of limitations. The system does not support the complete gesture vocabulary in 3D due to the ambiguities introduced by the 3D mapping. These can be overcome in part by making some motions context-specific or by introducing user-specified modes to resolve such ambiguities. For example, in a figure-skating context, drawn loops are probably better reserved for spins around the vertical axis rather than the head-over-heels flips that the current system is capable of.

The system is not suitable for animation that requires unique or detailed motions. A partial solution would be to animate a character in a series of successive passes with a motion layering approach[Sturman 1998; Oore et al. 2002; Dontcheva et al. 2003].

A possible improvement on our sketch segmentation scheme would be to develop a system that can be "trained by example" to recognize specific desired sets of gestures[Rubine 1991]. However, the work presented in [Rubine 1991] is not directly applicable to our problem domain because of the cursive nature of our gestures; one stroke represents a compound sequence of parameterized (and therefore variable) gestures instead of a single gesture.

6 Summary

As kinematic and dynamic methods for synthesizing motions from constraints become increasingly mature, the specification of constraints becomes a bottleneck, particularly to novice animators. We have presented a cursive language for sketching 2D and 3D character motions. This is implemented in a system that allows novices to quickly learn to sketch-and-animate a human or "mech-bot" character of their own design in tens of seconds. The technique is well-suited to take advantage of Tablet PCs and electronic whiteboards. The system is simple enough for children to use, and has other potential applications in storyboarding and the choreography of athletic routines. It is our hope that this method and its future variations play a role in making animation a more accessible media.

Acknowledgements

We would like to thank the anonymous Siggraph reviewers for their comments on earlier versions of the paper. This research was supported by NSERC post-graduate scholarships and an NSERC Discovery Grant.

References

ARIKAN, O., AND FORSYTH, D. A. 2002. Interactive motion generation from examples. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002) 21*, 3 (July), 483–490.

BALAGUER, J., AND GOBBETTI, E. 1995. Sketching 3D animations. *Computer Graphics Forum 14*, 3, 241–258.

CHETVERIKOV, D., AND SZAB, Z. 1999. A simple and efficient algorithm for detection of high curvature points in planar curves. *Proc. 23rd Workshop of the Austrian Pattern Recognition Group*, 175–184.

DAVIS, J., AGRAWALA, M., CHUANG, E., POPOVIC', Z., AND SALESIN, D. 2003. A sketching interface for articulated figure animation. In *Proc. of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 320–328.

DONTCHEVA, M., YNGVE, G., AND POPOVIĆ, Z. 2003. Layered acting for character animation. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22*, 3 (July), 409–416.

GIRARD, M. 1987. Interactive design of computer-animated legged animal motion. *IEEE Computer Graphics and Applications 7*, 6, 39–51.

HAMMOND, T., AND DAVIS, R. 2003. Ladder: A language to describe drawing, display, and editing in sketch recognition. *Proceedings of IJCAI 2003*.

HUTCHINSON, A., AND BALANCHINE, G. 1987. *Labanotation: The System of Analyzing and Recording Movement*. Theatre Arts Books.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3d freeform design. In *Proceedings of SIGGRAPH 99*, 409–416.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002) 21*, 3 (July), 473–482.

LASZLO, J., VAN DE PANNE, M., AND FIUME, E. L. 2000. Interactive control for physically-based animation. In *Proceedings of ACM SIGGRAPH 2000*, 201–208.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002) 21*, 3 (July), 491–500.

OORE, S., TERZOPOULOS, D., AND HINTON, G. 2002. A Desktop Input Device and Interface for Interactive 3D Character Animation. In *Proc. Graphics Interface*, 133–140.

PARK, S. I., SHIN, H. J., AND SHIN, S. Y. 2002. On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM Press, 105–111.

POPOVIC', J., SEITZ, S. M., AND ERDMANN, M. 2003. Motion sketching for control of rigid-body simulations. *ACM Trans. Graph. 22*, 4, 1034–1054.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C*. Cambridge University Press.

RUBINE, D. 1991. Specifying gestures by example. *Computer Graphics, Proceedings of SIGGRAPH '91 25*, 4, 329–337.

STURMAN, D. J. 1998. Computer puppetry. *IEEE Computer Graphics and Applications 18*, 1 (Jan-Feb), 38–45.

SUTHERLAND, I. E. 1963. Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, vol. 23, 329–346.

THORNE, M. 2003. *Motion Doodles: A Sketch-based Interface for Character Animation*. Master's thesis, University of British Columbia. http://www.cs.ubc.ca/grads/resources/thesis/Nov03/Matthew_Thorne.pdf.

VAN DE PANNE, M. 1997. From footprints to animation. *Computer Graphics Forum 16*, 4, 211–224. ISSN 1067-7055.

WILKE, L., CALVERT, T., RYMAN, R., AND FOX, I. 2003. Animating the dance archives. In *4th International Symposium on Virtual Reality, Archeology, and Intelligent Cultural Heritage*.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH '96*, 163–170.

Eurographics/ACMSIGGRAPH Symposium on Computer Animation (2005)
K. Anjyo, P. Faloutsos (Editors)

Spatial Keyframing for Performance-driven Animation

T. Igarashi [1,3], T. Moscovich [2], and J. F. Hughes [2]

[1] The University of Tokyo [2] Brown University [3] PRESTO, JST

Abstract
This paper introduces spatial keyframing, a technique for performance-driven character animation. In traditional temporal keyframing, key poses are defined at specific points in time: i.e., we define a map from a set of key times to the configuration space of the character and then extend this map to the entire timeline by interpolation. By contrast, in spatial keyframing key poses are defined at specific key positions in a 3D space where the character lives; the mapping from the 3D space to the configuration space is again defined by interpolation. The user controls a character by adjusting the position of a control cursor in the 3D space; the pose of the character is given as a blend of nearby key poses. The user thus can make expressive motion in real time and the resulting motion can be recorded and interpreted as an animation sequence. Although similar ideas are present in previous systems, our system is unique in that the designer can quickly design a new set of keyframes from scratch, and make an animation without motion capture data or special input devices. Our technique is especially useful for imaginary characters other than human figures because we do not rely on motion-capture data. We also introduce several applications of the basic idea and give examples showing the expressiveness of the approach.

1. Introduction

The most popular approach to character animation is keyframing, where the designer manually specifies the pose of a character as a discrete set of frames (keyframes) and the computer synthesizes the poses in the remaining frames. However, novice users have difficulty in creating fluid motion using this approach and it is very labor-intensive work. Other approaches such as motion capture and physically based simulation are available, but they are expensive to use and are not suitable for designing expressive imaginary motions. Furthermore, motion capture is mainly designed for human figures and is not directly applicable to imaginary characters.

We describe here a method that lets novice users create lively animations for arbitrary 3D characters quickly and easily using a standard input device such as a mouse. The basic idea is to directly record the user's performance or actions, that is, the user's direct manipulation of the character. We believe that this is much faster and more intuitive than traditional temporal keyframing, because the user need not mentally translate static keyframes to temporal motion during design. In performance-driven animation, what you perform and see on the screen during recording is what you obtain as the final result.

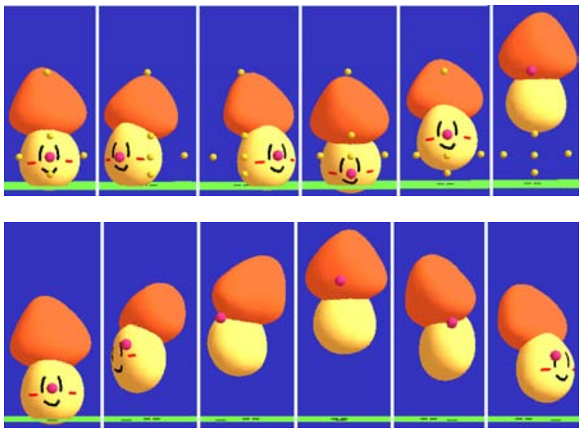


Figure 1: *Spatial keyframing with six key poses (top) and an example animation sequence using it (bottom). The user associates each pose with a location in space (yellow markers) in a preparation phase. During performance, the user moves the control cursor (red sphere) and the system synthesizes an animation sequence by blending nearby poses.*

The problem is that it is difficult to control complicated motion of a character in real time using standard input devices. A character may have many degrees of freedom (DOFs), including position, orientation, and angles at many joints, while a typical input device has only two DOFs. To control many DOFs in real time with input devices with limited DOFs, our system takes several predefined key poses and blends them in real time during performance. In a preparation phase, the user creates several key poses and associates them with specific positions in the 3D space (which we call spatial keyframes). During performance, the user moves a control cursor in the 3D space and the system synthesizes a corresponding pose by blending nearby spatial keyframes (Figure 1).

Synthesizing new poses by blending predefined poses is already common practice, and making animations from real-time performance is not new. The main contribution of this paper is to combine these methods in a practical system for making lively character animations from scratch without motion-capture data or special input devices. This paper also describes specific methods for blending poses to obtain reasonable results with very sparse key poses, as well as several extensions to the basic idea. We hope that this paper encourages the use of performance-driven approaches for character animation. The resulting animations are very different from those created using traditional methods, as they make apparent the animator's natural sense of timing. It is also much easier and more fun to create animations this way.

2. Related work

Performance-driven animation and puppeteering usually rely on a specialized input device or a motion-capture system [Stu98]. The direct mapping they assign between the character's DOFs and the device's DOFs requires a significant amount of training and expertise to control fluently. Shin et al. [SLG01] introduced methods for retargeting a motion-captured performer's full body motion to characters of different sizes. Dontcheva et al. [DYP03] proposed layered acting, where the user designs complicated motions by multiple acting passes. Our goal is to make performance-driven animation more accessible to casual users who lack expensive devices.

Synthesizing new poses by blending predefined poses is already done in many animation systems, but most of them are designed for large amounts of motion-capture data. Wiley and Hahn [WH97] associated key poses obtained by motion capture to a dense grid of points in space and linearly interpolated them. Rose et al. [RBC98] interpolated motion-capture sequences using radial basis functions to express the character's emotions. Kovar and Gleicher's system [KG04] automatically constructs a parameterized space of motions by analyzing large amounts of motion data and synthesizes a new pose by weighed interpolation of nearby poses. Our system is designed to work with very few key poses and does not require a large motion data set.

One work closely related to ours is the artist-directed inverse kinematics of Rose et al. [RSC01]. They also used radial basis functions to interpolate sparse examples in space. Our goal differs from theirs in that we focus on performance-driven animation authoring from scratch, while their focus was on controlling pre-authored motion data.

There are other related interactive systems for animation. Ngo et al. [NCD*00] used linear interpolation for manipulating 2D vector graphics. Key poses are embedded in a special structure called a simplicial configuration complex. Rademacher [Rad99] used interpolation for controlling view-dependent geometry. Key geometries are associated with specific view directions and are blended according to the current view direction. Laszlo et al. [LvPF00] combined interactive character control with physics-based simulations. They showed an example in which the horizontal and vertical motions of the mouse were directly mapped to the character's individual DOFs. The "motion doodle" system lets the user sketch the intended motion path; the system then synthesizes an appropriate motion by combining pre-authored keyframe animations [TBvP04]. Terra and Metoyer used performance for timing pre-authored key frame animation [TM04]. Donald and Henle proposed to use a haptic input device to manipulate motion capture data [DH00].

3. The User Interface

Our system consists of two subsystems. One is for designing spatial keyframes and the other is for making animation using these keyframes.

3.1 Designing spatial keyframes

The user's first task is to design a set of spatial keyframes, that is, to set poses of a character and associate them with positions in the 3D space. The user first imports a 3D articulated model into the system. We provide a standard direct-manipulation interface for the 3D model. The user can change the position of the character by dragging it within the space and change its pose by rotating its parts; the object can also be moved parallel to the ground by dragging its shadow [HZR*92].

Once the user is satisfied with a pose, the next task is to mark it as a new spatial keyframe by associating it with a position in the 3D space. To do so, the user moves the red control cursor to the target position and presses the "set" button. A small yellow ball now appears at the location of the control cursor that indicates the existence of the spatial keyframe. A spatial keyframe consists of two elements: (1) a character pose (e.g. joint angles) and (2) the xyz cursor position that corresponds to that pose. The user defines a set of these spatial keyframes by repeating the above process and these keyframes define a mapping from the control space to the set of the character's poses via an interpolation procedure described in the next section. The user can examine this mapping at any time by dragging the control cursor with the right mouse button down: the system blends the neighboring spatial keyframes around the control cursor and continuously displays the result. This synthesis is not done when the left mouse button is used; the left button is reserved for setting new poses.

The important feature of the system is that the user can set spatial keyframes at arbitrary positions in the 3D space, and that the user can start testing interesting motions with very few spatial keyframes. With only three keyframes, the user can make interesting full body motion, as shown in Figure 2. This is in contrast to linear interpolation systems that require many keyframes specified in a grid structure [WH97; NCD*00].

Note that the spatial keyframes are overlayed in the same 3D space inhabited by the character. This is in contrast to Ngo et al.'s system [NCD*00] in which keys are placed in a special configuration space. This makes it possible to establish an intuitive correspondence between the location of a spatial keyfmrame and the associated pose. For example, the keyframe for "look left" is likely to be on the left side of the screen and "look right" on the right (as in Figure 2). This intuitive mapping may be difficult to achieve when using automatic mapping such as the method introduced in [GMH04].

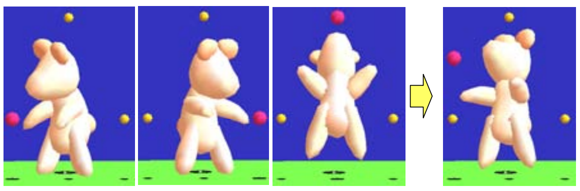


Figure 2: *A simple example with three key poses. Three key poses are associated with the three yellow balls (left). As the user drags the red control cursor with the right mouse button down, the system synthesizes a new pose by blending the three (right). Note that many joint angles as well as the character's position are controlled together.*

3.2 Making animation by performance

Having set the necessary spatial keyframes, the animator can use them to begin performing animations. In this phase, the character's pose can no longer be adjusted directly. The user moves the control cursor and the system shows the synthesized pose on the screen (Figure 3). Recording starts when the user starts dragging the control cursor after pressing the "record" button and finishes when the mouse button is released. The user can watch the resulting motion immediately by pressing the "play" button, and can watch it from any direction by changing the camera position.

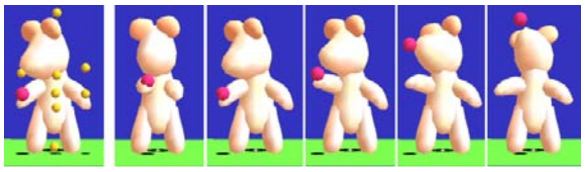


Figure 3: *Juggling. The user first sets the nine key poses as shown on the left. As the user drags the control cursor, the character performs a smooth motion as shown on the right.*

3.3 Discussion

In the current system a 2D mouse is used to control the position of the 3D control cursor and the control cursor moves parallel to the screen during dragging, so the motion of the 3D control cursor is actually 2D motion. Although interesting animations can be designed with this setup, we plan to investigate the possibility of using 3D input devices. Three-dimensionally distributed key poses may also be helpful for scripting purposes when specifying the 3D motion of the control cursor (see Section 5.3).

Designing animations in this way is really intuitive and fast. The spatial keyframe examples in this paper took only 10 to 20 minutes to design. This includes several iterations to adjust the resulting motion. After setting the keyframes, the only thing the user needs to do is to drag the control cursor. There is no need to directly edit each frame or repeat performance, as is required in layered acting [DYP03]. The time necessary to make an animation sequence is the same as the time to play it. In addition, the resulting motion is very lively because the user's direct hand motion is (indirectly) present in the animated motion. This is in contrast to the unnatural, robotic motion designed by novice users using standard keyframing. This idea is similar in spirit to the technique introduced by Terra and Metoyer [TM04], but they used performance to adjust only the timing of a predefined keyframe animation while our system allows the user to control timing and pose simultaneously.

A possible concern is that each mapping is usually specific to a single motion and thus not very reusable. This is true to some extent; the mapping defined for juggling makes little sense for other motions. However, our method allows the user to easily experiment with and design a wide variety of motions within a specific class of motion, e.g. in juggling the user can move the ball fast or slowly, high or low, clockwise or anti-clockwise. This flexibility is missing in traditional temporal keyframing methods and is critical for designing convincing motions.

4. Algorithm

This section describes the algorithm we use in the current implementation. Note that the main purpose of the following description is to provide the necessary information to implement the system, not to propose a better algorithm for motion blending. The blending of motion is essentially a difficult problem. Many techniques have been proposed, such as quaterenions and the exponential map, each with its own strengths and weaknesses. We chose this particular method because it is easy to implement, works well in practice, and satisfies certain reasonable user expectations It is our future work to implement and compare other approaches in detail.

The system takes the xyz-coordinates of the control cursor and a set of spatial keyframes (the xyz-coordinates of anchors and associated character poses) as input and returns a blended character pose. A pose is defined as a set of local transformations of the body parts. We currently do not allow translation for any part except the root, so we have 3×3 rotation matrices for all parts and one xyz translation vector for the root part. We interpolate each entry of the matrix using a radial basis function and orthonormalize the resulting matrix.

4.1 Interpolation using a radial basis function

Radial basis interpolation is useful for scattered data interpolation [Powell87]. We use the interpolation method described by Turk and O'Brien [TO02]. Each entry of each matrix is treated as a real-valued function on the control space, expressed in the form

$$f(\boldsymbol{x}) = \sum_{j=1}^{j=n} d_j \Phi(\boldsymbol{x} - c_j) + P(\boldsymbol{x})$$

where *F(x)* is a radial basis function, c_j are the marker positions, d_j are the weights, and *P(x)* is a degree-one polynomial. We currently use *F(x)* = | *x* | as a basis function. We chose this function empirically after several experiments, largely because the interpolation result follows the control cursor most faithfully. Other functions are smoother but show some oscillation effects.

The system solves for values d_j such that $f(x)$ represents the given pose at the marker locations: supposing $h_j=f(c_j)$, the constraint is represented as

$$h_i = \sum_{j=1}^{j=n} d_j \Phi(c_i - c_j) + P(c_i)$$

Since this equation is linear with respect to the unknowns, d_j and the coefficients of $P(x)$, it can be formulated as the following linear system:

$$\begin{bmatrix} \Phi_{11} & \cdots & \Phi_{1n} & 1 & c_1^x & c_1^y & c_1^z \\ \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots \\ \Phi_{n1} & \cdots & \Phi_{nn} & 1 & c_n^x & c_n^y & c_n^z \\ 1 & \cdots & 1 & 0 & 0 & 0 & 0 \\ c_1^x & \cdots & c_n^x & 0 & 0 & 0 & 0 \\ c_1^y & \cdots & c_n^y & 0 & 0 & 0 & 0 \\ c_1^z & \cdots & c_n^z & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ \vdots \\ d_{1n} \\ p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} h_1 \\ \vdots \\ h_n \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $c_i = (c_i^x, c_i^y, c_i^z)$, $\Phi_{ij} = \Phi(c_i - c_j)$,

$$P(\boldsymbol{x}) = p_0 + p_1 x + p_2 y + p_3 z \cdot$$

We obtain the interpolation function *f*(*x*) by solving the above linear system. We need to solve the linear system for each entry of the 3×3 rotation matrix. However, the large coefficient matrix on the left hand side of the above equation is identical for all nine entries, so we only need to invert the large matrix once for each joint. For the root part, we also compute each entry of the translation vector using this method.

Special care must be taken when there are fewer than four spatial keyframes and when the spatial distribution of the markers is degenerate (linearly or two-dimensionally distributed). In these cases, we apply the interpolation in a space of reduced dimensions by mapping the marker locations to the reduced space before applying the above procedure. The dimension of *P*(*x*) is also reduced accordingly. The choice of *F*(*x*) must also change to obtain true thin-plate interpolation [TO02], but we currently use *F*(*x*) = | *x* | for all dimensions and it works well.

4.2 Orthonormalization

The interpolated matrix obtained by the above procedure is not in general an orthonormal rotation matrix; we need to orthonormalize it. In some methods for orthonormalization such as Gram-Schmidt, the result is not necessarily close to the original matrix. We currently use the following iterative refinement method to orthonormalize the matrix by maintaining the balance between the three axes (Figure 4).

Suppose we have three basis vectors $\vec{x}_0, \vec{y}_0, \vec{z}_0$ and want to orthonormalize them. We first normalize them. We then compute $\vec{u}_0 = \vec{y}_0 \times \vec{z}_0$, $\vec{v}_0 = \vec{z}_0 \times \vec{x}_0$, $\vec{w}_0 = \vec{x}_0 \times \vec{y}_0$ and normalize these. Then we compute $\vec{x}_1 = (\vec{x}_0 + \vec{u}_0)/2$, $\vec{y}_1 = (\vec{y}_0 + \vec{v}_0)/2$, $\vec{z}_1 = (\vec{z}_0 + \vec{w}_0)/2$ and normalize them. We repeat the above procedure until the residual $r = (\vec{x}_n \cdot \vec{y}_n)^2 + (\vec{y}_n \cdot \vec{z}_n)^2 + (\vec{z}_n \cdot \vec{x}_n)^2$ is below a threshold or the number of repetition exceeds a predefined count. We currently use 0.000001 for the residual threshold and 10 for the count. It usually takes fewer than 3 iterations to obtain visually pleasing results. The maximum number, 10, is sufficient to detect a degenerate case.

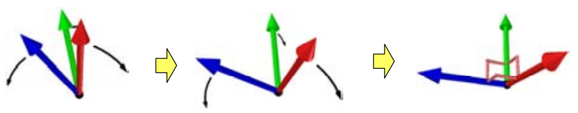


Figure 4: *Orthonormalization process. The system gradually makes the basis vectors perpendicular to each other.*

The above method is empirically designed with the goal of obtaining reasonable results robustly and quickly with a simple implementation. The outcome is satisfactory in our experience. It does not work for degenerate cases such as $\vec{x}_0 = \vec{y}_0 = \vec{z}_0 = \vec{0}$ that can arise, for example, when the control cursor is in the middle of two key poses that face completely opposite directions. In that case, the convergence fails and a skewed result is shown on the screen. Degenerate cases like this are unavoidable when creating a sufficiently nice mapping from 2-space or 3-space to the rotation group ("Sufficiently nice" in this case means that if two control points correspond to nearby rotations, then the line segment between them must correspond to a path near the geodesic path between these two rotations, which is meant to match user expectations). However, the answer is not well defined anyway in such cases from the user's point of view. The user naturally understands the existence of the degeneracy and avoids it during performance.

The method also has the nice mathematical property that if R is a rotation and M is a small perturbation to R that's orthogonal to the manifold of rotation matrices, considered as a submanifold of the manifold of all 3×3 matrices, then our process, applied to $R + tM$, yields a matrix that agrees with R to second order in t, i.e., it's essentially an orthogonal projection onto the rotation group, a property not shared by Gram-Schmidt, for instance, as can be seen by perturbing the identity by a small multiple of $\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

4.3 Why not angular parameterization?

One might ask why we do not use angular parametrizatoins such as euler angles, quaternions [Sho85], or the exponential map [Gra98] for the interpolation. Our short answer is that our domain is three-dimensional space and not sequential time, as is often the case when the typical interpolation happens. We describe two example issues that arise in this domain. Note that we only claim here that straightforward application of other approaches does not work well in our target domain. It may be possible to obtain similar results to ours by elaborating angular parameterizations [PSS02][BF01]. We leave further discussion to future publications.

The first issue is that simple angular parameterization does not behave as expected for extrapolation in our system (Figure 5). Suppose we have the two keyframes on the left. As we move the control cursor to the right, the head continues to turn if we use angular parameterization. In contrast, our method naturally keeps the head looking at the cursor. This is a design issue rather than a theoretical problem, but the basic idea behind spatial keyframing is to associate the pose with a position in space and angular parameterization breaks the natural mapping.

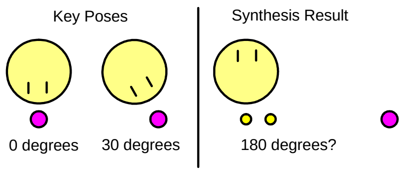


Figure 5: *An example synthesis result with straightforward angular parameterization. As the user drags the control cursor, the character rotates continuously. With our approach the character appropriately looks at the control cursor.*

The second issue is that there is a discontinuity when a part rotates 360 degrees. When using euler angles, the discontinuity is apparent. Even when using quaternions, the pose at 0 degrees and 360 degrees are located at opposite poles of the unit sphere in 4D space. This is not a problem when using a linear parameter space such as time, but we are working in two- or three-dimensional continuous parameter space. Figure 6 shows what happens when we apply angular parameterization naively. Suppose we have the four spatial keyframes shown on the left. If we move the control cursor between the first and last marker, the resulting pose is something between the second and third key pose on the left, because there is a discontinuity in angular parameter space between the first and last spatial keyframe. It might be possible to design methods that avoid this problem by elaborating on angular parameterization, but we believe that our approach (directly interpolating the rotation matrix) is more straightforward and easier to implement for our particular application domain.

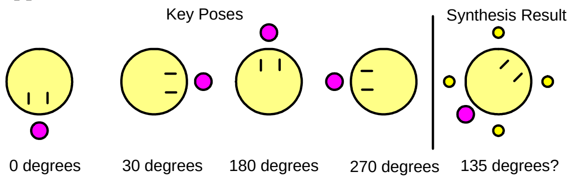


Figure 6: *Another synthesis result when using straightforward angular parameterization. If we have the four key poses shown on the left and places the control cursor at the place as shown on the right, the result is the blend of the four angles as shown on the right. Our system returns natural results by interpolating each component independently.*

Previous pose interpolation systems used angular parameterization [WH97; RSC01]; this was a reasonable decision because these systems were designed primarily to blend existing motions and the problems described above do not arise. However, our goal is the creation of a new motion from scratch by performance and it is crucial to be able to support dynamic behavior such as that shown in Figure 1.

5. Extensions

This section describes some extensions to the basic framework. These extensions are applications of existing ideas for spatial keyframing and we do not claim significant novelty here. We describe them here in order to show the possibilities of our technique and to inspire further exploration.

5.1 Inverse kinematics

Inverse kinematics (IK) is the process of determining the joint configuration required to place a particular part of an articulated character at a particular location in space. The most popular approach is to incrementally update the joint angles to satisfy the given constraints using Jacobian iteration [G M85; WW92]. In other words, the system gradually pulls the grabbed part to the target location. This means that the resulting pose is dependent on the previous pose, which can easily lead to very unnatural poses. Many solutions to this problem have been proposed, for instance using biomechanics knowledge [GGL96], constraint solving [YN03], and example-based optimization [GMH04]. However, they require manual encoding of various low-level constraints or large motion-capture datasets. Furthermore, it is difficult to include artistic control in the process.

Our spatial keyframing can be useful in adding artistic control to the inverse kinematics process. The algorithm is very simple. Instead of starting Jacobian-based refinement from the previous frame, we start the process from the synthesis result using spatial keyframing (Figure 7). This makes the resulting pose very stable. Regardless of the pose in the previous frame, the resulting pose is always the same for a given control cursor position. Our method can be seen as a subset of the one presented in Rose et al. [RSC01]. A similar technique is also used in [YKH04].

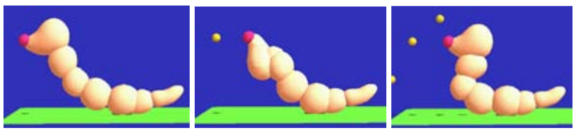


Figure 7: *Initial pose (left), standard IK result (middle) and IK with spatial keyframing (right). Standard IK can produce very strange poses after continuous operation. In contrast, IK with spatial keyframing returns stable results regardless of previous pose.*

5.2 Locomotion

Basic spatial keyframing is designed for controlling the character's pose at a fixed base position, and does not work well for animation involving travel or locomotion. The user can certainly represent a small positional change by setting the character in different places as independent spatial keyframes (as in Figure 1), but a long traveling sequence, such as walking, requires too many spatial keyframes.

One way to address this issue is to automatically change the character's position with respect to its body motion. We were inspired by interactive character control by Laszlo et al. [LvPF00], in which the user controls the character's limbs and the locomotion is generated as a result of a physically-based simulation. We would like to test similar physically-based simulation in the future, but currently use a simple rule to generate horizontal position change from the character's poses; at each point in time, the lowest point of the character is fixed relative to the ground, and the character's base position slides to satisfy the constraint [OTH02] (as it is too time-consuming to check all vertices, we manually mark the tip of each toe beforehand and use these marks for computing locomotion). When the lowest part is above the ground, the base position travels according to inertia; the system remembers the horizontal traveling speed just before the lowest point leaves the ground and continues to slide the ground with the same speed until another point touches the ground. The camera is fixed relative to the character's base position during recording.

It is possible that some point on the free leg dips lower than the supporting leg. In this case, the contact point suddenly switch, causing the character to start moving backwards. This problem can be serious if we consider all vertices of the mesh as possible contact points, but we can avoid most of the problem by using manually marked vertices only. In practice, we do experience some "waddling" motion when creating various walking motions, but the result is acceptable for novice users to quickly create simple animations. It is also very easy to interactively fix the problem by adjusting key poses and cursor trajectory when a problem occurs.

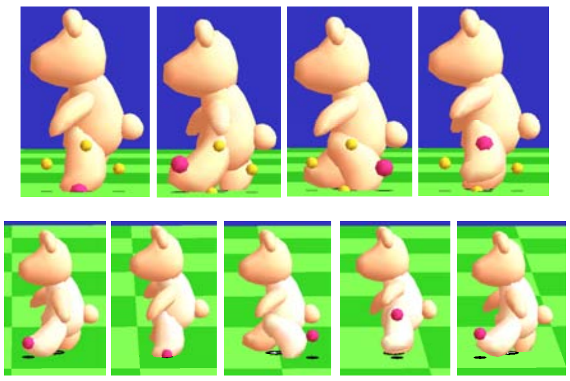


Figure 8: *Walking. Four key poses (top) and a walking animation using them (bottom). Observe that the ground slides along with the foot on the ground.*

Figure 8 shows an example. It has four key poses that represent a walking cycle. As the user moves the control cursor counterclockwise, the character makes a walking motion and the ground slides with the lower foot. The faster the user moves the cursor, the faster the ground slides, ensuring that foot-skating artifacts do not occur. Figure 9 shows another example. The three key poses represent a galloping motion. The system slides the ground to simulate inertia when the character is in the air.

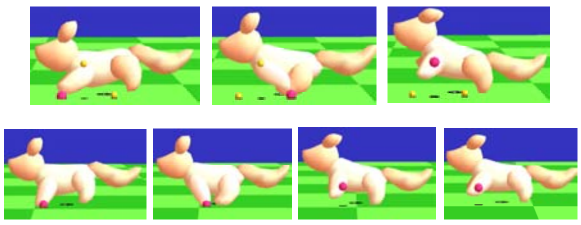


Figure 9: *Galloping. Three key poses (top) and a galloping animation using them (bottom). Observe that the ground slides along with the foot on the ground.*

5.3 Scripting

We developed spatial keyframing primarily for interactive puppetry and animation authoring by direct performance. However, the central idea behind it is to create a compact, low-DOF representation for high-DOF character poses, which should be useful for applications other than direct manipulation via a control cursor. One possibility is to use simple scripting for animation authoring. When using scripts to control a standard articulated character, individual joint angles must be specified explicitly. But using spatial keyframing, one can control rich character movement just by specifying the behavior of the control cursor in a script. Scripting with spatial keyframing is also useful for controlling mutually interacting characters.

We imagine that a set of spatial keyframes would be designed for each character, and that they would be packaged together (like the model and "rigging" of characters in animation studios). Then, the script authors would import the character with the spatial keyframe set and start writing scripts that select appropriate spatial keyframes and control the control cursor. In traditional scripting systems authors usually directly specify each joint angle [CDP00], so the spatial keyframe technique can significantly lower the bar and enrich the resulting animations. This is similar to a blend-shape interface where a character model is shipped with many adjustable control parameters, but our spatial keyframing is unique in that the control cursor lives in the same space as the character.

6. Implementation and Results

The current prototype system is implemented in Java (JDK1.4) and uses DirectX8 for 3D rendering. It currently uses articulated 3D models consisting of multiple rigid parts embedded in a hierarchical structure. Freeform surfaces guided by embedded bones are not currently supported, but it is straightforward to apply spatial keyframing to such bone structures. We use an extension to the Teddy system [IMT99] as a primary 3D modeling system, in which the user can design a painted articulated 3D character very rapidly (~10 minutes). Figure 1 and Figure 10 show example animations designed by the author. Acting with spatial keyframing is useful for expressing the characters' rich emotions in these simple actions.

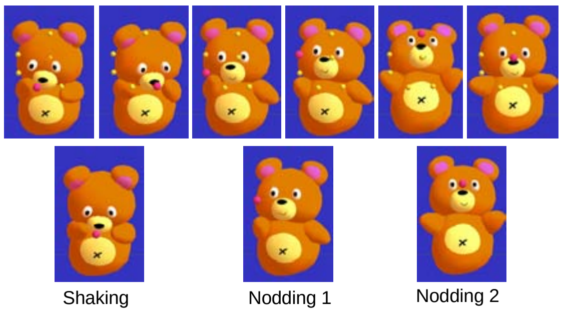


Figure 10: *An example animation. Using the six key poses (top), one can design an animation sequence in which the bear shakes his head, makes a small nod, and makes a large nod in turn seamlessly.*

Figure 11 shows an example of a highly articulated character. We experimented with various motions and found that our algorithm works well for these kinds of characters especially when the target animation is a general whole body motion such as dancing and gesturing. If the target animation requires precise placement of end-effectors, it might be better to interpolate the position of the end-effectors first and then apply inverse kinematics as in [YKH04]. It is our future work to implement this and compare the results.

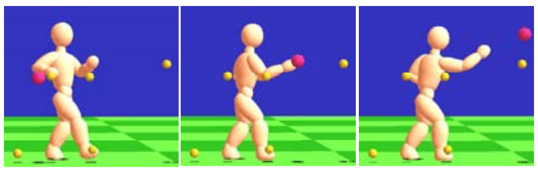


Figure 11: *An example of a highly articulated character.*

We have asked two professional artists to try the prototype system, one with extensive experience in 3D graphics and the other mainly in 2D animations. They both understood the concept quickly and started creating animation within 30 minutes. Examples are shown in Figure 12 and 13. They commented that the system was fun to use and the experience was very novel. However, the current implementation is still preliminary and the test revealed its limitations. The 2D artist had difficulty in setting individual poses with a mouse. Both wanted a mechanism to prepare multiple sets of spatial keyframes for different motions and switch from one motion to another to create meaningful stories. They noted that the system might not be immediately useful for professional production because they need precise control for each frame. They suggested that the system could be useful for real-time performance in front of audiences and for novice users.

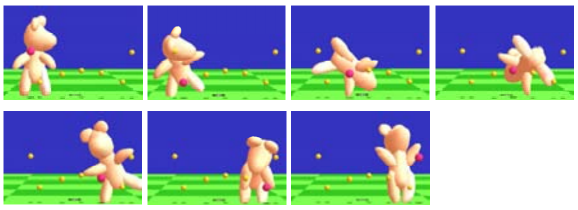


Figure 12: *An example spatial keyframes designed by a 3D expert with experience using standard keyframing. He found that spatial keyframing is much more fun to use, and that the resulting motion is very different from those created using existing methods.*

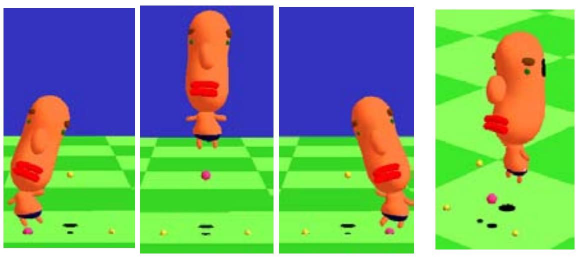


Figure 13: *An example animation created by a 2D artist. He found the system very fun to play with and inspiring but also found that it is still difficult to specify individual 3D poses with a mouse.*

7. Limitations and Future Work

The main limitation of our technique is that spatial keyframing is not directly applicable to some kinds of motions. It is very natural and effective for motions that are semantically associated with specific points in space, such as gazing and object manipulation, but is difficult to apply to more complicated motions such as speaking in sign language. Another problem is that spatial keyframing can represent only one type of motion at a time. We found that reasonably interesting animations can be designed with a single set of spatial keyframes by carefully distributing the key poses in space, but there certainly is a limit. To address these problems, we plan to investigate mechanisms for combining multiple spatial keyframe sets and achieving smooth transitions between them. How well a typical user can remember the different mappings is still an open question which we hope to answer in future research.

Spatial keyframing can easily be combined with existing methods for animation authoring. One can design more complicated motion by using spatial keyframing in layered acting [DYP03]. It is also straightforward to combine it with interactive physically based simulation to generate realistic motion automatically [LvPF00]. Motion doodles can be used to specify the trajectory of the character's locomotion while using spatial keyframing to control its pose [TBvP04].

Spatial keyframing can be seen as supplemental information added to a rigged character; skilled designers design a 3D character with predefined spatial keyframing and end users quickly author their own motion with it. We plan to develop tools to support widespread use of this framework. Examples include plug-ins for commercial 3D modeling and animation systems, 3D animation players that supports spatial keyframing, and a library of 3D articulated characters with pre-authored spatial keyframes.

References

[BF01] Buss, S.R., Fillmore, J.: Spherical Averages and Applications to Spherical Splines and Interpolation, ACM Transactions on Graphics, 20, 2, (2001), 95-126.

[CDP00] Cooper S., Dann W., Pausch R.: Alice: A 3-D Tool for Introductory Programming Concepts. *Journal of Computing Sciences in Colleges, 15,* 5 (2000), 107-116.

[DH00] Donald, B. R., Henle, F.: Using haptic vector fields for animation motion control. In Proceedings of IEEE International Conference on Robotics and Automation, (2000).

[DYP03] Dontcheva M., Yngve G., Popovic' Z.: Layered Acting for Character Animation. ACM Transactions on Graphics, 22, 3 (2003), 409-416.

[GGL96] Gullapalli V., Gelfand J. J., Lane S. H.: Synergy-based Learning of Hybrid Position/Force Control for Redundant Manipulators. In *Proceedings of IEEE Robotics and Automation Conference,* (1996), 3526-3531.

[GM85] Girard M., Maciejewski A. A.: Computational Modeling for the Computer Animation of Legged Figures. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85), 19,* 3 (1985), 263-270.

[GMH04] Grochow K., Martin S. L., Hertzmann A. Popovic' Z.: Style-based Inverse Kinematics. *ACM Transactions on Graphics, 23,* 3 (2004), 522-531.

[Gra98] GRASSIA, F. S.: Practical parameterization of rotations using the exponential map, Journal of Graphics Tools archive, 3, 3, (1998), 29-48.

[HZR*92] HERNDON K. P., ZELEZNIK R. C., ROBBINS, D. C., CONNER, D. B. SNIBBE, S. S., VAN DAM A.: Interactive Shadows. In *Proceedings of UIST '92*, (1992), 1-6.

[IMT99] IGARASHI T., MATSUOKA S., TANAKA, H.: Teddy: A Sketching Interface for 3D Freeform Design. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, Los Angeles, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, (1999), 409-416.

[KG04] KOVAR L., GLEICHER M.: Automated Extraction and Parameterization of Motions in Large Data Sets. *ACM Transactions on Graphics*, *23*, 3 (2004), 559-568.

[LvPF00] LASZLO J., VAN DE PANNE, M., FIUME, E.: Interactive Control for Physically-based Animation. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 2000, 201-208.

[NCD*00] NGO T., CUTRELL D., DANA J., DONALD B., LOEB L., ZHU S.: Accessible Animation and Customizable Graphics via Simplicial Configuration Modeling. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 2000. 403-410.

[OTH02] OORE, S., TERZOPOULOS, D., HINTON, G.: A desktop input device and interface for interactive 3D character animation, *Proceedings of Graphics Interface 2002*, (2002), 133-140.

[Pow87] POWELL M. J. D.: Radial Basis Functions for Multivariable Interpolation: A Review. In *Algorithms for Approximation*, J. C. Mason and M. G. Cox, Eds. Oxford University Press, Oxford, UK, (1987), 143-167.

[PSS02] PARK, S. I., SHIN, H. J., SHIN, S. Y.: On-line Locomotion Generation Based on Motion Blending, In Proceedings of Symposium on Computer Animation, (2002), 105-111.

[Rad99] RADEMACHER P.: View-Dependent Geometry. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, (1999), 439-446.

[RBC98] ROSE C., BODENHEIMER B., COHEN M.: Verbs and Adverbs: Multidimensional Motion Interpolation Using Radial Basis Functions. *IEEE Computer Graphics and Applications 18*, 5 (1998), 32-40.

[RSC01] ROSE III C. F., SLOAN P.-P. J., COHEN M. F.: Artist-Directed Inverse Kinematics Using Radial Basis Function, Interpolation. *Computer Graphics Forum*, *20*, 3 (2001), 239-250.

[SLGS01] SHIN H. J., LEE J., GLEICHER M., SHIN, S. Y.: Computer Puppetry: An Importance-Based Approach. *ACM Transactions on Graphics*, *20*, 2 (2001), 67-94.

[Sho85] SHOEMAKE, K.: Animating Rotations with Quaternion Curves. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, *19*, 3 (1985), 245-254.

[Stu98] STURMAN, D. J.: Computer Puppetry. *IEEE Computer Graphics and Applications*, *18*, 1 (1998), 38-45.

[TBvP04] THORNE M., BURKE, D., VAN DE PANNE M.: Motion Doodles: An Interface for Sketching Character Motion. *ACM Transactions on Graphics*, *21*, 3 (2004), 424-431.

[TM04] TERRA S.C.L., METOYER R.A.: Performance timing for keyframe animation. In *Proceedings of SCA 2004*, (2004), 253 - 258.

[TO02] TURK G., O'BRIEN J. F.: Modelling with Implicit Surfaces That Interpolate. *ACM Transactions on Graphics*, *21*, 4 (2002), 855-873.

[WH97] WILEY D.J., HAHN J.K.: Interpolation Synthesis of Articulated Figure Motion. *IEEE Computer Graphic and Applications*, *17*, 6 (1997), 39-45.

[WW92] WATT A., WATT M.: *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.

[YKH04] YAMANE, L., KUFFNER, J. J., HODGINS, J. K.: Synthesizing animations of human manipulation tasks. *ACM Transactions on Graphics*, *23*, 3 (2004), 532-539.

[YN03] YAMANE K., NAKAMURA Y.: Natural Motion Animation Through Constraining and Deconstraining at Will. *IEEE Transaction on Visualization and Computer Graphics*, *9*, 3 (2003), 352-360.

EUROGRAPHICS 2005 / J. Marks and M. Alexa
(Guest Editors)

Volume 24 (2005), Number 3

Interactive Design of Botanical Trees using Freehand Sketches and Example-based Editing

Makoto Okabe[1], Shigeru Owada[1,2] and Takeo Igarashi[1,3]

The University of Tokyo[1], Sony CSL[2] and JST PRESTO[3]

Abstract

We present a system for quickly and easily designing three-dimensional (3D) models of botanical trees using freehand sketches and additional example-based editing operations. The system generates a 3D geometry from a two-dimensional (2D) sketch using the assumption that trees spread their branches so that the distances between the branches are as large as possible. The user can apply additional gesture-based editing operations such as adding, cutting, and erasing branches. Our system also supports example-based editing modes in which many branches and leaves are generated by using a manually designed tree as an example. User experience demonstrates that our interface lets novices design a variety of reasonably natural-looking trees interactively and quickly.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques - Interaction Techniques

1. Introduction

3D models of botanical trees are important in geographical landscape simulation, cityscape design, virtual reality, consumer games, and other fields of 3D graphics. However, designing tree models is challenging because trees have enormous structural complexity.

There are two major methods for obtaining tree models. One is a rule-based approach, such as L-systems, and the other method places predefined generic models in a library or modifies their parameters. Rule-based systems allow users to design a wide variety of realistic trees. However, it is difficult for novice users to design trees of the desired appearance using rule-based systems because the inputs to such systems (rules and parameters) are very different from their output (3D geometry). Moreover, while one can quickly obtain typical trees by using predefined generic models, it is often difficult or impossible to design a desired tree by modifying predefined models.

This paper proposes a system for designing 3D botanical trees based on a sketching interface and example-based control. Sketch-based interfaces [ZHH96][IMT99] facilitate the rapid construction of 3D models and programming-by-example interfaces [Cyp91][MWK89] by automating repetitive operations. Our contribution is to propose intuitive modeling interfaces for trees, which free the user from complicated rules or parameters. The modeling process in rule-based systems can be seen as a deductive process, in that the final 3D geometry is derived from abstract production rules. Our sketch-based method can be seen as an inductive process, in that the user specifies the 2D appearance of the model directly and the system then generates a 3D structure by inferring hidden parameters.

The main purpose of our system is to construct 3D tree geometries from users' 2D sketches based on the simple assumption that botanical trees tend to spread their branches in such a way that the distances between branches are as large as possible. This enables the user to design 3D tree geometries intuitively using standard 2D input devices. This assumption is an overly simplistic description of the growth process of real trees, but it is fast and general enough for quickly designing reasonable-looking trees from typical sketches. Our algorithm also considers the fact that most users draw branches that extend sideways and omit those that extend toward or away from the screen.

In addition to the sketching interface, our system also supports three example-based editing modes: branch multiplication, leaf arrangement, and branch propagation. These editing modes allow the user to construct complicated trees by providing a few examples and free the user from drawing

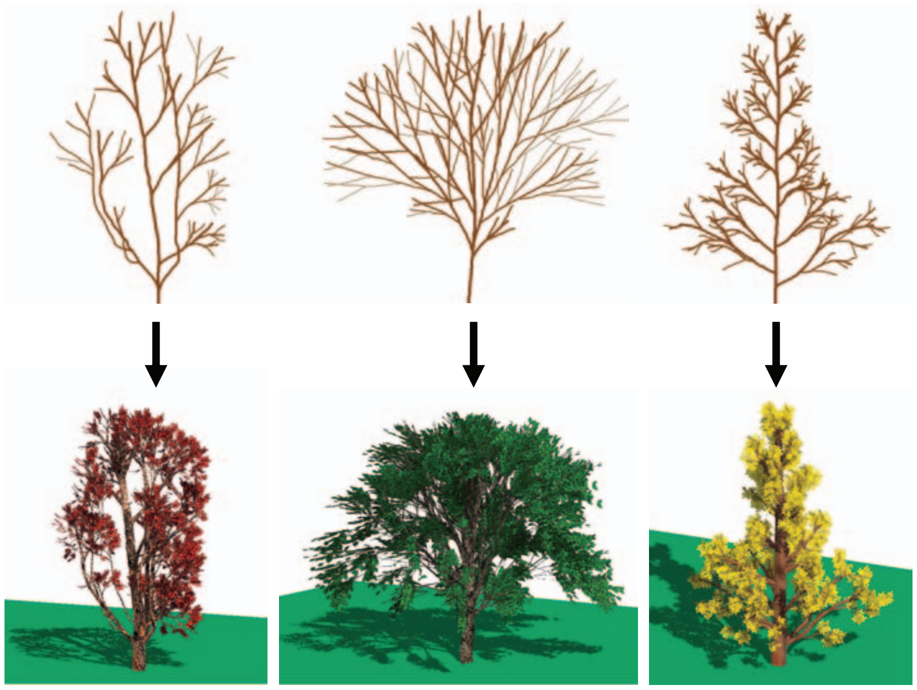


Figure 1: *Simple 2D sketches of botanical trees (above) and 3D polygon models, which are a red young tree, a zelkova, and a maidenhair tree (below). The initial 3D trees were created from the 2D sketches automatically using our algorithm. Additional branches and leaves were then attached to the trees using our prototype system. Each process took several minutes.*

all individual branches or specifying rules or parameters directly. Figure 1 shows sample 2D sketches and the resulting 3D trees created using our system.

Our goal is not to simulate the principles of nature, but to assist the user's creative design process. Our system supports several semi-automatic modeling functions that simulate some of the morphological properties of natural trees, but the user's intention expressed in the input sketch always has higher priority. Our current prototype system is designed only for trees, but many of the proposed interfaces can be used to design other 3D plants, such as flowers.

2. Previous Work

2.1. Modeling Plants

Lindenmayer proposed the formalism of L-systems [Lin68], and Prusinkiewicz improved them [PHHM96]. Subsequent research has expanded L-systems for simulating a wide range of interactions between a plant and its environment [PJM94][MP96], and for increasing realism and supporting the interactive modeling process by using positional information [PMKL01]. Boudon proposed a method to create trees more intuitively and interactively taking advantage of L-systems and demonstrated their method by creating models of bonsai trees [BPF*03]. Some other rule-based approaches have also been proposed to address the limitations of older L-systems [AK84][WP95]. Deussen and Lintermann developed the Xfrog system in order to combine the power of a rule-based approach with the intuitiveness of generic tree methods [DL97][LD99]. Deussen also proposed a non-photorealistic rendering method for 3D trees [DS00] that represents leaves as simple particles.

Other research has attempted to reconstruct 3D tree geometry from multiple photographs. Sakaguchi derived 3D volume data from multiple photographs of a tree and reconstructed 3D tree geometry by growing it upwards from its roots [SO99], but this method can produce undesirable geometries and needs some heuristics. Shlyakhter proposed a method that uses a visual hull to construct a trunk and major branches and an L-system for finer branches [SRDT01]. Reche proposed a method to capture a real-world tree as a volume with opacity and color values [RMMD04]. Maierhofer and Tobler proposed a user interface that makes it easier to specify modeling parameters by replacing numerical parameters with a more intuitive set of graphical modeling primitives [MT02]. Ijiri adopted the notion of floral diagrams and inflorescences, and proposed a method to design flowers, while preserving correct botanical structures [IOOI05].

2.2. Sketch-based 3D Modeling

Many researchers have proposed methods for constructing 3D models from user-defined 2D drawings. These include the reconstruction of rectilinear models covered by planar faces by solving constraints [EyHBE97] or using optimization-based algorithms [SL96a][SL96b], the reconstruction of the 3D geometry of a 3D curve using energy minimization [PK89], and using symmetric relations [TNT89].

Our particular interest is an interactive sketching interface for designing 3D models using 2D gestures. The Sketch system [ZHH96] is for designing 3D scenes consisting of simple primitives, and the Teddy system is for designing freeform models [IMT99]. Several extensions of the original Teddy system have recently been proposed [ONOI04].

2.3. Example-based Interfaces

The user interface research community has been investigating example-based user interfaces in which the computer automates some of the repetitive tasks by observing a user's example operation [Cyp93]. The Eager system detects repetition in the user's operation and suggests automation [Cyp91]. In the Metamouse system, the user explicitly trains the system by demonstration [MWK89]. Igarashi et al. [IMKT97][IKTM98] implemented a prediction mechanism on top of their beautification-based 2D drawing system. Their system predicts the user's next drawing operation based on the drawing already in the scene and displays the predicted results as multiple candidates. This helps the user to design relatively complicated scenes without drawing all of them manually. This idea has been extended to 3D-model design [IH01]; like the 2D version, this system suggests operations that the user is likely to do next.

3. User Interface for Modeling Trees

3.1. Overview

First, we overview the process of modeling a typical 3D tree (Figure 2). The user begins to model a 3D tree by sketching a simple 2D tree using a mouse or pen tablet (Figure 2 (a)). After sketching a tree, the user presses the "3D" button and the system constructs 3D tree geometry from the 2D sketch (Figure 2 (b)). This process takes several seconds. Now the user has a 3D tree, and manually adds or removes branches with simple gestures. The user can also apply example-based editing modes to generate complicated trees. The user switches between modes by pressing the corresponding buttons. In branch multiplication mode, the user can add more child branches to a designated parent branch using the existing child branches as examples (Figure 2 (c)). The leaf-arrangement mode lets the user place leaves around a branch following typical leaf arrangement patterns by providing a few examples (Figure 2 (d)). Finally, branch-propagation mode copies the child branches of a parent branch to other parent branches (Figure 2 (e)). The user can undo or redo any actions while modeling a tree by pressing the "Undo" or "Redo" button.

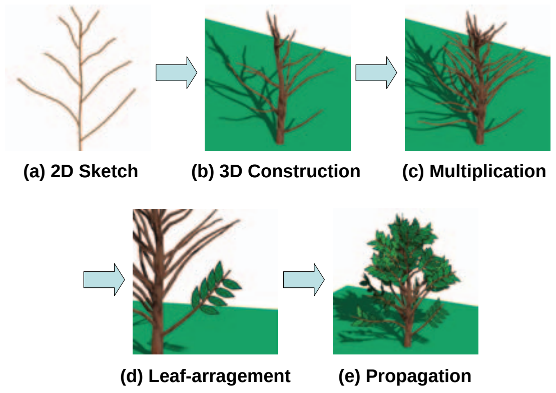


Figure 2: *(a) The user draws a 2D sketch of a tree; (b) the system generates a 3D tree when the user presses the 3D button; (c) a denser 3D tree model is obtained in multiplication mode; (d) leaves are added to a branch in leaf-arrangement mode; and (e) leaves are propagated to other branches in propagation mode.*

3.2. Sketching a 2D Tree

The user begins to model a 3D tree by sketching simple 2D strokes that represent branches or leaves. An open stroke makes a branch and a loop stroke makes a leaf polygon as a bounding box for the stroke (Figure 3). An incoming stroke is connected to the nearest existing branch and becomes its child branch or leaf. When a branch is attached to a parent it changes its form so that it is connected to its parent precisely. The base point moves to the nearest point along the parent branch; the terminal point remains fixed; and the displacements of the intermediate points are interpolated.

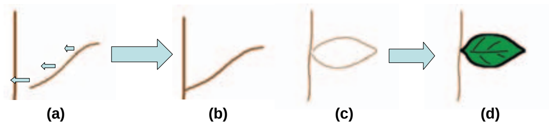


Figure 3: *Drawing a branch (a-b) and drawing a leaf (c-d).*

When the user draws a branch using a single stroke, the system applies default geometry (varying radii along the branch) to the branch. Optionally, the user can draw a pair of almost parallel strokes to define the detailed shape of the branch. The two strokes correspond to the silhouette of the branch (Figure 4). The system sweeps a circle along the two strokes to construct the branch geometry. After a detailed geometry has been specified using a pair of strokes, that geometry becomes the default setting and is then applied to subsequent branches drawn as single strokes.

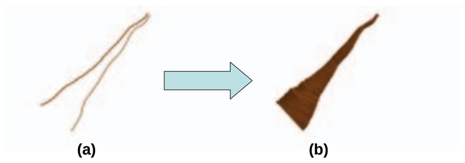


Figure 4: *Two strokes representing a branch's silhouette (a) and the resulting 3D branch geometry (b).*

The user can edit branches using two gestural editing operations: cutting and erasing. An editorial stroke intersecting a branch once cuts the branch. The distal end of the branch is erased, as are all the descendant nodes spawned on that side. An editorial stroke intersecting a branch two or more times erases the branch. In this case, all the descendant nodes are erased.

3.3. Creating a 3D Tree from a 2D Sketch

The system constructs a 3D tree when the user finishes drawing a 2D tree and requests its construction by pressing the "3D" button. (This takes several seconds.) After construction, the user can rotate the tree and see it from different viewpoints. Figure 5 shows some examples. The basic strategy is to make trees spread their branches so that the distances between them are as large as possible. The system also gives detailed depth modulation to each branch. The algorithm is described in Section 4.

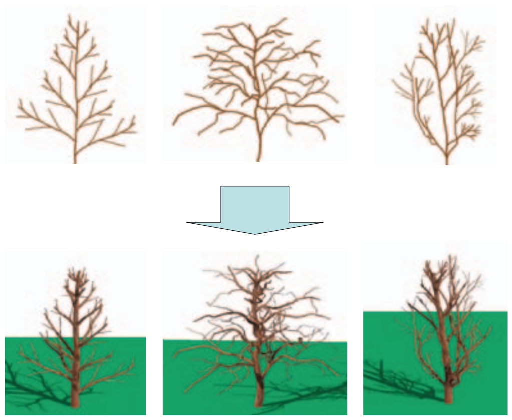


Figure 5: *The upper snapshots are freehand sketches, and the lower snapshots are the results of 3D construction.*

After constructing a 3D tree, the user can add a branch or a leaf by drawing a stroke as in the 2D case. The 2D strokes are projected onto a plane that is parallel to the screen and passes through the base point on the parent branch. The user can also edit branches or leaves by drawing cutting and erasing strokes.

3.4. Example-based Branch Multiplication

In this editing mode, the user can click a branch to increase the density of its child branches, while preserving the overall appearance of the tree. Each click adds a new branch to the parent branch; the user can add as many branches as desired by successive clicking. Figure 6 shows an example. The detailed algorithm for computing the position, orientation, and shape of the new branch is given in Section 4.3.

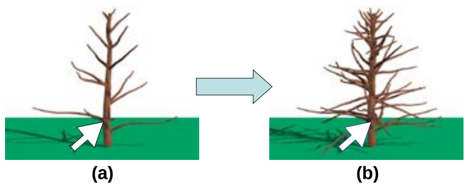


Figure 6: *The user adds child branches to a branch by successive clicking.*

3.5. Example-based Leaf Arrangements

This mode allows the user to place leaves according to typical leaf-arrangement patterns (alternating patterns and whorled patterns). The interface is similar to the user interfaces proposed by Igarashi and Hughes [IH01]. The user adds a few sample leaves manually and the system infers possible arrangement patterns from these examples.

The system then generates further leaves based on the inferred patterns and shows the results as thumbnail previews (Figure 7). When the user likes a result, he or she clicks the corresponding thumbnail to use it. If the user does not like the result, he or she can ignore it and proceed to the next operation. The current implementation supports three suggestion engines. One adds leaves to the base of a given set of whorled leaves. Figure 8 shows an example. When the system observes a set of leaves at the same base position, it infers that the user wants to use the whorled pattern and adds a leaf to the set. The system also rearranges the existing leaves so that they are distributed around the parent branch equally. The user can increase the number of leaves by clicking the corresponding thumbnail successively (Figure 8).

Another engine extends the whorled leaves along the parent branch (Figure 9). When the system observes a set of whorled leaves attached to a base position and a leaf at a different position, it infers that the user wants to add more sets of whorled leaves with the given spacing. The system fills the remaining region of the parent branch with sets of whorled leaves.

The final engine extends alternate leaves along the parent

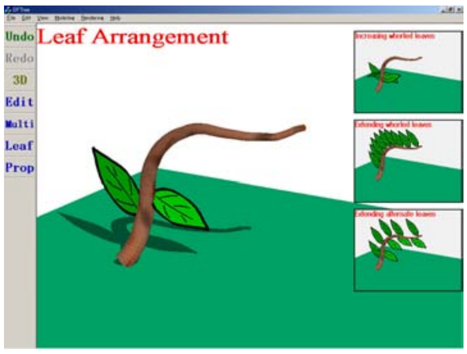


Figure 7: *Thumbnails are presented to the user. The user can adopt a prediction result by clicking the corresponding thumbnail.*

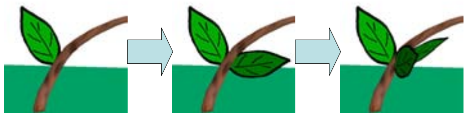


Figure 8: *Increasing the number of leaves in a set of whorled leaves.*

branch (Figure 10). When the system observes two leaves placed at different positions, it infers that the user wants to use the alternating pattern and adds additional leaves along the parent branch. The system uses the rotational angle between the sample leaves to place new branches, generating spirally arranged leaves.

3.6. Example-based Branch Propagation

This editing mode lets the user propagate the local arrangement of branches and leaves to the entire tree. When the user clicks a branch (reference branch), the system copies the child branches and leaves of the reference branch and pastes them on all other branches (target branches) on the tree. The current system supports two types of propagation;

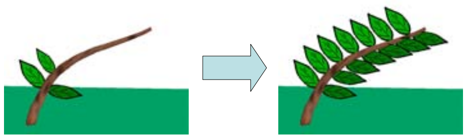


Figure 9: *Extending whorled leaves along the parent branch.*

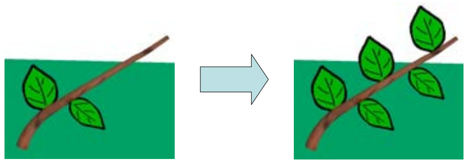


Figure 10: *Extending alternating leaves along the parent branch.*

the user switches between them using a toggle button on the screen.

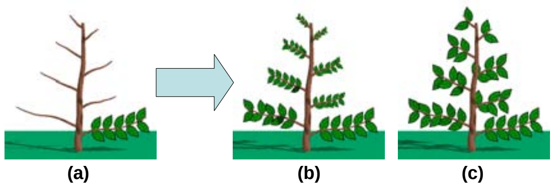


Figure 11: *The original state of a tree (a), propagation with scaling (b) and propagation without scaling (c).*

One type of propagation is that with scaling (see Figure 11 (b)). When the system pastes the child branches and leaves on a target branch, it scales them so that the reference branch matches the target branch. This mode is useful for propagating detailed branching to other branches. In propagation without scaling (Figure 11 (c)), the system pastes the child branches and leaves without scaling. The tip of the reference branch is placed at the tip of the target branch. If the target branch is shorter than the reference branch, the system uses only the child branches and leaves near the tip of the reference branch. This mode is useful for propagating leaves around the reference branch because all the leaves are of similar size and spacing all over the tree.

3.7. Reproduce a Tree from Overall Sketching

Once the entire modeling process is completed, the user can duplicate the finished model using two strokes representing the trunk and the silhouette of the new one (see Figure 12 and Figure 19). Several previous systems take advantage of silhouettes to specify the overall shape of a tree, and these inspired our user interface [BPF*03][PJM94][PMKL01][WP95]. The system changes the shape of the trunk and adjusts the length of each segment to fit into the silhouette, while other properties, such as the branching structure or the number of segments, remain fixed. This operation is useful for generating multiple trees that have the same structure, but different appearances.

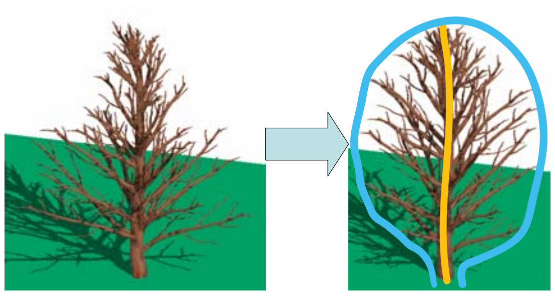


Figure 12: *The original tree (left) and a tree reproduced by sketching the trunk (orange) and the overall shape (blue)(right).*

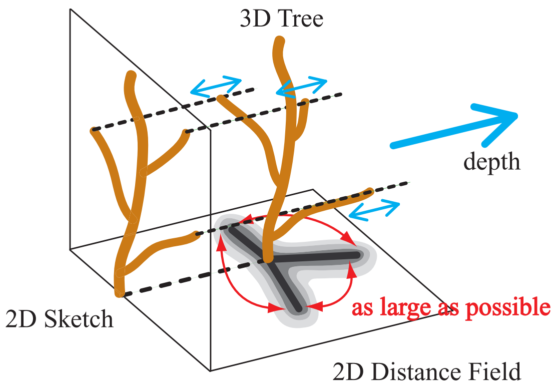


Figure 13: *Computing the depth information for branches with the 2D distance field.*

4. Algorithm

This section provides a detailed explanation of the algorithms used to compute 3D tree geometries based on sketches and examples. We focus on only the most relevant aspects, omitting the explanation of relatively straightforward processes because of space limitations.

4.1. Creating a 3D Tree from a 2D Sketch

The task here is to compute the depth information for the branches. The goal is to make the tree look similar when viewed from all angles, preserving the appearance from the original viewpoint. Our basic strategy is to adjust the orientation of a branch so that the distances between it and other branches are as large as possible (Figure 13). We use a greedy approach (adding branches one by one) instead of computing a globally optimal branch placement. The order of processing sibling branches is random. Ideally, we would construct a 3D volume whose voxels contain the distance to the nearest branch and then place the branch whose voxels have maximum distance values. Unfortunately, it is too computationally expensive to update the volumetric distance field for each added branch. Therefore, we project all branches to the ground and construct a 2D distance field, computing distances from the projected branches to each pixel (Figure 13, bottom). Our current implementation, which uses a 128×128 distance field, makes an exhaustive search for the optimal placement.

The projection of 3D branches onto the screen must fit the 2D sketch. This means that if we extended a branch in the direction almost perpendicular to the screen, the distance field alone could not prevent branches from protruding excessively from the overall silhouette (the maximum distance value is a branch of infinite length). To prevent this, we restrict the search to within a 3D hull constructed from a 2D convex hull around the given sketch. To construct the 3D hull, we simply sweep a circle along the 2D convex hull, starting from the bottom and ending at the top, and enlarge it by a constant value ($\sqrt{2}$) (Figure 14). Magnification is necessary in order to give the branches that touch the hull the freedom to move away from the original plane.

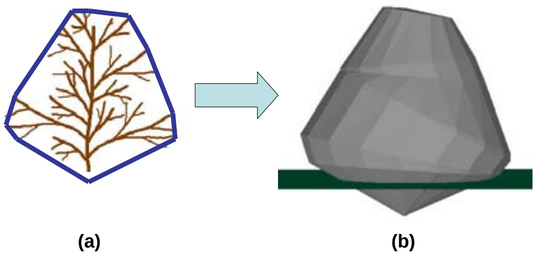


Figure 14: *A 2D convex hull of the original sketch (a). The resulting 3D convex hull after magnification (b).*

Our algorithm also constrains the lengths of branches, because a branch is generally shorter than its parent branch. We use the formula introduced in Weber and Penn [WP95] to calculate the length of a child branch. To relax the constraint, we magnify the calculated value using a constant (the current implementation uses 1.2). We use this constraint over branches other than the trunk and its child branches. Our algorithm further constrains the angles between a parent branch and its child branches. We first calculate the maximum angle between a parent branch and its child branches in the 2D sketch. Then, we compute a limit angle by multiplying the maximum angle times a constant (we use 1.2). Finally, we constrain the angles between the parent branch and its children to be smaller than the limit angle. During this process, for simplicity and efficiency we treat each branch as a straight-line segment that connects the base and terminal points. After constructing a 3D tree consisting of straight branches, we assign depth modulation to the curves shown in the original 2D sketch. For depth modulation, we adopt the algorithm described in [IOOI05], which was proposed to add appropriate depth to a user-drawn 2D stroke in an inflorescence editor.

4.2. Extension of the Basic Algorithm

People tend to draw branches that extend sideways and omit branches extending toward or away from the screen. However, our basic algorithm tries to spread branches in all directions uniformly, and the resulting 3D tree looks very different when viewed from the side (Figure 15).

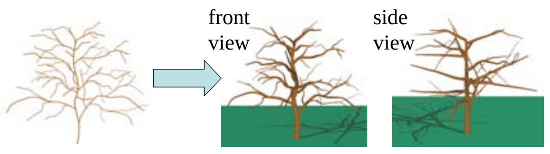


Figure 15: *The resulting tree is strange when viewed from the side.*

To solve this problem, we assume that the user draws only branches that extend sideways, and we make the system automatically add branches that extend toward and away from the screen. We do this by constraining the direction of a branch within specific angles relative to the viewing direction (we use angles between 45 and 135 degrees). The system constructs two 3D trees using the algorithm described above, rotates one of them 90 degrees around the vertical vector, and merges it with the original tree, except for the main trunk (Figure 16). The two trees are slightly different because our algorithm spreads sibling branches in a random order. This simple ad hoc trick works very well and is an indispensable tool in the system. The merged tree looks similar when viewed from both the front and the side (Figure 17).

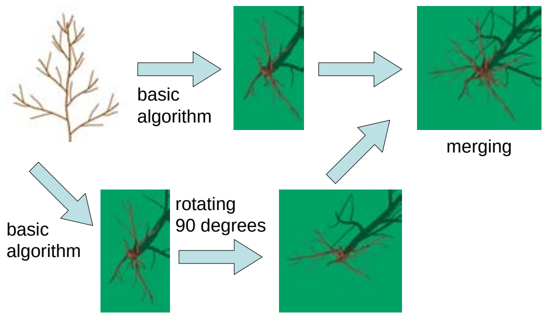


Figure 16: *The system adds branches to the front and the back. Snapshots, except for the first sketch, are seen from the top.*

4.3. Example-based Branch Multiplication

This operation adds a new branch to a given parent branch using the existing child branches as examples. To add a

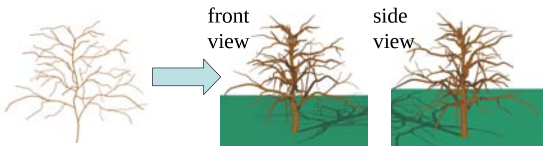


Figure 17: *The merged tree looks similar when viewed from both the front and the side.*

branch, the system must determine its position, length, orientation, and shape. Position, length, and shape are simple: the system places the new branch between the most separated pair of neighboring branches. The length of the branch is the average of the neighboring sibling branches. As for the shape, we randomly choose one of the sibling branches on the parent branch and copy it.

Orientation is a bit more complicated. The system first assigns consistent local coordinate systems along the parent branch using the "turtle" of L-systems [PHHM96]. The orientation of a child branch is determined by two angles in this local coordinate system: a "rotation angle" defined in the plane perpendicular to the parent branch, and a "down angle", which is the angle between the parent branch and the child branch (Figure 18).

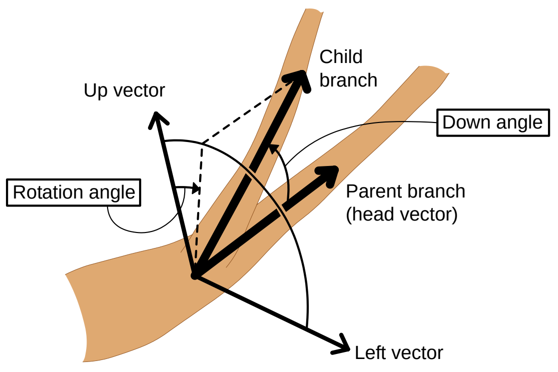


Figure 18: *A direction vector of a new child branch consisting of a rotation angle and a down angle.*

The rotation angle is calculated so that the new child branch spreads uniformly when seen along the head vector. Since a natural branch tends not to grow downward because of tropisms, we mimic the effect by calculating the rotation angle so that the child branch does not make an angle larger than 120 degrees with the upward vector perpendicular to the ground. A down angle is calculated as the average of the down angles of the neighboring sibling branches, as is the case with length.

4.4. Reproduce a Tree from Overall Sketching

The user-guided duplication algorithm is as follows:

- Generate a new trunk that follows the user-drawn trunk stroke.
- The user-drawn silhouette stroke is converted into a 3D convex hull, using the algorithm described in Section 4.1.
- The length of the first-generation branches is adjusted so that they touch the 3D convex hull. The ratio is stored in each first-generation branch.
- Modify the length of second-generation or younger branches using the ratio stored in the ancestral first-generation branch. If the resulting length is longer than the 3D convex hull, adjust the length so that it touches the hull.

Since first-generation branches tend to decide the overall shape of a tree [WP95], they are processed differently from subsequent-generation branches.

5. Results

Figure 1 and Figure 20 show 3D tree models designed by the authors. In Figure 1, the young red tree consists of 7,900 nodes (branches and leaves), the zelkova consists of 30,000 nodes, and the maidenhair tree consists of 4,900 nodes. We spent less than 10 minutes on average to design each of these models. We created broadleaf trees mainly from a 2D sketch and by propagating branches and leaves. The automatic multiplication was useful for designing branches of an acicular tree.

We performed a user study to test the usability of our prototype system. The subjects were seven students in the Computer Science department who were novice users of our system. After having them read a tutorial and learn how to use our system, we asked them to create as many 3D trees as they liked. Most subjects spent approximately 1 hour on the study, while one subject was fascinated by the system and spent a day playing with it. Figure 21 shows tree models designed by four users and the time to complete each tree model. Some of these model trees are not natural in appearance, but they are what the users wanted. These unique trees might be difficult to design using rule-based systems or by modifying predefined tree models.

We also performed another informal study to compare our system with existing methods. We used cpfg [PMKL01] as an example of a text-based system and Xfrog as an example of a graphical system. We recruited three novice users to join the study and asked them to create 3D trees like the target shown in Figure 22 (a). Figure 22 shows the resulting tree models created using cpfg (b), Xfrog (c), and our system (d and e). Two test users worked together for approximately 60 minutes to create the cpfg model. Another test user spent 30 minutes to create the Xfrog model. Finally, each test user worked individually using our system and took 10 minutes to create their models. These results show that our system is good at reconstructing the major branching structures of a tree, while the other systems are good at reconstructing detailed structures. We would like to combine these two complementary approaches in the future.

6. Limitations and Future Work

Our system allows the user to design various interesting tree-like shapes quickly. This rapid construction is possible because we ignore some natural principles. As a result, the final models sometimes exhibit artifacts not seen in conventional tree-modeling systems. In the future, we plan to explore methods to fill the gap between speedy systems and systems that adhere to the processes of nature. For example, the current implementation does not handle tropisms explicitly. We plan to estimate tropisms from user-defined branches and apply them to system-created branches. We have designed our prototype so that it uses linear workflow, except for undo or redo. To make the system usable for practical applications, it should be able to edit trees. In the future, we plan to develop more sophisticated 3D interactions in our prototype system, which would allow the user to move, rotate, or bend individual or groups of branches on a completed tree without destroying leaves.

Our user interfaces can be applied to 3D plants other than trees by implementing some additional interfaces. For example, we plan to implement sketch-based 3D interfaces for generating more complex leaf and flower models. We also plan to develop more leaf-arrangement engines, to handle other leaf patterns.

The current system is designed to construct a single tree. We are also interested in the construction of similar, but slightly different, trees to create a forest, and are experimenting with algorithms that generate slightly different trees by tweaking the parameters used in the construction process.

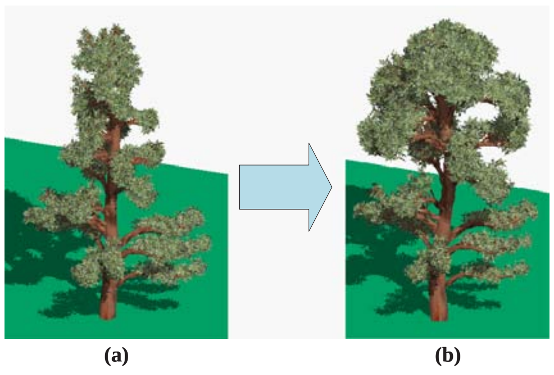


Figure 19: *An example of reproduction of a tree. (a) an original tree, (b) the reproduced version.*

References

[AK84] AONO M., KUNII T. L.: Botanical tree image generation. *IEEE Computer Graphics and Applications 4*, 5 (1984), 10–34. 2

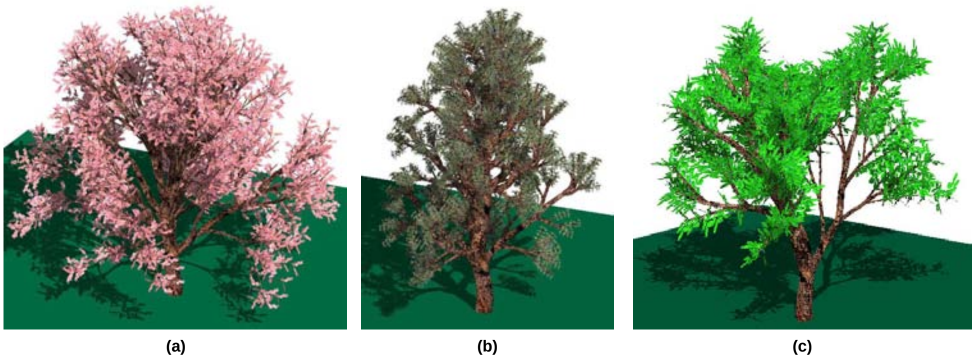


Figure 20: *3D trees designed by one of the authors. (a) Cherry, (b) Pine, (c) Oriental Plane.*

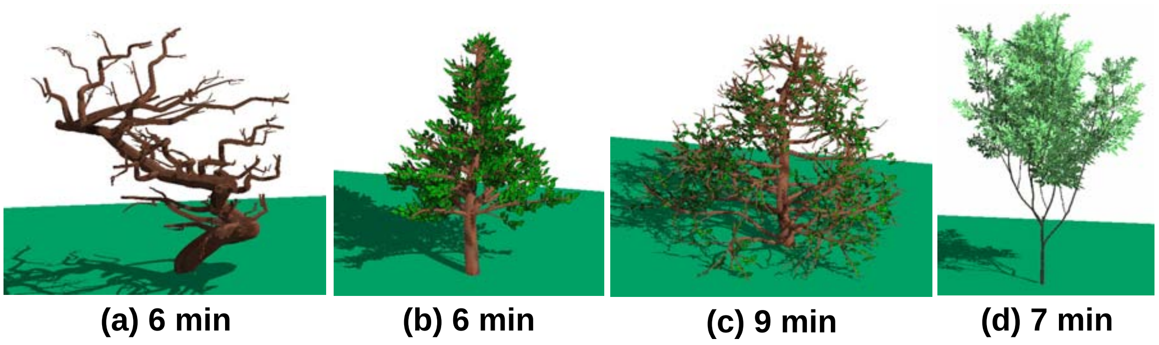


Figure 21: *3D tree models designed by the test users and the time to complete each tree model.*

[BPF*03] BOUDON F., PRUSINKIEWICZ P., FEDERL P., GODIN C., KARWOWSKI R.: Interactive design of bonsai tree models. *Comput. Graph. Forum 22*, 3 (2003), 591–600. 2, 5

[Cyp91] CYPHER A.: Eager: Programming repetitive tasks by example. In *Proceedings of CHI '91* (New Orleans, LA, 1991), pp. 33–39. 1, 3

[Cyp93] CYPHER A.: *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press, 1993. 3

[DL97] DEUSSEN O., LINTERMANN B.: A modelling method and user interface for creating plants. In *Graphics Interface* (May 1997), Davis W., Mantei M.„ Klassen V., (Eds.), pp. 189–197. 2

[DS00] DEUSSEN O., STROTHOTTE T.: Computer-generated pen-and-ink illustration of trees. In *Proceedings of SIGGRAPH 2000* (2000), ACM Press, pp. 13–18. 2

[EyHBE97] EGGLI L., YAO HSU C., BRÜDERLIN B. D., ELBER G.: Inferring 3d models from freehand sketches and constraints. *Computer-Aided Design 29*, 2 (1997), 101–112. 3

[IH01] IGARASHI T., HUGHES J. F.: A suggestive interface for 3d drawing. In *Proceedings of UIST 2001* (2001), ACM Press, pp. 173–181. 3, 4

[IKTM98] IGARASHI T., KAWACHIYA S., TANAKA H., MATSUOKA S.: Pegasus: a drawing system for rapid geometric design. In *Proceedings of CHI '98* (1998), ACM Press, pp. 24–25. 3

[IMKT97] IGARASHI T., MATSUOKA S., KAWACHIYA S., TANAKA H.: Interactive beautification: a technique for rapid geometric design. In *Proceedings of UIST '97* (1997), ACM Press, pp. 105–114. 3

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: a sketching interface for 3d freeform design. In *Proceedings of SIGGRAPH '99* (1999), ACM Press, pp. 409–416. 1, 3

[IOOI05] IJIRI T., OWADA S., OKABE M., IGARASHI T.: Floral diagrams and inflorescences: Interactive flower modeling using botanical structural constraints. In *Proceedings of SIGGRAPH 2005* (2005), ACM Press. 2, 6

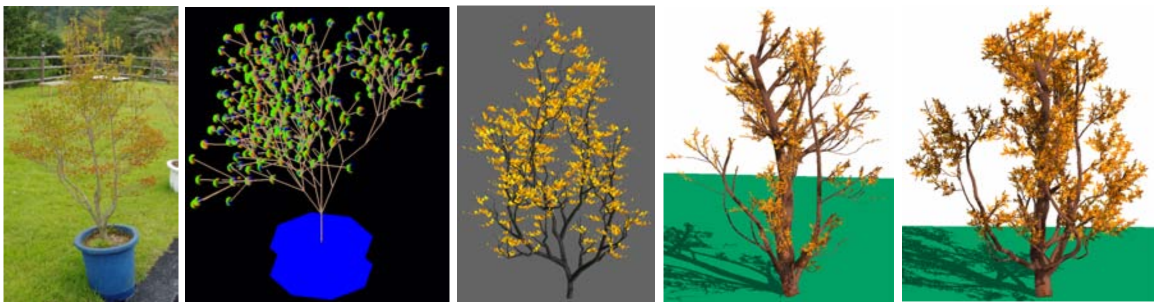


Figure 22: *3D tree models designed using the different systems by the test users. (a) The target tree, (b) the result of cpfg by two users, (c) the result of XFrog and (d, e) the results of our system.*

[LD99] LINTERMANN B., DEUSSEN O.: Interactive modeling of plants. *IEEE Comput. Graph. Appl. 19*, 1 (1999), 56–65. 2

[Lin68] LINDENMAYER A.: Mathematical models for cellular interactions in development. *Journal of Theoretical Biology 18* (1968), 280–315. 2

[MP96] MĚCH R., PRUSINKIEWICZ P.: Visual models of plants interacting with their environment. In *Proceedings of SIGGRAPH '96* (1996), ACM Press, pp. 397–410. 2

[MT02] MAIERHOFER S., TOBLER R. F.: Creation of realistic plants using semi-automatic parametric extraction from photographs. *Technical report 2002-002, VRVis Research Center* (2002). 2

[MWK89] MAULSBY D. L., WITTEN I. H., KITTLITZ K. A.: Metamouse: specifying graphical procedures by example. In *Proceedings of SIGGRAPH '89* (1989), ACM Press, pp. 127–136. 1, 3

[ONOI04] OWADA S., NIELSEN F., OKABE M., IGARASHI T.: Volumetric illustration: designing 3d models with internal textures. *ACM Trans. Graph. 23*, 3 (2004), 322–328. 3

[PHHM96] PRUSINKIEWICZ P., HAMMEL M., HANAN J., MĚCH R.: L-systems: from the theory to visual models of plants. *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences.* (1996). 2, 7

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *Proceedings of SIGGRAPH '94* (1994), ACM Press, pp. 351–358. 2, 5

[PK89] PENTLAND A., KUO J.: The artist at the interface. *Vision Science Technical Report 114.* (1989). 3

[PMKL01] PRUSINKIEWICZ P., MÜNDERMANN L., KARWOWSKI R., LANE B.: The use of positional information in the modeling of plants. In *Proceedings of SIGGRAPH 2001* (2001), ACM Press, pp. 289–300. 2, 5, 8

[RMMD04] RECHE-MARTINEZ A., MARTIN I., DRETTAKIS G.: Volumetric reconstruction and interactive rendering of trees from photographs. *ACM Trans. Graph. 23*, 3 (2004), 720–727. 2

[SL96a] SHPITALNI M., LIPSON H.: Identification of faces in a 2d line drawing projection of a wireframe object. *IEEE Trans. Pattern Anal. Mach. Intell. 18*, 10 (1996), 1000–1012. 3

[SL96b] SHPITALNI M., LIPSON H.: Optimization-based reconstruction of a 3d object from a single freehand line drawing. *Computer-Aided Design 28*, 8 (1996), 651–663. 3

[SO99] SAKAGUCHI T., OHYA J.: Modeling and animation of botanical trees for interactive virtual environments. In *Proceedings of VRST '99* (1999), ACM Press, pp. 139–146. 2

[SRDT01] SHLYAKHTER I., ROZENOER M., DORSEY J., TELLER S.: Reconstructing 3d tree models from instrumented photographs. *IEEE Comput. Graph. Appl. 21*, 3 (2001), 53–61. 2

[TNT89] TANAKA T., NAITO S., TAKAHASHI T.: Generalized symmetry and its application to 3d shape generation. *The Visual Computer 5*, 1&2 (1989), 83–94. 3

[WP95] WEBER J., PENN J.: Creation and rendering of realistic trees. In *Proceedings of SIGGRAPH '95* (1995), ACM Press, pp. 119–128. 2, 5, 6, 8

[ZHH96] ZELEZNIK R. C., HERNDON K. P., HUGHES J. F.: Sketch: an interface for sketching 3d scenes. In *Proceedings of SIGGRAPH '96* (1996), ACM Press, pp. 163–170. 1, 3

Floral diagrams and inflorescences: Interactive flower modeling using botanical structural constraints

Takashi Ijiri† Shigeru Owada‡ Makoto Okabe† Takeo Igarashi†§
†The University of Tokyo ‡Sony CS Laboratories Inc. §PRESTO/JST

Abstract

We present a system for modeling flowers in three dimensions quickly and easily while preserving correct botanical structures. We use *floral diagrams* and *inflorescences*, which were developed by botanists to concisely describe structural information of flowers. Floral diagrams represent the layout of floral components on a single flower, while inflorescences are arrangements of multiple flowers. Based on these notions, we created a simple user interface that is specially tailored to flower editing, while retaining a maximum variety of generable models. We also provide sketching interfaces to define the geometries of floral components. Separation of structural editing and editing of geometry makes the authoring process more flexible and efficient. We found that even novice users could easily design various flower models using our technique. Our system is an example of application-customized sketching, illustrating the potential power of a sketching interface that is carefully designed for a specific application.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction Techniques.

Keywords: 3D Modeling, floral diagram, flower, inflorescence, sketch-based modeling.

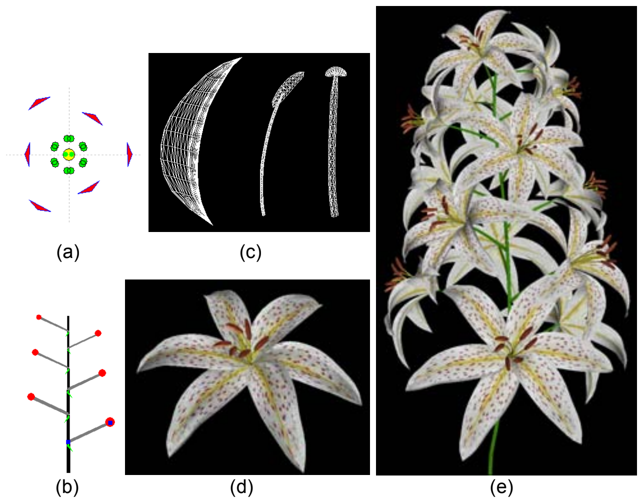


Figure 1: Lily model. The structural information is given as a floral diagram (a) and an inflorescence (b). The floral diagram consists of one pistil, six stamens, and six petals. The inflorescence pattern is raceme. The geometry models are designed in the sketch-based editor (c). The user creates a flower (d) and the entire model (e) of a lily combining the structural information and the geometries.

1 Introduction

Flowers pose an interesting and important challenge for three-dimensional (3D) computer graphics modeling. They have a great number of components, such as petals, stems, and pistils, which take on highly varied 3D shapes and which are connected with intricate structures. To create a flower, users must design each component as a freeform surface and lay them all out in 3D space. The geometric and structural complexity makes this a difficult and time-consuming task even for experienced users; for novice users, creation of beautiful and biologically plausible flowers using traditional tools is almost impossible.

Various botanic modeling systems have been created to support the design of plants. These can be classified into two groups according to their purposes. The first group concentrates mainly on visual plausibility rather than botanical correctness [Deussen and Lintermann 1999]. This type of modeler tends to offer a simple user interface, but its underlying method is to use a predefined library, and it is therefore difficult to design models that are not in the library. The second group tries to build a theoretical framework based on biological knowledge. For example, the L-System, one of the best known plant modeling systems, defines plant structures using a set of rewriting rules [Prusinkiewicz and Lindenmayer 1990]. However, it is very difficult to encode and decipher the behavior of real-world plants in such a simple form, and users must also have specific biological knowledge about plants. Furthermore, while an L-system encodes various characteristics of the gross structure of a plant, the actual geometry of the individual components; leaves, petals, stems, etc. remains to be determined by the user.

Our goal is to strike a balance between these two approaches to modeling, that is, to provide an easy-to-use interface, while allowing users to model a wide variety of biologically plausible flower models. When guiding the modeling process, we incorporate *floral diagrams* and *inflorescences* as general and compact frameworks to describe most real-world flowers. A floral diagram is an iconic description of a flower's structural characteristics (Figure 1a); we use it to design individual flowers. An inflorescence is a branch with multiple flowers and its branching pattern is represented in a pictorial form; we use it to design models that consist of many flowers, such as bostryx, lavender, and lilies (Figure 1b). These two frameworks define the structure of a flower model; it is also necessary to specify the geometry of each component, such as the floral receptacle, pistil, stamen, petal, and sepal. To make geometric modeling intuitive and efficient, we use a customized freeform sketching interface.

This paper describes the user interface of our prototype flower modeling system based on these ideas. The structure editor consists of two sub-systems: one is for individual flowers, driven by floral diagrams, the other for arrangements of multiple flowers, based on inflorescences. The geometry editor also has two sub-systems: one for floral elements, the other for inflorescences. We believe that this separation of structure editing from geometry editing is applicable to general modelers, simplifies the modeling process, and achieves high configurability and reusability. Without this separation, it is very difficult to change a basic structure after details have been completed. Using our system, once a whole model has been created, it is possible to apply the model to different geometry to create a new model with the same or a similar structure.

Note that our contribution is in simplifying the process of flower modeling, not in improving the final results. The resulting flower models can be replicated by existing modeling systems, but the process is different. With customized and well-designed high-level editors for particular classes of objects, the modeling process becomes much more intuitive and efficient.

We describe the user interface of these editors in the following sections after discussing related work and the basic background. Our results show that users can design interesting flower models, such as the one shown in Figure 1, with little training. A user spent only 30 minutes creating this model from scratch.

2 Related Work

Lindenmayer [1968] formulated the L-System and Prusinkiewicz and Lindenmayer [1990] later introduced it to the computer graphics community. The L-System has been extended to simulate a wide variety of interactions between plants and their environments [Měch and Prusinkiewicz 1996; Prusinkiewicz et al. 1994; 1996]. Prusinkiewicz et al. [2001] also proposed using positional information to control parameters along a plant axis. Boudon et al. [2003] proposed an L-system-based process for designing Bonsai tree models; it uses decomposition graphs to make it easier to manipulate various parameters.

Deussen and Lintermann [1997; 1999; Lintermann and Deussen 1996] developed the Xfrog system, which combines the power of a rule-based approach and intuitive user interfaces using a graph representation. Users design a graph representing the branching structures of a plant with 11 node types. This system offers an intuitive user interface and the resulting models are highly realistic, but the graph is designed heuristically and is too general for flower modeling (i.e., the graph can create structures other than plants). Furthermore, the graph representation includes geometric components such as FFD, so it is not possible to separate structural and geometric definitions completely.

Over the past decade, sketch-based modeling has become popular; instead of creating precise, large-scale objects, a sketching interface provides an easy way to create a rough model that quickly conveys a user's intentions. The main focus is on inferring 3D shapes from two-dimensional (2D) sketches. Previous work has reconstructed rectilinear models covered by planar faces by solving constraints [Pugh 1992; Eggli et al. 1997] or by using optimization-based algorithms [Lipson and Shpitalni 1996]. The SKETCH system [Zeleznik et al. 1996] allows users to design 3D scenes consisting of simple primitives, while the Teddy system allows users to design freeform models [Igarashi et al. 1999]. Generating 3D curves through sketching is also a rich research domain; Pentland and Kuo [1989] generated a 3D curve from its 2D projection using energy minimization, while Tanaka et al. [1989] used symmetric relations. Another strategy for defining a 3D curve is to draw strokes twice, for example, a screen projection of a curve and its shadow [Cohen et al. 1999; Tobita and Rekimoto 2003].

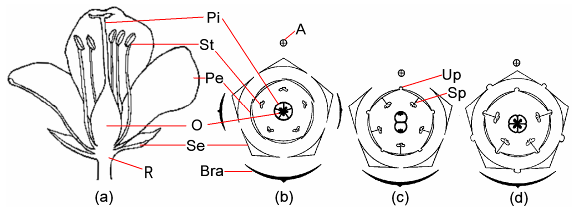


Figure 2: Examples of floral diagrams. A: axis, Bra: bract, O: ovary, Pe: petal, Se: sepal, St: stamen, Sp: sepal adnate to stamen, R: floral receptacle, Up: petal connate to petal.

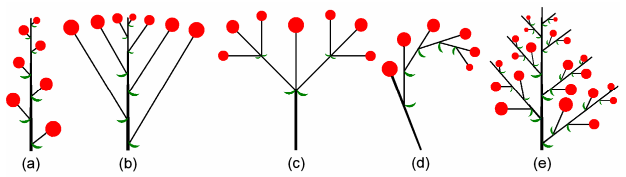


Figure 3: Examples of inflorescence patterns. The two on the left are indeterminate inflorescences: *raceme*(a) and *corymb*(b). The next two are determinate inflorescences: *dichasium*(c) and *drepanium*(d). The last is a compound inflorescence: *compound-raceme*(e).

2.1 Floral Diagrams and Inflorescences

Floral diagrams and *inflorescences* are technical representations used in the study of plant morphology, which uses plant structure to explore their evolution, ecology, and systematics [Hara 1994; Shimizu 2001; Bell 1991].

A *floral diagram* pictorially represents the layout of four kinds of floral elements on a receptacle (the base of a flower): pistils, stamens, petals, and sepals (Figure 2). A floral diagram also describes additional information, such as the stem cross-section, number of ovules, and whether petals are connate. However, it does not describe the 3D geometry of floral components or their relative sizes. There is no universal definition of a floral diagram, and various forms of floral diagram exist.

An *inflorescence* represents a branch bearing multiple flowers. In an inflorescence, flowers are generally arranged in one of a fixed number of patterns specific to their species. There are three inflorescence groups: indeterminate, determinate, and compound. In indeterminate inflorescences, lower flowers bloom first and higher flowers follow. In determinate inflorescences, top or central flowers bloom first and lower or lateral flowers follow. Compound inflorescences are a mixture of the other two patterns. Simple 2D figures can be used to represent all branching patterns (Figure 3). Here, black lines represent the central axis and its branches, red circles represent flowers, and green crescents represent bracts. Larger circles indicate older flowers.

3 Overview of the Modeling Process

Our system consists of a set of independent editors, which can be basically categorized into two groups: structure editors and geometry editors. The structure editor consists of a floral diagram editor and an inflorescence editor. Users can alternate between these two editors. A typical scenario is as follows (Figure 4).

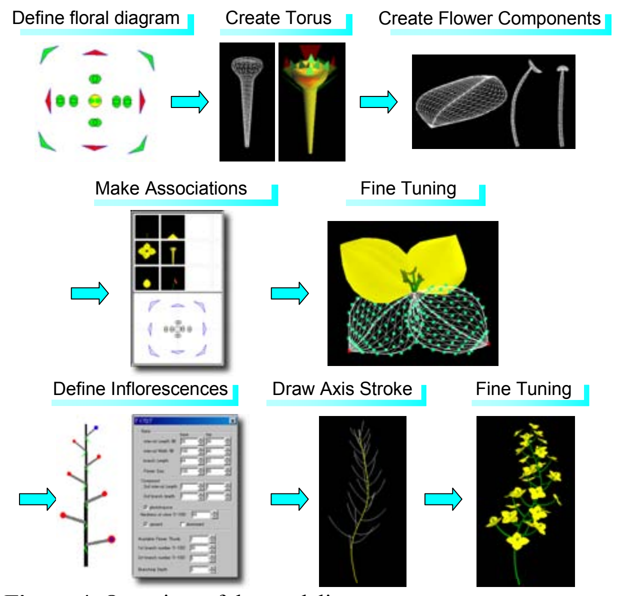


Figure 4: Overview of the modeling process

The user first defines the flower's structure in the floral diagram editor by editing the layout of the floral components. The user then models the shapes of the floral receptacle and floral components using the sketching interface in the geometry editor. The resulting receptacle model appears at the bottom of the floral diagram editor and the component thumbnails are listed on the right side of the window. Next, the user associates geometries of floral components with corresponding elements in the floral diagram using drag-and-drop operations. The system automatically places geometric objects on the receptacle model. The user can interactively adjust the angle of attachment, size, and shape of the components in the geometry editor. The user can also modify layout using the floral diagram editor.

After designing individual flowers, the user models the inflorescence. The user first defines the structure in the inflorescence editor, choosing one pre-defined inflorescence pattern from the list and making basic adjustments to various parameters. Then the user defines the central axis geometry by drawing a freeform stroke in the geometry editor. The system creates a three-dimensional inflorescence along the axis. The user adjusts the angles of flower and branch attachment using the geometry editor and can adjust parameters such as branching angle, branch length, etc. using the inflorescence editor.

4 Structure Editors

4.1 Floral Diagram Editor

A standard floral diagram represents not only the structure of a flower but also some geometric information. However, our floral diagram editor focuses on the layout of floral components, and geometries are modeled separately in the geometry editor. Floral components (pistil, stamen, petal, and sepal) are represented as icons (Figure 5b). Users first specify the number of parts by typing the number, then specify layout by dragging and moving icons in the diagram.

Floral components are often arranged in radial symmetry, so our editor provides a function to arrange them in radial symmetry.

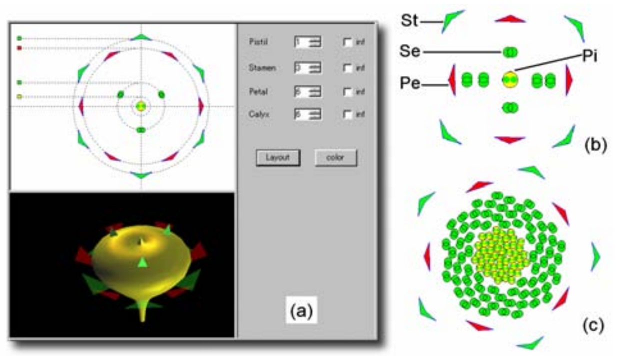


Figure 5: A snapshot of the floral diagram editor (a) and examples of floral diagrams: Brassica rapa (b) and Ranunculus acris (c). Pi: pistil, St: stamen, Pe: petal, Se: sepal.

There are four circular regions in the diagram editor and users can modify their size by dragging borders. If users press the "layout" button, the system distributes the parts uniformly in each region. Some species (*e.g.*, Ranunculus acris) have an indefinite number of components. In this case, a specific region of the flower is filled by as many corresponding components as possible. In our system, if users check the "indefinite" box, the corresponding region is filled by as many icons as possible (Figure 5c). We use a filling algorithm introduced by Prusinkiewicz et al. [2001].

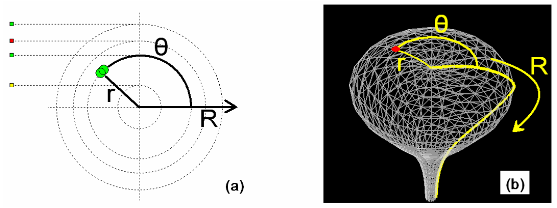


Figure 6: Mapping the 2D diagram onto the 3D receptacle. A stamen object on a floral diagram (a) and its position in 3D (b).

A floral diagram is a 2D representation of a layout, so in order to construct the final 3D flower, the system has to convert the 2D layout into a 3D composition of geometric objects. A floral receptacle is represented as a surface of revolution, the outline of which is drawn by the user. The system uses a polar coordinate system on this surface, shown in Figure 6b. In our implementation, the receptacle's 3D view is located underneath the floral diagram view (Figure 5a). A change using the floral diagram editor is immediately reflected in the 3D view. We currently do not allow users to use the 3D view to directly manipulate the layout; this remains for future work.

4.2 Inflorescence Editor

In the inflorescence editor, users select a branching pattern from the list and modify parameters by dragging handles in the visual pattern display (Figures 7b, c). We have implemented 8 of 22 patterns reported in the literature [Bell 1991]. The variety of adjustable parameters depends on the pattern selected. Figure 13 shows all patterns and their parameters. Using a raceme as an example, branch angle, branch length, and flower size at the top and bottom of the axis can be modified using the handles (Figure 7c). Values between the top and bottom are linearly interpolated. Parameters such as the existence of tropism or stem hardness are specified in dialog boxes, since these parameters are difficult to represent in a 2D illustration. In future research, we plan to allow for more flexible positional control [Prusinkiewicz et al. 2001].

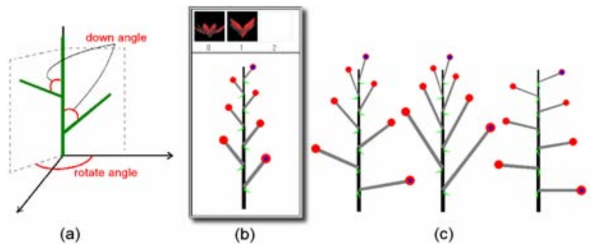


Figure 7: (a) Down angle and Rotate angle. (b) Inflorescence editor. (c) Inflorescence pattern of a raceme with various parameters.

To determine each branch's 3D direction, the system must compute branch angle to the stem; we call this the rotation angle (Figure 7a). In certain inflorescences, branches have one rotation angle value, which can be described as follows:

$$angle = \frac{F_n}{F_{n+2}} \times 360 \qquad n = 0, 1, 2, 3 \cdots$$

$$F_n : fibonacci\ sequence$$

This formula produces the following values: 180, 120, 144, 135, 138.45, 137.14, and 137.65, covering almost all species [Bell 1991]. These values are listed, and users can simply choose the desired value. Users can also specify an arbitrary angle when necessary.

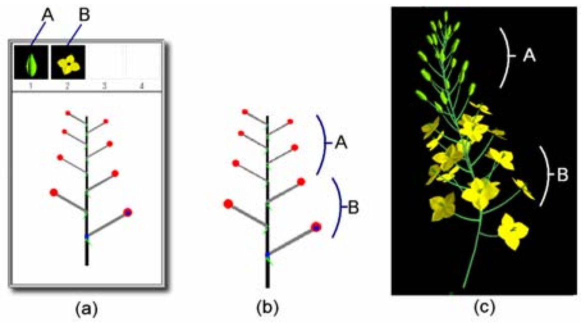


Figure 8: (a) Bud and blooming flower models (A and B) are specified. (b) (c) Buds are placed on the higher (younger) half of the branches. Blooming flowers are placed on the lower (older) half of the branches.

Users associate flower models (created in the floral diagram editor) with inflorescence branch terminals. Aging of a flower is represented simply by multiple flower models; as shown in Figure 8a, users import multiple models of different ages into the inflorescence editor top row in ascending order of age. The age is also linearly interpolated depending on the pattern (see section 2.1). For instance, when two flower models are provided for an indeterminate inflorescence pattern, the lower half is associated with the old flower model and the upper half is associated with the young flower model (Figures 8b, c).

After adjusting parameters, users add geometric information to the inflorescence in the geometry editor. If desired, users can return to the structure editor and adjust parameters. The system provides immediate visual feedback to the 3D inflorescence model during the parameter adjustment process.

There are special inflorescence patterns called *head* and *spadix*. A head is a pattern in which small flowers cover a base called a disc, *e.g.* sunflowers. A spadix is a pattern in which many flowers are densely arranged on a thick stalk, *e.g.* Lysichiton camtschatcense. These inflorescence patterns can be compactly represented in floral diagrams, so we work with them in the floral diagram editor, allowing users to arrange flowers on the receptacle as well as arranging standard floral components.

5 Geometry Editors

Flower model components are 3D freeform shapes. We use a sketch-based interface to allow quick and intuitive modeling. Sketch-based modeling systems [Zeleznik et al. 1996; Igarashi et al. 1999] allow users to design interesting 3D geometry by drawing strokes on the screen; by contrast, traditional modeling systems require users to work with menus and many control points. A key aspect of sketch-based systems is that they make strong assumptions in interpreting user input to maintain a simple user interface. Our system simplifies the interface by providing a customized modeling interface for each floral component. Traditional modeling interfaces are generally suitable for careful editing by expert users; sketching interfaces are suitable for quick exploration by novices or casual users.

5.1 Floral Receptacles and Floral Components

In the geometry editor, users can create the geometries of the floral receptacle, pistil, stamen, petal, and sepal.

A floral receptacle is defined as a surface of revolution, the profile of which is given by a user as a freeform stroke. A pistil is modeled using an inflation algorithm similar to "extrusion" in the Teddy system [Igarashi et al. 1999]. A stamen is defined as the sweep surface of a circle along a central axis drawn by the user. The user then draws another stroke to describe the axis of the stamen's *anther* and the system creates a mesh by warping an ellipsoid along this stroke.

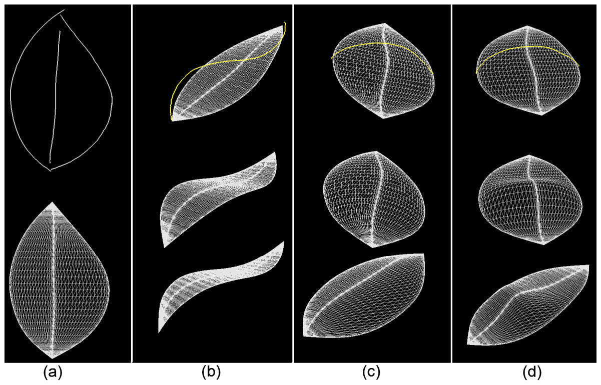


Figure 9: Petal modeling. (a) Initial creation. (b) Transforming an object along the center vein. (c) Transforming an object in global mode and (d) in local mode.

The petal and sepal share a common user interface (Figure 9). A user first draws three strokes to represent the outline and central vein of the petal (the central stroke may be omitted). The system returns a flat petal object (Figure 9a). Next, the user draws modifying strokes; these strokes are interpreted as cross-sections of the object (Figures 9b, c, d). Modifying strokes have two modes: global and local. In the global mode, a modifying stroke deforms the entire object, while in the local mode, only part of the object is deformed (Figure 9d). Users can switch between the two modes by selecting a button. To add realism, users can also add noise and texture.

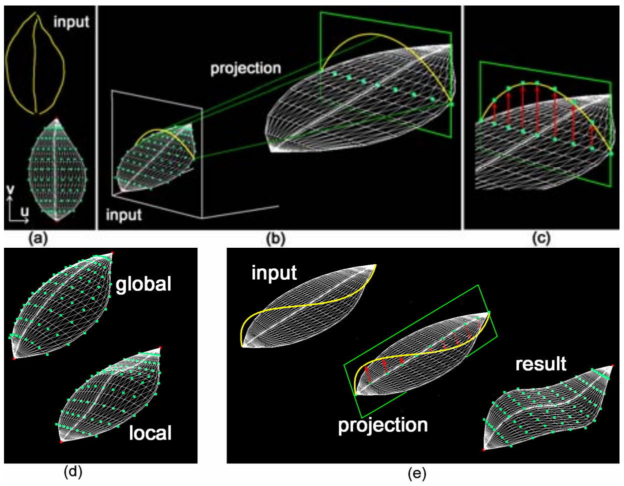


Figure 10: Petal modeling. (a) Initial creation. (b) (c) The system maps the 2D stroke. (d) Resulting geometry in global and local modes. (e) An example of a modifying stroke along the vertical direction.

A petal object is implemented as a B-spline surface. When the initial three outline strokes are drawn, the system generates control points of the B-spline surface, shown in Figure 10a. We parameterize the surface using *u* and *v* coordinates, where the *u*-axis corresponds to horizontal direction and the *v*-axis corresponds to vertical direction. The system saves the plane on which the initial surface lies as a base plane. Modifying strokes move control points perpendicular to this base plane. If a user draws a modifying stroke in the *u* direction, the system first finds the control point nearest to the stroke's starting point on the screen. Control points that have the same *v* value as the base point are marked as target control points. The system projects the stroke on a plane that passes through target control points and is perpendicular to the base plane (Figure 10b). Next, the system moves target control points to the projected stroke (Figure 10c). In the global mode, the system moves all control points on the surface, and in the local mode it moves only neighboring points (Figure 10d). The displacement amount smoothly decays toward the petal's top and bottom. When a modifying stroke is drawn in the *v* direction, the system projects the stroke to a plane containing the central axis, perpendicular to the base plane (Figure 10e). The system then moves control points so that all points with the same *v*-coordinates move the same amount. In this case, there is no difference between global and local modes.

5.2 Inflorescence

The interface for modeling the geometry of an inflorescence is very simple. After selecting an inflorescence pattern and adjusting its parameters in the structure editor (Figure 7), the user draws the selected inflorescence's central axis as a 2D freeform stroke. The system then creates the 3D geometry of the inflorescence, displaying the curves that represent the axis and branches during the drawing operation. When the user completes drawing the stroke, the system creates a mesh for the stem and places the flower objects on branch terminals (Figure 11).

Our system automatically adds appropriate depth to a user-drawn 2D stroke. Typical existing approaches first define a work plane that is almost perpendicular to the view direction and project the user-drawn stroke onto it [Cohen et al. 1999; Tobita and Rekimoto

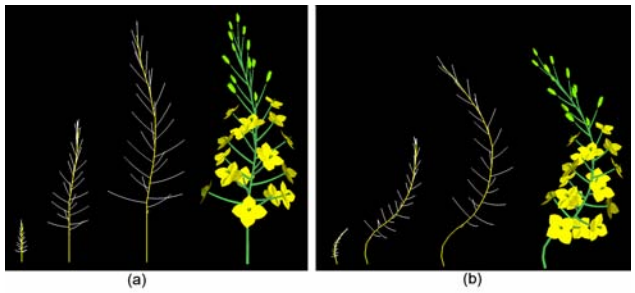


Figure 11: The geometry editor for inflorescences. The user draws the axis of the inflorescences freehand and the system provides the real time feedback during drawing.

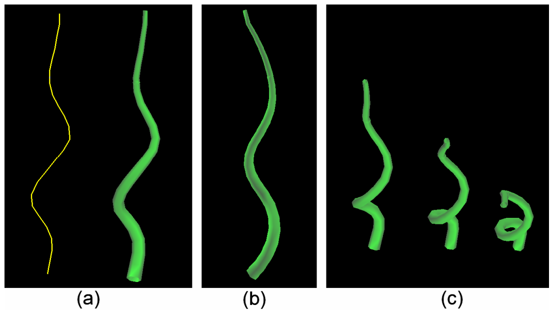


Figure 12: (a) A stroke drawn by the user and the resulting 3D geometry models. (b) The model viewed from the right side. (c) The model viewed from higher perspectives.

2003]. A drawback of this approach is that it cannot create the typical shapes of stems such as spirals, and it requires that strokes be drawn twice. Our approach requires input of a single stroke and generates a 3D curve with a similar appearance regardless of viewing direction around the axis. For example, when a user draws a sine curve, it creates a 3D spiral stroke. We achieve this effect by adding depth to the curve, so that the resulting curve has a constant curvature in 3D space (Figure 12). Our algorithm is a specialized version of the energy-minimizing curve reconstruction proposed by Pentland and Kuo [1989]. The detailed algorithm is as follows.

We assume that a user draws a stroke on the *x*-*y* plane and that the viewing direction is in the positive *z* direction. The initial stroke is represented as follows:

$$stroke = \{ v_i \mid v_i = (x_i, y_i, z_i), z_i = 0 \}$$

where the *y*-axis corresponds to the vertical direction. We resample the input stroke so that vertices are equally spaced along the *y* direction. Our algorithm receives the stroke with *x* and *y* values as input and returns a new stroke with appropriate *z* values. To achieve this, our algorithm assumes that the resulting stroke has a constant curvature in 3D space along the *y*-axis, *i.e.*:

$$\left(\frac{d^2x}{dy^2}\right)^2 + \left(\frac{d^2z}{dy^2}\right)^2 = const$$

We compute *z* values by solving this equation. We first decide the constant value by taking the maximum squared value of the second derivatives of *x* along the axis. Given the constant value and the second derivatives of *x*, we can calculate absolute values of the second derivative of *z* by solving the above equation.

Direct solution of this formula yields only absolute values. The next task is to determine the signs of the second derivatives of *z*. We assume that the second derivative of z_0 is positive and determine the signs sequentially, so that successive signs change when the first derivatives of *x* cross zero.

Given the signed second derivatives of *z*, we calculate values for *z* by integrating them twice. We set z_0 to be 0 and adjust the first derivative of z_0 (the initial branch slope in the depth direction) so that the last *z* also becomes zero.

6 Discussion

In this paper, we propose a system for efficiently modeling flowers with correct botanical structures. We introduce floral diagrams and inflorescences, which were developed by botanists to describe structural information about flowers. We also propose a specialized sketch-based geometry editor for floral elements. Our current implementation supports eight inflorescence branching patterns, shown in Figure 13. These are typical patterns selected from three inflorescence groups: indeterminate, determinate, and compound. Our results show that we can model plants successfully using these patterns, and it is probable that other branching patterns can be supported in a similar manner.

Figure 14 shows flower models designed using our system with the corresponding floral diagrams and inflorescence patterns. Since our system provides a simple, intuitive user interface for defining complex structures and geometries, it took less than 40 minutes to design these complete flower models from scratch. We also performed a preliminary user study to test the usability of our prototype system. We tutored four university students who were novice users for less than 20 minutes, and then asked them to create 3D flower models. Subjects were allowed to consult books to learn the structure of the target plants. It took less than 40 minutes for them to design the complete flower models shown in Figure 14 from scratch.

One limitation of the current system is that our inflorescence editor is not able to support the creation of a gradual progression of developmental flower stages. In addition, there are a few shapes that our geometry editor cannot create; for example, it is impossible to create petal-like shapes that do not have an elliptical outline.

The basis of our approach is the importance of separating structure editing from geometry editing. Our approach could be useful for modeling other targets with complicated structures and geometry, such as trees, insects, four-footed animals, etc.; in the future we would like to deal with these targets. Another interesting direction would be to extend our system to support entire plant structures. We are also interested in creating a flower arrangement application; this application would require a combination of biological and artistic knowledge, and would therefore be an interesting challenge.

We consider this work to be an example of an application-customized sketch-based interface; the success of the interface depends in part on balancing correct choice in expressive interface components against application needs: too-general components may allow users to make mistakes easily; too-limited ones may restrict user ability to reach goals, and may require a greater variety of components, which will be difficult to learn. The proper design rules for making such choices have yet to be elucidated; we hope that our system provides an instance from which such rules may someday be drawn.

Acknowledgements

We would like to thank Prof. John F. Hughes, Prof. Etsuya Shibayama, Dr. Shin Takahashi, Prof. Katsuhiko Kakei, and Prof. Ikuo Takeuchi for their comments and advice based on their experience and deep insights. This work was funded in part by grants from the Japanese Information-Technology Promotion Agency (IPA).

References

BELL, A. D. 1991. *Plant Form: An Illustrated Guide to Flowering Plant Morphology*. Oxford University Press.

BOUDON, F., PRUSINKIEWICZ, P., FEDERL, P., GODIN, C., and KARWOWSKI, R. 2003. Interactive Design of Bonsai Tree Models. In *Proceedings of Eurographics 2003*: Computer Graphics Forum, 22, 3, 591-599.

COHEN, J., MARKOSIAN, L., ZELEZNIK, R., HUGHES, J., and BARZEL, R. 1999. An Interface for Sketching 3D Curves. In *Proceedings of ACM I3D 99*, 17-21.

DEUSSEN, O. and LINTERMANN, B. 1997. A Modeling Method and User Interface for Creating Plants. In *Proceedings of Graphics Interface 97*, 189-197.

DEUSSEN O. and LINTERMANN, B. 1999. Interactive Modeling of Plants. *IEEE Computer Graphics and Applications*, 19, 1, 56-65.

EGGLI, L., HSU, C., ELBER, G., and BRUDERLIN, B. 1997. Inferring 3D Models from Freehand Sketches and Constraints. *Computer-Aided Design*, *29*, 2, 101-112.

HARA, N., 1994. *Syokubutu Keitaigaku* (Plant Morphology). Sasakura Shoten 1994 (In Japanese).

IGARASHI, T., MATSUOKA, S., and TANAKA, H. 1999. Teddy: A Sketching Interface for 3D Freeform Design. In *Proceedings of ACM SIGGRAPH 99*, ACM, 409-416.

LINDENMAYER, A. 1968. Mathematical Models for Cellular Interactions in Development, I & II. *Journal of Theoretical Biology*, 280-315.

LINTERMANN, B. and DEUSSEN, O. 1996. Interactive Modelling and Animation of Branching Botanical Structures. In *Proceedings of Eurographics Workshop on Computer Animation and Simulation 96*, 139-151.

LIPSON, H. and SHPITALNI, M. 1996. Identification of Faces in a 2D Line Drawing Projection of a Wireframe Object. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18, 10, 1000-1012.

MĚCH, R. and PRUSINKIEWICZ, P. 1996. Visual Models of Plants Interacting with Their Environment. In *Proceedings of ACM SIGGRAPH 96*, ACM, 397-410.

PENTLAND, A. and KUO, J. 1989. The Artist at the Interface. *Vision and Modeling Technical Report 114*, MIT Media Lab.

PRUSINKIEWICZ, P., and LINDENMAYER, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

PRUSINKIEWICZ, P., HAMMEL, M., HANAN, J., and MĚCH, R. 1996. L-systems: From the Theory to Visual Models of Plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences.*

PRUSINKIEWICZ, P., JAMES, M., and MĚCH, R. 1994. Synthetic Topiary. In *Proceedings of ACM SIGGRAPH 94*, ACM, 351-358.

PRUSINKIEWICZ, P., MÜNDERMANN L., KARWOWSKI, R., and LANE, B. 2001. The Use of Positional Information in the Modeling of Plants. In *Proceedings of ACM SIGGRAPH 2001*, ACM, 289-300.

PUGH, D. 1992 Designing Solid Objects Using Interactive Sketch Interpretation, *Computer Graphics,* 25, 2, 117-126.

SHIMIZU, T., 2001. *Syokubutu Yougo Jiten* (Dictionary of Botanical Terms). Yasaka Shobou (in Japanese).

TANAKA, T., NAITO, S., and TAKAHASHI, T. 1989. Generalized Symmetry and its Application to 3D Shape Generation. *Visual Computer*, 5, 83-94.

TOBITA, H., and REKIMOTO, J. 2003. Flat3D: A Shared Virtual 3D World System for Creative Activities and Communication, *IPSJ JOURNAL*, 44, 2, IPSJ, 245-255 (in Japanese).

ZELEZNIK, R. C., HERNDON, K. P., and HUGHES, J. F. 1996. SKETCH: An Interface for Sketching 3D Scenes. In *Proceedings of ACM SIGGRAPH 96*, ACM, 163-170.

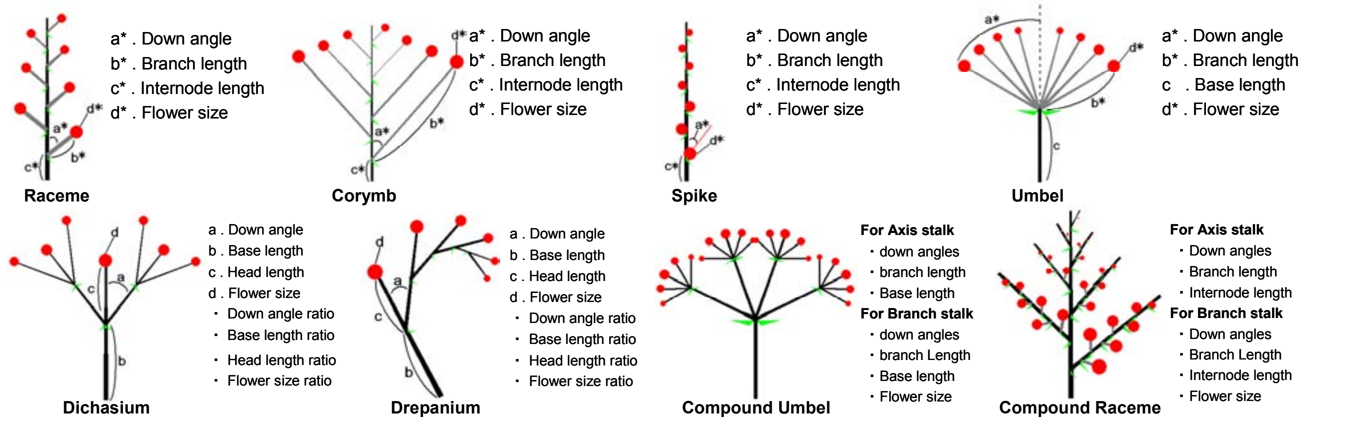
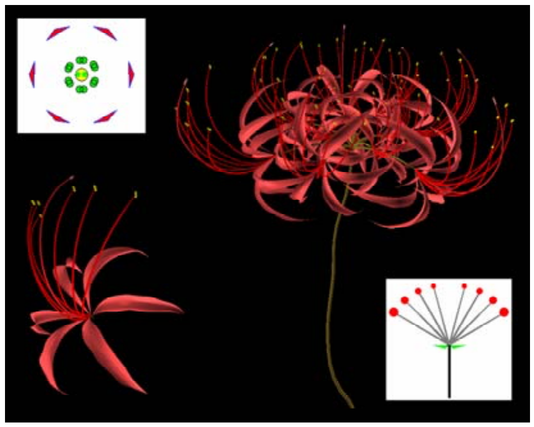
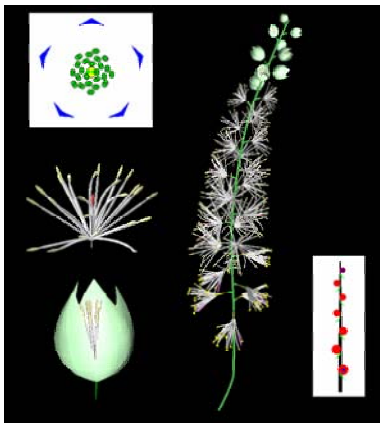
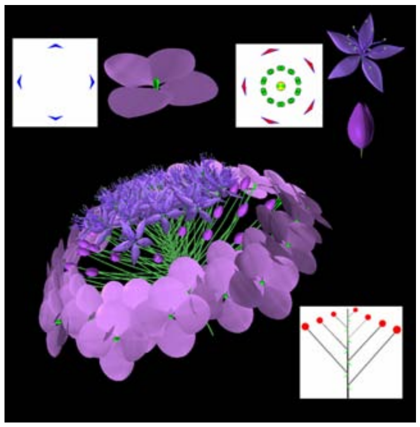
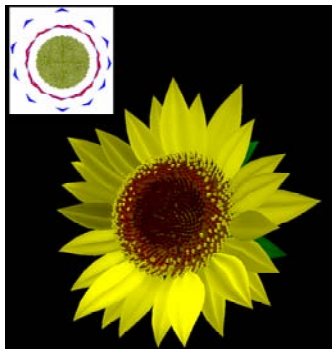


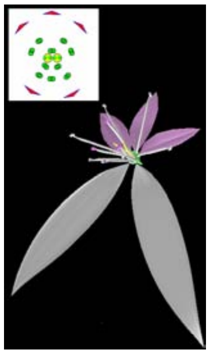
Figure13: Inflorescence patterns and their parameters in our current implementation. The parameters with the superscript '*' are pair of numbers to be linearly interpolated along the stem. There are also some common parameters that are not shown in the figure: phototropism direction, stem hardness, stem width, rotate angle, and the number of branches. Dichasium and Drepanium patterns have additional "ratio" parameters for all parameters that determine the ratio of a child branch's parameter values to those of a parent branch.

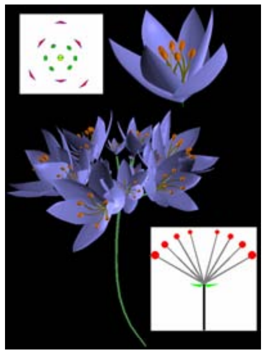

(a) Lycoris radiate (40 min)

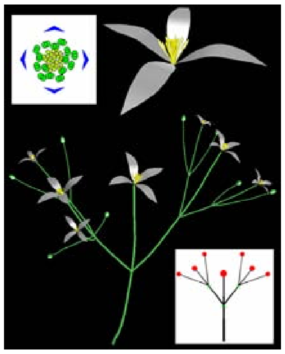

(b) Cimicifuga acerina (30 min)


(c) Hydrangea (40min)


(d) Sun flower (40min)


(e) Saxifraga stolonifera (15min)


(f) Allium roseum (30min)


(g) Clematis terniflora (30min)

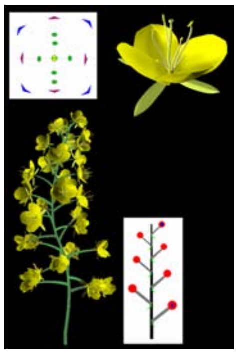

(h) Brassica rapa (30min)

Figure 14: Example models and the approximate time to complete each model. (a), (b), (d), and (e) are modeled by the author. (c), (f), (g), and (h) are designed by the test users.

EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2004)
John F. Hughes and Joaquim A. Jorge (Guest Editors)

Sketching garments for virtual characters

Emmanuel Turquin[1], Marie-Paule Cani[1] and John F. Hughes[2]

[1] Laboratoire GRAVIR (CNRS, INP Grenoble, INRIA, UJF)
[2] Brown University

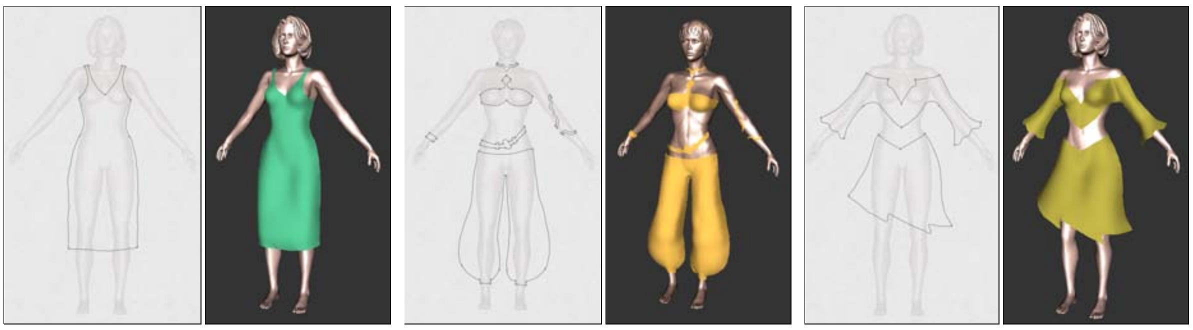


Figure 1: *Three examples of garments of various styles, respectively generated from the sketch on their left using our approach.*

Abstract

We present a method for simply and interactively creating basic garments for dressing virtual characters in applications like video games. The user draws an outline of the front or back of the garment, and the system makes reasonable geometric inferences about the overall shape of the garment (ignoring constraints arising from physics and from the material of the garment). Thus both the garment's shape and the way the character is wearing it are determined at once. We use the distance from the 2D garment silhouette to the character model to infer the variations of the distance between the remainder of the garment and the character in 3D. The garment surface is generated from the silhouette and border lines and this varying distance information, thanks to a data-structure that stores the distance field to the character's body. This method is integrated in an interactive system in which the user sketches the garment over the 3D model of the character. Our results show that the system can be used to create both standard clothes (skirts, shirts) and other garments that may be worn in a variety of ways (scarves, panchos).

Key words; Sketch-based interfaces, virtual garment, shape modeling.

Categories and Subject Descriptors (according to ACM CCS):
I.3.5 [Computational Geometry and Object Modeling]: Modeling packages
I.3.6 [Methodology and Techniques]: Interaction techniques

1. Introduction

Nowadays, 3D virtual characters are everywhere: in video games, feature films, TV shows... Obviously, these characters need clothing: when we encounter a person, we instantly notice his or her clothes, and feel that these tell us something about the person. Thus clothing on virtual characters provides a way to easily advance the story being told to the viewer.

The range of approaches used for clothing virtual characters is large: for incidental characters, the clothing may be no more than a texture map. For lead characters in feature films, full-fledged physical simulation of detailed cloth models may be used. And in the midrange, simple skinning techniques, combined with texture mapping, are common, providing some deformation of clothing as the character moves, but no physical realism. There are three problems one can associate with clothing virtual characters: the design of the clothes (*tailoring*), placing them on the character (*dressing*), and making them look physically correct (typically through *simulation*).

The process of tailoring, for human beings, involves choosing cloth and fitting it to the body, often making adjustments in the patterns of the cloth to adapt it to the particular person's form, and then sewing it. For virtual characters, clothing often has no "patterns" from which it is sewn, instead being represented by a simple polygon mesh that's constructed to fit the body. It's currently tedious to construct such meshes even without the issues of patterns and stitching. It's sometimes done by directly incorporating the cloth mesh into a character's geometric model, so that the character doesn't actually have *legs*, for instance, but just *pants*. In this case physical simulation is no longer a possibility, and when a character needs new clothes, it must be largely remodeled. An alternative approach involves drawing pattern pieces for a garment and positioning them over the naked form of the character, defining stitching constraints, etc. This can be tedious, especially when the character is not important enough to merit this level of effort; it also requires an understanding of how cloth fits over forms, although the actual pattern-and-stitching information may not be relevant after the tailoring is completed (except in the rare case where the physical properties of the cloth — was it cut on the bias? Does the cloth resist folding along one axis? — are later used in a full-fledged physical simulation).

Our approach combines tailoring and dressing into a single step to create a mesh that's suitable for later simulation or skinning approaches. The idea is to make it easy to generate simple garments that are adapted to an existing model. We believe that most people know better how to draw garments than the patterns which are needed to sew them. The aim of this work is thus to explore the use of a sketch-based interface for quickly constructing 3D virtual garments over a character model. This paper describes simple solutions to the problems of *shape generation* and *placement* of the clothing. The resulting system is so easy to use that it takes only minutes to create a simple garment.

1.1. Related work

Current interactive systems [HM90, WMTT93, BGR02] for designing garments and dressing virtual actors with them can be quite tedious: typically the user must draw each pattern piece on a planar virtual cloth, specify the edges to be stitched together, position the pieces around the virtual actor, and then finally run a simulation to obtain a convincing rest shape for the garment around the character model.

Sketch-based modeling systems such as [ZHH96, EyHBE97, IMT99, FBSS04] have became popular for interactively generating 3D geometry from sketches. Most of them only generate solid objects, as opposed to such surfaces as garments.

Two works have combined the idea of sketching with the goal of designing clothes: Bourguignon [BCD01] provided a 3D sketching method and used it to design garments over virtual actors. The sketch could be viewed from arbitrary viewing angles, but no 3D surface was reconstructed for the garment. Igarashi [IH02] described a sketch-based method for positioning garment patterns over a 3D body, but the user could not use the system for directly sketching the desired garment and still must know which pattern shapes will result in the garment he desires. That is to say, they addressed the *dressing* problem through sketching, but not the *tailoring* problem.

1.2. Overview

This paper presents a method for reconstructing the 3D geometry and placement of garments from a 2D sketch. As in [BCD01], the user sketches the garment directly on the 3D virtual actor body model. However, our method outputs a full 3D geometry for the garment, using the distance from the 2D garment silhouette to the character body model to infer the variations of the distance between the garment and the character in 3D.

More precisely, the reconstruction is performed in 3 steps: the lines of the 2D drawing are first automatically classified either as *silhouettes* (lines that do not cross the body) or as *border lines* (lines that do cross the body). A distance-to-the-body value is computed for each point of a silhouette segment and these distances are then used to determine desired point-to-body distances for the border lines. This distance information is then propagated in 2D to find desired point-to-body distances, which are then in turn used to determine the 3D position of the garment.

2. Sketch-based interface

2.1. Typical user interaction

To give a sense of the system's performance, we describe a typical interaction, in which a user sketches a skirt on a female model. The user first draws a line across the waist (see figure 2 (left)), indicating the top of the skirt, and then a line down the side, indicating the silhouette of the skirt, then a line across the bottom in a vee-shape indicating that he wants the front of the skirt to dip down, and finally the last side. A simple corner-detection process is applied to break the sketch into parts; one extra corner is detected by accident (at the bottom of the vee) and the user can delete it with a deletion gesture. He could also add new breakpoints as well, but none are necessary. Breakpoints play an important role in the 3D positioning process, since they determine

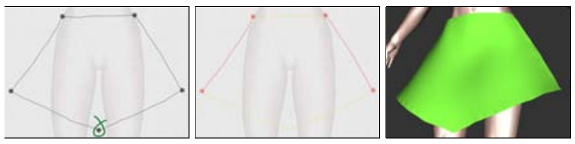


Figure 2: ***(left)*** *The user has drawn a few lines to indicate the shape of the skirt; the corner-detector has detected a breakpoint that the user does not want. The user will make a deletion gesture (a curve in the shape of an* α *enclosing the mistaken point) to delete it.* ***(middle)*** *The user has deleted the breakpoint, and the lines have been classified: the* sil-houettes *are in red and the* borders *in yellow.* ***(right)*** *The surface inferred by the system once the user has required a reconstruction.*

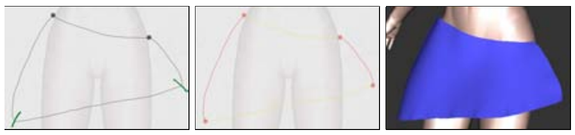


Figure 3: ***(left)*** *The user drew the outline of the skirt without sharp corners at the bottom, and the corner-detector failed to put breakpoints there. The user therefore makes gestures (overdrawn in green here) to indicate the need for new breakpoints.* ***(middle)*** *The new breakpoints have been inserted.* ***(right)*** *The skirt is reconstructed.*

the global 3D position of the cloth with respect to the body. The way they are used is detailed in Section 4.3. The two lines on the sides are classified as silhouettes, and the others are classified as border lines, as shown in the figure.

Now the user asks to see the garment inferred by the system; a surface matching the drawn constraints, but adapted to the shape of the underlying form (look near the waistine, for instance) appears almost instantly (see figure 2 (right)).

Sometimes, as in figure 3, the breakpoint-inference fails to detect all the points the user wants; in this case she or he can make a gesture to add new breakpoints.

In the current state of our system, only the front of the garment is generated; the back would have to be generated in a second pass, possibly through a simple mirror image of the silhouette strokes and user-applied modification of the border strokes. There's no texture on the surface, no indication of how a surface of this form could be constructed from flat cloth, no method for indicating that the front and back should be joined. Thus the current interface concentrates simply on the way that shape and placement can be inferred from simple strokes, not on the entire tailoring and dressing process.

2.2. Gestural interface components

The user's marks are interpreted as gestures, with the default being the construction of silhouette and border line segments. Other gestures add breakpoints for the classification process, delete breakpoints, delete a segment or an entire chain of segments, and clear all segments, as shown schematically in figure 4. (Our gestures are similar to those of Tsang et al. [TBSR04]).

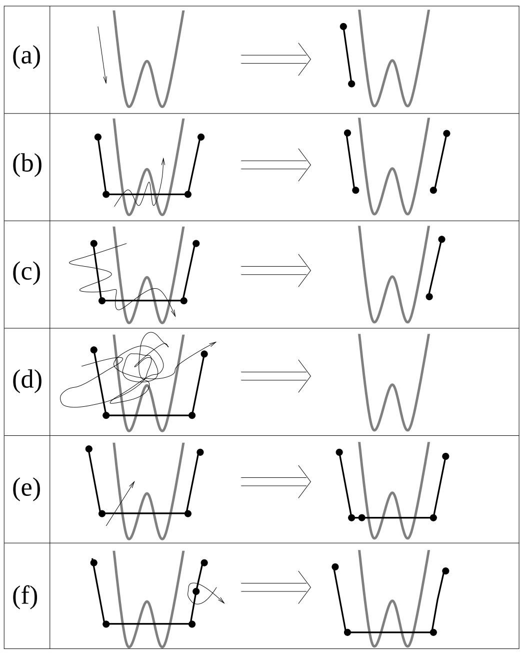


Figure 4: *The gestures in the interface; the model is drawn in thick lines, prior strokes as medium lines with dots at breakpoints, and new strokes with thin lines; the arrowhead on the new-stroke lines is not actually drawn by the user, but merely indicates the direction in which the stroke is drawn.* ***(a)*** *adding a segment;* ***(b)*** *deleting a segment (stroke must intersect the segment at least five times);* ***(c)*** *deleting several segments (the stroke must intersect more than one segment, and intersect segments at least five times. If the stroke intersects segments from two different chains, both chains are deleted entirely.);* ***(d)*** *clearing all segments (the stroke must intersect some segment and count at least fourty intersections)* ***(e)*** *adding a breakpoint* ***(f)*** *deleting a breakpoint (the stroke must intersect the segments on either side of the breakpoint, and intersect itself once).*

The breakpoint-deletion gesture matches well with the standard typsetting deletion-mark; the other deletion gestures require multiple intersections with existing strokes to prevent accidental deletions.

3. Interpreting sketches of garments: basic ideas

For the sake of clarity, we'll assume that the character is aligned with the *xy*-plane, viewed along the *z* direction.

The user draws the contours of the 2D projection of the garment in the (x, y) plane over the rendered character model. From these, we need to infer the *z*-information at every point of the contour and the interior of the garment. Clearly this problem is under-constrained; fortunately, by considering the special nature of clothing, we can generate a plausible guess of the user's intentions. In particular, silhouettes of clothing not only indicate points where the tangent plane to the cloth contain the view direction; they usually occur as the clothing passes around the body, so we can estimate the z-depth of a silhouette edge as being the z-depth of the nearest point on the body (or, for a body placed fairly symmetrically about the xy-plane, simply use $z = 0$ as a quicker estimate). Moreover, the distance from a silhouette to the body helps us to infer the distance elsewhere, since a cloth which tightly fits the body in the (x,y) plane will also tend to fit it in the z direction, while a loose cloth will tend to float everywhere.

3.1. Overview of the algorithm

Our algorithm, whose different steps will be detailed in the following sections, develops as follows:

1. Process the 2D sketch of the garment:
   - Find high-curvature points (breakpoints) that split the contours into segments;
   - Let user add and/or delete breakpoints.
   - Classify segments of curve between breakpoints into *border lines* (which cross the character's body) or *silhouette lines*.
2. Infer the 3D position of silhouette and border lines:
   - For each breakpoint that does not lie over body, find distance to body, d, and set the point's depth, z, to the depth of the closest point on the body.
   - For each silhouette line, interpolate z linearly over the interior of the line, and use interpolated z-values to compute d-values (distance to the body in the 3D space) over the interior of the line.
   - For each border line, interpolate d over interior linearly to establish a desired distance to the model and set a z value for each point on the line;
3. Generate the interior shape of the garment:
   - Create a mesh consisting of points within the 2D simple closed curve that the user has drawn, sampled on a rectangular grid in the (x,y) plane.
   - Extend the values of d, which are known on the boundary of this grid, over the interior.
   - For each interior grid point (x,y), determine the value of z for which the distance from (x,y,z) to the body is the associated value of d.
   - Adjust this tentative assignment of z values to take into account the surface tension of the garment between two limbs of the character.
   - Tesselate the grid with triangles, clipped to the boundary curves, and render the triangles.

3.2. Pre-computing a distance field

To accelerate steps 2 and 3 of the algorithm above, a distance field to the character's model is pre-computed when the model is loaded: for each point of a 3D grid around the model, we determine and store the distance to the nearest point of the model, using the octree-based algorithm in [JS01].

The distance field will be used each time we need to find the z coordinate to assign to a point $p(x_0,y_0)$ so that it lies at a given distance from the model. This can easily be done by stepping along the ray $R(z) = (x_0,y_0,z)$ and stopping when the adequate distance value is reached (we interpolate trilinearly to estimate distances for non-grid points). When this computation is performed during a sweeping procedure, the stepping starts at the z value already found at a neighbouring pixel, which ensures the spatial coherence of the result and efficiency. Else, the process starts near the near plane of our rectangular frustum, in which the distance field has been computed.

The quality of the results depends directly on the resolution of the 3D grid storing the distance field, as does computation time. The size of the 3D grid is user-configurable, but we have generally used a $32 \times 32 \times 32$ grid.

4. Processing of contour lines

4.1. 2D processing

First, one must classify the parts of the user's sketch. As the user sketches, a new line that starts or ends near (within a few pixels of) an endpoint of an already-sketched line is assumed to connect to it. When the sketch is complete (i.e., forms a simple closed curve in the plane), we classify the parts:

- First the sketched lines are broken into segments by detecting points of high (2D) curvature (breakpoints) (this is actually done on-the-fly, as the user draws the strokes).
- Each segment is classified as a silhouette or border line: border lines are ones whose projection meets the projection of the body in the xy-plane, silhouettes are the others. The mask of the body's projection is pre-computed and stored in a buffer called the *body mask*, in order to make this classification efficient.
- The resulting segmentation is visible to the user, who may choose if necessary to add or delete breakpoints indicating segment boundaries, after which segments are reclassified.

4.2. Distance and z-value at breakpoints

Breakpoints that are located over the body model (which is tested using the body mask) are used to indicate regions of the cloth that fit very tightly to the body. They are thus assigned a zero distance to the model, and their z value is set to the body's z at this specific (x,y) location.

To assign a distance value d to a breakpoint that does not lie over the body, we step along the ray from the eye in the direction of the breakpoint's projection into the xy-plane, checking distance values in the distance field data structure as we go, and find the z-value at which this distance is minimized. We assign to the breakpoint the discovered z and d values, thus positioning the breakpoint in 3D.

4.3. Line positioning in 3D

Our aim is to use the 3D position of the breakpoints we just computed to roughly position the garment in 3D, while the garment's shape will mostly be inferred from distances to the body along the sketch silhouettes.

To position the silhouette lines in 3D, we interpolate z linearly along the edge between the two breakpoints at the extremities of the silhouette. We then set the d-values for interior points of the silhouette to those stored in the pre-computed distance field. (We could instead interpolate d directly, and compute associated z-values, but if the body curves away from the silhouette curve, there may be no z value for which the interpolated d-value is the distance to the body; alternatively, we could compute d directly for each interior point and then assign the z-value of the closest body point, as we did with the breakpoints, but in practice this leads to wiggly lines because of the coarse grid on which we pre-compute the approximate distance-to-body.)

Having established the values of z and d along silhouette edges, we need to extend this assignment to the border lines. We do this in the simplest possible way: we assign d linearly along each border line. Thus, for example, in the skirt shown above, the d-values at the two ends of the waistline are both small, so the d-value for the entire waistline will be small, while the d-values for the ends of the hemline are quite large, so the values along the remainder of the hemline will be large too.

5. 3D Reconstruction of the garment's surface

5.1. Using distance to guess surface position

As for the contour lines, the interpolation of distances to the body will be our main clue for inferring the 3D position of the interior of the garment. The first process thus consists into propagating distance values inside the garment. This is done in the following way:

The 2D closed contour lines of the garment are used to generate a (x,y) buffer (sized to the bounding box of the sketch) in which each pixel is assigned one of the values 'in', 'out' or 'border', according to its position with respect to the contour. A border pixel is one for which some contour line intersects either the vertical or horizontal segment of length one passing through the pixel-center (see figure 5); other pixels are inside or outside, which we determine using

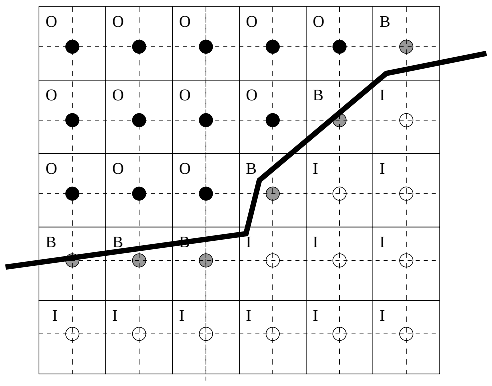


Figure 5: *Each grid point represents a small square (solid lines). If a contour (heavy line) meets a small vertical or horizontal line through the pixel center (dashed lines), the pixel is classified as a boundary; boundary pixels are marked with a "B" in this figure. Others are classified as inside ("I") or outside ("O") by winding number.*

a propagation of 'out' values from the outer limits of the buffer.

The distance values already computed along the silhouette and border lines are assigned to the border pixels. The distances for the 'in' pixels are initialized to the mean value of distances along the contour.

Then, distance information is propagated inside the buffer by applying several iterations of a standard smoothing filter, which successively recomputes each value as an evenly weighted sum (with coefficients summing to 1) of its current value and those of its "in" or "boundary" orthogonal or diagonal neighbours (see figure 6). The iterations stop when the maximum difference between values obtained after two successive iterations is under a threshold, or if a maximum number of steps has been reached. Convergence generally seems to be rapid, but we have not proven convergence of the method in general.

We then sweep in the 2D grid for computing z values at the inner pixels: as explained in section 3.2, the z value is computed by stepping along a ray in the z direction, starting at the z value we already assigned for a neighbouring point, and taking the z the closest to the desired distance value, as stored in the pre-computed distance field.

5.2. Mimicking the cloth's tension

The previous computation gives a first guess of the garment's 3D position, but still results in artefacts in regions located between two limbs of the character: due to surface tension, a cloth should not tightly fit the limbs in the in-between region (as in figure 7 (top)), but rather smoothly interpolate the limb's larges z value, due to its surface tension.

To achieve this, we first erode the 2D body mask (already mentioned in section 4.1) of a proportion that increases with

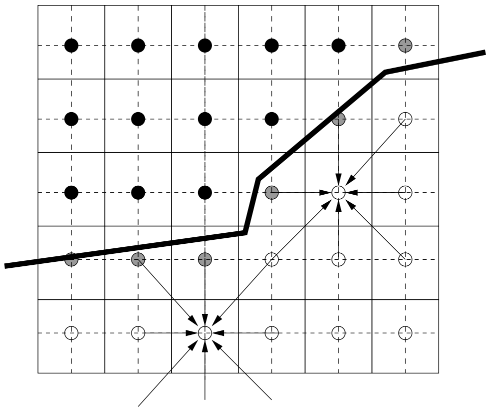


Figure 6: *The arrows indicate which neighboring pixels contribute to a pixel during the smoothing process: only neighboring* boundary *or* inside *pixels contribute to the average.*

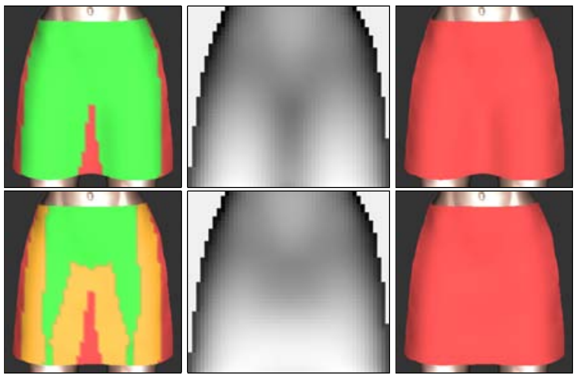


Figure 7: *Surface reconstruction without* ***(top)*** *versus with* ***(bottom)*** *taking tension effects into account. The part of the surface over the body mask is shown in green in the left images. At bottom left, the body mask has been eroded and Bézier curves used to infer the z-values between the legs. The resulting z-buffers are shown in the middle images and the reconstructed surfaces on the right.*

the underlying d value (see figure 7) (bottom left)). We then use a series of Bezier curves in horizontal planes to interpolate the z values for the pixels in-between. We chose horizontal gaps because of the structure of the human body: for an upright human (or most other mammals), gaps between portions of the body are more likely to be bounded by body on the left and right than to be bounded above and below.

To maintain the smoothness of the garment surface near the re-computed region, distances values are extracted from the new z values and the distance field. Some distance propagation iterations are performed again in 2D, before recomputing the z values int the regions not lying over the body that haven't been filled with the Bezier curves, as was done previously (see figure 7 (right)).

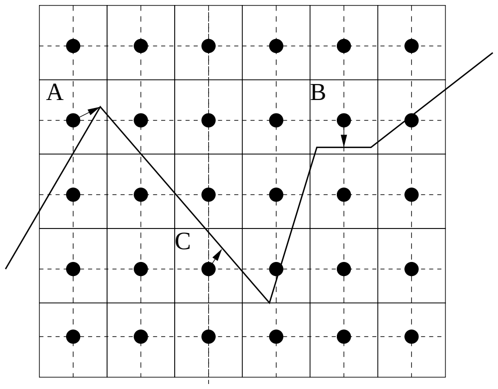


Figure 8: *When a segment ends in a pixel's box, the pixel center gets moved to that endpoint as in the box labelled "A." If more than one segment ends, the pixel center is moved to the average, as in "B". Otherwise, the pixel center is moved to the average of the intersections of segments with vertical and horizontal mid-lines, as in "C".*

We finally add a smoothing step on the z values in order to get a smoother shape for the parts of the cloth that float far from the character's model. This is done computing a smoothed version of the z-buffer from the application of a standard smoothing filter, and then by taking a weighed average, at each pixel, of the old and smoothed z-values, the weighing coefficients depending on the distance to the model at this pixel.

5.3. Mesh generation

Finally, the triangles of the standard diagonally-subdivided mesh are used as the basis for the mesh that we render (see figure 8): all "inside" vertices are retained; all outside vertices and triangles containing them are removed, and "boundary" vertices are moved to new locations via a simple rule:

- If any segments end within the unit box around the vertex, the vertex is moved to the average of the endpoints of those segments. (Because segments tend to be long, it's rare to have more than one endpoint in a box).
- Otherwise, some segment(s) must intersect the vertical and/or horizontal mid-lines of the box; the vertex is moved to the average of all such intersections.

6. Results

The examples presented in figure 1 and figure 9 were drawn in no more than 5 minutes each. We want to point out that the jagged appearance of the strokes in the drawings is simply due to the use of a mouse instead of a more adequat graphics tablet as the input device. The gallery includes simple clothes such as pullovers, skirts, robes and shirts, but also less standard ones such as an "arabian like" pair of trousers

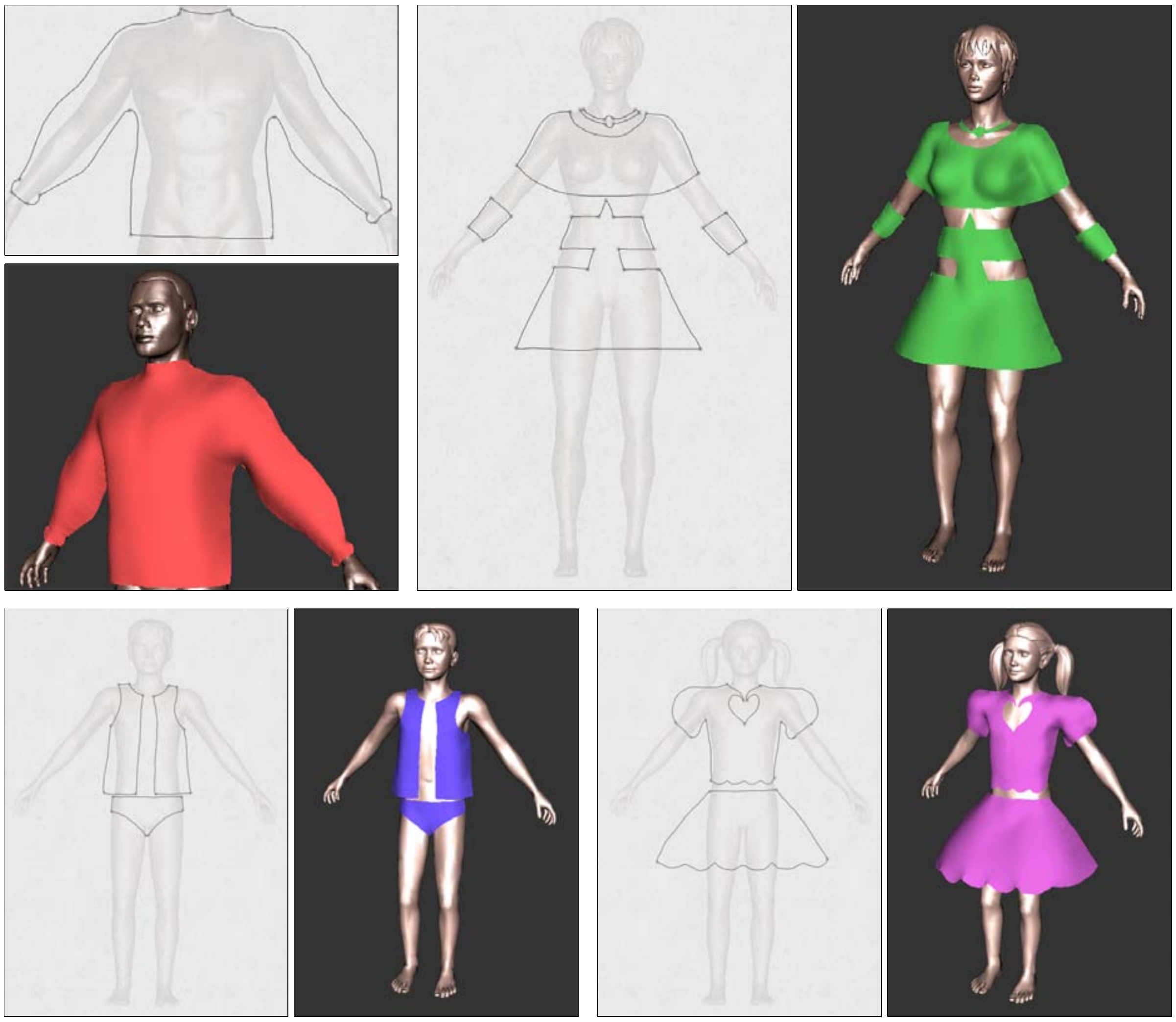


Figure 9: *Gallery of examples of garments created with our system.*

along with basic jewelry, or exantric outfits that would certainly fit nicely in a *Haute couture* collection. This wide range of types of cloth gives an idea of the expressivity our system provides the user with.

7. Discussion

We have presented a method for dressing virtual characters from 2D sketches of their garments. We used the distance from the 2D garment silhouette to the character model to infer the variations of the distance between the garment and the character in 3D. The garment surface is generated from the silhouette and border lines and this varying distance information, thanks to a data-structure which stores the distance field to the character's body. This method has been integrated in an interactive system in which the user can interactively sketch a garment and get its 3D geometry in a few minutes.

There are other approaches that could be used as well: one could imagine a parametric template for shirts, for instance, and a program that allowed the user to place the template over a particular character and then adjust the neckline, the overall size of the shirt, etc. But this limits the realm of designs to those that exist in the library of pre-defined templates (how could one design an off-one-shoulder shirt if it wasn't already in the library), and limits it to standard forms as well: dressing the characters in a movie like *Toy Story* becomes impossible, since many of them are not human forms. Nonetheless, the model-based approach is a very reasonable one for many applications, such as a "virtual Barbie Doll," for instance.

The automated inference of the shape is simple and easy-to-understand, but may not be ideal in all cases, and we have not yet provided a way for the user to edit the solution to make it better match the idea that she or he had when sketching.

8. Future work

To infer the shape of a full garment, we could use symmetry constraints around each limb to infer silhouettes for the invisible parts, and ask the user to sketch the border lines for the back view, for instance.

We'd also like to allow the user to draw folds, and take them into account when reconstructing the 3D geometry, so that even for a closely-fitting garment, there can be extra material, as in a pleated skirt.

Most cloth is only slightly deformable, so the garment we sketch should be locally developable into a plane everywhere. We did not take this kind of constraint into account in our model, but this would be an interesting basis for future work, including the automatic inference of darts and gussets as in [MHS99].

Finally, we have sketched clothing as though it were simply a stiff polygonal material unaffected by gravity. We would like to allow the user to draw clothing, indicate something about the stiffness of the material, and see how that material would drape over the body. The difference between silk (almost no stiffness), canvas (stiff), and tulle (very stiff) can generate very different draping behaviors. In the much longer term, we'd like to incorporate a simulator that can simulate the difference between bias-cut cloth and straight-grain, the former being far more "clingy" than the latter.

References

[BCD01] Bourguignon D., Cani M.-P., Drettakis G.: Drawing for illustration and annotation in 3D. *Computer Graphics Forum 20*, 3 (2001). ISSN 1067-7055. 2

[BGR02] Bonte T., Galimberti A., Rizzi C.: *A 3D graphic environment for garments design*. Kluwer Academic Publishers, 2002, pp. 137–150. 2

[EyHBE97] Eggli L., yao Hsu C., Bruderlin B. D., Elber G.: Inferring 3d models from freehand sketches and constraints. *Computer-Aided Design 29*, 2 (1997), 101–112. 2

[FBSS04] Fleisch T., Brunetti G., Santos P., Stork A.: Stroke-input methods for immersive styling environments. In *Proceedings of the 2004 International Conference on Shape Modeling and Applications, June 7–9, Genova, Italy* (2004), pp. 275–283. 2

[HM90] Hinds B., McCartney J.: Interactive garment design. *The Visual Computer 6* (1990), 53–61. 2

[IH02] Igarashi T., Hughes J. F.: Clothing manipulation. In *Proceedings of the 15th annual ACM symposium on User interface software and technology* (2002), ACM Press, pp. 91–100. 2

[IMT99] Igarashi T., Matsuoka S., Tanaka H.: Teddy: a sketching interface for 3d freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 409–416. 2

[JS01] Jones M. W., Satherley R.: Using distance fields for object representation and rendering. In *Proceedings of the19th Ann. Conf. of Eurographics (UK Chapter)* (2001), pp. 91–100. 4

[MHS99] McCartney J., Hinds B. K., Seow B. L.: The flattening of triangulated surfaces incorporating darts and gussets. *Computer Aided Design 31*, 4 (1999), 249–260. 8

[TBSR04] Tsang S., Balakrishnan R., Singh K., Ranjan A.: A suggestive interface for image guided 3d sketching. In *Proceedings of CHI 2004, April 24–29, Vienna, Austria* (2004), pp. 591–598. 3

[WMTT93] Werner H. M., Magnenat-Thalmann N., Thalmann D.: User interface for fashion design. In *Proceedings of the IFIP TC5/WG5.2/WG5.10 CSI International Conference on Computer Graphics* (1993), North-Holland, pp. 197–204. 2

[ZHH96] Zeleznik R. C., Herndon K., Hughes J. F.: Sketch: An interface for sketching 3d scenes. In *Computer Graphics Proceedings, SIGGRAPH'96* (New Orleans, Louisiana, August 1996), Annual Conference Series, ACM. 2

Clothing Manipulation

Takeo Igarashi
Computer Science Department, University of Tokyo
7-3-1 Hongo, Bunkyo, Tokyo, 113-0033 Japan
E-mail: takeo@acm.org

John F. Hughes
Computer Science Department, Brown University
Providence, RI 02912, USA
E-mail: jfh@cs.brown.edu

ABSTRACT

This paper presents interaction techniques (and the underlying implementations) for putting clothes on a 3D character and manipulating them. The user paints freeform marks on the clothes and corresponding marks on the 3D character; the system then puts the clothes around the body so that corresponding marks match. Internally, the system *grows* the clothes on the body surface around the marks while maintaining basic cloth constraints via simple relaxation steps. The entire computation takes a few seconds. After that, the user can adjust the placement of the clothes by an enhanced dragging operation. Unlike standard dragging where the user moves a set of vertices in a single direction in 3D space, our dragging operation moves the cloth *along the body surface* to make possible more flexible operations. The user can apply pushpins to fix certain cloth points during dragging. The techniques are ideal for specifying an initial cloth configuration before applying a more sophisticated cloth simulation.

KEYWORDS: User Interface, Clothing.

INTRODUCTION

Putting clothes on a 3D character is often a tedious, time-consuming task. A typical approach is to place parts of the clothes around the target body as rigid thin plates and use a simulation to enforce “stitch-together” constraints and show the effects of gravity [23]. The 3D character may be placed in a particular pose (e.g., arms outstretched) and then some “throwaway” animation may be used to get the character into a desired pose [3]. However, placing thin plates in free 3D space using a 2D input device is difficult, and it is not very flexible for exploring various nonstandard ways of wearing clothes. Recent fast cloth simulation systems enable real-time manipulation of clothes: the user can grab a piece of clothing and drag it around in 3D space [7,8]. But this is like manipulating clothes with chopsticks; it’s not ideal for putting clothes on a 3D character.

In this paper we introduce a set of *interaction techniques* for putting clothes on a 3D character (here called the *body*) quickly and intuitively using 2D input devices. The techniques are designed for specifying an approximate initial cloth configuration before applying a high-quality cloth simulation to obtain a final, good-looking cloth shape or animation. The intention is that the interface should also be useful for exploring various cloth configurations quickly during the design process, both in 3D character design and real-world fashion design. The interaction techniques are supported by an underlying *approximate simulation*

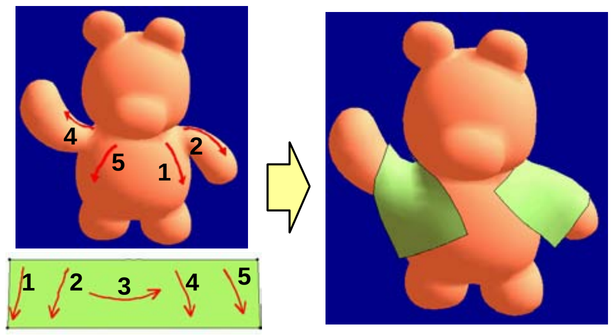


Figure 1: Wrapping. The user paints pairs of freeform marks on the target body and on the clothes (left); the system places the clothes on the body so that the corresponding marks match (right). The result appears almost instantly. (The mark numbering has been added by hand to clarify the correspondences.)

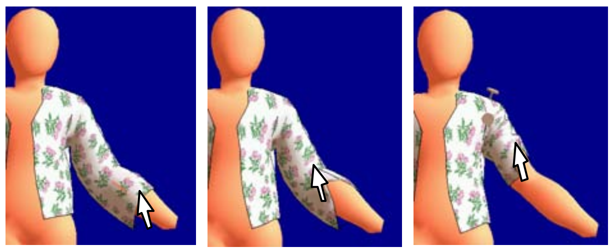

Before dragging Vertex dragging Surface dragging

Figure 2: Surface dragging. A typical vertex-dragging operation moves only one vertex explicitly, causing large local distortion. Surface dragging explicitly propagates motion across the clothes, enabling global manipulation. The pushpin on the shoulder blocks further propagation.

technique whose details we describe briefly, particularly as they relate to the interactions, but which could be replaced by any other sufficiently rapid simulation. Our sole requirement is that both the clothing and the 3D character be represented as polygonal manifold meshes.

The first technique, *wrapping*, is for putting the clothes on the body from scratch. The user paints freeform marks on the clothes and corresponding marks on the body, and in a few seconds the system places the clothes on the body in such a way that the corresponding marks match (Figure 1). The second technique, *surface dragging,* is for adjusting the configuration of clothes already on the body. While a typical cloth-dragging operation moves a set of vertices in a single direction in 3D space, our dragging operation moves the cloth along the body surface (Figure 2). The user can also place pushpins to hold some clothing parts fixed during dragging. We describe the user interface of the system first and describe the implementations of those operations later.

RELATED WORK

The computer graphics community has been interested in cloth modeling for decades [14,23]. Early approaches were purely geometric [24], but recent systems use physically based simulations for generating realistic pictures and animations [2,5,6]. Some systems also allow real-time cloth manipulation [8]: the user can drag the cloth around in a 3D space with appropriate cloth-like behavior, and can even get haptic feedback [7].

Cloth simulation is common in commercial 3D computer graphics programs today [20,22]. The typical interface for putting clothes on a 3D character is to set the character in a canonical dress-up pose, place the clothes around it as rigid objects, and then start a simulation to let the clothes fall into a natural position. Some systems let the user specify various constraints or motion paths for specific cloth vertices to control the simulation.

The garment-design industry has been using 2D pattern design programs (apparel CAD) for years [10,17], and recently started incorporating 3D features [1,9]. They use predefined mappings between 2D cloth patterns and a 3D mannequin surface, where the manipulations in the 2D editor appear simultaneously in 3D space.

Our interface is motivated by the recent sketching interfaces for 3D modeling [12,25]. These tools are designed for exploratory design and for communication during discussion; they are designed to support ease of use in rapid model creation rather than the refined kinds of modeling needed in the final design stages. Our goal is to develop similar easy-to-use design tools for clothes.

THE USER INTERFACE

The system has two windows: the *pattern-design window* for editing 2D cloth patterns and the *3D window* for manipulating cloth on a 3D character (the *body*)(Figure 3). The user first edits a 2D cloth pattern in the pattern-design window and then puts the clothes on the body using the wrapping operation. The user can then manipulate the clothing using surface dragging and pushpins.

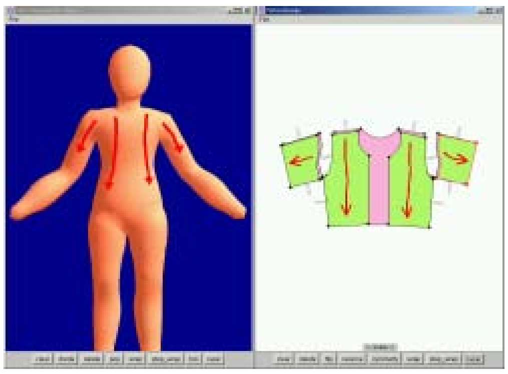


Figure 3: A screen snapshot of the system. Pink is the inside surface of the cloth, green is the outside, and the small gray lines indicate sewing constraints applied by the user.

The 2D pattern editor is a specialized 2D drawing program. The user draws pieces of cloth as closed polygons that can be freely moved and scaled. Each piece has distinct front and back sides, and the user can flip a piece to see its other side. The user can also indicate that two edges from different pieces are to be connected by specifying *sewing constraints*. The system maintains equality of the lengths of connected edges during pattern editing. We also provide simple editing operations such as duplication of pieces and making a piece horizontally symmetric. The implementation of the pattern editor is straightforward and hence not described here.

The 3D viewing window works as a typical 3D object viewer. The user can rotate and move the body three-dimensionally using the right mouse button [15].

Wrapping

To put 2D clothes on the body, the user paints freeform marks on the cloth pattern and the body using the left mouse button (Figure 1). The marks are numbered internally based on the order of painting independently on the clothes and the body, and marks with corresponding numbers are associated with each other.

After painting the marks, the user presses the “wrap” button. The system calculates the desired 3D cloth configuration on the body and shows the result in the 3D window. For the examples shown in the figures here, wrapping takes a few seconds in our current implementation. After presenting the initial result, the system continuously refines the cloth configuration via a relaxation operation. Similar mark-based interaction technique is used in a feature-based image morphing [4].

Wrapping is a best-effort operation. The system tries for a reasonable result satisfying the constraints specified by the user, but undesirable results can be generated depending on the configuration of marks. If the problem is small, the user can adjust the cloth placement via the surface dragging operation. However, if the result has serious topological problems such as the body penetrating the cloth, the user must cancel the wrapping operation and adjust the configuration of marks.

For the user's convenience, the system includes a "laser-paint" mode [11] in which the user's mark is automatically painted on both the front and back sides of the body. In the pattern-design window a laser-painted mark is painted on the front-facing cloth piece and the underlying back-facing piece (if any).

Wrapping can also be used to adjust the configuration of clothes on the body (called *rewrapping*) (Figure 4). Pushpins may be used to restrict the rewrapped area. Rewrapping is especially useful for edits involving topological changes (Figure 4 right).

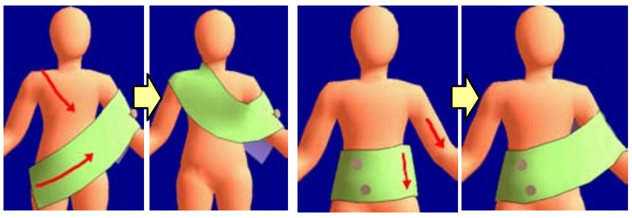


Figure 4: Rewrapping clothes already on the body. The pushpins limit the rewrapping region.

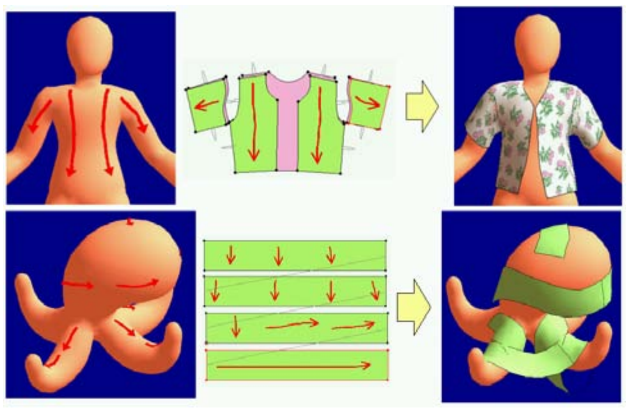


Figure 5: Examples of wrapping. In the first, laser-paint is used to duplicate marks front and back; in the second, we put a scarf on an octopus.

Figure 5 shows two examples of wrapping. The system generally returns the desired results, but the user must provide enough marks to avoid undesirable effects. For example, both the front and back sides must be painted to put a sleeve around an arm. Otherwise the cloth stays on one side of the arm. It is difficult to see the correspondence between pairs of marks in a still picture, but is easy for the user, who can paint corresponding marks on the pattern and the body alternately.

Surface Dragging

After putting clothes on the body, the user can adjust the placement of the clothes using *surface dragging*. The user clicks and drags the clothes using the right mouse button. This operation is superficially the same as the typical dragging operations in interactive real-time cloth-simulation systems.

In typical cloth-simulation systems [8], a user's dragging operation applies a force to a single vertex, and the system simulates the consequent forces on the rest of the cloth to create a larger-scale effect. This approach (which we call *vertex dragging*) is useful for adjusting very local cloth shape, but is inconvenient for more global cloth manipulations such as revolving a skirt around a body or pulling the sleeve upwards (see Figure 6). Single vertex dragging induces large deformations near the vertex, since other vertices resist the motion because of the friction against the body. In addition, vertex dragging can only *pull* the cloth and cannot *push* it – if the user tries to push the cloth, flips and folds result near the vertex. Finally, vertex dragging is often implemented as an unconstrained 3D movement and is therefore difficult to control with 2D input devices. Some commercial systems allow the simultaneous modification of multiple points, possibly with an attenuation factor to ease out the deformation, but these vertices are all moved in the same direction in 3D space and so it is still cumbersome to move clothing along the body surface.

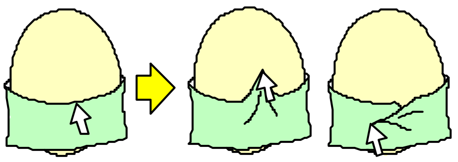


Figure 6: Limitations of conventional vertex dragging: it causes large stretch and folds instead of the desired upward slide or horizontal rotation of the entire cloth.

Our surface-dragging operation explicitly propagates the user's input motion across the clothes along the body surface to create a global effect. For example, if the user drags a vertex upwards, the system explicitly moves the surrounding cloth vertices upwards at the same time, and if the user drags the front side of a skirt to the right, surface dragging actually rotates the skirt horizontally around the body (Figure 7). Just as in wrapping, we apply a relaxation step after each dragging step to maintain the basic cloth constraints.

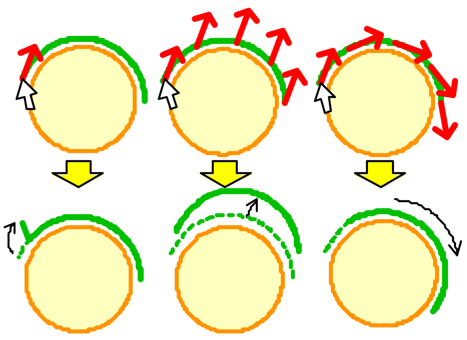


Figure 7: Various dragging approaches: vertex dragging (left), rigid dragging (center), surface dragging (right).

Surface dragging is constrained to directions parallel to the associated body surface and the user cannot pull the clothes away from the body[1]. The mouse cursor is projected onto the tangent plane to the body surface at the clickpoint. This makes dragging with a 2D input device much simpler and easier than completely free 3D motion. This constraint caused us no practical problems during typical operations (Figure 8).

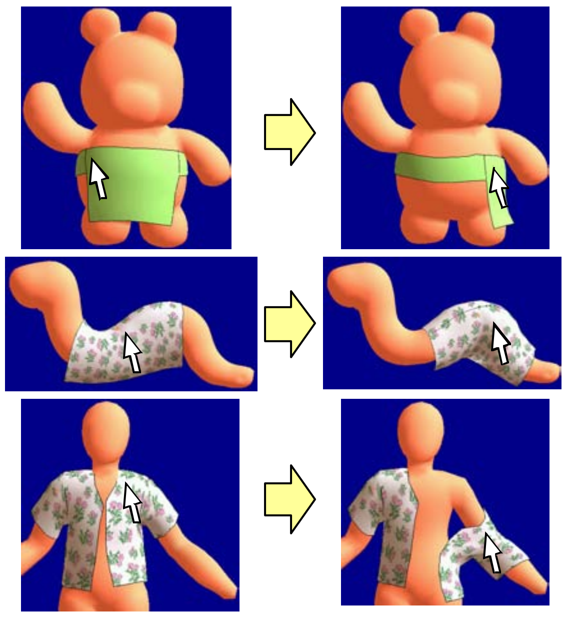


Figure 8: Surface dragging. The third example uses two pushpins on the back to block propagation. All examples run at a few frames per second.

Pushpins

The user can control the behavior of the clothes during surface dragging and the subsequent relaxation steps by putting *pushpins* on the clothes (Figure 9). The user places or removes a pushpin by clicking on the clothes with the left mouse button. A pushpin fixes a cloth vertex at some position on the body, thus helping in local cloth adjustments by blocking the propagation of motion during surface dragging. Pushpins are especially useful because dragging is a single-mouse operation—pushpins are often necessary to perform operations that require two hands in the real world.

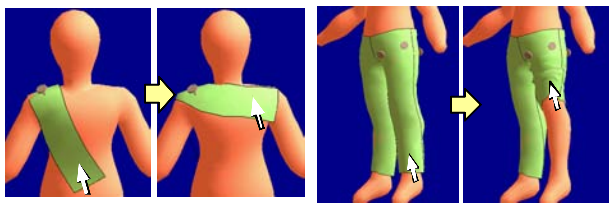


Figure 9: Surface dragging with pushpin.

ALGORITHMS

This section describes the algorithms for calculating the cloth configuration during manipulations. First the immediate goal position for each cloth vertex is computed in response to user input, and then relaxation steps adjust the positions to preserve basic cloth constraints such as prevention of penetration and limiting stretch. These two phases are actually closely integrated, but we describe them separately for clarity. We first discuss how to calculate immediate goal positions for cloth vertices during wrapping and surface dragging, and then describe how to preserve the cloth constraints.

The body and the clothes are represented as standard triangular meshes, and each cloth edge has an associated *rest length*. The parameters defining the behavior of the algorithms must be set accordingly to the characteristics of the target polygonal models. Our current implementation uses body models of 1.0~2.0 units in height and width that consist of a few thousand polygons.

Wrapping

Wrapping tries to put the clothes on the body so that the freeform marks on the clothes match the corresponding marks on the body. We begin by triangulating the cloth[2], since each cloth piece is initially a single polygon. Then we construct a single continuous mesh structure by combining the pieces of cloth according to the sewing constraints. Finally, we compute the geometry of the clothes by building a piecewise-linear map *f* from this mesh to 3-space by mapping the vertices one at a time (Figure 10). We'll say that an edge is *mapped* if both its vertices have been mapped, and that a triangle is mapped if all three of its vertices are mapped. The steps are

[1] Note that gravity can pull the clothes away from the body during the relaxation steps.

[2] We start with a constrained Delaunay triangulation and refine it iteratively, as in the "skin" algorithm [19]. An alternative triangulation algorithm [21] could work as well. The triangles must be small enough faithfully to represent the geometry of the body. We use a triangle edge length of 0.07~0.08 units.

1 Paste the clothes around the marks by defining *f* on the *root edges*, i.e., edges that cross marks on the clothes.

2 Grow the clothes by repeating the following process until all triangles are mapped:

(1) Find triangles with one unmapped vertex;
(2) Order the unmapped vertices;
(3) Map the vertices, performing relaxation after each.

To define *f* on a vertex *v* of a root edge *e* that crosses a mark *m* on the cloth corresponding to a mark *M* on the body, we first find the point *p* where *e* intersects *m*. The point *p* is some fraction of the way along the mark *m*; we find the point *P* that's a corresponding fraction of the way along *M* (we'll call this a *proportional correspondence* between *m* and *M*). The edge *e* makes some angle *a* with the tangent vector to the mark *m* at *p*, and the vertex *v* is some distance *d* from the mark *m*. We construct a ray tangent to the body at *P* with angle *a* to the tangent to *M* at *P* (see Figure 11) and walk[3] a distance *d* in this direction; the resulting point is defined to be *f(v)*. If *v* is an endpoint of multiple root edges, this calculation is carried out for each edge and *f(v)* is defined to be the average of the result.[4]

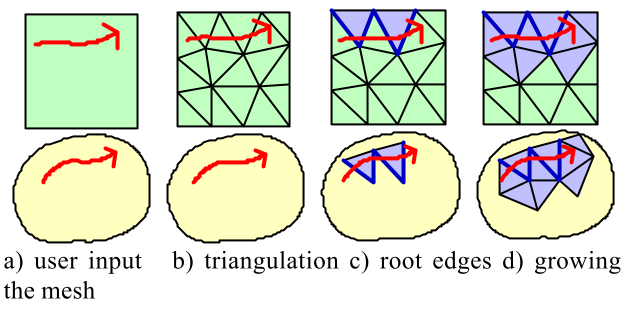


Figure 10: Overview of the wrapping process.

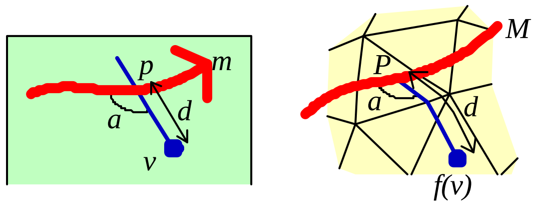


Figure 11: Mapping root-edge endpoints.

The remaining task is to grow the cloth mesh starting from the already mapped root edges (Figure 10d). For each triangle that is already *mapped,* we check whether the vertices around the triangle (i.e., the vertices of triangles that share an edge with this one) are already mapped or not. If a vertex is not mapped yet, the system marks it as *ready* (Figure 12) and places it in a priority queue with priority given by mesh distance to the nearest mark[5]. The system dequeues the lowest-priority vertex, maps it to the 3D body space (see below), and updates the queue based on the result. This process is repeated until all cloth vertices are mapped. We apply this procedure to the *merged* cloth mesh; sewn edges are treated as a single edge and the clothes grow across the sewn edge as usual.

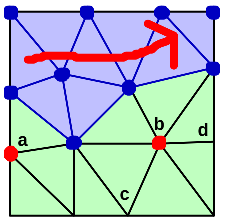


Figure 12: Growing process. Blue points and triangles are mapped vertices and triangles, red points are ready vertices. Vertex *a* has one parent triangle and *b* has two parents. Vertices *c* and *d* will become ready once *b* is mapped.

The position of a newly mapped vertex in the 3D space is calculated based on the triangles (which we call *parent triangles*) around the vertex that are already mapped. We'll describe the computation done for each parent triangle; the final value is the average of the results.

Let *P* be the parent triangle, sharing an edge *e* with another triangle *T* whose other vertex *v* needs to be mapped. Because the vertices of *P* are already mapped, there's a plane *H* in 3-space that contains *f*(*P*). Consider the segment *f*(*e*) in the plane *H* oriented so that *f*(*T*) lies to its right (Figure 13). The basepoint of an altitude from *v* to *e* lies somewhere along the line containing *e*; the distance from *v* to this point is some number *d*. Find (using a proportional mapping) the corresponding point on the line containing *f*(*e*), and go a distance *d* to the *right* of the directed segment to find the point *f(v)* where the vertex *v* is mapped relative to this parent.

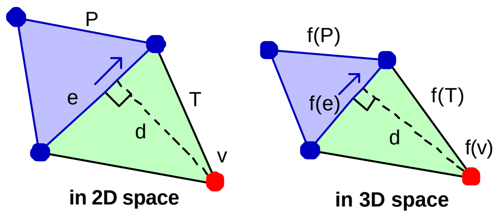


Figure 13: Calculating the position of a newly mapped vertex based on a parent triangle. The vertex is placed on the plane that contains the parent triangle in the 3D space.

[3] The "walk" in a given direction is found by traversing the mesh, as shown in Figure 11.

[4] The average is computed in space and the skin algorithm's surface tracking is used to find a closest surface point to this result.

[5] The *mesh distance*, computed by finding the shortest sequence of edges between two points, is used instead of geodesic distance because it can be computed quickly.

After each vertex is mapped, we apply a relaxation process (described below) that tries to keep the edge lengths of the already mapped mesh close to the corresponding "rest" lengths and prevents flipping of triangles on the body (i.e., tries to ensure that the map is orientation-preserving).

This growing algorithm extends the clothes using the already mapped vertices as guide and ignoring the body as long as the cloth does not collide with the body (the relaxation step detects and fixes such intersections). An alternative is to grow the cloth using the body as guide (see Figure 14). We experimented with this, but rejected it because of undesirable artifacts such as that shown in Figure 15. In general, body-guiding tends to create visually distracting folding and overlapping that are difficult for relaxation process to fix.

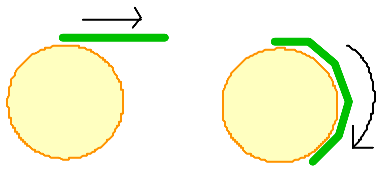
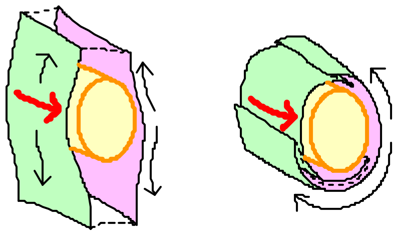


Figure 14: Two possible approaches for growing: current approach (left) and growing-on-the-body approach (right).



a) current approach b) growing-on-the-body approach

Figure 15: Putting a loose sleeve around an arm. The growing-on-the-body approach causes undesirable folds.

Surface Dragging

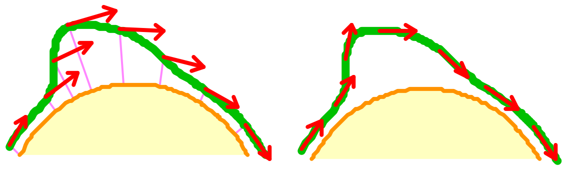
As discussed before, surface dragging explicitly propagates the dragging effect across the cloth vertices (Figure 7). At the beginning of a dragging operation, the system constructs a dependency graph whose root is near the click-point. Then as the user drags the grabbed vertex the system propagates the motion vector across the cloth according to the dependency graph. The system inserts a relaxation step after each dragging step.

Dependency graphs for surface dragging look similar to those for wrapping, but have two major differences. First, the dependency graph for surface dragging starts from a single root *vertex*, while that for wrapping starts from multiple root *edges*. Second, each vertex is dependent on multiple *parent vertices* in surface dragging, while each vertex is dependent on *parent triangles* in wrapping.

We build the dependency graph incrementally: we start with the grabbed vertex as the root, and insert its neighbors in a priority queue, with priority given by the distance to the root. Distances are calculated based on the edges' target rest lengths (i.e., the lengths of the corresponding edges in the cloth mesh). Now vertices are extracted from the priority queue and processed until the queue is empty. To process a vertex *v*, we first insert it into the graph and then examine its neighbors: if the neighbor vertex *n* is already in the graph, we add a directed edge from *n* to *v*. If not, we add the length of the edge *nv* to the priority of *v* to get a priority for *n*, which we insert in the queue. This process generates a directed acyclic graph of vertices with the grabbed vertex as the root.

We now describe how to propagate a motion vector from the root node to all other nodes. Just as in the wrapping algorithm, we compute a motion vector for each vertex from the motion vector for each of its parent vertices and then average the results to get the true motion vector (which may be zero).

There are two ways to propagate the motion over the clothing, one based on the body geometry and the other based on the cloth geometry. Figure 16 illustrates the difference between the two. The first approach works better when the clothes are close to the body surface, but causes undesirable motion when the clothes are far from the body. The second approach works better when the clothes are away from the body, but can be unstable because of its recursive nature, especially if significant wrinkles are present. Our current implementation uses the first approach because of its stability. In addition, the system slightly pulls the clothes near the body towards the body at each surface dragging step to make it stable (currently, a cloth vertex moves towards the nearest body surface so as to halve the distance when the distance is less than 0.036).



a) along body geometry b) along cloth geometry

Figure 16: Two possible approaches to surface dragging. Our current implementation uses the first one.

We now describe how to compute a child vertex's motion vector from that of its parent vertex. We first define a local coordinate system for each vertex. We use the normal vector of the corresponding body surface as the *z*-axis; we let $\vec{u}$ be a unit vector along the directed edge from the parent to the child, and use $\vec{u}-(\vec{u}\cdot\vec{z})\vec{z}$ as the direction of the *x*-axis and the cross product of the two as the *y*-axis. The motion vector for the child vertex is defined as the parent's motion vector mapped from the parent's coordinate system to the child's.

Pushpins

Pushpins provide additional control for surface dragging. A naïve approach to implementing them is simply to fix the pinned vertex and move the other vertices normally, but this generates large distortions around the pinned vertex (Figure 17).

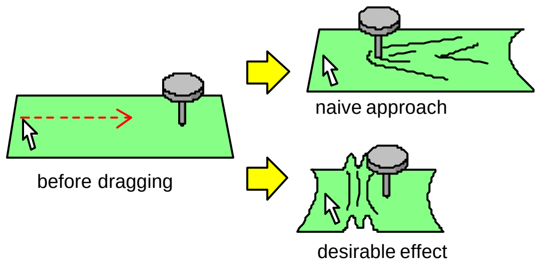
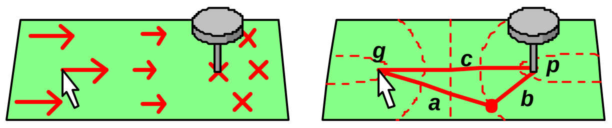


Figure 17: Pushpin effect. Naïve approach causes distortion.

To obtain the desirable effect in Figure 17, we attenuate the dragging vectors at the cloth vertices near the pushpin and diminish them on the other side of the pushpin (Figure 18a). This is done by calculating an *attenuation ratio* for each cloth vertex at the beginning of surface dragging; for each cloth vertex *v*, the system computes the mesh distance *a* to the grabbed vertex *g*, and the mesh distance *b* to the pinned vertex *p* (see Figure 18b). The system also computes the distance *c* between *g* and *p*.



a) attenuation of vectors b) calculation of distances

Figure 18: Calculating the attenuation ratio.

Given these distances, the attenuation ratio for the vertex is defined as

$$\begin{array}{lll} 1 & \text{if} & a-b=-c, \\ (c-a+b)/2c & \text{if} & -c<a-b<c, \\ 0 & \text{if} & c=a-b \end{array}$$

where 1 means full motion and 0 means no motion.

If multiple pushpins are used, the system calculates the attenuation ratio for each pushpin and uses their minimum[6]. The user can conveniently block the surface dragging effect by putting in a few pushpins in a row.

Pushpins are also important in controlling rewrapping (see Figure 4). Rewrapping first removes the cloth triangles from the 3D scene and then pastes them back around newly placed marks. But the removal of triangles is blocked by the pushpins – the system does not move vertices whose distance from the mark is greater than the distance between the mark and the pushpins.

[6] It is possible to use a blend function or the product of pin attenuations, but our simple approach shows satisfying results and we opt for the simplicity.

Keeping Clothes on the Body

We now describe the algorithms for maintaining basic cloth constraints during wrapping and surface dragging. This section describes the algorithm for handling cloth-body collision, and the next section describes the algorithm for preventing excessive stretching and folding.

Collision detection is the most time-consuming part in cloth simulation in general [14,23]. In addition, exact collision detection can impede placing cloth in the intended position. To achieve real-time operation, we ignore cloth-cloth collision and handle cloth-body collision in a limited way, by simply preventing cloth vertices from sinking into the body at each step and ignoring collisions between cloth edge and body edge. The system also ignores possible collisions during transitions. This simplified strategy obviously exhibits flaws in some situations, but it is fast and works well for our purpose.

To detect collisions between a cloth vertex and the body surface efficiently, the system keeps track of the nearest point on the body surface (*track point*) for each cloth vertex (this is the strategy used in the skin algorithm [19] for tracking the nearest skeleton surface for each skin vertex). Whenever a cloth vertex is moved, the system updates its track point by locally searching the body surface (Figure 19 left). Given the track point, detecting collision is a straightforward. If the cloth vertex is inside the body surface, the system pushes the cloth vertex back to above the body surface (Figure 19 right). The system actually keeps the cloth vertices a bit away from the body so that cloth edges do not penetrates the body surface (the current offset is 0.012). A vertex also shares information with its immediate neighbors so as to jump from a local solution to a distant solution (Figure 20). This migration feature is important when a garment spans separate body regions such as an arm and a torso.

This simple approach cannot detect collisions with body parts approaching from above or with separate body parts that were not covered by the cloth before. This causes no practical problems in our experience, but the current simple approach must be extended to handle more complex cases (see "Additional Algorithm Details" section).

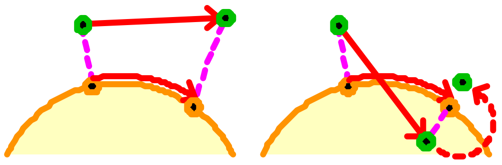


Figure 19: Each cloth vertex is associated with the nearest body surface. A vertex inside of the body is pushed back to the body surface.

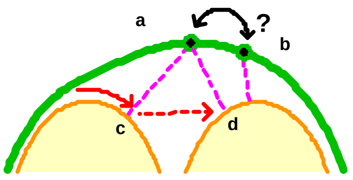


Figure 20: Cloth vertices share information with their neighbors. This enables vertex *a* to find the true solution *d* instead of being stuck with local solution *c*.

Relaxation Steps

Relaxation steps are inserted during wrapping and surface dragging to keep the clothes visually plausible by preventing excessive stretch and folds. Note that the purpose of this relaxation step is to move the cloth towards a class of desirable static configurations. Our goal is to add useful behavior to cloth so as to help the user put clothes on characters, *not* to mimic physically realistic behavior. For example, our relaxation steps automatically unfold flipped clothes, which does not happen in the real world.

A relaxation step has four parts. First, we try to make each edge's length closer to its rest length to prevent stretch and shrinkage. Second, we try to recover flipped triangles to prevent folds. Third, we try to flatten the cloth at each edge of triangle; this corresponds to a dihedral-angle spring and helps generate attractive wrinkles. Finally, we mimic the effects of gravity and friction.

Preventing stretch and flip The system addresses the first two goals simultaneously by adjusting vertex positions so that each triangle *T* recovers its rest shape (called the *reference triangle*) on the body surface. The reference triangle is uniquely defined by the rest length of the edges. The system places a copy *U* of the reference triangle as close to *T* as possible, and moves each vertex of *T* towards its corresponding vertex in this copy of *U* (Figure 21). *U* is placed in a plane (described below) with the centers of gravity, *O* and *O'*, of *T* and *U* aligned, and is rotated as follows. The system computes *B''* by rotating *B* by *ÐB'OA'* around *O*, and computes *C''* by rotating *C* by *ÐC'OA'* around *O*. The system rotates *U* so that $\overline{OA'}$ parallels $\overline{OA}+\overline{OB''}+\overline{OC''}$. A similar technique is used in automatic texture coordinate optimization [18].

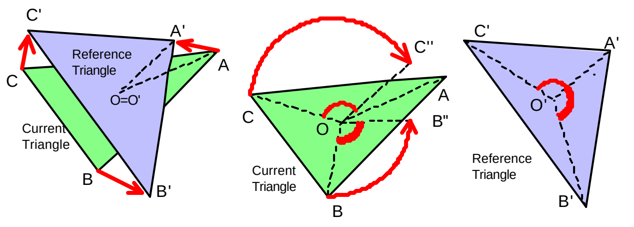
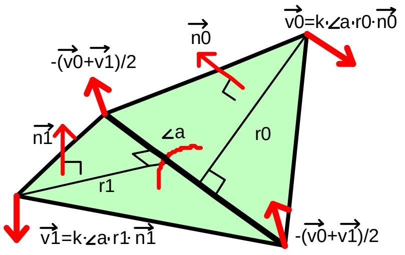


Figure 21: Matching a triangle and its reference triangle.

We've found that this triangle-based strategy works faster than an edge-based strategy (e.g. [8]) and generates better results for our purpose. In addition, it automatically recovers flipped triangles if we place the reference triangle front-face up. "Face up" is determined by a "temporary normal vector." The temporary normal is the body surface normal when the cloth is near the body surface (distance < 0.012), but is the cloth surface normal when the cloth is far from the body (distance > 0.08); the normals are blended in the intermediate region. The triangle-based relaxation is done on the plane perpendicular to this temporary normal: the system projects *T* to that plane, applies the above method above, and then moves the vertices according to the resulting vectors (which are parallel to the plane).

Flattening the cloth We flatten the cloth by moving vertices so as to make the dihedral angle at each edge closer to 180 degrees. We compute the vectors shown in Figure 22 for the four vertices associated with each edge; the sum of these is then applied to the vertices. This corresponds to the dihedral-angle spring found in typical cloth simulations [2].


$\vec{v0}$=k·∠a·r0·$\vec{n0}$
$\vec{n0}$
-($\vec{v0}$+$\vec{v1}$)/2
$\vec{n1}$
∠a
r0
r1
$\vec{v1}$=k·∠a·r1·$\vec{n1}$
-($\vec{v0}$+$\vec{v1}$)/2
Figure 22: Each edge on a ridge moves four adjacent vertices to become flat.

Gravity and friction To mimic the effects of gravity, we move each cloth vertex downward by a predefined amount unless it collides with the body surface (i.e., we make clothes fall at a constant speed). Friction is mimicked by not allowing any vertex to be moved if 1) the vertex is in contact with the body surface, 2) the requested motion vector heads downwards with respect to the underlying body surface, and 3) the requested motion vector is smaller than a predefined threshold.

ADDITIONAL ALGORITHM DETAILS

This section describes some further implementation details. The features described are optional: one can manipulate clothes reasonably with the basic algorithms alone, but these features help make the system robust and improve the user experience.

Adaptive Subdivision

As discussed in the previous section, we prevent the vertices from sinking into the body but do not prevent edges from sinking into the body. This works well when the underlying body surface is reasonably flat, but causes serious problems for high-curvature regions such as arms and legs. The body surface appears on top of the clothes and is very distracting.

This is essentially an aliasing problem, and our solution is to adaptively change mesh resolution according to the curvature of the body surface. A cloth edge is automatically split when it intersects the body, and restored when it no longer intersects the body (the original edge no longer exists in the mesh if it is split, but the system remembers the original edge information). We use the $\sqrt{3}$ subdivision scheme [16] because it allows edgewise split/merge and generates reasonable mesh patterns (Figure 23). Our current implementation allows only one subdivision step for simplicity, and this hides most problems sufficiently.

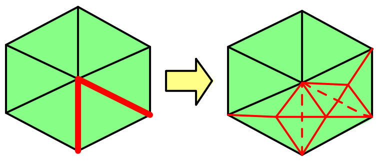


Figure 23: Splitting two edges with $\sqrt{3}$ subdivision.

We use Figure 24 to describe how the system decides whether to split an edge. Here, the system needs to know whether edge *AB* intersects the body. It is too expensive to do precise collision detection by traversing the body surface, so the system performs an approximate computation using local information, that is, the locations of *A*, *B*, and *P*, where *P* is the nearest point on the body to *A*, which is always available from the "skin" algorithm. It is obviously not possible to detect actual collision, so we approximate it by testing collision with a sphere of radius L = length(AB) that is tangent to the surface at *P*. The test is approximated by $(d + L)\sin q < L$ and $(d + L)\cos q < L$; if both inequalities hold, we split the edge *AB*. The justification for using L as the body's local radius is that it provides a minimum radius that we must worry about. If the body radius is actually larger than L, the "vertex is always on the body surface" constraint approximately guarantees that the edge does not sink below the surface. The system performs the equivalent test at *B* as well.

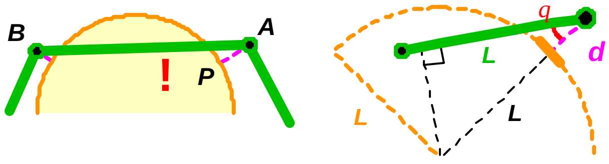


Figure 24: Testing an edge for collision with an approximating sphere.

Collision Detection with Bones

Adaptive subdivision effectively prevents most edge-to-edge penetration, but excessive movement can cause the clothes to penetrate the body. We can ignore small amounts of "sinking" because the relaxation process gradually recovers from the error, but the system cannot recover from significant topological errors such as the cloth penetrating the body all the way from one end to the other. This happens typically where thin parts such as a neck or an arm stick out from the body (Figure 25 center). To prevent this, we implemented collision detection against simple "bone structures" (Figure 25 right). A *bone* is a simple edge defined by two end points, and collision with all bones is checked whenever a cloth edge is moved. If a collision occurs, the system pushes the cloth edge back to prevent penetration. For the human body in Figure 2, we used six bones.

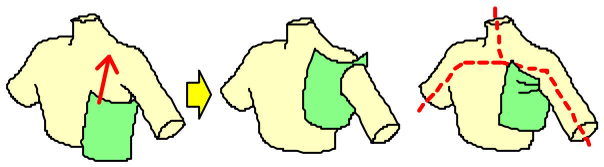


Figure 25: Collision detection against simple bones.

IMPLEMENTATION AND RESULTS

The current prototype system is implemented in Java™ (JDK1.4), and uses directX7 for 3D rendering. Figure 26 shows some clothing designed using the system. The clothes have a few hundred triangles and the system maintains reasonable frame rates during surface dragging on a high-end PC (AMD Athron™ 1.5GHz).

We have begun an informal user study. It took approximately 20 minutes before a user started using the system fluently under our supervision. The last image in Figure 26 was created by the test user. It took a while for the user to learn the peculiar behavior of the clothes in our system. The user tended to drag the clothes long way in a single interaction, making the system unstable; clothes must be moved gradually towards the goal position instead. It's also necessary to release the mouse occasionally during the dragging so that relaxation steps can dissolve the accumulated distortion. The user also had difficulty in designing the clothes of an appropriate size. It would be helpful if one could adjust the size of the clothes after putting them on the characters.

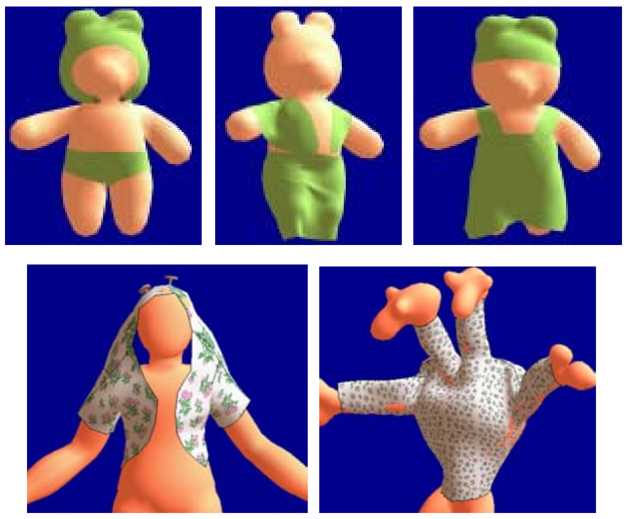


Figure 26: 3D characters in various clothes.

LIMITATIONS AND FUTURE WORK

Our current system has several limitations: our techniques are designed specifically for clothing a character and not for manipulating clothes away from a body. We support only a single layer of clothes on a body, although we plan to extend the system to support multilayer clothes. We also plan to support explicit folding of clothes (e.g., collars). But to support these, we need to track the nearest object on top of each cloth vertex as well as the nearest object under the vertex.

Cloth-cloth collision is ignored in the current implementation. Although we believe that this is a reasonable decision given current processor performance, we need to incorporate cloth-cloth collision detection in the future. That will let us explore more interesting cloth-manipulation techniques such as tying a tie.

Wrinkles are an important part of clothes design [13]. We plan to develop interaction techniques for explicitly placing wrinkles on clothes. For example, it might be useful for the system to automatically adjust global cloth configuration so that wrinkles appear where the user paints freeform marks.

Some basic interface improvements would be very useful: An obvious extension is to let a user edit the clothes in 2D and 3D space simultaneously [8]. We are considering several operations such as cutting, stitching, and resizing. And our dragging operation should probably interleave "relaxation steps" during long drags.

We believe it is reasonably easy to incorporate our cloth-manipulation techniques into existing 3D graphics systems because we use standard triangular mesh structures for the cloth and the body. A potential difficulty is finding appropriate values for the many ad hoc parameters in our algorithms (the current values are chosen as the result of many experiments). They must be carefully adjusted according to object geometry and the user input.

ACKNOWLEDGEMENTS

We would like to thank the Brown University computer graphics group for thoughtful discussions, and the CMU stage3 research group, especially Dennis Cosgrove, for allowing us to use their Jalice scenegraph. We also thank reviewers for their constructive critique.

REFERENCES

1. Asahi AGMS, www.agms.co.jp
2. D. Baraff and A. Witkin. Large steps in cloth simulation. *SIGGRAPH 98 Conference Proceedings*, pages 43-4, 1998.
3. D. Baraff, PIXAR. Personal communication, 2001.
4. T. Beier and S. Neely. Feature-based image metamorphosis. *SIGGRAPH 92 Conference Proceedings*, pages 35-42, 1992.
5. D.E. Breen, D.H. House, and M.J. Wozny. Predicting the drape of woven cloth using interacting particles. *SIGGRAPH 94 Conference Proceedings*, pages 365-72, 1994.
6. M. Carignan, Y. Yang, N. Magnenat-Thalmann, and D. Thalmann. Dressing animated synthetic actors with complex deformable clothes. *SIGGRAPH 92 Conference Proceedings,* pages 99-104, 1992.
7. F. Dachille IX, J. El-Sana, H. Qin and Arie E. Kaufman. Haptic sculpting of dynamic surfaces. *Proc. of Interactive 3D Graphics 1999*, pages 103-110, 1999.
8. M. Desbrun, P. Schroder, and A. Barr. Interactive animation of structured deformable objects. *Proc. of Graphics Interface '99*, pages 1-8, 1999.
9. DressingSim, www.dressingsim.com
10. Gerber Technologies, www.gerbertechnology.com
11. T. Igarashi and D. Cosgrove. Adaptive unwrapping for interactive texture painting. *Proc. of Interactive 3D Graphics 2001*, pages 209-216, 2001.
12. T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: A sketching interface for 3D freeform design. *SIGGRAPH 99 Conference Proceedings*, pages 409-416, 1999.
13. S. Hadap, E. Bangerter, P. Volino, N. Magnenat-Thalmann. Animating wrinkles on clothes. *Proc. of the Conference of Visualization '99*, pages 175-182, 1999.
14. D. House and D. Breen. *Cloth Modeling and Animation.* AK Peters, 2000.
15. J. Hultquist. A virtual trackball. *Graphics Gems* (ed. A. Glassner). Academic Press, pages 462-463, 1990.
16. L. Kobbelt, $\sqrt{3}$ subdivision, *SIGGRAPH 2000 Conference Proceedings*, pages 103-112, 2000.
17. Lectra, www.lectra.com
18. H. Malan, Righthemisphere Inc. Personal communication, 2001.
19. L. Markosian, J.M. Cohen, T. Crulli and J.F. Hughes. Skin: a constructive approach to modeling free-form shapes. *SIGGRAPH 99 Conference Proceedings*, pages 393-400, 1999.
20. Maya Cloth, www.aliaswavefront.com
21. J.R. Shewchuk. Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. *First Workshop on Applied Comp. Geometry Proc.*, pages 124-133, 1996.
22. 3ds MAX, www.ktx.com
23. P. Volino, N. Magnenat-Thalmann, *Virtual Clothing Theory and Practice*, Springer-Verlag, 2000.
24. J. Weil. The synthesis of cloth objects. *Computer Graphics (Proc. of SIGGRAPH)*, Vol. 20, No. 4, pages 49-53, 1986.
25. R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. SKETCH: an interface for sketching 3d scenes. *SIGGRAPH 96 Conference Proceedings*, pages 163-170, 1996.

Sketching for Mechanical Design and CAD

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Computers & Graphics 29 (2005) 563–575

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

A freehand sketching interface for progressive construction of 3D objects☆

M. Masry*, D. Kang[1], H. Lipson

Sibely School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY 14853, USA

Abstract

This paper presents an intuitive, freehand sketching application for Computer Aided Design (CAD) that can reconstruct a 3D object from a single, flat, freehand sketch. A pen is used to draw 2D sketches consisting of straight and curved strokes connected at vertices. The sketches are processed by a reconstruction algorithm that uses the angular distribution of the strokes and their connectivity to determine an orthogonal 3D axis system whose projection correlates with the observed stroke orientations. The axis system is used to determine a plausible depth for each vertex. This approach works well for drawings of objects whose edges predominantly conform to some overall orthogonal axis system. A second, independent optimization procedure is then used to reconstruct each curved stroke in the original sketch, assuming that the curve is planar. New strokes can be attached to the 3D object, or drawn directly onto the object's faces. An implementation of the reconstruction algorithm based on Levenberg–Marquardt optimization allows objects with over 50 strokes to be reconstructed in interactive time.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: 3D sketching; Pen-based computing; Optimization; Human–computer interaction; Computer graphics

1. Introduction

Visual methods of communication are often the simplest and most efficient way of conveying information about the shape, composition and relationships of an object's components. Furthermore, visual information often transcends the limitations imposed by spoken or written languages and is necessary in engineering: a major portion of engineering information is conceived, recorded and transmitted in a visual, nonverbal language [1]. In spite of this, little work has been done to create fast, intuitive sketch-based computer aided design (CAD) interfaces for engineers and designers. Conventional CAD user interfaces are typically cumbersome to use and hamper creative flow.

Freehand sketching, the informal drawing of shapes using freeform lines and curves, has remained one of the most powerful and intuitive tools used at the conceptual design stage. Sketches, in contrast to typical Computer Aided Designs, can quickly and easily be created to convey shape information. Simple paper-based sketching also has many drawbacks: the viewpoint is fixed and cannot be changed in mid drawing; the sketch is passive and cannot be directly simulated or analyzed using computational engineering tools (e.g. structural analysis or kinematic simulation); the sketch is tentative and if a

☆This research has been supported in part by US National Science Foundation (NSF) Grant IIS-0428133 and by a Microsoft University Relations research grant for Tablet Computing.

*Corresponding author. Tel.: +1 607 255 0396.
E-mail addresses: mark.masry@cornell.edu (M. Masry), hod.lipson@cornell.edu (H. Lipson).

[1]D. Kang is now with Tongmyong University of IT. Busan, Korea.

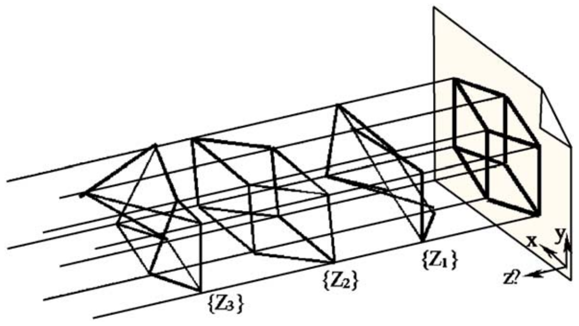


Fig. 1. A user creating, rendering and rotating a shape with a see through hole using the proposed system on a Tablet PC.

final, accurate model is desired, it must be recreated from scratch. The ideal solution from a designer's point of view should combine both the speed and ease of freehand sketching with the flexibility and analytical abilities of CAD tools.

This paper presents an intuitive, pen-based sketching tool that can reconstruct a 3D object from a single, flat, freehand sketch without relying on a database of existing models. As shown in Fig. 1, a user can make an initial sketch, reconstruct it, and add detail using a consistent sketching interface. The proposed system can reconstruct sketches consisting of both straight lines and planar curves. A series of optimization-based reconstruction algorithms are used to achieve this goal. The optimization algorithms run in interactive time on complex sketches, providing a seamless interface for the construction and refinement of 3D objects.

2. Previous work

Systems that use sketch-based input have been the focus of much research. Stahovich et al. [2] demonstrated a system that could interpret the causal functionalities of a 2D mechanism depicted in a sketch, and generate alternative designs. Davis [3] recently showed a system that simulated rigid-body dynamics of a sketched 2D mechanism. These systems are largely 2D.

Fig. 2 outlines the reconstruction of a 3D object from a 2D sketch, in which any arbitrary set of depths $\{Z\}$ that are assigned to the vertices in the sketch constitutes a 3D configuration whose projection will match the given sketch exactly. In principle, each such assignment yields a valid candidate 3D reconstruction. A considerable amount of research has focused on the reconstruction of polyhedral objects from straight-line sketches. Line labeling approaches [4,5] classify each line as convex, concave or occluding edge without explicitly reconstructing 3D shapes. Several methods construct relationships between the slope of sketch lines and the gradients of the associated 3D faces in an attempt to constrain the number of possible interpretations [6,7].

Other methods construct 3D objects incrementally by attaching facets sketched by the user in 2D [8,9]. A gesture-based system for interactively constructing 3D rectilinear models was proposed by Zeleznik et al. [10]. Other approaches to the reconstruction problem require the assumption that the 3D elements in a scene are specified entirely by known primitives [11]. Though restrictive, this allows the reconstructed scene to be specified with a convenient solid geometry.

Optimization-based reconstruction determines the depth assigned to the sketch vertices by optimizing a target function. These methods are more general than the approaches above and can be used to reconstruct relatively complex 3D objects. Optimization-based approaches characterize the relationship of a 2D sketch to an underlying 3D object using systems of linear equations for which the existence of solutions is a sufficient criterion for reconstruction. Linear programming optimization techniques may provide these solutions [12,13]. Another approach is taken by Lipson and Shpitlani [14]: 2D sketches are converted to line and vertex graphs, which are analyzed for regularities such as parallelism, perpendicularity and symmetry. Regularities in the 2D sketch plane are then weighted according to the probability that they correspond to 3D geometrical relationships, and summed to produce an overall compliance function that estimates how well the 3D construction conforms to the regularities in the 2D sketch. Reconstruction proceeds by optimizing this compliance function. There are also statistical approaches to optimization-based reconstruction [15,16]. The correlation between the 2D angles formed by lines in the sketch plane and the angle between these lines in 3D space are learned from a large number of computer-generated 3D shapes and the corresponding projections of these shapes onto a viewing plane. These 2D–3D geometric correlations are then used to determine the most likely 3D shape corresponding to a set of 2D angles, by optimizing over possible assignments of depth values.

While flexible, optimization-based methods suffer because the optimization surface itself may contain many local minima that make it difficult to find the global minimum, while the computational complexity of the optimization process grows rapidly in the number of vertices and lines in a sketch. In contrast to the number of approaches for reconstructing polyhedral objects specified by straight line sketches, there are relatively few reconstruction algorithms that can be applied to sketches of 3D objects with curves. The best-known such work is the Teddy system proposed by Igarashi et al. [17], which uses a sketch-based interface to specify the

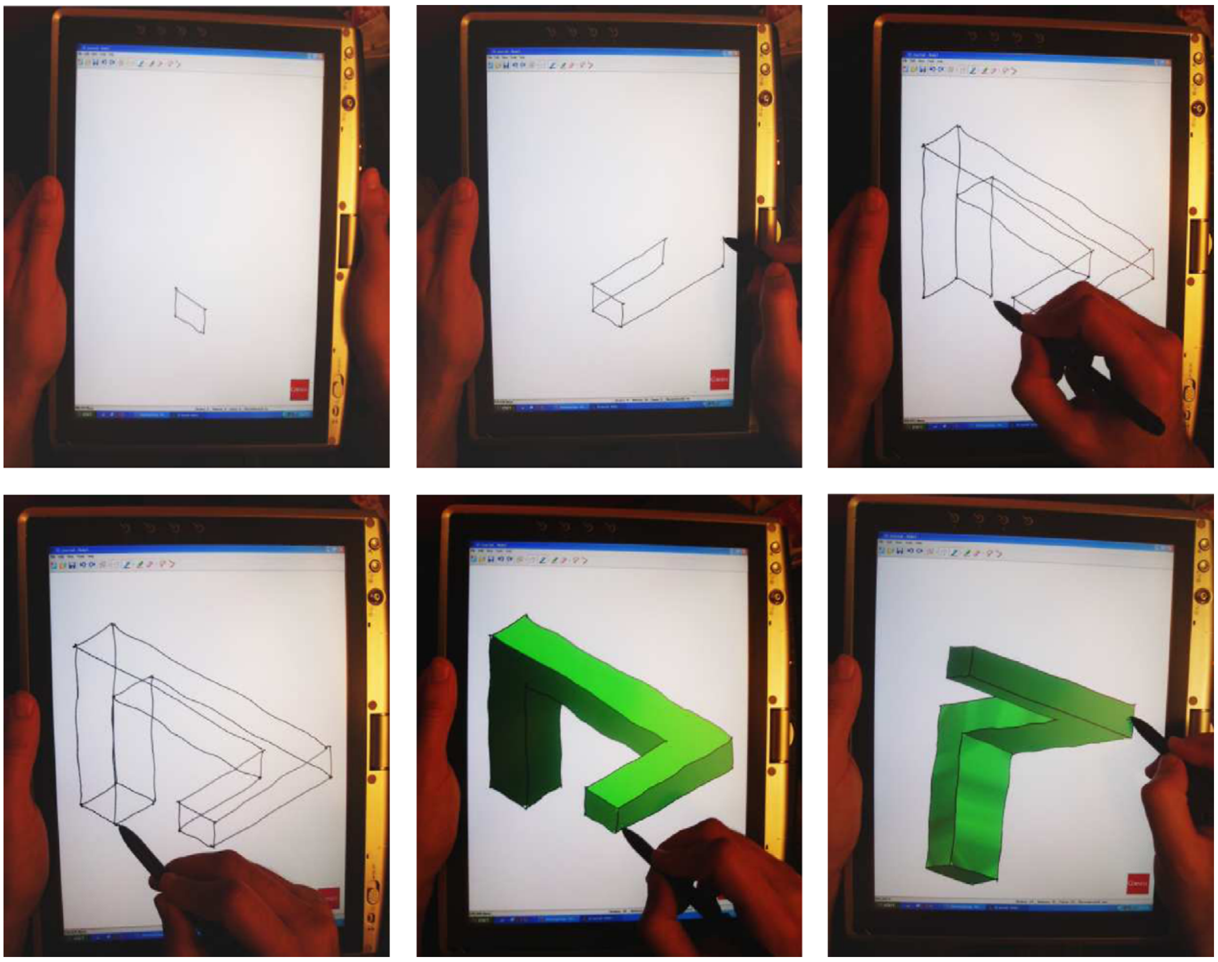


Fig. 2. A sketch provides only two of the coordinates (x, y) of object vertices. A 3D reconstruction must recover the unknown depth coordinate z. In parallel projections, this degree of freedom is perpendicular to the sketch plane; there are an infinite number of candidate objects—the problem is indeterminate. Each candidate object is represented by a unique set of Z coordinates, e.g. sets $\{Z_1\}$, $\{Z_2\}$ and $\{Z_3\}$.

boundaries for reconstruction of a curved solid. This system cannot be used to reconstruct polyhedral objects, or objects that mix straight lines and curves.

The 3D sketching system proposed in this paper uses a fast, optimization-based reconstruction algorithm that chooses a plausible three-connected sketch vertex to serve as a 3D axis origin based on the angular distribution of the lines in a sketch, and reconstructs the depths of the three vertices at the opposite ends of the attached strokes. Depths are then assigned to the other sketch vertices by propagation across the connectivity graph given by the sketch. This approach allows the reconstruction of 3D objects with a connectivity graph whose edges conform to an underlying, orthogonal axis system. Following reconstruction of the sketch vertices, a second optimization procedure reconstructs each curved stroke.

3. 3D sketching system

The sketching system attempts to create an experience similar to drawing with pencil and paper. The application was implemented using the Microsoft Tablet PC API. An example session is shown in Fig. 2. The system allows users to make an initial sketch by drawing strokes using the pen, reconstruct it, and subsequently add new strokes. Following the initial reconstruction, the sketch can be rotated, rescaled or resized. Each stroke is treated as an independent object, and can be erased or modified by the user either pre- or post-reconstruction.

The user interface relies entirely on the pen. A session begins with an initial sketch specified by a set of loosely connected strokes in the sketch plane given by the digitizer surface. Each potentially curved stroke is assumed to be piecewise linear, and is represented internally by the location of its two endpoints and a series of values specifying the location of each point along the length of the stroke.

Each new stroke is split into smaller strokes if one or more corners are detected using the methods proposed by Shpitlani and Lipson [18]. Strokes may intersect in the sketch plane, but these intersections are not taken to represent intersections in 3D space; at this stage, strokes may be joined only at the endpoints. The reconstruction process is triggered by pressing on the pen's barrel button. As a first step toward reconstruction, all stroke endpoints within a specified distance of one another are connected using an approach given by Shpitlani and Lipson [18]. The 2D sketch can now be interpreted as a connectivity graph (or straight-line graph) representing the 2D orthographic projection of a 3D object onto the plane $z = 0$, with vertices given by the connections between strokes and edges specified by the straight line connections between vertices.

The reconstruction process first determines the 3D position of all sketch vertices, while all curved strokes are treated as straight line connections between vertices, after which all points along each curve are reconstructed. The system then identifies circuits in the connectivity graph and constructs triangulated faces for each circuit. The reconstruction algorithms run in interactive time, allowing for a fluid interaction with the system. The reconstructed shape can be rotated and resized by dragging using the pen.

Strokes can be added, deleted or partially erased at any time. If a new stroke's endpoint is near a reconstructed 3D feature (vertex, stroke or face), its position is automatically interpolated from the 3D object. Strokes sketched directly onto a planar face are automatically reconstructed by interpolating the position of the stroke's points from the face. A stroke deletion or erasure will automatically cause removal of any faces containing the stroke.

The reconstruction algorithm and the methods by which subsequent strokes can be added to the sketch are described in the following sections.

4. Sketch reconstruction

Since the (x, y) coordinates of each vertex are given in the sketch, reconstructing a 3D object requires assigning a z coordinate (also termed the *depth* value) to each vertex, subject to constraints on the characteristics of the resulting 3D object. It is assumed that the sketch vertices are *connected* i.e. that a path can be constructed from each vertex to every other vertex. It is further assumed that none of the vertices or strokes in the sketch completely obscures other elements of the same kind. Though the reconstruction algorithm proposed in this paper requires that at least one vertex be connected to three strokes that represent projections of the 3D axis system, the algorithm can also be adapted to reconstruct an independent 3D axis system that is not directly associated with any vertex in the sketch.

The algorithm is intended to reconstruct 3D objects whose vertices can be connected by a spanning tree consisting of straight lines aligned with one of 3 orthogonal axes. Sketches consisting of connected planar curves can also be reconstructed, provided that the underlying straight line connectivity graph satisfies this requirement. Though these requirements are restrictive, this approach works well for drawings of objects whose edges predominantly conform to some overall orthogonal axis system, which includes a wide range of engineering design drawings. Objects without an underlying rectilinear frame cannot be reconstructed using this method, but this approach can still be to partially reconstruct the object before a more general optimization-based approach [14] is used to complete the reconstruction.

The reconstruction process proceeds as follows:

(1) The distribution of the 2D angles of all straight lines connecting sketch vertices is tested for the presence of three or more significant peaks. If these are found, the sketch is considered to have one or more underlying 3D axis systems, which are then identified.
(2) The identified 3D axis systems are reconstructed from their 2D projections in the sketch plane and used to reconstruct all sketch vertices.
(3) The points along each curved stroke in the original sketch are reconstructed using a separate reconstruction procedure, under the assumption that each stroke is planar.
(4) Connected circuits of approximately coplanar sketch vertices are identified and used to generate the object's faces.

Once the 3D object has been reconstructed, the user can rotate it, add additional strokes, or sketch directly onto the object's faces.

The reconstruction steps are described in more detail below.

4.1. Identifying axis systems

Since orthogonality is the prevailing trend in most engineering drawings, and the easiest to identify, a statistical analysis of the direction of lines in the sketch is performed to determine whether these are consistent with the projections from an underlying orthogonal axis system. The angular distribution graph (ADG) for a set of lines is a discrete histogram of the 2D angles of the lines relative to the sketch plane.

The ADG is constructed by taking each angle to be the mean of a Gaussian distribution with a fixed variance to reduce sensitivity to noise. The resulting sum of Gaussians is then sampled at 1° intervals to yield the discrete ADG. A discrete, rather than continuous, graph is used to facilitate correlation-based similarity measures. Peaks in the ADG show the prevailing sketch angles; the ADGs of most polyhedral 3D objects have clear peaks. The reconstructed object's axis system should thus have a spatial orientation such that it projects onto the sketch plane at angles corresponding to maxima in the ADG. Fig. 3 shows a 2D sketch and the ADG of the associated straight line sketch.

The local ADG of a vertex is the ADG of all lines attached to the vertex. The first step in the reconstruction process is to select a vertex whose local ADG is most similar to the ADG of a representative set of lines. The similarity between any two discrete ADGs is measured using linear correlation. The vertex whose local ADG has the highest correlation with the ADG of the representative set of lines is chosen to be the origin of an axis system. The three lines attached to this vertex represent the projection of the axes onto the sketch plane.

The depth of each vertex is determined by the projection of the connected lines onto the main axis system. Given that the sketch graph is connected, it is possible to construct a spanning tree that connects each vertex to the axis origin. Depth values are then propagated along this tree, beginning at the axis

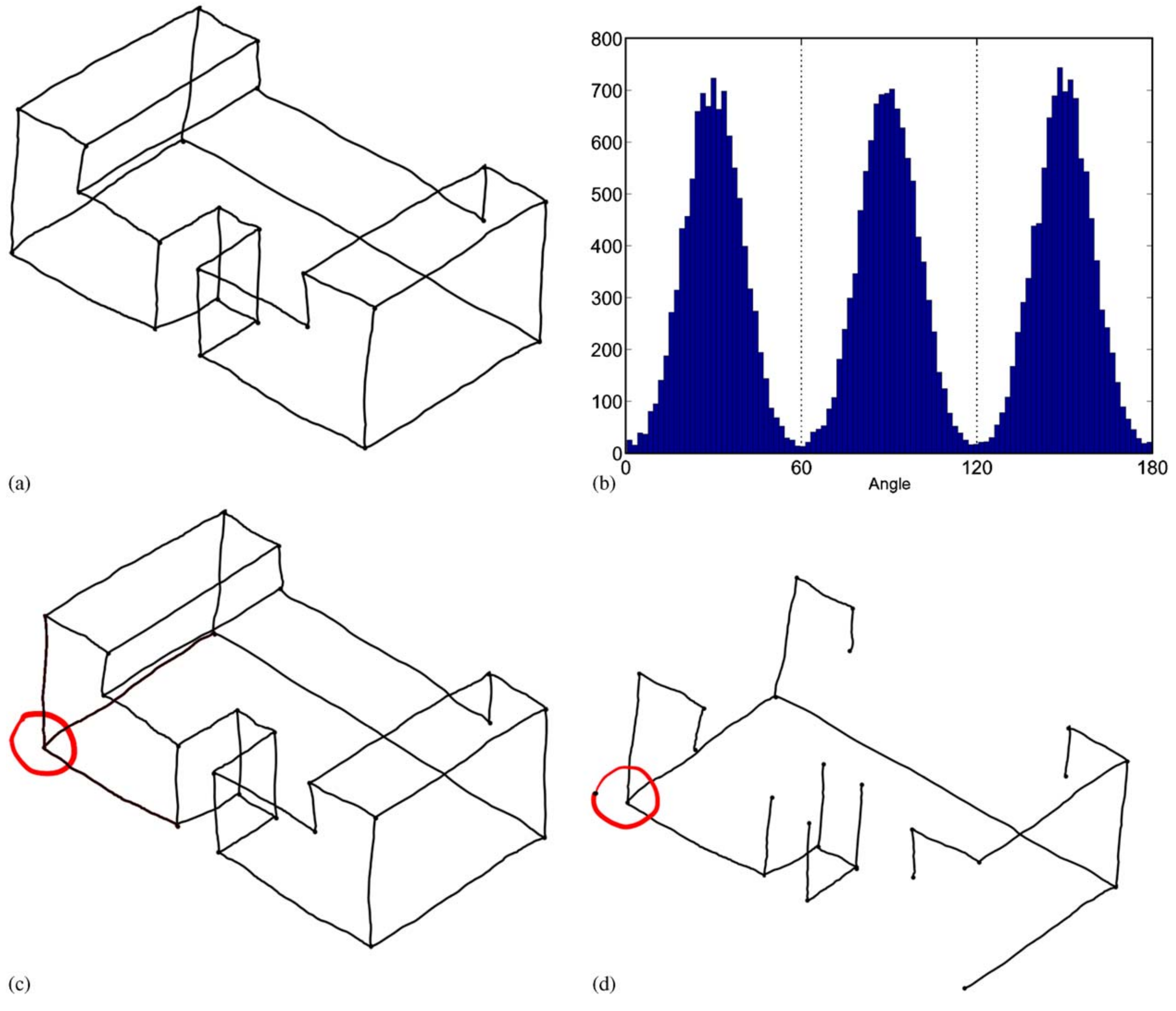


Fig. 3. (a) A 2D sketch with a single distinct axis system, (b) its angular distribution graph (ADG). (c) The sketch with the identified axis vertex circled in red, (d) the sketch's minimum spanning tree (MST) rooted at the selected vertex.

endpoints. The weight assigned to the 2D line direction vector $\mathbf{v}_n = [x_n, y_n]$ associated with line n is given by

$$\max_{\mathbf{v}_a \in \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}} \frac{|x_n x_a + y_n y_a|}{\sqrt{x_n^2 + y_n^2}\sqrt{x_a^2 + y_a^2}}, \tag{1}$$

where $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ is the set of 2D vectors that make up the axis system.

A maximum weight spanning tree (MST) is used to determine the propagation path from the main axis to each vertex. The MST is the tree that connects all sketch vertices such that the sum of the weights of all edges in the tree is maximized. The MST constructed with the weights given by Eq. (1) connects all of the vertices in the sketch while avoiding those lines that are not highly aligned with the main axes; using it to determine the depth values of each vertex therefore minimizes the propagation of reconstruction errors. The MST is determined using Prim's algorithm [19]. The algorithm begins using only the main axis vertex and the projected axes. The tree is then iteratively expanded by selecting the connected line with the highest weight.

The representative ADG and MST for the sketch are determined iteratively in order to minimize the effects of atypical lines using the following algorithm:

(1) Select all lines
(2) Build the ADG of selected lines
(3) Select an axis vertex using the ADG and assign line weights
(4) Generate the sketch MST and select the lines in the MST
(5) If this selection differs from the previous selection, goto step 2

It is assumed that a single MST can be constructed such that all vertices are connected by strokes that are aligned with a single set of 3D axes. If this process does not converge on a single MST after a several iterations, the 2D sketch cannot be reconstructed with the proposed method. An example MST is shown in Fig. 3.

The class of sketches with two distinct axis systems cannot be reconstructed in their entirety with the proposed algorithm. Fig. 4(a) shows a sketch with two axis systems. The sketch ADG shown in Fig. 4(b) has five distinct peaks, as opposed to the three found in the ADGs of sketches with a single axis system such as Fig. 3. If the sketch was drawn using two or more distinct axis systems, the MST construction process will not converge.

4.2. Reconstructing vertex depths

The origin of the main axis system is assumed to have a depth of zero. The depth component of the axis line vectors must be determined in order to reconstruct the main axis system. The x and y components of each axis are given by the vectors $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$. The unknown z components z_1, z_2, and z_3 are the values that minimize an optimization function based on two assumptions about an ideal sketch:

(1) Since the axis vectors are, ideally, perpendicular to one another in 3D space, the angle between any two axes should be 90° and the cosine of the angle should be 0.
(2) If the length of each line in the sketch plane corresponds to its length in 3D, the difference between the ratio of the axis lengths in the sketch plane and the ratio of their lengths in 3D space should be 0.

Note that the second assumption imposes restrictions on the sketch viewpoint: viewpoints where the orthographic

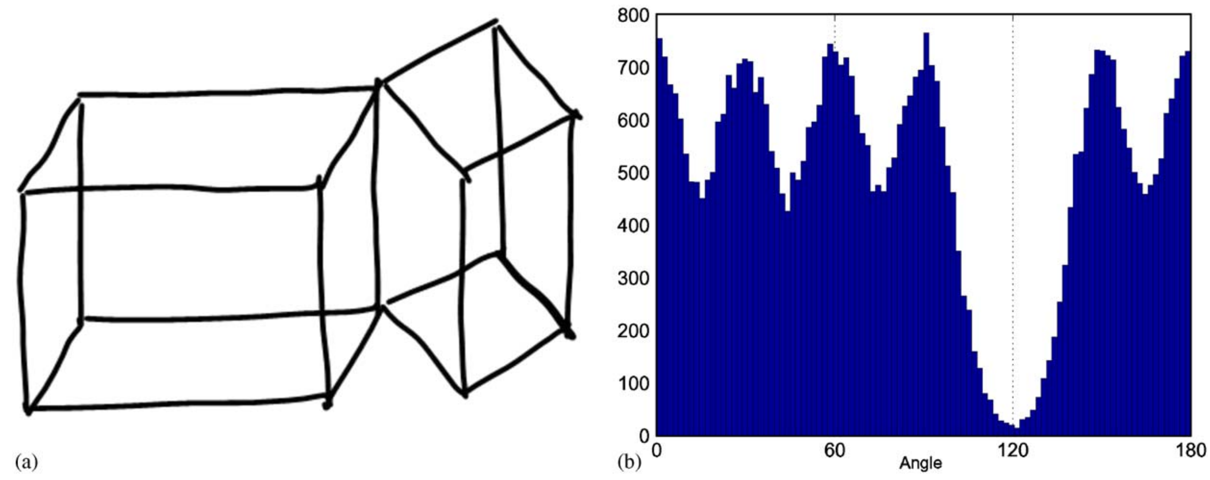


Fig. 4. (a) A 2D sketch with two distinct axis systems and (b) its angular distribution graph (ADG).

projection of the object onto the 2D sketch plane produces axes with very different lengths will result in reconstructed 3D axes with very different lengths.

The optimization goal is therefore to minimize the following cost function $f(z_1, z_2, z_3)$:

$$\begin{aligned} f(z_1,z_2,z_3) &= \cos^2\theta_{21} + \cos^2\theta_{32} + \cos^2\theta_{31} \\ &\quad + \omega\left(\left(r_{21} - \frac{|\mathbf{p}_2|}{|\mathbf{p}_1|}\right)^2 + \left(r_{32} - \frac{|\mathbf{p}_3|}{|\mathbf{p}_2|}\right)^2 \right. \\ &\quad \left. + \left(r_{31} - \frac{|\mathbf{p}_3|}{|\mathbf{p}_1|}\right)^2\right), \\ &= \left(\frac{\mathbf{p}_1\cdot\mathbf{p}_2}{|\mathbf{p}_1||\mathbf{p}_2|}\right)^2 + \left(\frac{\mathbf{p}_3\cdot\mathbf{p}_2}{|\mathbf{p}_3||\mathbf{p}_2|}\right)^2 + \left(\frac{\mathbf{p}_1\cdot\mathbf{p}_3}{|\mathbf{p}_1||\mathbf{p}_3|}\right)^2 \\ &\quad + \omega\left(\left(r_{21} - \frac{|\mathbf{p}_2|}{|\mathbf{p}_1|}\right)^2 + \left(r_{32} - \frac{|\mathbf{p}_3|}{|\mathbf{p}_2|}\right)^2 \right. \\ &\quad \left. + \left(r_{31} - \frac{|\mathbf{p}_3|}{|\mathbf{p}_1|}\right)^2\right), \end{aligned} \tag{2}$$

where $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ are the 3D axis vectors, r_{mn} is the ratio of the length of axis lines $\mathbf{p}_m$ and $\mathbf{p}_n$ as measured in the sketch plane, and θ_{mn} is the angle between $\mathbf{p}_m$ and $\mathbf{p}_n$. The weighting factor ω allows a tradeoff between the angular and length constraints. Note that this function has a global minimum at 0.

Since Eq. (2) can be explicitly differentiated, the Levenberg–Marquardt method [20] can be used to determine a solution for this nonlinear optimization problem. This fast nonlinear optimization method is an iterative variation of the Newton method for nonlinear optimization and relies on computation of the Jacobian matrix

$$\mathbf{J} = \left[\frac{\delta f}{\delta z_1}, \frac{\delta f}{\delta z_2}, \frac{\delta f}{\delta z_3}\right]. \tag{3}$$

The partial derivative $\delta f/\delta z_1$, for example, is given by

$$\begin{aligned} \frac{\delta f}{\delta z_1} = &-2\left(\frac{\mathbf{p}_1\cdot\mathbf{p}_2}{|\mathbf{p}_1||\mathbf{p}_2|}\right)\left(\frac{z_2}{|\mathbf{p}_1||\mathbf{p}_2|} + \frac{\mathbf{p}_1\cdot\mathbf{p}_2}{|\mathbf{p}_2|}\right)\left(\frac{z_1}{|\mathbf{p}_1|^3}\right) \\ &-2\left(\frac{\mathbf{p}_3\cdot\mathbf{p}_1}{|\mathbf{p}_3||\mathbf{p}_1|}\right)\left(\frac{z_3}{|\mathbf{p}_3||\mathbf{p}_1|} + \frac{\mathbf{p}_3\cdot\mathbf{p}_1}{|\mathbf{p}_3|}\right)\left(\frac{z_1}{|\mathbf{p}_1|^3}\right) \\ &+2\omega\left(r_{21} - \frac{|\mathbf{p}_2|}{|\mathbf{p}_1|}\right)\left(\frac{z_1|\mathbf{p}_2|}{|\mathbf{p}_1|^3}\right) \\ &+2\omega\left(r_{31} - \frac{|\mathbf{p}_3|}{|\mathbf{p}_1|}\right)\left(\frac{z_1}{|\mathbf{p}_1||\mathbf{p}_2|}\right). \end{aligned} \tag{4}$$

Once the values of z_1, z_2, and z_3 have been determined, the four vertices attached to the axis system are considered to have been reconstructed. The MST can then be used to determine the depths of all other sketch vertices. The depth of any vertex attached to an already reconstructed vertex by a line with direction vector $\mathbf{p}_n = [x_n, y_n, z_n]$ in the MST is determined by first reconstructing the missing depth component of this vector, then adding it to the depth value of the already reconstructed endpoint. The unknown depth component z_n is reconstructed by first selecting the axis vector $\mathbf{p}_a \in \{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ that maximizes the 2D projection $(x_n x_a + y_n y_a)/(\sqrt{x_n^2 + y_n^2}\sqrt{x_a^2 + y_a^2})$, then choosing the value of z_n that minimizes the equation

$$\left(1 - \frac{x_a x_n + y_a y_n + z_a z_n}{\sqrt{x_a^2 + y_a^2 + z_a^2}\sqrt{x_n^2 + y_n^2 + z_n^2}}\right)^2. \tag{5}$$

Note that this equation will reach a global minimum of 0 when the direction vector is coincident with a 2D axis vector. This process begins at the axis origin, which has depth 0, and proceeds first to the other vertices connected to the attached axis lines, then throughout the remainder of the MST.

Many alternative methods of constructing the 3D axis system are possible within the formulation presented above. For example, rather than select an axis vertex, an alternative approach is to construct an independent, unattached 3D axis system that will fit the peaks of the ADG. This produces an orthogonal axis system except in degenerate cases, but is more computationally intensive than the ADG algorithm.

4.3. Reconstructing curved strokes

Sections 4.1 and 4.2 dealt with the reconstruction of sketch vertices using a connected straight line graph extracted from the original sketch. Those strokes in the original sketch that cannot be represented by a straight line must also be reconstructed. This is accomplished by a second reconstruction algorithm that processes each stroke independently. The (x, y) locations of every point in a curved stroke are specified in the sketch; the depth of each point in the curve must be determined by the reconstruction process.

Though a stroke can specify an arbitrary path in three dimensions, it is difficult to sketch an arbitrary, unambiguous 3D path entirely by projection onto the sketch plane. The stroke reconstruction algorithm therefore relies on the underlying assumption that each curved stroke is planar, though the parameters of the planar equation are unknown. The goal of the curve reconstruction process is to determine a plane onto which the user might plausibly have drawn the stroke. The depth of each point in the curved stroke is then determined by projection onto the plane (Fig. 5).

The planar equation $a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$ has 3 unknowns $[a, b, c]^T$, which specify the planar normal vector; these must be determined by the reconstruction algorithm. Since the plane is constrained by the requirement that it contain the line $\mathbf{v}$ passing through both of the curve's end points, it is possible to determine the planar normal by optimization over a single variable. An initial planar normal $\mathbf{n}_0 = [a_0, b_0, c_0]^T$

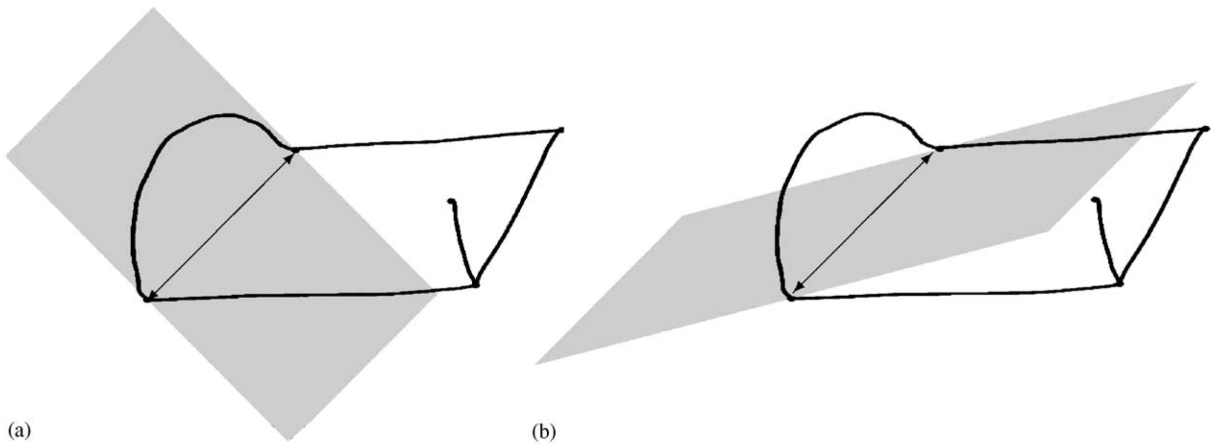


Fig. 5. A 3D axis system with an attached curved stroke, and two possible stroke planes, indicated in light gray. Each projection plane contains the line connecting the two curve endpoints (indicated by the vector). The depth value z_n (where depth is defined into the screen) for each point (x_n, y_n) along the curve are recovered by projection onto the underlying plane. The plane in (a) has the optimal orientation as given by the solution to Eq. (10). The plane in (b) is an implausible plane, which will yield projected depth points well outside the range specified by the two endpoints.

is constructed so that it is perpendicular to $\mathbf{v}$. All other allowable normals can then be constructed by rotating the initial normal by an angle θ around $\mathbf{v}$ to yield the rotated normal $\mathbf{n}_\theta = [a_\theta, b_\theta, c_\theta]^{\mathrm{T}}$:

$$\begin{bmatrix} a_\theta \\ b_\theta \\ c_\theta \end{bmatrix} = [\mathbf{A}_\theta] \begin{bmatrix} a_0 \\ b_0 \\ c_0 \end{bmatrix}, \tag{6}$$

where $\mathbf{A}_\theta$ is a 3×3 rotation matrix that specifies a rotation of angle θ around $\mathbf{v}$. Following rotation of the normal, the equation of the projection plane is given by $a_\theta(x - x_a) + b_\theta(y - y_a) + c_\theta(z - z_a) = 0$, where (x_a, y_a, z_a) is the first stroke endpoint, which is located at one of the reconstructed sketch vertices. The planar equation can likewise be specified by $a_\theta(x - x_b) + b_\theta(y - y_b) + c_\theta(z - z_b) = 0$, where (x_b, y_b, z_b) is the second, similarly reconstructed stroke endpoint.

The optimization function relates the depth value for a particular stroke point to the depths of the stroke's endpoints. The optimization function is based on the assumption that the most likely stroke plane for most simple planar curves is the one that causes the mean depth of the stroke points to fall between the depths of the stroke endpoints:

$$f(n) = (z_n - z_a)^2 + (z_n - z_b)^2. \tag{7}$$

It can be shown that this function has a minimum when $z_n = (z_b + z_a)/2$; minimizing this function over the collection of stroke points will produce mean projected depths that are close to the midpoint of the range between z_b and z_a. Since z_n is determined by projection onto the plane with normal $\mathbf{n}_\theta$ passing through the stroke endpoints (x_a, y_a, z_a) and (x_b, y_b, z_b), Eq. (7) may be rewritten as a function of θ and the stroke points (x_n, y_n)

$$f(\theta, n) = \left(\frac{a_\theta(x_n - x_a) + b_\theta(y_n - y_a)}{c_\theta}\right)^2 \tag{8}$$

$$+\left(\frac{a_\theta(x_n - x_b) + b_\theta(y_n - y_b)}{c_\theta}\right)^2. \tag{9}$$

The optimization goal for a curved stroke with N points is to find $\theta_{\min}$, where

$$\theta_{\min} = \underset{\theta}{\operatorname{argmin}} \sum_{n=1}^{N} f(\theta, n). \tag{10}$$

The normal of the optimal projection plane is given by $\mathbf{n}_{\theta_{\min}}$.

In many sketches, the projection plane for curved strokes is often related to the other sketch elements. In particular, the normal of the projection plane is often aligned with one of the sketch axes, or with one of the other lines connecting the stroke endpoints. In practice, the optimal projection plane normal is determined by first searching over quantized values of θ to determine a set of normal vectors $\mathbf{n}_\theta$, each of which maximizes the projection onto a single sketch line vector. The optimal normal is chosen from this set as the one that minimizes $\sum_{n=1}^{N} f(\theta, n)$. If the smallest value of $\sum_{n=1}^{N} f(\theta, n)$ measured over this set exceeds a specified threshold, $\theta_{\min}$ is determined by exhaustively searching over a set of quantized angular values.

4.4. Constructing faces

The object's constituent faces must be identified in order to convert the set of 3D vertices and strokes generated by the reconstruction algorithms into a solid. Several works (e.g. [21]) specify methods of identifying faces in 2D sketches. These algorithms are computationally intensive and relatively complex to implement, however. In this work, faces are identified by recursively searching the connectivity graph of the reconstructed 3D object for cycles of approximately coplanar lines using a fast, greedy search algorithm.

Since each of the object's faces are subtended by a plane, the sketch faces are triangulated by projecting each stroke's points onto the underlying plane, assuming that each descendant face is a hole. Holes are handled by projecting the hole contour onto the underlying plane. The resulting set of 2D contours are then triangulated using a Delauney triangulator capable of handling holes [22]. The resulting triangle mapping is then applied to the 3D points making up each stroke. This approach works well for faces comprised of approximately linear strokes. For faces composed of four strokes, with curved and straight lines appearing in alternate progression, the triangulation is determined using ruled surfaces [23]. More advanced methods (e.g. Coons patches) must be investigated to handle more complex cases.

5. Adding features

The endpoints of all new strokes that are added to the sketch are classified as one of the following:

(1) overlapping an existing reconstructed vertex, in which case this endpoint is linked to the vertex;
(2) lying on a stroke in the 3D object, in which case depth information for the endpoint is determined by interpolating along the 3D stroke;
(3) embedded within a face, in which case the endpoint's depth is determined by projection onto the face. A cycle of strokes whose vertices are all embedded into the same face are said to constitute a descendant face. Faces are stored as a tree; the immediate descendants of a face at the top level are treated as holes.

If none of the above are true, the endpoint's depth is determined using a general but computationally intensive reconstruction algorithm based on the work by Lipson and Shpitlani [19]. The optimization cost function is the sum of separate cost functions given by the parallelism, isometry and orthogonality of the resulting 3D shape:

$$f(\mathbf{v}_m, \mathbf{v}_n) = (1 - |\langle \mathbf{v}_m, \mathbf{v}_n \rangle|) + \left(1 - \frac{\min(|\mathbf{v}_m|, |\mathbf{v}_n|)}{\max(|\mathbf{v}_m|, |\mathbf{v}_n|)}\right) + (|\langle \mathbf{v}_m, \mathbf{v}_n \rangle|), \tag{11}$$

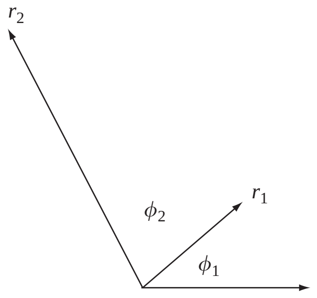
where the vector $\mathbf{v}_n = [x_n, y_n, z_n]$ is the vector given by the difference between the endpoints of stroke n, $\langle \mathbf{v}_m, \mathbf{v}_n \rangle$ is the normalized inner product of $\mathbf{v}_m$ and $\mathbf{v}_n$, and $|\mathbf{v}_n|$ is the vector magnitude. The optimization cost for a sketch consisting of N strokes is given by $\sum_{m=1}^{N} \sum_{n=m}^{N} f(\mathbf{v}_m, \mathbf{v}_n)$. A hill-climber [24] is used to minimize the total cost.

6. Results

The performance of the axis reconstruction algorithm, and the effect of the length ratio weight ω on the reconstructed axis systems, were verified by generating and reconstructing multiple random 2D axis systems. The first axis vector was taken to be the 2D vector $(0, 1)$. The second axis was generated by rotating the vector $(0, 1)$ by a random angle ϕ_1 and setting its length to a random factor r_1. The third axis vector was generated by rotating the second by a factor ϕ_2 and randomly setting its length to be r_2. The values were chosen such that $0 \leqslant \phi_1 + \phi_2 \leqslant 180^\circ$ and $0.5 \leqslant r_1, r_2 \leqslant 3$. A successful reconstruction generated an assignment of depth values to each axis such that the mean angle $\mu_{\theta_{mn}}$ between the resulting angle satisfied $70 < \mu_{\theta_{mn}} < 110$. Fifty different randomly generated axes were used to evaluate performance for five different values of ω ranging from 1 to 0. The relative importance of length ratios in the optimization function increases as ω increases.

Fig. 6 shows the mean angle between the reconstructed axes (in degrees) and the mean ratio between the longest and shortest reconstructed axes as a function of the weighting term ω, as well as the percentage of random axes that were successfully reconstructed. The percentage of successful reconstructions is encouragingly high, though it drops as ω increases; this is to be expected since the length ratio criterion necessarily limits the number of 3D axis systems that can be successfully reconstructed from a given 2D axis. The mean inter-axis angle $\mu_{\theta_{mn}}$ moves toward the ideal, value of 90° as the effect of axis lengths are reduced, though the mean length ratio μ_{length} increases, resulting in axis lengths that vary considerably. This suggests that the value of ω can be used to determine the elongation of the reconstructed 3D shape.

The curved stroke reconstruction algorithm was tested on several different symmetric and asymmetric curves where the distance $|z_b - z_a|$ between the depths of the two stroke endpoints was varied from 0 to ∞. The algorithm generated plausible reconstructions for curved strokes at different orientations drawn using the same set of fixed endpoints. The reconstruction was most plausible for small to moderate values of $|z_b - z_a|$, and for strokes with a high degree of curvature, though this is to some extent subjective for each user. The stroke planes for strokes with little to no curvature were difficult to determine, largely because the relative



ω	1	0.1	0.01	0.001	0
$\%_{success}$	80%	84%	96%	98%	96%
$\mu\theta_{mn}$	86.98°	89.78°	89.69°	90.18°	90.18°
μ_{length}	1.25	1.24	1.32	1.58	2.09

Fig. 6. A 2D axis system composed of a single fixed axis and two randomly constructed axes with lengths r_1 and r_2 and relative rotation angles ϕ_1 and ϕ_2, respectively. The table shows the percentage of successfully reconstructed 3D axis systems, as well as the mean inter-axis angle $\mu_{\theta_{mn}}$ and mean length ratio μ_{length} measured over the reconstruction of 50 randomly constructed 2D axes.

importance of the orientation of the projection plane decreases as the stroke curves become less pronounced. The reconstruction algorithm was not found to incur significant computational overhead for angular increments of 0.05 rad, and sketches containing up to 10 curved strokes. Since the error term is analytically differentiable, fast optimization techniques may also be employed to reduce computational overhead.

The curved stroke reconstruction algorithm exhibited a strong dependence on viewpoint. In cases where $|z_b - z_a|$ was small, curved strokes were always reconstructed using a projection plane that was nearly parallel to the sketch plane. Furthermore, relatively small variations in the shape of the sketched curve could result in very different reconstructions. Finally, since the curve reconstruction process occurs after the sketch vertices have been reconstructed, the reconstructed vertices may occupy slightly different positions in 3D space than intended by the user, which affects the correspondence of the 2D curve with the intended 3D curve. Further study is required to establish the impact of these considerations on the correspondence between the reconstructed 3D curve and the intended 3D curve.

The reconstruction algorithm performed best on sketches that exhibited strong orthogonal trends, and whose strokes were highly correlated with the underlying axis system, a class of sketches that includes the majority of engineering diagrams. Because the computationally intensive nonlinear optimization is used only to reconstruct the main axis system, rather than depths of all sketch vertices directly, the algorithm can process sketches composed of 50 or more strokes in interactive time on a Pentium 4 Tablet PC notebook computer. An example demonstrating the reconstruction of several sketches incorporating both straight and curved strokes is given in Figs. 7 and 8. The example also demonstrates the addition of holes to the 3D objects, and the performance of the general reconstruction process that optimizes Eq. (11). In all of these examples $\omega = 0.01$.

7. Conclusions and future work

This paper presented a pen-based sketching system for progressively constructing 3D objects from single free-hand sketches. Sketches representing the orthographic projection of a 3D object onto a sketch plane are treated as graphs consisting of vertices connected by the sketch strokes. The reconstruction problem consists of assigning a depth value to each vertex, and subsequently reconstructing each curved stroke. This is accomplished by a two stage reconstruction process. The first stage tests a straight line sketch extracted from the original for the presence of prevailing angular trends and uses these to determine a 3D axis system that maps 2D lines onto 3D lines. This axis system is used in combination with the sketch connectivity graph to assign a depth to each vertex. The second stage reconstructs curved strokes, under the assumption that they are planar. Following reconstruction, the 3D object's faces are identified and triangulated. Users can then add additional strokes, and sketch directly onto the object's faces. The system is implemented with a consistent pen-based interface that mimics pencil and paper sketching, and can reconstruct sketches of up to 50 strokes in interactive time.

Future work on the ADG reconstruction algorithm will address the reconstruction of shapes that do not exhibit pronounced angular trends, or that have strokes with large deviations from the underlying axis system. Future work on the curve reconstruction algorithm will address the problem of jointly reconstructing multiple curves in order to increase the symmetry of the 3D object. Furthermore, examination of the optimization surface for several curved strokes showed a plateau around the global minimum, suggesting that there are a range of possible stroke planes that might yield very similar results. Adding a term that maximizes the projection of the planar normal onto the sketch axis system to the optimization function might differentiate between these planes, and allow the overall angular trends in the sketch to be incorporated into the curve

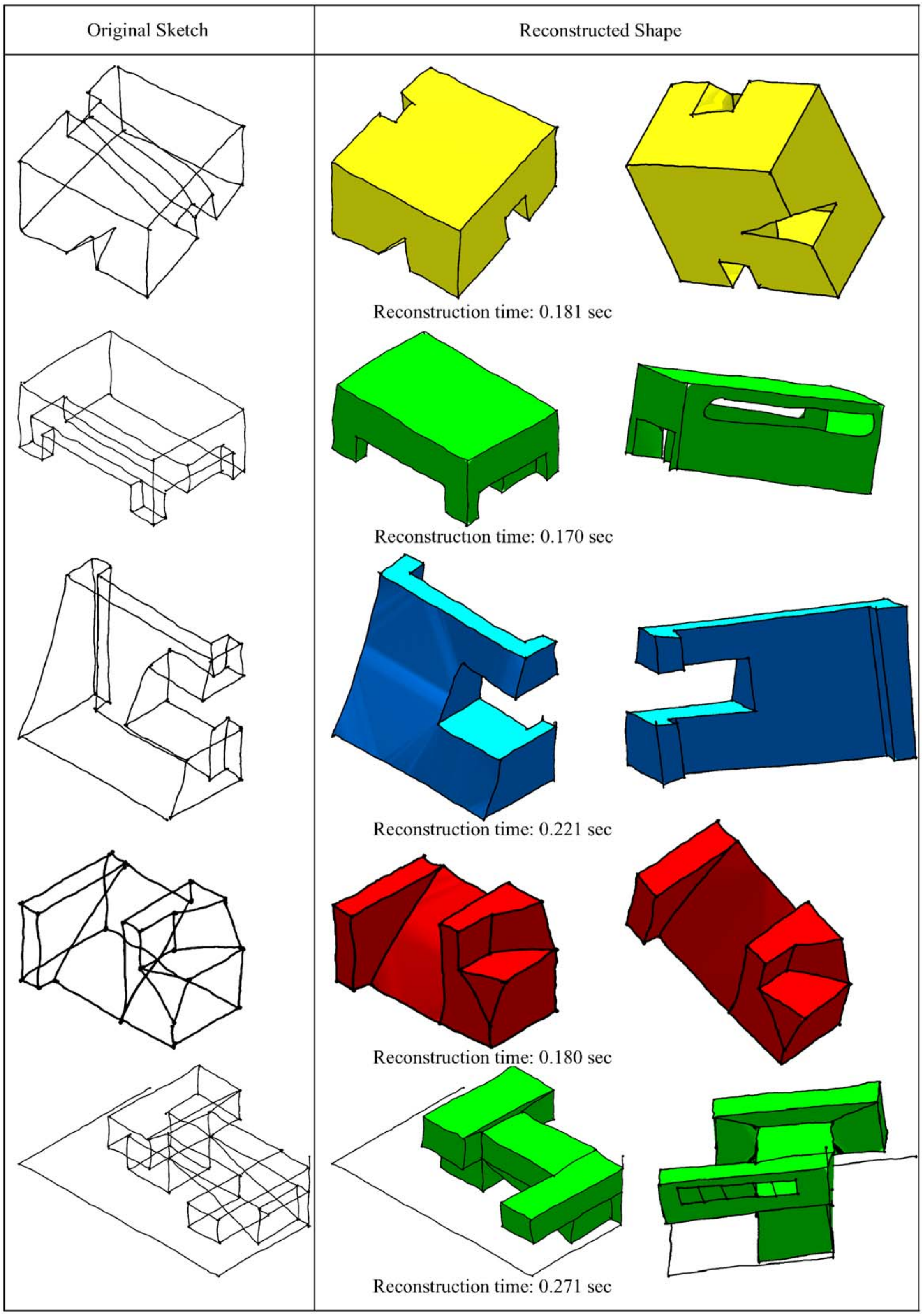

Original Sketch
Reconstructed Shape
Reconstruction time: 0.181 sec
Reconstruction time: 0.170 sec
Reconstruction time: 0.221 sec
Reconstruction time: 0.180 sec
Reconstruction time: 0.271 sec

Fig. 7. (a) Symmetrical unreconstructed straight-line 2D sketches and the accompanying reconstructed 3D shapes from two viewpoints. Reconstruction times are given for a Pentium 4M 1.7 Ghz Tablet PC.

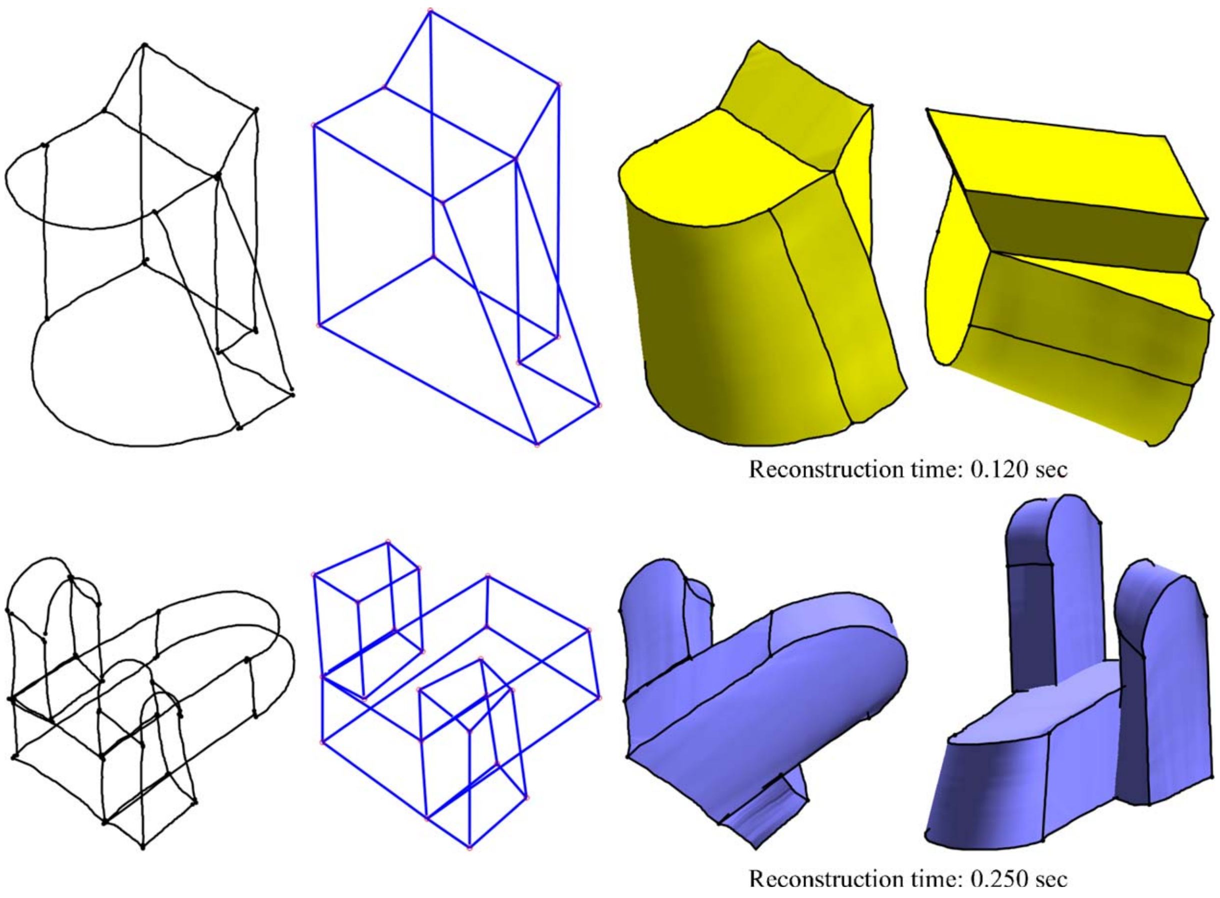

Reconstruction time: 0.120 sec
Reconstruction time: 0.250 sec

Fig. 8. Two sketches of shapes with mixed curved and straight lines, the underlying straight-line graphs used to reconstruct vertex depths, the reconstructed objects, an alternate viewpoint.

reconstruction process. Finally, research on the construction of plausible 3D surfaces from object faces consisting of multiple curved planar strokes is also required.

References

[1] Ferguson ES. Engineering and the minds eye. Cambridge, MA: MIT Press; 1992.
[2] Stahovich TF, Davis R, Schrobe J. Generating multiple new designs from a sketch. Artificial Intelligence 1998;104: 211–64.
[3] Davis R. Position statement and overview: sketch recognition at MIT. In: AAAI Sketch Understanding Symposium. Stanford, CA; 2002.
[4] Huffman DA. Impossible objects as nonsense sentences. In: Meltzer B, Mitchie D, editors. Machine intelligence. Edinburgh: Edinburgh University Press; 1971.
[5] Clowes MB. On seeing things. Artificial Intelligence 1971;2(1):79–112.
[6] Mackworth AK. Interpreting pictures of polyhedral scenes. Artificial Intelligence 1973;4:121–37.
[7] Wei X. Computer vision method for 3D quantitative reconstruction from a single line drawing. PhD thesis, Department of Mathematics, Beijing University, China [in Chinese; for a review in English see Wang and Grinstein; 1993].
[8] Lamb D, Bandopdhay A. Interpreting a 3D object from a rough 2D line drawing. In: Proceedings of visualization '90; 1990.
[9] Fukui Y. Input method of boundary solid by sketching. Computer aided design 1998;20(8):434–40.
[10] Zeleznik R, Herndon K, Hughes J. SKETCH: an interface for sketching 3d scenes. In: Proceedings of SIGGRAPH; 1996.
[11] Wang W, Grinstein G. A polyhedral object's CSG-Rep construction from a single 2D line drawing. In: Proceedings of the 1989 SPIE intelligent robots and computer vision III: algorithms and techniques, vol. 1192; 1989.
[12] Sugihara K. The interpretation of line drawings. Cambridge, MA: MIT Press; 1986.
[13] Grimstead IJ, Martin RR. Creating solid models from single 2D sketches. In: Solid modeling '95. Salt Lake City, UT, USA; 1995.

[14] Lipson H, Shpitlani M. Optimization-based reconstruction of a 3D object from a single freehand line drawing. Journal of Computer Aided Design 1996;28(8):651–63.

[15] Lipson H, Shpitlani M. Conceptual design and analysis by sketching. Journal of AI in Design and Manufacturing (AIEDEM) 2000;14:391–401.

[16] Lipson H, Shpitlani M. Correlation-based reconstruction of a 3D object from a single freehand sketch. In: AAAI spring symposium on sketch understanding; 2002.

[17] Igarashi T, Matsuoka S, Tanaka H. Teddy: sketching interface for 3d freeform design. In: Proceedings of SIGGRAPH; August 1999.

[18] Shpitlani M, Lipson H. Classification of sketch strokes and corner detection using conic sections and adaptive clustering. ASME Journal of Mechanical Design 1997;119(2): 131–5.

[19] Cormen TH, Leiserson CE, Rivest RL. Introduction to algorithms. Cambridge, MA: MIT Press; 1999.

[20] Heath MT. Scientific computing, an introductory survey, 2nd ed. New York, NY: McGraw-Hill; 2002.

[21] Shpitlani M, Lipson H. Identification of faces in a 2D line drawing projection of a wireframe object. IEEE Transactions on Pattern Analysis and Machine Intelligence 1996;18(10):1000–12.

[22] Shewchuk JR. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin MC, Manocha D, editors. Applied computational geometry: towards geometric engineering, Lecture Notes in Computer Science, vol. 1148, Springer, Berlin; p. 203–22, from the First ACM Workshop on Applied Computational Geometry.

[23] Schneider PJ, Eberly DH. Tools for computer graphics, 2003.

[24] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical Recipes in C++: The art of scientific computing, 2nd ed. London, New York: Cambridge University Press; 2003.

2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling (2005)
T. Igarashi (Chair)

A Sketch-Based Interface for Iterative Design and Analysis of 3D Objects

M. Masry and H. Lipson

mark.masry,hod.lipson@cornell.edu
Sibely School of Mechanical and Aerospace Engineering
Cornell University

Abstract

This paper presents a freehand, sketch-based interface for Computed Aided Design (CAD) engineering design and finite element analysis. After a user sketches a two dimensional sketch consisting of connected straight and curved strokes, the sketch is processed by two optimization-based reconstruction algorithms that can reconstruct sketches of 3D objects made up of straight lines and planar curves. The proposed implementation allows certain types of objects with over 50 strokes to be reconstructed in interactive time. Following reconstruction, the structural properties of the 3D shape can be examined using finite element analysis. The object can quickly be modified using the pen-based interface according to the results of the analysis. The combination of a rapid, sketch-based design interface and finite element analysis allows users to iteratively design, analyze and modify 3D objects in an intuitive and flexible way.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Interaction Techniques

1. Introduction

Visual methods of communication are often the simplest and most efficient way of conveying information about the shape, composition and relationships of an object's components. Visual information often transcends the limitations imposed by spoken or written languages. Engineering information, in particular, is often conceived, recorded and transmitted in a visual, non-verbal language. Little work has been done, however, to create fast and intuitive conceptual interfaces based on visual input. Conventional CAD user interfaces are typically cumbersome to use and hamper creative flow.

Freehand sketching, the informal drawing of shapes using freeform lines and curves, has remained one of the most powerful and intuitive tools used at the conceptual design stage. Sketches, in contrast to typical CAD designs, can quickly and easily be created to convey shape information. Simple paper-based sketching also has many drawbacks: the viewpoint is fixed and cannot be changed in mid drawing; the sketch is passive and cannot be directly simulated or analyzed using computational engineering tools (e.g. structural analysis or kinematic simulation); the sketch is tentative and if a final, accurate model is desired, it must be recreated from scratch.

The ideal solution from a designer's point of view should combine both the speed and ease of freehand sketching with the flexibility and analytical abilities of computational analysis tools. Sketch reconstruction algorithms allow designers to quickly specify a 3D object using a single, freehand sketch; the object can then be subjected to real-time physical simulations such as structural, fluid, or manufacturing analysis. The combination of sketching and physical simulation allows for a revolutionary, *iterative* design process: users can sketch an object, gain immediate insight into its physical properties, and revise the sketch until the design concept matures. By eliminating the restrictions of traditional CAD interfaces, analysis tools can be brought much earlier into the critical early conceptual design stage.

In addition to its practical applications, an interface for 3D sketching and analysis has significant educational benefits. By removing the barrier between sketch and simulation students can quickly explore and understand the physical properties, the advantages and the weaknesses of their design. Instructors can quickly convey design concepts and physical

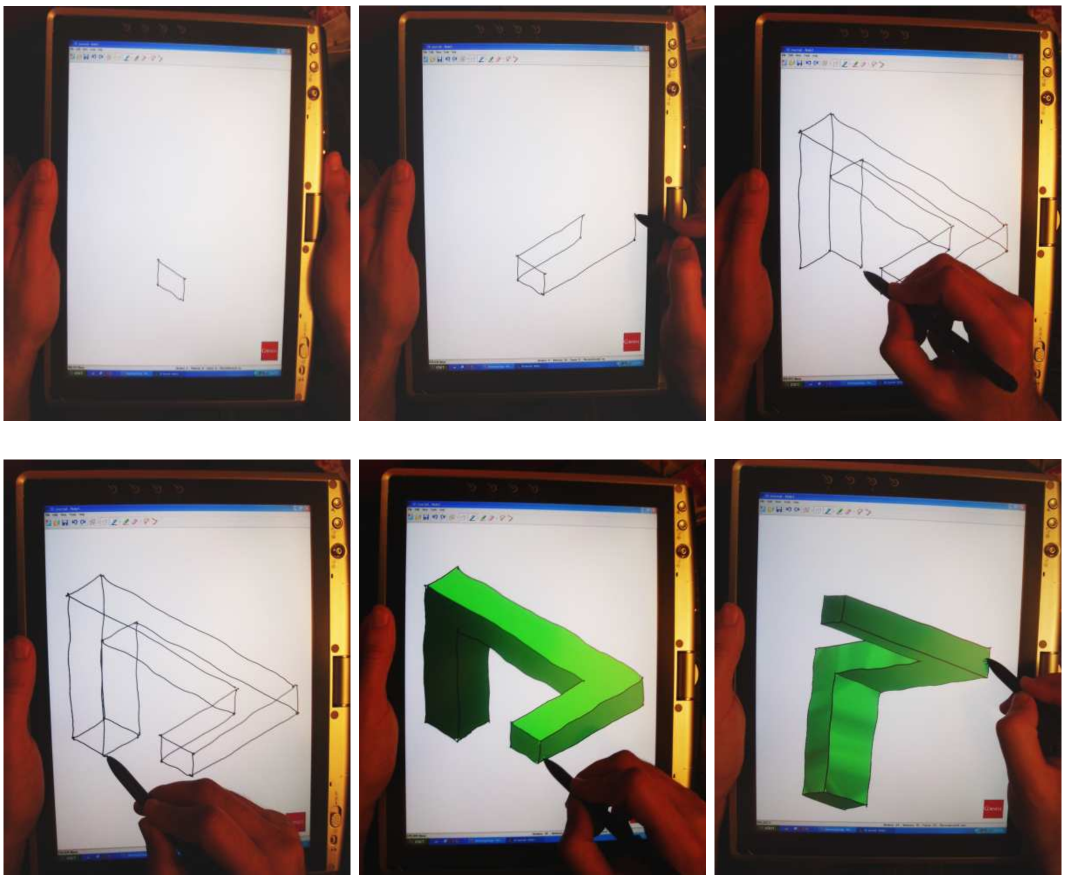


Figure 1: *A user creating, rendering and rotating a shape using the proposed system on a Tablet PC*

properties during lectures, making the classroom experience more dynamic and facilitating learning.

This paper presents an intuitive, pen-based sketching tool that combines sketch-based design of 3D objects with finite element analysis (FEA) in order to achieve these goals. The proposed approach consists of two parts: a reconstruction stage that reconstructs a 3D object from a single orthographic sketch, and an analysis stage that performs a finite element analysis on the resulting 3D object and displays the results in the sketch plane. The reconstruction algorithms can process sketches consisting of both straight lines and planar curves, and run in interactive time on certain types of complex sketches. Once the analysis has been performed, users can modify the shape using a consistent pen-based interface, and perform subsequent analyses until the design is complete.

2. Previous work

Several works have investigated the use of sketch-based interfaces. Stahovich et al. [SDS98] demonstrated a system that could interpret the causal functionalities of a two dimensional mechanism depicted in a sketch, and generate alternative designs. Davis et al. [Dav02] recently showed a system that simulated rigid-body dynamics of a sketched two-dimensional mechanism. These systems are mostly two dimensional, and the few that are 3D require additional steps that break the flow of sketching.

Figure 2 outlines the reconstruction of a 3D object from a 2D sketch, in which any arbitrary set of depths $\{Z\}$ that are assigned to the vertices in the sketch constitutes a 3D configuration whose projection will match the given sketch exactly. In principle, each such assignment yields a valid candidate 3D reconstruction. A considerable amount of research has focused on the reconstruction of polyhedral objects from straight-line sketches. Several methods construct relationships between the slope of sketch lines and the gradients of the associated 3D faces [Mac73, Wei87]. Other approaches include incremental construction [LB90, Fuk98, ZHH96], and construction using known primitives [WG89]. Detailed surveys are given in [LS00] and [CPC04].

Optimization-based reconstruction for polyhedral objects

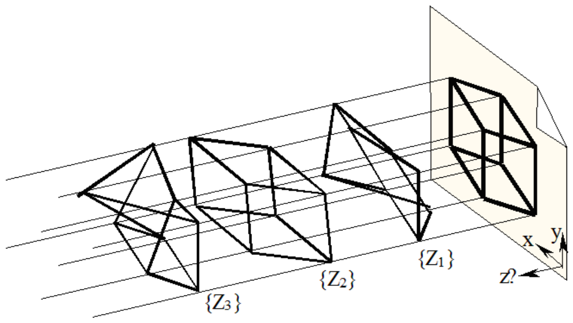


Figure 2: *A sketch provides only two of the coordinates (x,y) of object vertices. A 3D reconstruction must recover the unknown depth coordinate z. In parallel projections, this degree of freedom is perpendicular to the sketch plane; there are an infinite number of candidate objects – the problem is indeterminate. Each candidate object is represented by a unique set of Z coordinates, e.g. sets $\{Z_1\}$, $\{Z_2\}$ and $\{Z_3\}$*

specified by straight line sketches determine the depth assigned to each sketch vertex by optimizing a target function. These methods are more general than the approaches above and can be used to reconstruct relatively complex 3D objects, though they suffer from high computational complexity and susceptibility local minima in the target function. A comprehensive review of optimization-based reconstruction techniques can be found in [LS96, LS00]. We proposed an approach in which 2D sketches were converted to line and vertex graphs and analyzed for regularities such as parallelism, perpendicularity and symmetry [LS96]. These regularities were then summed to produce an overall compliance function that measures how well the 3D construction conforms to the regularities in the 2D sketch. Reconstruction proceeds by optimizing this compliance function. We have also investigated machine learning approaches to sketch reconstruction [LS02]. Optimization-based approaches to 3D reconstruction were used by Shesh et al. [SC04] in conjunction with incremental shape construction methods. Optimization-based approaches to sketch beautification and reconstruction were also used by Company et al. [CCCP04]. Neither of these two methods, however, address the problem of curve reconstruction, or incorporate physical analysis into the design process.

In contrast to the number of approaches for reconstructing polyhedral objects specified by straight line sketches, there are relatively few reconstruction algorithms that can be applied to sketches of 3D objects with curves. The best-known such work is the Teddy system [IMT99], which uses a sketch-based interface to specify the boundaries for reconstruction of a curved solid. This system cannot be used to reconstruct polyhedral objects, or objects that mix straight lines and curves. Approaches that deform a template (e.g. [VTMS04]) to fit a curved structure have also been developed, though these require that a 3D template be formed prior to curve reconstruction, and can therefore not reconstruct arbitrary single curves.

The 3D sketching system proposed in this paper uses a fast reconstruction algorithm that chooses a plausible three-connected sketch vertex to serve as a 3D axis origin based on the angular distribution of the lines in a sketch, and reconstructs the depths of the three vertices at the opposite ends of the attached strokes. Depths are then assigned to the other sketch vertices by propagation across the connectivity graph given by the sketch. This approach allows the reconstruction of 3D objects with a connectivity graph whose edges conform to an underlying, orthogonal axis system. Following reconstruction of the sketch vertices, a second optimization procedure reconstructs each curved stroke.

In this work we wish to explore the utility of 3D sketch understanding as a design tool, when combined with conventional analysis capabilities. We chose to link the sketch interpretation with standard finite element analysis code [RP05], and demonstrate a complete cycle of 3D modeling and result presentation performed entirely in the sketch medium. We chose Finite element analysis as a commonly used form of engineering analysis with a broad spectrum of applications, though clearly other analysis codes could be used.

3. 3D Sketching System

This sketching system attempts to create an experience similar to drawing with pencil and paper. The application was implemented on a tablet PC with a pen input device. An example session is shown in Figure 1. The user interface relies entirely on the pen. A session begins with an initial sketch specified by a set of loosely connected strokes in the sketch plane given by the digitizer surface. Each potentially curved stroke is assumed to be piecewise linear, and is represented internally by the location of its two endpoints and a series of values specifying the location of each point along the length of the stroke. Strokes may intersect in the sketch plane, but these intersections are not taken to represent intersections in 3D space; at this stage, strokes may be joined only at the endpoints. Users can erase all or part of a stroke at any time.

The reconstruction process when a user attempts to spin the 3D object by pressing on the pen's barrel button. As a first step toward reconstruction, all stroke endpoints within a specified distance of one another are connected using an approach given by Shpitlani et al. [SL97]. The 2D sketch can now be interpreted as a connectivity graph (or straight-line graph) representing the 2D orthographic projection of a 3D object onto the plane $z = 0$, with vertices given by the connections between strokes and edges specified by the straight line connections between vertices.

Following the initial reconstruction, the sketch can be rotated, rescaled or resized. Each stroke is treated as an independent object, and can be erased or modified by the user either pre- or post-reconstruction. Following reconstruction, the 3D object's faces are identified and triangulated. The triangulated faces are used to facilitate user interaction with the object, and as part of a hidden line removal algorithm that determines the visible parts of each stroke. The triangulated faces are also used to construct the object manifold surface that is required for finite element analysis.

The 3D strokes making up the reconstructed object may be altered and erased in the same way as the 2D strokes in the original sketch. Strokes may also be added to the 3D object, in which case the reconstruction procedure is used to determine the 3D position of any stroke vertices that are not coincident with object features.

The same interface can be used to specify the face parameters required to perform Finite Element Analysis (FEA). The analysis itself is triggered using the application's toolbar, at which point a tetrahedral mesh is generated for the reconstructed solid and finite element analysis is performed. The resulting deformation is superimposed directly over the strokes making up the original sketch.

The following sections describe the reconstruction and analysis stages in more detail.

4. Sketch Reconstruction

Since the (x, y) coordinates of each vertex are given in the sketch, reconstructing a 3D object requires assigning a z coordinate (also termed the *depth* value) to each vertex, subject to constraints on the characteristics of the resulting 3D object. It is assumed that the sketch vertices are *connected* i.e. that a path can be constructed from each vertex to every other vertex. This restriction necessarily restricts the reconstruction algorithm to sketches of single objects. Further research is required to determine the best method by which sketches of multiple objects can be reconstructed with suitable relative positions. It is further assumed that none of the vertices or strokes in the sketch completely obscure other elements of the same kind.

The algorithm is intended to reconstruct 3D objects whose vertices can be connected by a spanning tree consisting of straight lines aligned with one of 3 orthogonal axes. Sketches consisting of connected planar curves can also be reconstructed, provided that the underlying straight line connectivity graph satisfies this requirement. Though these requirements are restrictive, this approach works well for drawings of objects whose edges predominantly conform to some overall orthogonal axis system, which includes a wide range of engineering design drawings. The sketch vertices need not necessarily be trihedral (see objects 1 and 4 in Figure 4), though sketches with vertices connected to the spanning tree. In these cases, the proposed approach can be used to reconstruct part of the object before a more general optimization-based algorithm (e.g. [LS96]) is used to complete the reconstruction.

Reconstruction proceeds in two stages: the depths of the sketch vertices are determined while treating all curved strokes as straight line connections between vertices, following which all points along each curved stroke are reconstructed, assuming that the resulting 3D curve is planar. The reconstruction algorithms run in interactive time, allowing for a fluent interaction with the system.

4.1. Reconstructing vertex depths

The vertex reconstruction algorithm used in this paper was given by the authors in an earlier work [KML04], and is summarized here for convenience.

Since orthogonality is the prevailing trend in most engineering drawings, and the easiest to identify, a statistical analysis of the direction of the lines connecting the sketch vertices is performed to determine whether these are consistent with the projections from an underlying orthogonal axis system. The reconstruction process begins with the selection of the three-connected vertex whose attached lines are most representative of the angular distribution of a representative set of sketch lines; this vertex is the origin of the main sketch axis system, and the attached strokes are taken represent the orthographic projection of the 3D axes onto the 2D sketch plane.

The origin of the main axis system is assumed to have a depth of zero. The depth of the opposite vertex of each of the three attached axis line vectors must be determined in order to reconstruct the axis system. The unknown depths are determined as the minimizing solution of an optimization function based on two assumptions about an ideal sketch:

1. The 3D axis vectors should be as close to mutually orthogonal as possible.
2. The ratio of the axis lengths in the sketch plane is equal to the ratio of their lengths in 3D space.

Note that the second assumption imposes restrictions on the sketch viewpoint: viewpoints where the orthographic projection of the object onto the 2D sketch plane produces axes with very different lengths will result in reconstructed 3D axes with very different lengths. The optimization goal is to minimize a cost function that attains a value of 0 when all three axes are orthogonal in 3D and the ratio of the axis lengths in 3D is equal to their ratio in the sketch plane. Minimization of this cost function can be performed with a fast Levenberg-Marquardt algorithm. Since optimization is required only to reconstruct the three vertices of the axis system, it does not depend on the sketch allowing complicated sketch graphs to be reconstructed in real-time.

Since the sketch graph is connected, it is possible to construct a spanning tree that connects each vertex to the origin vertex. The spanning tree is given by the Maximum weight

Spanning Tree (MST), where the weight of each edge is given by the projection of the edge line onto the 2D axis lines. Once the axis system vertices have been reconstructed, the depths of the remaining vertices are determined by propagating depth values along this tree, beginning with the axis origin.

The depth of the endpoint vertices of all strokes added to the sketch post-reconstruction are determined by interpolating from an underlying, reconstructed sketch feature. If the endpoint does not lie over an existing feature, reconstruction is performed using a general, axis-independent but computationally intensive reconstruction algorithm based our earlier work [LS96]. A hill-climber [PTVF03] is used to minimize the total cost. This approach is also used for the initial reconstruction if there exists no vertex that can serve as the origin of the main axis system.

The curve reconstruction algorithm only requires that the vertices be reconstructed and planar faces be identified in order to run; there are several alternative approaches for reconstructing vertex depths. Company et. al [CCCP04] proposed an optimization-based reconstruction engine that also makes use of a minimum spanning tree, as well as a comprehensive set of observed shape regularities and two separate inflation methods. Their approach also includes an optimization-based sketch beautification strategy that may add to computational cost. A wide range of shapes can be reconstructed, although these occasionally require the use of sketch regularities such as planarity and corner orthogonality that are not considered in this work. Varley et al. [VMS04] proposed a framework for labeling the lines in a sketch that begins by analyzing the angular distribution of lines in the sketch, then constructing a 3D axis system by solving a system of linear equations. The reconstructed axis system is used along with line parallelism to determine the depths of all sketch vertices prior to assigning line labels. This methods is also capable of reconstructing a wide range of sketches, though these appear to also conform to an orthogonal axis system. Furthermore, the method was developed for natural sketches, rather than the wireframes under consideration here.

4.2. Reconstructing Curves

The (x,y) locations of every point in a curved stroke are specified in the sketch; once the vertex depths have been reconstructed, a second reconstruction algorithm reconstructs the depths of each stroke point. Though a stroke can specify an arbitrary path in three dimensions, it is difficult to sketch an arbitrary, unambiguous 3D path entirely by projection onto the sketch plane. The stroke reconstruction algorithm therefore relies on the underlying assumption that each curved stroke is planar, though the parameters of the planar equation are unknown. The goal of the curve reconstruction process is to determine a plane onto which the user might plausibly have drawn the stroke. The depth of each point in the curved stroke is then determined by projection onto the plane. All curves are treated as piecewise linear collections of points in the sketch plane. Special curves (e.g. conic sections) are not treated independently.

The planar equation $a(x-x_0)+b(y-y_0)+c(z-z_0)=0$ has 3 unknowns $[a,b,c]^T$, which specify the planar normal vector; these must be determined by the reconstruction algorithm. Since the plane is constrained by the requirement that it contain the line $\mathbf{v}$ passing through both of the curve's end points, it is possible to determine the planar normal by optimization over a single variable. An initial planar normal $\mathbf{n}_0=[a_0,b_0,c_0]^T$ is constructed so that it is perpendicular to $\mathbf{v}$. All other allowable normals can then be constructed by rotating the initial normal by an angle θ around $\mathbf{v}$ to yield the rotated normal $\mathbf{n}_\theta=[a_\theta,b_\theta,c_\theta]^T$:

$$\begin{bmatrix} a_\theta \\ b_\theta \\ c_\theta \end{bmatrix} = [A_\theta] \begin{bmatrix} a_0 \\ b_0 \\ c_0 \end{bmatrix} \qquad (1)$$

where $\mathbf{A}_\theta$ is a 3×3 rotation matrix that specifies a rotation of angle θ around $\mathbf{v}$. Following rotation of the normal, the equation of the projection plane is given by $a_\theta(x-x_a)+b_\theta(y-y_a)+c_\theta(z-z_a)=0$, where (x_a,y_a,z_a) is the first stroke endpoint, which is located at one of the reconstructed sketch vertices. The planar equation can likewise be specified by $a_\theta(x-x_b)+b_\theta(y-y_b)+c_\theta(z-z_b)=0$, where (x_b,y_b,z_b) is the second, similarly reconstructed stroke endpoint.

The optimization function relates the depth value for a particular stroke point to the depths of the stroke's endpoints. The optimization function is based on the assumption that users intended the depth of the stroke points to fall between the depths of the two stroke endpoints. We therefore choose the stroke plane as the one that causes the depth of each stroke points to fall between the depths of the stroke endpoints, which can be satisfied by minimizing the sum squared distance between both endpoints:

$$f(n)=(z_n-z_a)^2+(z_n-z_b)^2 \qquad (2)$$

It can be shown that this function has a minimum when $z_n=\frac{z_b+z_a}{2}$; minimizing this function over the collection of stroke points will produce mean projected depths that are close to the midpoint of the range between z_b and z_a. Since z_n is determined by projection onto the plane with normal $\mathbf{n}_\theta$ passing through the stroke endpoints (x_a,y_a,z_a) and (x_b,y_b,z_b), Equation 2 may be rewritten as a function of θ and the stroke points (x_n,y_n)

$$f(\theta,n) = \left(\frac{a_\theta(x_n-x_a)+b_\theta(y_n-y_a)}{c_\theta}\right)^2 \qquad (3)$$

$$+\left(\frac{a_\theta(x_n-x_b)+b_\theta(y_n-y_b)}{c_\theta}\right)^2 \qquad (4)$$

The optimization goal for a curved stroke with N points is to find θ_{min}, where

$$\theta_{min} = arg\min_\theta \sum_{n=1}^{N} f(\theta,n) \qquad (5)$$

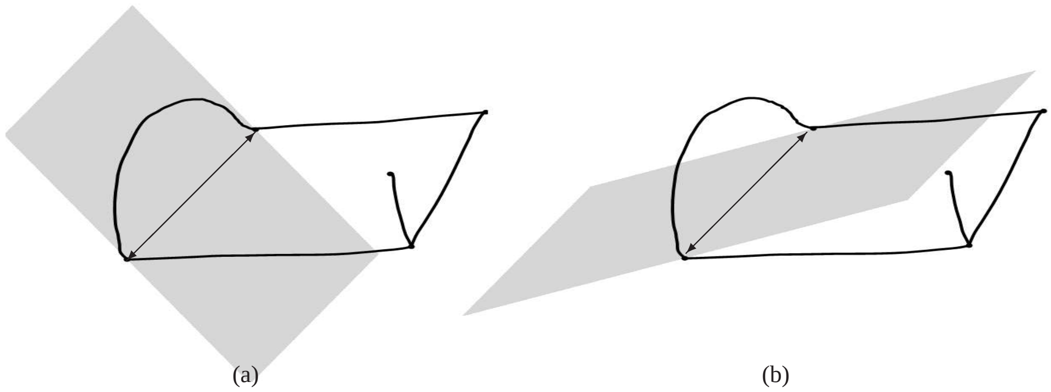


Figure 3: *A 3D axis system with an attached curved stroke, and two possible stroke planes, indicated in light gray. Each projection plane contains the line connecting the two curve endpoints (indicated by the vector). The depth value z_n (where depth is defined into the screen) for each point (x_n, y_n) along the curve are recovered by projection onto the underlying plane. The plane in (a) has the optimal orientation as given by the solution to Equation 5. The plane in (b) is an implausible plane, which will yield projected depth points well outside the range specified by the two endpoints.*

The normal of the optimal projection plane is given by $\mathbf{n}_{\theta_{min}}$. A gradient steepest descent algorithm can be used to find θ_{min}. In many sketches, the projection plane for curved strokes is often related to the other sketch elements. In particular, the normal of the projection plane is often aligned with one of the sketch axes, with one of the other lines connecting the stroke endpoints, or with a the normal of a planar face plane in the reconstructed straight line object. In practice, the optimal projection plane normal is determined by first searching over quantized values of θ to determine a set of normal vectors $\mathbf{n}_\theta$, each of which maximizes the projection onto a single sketch vector. The optimal normal is chosen from this set as the one that minimizes $\sum_{n=1}^{N} f(\theta, n)$.

The curved stroke reconstruction algorithm was tested on several different symmetric and asymmetric curves where the distance $|z_b - z_a|$ between the depths of the two stroke endpoints was varied from 0 to ∞. The algorithm generated plausible reconstructions for curved strokes at different orientations drawn using the same set of fixed endpoints. The reconstruction was most plausible for small to moderate values of $|z_b - z_a|$, and for strokes with a high degree of curvature, though this is to some extent subjective for each user. The stroke planes for strokes with little to no curvature were difficult to determine, largely because the relative importance of the orientation of the projection plane decreases as the stroke curves become less pronounced. The reconstruction algorithm was not found to incur significant computational overhead for angular increments of 0.05 radians, and sketches containing up to 10 curved strokes. Since the error term is analytically differentiable, fast optimization techniques may also be employed to reduce computational overhead.

5. Analysis

Once the 3D object has been reconstructed in its entirety, the object's constituent faces must be identified in order to generate a solid suitable for finite element analysis. In this work, faces are identified by recursively searching the connectivity graph of the reconstructed 3D object for cycles of approximately coplanar lines using a fast, greedy search algorithm. The sketch faces are then triangulated. If the face is approximately planar, a Delaunay triangulator [She96] is used. If the face is composed of 3 or 4 non-coplanar curves, the face is triangulated using Coons patches. Faces composed of more than 4 non-planar curved strokes are not triangulated. Note that the use of Coons patches to triangulate curved surface necessarily limits the interpretation of the curved surface. The correspondence between the reconstructed surface and the user-intended surface requires further investigation.

Once the faces in the object have been triangulated, the object's properties can be analyzed and, if necessary modified by the addition of further object features. Though there many different types of analysis can be applied to the 3D shape (e.g. structural, fluid, or manufacturing analysis) we have implemented a linear approximation to the laws for elastic deformation, since this is representative of engineering design analysis.

Finite Element Analysis (FEA) decomposes a continuous function over a global domain into a piecewise series of functions applied to a set of smaller elements whose union represents the original domain. Prior to performing FEA, the reconstructed 3D object is first checked to see if its triangulated faces can be combined to form a triangulated manifold surface. If so, the surface is decomposed into a set of tetrahedral elements using a meshing algorithm [Si05]. The mesh complexity is limited only by the requirements of the mesh-

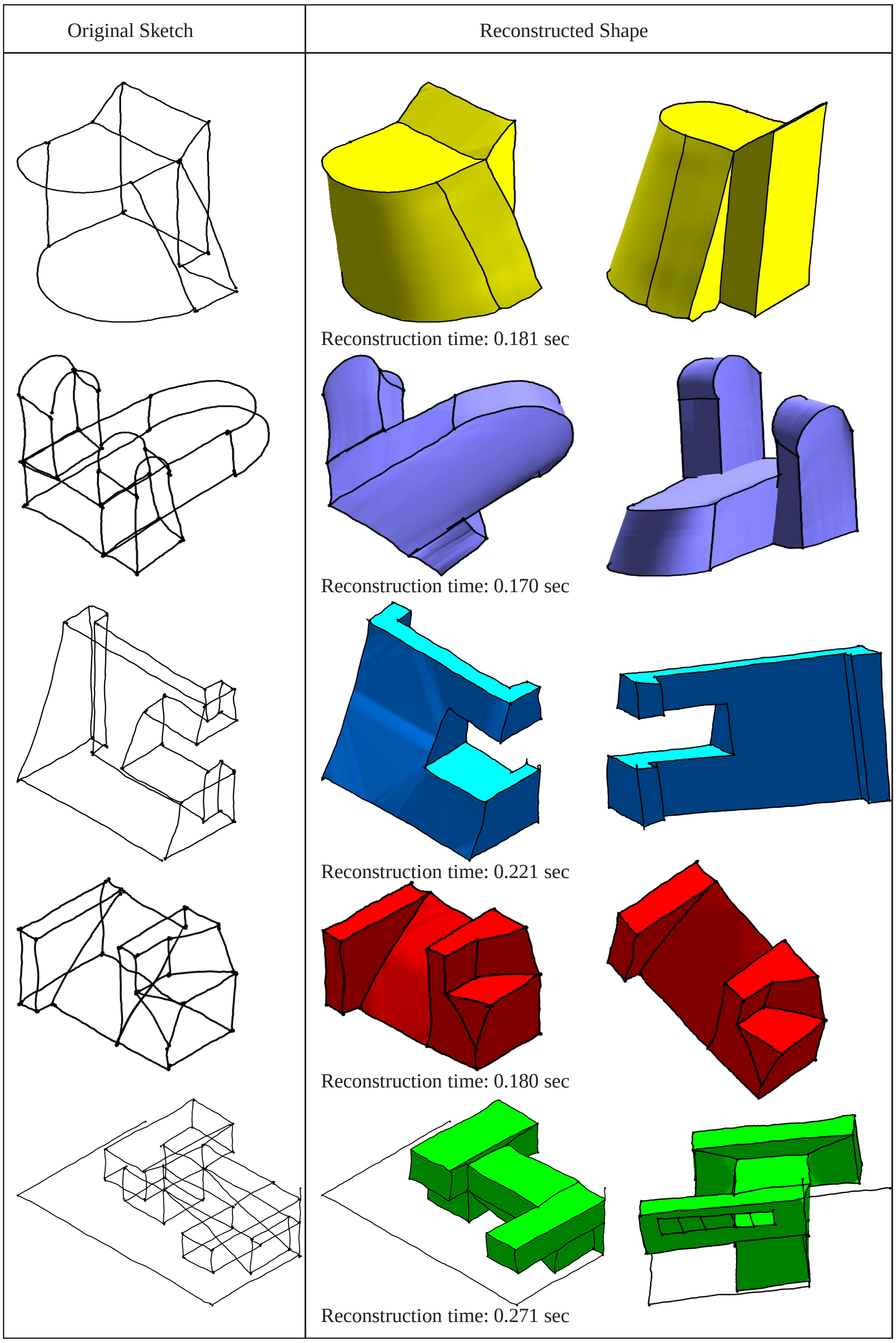

Original Sketch
Reconstructed Shape
Reconstruction time: 0.181 sec
Reconstruction time: 0.170 sec
Reconstruction time: 0.221 sec
Reconstruction time: 0.180 sec
Reconstruction time: 0.271 sec

Figure 4: *(a) Symmetrical unreconstructed straight-line 2D sketches and the accompanying reconstructed 3D shapes from 2 viewpoints. Reconstruction times are given for a Pentium 4 M 1.7Ghz Tablet PC.*

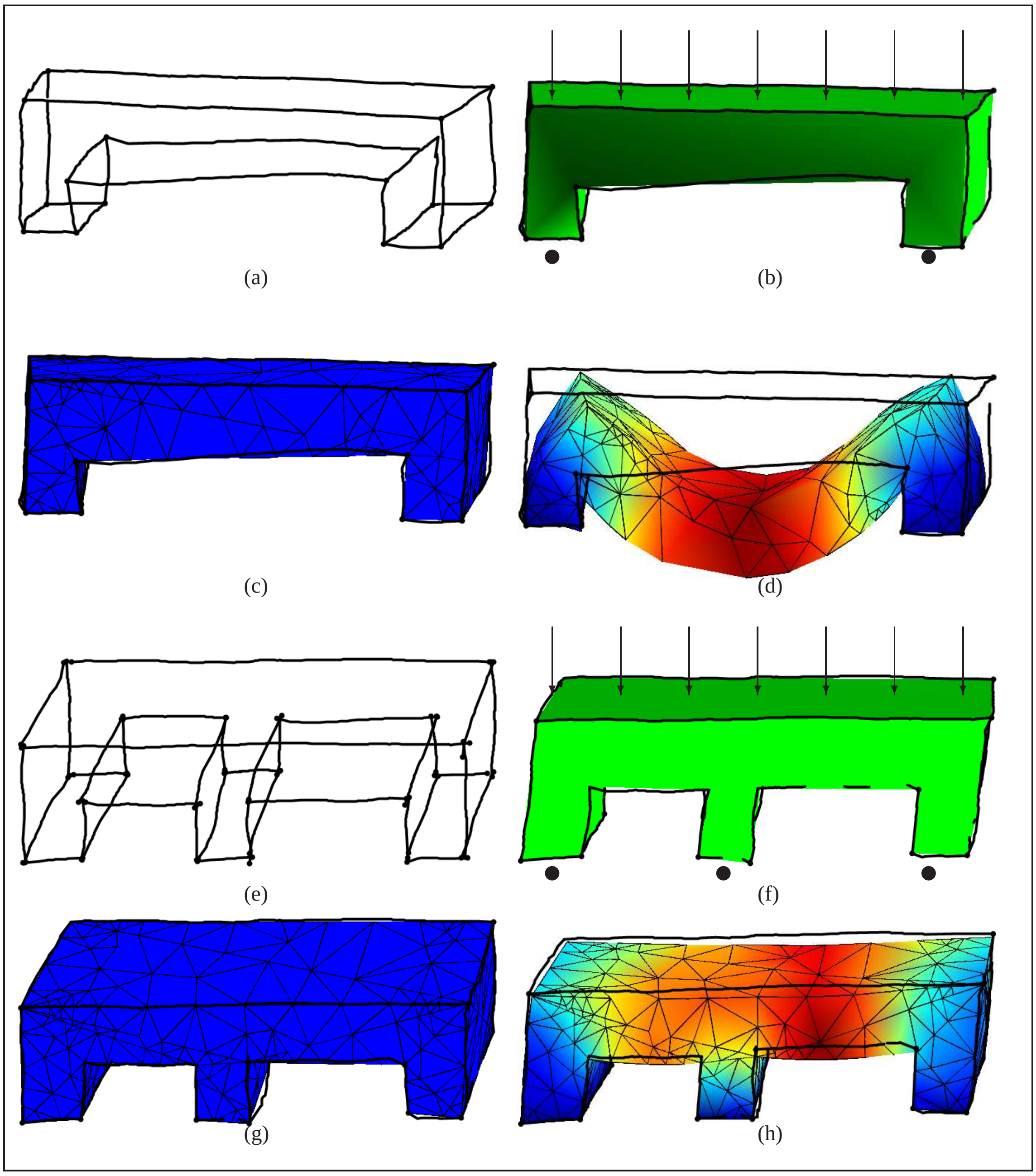


Figure 5: *A an example of a simple, iterative reconstruction and analysis session. (a) An initial sketch of a table-like structure with two legs (b) the reconstructed manifold solid. Boundary faces on the bottom of each leg are indicated by dark circles. A 3N downward force is applied to the topmost face. The coefficients of elasticity are relatively large, allowing for a high degree of deformation (c) the object's tetrahedral solid mesh superimposed over the original strokes (d) the tetrahedral mesh following finite element simulation to determine the displacement produced by the 3N force. The original sketch lines are still visible. Dark blue colors indicate nodes with minimal displacement, orange colors indicates nodes with the maximum magnitude displacement (e) a modified version of the original sketch with an additional bracing leg (f) the reconstructed manifold solid with boundary faces on the bottom of each leg. A 3N downward force is applied to the topmost face (g) the object's tetrahedral solid mesh superimposed over the original strokes (h) the tetrahedral mesh following FEA. Note that the additional leg considerably reduces deformation, and that peak deformation occurs on the longer of the two inter-leg sections.*

ing algorithm, which are discussed in detail in [Si05]. Construction of a practical mesh requires limiting the number of samples used to represent each stroke to less than 10 to ensure that the manifold surface is not overly complex, and ensuring that the boundary of each face is consistent with the boundary of any adjacent face so that the manifold surface can be constructed by linking the points along the boundary of each triangulated face. In practice, these restrictions do not limit the usefulness of the FEA simulation as an analysis tool for sketched shapes.

The FEA solution to the elasticity problem requires that at least one face of the manifold solid serve as a boundary that will remain stationary when forces are applied to the solid. Forces may be applied to one or more faces. Faces may be selected directly by clicking with the pen, after which the forces and boundary information are specified with a dialog. The analysis itself is performed using GetFem++ [RP05], a publicly available finite element library. The results of the analysis are superimposed over the strokes making up the sketch, allowing users to very quickly interpret the results and modify the object accordingly.

5.1. Results

The reconstruction algorithm performed best on sketches that exhibited strong orthogonal trends, and whose strokes were highly correlated with the underlying axis system, a class of sketches that includes many engineering diagrams. Because the computationally intensive nonlinear optimization is used only to reconstruct the main axis system, rather than depths of all sketch vertices directly, the algorithm can process sketches composed of 50 or more strokes in interactive time on a Pentium 4 Tablet PC notebook computer. An example demonstrating the reconstruction of several sketches incorporating both straight and curved strokes is given in Figure 4. Examples of non-trihedral vertices can be seen in the first and fourth objects.

An example FEA session is shown in Figure 5. Users were quickly able to design parts and verify their structural integrity under various loads. In practice, the time required to perform the finite element analysis proved to be the limiting factor. Finite element analysis scales to the third order with the number of tetrahedra required to represent the manifold solid, which in turn scales with the complexity of the object. Very simple shapes could be analyzed in several seconds. As complexity increased, however, the iterative analysis became time consuming and the design flow. Though finite element analysis is necessarily computationally complex, analysis time can be decreased by using only coarse approximations to the sketched solid during the iterative design phase.

6. Conclusions and Future Work

This paper presented a pen-based sketching system for progressively constructing and analyzing 3D objects using freehand sketches. Sketches representing the orthographic projection of a 3D object onto a sketch plane are treated as graphs consisting of vertices connected by the sketch strokes. The 3D object is reconstructed from this sketch, after which the object's faces are identified and triangulated.

The system is implemented with a consistent pen-based interface that mimics pencil and paper sketching, and can reconstruct sketches of up to 50 strokes in interactive time. Users can add additional strokes, or erase strokes, and sketch directly onto the object's faces. The reconstructed object can be submitted to a finite element analysis in order to investigate its physical properties, after which it can be modified if necessary. Other analysis codes can easily be used in place of FEA.

This work was performed to investigate the use of 3D sketching combined with engineering analysis as a conceptual design exploration tool. Future investigation will attempt to further elucidate the types of design tasks for which this form of interface is advantageous. Additional research will also be dedicated to improving the reconstruction algorithm, including extensions to sketches that cannot be described by a single, connected graph, and sketches with multiple axis systems.

7. Acknowledgements

This research has been supported in part by US National Science Foundation (NSF) Grant IIS-0428133 and by a Microsoft University Relations research grant for Tablet Computing.

References

[CCCP04] COMPANY P., CONTERO M., CONESA J., PIQUER A.: An optimization-based reconstruction engine for 3D modeling by sketching. *Computers & Graphics 28* (2004), 955–979.

[CPC04] COMPANY P., PIQUER A., CONTERO M.: On the evolution of geometrical reconstruciton as a core technology to sketch-based modeling. In *EUROGRAPHICS Workshop on sketch-based interfaces and modeling* (2004).

[Dav02] DAVIS R.: Position statement and overview: Sketch recognition at MIT. In *AAAI Sketch Understanding Symposium* (Stanford, CA, 2002).

[Fuk98] FUKUI Y.: Input method of boundary solid by sketching. *Computer aided design 20*, 8 (1998), 434–440.

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: sketching interface for 3d freeform design. In *Proceedings of SIGGRAPH* (August 1999), pp. 409–416.

[KML04] KANG D. J., MASRY M., LIPSON H.: Reconstruction of a 3d object from main axis system. In *AAAI Fall Symposium Series: Making Pen-Based Interaction Intelligent and Natural* (2004). To Appear.

[LB90] LAMB D., BANDOPDHAY A.: Interpreting a 3D object from a rough 2D line drawing. In *Proceedings of Visualization '90* (1990), pp. 59–66.

[LS96] LIPSON H., SHPITLANI M.: Optimization-based reconstruction of a 3D object from a single freehand line drawing. *Journal of Computer Aided Design 28*, 8 (1996), 651–663.

[LS00] LIPSON H., SHPITLANI M.: Conceptual design and analysis by sketching. *Journal of AI in Design and Manufacturing (AIEDEM) 14* (2000), 391–401.

[LS02] LIPSON H., SHPITLANI M.: Correlation-based reconstruction of a 3D object from a single freehand sketch. In *AAAI Spring Sumposium on Sketch Understanding* (2002), pp. 99–104.

[Mac73] MACKWORTH A. K.: Interpreting pictures of polyhedral scenes. *Artificial Intelligence 4* (1973), 121–137.

[PTVF03] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical Recipes in C++: The Art of Scientific Computing*, second ed. Cambridge University Press, 2003.

[RP05] RENARD Y., POMMIER J.: Getfem++: a generic finite element library in c++, version 1.7, 2005. http://www-gmm.insa-toulouse.fr/getfem.

[SC04] SHESH A., CHEN B.: SMARTPAPER–an interactive and user-friendly sketching system. In *Proceedings of Eurographics 2004* (August 2004).

[SDS98] STAHOVICH T. F., DAVIS R., SCHROBE J.: Generating multiple new designs from a sketch. *Artificial Intelligence 104* (1998), 211–264.

[She96] SHEWCHUK J. R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, Lin M. C., Manocha D., (Eds.), vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1996, pp. 203–222. From the First ACM Workshop on Applied Computational Geometry.

[Si05] SI H.: Tetgen, a quality tetrahedral mesh generator and three-dimensional delaunay triangulator, 2005. http://tetgen.berlios.de/.

[SL97] SHPITLANI M., LIPSON H.: Classification of sketch strokes and corner detection using conic sections and adaptive clustering. *ASME Journal of Mechanical Design 119*, 2 (1997), 131–135.

[VMS04] VARLEY P. A. C., MARTIN R. R., SUZUKI H.: Making the most of using depth reasoning to label line drawings of engineering objects. In *9th ACM Symposium on Solid Modeling and Applications SM'04* (2004), Elber G., Patrikalas N.,, Brunet P., (Eds.), pp. 191–202.

[VTMS04] VARLEY P., TAKAHASHI Y., MITANI J., SUZUKI H.: A two-stage approach for interpreting line drawings of curved objects. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2004), Hughes J., Jorge J., (Eds.), pp. 117–126.

[Wei87] WEI X.: *Computer Vision Method for 3D Quantitative Reconstruction from a Single Line Drawing*. PhD thesis, Department of Mathematics, Beijing University, China, 1987. (in Chinese; for a review in English see Wang and Grinstein, 1993).

[WG89] WANG W., GRINSTEIN G.: A polyhedral object's CSG-Rep reconstruction from a single 2D line drawing. In *Proceedings of 1989 SPIE Intelligent Robots and Computer Vision III: Algorithms and Techniques* (1989), vol. 1192, pp. 230–238.

[ZHH96] ZELEZNIK R., HERNDON K., HUGHES J.: SKETCH: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH* (1996), pp. 163–170.

Sketching and Education

NEXT-GENERATION EDUCATIONAL SOFTWARE

Why We Need It & a Research Agenda for Getting It

By Andries van Dam, Sascha Becker, and Rosemary Michelle Simpson

The dream of universal access to high-quality, personalized educational content that is available both synchronously and asynchronously remains unrealized. For more than four decades, it has been said that information technology would be a key enabling technology for making this dream a reality by providing the ability to produce compelling and individualized content, the means for delivering it, and effective feedback and assessment mechanisms. Although IT has certainly had some impact, it has become a cliché to note that education is the last field to take systematic advantage of IT. There have been some notable successes of innovative software (e.g., the graphing calculator, the Geometer's Sketchpad, and the World Wide Web as an information-storage and -delivery vehicle), but we continue to teach—and students continue to learn—in ways that are virtually unchanged since the invention of the blackboard.

Andries van Dam is Professor of Computer Science and Vice President for Research at Brown University. He has been working on electronic books with interactive illustrations for teaching and research since the late 1960s. Sascha Becker is a multidisciplinary software designer, developer, user, and critic, currently working as a research programmer in the Brown University Department of Computer Science. Rosemary Michelle Simpson is an information structures designer currently working in the Brown University Department of Computer Science. This article is drawn from a presentation delivered at the 2004 Aspen Symposium of the Forum for the Future of Higher Education.

Illustration by Steve McCracken, © 2005

There are many widely accepted reasons for the lack of dramatic improvement:

- Inadequate investment in appropriate *research and development* of authoring tools and new forms of content
- Inadequate investment in the creation of new dynamic and interactive *content* that takes proper advantage of digital hypermedia and simulation capabilities (as opposed to repurposed print content) at all educational levels and across the spectrum of disciplines
- Inadequate investment in appropriate IT *deployment* in schools (e.g., although PCs are available in K-12, there are too few of them, they are underpowered, and they have little content beyond traditional "drill-and-kill" computer-aided instruction, or CAI; at the postsecondary level there is more availability of computers and software, plus routine use of the Internet, but still a dearth of innovative content that leverages the power of the medium)
- Inadequate support for *teacher education* in IT tools and techniques and for the incorporation of IT-based content into the curriculum
- The general *conservatism* of educational institutions

Despite this disappointing record, we remain optimistic. The dramatic advances in hardware technology, especially during the last decade, provide extraordinary new capabilities, and the desire to "do something" to address the need for lifelong, on-demand learning is finally being widely recognized. The ubiquity and accessibility of the Internet has given rise to a new kind of learning community and environment, one that was predicted by Tim Berners-Lee in his 1995 address to the MIT/Brown Vannevar Bush Symposium[1] and that John Seely Brown elaborated into the rich notion of a learning ecology in his seminal article "Growing Up Digital: How the Web Changes Work, Education, and the Ways People Learn."[2] There is great hope that this emergent learning environment will in time pervade all organizations, binding learners and teachers together in informal, ever-changing, adaptive learning communities.

Here we will first recapitulate some well-known technology trends that make

Despite predictions that we will hit a technological wall in the coming decade, new advances repeatedly push any such wall out into the indefinite future.

current platforms so exciting, and then we will briefly discuss leveraging this technology into highly desirable forms of learning. Next we will examine an IT-oriented education research agenda prepared by a consortium called the Learning Federation and will present some promising educational software experiments being conducted at Brown University. Finally we will describe an as-yet-unrealized concept called "clip models": simulation-based interoperable families of components that represent multiple levels of explanatory power and simulation fidelity designed to be assembled into systems. We make no attempt here to present a critical review of the entire field of educational software or of its impact. A variety of organizations, journals, and conferences is addressing the uses and impact of IT in education; in particular, EDUCAUSE and its Center for Applied Research (ECAR) provide a good introduction to resources and studies of IT in higher education.

Technology Trends

Exponential advances in computer architecture in the last two decades have enabled the creation of far more compelling and engaging educational software than we could have dreamed of in the Apple II days. Advances in four areas of IT will continuously raise the bar on user experiences: platform power used for computation and graphics/multimedia; networking; ubiquitous computing; and storage.

The commoditization of the necessary *platforms*, a trend described by Moore's "law," is epitomized by supercomputer power and high-end graphics/multimedia capabilities in desktop and laptop computers costing less than $1,000 and even in specialized game boxes costing less than $200. Alan Kay's Dynabook vision can at long last be realized,[3] and even the personalized and ever-evolving "young lady's illustrated primer" from Neal Stephenson's *Diamond Age* will leave the realm of science fiction.[4]

Advances in *networking* enrich user experiences with ubiquitous, always-on, high-bandwidth connections. Already, gigabit networking over local area networks is a reality, and the Internet2 project is creating the core of a massively broadband global network. Wireless connectivity is widely available in both developed and underdeveloped countries and will rapidly increase in bandwidth. The commoditization of bandwidth eliminates physical distance and carrier costs as a factor in providing resources to a worldwide audience.

Ubiquitous computing environments[5] have become commonplace; embedded sensors and microcomputers transform ordinary passive objects into intelligent objects that interact with each other and with us in a great diversity of form factors. Keeping pace with the hardware is ever-more sophisticated software that uses the results of artificial intelligence research in practical applications of the "smart objects."

Compelling experiences and work products alike require data *storage* that is reliable, fast, and inexpensive. A 1.44-megabyte floppy disk cost a few dollars in the early 1990s and couldn't be relied on to keep data safe during a bus ride home from school; today, blank DVDs can be permanently "burnt" with 4.7 gigabytes of data for less than a dollar each, and 20-gigabyte mobile storage devices cost approximately $200, making the Library of Congress accessible anywhere. In addition to raw capacity, however, data needs that must be addressed include security, privacy, validity, and format persistence.

Despite predictions that we will hit a technological wall in the coming decade, new advances repeatedly push any such wall out into the indefinite future. For example, developments in nanotechnology and quantum computing promise new capabilities in all four areas. Indeed, one can only wish that the same exponential improvement curves that apply to hardware also applied to software and content creation. Regrettably, both these hugely important areas have shown, at most, modest improvements, and there are no signs of breakthrough technology on the horizon—only continued slow, evolutionary progress. But it is precisely because of the revolutionary improvements in hardware that we can create breakthrough experiences and content. Now is the time to mount such an effort.

IT in Education: Appropriate Role?

So, what is the appropriate role for IT in education, in the broadest sense? As always, IT's role is to augment (not to replace) the teacher, to provide human-centered tools that encourage and support adaptability and flexibility, and to enable appropriate modes of learning (e.g., small team interaction and not just individual task performance).[6] Principles such as situated, active learning (i.e., learning by doing rather than just by listening)—principles that foster interactive involvement of the learner with the educational materials—are well supported by current technology trends. However, one size does *not* fit all in educational software. Unless new tools allow exploration at multiple levels of detail and accommodate diverse learning styles,[7] they will be just as limited as ordinary textbooks. But this is easier to say than to do: there is no collective experience in authoring at multiple levels of detail and multiple points of view. Such authoring requires the development of skills and tools of far greater power than we have experience with to date.

The most important task in the application of IT to education is to author stimulating content that is as compelling as "twitch games" or even as strategy games appear to be. New content dropped into existing curricula typically shows no improvement in outcomes; we must also redefine curricula to support learner-centered, on-demand exploration and problem-solving, and we must break down traditional disciplinary boundaries. We must also train educators to take advantage of these new capabilities. This will require massive investment, on a scale we have not encountered heretofore. This content creation, curricula adaptation, and educator training will also require a long period of experimentation, as well as tolerance for the false starts that are an inevitable part of all innovation processes. For example, whereas classical CAI was thought to hold great promise in the 1960s, its applicability turned out to be rather limited; the same held for Keller plan self-paced instruction and other innovations that are in fact now reappearing in different guises.

Content and curriculum alone are not sufficient. We must provide support for all aspects of learning, in both formal and informal education, not just in schools but in all venues, ranging from the home to the office and the factory floor—anyplace where learners gather, singly or in groups. In addition, we must provide support for all aspects of this process, including course administration (as WebCT and its competitors are doing), continuous assessment (a deep research problem), and digital rights management (still a very contentious and difficult societal, commercial, and research problem).

Returning to the topic of content, we must develop software that accommodates many different human-computer interactions, from single-user to massively collaborative multi-user. Genres must be equally diverse, from cognitive tutors such as CMU's Pump Algebra Tutor, or PAT (http://act.psy.cmu.edu/awpt/awpt-home.html), to simulation- and rule-based interactive models (microworlds), massive multiplayer games, and robots constructed and programmed to carry out specified tasks.

Even before such adaptive, personalized content is widely available, we must also rethink learning environments—that is, envision profound structural change at all levels of education to accommodate the kind of learning the new content facilitates. Early examples of experiments in structural change include Eric Mazur's Peer Instruction Physics (http://mazur-www.harvard.edu/education/educationmenu.php), the RPI Studio Model (http://ciue.rpi.edu/), and the Virginia Tech Math Emporium (http://www.emporium.vt.edu/). Both the RPI Studio Model and the Virginia Tech Math Emporium change not just the structure of the educational process but even the facilities required. This is just the beginning of rethinking college and university instruction from the ground up. In addition, distance learning, as embodied by the Open University (http://www.open.ac.uk/) and the University of Phoenix (http://www.phoenix.edu/), shows that non-campus-based instruction can work, although the materials used are not particularly innovative as yet. On a cautionary note, we should add that there have been many recent failures in commercial distance learning. Traditional colleges and universities with classroom/laboratory instruction will not soon be replaced, although they will certainly be augmented by newer, IT-based forms of learning.

Unless new tools allow exploration at multiple levels of detail and accommodate diverse learning styles, they will be just as limited as ordinary textbooks.

The Computing Research Association's "Grand Challenge 3"—"Provide a Teacher for Every Learner" (http://www.cra.org/reports/gc.systems.pdf)—describes some of the genres mentioned above, but the most important conclusion of that report, reflected in its title, is that by providing powerful tools, we offer the opportunity to rethink the relationship between teachers and learners. The appropriate use of IT will empower teachers to enhance their mentoring roles and can supplement such teacher support with peer and computer-based mentoring and tutoring to provide students with essentially full-time, on-demand, context-specific help. Building domain-specific mentoring, tutoring, and question-answering is scarcely a solved problem and will require a very significant research and development (R&D) effort.

Getting There: Learning Federation Research Roadmaps

To better understand the issues involved and to direct a focused research investment effort, Andries van Dam helped to found a small steering group that has proposed the creation of a nonprofit, industry-led foundation, called the Learning Federation (http://www.thelearning federation.org/), modeled on the highly successful Sematech Consortium (http://www.sematech.org). The Learning Federation is a partnership joining companies, colleges and universities, government agencies, and private foundations whose purpose is to provide a critical mass of funding for long-term basic and applied pre-competitive research in learning science and technology. This research, to be conducted by interdisciplinary teams, is meant to lead to the development not only of next-generation authoring tools but also of exemplary curricula for both synchronous and asynchronous learning.

The Federation's first task was to produce a Learning Science and Technology R&D Roadmap. This roadmap describes a platform-neutral research plan to stimulate the development and dissemination of next-generation learning tools, with an initial focus on postsecondary science, technology, engineering, and mathematics. The component roadmaps, which address five critical focus areas for learning science and technology R&D, were developed using expert input from companies, colleges and universities, government research facilities, and others with unique expertise during a series of specialized workshops, consultative meetings, and interviews. Each roadmap provides an assessment of the R&D needs, identifies key research questions and technical requirements, and specifies long-term goals and three-, five-, and ten-year benchmarks—the roadmap to the long-term goals. The following sections give the abstracts from the component roadmaps, along with the URLs where the full PDF files may be downloaded.

Instructional Design: Using Games and Simulations in Learning

"Learning environments that provide learners opportunities to apply their knowledge to solve practical problems and invite exploration can lead to faster learning, greater retention, and higher levels of motivation and interest. Unfortunately, these learning strategies are rarely used today because they are difficult to implement in standard classroom environments. Expected improvements in technology have the potential to significantly reduce the cost and complexity of implementing learning-by-doing environments. The combined forces of high-powered computing, unparalleled bandwidth, and advances in software architecture are poised to make realistic gaming and simulation environments more feasible and economical. Because these tools will be increasingly available, it is important to understand appropriate contexts and methods for implementation. The challenge is to understand how the tools should be used, with whom and for what?" See (http://www.thelearning federation.org/instructional.html).

Question Generation and Answering Systems

"Question generation is understood to play a central role in learning, because it both reflects and promotes active learning and construction of knowledge. A key challenge to researchers and practitioners alike is to find ways to facilitate inquiry by taking advantage of the benefits offered by emerging technologies. Further exploration is needed in the realm of intuitive interfaces that allow the learner to use spoken language, or coach the learner on how to ask questions, tools to enable answers to learners' questions—including linking learners to real people, as well as the creation of intelligent systems that ask the learner questions or present problems that require major attention and conversation." See (http://www.thelearningfederation.org/question.html).

Learner Modeling and Assessment

"Assessment generates data for decisions such as what learning resources to provide individual learners and who to select or promote into particular jobs. These decisions are only as valid as the data and interpretations that are available. Ideally, every educational decision-maker, from teacher to human resource director, would have access to real-time valid data to make a decision about an individual, group, or program. There is a critical need to articulate more precisely and reach consensus on many of the theoretical underpinnings and models that drive our assessment practices." See (http://www.thelearningfederation.org/learner.html).

Building Simulations and Exploration Environments

"Research has demonstrated that simulation environments are powerful learning tools that encourage exploration by allowing learners to manipulate their learning experience and visualize results. Simulations used in academic settings can enhance lectures, supplement labs, and engage students. In the workplace, simulations are a cost-effective way to train personnel. Despite important successes in the use of simulation and synthetic environments, there are still a number of limitations to current applications and programs. The goal of this R&D effort is to make simulations and synthetic environments easier to build and incorporate into learning environments." See (http://www.thelearningfederation.org/building.html).

Integration Tools for Building and Maintaining Advanced Learning Systems

"As specifications and standards have been developed to support web-based system directed learning systems, the means for creating interoperable and robust instructional content have emerged. However, these specifications have defined a technically complex infrastructure that is unfriendly to instructional designers. Designers should be able to focus entirely on content, the needs of students, and instructional theory and not on the mechanics of the software. A variety of authoring and integration tools are needed to make it easy to identify software resources and to combine these resources into a functioning system." See (http://www.thelearningfederation.org/integration.html).

Beginnings: Brown University Projects

Microworlds

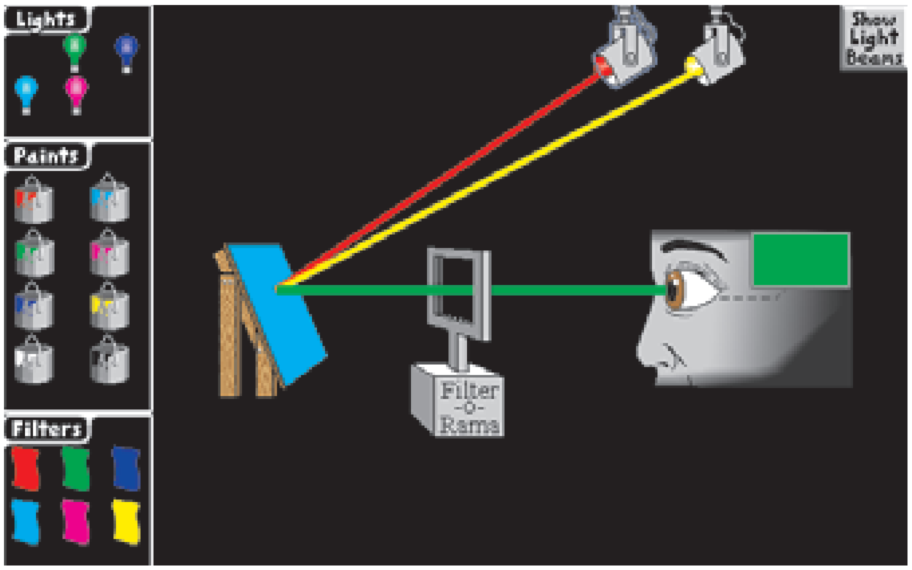
At Brown University, partially inspired by Kay's powerful Dynabook vision,[8] we have been particularly interested in building simulation and exploration environments, often referred to as *microworlds*. These can be used to teach abstract concepts, such as Newton's laws and the Nyquist limit for signal processing, and skills, such as spatial visualization and integration by parts. The combination of the Web and Java applets has resulted in a proliferation of applets across a broad range of subjects.

For the last decade, inspired by many applets on the Web, we have been developing computer graphics teaching applets called Exploratories (see Figure 1). These highly interactive, simulation-based applets are built from reusable software components and can be embedded in a Web-based hypermedia environment or used as downloadable components in a wide variety of settings. Their design builds on a geometric structure to simulate behavior. Users can control the environment and experiment with different variables through interface-exposed parameters. To date we have over fifty Exploratories in computer graphics alone (http://www.cs.brown.edu/exploratories/).

Gesture-Based Tablet PC Applications

In an increasingly ubiquitous world of iPods, digital camera cell phones, and wireless everything, the WIMP ("Windows, Icons, Menus, and Pointers") interface has been gradually augmented by post-WIMP interface techniques as mobile users experience the convergence of media and communication technologies. The laptop workhorse has been expanding its capabilities as well with the advent of the Tablet PC and its pen-based interface. Until the last decade, pen-based technology was not good or cheap enough for widespread use. Gesture recognition, handwriting recognition, and digitizer technology have signifi-

Figure 1. Exploratory on color mixing, highlighting the differences between mixing light and mixing paint



cantly improved in performance and availability in the last few years; now applications can be developed and deployed at retail scope.

Just as with microworlds, the Brown computer graphics group has been experimenting with gesture-based user interfaces and applications for many years[9] and is currently developing gestural interfaces for the Tablet PC. Two of these are MathPad^2 and ChemPad.

MathPad^2. Mathematical sketching is a pen-based, modeless gestural interaction paradigm for mathematics problem-solving. Whereas it derives from the familiar pencil-and-paper process of drawing supporting diagrams to facilitate the formulation of mathematical expressions, users can also leverage their physical intuition by watching their hand-drawn diagrams animate in response to continuous or discrete parameter changes in their written formulas. Implicit associations that are inferred, either automatically or with gestural guidance, from mathematical expressions, diagram labels, and drawing elements drive the diagram animation.

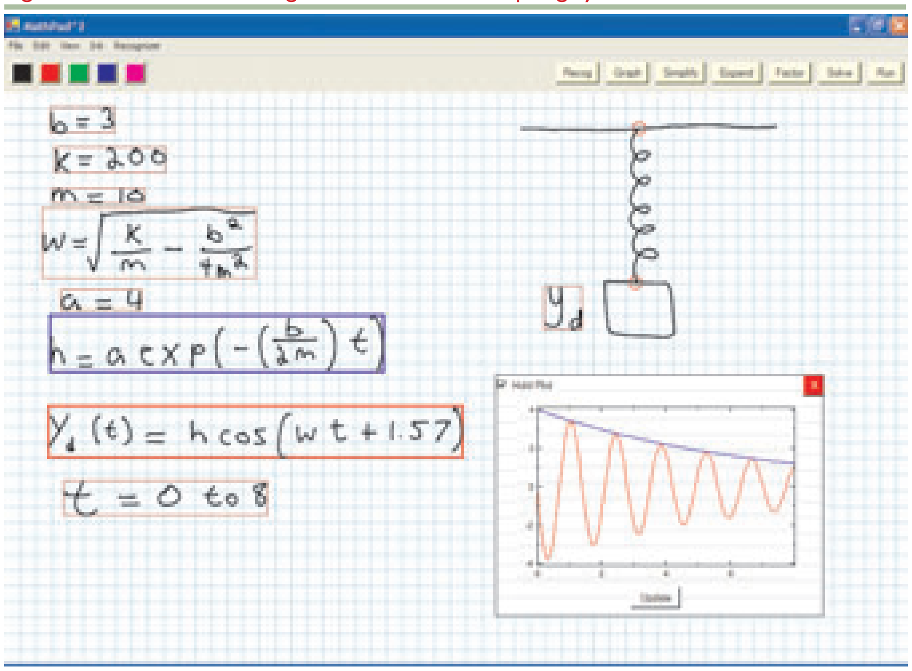
The modeless nature of mathematical sketching enables users to switch freely between modifying diagrams or expressions and viewing animations. Mathematical sketching can also support computational tools for graphing, manipulating, and solving equations.

The MathPad^2 mathematical sketching application currently uses MATLAB as its underlying math engine and provides a fully gestural interface for editing. Expressions can be deleted, edited, and rerecognized in a fully modeless operation (see Figure 2).

ChemPad. Organic chemistry is the study of the structure and function of carbon-based molecules. These molecules have complex, three-dimensional structures that determine their functions. Ideally, students would do all their thinking and drawing in three dimensions (3D), but whiteboards and paper notebooks support only 2D structures and projections of 3D and higher-dimensional structures. To compensate, organic chemists use a complicated 2D schematic notation for indicating the spatial arrangement of atoms in a molecule. With practice and insight, beginning chemists can develop the ability to look at such a 2D schematic description of a molecule and automatically construct a mental model of the 3D structure of that molecule.

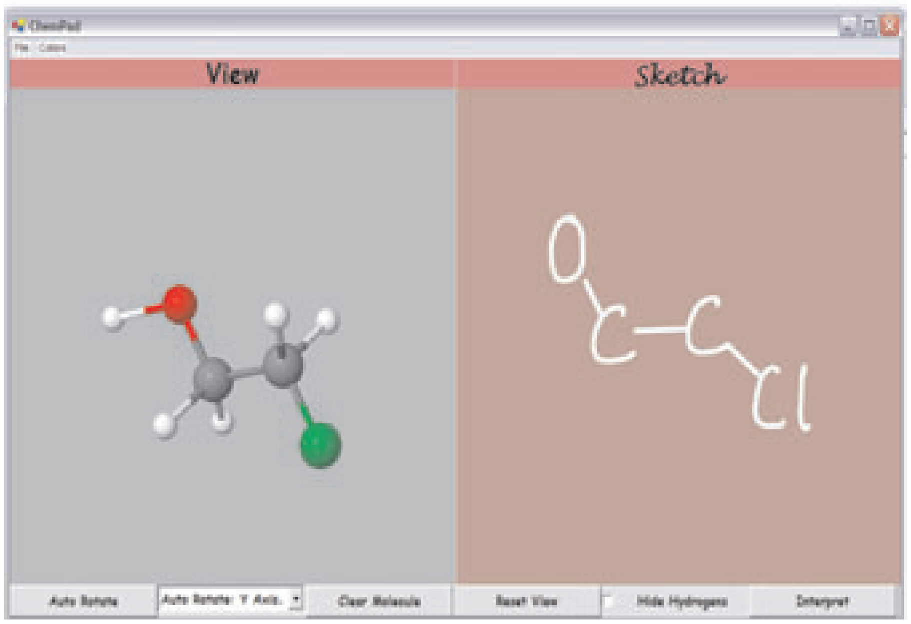
Teachers of organic chemistry identify this spatial understanding as a key determinant of whether students will succeed in organic chemistry. We have been designing and developing a software project, ChemPad, whose purpose is to help organic chemistry students develop an understanding of the 3D structure of molecules and the skill of constructing a 3D mental model of a molecule that matches a 2D diagram (see Figure 3). ChemPad fosters this understanding by allowing the student to sketch a 2D diagram and then to see and manipulate the 3D model described by the diagram.

Figure 2. MathPad^2 sketching interface of a mass spring system



A pen-based interface is particularly appropriate for drawing organic chemistry molecules because the existing software tools in this area are difficult to learn and use, which places them out of the reach of most students. Drawing with pen and paper, though, is not entirely satisfactory; it is difficult to produce neat drawings, and it is difficult to erase and correct errors neatly. ChemPad addresses both these issues, with a simple interface that mimics drawing on paper and with a "beautify" function that "tidies up" a student's drawing. ChemPad also provides validity-checking; many of the structures that beginning students draw do not describe physically possible molecules. Unlike paper and pencil, ChemPad can detect and indicate certain kinds of errors. One possible extension of this approach would be to add simulation capabilities so that the static ball-and-link 3D diagrams can start to approximate the actual dynamics of molecular interaction.

Figure 3. The ChemPad program displays an organic molecule, 2-chloro-ethanol. On the right side, the user has sketched the non-hydrogen atoms of the molecule using standard organic chemistry conventions. On the left side, the program has generated and rendered a 3D view of the molecule. The user is able to rotate the 3D representation and examine it from arbitrary points of view.



Limitations of Our Current Work
Although microworlds have been useful adjuncts to the undergraduate computer graphics course, they fall short of the goals for a far more ambitious vision. Microworlds and Exploratories are restricted to single concepts with a small set of parameters. However, because they are component- and parameter-based, they illustrate some of the fundamental principles that will be essential in fully functioning clip-model environments, and they open possibilities for evolving even more flexible structures. The combination of fluid and multi-POV (point of view) hypermedia information structures with component-based software architectures may provide a foundation on which we can build.

Tablet-PC-based gestural interfaces to applications are underdeveloped because the state-of-the-art in robust user-independent gesture recognition is still primitive. Furthermore, gesture sets are anything but self-disclosing, and they take considerable time to learn. Finally, our experiments thus far are essentially single-user in their orientation and don't facilitate a collaborative, team-based approach to learning. The next section addresses some of the issues involved in designing software that can be adapted to multiple needs, users, and levels of detail.

Clip Models: A Proposal for Next-Generation Educational and Research Software

Over forty years ago, Jerome Bruner proposed a radically new theory of education: "Any subject can be taught effectively in some intellectually honest form to any child at any stage of development."[10] Although many people have disputed the more extreme claims attached to that hypothesis, it is an admirable goal. One way to implement it is through the "spiral approach to learning," common to formal education, in which a learner encounters a topic multiple times throughout his or her education, each time at an increasing level of sophistication. Furthermore, at any stage, the learner should be able to mix and match educational modules at different levels of sophistication within the same general topic area. Simpler modules can offer overviews of a subject for review or provide context when the intent is to go more deeply into related topics.

The kinds of modules we are most interested in here are simulation- or rule-based modules that help create explorable models of subsystems, which can be linked into increasingly higher-level subsystems. Such modules can help simulate most aspects and components of the natural and man-made worlds. We will focus here on simulating subsystems of the human body at all levels, from the molecular to the cellular to the gross anatomical. Each subsystem of the human body must then be simulated at a level appropriate to the (educational) purpose. There is not just a single model/simulation for each component of the system (e.g., the heart or lungs) but a family of models/simulations varying in explanatory power and simulation fidelity—not to mention, ideally, in the learning style it is to match. Furthermore, since subsystems interact with each other, the models and their underlying simulations must be able to interoperate. We summarize the properties of these types of models with the term "clip models": simulation-based families of components that represent multiple levels of explanatory power and simulation fidelity designed to interoperate and to be assembled into systems. In particular, unlike clip art, which represents only images, clip models emphasize behavior, interaction/exploration, and interoperability.

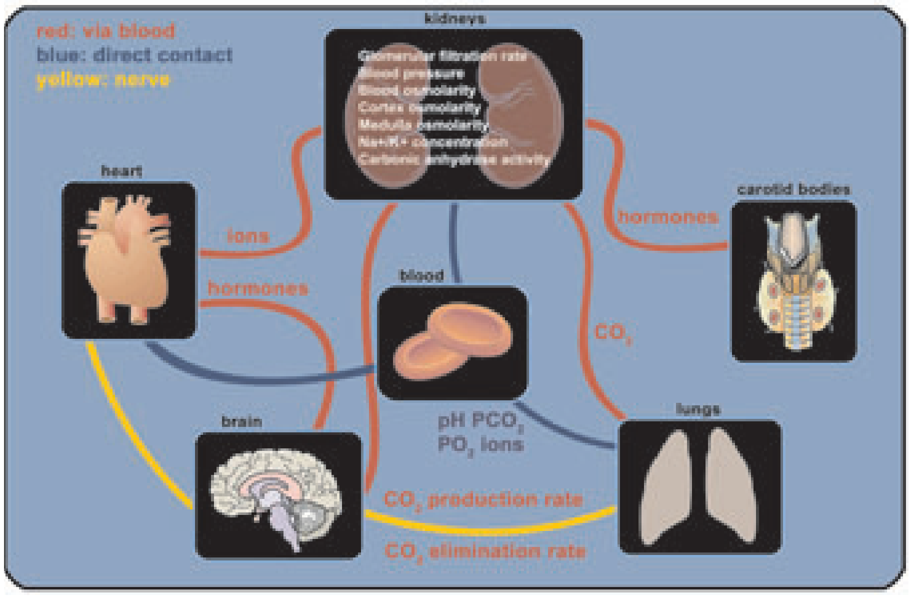
This concept of mix-and-match, multi-LOD (level of detail) models poses huge challenges to would-be implementers. The inherent challenges of building multi-resolution, multi-view, multi-complexity interoperating simulations have not yet been confronted because most simulation efforts have been standalone projects. In the same way, repositories of learning objects have stored only objects at a single level of explanatory power, and component frameworks in use by software developers have not been designed with the complexity of interoperation between components at different levels of detail in mind.

A Biological Scenario

The concept of clip models can best be explained with an illustration. The details don't really matter; the important thing is to note the complexity of the relationships between simulated components and the potential applications of this family of simulations for education and research.

Figures 4, 5, and 6 are an abstract representation of how the heart and vascular systems interact with other systems used by the human body to regulate oxygenation—that is, to make sure that we have enough oxygen in our blood, and not too much. This homeostasis, crucial for maintaining life, relies on several interconnected mechanisms, including functions of the kidney, the heart, the brain, the lungs, and chemoreceptors located throughout the body.

Figure 4. Elements in the system for control of oxygenation in the human body



All of these systems are connected by the blood, and each of them plays a slightly different role. The blood's behavior as an oxygen carrier is determined by macro- and microscopic factors, from the fraction of blood composed of red blood cells, visible to the human eye in a test tube, to the electrostatic attraction between oxygen and hemoglobin molecules at a scale too fine to be seen with any microscope. Hormones that regulate the actions of the kidney, the heart, and the lungs are generated by the kidney, the brain, the lungs, and the endocrine system, including endocrine glands located in or near the brain.

The kidneys monitor and correct various characteristics of the blood. To understand their function, we must first perceive them at a coarse level as organs that produce urine by filtering the blood. At this coarse scale, we must understand only that blood is delivered to and accepted from the kidneys by large blood vessels and that the kidneys produce urine; this level of understanding is appropriate for an elementary school student. To understand how the kidneys perform this function, a more advanced learner must examine their structure at a much finer scale, the scale of the nephron, of which each kidney has millions.

The heart rate and the volume of blood ejected per heartbeat control the rate of

Figure 5. A more detailed view of the kidney, a cross section with some of the internal structures

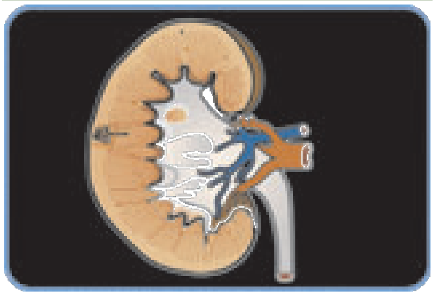
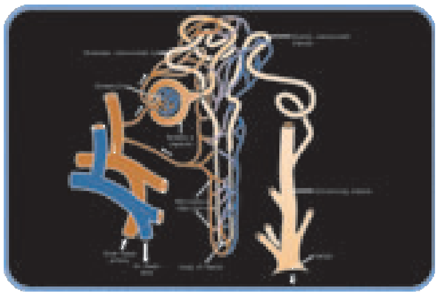
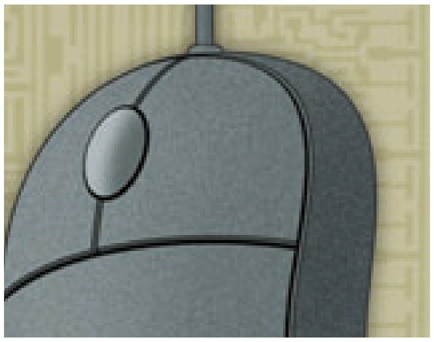


Figure 6. A more detailed view of a nephron, the microscopic functional unit of the kidney





The varied needs of audiences at many different levels of sophistication preclude a one-to-one mapping between a given concept and a single clip model.

distribution of oxygen to the body; these factors are jointly controlled by the brain and by the heart itself. The lungs' respiration rate and inhalation volume are controlled by the brain via nerves, but the oxygen absorption and the carbon dioxide elimination rate are also determined by the concentration of these gases in the blood. The carotid bodies, in the neck, monitor the oxygen concentration in the blood and inform the brain when more oxygen is needed. The brain then issues hormones and neural signals, carried by the blood and the nervous system to other organs, which adjust their operation to correct the problems.

In our simplified illustration, we show three levels of detail for examining the roles of the heart and the kidneys in homeostatic oxygenation. Figure 5 illustrates more detail of the kidney, a cross section with some of the internal structures. Figure 6 depicts a single nephron in the kidney. A learner can dynamically select which level to examine and may explore different levels at different times, depending on need. Clearly, clip-model exploration by itself may not suffice in an educational context. We must not only embed it in explanatory context (e.g., a hypermedia document) and organizational structure (e.g., a self-paced course) but also enrich it with some type of problem-solving and/or construction activity and continuous feedback and assessment.

Clip-Model Implications

The interconnected mechanisms in the example above, along with the fundamental interconnectedness of system components in all disciplines, cannot all be studied or understood at once, nor can they be understood with purely linear thought. The learner's exploratory process naturally follows the interconnected web of causality, but linear or hierarchical organizations (such as those in most software data sources and all textbooks) force the learner into an artificially linearized exploration. Linearization discourages the cross-disciplinary insights that fuel deep understanding, since it encourages compartmentalized rote knowledge.

As noted above, the varied needs of audiences at many different levels of sophistication preclude a one-to-one mapping between a given concept (such as the circulation of blood through the cardiovascular system) and a single clip model. Thus, instructional designers must think not in terms of creating a single clip model for a given topic but in terms of creating one or more components in a family of interrelated clip models that cover a broad range of explanations and their representations. These models must correctly reflect the ontology and semantics of the subject matter at each point along the multiple axes of age, knowledge level, task, and individual learning. We must also accommodate the variety of learning environments in which such clip models will be presented. These innovative and, by their nature, emergent learning environments must be made available online and on-site, in synchronous and asynchronous and in virtual and real classrooms, servicing both single on-demand learners and collaborative learners, either in impromptu virtual study groups or in formats yet to be defined. Another dimension we need to explore more deeply is team collaboration. Clearly, these requirements present a huge challenge for instructional design and learning technology.

The variety of pedagogical needs that clip models must satisfy is a complicating factor that makes their design immensely harder than that of ordinary components in standard software engineering. A potential approach to thinking about the problem may be to use an extension of the MVC (Model-View-Controller) paradigm of object-oriented programming to describe the necessary interrelationships between these different concept representations. Each concept or real-world object must be represented by a family of models (e.g., the heart as a pump, the heart as a muscle, the heart as an organ in the chest cavity), with widely different degrees of sophistication. Each model supports multiple views (e.g., simplified 3D models, realistic 3D models, 2D schematics, the sound heard through a stethoscope) and, for each view, multiple controllers that may present a learner-chosen UI (user interface) style. Multiple models that must interact, regardless of level of detail and simulation fidelity, geometrically complicate the single-model paradigm of classic MVC.

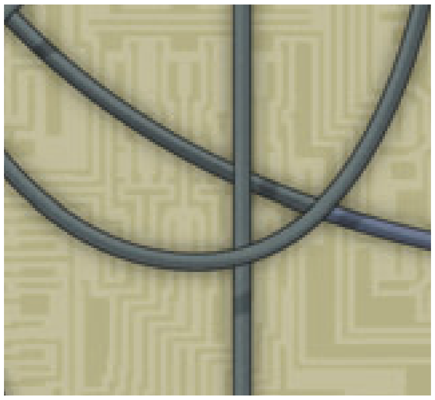
Challenges

This intersection of simulation science, software engineering, ontology (formal naming scheme) building, instructional design, and user interface design forms the technological aspect of this complex problem. In addition to the technological challenges, there are interdisciplinary organizational challenges: building clip models is essentially a new design discipline that requires collaborative teams of experts from cognitive science, social sciences, arts, design, story-telling professions, information structure design, and instructional design—working with teachers, domain experts, simulation scientists, and computer scientists. We can identify challenges for ontological engineering, simulation science, software engineering, and educational design.

As a prerequisite to interoperation, simulation elements must agree on the *ontology* of the conceptual realm they represent. Without a shared ontology or mappings between related ontologies, two simulation elements cannot interoperate if they disagree on the point in the nephron at which the "filtrate" becomes "urine" or the names for the lobes of the liver. Furthermore, the ontology must encompass not just (geometric) structure (anatomy, in the case of biologicial systems) but also behavior (biochemical, electrochemical, biomechanical, etc.), a largely untackled problem, at least for biology. As an additional complication, when you have a single author or a small team of authors writing a single book targeted at a single audience, the domain specification as seen in the definition and relationships of concepts and terms is an important but manageable task. When you expand the context as described above, the situation becomes orders of magnitude more complex. Who will define the master ontology? How will other classification schemes and vocabularies build a correspondence map? Some sort of collaborative approach to ontological engineering will have to be used in order to build an ontology that is acceptable to many members of a given field.

Simulation science does not have a sufficiently flexible framework for wiring together components of a simulation from various providers that were not designed to interoperate from the beginning. How to connect simulations from different problem domains for the same subsystem is still a difficult problem. For example, simulating the heart's operation biochemically, electrochemically, and with computational fluid dynamics, while dealing with flexible (nonrigid) and time-varying geometry and both normal and abnormal behavior, is still a daunting problem. Even with a standard vocabulary, adaptive multi-resolution simulations will be even harder to interoperate; how can they determine at what level of detail to share information? If we are running interactive simulations, should we allow algorithms to run with graceful degradation in order to meet time requirements? What is the nature of such approximations? How can the valid operating ranges of particular simulations be determined? How can the simulations report when they venture beyond those ranges? If these simulations are to be used in real science, as we hope, they must have a mechanism for comparing them with experimental results and for validating their calculations. How can a researcher compare predictions made by a Stanford heart model and by a Harvard heart model? How will a kidney model created by nephrologists at Johns Hopkins share data with a heart model from Stanford or a lungs model from Caltech not purposely designed to interoperate? How can a seventh-grade teacher in Nebraska use a fourth-grade teacher's set of human anatomy clip models as the basis for a more detailed model of the circulatory system?

The *software engineering* challenges range from the commercial and social difficulties of persuading scientists to work within a common model to the software design characteristics that will enable flexibility at all levels. Just as object-oriented programming is a vast improvement in the power of abstraction compared to assembly language, so must the clip-model framework design be to



Building clip models is essentially a new design discipline that requires collaborative teams of experts.

today's component frameworks; the challenges are simply too great to be addressed by incremental advances. A clip-model framework must address various questions. How can simulations ensure that they get the data they need, in the format they need, regardless of the level of fidelity at which connected clip models are running their simulation? For example, how will a heart model cope with changing stiffness in the valves if the valve model is not designed to adjust to stenosis? What protocols will keep all the simulation components synchronized in time, even if one runs in real time and another takes a day to compute a single time-step? Who will maintain the repository of code? Who will control the standards? How can interoperability be preserved when some components are proprietary?

The *educational design* challenges of our vision are the same problems facing today's educational software designers. How can teachers, learners, and scientists find the components that best meet their needs? How does a student figure out that he or she is an auditory learner if the student is bombarded with visual materials? How can users evaluate the reliability, correctness, bias, and trustworthiness of authors of the components?

Progress So Far

Our field is more prepared to address this challenge today than we were twenty years ago. Software engineers used to joke and complain about rewriting a linked list, a common data structure, in every new language, project, and environment. Since then, library standardization (especially the C++ Standard Library, the Standard

Template Library, Microsoft .NET, and Java's extensive built-in libraries) has made reusable data structures available to almost any software project. Reusable component libraries have advanced to include algorithms (generic programming in C++), user interface elements (Windows Forms, Java Swing), and structured data (XML). Our proposal for clip models follows this trend of abstraction and reuse. Although none of the efforts below address the full generality of our clip-model idea, there are a number of projects that are important stepping-stones toward our goals.

Various groups are working to build learning object repositories (http://elearning.utsa.edu/guides/LO-repositories.htm)—for example, the ARIADNE Foundation (http://www.ariadne-eu.org/) and MERLOT (http://www.merlot.org/Home.po)—and to develop standards and reference models for learning objects—for example, SCORM (http://www.adlnet.org/index.cfm?fuseaction=scormabt) and IEEE's WG12: Learning Object Metadata (http://ltsc.ieee.org/wg12/). Several efforts have begun to address some of the simulation science challenges identified above. The Center for Component Technology for Terascale Simulation Software is designing a Common Component Architecture (http://www.cca-forum.org/) as part of a program of the U.S. Department of Energy (DOE) Office of Science (SC). The Knowledge Web community is now starting to tackle the problem of identifying and encoding domain-specific ontologies for the Web. Clyde W. Holsapple and K. D. Joshi describe a collaborative approach to designing an ontology; the approach begins with independent ontological proposals from several authors and incorporates input from many contributors.[11] At the University of Washington, the Structural Informatics Group has been working on the Digital Anatomist Foundational Model of Anatomy (FMA), an ambitious anatomical ontology: "The FMA is a domain ontology that represents a coherent body of explicit declarative knowledge about human anatomy" (http://sig.biostr.washington.edu/projects/fm/AboutFM.html). The ambitious Digital Human Project (http://www.fas.org/dh/index.html), which uses the FMA ontologies, is intended to incorporate all biologically

Educational software R&D thus far has insufficient commercial appeal and must therefore be considered a strategic investment by funding agencies and companies.

relevant systems at all time and scale dimensions. They range from 10^{-14} sec chemical reactions to 10^9 sec lifetimes and perhaps 10^{12} for ecosystems, and from 10^{-9} meter chemical structures to meter-scale humans. The work thus far has focused on a series of scattered projects around the world, including the CellML (http://www.cellml.org/public/about/what_is_cellml.html) and other work at the University of Aukland. In the United States, the NIH (National Institutes of Health) (http://www.nih.gov/) has created an interagency group, is planning another meeting in 2005, and has started a number of bioinformatic centers that should help, while DARPA (Defense Advanced Research Projects Agency) has charged ahead with the Virtual Soldier Program (http://www.darpa.mil/dso/thrust/biosci/virtualsoldier.htm) and the BioSPICE program (https://community.biospice.org/).

Conclusion

Rethinking learning and education, in all of their dimensions, to successfully address the needs in this century is an overwhelmingly large and complex research and implementation agenda that will require a multi-decade—indeed, never-ending—level of effort on the part of all those involved in creating and delivering educational content. Nonetheless, a start *must* be made, as a national—indeed, global—imperative.

The start we're proposing here (the Learning Federation's R&D roadmaps) is another first step in the quest to build next-generation educational content and tools. This research agenda is meant to lead to the development not only of next-generation authoring tools and content but also of exemplary curricula in the broadest sense. The research agenda is predicated on our belief that "hardware will take care of itself" because of commoditization driven by market forces. Educational software R&D, on the other hand, thus far has insufficient commercial appeal and must therefore be considered a strategic investment by funding agencies and companies. Industry and government are certainly investing in computer-based training; much can be learned from their successful efforts.

To return to our biology example, we believe that the creation of families of interoperable clip models that will describe the human body as a system of interconnected biological components at all levels—from the molecular to the gross anatomical—will provide an unprecedented learning resource. Even though the creation of such families of clip models in a variety of disciplines will necessitate the integration of work from thousands of contributors over decades, even a beginning but very ambitious and comprehensive effort, such as the Digital Human Project, at building biological system components will have a payoff. We should not be daunted by the sheer magnitude of the task but should make steady progress along a clearly articulated path.

Furthermore, clip models are not, by themselves, the answer: there is no magic bullet, no single style of educational content that can encompass the enormously diverse set of requirements for this agenda. Creating high-quality next-generation educational content, across all disciplines and at all levels, will require a Grand Challenge effort on a scale such as the Manhattan Project, the Man on the Moon (Apollo) Project, and the Human Genome Project. The U.S., European, and several Asian economies certainly have both the ability and the need to cultivate the will to invest the same amount in creating exemplar interactive courses as they do in videogames and special-effects movies. Indeed, the U.S. Department of Defense is making significant modern IT-based investments for its training needs, mostly notably in "America's Army" (http://www.americasarmy.com/) and the funding of institutes such as the USC Institute for Creative Technologies (http://www.ict.usc.edu/). We cannot afford to have the civilian sector left behind. The Learning Federation has made a start in working with the government with the DOIT (Digital Opportunity Investment Trust) report (http://www.digitalpromise.org/), which articulates a potential funding mechanism based on spectrum sales.

The payoff from the huge investment of time, energy, and money cannot be overstated. Beyond education, the clip-model architecture will help advance science itself. The architecture will enable the "development" aspect of R&D to rapidly integrate advances in basic research. The coming avalanche of data in genomics and proteomics will require massively interconnected simulation systems; otherwise, how will the identification of a gene in Japan link to a class of pharmaceutical candidates for a rare disease being researched in Switzerland? Information sharing must be augmented by model sharing as an intrinsic part of the research process if connections are to be drawn between advances in different specialized fields—not sharing simply by publishing papers in research journals but by publishing information as software objects that can be *used* immediately (subject to accommodating the relevant IP and commercialization issues) in other research projects. We cannot predict the insights that will be revealed by happy accident when two or three unrelated strands of knowledge are unified in an integrated model, but we can eagerly anticipate the leverage that will be gained from the synergy. *e*

Notes

We gratefully acknowledge the support of our sponsors: NSF and Sun Microsystems, which supported the Exploratories Project led by Anne Morgan Spalter; NSF and Microsoft, which support the work on MathPad^2 led by Joseph LaViola; and Hewlett Packard, which provided twenty Tablet PCs to enable us to conduct a user study of ChemPad, led by Dana Tenneson, in an organic chemistry class. In addition we thank Janet Bruesselbach, the clip-model illustrator, and our four reviewers: Henry Kelly and Randy Hinrichs (who also are co-founders of the Learning Federation), Anne Morgan Spalter, and Debbie van Dam.

1. Rosemary Simpson, Allen Renear, Elli Mylonas, and Andries van Dam, "50 Years after 'As We May Think': The Brown/MIT Vannevar Bush Symposium," *ACM Interactions,* vol. 3, no. 2 (March 1996): 47–67.
2. John Seely Brown, "Growing Up Digital: How the Web Changes Work, Education, and the Ways People Learn," *Change,* March/April 2000, <http://www.aahe.org/change/digital.pdf>.
3. Alan Kay and Adele Goldberg "Personal Dynamic Media," IEEE Computer, vol. 10, no. 3 (March 1977): 31-41.
4. Neal Stephenson, *The Diamond Age; or, Young Lady's Illustrated Primer* (New York: Bantam Books, 1995).
5. See Mark Weiser's Web site: <http://www.ubiq.com/hypertext/weiser/weiser.html>.
6. We know all too little about effective group learning using digital media. A lot could be learned, for example, from studying the kind of informal learning ecology that typifies massive multiplayer games, such as "The Sims," "EverQuest," and "Asheron's Call."
7. Howard Gardner, *Frames of Mind: The Theory of Multiple Intelligences,* 10th anniversary ed. (New York: Basic Books, 1993).
8. Kay and Goldberg, "Personal Dynamic Media."
9. Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes, "SKETCH: An Interface for Sketching 3D Scenes," *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York: ACM Press, 1996), 163–70.
10. Jerome S. Bruner, *The Process of Education* (Cambridge: Harvard University Press, 1960), 33.
11. Clyde W. Holsapple and K. D. Joshi, "A Collaborative Approach to Ontology Design," *Communications of the ACM,* vol. 45, no. 2 (February 2002): 42–47.

Multi-Domain Sketch Understanding

Dynamically Constructed Bayes Nets for Multi-Domain Sketch Understanding

Christine Alvarado[*] **and Randall Davis**[†]
MIT CSAIL
Cambridge, MA 02139
{calvarad,davis}@csail.mit.edu

Abstract

This paper presents a novel form of dynamically constructed Bayes net, developed for multi-domain sketch recognition. Our sketch recognition engine integrates shape information and domain knowledge to improve recognition accuracy across a variety of domains using an extendible, hierarchical approach. Our Bayes net framework integrates the influence of stroke data and domain-specific context in recognition, enabling our recognition engine to handle noisy input. We illustrate this behavior with qualitative and quantitative results in two domains: hand-drawn family trees and circuits.

1 Introduction

There is a gap between how people naturally express ideas and the ability of computers to use that information. For example, while sketching provides a natural way to record design ideas in many domains, sketches are, unavoidably, static pictures. Computer aided design tools, on the other hand, offer powerful capabilities, but require designers to interact through buttons and menus. The hardware to draw on the computer exists; the missing element is the computer's ability to interpret hand-drawn symbols in a domain. To address this problem, we are constructing a general sketch recognition architecture applicable to a number of domains, and capable of parsing freely-drawn strokes in real time and interpreting them as depicting objects in the domain of interest.

Sketch recognition involves two related subproblems: stroke segmentation and symbol recognition. *Segmentation* determines which strokes should be grouped to form a single symbol. *Symbol recognition* determines what symbol a given set of strokes represents.

The difficulty of doing segmentation and recognition simultaneously has led previous approaches to place constraints on the user's drawing style, or focus on tasks where assumptions can greatly reduce segmentation complexity. For example, the multimodal approach in [Wu *et al.*, 1999] assumes that each symbol will be drawn independently, and that the user will often say the name of the symbol when drawing it. The approach in [Kara and Stahovich, 2004] assumes that the diagram (a feedback control system) consists of shapes linked by arrows, which is not true in other domains.

While these systems have proven useful for their respective tasks, we aimed to create a more general system, independent of drawing assumptions in any one domain. Our system is designed, instead, to be applied to a number of symbolic domains by giving it descriptions of shapes and commonly occurring combinations of shapes. We use these descriptions in a three-stage, constraint-based approach to recognition. Our system first relies on a rough processing of the user's strokes to generate a number of interpretation hypotheses. In the second stage, the system uses a novel form of dynamically constructed Bayes net to determine how well each hypothesis fits the data. In the third stage, it uses this evaluation to guide further hypothesis generation. This paper focuses specifically on the second stage, exploring hypothesis evaluation; the other two stages are described in [Alvarado, 2004].

Using Bayes nets for hypothesis evaluation offers two advantages over previous constraint-based recognition approaches (e.g., [Futrelle and Nikolakis, 1995; Hammond and Davis, 2004]). First, our system can interpret drawings as they develop, identifying shapes before they are complete. Second, the system's belief in a hypothesis can be influenced by both the strokes and the context in which the shape appears, allowing the system to cope with noise in the drawing.

We constructed a sketch recognition engine and used it in two domains: family tree diagrams and circuit diagrams. We show that the Bayes net successfully allows domain-specific context to help the system overcome noise in the stroke data, reducing interpretation errors compared to a baseline system.

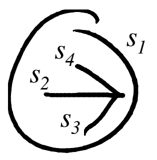
2 Dynamically Constructed Graphical Models

While time-based graphical models (e.g., Dynamic Bayes Nets) have been widely used, they are not suitable for sketch understanding, for two reasons. First, we must model shapes based on two-dimensional constraints (e.g., touches) rather than on temporal constraints (i.e., follows). Second, our models cannot simply unroll in time as data arrives: we cannot necessarily predict the order in which the user will draw the strokes, and things drawn previously can be changed.

While it is not difficult to use Bayes nets to model spatial relationships, static Bayes nets are not suitable for our task

[*]Currently at Harvey Mudd College, Claremont, CA, 91711

[†]This work was funded by the MIT iCampus project supported by Microsoft and MIT Project Oxygen.



Primitive hypotheses
E_1 = `ellipse-fit(`s_1`)`
L_1 = `line-fit(`s_2`)`
L_2 = `line-fit(`s_3`)`
L_3 = `line-fit(`s_4`)`

Arrow hypothesis A_1
Subshapes:
line hypotheses L_1, L_2, L_3
Constraints:
C_2 = (coincident `line-fit(`s_2`)`.p1 `line-fit(`s_3`)`.p1)
C_3 = (coincident `line-fit(`s_2`)`.p1 `line-fit(`s_4`)`.p1)
C_4 = (equalLength `line-fit(`s_3`)` `line-fit(`s_4`)`)
C_5 = (shorter `line-fit(`s_3`)` `line-fit(`s_2`)`)
C_6 = (acuteAngle `line-fit(`s_2`)`, `line-fit(`s_3`)`)
C_7 = (acuteAngle `line-fit(`s_2`)`, `line-fit(`s_4`)`)

CS hypothesis CS_1
Subshapes:
arrow hypothesis A_1, ellipse hypothesis E_1
Constraints:
C_1 = (contains `ellipse-fit(`s_1`)` `shape-fit(`s_2,s_3,s_4`)`)

Figure 1: A single current source hypothesis (CS_1) and associated lower-level hypotheses.

because we cannot predict *a priori* the number of strokes or symbols the user will draw. For sketch recognition, as in related dynamic tasks, models to reason about specific problem instances (e.g., a particular sketch) must be dynamically constructed in response the input. This problem is known as the task of *knowledge-based model construction* (KBMC).

Early approaches to KBMC focused on generating Bayes nets from probabilistic knowledge bases [Glessner and Koller, 1995; Haddawy, 1994]. A recently proposed representation uses generic template knowledge directly as Bayes net fragments that can be instantiated and linked together at run-time [Laskey and Mahoney, 1997]. Finally, Koller *et al.* have developed a number of object-oriented frameworks [Koller and Pfeffer, 1997; Pfeffer *et al.*, 1999; Getoor *et al.*, 1999]. These models represent knowledge in terms of relationships among objects and can be instantiated dynamically in response to the number of objects in a particular situation.

Although these frameworks are powerful, they are not directly suitable for sketch recognition. First, because of the size of the networks we encounter, it is sometimes desirable to generate only part of a complete network, or to prune nodes from the network. In reasoning about nodes in the network, we must account for the fact that the network may not be fully generated or relevant information may have been pruned from it (see [Alvarado, 2004] for details). Second, these previous models have been optimized for responding to specific queries about a single node in the network. In contrast, our model must provide probabilities for a full set of possible interpretations of the user's strokes.

3 Network Structure

Our Bayes net model is built around hierarchical descriptions of shapes in a domain, described in a language called LADDER [Hammond and Davis, 2003]. The basic unit in this language is a *shape*, which we use to mean a pattern recognizable in a given domain. Shapes may be *compound*, i.e., composed of *subshapes* fit together according to *constraints*. These subshapes also may be compound, but all shapes must be non-recursive. A shape that cannot be decomposed into subshapes (e.g., a line), is called a *primitive shape*. Primitive shapes may have named *subcomponents* that can be used when describing other shapes, e.g., the endpoints of a line, "p1" and "p2", used in Figure 1.

We refer to each shape description as a *template* with one *slot* for each subpart. A *shape hypothesis* is a template with an associated mapping between slots and strokes, generated during the recognition process. Similarly, a *constraint hypothesis* is a proposed constraint on one or more of the user's strokes. A *partial hypothesis* is a hypothesis in which one or more slots are not bound to strokes.

To introduce our Bayes net model, we begin by considering how to model a current source hypothesis (CS_1) for the stokes in Figure 1. A current source is a compound shape, so CS_1 involves two lower-level shape hypotheses—E_1, an ellipse hypothesis for stroke s_1; and A_1, an arrow hypothesis involving strokes s_2, s_3 and s_4—and one constraint hypothesis—C_1, indicating that an ellipse fit for stroke s_1 contains strokes s_2, s_3, and s_4. A_1 is also compound and is further broken down into three line hypotheses (L_1, L_2 and L_3) and six constraint hypotheses ($C_2, \ldots, C_7$) according to the description of an arrow. Thus, determining the strength of hypothesis CS_1 can be transformed into the problem of determining the strength of a number of lower-level shape and constraint hypotheses.

The Bayes net used to evaluate CS_1 is shown in Figure 2. There is one node in the network for each hypothesis; each node represents a boolean random variable that is true if the corresponding hypothesis is correct. The probability of each hypothesis is influenced both through its children by stroke data and through its parents by the context in which it appears, allowing the system to handle noise in the drawing.

The nodes labeled $O_1, \ldots, O_{11}$ represent measurements of the stroke data that correspond to the constraint or shape to which they are linked. The variables corresponding to these nodes have positive real numbered values that we discretize in our implementation[1]. For example, the variable O_2 is a measurement of the squared error between the stroke s_1 and the best fit ellipse to that stroke. Its raw value (later discretized) ranges from 0 to the maximum possible error between any stroke and an ellipse fit to that stroke. The boxes labeled $s_1, \ldots, s_4$ are not part of the Bayes net but serve to indicate the stroke or strokes from which each measurement, O_i, is taken (e.g., O_2 is measured from s_1). $P(CS_1 = t|ev)$ (or simply $P(CS_1|ev)$)[2], where *ev* is the evidence observed from the user's strokes, represents the probability that the hypothesis CS_1 is correct.

The direction of the links may seem counterintuitive, but there are two important reasons why the links are directed from higher-level shapes to lower-level shapes instead of the opposite direction. First, whether a higher-level hypothesis is true directly influences whether a lower-level hypoth-

[1]The BN software we used did not support continuous variables.
[2]Throughout this paper, t means true, and f means false.

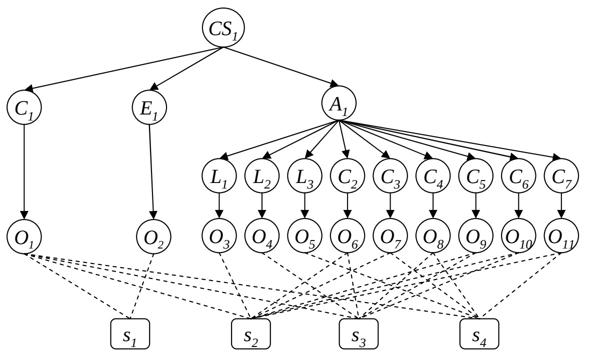


Figure 2: A Bayes net to verify a single current source hypothesis. Labels come from Figure 1.

esis is true. For example, if the arrow hypothesis A_1 is true, then it is extremely likely that all three line hypotheses, L_1, L_2, L_3, are also true. Second, this representation allows us to model lower-level hypothesis as conditionally independent given their parents, which reduces the complexity of the data needed to construct the network.

Each shape description constrains its subshapes only relative to one another. For example, an arrow may be made from *any* three lines that satisfy the necessary constraints. Based on this observation, our representation models a symbol's subshapes separately from the constraints between them. For example, node L_2 represents the hypothesis that stroke s_3 is a line. Its value will be true if the user intended for s_3 to be a line, *any* line regardless of its position, size or orientation. C_4 separately represents the hypothesis that the line fit to s_3 and the line fit to s_4 are the same length.

The conditional independence between shapes and constraints might seem a bit strange at first. For example, whether or not two lines are the same length seems to depend on the fact that they are lines. However, observation nodes for constraints are calculated in such a way that their value is not dependent on the true interpretation for a stroke. For example, when calculating whether or not two lines are the same length, we first fit lines to the strokes (regardless of whether or not they actually look like lines), then measure their length. How well these lines fit the original strokes is not considered in this calculation.

The fact that there is no edge between the constraint nodes and the shapes they constrain has an important implication for using this model to perform recognition: There is no guarantee in this Bayes net that the constraints will be measured between the correct subshapes because the model allows subshapes and constraints to be detected independently. For example, we want C_4 in Figure 2 to indicate that L_2 and L_3 (the two lines in the head of an arrow) are the same length, not simply that *any* two lines are the same length. To satisfy this requirement, the system must ensure that O_8 is measured from the same strokes that O_4 and O_5 were measured from. We use a separate mechanism to ensure that only legal bindings are created between strokes and observation nodes.

The way we model shape and constraint information has two important advantages for recognition. First, this Bayes

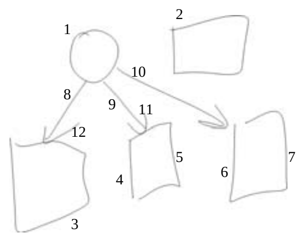


Figure 3: A partial sketch of a family tree. Quadrilaterals (Q) represent males (M); ellipses (E) represent females (F), and arrows (A) indicate a parent-child relationship.

net model can be applied to recognize a shape in any size, position and orientation. CS_1 represents the hypothesis that $s_1, ..., s_4$ form a current source symbol, but the exact position, orientation and size of that symbol is determined directly from the stroke data [3].

Second, the system can model competing higher-level interpretations for lower-level shapes. For example, the system may consider a stroke to be a line that is in turn part of either an arrow or a quadrilateral. Because the line hypothesis does not include any higher-level shape-specific constraint information, both an arrow hypothesis node and a quadrilateral hypothesis node can point to this same line hypothesis node. These two hypotheses then become alternate, competing explanations for the line hypothesis. We further discuss how hypotheses are combined below.

4 Recognizing a Complete Sketch

To recognize a complete sketch, we create a Bayes net similar to the one above for each shape. We call each of these Bayes nets a *shape fragment* because they can be combined to create a complete Bayes net for evaluating the whole sketch.

Given a set of hypotheses for the user's strokes (hypothesis generation is described in [Alvarado and Davis, 2004] and [Alvarado, 2004]), the system instantiates the corresponding shape fragments and links them together to form a complete Bayes net, called the *interpretation network*. To illustrate this process, consider a piece of a network generated in response to Strokes 6 and 7 in the example of Figure 3. Figure 4 shows the part of the Bayes net representing the hypotheses that the system generated for these strokes. Low level processing recognizes Strokes 6 and 7 as L-shaped polylines and breaks each into two individual lines ($L_1 ... L_4$) that meet.

4.1 Linking Shape Fragments

When Bayes net fragments are linked during recognition, each node H_n may have several parents, $S_1 \dots S_m$, where each parent represents a possible higher-level interpretation for H_n. We use a noisy-OR function to combine the influences of all the parents of H_n to produce the complete conditional probability table (CPT) for $P(H_n|S_1, \dots, S_m)$. The noisy-OR function models the assumption that each parent can independently cause the child to be observed. For example, a single

[3]Orientation-dependent symbols (e.g., a downward-facing arrow) can be recognized by including orientation-dependent constraints in their descriptions (e.g., vertical, below).

stroke might be part of a quadrilateral or an arrow, but both interpretations would favor that interpretation of the stroke as a line. We set $P(H_n = f|S_j = t) = 0$ for all parents S_j in which H_n is a required subshape or constraint, and we set $P(H_n = f|S_k = t) = 0.5$ for all parents S_k in which H_n is an optional subshape or constraint. A consequence of these values is that $S_j = t \Rightarrow P(H_n|S_1,\ldots,S_m) = 1$ for any S_j in which H_n is required, which is exactly what we intended.

We experimented with a noisy-XOR construct, implemented using a "gate node" similar to that described in [Boutilier *et al.*, 1996], but found that noisy-OR semantics were simpler and in fact produced better results. In effect, a noisy-OR node in a Bayes net with low prior probabilities behaves as a non-aggressive XOR. The fact that one parent is true does not actively prohibit the other parents from being true, but it causes their probabilities to tend back to their prior values because of the "explaining away" phenomenon.

4.2 Signal Level Noise

The bottom layer of the network deals with signal level noise by modeling the differences between the user's intentions and the strokes that she draws. For example, even if the user intends to draw L_1, her stroke likely will not match L_1 exactly, so the model must account for this variation. Consider $P(O_1|L_1 = t)$ (recall that O_1 is a discretized continuous valued variable). If the user always drew perfect lines, this distribution would be 1 when $O_1 = 0$ (i.e., the error is 0), and 0 otherwise. However, most people do not draw perfect lines (due to inaccurate pen and muscle movements), and this distribution allows for this error. It should be high when O_1 is close to zero, and fall off as O_1 gets larger. The wider the distribution, the more error the system will tolerate, but the less information a perfect line will provide.

The other distribution needed is $P(O_1|L_1 = f)$ which is the probability distribution over line error given that the user did not intend to draw an line. This distribution should be close to uniform, with a dip around 0, indicating that if the user specifically does not intend to draw an line, she might draw any other shape, but probably won't draw anything that resembles an line. Details about how we determined the conditional probability distributions between primitive shapes and constraints and their corresponding observation nodes are given elsewhere [Alvarado, 2004].

4.3 Missing Strokes

A_1 is a partial hypothesis—it represents the hypothesis that L_1 and L_2 (from Stroke 6) are part of an arrow whose other line has not yet been drawn. Line nodes representing lines that have not been drawn (e.g., L_5) are not linked to observation nodes because there is no stoke from which to measure these observations. We refer to these nodes (and their corresponding hypotheses) as *virtual*.

The fact that partial hypotheses have probabilities allows the system to assess the likelihood of incomplete interpretations based on the evidence it has seen so far. In fact, even virtual nodes have probabilities, corresponding to the probability that the user (eventually) intends to draw these shapes but either has not yet drawn this part of the diagram or the

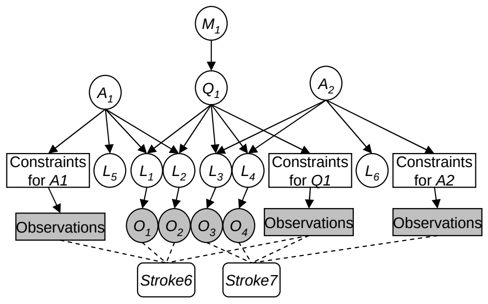


Figure 4: A portion of the interpretation network generated while recognizing the sketch in Figure 3.

correct low-level hypotheses have not yet been proposed (because of low-level recognition errors). A partial hypothesis with a high probability cues the system to examine the sketch for possible missed low-level interpretations during the hypothesis generation step.

5 Implementation and Bayesian Inference

Our system updates the structure of the Bayes net in response to each stroke the user draws, using an off-the-shelf, open source Bayes net package for Java called BNJ [bnj, 2004].

Generating and modifying the BNJ networks can be time consuming due to the exponential size of the CPTs between the nodes. We use two techniques to improve performance. First, networks are generated only when the system needs to evaluate the likelihood of a hypothesis. This on-demand construction is more efficient than continuously updating the network, because batch construction of the CPTs is often more efficient than incremental construction. Second, the system modifies only the portion of the network that has changed between strokes, rather than creating it from scratch every time.

We experimented with several inference methods and found that loopy belief propagation (loopy BP) [Weiss, 1997] was the most successful[4]. On our data, loopy BP almost always converged (messages initialized to 1, run until node values stable within 0.001). We further limited the algorithm's running time in two ways. First, we terminated the algorithm after 60 seconds if it had not yet converged. Second, we allowed each node to have no more than 8 parents (i.e., only 8 higher-level hypotheses could be considered for a single hypothesis), ensuring a limit on the complexity of the graphs produced. These restrictions had little impact on recognition performance in the family tree domain, but for complex domains such as circuit diagrams, more efficient inference algorithms or graph simplification techniques are needed to improve recognition results.

6 Application and Results

We applied our implemented system to two non-trivial domains—family trees and circuits—and found that it is capable of recognizing sketches in both domains without reprogramming. Qualitative and quantitative evaluation of our

[4]Junction Tree [Jensen *et al.*, 1990] was often too slow and Gibb's Sampling did not produce meaningful results.

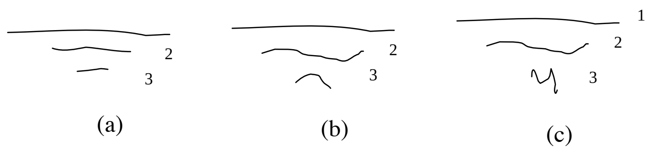


Figure 5: Part of three possible ground symbols.

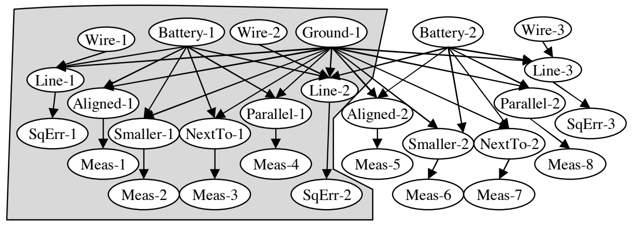


Figure 6: The Bayes net produced in response to the strokes in each ground symbol in Figure 5. The shaded area shows the network produced in response to the first two strokes.

Name	$P[Shape \mid stroke\ data]$					
	after 2 strokes			**after 3 strokes**		
	(a)	(b)	(c)	(a)	(b)	(c)
Wire-1	0.4	0.41	0.41	0.4	0.4	0.42
Wire-2	0.4	0.38	0.38	0.4	0.4	0.38
Wire-3	N/A	N/A	N/A	0.4	0.4	0.1
Battery-1	0.51	0.51	0.51	0.09	0.1	0.91
Battery-2	N/A	N/A	N/A	0.09	0.1	0.0
Ground-1	0.51	0.51	0.51	0.95	0.94	0.03

Table 1: Posterior probabilities for part of the network in Figure 6 for each sketch in Figure 5.

system illustrates its strengths over previous approaches and suggests some extensions.

Applying our system to a particular domain involves two steps: specifying the structural descriptions for the shapes in the domain, and specifying the prior probabilities for top-level shapes and domain patterns, i.e., those that will not have parents in the Bayes net. For each domain, we wrote a description for each shape and pattern in that domain. We hand-estimated priors for each domain pattern and top level shape based on our intuition about the relative prevalence of each shape. For example, in family-tree diagrams, we estimated that marriages were much more likely than partnerships, and set $P[Mar] = 0.1$ and $P[Part] = 0.001$. These priors represent the probability that a group of strokes chosen at random from the page will represent the given shape and, in most cases, should be relatively low. Although setting priors by hand can be tedious, we found through experimentation that the system's recognition performance was relatively insensitive to the exact values of these priors. For example, in the circuit diagrams, increasing all the priors by an order of magnitude did not affect recognition performance; what matters instead is the relative values of the prior probabilities.

Our system is capable of recognizing simple sketches nearly perfectly in both the family-tree and circuit domains. We also tested its performance on more complex, real-world data. As there is no standard test corpus for sketch recognition, we collected our own sketches and have made them available online to encourage others to compare their results with those presented here [Oltmans *et al.*, 2004]. In total, we tested our system on 10 family tree sketches and 80 circuit diagram sketches, with between 23 and 110 strokes each.

6.1 Qualitative Results

Qualitative analysis reveals that the Bayes net mechanism successfully aggregates information from stroke data and context, resolves inherent ambiguities in the drawing, and updates its weighting of the various existing hypotheses as new strokes are drawn. We show through an example that the Bayes net scores hypotheses correctly in several respects: it prefers to group subshapes into the fewest number of interpretations, it allows competing interpretations to influence one another, it updates interpretation strengths in response to new stroke data, and it allows both stroke data and context to influence interpretation strength.

To illustrate the points above, we consider in detail how the system responds as the user draws the three sets of strokes in Figure 5 (a ground symbol). To simplify the example, we consider a reduced circuit domain in which users draw only wires, resistors, ground symbols and batteries. Figure 6 shows the Bayes nets produced in response to the user's first two and first three strokes. After the first two strokes, the network contains only the nodes in the shaded area; after three strokes it contains all nodes shown. Continuous variables corresponding to observation nodes were discretized into three values: 0 (low error: stroke data strongly supports the shape or constraint), 1 (medium error: stroke data weakly supports the shape or constraint), and 2 (high error: stroke data does not support the shape or constraint).

Posterior probabilities are given in Table 1. For the relatively clean ground symbol (Figure 5(a)), the stroke data strongly supports all the shapes and constraints, so all shaded nodes in Figure 6 have value 0. After the first two strokes, the battery symbol and the ground symbol have equal weight, which may seem counterintuitive: after two strokes the user has drawn what appears to be a complete battery, but has drawn only a piece of the ground symbol. Recall, however, that the Bayes net models not what has been drawn, but what the user *intends to draw*. After two strokes, it is equally likely that the user is in the process of drawing a ground symbol. After the third stroke, the ground symbol's probability increases because there is more evidence to support it, while the battery's probability decreases. The fact that the ground symbol is preferred over the battery illustrates that the system prefers interpretations that result in fewer symbols (Okham's razor). Interpretations that involve more of the user's strokes have more support and get a higher score in the Bayes net. The fact that the battery symbol gets weaker as the ground symbol gets stronger results from the "explaining away" behavior in this Bayes net configuration: each of the low-level components (the lines and the constraints) are effectively "explained" by the existence of the ground symbol, so the battery is no longer needed to explain their presence.

Another useful property of our approach is that a small amount of noise in the data is counteracted by the context provided by higher-level shapes. The slightly noisy second and third strokes in Figure 5(b) cause the values of SqErr-2 and SqErr-3 to be 1 instead of 0 (all other shaded node values remain 0). Despite this noise, after three strokes the posterior probability of the ground symbol interpretation is still high (0.94), because Line-1 and all of the constraints are still strongly supported by the data. In addition, the probabilities for Line-2 and Line-3 are high (both 1.0, not shown in Table 1), indicating that the context provided by the ground symbol provides support for the line interpretations even when the evidence from the data is not as strong.

On the other hand, if the data is too messy, it causes the probability of the higher-level interpretations to decrease. The sketch in Figure 5(c) causes the value of SqErr-2 to be 1, and SqErr-3 to be 2, and results in a low posterior probability for Ground-1 (0.03) and a high posterior for Battery-1 (0.91), because the hypothesis Line-3 is contradicted by the user's third stroke. For now, the third stroke remains uninterpreted.

6.2 Quantitative Results

We ran our system on all of the sketches we collected and compared its performance to a baseline constraint-based recognition system that used a fixed threshold for detecting shapes and constraints and did not reinterpret low-level shapes. We measured recognition performance by determining the number of objects identified correctly in each sketch. Our system significantly outperformed the baseline system ($p \ll 0.001$), correctly recognizing 77% (F=0.83) of the shapes in the family tree diagrams and 62% (F=0.65) of the shapes in the circuit diagrams, while the baseline system correctly recognized 50% (F=0.63) and 54% (F=0.57).

While not yet real time, in general our system's processing time scaled well as the number of strokes increased. However, it occasionally ran for a long period. The system had particular trouble with areas of the sketch that involved many strokes drawn close together in time and space and with domains that involve more complicated or overlapping symbols. This increase in processing time was due almost entirely to increase in Bayes net complexity.

We believe we can speed up loopy BP based on the observation that it repeatedly sends messages between the nodes until each node has reached a stable value. When a stroke is added, our system resets all messages to 1, essentially erasing the work done the last time inference was performed, even though most of the graph is unchanged. The algorithm should instead begin with the messages that remain at the end of the previous inference step.

7 Conclusion

We have described a model for dynamically constructing Bayes nets to represent varying hypotheses for the user's strokes. Our model, specifically developed for the task of recognition, allows both stroke data and contextual data to influence the probability of an interpretation for the user's strokes. Using noisy-OR, multiple potential higher-level interpretations mutually influence each other's probabilities within the Bayes net. The net result is a sketch recognition approach that brings us a significant step closer to sketching as a natural and powerful interface.

References

[Alvarado and Davis, 2004] C. Alvarado and R. Davis. SketchREAD: A multi-domain sketch recognition engine. In *Proc. of UIST-04*, 2004.

[Alvarado, 2004] C. Alvarado. *Multi-Domain Sketch Understanding*. PhD thesis, MIT, 2004.

[bnj, 2004] Bayesian network tools in java (bnj), 2004. http://bnj.sourceforge.net.

[Boutilier *et al.*, 1996] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proc. of UAI '96*, 1996.

[Futrelle and Nikolakis, 1995] R. P. Futrelle and N. Nikolakis. Efficient analysis of complex diagrams using constraint-based parsing. In *ICDAR-95*, pages 782–790, Montreal, Canada, 1995.

[Getoor *et al.*, 1999] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proc. of IJCAI '99*, pages 1300–1309, 1999.

[Glessner and Koller, 1995] S. Glessner and D. Koller. Constructing flexible dynamic belief networks from first-order probabilistinc knowledge bases. In *Synbolic and Quantitatice Approaches to Reasoning and Uncertainty*, pages 217–226, 1995.

[Haddawy, 1994] P. Haddawy. Generating bayesian networks from probability logic knowledge bases. In *Proc. of UAI '94*, 1994.

[Hammond and Davis, 2003] T. Hammond and R. Davis. LADDER: A language to describe drawing, display, and editing in sketch recognition. *Proc. of IJCAI '03*, 2003.

[Hammond and Davis, 2004] T. Hammond and R. Davis. Automatically transforming symbolic shape descriptions for use in sketch recognition. *Proc. of AAAI '04*, 2004.

[Jensen *et al.*, 1990] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4, 1990.

[Kara and Stahovich, 2004] L. B. Kara and T. F. Stahovich. Hierarchical parsing and recognition of hand-sketched diagrams. In *Proc. of UIST '04*, 2004.

[Koller and Pfeffer, 1997] D. Koller and A. Pfeffer. Object-oriented bayesian networks. In *Proc. of UAI '97*, 1997.

[Laskey and Mahoney, 1997] K. B. Laskey and S. M. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Proc. of UAI '97*, 1997.

[Oltmans *et al.*, 2004] M. Oltmans, C. Alvarado, and R. Davis. Etcha sketches: Lessons learned from collecting sketch data. In *Making Pen-Based Interaction Intelligent and Natural*. AAAI Fall Symposium, 2004.

[Pfeffer *et al.*, 1999] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proc. of UAI '99*, 1999.

[Weiss, 1997] Y. Weiss. Belief propagation and revision in networks with loops. Technical report, MIT, November 1997.

[Wu *et al.*, 1999] L. Wu, S. L. Oviatt, and P. R. Cohen. Multimodal integration—a statistical view. *IEEE Transaction on Multimedia*, 1(4):334–341, December 1999.

SketchREAD: A Multi-Domain Sketch Recognition Engine

Christine Alvarado
MIT CSAIL
Cambridge, MA 02139 USA
calvarad@csail.mit.edu

Randall Davis
MIT CSAIL
Cambridge, MA 02139 USA
davis@csail.mit.edu

ABSTRACT
We present SketchREAD, a multi-domain sketch recognition engine capable of recognizing freely hand-drawn diagrammatic sketches. Current computer sketch recognition systems are difficult to construct, and either are fragile or accomplish robustness by severely limiting the designer's drawing freedom. Our system can be applied to a variety of domains by providing structural descriptions of the shapes in that domain; no training data or programming is necessary. Robustness to the ambiguity and uncertainty inherent in complex, freely-drawn sketches is achieved through the use of context. The system uses context to guide the search for possible interpretations and uses a novel form of dynamically constructed Bayesian networks to evaluate these interpretations. This process allows the system to recover from low-level recognition errors (e.g., a line misclassified as an arc) that would otherwise result in domain level recognition errors. We evaluated SketchREAD on real sketches in two domains—family trees and circuit diagrams—and found that in both domains the use of context to reclassify low-level shapes significantly reduced recognition error over a baseline system that did not reinterpret low-level classifications. We also discuss the system's potential role in sketch-based user interfaces.

Categories and Subject Descriptors: I.5.4 [**Pattern Recognition**]: Applications; H.5.2 [**User Interfaces**]: Interaction Styles

Additional Keywords and Phrases: Pen-based UIs, input and interaction technology, sketch recognition, intelligent UIs, Bayesian networks

1 INTRODUCTION

While in recent years there has been an increasing interest in sketch-based user interfaces [9, 13, 14], the problem of robust free-sketch recognition remains largely unsolved. Because existing sketch recognition techniques are difficult to implement, and are error-prone or severely limit the user's drawing style, many previous systems that support sketching perform only limited recognition. ScanScribe, for example, uses perceptual guidelines to support image and text editing, but does not attempt to recognize the user's drawing [14]. Similarly, the sketch-based DENIM system supports the design of web pages but recognizes very little of the user's sketch [13]. Finally, NuSketch reasons about spatial relationships in military diagrams, but does not recognize sketched symbols [5]. Systems of this sort involve the computer in the early design, making it easy to record the design process, but they do not always facilitate automatic transition from the early stage design tool to a more powerful design system.

To enable the construction of sketch-based interfaces for a number of domains, we have created SketchREAD (Sketch Recognition Engine for mAny Domains), a system capable of understanding freely-drawn, messy, two-dimensional diagrammatic sketches. SketchREAD "understands" a user's sketch in that it parses a user's strokes as they are drawn and interprets them as objects in a domain of interest. SketchREAD operates in the background while the user sketches; recognition results may be displayed after the user completes the sketch or at any time during the recognition process. Our engine does not assume it will receive user feedback for its recognition, because having to give feedback can distract the user during the design process. It may be applied to any domain in which sketches may be described in terms of diagrammatic symbols (e.g., circuit diagrams, military course of action diagrams). Although SketchREAD is not designed to recognize other types of sketches (e.g., three-dimensional sketches and free-form sketches common in domains such as architecture) the class of sketches it is designed to recognize is important for designers in many domains. This system both helps solve a challenging problem in sketch understanding and enables more natural interaction with design software.

One of the most difficult problems in creating a sketch recognition system is handling the tradeoff between ease of recognition and drawing freedom. The more we constrain the user's drawing style, the easier recognition becomes. For example, if we enforce the constraint that each component in the domain must be a carefully drawn symbol that can be created with a single stroke, it is relatively easy to build recognizers capable of distinguishing between the symbols, as was done with Palm Pilot Graffiti^{TM}. The advantage of using restricted recognizers is accuracy; the disadvantage is the designer is constrained to a specific style of sketching.

Previous recognition-intensive systems have focused on tasks where drawing style assumptions can greatly reduce recognition complexity. Long *et al.* focus on designing special graphical symbols that will not be confused easily by the computer [10]. This approach improves recognition, but it limits the designer to a specific set of single-stroke symbols that may be natural only for certain tasks. The Quickset

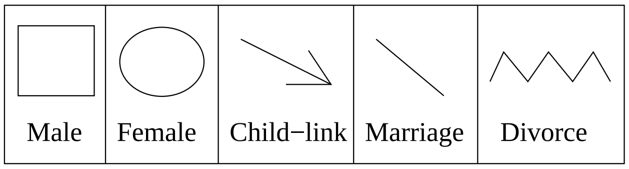


Figure 1: The symbols in the family tree domain.

system for recognizing military course of action (COA) diagrams uses multi-modal information to improve recognition of sketched symbols [20], but assumes that each symbol will be drawn independently, and that the user will likely speak the name of the symbol when drawing it. These assumptions aid recognition, but may fail for design tasks in other domains. In electrical engineering, for example, designers draw several symbols without pausing and probably do not speak the name of the symbols they draw. Other previous systems have similar constraints on drawing style or do not provide the level of recognition robustness we seek here [17, 6, 3].

While the previous systems have proven useful for their respective tasks, we aim to create a general sketch recognition system that does not rely on the drawing style assumptions of any one domain. To be usable, a sketch recognition-based system must make few enough mistakes that sketching is less work than using a more traditional (i.e., menu-based) interface. To be broadly effective the system's architecture should be easily applied across a variety of domains, without having to reengineer the system. Our system is a significant step toward achieving these goals.

Our approach makes four contributions to the field of sketch-based UIs. First, our engine separates information about basic shapes from their interpretation in a particular domain so that our engine can more easily be extended to multiple domains without having to recreate the shape recognizers. Second, we have developed a novel form of dynamically constructed Bayesian networks to allow both stroke data and higher-level shape information to influence the system's interpretation of the user's strokes. Third, our system uses this novel Bayesian network technique to guide its search for possible interpretations of the user's sketch, allowing it to recover from low-level interpretation errors (e.g., a line misclassified as an arc) that would otherwise prevent recognition of the sketch. Fourth, we gathered and tested our recognition engine on unconstrained freely-drawn data in two domains—family trees and circuits. We show that SketchREAD consistently reduced recognition errors over a baseline system that did not reinterpret low-level classifications. The results of these tests make concrete the strengths of our approach and the remaining challenges we face in building a recognition engine that can better handle real-world data.

We begin by exploring the challenges of recognizing real-world sketches. Next, we present our new approach to recognition, then analyze our system's performance on real data. We conclude with a discussion of how to extend our system's power and how it can be used in sketch recognition user interfaces (SkRUIs).

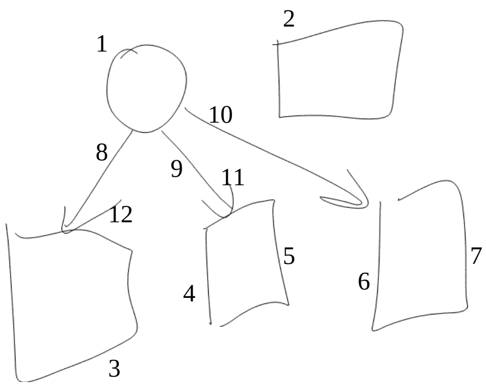


Figure 2: A partial sketch of a family tree.

2 THE CHALLENGES OF SKETCH UNDERSTANDING

Figure 2 shows the beginning of a sketch of a family tree, with the strokes labelled in the order in which they were drawn. The symbols in this domain are given in Figure 1. The user started by drawing a mother and a father, then drew three sons. He linked the mother to the sons by first drawing the shafts of each arrow and then drawing the arrowheads. (In our family tree diagrams, each parent is linked to each child with an arrow.) He will likely continue the drawing by linking the father to the children with arrows and linking the two parents with a line.

Although relatively simple, this drawing presents many challenges for sketch recognition. While previous recognition systems have addressed some of these challenges, SketchREAD is the first to address all of them using a general framework that can be extended to multiple domains.

The first challenge illustrated in Figure 2 is the incremental nature of the sketch process. Incremental sketch recognition allows the computer to seamlessly interpret a sketch as it is drawn and keeps the user from having to specify when the sketch is complete. To recognize a potentially incomplete sketch, a computer system must know when to recognize a piece of that sketch and when to wait for more information. For example, Stroke 1 can immediately be recognized as a female, but Stroke 6 cannot be recognized without Stroke 7.

The second challenge is that many of the shapes in Figure 2 are visually messy. For example, the center arrowhead (Stroke 11) looks more like an arc than two lines. Next, the stroke used to draw the leftmost quadrilateral (Stroke 3) looks like it is composed of five lines—the top of the quadrilateral is bent and could be reasonably divided into two lines by a stroke parser. Finally, the lines in the rightmost quadrilateral (Strokes 6 and 7) do not touch in the top left corner.

The third issue is segmentation: It is difficult to know which strokes are part of which shapes. For example, if the computer knew that Strokes 9 and 11 were part of one shape, the system would likely be able to match an arrow pattern to these strokes using a standard algorithm, such as a neural network. Unfortunately, segmentation is not an easy task. The shapes in this drawing are not clearly spatially segmented, and naively trying different combinations of strokes is prohibitively time consuming. To simplify segmentation, many previous systems (e.g., [13, 20]) assume each shape will be drawn with temporally contiguous strokes. This assumption does not hold here.

There are also some inherent ambiguities in how to segment the strokes. For example, lines in our domain indicate marriage, but not every line is a marriage-link. The shaft of the leftmost arrow (Stroke 8) might also have been interpreted as a marriage link between the female (Stroke 1) and the leftmost male (Stroke 3). In this case, the head of that arrow (Stroke 12) could have been interpreted as a part of the drawing that is not yet complete (e.g., the beginning of an arrow from the leftmost quadrilateral (Stroke 3) to the top quadrilateral (Stroke 2)).

Finally, how shapes are drawn can also present challenges to interpretation. The head of the rightmost arrow (part of Stroke 10) is actually made of three lines, two of which overlap to form one side of the arrowhead. In order to recognize the arrow, the system must know to collapse those two lines into one, even though they do not actually overlap. Another challenge arises because the same shape may not always be drawn in the same way. For example, the arrows on the left (Strokes 8 and 12, and Strokes 9 and 11) were drawn differently from the one on the right (Stroke 10) in that the user first drew the shaft with one stroke and then drew the head with another. This variation in drawing style presents a challenge for segmentation and recognition because a system cannot know how many strokes will be used to draw each object, nor the order in which the parts of a shape will appear.

Many of the difficulties described in the example above arise from the messy input and visual ambiguity in the sketch. It is the context surrounding the messy or ambiguous parts of the drawing that allows humans to interpret these parts correctly. We found that context also can be used to help our system recover from low-level interpretation errors and correctly identify ambiguous pieces of the sketch. Context has been used to aid recognition in speech recognition systems; it has been the subject of recent research in computer vision [18, 19] and has been used to a limited extent in previous sketch understanding systems [3, 6, 13]. We formalize the notion of context suggested by previous sketch recognition systems. This formalization improves recognition of freely drawn sketches using a general engine that can be applied to a variety of domains.

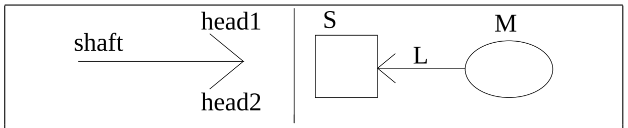
3 TECHNICAL APPROACH

We have developed and implemented a general framework for sketch recognition that handles the challenges presented in the previous section and that can be applied to a variety of domains by supplying domain specific pattern descriptions.

3.1 Knowledge Representation

We use a hierarchical shape description language to describe the shapes in a domain. A hierarchical representation is useful because it enables re-use of geometric shapes (e.g., arrows) in a variety of domains, and because many sketched symbols are compositional. Here we describe the language only briefly. For a more complete description, see [7].

Figure 3 shows a simple use of the language. The arrow is an example of a *shape*, which we use to mean a pattern recog-



DEFINE ARROW
(**Subshapes**
L1,L2,L3: (Line *shaft head1 head2*))
(**Constraints**

C1: (coincident *shaft.p1 head1.p1*)	*C3*: (equal-length *head1 head2*)
	C4: (shorter *head1 shaft*)
C2: (coincident *shaft.p1 head2.p2*)	*C5*: (acute-angle *head1 shaft*)
	C6: (acute-angle *head2 shaft*))

DEFINE MOTHER-SON
(**Subshapes**
M,S,L: (Female *M*) (Male *S*) (Child-link *L*)
(**Constraints**
C1: (touches *L.head S*)
C2: (touches *L.tail M*))

Figure 3: The description of the shape “arrow” and the domain pattern “mother-son.” Child-links are defined from arrows, males from quadrilaterals, and females from ellipses. Labels for the subshapes and constraints are used in Figure 4.

nizable in a given domain. The arrow is a *compound shape*, i.e., one composed of non-recursive *subshapes* fit together according to *constraints*. A line is a *primitive shape*—one that cannot be decomposed into subshapes. Although primitive shapes cannot be decomposed into subshapes, they may have named *subcomponents* that can be used when describing other shapes, e.g., the endpoints of a line, “p1” and “p2”, used in Figure 3. *Domain shapes* are shapes that have semantic meaning in a particular domain. For example, `line`, `arrow` and `child-link` are all shapes that may be recognized, but only a child-link has meaning in the family tree domain. *Domain patterns* are combinations of domain shapes that are likely to occur, for example a child-link pointing from a female to a male, indicating a relationship between mother and son. Recognition information for primitive shapes and constraints are built in to SketchREAD; compound shapes, including domain shapes and patterns, are recognized from primitive shapes and vary depending on the domain to which SketchREAD is applied.

The context in which a shape is likely to occur is given by the higher-level shapes and domain patterns in which it appears. For example, the domain pattern `mother-son` provides a context in which child-links (and in turn arrows), males and females are likely to occur. This representation allows our system to incorporate both domain knowledge and shape information in its interpretation of the user’s sketch. The separation between domain shapes and their geometric subshapes facilitates the re-use of geometric shapes in other domains (e.g., the arrow in electrical engineering symbols). The shapes and domain patterns for the family tree domain are listed in Table 1; constraints are omitted to save space.

3.2 Recognition Overview

Recognizing the sketch is a matter of parsing a user’s strokes according to the specified visual language. Visual language parsing has been studied [12], but most previous approaches

Shape/Pattern (Abbr.)	Subshapes
Line (L)	–
Ellipse (E)	–
Polyline (PL)	–
Arrow (A)	Line *h1, h2, shaft*
Quadrilateral (Q)	Line *l1, l2, l3, l4*
Marriage-link (ML)	Line *l*
Divorce-link (DL)	Polyline *pl*
Female (F)	Ellipse *e*
Male (M)	Quadrilateral *m*
Child-link (CL)	Arrow *a*
Divorce (Div)	Male *h*; Female *w*; DL *l*
Marriage (Mar)	Male *h*; Female *w*; ML *l*
Partnership-F (PartF)	Female *w1*, *w2*; ML *l*
Partnership-M (PartM)	Male *h1*, *h2*; ML *l*
Father-daughter (FD)	Male *f*; Female *d*; CL *l*
Mother-daughter (MD)	Female *m*, *d*; CL *l*
Father-son (FS)	Male *f*, *s*; CL *l*
Mother-son (MS)	Female *m*; Male *s*; CL *l*

Table 1: A complete list of the shapes and domain patterns in the family tree domain.

assume diagrammatic input free from low-level recognition errors and cannot handle realistic, messy, stroke-based input. Mahoney and Fromherz use mathematical constraints to cope with the complexities of parsing sketches of curvilinear configurations such as stick figures [11]. Shilman *et al.* [16] present a parsing method similar to our approach, with two differences. First, their work employs a spatially-bounded search for interpretations that quickly becomes prohibitively expensive. Second, their parsing method builds and scores a parse tree for each interpretation independently; we allow competing interpretations to influence each other.

As the user draws, our system uses a two-stage generate and test recognition process to parse the strokes into possible interpretations. This two-dimensional parsing problem presents a challenge for a real-time system. Noise in the input makes it impossible for the system to recognize low-level shapes with certainty or to be sure whether or not constraints hold. Low-level misinterpretations cause higher-level interpretations to fail as well. On the other hand, trying all possible interpretations of the user's strokes guarantees that an interpretation will not be missed, but is infeasible due to the exponential number of possible interpretations.

To solve this problem we use a combined bottom-up and top-down recognition algorithm that generates the most likely interpretations first (bottom-up), then actively seeks out parts of those interpretations that are still missing (top-down). Our approach uses a novel application of dynamically constructed Bayesian networks to evaluate partial interpretation hypotheses and then expands the hypothesis space by exploring the most likely interpretations first. The system does not have to try all combinations of all interpretations, but can focus on those interpretations that contain at least a subset of easily-recognizable subshapes and can recover any low-level subshapes that may have been mis-recognized.

3.3 Hypothesis Evaluation

Our method of exploring the space of possible interpretations depends on our ability to assess partial hypotheses. We

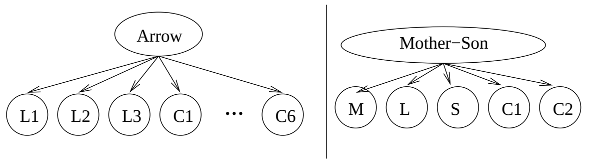


Figure 4: The Bayesian network fragment constructed from the description of an arrow and the domain pattern "mother-son" given in Figure 3.

use a dynamically constructed Bayesian network to evaluate the current set of hypothesized interpretations. (We discuss below how these hypotheses are generated.) We give an overview and illustration of this method here; more details are presented in [1].

Bayesian networks provide a framework for combining multiple sources of evidence to reason about uncertainty in the world. They consist of two parts: a directed acyclic graph that encodes *which* factors in the world influence each other, and a set of conditional probability distributions that specify *how* these factors influence one another. Each node in the graph represents something to be measured, and a link between two nodes indicates that there is a causal relationship from one node to another. Each node contains a conditional probability table specifying how it is influenced by its parents. For more information, see [4], which provides an intuitive overview of Bayesian networks.

Bayesian networks are traditionally used to model static domains in which the variables and relationships between those variables are known in advance. Static networks are not suitable for the task of sketch recognition because we cannot predict *a priori* the number of strokes or symbols the user will draw. Therefore, our network structure must be changed to reflect each new stroke. To allow the network to grow as new data arrives, we specify a library of Bayesian network fragments that describe shapes and domain patterns. This framework is similar to the Object-Oriented Bayesian Networks proposed in [8] but has been developed specifically for use in a real-time recognition system. As an example of our representation, the fragments for the descriptions from Figure 3 are given in Figure 4. As the recognition system proposes interpretations for the user's strokes, it makes copies of the corresponding fragments from the library and links them together to form a complete Bayesian network (as in Figure 5). Each node, n, in the network has two values, `true` and `false`, and represents a possible interpretation for a subset of the user's strokes or a constraint between interpretations. $P(n = \texttt{true})$ reflects the strength of that interpretation. The complete network contains the set of all the interpretations the system is considering.

Recall that links in the Bayesian network indicate causal relationships, so the arrow fragment in Figure 4 represents the hypothesis that the user intended to draw an arrow, which in turn "caused" her to produce three lines that obey a corresponding set of constraints (i.e. they are connected, two of them are the same length, etc.). Similarly, the user's intent to draw a mother-son relationship caused her to draw a male, a female and a child-link, which in turn caused her to draw an ellipse, an arrow and a quadrilateral, and so forth.

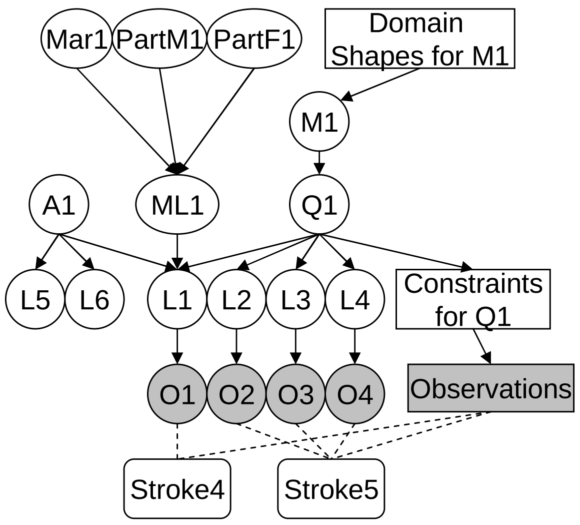


Figure 5: A portion of the interpretation network generated while recognizing the sketch in Figure 2. Abbreviations are given in Table 1

To explain the semantics of the Bayesian network further, we consider a piece of the network that is generated in response to Strokes 4 and 5 in the example given in Figure 2. Figure 5 shows the part of the Bayesian network representing the possible interpretations that the system generated for these strokes (which we call the *interpretation network*). Each node represents a hypothesized interpretation for some piece of the sketch. For example, Q1 represents the system's hypothesis that the user intended to draw a quadrilateral. A higher level hypothesis is compatible with the lower level hypotheses it points to. For example, if M1 (the hypothesis that the user intended to draw a male) is correct, Q1 (the hypothesis that the user intended to draw a quadrilateral) and L1-L4 (the hypotheses that the user intended to draw 4 lines) will also be correct. Two hypotheses that both point to the same lower level hypothesis represent competing interpretations for the lower level shape and are incompatible. For example, A1, ML1 and Q1 are three possible higher level interpretations for line L1, only one of which may be true.

Hypotheses are linked to stroke data with observation nodes that represent measurements taken from one or more strokes. For example, L1 is linked to O1, which is measured from Stroke 4. Consequently, L1 represents the hypothesis that Stroke 4 is a line, not simply that there is a line somewhere on the page. A1 is a partial hypothesis–it represents the hypothesis that L1 (and hence Stroke 4) is part of an arrow whose other two lines have not yet been drawn. Line nodes representing lines that have not been drawn (L5 and L6) are not linked to observation nodes because there is no stroke from which to measure these observations. We refer to these nodes (and their corresponding interpretations) as *virtual*.

The probability of each interpretation is influenced both by stroke data (through its children) and by the context in which it appears (through its parents), allowing the system to handle noise in the drawing. For example, the bottom edge of quadrilateral Q1 is slightly curved (see Figure 2); stroke data (O4) only weakly supports the corresponding line hypothesis (L4). However, the other three edges of Q1 are fairly straight, and O1-O3 raise the probabilities of L1-L3, respectively, which in turn raise the probability of Q1. Q1 provides a context in which to evaluate L4, and because Q1 is well supported by L1-L3 (and by the constraint nodes), it raises the probability of L4.

The fact that partial interpretations have probabilities allows the system to assess the likelihood of incomplete interpretations based on the evidence it has seen so far. In fact, even virtual nodes have probabilities, corresponding to the probability that the user (eventually) intends to draw these shapes but either has not yet drawn this part of the diagram or the correct low-level hypotheses have not yet been proposed because of low-level recognition errors. As we describe below, a partial interpretation with a high probability cues the system to examine the sketch for possible missed low-level interpretations.

3.4 Hypothesis Generation

The major challenge in hypothesis generation is to generate the correct interpretation as a candidate hypothesis without generating too many to consider in real-time. A naive approach to hypothesis generation simply would attempt to match all shapes to all possible combinations of strokes, but this would produce an exponential number of interpretations. Our method of evaluating partial interpretations allows us to use a bottom-up/top-down generation strategy that greatly reduces the number of hypotheses considered but still generates the correct interpretation for most shapes in the sketch.

Our hypothesis generation algorithm has three steps:

1. **Bottom-up step**: As the user draws, the system parses the strokes into primitive objects using a domain-independent recognition toolkit developed in previous work [15]. Compound interpretations are hypothesized for each compound object that includes these low-level shapes, even if not all the subshapes of the pattern have been found.
2. **Top-down step**: The system attempts to find subshapes that are missing from the partial interpretations generated in step 1, often by reinterpreting strokes that are temporally and spatially proximal to the proposed shape.
3. **Pruning step**: The system removes unlikely interpretations.

This algorithm, together with the Bayesian network representation presented above, deals successfully with the challenges presented in Section 2. Using the example in Figure 2, we illustrate how the system generates hypotheses that allow the Bayesian network mechanism to resolve noise and inherent ambiguity in the sketch, how the system manages the number of potential interpretations for the sketch, how the system recovers from low-level recognition errors, and how the system allows for variation in drawing style.

Based on low-level interpretations of a stroke, the bottom-up step generates a set of hypotheses to be evaluated using the Bayesian network mechanism presented in the previous section. In the sketch in Figure 2, the user's first stroke is correctly identified as an ellipse by the low-level recognizer, and from that ellipse the system generates the interpretation `ellipse`, and in turn, partial interpretations for mother-son,

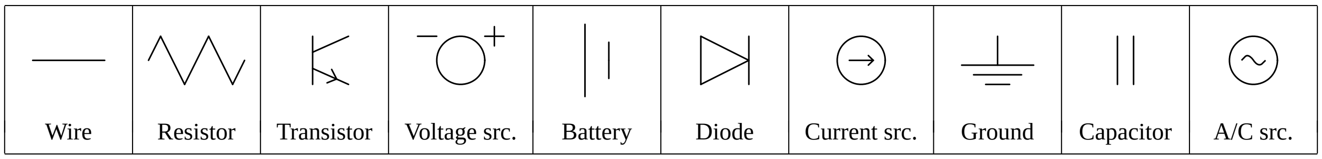


Figure 6: The symbols in the circuit domain.

mother-daughter, father-daughter, marriage, partner-female, and divorce. These proposed interpretations are called *templates* that have a *slot* for each subshape. Future interpretations will be filled into empty slots.

Naive bottom-up interpretation easily can generate too many hypotheses to consider in real-time. We employ three strategies to control the number of hypotheses generated in the bottom-up step. First, when an interpretation can be fit into more than one slot in a higher-level template (e.g., in Figure 5, L1 could be the shaft or either of the lines in the head of A1), the system arbitrarily chooses one of the valid slots rather than generating one hypothesis for each potential fit. Later, the system can shuffle the shapes in the template when it attempts to fit more subshapes.

Second, the system does not generate higher-level interpretations for interpretations that are only partially filled. The lines generated from Strokes 4 and 5 result in one partial hypothesis—`arrow` (A1)—and two complete hypotheses—`quadrilateral` (Q1) and `marriage-link` (ML1) (Figure 5). Continuing to generate higher-level templates from partial hypotheses would yield a large number of hypotheses (one hypothesis for each higher level domain pattern involving each existing partial hypothesis). To avoid this explosion, the system continues to generate templates using only the complete hypotheses (in this case, ML1 and Q1).

Third, when the system processes polylines, it assumes that all the lines in a single polyline will be used in one interpretation. While this assumption does not always hold, in practice we find that it is often true and greatly reduces the number of possible interpretations. The system recognizes Stroke 2 as a 4-line polyline. The bottom-up step generates only a quadrilateral because that is the only shape in the domain that requires four lines.

The top-down step allows our system to recover from low-level recognition errors. Stroke 3 is incorrectly, but reasonably, parsed into 5 lines by the low-level recognizer. Because the system does not know about any 5-line objects, but does know about things that contain fewer than 5 lines, it attempts to re-segment the stroke into 2 lines, 3 lines and 4 lines (with a threshold on acceptable error). It succeeds in re-segmenting the stroke into 4 lines and successfully recognizes the lines as a quadrilateral. Although the 4 line fit is not perfect, the network allows the context of the quadrilateral in addition to the stroke data to influence the system's belief in the 4-line interpretation. Also note that the 5 lines from the original segmentation remain in the interpretation network.

The system controls the number of interpretations in the network through pruning, which occasionally causes it to prune a correct hypothesis before it is complete. The top-down step regenerates previously-pruned hypotheses, allowing the system to correctly interpret a symbol despite variations in drawing order. The leftmost arrow in Figure 2 was drawn with two non-consecutive strokes (Strokes 8 and 12). In response to Stroke 8, the system generates both an arrow partial hypothesis and a marriage-link hypothesis (using the line hypothesis generated for this stroke). Because the user does not immediately complete the arrow, and because the competing marriage-link hypothesis is complete and has a high probability, the system prunes the arrow hypothesis after Stroke 9 is drawn. Later, Stroke 12 is interpreted as a 2-line polyline and a new arrow partial hypothesis is generated. The top-down step then completes this arrow interpretation using the line generated previously from Stroke 8, effectively regenerating a previously pruned interpretation.

3.5 Selecting an Interpretation

As each stroke is drawn, the sketch system uses a greedy algorithm to select the best interpretation for the sketch. It queries the Bayesian network for the strongest complete interpretation, sets aside all the interpretations inconsistent with this choice, chooses the next most likely remaining domain interpretation, and so forth. It leaves strokes that are part of partial hypotheses uninterpreted. Although the system selects the most likely interpretation at every stroke, it does not eliminate other interpretations. Partial interpretations remain and can be completed with the user's subsequent strokes. Additionally, the system can change its interpretation of a stroke when more context is added.

4 APPLICATION AND RESULTS

Applying SketchREAD to a particular domain involves two steps: specifying the structural descriptions for the shapes in the domain and specifying the prior probabilities for the domain patterns and any top-level shapes (i.e., those not used in domain patterns, which, consequently, will not have parents in the generated Bayesian network. See [1] for details on how probabilities are assigned to other shapes). We applied SketchREAD to two domains: family trees and circuits. For each domain, we wrote a description for each shape and pattern in that domain (Figures 1 and 6) and estimated the necessary prior probabilities by hand. Through experimentation, we found the recognition performance to be insensitive to the exact values of these priors.

4.1 Data Collection

SketchREAD can recognize simple sketches nearly perfectly in both the family tree and circuit domains, but we wanted to test its performance on more complex, real-world data. Our goal was to collect sketches that were as natural and unconstrained as the types of sketches people produce on paper to test the limits of our system's recognition performance. To collect these sketches, we used a data collection program for the Tablet PC developed by others in our group that allows the user to sketch freely and displays the user's strokes

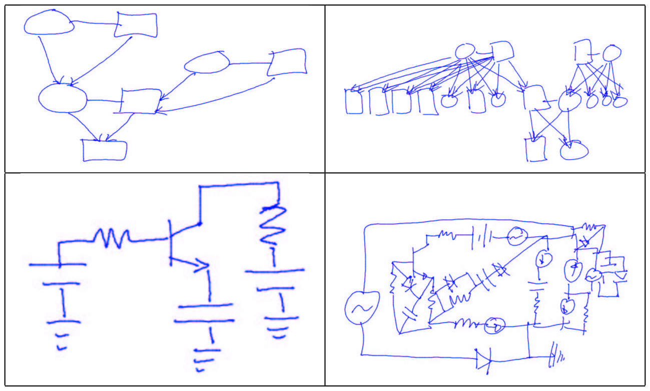


Figure 7: Examples that illustrate the range of complexity of the sketches collected.

exactly as she draws them, without performing any type of recognition. Most of our users had played with a Tablet PC before they performed our data collection task but had never used one for an extended period of time. None used any type of pen-based computer interface on a regular basis. The users first performed a few warm-up tasks, at the end of which all users expressed comfort drawing on the Tablet PC.

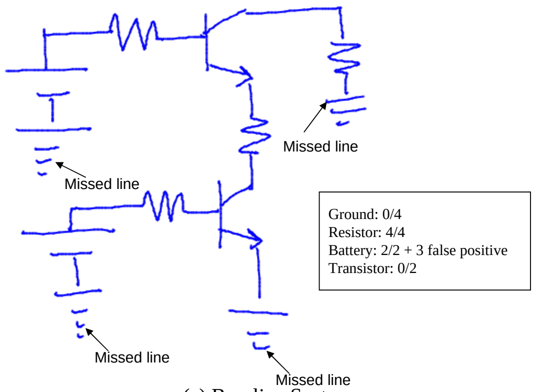
To collect the family tree sketches, we asked each user to draw her family tree using the symbols presented in Figure 1. Users were told to draw as much or as little of the tree as they wanted and that they could draw the shapes however felt natural to them. Because erasing strokes introduces subtleties into the recognition process that our system is not yet designed to deal with, users were told that they could not erase, and that the exact accuracy of their family tree diagram was not critical. We collected ten sketches of varying complexity.

We then recruited subjects with basic knowledge of circuit diagram construction and showed them examples of the types of circuits we were looking for. After a warm-up task, subjects were instructed to draw several circuits. We specified the number and types of components to be included in the circuit and then and asked them to design any circuit using those components. Subjects were instructed not to worry about the functionality of their circuit, only that they should try to produce realistic circuits. We collected 80 diagrams in all.

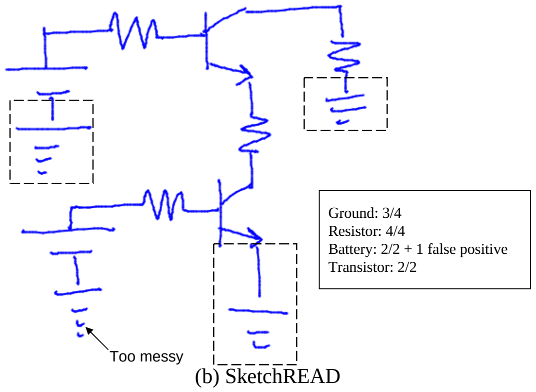
The circuit diagrams were considerably more complicated than the family tree diagrams. One limiting assumption that SketchREAD currently makes is that the user will not draw more than one symbol with a single stroke. Unfortunately, in drawing circuit diagrams, users often draw many symbols with a single stroke. To allow SketchREAD to handle the circuit diagrams, we broke apart strokes containing multiple objects by hand. This is clearly a limitation of our current system; we discuss below how it might be handled.

5 Performance Results

We ran SketchREAD on each family tree sketch and each circuit sketch. We present qualitative results, as well as aggregate recognition and running time results for each domain. Our results illustrate the complexity our system can currently handle, as well as the system's current limitations. We discuss those limitations below, describing how best to use the system in its current state and highlighting what needs to be done to make the system more powerful. Note that to apply the system to each domain, we simply loaded the domain's shape information; we did not modify the recognition system. Although SketchREAD does not perform perfectly on every sketch, its generality and performance on complex sketches illustrates its promise over previous approaches.



(a) Baseline System



(b) SketchREAD

Figure 8: Recognition performance example. Overall recognition results (# correct / total) are shown in the boxes.

Figure 8 illustrates how our system is capable of handling noise in the sketch and recovering from missed low-level interpretations. In the baseline case, one line from each ground symbol was incorrectly interpreted at the low-level, causing the ground interpretations to fail. SketchREAD was able to reinterpret those lines using the context of the ground symbol in three of the four cases to correctly identify the symbol. In the fourth case, one of the lines was simply too messy, and SketchREAD preferred to (incorrectly) recognize the top two lines of the ground symbol as a battery.

In evaluating our system's performance, direct comparisons with previous work are difficult, as there are few (if any) published results for this type of recognition task. The closest published sketch recognition results are for the Quickset system, which also uses top-down information (through multi-modal input) to recognize sketched symbols, but this system assumes object segmentation, making its recognition task different from ours [20]. We compared SketchREAD's recognition performance with the performance of a strictly bottom-up approach of the sort used in previous systems

	Size	#Shapes	% Correct	
			BL	SR
Mean	**50**	**34**	**50**	**77**
S1	24	16	75	100
S2	28	16	75	87
S3	29	23	57	78
S4	32	22	31	81
S5	38	31	54	87
S6	48	36	58	78
S7	51	43	26	72
S8	64	43	49	74
S9	84	49	42	61
S10	102	60	57	80

Table 2: Recognition rates for the baseline system (BL) and SketchREAD (SR) for each sketch for the family tree domain. The size column indicates the number of strokes in each sketch.

	Total	% Correct		# False Pos	
		BL	SR	BL	SR
AC Source	4	100	100	35	29
Battery	96	60	89	56	71
Capacitor	39	56	69	27	14
Wire	1182	62	67	478	372
Ground	98	18	55	0	5
Resisitor	330	51	53	7	8
Voltage Src.	43	2	47	1	8
Diode	77	22	17	0	0
Current Src.	44	7	16	0	0
Transistor	43	0	7	0	14

Table 3: Aggregate recognition rates for the baseline system (BL) and SketchREAD (SR) for the circuit diagrams by shape.

[3, 13]. This strictly bottom-up approach combined low-level shapes into higher-level patterns without top-down reinterpretation. Even though our baseline system did not reinterpret low-level interpretations, it was not trivial. It could handle some ambiguities in the drawing (e.g., whether a line should be interpreted as a marriage-link or the side of a quadrilateral) using contextual information in the bottom-up direction. To encourage others to compare their results with those presented here we have made our test set publicly available at http://rationale.csail.mit.edu/ETCHASketches.

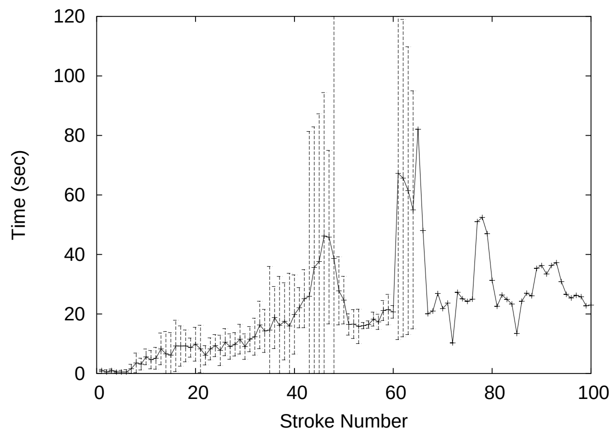
We measured recognition performance for each system by determining the number of correctly identified objects in each sketch (Table 2 and Table 3). For the family tree diagrams SketchREAD performed consistently and notably better than our baseline system. On average, the baseline system correctly identified 50% of the symbols while SketchREAD correctly identified 77%, a 54% reduction in the number of recognition errors. Due to inaccurate low-level recognition, the baseline system performed quite poorly on some sketches. Improving low-level recognition would improve recognition results for both systems; however, SketchREAD reduced the error rate by approximately 50% independent of the performance of the baseline system. Because it is impossible to build a perfect low-level recognizer, SketchREAD's ability to correct low-level errors will always be important.

Circuit diagrams present SketchREAD with more of a challenge for several reasons. First, there are more shapes in the circuit diagram domain and these shapes are more complex. Second, there is a stronger degree of overlap between shapes in the circuit diagrams. For example, it can be difficult to distinguish between a capacitor and a battery. As another example, a ground symbol contains within it (at least one) battery symbol. Finally, there is more variation in the way people draw circuit diagrams, and their sketches are messier causing the low-level recognizer to fail more often. They tend to include more spurious lines and over-tracings.

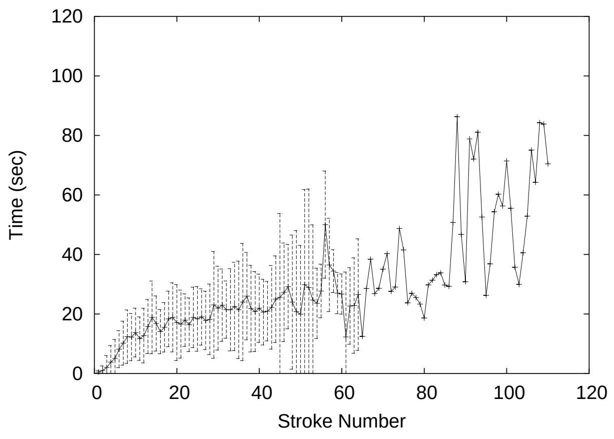
Overall, SketchREAD correctly identified 62% of the shapes in the circuit diagrams, a 17% reduction in error over the baseline system. It was unable to handle more complex shapes, such as transistors, because it often failed to generate the correct mapping between strokes and pieces of the template. Although the system attempts to shuffle subshapes in a template in response to new input, for the sake of time it cannot consider all possible mappings of strokes to templates. We discuss below how we might extend SketchREAD to improve its performance on complex domains such as circuit diagrams.

We measured SketchREAD's running time to determine how it scales with the number of strokes in the sketch. Figure 9 graphs the median time to process each stroke for each domain. The vertical bars in the graph show the standard deviation in processing time over the sketches in each domain. (One family tree diagram took a particularly long time to process because of the complexity of its interpretation network, discussed below. This sketch affected the median processing time only slightly but dominated the standard deviation. It has been omitted from the graph for clarity.) Three things about these graphs are important. First, although SketchREAD does not yet run in real-time, the time to process each stroke in general increased only slightly as the sketch got larger. Second, not every stroke was processed by the system in the same amount of time. Finally, the processing time for the circuit diagrams is longer than the processing time for the family trees.

By instrumenting the system, we determined that the processing time is dominated by the inference in the Bayesian network, and all of the above phenomena can be explained by examining the size and complexity of the interpretation network. The number of nodes in the interpretation network grows approximately linearly as the number of strokes increases. This result is encouraging, as the network would grow exponentially using a naive approach to hypothesis generation. The increase in graph size accounts for the slight increase in processing time in both graphs. The spikes in the graphs can be explained by the fact that some strokes not only increased the size of the network, but had more higher-level interpretations, creating more fully connected graph structures, which causes an exponential increase in inference time. After being evaluated, most of these high-level hypotheses were immediately pruned, accounting for the sharp drop in processing time on the next stroke. Finally, the fact that circuits take longer to process than family trees is related to the relative complexity of the shapes in the domain. There are more shapes in the circuit diagram domain and they are more complex, so the system must consider more interpretations for the user's strokes, resulting in larger and more connected Bayesian networks.



(a) Family Tree



(b) Circuit

Figure 9: The median incremental time it took the system to process each stroke in the family tree and circuit diagrams. Vertical bars show the standard deviation across the sketches in each domain.

6 SKETCH RECOGNITION USER INTERFACES (SKRUIS)

Our goal is to use SketchREAD in sketch recognition user interfaces (SkRUIs). In this section we discuss SketchREAD's strengths and limitations with two purposes. First, we discuss the type of interface in which SketchREAD can be used currently. Second, we discuss the system's limitations and how they can be addressed so that its overall performance can be improved to better handle more complicated domains.

6.1 Building Design Tools with SketchREAD

We presented results on sketches of family trees and circuits, but SketchREAD can be applied to any domain that has a clear language of symbols. Currently SketchREAD is best used in simple domains where the shapes have little overlap and are drawn spatially separated. As an example, SketchREAD was able to process the simpler family trees in near real-time. It ran into difficulty when encountering many overlapping shapes, such as the arrows the top right drawing in Figure 7. Based on these guidelines, we used SketchREAD to build a sketch-based system for constructing box and arrow diagrams in Power Point and conducted an informal user study on this system [2]. People sketched their diagrams freely while SketchREAD ran in the background. The system produced few errors in this simple domain, allowing us to investigate issues such as how and when to display recognition feedback, and how to allow people to edit their sketches.

While interpretations are selected after every stroke, determining when to display these interpretations to the user is by itself an interesting and difficult question of human computer interaction. One approach we are exploring is to use the system's determination of the completeness of an interpretation to inform it when to display an interpretation and when to wait for more strokes.

Even though SketchREAD reduces recognition error, it will never eliminate it. SketchREAD's interpretation-graph architecture could itself be used to reduce the burden placed on the user in correcting the system's errors. Often in complicated diagrams, errors are interconnected. If there are many competing interpretations, choosing the wrong interpretation for one part of the diagram often will lead the system to choose the wrong interpretation for the surrounding strokes. SketchREAD could display its recognition results on top of the user's original strokes, so that the user can see the context of these results. Then, the user could help the system by tracing over the strokes for one of the symbols that was misrecognized. This retracing would help the system recover from the error because the user's strokes would be cleaner and because the system would know that they were all part of a single symbol. Then, based on this new interpretation, the system could reevaluate the surrounding strokes and (hopefully) recover some of the missed interpretations that might still exist but simply were not currently chosen as the best interpretation.

Finally, one of the main causes of recognition error might be dealt with through UI design. The system had trouble with the fact that the users varied the structure of their symbols even though they were explicitly shown the desired structure for each symbol. For example, although we instructed people to draw a voltage source using a circle with a plus and a minus next to it, some people put the plus and minus inside the circle. SketchREAD is designed to handle variations in the way people draw, but cannot handle such unexpected changes to the basic shape of the symbol. Although ideally we would like to support all methods people have for drawing each object, this might never be possible. Instead, a simple interactive training step before a new user uses the interface could help eliminate this type of variation without imposing too many limitations on the user's drawing style.

6.2 Performance Improvement

SketchREAD significantly improves the recognition performance of unconstrained sketches. However, its accuracy, especially for complicated sketches and domains, is still too low to be practical in most cases. Here we consider how to improve the system's performance.

First, while SketchREAD always corrected some low-level interpretation errors, its overall performance still depended on the quality of the low-level recognition. Our low-level recognizer was highly variable and could not cope with some users' drawing styles. In particular, it often missed corners of polylines, particularly for symbols such as resistors. Other members of our group are working on a low-level recognizer that adapts to different users' drawing styles.

Second, although in general SketchREAD's processing time scaled well as the number of strokes increased, it occasionally ran for a long period. The system had particular trouble with areas of the sketch that involved many strokes drawn close together in time and space and with domains that involve more complicated or overlapping symbols. This increase in processing time was due almost entirely to in increase in Bayesian network complexity.

We suggest two possible solutions. First, part of the complexity arises because the system tries to combine new strokes with low-level interpretations for correct high-level interpretations (e.g., the four lines that make a quadrilateral). These new interpretations were pruned immediately, but they increased the size and complexity of the network temporarily, causing the bottlenecks noted above. In response, we are testing methods for "confirming" older interpretations and removing their subparts from consideration other higher-level interpretations as well as confirming their values in the Bayesian network so that their posterior probabilities do not have to be constantly re-computed. Second, we can modify the belief propagation algorithm we are using. We currently use Loopy Belief Propagation, which repeatedly sends messages between the nodes until each node has reached a stable value. Each time the system evaluates the graph, it resets the initial messages to one, essentially erasing the work that was done the last time inference was performed, even though most of the graph remains largely unchanged. Instead, this algorithm should begin by passing the messages it passed at the end of the previous inference step.

Finally, because our recognition algorithm is stroke-based, spurious lines and over-tracing hindered the system's performance in both accuracy and running time. A preprocessing step to merge strokes into single lines would likely greatly improve the system's performance. Also, in the circuit diagram domain, users often drew more than one object with a single stroke. A preprocessing step could help the system segment strokes into individual objects.

7 CONCLUSION

We have shown how to use context to improve online sketch interpretation and demonstrated its performance in SketchREAD, an implemented sketch recognition system that can be applied to multiple domains. We have shown that SketchREAD is more robust and powerful than previous systems at recognizing unconstrained sketch input in a domain. The capabilities of this system have applications both in human computer interaction and artificial intelligence. Using our system we will be able to further explore the nature of usable intelligent computer-based sketch systems and gain a better understanding of what people would like from a drawing system that is capable of understanding their freely-drawn sketches as more than just strokes. This work provides a necessary step in uniting artificial intelligence technology with novel interaction technology to make interacting with computers more like interacting with humans.

8 ACKNOWLEDGEMENTS

This work is supported by the Microsoft iCampus project and by MIT's Project Oxygen. We would like to thank our users for their time and the members of the MIT Design Rationale Group and Alex Snoeren for helpful editing comments. We would also like to thank Alex Snoeren and Aaron Adler for help with compiling recognition results.

REFERENCES

1. C. Alvarado. *Multi-Domain Sketch Understanding*. PhD thesis, Massachusetts Institute of Technology, 2004.
2. C. Alvarado. Sketch recognition and usability: Guidelines for design and development. In *AAAI Fall Symposium on Pen-Based Interaction*, 2004.
3. C. Alvarado and R. Davis. Resolving ambiguities to create a natural sketch based interface. In *Proc. of IJCAI*, 2001.
4. E. Charniak. Bayesian networks without tears: making bayesian networks more accessible to the probabilistically unsophisticated. *Artificial Intelligence*, 12(4):50–63, 1991.
5. K. D. Forbus, J. Usher, and V. Chapman. Sketching for military course of action diagrams. In *Proc. of IUI*, 2003.
6. M. Gross and E. Y.-L. Do. Ambiguous intentions: a paper-like interface for creative design. In *Proc. of UIST*, 1996.
7. T. Hammond and R. Davis. LADDER: A language to describe drawing, display, and editing in sketch recognition. In *Proc. of IJCAI*, 2003.
8. D. Koller and A. Pfeffer. Object-oriented bayesian networks. In *Proc. of UAI*, 1997.
9. J. A. Landay and B. A. Myers. Interactive sketching for the early stages of user interface design. In *Proc. of CHI*, 1995.
10. A. C. Long, Jr., J. A. Landay, L. A. Rowe, and J. Michiels. Visual similarities of pen gestures. In *Proc. of CHI*, 2000.
11. J. V. Mahoney and M. P. J. Fromherz. Three main concerns in sketch recognition and an approach to addressing them. In *AAAI Spring Symposium on Sketch Understanding*, 2002.
12. K. Marriott, B. Meyer, and K. Wittenburg. A survey of visual language specification and recognition. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 5–85. Springer-Verlag, 1998.
13. M. W. Newman, J. Lin, J. I. Hong, and J. A. Landay. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction*, 18(3):259–324, 2003.
14. E. Saund, D. Fleet, D. Larner, and J. Mahoney. Perceptually supported image editing of text and graphics. In *Proc. of UIST*, 2003.
15. T. M. Sezgin, T. Stahovich, and R. Davis. Sketch based interfaces: Early processing for sketch understanding. In *Proc. of PUI*, 2001.
16. M. Shilman, H. Pasula, S. Russell, and R. Newton. Statistical visual language models for ink parsing. In *AAAI Spring Symposium on Sketch Understanding*, 2002.
17. T. Stahovich, R. Davis, and H. Shrobe. Generating multiple new designs from a sketch. *Artificial Intelligence*, 104(1-2):211–264, 1998.
18. T. M. Strat and M. A. Fischler. Context-based vision: Recognizing objects using information from both 2-d and 3-d imagery. *IEEE Trans. on PAMI*, 13(10):1050–1065, 1991.
19. A. Torralba and P. Sinha. Statistical context priming for object detection. In *Proc. of ICCV*, 2001.
20. L. Wu, S. L. Oviatt, and P. R. Cohen. Multimodal integration—a statistical view. *IEEE Trans. on Multimedia*, 1(4):334–341, 1999.

Available online at www.sciencedirect.com
SCIENCE DIRECT
Computers & Graphics 29 (2005) 518–532
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

LADDER, a sketching language for user interface developers

Tracy Hammond[a,*], Randall Davis[b]

[a]MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., 32-239, Cambridge, MA 02141, USA
[b]MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., 32-237, Cambridge, MA 02141, USA

Abstract

Sketch recognition systems are currently being developed for many domains, but can be time consuming to build if they are to handle the intricacies of each domain. In order to aid sketch-based user interface developers, we have developed tools to simplify the development of a new sketch recognition interface. We created LADDER, a language to describe how sketched diagrams in a domain are drawn, displayed, and edited. We then automatically transform LADDER structural descriptions into domain specific shape recognizers, editing recognizers, and shape exhibitors for use in conjunction with a domain independent sketch recognition system, creating a sketch recognition system for that domain. We have tested our framework by writing several domain descriptions and automatically generating a domain specific sketch recognition system from each description.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Object recognition; Object modeling; Shape; Hierarchical scene analysis; Knowledge representation; Representation languages; Representations; Vision and scene understanding; User-centered design

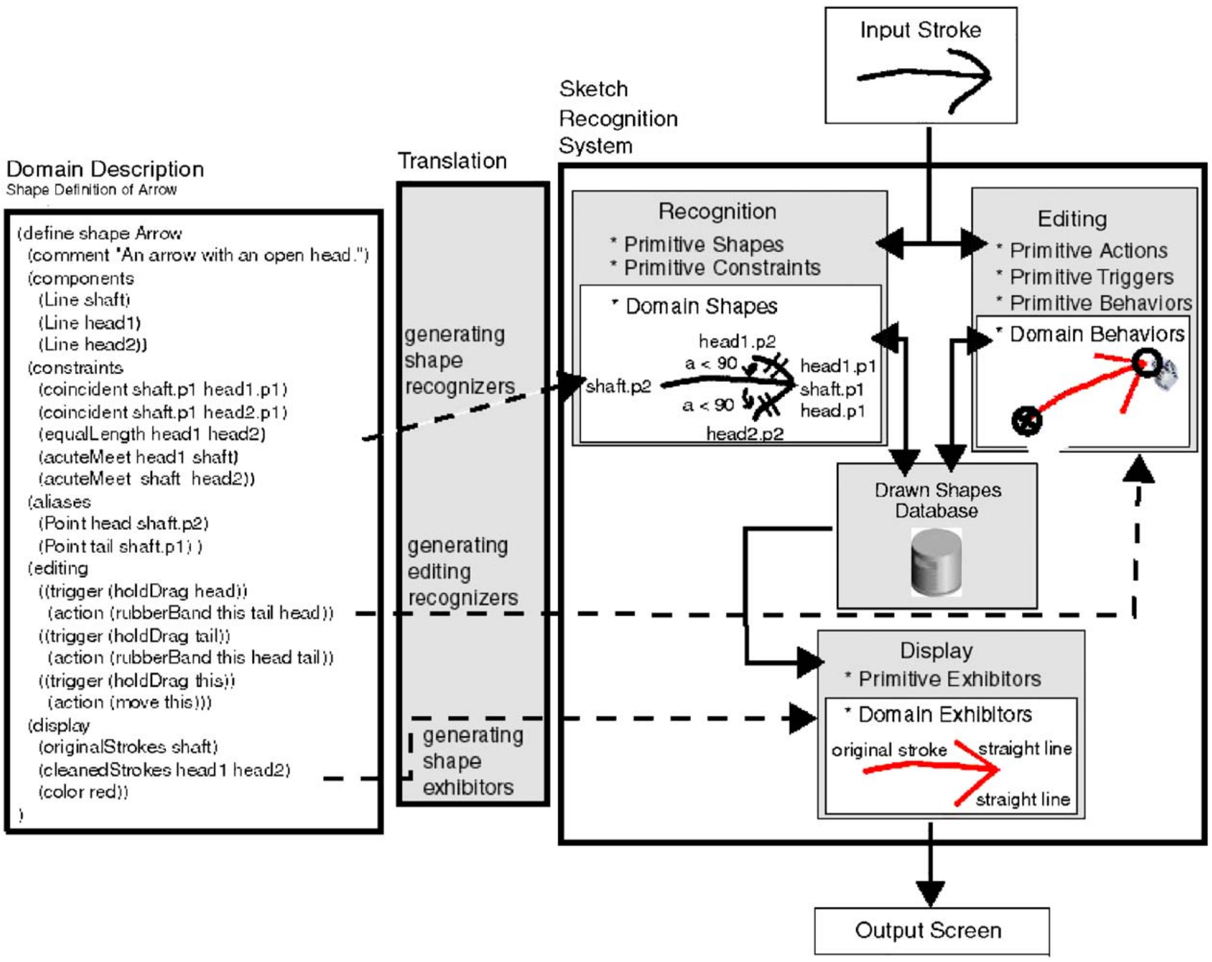
1. Introduction

As pen-based input devices have become more common, sketch recognition systems are being developed for many hand-drawn diagrammatic domains such as mechanical engineering [1–3], UML class diagrams [4–7], webpage design [8], GUI design [9,10], virtual reality [11], stick figures [12], course of action diagrams [13], and many others. These sketch interfaces (1) allow for more natural interaction than a traditional mouse and palette tool [14] by allowing users to hand sketch the diagram, (2) can automatically connect to a CAD system preventing the designer from having to enter the same information twice, (3) can offer real-time design advice from CAD systems, (4) allow more powerful editing since the shape is recognized as a whole, (5) provide diagram beautification to remove mess and clutter, and (6) use display as a trigger to inform the sketcher that the shapes have been correctly recognized. However, sketch recognition systems can be quite time consuming to build if they are to handle the intricacies of each domain. Also we would prefer that the builder of a sketch recognition system be an expert in the domain rather than an expert in sketch recognition at a signal level. Rather than build each recognition system separately, our group has been working on a multi-domain recognition system that can be customized for each domain.

Using our framework, in order to build a sketch recognition system for a new domain, a developer need only write a domain description which describes what the domain shapes look like, and how they should be displayed and edited after they are recognized. Thus, the writer of the domain description does not need to know how to program a system to perform sketch recognition. This domain description is then automatically translated

*Corresponding author.
E-mail addresses: hammond@csail.mit.edu (T. Hammond), davis@csail.mit.edu (R. Davis).

0097-8493/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2005.05.005



```
(define shape Arrow
  (comment "An arrow with an open head.")
  (components
    (Line shaft)
    (Line head1)
    (Line head2))
  (constraints
    (coincident shaft.p1 head1.p1)
    (coincident shaft.p1 head2.p1)
    (equalLength head1 head2)
    (acuteMeet head1 shaft)
    (acuteMeet shaft head2))
  (aliases
    (Point head shaft.p2)
    (Point tail shaft.p1) )
  (editing
    ((trigger (holdDrag head))
     (action (rubberBand this tail head))
    ((trigger (holdDrag tail))
     (action (rubberBand this head tail))
    ((trigger (holdDrag this))
     (action (move this)))
  (display
    (originalStrokes shaft)
    (cleanedStrokes head1 head2)
    (color red))
)
```

Fig. 1. System framework.

into shape recognizers, editing recognizers, and shape exhibitors for use with the customizable base domain independent recognition system creating a domain specific sketch interface that recognizes the shapes in the domain, displaying them and allowing them to be edited as specified in the description. The inspiration for such a framework stems from work in speech recognition [15,16], which has used this approach with some success.

This paper describes LADDER, the first sketch description language that can be used to describe how shapes and shape groups are drawn, edited, and displayed, and a first implemented prototype system that proves that such a framework is possible: that we can automatically generate a sketch interface for a domain from only a domain description. This work also shows that LADDER [17] is an acceptable language for describing sketch interfaces and enables us to automatically generate a sketch interface from only a LADDER domain description. To accomplish our goal, we have built (1) LADDER, a symbolic language to describe how shapes are drawn, displayed, and edited in a domain, (2) a base customizable multi-domain recognition system, and (3) a code generator [18] that parses a LADDER domain description and generates Java and Jess code to be used by the base recognition system so that it can recognize, display, and edit domain shapes. Fig. 1 shows how all three parts of the system fit together.

2. LADDER

LADDER allows interface designers to describe how shapes in a domain are drawn, displayed, and edited. LADDER descriptions primarily concern shape, but may include other information helpful to the recognition process, such as stroke order or stroke direction. The specification of editing behavior allows the system to determine when a pen gesture is intended to indicate editing rather than a stroke. Display information indicates what to display after strokes are recognized.

The language consists of predefined shapes, constraints, editing behaviors, and display methods, as well as a syntax for specifying a domain description. The difficulty in creating such a language is ensuring that domain descriptions are easy to specify, and that the descriptions provide enough detail for accurate sketch recognition. To simplify the task of creating a domain description, shapes can be built hierarchically, reusing low-level shapes. Shapes can extend abstract shapes, which describe shared shape properties, preventing the application designer from having to redefine these properties several times. The language has proven powerful enough to describe shapes from several domains. The language enables more accurate sketch recognition by supporting both top-down and bottom-up recognition. Descriptions of how shapes may combine can aid in top-down recognition and can be used to describe "chain reaction" editing commands.

A shape definition is structural and includes primarily geometric information, but can include other drawing information that may be helpful to the recognition process, such as stroke order or stroke direction.[1] We can specify that the shaft of an arrow must be drawn before the two lines representing the head with `(drawOrder shaft head1 head2)`. We can use the same constraint to specify stroke direction; for instance `(drawOrder shaft.p2 shaft.p1)` requires that the tail of the arrow be drawn before the head.

LADDER allows the developer to specify both hard and soft constraints. Hard constraints must be satisfied for the shape to be recognized, but soft constraints may not be. Soft constraints can aid recognition by specifying relationships that usually occur. For instance, in the left box of Fig. 1, we could have specified `soft(drawOrder shaft head1 head2)` to specify that the shaft of the arrow is commonly drawn before the head, but the arrow should still be recognized even if this is not satisfied.

Before creating the language, we performed a user study where 30 people described shapes with their natural vocabulary and with increasing levels of syntactical constraints in order to ensure an intuitive vocabulary and syntax. We chose a hierarchical symbolic shape-based language as we found it to be more intuitive to describe shapes in this manner, making descriptions easier to create, understand, and correct. We also noticed that not only are shape-based geometrical properties more intuitive than feature-based properties such as those used by [19,20] (since shape is the salient feature used in human recognition), but since the features (and thus recognition) are not based on drawing style, sketchers are able to draw as they do naturally, with no constraints on stroke number, order, or direction.

[1]This enables us to also describe sketching languages such as the Graffiti language for the Palm Pilot.

LADDER is the first language that not only can define how shapes are to be recognized, but also can define how shapes are displayed and edited. Display and editing are important parts of a sketch interface, and are different in each domain. The display gives the sketcher feedback that an object was recognized and beautification can be used to remove clutter from the diagram. Because the objects are recognized we can define more powerful and intuitive editing gestures, consisting of a trigger and action, for each shape. For instance, a developer may define that an arrow can be dragged in rubber-band fashion from its head or tail, or she may define that a wheel can be moved as a whole by dragging any point within the wheel's bounding box. Although we do encourage standardization between different domains by including some predefined editing behaviors, it is important that we allow the developer to define her own editing behaviors for each domain. The same gesture, such as writing an X inside of a rectangle, may be intended as a pen stroke in the one domain (a check inside of a checkbox, or the letter X in a textbox), or as an editing command (deletion of the box).

2.1. Description limitations

LADDER can be used to describe a wide variety of shapes, but we are limited to the following class of shapes.

- LADDER can only describe shapes with a fixed graphical grammar. The shapes must be diagrammatic or iconic such that they are drawn using the same graphical components each time. For instance we cannot describe abstract shapes, such as people or cats, that would be drawn in an artistic drawing.
- The shapes must be composed solely of the primitive constraints contained in LADDER and must be differentiable from the other shapes in the language using only the constraints available in LADDER.
- Pragmatically, LADDER can only describe domains that have few curves or where the curve details are not important for distinguishing between different shapes. This is because curves are inherently different to describe in detail because of the difficulty in specifying a curve's control points. Future work includes investigating more intuitive ways of describing curves.
- Pragmatically, LADDER can only describe shapes that have a lot of regularity and not too much detail.

If a shape is highly irregular and complicated so that it cannot be broken down into subshapes which can be described, it will be cumbersome to define.

2.2. Shape definition

New shapes are defined in terms of previously defined shapes and constraints between them. An example of arrow definition is shown on the left-hand side of Fig. 1. The definition of a shape contains the following parts.

- A list of *components* specifies the elements from which the shape is built. Note that the arrow is built from three lines.
- Geometric *constraints* define the relationships on those components. The arrow definition requires that the HEAD1 and SHAFT meet at a single point and form an acute angle in a counter-clockwise direction from HEAD1 to SHAFT. (Angles are measured in a counter-clockwise direction.)
- A set of *aliases* is used to simplify other elements in the description. The HEAD and TAIL have been added as aliases in the arrow definition to more easily specify the editing behaviors.
- *Editing* behaviors specify the editing gestures triggers and how the object should react to these editing gestures. The arrow definition specifies three editing behaviors: dragging the head, dragging the tail, and dragging the entire arrow. Each editing behavior consists of a trigger and an action. Each of the three defined editing commands are triggered when the sketcher places and holds the pen on the head, tail, or shaft, and then begins to drag the pen. The actions for these editing commands specify that the object should follow the pen either in a rubber-band fashion for the head or tail of the arrow or by translating the entire shape.[2]
- *Display* methods indicate what to display when the object is recognized. A shape or its components may be displayed in any color in four different ways: (1) the original strokes of the shape, (2) the cleaned-up version of the shapes, where the best-fit primitives of the original strokes are displayed, (3) the ideal shape, which displays the primitive components of the shape with the constraints solved, or (4) another custom shape that specifies which shapes (line, circle, rectangle, etc.) to draw and where. The arrow definition specifies that the arrow should be displayed in the color red, that head1 and head2 should be drawn using CLEANEDSTROKES (a straight line in this case), and that the shaft should be drawn using the original strokes.

[2]Rubber-banding allows sketchers to simultaneously rotate and scale an object, assuming a fixed rotation point is defined. This action has proved useful for editing arrows and other linking shapes.

The domain description is translated into shape recognizers (from the *components* and *constraints* sections), exhibitors (from the *display* section), and editors (from the editing section) which are used in conjunction with a customizable recognition system to create a domain sketch interface.

2.2.1. Hierarchical shape definitions

To simplify shape definitions, shapes can be defined hierarchically. For example, the TRIANGLEARROW in Fig. 2 is composed of an ARROW and a LINE.

2.2.2. Abstract shape definitions

In the domain of UML class diagrams, there are four different types of arrows: the regular arrow, an arrow with a triangle head, an arrow with a diamond head, and an arrow with a dashed shaft. All of these arrows have the same editing behaviors. Rather than repeat the editing behaviors four times, we instead create an ABSTRACTARROW (shown in Fig. 3 which specifies the repeated editing behaviors). The *is-a* section, used in Fig. 2, specifies any class of abstract shapes that the shape may be a part of. This is similar to the extends property in Java. All shapes extend the abstract shape SHAPE. Abstract shapes have no concrete shape associated with them; they represent a class of shapes that have similar attributes or editing behaviors. An abstract shape is defined similarly to a regular shape, except it has a *required* section instead of a *components* section. Each shape that extends the abstract shape must define each variable listed in the *required* section, in its *components* or *aliases* section.

2.2.3. Shape groups

A shape group is a collection of domain shapes that are commonly found together in the domain. Defining shape groups provides two significant benefits. Shape groups can be used by the recognition system to provide top-down recognition, and "chain reaction" editing behaviors can be applied to shape groups, allowing the movement of one shape to cause the movement of another. Below we have an example describing a shape group consisting of a FORCE and a BODY (a mechanical engineering term describing a physical mass). In the

```
(define shape TriangleArrow
  (description   "An arrow with a triangle-shaped head")
  (is-a AbstractArrow)
  (components
    (Arrow a)
    (Line head3))...)
```

Fig. 2. Description for an arrow with a triangle-shaped head.

```
(define abstract-shape AbstractArrow
  (required
    (Point head)
    (Point tail)
    (Line shaft))
  (editing
   ((trigger (holdDrag shaft))
     (action  (translate this))
   ((trigger (holdDrag head))
     (action (rubberBand this head tail))
   ((trigger (holdDrag tail))
     (action (rubberBand this tail head))))
```

Fig. 3. Description for the abstract class AbstractArrow.

```
(define shape Force
  (description "An arrow is a force only if the arrow head is
 pushing an object.")
  (component (Arrow a))
  (aliases (Point head a.head) (Point tail a.tail)))

(define shape Body
  (description "Any polygon")
  (component(Polygon p)))

(define shape-group ForcePushObject
  (components
    (Force f)
    (Body b))
  (constraints
    (meet f.head b)
    (drawOrder b f) ))
```

Fig. 4. Definition of a shape group for the force/body relationship in mechanical engineering.

mechanical domain, forces *push* bodies. Forces are represented by arrows and objects are represented by polygons. If a force is said to be pushing an object, then an arrow is pointing to the polygon. The shape group ForcePushObject defined in Fig. 4 states that the head of the arrow touches the body. It also specifies that the body must be drawn before the force. If a single shape in a sketch can be part of many instances of a shape group, then we place the key word *shared* before the component shape of the shape group (e.g. if a body could have several forces we would place the word *shared* in front of the `(Body b)` to show `shared(Body b)`. We can also define abstract shape groups.

2.3. Language contents

The power of the language is derived in part from carefully chosen predefined building blocks. The language consists of predefined shapes, constraints, editing behaviors, and display methods.

2.3.1. Predefined shapes

The language includes the primitive shapes Shape, Point, Path, Line, BezierCurve, Curve, Arc, Ellipse, and Spiral. The language also includes a library of predefined shapes built from these primitives including Rectangle, Diamond, etc. The language uses an inheritance hierarchy; Shape is an abstract shape which all other shapes extend. Shape provides a number of components and properties for all shapes, including *boundingbox*, *centerpoint*, *width*, and *height*. Each predefined shape may have additional components and properties; a Line, for example, also has *p1*, *p2* (the endpoints), *midpoint*, *length*, *angle*, and *slope*. Components and properties for a shape can be used hierarchically in shape descriptions. When defining a new shape the components and properties are those defined by Shape, and those defined by the *components* and *aliases* section.

2.3.2. Predefined constraints

A number of predefined constraints are included in the language, including *perpendicular*, *parallel*, *collinear*, *sameSide, oppositeSide*, *coincident*, *connected*, *meet*, *intersect*, *tangent, contains*, *concentric*, *larger*, *near*, *drawOrder*, *equalLength, equal*, *lessThan*, *lessThanEqual*, *angle*, *angleDir*, *acute*, *obtuse, acuteMeet,* and *obtuseMeet*. If a sketch grammar consists of only the constraints above, the shape is rotationally invariant.

There are also predefined constraints that are valid only in a particular orientation, including *horizontal*, *vertical, posSlope*, *negSlope*, *leftOf*, *rightOf*, *above*, *below*, *sameXPos, sameYPos*, *aboveLeft*, *aboveRight*, *belowLeft*, *belowRight*, *centeredBelow*, *centeredAbove*, *centeredLeft*, *centeredRight*, and *angleL*, where (*angleL line1 degrees*) specifies that the angle between a horizontal line pointing right and *line1* is *degrees*.

We have found that it is easier for many developers to describe shapes in an orientation dependent fashion. However, since the developer may still want a shape to be recognizable in any orientation, the language allows a developer to describe shapes in an orientation dependent fashion and then specify that the shape is rotatable. For this purpose, the language contains an additional constraint: *isRotatable*, which implies the shape can be found in any orientation. If *isRotatable* is specified along with an orientation dependent constraint, there must be an *angleL*, *horizontal*, or *vertical* constraint specified, which serves to define the orientation and set a relative coordinate system. For example, the two *angleMeet* constraints could have been replaced with:

```
(isRotatable)
(horizontal shaft)
(negSlope head1)
(posSlope head2)
(leftOf shaft.p1 shaft.p2)
(leftOf head1.p2 shaft.p2)
(leftOf head2.p2 shaft.p2),
```

in which case the *shaft* is the reference line.

2.3.3. Predefined editing behaviors, actions, and triggers

Describing editing gestures permits the recognition system to discriminate between sketching (pen gestures intended to leave a trail of ink) and editing gestures (pen gestures intended to change existing ink), and permits us to describe the desired behavior in response to a gesture.

In order to encourage interface consistency, the language includes a number of predefined editing behaviors described using the actions and triggers above. One such example is *dragInside*, and defines that if you hold the pen for a brief moment inside the bounding box of a shape and then start to drag the pen (specified by the trigger `(holdDrag Shape)`, the entire shape automatically translates along with the motion of the pen.

When defining a new editing behavior particular to a domain, there are two things to specify: the trigger—what signals an editing command—and the action—what should happen when the trigger occurs. The language has a number of predefined triggers and actions to aid in describing editing behaviors.

The arrow definition in Fig. 1 defines three editing behaviors. The first editing behavior says that if you click and hold the pen over the SHAFT of the ARROW, when you drag the pen, the entire ARROW will translate along with the movement of the arrow. The second editing behavior states that if you click and hold the pen over the HEAD of the arrow, the HEAD of the arrow will follow the motion of the pen, but the TAIL of the arrow will remain fixed and the entire ARROW will stretch like a rubber-band (translating, scaling, and rotating) to satisfy these two constraints and keep the ARROW as one whole shape. All of the editing behaviors also change the pen's cursor as displayed to the sketcher, and display moving handles to the sketcher to let the sketcher know that she is performing an editing command.

The possible editing actions include *wait*, *select*, *deselect*, *color*, *delete*, *translate*, *rotate*, *scale*, *resize*, *rubberBand*, *showHandle*, and *setCursor*. To give an example: `(rubberBand shape-or-selection fixed-point move-point [new-point])` translates, scales, and rotates the *shape-or-selection* so that the *fixed-point* remains in the same spot, but the *move-point* translates to the *new-point*. If *new-point* is not specified, *move-point* translates according to the movement of the pen.

The possible triggers include *click*, *doubleClick*, *hold*, *holdDrag*, *draw*, *drawOver*, *scribbleOver*, and *encircle*. Possible triggers also include any action listed above, to allow for "chain reaction" editing.

Shape groups allow designers to define "chain reaction" editing behaviors. For instance, the designer may want to specify that when we move a rectangle, if there is an arrowhead inside of this rectangle, the arrow should move with the rectangle.

2.3.4. Predefined display methods

An important part of a sketching interface is controlling what the sketcher sees after shapes are recognized, both of which can be used to clean up the sketch as desired for the domain and provide feedback to the sketcher that a shape has been recognized. The designer can specify that the original strokes should remain, or instead that a cleaned version of the strokes should be displayed. In the cleaned version, the original strokes are fit to straight lines, clean curves, clean arcs, or a combination.

Another option is to display the ideal version of the strokes where the constraints listed in the definition are solved. In this case, lines that are supposed to connect at their end points actually connect and lines that are supposed to be parallel are actually shown as parallel. In the ideal version of the strokes, all of the low-level signal noise from sketching is removed.

It may be that we do not want to show any version of the strokes at all, but some other picture. In this case, we can either place an image at a specified location, size, and rotation (using the method IMAGE), or we can create a picture built out of predefined shapes, such as circles, lines, and rectangles.

The predefined display methods include *originalStrokes*, *cleanedStrokes*, *idealStrokes*, *circle*, *line*, *point*, *rectangle*, *text*, *color*, and *image*. Each method includes color as an optional argument.

2.4. Vectors

The arrow defined in Fig. 1 contains a fixed number of components (3). However, many shapes that we would like to define, such as a POLYGON, POLYLINE, or DASHEDLINE, contain a variable number of components. A POLYLINE may contain a variable number of line segments. A variable number of components is specified by the key word *vector* and must specify the minimum and maximum number of components. If the maximum number can be infinite, the variable *n* is listed. For instance, the POLYLINE must contain at least two lines, and each line must be connected with the previous. The definition of a POLYGON easily follows from the definition of the POLYLINE (both are defined in Fig. 5).

Likewise, a DASHEDARROW is made from an ARROW, and a DASHEDLINE (both defined in Fig. 6), which in turn contains at least two line segments. When given a third argument specifying a length, the constraint *near* states that two points are near to each other relative to a given length.

```
(define shape PolyLine
  (components (vector Line vl[2,n]))
  (constraints (coincident vl[i].p2 vl[i+1].p1))
  (aliases (Point head vl[0].p1)(Point tail vl[n].p2)))

 (define shape Polygon
  (components(PolyLine poly))
  (constraints(coincident poly.head poly.tail)))
```

Fig. 5. Shape description of a polygon.

```
(define shape DashedLine
  (components (vector Line vl[2,n]))
  (constraints (collinear vl[i].p1 vl[i].p2 vl[i+1].p1)
    (not(intersect vl[i] vl[i+1])
    (near vl[i].p2 vl[i+1].p1 vl[i].length))
  (aliases (Point head vl[0].p1)(Point tail vl[n].p2)))

(define shape DashedOpenArrow
  (components (OpenArrow oa)(DashedLine dl))
  (constraints (near oa.tail dl.head oa.shaft))
  (aliases
    (Point head oa.head)(Point tail dl.tail)))
```

Fig. 6. Description of a dashed line and a dashed open arrow.

3. Multi-domain recognition system

3.1. Recognition of primitive shapes

The base customizable recognition system contains domain independent modules that can recognize, exhibit, and edit all of the primitive shapes in LADDER. These modules are noted by the shaded boxes without their inner white domain modules on the right side of Fig. 1.

When a stroke is drawn (and has not been identified as an editing gesture, described below), low-level recognition is performed on the stroke. The domain independent modules determine if the stroke can be classified as an Ellipse, Line, Curve, Arc, Point, Polyline or some combination using techniques by [21]. In many cases the stroke is ambiguous and has more than one interpretation. When this happens both interpretations are produced and sent off to the higher level recognizer.

We want to ensure that the domain shape recognition system only chooses one interpretation of a single stroke. In order to ensure that only one interpretation is chosen, each shape has an ID, and each shape keeps a list of its subshapes, including its strokes. At any particular time, each subshape is allowed to belong to only one final recognized domain shape. (A final shape is a chosen interpretation as opposed to the myriad of possible interpretations that are created and kept until one is finally chosen.) To give an example, the Stroke in Fig. 7A has two interpretations: a Line and an Arc. The figure specifies each shape's ID followed by the IDs of all of the subshapes. Note that the Line and the Arc share the same Stroke subpart. If a shape has a

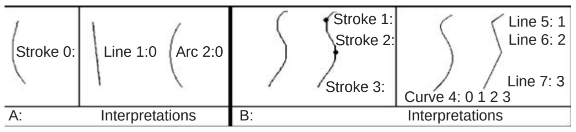


Fig. 7. (A) An ambiguous stroke that can be a Line or an Arc. (B) An ambiguous stroke that can be a Curve or a Polyline.

Polyline interpretation, or some other combination interpretation, the Stroke must be divided into segments. The original full Stroke then has the substrokes added as subparts. These substrokes are then included in any interpretation that uses the full stroke. For example in Fig. 7B the Curve contains all of the substrokes as subshapes. Since the set of final shapes cannot share any subshapes, this prevents the Curve and the Polyline from both being chosen in a final interpretation.

A limitation with this bottom-up recognition method is that if the primitive shape recognizer does not provide the correct interpretation of a stroke, the domain shape recognizer will never be able to correctly recognize a higher level shape using this stroke. In the future, it may be advantageous to add a top-down recognition process that re-examines lower level shapes if an item is missing from the template, such as that done in [22].

3.2. Recognition of domain shapes

Recognition of domain shapes occurs as a series of bottom-up opportunistic data driven triggers where the recognized shapes in the drawing represent the facts about the world. Domain shape recognition is performed by the Jess rule-based system [23]. When a new shape primitive shape is recognized, it is added as a fact into the Jess rule-based system. We created several Jess rules to perform higher level clean-up on the shapes, such as merging lines together.

Each domain shape recognizer is actually a Jess rule defined to recognize the shape (the translation process creating the Jess rule is explained below). The Jess rule-based system searches for all possible combinations of shapes that can satisfy the rule. When choosing between competing shapes we use Okhams razor, choosing the shape that accounts for more of the underlying data. If two choices are equivalent, we choose the shape created first, assuming that a sketcher prefers his shapes to remain constant on the screen. As the number of lower level shapes increases, the rule-based system slows down exponentially, since as each new stroke is drawn the system tries to join it with every other existing stroke to attempt to form higher level shapes. To improve efficiency and to allow the application to continue to react in real time, we have implemented a greedy algorithm that removes subshapes from the rule-based system once a final higher level shape is chosen. Our greedy approach is limited in that in cases of higher level ambiguity, the system may select the wrong higher level shape.

For the same reasons as above, the higher level recognizer also slows down if there are a large number of unrecognized strokes on the screen since the system continues to try to match each of the unrecognized strokes with every new stroke. A quick fix to this is to simply prune unrecognized strokes from the recognition tree after some time. However, this is not ideal as the user may decide to finish a shape after some time, and we would like to be able to recognize this shape. Future work will investigate ways of solving this problem in the future.

When the shapes are more constrained, the recognizer performs faster since the system will hold fewer partial templates to examine later. For instance, if a line direction is constrained, the recognizer only tries to fit the line in one direction, and if a line is horizontal, the recognizer only needs to try the subset of the lines that are horizontal.

We should note that even during these extreme cases when the higher level domain shape recognition begins to slow down and stops reacting in real time, the rest of the system continues to run in real time since the drawing panel, primitive recognition and, domain shape recognition all run in three separate threads, with the drawing panel being given priority to ensure that the pen markings always appear in real time.

3.3. Editing recognition

A stroke may be intended as an editing gesture rather than a drawing gesture. If an editing gesture such as click-and-hold or double-click occurs, the system checks to see (1) if an editing gesture for that trigger is defined for any shape, and (2) if the mouse is over the shape the gesture is defined for. If so then the drawing gesture is short-circuited and the editing gesture takes over (for instance, the shape may then be dragged). Other triggers, such as shape-over, may require that the drawing gesture be completed and recognized before the action, such as deleting the shape underneath, occurs. For example, consider the editing gesture `(trigger (drawOver Cross Shape)) (action (delete Shape) (delete Cross))`; in this example when a Cross is drawn over a Shape, the Shape is deleted (as is the Cross).

3.4. Constraint solver

As mentioned earlier, each shape can be displayed by its original strokes, best-fit primitives, the best-fit primitives with all of the constraints solved (which we will refer to as the shape's ideal strokes), or through Java Swing objects.

To display a shape's ideal strokes, the system uses a shape constraint solver which takes in a shape description and initial locations for all of the subshapes and outputs the shape with all of the constraints satisfied while moving the points as little as possible. Because the positions of the shape's components, its properties, and its constraints are all interrelated, we need to generate and solve algebraic equations demonstrating these relations. We have constructed this shape constraint solver using optimization functions from Mathematica.

To generate a shape we first convert each shape's components, properties (such as, width, height, area), and constraints into a set of algebraic equations. These equations are then solved to find a mathematical solution representing a shape that satisfies the description.

We translate each shape, its components, and its properties using the schema listed below. This produces a set of equations describing the object. For example, one equation produced is `arrow.area = = arrow.width * arrow.height`.

- Minimize: To prevent the shape from shifting too much, we minimize the distance from the value in the initial hand-drawn example and the final solved value for each component.
- Require: To prevent the lines from collapsing to a point, all lines must have a length greater than 10 pixels.
- We define the bounding box of a shape (*minx*, *miny*, *maxx*, *maxy*), so that we can enforce area-related constraints such as *equalArea*, *larger*, *contains*, as follows:
  - Define shape.minx recursively:
    ```
    if (shape is line):
        Require: shape.minx < shape.p1.x
        Require: shape.minx < shape.p2.x
    else for each component:
        Require: shape.minx < shape.component.minx
    ```
  - Define shape.miny, shape.maxx, shape.maxy similarly
  - Minimize: shape.maxx
  - Minimize: shape.maxy
  - Minimize: −1 * shape.minx
  - Minimize: −1 * shape.miny
- Require: shape.width = = shape.maxx - shape.minx
- Require: shape.height = = shape.maxy - shape.miny
- Require: shape.area = = shape.width * shape.height
- Require: shape.center.x = = (shape.minx + shape.maxx)/2

- Require: shape.center.y = = (shape.miny + shape.maxy)/2

Next we translate each constraint into a set of equations on the variables defined above. For example:

horizontal line1 becomes $line1.p1.y == line1.p2.y$
contains shape1 shape2 becomes
$(shape1.minx < shape2.minx)$ &&
$(shape1.miny < shape2.miny)$ &&
$(shape1.maxx > shape2.maxx)$ &&
$(shape1.maxy > shape2.maxy)$
equalLength line1 line2 becomes
$(line1.p1.x - line1.p2.x)^2 + (line1.p1.y - line1.p2.y)^2 ==$
$(line2.p1.x - line2.p2.x)^2$
$+(line2.p1.y - line2.p2.y)^2$
not sameX shape1 shape2 becomes
$(shape1.center.x + 20 < shape2.center.x)$
$||(shape1.center.x > shape.center.x + 20)$
(Because perceptually, small differences in ‘x’ values may not be detected, when constraining the shape to have different ‘x’ values we require the difference to be at least 20 pixels.)

Finally, we use the NMinimize function in Mathematica, which finds constrained global optima, to find a solution that satisfies all of the equations above. We now have new positions for each of the shape’s components which satisfy the constraints in the description. The system will then display the beautified shape.

4. Code generation

Domain shape recognizers, exhibitors, and editors are automatically generated during the translation process shown in the middle of Fig. 1. A shape definition is composed of three parts: how to recognize the shape, how to display the shape once it is recognized, and how to edit the shape once it is recognized. The translation process parses the description and generates code specifying how to recognize shapes and editing triggers as well as how to display the shapes once they are recognized and what action to perform once an editing trigger occurs.

The components and constraints sections of a shape description are automatically translated into a Jess rule defining how to recognize that shape. The Jess rule created for the arrow definition listed in Fig. 1 is shown in Fig. 9. The Jess rule created first searches for the appropriate combination of subshapes, and then tests the constraints between them.[3] We have built our customizable base recognition system in an effort to keep the translation process as simple as possible.

[3]Our current implementation does not yet support soft constraints.

If a shape consists of a variable number of components such as a polyline (as opposed to an arrow which is composed of a fixed (3) number of components), the shape description is translated into two Jess rules, one recognizing the base case (a polyline composed of two lines) and the other recognizing the recursive case (a polyline composed of a line and a polyline).

A shape exhibitor is automatically generated as a Java paint method for the shape, which calls functions in the base recognition system defined to work for any shape. A shape can be displayed by one or more of the following: its original strokes, its best-fit primitives, its best-fit primitives with the constraints solved, a collection of Java Swing shapes, or a bitmap image. To display the ideal strokes (the best-fit primitives with the constraints solved), an IDEAL-PAINT method is automatically generated that defines the constraints to be solved by the shape-based constraint solver.

A shape editor is automatically generated defining which triggers are turned on for the shape or its subshapes. If the trigger is turned on, then the corresponding actions are defined in an automatically generated method. The base recognition system includes methods to identify triggers and perform actions for each shape, and the generated method needs only to turn them on.

5. Evaluation

We have written LADDER domain descriptions for a variety of domains including UML class diagrams, mechanical engineering, finite state machines, flowcharts, and a simplified version of the course of action diagrams (Fig. 8). Using the system presented in this paper, the descriptions have been automatically translated into a sketch interface which recognizes, displays, and allows editing in real time as specified by the domain description. These descriptions include over one hundred shapes, some containing text.[4] Figs. 10–12 show the unrecognized and recognized strokes from a drawing made in an automatically generated mechanical engineering, flowchart, and finite state machine sketch interfaces.

6. Related work

This paper discusses in more detail work from [17,18]. In particular this paper gives more details on the

[4]Text is also a primitive shape. Text can be entered using a keyboard or a handwriting recognizer GUI provided in Microsoft Tablet XP. The text appears at the last typed place.

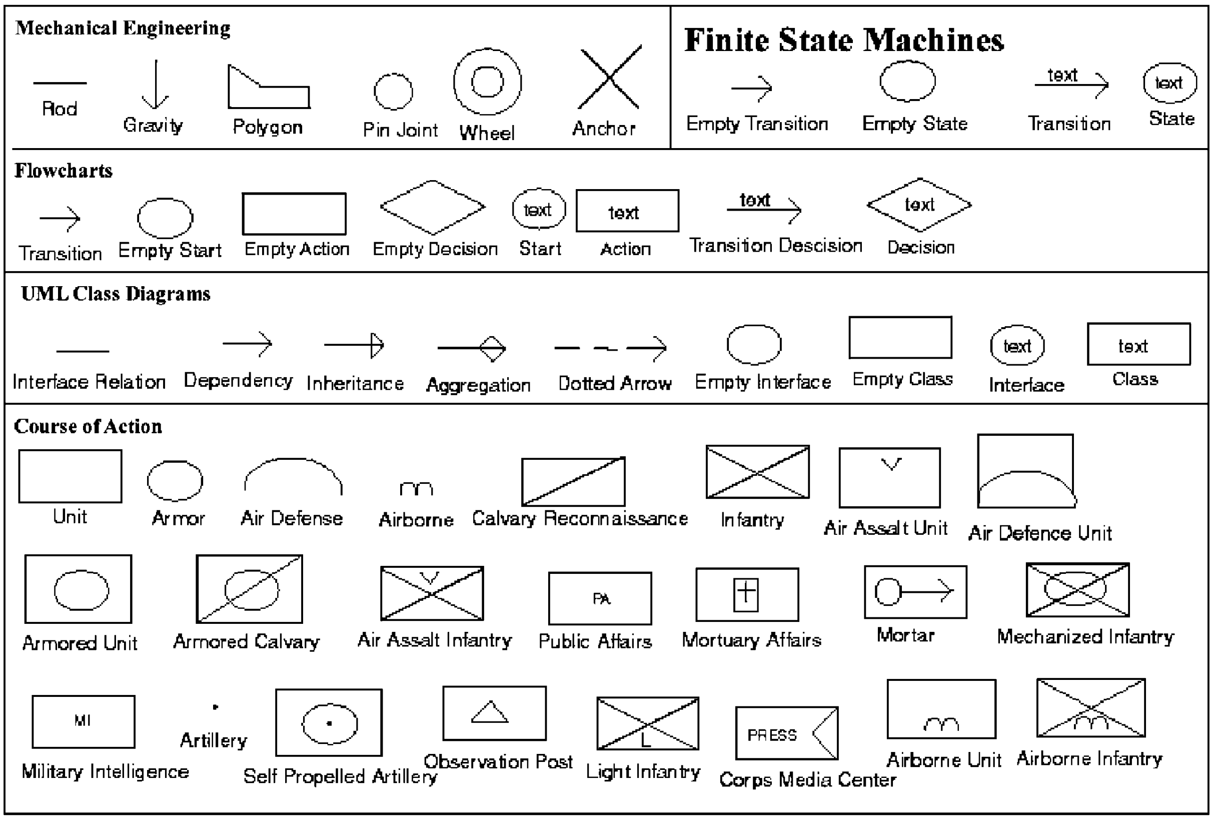


Fig. 8. Variety of shapes and domains described and auto-generated.

primitive and domain shape recognizers, the handling of ambiguous primitive shapes, and the techniques used for beautifying shapes.

6.1. *Visual or sketching languages*

Shape definition languages, such as shape grammars, have been around since the early 1970s [24]. Shape grammars are studied widely within the field of architecture, and many systems are continuing to be built using shape grammars [25]. Shape grammars have, however, been used largely for shape generation rather than recognition, and do not provide for non-graphical information, such as stroke order, that may be helpful in recognition. They also lack ways for specifying shape editing.

More recent shape definition languages have been created for use in diagram parsing [26]. These shape definition languages are not intended for use with an online system and do not provide ways for specifying how to display or edit a shape. Also, since they are not created with sketching in mind they do not provide ways for describing non-graphical information, such as stroke order or direction.

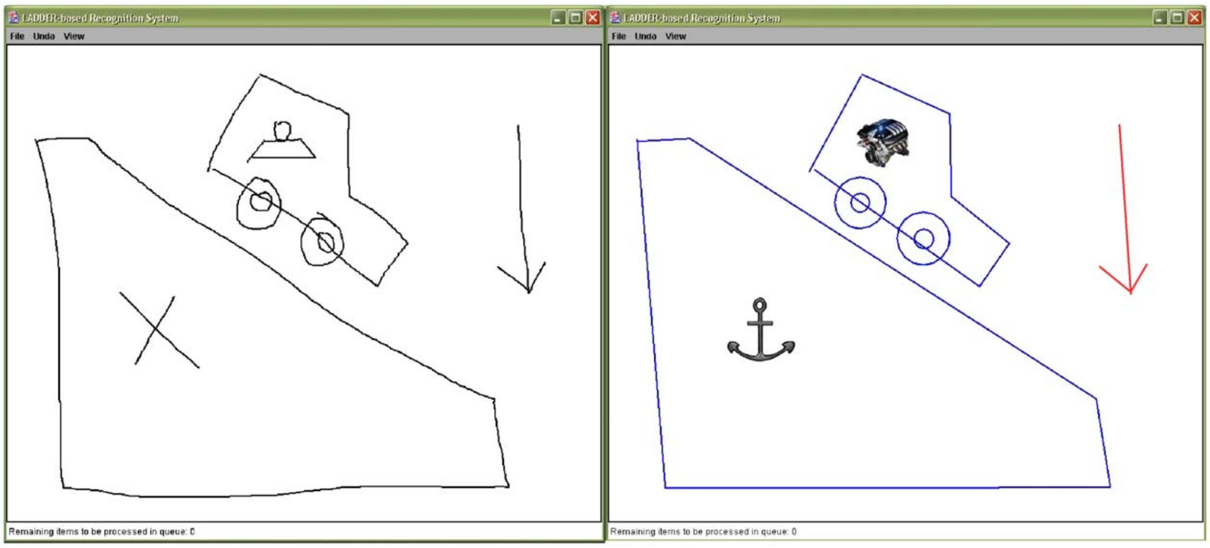
Within the field of sketch recognition, there have been other attempts to create languages for sketch recognition. Bimber describes a simple sketch language using a BNF-grammar [27]. The language describes three-dimensional shapes hierarchically. This language allows a programmer to specify only shape information and lacks the ability to specify other helpful domain information such as stroke order or direction and editing behavior, display, or shape interaction information.

Mahoney uses a language to model and recognize stick figures. The language currently is not hierarchical, making large objects cumbersome to describe [28]. Caetano et al. use fuzzy relational grammars to describe shape [29]. Both Mahoney and Caetano lack the ability to describe editing, display, or shape grouping information.

Shilman has developed a statistical language model for ink parsing with a similar intent of facilitating development of sketch recognizers. The language consists of seven constraints: *distance*, *delta X*, *delta Y*, *angle*, *width ratio*, *height ratio*, and *overlap*, and allows you to specify concrete values using either a range or gaussian [30]. We find it difficult to describe some shapes using this technique as the language requires providing quantitative discrete values about a shape's probable location. We feel it is more intuitive to say (contains shape1 shape2), rather than having to specify two *deltaX* and two *deltaY* constraints using discrete constraints, each of the form deltaX (shape1.WEST

```
(defrule ArrowCheck
  ;; get three lines
  ?f0 $<$- (Subshapes Line ?shaft \$?shaft_list)
  ?f1 $<$- (Subshapes Line ?head1 \$?head1_list)
  ?f2 $<$- (Subshapes Line ?head2 \$?head2_list)
  ;; make sure lines are unique
  (test (uniquefields \$?shaft_list \$?head1_list))
  (test (uniquefields \$?shaft_list \$?head2_list))
  (test (uniquefields \$?head1_list \$?head2_list))
  ;; get accessible components of each line
  (Line ?shaft ?shaft_p1 ?shaft_p2 ?shaft_midpoint ?shaft_length)
  (Line ?head1 ?head1_p1 ?head1_p2 ?head1_midpoint ?head1_length)
  (Line ?head2 ?head2_p1 ?head2_p2 ?head2_midpoint ?head2_length)
  ;; test constraints
  (test (coincident ?head1_p1 ?shaft_p1))
  (test (coincident ?head2_p1 ?shaft_p1))
  (test (equalLength ?head1 ?head2))
  (test (acuteMeet ?head1 ?shaft))
  (test (acuteMeet ?shaft ?head2))
  ;;deleted code: get line with endpoints swapped

=>  ;; FOUND ARROW (ACTION TO BE PERFORMED)
  ;; set aliases
  (bind ?head ?shaft_p1)
  (bind ?tail ?shaft_p2)
  ;; add arrow to sketch recognition system to be displayed properly
  (bind ?nextnum (addshape Arrow ?shaft ?head1 ?head2 ?head ?tail))
  ;; add arrow to Jess fact database
  (assert (Arrow ?nextnum  ?shaft ?head1 ?head2 ?head ?tail))
  (assert (Subshapes Arrow ?nextnum (union\$ \$?shaft_list
          \$?head1_list \$?head2_list)))
  (assert (DomainShape Arrow ?nextnum (time)))
  ;; remove Lines from Jess fact database for efficiency
  (retract ?f0) (assert (CompleteSubshapes Line ?shaft \$?shaft_list))
  (retract ?f1) (assert (CompleteSubshapes Line ?head1 \$?head1_list))
  (retract ?f2) (assert (CompleteSubshapes Line ?head2 \$?head2_list))
  ;;deleted code: retract line with endpoints swapped
)
```

Fig. 9. Automatically generated Jess rule for the arrow definition in the left box of Fig. 1.


LADDER-based Recognition System
File Undo View
LADDER-based Recognition System
File Undo View

Remaining items to be processed in queue: 0

Remaining items to be processed in queue: 0

Fig. 10. Auto-generated mechanical engineering interface.

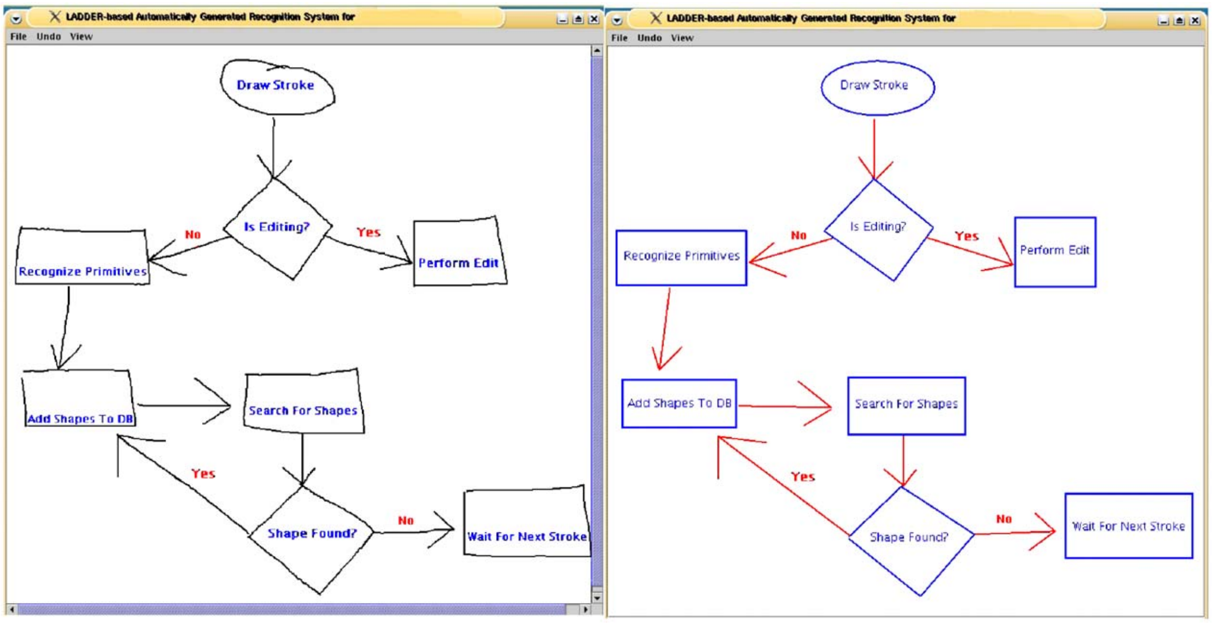


Fig. 11. Auto-generated flowchart interface.

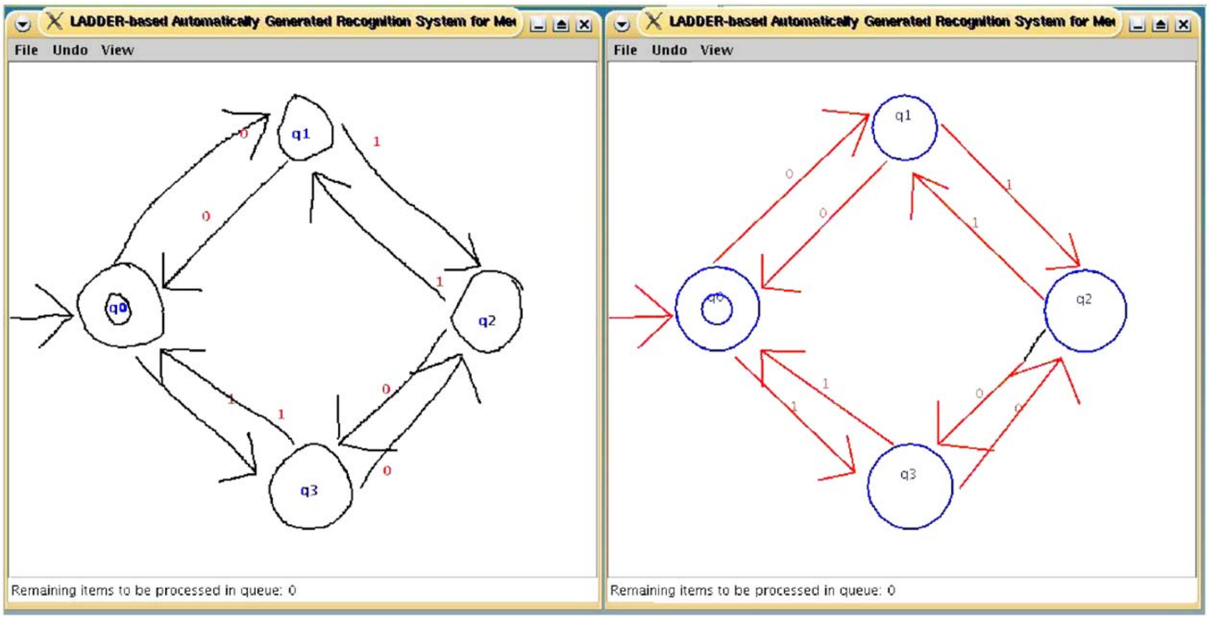

LADDER-based Automatically Generated Recognition System for Me
File Undo View
q0
q1
q2
q3
0
1
Remaining items to be processed in queue: 0

Fig. 12. Auto-generated finite state machine interface.

< shape2.WEST).range(0, 100)) This work also lacks the ability to describe editing and display.

Our recognition system is based on template filling of a shape's structural description. These structural descriptions are often represented in relational graphs. Lee performs recognition using attribute relational graphs [31]. Their attribute language differs from ours in that ours is more topological or geometrical, whereas their language is more quantitative, requiring specific details of the shape's position. Keating also performs recognition by matching a graph representation of a shape; the main difference between their limited graphical language and ours is that their language is statistical and specifies the probable location of each subpart, whereas our language is categorical and describes the ideal location of the shape [32]. Calhoun also uses a semantic network representing the shape in recognition, but as far as we can tell the language is limited, specifying only relative angles and the location of intersections [33].

6.2. Building recognition systems

Quill [20] is a feature-based graffiti-type domain-independent gesture recognition system that allows designers of a gesture recognition system to sketch the gestures to be recognized. The system then provides advice about how well the gestures will be recognized by the computer and how well they will be learned and remembered by people. The Quill framework differs from ours in using recognizers based on features and in focusing on the way the shape is drawn (e.g., the number of strokes, as well as stroke, speed, curvature, order, direction, etc.). In order for their strokes to be recognized, sketchers of this system must sketch a gesture in the same way as the developer who trained the system. Our focus is on removing as many sketching restrictions as possible, to provide a more natural sketching medium. We want users' sketches to be recognized no matter how many strokes they used or in what direction or order they were drawn. Thus, our framework includes a symbolic language for describing the geometry of shapes from which to base recognition.

The Electronic Cocktail Napkin project [34] allows users to define domain shapes by drawing them. A shape is described by the shapes it is built out of and the constraints between them. The Cocktail Napkin's language is able to describe only shape.

Jacob [35] has created a software model and language for describing and programming fine-grained aspects of interaction in a non-WIMP user interface, such as a virtual environment. The language is low level, making it difficult to define new interactions, and, in the domain of sketching, does not provide a significant improvement in comparison to coding the domain dependent recognition system from scratch.

6.3. Translation

The translation process is analogous to work done on compiler compilers, in particular, visual language compiler compilers by Costagliola et al. [36]. A visual language compiler compiler allows a user to specify a grammar for a visual language, and then compiles it into a recognizer which can indicate whether an arrangement of icons is syntactically valid. The main difference between Costagliola's work and ours is that (1) ours handles hand-drawn images and (2) their primitives are the iconic shapes in the domain whereas our primitives are geometric.

7. Future work

While we have attempted to make LADDER as intuitive as possible, shape definitions can be difficult to describe textually, and we are currently integrating Veselova's work to automatically generate shape descriptions from a drawn example [37]. However, even automatically generated shapes will need to be checked and modified. Thus we are in the process of building a graphical debugger which tests if shapes are over- or under-constrained by generating suspected near-miss example shapes [38].

We want to ensure that our framework and language are robust and thus we are continuing to test our system on more domains. For the same reason, we would like to test our syntax on a wide user base.

We are in the process of building an API to allow the designer to connect to a CAD system to build more sophisticated sketch systems. We would also like to allow users building systems in languages other than Java to access recognition events by registering for them.

8. Contributions

We have developed an innovative framework in which developers need to write only a LADDER domain description, and then this description is automatically transformed into a sketch recognition interface for that domain. We have implemented a prototype system and tested our framework by writing descriptions for several domains and then automatically generating a sketch interface for each of these domains. To accomplish our goal, we have created (1) LADDER, the first symbolic domain description language to describe how sketched diagrams in a domain are drawn, displayed, and edited, (2) a customizable base recognition system, which performs the domain independent parts of recognition usable for many domains, and (3) a code generator that translates a domain description into higher level domain specific recognition code to be used by the customizable base recognition system.

Acknowledgements

The authors would like to thank Eric Saund, Mike Oltmans, Jacob Eisenstein, Aaron Adler, Metin Sezgin, Christine Alvarado, Sonya Cates, Mark Finlayson, Raghavan Parthasarthy, Jan Hammond, Rob Miller, James Landay, and Tom Stahovich for their feedback. This work is supported in part by the MIT/Microsoft iCampus initiative and in part by MIT's Project Oxygen.

References

[1] Alvarado C. A natural sketching environment: bringing the computer into early stages of mechanical design. Master's thesis, MIT, 2000.
[2] Landay JA, Myers BA. Interactive sketching for the early stages of user interface design. In: Proceedings of CHI '95: Human Factors in Computing Systems, 1995. p. 43–50.
[3] Stahovich T. Sketchit: a sketch interpretation tool for conceptual mechanism design. Technical Report, MIT AI Laboratory, 1996.
[4] Hammond T, Davis R. Tahuti: a geometrical sketch recognition system for uml class diagrams, AAAI Spring Symposium on Sketch Understanding 2002; 59–68.
[5] Damm CH, Hansen KM, Thomsen M. Tool support for cooperative object-oriented design: gesture based modeling on an electronic whiteboard. In: CHI 2000, CHI; 2000. p. 518–25.
[6] Ideogramic, Ideogramic UML^{TM}, Ideogramic ApS, Denmark, http://www.ideogramic.com/products/, 2001.
[7] Lank E, Thorley JS, Chen SJ-S. An interactive system for recognizing hand drawn UML diagrams. In: Proceedings for CASCON 2000; 2000. p. 7.
[8] Lin J, Newman MW, Hong JI, Landay J. DENIM: finding a tighter fit between tools and practice for web site design. In: CHI Letters: Human Factors in Computing Systems, CHI; 2000. p. 510–7.
[9] Caetano A, Goulart N, Fonseca M, Jorge J. JavaSketchIt: issues in sketching the look of user interfaces, Sketch Understanding. Papers from the 2002 AAAI Spring Symposium.
[10] Lecolinet E. Designing guis by sketch drawing and visual programming. In: Proceedings of the International Conference on Advanced Visual Interfaces (AVI 1998), AVI, New York: ACM Press, 1998. p. 274–6. URLciteseer.nj.nec.com/lecolinet98designing.html
[11] Do EY-L. Vr sketchpad—create instant 3d worlds by sketching on a transparent window. In: de Vries B, van Leeuwen JP, Achten HH, editors. CAAD Futures 2001. 2001. p. 161–72.
[12] Mahoney JV, Fromherz MPJ. Handling ambiguity in constraint-based recognition of stick figure sketches. SPIE Document Recognition and Retrieval IX Conference, San Jose, CA.
[13] Pittman J, Smith I, Cohen P, Oviatt S, Yang T. Quickset: a multimodal interface for military simulations. Proceedings of the Sixth Conference on Computer-Generated Forces and Behavioral Representation, 1996. p. 217–24.
[14] Hse H, Shilman M, Newton AR, Landay J. Sketch-based user interfaces for collaborative object-oriented modeling. Berkley CS260 Class Project (December 1999).
[15] Zue, Glass, Conversational interfaces: advances and challenges. Proceedings of IEEE, 2000. p. 1166–80.
[16] VoiceXML Forum, http://www.voicexml.org/specs/VoiceXML-100.pdf, Voice eXtensible Markup Language, 1st ed. (07 March 2000).
[17] Hammond T, Davis R. LADDER: a language to describe drawing, display, and editing in sketch recognition. Proceedings of the 2003 International Joint Conference on Artificial Intelligence (IJCAI).
[18] Hammond T, Davis R. Automatically transforming symbolic shape descriptions for use in sketch recognition. Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04).
[19] Rubine D. Specifying gestures by example. Computer Graphics 1991;25(4):329–37.
[20] Long AC. Quill: a gesture design tool for pen-based user interfaces, Eecs department, computer science division, UC Berkeley, Berkeley, CA, December 2001.
[21] Sezgin TM. Feature point detection and curve approximation for early processing in sketch recognition. Master's thesis, Massachusetts Institute of Technology, June 2001.
[22] Alvarado C, Davis R. Sketchread: a multi-domain sketch recognition engine. In: Proceedings of UIST '04, 2004. p. 23–32.
[23] Friedman-Hill E, Jess, the java expert system shell. http://herzberg.ca.sandia.gov/jess, 2001.
[24] Stiny G, Gips J. Shape grammars and the generative specification of painting and sculpture. In: Freiman CV, editor. Information processing, vol. 71. Amsterdam: North-Holland, 1972. p. 1460–5.
[25] Gips J. Computer implementation of shape grammars. NSF/MIT Workshop on Shape Computation.
[26] Futrelle RP, Nikolakis N. Efficient analysis of complex diagrams using constraint-based parsing. In: ICDAR-95 (International Conference on Document Analysis and Recognition), Montreal, Canada; 1995. p. 782–90.
[27] Bimber O, Encarnacao LM, Stork A. A multi-layered architecture for sketch-based interaction within virtual environments. Computer and Graphics (Special Issue on Calligraphic Interfaces: Towards a New Generation of Interactive Systems) 2000; 24(6): 851–67.
[28] Mahoney JV, Fromherz MPJ. Three main concerns in sketch recognition and an approach to addressing them. In: Sketch Understanding, Papers from the 2002 AAAI Spring Symposium. Stanford, CA: AAAI Press, 2002. p. 105–12.
[29] Caetano A, Goulart N, Fonseca M, Jorge J. Sketching user interfaces with visual patterns. Proceedings of the First Ibero-American Symposium in Computer Graphics (SIACG02), 2002. p. 271–9.
[30] Shilman M, Pasula H, Russell S, Newton R. Statistical visual language models for ink parsing. In: Sketch Understanding, Papers from the 2002 AAAI Spring Symposium. Stanford, CA: AAAI Press, 2002. p. 126–32.
[31] Lee S-W. Recognizing circuit symbols with attributed graph matching. In: Baird H, Bunke H, Yamamoto K, editors. Structured document image analysis, 1992. p. 340–58.
[32] Keating JP, Mason RL. Some practical aspects of covariance estimation. Pattern Recognition Letters 1985; 3(5):295–350.
[33] Calhoun C, Stahovich TF, Kurtoglu T, Kara LB. Recognizing multi-stroke symbols. Sketch Understanding. Papers from the 2002 AAAI Spring Symposium, 2002. p. 15–23.
[34] Gross MD, Do EY-L. Demonstrating the electronic cocktail napkin: a paper-like interface for early design. 'Common Ground' appeared in ACM Conference on Human Factors in Computing (CHI), 1996. p. 5–6.

[35] Jacob RJK, Deligiannidis L, Morrison S. A software model and specification language for non-WIMP = user interfaces. ACM Transactions on Computer-Human Interaction 1999; 6(1):1–46 URLciteseer.nj.nec.com/jacob99software.html.

[36] Costagliola G, Tortora G, Orefice S, Lucia D. Automatic generation of visual programming environments. IEEE Computer 1995; 56–65.

[37] Veselova O. Perceptually based learning of shape descriptions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

[38] Hammond T, Davis R. Shady: a shape description debugger for use in sketch recognition. AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural.

Scale-space Based Feature Point Detection for Digital Ink

Tevfik Metin Sezgin and Randall Davis
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center 235
Cambridge MA, 02139
{mtsezgin,davis}@csail.mit.edu

Abstract

Feature point detection is generally the first step in model-based approaches to sketch recognition. Feature point detection in free-hand strokes is a hard problem because the input has noise from digitization, from natural hand tremor, and from lack of perfect motor control during drawing. Existing feature point detection methods for free-hand strokes require hand-tuned thresholds for filtering out the false positives. In this paper, we present a threshold-free feature point detection method using ideas from the scale-space theory.

Introduction

There is increasing interest in building systems that can recognize and reason about sketches. Among different approaches to sketch recognition, model-based recognition techniques model objects in terms of their constituent geometric parts and how they fit together (e.g., a rectangle is formed by four lines, all of which intersect at right angles at four distinct corners). In order to be able to match scene elements to geometric model parts, it is necessary to convert the free-hand strokes in the scene into geometric primitives, which results in a more concise and meaningful description of the scene compared to a raw representation only in terms of sampled pen positions. As described in (Sezgin *et al.* November 2001), feature point (i.e., corner) detection is a major part of generating such geometric descriptions.

Issues

The major issue in feature point detection is the noise in the data. We consider noise from two sources: imprecise motor control and digitization. We describe characteristics of each kind of noise with examples to make the distinction clear.

Imperfect motor control

Examples of noise due to imperfect motor control include line segments that were meant to be straight but are not, or corners that look round as opposed to having a precise turning point. This kind of noise gives sketches their “messy” appearance. Easiest way of characterizing this kind of noise is to ask if the noise would still be present if the user drew very carefully perhaps using a ruler. If the answer is negative, then the noise is due to imperfect motor control.

Digitization noise

Digitization noise is the kind of noise that cannot be removed even if one draws very carefully. Although visually less apparent, it hinders feature point detection because digitization corrupts curvature and speed data, which are primary sources of information in feature point detection. Digitization noise can be present in the (x, y) positions and in their timestamps. Source of the spatial digitization noise is the conversion to screen coordinates. For example, in the Acer C110 Tablet PC, the pen positions are digitized into a 1024x768 grid. Spatial digitization can be so poor that the point stream returned by digitization may occasionally have points with repeating (x, y) positions.

In the same platform, timestamps too have digitization noise. Because the concept of having digitization noise in timestamps is less intuitive, we illustrate the point with an example. Consider the stroke in Fig. 1 captured using an Acer c110 Tablet PC. In this platform, we know that the hardware samples points uniformly at a high resolution, digitizing the timestamps once. Then, the operating system digitizes the timestamps again at 100 Hz. Although the timestamps are good when read at the higher resolution directly using Microsoft’s Tablet PC API, they get corrupted during digitization. For the stroke in Fig. 1, Fig. 2 shows the deviation of the digitized timestamps from their predicted ground truth values computed by a least squares linear regression line. The slope of the least squares regression gives us the hardware sampling rate (which is about 133 Hz). The difference in the sampling frequencies causes a skew to accumulate between the real timestamps of the points and those obtained after digitization. The timestamps are occasionally adjusted for the skew by repeating a timestamp, which occurs about every four points with a standard deviation of 0.53. Furthermore, although less frequent, the digitizer consistently returns a point which is 11ms apart from the previous one (as opposed to the more frequent 10ms time difference). This happens roughly once every 92 points ($\mu = 92.25$, $\sigma = 0.95$). If we consider that the time resolution at a sampling rate of 100Hz is 10 ms, the deviations in Fig. 2 which range between $(-8, 6)$ with $\sigma = 3.02ms$ is quite significant. Digitization noise of this nature causes

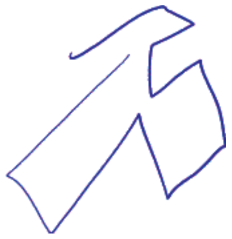


Figure 1: A free-hand stroke captured using Acer c110.

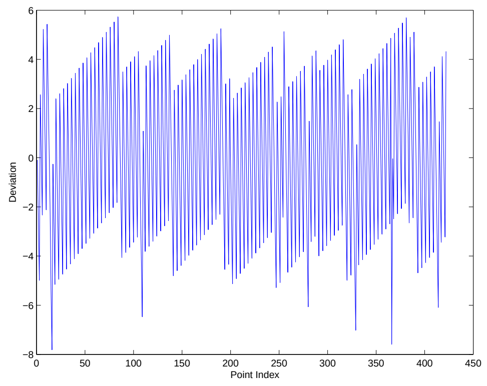


Figure 2: This graph shows the deviation of the timestamps from their linear regression line measured in milliseconds.

speed data computed by taking the time derivative of position to be noisy.

Similar digitization noise behavior is also present in the HP tc1100. Although different in nature, digitization noise is also present for mouse based interfaces, digitizing tablets such as the Wacom PL-400 and the Mimio, a whiteboard capture hardware. The case of Acer c110 and HP tc1100 is more interesting in part because there is a two layer digitization process.

The mainstream approach to dealing with noise is to use filtering criteria based on thresholds preset either by hand or learned from labeled calibration data.

Another approach to dealing with noise is down-sampling points in an effort to achieve a less noisy signal, but such methods throw away potentially useful information when they use fewer than all the points. Furthermore free-hand sketching data is already sparse[1]. Here, we describe a feature point detection system that doesn't depend on preset thresholds or constants, and uses all the points in the stroke.

System Description

Feature Point Detection

Feature point detection is the task of finding corners (vertices) of a stroke. We want to be able to find corners of piecewise linear strokes. For strokes that have curved parts (complex shapes), we want to be able to identify points where curved and straight segments connect. Our technique works for piecewise linear shapes and complex shapes. Requiring the ability to handle complex shapes complicates the problem significantly and rules out well studied piecewise linear approximation algorithms. [2] For strokes with curved portions, we would like to avoid picking points on the curved regions resulting in a piecewise linear approximation of the curved regions.

Our approach takes advantage of the availability of point timestamps during online sketching and combines information from both curvature and speed data, while avoiding a piecewise linear approximation.

Feature points are indicated by maxima of curvature[3] and the minima of pen speed. The strategy of corner detection through local extrema in curvature and speed data would work perfectly in an ideal noiseless setup. In practice it results in many false positives, because local extrema due to the fine scale structure of the noise and those due to the high level structure of the stroke get treated the same way.

One could try setting parameters to filter out these false positives but selecting a priori parameters has the problem of not lending itself to different scenarios where object features and noise may vary. Our experience with the average based feature detection method in (Sezgin *et al.* November 2001) is that its parameters need adjustment for different stroke capture hardware and sometimes for different users. For example, some people tend to make corners more rounded than others. This requires adjusting the parameters of the system for different conditions, a tedious task for the user who must supply data on each platform for calibration purposes, and for the programmer who should find a good set of parameters for each case. Our aim is to remove this overhead by removing the dependence of our algorithms on preset thresholds.

As indicated by our experiments, the extrema due to noise disappear if we look at the data at coarser scales while those due to the real feature points persist across coarser scales. We base our feature point detection technique on our observation that features due to noise and real features exist at different scales. We use the scale-space framework to derive coarser and smoother versions of the data and use the way the number of feature points evolves over different scales to select a scale where the extrema due to noise don't exist. We give details of how we achieve this after a brief introduction to the scale space concept.

Scale-space representation

An inherent property of real-world objects is that they exist as meaningful entities over a limited range of scales. The classical example is a tree branch. A tree branch is meaningful at the centimeter or meter levels, but looses its meaning at very small scales where cells, molecules or atoms make

[1]Data sampled using a traditional digitizing tablet or a Tablet PC may have resolution as low as 4-5 dpi as opposed to scanned drawings with up to 1200-2400 dpi resolution. This is because sometimes users draw so fast that even with high sampling rates such as 100Hz only few points per inch can be sampled.

[2]Vertex localization for piecewise linear shapes is a frequent subject in the extensive literature on graphics recognition. (e.g., (Rosin 1996) compares 21 methods).

[3]Defined as $|\partial\theta/\partial s|$ where θ is the angle between the tangent to the curve at a point and the x axis and s is the cumulative curve length.

sense, or at very large scales where forests and trees make sense.

A technique for dealing with features at multiple scales is to derive representations of the data through multiple scales. The scale-space representation framework introduced by Witkin (Witkin 1983) allows us to derive such multi-scale representations in a mathematically sound way.

The virtues of the scale-space approach are twofold. First, it enables multiple interpretations of the data. These interpretations range from descriptions with a fine degree of detail to descriptions that capture only the overall structure of the stroke. Second, the scale-space approach sets the stage for selecting a scale or a set of scales by looking at how the interpretation of the data changes and features move in the scale-space as the scale is varied.

The basic idea behind the scale-space representation is to generate successively higher level descriptions of a signal by convolving it with a filter. As our filter, we use the Gaussian defined as:

$$g(s,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-s^2/2\sigma^2}$$

where σ is the smoothing parameter that controls the scale. A higher σ means a coarser scale, describing the overall features of the data, while a smaller σ corresponds to finer scales containing the details. The Gaussian filter does not introduce new feature points as the scale increases. This means that as scales get coarser, the number of features (obtained by extrema of the data in question) either remains the same or decreases (i.e., neighboring features are merge causing a decrease in the total number of feature points). The Gaussian kernel is unique in this respect for use in scale-space filtering as discussed in (Yuille & Poggio 1986) and (Babaud *et al.* 1986).

In the continuous case, given a function $f(x)$, the convolution is given by:

$$F(x,\sigma) = f(x) * g(x,\sigma) = \int_{-\infty}^{\infty} f(u) \frac{1}{\sigma\sqrt{2\pi}} e^{(x-u)^2/2\sigma^2}\, du$$

We use the discrete counterpart of the Gaussian function which satisfies the property:

$$\sum_{i=0}^{n} g(i,\sigma) = 1$$

Given a Gaussian kernel, we convolve the data using the following scheme:

$$x_{(k,\sigma)} = \sum_{i=0}^{n} g(i,\sigma) x_{k-\lfloor n/2+1 \rfloor + i}$$

There are several methods for handling boundary conditions when the extent of the kernel goes beyond the end points. In our implementation, we assume that for $k - \lfloor n/2+1 \rfloor + i < 0$ and $k - \lfloor n/2+1 \rfloor + i > n$ the data is padded with zeroes on either side.

Scale selection

The scale-space framework provides a concise representation of the behavior of the data across scales, but doesn't tell us what scale(s) to attend to. In our case, we would like to

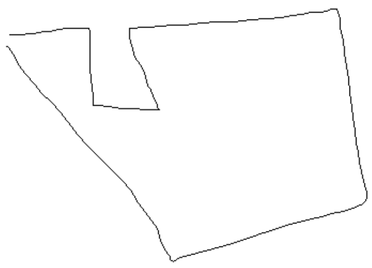


Figure 3: A freehand stroke.

know what scale to attend to for separating noise from real features. The next two sections explain how we used the feature count for scale selection for curvature and speed data.

Application to curvature data

We start by deriving direction and curvature data, then derive a series of functions from the curvature data by smoothing it with Gaussian filters of increasing σ. We build the scale-space by finding the zero crossings of the curvature at various scales.

Scale-space is the (x,σ)-plane where x is the dependent variable of function $f(.)$ (Witkin 1983). We focus on how maxima of curvature move in this 2D plane as σ is varied.

Fig. 3 shows a freehand stroke and Fig. 4 the scale-space map corresponding to the features obtained using curvature data. The vertical axis in the graph is the scale index σ (increasing up); the horizontal axis ranges from 0 to 178 indicating which of the points in the original stroke is calculated to be a feature point. The stroke in question contains 179 points. We detect the feature points by finding the negative zero-crossings of the derivative of absolute value of the curvature. We do this at each scale and plot the corresponding point (σ, i) for each index i in the scale-space plot. An easy way of reading this plot is by drawing a horizontal line at a particular scale index, and then look at the intersection of the line with the scale-space lines. The intersections give us the indices of the points in the original stroke indicated to be feature points at that scale.

As seen in this graph, for small σ (near the bottom of the scale-space graph), many points in the stroke are classified as vertices, because at these scales the curvature data has many local maxima, most of which are caused by the noise in the signal. For increasing σ, the number of feature points decreases gradually.

Our next step is to choose a scale where the false positives due to noise are filtered out and we are left with the real vertices of the data. We want to achieve this without having any particular knowledge about the noise[4] and without having preset scales or constants for handling noise.

The approach we take is to keep track of the number of feature points as a function of σ and find a scale that preserves the tradeoff between choosing a fine scale where the data is too noisy and introduces many false positives, and choosing a coarse scale where true feature points are filtered

[4]The only assumption we make is that the noise is at a different scale than the feature size.

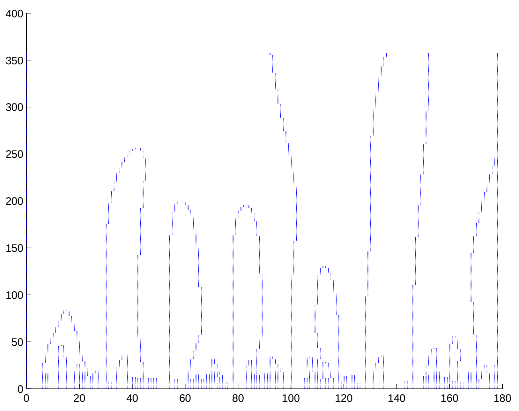


Figure 4: The scale-space for the maxima of the absolute curvature for the stroke in Fig. 3. This plot shows how the maxima move in the scale-space. The x axis is the indices of the feature points, the y axis is the scale index.

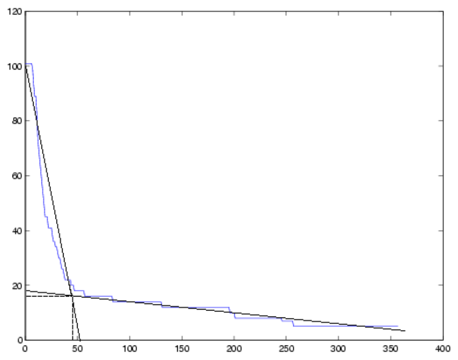


Figure 5: This plot shows the drop in feature point count (y axis) for increasing σ (x axis) and the scale selected by our algorithm for the stroke in Fig. 3.

out. For example, the stroke in Fig. 3, has 101 feature points for $\sigma = 0$. On the coarsest scale, we are left with only 5 feature points, two of which are end points. This means 4 actual feature points are lost by the Gaussian smoothing. Because the noise in the data and the shape described by the true feature points are at different scales, it becomes possible to detect the corresponding ranges of scales by looking at the feature count graph.

Fig. 5 gives the feature count graph for the stroke in Fig. 3. In this figure, the steep drop in the number of feature points that occurs for scales in the range $[0, 40]$ roughly corresponds to scales where the noise disappears, and the region $[85, 357]$ roughly corresponds to the region where the real feature points start disappearing. Fig. 6 shows the scale-space behavior during this drop by combining the scale-space with the feature-count graph. In this graph, the x, y, axis z, respectively correspond to the feature point index [0,200], σ [0,400], and feature count [0,120]. We read the graph as follows: given σ, we find the corresponding location in the y axis. We move up parallel to the z axis until we cross the first scale-space line.[5] The z value at which we

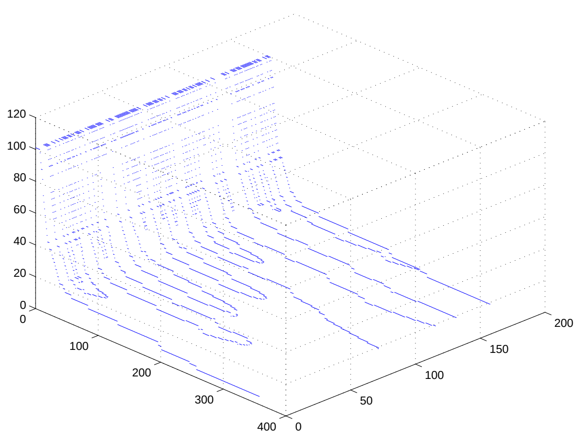


Figure 6: Joint scale-space feature-count graph for the stroke in Fig. 3, simultaneously showing feature point movements in the scale-space and the drop in feature point count for increasing σ.

cross the first scale-space line gives the feature count at scale index σ. Now, we draw a line parallel to the x axis. Movements along this line correspond to different feature indices, and its intersection with the scale-space plot corresponds to indices of feature points present at scale index σ. The steep drop in the feature count is seen in both Fig. 5 and Fig. 6.

Our experiments suggest that this phenomena (i.e., the drop) is present in all hand drawn curves, except in singular cases such as a perfectly horizontal or perfectly vertical line drawn at a constant speed. We model the feature count - scale graph by fitting two lines and derive the scale where the noise is filtered out using their intersection. Specifically, we compute a piecewise linear approximation to the feature count - scale graph with only two lines, one of which tries to approximate the portion of the graph corresponding to the drop in the number of feature points due to noise, and the other that approximates the portion of the graph corresponding to the drop in the number of real feature points. We then find the intersection of these lines and use its x value (i.e., the scale index) as the scale. Thus we avoid extreme scales and choose a scale where most of the noise is filtered out.

Fig. 5 illustrates the scale selection scheme via fitting two lines l_1, l_2 to the feature count - scale graph. The algorithm to get the best fit simply finds the index i that minimizes $OD(l_1, \{P_j\}) + OD(l_2, \{P_k\})$ for $0 \leq j < i$, $i \leq k < n$. $OD(l, \{P_m\})$ is the average orthogonal distance of the points P_m to the line l, P is the array of points in the feature count - scale graph indexed by the scale parameter, and $0 \leq i < n$ where n is the number of points in the stroke. Intuitively, we divide the feature count - scale graph into two regions, fit an ODR line to each region, and compute the orthogonal least squares error for each fit. We search for the division that minimizes the sum of these errors, and select the scale corresponding to the intersection of the lines for which the division is optimal (i.e., has minimum error).

Interestingly enough, we have reduced the problem of stroke approximation via feature detection to fitting lines to

[5]The first scale-space line corresponds to the zeroth point in our stroke, and by default it is a feature point and is plotted in the scale space plot. This remark also applies to the last point in the stroke.

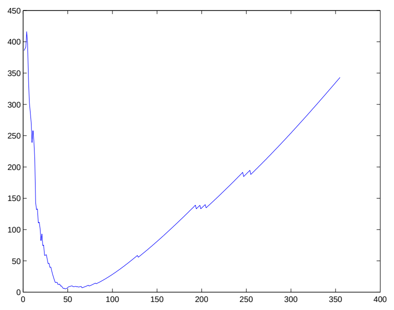


Figure 7: The summed error for the two lines fit to Fig. 5 during scale selection for the stroke in Fig. 3.

the feature count graph, which is similar in nature to the original problem. However, now we know how we want to approximate the data (i.e., with two lines). Therefore even an exhaustive search for i corresponding to the best fit becomes feasible. As shown in Fig. 7 the error as a function of i is U shaped. Thus, if desired, the minima of the summed error can be found using gradient descent methods, by paying special attention to not getting stuck in the local minima. For the stroke in Fig. 3, the scale selected by our algorithm is 47.

While we try to choose a scale where most of the false maxima due to noise are filtered out, feature points at this scale may still contain some false positives. This problem of false extrema in the scale space is also mentioned in (Rattarangsi & Chin 1992), where these points are filtered out by looking at their separation from the line connecting the preceding and following feature points. They filter these points out if the distance is less than one pixel.

The drawback of the filtering technique in (Rattarangsi & Chin 1992) is that the scale-space has to be built differently. Instead of computing the curvature for $\sigma = 0$ and then convolving it with Gaussian filters of larger σ to obtain the curvature data at a particular scale, they treat the stroke as a parametric function of a third variable s, path length along the curve. The x and y components are expressed as parametric functions of s. At each scale, the x and y coordinates are convolved with the appropriate Gaussian filter and the curvature data is computed. It is only after this step that the zero crossings of the derivative of curvature can be computed for detecting feature points. The x and y components should be convolved separately because filtering out false feature points requires computing the distance of each feature point to the line connecting the preceding and following feature points, as explained above. This means the Gaussian convolution, a costly operation, has to be performed twice in this method, compared to a single pass in our algorithm.

Because we convolve the curvature data instead of the x and y coordinates, we can't use the method mentioned above. Instead we use an alternate 2-step method to minimize the number of false positives. First we check whether there are any vertices that can be removed without increasing the least squares error between the generated fit and the original stroke points. The second step in our method takes the generated fit, detects consecutive collinear[6] edges and

[6]Measure of collinearity is determined by the task in hand. We consider lines with $|\Delta\theta| \leq \pi/32$ to be collinear.

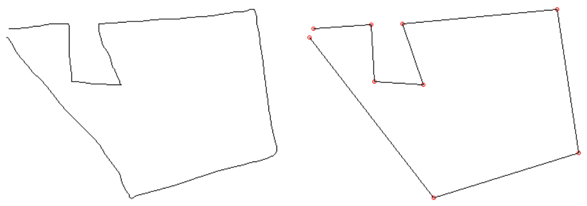


Figure 8: The input stroke (left) and the features detected by looking at the scale-space of the curvature (right).

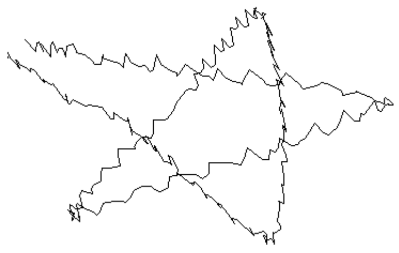


Figure 9: A very noisy stroke.

combines these edges into one by removing the vertex in between. After performing these operations, we get the fit in Fig. 8.

One virtue of the scale-space approach is that works extremely well in the presence of noise. In Fig. 9 we have a very noisy stroke. Figure 10 shows the feature-count and scale-space behaviors respectively. The output of the scale-space based algorithm is in Fig. 11. This output contains only 9 points. For comparison purposes, the output of the average based feature detection algorithm (Sezgin *et al.* November 2001) based on curvature is also given in Fig. 11. This fit contains 69 vertices. (The vertices are not marked for the sake of keeping the figure clean.)

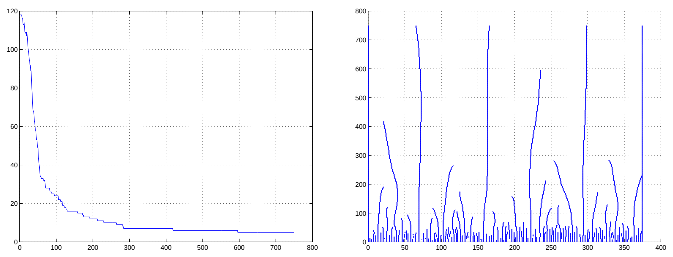
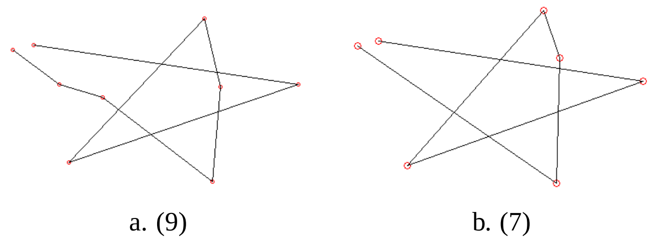


Figure 10: The feature count for increasing σ and the scale-space map for the stroke in Fig. 9. Even with very noisy data, the behavior in the drop is the same as it was for Fig. 3.

Application to speed change

We also applied the scale selection technique described above to speed data. The details of the algorithm for deriving the scale-space and extracting the feature points are similar to that of the curvature data except for obvious differences (e.g., instead of looking for the maxima, we look for the minima).

Fig. 12 has the scale-space, feature-count and joint graphs



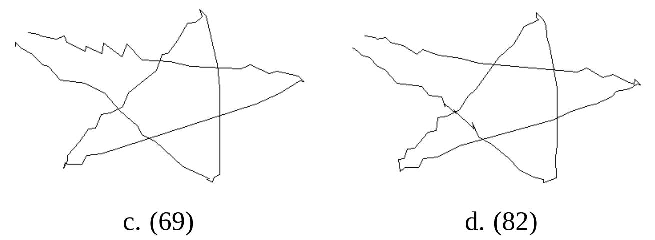


Figure 11: Above, curvature (a) and speed (b) fits generated for the stroke in Fig. 9 with scale-space filtering. Below, fits generated using average based filtering (c,d). For each fit, the number of vertices is given in parenthesis.

for the speed data of the stroke in Fig. 9. As seen in these graphs, the behavior of the speed scale-space is similar to the behavior we observed for the curvature data. We use the same method for scale selection. In this case, the scale index picked by our algorithm was 72. The generated fit is in Fig. 11 along with the fit generated by the average based filtering method using the speed data.

For the speed data, the fit generated by scale-space method has 7 vertices, while the one generated by the average based filtering has 82. In general, the performance of the average based filtering method is not as bad as this example may suggest. For example, for strokes as in Fig. 3, the performance of the two methods are comparable, but for extremely noisy data as in Fig. 9, the scale-space approach pays off when using curvature and speed data.

Because the scale-space approach is computationally more costly[7], using average based filtering is preferable for data that is less noisy. There are also scenarios where only one of curvature or speed data may be noisier. For example, in some platforms, the system-generated timing data for pen motion required to derive speed may not be precise enough, or may be noisy. In this case, if the noise in the pen location is not too noisy, one can use the average based method for generating fits from the curvature data and the scale-space method for deriving the speed fit. This is a choice that the user has to make based on the accuracy of the hardware used to capture the strokes, and the computational limitations.

Combining information sources

Above, we described two feature point detection methods but didn't give a way of combining the results of each method. The hybrid fit generation method described in (Sezgin *et al.* November 2001) can be used to combine the results from two methods to utilize both information sources.

[7]Computational complexity of the average based filtering is linear with the number of points where the scale-space approach requires quadratic time if the scale index is chosen to be a function of the stroke length.

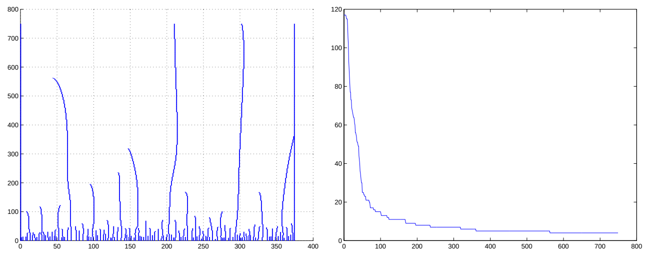
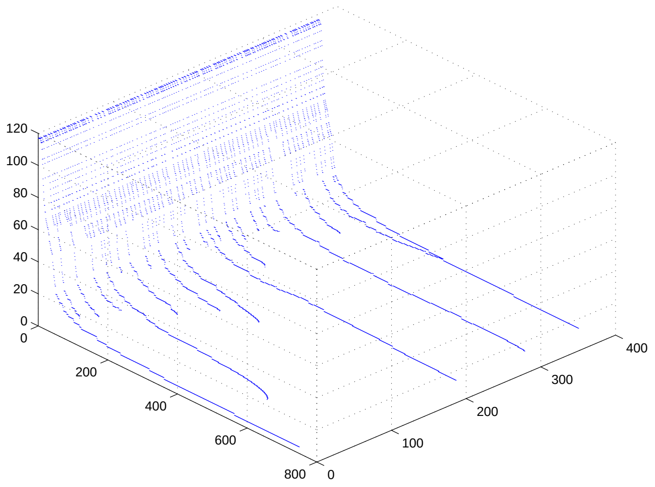



Figure 12: The scale-space, feature-count and joint graphs for the speed data of the stroke in Fig. 9. In this case, the scale selected by our algorithm is 72.

Handling complex strokes

As we mentioned earlier, we would like our method to work for strokes even if they have curved segments. In such cases, we would like to avoid piecewise linear approximations for the curved portions. In our framework, each curved region behaves like a big and smooth corner. Some arbitrary point on the curve (which happens to be the local extreme at the scale selected by our algorithm) gets recognized as a corner. This makes it possible to avoid a piecewise linear approximation of the curved segments. The curve detection method described in (Sezgin *et al.* November 2001) can be applied to detect the curved portions of a stroke.

Evaluation

We measured the performance of our scale-space based feature detection method on strokes from three different setups: Two Tablet PCs (an Acer c110 and an HP tc1100), and a Wacom digitizing LCD tablet PL-400. We chose the average based filtering method as our baseline method and compared our method's performance against it.

We collected data from 10 users. For each platform, the users were asked to draw three instances of 8 shapes. Six of these are shown in Fig. 13, the other two are rectangles rotated 45^o and -45^o. For each user on each platform, we counted the total number of errors (in our case either a

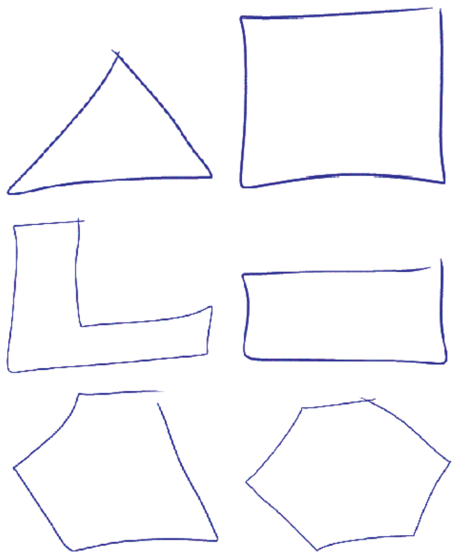


Figure 13: Shapes used in evaluation.

	Acer c110	HP tc1100	Wacom PL-400
T	14	9	11.5

Figure 14: T values for the Wilcoxon matched-pairs signed-ranks test for our feature point detection method and the baseline with data collected using three different setups.

false positive or a false negative) using our feature detection method and the baseline method. For the baseline method we used hand-tuned parameters that gave the best possible fitting results. For each platform, we compared the total number of errors made by each method using the Wilcoxon matched-pairs signed-ranks test (Siegel 1956) with the null hypothesis that the feature detection methods have comparable performance. The T values we obtained for each platform is given in table 14.

Although we were unable to reject the null hypothesis for any platform with a significance of 5% for a two tailed test, in one case we obtained a T value of 9, very close to the value 8 required for rejecting H_0 in favor of our method with level of significance 2.5% for a one tailed test. Overall, our approach compared favorably to the average based filtering method, without the need to hand-tune thresholds for dealing with the noise on each platform.

Related and Future Work

Previous methods on feature point detection either rely on preset constants and thresholds (Sezgin *et al.* November 2001; Calhoun *et al.* 2002), or don't support drawing arbitrary shapes (Schneider 1988; Banks & Cohen 1990).

In the pattern recognition community (Bentsson & Eklundh 1992; Rattarangsi & Chin 1992; Lindeberg 1996) apply some of the ideas from scale-space theory to similar problems. In particular (Bentsson & Eklundh 1992; Rattarangsi & Chin 1992) apply the scale-space idea to detection of corners of planar curves and shape representation, though they focus on shape representation at multiple scales and don't present a scale selection mechanism. The work by (Lindeberg 1996) presents a way of normalizing operator responses (feature strengths) for different σ values such that values across scales become comparable. He presents a scale selection mechanism which finds maxima of the data across scales. Although this method has the merit of making no assumptions about the data, its merit is also its weakness because it doesn't use observations specific to a particular domain as we do for scale selection. It may be an interesting exercise to implement this method and compare its performance to our approach.

Acknowledgements

The authors would like to thank Prof. Fatin Sezgin from Bilkent University for his contributions on the statistical analysis of the evaluation data.

References

Babaud, J.; Witkin, A. P.; Baudin, M.; and Duda, R. O. 1986. Uniqueness of the gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8:26–33.

Banks, M., and Cohen, E. 1990. Realtime spline curves from interactively sketched data. In *SIGGRAPH, Symposium on 3D Graphics*, 99–107.

Bentsson, A., and Eklundh, J. 1992. Shape representation by multiscale contour approximation. *IEEE PAMI 13, p. 85–93, 1992.*

Calhoun, C.; Stahovich, T. F.; Kurtoglu, T.; and Kara, L. B. 2002. Recognizing multi-stroke symbols. *In AAAI 2002Spring Symposium Series, Sketch Understanding.*

Lindeberg, T. 1996. Edge detection and ridge detection with automatic scale selection. *ISRN KTH/NA/P–96/06–SE, 1996.*

Rattarangsi, A., and Chin, R. T. 1992. Scale-based detection of corners of planar curves. *IEEE Transactionsos Pattern Analysis and Machine Intelligence* 14(4):430–339.

Rosin, R. 1996. Techniques for assessing polygonal approximations of curves. *7th British Machine Vision Conf., Edinburgh.*

Schneider, P. 1988. Phoenix: An interactive curve design system based on the automatic fitting of hand-sketched curves. Master's thesis, University of Washington.

Sezgin, T. M.; Stahovich, T.; and Davis, R. November 2001. Sketch based interfaces: Early processing for sketch understanding. *Proceedings of PUI-2001.*

Siegel, S. 1956. Nonparametric statistics: For the behavioral sciences. *McGraw-Hill Book Company*.

Witkin, A. 1983. Scale space filtering. *Proc. Int. Joint Conf. Artificial Intell., held at Karsruhe, West Germany, 1983, published by Morgan-Kaufmann, Palo Alto, California*.

Yuille, A. L., and Poggio, T. A. 1986. Scaling theorems for zero crossings. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8:15–25.

Sketch Based Interfaces: Early Processing for Sketch Understanding

Tevfik Metin Sezgin
MIT AI Laboratory
Massachusetts Institute of Technology
Cambridge MA 02139, USA
mtsezgin@ai.mit.edu

Thomas Stahovich
CMU Deptarment of Mechanical Engineering
Pittsburgh, PA 15213
stahov@andrew.cmu.edu

Randall Davis
MIT AI Laboratory
Massachusetts Institute of Technology
Cambridge MA 02139, USA
davis@ai.mit.edu

ABSTRACT

Freehand sketching is a natural and crucial part of everyday human interaction, yet is almost totally unsupported by current user interfaces. We are working to combine the flexibility and ease of use of paper and pencil with the processing power of a computer, to produce a user interface for design that feels as natural as paper, yet is considerably smarter. One of the most basic steps in accomplishing this is converting the original digitized pen strokes in a sketch into the intended geometric objects. In this paper we describe an implemented system that combines multiple sources of knowledge to provide robust early processing for freehand sketching.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User interfaces; H.5.2 [**User Interfaces**]: Input Devices and strategies; J.6 [**Computer Aided Engineering**]: CAD

General Terms

Design, Human Factors

Keywords

freehand sketching, natural interaction, multiple sources of knowledge

1. INTRODUCTION

Freehand sketching is a familiar, efficient, and natural way of expressing certain kinds of ideas, particularly in the early phases of design. Yet this archetypal behavior is largely unsupported by user interfaces in general and by design software in particular, which has for the most part aimed at providing services in the later phases of design. As a result designers either forgo tool use at the early stage or end up having to sacrifice the utility of freehand sketching for the capabilities provided by the tools. When they move to a computer for detailed design, designers usually leave the sketch behind and the effort put into defining the rough geometry on paper is largely lost.

We are working to provide a system where users can sketch naturally and have the sketches understood. By "understood" we mean that sketches can be used to convey to the system the same sorts of information about structure and behavior as they communicate to a human engineer.

Such a system would allow users to interact with the computer without having to deal with icons, menus and tool selection, and would exploit direct manipulation (e.g., specifying curves by sketching them directly, rather than by specifying end points and control points). We also want users to be able to draw in an unrestricted fashion. It should, for example, be possible to draw a rectangle clockwise or counterclockwise, or with multiple strokes. Even more generally, the system, like people, should respond to how an object looks (e.g., like a rectangle), not how it was drawn. This will, we believe, produce a sketching interface that feels much more natural, unlike Graffiti and other gesture-based systems (e.g., [9], [14]), where pre-specified motions (e.g., an L-shaped stroke or a clockwise rectangular stroke) are required to specify a rectangular shape.

The work reported here is part of our larger effort aimed at providing natural interaction with software, and with design tools in particular. That larger effort seeks to enable user to interact with automated tools in much the same manner as they interact with each other: by informal, messy sketches, verbal descriptions, and gestures. Our overall system uses a blackboard-style architecture [6], combining multiple sources of knowledge to produce a hierarchy of successively more abstract interpretations of a sketch.

Our focus in this paper is on the very first step in the sketch understanding part of that larger undertaking: interpreting the pixels produced by the user's strokes and producing low level geometric descriptions such as lines, ovals, rectangles, arbitrary polylines, curves and their combinations. Conversion from pixels to geometric objects is the first step in interpreting the input sketch. It provides a more compact representation and sets the stage for further, more abstract interpretation (e.g., interpreting a jagged line as a symbol for a spring).

2. THE SKETCH UNDERSTANDING TASK

Sketch understanding overlaps in significant ways with the extensive body of work on document image analysis generally (e.g., [2]) and graphics recognition in particular (e.g., [16]), where the task is to go from a scanned image of, say, an engineering drawing, to a symbolic description of that drawing.

Differences arise because sketching is a realtime, interactive process, and we want to deal with freehand sketches, not the precise diagrams found in engineering drawings. As a result we are not analyzing careful, finished drawings, but are instead attempting to respond in real time to noisy, incomplete sketches. The noise is different as well: noise in a freehand sketch is typically not the small-magnitude randomly distributed variation common in scanned documents. There is also an additional source of very useful information in an interactive sketch: as we show below, the timing of pen motions can be very informative.

Sketch understanding is a difficult task in general as suggested by reports in previous systems (e.g., [9]) of a recognition rate of 63%, even for a sharply restricted domain where the objects to be recognized are limited to rectangles, circles, lines, and squiggly lines (used to indicate text).

Our domain–mechanical engineering design–presents the additional difficulty that there is no fixed set of shapes to be recognized. While there are a number of traditional symbols with somewhat predictable geometries (e.g., symbols for springs, pin joints, etc.), the system must also be able to deal with bodies of arbitrary shape that include both straight lines and curves. As consequence, accurate early processing of the basic geometry–finding corners, fitting both lines and curves–becomes particularly important.

3. SYSTEM DESCRIPTION

Sketches can be created in our system using any of a variety of devices that provide the experience of freehand drawing while capturing pen movement. We have used traditional digitizing tablets, a Wacom tablet that has an LCD-display drawing surface (so the drawing appears under the stylus), and a Mimio whiteboard system. In each case the pen motions appear to the system as mouse movements, with position sampled at rates between 30 and 150 points/sec, depending on the device and software in use.

In the description below, by a single stroke we mean the set of points produced by the drawing implement between the time it contacts the surface (mouse-down) and the time it breaks contact (mouse-up). This single path may be composed of multiple connected straight and curved segments (see, Fig. 1).

Our approach to early processing consists of three phases *approximation*, *beautification*, and *basic recognition*. Approximation fits the most basic geometric primitives–lines and curves–to a given set of pixels. The overall goal is to approximate the stroke with a more compact and abstract description, while both minimizing error and avoiding overfitting. Beautification modifies the output of the approximation layer, primarily to make it visually more appealing without changing its meaning, and secondarily to aid the third phase, basic recognition. Basic recognition produces interpretations of the strokes, as for example, interpreting a sequence of four lines as a rectangle or square. (Subsequent recognition, at the level of mechanical components, such as springs, and pin joints is accomplished by another of our systems [1]).

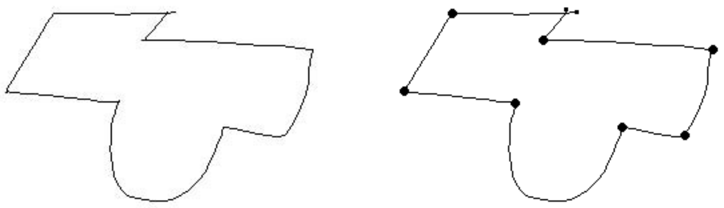


Figure 1: The stroke on the left contains both curves and straight line segments. The points we want to detect in the vertex detection phase are indicated with large dots in the figure on the right. The beginning and the end points of the stroke are indicated with smaller dots.

3.1 Stroke Approximation

Stroke processing consists of detecting vertices at the endpoints of linear segments of the stroke, then detecting and characterizing curved segments of the stroke.

3.1.1 Vertex detection

We use the sketch in Fig. 1 as a motivating example of what should be done in the vertex detection phase. Points marked in Fig. 1 indicate the corners of the stroke, where the local curvature is high.

Note that the vertices are marked only at what we would intuitively call the corners of the stroke (i.e., endpoints of linear segments). There are, by design, no vertices marked on curved portions of the stroke because we want to handle these separately, modeling them with curves (as described below). This is unlike the well studied problem of piecewise linear approximation [13].

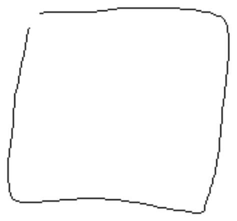


Figure 2: Stroke representing a square.

Our approach takes advantage of the interactive nature of sketching, combining information from both stroke direction and speed data. Consider as an example the square in Fig. 2; Fig. 3 shows the direction, curvature (change in direction with respect to arc length) and speed data for this stroke. We locate vertices by looking for points along the stroke that are minima of speed (the pen slows at corners) or maxima of the absolute value of curvature.[1]

While extrema in curvature and speed typically correspond to vertices, we cannot rely on them blindly because noise in the data introduces many false positives. To deal with this we use average based filtering.

[1]From here on for ease of description we use curvature to mean the absolute value of the curvature data.

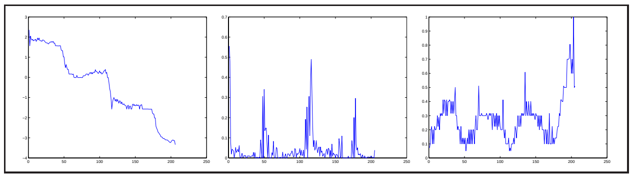


Figure 3: Direction, curvature and speed graphs for the stroke in Fig. 2

Average based filtering
We want to find extrema corresponding to vertices while avoiding those due to noise. To increase our chances at doing this, we look for extrema in those portions of the curvature and speed data that lie beyond a threshold. Intuitively, we are looking for maxima of curvature only where the curvature is already high and minima of speed only where the speed is already low. This will help to avoid selecting false positives of the sort that would occur say, when there is a brief slowdown in an otherwise fast section of a straight stroke.

To avoid the problems posed by choosing a fixed threshold, we set the threshold based on the mean of each data set.[2] We use these thresholds to separate the data into regions where it is above/below the threshold and select the global extrema in each region that lies above the threshold.

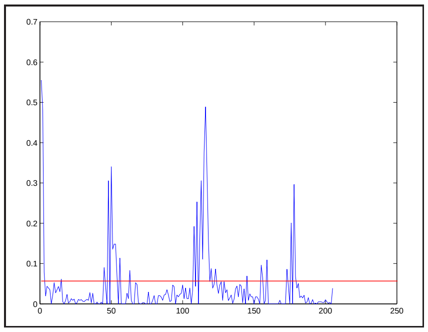


Figure 4: Curvature graph for the square in Fig. 2 with the threshold dividing it into regions.

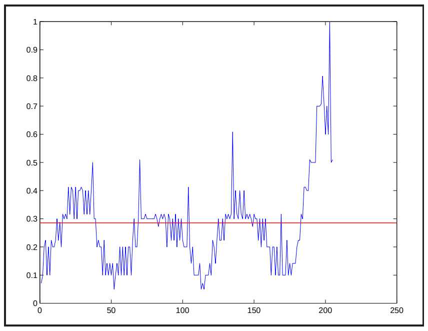


Figure 5: Speed graph for the stroke in Fig. 2 with the threshold dividing it into regions.

[2]The exact threshold has been determined empirically; for curvature data the threshold is the mean, while for the speed the threshold is 90% of the mean.

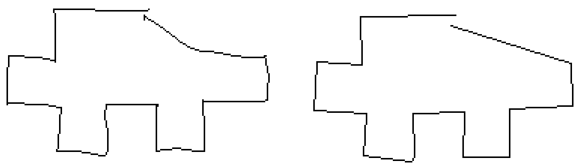


Figure 6: At left the original sketch of a piece of metal; at right the fit generated using only curvature data.

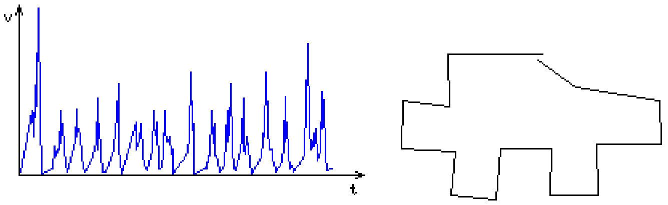


Figure 7: At left the speed graph for the piece; at right the fit based on only speed data.

Application to curvature data

Fig. 4 shows the curvature graph partitioned into regions of high and low curvature. Note that this reduces but doesn't eliminate the problem of false positives introduced by noise in the stroke. We deal with the false positives using the hybrid fit generation scheme described below.[3]

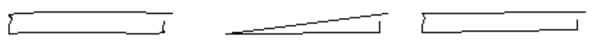
While average based filtering performs better than simply comparing the curvature data against a hard coded threshold, it is still clearly not free of empirical constants. As we explain when considering future work, scale space provides a better approach for dealing with noisy data without having to make a priori assumptions about the scale of relevant features.

Application to speed change
Our experience is that curvature data alone rarely provides sufficient reliability. Noise is one problem, but variety in angle changes is another. Fig. 6 illustrates how curvature fit alone misses a vertex (at the upper right) because the curvature around that point was too small to be detected in the context of the other, larger curvatures. We solve this problem by incorporating speed data into our decision as an independent source of guidance.

Just as we did for the curvature data, we reduce the number of false extrema by average based filtering, then look for speed minima. The intuition here is simply that pen speed drops when going around a corner in the sketch. Fig. 7 shows (at left) the speed data for the sketch in Fig. 6, along with the polygon drawn from the speed-detected vertices (at right).

[3]An alternative approach is to detect consecutive almost-collinear edges (using some empirical threshold for collinearity) and combine them into one edge, removing the vertex in between. Our hybrid fit scheme deals with the problem without the need to decide what value to use for "almost-collinear."

Using speed data alone has its shortcomings as well. Polylines formed from a combination of very short and long line segments can be problematic: the maximum speed reached along the short line segments may not be high enough to indicate the pen has started traversing another edge, with the result that the entire short segment is interpreted as the corner. This problem arises frequently when drawing thin rectangles, common in mechanical devices. Fig. 8 illustrates this phenomena. In this figure, the speed fit misses the upper left corner of the rectangle because the pen failed to gain enough speed between the endpoints of the short vertical segment. The curvature fit, by contrast, detects all corners, along with some other vertices that are artifacts due to hand dynamics during freehand sketching. This illustrates the utility of having both fits available.



(a) Input, 63 points (b) Using speed data, 4 vertices (c) Using curvature data, 7 vertices

Figure 8: Average based filtering using speed data misses a vertex. The curvature fit detects the missed point (along with vertices corresponding to the artifact along the left edge of the rectangle).

We use information from both sources, generating hybrid fits by combining the set of candidate vertices derived from curvature data F_d with the candidate set from speed data F_s, taking into account the system's certainty that each candidate is a real vertex.

Generating hybrid fits

Hybrid fit generation occurs in three stages: computing vertex certainties, generating a set of hybrid fits, and selecting the best fit.

Our certainty metric for a curvature candidate vertex v_i is the scaled magnitude of the curvature in a local neighborhood around the point, computed as $|d_{i-k} - d_{i+k}|/l$. Here l is the curve length between points S_{i-k}, S_{i+k} and k is a small integer defining the neighborhood size around v_i. The certainty metric for a speed fit candidate vertex v_i is a measure of the pen slowdown at the point, $1 - v_i/v_{max}$, where v_{max} is the maximum pen speed in the stroke. The certainty values are normalized to $[0, 1]$.

While both of these metrics are designed to produce values in $[0, 1]$, they have different scales. As the metrics are used only for ordering within each set, they need not be numerically comparable across sets. Candidate vertices are sorted by certainty within each fit.

The initial hybrid fit H_0 is the intersection of F_d and F_s. A succession of additional fits is then generated by appending to H_i the highest scoring curvature and speed candidates not already in H_i.

To do this, on each cycle we create two new fits: $H_i' = H_i + v_s$ (i.e., H_i augmented with the best remaining speed fit candidate) and $H_i'' = H_i + v_d$ (i.e., H_i augmented with the best remaining curvature candidate). We use least squares error as a metric of the goodness of a fit: the error ε_i is computed as the average of the sum of the squares of the distances to the fit from each point in the stroke S:

$$\varepsilon_i = \frac{1}{|S|} \sum_{s \in S} ODSQ(s, H_i)$$

Here $ODSQ$ stands for *orthogonal distance squared*, i.e., the square of the distance from the stroke point to the relevant line segment of the polyline defined by H_i. We compute the error for H_i' and for H_i''; the higher scoring of these two (i.e., the one with smaller least squares error) becomes H_{i+1}, the next fit in the succession. This process continues until all points in the speed and curvature fits have been used. The result is a set of hybrid fits.

In selecting the best of the hybrid fits the problem is as usual trading off more vertices in the fit against lower error. Here our approach is simple: We set an error upper bound and designate as our final fit H_f, the H_i with the fewest vertices that also has an error below the threshold.

3.1.2 Handling curves

The approach described thus far yields a good approximation to strokes that consists solely of line segments, but as noted our input may include curves as well, hence we require a means of detecting and approximating them.

The polyline approximation H_f generated in the process described above provides a natural foundation for detecting areas of curvature: we compare the Euclidean distance l_1 between each pair of consecutive vertices in H_f to the accumulated arc length l_2 between those vertices in the input S. The ratio l_2/l_1 is very close to 1 in the linear regions of S, and significantly higher than 1 in curved regions.

We approximate curved regions with Bézier curves, defined by two end points and two control points. Let $u = S_i$, $v = S_j$, $i < j$ be the end points of the part of S to be approximated with a curve. We compute the control points as:

$$c_1 = k\hat{t}_1 + v$$

$$c_2 = k\hat{t}_2 + u$$

$$k = \frac{1}{3} \sum_{i \le k < j} |S_k - S_{k+1}|$$

where $\hat{t}_1$ and $\hat{t}_2$ are the unit length tangent vectors pointing inwards at the curve segment to be approximated. The 1/3 factor in k controls how much we scale $\hat{t_1}$ and $\hat{t_2}$ in order to reach the control points; the summation is simply the length of the chord between S_i and S_j.[4]

As in fitting polylines, we want to use least squares to evaluate the goodness of a fit, but computing orthogonal distances from each S_i in the input stroke to the Bézier curve segments would require solving a fifth degree polynomial. (Bézier curves are described by third degree polynomials, hence computing the minimum distance from an arbitrary point to the curve involves minimizing a sixth degree polynomial, equivalent to solving a fifth degree polynomial.) A numerical solution is both computationally expensive and heavily dependent on the goodness of the initial guesses for

[4]The 1/3 constant was determined empirically, but works very well for freehand sketches. As we discovered subsequently, the same constant was independently chosen in [15].

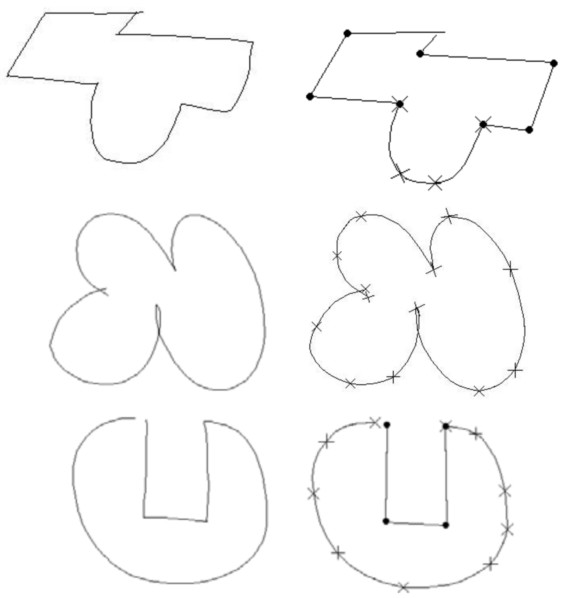


Figure 9: Examples of arbitrary stroke approximation. Boundaries of Bézier curves are indicated with crosses, detected vertices are indicated with dots.

roots [12], hence we resort to an approximation. We discretize the Bézier curve using a piecewise linear curve and compute the error for that curve. This error computation is $O(n)$ because we impose a finite upper bound on the number of segments used in the piecewise approximation.

If the error for the Bézier approximation is higher than our maximum error tolerance, the curve is recursively subdivided in the middle, where middle is defined as the data point in the original stroke whose index is midway between the indices of the two endpoints of the original Bézier curve. New control points are computed for each half of the curve, and the process continues until the desired precision is achieved.

Examples of the capability of our approach is shown in Fig. 9, a hastily-sketched mixture of lines and curves. Note that all of the curved segments have been modeled curves, rather than the piecewise linear approximations that have been widely used previously.

3.2 Beautification

Beautification refers to the (currently minor) adjustments made to the approximation layer's output, primarily to make it look as intended. We adjust the slopes of the line segments in order to ensure the lines that were apparently meant to have the same slope end up being parallel. This is accomplished by looking for clusters of slopes in the final fit produced by the approximation phase, using a simple sliding-window histogram. Each line in a detected cluster is then rotated around its midpoint to make its slope be the weighted average of the slopes in that cluster. The (new) endpoints of these line segments are determined by the intersections of each consecutive pair of lines. This process (like any neatening of the drawing) may result in vertices being moved; we chose to rotate the edges about their midpoints because this produces vertex locations that are close to those detected, have small least square errors when mea-

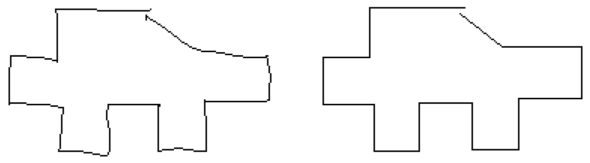


Figure 10: At left the original sketch of a piece of metal revisited, and the final beautified output at right.

sured against the original sketch, and look right to the user. Fig. 10 shows the original stroke for the metal piece we had before, and the output of the beautifier. Some examples of beautification are also present in Fig. 13.

3.3 Basic Object Recognition

The final step in our processing is recognition of the most basic objects that can be built from the line segments and curve segments produced thus far, i.e., simple geometric objects (ovals, circles, rectangles, squares).

Recognition of these objects is done with hand-tailored templates that examine various simple properties. A rectangle, for example, is recognized as a polyline with 4 segments all of whose vertices are within a specified distance of the center of the figure's bounding box; a stroke will be recognized as an oval if it has a small least squares error when compared to an oval whose axes are given by the bounding box of the stroke.

3.4 Evaluation

We have conducted a user study to measure the degree to which the system is perceived as easy to use, natural and efficient. Study participants were asked to create a set of shapes using our system and Xfig, a Unix tool for creating diagrams. Xfig is a useful point of comparison because it is representative of the kinds of tools that are available for drawing diagrams using explicit indication of shape (i.e., the user indicates explicitly which parts of the sketch are supposed to be straight lines, which curves, etc.) As in other such tools, XFig has a menu and toolbar interface; the user selects a tool (e.g., for drawing polygons), then creates the shapes piece by piece.

Thirteen subjects participated in our study, including computer science graduate students, computer programmers and an architecture student. Subjects were given sufficient time to get familiar with each system and then asked to draw a set of 10 shapes (examples given in Fig 11). All of the subjects reported our system being easier to use, efficient and more natural feeling. The subjects were also asked which system they would prefer when drawing these sort of informal shapes on a computer. All but one subject preferred our system; the sole dissenter preferred a tablet surface that had the texture and feel of paper.

Overall users praised our system because it let them draw shapes containing curves and lines directly and without having to switch back and forth between tools. We have also observed that with our system, users found it much easier to draw shapes corresponding to the gestures they routinely draw freehand, such as a star.

While the central point of this comparison was to deter-

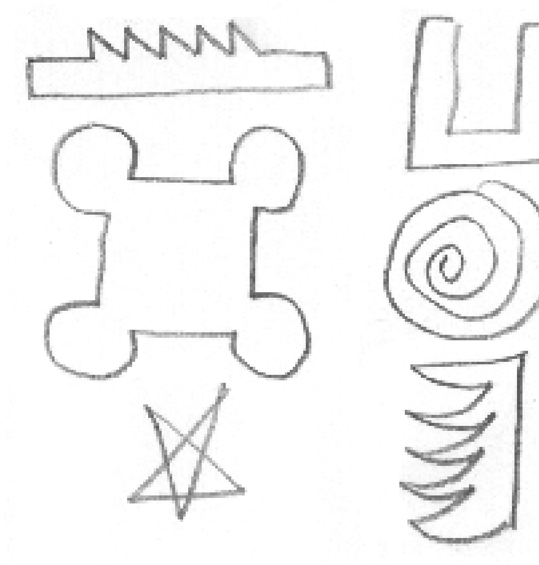


Figure 11: Examples of the shapes used in the user study.

mine how natural it felt to use each system, we also evaluated our system's ability to produce a correct interpretation of each shape (i.e., interpret strokes appropriately as lines or curves). Overall the system's identification of the vertices and approximation of the shapes with lines and curves was correct 96% of the time on the ten figures.

In addition to the user studies we have conducted, we wrote a higher level recognizer for evaluation purposes. The higher level recognizer takes the geometric descriptions generated by the basic object recognition module of our system and combines them into domain specific objects.

Fig. 13 shows the original input and the program's analysis for a variety of simple but realistic mechanical devices drawn as freehand sketches. The last two of them are different sketches for a part of the direction reversing mechanism for a tape player. Recognized domain specific components include gears (indicated by a circle with a cross), springs (indicated by wavy lines), and the standard fixed-frame symbol (a collection of short parallel lines). Components that are recognized are replaced with standard icons scaled to fit the sketch.

An informal comparison of the raw sketch and the system's approximations shows whether the system has selected vertices where they were drawn, fit lines and curves accurately, and successfully recognized basic geometric objects. While informal, this is an appropriate evaluation because the program's goal is to produce an analysis of the strokes that "looks like" what was sketched.

We have also begun to deal with overtracing, one of the (many) things that distinguishes freehand sketches from careful diagrams. Fig. 12 illustrates one example of the limited ability we have thus far embodied in the program.

4. RELATED WORK

In general, systems supporting freehand sketching lack one or more of the properties that we believe a sketching system should have:

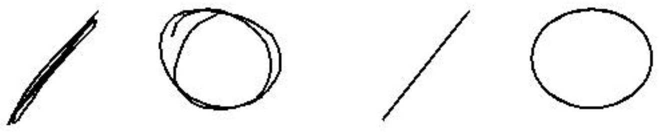


Figure 12: An overtraced oval and a line along with and the system's output.

- It should be possible to draw arbitrary shapes with a single stroke, (i.e., without requiring the user to draw objects in pieces).
- The system should do automatic feature point detection. The user should not have to specify vertex positions by hand.
- The system should not have sketching modes for drawing different geometric object classes (i.e., modes for drawing circles, polylines, curves etc.).
- The sketching system should feel natural to the user.

The Phoenix sketching system [15] had some of the same motivation as our work, but a more limited focus on interactive curve specification. While the system provided some support for vertex detection, its focus on curves led it to use Gaussian filters to smooth the data. While effective for curves, Gaussians tend to treat vertices as noise to be reduced, obscuring vertex location. As a result the user was often required to specify the vertices manually.

Work in [5] describes a system for sketching with constraints that supports geometric recognition for simple strokes (as well as a constraint maintenance system and extrusion for generating solid geometries). The set of primitives is more limited than ours: each stroke is interpreted as a line, arc or as a Bézier curve. More complex shapes can be formed by combinations of these primitives, but only if the user lifts the pen at the end of each primitive stroke, reducing the feeling of natural sketching.

The work in [3] describes a system for generating realtime spline curves from interactively sketched data. They focus on using knot removal techniques to approximate strokes known to be composed only of curves, and do not handle single strokes that contain both lines and curves. They do not support corner detection, instead requiring the user to specify corners and discontinuities by lifting the mouse button, or equivalently by lifting the pen. We believe our approach of automatically detecting the feature points provides a more natural and convenient sketching interface.

Zeleznik [7] describes a mode-based stroke approximation system that uses simple rules for detecting the drawing mode. The user has to draw objects in pieces, reducing the sense of natural sketching. Switching modes is done by pressing modifier buttons in the pen or in the keyboard. In this system, a click of the mouse followed by immediate dragging signals that the user is drawing a line. A click followed by a pause and then dragging of the mouse tells the system to enter the freehand curve mode. Our system allows drawing arbitrary shapes without any restriction on how the user draws them. There is enough information provided by the freehand drawing to differentiate geometric shapes such as curves, polylines, circles and lines from one another, so we believe requiring the user to draw things in a particular fashion is unnecessary and reduces the natural feeling of sketching. Our goal is to make computers understand what the user is doing rather than requiring the user to sketch in a way that the computer can understand.

Among the large body of work on beautification, Igarashi et al. [8] describes a system combining beautification with constraint satisfaction, focusing on exploiting features such as parallelism, perpendicularity, congruence and symmetry. The system infers geometric constraints by comparing the input stroke with previous ones. Because sketches are inherently ambiguous, their system generates multiple interpretations corresponding to different ways of beautifying the input, and the most plausible interpretation is chosen among these interpretations. The system is interactive, requiring the user to do the selection, and doesn't support curves. It is, nevertheless, more effective then ours at beautification, but beautification is not the main focus of our work and is present for the purposes of completeness.

The works in [15] and [3] describe methods for generating very accurate approximations to strokes known to be curves with precision several orders of magnitude below the pixel resolution. The Bézier approximations we generate are less precise but are sufficient for approximating free-hand curves. We believe techniques in [15] and [3] are excessively precise for free-hand curves, and the real challenge is detecting curved regions in a stroke rather than approximating those regions down to the numerical machine precision.

5. FUTURE WORK

We are working to link this early processing to other work in our group that has focused on recognition [1] of higher level mechanical objects. This will provide the opportunity to add model-based processing of the stroke, in which early operations like vertex localization may be usefully guided by knowledge of the current best recognition hypothesis.

In addition, incorporating ideas from scale space theory looks like a promising way of detecting different scales inherent in the data and avoiding *a priori* judgments about the size of relevant features. In the pattern recognition community [4], [11] and [10] apply some of the ideas from scale space theory to similar problems. We are currently working on ways of applying these techniques to speed and curvature data. We believe this may allow us to deal more effectively with sketches that contain relevant details at a variety of scales. There is no guaranteed way of deciding which scales are important at the geometric level, so using constraints and/or information provided by the domain of application may help in scale selection.

Humans naturally seem to slow down when they draw things carefully as opposed to casually, so another interesting research direction would be to explore the degree to which one can use the time it takes to draw a stroke as an indication of how careful and precise the user meant to be.

6. CONCLUSION

We have built a system capable of using multiple sources of information to produce good approximations of freehand sketches. Users can sketch on an input device as if drawing on paper and have the computer detect the low level geometry, enabling a more natural interaction with the computer, as a first step toward more natural user interfaces generally, and toward earlier use of automated tools in the design cycle in particular.

7. REFERENCES

[1] C. Alvarado. A natural sketching environment: Bringing the computer into early stages of mechanical design. Master's thesis, Massachusetts Institute of Technology, 2000.

[2] H. S. Baird, H. Bunke, and K. Yamamoto. Structured document image analysis. Springer-Verlag, Heidelberg, 1992.

[3] M. Banks and E. Cohen. Realtime spline curves from interactively sketched data. In *SIGGRAPH, Symposium on 3D Graphics*, pages 99–107, 1990.

[4] A. Bentsson and J. Eklundh. Shape representation by multiscale contour approximation. *IEEE PAMI 13, p. 85–93, 1992.*, 1992.

[5] L. Eggli. Sketching with constraints. Master's thesis, University of Utah, 1994.

[6] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12:213–253, 1980. Reprinted in: *Readings in Artificial Intelligence*, Bonnie L. Webber and Nils J. Nilssen (eds.)(1981), pp 349-389. Morgan Kaufman Pub. Inc., Los Altos, CA.

[7] A. Forsberg, K. Herndon, and R. Zeleznik. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH'96*, pages 163–170, 1996.

[8] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Interactive beautification: A technique for rapid geometric design. In *UIST '97*, pages 105–114, 1997.

[9] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer, vol. 34, no. 3, March 2001, pp. 56-64.*

[10] T. Lindeberg. Edge detection and ridge detection with automatic scale selection. *ISRN KTH/NA/P–96/06–SE, 1996.*, 1996.

[11] A. Rattarangsi and R. T. Chin. Scale-based detection of corners of planar curves. *IEEE Transactionsos Pattern Analysis and Machine Intelligence*, 14(4):430–339, Apr. 1992.

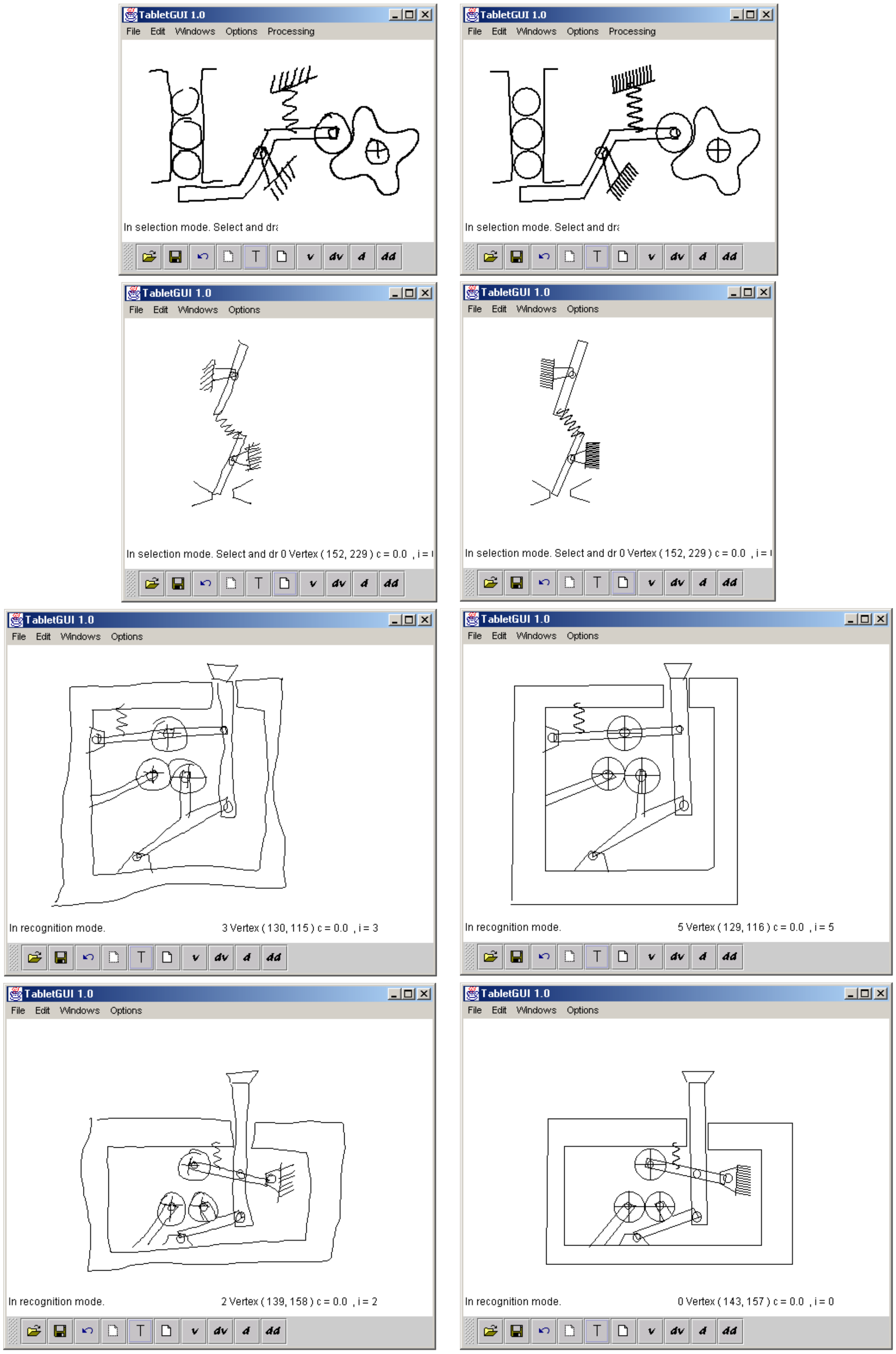
[12] N. Redding. Implicit polynomials, orthogonal distance regression, and closest point on a curve. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 191–199, 2000.

[13] R. Rosin. Techniques for assessing polygonal approximations of curves. *7th British Machine Vision Conf., Edinburgh*, 1996.

[14] D. Rubine. Specifying gestures by example. *Computer Graphics*, 25(4):329–337, 1991.

[15] P. Schneider. Phoenix: An interactive curve design system based on the automatic fitting of hand-sketched curves. Master's thesis, University of Washington, 1988.

[16] K. Tombre. Analysis of engineering drawings. In *GREC 2nd international workshop*, pages 257–264, 1997.



In selection mode. Select and dr

In selection mode. Select and dr

In selection mode. Select and dr 0 Vertex (152, 229) c = 0.0 , i =

In selection mode. Select and dr 0 Vertex (152, 229) c = 0.0 , i =

In recognition mode. 3 Vertex (130, 115) c = 0.0 , i = 3

In recognition mode. 5 Vertex (129, 116) c = 0.0 , i = 5

In recognition mode. 2 Vertex (139, 158) c = 0.0 , i = 2

In recognition mode. 0 Vertex (143, 157) c = 0.0 , i = 0

Figure 13: Performance examples: The first two pair are sketches of a marble dispenser mechanism and a toggle switch. The last two are sketches of the direction reversing mechanism in a tape player.

Designing Freeform Surfaces by Sketching

A Sketching Interface for Modeling the Internal Structures of 3D Shapes

Shigeru Owada[1], Frank Nielsen[2], Kazuo Nakazawa[3], and Takeo Igarashi[1]

[1] Department of Computer Science, The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku ,Tokyo 113-8654, JAPAN,
{ohwada|takeo}@is.s.u-tokyo.ac.jp
[2] Sony Computer Science Laboratories,Inc.,
Takanawa Muse Bldg., 3-14-13, Higashigotanda,
Shinagawa-ku, Tokyo 141-0022, JAPAN,
nielsen@csl.sony.co.jp
[3] National Cardiovascular Center,
5-7-1 Fujishiro-dai, Suita, Osaka 565-8565, JAPAN,
nakazawa@ri.ncvc.go.jp

Abstract. This paper presents a sketch-based modeling system for creating objects that have internal structures. The user input consists of hand-drawn sketches and the system automatically generates a volumetric model. The volumetric representation solves any self-intersection problems and enables the creation of models with a variety of topological structures, such as a torus or a hollow sphere. To specify internal structures, our system allows the user to cut the model temporarily and apply modeling operations to the exposed face. In addition, the user can draw multiple contours in the Create or Sweep stages. Our system also allows automatic rotation of the model so that the user does not need to perform frequent manual rotations. Our system is much simpler to implement than a surface-oriented system because no complicated mesh editing code is required. We observed that novice users could quickly create a variety of objects using our system.

1 Introduction

Geometric modeling has been a major research area in computer graphics. While there has been much progress in rendering 3D models, creating 3D objects is still a challenging task. Recently, attention has focused on sketch-based modeling systems with which the user can quickly create 3D models using simple freehand strokes rather than by specifying precise parameters for geometric objects, such as spline curves, NURBS patches, and so forth [15, 6]. However, these systems are primarily designed for specifying the external appearance of 3D shapes, and it is still difficult to design freeform models with internal structures, such as internal organs. Specifically, the existing sketch-based freeform modeling system [6] can handle 3D models only with spherical topology. This paper introduces a modeling system that can design 3D models with complex internal structures, while maintaining the ease of use of existing sketch-based freeform modelers. We used a volumetric data structure to handle the dynamically changing topology efficiently. The volumetric model is converted to a polygonal surface and is displayed using a non-photorealistic rendering technique to facilitate creative exploration. Unlike previous systems, our system allows the user to draw nested contours to design models with internal structures. In addition, the user can cut the model temporarily and apply modeling operations to the exposed face to design internal structures. The underlying volumetric representation simplifies the implementation of such functions. Moreover, our system actively assists the user by automatically rotating the model when necessary.

The heart of our technique is automatic "guessing" of 3D geometry from 2D gestural input, and it is done by making certain assumptions about the target geometry. To be specific, the system assumes that the target geometry has a rotund, smooth (low curvature) surface [6] other than the places where the user explicitly defined the geometry by the input strokes. In other words, the user specifies the information about important features (silhouette, intersection, and sweep path) and the system supplies missing information based on the above assumption.

Our system is designed to facilitate the communication of complicated geometric information, such as surgical plans. Like other sketch-based modeling systems, however, our system is not suitable for creating the final output of any serious production, because of its lack of accuracy.

2 Previous Work

Three-dimensional shape modeling systems that use a volumetric data structure directly are relatively new [14, 4] as compared with other popular modeling primitives, such as polygons, NURBS, and subdivision surfaces. Recently, a scripting language [2], octree [11], subdivision volume [10], and level set [1] have been used as volumetric modeling methodologies. Some systems use 3D haptic input devices [4, 3, 5, 10].

Sketch-based modeling using standard mouse operations became popular in the past decade. Instead of creating precise, large-scale objects, a sketching interface provides an easy way to create a rough model to convey the user's idea quickly. One of the earliest sketching systems was Viking [12], which was designed in the context of prototypic CAD models. Later works include SKETCH [15] and Teddy [6]. The SKETCH system is intended to sketch a scene consisting of simple primitives, such as boxes and cones, while the Teddy system is designed to create rotund objects with spherical topology. Although improvements to the original Teddy system have recently been proposed [7], extending the topological variety of creatable models is still an unsolved problem.

Although the user interface of our system is based on the Teddy system, our system is free from topological limitations, provides multiple interfaces for specifying internal structures, and actively assists the user by automatically rotating a model when necessary.

3 User Interface

The entire editing operation is performed in a single window. Modeling operations are specified by freeform strokes drawn on the screen and by pressing buttons on a menu bar. The freeform strokes provide necessary geometric information and the buttons apply specific modeling operations using the strokes as input. The drawing of strokes is assigned to the left mouse button and rotating the model is assigned to the right mouse button. The current implementation uses four buttons, as shown in Fig. 1. The leftmost button is used to initialize the current scene; the second one is to create items; the third is for the extrusion/sweep function; and the last is for undo.

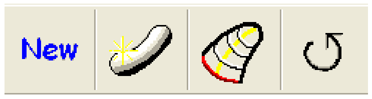


Fig. 1. Buttons in our system

3.1 Create

Objects are created by drawing one or more contours on the canvas and pressing the "Create" button. This operation inflates the intermediate region between the strokes leaving holes (Fig. 2).

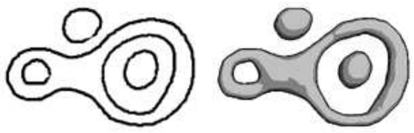


Fig. 2. Nested contours are allowed in the Create operation.

3.2 Extrusion

Extrusion is an operation that generates a protuberance or a dent on a model. The user draws a single closed stroke on the object's surface specifying the contour (Fig. 3 (b)) and presses the "Extrusion/sweep" button. After rotating the model (Fig. 3 (c)), the user draws a second stroke specifying the silhouette of the extruded area (Fig. 3 (d, f)). The user should place each end of the silhouette stroke close to each end of the projected surface contour (otherwise the second stroke is interpreted as a sweep path; see Section 3.4.) A protuberance is created if the second stroke is drawn on the outside of the object (Fig. 3 (d,e)). The user can also create a hole by drawing a stroke into the object (Fig. 3 (f,g)). Volumetric representation automatically prevents self-intersection problems, where specialized care must be taken when using a polygonal representation. A hidden silhouette is rendered as broken lines.

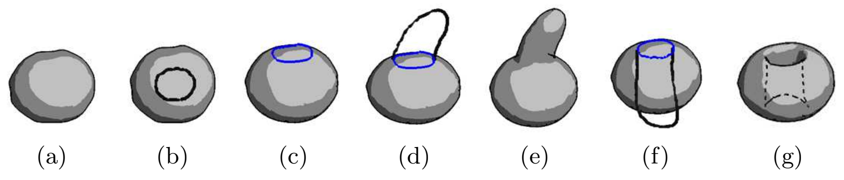


Fig. 3. Examples of Extrusion

3.3 Loop Extrusion

In addition, it is also possible to create a hollow object using extrusion. To do this, the user first cuts the model to expose the internal region (Fig. 4 (a-c)), then draws a contour on the exposed plane (Fig. 4 (d)), and finally draws a circular stroke that entirely surrounds the contour (Fig. 4 (e)). We call this operation "Loop Extrusion". The cutting operation that we use differs from the standard Cut operation in the Teddy system [6] in that the removed region is just deactivated temporarily. The system distinguishes these two operations by checking whether there is a corner at the end of a stroke. The system performs a standard cutting operation when there is no corner, while the system deactivates a region when there is a corner. The direction of the stroke end is used to determine which area to deactivate. The silhouette of the deactivated parts is rendered as broken lines.

Deactivation is provided in order to make the inside of an object accessible. The user can draw a contour and have it extrude on an internal surface in exactly the same way as done on an external surface (Fig. 5). The following sweep operation can also be used in conjunction with deactivation.

3.4 Sweep

After pressing the "Extrusion/Sweep" button, the user can also draw an open stroke specifying the sweep path. If a single contour is drawn in the first step,

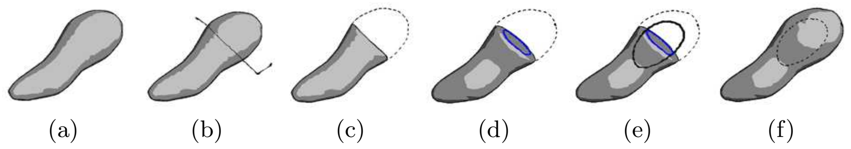


Fig. 4. An example of creating a hollow object: first, the user defines the desired cross-sectional plane by deactivating part of the object (a-c). Then, the user draws a contour on the cut plane (d). Finally, the user draws a extruding shape surrounding the contour, which we call "Loop Extrusion" (e). This creates a hollow object (f).

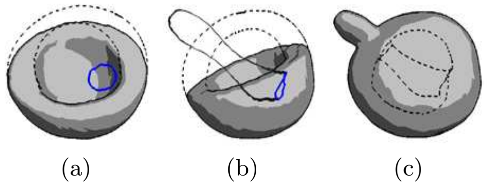


Fig. 5. A extrusion from an internal surface of an object using deactivation

both ends are checked to determine whether they are close to the projected contour. Unlike extrusion, the user can draw multiple contours to design tube-like shapes (Fig. 6).

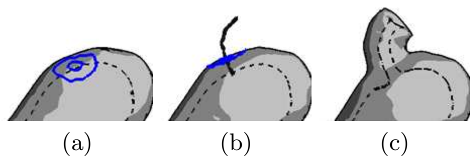


Fig. 6. Sweeping double contours: drawing contours on the surface of an object (a) and sweeping them (b) produces a tube (c).

3.5 Animation Assistance

In extrusion or sweep, the model must be rotated approximately 90 degrees after pressing the "Extrusion/Sweep" button to draw the last stroke. To automate this process, our system rotates the model after the "Extrusion/Sweep" button is pressed; the contours are then moved so that they are perpendicular to the screen (Fig. 7 (a-c)). This animation assistance is also performed after a Cut operation, because it is likely that a contour will be drawn on the cut plane in the next step. When a model is cut, it is automatically rotated so that the cut plane is parallel to the screen (Fig. 7 (d-f)).

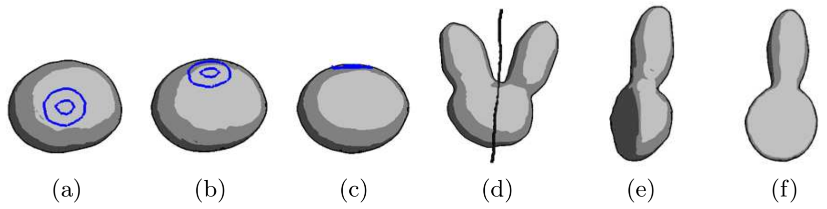


Fig. 7. Examples of animation assistance: as soon as the user presses the "Extrusion/Sweep" button, the model is rotated so that the contours are perpendicular to the screen (a-c). When the user cuts a model, the /model is automatically rotated so that the cut plane is parallel to the screen (d-f).

4 Implementation

We use a standard binary volumetric representation. The examples shown in this paper require approximately 400^3 voxels. The volumetric data are polygonized using the Marching Cubes algorithm [9]. The polygonized surface is then smoothed [13] and displayed using a non-photorealistic rendering technique [8]. The silhouette lines of invisible or deactivated parts are rendered as broken lines.

The Create and Extrusion operations can be implemented using the algorithms described in the original Teddy system, converting the resulting polygonal model into a volumetric model and performing a CSG operation. In Extrusion, our system adds the additional geometry to the original model when an outward stroke is drawn and subtracts it when an inward stroke is drawn. Note that complex "sewing" of polygons is not necessary and no self-intersection will occur because of the volumetric data structure. Loop Extrusion applies the standard inward (subtract) extrusion in both directions. The Sweep operation in our system requires two-path CSG operations to add a new geometry to the original model. First, the sweep volume of the outermost contour is subtracted from the original model (Fig. 8 (a-c)). Then, the regions between the contours are swept and the sweep volume is added to the model (Fig. 8 (d)). This avoids the inner space being filled with the original geometry.

The volumetric representation significantly simplifies the implementation of the Cut operation and enables the change in topology. A binary 2D image is computed from the cutting stroke in the screen space to specify a "delete" region and a "remain" region. Both ends of the cutting stroke are extended until they intersect or reach the edges of the screen. Then, one of the separated regions is set as the "delete" region (usually the region to the left of the stroke, following

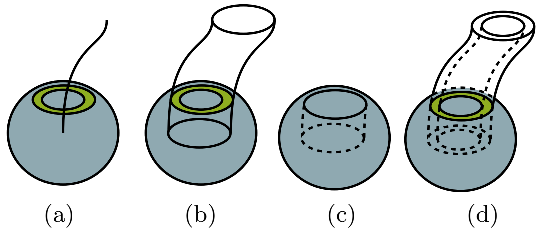


Fig. 8. Handling the sweep operation. The outmost contour is swept along the specified path (a,b) and extracted from the original model (c). Then, every contour is swept and added to the model.

the original Teddy convention). Each voxel is then projected to the screen space to check whether it is in the deleted region; if so, the voxel is deleted. This process is significantly simpler than traversing the polygonized surface and remeshing it.

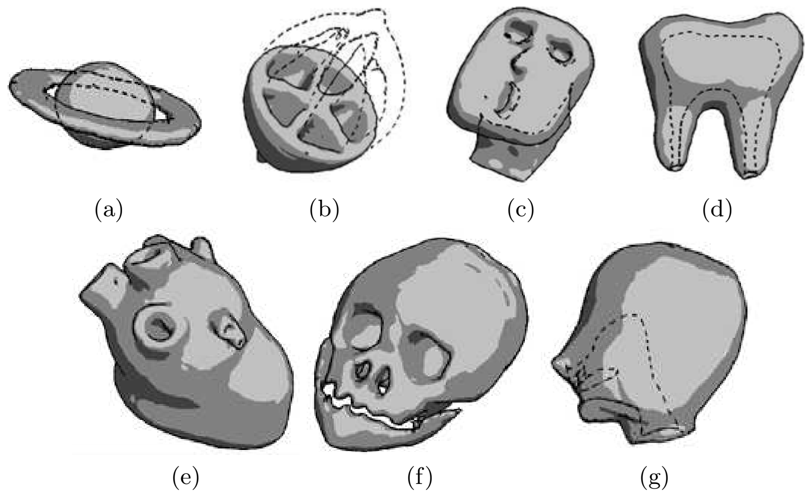


Fig. 9. Examples created using our system. (a-c) were created by novices, while (d-g) were created by an expert.

5 Results

We used a Dell Dimension 8200 computer that contained a Pentium 4 2-GHz processor and 512 MB of RAM. The graphics card was an NVIDIA GeForce3 Ti500 with 64 MB of memory. Users can create models interactively on this

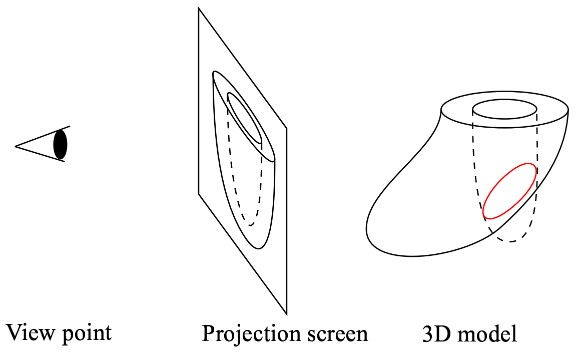


Fig. 10. An undesired effect caused by the lack of depth control. Since there is no depth information in the original model, the newly created cavity can pierce the wall.

machine. We also used a display-integrated tablet as an input device, with which the user can edit an object more intuitively. However, some users found it difficult to rotate an object because they needed to press a button attached to the side of the pen and move the pen without touching the display.

Figure 9 shows some models created using our system. Fig. 9 (a-c) were created by novices within fifteen minutes of an introductory fifteen-minute tutorial; the others were created by an expert. Our observations confirmed that users could create models with internal structures quickly and easily. Nevertheless, one limitation also became clear. The users occasionally found the behavior of Extrusion unpredictable because there was no depth control. Specifically, when a user tried to create a cavity in an object, the hole sometimes penetrated the wall of the original model (Fig.10).

6 Conclusions and Future work

We presented a sketch-based modeling system for creating objects with internal structures. The underlying volumetric data structure simplifies the handling of a dynamically changing topology. The user can modify the topology easily in various ways, such as by cutting an object, forming a extrusion, specifying multiple contours with create or sweep operations, or specifying internal structures in conjunction with temporal deactivation. In addition, automatic rotation of the object frees the user from tedious manual labor.

Our system is designed for the rapid construction of coarse models and is not appropriate for precise modeling. Currently, it is difficult to modify shapes locally and we are exploring ways to add small details. As mentioned above, the absence of depth control causes difficulty. Finally, our current implementation can produce only binary volumetric data and we plan to explore a new interface in which the user can define the internal *volumetric textures* of a model.

References

1. Bærentzen, J.A. and Christensen, N.J.: Volume Sculpting Using the Level-set Method. Proc. 2002 International Conference on Shape Modeling and Applications (2002) 175–182
2. Cutler, B., Dorsey, J., McMillian, L., Müller, M. and Jagnow, R.: A Procedural Approach to Authoring Solid Models. ACM Transactions on Graphics **21** 3 (2002) 302–311
3. Ferley, E., Cani, M.P. and Gascuel, J.D.: Practical Volumetric Sculpting. The Visual Computer **16** 8 (2000) 469–480
4. Galyean, T.A. and Hughes, J.F.: Sculpting: An Interactive Volumetric Modeling Technique. In Computer Graphics (Proc. SIGGRAPH 91) **25** 4 (1991) 267–274
5. Hua, J. and Qin, H.: Haptics-based Volumetric Modeling Using Dynamic Spline-based Implicit Functions. In Proc. 2002 IEEE Symposium on Volume Visualization and Graphics (2002) 55–64
6. Igarashi, T., Matsuoka, S. and Tanaka, H.: Teddy: A Sketching Interface for 3D Freeform Design. In Computer Graphics (Proc. SIGGRAPH 99) (1999) 409–416
7. Karpenko, O., Hughes, J.F. and Raskar, R.: Free-form Sketching with Variational Implicit Surfaces. Computer Graphics Forum **21** 3 (2002) 585–594
8. Lake, A., Marshall, C., Harris, M. and Blackstein, M.: Stylized Rendering Techniques for Scalable Real-Time 3D Animation. In Proc. Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000) (2000) 13–20
9. Lorensen, W.E. and Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In Computer Graphics (Proc. SIGGRAPH 87) **21** 4 (1987) 163–169
10. McDonnell, K.T. and Qin, H.: Dynamic Sculpting and Animation of Free-Form Subdivision Solids. The Visual Computer **18** 2 (2002) 81–96
11. Perry, R.N. and Frisken, S.F.: Kizamu: A System for Sculpting Digital Characters. In Computer Graphics (Proc. SIGGRAPH 2001) (2001) 47–56
12. Pugh, D.: Designing Solid Objects Using Interactive Sketch Interpretation. Computer Graphics (1992 Symposium on Interactive 3D Graphics) **25** 2 (1992) 117–126
13. Taubin, G.: A Signal Processing Approach to Fair Surface Design. In Computer Graphics (Proc. SIGGRAPH 95) (1995) 351–358
14. Wang, S.W. and Kaufman, A.E.: Volume Sculpting. Computer Graphics (1995 Symposium on Interactive 3D Graphics) (1995) 151–156
15. Zeleznik, R.C., Herndon, K.P. and Hughes, J.F.: SKETCH: An Interface for Sketching 3D Scenes. In Computer Graphics (Proc. SIGGRAPH 96) (1996) 163–170

Shape Modeling by Sketching using Convolution Surfaces

Anca Alexe[1], Loic Barthe[1],Marie Paule Cani[2], Véronique Gaildrat[1]
1. IRIT - CNRS, UPS Toulouse, France
2. GRAVIR - CNRS, INRIA, UJF, INP Grenoble, France
E-mail: alexe,lbarthe,gaildrat@irit.fr, Marie-Paule.Cani@imag.fr

Abstract

This paper proposes a user-friendly modeling system that interactively generates 3D organic-like shapes from user drawn sketches. A skeleton, in the form of a graph of branching polylines and polygons, is first extracted from the user's sketch. The 3D shape is then defined as a convolution surface generated by this skeleton. The skeleton's resolution is adapted according to the level of detail selected by the user. The subsequent 2D strokes are used to infer new object parts, which are combined with the existing shape using CSG operators. We propose an algorithm for computing a skeleton defined as a connected graph of polylines and polygons. To combine the primitives we propose precise CSG operators for a convolution surfaces blending hierarchy.

Our new formulation has the advantage of requiring no optimization step for fitting the 3D shape to the 2D contours. This yields interactive performances and avoids any non-desired oscillation of the reconstructed surface. As our results show, our system allows non-expert users to generate a wide variety of free form shapes with an easy to use sketch-based interface.

Keywords: sketch based modeling, implicit surfaces, convolution surfaces, CSG.

1. Introduction

The complexity of user interaction is the main obstacle to the use of standard modeling systems. This impacts both the user and the possibilities of expression this system provides. Achieving a simple and faithful translation of the user's idea without requiring sophisticated input and a long training process remains a challenge for the modeling software. One of the simplest and user-friendliest modeling metaphors is drawing. This kind of communication is useful in educational applications such as teaching, and already has industrial purposes such as story boarding. It is generally used in the early stages of design, because drawing a sketch is both much faster than creating a 3D model, and more convenient for expressing ideas. However, the obvious drawback of 2D sketches is their limitation to a single viewpoint. The user cannot move around the drawn object, nor view it from different angles, except in [4] where the sketch cannot be used for extracting a 3D shape. The aim of the sketch-based modeling is to combine the simplicity of 2D sketching with powerful 3D capabilities. Since the first sketch based interface [11] the concept has been largely developed and explored, from architectural design [5] to artistic design [8] and free form shapes [6, 7, 10]. The latter are difficult to model with sketches, though among the most interesting because of the large modeling possibilities they provide. The main difficulty in reconstructing a 3D model from a 2D contour is extrapolating lacking information. There are two main approaches for constructing smooth, rounded shapes from 2D contours. The first one consists in perspective projections of the contour point samples to reconstruct the 3D geometry. These points are then interpolated using variational implicit surfaces [7, 12, 5]. The second is to construct a skeleton from the 2D contour and use it to generate a 3D shape [6, 10, 1]. The main drawback for the first approach is that the surface has to be recalculated every time it is edited and the time taken to compute the coefficients for the variational implicit surfaces increases with the number of points. Also, small details are lost when blending the object parts because preserving them would require too many constraints and too much computation. Therefore we prefer the second approach. Previous research in this field has raised some difficulties. One of these is the necessity of an optimization step to adjust the implicit surface to the drawn contour. This leads to a better contour approximation in terms of error but the surface oscillates [10, 1]. Moreover it is time consuming and in the context of sketch based interface providing very accurate reconstruction is not necessary. Indeed, the user drawn contour is seldom noisy so we rather aim at getting a smooth shape with close appearance to the contour. Removing the optimization step saves time and reconstructs a smooth non-oscillating surface. Of course the contour approximation constraints have to remain satisfied. Another drawback of most of the previous approaches is that the shape thickness is automatically inferred so the result may differ from what the user wanted. For example if the user draws the shape of the palm of a hand, the fingers will be smoothly reconstructed as cylinders, whereas the palm will look like a sphere, far from the user’s expectation. In [10] the problem is addressed by asking the user to provide additional information about the cross section's profile.

This increases the complexity of the interface and for this reason the technique might not be intuitive enough for non-expert users.

Our contribution

We propose a representation that allows for a great variety of topological shapes, a richer collection of sketch-based operations, an adaptive level of detail for sketch modeling with precise control of the result up to small details, while keeping a very simple and friendly user interface. For this purpose we reconstruct the 3D shape using convolution surfaces [3] with both polylines and polygons skeletons. The primitives are composed with CSG blending in a blending hierarchy.

Section 2 presents our system from the user's point of view. Section 3 presents the application from the system's point of view, i.e. the algorithms and the techniques used. Section 4 shows and discusses some results and also draws the conclusions and perspectives of our work.

2. From the user viewpoint

The purpose of our system is to enlarge the possibilities offered by the paper-pencil 3D modeling metaphor, while keeping a simple and intuitive input interface. The modeling process iterates the following steps until modeling is complete:

1. The user draws one or several strokes
2. The strokes are interpreted to reconstruct a 3D object part
3. The part is added to the current object (or subtracted if carving)

As the user draws a stroke, its thickness and color intensity vary proportionally with the pressure on the digital pen, as to imitate the irregular density and thickness of the strokes produced by a real pen. Several strokes accumulated in the same pixel result in a darker color for that pixel. The other end of the pen is used as an eraser. As long as the stroke has not been reconstructed, the user is free to erase and modify it. This way the user's input is allowed to be noisy and irregular, as it is naturally on paper. To create a new shape, the user draws a contour on the graphic tablet using the digital pen. Once the contour has been completed the user presses the digital pen against the tablet. This produces the 3D reconstruction of the stroke (see Fig. 1 (a),(b)). To add part to an existing shape, the mechanism is the same as for creation. The first surface point hit by the user gives the depth of the shape to be constructed. When the stroke is complete, the user presses the stylus if he wishes to add the shape to the existing object, or the eraser (at the opposite pen's end) if he wishes to carve it into the object. The shape is reconstructed in such a manner that the projection of the shape on the screen fits the contour that has been drawn by the user (see Fig. 1 (c),(d),(e) and (f),(g),(h)). The user controls the thickness of the shape using the pen's bend (see Fig. 1 (i),(j),(k)). Small details can be modeled by zooming to get closer to the object. The large object parts will smoothly blend with each other, while the small details (e.g. eyes, nose of a character) will have a sharper blending. The user can paint directly on the objects or in space next to them. In this way additional information or annotation can be added to the model.

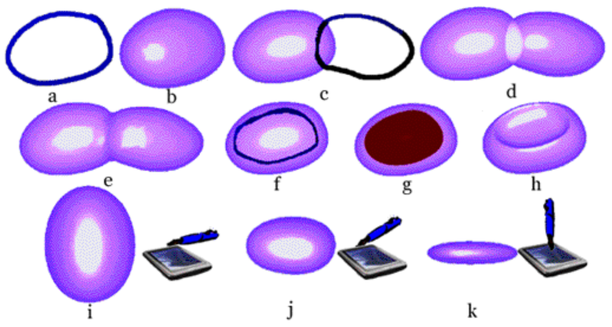


Figure 1: (a), (b) Creating a part. (c),(d),(e) Adding a part to an object. (f),(g),(h) Carving. (i),(j),(k) Thickness control (side view).

3. From the system viewpoint

A pressure threshold indicates that the drawing is finished. When the stylus pressure has reached this threshold, the strokes image is recovered as a 2D bitmap, then compressed and reduced in size using a pixel averaging technique. This also reduces the amount of computation for the skeleton. In order to perform the skeleton extraction we iteratively construct a connected pixels skeleton, which is then sampled in order to obtain a segments and triangles graph [13]. This will be used to define a convolution surface [3]. In order to obtain interactive modeling, we use the pseudo Cauchy [9] convolution kernel, which gives a closed form solution for the convolution integral for the primitives that we use (segments and triangles). See [13] for a full algorithm description.

The addition and subtraction operations are defined using CSG, for which we have adapted the composition model shown in [2] and rewrote the union and difference operators in order to allow hierarchical exact composition. The level of detail of the skeleton remains constant in the image space, but it is automatically adapted to the level of detail of the 3D shape, given by the distance between the object and the camera. The level of detail determines the blending parameters, the skeleton weights and the size of the polygonization cell for the shape to be reconstructed. The polygonization for the reconstructed stroke is computed and displayed immediately, while a process in the background computes the final surface polygonization. The final mesh is displayed as soon as it is available. This allows maintaining interactive rates and rapid application response during the modeling process so that the user feels free to pursue his modeling activity.

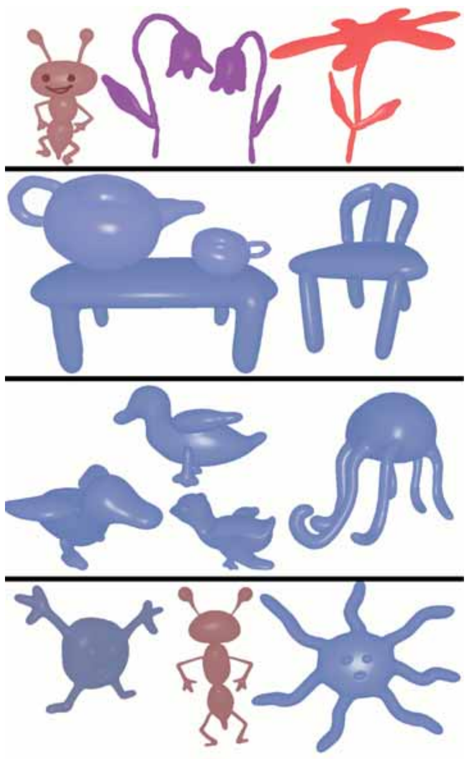


Figure 2: Objects modelled with our system. The user took 2 to 5 minutes overall modelling time and 3 to 9 strokes for each object.

5. Results and conclusions

Convolution surfaces allow much better shape representation than standard skeleton based implicit surfaces, due to their possibility to represent flat surfaces, as well as a large topological variety. Fig. 2 shows objects modeled with our system. The system provides a real simplicity for the non-expert user, for example three strokes only are necessary to create each one of the birds in Fig. 2 (with symmetry enabled for the wings and legs). The Fig. 2 also shows flat surfaces (table and chairs). The shapes have no oscillations and no bulges. The CSG composition is a generalized composition more flexible and accurate than the simple sum, allowing a better blending control, from smooth to sharp transitions. The small details of the objects are well preserved due to the parametrable CSG.

For example, the sun's eyes and mouth are small details compared to the face but they are well preserved by the blending. The shape may have various topologies (ex. chairs, teapot) and can be carved (teapot, mugs). The applications of our system are educational, but also story boarding for films making (ex. cartoons, see Fig. 2) where the scenarios writer is not necessary a 3D designer. The system could be extended to design the internal structure of organic shapes because the composition model is suitable for this.

In the future we would also like to focus on accelerating the polygonization time for generating the final implicit surface and investigate adaptive polygonization.

References

[1] A. Alexe, V. Gaildrat, and L. Barthe, “Interactive modeling from sketches using spherical implicit functions”, *AFRIGRAPH*, Stellenbosch, November 2004.

[2] L. Barthe, V. Gaildrat, and R. Caubet, “Combining implicit surfaces with soft blending in a CSG tree”, *CSG Conference Series*, April 1998.

[3] J. Bloomenthal and K. Shoemake, “Convolution surfaces”, *Computer Graphics*, 25(4):251-256, 1991.

[4] D. Bourguignon, M.-P. Cani, and G. Drettakis, “Drawing for illustration and annotation in 3D”, *Computer Graphics Forum*, vol. 20, pages 114-122., Blackwell Publishers, September 2001.

[5] A. Cuno, C. Esperança, P. R. Cavalcanti, R. F. Cavalcanti, and R. Farias, “3D free free-form modeling with variational surfaces”, *WSCG*, February 2005.

[6] T. Igarashi, S. Matsuoka, and H. Tanaka. “Teddy: A sketching interface for 3d freeform design”, *SIGGRAPH 99*, August 1999, pages 409-416.

[7] O. Karpenko, J. F. Hughes, and R. Raskar, “Free-form sketching with variational implicit surfaces”, *Computer Graphics Forum* 21(3):585-594, September 2002.

[8] D. F. Keefe, D. Acevedo, Moscovich, Tomer, Laidlaw, H. David, and J. J. LaViola. “Cavepainting: A fully immersive 3d artistic medium and interactive experience”, *Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 2001.

[9] A. Sherstyuk, “Kernel functions in convolution surfaces: a comparative analysis”, *The Visual Computer*, 15(4), 1999.

[10] C.-L. Tai, H. Zhang, and C.-K. Fong, “Prototype modeling from sketched silhouettes based on convolution surfaces”, Computer Graphics Forum, 23(4):855{83, 2004.

[7] R. Zeleznik, K. Herndon, and J. Hughes, “Sketch: An interface for sketching 3d scenes”, *ACM Transactions on Graphics*, Proceedings of SIGGRAPH, 1996.

[12] R. Zenka and P. Slavik, “New dimension for sketches”, *In SCCG 2003.*

[13] A. Alexe, L. Barthe, M.P. Cani, V. Gaildrat, “A Sketch-Based Modelling system using Convolution Surfaces”, *Technical Report* IRIT/2005-17-R, July 200

SmoothSketch: 3D free-form shapes from complex sketches

Olga A. Karpenko* Brown University
John F. Hughes† Brown University

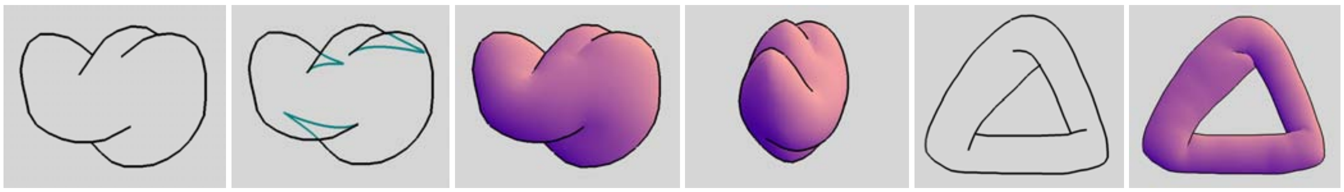


Figure 1: The user draws the visible contours of a shape; our program infers the hidden contours, including hidden cusps, and then creates a fairly smooth 3D shape matching those contours. The 3D shape can be viewed from any direction. A smoothed version of Penrose's polyhedral impossible triangle shows that the algorithm can handle objects with complex holes.

Abstract

We introduce SmoothSketch—a system for inferring plausible 3D free-form shapes from visible-contour sketches. In our system, a user's sketch need not be a simple closed curve as in Igarashi's Teddy [1999], but may have cusps and T-junctions, i.e., endpoints of hidden parts of the contour. We follow a process suggested by Williams [1994] for inferring a smooth solid shape from its visible contours: completion of hidden contours, topological shape reconstruction, and smoothly embedding the shape via relaxation. Our main contribution is a practical method to go from a contour drawing to a fairly smooth surface with that drawing as its visible contour. In doing so, we make several technical contributions:

- extending Williams' and Mumford's work [Mumford 1994] on figural completion of hidden contours containing T-junctions to contours containing cusps as well,
- characterizing a class of visible-contour drawings for which inflation can be proved possible,
- finding a topological embedding of the combinatorial surface that Williams creates from the figural completion, and
- creating a fairly smooth solid shape by smoothing the topological embedding using a mass-spring system.

We handle many kinds of drawings (including objects with holes), and the generated shapes are plausible interpretations of the sketches. The method can be incorporated into any sketch-based free-form modeling interface like Teddy.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Modeling Packages; I.2.10 [Artificial Intelligence]: Vision and Scene Understanding—Shape

*e-mail: koa@cs.brown.edu
†e-mail: jfh@cs.brown.edu

1 Introduction

Our visual system, presented with a line-drawing, makes nearly-instant inferences about the shape that the drawing represents. Some aspects of this inference mechanism are surely based on expectation — when we see something that looks like an ear, it's easy to infer that the nearby protrusion must be a nose, for instance. But other aspects depend on local cues — a contour that ends, or that disappears behind another — and a gestalt view that helps us integrate these local cues into a coherent whole [Hoffman 2000].

Dual to this recognition ability is our ability to learn to draw contours of objects in a way that lets us communicate their shape to others. While drawing *well* can be difficult, even children can draw easily recognizable shapes. On the other hand, while drawing outlines or contours is relatively easy, we know few people who can reliably draw the hidden contours of even simple shapes.

When we seek to create shapes with a computer, however, there are few interfaces based directly on drawing; inferring a shape from a complex contour-sketch has generally proved too difficult. The value in doing so, from the sketching point of view, is that it allows a user to draw what he or she is thinking of directly. Teddy's inflation algorithm is a good step, but limited to simple closed curve contours. Our work extends this substantially, although it is by no means a final answer. Such a final answer may never be found, though—it's easy to draw contour sets that are so complicated that different viewers make different inferences about them. The best one can hope for is to create plausible shapes for a fairly large class of contours on which users agree on the interpretation. That is what our work does.

Our work, therefore, is not about a *system* like Teddy; it's about a *component* that can be used in a free-form-sketching interface like Teddy. We believe that a sketching program should let the user and the computer share the work, each doing what it does best. The computer can infer a plausible shape from a moderately complex contour like the ones shown in this paper. Then, to create more complex objects (or to, say, modify the thickness of the inflated models), a user would use various gestures like the ones available in Teddy and other sketch-based systems. We believe we succeeded in the first step, and thus provided a new starting-point for sketch-based systems.

Our system takes a user's contour-drawing of a smooth, compact, oriented, embedded surface-without-boundary (which we'll call a *good* surface) and determines a 3D surface whose contours match those that the user drew. Because this is an under-determined problem—is that circle a contour drawing of a pancake? a sphere? a cigar viewed end-on?—we aim to generate surfaces that have generally low curvatures, to the degree allowed by the constraints of the contours. The contours must be *oriented*, i.e., drawn so the surface lies on the left. Thus to draw a torus, a user would draw a counter-clockwise outer stroke and a clockwise inner stroke.

Williams' thesis [1994] and subsequent work lay out a plan for finding a surface fitting a given collection of visible contours[1]. The steps involved are

1. *Complete* the drawing by inferring the hidden contours, and provide a Huffman-labeling for it [Huffman 1971].
2. *Convert* the completed-drawing to an abstract topological surface, and map this surface to $\Re^2$ so that the "folds" of the mapping match the contours of the drawing.
3. *Lift* this mapping to a smooth embedding in $\Re^3$ whose projection is the mapping to $\Re^2$.

Having laid out this scheme, Williams then completes several significant parts: he completes step 1 for "anterior surfaces" – roughly the front-facing parts of scenes, which generically have no cusps, and does step 2 for both these *and* for drawings of smooth surfaces. For our purposes, we need step 1 for good surfaces, and we need step 3, which we do in two steps: first we lift to a topological embedding in $\Re^3$, and then we smooth that. We cannot claim to produce a *smooth embedded* manifold; with our current tuning constants, our results are usually immersed (i.e., have self intersections) rather than embedded. Furthermore, the contours of the surfaces we construct cannot in general project exactly to the input drawing, because, for example, the projection of the contour near a cusp always has infinite curvature at the cusp [Koenderink 1990], while our user's input may not satisfy that constraint. We produce a fairly smooth mapping of the manifold into 3-space, whose visible-contour projection nearly matches the user's drawing.

In carrying out step 2, Williams (a) observes that the projection of the complete contours of a generic view of a good surface onto an image plane gives a knot-drawing-with-cusps which can be labeled by Huffman's labeling scheme for smooth surfaces (see figure 2) [Huffman 1971], and (b) gives a method (the *paneling construction*) [Griffiths 1981] to build an abstract manifold M that can be embedded so that its projection has contours matching the knot-drawing. (He does not actually construct an embedding $e : M \to \Re^3$, but instead describes a map $f : M \to \Re^2$ with the property that if such an embedding e exists, and if P is projection onto the drawing plane, then $f = P \circ e$.)

To carry out step 1 for good surfaces, we must establish which drawings of visible contours can be extended to complete contour drawings; not all can, as figure 3 shows. We partially solve this problem by exhibiting a large class of drawings that admit such extensions; the general problem of characterizing extendable visible-contour drawings remains open, however.

For the anterior-surface case, Williams and Jacobs [1997] (and Mumford [1994]) describe an approach to completing the hidden contours, which generically join tee-points in the drawing (see figure 4). To join a pair of tees, they consider all C^1 random walks (i.e., random walks in which the tangent direction θ changes by an amount X at each point, where X is a Gaussian random variable) starting at the first tee, headed in the right direction, and ending at the second, and assign to each a probability based on the product of the probabilities of each angle-change and $e^{-\lambda}$, where λ is the

[1]The reader interested in implementing the ideas of this paper will need first to become acquainted with Williams' work.

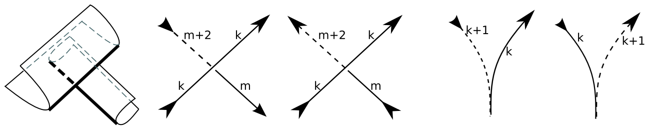


Figure 2: Huffman's labeling scheme for contours of generic smooth projections. Labels indicate the number of surfaces in front of the contour (visible contours have label zero); the surface on which the contour lies is to the left when you traverse it in the direction shown by the arrow. *Any* cusped knot diagram—i.e., collection of circles in the plane, possibly intersecting themselves and each other, and smoothly immersed except at finitely many cusp points, which are distinct from the crossing points—that can be so labeled corresponds to the projection of a smooth surface in 3-space (mostly proved by Williams), and all generic projections of smooth surfaces have this property (proved by Huffman). The first picture shows the surfaces corresponding to the first case of Huffman's labeling (the second picture). Figure 6 (left) corresponds to the last case.

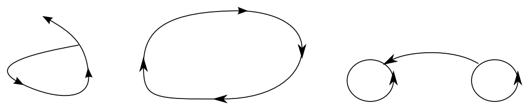


Figure 3: None of these drawings can be extended by invisible contours to be the contour set of any good manifold projection. They exhibit two problems: in the first, a cusp appears in the outer region of the plane surrounding the figure. In the second, the outermost path around the drawing is clockwise. Although the third has neither of these problems, it is still not extendable.

length of the curve. They posit that the maximum-likelihood random walk is a good candidate for the completion; when multiple pairs of tees might be joined, they compute which pairings have largest likelihoods and choose those.

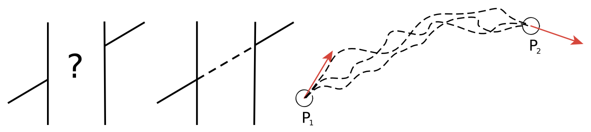


Figure 4: How can we join the two tee-points on the left? With an optimal completion, as shown in the middle. Optimality is determined by choosing, among all C^1 random walks from p_1 to p_2, the most likely one, under a simple probabilistic model. Mumford shows such curves are elastica, which had been studied by Euler.

We extend this approach, in Section 4.2, to the cases where a T-point must be joined to a cusp, or two cusps must be joined. To determine which visible endpoints (tees or cusps) should be joined to which, we use a greedy search similar to Nitzberg *et al.* [1993].

To carry out step 3, we take the results of Williams' paneling construction — an abstract manifold and a continuous mapping f of it to $\Re^2$ and "lift" it to a mapping e into $\Re^3$ whose projection is f. We construct e a dimension at a time, first placing the vertices of the paneling construction, then embedding the edges, and finally each panel. This algorithm is described in section 5.1. The result is a topological embedding (i.e., a 1-1 continuous map from the surface into $\Re^3$). Finally, in section 6 we talk about smoothing out the creases in this topological embedding by an ad-hoc mass-spring system to produce the desired fairly smooth mesh in 3-space.

2 Related Work

Shape from drawings. The problem of inferring 3D shape from 2D drawings has been studied in a great many forms; if one extends it to include determining drawings from images as a first step, it occupies much of the computer vision literature [Witkin 1980]. We'll only describe the work most closely related to this paper.

Much early shape-from-drawing work applied to blueprint-like drawings of machined surfaces. The important features of such shapes are sharp bends, like the edges of a cube – and their trihedral intersections. Lipson and Shpitalni [1997] introduced a system in which a user sketches both visible and hidden contours and boundaries of a rectilinear CAD-like geometric object, and the system infers a shape. Their approach is based on correlations among arrangements of lines in the drawing and lines in space.

Pentland and Kuo [1989] presented a system that infers simple 3D curves and surface patches from 2D strokes by minimizing the energy of the corresponding *snakes*.

A classic paper in this area is by Huffman [1971], who developed two labeling schemes—one for objects made from planar surfaces, one for smooth objects—and proved that their complete contour drawings must have the corresponding sorts of labeling. Williams [1994] [1997] did the defining work in inverting the smooth-surface labeling scheme, as described in the introduction.

Contour completion. Of course, Huffman labelings are for complete contour projections — the projections of both the visible and invisible parts of an object's contours. Given a drawing of the visible parts of a contour, we must infer where the invisible parts lie. Kanizsa's work [1979] on contour completion (and its relationship to the mechanisms of the human visual system) forms the basis for much of the later work in the area.

A solution proposed first by Grenander [1981] was to use a stochastic process to model the space of all possible edges. Mumford proves that elastica that arise in the completion problem described in the introduction could be modeled by a white noise stochastic process and gives needed formulas. Williams [1997] approximates the solution by considering a sampling of the space of all random walks (with varying $\Delta\theta$ — the direction of the walk) starting from the first point with the first direction and coming to the second point with the second direction, and taking the random walk with the highest probability as the best path connecting two edges.

Although contour completion is a well-studied research topic, many problems are still open; in section 4.2 we propose our solution, inspired by the work of Williams and Mumford, to the problem of finding a hidden cusp for a cusp-contour completion case.

Sketching interfaces. Several gestural interfaces for sketching 3D shapes have been developed for different classes of models. For rectilinear objects, the Sketch system described by Zeleznik *et al.* [1996] lets a user create and edit models through gestural interface, where geometric aspects of gestures determine numerical parameters of the objects; a cuboid is created by drawing three lines meeting at a point; the lengths of the lines and position of the point determine the geometry of the cuboid. These ideas were extended by several research groups [Shesh and Chen 2004] [Pereira et al. 2004], and appear in the SketchUp [SketchUp] architectural design software.

For free-form objects, Igarashi's Teddy [1999] was the first interface for free-form modeling via sketching. In it, a user inputs a simple closed curve and the system creates a shape matching this contour. Then the user can add details by editing the mesh with operations like extrusion, cutting and bending, all done gesturally. The Smooth Teddy [Igarashi and Hughes 2003] system extended this by adding algorithms for beautification and mesh refinement, as well as organizing the shapes into a hierarchy.

Karpenko *et al.* [2002] described a system for creating shapes from free-form sketches; the primitive objects were variational implicit surfaces, which facilitated operations like surface blending. ShapeShop [Schmidt et al. 2005] uses hierarchical implicit volume models to let a user interactively edit complex models via a sketching interface. Alexe *et al.* [2005] extract the skeleton from the sketch and then construct a convolution surface. None of these systems handle complex strokes containing tees and cusps.

Nealen *et al.* [2005] presented a sketch-based interface for laplacian mesh editing where a user draws reference and target curves on the mesh to specify the mesh deformation. A similar interface was developed by Kho and Garland [2005] for posing 3D characters, in particular, bodies and limbs.

Finally, Karpenko and Hughes [2005] demonstrated a method for inferring certain free-form shapes from sketches by detecting 'templates' in the sketches and building a part of the 3D surface from a standard recipe for each template.

Pseudo-3D models. Tolba *et al.* [2001] describe a system that lets a user draw a scene with 2D strokes and then view it from several new locations as if a 3D scene had been created. This is done by projecting the 2D strokes on a sphere centered at the eye point and then viewing them in perspective. Bourguignon *et al.* [2001], describe a system that takes a set of 3D strokes representing contours and creates a small piece of surface near each stroke whose contour is the given stroke; contours of this surface, seen from nearby viewpoints, give the appearance of a full-fledged 3D model, although in distant viewpoints the illusion is lost. Johnston [2002] computes lighting on 2D drawings without reconstructing 3D geometry by estimating surface normals from the drawing.

Shape from contours for special classes, and other shape-from methods. Ulupinar *et al.* [1995] solve the "shape-from-contour" problem for images by considering only a special class of symmetrical 3D shapes: straight homogeneous- and constant cross section generalized cylinders. Apart from inferring the shape from the contour, researchers have long tried to infer shape from texture, shading, and other cues. An overview of some of these methods can be found in the paper by Ulupinar [1993].

3 Notation and problem formulation

Much of the material that follows relies on ideas from differential geometry and combinatorial and differential topology. We refer the reader to the books of Guillemin and Pollack [1974] and Koenderink [1990] for clear expositions of the necessary background.

Suppose that S is a smooth, closed, compact, orientable surface-without-boundary (i.e., a *good* surface) embedded in the $z > 0$ half-space of $\Re^3$. The orthogonal projection of S onto the $z = 0$ plane will have a compact image. Following Williams and Mumford, we will assume that the embedding and this projection are *generic*, i.e., that no probability-zero events occur, e.g., no projector meets three contours, no cusp projects to a point on another contour, etc. If the projector through the point $s \in S$ lies in the tangent plane at s, then s is called a *contour point*; if the projector first meets S at s, then s is a *visible* contour point (see figure 5). For a generic projection, the set of all contour points forms a compact 1-manifold-without-boundary C in S, i.e., a collection of disjoint topological circles in S. The set of visible contour points form a compact 1-manifold-with-boundary, V in S, i.e., a collection of disjoint topological circles and line-segments. The projection of C to the $z = 0$ plane is the *contour drawing* of S; the projection of V to the $z = 0$ plane is the *visible contour drawing* of S.

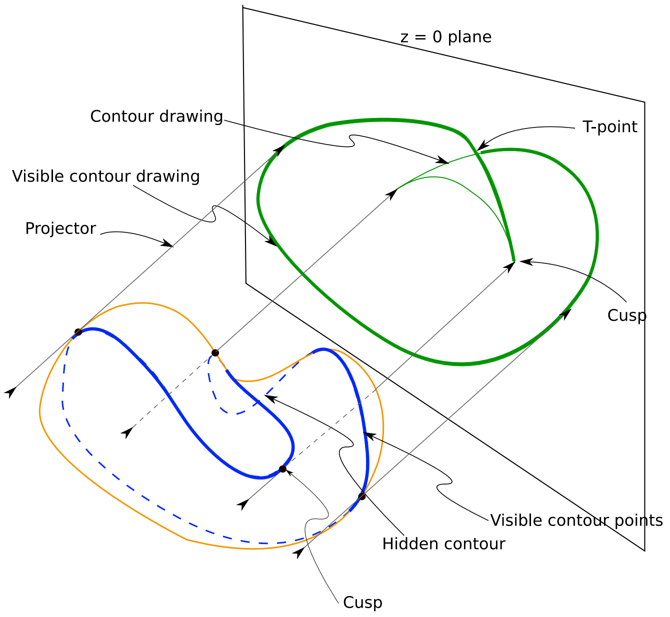


Figure 5: (Adapted from Williams' [1997]) The contour, in blue, of a good surface embedded generically in 3-space projects to a contour drawing, in green; the visible contour (drawn bold) projects to the visible-contour drawing. A point where the projector is tangent to the contour projects to a cusp in the contour drawing. The restriction of the projection to just the contour is 1-1 except at finitely many points, where two contours cross in the drawing; these are called T-points.

The projection from the contour to the contour drawing is an embedding at most points; the exceptions are *crossings*, where two contours meet, and *cusps*. A cusp is a point $s \in S$ where the projector through s is tangent to C at s. The projection of a cusp appears as a point where the contour drawing "reverses direction" (see figure 6, left). When an arc of the visible contour drawing reaches a crossing, it appears as a *T-point*: one part of the contour becomes invisible there.

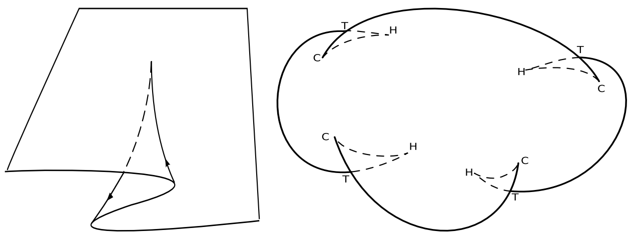


Figure 6: (Left) The generic projection of a contour at a cusp reverses direction at the cusp. (Right) A drawing with the tee-points and cusps marked; hidden contours and hidden cusps that must be inferred are shown in dotted lines.

For a generic smooth surface and viewpoint, tees and cusps of the contour will be isolated, as will curvature zeroes of the contour; this guarantees a unique osculating plane at a cusp, which means the projected contour must reverse direction rather than emanating from the cusp in any other direction (see figure 6, left).

The "bean" example (figure 5) is something of an archetype for the method described in this paper, in the sense that it's the simplest shape that has a cusp; the way that this single cusp is processed is the key to processing more general drawings, hence we use the bean as an example throughout.

In Figure 6 (right) we show in solid lines a typical input drawing; in dotted lines are the projections of invisible contours. Certain hidden contour points are also cusp-points; the visible cusps are marked with a "C" while the hidden cusps are marked with an "H".

Note that the user input is the part of the contour drawn in solid lines. Everything marked by a dashed line is a part of a *hidden* contour and needs to be inferred by our program.

With this terminology, our goal is to take a user-provided directed visible-contour drawing of a good surface as above and to determine a surface S whose visible contours match the given drawing. Note that we do not seek to reconstruct exactly the surface that the user was drawing; the map from surfaces to drawings is many-to-one, hence non-invertible. Note too that we require that the drawing arise from *some* surface, so that the problem has at least one solution; a drawing consisting of a single line segment, for instance, cannot be the projection of the visible contours of any surface.

Our system currently produces *a* surface consistent with the user's drawing, and one which we generally find to be plausible. Eventually, we would like to solve a more general problem: we want not only to produce *one* of the reasonable-looking shapes, we would like to return *the most natural shape*. Of course, "most natural" can be very subjective and depend on a user's preferences, but experience shows that people generally agree about what a drawing conveys. We could take cartoon illustrations (see figure 7) as an example. These pictures vary from simple to very sophisticated, but their expressiveness is such that people interpret them immediately. Such a degree of "naturalness" (or indeed, any way of measuring it) appears to be a very long-range goal.

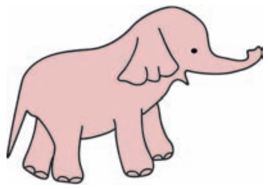


Figure 7: This cartoon-like illustration shows us how even the simplest drawings can have complex contours.

4 Figural Completion for Smooth Surfaces

Given the visible contour drawing, in the $z = 0$ plane, for a good surface in $\Re^3$, we describe an approach to completing the drawing, i.e., adding hidden contours so that the resulting drawing can be Huffman-labeled. The approach works in a large number of cases, although not all. We begin by showing a construction that provably works for a large class of drawings, but often produces "unlikely" completions according to the Williams-Mumford measure of likelihood; we then describe our actual implementation, which approximates the construction while preferring 'more likely' completions that occasionally lead to problems.

4.1 Completable drawings

Although the class of visible-contour drawings that can be completed has not been characterized, to the best of our knowledge, we can demonstrate that a large class *can* be completed. Once again we consider the 'bean' as our example (see figure 8). Taking a directed visible contour as input, we consider the regions into which it divides the plane; traversing the boundaries of these regions, and pushing slightly into each region, gives a collection of disjoint embedded curves (which we call *red curves*), each of which may pass by some number of T-points or cusps. We give the key steps in an argument that if (a) the curve for the outermost region passes no such points, and has turning number one (i.e., can be smoothly deformed to a counter-clockwise circle) and (b) all other curves pass an equal number of "starting" and "ending" points, then there is a Huffman-label-able completion of the visible contour. The formal proof involves careful application of the tubular neighborhood theorem, the isotopy extension theorem, and other standard techniques from topology; we present only the essential insights here.

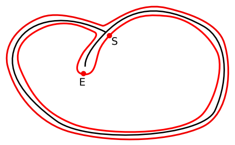


Figure 8: The visible contours of the bean divide the plane into two regions; traversing a path slightly displaced from the boundaries of these regions gives the two red curves shown. The inner one passes the cusp and then the T-point (we don't count where it makes a turn at the T-point as "passing" it); the passages are marked with dots. The outer one passes no endpoints at all.

The first step is to assign depth 0 to all visible edges. Now consider one of the red curves that meets at least two endpoints. Starting from any point of the red curve, we traverse it, noting whether the points we encounter are "starting points" (S) or "ending points" (E) of the visible contour arcs. The resulting circular sequence of Ss and Es contains at least one of each, by hypothesis; there must therefore be an adjacent pair of points, one a starting point and one an ending point. We'll show how to remove these from the sequence by completing the two contours; induction then shows that all contours can be completed and we are done.

The adjacent S and E points can be either cusps or Ts. Figure 9 shows how these can be joined. In the cusp-tee case, we can add a hidden contour and a hidden cusp, all within the region between the red curve and the contour between the two points being processed; after this addition, the red arc can be redrawn; the points S and E are no longer arc endpoints, and thus the start-end sequence for the arc is now two characters shorter. Similarly, in the T-T case, we can add a short completion arc. The only remaining case is the cusp-cusp case. Depending on whether the cusps appear in S-E order or E-S order, one of two standard solutions shown provides the necessary completion.

We note that there are drawings that do not satisfy the criterion above, but which nonetheless admit completions, so this class, although large, is not exhaustive.

4.2 Practical contour completion

The completions described in the previous section are formally correct, but since many of them have sharp turns in the hidden contours, they are, from the Williams-Mumford random-walk perspective, unlikely. As a practical matter, therefore, we take a different approach in our program: we consider all visible-contour endpoints, and estimate the likelihood of a hidden contour joining each possible pair. Following Nitzberg *et al.* [1993], we pair up points using their greedy algorithm, testing multiple configurations for (a) probability, and (b) consistency (can they be Huffman-labeled?); if the most-likely configuration is inconsistent, we move to the next-most-likely, and so on.

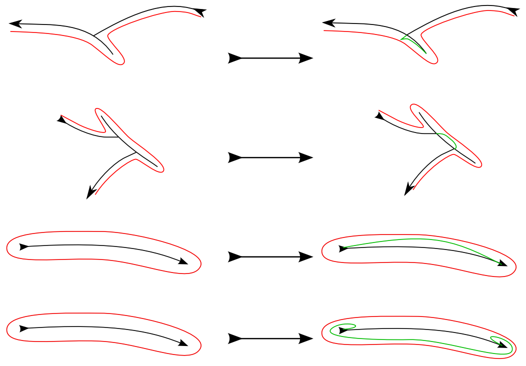


Figure 9: Adjacent cusp-T pairs can be joined with two arcs and a hidden cusp; adjacent T-T pairs can be joined with a single arc, as can adjacent cusp-cusp pairs; the side on which the arc lies, in this case, depends on the order in which the two cusps were encountered.

Pairwise completions. First, for each pair of endpoints of the visible contour we compute an initial estimate of the probability that they are connected by a hidden contour. Each endpoint has a location and associated direction for the completion curve (for T-points, the direction is given by the tangent ray of the visible contour; for cusps it is the opposite). To compute the likelihood of joining two tees or two cusps, we compute an energy function for the pairing, inversely proportional to the likelihood. The energy function of the pairing is a sum of two energy functions $E = E_{curve} + E_{endpoints}$, where E_{curve} is the energy of the curve that would connect them were they to be matched and $E_{endpoints}$ is the energy corresponding to the heuristic defined by the endpoint tangent directions. We now describe each in detail. First, we approximate the elastica curve with a Bézier spline connecting the endpoints given their tangent vectors. The Bézier curve is defined by the two endpoints and the points displaced from the endpoints along the tangent vectors. The distance by which the endpoints are displaced along the tangents is $\frac{1}{3}$ of the distance between the endpoints. The Bézier curve is then uniformly sampled and the energy function of the resulting polyline is computed as follows (see figure 10):

$$E_{curve} = e^{\sum_i l_i} \cdot \sum_i \Delta\theta_i$$

where l_i is the length of the i-th segment of the polyline, and $\Delta\theta_i$ is the absolute value of the angle change between two consecutive segments of the polyline. $E_{endpoints}$ corresponds to another heuristic similar to [Nitzberg et al. 1993], where we use the tangents at the endpoints to estimate the likelihood of the matches. Intuitively, if the tangent directions at the two endpoints are very similar, it is likely for them to be paired even if the length of the curve connecting them would be long (think of a fat snake whose tail passes behind its body). Similarly, if the tangents at the endpoints are very different, it should be pretty unlikely for them to be paired up. Currently, $E_{endpoints}$ is a constant 1.0 if the angle between the tangents at the endpoints is between $\varepsilon_1 = 0.3$ and $\varepsilon_2 = 2.5$, 0 if the angle between them is $\leq \varepsilon_1$ and proportional to e^{angle} if the angle is $\geq \varepsilon_2$.

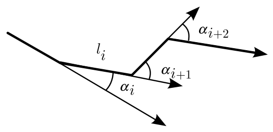


Figure 10: The energy of the polyline approximating the Bézier spline is computed as a product of the sum of angle changes between the consecutive segments and the exponent of the sum of the segment lengths.

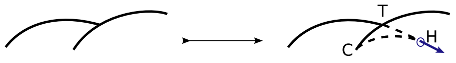


Figure 11: When we have a T-point and a cusp to match it to, we seek the location of a hidden cusp such that the two hidden contour parts joining our points to the hidden cusp have the highest probability.

When we want to join a T and a cusp (see figure 11), we ask the question "for all possible locations of a hidden cusp, and all possible tangent directions there, what are the probabilities of a C^1 random walk joining the T to this cusp and of another joining the visible cusp to the hidden cusp?" We treat the product of these probabilities as the probability of this point-and-direction being the hidden cusp. We posit that the ideal location of the hidden cusp is the one with the highest probability. Unfortunately, computing this probability directly by generating many random walks, etc., is impractical and slow. We therefore did this once, offline, and stored the precomputed probabilities in a table. That is to say, we placed one point at the origin, with a tangent ray along the positive x-axis; we generated many (ca. 10^8) C^1 random walks from there and recorded (in a discretized form) where each ended, and in which direction it was going. We did the same for a second point, situated on a unit circle, and with a given initial direction; we then found the point and direction that was most likely to be an endpoint of both sets of random walks. This was repeated for multiple points on the unit circle, and multiple initial directions, and the results stored in a table. This table, then, represents the function $(P,\phi) = h(Q,\theta) =$ the location and direction of the hidden cusp when one point is at the origin and has horizontal tangent, and the other is at Q and has tangent direction θ. Given an actual tee and cusp, we can translate one to the origin and rotate so that its tangent is in the positive-x direction; we then use the other point's location and direction to look up an answer in the table. (Note that this assumes that the optimal answer is scale-invariant, in that the second point may not be at a unit distance from the first, and we must scale this distance to one in order to use our table.) We connect the hidden cusp to the tee point and the visible cusp with Bezier curves, and compute E_{curve} for their union as described above.

Greedy search for the best configuration. After a likelihood for each pair of endpoints is computed, we need to match up pairs to find the best total *configuration* (a configuration consists of endpoint pairs, where each endpoint appears in only one pair). For instance, if we have 4 endpoints numbered 1 to 4, the possible configurations are: $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$ and $\{(1,4),(2,3)\}$. The likelihood of a configuration is defined as the product of likelihoods of its pairs. It is not practical to compute the likelihoods of all possible configurations as the number of them grows exponentially in the number of tees and cusps. Instead, we do a greedy search similar to [Nitzberg et al. 1993]; starting with several best pairs, for each of them we choose the next best pairs from the set of valid configurations, and so on; we keep track of the 10 best configurations at any time.

4.2.1 Limitations of the figural completion algorithm

The figural completion approach that we presented has a number of limitations.

The location of the hidden cusp provided by the method above may be unsatisfactory. Indeed, in the bean-like case shown in figure 12, the hidden cusp is estimated to lie at a point that is not, in fact, hidden. Figure 12 (right) shows another example where the locations of hidden cusps are estimated incorrectly because of the failed assumption that the precomputed positions of hidden cusps are scale-invariant.

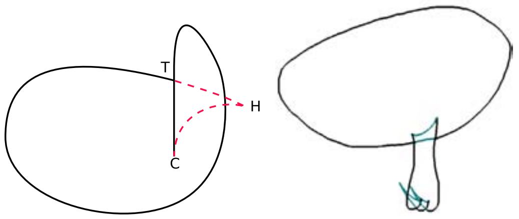


Figure 12: Problem cases: our method can produce a contour completion which places the hidden cusps in impossible locations. This happens because our method only considers local probabilities, and not the shape of the remainder of the visible contour.

Consider a dog's body with one leg on the left hand side, seen from the right hand side (see figure 13). This is a case that our contour-completion algorithm cannot handle. The "completion" of the obscured contours consists of two hidden cusps connected by a U-shaped hidden contour, and two "straight" segments connecting the hidden cusps to two t-points. In the two-hidden-cusp completion, the location of the two cusps is ambiguous. The algorithm for finding a hidden cusp for a t-point/visible-cusp pair will not work for this case, because there are no visible cusps.

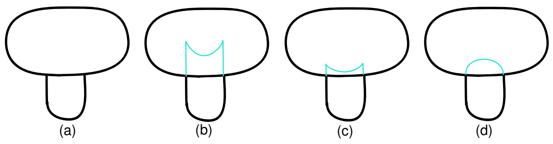


Figure 13: (a) The back-leg drawing case; (b),(c), and (d) show possible completions; our system would produce completion (d), but would fail at later stages (although if we allowed multiple-component surfaces, then (d) could be created).

The figural completion for this case could equally well consist of just an arc joining the two t-points — there's no a priori reason for the system to assume that the shape being drawn has only one connected component. Without the context (knowing that this is a leg), we do not know of any principled algorithm to guess the locations of the hidden cusps.

The back legs issue is something that can be handled pretty easily by adding gestures to our system (where, say, a user could change the view and draw a stroke corresponding to the back leg), or, by incorporating our system as an inflation component into one of the existing free-form sketch-based interfaces like Teddy. Such a system would also ideally include the ability to sketch or edit the hidden contours (i.e., provide user-guidance to the optimization algorithm).

Our contour-completion algorithm, based on the table-lookup, should probably be improved. We would like to find a good approximation to the data in the table so that a lookup (and the computation and storage of the table) is unnecessary; if such a function were based on a deep understanding, the unprincipled “scale-invariance” assumption might also be eliminated.

Given that figural completion is an expensive search problem, it becomes slower for a large number of tee/cusps (say, more than 15) and is more likely to make incorrect inferences as the drawing gets more and more complicated (we find only an approximate solution to the optimization problem to make it tractable; as a result, the approximate minimum is not always the global minimum).

We have shown that a large class of visible contour drawings admit completions; it is known that others do not. But we do not know a complete characterization of which drawings admit completions.

5 From drawing to topological embedding

At this point we have a completed contour drawing, with a Huffman-labeling. This drawing consists of directed arcs (which we call *edges*) between *vertices* corresponding to T-points and cusps. The drawing partitions the plane into regions; Williams' paneling construction tells how to take a disjoint union of multiple copies of these regions and identify edges in pairs to produce an abstract manifold. Williams' description misses one subtlety: the regions he is identifying must be closed sets so that they contain their boundary points, for it is boundary points that are identified in pairs. For regions like the large region of the bean, the “crossing point” at the top must be counted twice – once as a point on the left half of the contour, and once as a point of the right half, or the object resulting from the identification of edges will not in fact be a manifold. This can be addressed by examining the boundary of each region for self-intersections and, if any are present, subdividing the region into two smaller regions; the details are finicky but not difficult. We'll simply treat that crossing point as two ever-so-slightly-separated points for the purpose of this explanation.

In our implementation, each copy of the region at this point is a 2D mesh created by triangulating the boundary of the region. We use Triangle [Shewchuk 1996] which performs the constrained Delaunay triangulation algorithm on the given boundaries of the regions.

We consider (see figures 14, 15) the disjoint copies of a region R as being of the form $R_i = R \times \{i\}$, where the index i never appears more than once in all copies of all regions. A typical identification in Williams' scheme is then that the point (r,i) is identified with (r,j), where r is a point on the boundary of region R, and (r,i) and (r,j) lie in R_i and R_j respectively; another might be that (r,i) is identified with (s,j), where R and S are adjacent regions in the plane both containing the point $r=s$ on their boundaries, $(r,i) \in R_i$ and $(s,j) \in S_j$. The disjoint union of all the copies of all the regions will be called U; there's a natural map $\pi : U \to \Re^2 : (r,i) \mapsto r$ in which the multiple copies of any point r are all mapped to r.

For a point P in the plane, the set $\pi^{-1}(P)$ is a set of points of the form (P,i); we call this the “stack over P.” Similarly, we can consider the stack of edges over an edge in the plane, or the stack of panels over a panel in the plane. If an edge e in the plane goes from P to Q, we write $\partial e = (P,Q)$ to denote that the boundary of edge e consists of the points P and Q, in that order. If e_i is an edge in the stack over e, then $\partial e_i = (P_i, Q_i)$ as well.

Williams identifies certain panel edges in pairs (see figure 15), that is, for certain i and j, he declares that e_i is to be identified with e_j, which means that the point $(x,i) \in e_i$, is identified with the point $(x,j) \in e_j$. This identification induces an identification on the stacks above vertices: if e_i is identified with e_j, and $\partial e = (P,Q)$, we

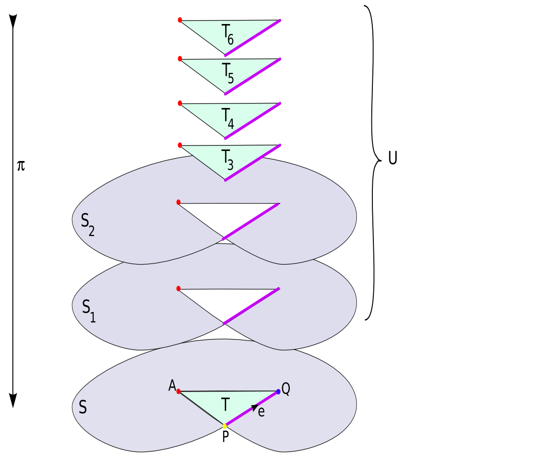


Figure 14: Schematic view of the disjoint union of panels that are glued to form the topological manifold homeomorphic to the bean. Each copy of each panel lies in a different layer; the union of all these copies is called U. The map π is “projection back to $\Re^2$ along z.” The collection of all points that project to A (the red dots) is called the “stack above A”. The magenta edges are the stack above the edge e. Each panel is indexed by its height in z, so all panels have different indices.

declare $P_i \sim P_j$, and $Q_i \sim Q_j$. The transitive closure of the relation $\sim$ partitions stacks into equivalence classes that we call *clusters*; each cluster in each stack corresponds to a vertex in Williams' surface, which we'll eventually embed.

Ordering the clusters. Williams' construction gives a depth order to the panels in each panel-stack; this order is generally unrelated to the indices above. This order induces an order on the clusters as follows: if P_i and P_j are in two clusters, and R is a region containing $P = \pi(P_i) = \pi(P_j)$ consider all the faces in the stack over R that are adjacent to vertices in the first cluster, and all those adjacent to vertices in the second cluster. By Williams' construction, faces in the first group will either be all in front of or all behind the faces in the second group; we say that the first cluster is in front of or behind the second group accordingly. Again by construction, this order is independent of the adjacent region R that we choose.

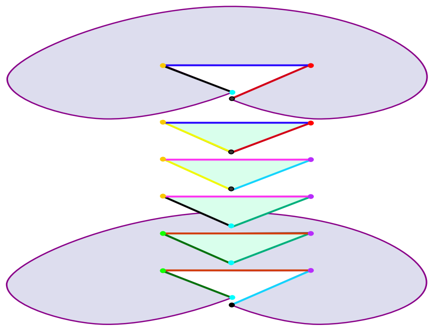


Figure 15: The panels, re-ordered for visibility; edges with the same colors are identified. This identifies *clusters* of vertices in each stack; vertices with the same color form a cluster. Note that the near vertex in the two large panels has been split into two copies.

Extra vertices. One important issue remains: if two edges e and e' in the same edge-stack have the same clusters as their endpoints but are *not* identified in the topological manifold, these distinct edges would be assigned the same depth in the constructed surface, which would result in a non-embedding. Figure 16 shows two such edges in the lower portion of the leg case. In such cases, we add a new vertex at the midpoint of each of the edges e and e' of the contour (and to any other edges that are identified with these). The stacks and the clusters within these stacks are then created for these newly-inserted points in the same way we described above.

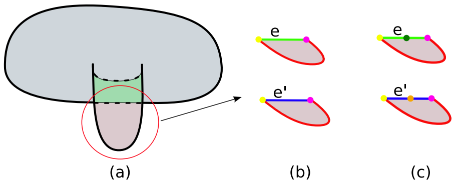


Figure 16: (a) A contour-completed drawing of a leg attached to a body, with panels colored. (b) The two panels for the bottom of the leg, colored to show edge identifications and vertex clusters. Note that the top edges e and e' share endpoints but are not identified. (c) We add mid-edge vertices, sort, and cluster them as before.

5.1 Constructing a topological embedding

We now present a novel algorithm that constructs a topological embedding from Williams' abstract manifold.

Embedding vertices To each cluster of the vertex stack over a vertex P, we associate a vertex whose xy-coordinates are those of P, and whose z coordinate is yet to be determined (we call these *cluster vertices*). We determine the z-placements using a mass-spring system. Suppose that the vertices corresponding to the clusters of one stack are X_α, where α ranges over the clusters. If cluster α is behind cluster β, we want the z-coordinate, z_α of X_α to be less than that of X_β. For each such order-relation between two of the Xs, we attach a spring whose rest-length z_0 is one, and for which the spring force follows the rule

$$F(d) = \begin{cases} 0 & d \geq 1, \\ Ce^{1-d} & d < 1 \end{cases}$$

which ensures that if the z-order is inverted, there's a substantial force pushing back towards the proper ordering.

This ordering and set of z-values could also be found by simply sorting the vertices; we use the mass-spring system as a way to relate the z-depths for vertices in separate stacks. In particular, if P and Q are distinct vertices joined by an edge e, then each cluster over P is joined to one or more clusters over Q by edges in the stack over e. For each such connection, we add a spring with rest-length zero between the corresponding cluster-vertices; we use a sufficiently small spring constant that the intra-stack ordering is not disturbed. Our goal is to make each edge want to be somewhat parallel to the $z = 0$ plane, rather than having vertices associated with one stack be far in front of all others, for instance.

The mass-spring system acts on the points, which are constrained to move only in z. Clearly if the spring constant for the inter-stack springs is small enough, each stack will be ordered correctly. In our implementation, we use the constant 1.3, which seems to perform well on examples like the ones shown in this paper and the associated video. The points of the drawings in our system lie in the bounding box of -1.0 to 1.0 in each direction.

Embedding edges Having embedded the cluster vertices (i.e., the vertices of the manifold that Williams constructs), we can extend the embedding to edges by linearly interpolating depth along each edge. The ordering of edges in Williams' construction is generally sufficient to show that if e_i and e_j are distinct edges of the manifold corresponding to contour edge e, then they do not intersect except, perhaps, at endpoints which they share. In the event that e_i and e_j share *both* their endpoints, linear interpolation would assign them the same depths at all points, and our mapping would not be an embedding. Fortunately, the "extra vertices" step above inserts points exactly when necessary to prevent this; thus we have an embedding of both the vertices and the edges of Williams' manifold.

Embedding faces We extend the embedding over the panel interiors using Poisson's formula to find a harmonic function on the panel whose values on the boundary are the given depth values that we've already assigned to the edges of the panel. Each interior point is assigned a depth that is a weighted average of the depths of points on the boundary edges. To prove that two panel interiors in the same stack never intersect, suppose that P is a point of some panel R, and that X and Y are points in the panel-stack over R, and that $\pi(X) = \pi(Y) = P$. Suppose that the panel to which X belongs, R_i, is in front of the panel to which Y belongs, R_j, so that the z-value for X should be larger than the z-value for Y. Then points on the edges of R_i are in front of (or equal to) the corresponding points on the edges of R_j. The z-coordinates of corresponding points cannot all be equal unless the boundaries of R_i and R_j are identical, in which case the union of R_i and R_j is a spherical connected component of the manifold, and is handled as a special case. In the remaining cases, since the z-values for R_i are greater than or equal to the corresponding values for R_j, and the z value for X is a weighted sum of these values *with all nonzero weights*, and the z-values for Y is the corresponding weighted sum of the other z-values, with the same weights, we find that the z value for X is strictly greater than that for Y. Thus the interiors of faces do not intersect. We have thus constructed a continuous 1-1 map from Williams' abstract manifold into $\Re^3$, i.e., a topological embedding.

6 Smoothing the embedding

Now that mesh vertices corresponding to each panel have been assigned depths, we "stitch" the meshes of individual panels into a single mesh. We start with the first panel, and stitch panels to it one at a time. If two panels are identified along an edge e, we alter the vertex indices on second to match those of the first. The edge correspondences for the stitched panel (excluding the edge we stitched along) come from the correspondence information of the two component panels. Although the resulting stitched mesh has the proper "contour projection" (for an appropriately modified definition of "contour"), its shape is generally unsatisfactory, as can be seen in the accompanying video. We therefore perform several optimization steps. During these steps, we constrain the vertices lying on the visible silhouette to remain on the silhouette so that the contours will match the drawing. That is, these vertices can only move in z, while others may move in x, y, and z.

First, we remesh the model using the algorithm proposed in [Kobbelt et al. 2000] in order to create more regular triangles, as the behavior of the mass-spring system is sensitive to the quality of the triangulation. Then, ten iterations of Taubin's λ/μ smoothing [1995] are applied to the mesh.

At this point, the mesh is smoother, but rather flat and sharp along the edges (because the silhouette constraints have not been incorporated smoothly). The next goal is to "inflate" the model, making it more rounded. To achieve this, we construct a mass-spring system on the initial mesh, with masses at the vertices and with two types of springs: length springs and what we call "pressure force" springs. The length springs try to keep the length of each edge as close to zero as possible, while the "pressure springs" simply push each triangle outward along its normal with a force proportional to the area of the triangle. We relax this mass-spring system and although the convergence in general is not guaranteed, in practice it converges quite fast. A model like the ones shown in the paper inflates in several seconds on an AMD Athlon 64 3000+ processor.

Our mass-spring approach has several drawbacks; some of them common to all mass-spring systems, others are particular to our choice of springs. First, most mass-spring systems approximate the physics of deformable models very crudely. Further, in our case, even the underlying "physical" model is quite *ad hoc*. We intuitively think of the current model as inflating the initial flat shape as a balloon, but with the restriction on the movement of silhouette points and disregard for surface curvature, it is a very weak analogy.

Secondly, our mass-spring system has several tuning constants that have to be chosen so that they work for most of the examples user draws. Thirdly, there is currently no mechanism in the system to prevent self-penetrations of the surfaces. We would like to address this issue in the future. Sometimes, though, we would like to allow self-penetrations: think of a body-with-two-legs example; there, the legs being slightly pushed inside the body is often more desirable than having them stick out far away from the body. Finally, there is a known problem of stiffness [Gibson and Mirtich 1997] with all mass-spring systems that leads to their potential instability.

Having said all this, the mass-spring system we created seems to work reasonably well on most examples. The results of the relaxation of the mass-spring system are satisfactory except at the areas that were completely flat and skinny initially (like the tips of the legs). Finally, we may choose to apply a few iterations of Taubin's anisotropic smoothing [Taubin 2001], which first filters the normals using λ/μ algorithm, and then filters the vertices, integrating the new normal field in the least squares sense. The final results are shown in figure 17.

7 Discussion, Limitations, Conclusions

Our system creates 3D shapes for a wide class of contour drawings; certain limitations prevent it from working universally. One is that the contour-completion approach is local–the completed contour shape depends on the geometry of the starting and ending points, but ignores the remainder of the input shape; it will require a much deeper understanding of contour completion to address this.

Williams' topological manifold construction, followed by our lifting, creates a mesh embedding with "folds" matching the drawn contour. But mesh contours and smooth contours are different. In particular, the curvature of a smooth contour at a cusp goes to infinity, which a mesh-cusp simply projects to some non-zero angle in the contour-drawing. The problem of exactly fitting the drawing is therefore generally impossible, unless users respect the conditions on curvature at cusps. We need to develop a means to characterize when we have adequately approximated the user's drawing.

Our inflation algorithm currently requires tuning constants; the constants that produce the most satisfactory-looking results actually produce self-intersecting surfaces, especially in locations like "armpits" (i.e., between a limb and a body). We would like to find an algorithm that produces embeddings instead. A more principled approach would optimize something about expected shapes of the inflated surface, conditioned on the known shapes of the visible contours, but lacking a prior distribution on all smooth surface shapes, such an approach seems intractable. We anticipate that a minimization of some fairness functional might hold promise.

We would also like to extend our work to include minor surface discontinuities—things like ridges or creases on a surface, which often are perceptually significant (and indeed, in cases like armpits, creased shapes are what a user might want to create).

Finally, although we have developed our system to be agnostic about shape, treating it purely geometrically, users *are* familiar with many shapes. We imagine the possibility of a hybrid system, in which the user's sketch is both inflated *and* matched against a large database of known forms, for possible suggestions ("You seem to be drawing a dog; would you like us to add the hidden legs for you?"). The problems of searching such a database and forming reliable hypotheses, however, seems daunting.

Acknowledgements

We would like to thank Prabhat, Gabriel Taubin, Anatoly Karpenko, Tomer Moscovich and Peter Sibley for many useful suggestions and helpful discussions, and the reviewers for their feedback. This work was partly supported by a gift from Pixar Animation Studios.

References

ALEXE, A., BARTHE, L., CANI, M.-P., AND GAILDRAT, V. 2005. Shape modeling by sketching using convolution surfaces. In *Pacific Graphics*, Short paper.

BOURGUIGNON, D., CANI, M.-P., AND DRETTAKIS, G. 2001. Drawing for illustration and annotation in 3D. *Computer Graphics Forum 20*, 3, 114–122.

GIBSON, S., AND MIRTICH, B. 1997. A survey of deformable modeling in computer graphics. Tech. Rep. TR-97-19, Mitsubishi Electric Research Lab., Cambridge, MA.

GRENANDER, U. 1981. *Lectures in Pattern Theory*, vol. 1-3. Springer-Verlag.

GRIFFITHS, H. B. 1981. *Surfaces*. Cambridge University Press.

GUILLEMIN, V., AND POLLACK, A. 1974. *Differential Topology*. Prentice Hall.

HOFFMAN, D. D. 2000. *Visual Intelligence: How We Create What We See*. W. W. Norton.

HUFFMAN, D. A. 1971. Impossible objects as nonsense sentences. In *Machine Intelligence 6*, B. Meltzer and D. Michie, Eds. American Elsevier Publishing Co., New York.

IGARASHI, T., AND HUGHES, J. F. 2003. Smooth meshes for sketch-based freeform modeling. In *Symposium on Interactive 3D Graphics*, 139–142.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3D freeform design. In *Proceedings of SIGGRAPH 99*, 409–416.

JOHNSTON, S. F. 2002. Lumo: illumination for cel animation. In *Proceedings of the Symposium on Non-photorealistic animation and rendering*, 45–52.

KANIZSA, G. 1979. *Organization in vision*. Praeger, New York.

KARPENKO, O., AND HUGHES, J. 2005. Inferring 3d free-form shapes from contour drawings. In *Siggraph 2005 Sketches Program*.

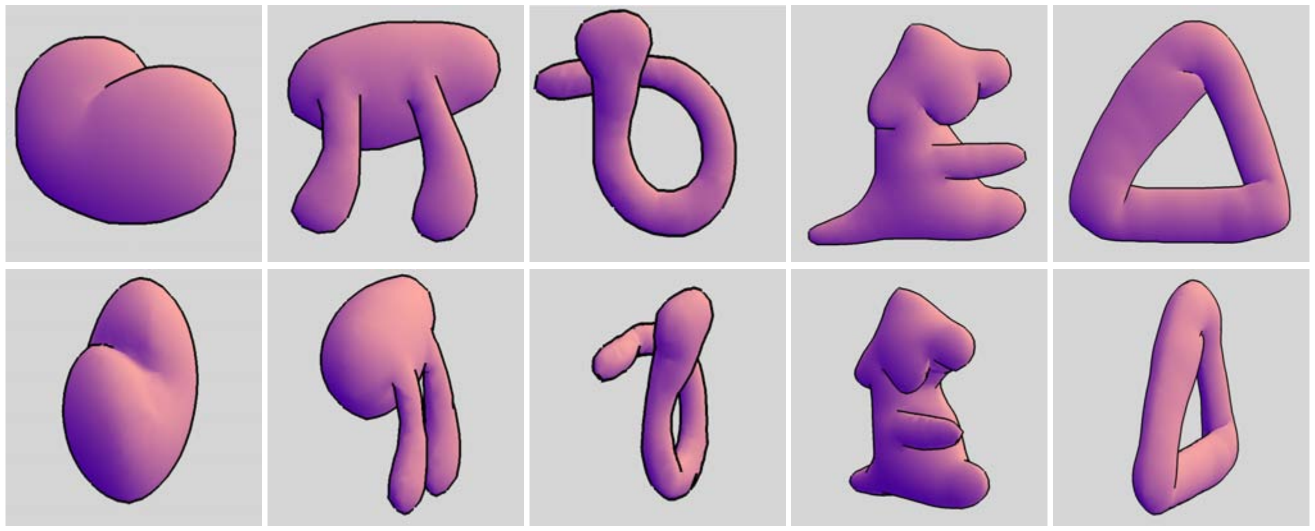


Figure 17: The examples of the shapes created with our system from user drawings. The top row shows the shapes from the sketching viewpoint, and the bottom row shows them from a different view.

KARPENKO, O., HUGHES, J., AND RASKAR, R. 2002. Free-form sketching with variational implicit surfaces. In *Eurographics Computer Graphics Forum*, vol. 21/3, 585–594.

KHO, Y., AND GARLAND, M. 2005. Sketching mesh deformations. In *Symposium on Interactive 3D Graphics and Games 2005*, 147–154.

KOBBELT, L. P., BAREUTHER, T., AND SEIDEL, H.-P. 2000. Multiresolution shape deformations for meshes with dynamic vertex connectivity. *Computer Graphics Forum 19*, 3.

KOENDERINK, J. J. 1990. *Solid Shape*. MIT Press.

LIPSON, H., AND SHPITALNI, M. 1997. Conceptual design and analysis by sketching. In *AIEDAM-97*, vol. 14, 391–401.

MUMFORD, D. 1994. Elastica and computer vision. In *Algebraic Geometry and Its Applications*, C. L. Bajaj, Ed. Springer-Verlag New York Inc.

NEALEN, A., SORKINE, O., ALEXA, M., AND COHEN-OR, D. 2005. A sketch-based interface for detail-preserving mesh editing. *ACM SIGGRAPH Transactions on Graphics*, 1142–1147.

NITZBERG, M., MUMFORD, D., AND SHIOTA, T. 1993. *Filtering, Segmentation, and Depth*. Springer-Verlag.

PENTLAND, A., AND KUO, J. 1989. The artist at the interface. Tech. Rep. 114, MIT Media Lab.

PEREIRA, J. P., BRANCO, V. A., JORGE, J. A., SILVA, N. F., CARDOSO, T. D., AND FERREIRA, F. N. 2004. Cascading recognizers for ambiguous calligraphic interaction. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*.

SCHMIDT, R., WYVILL, B., SOUSA, M. C., AND JORGE, J. A. 2005. Shapeshop: Sketch-based solid modeling with blobtrees. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 53–62.

SHESH, A., AND CHEN, B. 2004. Smartpaper: An interactive and user-friendly sketching system. In *Eurographics Computer Graphics Forum*, vol. 23/3, 301–310.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148. May.

SKETCHUP. SketchUp software: 3D sketching software for the conceptual phases of design. http://www.sketchup.com.

TAUBIN, G. 1995. A signal processing approach to fair surface design. *Computer Graphics 29*, Annual Conference Series, 351–358.

TAUBIN, G. 2001. Linear anisotropic mesh filtering. Tech. Rep. RC-22213, IBM Research.

TOLBA, O., DORSEY, J., AND MCMILLAN, L. 2001. A projective drawing system. In *2001 ACM Symposium on Interactive 3D Graphics*, 25–34.

ULUPINAR, F., AND NEVATIA, R. 1993. Perception of 3D surfaces from 2D contours. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15/1, 3–18.

ULUPINAR, F., AND NEVATIA, R. 1995. Shape from contour: Straight homogeneous generalized cylinders and constant section generalized cylinders. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17/2, 120–135.

WILLIAMS, L. R., AND JACOBS, D. W. 1997. Stochastic Completion Fields: A Neural Model of Illusory Contour Shape and Salience. In *Neural Computation*, vol. 9/4, 837–858.

WILLIAMS, L. R. 1994. *Perceptual Completion of Occluded Surfaces*. PhD thesis, University of Massachusetts.

WILLIAMS, L. R. 1997. Topological reconstruction of a smooth manifold-solid from its occluding contour. *Intl. Journal of Computer Vision 23*, 1, 93–108.

WITKIN, A. P. 1980. *Shape from contour*. PhD thesis, MIT.

ZELEZNIK, R. C., HERNDON, K., AND HUGHES, J. 1996. Sketch: An Interface for Sketching 3D Scenes. In *Proceedings of SIGGRAPH 96*, 163–170.

Sketching Mesh Deformations

Youngihn Kho[*] Michael Garland[†]

University of Illinois at Urbana-Champaign

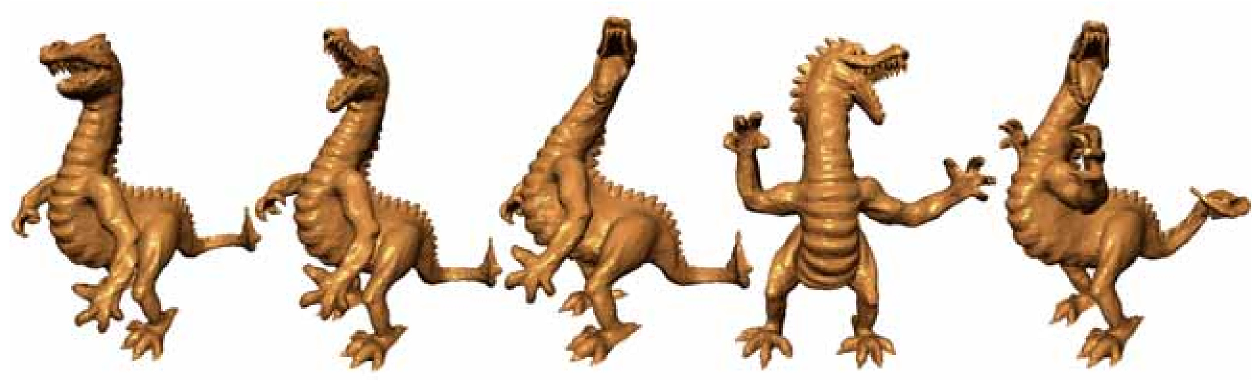


Figure 1: Step-by-step editing of a dragon character in under 3 minutes using our system. Each step represents 1–3 individual deformations.

Abstract

Techniques for interactive deformation of unstructured polygon meshes are of fundamental importance to a host of applications. Most traditional approaches to this problem have emphasized precise control over the deformation being made. However, they are often cumbersome and unintuitive for non-expert users.

In this paper, we present an interactive system for deforming unstructured polygon meshes that is very easy to use. The user interacts with the system by sketching curves in the image plane. A single stroke can define a free-form skeleton and the region of the model to be deformed. By sketching the desired deformation of this reference curve, the user can implicitly and intuitively control the deformation of an entire region of the surface. At the same time, the reference curve also provides a basis for controlling additional parameters, such as twist and scaling. We demonstrate that our system can be used to interactively edit a variety of unstructured mesh models with very little effort. We also show that our formulation of the deformation provides a natural way to interpolate between character poses, allowing generation of simple key framed animations.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Transformations I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques

Keywords: interactive mesh deformation, sketch-based editing, intuitive interfaces

[*]e-mail: kho@uiuc.edu
[†]e-mail: garland@uiuc.edu

1 Introduction

While many techniques for geometric mesh deformation have been developed, finding effective interactive techniques is still a challenging topic in modeling and animation. Observing that direct interaction in 3-D space is often a confusing task for non-expert users, we propose an intuitive interface for mesh deformation by sketching curves in the image plane. Our system allows a user to easily apply a broad range of deformations to unstructured polygon meshes.

In our system, users sketch a reference curve in the image plane both to determine a region of interest and to serve as a means of controlling an individual deformation. By sketching a second curve indicating the desired deformation of the reference curve, users can easily achieve the deformation of the entire region of interest specified by the reference curve. By constructing a mapping of the region of interest onto the reference curve, our system also provides a simple method for controlling additional parameters such as local twisting or scaling.

Our system provides a great deal of flexibility to the user. It can accept as input triangulated manifold meshes of any genus, containing any number of boundary loops. No further structural information about the object is required. The user can use free-form reference curves without being constrained by geometrically defined skeletal structures. The deformation curves can be significantly different than the "natural" skeleton of the surface, and work well even when such skeleton structures would be ill-defined.

By using a sketch-based screen space interface, we avoid the need for complex 3-D interactions that can be cumbersome for non-expert users. Users can achieve relatively complex deformations by simply drawing two strokes on the screen. Furthermore, we can use these sketch-based deformations to achieve a natural interpolation between character poses, thus producing simple key framed animations.

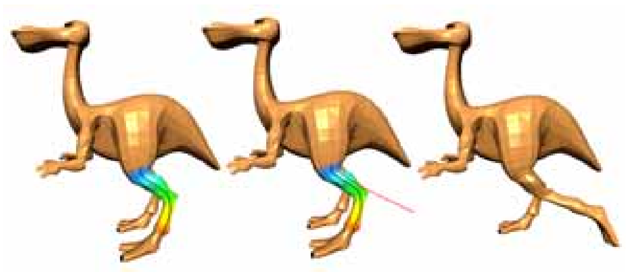


Figure 2: A simple sketch-based mesh deformation. The user draws a reference curve along the leg, followed by a second target curve. This induces a deformation of the leg itself.

2 Related Work

There has been a great deal of work done in the past on developing techniques for the modeling and deformation of geometric objects. Here we survey only the most relevant work, with an emphasis on techniques for interactive deformation.

Free-Form Deformations (FFD) are one of the most important techniques for deforming surfaces [Sederberg and Parry 1986; Coquillart 1990; MacCracken and Joy 1996]. While their complex control lattices provide very precise control over the resulting deformation, editing these lattices can be an unintuitive and time-consuming process. Handle-manipulation approaches [Kobbelt et al. 1998; Bendels and Klein 2003; Yu et al. 2004] are also popular and can produce pleasingly smooth deformations. However, the range of deformations that can be produced with a single handle manipulation is generally quite limited. Therefore a user must perform a sequence of several individual step to achieve a more complex result.

Curve-based deformation approaches such as Wires [Singh and Fiume 1998] or medial-based shape deformations [Bloomenthal 2002; Yoshizawa et al. 2003] have received considerable attention in recent years. Curves or skeletons in this type of methods can provide a natural means of capturing the structure of surfaces. Thus, deforming surfaces by editing those entities provides a good means of achieving large scale deformations. For this reason, our system uses a curve-based approach with an emphasis on providing an extremely easy method for specifying free-form control curves.

There has been substantial interest of late in developing intuitive interactive techniques, such as deformation by painting over surfaces [Lawrence and Funkhouser 2003]. Sketch-based interfaces have emerged as one of the more popular approaches to building user-friendly deformation tools. In Teddy [Igarashi et al. 1999], users can create and edit objects by simply sketching strokes in the screen. The system also proposes a deformation method based on *warp* [Corrêa et al. 1998]. Recently, a sketching interface to FFDs has been developed [Hua and Qin 2003]. Here sketch strokes are used to manipulate scalar field embeded in 3-D space. In our system, we employ sketching both to specify and deform the regions of interest.

3 Overview

In our system, the user initiates a deformation by drawing a reference curve on the image plane. This curve implicitly defines a *region of interest* — that part of the surface which will be deformed. The user then applies the deformation either by sketching a new target shape for the reference curve or by directly manipulating a deformation parameter such as twist or scaling.

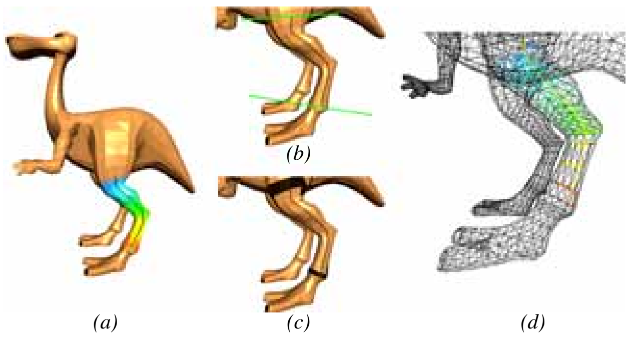


Figure 3: Preparing for deformation of the left leg. We compute two cutting planes (b) that will define the two loops bounding the region of interest (c). Each vertex in the region is mapped to the closest point on the reference curve (d).

Figure 2 shows a simple example of our system in action. The user begins by drawing a reference curve along the leg. The region of interest is highlighted with a red-to-blue color ramp. The user then draws a target curve indicating the desired deformation. From this pair of curves, the system automatically generates the deformation of the leg. A more complex editing session is shown in Figure 1. Each step in this editing sequence corresponds to 1–3 individual deformations.

4 Beginning a Deformation

The user begins the process of deformation by drawing a reference curve, which must be projected into the 3-D world space. From the reference curve, we implicitly recognize the region of the surface that the user wishes to deform. The user can optionally refine this region selection using an interactive partitioning scheme. Once the region of interest is identified, a "skinning" step associates each vertex within this region with the closest point on the 3-D reference curve. This basic process is illustrated in Figure 3.

4.1 Building the Reference Curve

We begin with a free-form sketch of the reference curve in the image plane. We represent the raw sketch curve as a collection of line segments taken directly from mouse events produced by the user's stroke. This raw curve is likely to be fairly noisy, especially when drawn with a mouse rather than a tablet device. Therefore, before proceeding, we smooth and regularize the raw sketch. We apply a simple averaging filter and simplify the polyline by merging neighboring segments so that each segment will be at least 5 pixels in length.

Having regularized the reference curve in the image plane, we must project it into the 3-D world space of the model. We first compute the point of intersection of a ray from the view point through the first point on the sketch curve. This hit point, along with the normal of the viewing plane, defines a plane in world space parallel to the image plane. We project the sketch curve onto this plane to compute the 3-D reference curve.

4.2 Recognizing the Region of Interest

After the user-drawn reference curve is mapped into 3-D space, we implicitly partition the model into three parts: (1) a static component, (2) the region of interest, and (3) a rigid component. The static component of the mesh will be unchanged by the deformation. The

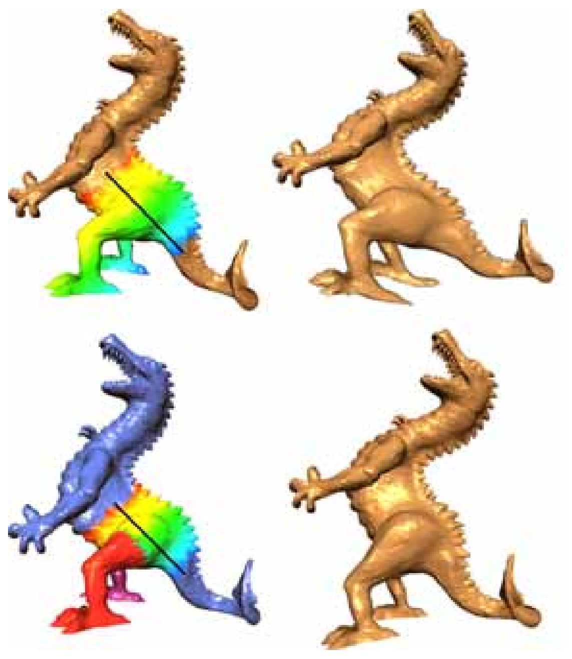


Figure 4: (Top) Deformation of the body without partitioning. The legs and feet are undesirably distorted. (Bottom) The two legs are partitioned so that they can be transformed rigidly, producing a more natural result.

region of interest is that part of the mesh to which the deformation will actually be applied. The rigid component will be transformed rigidly to maintain its connectivity with the region of interest. Figure 3 shows a simple example in which (1) the whole body beyond the upper thigh forms the static component, (2) the leg is the region of interest, and (3) the foot below the ankle is the rigid component.

The underlying assumption of our system is that the region of interest is the part of the surface "covered" by the user's sketch curve. At each end point of the curve, the system computes a cutting plane perpendicular to the reference curve (see Figure 3b). The intersection of these planes with the surface define triangle loops that partition the input mesh. We define these triangle loops using a graph cut formulation outlined in Section 4.2.1. In cases where a plane defines multiple intersection loops, we select the loop containing the nearest triangle to the end point. Having selected one boundary loop per cutting plane, we now have two triangle loops bounding the region of interest (as in Figure 3c). The vertices in each of the 3 regions are labelled by a breadth first style "flood fill". The static component will be the region bounded by the loop created by the starting point of the reference curve and the rigid component is bounded by its ending point.

In higher genus cases, a single loop may fail to cut the object into two parts. In such cases, multiple loops are required for the partition. We begin with the set of all loops defined by the cutting plane; these *must* collectively partition the object, as the plane itself does. We then consider each loop other than the initially one in succession. We remove a candidate loop from the cut only if does not merge the two components separated by the initially selected loop. This process eventually produces exactly two disjoint components.

Our implicit recognition scheme works well in many cases. However, additional explicit partitioning is useful in certain circumstances. For example, in Figure 4, when we lift up the back part of the dragon, we probably want to rigidly transform each leg, while the body is smoothly deformed. To do this, users can interactively augment (or override) the automatically generated partition.

4.2.1 Interactive Partitioning

In our system, we adopt a graph cut partitioning scheme controlled by the selection of cutting planes. The user draws a line to define a cutting plane containing that line and perpendicular to the view plane. In general, this cutting plane will not follow existing edges in the mesh but will cut across many triangles. Therefore, we apply a fuzzy decomposition technique [Katz and Tal 2003] to find the actual boundaries. We collect all triangles within a certain screen-space distance from the cutting plane — we typically adopt a 5 pixel distance limit. In general, the cutting plane might create multiple separated fuzzy regions. In this case, the system picks the region which is nearest to the view point. Then this set of triangles serves as a fuzzy region. To compute the boundary, a dual graph of the fuzzy region is created, where the weight of an edge in the dual graph is the dihedral angle of the corresponding primal edge multiplied by its length. Finally, a min-cut method on the dual graph produces a cut corresponding to the boundary. The method produces natural and smooth boundaries since the resulting cut follows relatively lower dihedral angles which are a good criterion for natural boundaries. The freely deformable region is bounded by only one partition. All other parts will be rigidly transformed.

4.3 Skinning and Parameterization

At this point, all the vertices in the region of interest have been collected. We skin the surface by associating each vertex with the closest point on the reference curve. Note that these closest points are simply required to be on the curve; they need not be vertices of the curve. Figure 3d shows an example of this association, connecting each vertex with its corresponding point on the curve.

We represent each point on the reference curve by its *normalized arc length s*. That is, for a given point we add up the length of all segments from the origin of the curve to the point. We normalize these values so that they range between 0 and 1. Thus $s = 0$ and $s = 1$ are, respectively, the origin and end points of the curve. This induces a parameterization of the region of interest onto the range $[0, 1]$. For each vertex $\mathbf{v}$ we have the normalized arc length parameter $s(\mathbf{v})$ corresponding to the associated point on the reference curve. This parameterization is shown by the color ramp on the leg in Figure 3a, with blue corresponding to $s = 0$ and red to $s = 1$.

As outlined earlier, one or more regions of the surface will be rigidly transformed. To assign the single transformation to each partition, we introduce a *virtual vertex* for each of them. The virtual vertex is placed in the center of the boundary between partitions and is associated with the closest point on the reference curve. The transformation of a virtual vertex determines the transformation of all vertices in its partition.

5 Deformation Techniques

In the previous section, we have discussed the preparation steps necessary to begin a deformation. The user draws a reference curve, which is filtered and projected into 3-D. The region of interest is determined, and each vertex within this region is mapped to a point on the 3-D reference curve. Once this initial phase is complete, the user can apply any one of the fundamental deformations described in this section.

5.1 Sketch-Based Mesh Deformation

The primary deformation that we support is accomplished by sketching. The user simply draws a new *target* curve, which we

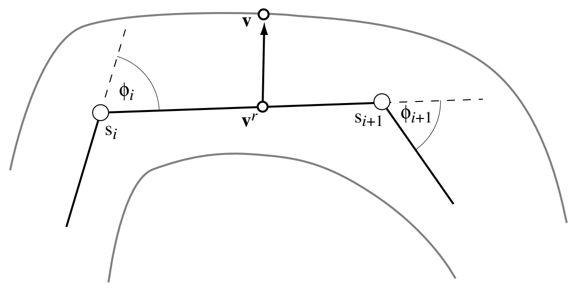


Figure 5: Illustration of sketch-based deformation. For each vertex $\mathbf{v}$ in the mesh, we compute the closest point $\mathbf{v}^r$ in the reference curve. The total turning angle for the vertex $\mathbf{v}$ is linearly interpolated in the line segment $\mathbf{v}^r$ lies on.

interpret as a deformation of the original *reference* curve. Our system then deforms the entire region of interest in an analogous way. This provides the user with intuitive and flexible control over the object's shape. Figure 2 shows a simple example of this style of deformation.

Recall that we have already assigned a value $s(\mathbf{v})$ to each vertex $\mathbf{v}$ which connects it to a corresponding point on the reference curve. We can thus think of the deformation process as follows. The reference curve is scaled and bent to match the target curve. The vertices of the mesh are connected to the reference curve through rigid iron wires, so they are translated and rotated along with the reference curve.

Preparation We begin by filtering and smoothing the target curve in the image plane, just as we did the reference curve. To prevent the mesh from tearing, we translate the target curve so that it's starting point is coincident with the starting point of the reference curve. We then map the target curve to the sketch plane constructed in Section 4.1. Consequently, the user is not allowed to change viewing parameters while drawing the two curves.

For each vertex $\mathbf{v}$ in the region of interest, we have a corresponding normalized arc length value $s(\mathbf{v})$ that provides us with the corresponding closest point $\mathbf{v}^r$ on the reference curve. Similarly, we use the same normalized arc length parameter to compute the corresponding point $\mathbf{v}^t$ on the target curve. This provides us with all the information necessary to compute to desired transformation at $\mathbf{v}$.

Computing Rotational Angles We deform the surface by rotating each vertex $\mathbf{v}$ about the corresponding reference point $\mathbf{v}^r$. The axis of rotation is simply the normal of the sketch plane and the rotational angle $\theta(\mathbf{v})$ is the signed angle between the tangents at $\mathbf{v}^r$ and $\mathbf{v}^t$. However, since our curves are sequences of line segments, we must interpolate rotational angles along the curve in order to avoid significant discontinuities.

To compute the rotational angle $\theta(\mathbf{v})$, we must first locate the line segment $[s_i, s_{i+1}]$ of the reference curve on which $\mathbf{v}^r$ lies. We define ϕ_i to be the signed exterior turning angle of the curve at node i (see Figure 5). We will define the tangent direction at the node s_i by the total turning angle Φ_i:

$$\Phi_i = \sum_{j=0}^{i-1} \phi_j + \frac{\phi_i}{2} \tag{1}$$

Note that we use the half-angle $\frac{\phi_i}{2}$ so that the tangent at s_i will be the average direction of the two incident line segments. This defines

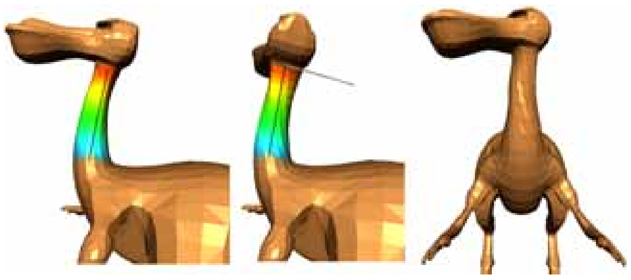

(a) (b) (c)

Figure 6: An example of twisting the neck of a dinopet model. (a) We first specify the approximate part of the neck to twist. (b) The neck is twisted by specifying the rotation axis (black) and the amount of angles (grey). Note that in this example, rotational angle is linear to the parametrization which is color ramp coded. (c) The result from a different view point.

the total turning angle at the nodes of the curve. For a point on the interior of a segment $[s_i, s_{i+1}]$ we interpolate the turning angle

$$\Phi(s) = \Phi_i + \frac{\phi_i}{2} b(2\alpha) + \frac{\phi_{i+1}}{2} b(2\alpha - 1) \tag{2}$$

where

$$\alpha = \frac{s - s_i}{s_{i+1} - s_i} \tag{3}$$

and

$$b(x) = \begin{cases} 1 & \text{if } x > 1, \\ x & \text{if } 0 \le x \le 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The blending function b is chosen so that $\Phi(\frac{s_i + s_{i+1}}{2}) = \Phi_i + \frac{\phi_i}{2}$. In other words, so that the tangent direction at the midpoint of the segment will be parallel to the segment.

We can compute these total turning angles for both the reference and target curves. They tell us the signed angle difference between the initial segment of the curves and the given points $\mathbf{v}^r$ and $\mathbf{v}^t$. We also need to account for the global rotation θ_g, which is the angle between the initial segments of the two curves. Our desired rotational angle is now simply

$$\theta(\mathbf{v}) = \Phi^r(s(\mathbf{v})) - \Phi^t(s(\mathbf{v})) + \theta_g \tag{5}$$

Once we have computed the target position and the desired rotational angle for a vertex, the final deformed position $\mathbf{v}'$ of the vertex $\mathbf{v}$ will be

$$\mathbf{v}' = T(\mathbf{v}^t) R(\theta(\mathbf{v})) T(-\mathbf{v}^r) \mathbf{v} \tag{6}$$

where T indicates translation and R indicates rotation about the normal of the sketch plane.

5.2 Twisting

We can also achieve twisting deformations by locally rotating the the region of interest about the reference curve. For example, consider the simple twisting operation shown in Figure 6. The reference curve is now the rotational axis and the user's second mouse stroke is used to control the amount of twisting being performed.

To use the reference curve as a rotational axis, it must be placed inside of the the model. To do this, we must alter the way in which we project the reference curve into world space. We first compute a set of *joints*. For each node in the reference curve in the image plane, a joint is defined as the average of hit points on the nearest front face and the back face by the ray from the viewpoint to the vertex. Consequently, we disallow twisting if any of nodes in the sketch curve has only one hit point. Now, the rotational axis is the curve connecting these joints. Note that this new curve looks the same from the user's perspective, as it still projects to the same image space sketch curve. As we did in sketch-based deformation, the rotational axis for the vertex $\mathbf{v}$ (i.e., the tangent direction at $s(\mathbf{v})$) is linearly blended.

We must now compute the rotational angle $\theta(\mathbf{v})$. Obviously using the same rotational angle at all vertices would not produce the desired result. We have found that the most natural twisting is achieved when the rotational angles $\theta(\mathbf{v})$ vary linearly with the normalized arc length $s(\mathbf{v})$. Thus, we compute a maximum angle $\theta_{\max}$ proportional to the length of the second line drawn by the user and use a rotational angle $\theta(\mathbf{v}) = s(\mathbf{v})\theta_{\max}$ at the vertex $\mathbf{v}$. The new position for vertex $\mathbf{v}$ will therefore be

$$\mathbf{v}' = T(\mathbf{v}^r)R(\mathbf{t}^r(\mathbf{v}), \theta(\mathbf{v}))T(-\mathbf{v}^r)\mathbf{v} \tag{7}$$

where $\mathbf{t}^r(\mathbf{v})$ is the interpolated tangent direction of the reference curve at $s(\mathbf{v})$.

5.3 Indirect Control Using Parameterization

The parameterization $s(\mathbf{v})$ that we have established to map the region of interest onto the reference and target curves also provides a natural mechanism for adjusting deformation parameters. We allow the the user to gain finer control over the deformation by using a standard spline control to specify modifications of the deformation parameters as a function of s. In this section, we briefly outline three such controls.

Adjusting Target Curve Turning Angles In our sketch-based deformation, the end result is obviously controlled by the shape of the target curve. By fine tuning the target curve, we can fine tune the deformation. This allows the user to draw a fairly simple base target curve and then interactively adjust its shape to achieve a specific intended deformation.

We control the shape of the target curve by adjusting the exterior turning angles ϕ_i, which were discussed in Section 5.1. By controlling these angles, we can radically alter the shape of the target curve.

We present to the user a standard spline box with which they can define an offset function $F(s)$, which we initialize to the identity function $F(s) = s$. For each vertex at position s_i along the target curve, we compute an adjusted turning angle ϕ_i' as:

$$\phi_i' = \phi_i + F(s_i) - s_i \tag{8}$$

and use these adjusted turning angle to compute an adjusted deformation.

An example of this type of deformation is shown in Figure 7. We begin by sketching a very simple deformation that achieves a coarse deformation of the overall shape. We then adjusted the rotational angles to achieve a final S-shaped shark.

Scaling Control The scaling control allows users to locally inflate or deflate the surface along the reference curves. This technique can be thought as an interactive version of generalized cylinders [Snyder and Kajiya 1992]. We achieve local scaling by controlling the magnitude of the offset vector $\mathbf{v} - \mathbf{v}^r$ of a vertex $\mathbf{v}$. We use the same skeleton curve connecting joints as in twisting for computing these offsets. The magnitude of the offset vectors is scaled by a function $F(s)$ which is initially the constant function $F(s) = 1$. We see a typical example in Figure 8. We have locally inflated and deflated the left Queen along a vertical reference curve using the spline function shown on the right.

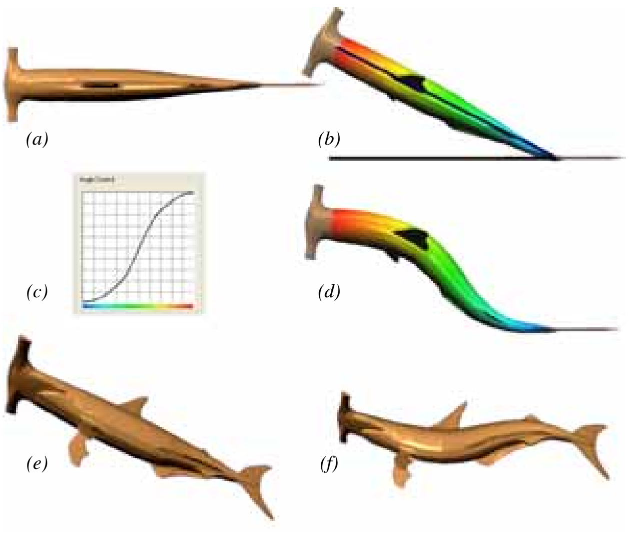


Figure 7: Adjusting target curve turning angles can fine tune the deformation. An initial coarse deformation (b) is adjusted using a spline control (c) to produce a more nuanced pose (d–f).

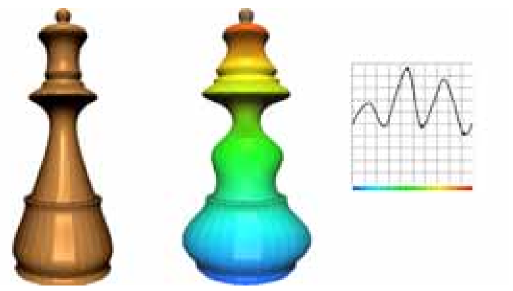


Figure 8: By locally scaling offset vectors, we can locally inflate and deflate the initial shape.

Rotational Angle Control for Twisting As discussed in Section 5.2, we linearly increase the twisting angle as a function of s. We can just as easily add an additional scaling adjustment to produce $\theta(\mathbf{v}) = F(s(\mathbf{v}))\theta_{\max}$. This allows the user to control the relative "speed" of the twist along the region of interest.

5.4 Increasing Smoothness

In most cases, our techniques produce pleasingly smooth deformations. However, local jaggedness can occur as the result of factors such as excessive noise in the user's sketch or the limited screen resolution. For these circumstances, we introduce a deformation optimization technique that can automatically smooth away such artifacts. It also provides a straightforward mechanism to blend the boundaries between rigidly transformed components and the freely deformable region.

Any one of the many general mesh smoothing algorithms could be applied to smooth the deformation. However, this would have the undesirable side-effect of removing actual small-scale *features* from the surface as well. Therefore, we seek to directly optimize deformation parameters to produce a smooth result. The central idea is neighboring vertices should undergo similar transformations to maintain smoothnes, while still remaining faithful to the user's

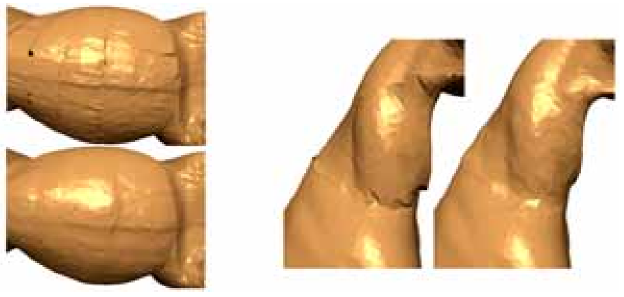


Figure 9: The benefits of automatically smoothing the deformation. (Left) Sketch-based deformation. The top picture is before the optimization, and the bottom picture is after the process. (Right) Twist deformation. The left one shows non-optimized twisting deformation, and the right one shows the result of the optimization.

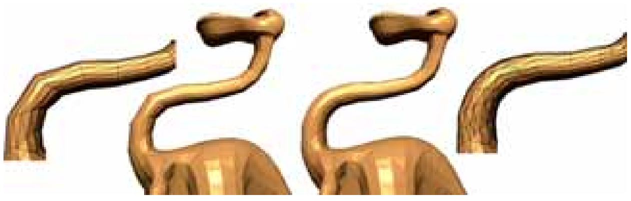


Figure 10: When performing significant deformations, the result can appear jagged if the input mesh is too coarse. Adaptive mesh refinement removes these artifacts.

specified transformation.

For each vertex $\mathbf{v}_i$ in the region of interest, let us consider its deformation parameter u_i. For sketch-based deformation, we would separately consider both rotational angles $\theta(\mathbf{v}_i)$ and the normalized arc length $s(\mathbf{v}_i)$. Starting from an initial set of parameters $(u_1,\dots,u_n)$ generated from the user's input, we wish to find a new set of parameters $(u'_1,\dots,u'_n)$ that minimizes the following energy:

$$E = \sum_i \left(\sum_{j \in N_i} w_{ij}(u'_i - u'_j)^2 \right) + w_c(u_i - u'_i)^2 \tag{9}$$

where N_i is the set of vertices adjacent to vertex $\mathbf{v}_i$.

The first term of this energy function provides a measure of the smoothness of the parameter u_i. We choose the edge weights $w_i j$ to be the inverse edge length $w_{ij} = \|\mathbf{v}_j - \mathbf{v}_i\|^{-1}$. The second term in the summation measures the deviation of the new parameters u'_i from the parameters u_i derived from the user's input. Empirically, we find that setting the weight w_c such that $\sum_{j \in N(i)} w_{ij} = 10 w_c$ provides a generally good balance between smoothness and this fidelity term.

We solve this optimization problem using a traditional Newton method. The initial values are simply those before the optimization, which makes the second cost term zero. Since our cost function is quadratic and the number of neighboring vertices are usually very small, the Hessian matrix is constant and sparse. Therefore, the optimization problem can be very efficiently computed by solving a linear system through one-time LU factorization. Furthermore, the initial values tend to be fairly close to the optimal solution, so we observe that the number of iterations necessary is generally very small.

The result of this automatic optimization process are shown in Figure 9. The fairly obvious artifacts in the unoptimized deformation are entirely removed following the optimization process.

5.5 Adaptive Mesh Refinement

If the user applies a significant deformation to the model, the resolution of the input mesh may be insufficient to support smooth deformation. The result will be a locally jagged deformation. To account for this, we propose a simple adaptive refinement scheme for smooth deformation. Our refinement primitive is the *edge split*. For each edge, we test two criteria to decide whether to split or not. We split the edge only if the two criteria are both satisfied.

The first criterion is edge *stretching*, defined as the ratio of the edge length in the original mesh to its length in the deformed mesh. We will consider splitting any edge whose ratio exceeds a specified limit. We also apply this criterion recursively to new edges produced by splitting previous edges. Experimentally, we find that a ratio of 1.5 works well as a threshold; here an edge will be split if it is stretched by more than 50%. This criterion will tend to refine more along the direction of deformation, but less in the orthogonal direction, as can be seen in Figure 10. In this example, we subdivide more along the direction in which the neck is stretched, but relatively less around the neck.

The second criterion is edge *curvature*. We only wish to split those edges that are sufficiently bent by the deformation. Each candidate edge in the original mesh is tentatively splitted by inserting a vertex at the midpoint of the edge. Then the inserted vertex is also transformed to a deformed position. If the angle between the two new edges in the deformed mesh is less than some threshold, we accept the split. Otherwise, the split is cancelled. This means that if the split edges are close to parallel, the split is not necessary. We have found that an angle threshold of $\pi - \frac{\pi}{24}$ produces good results. In Figure 10, the bent part in the lower neck is more refined, while the middle part is less refined.

Our refinement scheme is designed to minimize the number of faces added while maintaining the smoothness of the deformation. However, the resulting mesh might have many triangles of bad aspect ratio. If more nearly equilateral triangles are desired, then a regularization process that can perform additional operations such as edge flipping could be considered [Welch and Witkin 1994; Kobbelt et al. 2000].

6 Results

In this section, we consider several examples of using our system to edit unstructured polygon meshes. All results were generated by interactive editing on a standard consumer-level Windows PC. We render all sample models with flat shading in order to better highlight the structure of the surface mesh.

Figure 1 provides a step-by-step illustration of an editing session in which we repose a dragon character. We begin by opening the mouth, which requires only two sketch-based deformations or exactly 4 lines to be drawn by the user. We subsequently twisted both arms and the neck. We conclude by using two sketch-based deformations to bend the tail. The entire editing session — including program startup, loading the mesh from disk, and interactive editing — required less than 3 minutes.

In Figure 11 we see an example of using sketch-based deformation to bend an initial cylinder into multiple letter forms. Each letter was created by drawing a single reference curve on the cylinder followed by a single target curve in the shape of the intended letter. Even though the original cylinder has been stretched and bent fairly significantly, the deformed surface remains smooth. Note that, in order to keep the number of triangles fixed across all examples, we have not applied our adaptive refinement scheme in this case.

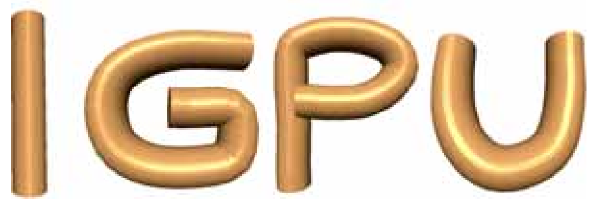


Figure 11: Applying significant deformations to bend a cylinder into various letter forms still results in smooth surfaces.

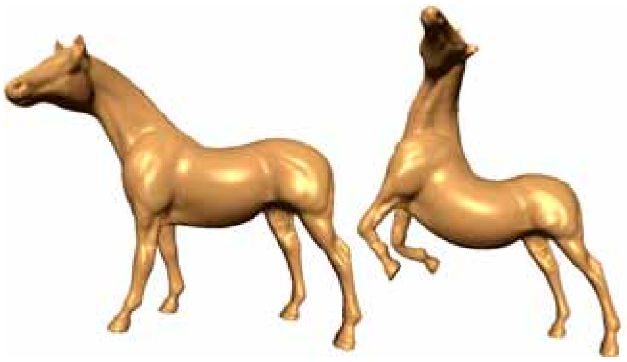


Figure 12: Reposing a horse with 4 simple deformations.

Figure 12 demonstrates the deformation of another more complex figure. We begin by interactively partitioning the front two legs. The body is then deformed by a single sketch-based deformation (requiring only 2 strokes). We conclude by twisting the neck, and bending the front legs by a single sketch-based deformation for each.

We can deform the hand shown in Figure 13 into a number of other poses quite easily. For the examples shown, we applied 1–2 sketch-based deformations to each of the fingers. Total modelling time is a mere 1 minute per hand pose. Note that the input is fairly smooth and the mesh is rather dense. Our system can still handle this mesh at interactive speeds and the deformed meshes are just as smooth as the input.

Figure 14 demonstrates a leg deformation using three different reference curves. We can clearly see that the system behaves well even when given quite different reference curves. The user is not constrained to draw a reference curve that follows the "natural" skeleton of the leg, and indeed can even draw a reference curve beyond the edge of the surface. All of these reference curves are sufficient to unfold the leg. We can also see that by drawing somewhat different reference curves, the user can easily excercise control over the nuances of the deformation result. For instance, the middle result bends the leg more rigidly than the others because the reference curve follows the shape of the leg less closely.

6.1 Skeleton Based Morphing

In our sketch-based deformation, the reference curves can be thought of as skeletons for the regions of interest. Utilizing these implicit skeletons, we can produce quite natural pose transitions that are much more pleasing than using simple linear interpolation (see Figure 15).

To morph using the skeletons at each intermediate frame, we must interpolate the curve from the reference and the target curves

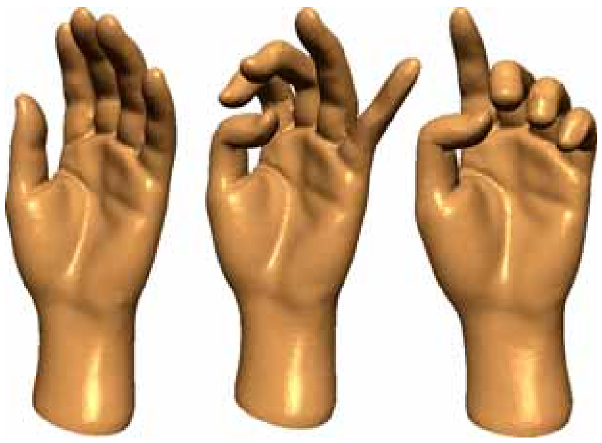


Figure 13: The original hand model (left) and two deformed hands.

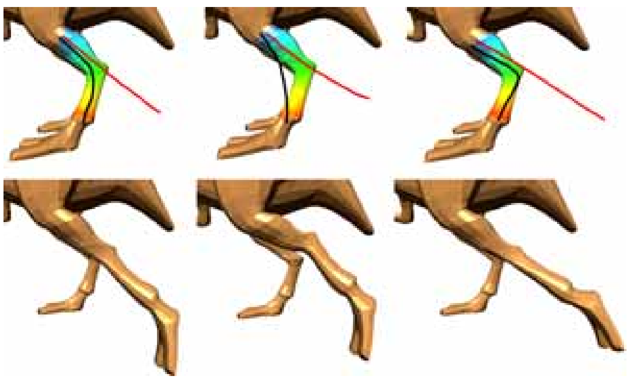


Figure 14: By using different reference curves, the user can produce subtly different results.

then deform the object according to the *interpolated curve*. The interpolated curve is computed automatically by linearly interpolating lengths and rotational angles. Its total length is linearly interpolated from the lengths of the reference and target curves. As defined in Section 5.1, there is a rotational angle $\theta(\mathbf{v})$ at each vertex of the reference curve that deforms it into the target curve. The rotational angles for the interpolated curve are simply $\alpha\theta(\mathbf{v})$ for $\alpha \in [0,1]$. This approach is similar to as-rigid-as-possible interpolation [Alexa et al. 2000], in the sense that we separate the rotation and the scale components.

Once the interpolated curve is computed, the vertex positions are computed in the same way as in the sketch-based deformation described in Section 5.1. The only difference is that we substitute the interpolated curve as the target curve.

7 Conclusion and Future Work

We have proposed a new and intuitive approach to interactive deformation of unstructured polygon meshes. Users of our system can make significant edits to 3-D objects by simply drawing a pair of curves on the image plane. The reference curves drawn by the user simultaneously partition the mesh, serve as a control handle for the deformation, and provide a scalar field that parameterizes the region of interest. This parameterization can be used to easily

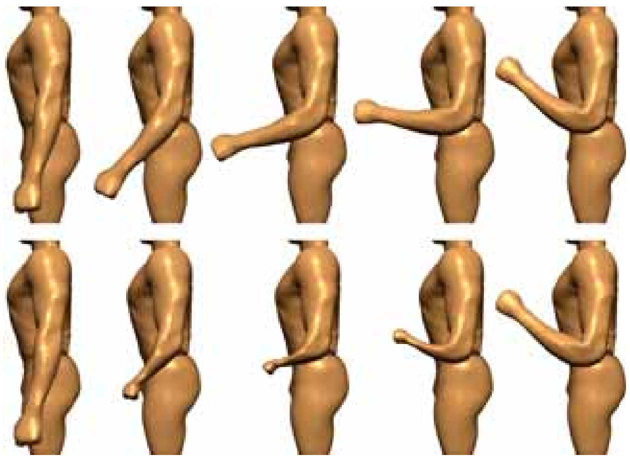


Figure 15: Interpolating our deformation parameters generates much more natural in-between frames than linear interpolation of vertex positions.

control additional deformation parameters such as twist and scaling. As our method does not use any fixed skeletal structure, users have great freedom in choosing the way in which the surface will be deformed. Since our system is based on simple 2-D sketching operations, it is both intuitive and easy to use, and allows users to quickly create deformed objects. It is also relatively appealing to non-expert users. Furthermore, our deformation method can provide for natural morphing between two key frames by using the sketch curves as implicit skeletons.

Our current system is a very effective tool, but there are also numerous ways in which it could be improved and extended. Our current approach is targeted more towards reposing bodies and limbs. Editing fine-grained surface features, such as the shape of the eyes on a face, is more difficult. Extending our sketch-based methodology to support editing of such surface features would be very desirable. Although our method is quite fast and suitable for interactive editing of fairly large models, the performance could be improved by incorporating multiresolution techniques such as [Lee et al. 2000]. Extending skeleton-based morphing into a complete system for easily creating simple key-framed animations is another very appealing direction.

References

ALEXA, M., COHEN-OR, D., AND LEVIN, D. 2000. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 157–164.

BENDELS, G. H., AND KLEIN, R. 2003. Mesh forging: Editing of 3d-meshes using implicitly defined occluders. In *Symposium on Geometry Processing 2003*.

BLOOMENTHAL, J. 2002. Medial-based vertex deformation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM Press, 147–151.

COQUILLART, S. 1990. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM Press, 187–196.

CORRÊA, W. T., JENSEN, R. J., THAYER, C. E., AND FINKELSTEIN, A. 1998. Texture mapping for cel animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 435–446.

HUA, J., AND QIN, H. 2003. Free-form deformations via sketching and manipulating scalar fields. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, ACM Press, 328–333.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3d freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 409–416.

KATZ, S., AND TAL, A. 2003. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Trans. Graph. 22*, 3, 954–961.

KOBBELT, L., CAMPAGNA, S., VORSATZ, J., AND SEIDEL, H.-P. 1998. Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 105–114.

KOBBELT, L. P., BAREUTHER, T., AND SEIDEL, H.-P. 2000. Multiresolution shape deformations for meshes with dynamic vertex connectivity. In *EUROGRAPHICS 2000*.

LAWRENCE, J., AND FUNKHOUSER, T. A. 2003. A painting interface for interactive surface deformations. In *Pacific conference on computer graphics and applications*.

LEE, A., MORETON, H., AND HOPPE, H. 2000. Displaced subdivision surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 85–94.

MACCRACKEN, R., AND JOY, K. I. 1996. Free-form deformations with lattices of arbitrary topology. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 181–188.

SEDERBERG, T. W., AND PARRY, S. R. 1986. Free-form deformation of solid geometric models. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM Press, 151–160.

SINGH, K., AND FIUME, E. 1998. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 405–414.

SNYDER, J. M., AND KAJIYA, J. T. 1992. Generative modeling: a symbolic system for geometric modeling. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM Press, 369–378.

WELCH, W., AND WITKIN, A. 1994. Free-form shape design using triangulated surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM Press, 247–256.

YOSHIZAWA, S., BELYAEV, A. G., AND SEIDEL, H.-P. 2003. Free-form skeleton-driven mesh deformations. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, ACM Press, 247–253.

YU, Y., ZHOU, K., XU, D., SHI, X., BAO, H., GUO, B., AND SHUM, H.-Y. 2004. Mesh editing with poisson-based gradient field manipulation. *ACM Trans. Graph. 23*, 3, 644–651.

A Sketch-Based Interface for Detail-Preserving Mesh Editing

Andrew Nealen, TU Darmstadt
Olga Sorkine, Tel Aviv University
Marc Alexa, TU Darmstadt
Daniel Cohen-Or, Tel Aviv University

Abstract

In this paper we present a method for the intuitive editing of surface meshes by means of view-dependent sketching. In most existing shape deformation work, editing is carried out by selecting and moving a *handle*, usually a set of vertices. Our system lets the user easily determine the handle, either by silhouette selection and cropping, or by sketching directly onto the surface. Subsequently, an edit is carried out by sketching a new, view-dependent handle position or by indirectly influencing differential properties along the sketch. Combined, these editing and handle metaphors greatly simplify otherwise complex shape modeling tasks.

Keywords: Sketch Based Model Editing, Laplacian Surface Editing, Differential Geometry, Sketching, Deformations

1 Introduction

A few strokes suffice to sketch the main features of a shape. This is why designers still prefer using pen and paper to invent and communicate, and explains the great success of sketch-based shape modeling approaches, such as SKETCH [Zeleznik et al. 1996] and Teddy [Igarashi et al. 1999]. In this work, we add to the existing toolbox of sketch based shape modeling techniques. Our contribution is a tool for sketching significant shape details on already existing coarse or detailed shapes. We believe the important first step of creating the basic shape from scratch is essentially solved: either based on sketching (apart from the pioneering works mentioned above, see also [Karpenko et al. 2002; Igarashi and Hughes 2003; Bourguignon et al. 2004]) or using other modeling techniques. Ideally, a sketch-based modeling system for 3D shapes should use the very same sketches that designers would draw on a piece of paper to convey the shape. What are these lines? As pointed out by Hoffman and Singh [1997], the human visual system uses silhouettes as the first index into its memory of shapes, making everyday objects recognizable without color, shading or texture, but solely by their contours. In the area of non-photorealistic rendering (NPR), silhouettes have been used extensively [Gooch and Gooch 2001] and recently they have been extended to *suggestive* contours: curves on the shape that might be silhouettes in nearby views [DeCarlo et al. 2003]. The apparent presence of a feature line in a picture of a shape results from an abrupt change in illumination. Apart from view dependent features for which this happens or might happen in a nearby view, change of illumination generally correlates with curvature. Lines of curvature extrema (i.e. ridges and ravines) have, therefore, also been used in NPR for conveying shape.

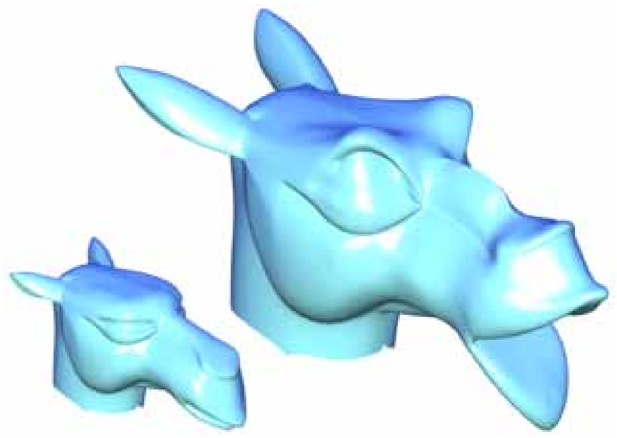


Figure 1: With a few strokes we have greatly increased the expressiveness of the CAMEL model (bottom left). See Fig. 2 for details.

We come to the conclusion that **sketching** a shape is **inverse NPR**. Consequently, we design a sketch-based modeling interface using silhouettes and sketches as input, and producing contours, or suggestive contours, and ridges/ravines. The user can sketch a curve, and the system adapts the shape so that the sketch becomes a feature line on the model, while preserving global and local geometry as much as possible. As the requested properties of the sketch cannot or should not always be accommodated exactly, users only *suggest* feature lines.

It might seem obvious to let users alter contours, or ask for a line in space to be a feature line. Interestingly, our concurrent goals of preserving the global and local geometry during the edit while using feature lines for defining the edit are difficult to implement using traditional approaches: typical sketching tools [Igarashi et al. 1999; Karpenko et al. 2002; Fleisch et al. 2004] do use silhouettes, however, they create only smooth shapes. Some operations of sketching techniques might preserve geometric detail, however, they are not based on inserting feature lines into the shape [Draper and Egbert 2003; Kho and Garland 2005]. In general modeling environments, such as space deformation techniques (e.g., [Sederberg and Parry 1986; Singh and Fiume 1998]) and multi-resolution or subdivision mesh modeling approaches [Zorin et al. 1997; Kobbelt et al. 1998; Biermann et al. 2000], it can be difficult to incorporate the displacement of a feature curve: these approaches provide a basis that spans a space of shapes; the requested displacement has to be translated into coefficients of this basis. In general, this might be impossible, and an approximate solution typically leads to a difficult inverse problem (see also Botsch and Kobbelt [2004]). Our idea becomes realizable through the recent trend to cast mesh modeling problems as discrete Laplace or Poisson models [Alexa 2003; Botsch and Kobbelt 2004; Sorkine et al. 2004; Yu et al. 2004; Sumner and Popović 2004]. Within this framework, it is easy to displace a set of edges (e.g., sketch a new position of an identified contour) while preserving the geometric details of the surface as much as possible. However, most of the feature lines we want to use have specific differential properties, either absolute or relative to the viewing direction, and they might not coincide with edges on the mesh. We therefore extend the framework of Laplace/Poisson mesh modeling in the following ways: (a) we accommodate constraints on the normals and the curvature; (b) we allow constraints to be placed

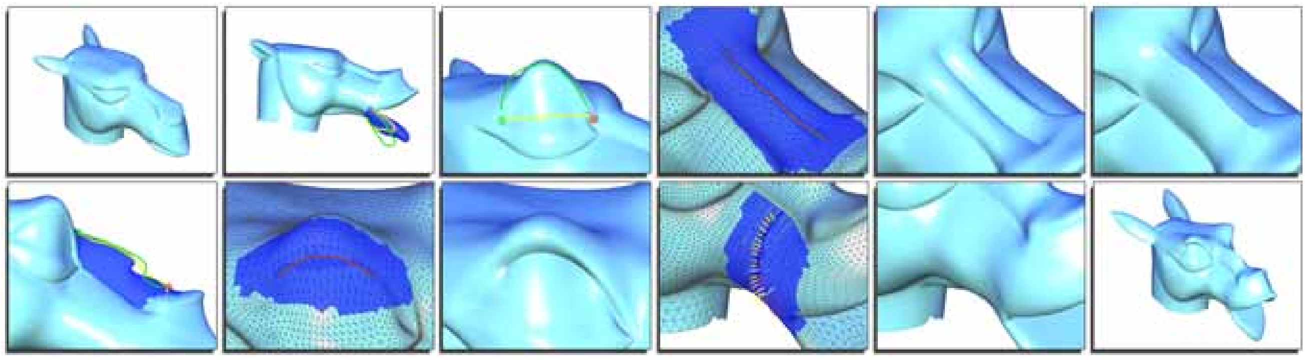


Figure 2: Our mesh editing tool in action. Top row [(1)-(6)]: First, we open the mouth of the CAMEL model (1) by detecting an object silhouette, and sketching an approximation of the lip shape we want (2) (See Section 3). Note that in (2) the yellow curve is the original object silhouette, the green curve is the user sketch, and the dark blue region is the result of a previously placed sketch. By sketching directly onto the model (3) we produce a handle (yellow) by which we can lift the eyebrow with the green sketch. For the creation of sharp features we sketch the feature line (4) and then scale the affected Laplacians to produce either a ravine (5) or a ridge (6) (See Section 4.2). Bottom row [(7)-(12)]: If we are not yet satisfied with the ridge in (6), we can edit the newly created object contour using our silhouette tool (7). Sketching a ravine under the eye by geometry adjustment (See Section 4.1) is shown in (8) and (9). Finally, we sketch a subtle suggestive contour near the corner of the mouth in (10) and (11) (See Section 4.3), resulting in the SCREAMING CAMEL model (12), shown in Fig. 1.

on virtual vertices, i.e. vertices placed on edges that only serve to implement the constraints but are never added to the mesh; (c) we incorporate a tangential mesh regularization, which moves edges onto sharp features while ensuring well-shaped triangles.

This mesh modeling framework together with a user-interface mostly based on sketching suggested feature lines onto or around a shape, indeed, yields an intuitive shape modeling technique.

2 Mesh modeling framework

The basic idea of the modeling framework is to satisfy linear modeling constraints (exactly, or in the least squares sense), while preserving differential properties of the original geometry. This technique has recently been presented in various fashions and we only briefly explain the main concepts. For more detailed explanations see the references given below. One way of deriving these linear constraints is to ask that the Laplacian of the original geometry be preserved in the least squares sense [Alexa 2003; Lipman et al. 2004]. Let the mesh be represented as a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, consisting of vertices $\mathbf{V}$ and edges $\mathbf{E}$. Let $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$, $\mathbf{v}_i = (v_{i_x}, v_{i_y}, v_{i_z}) \in \mathbb{R}^3$ be the original geometry and Δ the Laplace operator, then the deformed geometry $\mathbf{V}'$ is defined by the constrained minimization

$$\mathbf{V}' = \arg\min_{\mathbf{W}} \|\Delta\mathbf{V} - \Delta\mathbf{W}\|^2, \tag{1}$$

where the vertices might be weighted differently to trade-off between modeling constraints and the reproduction of original surface geometry. Note that this is equivalent to solving a linear system of the form $\mathbf{AV}' = \mathbf{b}$ in the least squares sense. If the original surface was a membrane, the necessary constraints for the minimizer lead to $\Delta^2\mathbf{V}' = 0$, which has been advocated by Botsch and Kobbelt [2004] in the context of modeling smooth surfaces. If, in contrast, the original surface contained some detail, the right-hand side is non-zero and we arrive at a variant of the discrete Poisson modeling approach of Yu et al. [2004].

The modeling operation is typically localized on a part of the mesh. This part of the mesh is selected by the user as the region of interest (ROI) during the interactive modeling session (with a lasso tool). The operations are restricted to this ROI, padded by several layers of anchor vertices. The anchor vertices yield positional constraints $\mathbf{v}'_i = \mathbf{v}_i$ in the system matrix $\mathbf{A}$, which ensure a gentle transition between the altered ROI and the fixed part of the mesh. Based on the constraints formulated so far, local surface detail is preserved if parts of the surface are translated, but changes with rotations and scales. One way of dealing with this is to define local rotations per vertex a priori. Lipman et al. [2004] compute these rotations from a smoothed solution of Eq. 1, Yu et al. [2004] let the user specify a few constraint transformations and then interpolate them over the surface. However, we would like to incorporate the treatment of directions into the modeling phase so that some of the details have a fixed (normal) orientation, while others may rotate. Thus, we adopt the approach of Sorkine et al. [2004], who define the local rotations and scales by comparing one-rings between $\mathbf{V}$ and $\mathbf{V}'$. However, we discretize the Laplace operator using cotangent weights as recommended by Meyer et al. [2003]. The conditions to be satisfied lead to an overdetermined system of linear equations of the form $\mathbf{AV}' = \mathbf{b}$, which we solve in the least squares sense according to the normal equations $\mathbf{A}^T\mathbf{AV}' = \mathbf{A}^T\mathbf{b}$. For information on how to derive the rows resulting from Eq. 1 see [Sorkine et al. 2004].

We extend this framework towards constraints on arbitrary points on the mesh. Note that each point on the surface is the linear combination of two or three vertices. A point on an edge between vertices i and j is defined by one parameter as $(1-\lambda)\mathbf{v}_i + \lambda\mathbf{v}_j$, $0 \le \lambda \le 1$. Similarly, a point on a triangle is defined by two parameters. We can put positional constraints $\hat{\mathbf{v}}_{ij}$ on such a point by adding rows to the system matrix $\mathbf{A}$ of the form

$$(1-\lambda)v'_{i_x} + \lambda v'_{j_x} = \hat{v}_{ij_x}, \ldots . \tag{2}$$

Furthermore, we extend the framework by using other forms of differentials to achieve some additional effects. Let δ_i be the Laplacian of $\mathbf{v}_i$, the result of applying the discrete Laplace operator to $\mathbf{v}_i$, i.e.

$$\delta_i = \mathbf{v}_i - \sum_{\{i,j\}\in E} w_{ij}\mathbf{v}_j, \tag{3}$$

where $\sum_{\{i,j\}\in E} w_{ij} = 1$, and the weights w_{ij} are determined using the cotangent weights [Meyer et al. 2003]. An important benefit of this weighting is that δ_i points in the normal direction, and the length $\|\delta_i\|$ is proportional to the mean curvature around vertex i. This allows us to prescribe a certain normal direction and/or curvature for a vertex, simply by adding a row to $\mathbf{A}$ of the form

$$\mathbf{v}'_i - \sum_{\{i,j\}\in E} w_{ij}\mathbf{v}'_j = \delta'_i. \tag{4}$$

Setting the normal direction is necessary for contours and suggestive contours, setting the curvature – for ridges or ravines.

To access the tangential location of vertices, we use the umbrella operator [Kobbelt et al. 1998] as a discrete Laplacian and relate it to the cotangent weighted Laplace operator. We exploit this for regularizing the mesh in tangential direction, by asking that

$$\mathbf{v}'_i - d_i^{-1} \sum_{\{i,j\}\in E} \mathbf{v}'_j = \mathbf{v}_i - \sum_{\{i,j\}\in E} w_{ij}\mathbf{v}_j, \tag{5}$$

where d_i is the degree of vertex i. The rationale behind this operation is this: the uniformly weighted operator generates a tangential component, while the cotangent weighting does not. Asking that they are equivalent is essentially solving the Laplace equation but only for the tangential components. The result is a mesh with well shaped triangles, preserving the original mean curvatures as long as the tangential offset is not too large. Note that we typically restrict this operation to small regions, so that large tangential drift cannot occur.

In the following sections, we explain how to use these basic building blocks for satisfying user-defined feature lines on a mesh.

3 Silhouette sketching

Our goal is to identify areas of the model which are easily recognized, and for which our memories hold vast databases of possible variations, and then apply these variations by sketching them. The idea is simple yet effective: after defining a region of interest on the surface and a camera viewpoint, we select (and trim) one of the resulting silhouettes, and then sketch a new shape for this silhouette (see Fig. 3).

For the computation of silhouettes on polygonal meshes, various methods are available, see [Hertzmann 1999]. We have chosen to use object space silhouettes, and include the ability to switch between edge silhouettes (mesh edges, for which one adjacent face is front-facing and one is back-facing) and smooth surface silhouettes [Hertzmann and Zorin 2000]. Hertzmann and Zorin [2000] determine the silhouette on mesh edges $e = (\mathbf{v}_i, \mathbf{v}_j)$ by linearly interpolating corresponding vertex normals $\mathbf{n}_i, \mathbf{n}_j$: a silhouette point $\mathbf{p} = (1-\lambda)\mathbf{v}_i + \lambda\mathbf{v}_j$ on e has to satisfy $((1-\lambda)\mathbf{n}_i + \lambda\mathbf{n}_j)\cdot(\mathbf{p}-\mathbf{c}) = 0$, where $\mathbf{c}$ is the viewpoint. Silhouette points on edges are connected by segments over faces.

During editing, the user first picks one of the connected components, and then interactively adjusts the start and end point by dragging them with the mouse. Note that degenerate silhouette edge paths might lead to multiply connected curves, resulting in non-intuitive user interaction. Smooth silhouettes [Hertzmann 1999] remedy this problem on smoothly varying surfaces, and only for models with distinct sharp features (such as CAD models), mesh edges are used as silhouettes. In any case, the selected silhouette segment is represented as a set of points $\mathbf{q}_i$ on the mesh.

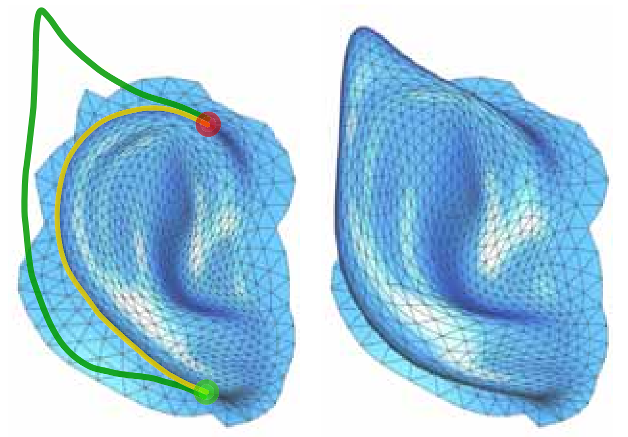


Figure 3: Sketching a very recognizable ear silhouette: we detect, select, crop and parameterize an object silhouette (yellow, the green and red balls represent begin and end vertices respectively), and then sketch a new desired silhouette (green).

After selecting a silhouette segment, the user sketches a curve on the screen, representing the suggested new silhouette segment. The sketch is represented as a polyline in screen space. The vertex locations $\mathbf{s}_i$ on this polyline result in constraints on mesh vertices as follows: First, silhouette vertices $\mathbf{q}_i$ are transformed to screen space, i.e. the first two components contain screen space coordinates, while the third contains the z-value. Then, both curves are parameterized over $[0,1]$ based on edge lengths of the screen space polylines. This induces a mapping from $\mathbf{q}_i$ to $\{\mathbf{s}_j\}$, defining a new screen space position $\mathbf{q}'_i$ (note that $\mathbf{q}'_i$ retains the z-value of $\mathbf{q}_i$).

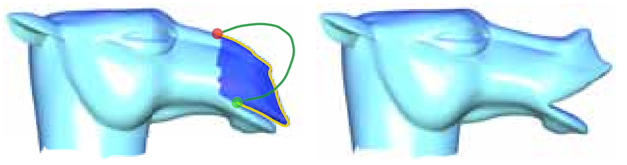


Figure 4: Sketching an *approximate* CAMEL lip by reducing the weights on the positional constraints for silhouette vertices.

The new position $\mathbf{q}'_i$ in screen space is transformed back to model space and serves as a positional constraint. Note that when using smooth surface silhouettes, on-edge constraints have to be used (see Eq. 2). Additionally, varying the weighting of positional constraints along the silhouette against Laplacian constraints leads to a trade off between the accurate positioning of silhouette vertices under the sketch curve, and the preservation of surface details in the ROI. To achieve this, we simply multiply the affected rows in $\mathbf{A}$ and $\mathbf{b}$ with the selected weighting factor. For example, the result in Fig. 3 follows the sketch closely, whereas the sketch in Fig. 4 only hints at the desired lip position.

This method works well even for moderately noisy and bumpy surfaces and preserves details nicely (see Fig. 5). Note that for very noisy surfaces, object space silhouette paths and loops may become arbitrarily segmented, in which case our silhouette sketching method is no longer applicable. In such cases, sketch editing can be performed relative to any user-defined curve sketched manually onto the surface, as was done for lifting the eyebrows of the CAMEL, see Fig. 2(3).

The matrix $\mathbf{A}^T\mathbf{A}$ is computed and factored once for each ROI and silhouette curve selection, and we simply solve for each sketch by back substitution [Toledo 2003]. Some editing results in Fig. 1 were obtained by using the silhouette editing capabilities of our system: sketching larger ears, opening the mouth and modifying the nose contour.

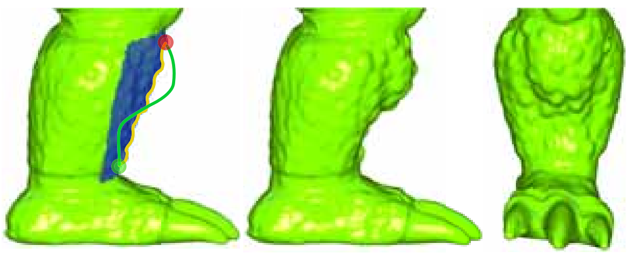


Figure 5: Editing the bumpy ARMADILLO leg: although the silhouette (yellow) in the ROI (blue) has substantial depth variation and the desired silhouette (green) is smooth, properly weighting the positional constraints retains the surface characteristics after the edit.

4 Feature and contour sketching

4.1 Geometry adjustment

Suppose we intend to create a potentially sharp feature where we have drawn our sketch *onto* the mesh. To create a meaningful feature (i.e. a ridge, ravine or crease) on a mesh, we must first adjust the mesh geometry to accommodate such a feature directly under the sketch, since in our setting the sketch need not run along an edge path of the mesh. To illustrate this, see Fig. 6(a), where the sketch path $\{\mathbf{s}_i\}$ (green) follows the edges on the left, but runs perpendicular to them on the right. By applying repeated subdivision we could have locally adjusted the mesh resolution, but for situations similar to the one in Fig. 6(a), many levels of subdivision would be necessary to properly approximate the sketch with an edge path. Another option would be to cut the mesh along the sketch; however, we have found a simpler method that avoids increasing the mesh complexity, yields nice feature lines and well-shaped triangles while retaining the original mesh topology. In detail:

The triangles in the ROI are transformed to screen space; triangles intersecting $\{\mathbf{s}_i\}$ are gathered (Fig. 6(a), dark triangles) and the begin and end mesh vertices are identified.

An edge path $\mathbf{V}_p = (\mathbf{v}_{p_1}, \mathbf{v}_{p_2}, \dots, \mathbf{v}_{p_n})$ that is close to $\{\mathbf{s}_i\}$ is computed by solving a weighted shortest path problem in the edge graph of the ROI. The weight for each edge is the sum of its vertices' screen space distance to $\{\mathbf{s}_i\}$. The resulting edge path vertices are generally not on, but close to $\{\mathbf{s}_i\}$ (shown in red in Fig. 6(a)).

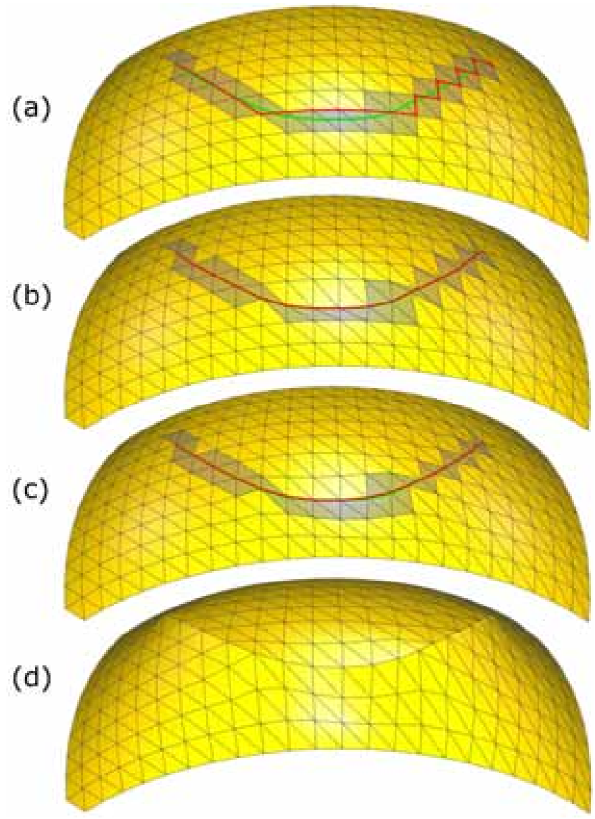


Figure 6: Creating a ravine-like crease: in (a) the green sketch given by the user is approximated by the red edge path on the original geometry. We adjust the geometry to lie directly under the sketch by orthogonal projection along the tangent plane (b), and then relax the area around the sketch (c). Now we can create the crease by scaling the Laplacians along the edge path (d), resulting in a sharp feature, even for this coarsely sampled surface.

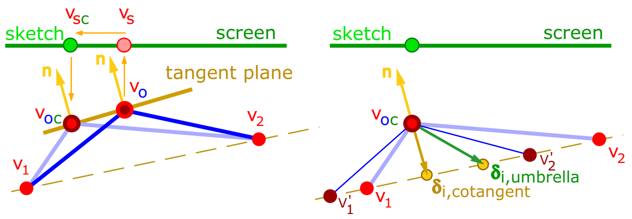


Figure 7: Adjusting edge path vertices to lie under the sketch curve (left): an object-space edge path vertex $\mathbf{v}_o$ is projected to $\mathbf{v}_s$ in screen space, from there orthogonally projected onto $\mathbf{v}_{sc}$ on the sketch curve, and then projected back onto the tangent plane defined by the normal at $\mathbf{v}_o$, yielding the new vertex position $\mathbf{v}_{oc}$. Relaxing the sketch region (right): to ensure a good triangulation after adjusting the geometry, we perform a relaxation of the edge-path vertices (allow them to move along the sketch path) and nearby vertices by constraining $\delta_{i,umbrella}$ to $\delta_{i,cotangent}$ in the least squares sense. Qualitatively, this moves $\mathbf{v}_1$ and $\mathbf{v}_2$ to $\mathbf{v}'_1$ and $\mathbf{v}'_2$, while keeping $\mathbf{v}_{oc}$ under the sketch.

The path vertices $\mathbf{V}_p$ are mapped onto closest edges of the sketch path $\{\mathbf{s}_i\}$ in screen space; corresponding z-values are computed from restricting each vertex to move on its tangent plane, as defined by the original vertex normal (Fig. 7, left). The resulting edge path closely follows the sketch curve (Fig. 6(b)), yet may introduce badly shaped triangles.

We improve triangle shapes by relaxing vertices close to the sketch so that their umbrella Laplacian equals the cotangent Laplacian in the least squares sense (See Fig. 7, right, and Section 2). For the vertex relaxation we must solve a linear system, much like the actual editing solver, but with constraints given by Eq. 5. Obviously, the edge path vertices must remain under the sketch path during this procedure. To ensure this, while also giving the edge path vertices a valid degree of freedom, we add them as positional constraints (Section 2), and additionally add averaging constraints of the form

$$\mathbf{v}'_{p_i} - \frac{1}{2}\mathbf{v}'_{p_{i-1}} - \frac{1}{2}\mathbf{v}'_{p_{i+1}} = 0, \tag{6}$$

for all vertices in $\mathbf{V}_p$ excluding the begin and end vertices. The averaging constraint *loosens* the positional constraint, allowing edge path vertices to move between their adjacent vertices in the path. Adjusting the ratio of weights between positional and averaging constraints leads to a trade-off between accurately approximating the sketch, and some possibly desired path smoothing.

We have experienced no detrimental effects when applying this procedure on meshes which approximate the underlying smooth surface well, even in areas of high curvature. Also, small changes might be tolerable, as this region will be subsequently edited.

After the geometry adjustment step, the surface is prepared for editing operations in the vicinity of the sketch.

4.2 Sharp features

To create a sharp feature along the edge path, we adjust the Laplacians of path vertices when constructing the $\mathbf{A}$ matrix by prescribing the Laplacian transform for sketch vertices without flexibility to rotate or scale (i.e., as in Eq. 4). Since we discretize the Laplacian using the cotangent weights, we can simply scale the Laplacians of edge path vertices, resulting in a ridge or ravine, depending on the sign. If the Laplacian evaluates to zero, as is the case for flat surfaces, we instead scale the surface normal and prescribe it as the new Laplacian. As described in Section 3, we factor the matrix $\mathbf{A}^T\mathbf{A}$ once we have selected a sketch, and can then quickly evaluate the results of varying scales by dragging the mouse up and down. The creation of a sharp ridge is shown in Fig. 6(d). Alternatively, we can add some amount to the Laplacians, making the change absolute rather than relative. This works well in regions with high curvature variation along the sketch.

We have found it to be very convenient to create a ridge using our modeling framework, and thereafter treat it as a silhouette from a different camera position and edit it as outlined in Section 3. This technique was applied in the creation of the wavy ridge along the nose of the CAMEL model in Figures 1 and 2(7).

4.3 Smooth features and suggestive contours

Applying the editing metaphor described in the previous section can only create sharp features. To enable smooth features or suggestive contours, we need to influence the Laplacians of more vertices than only those lying on the edge path. Additionally, for suggestive contours, we intend to manipulate curvature in the viewing direction. Thus, we need to rotate the Laplacians w.r.t. an axis which is orthogonal to both viewing and normal vectors. After performing the geometry adjustment of Section 4.1, given the viewing position $\mathbf{c}$, we gather and segment vertices within a user-defined sketch region around the edge path as follows (Fig. 8, top):

- For each path vertex $\mathbf{v}_{p_i}$ with normal $\mathbf{n}_{p_i}$ (the yellow vectors in Fig. 8) we compute the radial plane $\mathbf{r}_i$, which passes through $\mathbf{v}_{p_i}$ with plane normal $\mathbf{n}_{r_i} = (\mathbf{v}_{p_i} - \mathbf{c}) \times \mathbf{n}_{p_i}$ (the blue vectors in Fig. 8). Now we can segment the vertices in the sketch region $\mathbf{V}_s = (\mathbf{v}_{s_1}, \mathbf{v}_{s_2}, \ldots, \mathbf{v}_{s_n})$ such that each sketch region vertex is associated with one such plane (ergo, each vertex in $\mathbf{V}_s$ *belongs* to one edge path vertex).
- Each vertex in $\mathbf{V}_s$ is assigned to the radial plane it is closest to, where the distance of $\mathbf{v}_{s_j}$ to plane $\mathbf{r}_i$ is measured as $d_j = orthodist(\mathbf{r}_i, \mathbf{v}_{s_j}) + dist(\mathbf{v}_{p_i}, \mathbf{v}_{s_j})$. Here, *orthodist* measures orthogonal distance to the plane, and *dist* is the Euclidean distance between $\mathbf{v}_{p_i}$ and $\mathbf{v}_{s_j}$. We take Euclidean distance into account to avoid problems which occur when two different path vertices have similar radial planes, and furthermore to limit the support of the sketch region.

In Fig. 8 (top image), we show one such segmentation, where the edge path vertices are highlighted with red circles and the segmentation is color coded (i.e. all vertices of the same color are associated with the path vertex of that color).

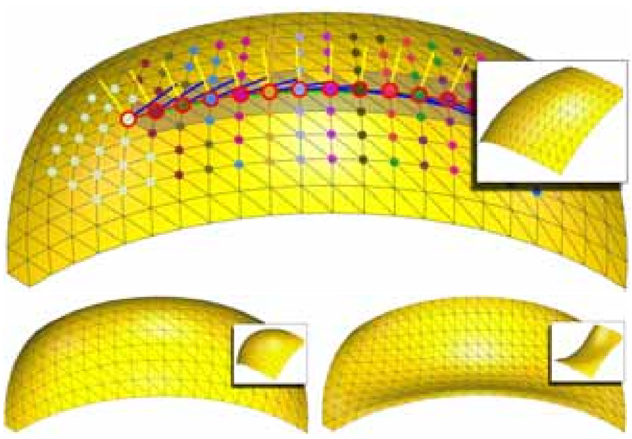


Figure 8: Top: view dependent vertex segmentation and rotation axis assignment. Bottom left: scaling all Laplacians in the sketch region by the same factor produces smooth ridges and ravines. Bottom right: rotating all Laplacians by an angle of $-\pi/2$ w.r.t. the blue rotation axes results in a suggestive contour.

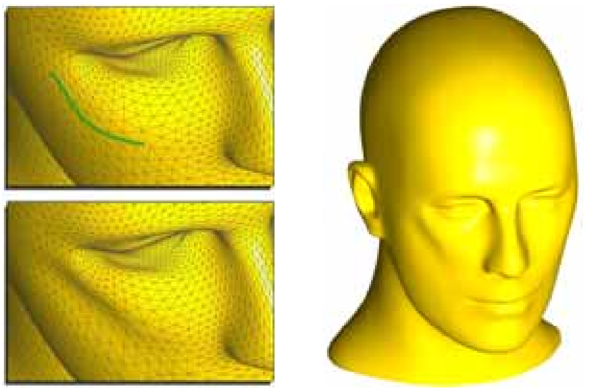


Figure 9: Adding a strong cheekbone to the MANNEQUIN model by sketching a suggestive contour.

Once we have this segmentation, one possible operation is to uniformly scale (or add to) the Laplacians of all sketch region vertices. Complementing the sharp features of Section 4.2, this operation gives us smooth bumps and valleys (Fig. 8, bottom left). By setting the Laplacians to zero we can flatten specific regions of the mesh.

An alternative editing behavior results from rotating all Laplacians w.r.t. their respective rotation axes (given by above segmentation) by a user-defined angle, determined by dragging the mouse left or right. Note that rotation by π is identical to scaling by minus one. For angles in the ranges $[0, \pi)$ and $(\pi, 2\pi]$ we create varying radial curvature inflection points (Fig. 8, bottom right), resulting in suggestive contours [DeCarlo et al. 2003] such as the cheekbone shown in Fig. 9. Note that these inflection points are not necessarily directly under the sketch, since they result from the Laplacian surface reconstruction and the boundary constraints around the ROI.

5 Discussion

Generating plausible and visually pleasing shapes and deformations is far from trivial: while our capability to derive a mental model from everyday shapes around us is well developed, we fail to properly communicate this to a machine. This is why we have to model in a loop, constantly correcting the improper interpretation of our intentions.

The quality of shape editing, therefore, depends on two factors: the time required by the system to update the shape after user commands and how well the shape change reflects our mental model of that process. The update time is a potential bottleneck in our approach, as the necessary matrix factorization and back substitution depend on the number of vertices and not the complexity of the edit operation. For example, ROI sizes of 5.5K/12K/33K vertices require 0.7/2.5/7.0 seconds for factorization and 0.035/0.07/0.25 seconds for back substitution on an Intel P4/2.0 GHz. On the other hand, we believe we have improved the match between the mental model and shape updates, though this is obviously hard to quantify.

From a user's point of view, our system is similar to other sketch-based editing interfaces [Igarashi et al. 1999; Karpenko et al. 2002; Draper and Egbert 2003; Kho and Garland 2005], while it differs algorithmically: the above methods are based on space warps and variational implicits, whereas our representation is aimed at surface detail preservation. Our method inherits the simplicity of the user interface, and enables the creation of interesting and useful surface edits, both for inexperienced users and modeling professionals.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by

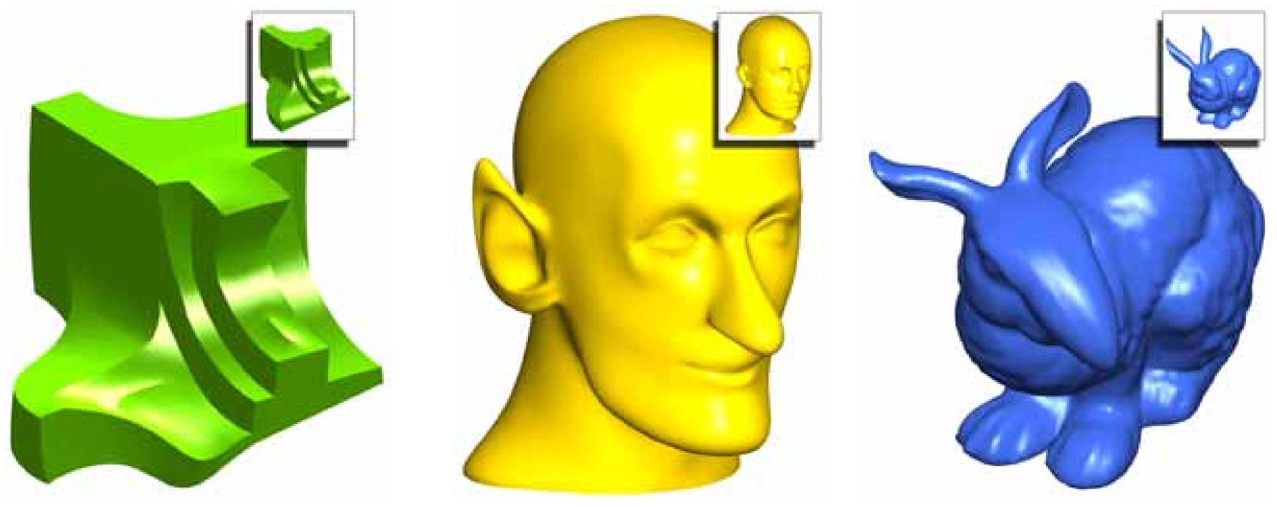


Figure 10: Some results: a deformed FANDISK with a few more sharp features, a rather surprised MANNEQUIN with more than just an extra contour around the eye, and droopy-eared, big-nose BUNNY with large feet.

grants from the European Network of Excellence AIM@SHAPE (FP6 IST NoE 506766), the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and by the Israeli Ministry of Science.

References

ALEXA, M. 2003. Differential coordinates for local mesh morphing and deformation. *The Visual Computer 19*, 2, 105–114.

BIERMANN, H., LEVIN, A., AND ZORIN, D. 2000. Piecewise smooth subdivision surfaces with normal control. In *Proceedings of SIGGRAPH 2000*, 113–120.

BOTSCH, M., AND KOBBELT, L. 2004. An intuitive framework for real-time freeform modeling. *ACM Trans. Graph. 23*, 3, 630–634.

BOURGUIGNON, D., CHAINE, R., CANI, M.-P., AND DRETTAKIS, G. 2004. Relief: A modeling by drawing tool. In *First Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 151–160.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Trans. Graph. 22*, 3, 848–855.

DRAPER, G., AND EGBERT, P. 2003. A gestural interface to freeform deformation. In *Proceedings of Graphics Interface 2003*, 113–120.

FLEISCH, T., RECHEL, F., SANTOS, P., AND STORK, A. 2004. Constraint stroke-based oversketching for 3D curves. In *First Eurographics Workshop on Sketch-Based Interfaces and Modeling*.

GOOCH, B., AND GOOCH, A. 2001. *Non-Photorealistic Rendering*. A.K. Peters.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proceedings of SIGGRAPH 2000*, 517–526.

HERTZMANN, A. 1999. Introduction to 3D non-photorealistic rendering: Silhouettes and outlines. In *Non-Photorealistic Rendering. SIGGRAPH 99 Course Notes.*

HOFFMAN, D. D., AND SINGH, M. 1997. Salience of visual parts. In *Cognition*, vol. 63(1), 29–78.

IGARASHI, T., AND HUGHES, J. F. 2003. Smooth meshes for sketch-based freeform modeling. In *2003 ACM Symposium on Interactive 3D Graphics*, 139–142.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3D freeform design. In *Proceedings of SIGGRAPH 99*, 409–416.

KARPENKO, O., HUGHES, J. F., AND RASKAR, R. 2002. Freeform sketching with variational implicit surfaces. *Computer Graphics Forum 21*, 3, 585–594.

KHO, Y., AND GARLAND, M. 2005. Sketching mesh deformations. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, 147–154.

KOBBELT, L., CAMPAGNA, S., VORSATZ, J., AND SEIDEL, H.-P. 1998. Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of SIGGRAPH 98*, 105–114.

LIPMAN, Y., SORKINE, O., COHEN-OR, D., AND LEVIN, D. 2004. Differential coordinates for interactive mesh editing. In *International Conference on Shape Modeling and Applications*, 181–190.

MEYER, M., DESBRUN, M., SCHRÖDER, P., AND BARR, A. H. 2003. Discrete differential-geometry operators for triangulated 2-manifolds. *Visualization and Mathematics III*, pages 35–57.

SEDERBERG, T. W., AND PARRY, S. R. 1986. Free-form deformation of solid geometric models. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, vol. 20, 151–160.

SINGH, K., AND FIUME, E. L. 1998. Wires: A geometric deformation technique. In *Proceedings of SIGGRAPH 98*, 405–414.

SORKINE, O., LIPMAN, Y., COHEN-OR, D., ALEXA, M., RÖSSL, C., AND SEIDEL, H.-P. 2004. Laplacian surface editing. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry processing*, 179–188.

SUMNER, R., AND POPOVIĆ, J. 2004. Deformation transfer for triangle meshes. *ACM Trans. Graph. 23*, 3, 399–405.

TOLEDO, S. 2003. TAUCS*: A Library of Sparse Linear Solvers*. Tel Aviv University.

YU, Y., ZHOU, K., XU, D., SHI, X., BAO, H., GUO, B., AND SHUM, H.-Y. 2004. Mesh editing with poisson-based gradient field manipulation. *ACM Trans. Graph. 23*, 3, 644–651.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: An interface for sketching 3D scenes. In *Proceedings of SIGGRAPH 96*, 163–170.

ZORIN, D., SCHRÖDER, P., AND SWELDENS, W. 1997. Interactive multiresolution mesh editing. In *Proceedings of SIGGRAPH 97*, 259–268.

EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2005)
J.A. Jorge and T. Igarashi (Guest Editors)

ShapeShop: Sketch-Based Solid Modeling with BlobTrees

R. Schmidt[1], B. Wyvill[1], M. C. Sousa[1], J. A. Jorge[2]

[1]Dept. Computer Science
University of Calgary, Canada

[2] Dept. Information Systems and Computer Engineering
TU Lisbon, Portugal

Abstract

Various systems have explored the idea of inferring 3D models from sketched 2D outlines. In all of these systems the underlying modeling methodology limits the complexity of models that can be created interactively. The ShapeShop sketch-based modeling system utilizes Hierarchical Implicit Volume Models (BlobTrees) as an underlying shape representation. The BlobTree framework supports interactive creation of complex, detailed solid models with arbitrary topology. A new technique is described for inflating 2D contours into rounded three-dimensional implicit volumes. Sketch-based modeling operations are defined that combine these basic shapes using standard blending and CSG operators. Since the underlying volume hierarchy is by definition a construction history, individual sketched components can be non-linearly edited and removed. For example, holes can be interactively dragged through a shape. ShapeShop also provides 2D drawing assistance using a new curve-sketching system based on variational contours. A wide range of models can be sketched with ShapeShop, from cartoon-like characters to detailed mechanical parts. Examples are shown which demonstrate significantly higher model complexity than existing systems.

1. Introduction

A variety of underlying shape representations have been used in sketch-based free-form modeling systems, including triangle meshes [IMT99], subdivision surfaces [IH03], variational implicit surfaces [KHR02][AJ03], convolution surfaces [TZF04], spherical implicit functions [AGB04], and discrete volume data sets [ONNI03]. A common attribute of these systems is that the underlying shape representation heavily influences the sketch-based modeling operations that are implemented. For example, supporting automatic blending with triangle meshes is relatively complex, compared to implicit representations. These issues tend to limit prototype sketch-based modeling systems to operations that are practical to implement, which in turn restricts the types of models that can be sketched by the intended users.

None of the existing systems have been shown to support creation of complex models while retaining interactive performance. Again, the underlying shape representation can fundamentally restrict scalability. For example, variational implicit surfaces [KHR02][AJ03] are generated by solving a large matrix, which is not feasible in real-time except for relatively simple models.

In an attempt to mitigate these issues, we propose Hierarchical Implicit Volume Models (BlobTrees) [WGG99] as an underlying shape representation for sketch-based free-form modeling. A BlobTree procedurally defines an implicit volume using a tree of basic volumes (primitives) and composition operators, such as CSG and blending. Inside this framework, shape-modeling operations such as hole-cutting are easy to implement. The underlying model tree is also a construction history which supports non-linear editing of the model. Using a hierarchical spatial caching scheme [SWG05], complex models can be constructed and manipulated interactively.

We describe *ShapeShop*, a sketch-based 3D BlobTree modeling system in the style of Teddy [IMT99]. ShapeShop includes several sketch-based operations for hole cutting, oversketched blending, and adding surface detail (Section 3). We also introduce a technique for assisting the user with sketching smooth 2D curves, and describe some other gestural interface tools (Section 4).

Traditionally, BlobTree systems have used *skeletal primitives*, essentially offset surfaces from geometric entities. It is non-trivial to define a skeletal primitive such that the offset surface fits a sketched 2D contour [AGB04]. To support sketch-based modeling, we introduce a free-form BlobTree primitive that closely approximates a closed 2D contour using variational interpolation (Section 5). The surface of the primitive can be defined such that it mimics the "inflation" algorithms of existing sketch-based systems [IMT99].

Finally, we provide several examples of models sketched with ShapeShop that display significantly higher surface complexity than previous systems (Section 7).

2. Related Work

Sketch-based 3D modeling systems can be categorized based on how the system creates 3D shapes in response to user input (sketches). *Suggestive* sketch-based modeling systems attempt to map rough sketches to linear geometry such as lines, planes, and polyhedra. These systems frequently use *expectation lists* which allow the user to resolve ambiguous situations. Examples of these systems include SKETCH [ZHH96], Chateau [IH01], and GiDES++ [JSC03].

In contrast, *Literal* sketch-based modeling systems create 3D surfaces directly from user strokes. Examples include Teddy [IMT99], BlobMaker [AJ03], and ConvMo [TZF04]. A fundamental operation in these systems is *inflation*, where user-sketched closed 2D contours become the 3D silhouettes of rounded shapes. Various systems support different sketch-based editing operations on inflated surfaces, including extrusion, cutting, and blending. These systems are frequently classified as *free-form* modeling tools. Our system falls into the Literal sketch-based modeling category.

The Teddy system [IMT99] pioneered the free-form sketch-based modeling concept. Closed triangle meshes were created by inflating user-sketched 2D contours using the *chordal axis* of the 2D polygon. Sketch-based extrusion, cutting, and smoothing operations were supported. Further work produced smoother results by re-meshing the surface based on local quadratic implicit surface approximation [IH03] [MCCH99]. The system was limited to models with spherical topology (genus 0) and low surface complexity.

A recent work by Cherlin et al. [CSSJ05] implements sketch-based modeling using interpolating parametric surfaces. A wide variety of shapes are created using a novel generalized surface-of-revolution scheme. No composition or grouping operations are supported, each surface is independent. The system can scale to a large number of individual surfaces, however each must be manually positioned to give the impression of a solid 3D model. While complex models can be created, the authors note that the requisite manual positioning is very time consuming.

Several attempts have been made to improve on Teddy using implicit surfaces. Variational implicit surfaces were used by Karpenko et al. [KHR02] and the BlobMaker system [AJ03]. Shape-editing was limited to blending and oversketching. In both cases blending was not procedural, the existing surfaces were replaced with a single combined surface. Karpenko's system did maintain a hierarchy of individual components, but this hierarchy was only used for maintaining spatial relationships. An $O(N^3)$ matrix inversion is necessary to solve for the variational function, limiting the number of constraint points (and hence the surface complexity).

Other implicit-based sketching systems have used convolution surfaces [TZF04] and spherical implicit functions [AGB04]. Neither system supports sharp edges, and in both cases only low-complexity models are shown.

A binary volume data set is used by Owada et al. [ONNI03] in a sketching system based on Teddy. The topological restrictions of Teddy are mitigated by the use of a volumetric representation. Novel methods for sketching internal cavities are presented which allow for more detailed models. This system is fundamentally limited by the resolution of the underlying volume data set.

We note that none of the literal sketch-based modeling systems published to date have been shown to scale to even moderately complex models. The stated goal of these tools is generally to support 3D modeling in the conceptual-design phase, and not replace existing shape modeling tools [IMT99][AJ03][TZF04]. However, it is unclear that low-complexity models can adequately represent the often highly-detailed sketches produced in conceptual design.

Tai et al. [TZF04] classify free-form sketch-based modeling systems as either *boundary-based* or *volume-based*. Of the above systems, only two are boundary-based [IMT99][CSSJ05]. However, only Owada et al.'s system [ONNI03] takes advantage of the benefits provided by a volumetric representation. The implicit-based systems largely ignore the extensive framework provided by hierarchical implicit volume modeling [WGG99], and instead focus on surface-smoothness properties. We address the benefits provided by integration of these concepts into a sketch-based modeling system in the following sections.

3. Sketch-Based Modeling Operations

We support construction of three types of surfaces based on sketches - "blobby" inflation in the style of Teddy, linear sweeps, and surfaces of revolution. Based on these three shapes, sketch-based cutting and blending operations are implemented using BlobTree composition operators.

A key benefit of BlobTrees is that the current volume is procedurally defined by the underlying model tree (Section 5.1). This tree represents both a scene graph and a full construction history. Single primitives, as well as entire portions of the tree, can be modified or removed at any time. This flexibility is exposed in ShapeShop mainly via gestural commands and 3D widgets (Section 4.4). However, we also implement a sketch-based resize operation that takes advantage of the BlobTree hierarchy.

3.1. Blobby inflation

A closed 2D contour can be inflated into a "blobby" shape using the technique described in Section 5.2. The 2D sketch (Figure 1a) is projected onto a plane through the origin parallel to the current view plane, and then inflated in both directions (Figure 1b). After creation, the width of the primitive can be manipulated interactively with a slider (Figure 1c). The inflation width is functionally defined and could be manipulated to provide a larger difference between thick and thin sections. One advantage of an implicit representation is that holes and disjoint pieces can be handled transparently.

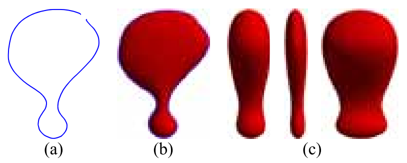


Figure 1: *Blobby inflation converts the 2D sketch shown in (a) into the 3D surface (b) such that the 2D sketch lies on the 3D silhouette. The width of the inflated surface can be manipulated interactively, shown in (c).*

3.2. Sweep surfaces

Our blobby inflation scheme is based on an underlying sweep surface representation which also supports linear sweeps (Figure 2a) and surfaces of revolution (Figure 2b). Linear sweeps are created in the same way as blobby shapes, with the sweep axis perpendicular to the view-parallel plane. The initial length of the sweep is proportional to the screen area covered by the bounding box of the 2D curve, but can be interactively manipulated with a slider. Surfaces of revolution are created by revolving the sketch around an axis lying in the view-parallel plane. Revolutions with both spherical and toroidal topology can be created.

Existing sketch-based systems have generally not included these types of shapes, with the exception of [CSSJ05]. However, we have found them invaluable. Surfaces of revolution are a class of shape that cannot be reproduced with blobby inflation.

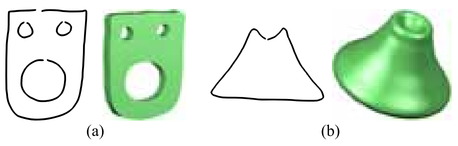


Figure 2: *Sketched 2D curves can also be used to create (a) linear sweeps and (b) surfaces of revolution.*

3.3. Cutting

Since our underlying shape representation is a true volume model, cutting operations can be easily implemented using CSG operators. Users can either cut a hole through the object or remove volume by cutting across the object silhouette. Once a hole is created the user may transform the hole interactively. We provide a slider control to modify the depth of cutting operations. Cut regions are represented internally as linear sweeps, no additional implementation is necessary to support cutting in the BlobTree. As example is shown in Figure 3. This CSG-based cutting operation is both more precise and less restrictive than in existing systems.

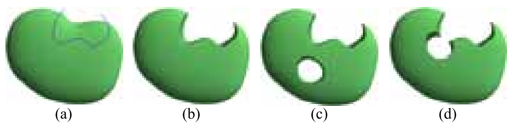


Figure 3: *Cutting can be performed (b) across the object silhoutte or (c) through the object interior. Holes can be interactively translated and rotated. Intersection with other holes is automatically handled, as shown in (d).*

3.4. Blending

We allow the user to blend new blobby primitives to the current volume via oversketching. To position the new blobby primitive, we intersect rays through the sketch vertices with the current implicit volume. The new primitive is centered at the average z-depth of the intersection points. The width of the new blobby primitive can be manipulated with a slider, as can the amount of blending. Blended volumes can be transformed interactively, an example is shown in Figure 4. Karpenko et al [KHR02] supported sketching of the blend profile but also noted that this technique was not robust and is very slow to compute. The level of interactive control over the blend surface in our system has not been previously available.

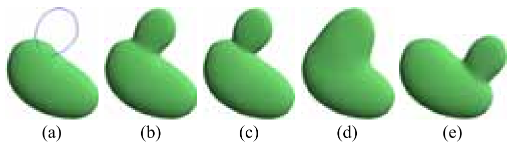


Figure 4: *The sketch-based blending operation (a) creates a new blobby inflation primitive (b) and blends it to the current volume. The blending strength is parameterized and can be interactively manipulated, the extreme settings are shown in (c) and (d). The blend region is recomputed automatically when the blended primitives move, as shown in (e).*

3.5. Surface drawing

Any BlobTree primitive can be used to add surface detail based on sketches. As an initial experiment, ShapeShop supports "surface-drawing". Rays through the 2D sketch are intersected with the current implicit volume. Point primitives, which produce spherical volumes, are placed at intersection points and blended together. Slider controls are provided to manipulate the radius of the point primitives. Results are shown in Figure 5. We are developing a more robust technique involving 3D sweep primitives passing through the intersection points.

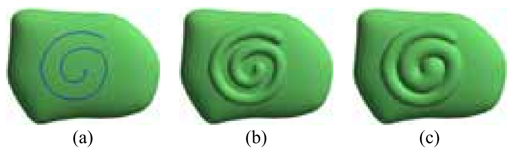


Figure 5: *Surface-drawing is specified by a 2D sketch, as shown in (a). Blended skeletal implicit point primitives are placed along the line at intersection points with the model, shown in (b). In (c) the radius of the points is increased and then tapered along the length of the 2D curve.*

Surface drawing with implicit volumes is a very flexible technique. Any pair of implicit primitive and composition operator can be used as a type of "brush" to add detail to the current surface. For example, creases could be created by subtracting swept cone primitives using CSG operations. Implementing these alternative tools is trivial. In addition, since each surface-drawing stroke is represented independently in the model hierarchy, individual surface details can be modifed or removed using our existing modeling interface.

3.6. Sketch-Based Sweep Manipulation

We provide a sketch-based mechanism for resizing and repositioning linear sweeps and blobby shapes, similar to the method used in the SKETCH system [ZHH96]. The user selects a sweep primitive and rotates the view such that the sweep axis is perpendicular to the view direction. The user then draws a straight line which determines the new extents of the shape. Holes can be manipulated with this technique as well, since they are created using linear sweeps (Figure 6). This operation largely eliminates the need for slider widgets to control sweep length and blobby inflation width, except when very fine-grained manipulation is desired.

4. Modeling Interface

Our sketch-based modeling interface has been designed primarily to support use on large interactive displays, such as the touch-sensitive SmartBoard (Figure 7). These input systems lack any sort of modal switch (buttons). In some

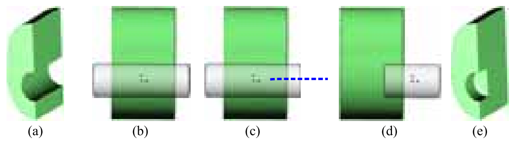


Figure 6: *The linear sweep volume subtracted from (a) is hilighted in (b). By drawing a straight-line stroke parallel to the sweep axis (c), the sweep can be repositioned and resized (d). The new surface is shown in (e).*

sense this is desirable, as pencils also lack buttons. However, tasks commonly initiated with mode-switching (such as keypresses or right mouse buttons) must be converted to alternate schemes, such as gestures or 2D widgets.

Since many 2D widgets can be difficult to use with large-display input devices (which frequently exhibit low accuracy and high latency), we borrow the stroke-based widget interaction techniques of CrossY [AG04]. For example, a button is "pressed" by drawing a stroke that crosses the buttton.

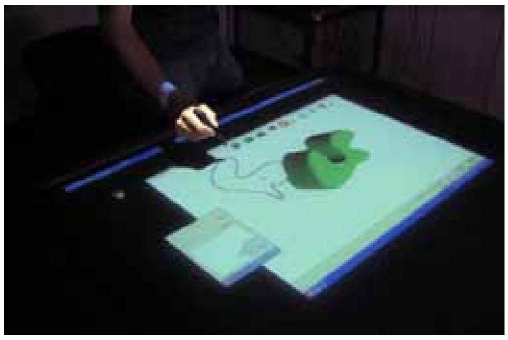


Figure 7: *Our sketch-modeling interface is designed to support non-modal input devices, like this touch-sensitive horizontal tabletop display.*

4.1. 2D Sketch Editing

Two-dimensional sketches form the basis for 3D shape creation in ShapeShop. We have implemented a 2D sketching system that assists with the creation of smooth 2D contours. This system is related to the *interactive beautification* techniques used in the Pegasus system [IMKT97]. Due to space constraints, we will only provide a high-level overview of these techniques.

A fundamental limitation of most standard input devices is that they provide only point samples to the operating system. This discrete data can be converted to a poly-line by connecting temporally-adjacent point samples. However, in the case of curves the poly-line is only an approximation to the smooth curve the user desires. In our system

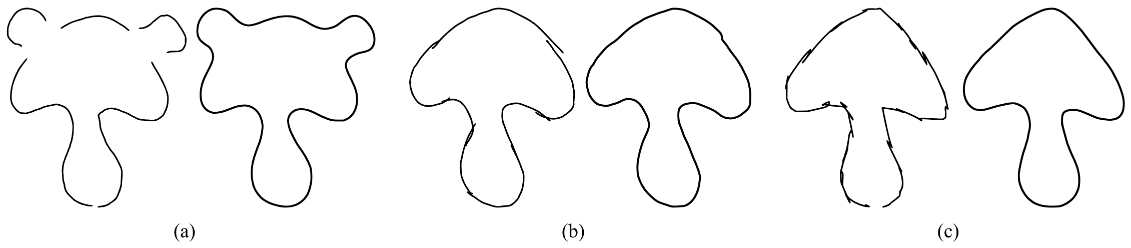


Figure 8: *The gap-filling and smoothing properties of variational curves simplify 2D curve sketching. In (a), multiple disjoint strokes are automatically connected by fitting a variational curve to the input samples. In (b), smoothing parameters are used to handle intersections between multiple strokes. Rough sketches with many self-intersections can also be automatically smoothed, as shown in (c).*

we do not create an approximate poly-line, but instead fit a smooth 2D variational implicit curve [TO02] to the discrete samples. Curve normals derived from the discrete poly-line are used to generate the necessary off-curve constraint points [CBC*01]. Variational curves provide many benefits, such as automatic smoothing and gap-closing with minimal curvature (Figure 8).

ShapeShop supports sketch-based editing of the set of point samples, but not the final variational curve. To simultaneously visualize these two different components, we render the current variational curve in black and the point sample poly-line in transparent blue (Figure 9).

We have implemented three gestural commands to assist users when drawing 2D sketches. The first, *eraser*, is initiated with a "scribble", as shown in Figure 9a. An oriented bounding-box is fit to the scribble vertices and used to remove point samples from the current 2D sketch. The variational curve is re-computed using the remaining samples.

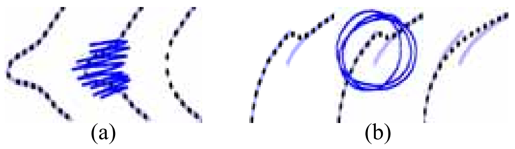


Figure 9: *Examples of the* eraser *gesture (a) and* smooth *gesture (b). These gestures manipulate the parameters used to compute the final variational curve (dashed line).*

The second gestural command is *smooth*, initiated by circling the desired smoothing region a minimum of 2 times. Each point sample has a smoothing parameter associated with it which is incremented if the point is contained in the circled region. The variational curve is then re-computed with the new smoothing parameters (Figure 9b). This gesture can be applied multiple times to the same point samples to further smooth the 2D sketch.

Finally, the *pop* gesture is used to manipulate entire 2D sketches. Using the *erase* command to repair large sketching errors is tedious. Hence, we store individual sketches in a stack. The *pop* gesture, which is input as a quick stroke straight to the left, pops the topmost sketch and discards it.

We have found this system to be very effective for creating smooth 2D sketches. This in turn improves the efficiency of 3D modeling, since fewer corrections need to be made to the 3D shape. One current limitation is that sharp creases in the input sketch are lost, since the underlying variational curve is always C^2 continuous. We are developing additional gesture operations to allow specification of creases.

4.2. Expectation Lists

In ShapeShop the user specifies only 2D silhouettes of the desired 3D shapes. Under this constraint there is an unavoidable ambiguity regarding what shape-modeling operation the user intends. For instance, a given 2D contour can always be interpreted as a blobby shape and a linear sweep. One option is to require additional sketches or gestures to resolve this ambiguity. It is unclear that this extra complexity is more efficient than a visual representation. Hence, we have borrowed the *expectation lists* used in various sketch-based modeling systems [IH01][AJ03][FFJ04].

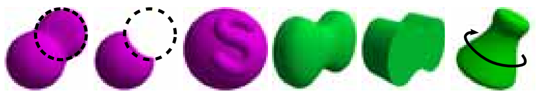


Figure 10: *An example of an expectation list in ShapeShop. The icons denote (from left to right) blend, cut, surface-drawing, blobby inflation, linear sweep, and surface of revolution. The icons are color-coded - green icons create new volumes while magenta icons modify the current volume.*

Existing systems have generally rendered small images of what the updated surface would look like for each expectation list icon. For complex models the user may be required to carefully inspect each image to find the desired action. Instead, We use color-coded iconic representations (Figure 10) which can be easily distinguished. Mistakes can be quickly corrected by erasing nodes from the BlobTree.

The set of icons displayed in the expectation list is context-dependent. For example, if the user draws a stroke which produces a variational curve that is not closed, no shape-creation icons are displayed. However, in many contexts a single stroke can be interpreted as any sketch action. As the set of operations increases, additional strokes may be necessary to prevent the expectation list from becoming too large.

4.3. Dynamic 3D Clipping

Most of the sketch-based shape editing operations described in Section 3 are based on view-parallel planes and ray-surface intersections. It is frequently the case that the desired editing region is obscured by some other part of the current volume. To deal with this situation we use a dynamic cutting plane. Owada [ONNI03] used a dynamic cutting plane to support sketching of internal volumes. While this is possible in ShapeShop, we have found that the primary use for our dynamic cutting plane is in resolving viewing issues and depth-determination ambiguities.

The cutting plane is initiated by the **L**-shaped *clip* gesture. The user draws a straight line across the surface followed by a small perpendicular "tick" (Figure 11). The initial straight line determines the cutting plane orientation, the tick direction determines which side of the plane to clip. Owada's system kept the "right" side of the line, which we found unintuitive when drawing horizontal lines.

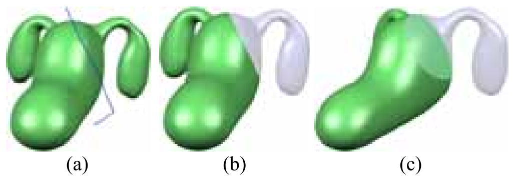


Figure 11: *A temporary cutting plane can assist with sketch-based editing. In (a), the user draws an L-shaped gesture to mark the cutting plane and orientation. Two different views are shown after cutting in (b) and (c).*

4.4. 3D Selection and Transformation

Procedurally-defined BlobTree volumes inherently support non-linear editing of internal tree nodes. However, before a primitive can be manipulated it must be selected. One option is to cast a ray into the set of primitives and select the first-hit primitive. This technique is problematic when dealing with blending surfaces, since the user may click on the visible surface but no primitive is hit.

Instead we implement picking by intersecting a ray with the current volume, then select the primitive which contributes most to the total field value at the intersection point. This algorithm selects the largest contributor in blending situations, and selects the 'hole' primitive when the user clicks on the inside of a hole surface. Since the shape of the selected primitive may not be obvious (if it is part of a blend), we have experimented with several rendering modes (Figure 12) that display the selected internal volume.

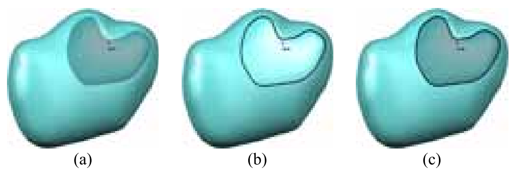


Figure 12: *Internal volumes can be displayed using (a) transparency, (b) silhouette lines, or (c) transparency and silhouttes.*

This selection system only allows for selection of primitives. To select composition nodes we implement a *parent* gesture, which selects the parent of the current node. The *parent* gesture is entered as a straight line towards the top of the screen. Other tree editing operations, such as cut-and-paste, currently require the use of a standard tree widget. An integrated tree visualization tool with gesture-based editing is a feature that we plan on exploring.

A selected primitive or composition node can be removed using the *eraser* gesture described in Section 4.1. Removing a compostion node is equivalent to cutting a branch from the model tree - all children are also removed.

To support 3D manipulation we have implemented standard 3D translation and rotation widgets. These widgets provide both free-translation/rotation in the view-parallel plane as well as constrained manipulation with respect to the unit axes. Compared to the fluid gestural commands used elsewhere in ShapeShop, these 3D widgets are rather crude.

5. Implementation Details

5.1. Hierarchical Implicit Volume Modeling

Given a continuous scalar function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, we can define a volume $\mathcal{V}$:

$$\mathcal{V} = \left\{ \mathbf{p} \in \mathbb{R}^3 : f(\mathbf{p}) \geq v_{iso} \right\} \qquad (1)$$

where v_{iso} is called the *iso-value*. We call $\mathcal{V}$ an *implicit volume*. The surface $\mathcal{S}$ of this volume is defined by replacing the inequality in (1) with an equality. We call this surface the *implicit surface* [Blo97]. This definition also holds in 2D, where $\mathcal{S}$ is a contour.

Two implicit volumes, defined by scalar functions f_1 and f_2, can be combined functionally using a composition operator $\mathcal{G}(f_1, f_2) \in \mathbb{R}_+$. Since $\mathcal{G}$ is also scalar function, composition operators can be applied recursively. A variety of operators are available for performing Computational Solid Geometry (CSG), blending, and space deformation [Blo97].

Recursive application of composition operators results in a tree-like data structure with implicit volumes (*primitives*) at the leaves and composition operators at tree nodes. The final scalar field is evaluated at the root composition operator, which recursively evaluates its children, and so on. This type of procedurally-defined implicit volume model is often called a BlobTree [WGG99].

We restrict the set of primitives we use to those with *bounded* [†] scalar fields. A scalar field f is bounded if $f = 0$ outside some sphere with finite radius. Bounded fields guarantee local influence, preventing changes made to a small part of a complex model from affecting distant portions of the surface. Local influence preserves a "principle of least surprise" that is critical for interactive modeling.

One type of implicit volume primitive with a bounded scalar field is the *skeletal primitive*, defined by a geometric skeleton E (such as a point or line) and a one-dimensional function $g : \mathbb{R}_+ \longrightarrow \mathbb{R}_+$. The scalar function f is then:

$$f_{\mathsf{E},g}(\mathbf{p}) = g \circ d_{\mathsf{E}}(\mathbf{p}) \tag{2}$$

where d_{E} is a function that computes the minimum Euclidean distance from $\mathbf{p}$ to E. The shape of a skeletal primitive is primarily determined by E. We use the following function for g [Wyv05]:

$$g_{wyvill}(x) = (1 - x^2)^3 \tag{3}$$

where x is clamped to the range $[0,1]$. This polynomial smoothly decreases from 1 to 0 over the valid range, with zero tangents at each end. We choose 0.5 as the iso-value.

The basic tree data structure can be augmented by attaching an affine transformation to each node, producing a scene graph suitable for animation. To avoid useless field value queries, a bounding volume containing the non-zero portion of the scalar field can be attached to each node.

To improve interactivity, we use Hierarchical Spatial Caching [SWG05]. *Cache nodes* containing lazily-evaluated discrete volume datasets are inserted into the BlobTree to approximate portions of the model tree. This technique provides interactive performance for complex models.

5.2. A Sketch-Based BlobTree Primitive

Our algorithm for inflating a 2D curve $\mathcal{C}$ consists of two steps. First, we create a bounded 2D scalar field f_M, such that the iso-contour $f_M = v_{iso}$ closely approximates $\mathcal{C}$. Then, we sweep this 2D field along an infinite 3D axis and bound it using g_{wyvill} (Equation 3). The following description is necessarily brief, see our technical report [SW05] for a detailed discussion of our blobby inflation technique, linear sweeps, and surfaces of revolution.

Computing the 2D scalar field f_M also consists of two steps, first creating an unbounded field and then bounding it with g_{wyvill}. We create an unbounded scalar field $f_{\hat{M}}$ such that the iso-contour $f_{\hat{M}} = g^{-1}_{wyvill}(0.5)$ approximates $\mathcal{C}$ by fitting a variational curve to a set of sample points lying on $\mathcal{C}$. To adequately constrain the result, we must also consider off-curve points when fitting the variational solution. We automatically generate inside and outside off-curve points along vectors normal to $\mathcal{C}$, similar to the normal constraints used in 3D variational surface fitting [CBC*01] [TO02]. Additional constraint points are created at a constant radius r_c from $\mathbf{c}$, the center of the bounding box of $\mathcal{C}$. The purpose of these additional constraint points is to attempt to force $f_{\hat{M}}$ to more closely approximate the distance field of $\mathcal{C}$. Distance fields are not used because they contains C^1 discontinuities which create the appearance of creases in the inflated surface.

Once we have computed the 2D variational scalar field $f_{\hat{M}}$, we define f_M at 2D points $\mathbf{u}$:

$$f_M(\mathbf{u}) = g_{wyvill}\left(f_{\hat{M}}(\mathbf{u})\right) \tag{4}$$

which is bounded inside a circle of radius 2 if $\mathcal{C}$ is scaled to fit inside a unit box before computing $f_{\hat{M}}$ and a value of 2 is used for r_c. The resulting scalar field is C^2 smooth and the iso-contour $f_M = v_{iso}$ closely approximates $\mathcal{C}$ (Figure 13).

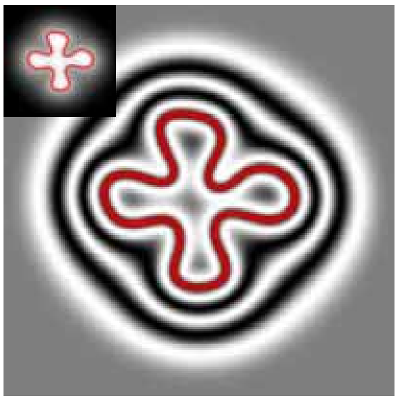


Figure 13: *2D scalar field created using Equation 4. Iso-contours hilighted using* sin *function before mapping to grayscale. Iso-surface is marked in red.*

Creating a 3D bounded scalar field based on f_M is relatively straightforward. Given an origin $\mathbf{o}$, normal $\mathbf{n}$, and two mutually perpendicular vectors $\mathbf{k}_1$ and $\mathbf{k}_2$ in the plane defined by $\mathbf{n}$, we can define an infinite linear sweep of the field f_M. To evaluate f_M at some 3D point $\mathbf{p}$, we require a function

[†] We use the term *bounded*, rather than *compact support*, in an attempt to draw an analogy to the concept of bounding boxes that is ubiquitous in computer graphics.

$\mathbf{F}$ that maps $\mathbf{p}$ to a 2D point $\mathbf{u}$:

$$\mathbf{F}(\mathbf{p}) = \mathbf{Rot}\left[\mathbf{k}_1 \quad \mathbf{k}_2 \quad \mathbf{n}\right] \cdot \mathbf{Tr}\left[-(\mathbf{o}+s\mathbf{n})\right] \cdot \mathbf{p} \qquad (5)$$

where $s = (\mathbf{p}-\mathbf{o})\cdot\mathbf{n}$, $\mathbf{Rot}[\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3]$ is a homogeneous transformation matrix with upper left 3x3 submatrix $[\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3]^\top$ and $\mathbf{Tr}[\mathbf{e}_t]$ is a homogeneous translation matrix with translation component $\mathbf{e}_t$. The z coordinate of $\mathbf{F}(\mathbf{p})$ is dropped, resulting in a 2D point $\mathbf{u}$.

The linear sweep scalar field f_{linear} is then defined as

$$f_{linear}(\mathbf{p}) = f_M\,(\,\mathbf{F}(\mathbf{p})\,)$$

This function f_{linear} defines an scalar field of infinite extent along $\mathbf{n}$. To bound the field, we multiply f_{linear} by g_{wyvill}:

$$f_{inflate}(\mathbf{p}) = g_{wyvill}\left(\frac{|s|}{d_{endcap}}\right)\cdot f_{linear}(\mathbf{p}) \qquad (6)$$

where d_{endcap} determines the width of the falloff region. The width of the implicit surface varies (Figure 1) because f_M has increasing values inside the 2D contour, as can be seen in Figure 13. Larger values of f_{linear} extend further along $\mathbf{n}$, producing a variable-width surface that mimics the inflation techniques of Teddy [IMT99] and other systems.

Equation 6 is computationally expensive because evaluating $f_{\hat{M}}$ is is $O(N)$ in the number of constraint points. The non-zero region of f_M can be discretely approximated using a *field image*. An example is shown in the inset of Figure 13. The field image is sampled in constant time using a C^1 continuous biquadratic reconstruction filter [BMDS02]

The chordal axis techniques used in previous sketch modeling systems [IMT99] can be adapted to create implicit surfaces based on the skeletal primitive approach (Equation 2). However, the resulting scalar field contains C^1 discontinuities which produce unintuitive blending behavior. In addition, this skeletal approach is much slower than the field image-based technique we have described.

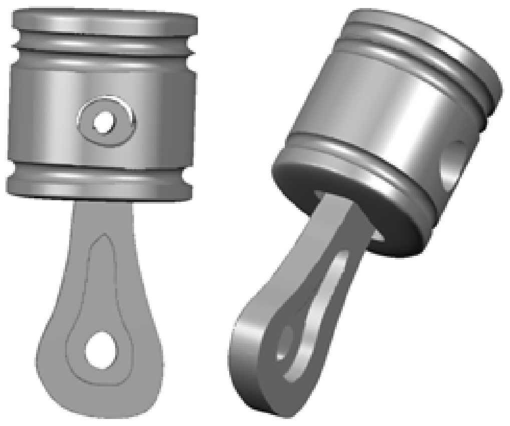


Figure 14: *A Mock-up of a mechanical part sketched with ShapeShop. This model was sketched in under 10 minutes.*

5.3. Sketch Modeling Implementation

Sketch-modeling operations are implemented by replacing the root node of the current BlobTree with a new composition operator. The existing root node is added as the first child, and the new primitive as the second. To implement cutting (Section 3.3), we create a new CSG difference node which subtracts a linear sweep from the current volume. We use a C^1 CSG difference function [BWdG04] which prevents unsightly gradient discontinuities. Blending (Section 3.4) is implemented with the parameterized Hyperblend [Ric73] [WGG99], which affords some control over the blend surface.

To visualize the implicit surface we use an optimized version of Bloomenthal's polygonizer [Blo94]. We provide control over the polygonizer resolution, allowing the user to determine the trade-off between accuracy and interactivity. The images in this paper were all rendered using high-resolution polygonizations which take approximately 5-10 seconds to compute. The Extended Marching Cubes [KBSS01] polygonization algorithm is used to recover sharp features (Figure 14).

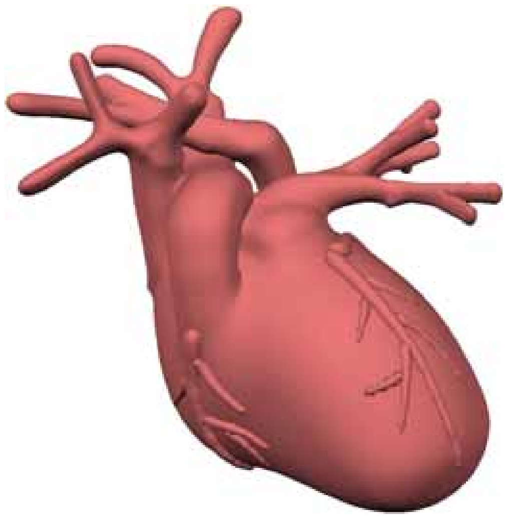


Figure 15: *Heart model sketched in approximately 30 minutes. Complex branching structures can be created quickly by blending simple parts. Our surface-drawing technique is useful for creating anatomical details such as veins.*

6. Results

The benefit of an underlying analytic representation is particularly apparent in CAD-style models (Figure 14). Sharp edges created with CSG are mathematically precise. Since the BlobTree is also a scene graph, the separate parts in this model can be animated. This could, for example, allow an engineer to easily create an interactive assembly manual.

The flexible blending capabilities of implicit modeling are useful when constructing biological models (Figure 15). Since smooth surface transitions are automatic, complex topologies can be assembled quickly from simple parts. The volumetric BlobTree representation supports sketching of internal volumes (Figure 16), which can be applied to biological models to aid in visualization and communication.

Many of the free-form sketch-based systems described in Section 2 have explored character modeling. However, the hierarchical nature of the BlobTree allows our character models to be fully articulated, even when the internal components are blended to form smooth surfaces (Figure 17). These articulated models can be animated directly.

7. Discussion

Hierarchical implicit volume modeling is a useful tool for a wide range of modeling tasks. Employing BlobTrees as an underlying shape representation has allowed us to design an interactive system that supports sketch-based creation of complex 3D models. The models displayed in Figures 14-17 exhibit significantly higher surface complexity than the models demonstrated in existing systems. Further, these models do not indicate the complexity limit of ShapeShop, but rather the point at which these models were considered “finished” by the creator (the primary author).

Only informal observations of graduate students using ShapeShop have been performed. The area that caused the most confusion was selection of non-primitive nodes. Users must understand the hierarchical BlobTree concept, however currently we do not visualize the tree and inferring it’s structure by inspection is difficult. This must be improved in future systems. The 3D transformation widgets were also problematic, we plan on exploring techniques based on those described in the SKETCH system [ZHH96]. Many aspects of the sketch-based interface should be analyzed with formal usability studies.

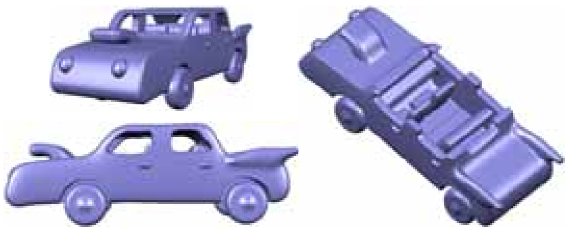


Figure 16: *The body of this car model was initially sketched, and then the internal structure was carved out. Right image shows cut-away view.*

The 2D curve-sketching technique described in Section 4.1 is limited to smooth contours. Integration of methods for adding sharp creases would be beneficial, particularly in the case of CAD models. Techniques such as those described in suggestive sketch-based systems [ZHH96] [JSC03] would also be useful to assist with sketching CAD-style models.

A key property of implicit volume modeling is that composition operators do not depend on the shape of underlying volumes. We have demonstrated that with BlobTrees, CAD-style solid modeling and free-form modeling can be integrated into a single interface. While generality has practical advantages, a more fundamental benefit may come from giving designers a modeling tool which does not prescribe a particular modeling “style”.

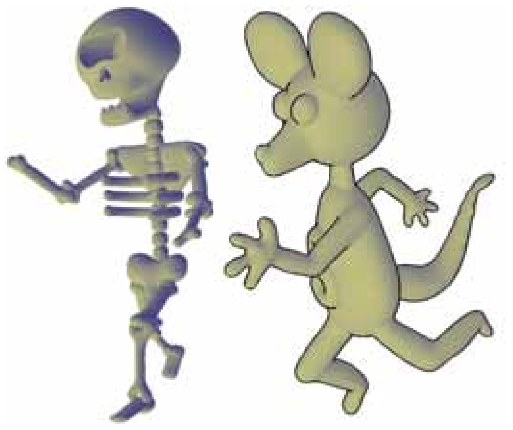


Figure 17: *Character models created with ShapeShop. The skeleton model (left) is composed of 36 primitives in a hierarchical arrangement that is suitable for direct animation.*

Acknowledgements

We would like to thank the anonymous reviewers for their comments and suggestions. This work was supported by the National Sciences and Engineering Research Council of Canada and iCore.

References

[AG04] APITZ G., GUIMBRETIÉRE F.: Crossy: a crossing-based drawing application. In *Proceedings of ACM UIST 2004* (2004), pp. 3–12. 4

[AGB04] ALEXE A., GAILDRAT V., BARTHE L.: Interactive modelling from sketches using spherical implicit functions. In *Proceedings of AFRIGRAPH 2004* (2004), pp. 25–34. 1, 2

[AJ03] ARAÚJO B., JORGE J.: Blobmaker: Free-form modelling with variational implicit surfaces. In *Proceedings of 12th Encontro Português de Computação Gráfica* (2003). 1, 2, 5

[Blo94] BLOOMENTHAL J.: *Graphics Gems IV*. Academic Press Professional Inc., 1994, ch. An implicit surface polygonizer, pp. 324–349. 8

[Blo97] BLOOMENTHAL J. (Ed.): *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., 1997. 6, 7

[BMDS02] BARTHE L., MORA B., DODGSON N., SABIN M.: Interactive implicit modelling based on c^1 reconstruction of regular grids. *International Journal of Shape Modeling 8*, 2 (2002), 99–117. 8

[BWdG04] BARTHE L., WYVILL B., DE GROOT E.: Controllable binary csg operators for soft objects. *International Journal of Shape Modeling 10*, 2 (2004), 135–154. 8

[CBC*01] CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MCCALLUM B. C., EVANS T. R.: Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of ACM SIGGRAPH '01* (2001), pp. 67–76. 5, 7

[CSSJ05] CHERLIN J. J., SAMAVATI F., SOUSA M. C., JORGE J. A.: Sketch-based modeling with few strokes. In *Proceedings of the Spring Conference on Computer Graphics* (2005). 2, 3

[FFJ04] FONSECA M. J., FERREIRA A., JORGE J. A.: Towards 3d modeling using sketches and retrieval. In *Proceedings of the First Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2004). 5

[IH01] IGARASHI T., HUGHES J. F.: A suggestive interface for 3d drawing. In *Proceedings of ACM UIST 2001* (2001), pp. 173–181. 2, 5

[IH03] IGARASHI T., HUGHES J. F.: Smooth meshes for sketch-based freeform modeling. In *Proceedings of the 2003 symposium on Interactive 3D graphics* (2003), pp. 139–142. 1, 2

[IMKT97] IGARASHI T., MATSUOKA S., KAWACHIYA S., TANAKA H.: Interactive beautification: a technique for rapid geometric design. In *Proceedings of ACM UIST '97* (1997), pp. 105–114. 4

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: A sketching interface for 3d freeform design. In *Proceedings of ACM SIGGRAPH 99* (1999), pp. 409–416. 1, 2, 8

[JSC03] JORGE J. A., SILVA N. F., CARDOSO T. D.: Gides++. In *Proceedings of 12th Encontro Português de Computação Gráfica* (2003). 2, 9

[KBSS01] KOBBELT L. P., BOTSCH M., SCHWANECKE U., SEIDEL H.-P.: Feature-sensitive surface extraction from volume data. In *Proceedings of ACM SIGGRAPH 2001* (2001), pp. 57–66. 8

[KHR02] KARPENKO O., HUGHES J., RASKAR R.: Free-form sketching with variational implicit surfaces. *Computer Graphics Forum 21*, 3 (2002), 585 – 594. 1, 2, 3

[MCCH99] MARKOSIAN L., COHEN J. M., CRULLI T., HUGHES J. F.: Skin: A constructive approach to modeling free-form shapes. *Proceedings of SIGGRAPH 99* (1999), 393–400. 2

[ONNI03] OWADA S., NIELSEN F., NAKAZAWA K., IGARASHI T.: A sketching interface for modeling the internal structures of 3d shapes. In *Proceedings of the 4th International Symposium on Smart Graphics* (2003), pp. 49–57. 1, 2, 6

[Ric73] RICCI A.: A constructive geometry for computer graphics. *Computer Graphics Journal 16*, 2 (1973), 157–160. 8

[SW05] SCHMIDT R., WYVILL B.: *Implicit Sweep Surfaces*. Tech. Rep. 2005-778-09, University of Calgary, 2005. 7

[SWG05] SCHMIDT R., WYVILL B., GALIN E.: Interactive implicit modeling with hierarchical spatial caching. In *Proceedings of Shape Modeling International 2005* (2005), pp. 104–113. 1, 7

[TO02] TURK G., O'BRIEN J. F.: Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics 21*, 4 (2002), 855–873. 5, 7

[TZF04] TAI C.-L., ZHANG H., FONG J. C.-K.: Prototype modeling from sketched silhouettes based on convolution surfaces. *Computer Graphics Forum 23*, 1 (2004), 71–83. 1, 2

[WGG99] WYVILL B., GUY A., GALIN E.: Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum 18*, 2 (1999), 149–158. 1, 2, 7, 8

[Wyv05] WYVILL G.: Wyvill function. Personal Communication, 2005. 7

[ZHH96] ZELEZNIK R. C., HERNDON K. P., HUGHES J. F.: Sketch: an interface for sketching 3d scenes. In *Proceedings of ACM SIGGRAPH 96* (1996), pp. 163–170. 2, 4, 9

Creating Geometry from Sketch-Based Input

Correlation-Based Reconstruction of a 3D Object From a Single Freehand Sketch

Hod Lipson[1] **and Moshe Shpitalni**[2]

[1]Cornell Computer Aided Design Lab, Mechanical & Aerospace Engineering, Cornell University, Ithaca NY 14853, USA
[2]NSF Engineering Research Center for Reconfigurable Machining Systems, University of Michigan, Ann Arbor, MI 48109, USA[*]
hod.lipson@cornell.edu

Abstract

We propose a new approach for reconstructing a three-dimensional object from a single two-dimensional freehand line drawing depicting it. A sketch is essentially a noisy projection of a 3D object onto an arbitrary 2D plane. Reconstruction is the inverse projection of the sketched geometry from two dimensions back into three dimensions. While humans can do this reverse-projection remarkably easily and almost without being aware of it, this process is mathematically indeterminate and is very difficult to emulate computationally. Here we propose that the ability of humans to perceive a previously unseen 3D object from a single sketch is based on simple 2D-3D geometrical correlations that are learned from visual experience. We demonstrate how a simple correlation system that is exposed to many object-sketch pairs eventually learns to perform the inverse projection successfully for unseen objects. Conversely, we show how the same correlation data can be used to gauge the understandability of synthetically generated projections of given 3D objects. Using these principles we demonstrate for the first time a completely automatic conversion of a single freehand sketch into a physical solid object. These results have implications for bi-directional human-computer communication of 3D graphic concepts, and might also shed light on the human visual system.

Introduction

In a survey of adequacy of CAD tools for conceptual design (Puttre, 1993), an industrial designer relating to an existing CAD system is quoted saying "The interface is just not for us. I can do thirty sketches on paper by the time it takes me to do two on the computer". Indeed, it is interesting to watch how a designer, when given a 3D design problem, instinctively reaches for a pencil and paper. Despite the abundance of computerized 3D graphic software and CAD systems, raw sketching has remained one of the most useful and intuitive tools at the conceptual design stage. When designing 3D artifacts, user interfaces that deal with spatial construction are typically cumbersome to use and hamper creative flow. Freehand sketching, on the other hand, still provides a fluent method for conveying 3D information among designers, even though it uses an inherently flat medium. Humans seem to be able to understand 3D spatial concepts even when they are depicted on 2D medium in the form of simple and inaccurate line drawings. It is exactly this ability – to understand and generate sketches – that we wish to emulate.

The importance of sketching in design has been a subject of intensive study (Herbert, 1987; Larkin and Simon, 1987; Fang, 1988; Walderon and Walderon 1988; Ullman *et al*, 1990; Jenkins and Martin, 1995). These studies agree that sketching appears to be important for the following reasons:

- It is fast, suitable for the capacity of short term memory,
- It is implicit, i.e. describes form without a particular construction sequence,
- It serves for analysis, completeness check and simulation,
- It is inexact and abstract, avoiding the need to provide unnecessary details,
- It requires minimal commitment, is easy to discard and start anew

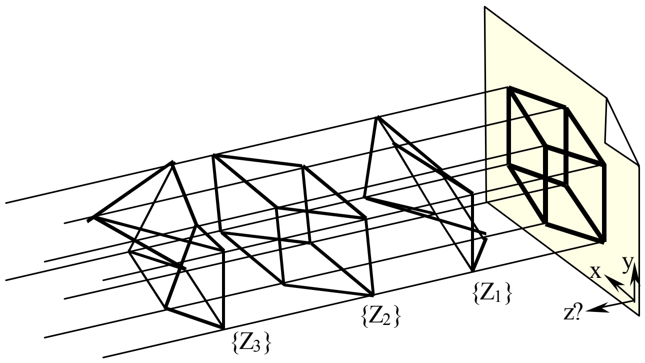


Figure 1: A sketch provides two of the coordinates (the *x,y*) of object vertices. A reconstruction must recover the unknown depth coordinates (*Z*). In parallel projections, these degrees of freedom are perpendicular to the sketch plane; in a perspective projection, they run along lines that meet at the viewpoint (not shown).

[*] On sabbatical leave from the Technion – Israel Inst. of Technology, Haifa, Israel

Background

A sketch is inherently a collection of lines (edges) on a flat surface (paper), representing an arbitrary projection of an arbitrary object. In this work we assume all edges of an object are sketched (wireframe) and are straight lines. We also assume an online source, so that each stroke corresponds to a single edge and edges meet at stroke endpoints. The projection transformation removes the depth information from each vertex of the edge-vertex graph. Consequently, any arbitrary set of depths {Z} that are re-assigned to the vertices of the graph constitutes a 3D configuration whose projection will match the given sketch, and so is a candidate reconstruction (Figure 1).

To recover the lost depth information, a system needs to extract spatial information from the inherently flat sketch. Although this step is mathematically indeterminate, humans seem to be able to accomplish this with little difficulty. Moreover, despite the infinitely many possible candidate objects, most observers of a sketch will agree on a particular interpretation. This consensus indicates that a sketch may contain additional information that makes observers agree on the most plausible interpretation.

There are several reports of methods used to reconstruct a 3D object from multiple views by matching features between different views. However, these approaches are not suitable for analysis of a single sketch. The computer-vision literature also deals extensively with techniques for extracting spatial information from images. These methods typically rely on various depth cues such as shading, lighting, occlusion, shadows, perspective, optical flow, stereo and motion cues. These cues are not available in our problem since we are dealing with a single non-imaging source. We are left with only the raw sketch strokes representing edges of the depicted object.

The literature contains several fundamentally different approaches to interpretation and reconstruction of objects and scenes from single-view line drawings. These are briefly described below. Many reported systems use a mixture of these approaches to enhance their performance.

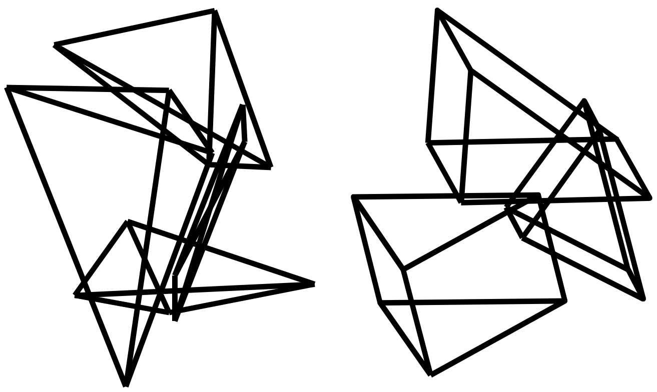


Figure 2: Humans are better at understanding sketches of regular objects (a) A random scene composed of arbitrary polyhedra, (b) A man-made scene composed of right-angled wedges

Line labeling is a form of interpretation of a line drawing; it provides spatial information about the scene but does not yield an explicit 3D representation. Each line in the drawing is assigned one of three meanings: convex, concave, or occluding edge. Junction dictionaries and constraint graphs are used to find consistent assignments (Huffman, 1971; Clowes, 1971, and many works since).

The gradient space approach draws a relationship between the slope of lines in the drawing plane and the gradient of faces in the depicted 3D scene. Assuming a particular type of projection, an exact mathematical relationship can be computed, and possible interpretations of the drawing can be constrained (Mackworth, 1973; Wei 1987).

The linear System approach uses a set of linear equalities and inequalities defined in terms of the vertex coordinates and plane equations of object faces, determined by whether vertices are on, in front of, or behind the polygon faces. The solvability of this linear program is a sufficient condition for the reconstructability of the object (Sugihara, 1986; Grimstead and Martin, 1995). Linear programming optimization may yield a solution.

Interactive methods gradually build up the 3D structure by attaching facets one after the other as sketched and specified by a user. The aim is to provide a practical method for constructing 3D models in an interactive CAD/CAM environment (Fukui, 1988; Lamb and Bandopadhay, 1990).

The primitive identification approach reconstructs the scene by recognizing instances or partial instances of known primitive shapes, such as blocks, cylinders etc. This approach contains a strict assumption that the depicted 3D object is composed entirely of known primitives, but has the benefit of yielding the final 3D structure in a convenient constructive solid geometry (CSG) form (e.g. Wang and Grinstein, 1989).

The minimum standard deviation approach focuses on a single and simple observation; that human interpretation of line drawings tends towards the most 'simple' interpretation. Marill (1991) defined simplicity as an interpretation in which angles created between lines at junctions are as uniform as possible across the reconstructed object, inflating the flat sketch into a regularized 3D object (Leclerc and Fiscler, 1992).

Analytical Heuristics approaches use coded soft geometrical constraints such as parallelism, skewed symmetry and others to seek the most plausible reconstruction (Kanade, 1980; Lipson and Shpitalni, 1996).

Geometric Correlations

The prevailing common assumption we abandon in this work is *generality*: The misconception that humans' remarkable ability to interpret sketches applies in general to arbitrary objects. In a simple experiment we tried a reversal of roles: We let the computer generate sketches of arbitrary 3D scenes, and asked users to interpret those sketches. We observed that when the scene contained random objects (such as tetrahedrons, Fig 2a), subjects were not able to reconstruct the scene correctly at all from a single sketch; however, when the scene depicted man-made objects (such as right-angled wedges, Fig 2b), the scenes were more readily reconstructable. Hence we concluded that the ability of humans to correctly perceive 3D scenes depicted in sketches is not general, but relies on visual experience. We thus offer the alternative reconstruction approach of acquired geometrical correlations: Humans learn correlations between 3D geometry and its corresponding 2D projected pattern. For example, human might learn to correlate 3D tactile geometric information with 2D images projected on their retina. The following section suggests one way of synthetically capturing and using this information.

At the base of our approach is the need to gather correlations between 3D geometry and its corresponding 2D projections. Whereas general geometrical relationships can be derived analytically (Ulupinar and Nevatia, 1991; Ponce and Shimshoni, 1992), these relationships become more elaborate when collected for non-general scenes. We chose to collect these relationships empirically by generating many 3D scenes (like Fig. 2b) and projecting them with noise (normal distribution σ=2% of object width).

We now define a 3D-2D geometric correlation as probability of a certain 2D configuration to represent a certain 3D configuration. For example, consider Figure 3a. The 3D line-pair *AB* creates a 3D angle $\alpha_{3D}=\angle AB$. When the line pair is projected onto the sketch plane, it produces line-pair *ab*. The projected angle is $\alpha_{2D}=\angle ab$. Measuring this correlation over many arbitrary projections of objects in a certain repertoire, we can derive the probability density function *pdf*(α_{3D}, α_{2D}) for that repertoire of objects. We can then use this probability function to determine the likelihood of a candidate reconstruction.

Instead of just measuring angles, we can measure also line lengths. Here we would measure the correlation between length ratio in 3D ρ_{3D}=*A/B* to length ration in 2D ρ_{2D}=*a/b*. Similarly, we might chose to correlate *A/B* with *Ðab*, or *ÐAB* with *a/b*, and so forth. Moreover, we can expand these correlations to third order, by correlating various length-angle relationships among three lines, such as the cone angle of three lines in 3D *A´BˣC* versus the cone angle in 2D *min(aˣb, bˣc, cˣa)*, see Fig 3b.

Higher order correlations may also be recorded in the form of trivariate probability density functions such as *pdf*(α_{3D}, α_{2D}, ρ_{2D}), and even higher orders. In our experiments we used only bivariate probabilities. These were collected for 100,000 random scenes and stored in tables such as those shown in Fig. 4. Note that more efficient correlation memory representations could have been used, such as neural networks or Bayesian networks.

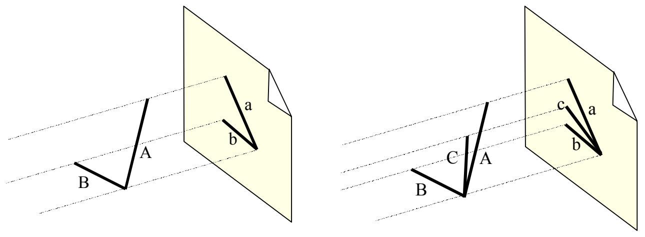


Figure 3: Measuring 2D-3D correlations. (a) second order, (b) third order.

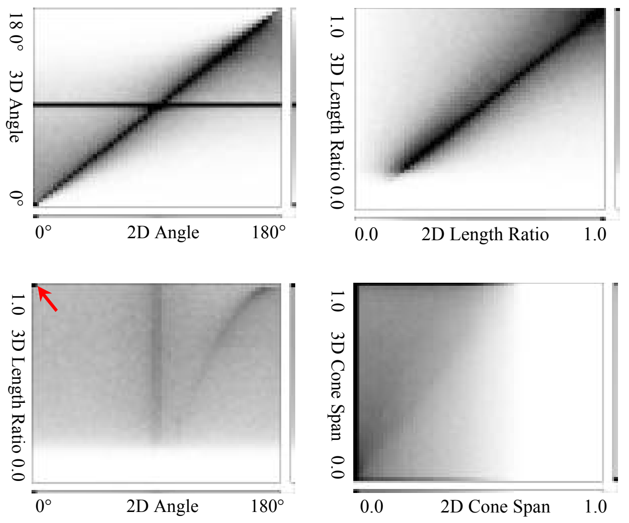


Figure 4: Measuring 2D-3D second order correlations. Dark areas show high correlation. Strips on right and bottom of each table show marginal probabilities. Note dark top-left corner of bottom-left plot.

Once geometric correlation functions are known, it is possible to compute the probability of a particular 3D object being the source of a given 2D sketch. This amounts to measuring a 3D angle α_{3D} of line pairs in the candidate reconstruction, and the corresponding 2D angle α_{2D} in the sketch, and using $p(\alpha_{3D,}\ \alpha_{2D})$ to estimate the probability of α_{3D} given α_{2D}:

$$p(a_{3d}/a_{2d})=\frac{pdf(a_{3d},a_{2d})\cdot da_{3d}\cdot da_{2d}}{pdf(a_{2d})\cdot da_{2d}}$$

where $\delta\alpha$'s are the tolerances of the measurements. This probability is accumulated (multiplied) for all line pairs/triplets in the candidate object and sketch in question

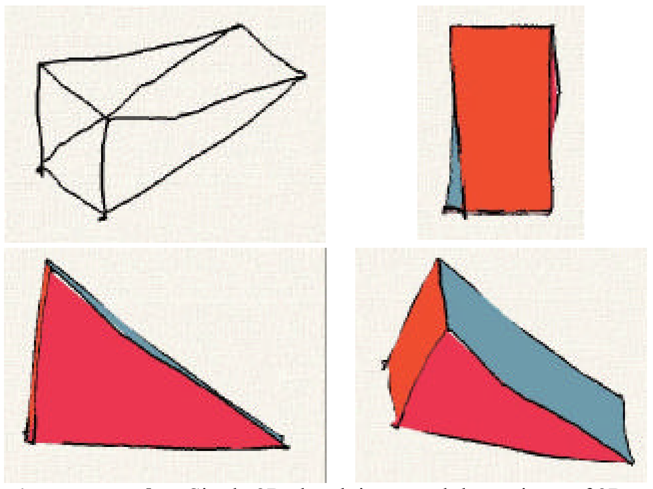


Figure 5: Wedge: Single 2D sketch input and three views of 3D output reconstruction.

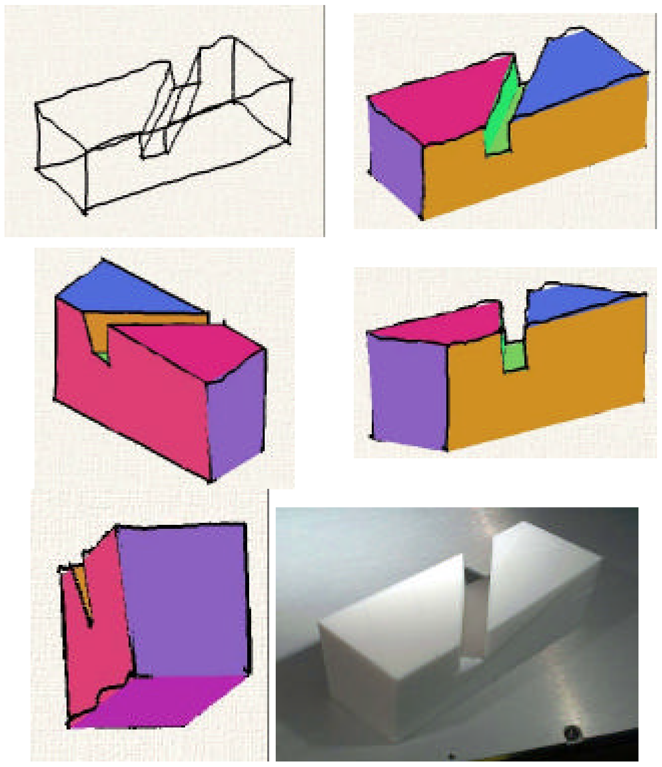


Figure 6: Object "Slot": 2D Single freehand sketch input (top left) and several views of automatically generated 3D reconstruction. Bottom right is automatically generated final physical solid object output. Reconstruction required approx. 5000 hill-climbing cycles (50 Minutes on P500)

over all correlation tables learned, to yield the overall candidate probability.

Once the probability of a candidate reconstruction can be evaluated, then the reconstruction process amounts to an optimization problem, where the objective is to find a set of depth coordinates {**Z**} that maximizes the probability.

However, the optimization process is far from easy. The relatively high degree of coupling among vertices make the optimization landscape rugged with local minima. Moreover, the high dimensionality of the search space (equal to the number of vertices in the sketch minus one) makes brute force search techniques impractical. We have been trying various techniques ranging from straightforward random search to hill-climbing, simulated annealing, and genetic algorithms. Note that for any given solution set **Z**, the inverse solution –**Z** is also equally valid (this is known as the Necker cube illusion). Similarly, the trivial solution **Z**=*constant* also has relatively high probability. This multi-modal nature creates many local optima, and good optimization techniques for this problem still require more research.

The reconstruction step outlined above generates only a 3D wireframe object. In order to complete the transition into a true solid, it is still necessary to identify which of the edge circuits constitute faces of the object, and what is the material side of each face. We use a topological face identification algorithm (Shpitalni and Lipson, 1996) to

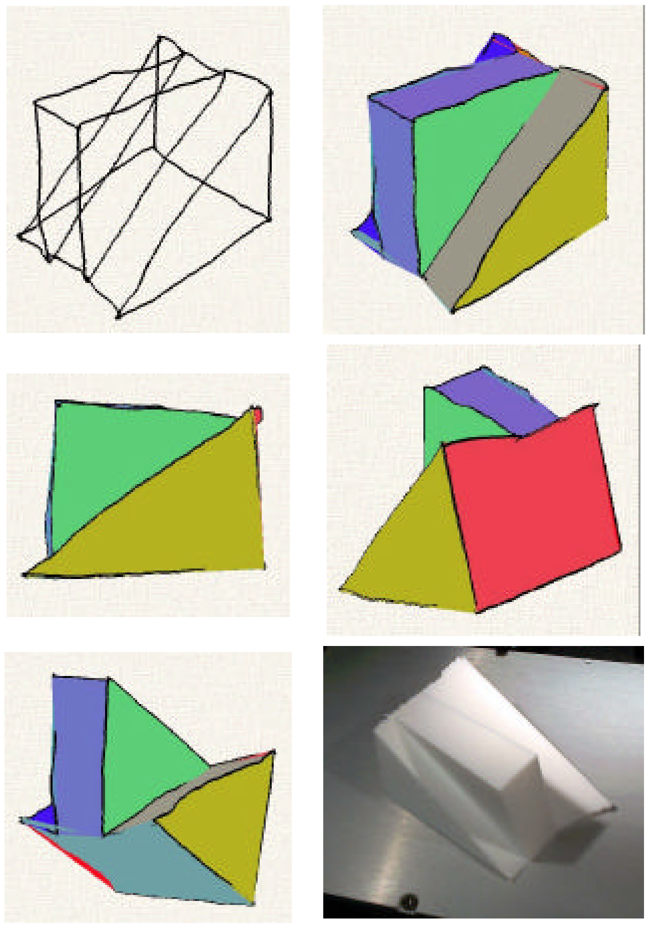


Figure 7: Object "Slide": 2D Single freehand sketch input (top left) and several views of automatically generated 3D reconstruction. Bottom right is final physical solid object output. Reconstruction required approximately 500 hill-climbing cycles (5 Minutes on P500)

mark faces, and then chose outward-pointing normals so that joined faces are consistent and the total object volume is positive. Once the 3D solid model exists, it can be tessellated for rendering and for production using commercial 3D Printing (rapid prototyping). The automatic production of a physical model constitutes the ultimate confirmation of the rigor of the interpretation and its topology.

Results

Since this article is written on a flat paper, resulting 3D reconstructions will be exhibited themselves as 2D drawing, thereby re-creating the very problem they are trying to solve. Nevertheless, we display each 3D solution rendered from multiple viewpoints to make the interpretation clear. Object faces were colored arbitrarily.

First, Fig 5 shows how a sketch of a right-angled wedge is reconstructed into a 3D wedge. This is merely a confirmation, since the geometric correlations were collected for wedge-based scenes. Figure 6 shows how a

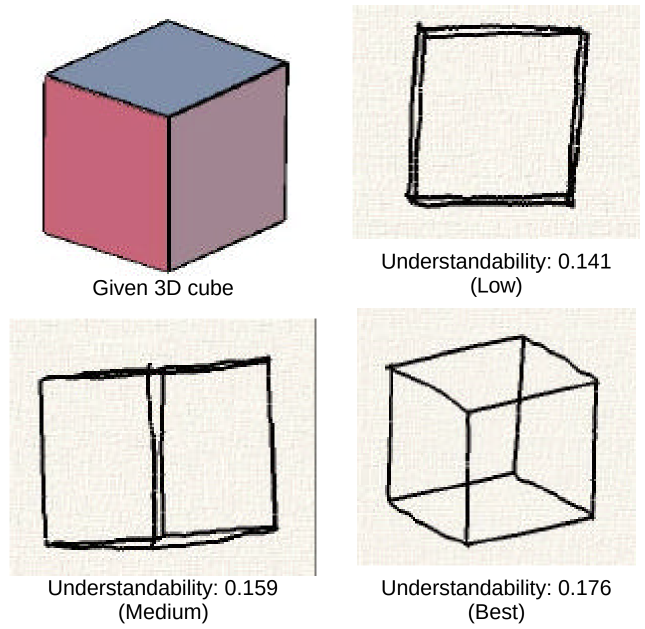


Figure 8: Measuring understandability of sketches: Given 3D Cube (top left), and three projections with varying degrees of correlations, corresponding to degree of "understandability".

more complex structure, not seen by the system during its training period, is reconstructed correctly. Note that the reconstructed object is not accurate – it is a rough 3D object, resembling the roughness of its input. While reconstruction of an accurate 3D model from a rough 2D sketch requires more information (like dimensions and specific constraints), a rough 3D model is useful for many applications. Figure 7 shows an additional example.

Assessing view quality

The correlation tables provide the probability that a 3D model is the source of a 2D sketch. This information can also be used to assess the quality, or "understandability" of a given sketch for a particular 3D scene. This is because a low sketch correlation will make the identification of the correct reconstruction harder. We can thus use the correlation factor as a figure of merit for selecting good sketch viewpoints (Figure 8).

Applications

Enabling a computer to correctly perceive hand-drawn sketches as 3D objects opens an opportunity for new forms of human-computer communication of 3D concepts, and for performing computer aided engineering (CAE) analysis at earlier preliminary design stages (Lipson and Shpitalni, 2000).

Here we investigate the possibility of combining the advantages of non-manipulable pencil-and-paper sketching on real paper, with convenience of 3D manipulation of objects on a computer. Traditionally, a paper sketch is a more fluent medium to describe a 3D object, but once the sketching has commenced, the viewpoint cannot be changed. On a CAD system, viewpoint is easily changed

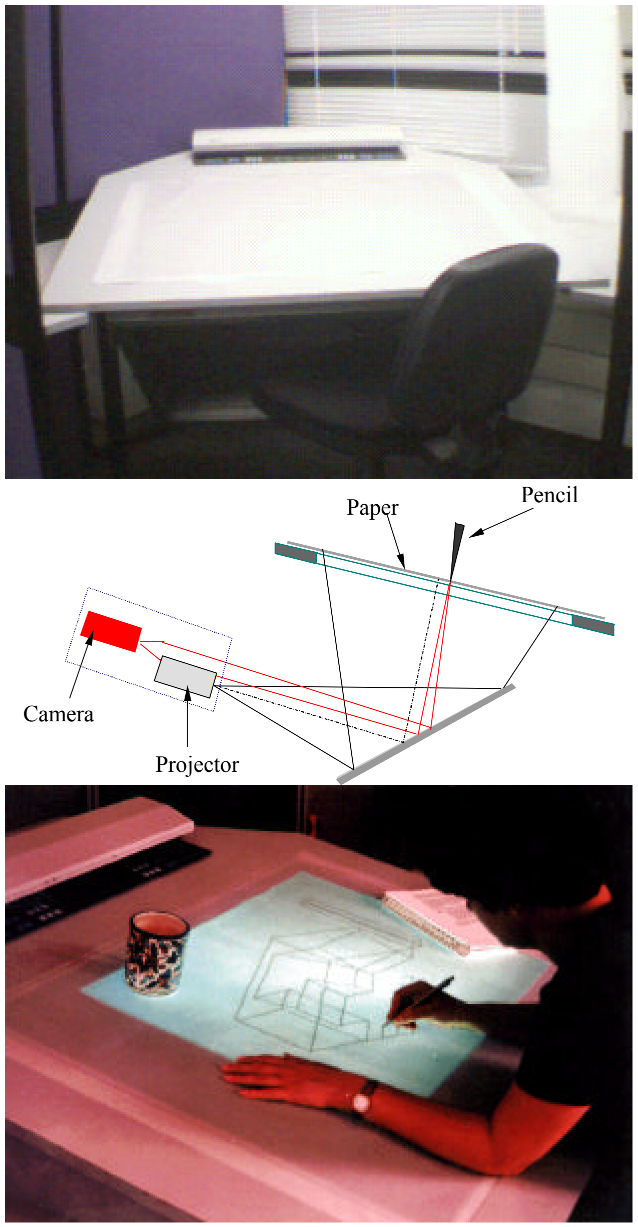


Figure 9: Sketch hardware setup combines natural pencil-and-paper sketching environment but allows changing viewpoint in midst of sketching.

but the direct pencil-and-paper input is lost. However, once a computer is capable of perception of spatial relationships in the sketch, the viewpoint can be changed in midst of sketching. In an experimental system shown in Figure 9 we set up a glass drawing board on which we mounted translucent sheet of paper. The user operated an infrared pen whose motion could be detected by a camera with an IR filter. User's pen strokes on the paper were converted into graphic strokes that were projected back onto the paper. This setup provided an emulation of real pencil-and-paper drawing, while still permitting full interactive ability. After completing partial drawing, the scene could be reconstructed and rotated, so that sketching could be resumed from a different viewpoint. Processing speed, however, still needs to be improved before this can be a truly useful system.

Conclusions

In this paper we have shown how a 2D line drawing can be reverse-projected into three dimensions based on optimizing learned 2D-3D geometric correlations. These correlations are acquired from analyzing many 3D scenes and their corresponding 2D views. Moreover, the same information can be used to judge quality of projections. We demonstrated this approach using four simple second order correlations (angle and length permutations) and one third-order correlation (2D/3D cone angles). We used matrices to store the correlations, and used hill climbing to seek the optimal reconstruction. We demonstrated how this approach could ultimately be used to automatically convert a single rough sketch into a physical solid object without external assistance.

While we appreciate that more advanced correlation representation methods could be used to store higher order correlations more efficiently (e.g. neural networks or Bayesian networks), and more specialized optimization algorithms could enhance our results, we hypothesize that this statistical approach is simpler and perhaps more biologically plausible than traditional constraint-solution techniques proposed to date in the literature. Similarly, while reconstruction time and success rate is still far from being viable for interactive applications, we hope these results may have implications for bi-directional human-computer communication of 3D graphic concepts, and might also shed light on the workings of the human visual system.

References

Clowes M.B., 1971, “On Seeing Things,” *Artificial Intelligence*, Vol. 2(1), pp. 79-112.

Fang R. C., 1988, “2D free hand recognition system”, Master’s report, Oregon State University, Corvallis

Fukui Y., 1988, “Input method of boundary soild by sketching”, *Computer Aided Design*, Vol. 20, No. 8, pp. 434-440

Grimstead I. J., Martin R. R., 1995, “Creating solid models from single 2D sketches”, *Solid Modeling* ’95, Salt Lake City, Utah, USA, pp. 323-337

Herbert D., 1987, “Study drawings in architectural design: Applications of CAD systems”, in Proceedings of the 1987 workshop of the association for computer aided design in architecture (ACADIA)

Huffman D.A., 1971, “Impossible objects as nonsense sentences,” Machine Intelligence, pp. 295-323, Edinburgh University Press, Edinburgh, B. Meltzer and D. Michie, eds.

Jenkins D. L., Martin R. R., 1993, “The importance of free hand sketching in conceptual design: Automatic sketch input”, ASME Conference on Design theory and Methodology (DTM’93), DE-Vol 53, pp. 115-128

Kanade T., 1980, “Recovery of the three-dimensional shape of an object from a single view” *Artificial Intelligence* Vol. 17, pp. 409-460

Lamb D., Bandopadhay A., 1990, “Interpreting a 3D Object from a Rough 2D Line Drawing,” *Proceeding of Visualization* ’90, pp. 59-66.

Larkin J., Simon H., 1987, “Why a diagram is (sometimes) worth a thousand words”, *Cognitive Science,* Vol. 11, pp. 65-99

Leclerc Y. G., Fiscler M. A., 1992, “An optimization based approach to the interpretation of single line drawings as 3D wire frames” *Int. J. of Computer Vision* Vol 9 No 2 pp. 113-136

Lipson H., Shpitalni M., 1996 “Optimization-Based Reconstruction of a 3D Object From a Single Freehand Line Drawing,” *Journal of Computer Aided Design,* Vol. 28 No 8, 651-663.

Lipson H, Shpitalni M., 2000, “Conceptual design and analysis by sketching”, *Journal of Artificial Intelligence in Design and Manufacturing* (AIDAM), Vol. 14, pp. 391-401.

Mackworth A.K., 1973, “Interpreting Pictures of Polyhedral Scenes,” Artificial Intelligence, Vol. 4, pp. 121-137.

Marill T., 1991, “Emulating the human interpretation of line drawings as three-dimensional objects” *Int. J. of Computer Vision* Vol. 6 No. 2, pp. 147-161

Ponce J., Shimshoni I., 1992, “An algebraic approach to line drawing analysis in the presence of uncertainty”, Proceedings of the 1992 IEEE *Int. Conf. On Robotics and Automation*, Nice, France, pp. 1786-1791

Puttre M., 1993, “Gearing up for conceptual design”, *Mechnical Engineering*, March 93, pp. 46-50

Shpitalni, M. and Lipson, H., 1996, "Identification of Faces in a 2D Line Drawing Projection of a Wire frame Object", *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI), Vol. 18, No. 10, pp. 1000-1012

Sugihara K., 1986, *Interpretation of Line Drawings*, The MIT Press.

Ullman, D.G., Wood, S., Craig, D., 1990, “The Importance of Drawing in the Mechanical Design Process,” *Computers & Graphics*, Vol. 14 No. 2, pp. 263-274

Ulupinar F., Nevatia R., 1991, “Constraints for interpretation of line drawings under perspective projections”, *Computer Vision Graphics Image Processing* (CVGIP): Image Understanding, Vol. 53, No. 1, pp. 88-96.

Walderon M. B., Walderon K. J., 1988, “Conceptual CAD tools for mechnical engineers”, in Patton E. M. (Ed.), *Proceedings of Computers in Engineering Conference*, Vol. 2, pp. 203-209, Computer and Graphics, 1988

Wang W., Grinstein G., 1989, “A polyhedral object’s CSG-Rep reconstruction from a single 2D line drawing,” Proc. Of 1989 *SPIE Intelligent Robots and Computer Vision III: Algorithms and Techniques*, Vol. 1192, pp. 230-238.

Wei X., 1987, “Computer Vision Method for 3D Quantitative Reconstruction from a Single Line Drawing,” PhD Thesis, Department of Mathematics, Beijing University, China (in Chinese; for a review in English see Wang and Grinstein, 1993).

ELSEVIER

0010-4485(95)00081-X

Computer-Aided Design, Vol. 28, No. 8, pp. 651–663, 1996
Copyright © 1996 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0010-4485/96 $15.00+0.00

Optimization-based reconstruction of a 3D object from a single freehand line drawing

H Lipson and M Shpitalni

This paper describes an optimization-based algorithm for reconstructing a 3D model from a single, inaccurate, 2D edge-vertex graph. The graph, which serves as input for the reconstruction process, is obtained from an inaccurate free-hand sketch of a 3D wireframe object. Compared with traditional reconstruction methods based on line labelling, the proposed approach is more tolerant of faults in handling both inaccurate vertex positioning and sketches with missing entities. Furthermore, the proposed reconstruction method supports a wide scope of general (manifold and non-manifold) objects containing flat and cylindrical faces. Sketches of wireframe models usually include enough information to reconstruct the complete body. The optimization algorithm is discussed, and examples from a working implementation are given. Copyright © 1996 Elsevier Science Ltd.

Keywords: drawing to model, optimization, robustness

INTRODUCTION

Despite significant developments in the field of computer-aided design, conceptual designers still tend to prefer pencil and paper to CAD systems. This tendency is especially noticeable in the preliminary stages of product formation, when a design is nothing more than a collection of abstract ideas. The reason most often given for not using a computer at this stage of conceptual design is that the interface is not suitable for sketching very basic ideas. Although the interface is user-friendly, it still seems to disturb the designer's flow of ideas and to interfere with creativity and concentration.

A new interface for conceptual design based on 3D object reconstruction from a single 2D sketch was introduced by the authors[1]. The system concept is illustrated in *Figure 1*. The system analyses and interprets single 2D inaccurate line drawing (projection) input and reconstructs the 3D object most likely to have been selected by a human observer. Once the 3D model is obtained, it can be manipulated or modified, and further detail can be sketched in, perhaps from a more convenient point of view. The output of the system is an abstract 3D model that was constructed without any commands or keyboard input. Although this model does not have exact dimensions, it is sufficient to convey the design concept to other designers or to a conventional CAD system. The proposed approach is intended to provide a designer with the means to convey his ideas to a CAD system in a way similar to the way in which designers communicate among themselves.

Much work has been devoted to reconstructing 3D information from a 2D source. The research presented in this paper investigates the ability to reconstruct the 3D model of an object depicted by a 2D freehand sketch.

The line drawing is assumed to represent a projection of a general object, in wireframe representation, as seen from some arbitrary point of view. Since the 2D line drawing lacks depth information, the reconstruction process is non-deterministic. A simple line drawing can represent the projections of an infinite number of possible 3D objects (*Figure 2*). Interpretation heuristics regarding regularities in the image, however, allow additional implicit information to be extracted from the line drawing and make it possible to reconstruct the object most likely represented by the input drawing.

The reconstruction algorithm proposed and discussed in this paper is executed after several preprocessing stages in which the initial 2D rough sketch is transformed into a 2D *line* and *junction* graph. In this graph, each line is assumed to correspond to the projection of exactly one *edge* of the depicted 3D object and each junction of lines to one *vertex* of the object. The graph is 2D and contains no information regarding the 3D object represented, its type or its position with respect to the viewpoint.

The preprocessing stage is described in detail in References 1 and 2. In brief, the initial (raw) sketch strokes are smoothed, classified into geometrical entities and then linked together at their end-points to form a projected topological edge–vertex graph. The original sketch is obtained from an on-line sketching interface, and it is assumed that each edge is drawn as a continuous

Laboratory for Computer Graphics and CAD, Faculty of Mechanical Engineering, Technion, Haifa 32000, Israel
Paper received: 8 July 1995. Revised: 4 November 1995

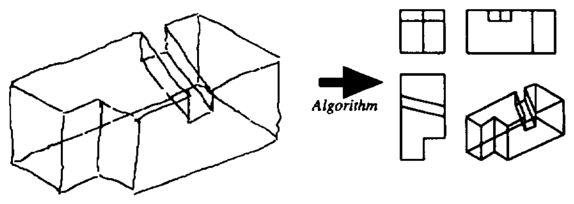


Figure 1 Reconstruction of a 3D object from a single 2D line drawing

sketch stroke. This assumption facilitates the automatic distinction between real vertices and accidental crossing of lines. Then, faces of the depicted 3D wireframe object are determined in the 2D topological graph. Because general objects are considered (both manifold and non-manifold, possibly including cylindrical faces), the face identification process is complex. A separate paper by the authors[2] dedicated to this subject has been submitted.

In this paper, the motivation for this work is discussed in the introduction. Next, related work is reviewed. Then, the requirements are set, and the reconstruction problem and underlying assumptions are discussed. The reconstruction method is then developed, and examples from a working implementation are given.

PROBLEM DEFINITION AND ASSUMPTIONS

Input to the reconstruction algorithm is a single 2D projection of some 3D object in the form of a *2D line–junction* graph. The graph originated from a previous rough freehand sketch and may still contain inaccuracies both in position of vertices and in connectivity of edges. The goal of the algorithm is to restore the original *rough* 3D object using information derived from the projection alone. It should be noted again that at this stage the graph is still *two-dimensional* and represents a projection of a 3D object. Furthermore, no additional information is available regarding the 3D object itself, its type or its position with respect to the viewpoint.

The problem can be defined more precisely by setting out a number of assumptions and requirements.

Assumptions

- The input to the system consists of a single 2D line drawing only, which is given as a graph of connected entities. (This graph may be the result of previous processing of a raw input sketch.)
- The input projection represents a wireframe model of a general object that may be manifold, non-manifold or an assembly of such objects. However, no information is provided to the system about the 3D object itself, its type or its position relative to the viewpoint.
- The projection is drawn from a general—non-accidental—viewpoint that reveals all edges and vertices. That is, none of the edges or vertices of the object coincide accidentally, and none of them accidentally appear to be joined in the projection. This assumption also requires that the topology of the projected edge–vertex graph will not change if the viewpoint is perturbed slightly within its neighbourhood. (Although this requirement seems limiting, it complies with regular practice in engineering communication.)
- All drawn lines and curves in the projection represent real edges, silhouette curves or intersections of faces in the 3D object. No shadow lines or surface marks are allowed.
- The sketch is assumed to depict the object in a parallel (or nearly parallel) projection. Although a perspective projection could convey more spatial information, this source relies too heavily on the user's sketching skills and is therefore prone to inaccuracies.

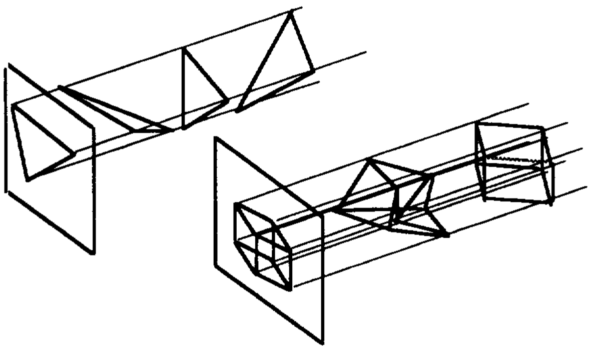


Figure 2 A single 2D sketch can be a projection of an infinite number of 3D objects

Requirements

- The algorithm must arrive at the most plausible reconstruction. A 2D sketch might be a projection of an infinite number of possible 3D objects, as is illustrated in *Figure 2*. It is therefore impossible to uniquely identify the source object. Human observers, however, seem to have quite definite ideas regarding the 3D object that gave rise to a particular projection. The reconstruction, then, must arrive at the *most plausible* 3D object described by a given projection, that is, the object that human observers are most likely to select.
- The algorithm may not consult the user at any point during the reconstruction (non-interactive execution).
- The algorithm is required to arrive at a *rough* 3D reconstruction. For example, if a sketch of a cube is given, the algorithm is required to produce a rough 3D cube where: (1) dimensions bear a rough resemblance to those of the sketch; (2) lines are approximately perpendicular; and (3) faces are approximately planar (see examples in *Figure 12*). Although this requirement is somewhat 'loose', a user can easily determine if the computer has come up with the 'correct' solution, even if it is rough. To be more specific, the resulting reconstruction must exhibit correct 3D topology (i.e. relative position of faces, lines and vertices, such as above, below, between, adjacent, intersecting, etc.) and approximate geometrical qualities (parallelism, perpendicularity, proportional lengths, symmetry, etc.). Currently, there is no accurate or absolute measure of the correctness of a reconstruction. A classical example is the question of *generic reconstructibility* addressed by Sugihara[3]. The sketch shown in *Figure 3a* is *not* mathematically reconstructable with planar faces, since the extended three lines do not meet at

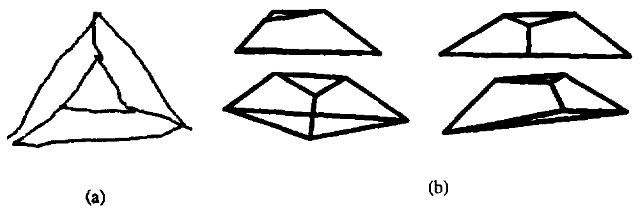


Figure 3 (a) Original sketch of an object unable to be reconstructed generically. (b) Views of the reconstructed 3D object (2 sec)

the same point. However, observers of the sketch immediately understand it to represent a truncated pyramid with planar faces. It is exactly this capability that we wish to emulate. Indeed, the reconstruction shown in *Figure 3b* does reflect the correct notion of the object, though not strictly accurately. For this reason, the reconstruction technique does not rely, at any point, on exact solution of equations, as in previous approaches. In fact, this tolerance is what makes the reconstruction possible. Some tests showed that forcing *strict planarity* on a model originating in an inaccurate sketch produces reconstructions which are not plausible. For conveying qualitative conceptual information, this kind of reconstruction will suffice. Clearly, for the reconstructed object to be of further use in detail design, it must be transformed into an accurate model by means of further information, such as explicit dimensions and a variational geometry process. This stage is currently under research.

- The procedure for identifying the most probable solution must be able to tolerate faults due to: (a) inaccurate positioning of vertices; (b) inaccurate connectivity between entities; and (c) missing or excess entities. In fact, this flexibility is a key principle in determining whether the proposed reconstruction algorithm can be used in practice. Freehand sketches are often rough and inaccurate; it is actually the inaccurate nature of the sketch that makes it a fast, comfortable and natural communication language for conveying conceptual ideas. In order to preserve this important quality, fault tolerance is required. Similarly, engineers may express a geometrical concept by occasionally eliminating or adding some lines beyond the correct technical drawing regulations. Although these deviations may render the drawing topologically incorrect, they often do not impede an observer's ability to understand them, as is illustrated in the two incorrect drawings in *Figure 4*. We wish to emulate this tolerance.

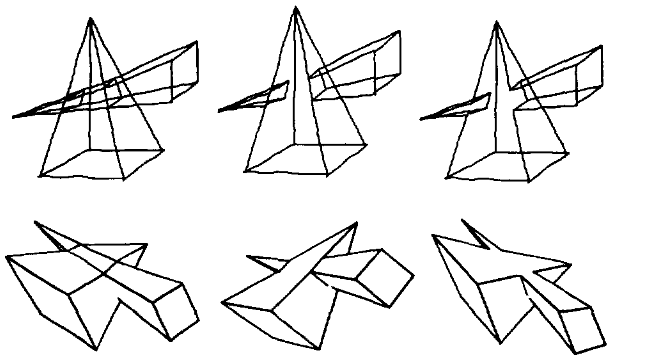


Figure 4 Different (incorrect) versions of sketches of an intersection of two pyramids, and (correct) 3D reconstructions

RELATED WORK

Much work has been done on reconstruction of a 3D object from three (and sometimes two) orthogonal projections; some are surveyed in Reference 4. Most of these methods rely on correlating between different views while relying on their mutual orthogonality for deriving spatial information. Since a single projection is given, these works are not applicable here. A more elaborate survey of 3D reconstruction methods from both single views and multiple views is provided in Reference 5.

Single view scenes with hidden lines *removed* have been studied qualitatively by Huffman[6] and Clowes[7] and quantitatively by others, e.g., Kanade[8] and Sugihara[3], resulting in various line labelling schemes based on junction libraries. According to these techniques, all line segments in a drawing are labelled either as a concave or convex junction of faces or as an edge of an occluding face. A set of consistent line labels is searched for in a library of possible junction configurations. Several such sets may be generated. In general, however, line labelling procedures serve for drawing *interpretation* rather than actual *reconstruction* of the depicted 3D scene. Kanade[8] and Sugihara[3] perform 3D reconstruction using additional information derived from image intensities or geometrical regularities consistent with a chosen labelling set. Line labelling methods, however, are not suitable for handling inaccurate drawings and possible missing entities. A missing line may render a complete drawing insoluble. It should be noted that these works are confined to scenes with hidden lines removed (thus simplifying the face identification and line labelling processes), and polyhedral object domain (thus limiting the possible edge and face configurations).

An optimization-based approach for reconstruction from an accurate view has been suggested by Marill[9]. In his work, the depth of vertices of a body represented in a line drawing projection is optimized until the minimum-standard-deviation-of-angles (MSDA) between line pairs at junctions is reached. This approach has been amended by Leclerc[10] to consider face planarity as well; however, these two criteria alone appear to be sufficient for reconstructing only a limited set of object types. An interactive reconstruction system based on line labelling has been implemented by Lamb *et al.*[11] Their system tries to identify the principal axis, and uses symmetry detection and user intervention in cases of ambiguity. Marti *et al.*[12] have dealt with interpretation of sketches of origami-world objects with hidden lines visible. The interpretation is performed by preprocessing a scanned paper sketch and then using line labelling procedures. A recent work by Grimstead *et al.*[13] constructs a trihedral object from an orthographic view with hidden lines removed. The reconstruction is based on line labelling and iterative least-squares solution of a system of equations derived from line labels and image regularities. Work on the general application of machine interpretation of line drawings including non-geometrical data is given by Dori[14].

This work attempts to provide a basis for reconstructing a larger variety of object types depicted in inaccurate sketches with hidden lines visible, as is often the case in mechanical engineering drawings.

THE PROPOSED APPROACH

The key principle in the reconstruction process is the perception that although the input edge–vertex graph itself is merely 2D, it does hold some implicit spatial information. This implicit information enables a human observer to have a 'feel' for the 3D object depicted by the graph. The implicit 3D information originates from three sources:

- image regularities;
- face topology; and
- statistical configuration of entities.

In the following sections, image regularity and statistical configuration sources of information are described. For a detailed description of face topology, refer to Reference 2.

Image regularities

Image regularities are special geometrical relationships between individual entities or within groups of entities. The heuristic rule is that the image regularities do not appear in the drawing accidentally, but rather correspond to some real geometrical regularities existing in the 3D object[8]. An example of a typical regularity is *parallelism*. The heuristic rule for the *parallelism* regularity is that if two lines are parallel in the sketch plane, they probably represent parallel lines in the 3D object, although mathematically this is not necessarily the case. This heuristic rule has a sound statistical basis: two crossing lines in space will appear parallel only when viewed from a limited scope of viewpoints. Parallel lines in space, however, will appear parallel when viewed from *any* viewpoint. Hence, if two lines are parallel in the sketch plane, it is *more likely* that they represent true spatially parallel lines. Most of the following image regularities are based on similar grounds. The notion of image regularities is so deeply rooted in the human visual system that an image failing to comply with them often perplexes the viewer. An excellent example is found in M C Escher's puzzling drawings, where some of the scenes contain parallel lines that do not correspond to parallel lines in the 3D scene.

A 3D reconstruction is now sought by moving the vertices of the graph in the depth direction (z) perpendicular to the drawing plane while retaining their projected position (x, y). The vertices are moved to conform with image regularities detected in the given 2D edge–vertex graph.

Some image regularities have been used (e.g. by Kanade[8]) to create a set of equations, which are then solved to obtain the 3D object from an accurate drawing. This approach, however, has been found inappropriate for practical use when dealing with inaccurate drawing sources. The basic problems with regularities in such cases are: (a) they do not *necessarily* represent a 3D relationship because they might be accidental; and (b) in an inaccurate drawing where parallel lines, for instance, are not exactly parallel in the drawing, it may be difficult to detect these regularities with certainty. As a result of these issues, the information available in image regularities is used as soft constraints which do not necessarily require full compliance. In the proposed algorithm, a set of soft constraints has been constructed by checking all regularities over all the combinations of entities in the topological graph. Each constraint is weighted according to the probability that the regularity detected in the sketch indeed represents a 3D geometrical relationship. For a proposed 3D reconstruction, the regularities are evaluated using mathematical terms, multiplied by their weight-coefficient and summed to produce an overall compliance function. This compliance function estimates how well the specific 3D reconstruction conforms with the regularities identified in its 2D projection. A 3D configuration of vertices is then searched for by optimizing the compliance function. This approach tolerates inaccuracies while arriving at a reconstruction that will approximately satisfy *most* of the regularities. All applications seek to bring the compliance function to an extremum; however, different weighted compositions of regularities in the compliance function may be necessary for different types of sketches. The composition used here is suitable for general purposes (all the examples in *Figure 12* have been computed with the same setting). Better compositions for a more specific object domains can be found. The success of the optimization procedure depends on the variety of regularities used, their mathematical formulation, and the optimization algorithm itself. After a discussion of consistency of interpretation, the regularities are described and discussed.

Consistency of interpretation

An important principle in the reconstruction process is that interpretation remains invariant despite small changes in the input sketch. An infinitesimal change in a sketch will not make a human observer interpret the sketch differently. Consider the following example: one of the important geometrical relationships between two entities in a sketch is parallelism. Some sketchers in existing CAD systems use 7° of angular difference as a threshold for determining parallelism, implying that an angular difference of 7.0001° would be interpreted as *not* parallel. Such behaviour directly conflicts with the consistency of interpretation principle. To avoid this, a *continuous* compliance factor $\mu(a)$ is defined, where a is the angular difference between the two lines in question. The compliance factor μ ranges from 1.0 for exact parallelism ($a = 0$) and descends to zero like a standard normal distribution curve with $\sigma = 7°$, as a approaches 90°. In computations of the target function, the parallelism criterion would be weighted according to its compliance factor. Intuitively, it could be said that 'the more the lines are parallel in the sketch plane, the more they are required to be parallel in space'. Thus, by avoiding a clear cut threshold, a small change in the angle of a line would not cause a step difference in interpreting parallelism. A more general formulation of the function $\mu(x)$ is given by:

$$\mu_{a,b}(x) = e^{-((x-a)/b)^2} \qquad (1)$$

where x represents the value to be checked, a is a nominal value (e.g. 90° for perpendicularity), and b is a reasonable deviation (e.g. 7°).

Here $\mu(x)$ has been defined so that it evaluates to 1.0 when $x = a$ exactly, and degenerates to 0.0 like a standard distribution curve with $\sigma = b$ as x retreats from a. For practical purposes, Equation 1 has been modified to eliminate values close to zero that may otherwise be weak regularities. That is:

$$\mu_{a,b}(x) = \max\left[0, 1.1 \cdot e^{-((x-a)/b)^2} - 0.1\right] \tag{2}$$

The inclusion of the consistency principle is a significant contribution to the robustness of the interpretation.

Formulation of image regularities

Image regularities are heuristic in their nature and can be classified into three groups.

(1) Regularities reflecting some spatial relationship among individual entities, for example, parallelism between two entities is likely to reflect parallelism in 3D.
(2) Regularities reflecting some spatial relationship among a selected group of entities, such as entities in a contour or a chain of entities, for example, skewed symmetry of a projected contour is likely to represent a symmetrical contour in 3D.
(3) Regularities affecting all the entities in the drawing. For example, the isometry regularity requires that the 3D length of entities will be uniformly proportional to the length of their projection.

Each regularity is expressed by a mathematical term and is included in the compliance function. The term evaluates to a value denoting how well a specific image regularity agrees with its associated 3D configuration. Since the compliance function serves as a target for minimization, the terms included in it must be formulated to exhibit uniform mathematical behaviour. Consequently, three guiding principles have been set:

(1) All terms must vanish for complete compliance and must evaluate to approximately n for clear non-compliance, where n is the number of vertices participating in the evaluation of the term. Consequently, whenever a vertex is repositioned, its influence is proportional to the number of constraints it affects. It is not mathematically possible to make values of different units comparable, for example, units of length can take arbitrary magnitudes compared with units of angular deviation. In our case, however, the sketch dimensions, in pixels, are limited and are typically around half of the screen width. When such limitations exist, all units can be normalized to become dimensionless and can be successfully weighted and compared.
(2) The terms must evaluate the square of the linear deviation from compliance. To achieve linearity, inverse trigonometric functions are often used to compensate for non-linearity of vector operations. Although these functions may slow the computation, they ensure that changes in deviation uniformly affect the model *compliance function*.
(3) The terms must include a weighting factor to accommodate uncertainty in identifying regularity due to sketch inaccuracies.

The following notation is used:

a	represents the criteria value. These criteria are summed up and used as a minimization target;
w	represents the weight given to a constraint, based on the dominance of its associated regularity;
$\mathbf{v}, \mathbf{v}'$	are vectors that, respectively, represent a vertex belonging to a 3D object and its 2D projection in the sketch projection;
$\mathbf{l}, \mathbf{l}'$	are unit vectors that, respectively, represent a direction in 3D and in the sketch plane.

Following is a list of image regularities. Each regularity includes an observable geometrical relationship in 2D and the associated configuration presumed in 3D. The mathematical terms used to evaluate the regularity and its weighing coefficient are given.

Face planarity

A face contour consisting entirely of straight lines reflects a planar face in 3D. The evaluation of this condition is performed in two stages: first, the best-fit surface for the contour vertices is found; then, the deviation of each vertex from that surface is computed, squared, and summed.

The best-fit plane is assumed to be in the form:

$$ax + by + cz + d = 0 \tag{3}$$

The plane coefficients a, b, c are computed by solving the linear system using the given list of points $(x_i; y_i; z_i$ $i = 1, \ldots, n)$ lying on the plane, and arbitrarily assuming $d = 1$.

$$\Sigma \begin{bmatrix} x_i^2 & x_i y_i & x_i z_i \\ x_i y_i & y_i^2 & y_i z_i \\ x_i z_i & y_i z_i & z_i^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \Sigma \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{4}$$

The coefficients are then normalized by having $\sqrt{a^2 + b^2 + c^2} = 1$ with d scaled appropriately and the deviation of a point from the plane taken as the absolute value $|ax_i + by_i + cz_i + d|$.

Note that in order to use this regularity, a preprocessing stage is necessary in which edge-circuits corresponding to the faces of the depicted 3D object are identified *in the 2D graph*. This process is described in detail for general (non-manifold) objects in Reference 2.

Other methods exist for evaluating the planarity of a set of points in space. However, methods relying on checking the planarity of every four-point sequence around the contour (e.g. Leclerc[10]) are not sufficient because local planarity of contour points does not ensure its global planarity. Such a situation may occur whenever there is a sequence of three collinear points.

Line parallelism

A parallel pair of lines in the sketch plane reflects parallelism in space. The term used to evaluate the parallelism of a line pair is

$$\begin{aligned} \alpha_{\text{parallel}} &= w_{1,2}[\cos^{-1}(\hat{\mathbf{l}} \cdot \hat{\mathbf{l}}_2)]^2 \\ w_{1,2} &= \mu_{0^\circ,7^\circ}(\cos^{-1}(\hat{\mathbf{l}}'_1 \cdot \hat{\mathbf{l}}'_2)) \end{aligned} \tag{5}$$

where $\hat{\mathbf{l}}_1$ and $\hat{\mathbf{l}}_2$ are the unit direction vectors of the first and second lines, respectively. The weight $w_{1,2}$ given to this regularity for a specific pair of lines depends on how the two lines are parallel in the sketch plane.

Line verticality

A line that is vertical in the sketch plane (parallel to the *y* axis of the drawing page) is 'vertical in space', i.e. its two end-points have similar depth (*z*) coordinates. The term used to evaluate the verticality of a line is

$$\alpha_{\text{vertical}} = w_l[\cos^{-1}(\hat{\mathbf{l}}_y)]^2$$
$$w_l = \mu_{0^\circ,7^\circ}(\cos^{-1}(\hat{\mathbf{l}}'_y)) \tag{6}$$

where $\hat{\mathbf{l}}_y$ is the *vertical* component of the line's direction vector and $\hat{\mathbf{l}}'_y$ is its *vertical* component in the sketch plane.

Isometry

Lengths of entities in the 3D model are uniformly proportional to their lengths in the sketch plane. The term to account for non-uniformity corresponds to the standard deviation of scales.

$$\alpha_{\text{isometry}} = n \cdot \sigma^2(r_{i=1,\dots,N_e}) \qquad r_i = \frac{\text{length (entity}_i)}{\text{length}'(\text{entity}_i)} \tag{7}$$

where *n* is the number of entities, r_i is the ratio between the current length of entity *i* and its length in the sketch plane, and *s* is the standard deviation of the series of r_i.

Corner orthogonality

A junction of three lines that mathematically qualifies as a projection of a 3D orthogonal corner *is* orthogonal in space. To determine whether a junction of three lines in a plane qualifies as a projection of an orthogonal corner, the following test is applied, based on the fact that the projection of an orthogonal corner spans at least 90°. A junction of three lines has eight variants, created by flipping the direction of each line and considering the eight resulting permutations (see *Figure 5*). The eight variants of the junction *in the sketch plane* are tested; for each variant, three lines exist $\mathbf{l}'_{i=1,\dots,3}$, forming three pairs between themselves, $\mathbf{l}'_{i=1,2}$, $\mathbf{l}'_{i=2,3}$, $\mathbf{l}'_{i=3,1}$. Each line is described by a 2D unit direction vector $\hat{\mathbf{l}}'_i$ in the sketch plane, pointing from the junction outwards. If a junction variant spans less than 90° (i.e. is not a projection of an orthogonal corner), all of the three dot-products of its direction-vector pairs will be positive. If a three-line junction is a projection of an orthogonal corner, all of its eight variants must span at least 90°. Thus, if any one of the eight variants appears to span less than 90° (shows such an 'all-positive' condition), the tested junction is unlikely to represent an orthogonal corner. Consequently, the term used to evaluate the corner orthogonality condition is

$$\alpha_{\text{corner}} = w_{\text{corner}} \sum_{\text{Pair}=1}^{3} [\sin^{-1}(\hat{\mathbf{l}}_1 \cdot \hat{\mathbf{l}}_2)]^2 \tag{8}$$

$$w_{\text{corner}} = \begin{cases} 1 & \text{if } \beta \le 0 \\ \mu_{0,0.1} & \text{if } \beta > 0 \end{cases}$$
$$\beta = \max_{8\text{ variants}} \left[\min_{3\text{ pairs}} (\hat{\mathbf{l}}'_a \cdot \hat{\mathbf{l}}'_b) \right] \tag{9}$$

Skewed facial orthogonality

A face contour that shows skewed orthogonality is probably orthogonal in space. If entities on the contour of a planar face join only at right angles, then the contour can be said to be orthogonal. If this contour is viewed from an arbitrary viewpoint, it will exhibit skewed orthogonality, as is illustrated in *Figure 6a*. Faces or entity chains showing skewed orthogonality are easily detected by alternating their boundary lines between two main directions which correspond to the main axis directions of the original shape (see *Figure 6a*). The statistical behaviour of the alternating values produced by multiplying the scalar-product and the cross-product of adjacent 2D lines in the sketch plane is used for detection. Consistent behaviour is likely to represent skewed orthogonality. The amount by which the face is considered to have skewed orthogonality is represented by the value of the weighting coefficient $w_{\text{skewed orthogonality}}$. The terms to evaluate the above for a face are

$$\alpha_{\substack{\text{skewed}\\\text{orthogonality}}} = w_{\substack{\text{skewed}\\\text{orthogonality}}} \sum_{i=1}^{n} [\sin^{-1}(\hat{\mathbf{l}}_i \cdot \hat{\mathbf{l}}_{i+1})]^2 \tag{10}$$

$$w_{\substack{\text{skewed}\\\text{orthogonality}}} = \mu_{0,0.2}(\sigma(\beta_{i=1,\dots,n}))$$
$$\beta_i = (-1)^i \cdot [\hat{\mathbf{l}}'_i \cdot \hat{\mathbf{l}}'_{i+1}] \cdot [\hat{\mathbf{l}}'_i \times \hat{\mathbf{l}}'_{i+1}] \tag{11}$$

where *n* represents the number of lines along the face contour. Faces in which only part of their contour shows skewed orthogonality, such as in *Figure 6b*, can also be accepted, with the line list $\mathbf{l}'_{i=1,\dots,n}$ reduced to a subsection along the contour path. This requires detection of subsections with consistent b_i (low σ).

Skewed facial symmetry

A face showing skewed symmetry in 2D denotes a truly symmetrical face in 3D (see *Figure 7*). Algorithms for the

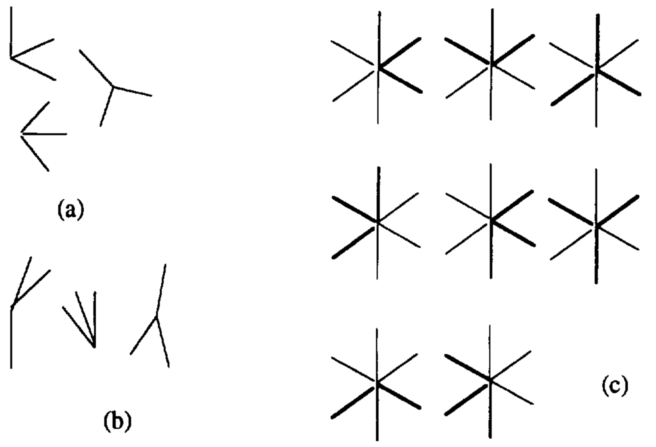


Figure 5 (a) Some three-line junctions may appear to form orthogonal corners; (b) some not; (c) a three-line junction has eight variants

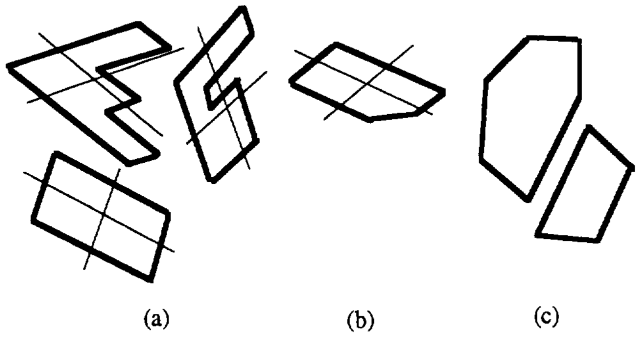


Figure 6 (a) Faces showing skewed orthogonality with their skewed principal axis; (b) partial orthogonality; and (c) none

detection of skewed symmetry have been the subject of extensive research (see, for example, References 15 and 16). A simplification used here is that if skewed symmetry exists in a polygonal shape, its axis intersects the contour at two points, each either a vertex or mid-point of an entity. Assuming also that the number of entities on both sides of the symmetry axis in a truly symmetrical shape is equal, the number of possible symmetry-axis candidates is reduced significantly to n, where n is the number of vertices in the shape. Each possible candidate symmetry-axis passes through the vertices $\mathbf{v}_k$ and $\mathbf{v}_{k+n/2}$, where $k = 1/2, 2/2, 3/2, \ldots, n/2$ and, for example, $\mathbf{v}_{2_{1/2}} = (\mathbf{v}_2 + \mathbf{v}_3)/2$.

The relationship between the vertices of the shape and the candidate symmetry-axis determines whether the axis can serve as a skewed symmetry axis. This relationship is represented, per axis k, by the weighting coefficient w_k. The maximal w_k determines the selected symmetry-axis if the face possesses the skewed symmetry characteristic at all.

$$w_{\substack{\text{skewed}\\\text{symmetry}}} = \max_{k=1/2,2/2,3/2,\ldots,n/2} [w_k]$$

$$w_k = \mu_{0,0.2}\left(\sigma_{i=1/2,2/2,3/2,\ldots,n/2}(\text{skew}_i) + \sigma_{i=1/2,2/2,3/2,\ldots,n/2}(\text{sym}_i)\right) \tag{12}$$

$$\text{skew}_i = \left[\frac{(\mathbf{v}'_k - \mathbf{v}'_{k+n/2})}{\|\mathbf{v}'_k - \mathbf{v}'_{k+n/2}\|} \cdot \frac{(\mathbf{v}'_{k+1} - \mathbf{v}'_{k-1})}{\|\mathbf{v}'_{k+1} - \mathbf{v}'_{k-1}\|}\right] \times \left[\frac{(\mathbf{v}'_k - \mathbf{v}'_{k+n/2})}{\|\mathbf{v}'_k - \mathbf{v}'_{k+n/2}\|} \times \frac{(\mathbf{v}'_{k+1} - \mathbf{v}'_{k-1})}{\|\mathbf{v}'_{k+1} - \mathbf{v}'_{k-1}\|}\right] \tag{13}$$

$$\text{sym}_i = \frac{\text{dist of } \mathbf{v}_{k+i} \text{ from axis}}{\text{dist of } \mathbf{v}_{k-i} \text{ from axis}} - 1 \tag{14}$$

Note that the vertices $\mathbf{v}'_k$ are in the 2D sketch plane. Two conditions are required for skewed symmetry to occur. First, corresponding points must be evenly distanced from the symmetry axis, a condition denoted by the 'sym' term above; second, lines stretched between corresponding points must form a consistent angle with the symmetry axis, a condition denoted by the 'skew' term above. If skewed symmetry has been detected, the optimization term will be

$$\alpha_{\substack{\text{skewed}\\\text{symmetry}}} = w_{\substack{\text{skewed}\\\text{symmetry}}} \sum_{i=1}^{n/2} \left[\sin^{-1}\left(\left[\frac{(\mathbf{v}_{k+1} - \mathbf{v}_{k-1})}{\|\mathbf{v}_{k+1} - \mathbf{v}_{k-1}\|}\right] \times \left[\frac{(\mathbf{v}_k - \mathbf{v}_{k+n/2})}{\|\mathbf{v}_k - \mathbf{v}_{k+n/2}\|}\right]\right)\right]^2 \tag{15}$$

where k denotes the axis that has been selected.

Line orthogonality

All line pairs in a junction *except those that are collinear* are perpendicular in 3D. This statement does not represent a pure regularity in the sense that it does not depend entirely on the appearance of the entity in the image plane apart from the exception clause. For a junction, the regularity serves mainly to initially 'inflate' the flat projection into 3D. The term used for this computation is:

$$\alpha_{\substack{\text{junction}\\\text{MSDP}}} = \sum_{i=1}^{n} w_i[\sin^{-1}(\hat{\mathbf{l}}_1 \cdot \hat{\mathbf{l}}_2)]^2$$
$$w_i = 1 = \mu_{0^\circ,7^\circ}(\cos^{-1}(\hat{\mathbf{l}}'_1 \cdot \hat{\mathbf{l}}'_2)) \tag{16}$$

where n is the number of non-collinear pairs of lines meeting at the junction. This regularity is termed MSDP (minimum sum of dot products).

Minimum standard deviation of angles

All angles between all pairs of lines meeting at junctions must be similar (MSDA = minimum standard deviation of angles). The term used for this computation for the entire body is

$$\alpha_{\text{MSDA}} = n \cdot \sigma^2(\cos^{-1}(\hat{\mathbf{l}}_1 \cdot \hat{\mathbf{l}}_2)) \tag{17}$$

where $\hat{\mathbf{l}}_1$ and $\hat{\mathbf{l}}_2$ represent the unit direction vectors of all possible line pairs meeting at vertices of the object. When arcs are introduced into the object domain, their tangents at the end-points are used in evaluating the MSDA and MSDP. The MSDA criterion has been suggested by Marill[9].

Face perpendicularity

All adjacent faces must be perpendicular. Again, this criterion serves to initially 'inflate' the flat projection to a convex shape in 3D space from which optimization is more easily achieved. The term used here is

$$\alpha_{\text{Perp faces}} = \sum_{i=1}^{n} [\sin^{-1}(\hat{n}_1 \cdot \hat{n}_2)]^2 \tag{18}$$

where $\hat{n}_1$ and $\hat{n}_2$ denote all possible combinations of normals of adjacent faces, and n is the number of such combinations.

Prismatic face

A face joining two parallel elliptic arcs in the projection plane is prismatic in space. Being prismatic, it can resolve both cylindricality and planarity (see *Figure 8*). The term to evaluate this behaviour sums up the misalignment of lines. For an exact prismatic surface, this value should become zero.

$$\alpha_{\text{prismatic}} = \sum_{i=1}^{n} [\cos^{-1}(\hat{\mathbf{l}}_1 \cdot \hat{\mathbf{l}}_{i+1})]^2 \tag{19}$$

where $\hat{\mathbf{l}}_i$ denotes an extrusion line and n represents the number of subdivisions made to the arcs. In our implementation, this condition is sought by segmenting the elliptic arcs at the edge of the cylindrical face into linear components at constant angular steps. Although the arc has been transformed into a group of lines and vertices, the solver ensures they remain on the same plane. The cylindrical face is then approximated with a set of rectangular faces. The parallelism criterion along with the face-planarity criterion serve to implement the prismatic-face regularity.

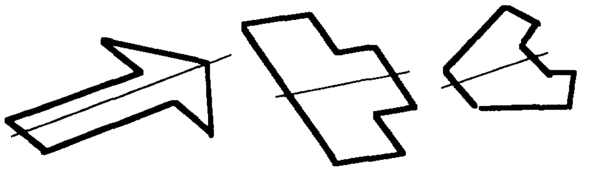


Figure 7 Faces showing skewed symmetry

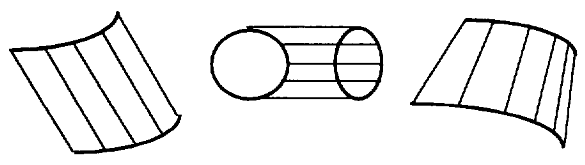


Figure 8 Equi-parameter extrusion lines are parallel for prismatic surfaces

Line collinearity

Lines collinear in the sketch plane are collinear in space. The term used to denote this heuristic is

$$\alpha_{\text{collinear}} = \sum_{i=1}^{n} w_i \cdot \max_{j=1,\ldots,4} \times \left[\frac{\det |\bar{\mathbf{v}}_j, \bar{\mathbf{v}}_{j+1}, \bar{\mathbf{v}}_{j+2}|}{\max (\|\bar{\mathbf{v}}_j - \bar{\mathbf{v}}_{j+1}\|, \|\bar{\mathbf{v}}_{j+1} - \bar{\mathbf{v}}_{j+2}\|, \|\bar{\mathbf{v}}_{j+2} - \bar{\mathbf{v}}_j\|)} \right]^2 \quad (20)$$

$$w_i = 1 = \mu_{0^\circ,7^\circ}(\cos^{-1}(\hat{\mathbf{l}}'_1 \cdot \hat{\mathbf{l}}'_2)) \quad (21)$$

where n is the number of such collinear pairs and $\mathbf{v}_{j=1,\ldots,4}$ are the four 3D end-vertices of the two lines.

Planarity of skewed chains

When a *chain* of entities is found to exhibit skewed symmetry of skewed orthogonality, both *planarity* and *symmetry* or *orthogonality* are required, through application of the corresponding formulation above (see *Figure 9*).

OPTIMIZATION-BASED RECONSTRUCTION

When the reconstruction process begins, the given 2D edge–vertex graph is analysed and image regularities are identified. For each regularity, the corresponding weighting coefficient is computed according to the formulations presented in the previous section. A 3D configuration can be represented by a vector **Z** containing the z coordinates of the vertices. A compliance function **F**(**Z**) can then be computed for any 3D configuration by summing the contributions of the regularity terms. Regularities are prefixed by a global balancing coefficient vector **W**. The final compliance function to be optimized takes the form

$$\mathbf{F}(\mathbf{Z}) = \mathbf{W}^{\mathrm{T}}\Sigma[\alpha] \quad (22)$$

where

$$[\alpha] = \begin{bmatrix} \alpha_{\text{planarity}} \\ \alpha_{\text{parallel}} \\ \alpha_{\text{vertical}} \\ \alpha_{\text{isometry}} \\ \alpha_{\text{corner}} \\ \alpha_{\text{skewed orthogonality}} \\ \alpha_{\text{skewed symmetry}} \\ \alpha_{\text{MSDP}} \\ \alpha_{\text{MSDA}} \\ \alpha_{\text{perpendicular faces}} \\ \alpha_{\text{prismatic}} \\ \alpha_{\text{collinear}} \end{bmatrix}$$

The appropriate configuration of zs (the vector **Z**) is sought using optimization. *Figure 10* shows various stages of the object as it evolves from a flat 2D projection to a full 3D reconstruction.

The process of manipulating the zs while seeking the best reconstruction is a full n-dimensional non-linear optimization problem, where n denotes the number of vertices. In practice, the first z can be arbitrarily set to 0, so an $n - 1$-dimensional problem is to be solved. Since all the regularities (not the weights, which are constant during the optimization) are defined using continuous and differentiable algebraic terms, the compliance function is well defined and continuous, and its derivatives defined everywhere. Under such circumstances, the optimization procedure is guaranteed to converge, since it can always proceed from the current point downhill in some direction until it cannot go any further.

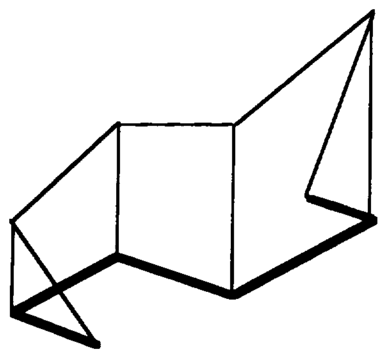


Figure 9 A skewed orthogonal chain of entities

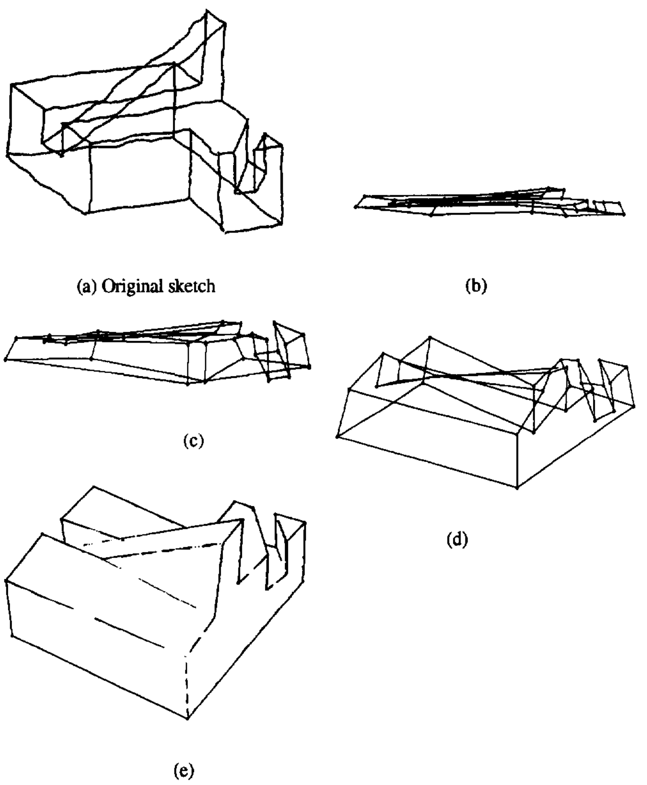


Figure 10 A 2D sketch 'inflated' into 3D by optimization

However, the point of convergence is not guaranteed to be the global optimum. This means that the reconstruction algorithm will always return a solution, but not necessarily the correct one. According to the requirements posed earlier, an incorrect solution can be any collection of lines and faces randomly positioned in space, in a form that happens to comply with the given projection from some point of view. These cannot be considered as valid objects, and some heuristic measures of correctness suggested in Reference 1 can be used to identify such situations and acknowledge failure or restart at a different point. There is no method, as yet, to determine in advance whether the optimization approach will succeed or fail. Mathematical inconsistency does not necessarily indicate that the object is not reconstructable, as shown in the truncated pyramid example in *Figure 3*. Similarly, there are no restrictions to any specific object domain, such as manifold objects or trihedral polyhedrons, and there are no means of confirming the consistency of the input graph. The input sketch could depict a skeletal structure with no faces at all. The reconstruction algorithm proposed in this paper relies to some extent on the fact that it is embedded in an interactive system, and the user can remove and re-sketch incorrect reconstructions. Some interesting projections can have several topologically different solutions. Such cases can usually be identified mathematically in advance. Refer to Reference 2 for more details on the analysis of general non-manifold sketch topology. The famous Neckers-Cube illusion can easily be overcome by reversing the signs of the depth coordinates.

Multiple-dimensional non-linear optimization (without derivatives) is in itself a subject for extensive research and is beyond the scope of this work. Several optimization methods, however, have been examined in order to perform the optimization.

Brent's minimization[17]

This 1D minimization technique is based on parabolic interpolation and is adequate for squared deviation formulations of image regularity terms. This method is applied cyclically, vertex by vertex, until the system appears to reach equilibrium. This method cannot be guaranteed to arrive at the global optimum, but it is fairly robust.

Conjugate gradients[18]

This method uses a 1D minimization procedure to minimize the target function separately in each direction and then computes a modified set of directions to travel along in the next iteration. If the target function can be minimized faster by travelling along a direction that does not correspond to one of the main axes of the n dimensions, then this method is likely to find that direction. Again, this method typically converges rapidly but is less robust and may find a local optimum.

Genetic algorithm[19]

This algorithm consists of: a mechanism for generating a large population of random objects corresponding to the projection; a mechanism for comparing solutions (the compliance function); and a mechanism for selecting parents and combining them to introduce new solutions to the population. Some individual solutions are randomly mutated from time to time. After a large number of iterations, a good solution can be found among the population. This method is slow but has a better chance of finding the global optimum.

Of the three methods tested, cyclic application of Brent's minimization seemed most promising. As opposed to the other multiple-dimensional, non-linear optimization techniques mentioned above, Brent's method allows for intervention by various operations, such as control over dynamic vertex ordering and restrictive vertex displacement. The conjugate gradient method generally converges in one-third of the time, but in some cases does not converge at all and thus is less robust. The genetic approach has the important benefit of successfully avoiding local minima which are problematic in multi-dimensional optimization problems. However, for a genetic algorithm to reach a good solution, a very large number of iterations is necessary.

COMPLEXITY OF THE RECONSTRUCTION ALGORITHM

The complexity of the reconstruction algorithm is a multiple of two factors: (a) the complexity of the optimization procedure optimizing the compliance function C; and (b) the complexity of evaluating the compliance function C itself. The complexity of an

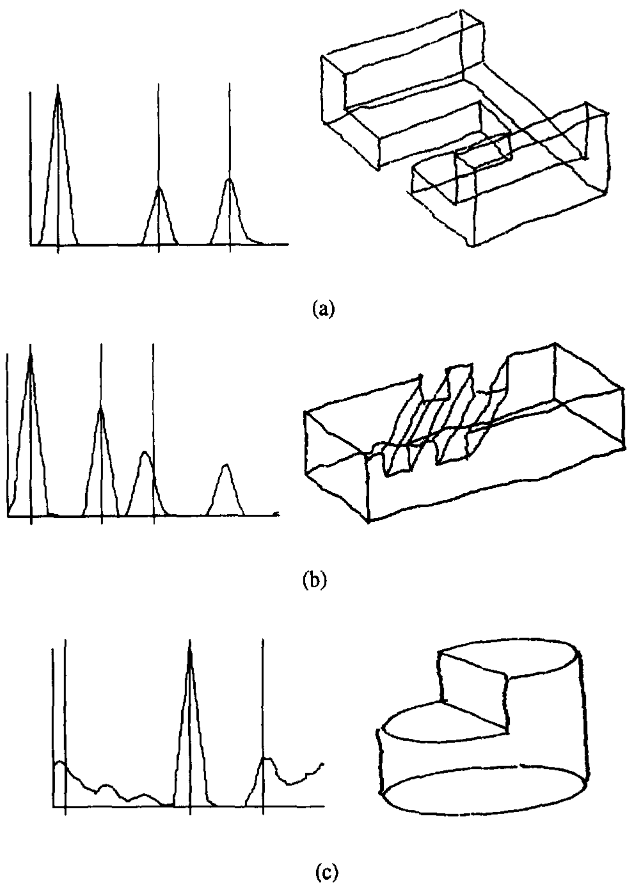


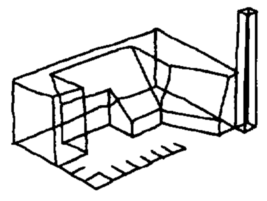
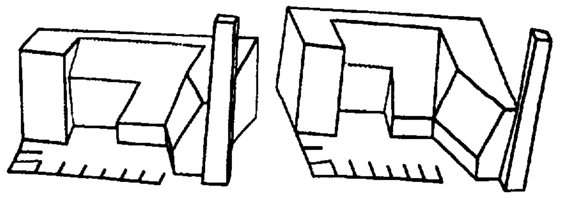
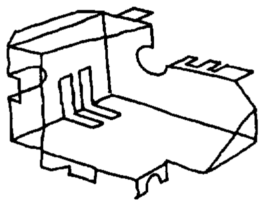
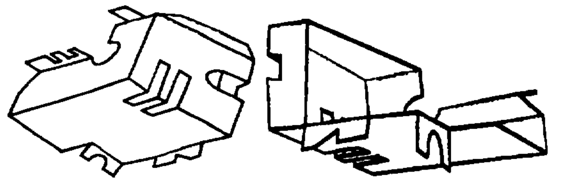
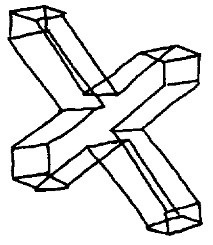
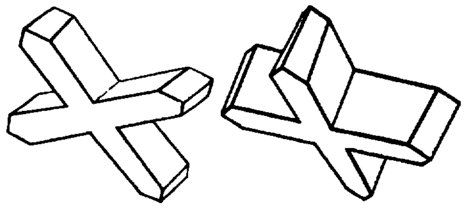
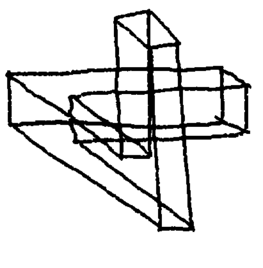
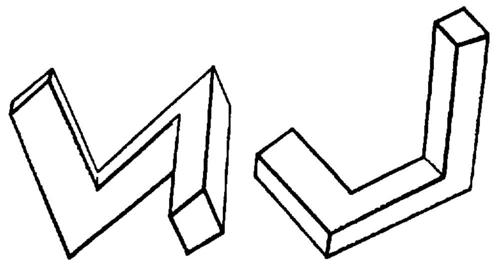
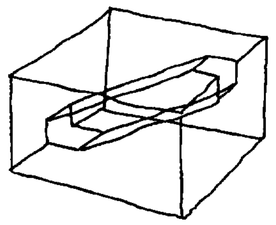
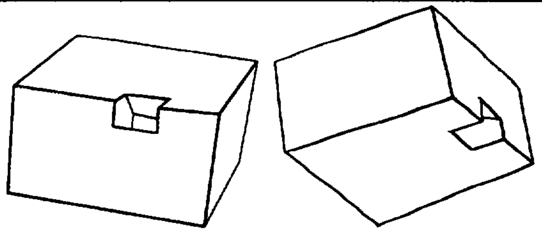
Figure 11 Angular distribution graphs (ADGs) of various sketches

optimization procedure is, at least, $O(n)$ (n^2 for Brent's optimization), where n is the number of dimensions, or, in this case, the number of vertices. In each evaluation of the compliance function, all the regularity terms are evaluated and summed. Assume that there are k regularity types; then, in the worst case, k regularities are observable between each pair of entities in the sketch. Noting that in an object with n vertices there are $O(n)$ edges (Euler's equation), it can be concluded that each evaluation of the compliance function may be, in the worst case, of complexity $O(kn^2)$, i.e. $O(n^2)$, since k is constant. The overall complexity of the optimization process is thus at least $O(n^3)$, depending on the optimization procedure.

Accelerating and improving convergence: angular distribution graph

As is evident from the complexity analysis, the basic drawbacks of the optimization procedure are its slow convergence and susceptibility to arriving at local minima. These problems become more severe when the dimensionality of the problem increases, as is the case for complex models with a large number of vertices. A partial solution is an application using sketch input is part-by-part sketching of the input model. Thus, each part in itself is a smaller reconstruction problem.

Another approach is based on obtaining a preliminary approximation of the model before starting the

Original 2D Sketch	3D Reconstruction
 530 secs.*	 A mixed-dimensions non-manifold object
 1674 secs.*	 A non-manifold sheet-metal product with holes and some curved edges
 133 secs.*	 A concave extruded solid
 43 secs.*	 An orthogonal object
 45 secs.*	 A manifold solid with a piercing hole

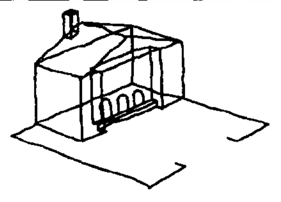
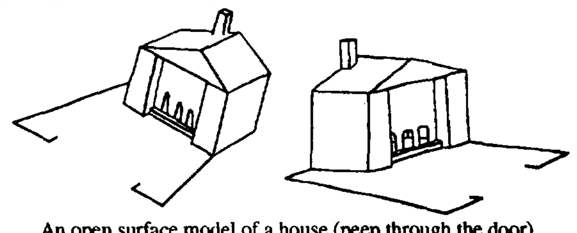
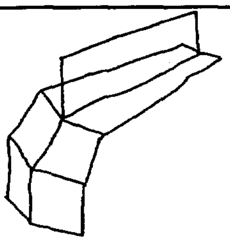
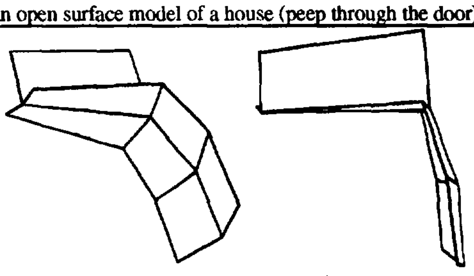
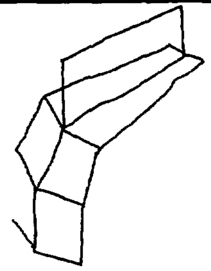
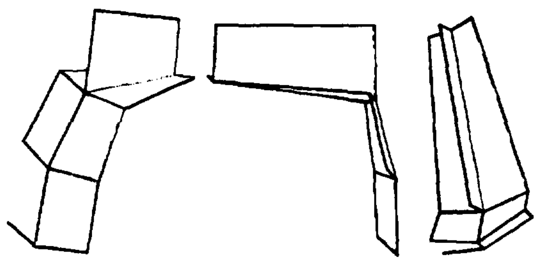
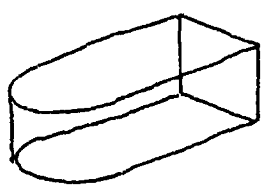
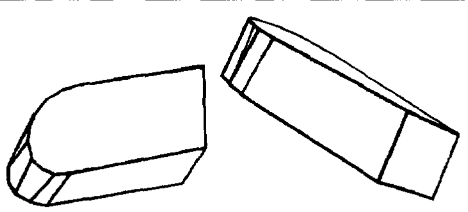
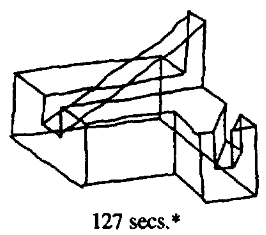
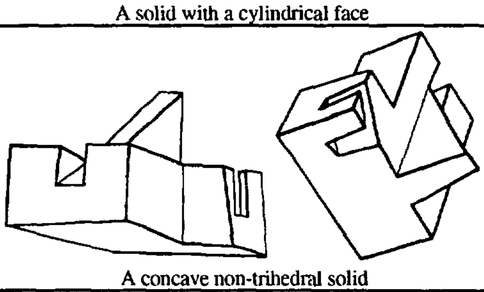
*Reconstruction time on SG Personal Iris. (4D)

Figure 12 Some examples of input sketches and their interpretations. Note that the reconstruction was performed from the sketch alone, without any prior knowledge about the 3D object represented or its type

optimization process, namely, starting with an improved initial position. As in many numerical non-linear optimization problems, the distance of the initial guess from the target has the greatest influence both on convergence time and on ability to converge to the global minimum. The initial guess would be a preliminary estimation of the shape of the body represented by the sketch.

The human process of understanding a line drawing may also go through some sort of preliminary understanding phase in which the basic and overall shape is conceptualized but the fine details and features are not yet comprehended. It is not clear how this understanding is accomplished, but it is possible that the human observer identifies some kind of main trend in the direction of the lines corresponding to the main dimensions of the body. The main trend may suggest that the object is basically orthogonal, spherical or perhaps cylindrical in shape. It is also possible that different areas of a sketch encompass different trends, thus implying that the object is perhaps spherical on one side and orthogonally thin and tall on the other. A general sketch stroke trend might also identify some sort of symmetry or extrusion or the existence of two distinct parts separated by an angle of 30°.

To understand the term *trend* more specifically, it is necessary to analyse statistically the direction of strokes in the sketch to find some general consistency. In this study, orthogonality is sought because it is the prevailing trend in most engineering drawings and also the simplest to identify.

Original 2D Sketch	3D Reconstruction
 1183 secs.*	 An open surface model of a house (peep through the door)
 18 secs.*	 A non-manifold shell surface
 12 secs.*	 Same shell as above, with a line omitted from the sketch
 23 secs.*	 A solid with a cylindrical face
 127 secs.*	 A concave non-trihedral solid

*Reconstruction time on SG Personal Iris. 4D

Figure 12 Continued

The distribution of strokes is analysed by means of an angular distribution graph (ADG). The ADG is constructed by sampling the angle of every entity in the sketch and plotting it on an angular distribution histogram. When an angle is added to the graph, a Gaussian distribution curve with $s = 7$ is superpositioned onto the graph to account for the inaccuracy of the sketch. After all the curves have been superpositioned and combined with their different weights, the graph is normalized with its maximum at 1.0. The resulting ADG qualitatively shows three prevailing angles in the sketch, as can be seen in the examples shown in *Figure 11*.

Figure 11 shows that strictly orthogonal objects have clear three hump ADG's. Other objects exhibiting some orthogonality still show a few noticeable humps surrounded by angular 'noise'. It is possible to identify the main axis directions of the intended object from this graph. To do so, an axis system must be found at a spatial orientation such that it will be projected onto the sketch plane with angles corresponding to maxima in the ADG.

Once the prevailing axis system is identified, lines in the sketch are associated with one of the axes and assigned a compatibility factor. For example, lines that are found nearly parallel (in the sketch plane) to the X axis are marked as parallel to X, with a compatibility factor denoting the angular difference between the line and the axis (in the sketch plane). The term 'X' is an arbitrary label assigned to any one of the axes. Vertex positions are computed starting at an arbitrary origin point with an arbitrary depth of zero and advancing to adjacent points using lines with the best compatibility factors. The resulting reconstruction process is extremely simple and is equivalent to a sorting procedure with complexity of $O(n \log n)$, where n is the number of vertices. It provides a good initial guess for most typical engineering parts exhibiting some degree of orthogonality. The initial phase of building the ADG is a process of $O(n)$ and analysing it is a constant procedure independent of n. It appears that if this process were to be further developed to identify different groups of faces with corresponding ADGs, a better initial guess could be found for complex parts.

Results

Figure 12 shows some reconstruction results on a variety of sketches representing objects of various types. The sketches shown here consist of 10–70 vertices and are more complex than those appearing in previous publications.

SUMMARY

In this research, an optimization-based method for reconstructing a 3D object from a single sketch has been developed. The process is based on identifying and formulating geometrical regularities and seeking their associated 3D configuration. The algorithm presented here is capable of reconstructing objects from a wide object domain with considerable fault tolerance. Improved performance has been achieved through ADG analysis prior to processing.

The reconstruction results appear to correctly reflect the concept of the depicted object but tend to produce a somewhat distorted 3D model. In part, this distortion is due to the inherent inaccuracies in the sketch, but it also relates to the failure to accurately distinguish between important and less important regularities. Evidently, more research is required pertaining to this distinction ability. In addition, the reconstruction process is far more prone to errors when the object involves curved faces. This can be attributed to the fact that the majority of regularities used are applicable only to straight line segments. More research in curved-face regularities is necessary as well.

When these difficulties are overcome, we believe that this approach to sketch analysis can be incorporated into CAD systems to produce a new kind of natural user interface aimed particularly at the conceptual design stage. This type of interface will enable designers to communicate more freely with the computer and among themselves.

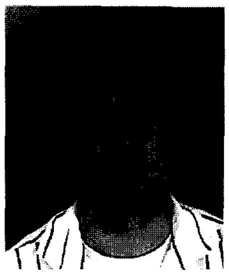
ACKNOWLEDGMENTS

This work was carried out in partial fulfillment of the requirements for the degree of DSc in Mechanical Engineering at the Technion (Hod Lipson). This research has been supported in part by the Fund for the Promotion of Research at the Technion, Research No. 033-028.

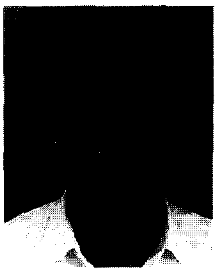
REFERENCES

1 Lipson, H and Shpitalni, M 'A new interface of conceptual design based on object reconstruction from a single freehand sketch' *Ann. CIRP* Vol 44 No 1 (1995) pp 133–136
2 Shpitalni, M and Lipson, H 'Identification of faces in a 2D line drawing projection of a wireframe object' *IEEE Trans. Pattern Analysis & Machine Intell.* (submitted)
3 Sugihara, K *Machine Interpretation of Line Drawings* MIT Press (1986)
4 Nagendra, I V and Gujar, U G '3D objects from 2D orthographic views—a survey' *Comput. Graph.* Vol 12 No 1 (1988) pp 111–114
5 Wang, W and Grinstein, G 'A survey of 3D solid reconstruction from 2D projection line drawings' *Comput. Graph. forum* Vol 12 (1993) pp 137–158
6 Huffman, D A 'Impossible objects as nonsense sentences' *in Machine Intelligence* Edinburgh University Press (1971) pp 295–323
7 Clowes, M B 'On seeing things' *Artificial Intelligence* Vol 2 (1971) pp 79–116
8 Kanade, T 'Recovery of the three dimensional shape of an object from a single view' *Artificial Intelligence* Vol 17 (1980) pp 409–460
9 Marill, T 'Emulating the human interpretation of line drawings as three dimensional objects' *Int. J. Comput. Vision* Vol 6 No 2 (1991) pp 147–161
10 Leclerc, Y G and Fiscler, M A 'An optimization based approach to the interpretation of single line drawings as 3D wire frames' *Int. J. Comput. Vision* Vol 9 No 2 (1992) pp 113–136
11 Lamb, D and Bandopadhay, A 'Interpreting a 3D object from a rough 2D line drawing' *Proc. First IEEE Conf. on Visualization 90* (1990) pp 59–66
12 Marti, E, Regomcós, J, López-Krahe, J and Villanueva, J J 'Hand line drawing interpretation as three dimensional object' *Signal Process.* Vol 32 (1993) pp 91–110
13 Grimstead, I J and Martin, R R 'Creating solid models from single 2D sketches' *Proc. Third Symp. on Solid Modeling Applications, ACM Siggraph* (1995) pp 323–337
14 Dori, D 'Dimensioning analysis—toward automatic understanding of engineering drawings' *Commun. ACM* Vol 35 No 10 (1992) pp 92–102
15 Friedberg, S A 'Finding axes of skewed symmetry' *Comput. Vision, Graph. & Image Process.* No 34 (1986) pp 138–155

16 Yip, R K K, Tam, P K S and Leung, D N K 'Application of elliptic Fourier descriptors to symmetry detection under parallel projection' *IEEE PAMI* Vol 16 No 3 (1994) pp 277–286
17 Brent, R P *Algorithms for Minimization without Derivatives* Prentice Hall, Englewood Cliffs, NJ (1973) Chap 7
18 Polak, E *Computational Methods in Optimization* Academic Press, New York (1971) Chaps 2 and 3
19 Holland, J H *Adaptation in Natural and Artificial System* University of Michigan Press, Ann Arbor (1975)



Hod Lipson was born in Haifa, Israel in 1967. He received a BSc (1989) in mechanical engineering from the Technion—Israel Institute of Technology. From 1990 to 1994, he worked in the CAD *software industry in the fields of naval architecture and sheet metal design. He is currently pursuing a PhD degree in mechanical engineering at the Technion. His research interests include artificial intelligence in design, image understanding, and geometric modelling.*



Moshe Shpitalni received a BSc (1972), an MSc (1975) and a PhD (1980) in mechanical engineering from the Technion—Israel Institute of Technology. In 1983, after two years at Rensselaer Polytechnic Institute, USA, he joined the faculty of the Technion, where he is now an associate professor and head of the Laboratory for Interactive Computer Graphics and CAD. *His research interests are in the applications of geometrical modelling and reasoning to design and manufacturing systems,* CIM, *automatic process planning, and CNC. He is particularly interested in manufacture of sheet metal products and automatic assembly.*

Mathematical Sketching

MathPad^2: A System for the Creation and Exploration of Mathematical Sketches

Joseph J. LaViola Jr. Robert C. Zeleznik

Brown University [*]

Abstract

We present *mathematical sketching*, a novel, pen-based, modeless gestural interaction paradigm for mathematics problem solving. Mathematical sketching derives from the familiar pencil-and-paper process of drawing supporting diagrams to facilitate the formulation of mathematical expressions; however, with a mathematical sketch, users can also leverage their physical intuition by watching their hand-drawn diagrams animate in response to continuous or discrete parameter changes in their written formulas. Diagram animation is driven by implicit associations that are inferred, either automatically or with gestural guidance, from mathematical expressions, diagram labels, and drawing elements. The modeless nature of mathematical sketching enables users to switch freely between modifying diagrams or expressions and viewing animations. Mathematical sketching can also support computational tools for graphing, manipulating and solving equations; initial feedback from a small user group of our mathematical sketching prototype application, MathPad^2, suggests that it has the potential to be a powerful tool for mathematical problem solving and visualization.

CR Categories: H.5.2 [Information Interfaces and Presentation]: User Interfaces—Interaction Styles G.4 [Mathematics of Computing]: Mathematical Software—User Interfaces;

Keywords: pen-based interfaces, mathematical sketching, gestures

1 Introduction

Diagrams and illustrations are often used to help explain mathematical concepts. They are commonplace in math and physics textbooks and provide a form of physical intuition about abstract principles [Hecht 2000; Varberg and Purcell 1992; Young 1992]. Similarly, students often draw pencil-and-paper diagrams for math problems to help in visualizing relationships among variables, constants, and functions. With the drawing as a guide, they can write the appropriate math to solve the problem. However, such static diagrams generally assist only in the initial formulation of mathematical expressions, not in the “debugging” or analysis of those expressions. This can be a severe limitation, even for simple problems with natural mappings to the temporal dimension, or for problems with complex spatial relationships.

By animating sketched diagrams from changes in associated mathematical expressions, users can evaluate different formulations with

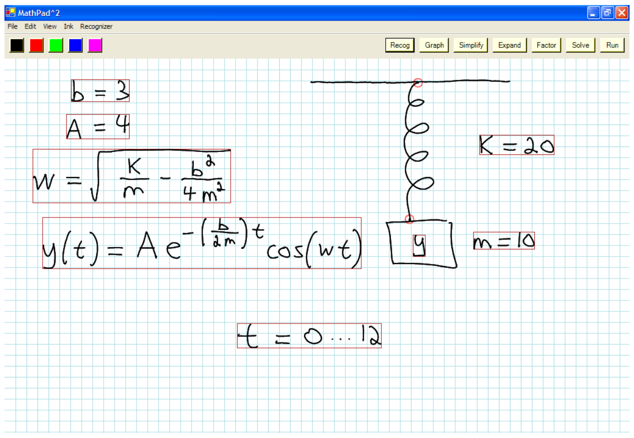


Figure 1: A mathematical sketch used to explore damped harmonic oscillation. It shows a spring and mass drawing and the necessary equations for animating the sketch. The label inside the mass associates the mathematics with the drawing.

their physical intuitions about motion. They can sense mismatches between animated and expected behaviors and can often see that a formulation is incorrect and also make better educated guesses as to why. Alternatively, correct formulations can be explored intuitively, perhaps to home in on an aspect of the problem to study with a more conventional numerical or graphing technique. It is beyond the scope of this paper to evaluate the educational merits of mathematical sketching; however, we are convinced that the rapid creation of mathematical sketches can unlock a range of insight, even for such simple formulations as the ballistic 2D motion of a spinning football, where correlations among position, rotation and their derivatives can be challenging to comprehend.

This paper presents MathPad^2, a prototype application for creating mathematical sketches (see Figure 1). MathPad^2 incorporates a novel gestural interaction paradigm that lets users modelessly create handwritten mathematical expressions using familiar math notation and free-form diagrams, including associations between the two, with only a stylus. We posit that because users must write down both the math and the diagrams themselves, MathPad^2 will not only be general enough to apply to a full spectrum of problems, but may also support deeper mathematical understanding than alternative approaches including professionally created interactive illustrations. Thus, MathPad^2's central design principle is to be broadly applicable by enforcing the notion that as much behavior as possible be specified by user-written mathematical expressions, not by rules or behaviors implicitly embedded in the system.

To present MathPad^2 from a system's perspective, we describe various user interface and recognition options and discuss why we chose our current design. The next section considers related work and is followed by sections on MathPad^2's gestural user interface and on its visual parsing engine. We then present an example scenario of how an introductory physics student might use the system. Finally, we discuss informal feedback from a small user group and

[*]Email: {jjl,bcz}@cs.brown.edu

how mathematical expression recognition and parsing accuracy affects system usability.

2 Related Work

The idea of using computers to create dynamic illustrations of mathematical concepts has a long history. One of the earliest dynamic illustration environments was Borning's ThingLab, a simulation laboratory environment for constructing dynamic models of experiments in geometry and physics, that relied heavily on constraint solvers and inheritance classes [Borning 1979]. Other systems such as Interactive Physics$^{\text{TM}}$ and The Geometer's SketchPad$^{\text{TM}}$ also let the user create dynamic illustrations; these systems are all WIMP-based (Windows, Icons, Menus, Pointers) resulting in a significant amount of mode switching and loss of fluidity within the interface. In addition, they do not allow the user to write handwritten mathematics to create these illustrations. Because MathPad^2 uses handwritten mathematical expressions, users can leverage their knowledge of mathematical notation in order to create mathematical sketches. Java applets that provide both interactive and dynamic illustrations have also been developed for exploring various mathematical principles [Laleuf and Spalter 2001; Spalter and Simpson 2000]. However, these applets are not general, typically provide limited control over the illustration, and rarely show the user the mathematics behind the illustration.

One of the key contributions of the MathPad^2 system is its gestural user interface. This type of interface has been used in many different applications including cooperative objected-oriented design [Damm et al. 2000], conceptual design in 2D [Gross and Do 1996] and 3D [Igarashi et al. 1999; Zeleznik et al. 1996], musical score creation [Forsberg et al. 1998], prototyping user interfaces [Landay and Myers 1995], and whiteboard systems [Mynatt et al. 1999; Moran et al. 1997]. While these gestural interfaces have worked well for their particular applications, they are either modal or have limited drawing domains. In contrast, MathPad^2 strives for a modeless gestural interface that allows fluid transitions among drawing free-form shapes, writing mathematics, and performing gestural actions.

Alvarado [Alvarado 2000] and Kara [Kara et al. 2004] let the user make sketched diagrams that are recognized as drawing primitives with domain knowledge from specific disciplines and then animated. Although these systems provide powerful illustrations of physics and mathematical concepts, they are limited because of their domain knowledge and because they hide the underlying mathematical formulations from the user.

The primary focus of systems such as Mathematica$^{\text{TM}}$, Maple$^{\text{TM}}$, MathCad$^{\text{TM}}$, and Matlab$^{\text{TM}}$ has been entering mathematics for computation, symbolic mathematics, and illustration. These tools can create dynamic illustrations using mathematics as input. However, the mathematical notation used in these systems is one-dimensional, requiring unconventional notation for concepts that would be intuitive using 2D handwritten mathematics. In addition, these systems do not let the user create diagrams in a natural pencil-and-paper style.

Finally, a number of systems let users enter 2D handwritten mathematics in the context of math recognition and parsing, such as those found in [Zanibbi et al. 2002; Chan and Yeung 2000a; Matsakis 1999; Miller and Viola 1998]. Only a few of these systems go beyond just exploring and developing recognition technology. For example, Chan developed a simple pen-based calculator [Chan and Yeung 2001] while xThink, Inc. developed MathJournal$^{\text{TM}}$, a system designed to solve equations, perform symbolic manipulation, and make graphs. However, we are not aware of any system that lets users make dynamic illustrations with handwritten 2D mathematics.

3 MathPad^2 and Mathematical Sketching

Mathematical sketching is the process of making simple illustrations from a combination of handwritten 2D mathematical expressions and sketched diagrams. Combining mathematical expressions with diagram elements, called an *association*, is done either implicitly using diagram labels as input to an inferencing engine or manually using a simple gestural user interface. The formulations in the mathematical sketch of Figure 1 are written as digital ink and converted, using our mathematical expression recognition and parsing engine, for further processing into equivalent one-dimensional string representations required by our computational back end. Individual diagram elements can be associated with various mathematical expressions and will behave accordingly. The user can view the sketch results as an animation that is based on the underlying mathematical specification.

The following three sections describe the user interface, the sketch parser, and the animation engine of our MathPad^2 system.

3.1 User Interface

An important goal of the MathPad^2 prototype is to facilitate mathematical problem solving without imposing any interaction burden beyond what would arise with traditional media. Since pencil-and-paper users switch fluidly between writing equations and drawing supporting diagrams, a modeless interface is highly desirable. Although a simple free-hand drawing pen would suffice to mimic pencil and paper, we want to support computational activities including formula manipulation, diagram rectification (see section 3.2.4), and animation. This functionality requires extending the notion of a free-hand pen, either implicitly by parsing the user's 2D input on the fly or explicitly by letting the user perform gestural operations. We chose an interface that combines both in an effort to reduce the complexity and the ambiguities that arise in many hand-drawn mathematical sketches – we use parsing to recognize mathematical expressions and make associations, while we use gestures to segment expressions and perform editing operations.

The challenge for MathPad^2's gestural user interface is for its gestures not to interfere with the entry of drawings or equations and yet still be direct and natural enough to feel fluid. The following sections describe the gestural interface summarized in Figure 2.

3.1.1 Writing Mathematical Expressions

Inking Writing mathematical expressions and diagrams in MathPad^2 is straightforward: users draw with a stylus on a Tablet PC as they would using pencil and paper. The only complication in writing expressions is how errant strokes are corrected. Although the stylus can be flipped over to use its eraser, we found that a gestural action not requiring flipping was both more accurate (because of hardware characteristics of the stylus) and more convenient. We therefore first designed a *scribble erase* gesture in which the user scribbles with the pen back and forth over the ink strokes to be deleted. The drawback of this first implementation was that it created too many false positives, recognizing scribble erase gestures

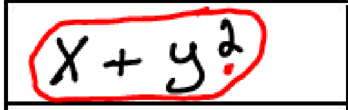
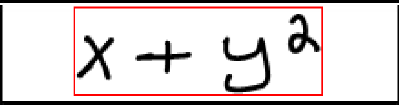
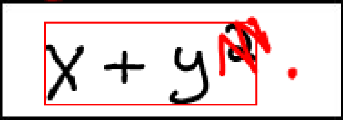
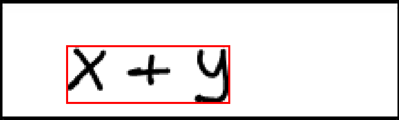
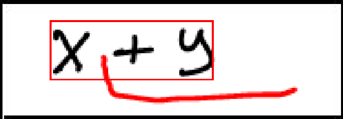
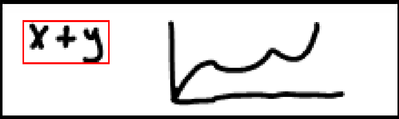
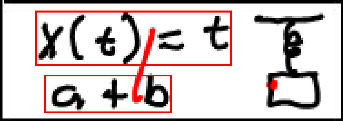
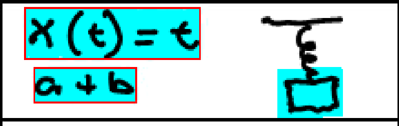
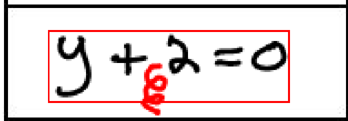
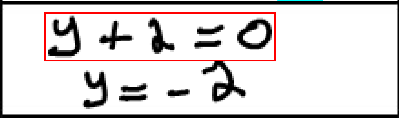
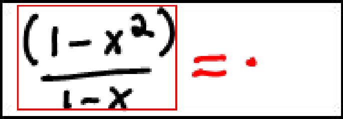
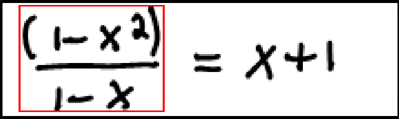
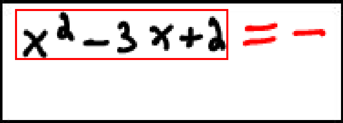
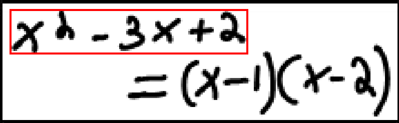
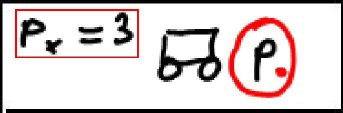
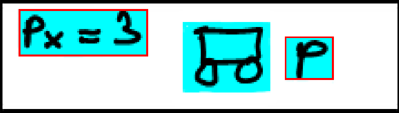
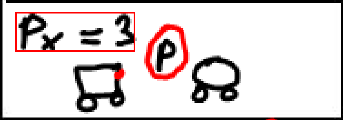
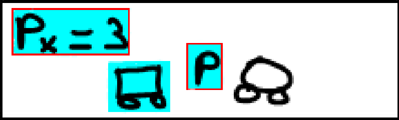
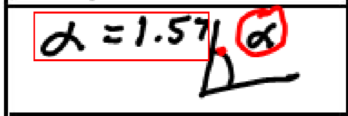
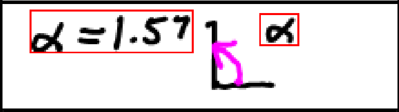
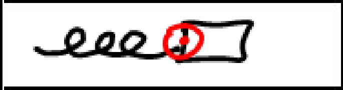
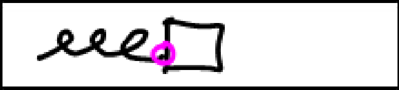
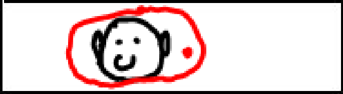
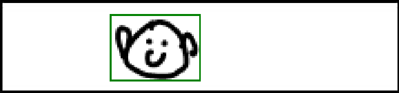
Gesture	Result	Description
		Lasso and tap to recognize an expression
		Scribble and tap to delete ink
		Create a graph, Line starts in recognized math, no cusps or intersections
		Line through math and click on drawing makes association, Release makes rotation point
		Solve equation
		Simplify an expression
		Factor an expression
		Make implicit association using label family 'P'
		Make implicit association with explicit tap on object
		Implicit angle association and rectification (tap location indicates which line moves)
		Nail two drawing elements by small circle and tap inside
		Group strokes (lasso content determines grouping or recog)

Figure 2: Gestures for interacting with MathPad^2. Gesture strokes in the first column are shown here in red for clarity. In the second column, cyan-highlighted strokes provide association feedback (the highlighting color changes each time a new association is made), and magenta strokes show nail and angle association/rectification feedback. Note that the first and last gesture are equivalent: the system examines the ink strokes inside the lasso to determine whether to perform stroke grouping or recognition.

when in fact the user had intended to draw ink and not erase anything. To alleviate this problem we decided on the use of a compound gesture because of its relative simplicity and ease of execution. Thus our current definition of scribble erase is the same scribble stroke as before followed directly by a tap. In practice, users found this compound gesture easy to learn, effective in eliminating false positives, and not significantly more difficult or slower than the simple scribble gesture.

Parsing Expressions Once mathematical expressions are drawn, they must be parsed by the system and converted to strings for use by a computational engine (Matlab in this case). Our initial attempt, clicking on a recognize button that attempted to recognize all math on the page, was problematic because it was hard to algorithmically determine "lines of math" accurately, especially when the expressions were closely spaced, at unusual scales or in unusual 2D arrangements. We therefore opted for a manual segmentation alternative in which the user explicitly selects a set of strokes comprising a single mathematical expression by drawing a lasso. Since in a modeless interface making a lasso cannot be distinguished from drawing a closed curve, we needed to disambiguate the two actions. An approach that worked well for some was to lasso lines of math while pressing the barrel button on the stylus. However, many people inadvertently triggered this button when trying to draw, while others found pressing a barrel button awkward. Once again, the solution was to use compound gestures – this time drawing a lasso around a line of math followed by a tap. This modified gesture has worked robustly within our dozen-user feedback group.

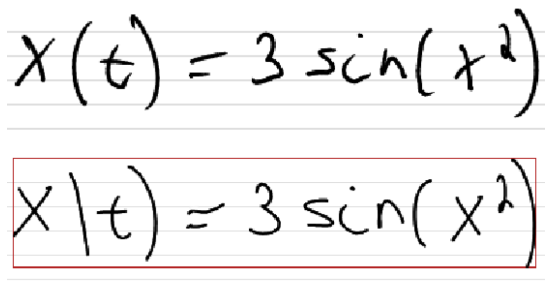


Figure 3: A written mathematical expression and a recognized one. Even though the recognized expression is presented in the user's own handwriting, recognition errors are easily discernible.

Feedback Ideally, recognition would not require any feedback to the user – the system would simply understand what the user had written. However, due to the complexity and ambiguities of mathematical notation, it is essential that users know how MathPad^2 interprets their input expressions. We chose to show the system's recognition in two ways. First, a bounding box rectangle is drawn around the user's digital ink for each recognized expression. Second, each ink symbol within a recognized expression is replaced with the corresponding canonical version of that symbol (a training example of the user's own handwriting) that occupies the same bounding box as the original stroke (see Figure 3). Our theoretical basis for this feedback is twofold: users can generally disambiguate characters of their own handwriting even if they are quite similar, and users often want to preserve the look, feel, and spatial relationships of their notation for reasons of esthetics, subtle information detail, and ease of editing [Zanibbi et al. 2001]. Although we had earlier provided an option for modifying the layout of the recognized ink to correspond to the 2D parse relationships of the math, this was often distracting since our implementation did not preserve all the layout subtleties of the user's original handwriting; nonetheless, we are currently investigating techniques for improved 2D layout constrained to the style of the user's input handwriting. Some users have expressed an interest in a high-quality typeset feedback option to be used primarily as an alternate view in a less technically interesting "clean-up" phase.

Correcting Recognition Errors When users identify a recognition error, they can correct it simply by scribble erasing the offending symbols and rewriting them, or they can invoke a pull-down menu of alternatives. Whenever the stylus hovers over a recognized expression, a small green button appears in the lower right corner of its bounding box. Pressing on this button displays a menu of alternate recognitions and the displayed ink is updated to reflect the alternate expression selected.

3.1.2 Making Diagrams

Diagrams are sketched in the same way as mathematical expressions except the diagrams need not be recognized. In balancing the value of a primitives-based drawing system against the added interaction overhead of specifying primitives, we decided that for our initial prototype our only primitive would be unrecognized ink strokes. We believe that a primitives-based approach not only would require a more elaborate user interface, but would also make it more difficult to find the source of an error since the diagram would be in part user-specified and in part specified by the primitive's hidden rules or behaviors.

Nailing Diagram Components In reviewing a broad range of mathematical illustrations, we noticed that the single low-level behavior of stretching a diagram element could be a very powerful technique. Thus, we support the concept of "nails" to pin a diagram element to the background or to pin a point on a diagram element to another diagram element. If either diagram element is moved, the other element either moves rigidly to stay attached at the nail if it has only one nail, or stretches so that all its nails maintain their points of attachment. Nails, although used primarily to create non-rigid objects, can also create binary grouping relationships. We do not currently detect or support cyclic nail relationships.

The user creates a nail by drawing a circle around the appropriate location in the drawing and making a tap gesture inside it (the tap disambiguates the nail gesture from a circle that is part of a drawing). A nail is distinguished from a math recognition gesture (see above) because the nail circle must intersect one or two drawing elements but fully contain neither, whereas the math recognition lasso must fully contain at least one stroke. The system finds and links together all drawing elements that intersect the circle of the nail gesture. The link is then symbolized by centering a small red circle on the nail location.

Grouping Diagram Components Since many drawings involve creating one logical object from a set of strokes (drawing elements), we overload the math recognition gesture to perform a grouping operation if the lasso is drawn around diagram strokes. We use the Microsoft Tablet PC SDK Divider API to classify the strokes within a lasso as being either drawings or text. If the strokes are drawings, they are grouped, otherwise they are considered to be an expression and mathematical recognition is performed. However, due to the complexity of this classification and the immaturity of Microsoft's implementation, this approach is not yet reliable and will require further research. In the meantime, we also provide an explicit gesture for grouping objects that can be distinguished from a recognition gesture based on the location of the tap. If the tap falls on the lasso, then we perform a grouping operation, otherwise we recognize the expression.

3.1.3 Associations

The essence of a mathematical sketch comes from making associations between mathematical expressions and diagrams. Associations are made between scalar mathematical expressions and angle arcs or one of the three special values of a diagram element, its *x*, *y*, or rotation coordinate. After an association is made, changes in mathematical expressions can be reflected as changes in the diagram and vice versa.

Associations between mathematical expressions and drawing elements can be made both explicitly and implicitly. Implicit associations are based on the familiar variable names and constant labels found in math and physics text illustrations. To create an implicit association, users draw a variable name or constant value near the intended drawing element and then use the math recognition gesture to recognize the label. If the recognition gesture's tap falls within the gesture's lasso, then the label is linked to the nearest drawing primitive; otherwise, the tap location is used to specify both the drawing element to be linked to the label and the drawing element's center of rotation (this point is only used for rotational labels). When labeling an angle arc, the location of the tap on the arc determines the *active line* – the line attached to the arc that will move when the angle changes. The apex of the angle is then marked with a green dot, and the active line is indicated with an arrowhead on the angle arc. Note, we do not detect or support cyclical association relationships, such as the specification of each angle in a triangle. MathPad^2 uses the recognized label and linked drawing element to infer associations with other expressions on the page (see Section 3.2.2).

For slightly more control over associations and to reduce the density of information in a diagram, associations can also be created explicitly without using variable name labels. The user can make an explicit association by drawing a line through a set of related math expressions and then tapping on a drawing element. After this line is drawn, drawing elements change color as the stylus hovers over them to indicate the potential for an association. However, this technique provides greater flexibility than the implicit association technique in specifying the precise point of rotation because, instead of just tapping on the drawing element (which sets the point of rotation to be the center of the drawing element), the user can press down on the element to select it, move the stylus, and then lift it to the desired center of rotation, even if it is not on the drawing element.

With both implicit and explicit associations, MathPad^2 provides an option for visualizing which drawing elements, labels and expressions are associated by filling the bounding boxes of all associated components with the same semi-transparent pastel color.

3.1.4 Using MathPad^2's ToolSet

In addition to diagram associations, MathPad^2 supports a range of computational functions on recognized mathematical expressions, including graphing, solving, simplifying, and factoring. This set of functions is rudimentary but gives some representative early results on adding advanced features to the system.

Users can graph recognized functions with a simple line gesture that begins on the function and ends at the graph's intended location. This line gesture is distinguished from other drawn lines by starting within the function expression's bounding box, being too long to be a mathematical symbol, and having no cusps or self-intersections. This gesture produces a movable, resizable graph widget displaying a plot of the function (see Figure 4). Additional graph gestures that end on this widget will overlay or replace the function being

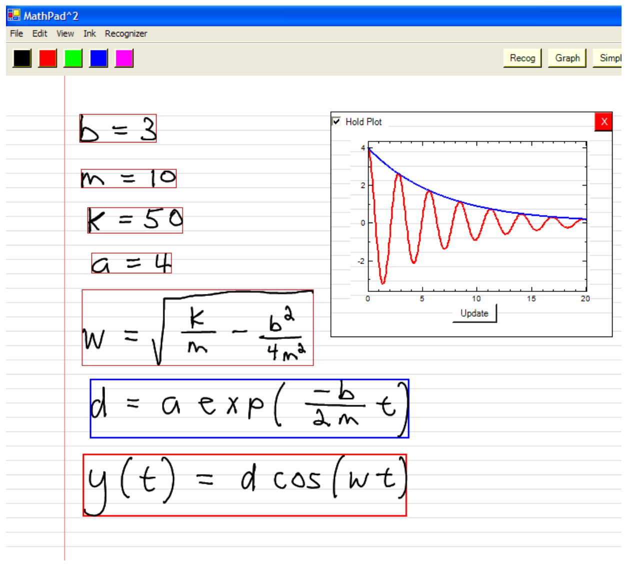


Figure 4: Two plots created using graph gestures. Expression bounding boxes are colored to correspond with plot lines.

graphed depending on the state of the 'hold plot" check box found in the upper left corner of the widget.

The graph widget uses default values for the domain of plotted functions based on a very simple heuristic: the domain is 0...5 for functions of *t* and −5...5 for functions of any other variable. We are currently developing ways to choose better defaults based on function characteristics. In any case, users can change the domain or range by selecting a region of the graph to zoom in on or by writing a new value below the start and/or end of the graph and then clicking the update button. Optional visual feedback is provided to show correspondences between a plot line and a mathematical expression by coloring the line the same as the expression bounding boxes.

In addition to graphing, MathPad^2 enables the user to solve equations. Users invoke the solver by making a squiggle gesture that starts inside the bounding box of a recognized mathematical expression. The system presents the solution to the user at the end of the squiggle gesture either as typeset symbols or ink in the user's handwriting style. Closely related gestures for simplifying and factoring expressions are presented in Figure 2.

3.2 Sketch Parser

We have focused so far on the user interface for specifying the input description of a mathematical sketch. We now discuss the components that prepare a sketch for execution by recognizing expressions, inferring associations, and rectifying drawings.

3.2.1 Mathematical Expression Recognition

There have been many different approaches to the recognition and parsing of mathematical expressions [Chan and Yeung 2000b]. After evaluating these recognition and parsing techniques and because of the lack of publicly available systems, we decided to implement our own custom recognizer and parsing engine. We chose a writer-dependent system because these systems tend to be more accurate than independent systems since the recognizer can be tailored to a particular user's handwriting [Connell and Jain 2002]. In addition, we could use the user's writing samples to present recognition results to the user in her own handwriting, so as to keep a pencil-and-paper look and feel.

Our multistage recognizer uses three different recognition techniques to form a hybrid solution [Li and Yeung 1997; Connell and Jain 2000; Smithies et al. 1999]. The first stage is the preprocessing step where ink strokes are normalized and filtered to reduce noise and made size- and translation-invariant. In addition, dominant points [Li and Yeung 1997] and their directions as well as other statistical features are calculated for use in the classification steps. The second stage in the recognition algorithm, coarse classification, is used to decrease the number of possible mathematical symbol candidates by rejecting unlikely ones and is intended to be fast and not fully exhaustive. The coarse classification scheme uses two separate algorithms. The first uses the direction information as input to a dynamic programming algorithm that calculates the optimal degree of difference or similarity between two characters [Li and Yeung 1997]. The second algorithm performs a statistical feature-set classification based on [Smithies et al. 1999]. The results of both classifiers are then merged using an averaging scheme normalized by their respective standard deviations. The third stage of the recognizer is fine classification, which takes the list of mathematical symbol candidates from the coarse classifier and uses dynamic programming once again to determine the best possible match with the training data [Connell and Jain 2000]. The fine classification's dynamic program is similar to that used by the coarse classifier but also uses dominant points as input, which helps detect small differences between similar mathematical symbols.

After the mathematical symbols are recognized they must be two-dimensionally parsed in terms of exponents, subscripts, fractions, and others constructs [Blostein and Grbavec 2001]. Therefore, using a simple grammar, our parser takes into account not only the symbols but their bounding boxes as well. Because Matlab^{TM} is our computational back end, the parsing system converts the 2D mathematical expressions into 1D Matlab specific expressions.

3.2.2 Association Inferencing System

As mentioned in section 3.1.3, implicit associations require a technique for determining the written mathematical expressions that should be associated with a particular drawing element based on the variable label linked to the element. An expression should be associated with a drawing element if that expression takes part in the computation of any variable that falls in the *label family* of the drawing element's variable label.

A label family is defined by its name, a root string. Members of the label family are variables that include that root string and a component subscript (e.g., x for its x-axis component) or a function specification. For example, if the user labels a drawing element ϕ_o, the inferencing system determines the label family to be ϕ and finds all mathematical expressions having members of the ϕ label family on the left-hand side of the equal sign: ϕ, ϕ_o, $\phi(t)$, $\phi_x(t)$, and so on. The inferencing system then finds all the variable names appearing on the right-hand side, determines their label families, and then continues the search. This process terminates when there are no more variable names to search for. Once all the related mathematical expressions have been found, they are sorted to represent a logical flow of operations that can be executed by a computational engine (e.g., Matlab), and the implicit association is made. For example, in Figure 1 the system would first store all of the terminal expressions, then the expressions that are not functions of t, and finally the time dependent functions. Note that our prototype does not currently handle interrelated equations such as $x(t) = t^2y(t)$ and $y(t) = x(t) - t$. However, in future versions of the system, we plan to support these types of dependencies by detecting them, solving them with our computational engine, and providing the user with the ability to interactively select the appropriate solutions to use in their sketches.

3.2.3 Defining Drawing Dimensions

Although an explicit Cartesian coordinate system can be created within a mathematical sketch diagram, many diagrams contain enough information that this can be done implicitly. MathPad^2 can infer coordinate systems in two ways: by using the initial locations of diagram elements and by labeling linear dimensions within a diagram (see Figure 5).

When two different drawing elements are associated with expressions so that each drawing element has a different value for one of its coordinates (x or y), then an implicit coordinate system can be defined. The distance along the coordinate shared between the two drawing elements establishes a dimension for the coordinate system, and the location of the drawing elements implies the location of the coordinate system origin.

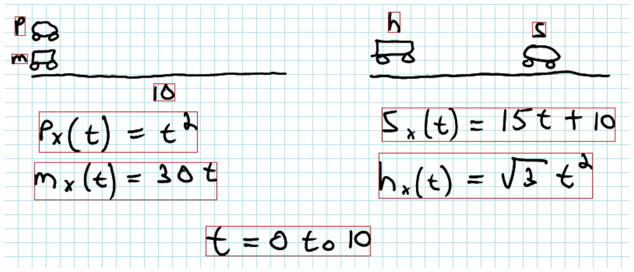


Figure 5: Two methods for inferring coordinate systems: the mathematical sketch on the left uses labeling of the ground line while the one on the right uses initial conditions.

Alternatively, if only one drawing element is associated with math, then the dimension of the coordinate system can still be inferred if another drawing element is associated with a numerical label. Whenever a numerical label is applied to a drawing element, MathPad^2 analyzes the drawing element: if it is a horizontal or vertical line, the corresponding x or y axis dimension is established; otherwise, we apply the label to the best-fit line to the drawing element and then establish the dimension of both coordinate axes.

If not enough information has been specified to define a coordinate system implicitly, then a default coordinate system is used. Like our graph defaults, we expect that we can improve these defaults from inferences based on an analysis of the function characteristics of the associated expressions.

3.2.4 Drawing Rectification

Mathematical sketches often have inherent ambiguities between what the mathematics specifies and what the user draws: that is, mathematical expressions often do not jibe precisely with their associated visual relationships in user drawings. *Rectification* is the process of fixing the correspondence between drawings and mathematics so that something meaningful is displayed. For example, a user might draw a diagram containing an angle, and then write some math that specifies that angle numerically. The drawn angle is unlikely to match the math description exactly and so either the drawing must be adjusted to match the math or vice versa.

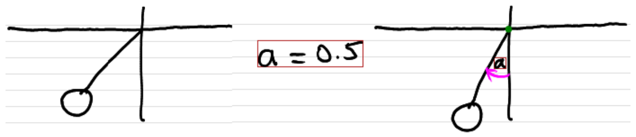


Figure 6: The effects of labeling an angle: the drawing is rectified based on the initial value of a (in radians). The green dot shows the rotation point and the magenta arrow indicates which part of the drawing will rotate.

Rectification is a difficult problem tantamount to the general constraint satisfaction problem. However, rectification in MathPad^2 is simplified since it is designed to handle only simple acyclic relationships between drawing elements and we do not check for cycles during this process. Our current implementation demonstrates rectification in the context of angle relationships. Mismatches between numerical descriptions of angles and their diagram counterparts are readily discernible. When an angle such as a in Figure 6 is associated with math, MathPad^2 rectifies the drawing in one of two ways. First, the angle between the two lines connected by the angle arc is computed. Next, the system determines if a mathematical expression corresponding to the angle label already exists. If so, it moves the active line, as determined by the angle arc, to the correct place based on the mathematical specification. If not, it uses the angle computed from the drawing as the numerical specification of the angle's value. Currently, this angle is represented internally and used during simulation.

A similar rectification process occurs when the same coordinate of more than two drawing elements is specified. If one or two coordinates are specified, no rectification is typically necessary since the coordinates of the elements can be used to establish dimensions of a coordinate system (see Section 3.2.3). However, when a third drawing element is added, it must be rectified to the coordinate system defined by the previous two drawing elements. A full treatment of the rectification issues of multi-element drawings is still under development.

3.3 Mathematical Sketch Animation and Output

The final subsystem of MathPad^2 is the animation engine. When users create a diagram and associates it with math, they are effectively drawing the initial conditions of an animation. MathPad^2 defines initial conditions as the value of expressions when the variable t is at its initial value, as defined by users when they write the mathematical expression $t = T_{initial} \ldots T_{final}$.[1]

In order to compute the animation, the animation subsystem first checks all drawing elements with associations to see if they are *animatable*. Animatable drawing elements are associated with functions of time. Currently the system supports x and y translational movement, rotation about a given point, and changing the value of an arc. The system takes these animatable drawing elements and sends their associated math to Matlab, which executes the math and returns the results.

[1]In actuality, the user draws a combination of initial values (for x, y, and rotation coordinate associations) and initial constants (for arc associations). In typical diagrams, however, initial constants are equivalent to initial conditions.

By default, MathPad^2 maps all animations so that they last four seconds in wall-clock time. Once again, our default is not appropriate for many animations, and so it can be overridden by specifying the duration as part of the time specification:

$t = T_{initial} \ldots T_{final}$ *in Seconds.*

4 An Example 2D Projectile Motion Scenario

We now present a scenario for how a user, say a high school physics student, might use MathPad^2 and mathematical sketching as a tool for better understanding mathematical concepts.

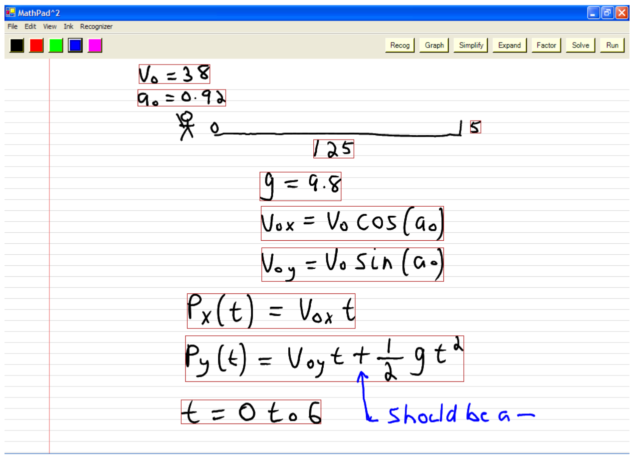


Figure 7: An ill-specified mathematical sketch for determining whether a baseball will fly over a fence for a home run. After running the sketch, the user will clearly see the error, since the ball will fly upward in a parabolic fashion. The remedy is shown in blue.

The student is interested in determining whether a baseball player can hit the ball over a fence given an initial velocity and angle. She first writes down a simple playing field as shown in Figure 7. Next she writes down the known quantities: the initial angle a_o, initial velocity v_o, and a gravitational term g. From her knowledge of projectile motion, she then writes down the mathematics as shown in Figure 7, labels the drawing making the required associations, and runs the simulation.

The simulation shows the ball moving upward against gravity, which does not look correct. The user then checks the equations and realizes that the equation $P_y(t) = v_{oy}t + \frac{1}{2}gt^2$ has a sign error. She scratches out the $+$, writes in a $-$, and re-recognizes the expression. She then runs the simulation again: the ball takes on the correct motion and barely makes it over the fence.

Next she wants to see how much farther the ball will go if v_o is increased. She scratches out the current value, writes in a larger one, and runs the simulation again. The ball does go farther but it also stops short of the ground, which leads her to the question "when will the ball hit the ground with these new parameters?" So, she takes equation $P_y(t)$, sets it equal to zero, and solves it with the squiggle gesture. Finally, she takes the second value for t, changes the time field, runs the simulation, and finds this time value is correct for the ball to hit the ground under the new parameters.

In this scenario, the student's understanding of the mathematical concept is augmented not only because she could visualize the behavior of the mathematical system, but also because she had to write down the mathematics in order to generate the animation. It is this unique feature that makes MathPad^2 and mathematical sketching a powerful problem-solving tool.

5 Discussion

About a dozen users have seen and tried the MathPad^2 prototype. Their response was quite positive. Users found the gestural interface easy to learn and use and commented on the fluidity of the modeless interaction style. One user stated that he wished he had had this software in high school. Many users asked whether they could solve more complex problems, such as a double pendulum, that often require open-form solutions. We believe that a much broader class of problems, including multi-body collisions and simple ordinary and partial differential equations, could be supported by extending mathematical sketching to handle basic programming constructs, such as loops and conditionals. We also recognize the desire for a macro facility that would let mathematical sketches be saved as functions for reuse within other sketches.

The user feedback also highlighted the effect of misrecognition and incorrect parsing on the overall usability of MathPad^2. In general, recognition and parsing errors increase in proportion to the complexity of the mathematical expressions. Users could generally resolve errors by simply scratching out the offending symbols, redrawing them and then re-recognizing the expression, although in some cases, they would need to perform this process multiple times, since our recognition algorithms do not adjust on the basis of previous attempts.

By training the symbol recognizer for each user, symbol recognition was generally reasonably robust. However, some users did encounter persistent symbol-recognition errors that can be attributed to ambiguous entry of specific pairs or groups of symbols, for instance, when distinguishing between 5 and s, x and $+$, and c, 1, and (. In some cases, we tried to exploit contextual knowledge to avoid common conflicts [Lee and Wang 1995]. For example, we would replace the 0 in $l0g$ with an o to form log, while a 5 in $5in$ would be replaced with an s to form sin. A few users chose to reduce the set of trained symbols to improve accuracy at the cost of limiting the scope of mathematical expressions. Other users changed the way they drew certain symbols such as drawing a 5 with two strokes to differentiate from a single stroke s.

We did not provide any means for the user to train the expression parser. Instead, we used a constant set of parsing rules based on the spatial relationships between symbol bounding boxes. As a result, parsing performance exhibited greater per-user variance depending on how closely the user's natural handwriting corresponded to our rules. Common parsing errors were to mischaracterize superscripts when the base symbol was an ascender letter (e.g., d, b) or subscripts when the base symbol was a descender letter (e.g., y, p). Also, some users wrote rather large subscripts and superscripts, which often resulted in parsing errors. We believe that most of these errors can be avoided by training the parser as well as the symbol recognizer. In other cases, parsing errors are more complicated, as when a misrecognized symbol triggers a parsing error. For example, in Figure 1, if the closing parenthesis of the exponential is misrecognized as a 1, our parser would aggregate the $cos(wt)$ as a subscript of the 1. However, we believe these kinds of errors can largely be avoided by exploiting context both within and between expressions, for example by avoiding subscripts for integers. Last, we need to present parsing results in a more natural way than our 1D representation, as by the technique described in [Zanibbi et al. 2001].

The current state of MathPad^2's recognition and parsing system can be a limiting factor for some users, especially those who must adjust their handwriting significantly to be neat and to correspond to our notion of the "correct" way to lay out expressions. Still, with adequate training most users find the system usable for the domain of mathematical sketches we currently support. To achieve broader acceptance and to support more complex mathematical sketches, we must continually incorporate the state of the art in mathematical recognition and parsing techniques.

6 Conclusion

We have discussed the novel concept of mathematical sketches – combinations of handwritten mathematics and free-hand diagrams – that allow rapid animated visualization of mathematical formulations. In addition, we described the system design issues encountered in creating a prototype mathematical sketching system, MathPad^2, including the design of a modeless gestural interface and techniques for implicitly associating and rectifying diagrams with mathematical expressions. Despite the restriction of the current prototype to closed-form expressions, we conjecture that mathematical sketching can be a powerful assistant in formulating and visualizing mathematical concepts.

Acknowledgements

We are grateful to Kazutoshi Yamazaki for implementing our custom recognition and parsing engine and to David Salesin for suggestions on improving an earlier version of MathPad^2. We also thank Andries van Dam, John Hughes, David Laidlaw, and the members of the Brown University Computer Graphics Group for valuable guidance and discussion, as well as the anonymous SIGGRAPH reviewers for their feedback. This work is supported in part by a gift from Microsoft and grants from the National Science Foundation and the Joint Advanced Distributed Co-Laboratory.

References

ALVARADO, C. J. 2000. *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Master's thesis, Massachusetts Institute of Technology.

BLOSTEIN, D., AND GRBAVEC, A. 2001. Recognition of mathematical notation. In *Handbook of Character Recognition and Document Image Analysis*, World Scientific, H. Bunke and P. Wang, Eds., 557–582.

BORNING, A. 1979. *ThingLab: A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University.

CHAN, K.-F., AND YEUNG, D.-Y. 2000. An efficient syntactic approach to structural analysis of on-line handwritten mathematical expressions. *Pattern Recognition 33*, 3, 375–384.

CHAN, K.-F., AND YEUNG, D.-Y. 2000. Mathematical expression recognition: A survey. *International Journal on Document Analysis and Recognition 3*, 1, 3–15.

CHAN, K.-F., AND YEUNG, D.-Y. 2001. Pencalc: A novel application of on-line mathematical expression recognition technology. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, 774–778.

CONNELL, S. D., AND JAIN, A. K. 2000. Template-based on-line character recognition. *Pattern Recognition 34*, 1, 1–14.

CONNELL, S. D., AND JAIN, A. K. 2002. Writer adaptation for online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 3, 329–346.

DAMM, C. H., HANSEN, K. M., AND THOMSEN, M. 2000. Tool support for cooperative object-oriented design: Gesture-based modelling on an electronic whiteboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 518–525.

FORSBERG, A., DIETERICH, M., AND ZELEZNIK, R. 1998. The music notepad. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 203–210.

GROSS, M. D., AND DO, E. Y.-L. 1996. Ambiguous intentions: A paper-like interface for creative design. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 183–192.

HECHT, E. 2000. *Physics: Calculus*. Brooks/Cole.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3d freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., 409–416.

KARA, L. B., GENNARI, L., AND STRAHOVICH, T. F. 2004. A sketch-based interface for the design and analysis of simple vibratory mechanical systems. In *Proceedings of ASME International Design Engineering Technical Conferences*.

LALEUF, J. R., AND SPALTER, A. M. 2001. A component repository for learning objects: A progress report. In *Proceedings of the First ACM/IEEE-CS Joint Conference on Digital Libraries*, ACM Press, 33–40.

LANDAY, J. A., AND MYERS, B. A. 1995. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press/Addison-Wesley Publishing Co., 43–50.

LEE, H.-J., AND WANG, J.-S. 1995. Design of a mathematical expression recognition system. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE Press, 1084–1087.

LI, X., AND YEUNG, D.-Y. 1997. On-line handwritten alphanumeric character recognition using dominant points in strokes. *Pattern Recognition 30*, 1, 31–44.

MATSAKIS, N. E. 1999. *Recognition of Handwritten Mathematical Expressions*. Master's thesis, Massachusetts Institute of Technology.

MILLER, E. G., AND VIOLA, P. A. 1998. Ambiguity and constraint in mathematical expression recognition. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 784–791.

MORAN, T. P., CHIU, P., AND VAN MELLE, W. 1997. Pen-based interaction techniques for organizing material on an electronic whiteboard. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 45–54.

MYNATT, E. D., IGARASHI, T., EDWARDS, W. K., AND LAMARCA, A. 1999. Flatland: New dimensions in office whiteboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 346–353.

SMITHIES, S., NOVINS, K., AND ARVO, J. 1999. A handwriting-based equation editor. In *Proceedings of Graphics Interface'99*, 84–91.

SPALTER, A. M., AND SIMPSON, R. M. 2000. Integrating interactive computer-based learning experiences into established curricula: A case study. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, ACM Press, 116–119.

VARBERG, D., AND PURCELL, E. J. 1992. *Calculus with Analytic Geometry*. Prentice Hall.

YOUNG, H. D. 1992. *University Physics*. Addison-Wesley Publishing Company.

ZANIBBI, R., NOVINS, K., ARVO, J., AND ZANIBBI, K. 2001. Aiding manipulation of handwritten mathematical expressions through style-preserving morphs. In *Proceedings of Graphics Interface 2001*, 127–134.

ZANIBBI, R., BLOSTEIN, D., AND CORDY, J. 2002. Recognizing mathematical expressions using tree transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 11, 1–13.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: An interface for sketching 3d scenes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, 163–170.

EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2006)
Thomas Stahovich and Mario Costa Sousa (Editors)

An Initial Evaluation of a Pen-Based Tool for Creating Dynamic Mathematical Illustrations

Joseph J. LaViola Jr.

Brown University, Department of Computer Science, USA
Email: jjl@cs.brown.edu

Abstract

MathPad^2 is a pen-based application prototype for creating mathematical sketches. Using a modeless gestural interface, it lets users make dynamic illustrations by associating handwritten mathematics with free-form drawings and provides a set of tools for graphing and evaluating mathematical expressions and solving equations. In this paper, we present the results of an initial evaluation of the MathPad^2 prototype, examining the user interface's intuitiveness and the application's perceived usefulness. Our evaluations are based on both performance and questionnaire results including first attempt gesture performance, interface recall tests, and surveys of user interface satisfaction and perceived usefulness. The results of our evaluation suggest that, although some test subjects had difficulty with our mathematical expression recognizer, they found the interface, in general, intuitive and easy to remember. More importantly, these results suggest the prototype has the potential to assist beginning physics and mathematics students in problem solving and understanding scientific concepts.

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [Information Interfaces and Presentation]: User Interfaces — Interaction Styles, Evaluation/Methodology

1. Introduction

MathPad^2(see Figure 1) is a pen-based, Tablet PC application prototype for creating dynamic illustrations used for exploring mathematics and physics concepts [LZ04]. The fundamental technology behind MathPad^{2}is mathematical sketching, a pen-based gestural interaction paradigm for mathematics problem solving that derives from the familiar pencil-and-paper process of drawing supporting diagrams to facilitate the formulation of mathematical expressions; however, with mathematical sketching, users can also leverage their physical intuition by watching their hand-drawn diagrams animate in response to continuous or discrete parameter changes in their written formulas [LaV05]. Diagram animation is driven by associations that are inferred, either automatically or with gestural guidance, from handwritten mathematical expressions, diagram labels, and drawing elements.

The essential goal in developing the MathPad^{2}user interface was that it be as similar and fluid as pencil and paper, since mathematics and physics problems are often solved using this medium. Thus, we did not want to use any additional hardware (e.g., a modifier key or stylus button) or software (e.g., buttons) modes. Instead, we wanted all interaction to be derived from using digital ink. We developed a gestural user interface for invoking different operations in MathPad^{2}because we wanted users able to work as fluidly as possible with the mathematics and drawings they create. We wanted to explore whether our choice of gestures, which by themselves are not part of pencil-and-paper interaction, are thought of as intuitive or at least complimentary to pencil and paper.

Given the foundations for MathPad^2, we performed an initial usability evaluation to gauge users' performances and reactions to the prototype to validate its design and potential benefit and determine if further, more in-depth studies are needed. More specifically, we are interested in how easy it is for users to use MathPad^{2}with only a visual demonstration of how to invoke gestural operations, and in how many mistakes they make in performing various MathPad^{2}tasks. We are also interested in how well subjects remember various gestural commands, since this is a good indicator of intuitiveness. Using interface satisfaction [CDN88] and perceived usefulness [Dav89] questionnaires, we are addition-

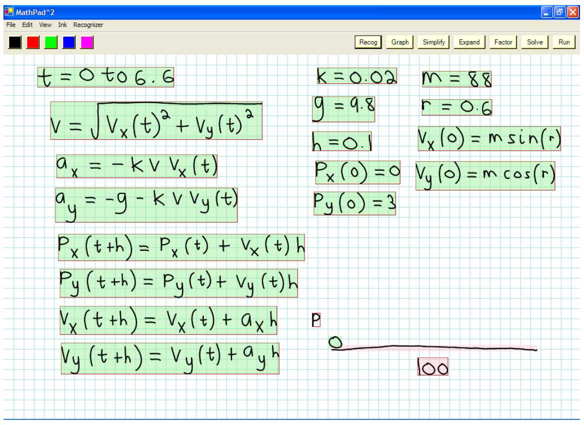

MathPad^2
File Edit View Ink Recognizer
Recog Graph Simplify Expand Factor Solve Run
$t = 0$ to 6.6
$k = 0.02$
$m = 88$
$V = \sqrt{V_x(t)^2 + V_y(t)^2}$
$g = 9.8$
$r = 0.6$
$h = 0.1$
$V_x(0) = m\sin(r)$
$a_x = -kVV_x(t)$
$P_x(0) = 0$
$V_y(0) = m\cos(r)$
$a_y = -g - kVV_y(t)$
$P_y(0) = 3$
$P_x(t+h) = P_x(t) + V_x(t)h$
$P_y(t+h) = P_y(t) + V_y(t)h$
$V_x(t+h) = V_x(t) + a_x h$
$V_y(t+h) = V_y(t) + a_y h$
P
100

Figure 1: *A mathematical sketch, created in MathPad^2, illustrating how air drag affects a ball's 2D motion. Associations between mathematics and drawings are color-coded.*

ally interested in whether subjects would use mathematical sketching in their work and why.

2. Related Work

The idea of using computers to create dynamic illustrations of mathematical concepts has a long history. One of the earliest dynamic illustration environments was Borning's ThingLab, a simulation laboratory environment for constructing dynamic models of experiments in geometry and physics, that relied heavily on constraint solvers and inheritance classes [Bor79]. Other systems such as Interactive Physics^{TM} and The Geometer's SketchPad^{TM} also let the user create dynamic illustrations; these systems are all WIMP-based (Windows, Icons, Menus, Pointers) resulting in a significant amount of mode switching and loss of fluidity within the interface. In addition, they do not allow the user to write handwritten mathematics to create these illustrations. Because MathPad^2 uses handwritten mathematical expressions, users can leverage their knowledge of mathematical notation in order to create mathematical sketches. Java applets that provide both interactive and dynamic illustrations have also been developed for exploring various mathematics and physics principles [CT98]. However, these applets are not general, typically provide limited control over the illustration, and rarely show the user the mathematics behind the illustration.

Alvarado [Alv00] and Kara [KGS04] let the user make sketched diagrams that are recognized as drawing primitives with domain knowledge from specific disciplines and then animated. Although these systems provide powerful illustrations of physics and mathematical concepts, they are limited because of their domain knowledge and because they hide the underlying mathematical formulations from the user. Pen-based systems have also been developed for other types of dynamic illustration. For example, Pickering et al. developed a system for sketching football plays, simulating them, and then creating a dynamic illustration of the play outcome [PBLP99] while Davis et al. developed a pen-based system for creating traditional animations [DACP04].

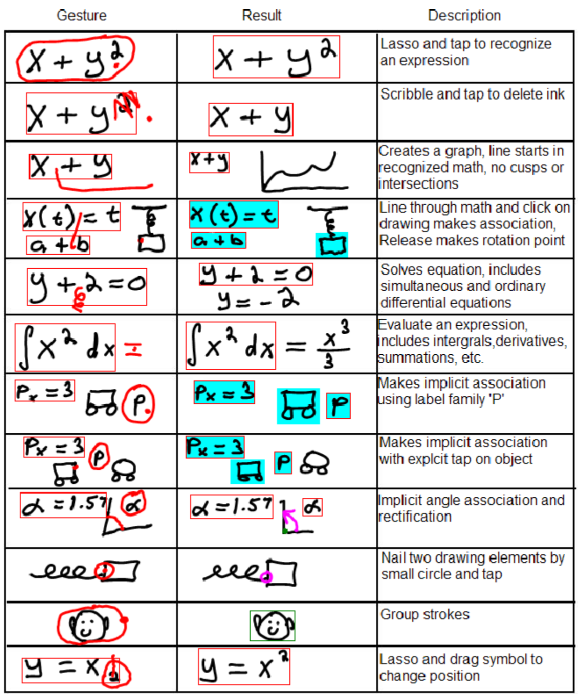
MathJournal, developed by xThink, Inc., is the closest in spirit to MathPad^{2}because its animation controls let users write down and recognize mathematics, make drawings, and assign the mathematics to the drawings. However, a key limitation of MathJournal's animation control is that users must keyframe their animations (typically providing a starting and ending frame), making the user interface less fluid and contravening how users would make diagrams with pencil and paper. In addition, MathJournal's animation control lacks the iteration and conditional constructs, diagram rectification, and modeless gestural user interface that mathematical sketching supports.

3. The MathPad^2 User Interface

To make mathematical sketches in MathPad^2, users write down mathematics, make drawings, and make associations between the two. Additionally, users can invoke mathematical tools such as graphing, function evaluation, and equation solving to help create and manipulate their sketches. In this section, we describe how users perform these tasks with MathPad^2's modeless gestural user interface. A summary of the commands are found in Figure 2.

When designing our modeless gestural interface, we wanted the gestures not to interfere with the entry of drawings or equations and still be direct and natural enough to feel fluid. To accomplish this, we use context sensitivity to determine what operations to perform with a single gesture. We also use the notion of punctuated gestures, compound gestures with one or more strokes and terminal punctuation, to help disambiguate gestures from mathematics and drawings. We also wanted to ensure that gestures which seem logical for more than one command should be used for all of those commands. For example, if a particular gesture makes sense for two or three different operations, then we want that gesture to invoke all those operations. More details on the design of and methodology behind these gestures can be found in [LZ04, LaV05].

To write mathematical expressions, users simply write them down using the stylus as if they were using pencil-and-paper. To have the system recognize a mathematical expression, users must lasso the expression and make a tap inside the lasso. Recognized symbols are presented to users in their own handwriting since MathPad^{2}has handwriting samples from individual users as a result of our writer-dependent mathematical expression recognition engine. When users move the stylus over the bounding box of the recognized mathematical expression, a green button appears in the box's lower right corner, and when pressed, shows whether the expression was parsed correctly. If a mathematical expression



Gesture	Result	Description
		Lasso and tap to recognize an expression
		Scribble and tap to delete ink
		Creates a graph, line starts in recognized math, no cusps or intersections
		Line through math and click on drawing makes association, Release makes rotation point
		Solves equation, includes simultaneous and ordinary differential equations
		Evaluate an expression, includes intergrals,derivatives, summations, etc.
		Makes implicit association using label family 'P'
		Makes implicit association with explcit tap on object
		Implicit angle association and rectification
		Nail two drawing elements by small circle and tap
		Group strokes
		Lasso and drag symbol to change position

Figure 2: *MathPad^2's gestural commands. Gesture strokes in the first column are shown here in red. In the second column, cyan-highlighted strokes provide association feedback (the highlighting color changes each time a new association is made), and magenta strokes show nail and angle association/rectification feedback.*

is recognized incorrectly, users can simply erase the offending symbols using a scribble erase gesture followed by a tap and then re-recognize the expression. Users can also tap on a recognized symbol to get a list of alternates. If there is a parsing error with the mathematical expression, users can lasso the offending symbols and interactively move them to a new location where the complete expression will be reparsed.

Users make drawings in the same way they write mathematical expressions except that the ink strokes need not be recognized. We refer to these ink strokes as drawing elements and they can be grouped together to form composite drawing elements. Users lasso the drawing elements they want to composite and make a tap on the lasso line. Tapping on the lasso line distinguishes this operation from recognizing mathematical expressions. Users can also nail drawing elements together by drawing a small circle over them and making a tap inside the circle. Nailing drawing elements together lets users make stretchable objects. Note that the drawn circle must not completely contain any drawing elements in order to be recognized as a nail gesture. This constraint distinguishes it from the gesture for making composite drawing elements and recognizing mathematical expressions.

One of the most important components of MathPad^{2}is the ability to associate mathematics to drawing elements so they know how to behave during an animation. Users can make associations either explicitly or implicitly. Users make explicit associations by simply drawing a line through the bounding boxes of all the necessary mathematical expressions and tapping on a particular drawing element. As the stylus hovers over drawing elements, they highlight to give users feedback about which drawing element they will select. Implicit associations are made by labeling a drawing element with a variable name or constant value and can be either point or angle associations. Point associations are made in the same way that mathematical expressions are recognized except the tap is made on the drawing element instead of inside a lasso. Angle associations are made by drawing an angle arc and label. Then users lasso the label and make a tap whose location on the arc determines the *active line* — the line attached to the arc that will move when the angle changes. The apex of the angle is then marked with a green dot, and the active line is indicated with an arrowhead on the angle arc. In either case, MathPad^{2}uses the label to find all of the required mathematical expressions that should be associated to the drawing element.

Finally, MathPad^{2}provides users with a mathematical toolset for graphing and evaluating functions as well as solving equations that can assist users in making mathematical sketches. Users graph functions by simply drawing a sufficiently long, smooth line with no self-intersections, starting inside the bounding box of a recognized mathematical expression, intersecting any other functions along the way, and ending outside all expression bounding boxes. This gesture creates a graph control widget where users can view plots of the functions the graph gesture has intersected and also change the domain and range of the functions by writing down the values and pressing the update button.

Users evaluate mathematical expressions such as integrals, summations, and derivatives by writing an equal sign to the right of the expression and making a tap inside the equal sign's bounding box. The results are then displayed to the right of the drawn equal sign. Users solve single, simultaneous, or ordinary differential equations, by making a squiggly gesture (see Figure 2). This gesture is identical to the graphing gesture except the line must contain two self-intersections. The results are then displayed underneath the last intersected equation.

4. MathPad^2 Evaluation

4.1. Experimental Design and Tasks

The goal of our initial usability experiment is to get users' reactions to the prototype to validate the user interface design and its potential benefit as well as determine if further, more in-depth studies are needed. More specifically, we wanted to evaluate the intuitiveness of MathPad^2's user interface and gauge the perceived usefulness of the tool. Writing down mathematical expressions and making drawings is a fairly intuitive task, and although our gestural commands need to be taught, we felt they were designed so that they should be easy to understand given simple demonstrations of their use.

In the experiment, subjects must complete six tasks representing common interactions that a student or teacher would perform with MathPad^2. Before a subject performs each task, the experimenter shows the subject how to perform the required gestures for that task via demonstration only. Tasks 1–3 were designed to test how well users were able to use the graph, equation solving, and expression evaluation gestures. First, subjects are shown how to write and recognize mathematical expressions using the lasso and tap gesture, how to erase ink using the scribble erase gesture, and how to use the correction user interface. Then, they are shown how to perform each task specific gesture or command. For task 1 (Graphing), after being shown the required gestural commands, the subjects write, recognize, and then graph $y = x$, $y = x^3$, and $y = cos(x)e^x$. Then subjects change $y = x^3$ to $y = x^2$, graph the function, and change the function's domain from $-5\ldots5$ to $0\ldots8$. For task 2 (Equation Solving) task, subjects write down and recognize $x^2 - 16x + 13 = 0$ and solve the equation. Next, subjects write and recognize $x^2y + 2y = 4$ and $3x + y = 2$ and solve this set of simultaneous equations. For task 3 (Expression Evaluation), subjects write down the following expressions and evaluate them:

- $\int_0^2 x^2 dx$
- $y = \int x^2 cos(x) dx$
- $\frac{dy}{dx}$
- $\frac{d^2y}{dx^2}$
- $\sum_{l=0}^{5}(l-1)^2$.

In all tasks, subjects are instructed to use the correction user interface if the recognizer incorrectly recognizes symbols or expressions.

Tasks 4–6 were designed to lets users make mathematical sketches and evaluate whether they prefer to use implicit or explicit associations. Task five also was designed to evaluate how well subjects can make nails. Note that only task four required subjects to write down the necessary mathematical expressions. Tasks five and six used prewritten mathematical expressions because we felt having them write and recognize these expressions was not needed, given the many expressions they had already written in the mathematical expression recognition study (see Section 4.4). However, with task four, we wanted to see how well subjects could make a mathematical sketch from beginning to end.

The fourth task (Bouncing Ball), has subjects create a complete mathematical sketch of an object bouncing along the ground. Subjects write and recognize the four mathematical expressions shown in Figure 3, make a drawing with a horizontal line representing the ground and a composite drawing element consisting of three circles drawn near the start of the horizontal line. Next, subjects write the number 20 and associate it to the horizontal line. Finally, subjects associate the mathematics to the composite drawing element, either choosing an explicit association or using an implicit association with the letter "p" as a label, and run the sketch. Note that if MathPad^2 fails to recognize subjects' mathematical expressions after several attempts, we provide them with prewritten expressions. However, we do not make them aware of this when the instructions for this task are given.

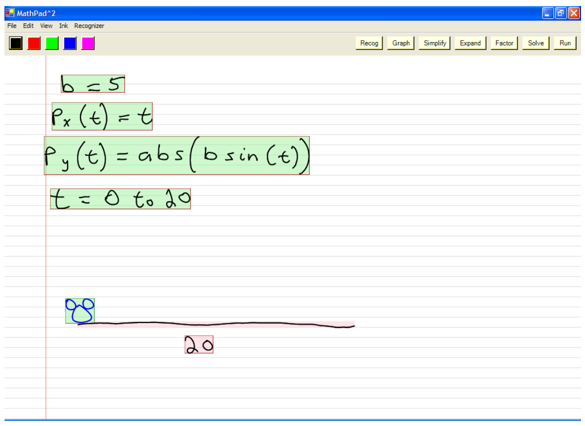


Figure 3: *The fourth task in the MathPad^2 usability test.*

The fifth task (Oscillator) has subjects create a mathematical sketch illustrating damped harmonic oscillation. The experimenter instructs subjects to first draw a line and make seven nail gestures along that line. This subtask does not have anything to do with the mathematical sketch itself, but gives us additional accuracy data on how well subjects can perform the nail gesture. Subjects make a drawing consisting of a horizontal line, a spring underneath the line, and a box underneath the spring (see Figure 4). Subjects then use two nail gestures to nail the horizontal line to the spring and the spring to the box. Next, subjects associate the mathematics to the box, using an explicit or implicit association with the letter "y" as a label, and run the sketch.

In the last task (2D Motion), subjects create a mathematical sketch illustrating 2D projectile motion subject to air resistance (see Figure 1). Subjects draw a horizontal line and a ball near the left side of the horizontal line. They then associate the number 100 to the horizontal line. Finally, subjects associate the mathematics to the ball, using an explicit or implicit association with the letter "p" as a label, and run the sketch. After all six tasks are completed, subjects answer a post-questionnaire.

4.2. Participants

Seven subjects (four men and three women), participated in the MathPad^2 usability evaluation. Subjects were recruited from the Brown University undergraduate population and

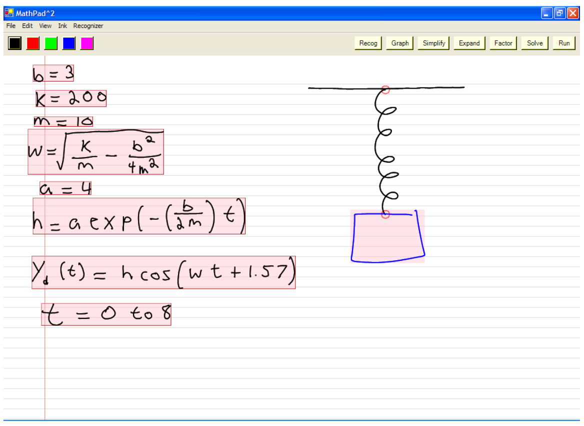

MathPad^2
File Edit View Ink Recognizer
Recog Graph Simplify Expand Factor Solve Run
$b = 3$
$k = 200$
$m = 10$
$w = \sqrt{\frac{k}{m} - \frac{b^2}{4m^2}}$
$a = 4$
$h = a\exp\left(-\left(\frac{b}{2m}\right)t\right)$
$y_1(t) = h\cos(wt + 1.57)$
$t = 0$ to 8

Figure 4: *Subjects create a damped harmonic oscillator in the fifth task.*

were either physics or applied mathematics majors. We chose this particular user population because MathPad^2 was designed for mathematics and physics students. Subjects' ages ranged from 19 to 23 and all were right-handed; only one had used a pen-based computer before (a PDA). All seven subjects were asked prior to the study if they had used mathematical software before and which packages: six subjects answering yes and had used a variety of different packages including Matlab, Mathematica, and Maple. All seven subjects were paid $30 for their time and effort.

4.3. Evaluation Measures

We evaluate MathPad^2's usability using quantitative and qualitative data from subjects' task performances and from a post-questionnaire. As subjects perform the six experimental tasks, the experimenter records important information about subjects' performances in completing each task, the decisions they made, and counts their mistakes. Performance is characterized by whether subjects can complete each task and how well they do on each subtask. Therefore, the experimenter records whether or not subjects make the appropriate gestures correctly and, if so, whether on the first attempt. Knowing how well subjects perform gestural operations on their first attempt is an important measure because it tells us how easy the gestures are to make and remember. The experimenter also records subjects' choices of implicit and explicit associations in tasks 4–6 so as to get a quantitative metric for their preferences.

After subjects have completed all six tasks they are given a post-questionnaire designed to get their reactions to the MathPad^2 user interface and its perceived usefulness as well as assess how well they remember certain gestures. The post-questionnaire consists of four parts. The first and second parts are adapted from Chin's Questionnaire for User Interface Satisfaction [CDN88] and asks subjects to rate MathPad^2's user interface as a whole and its individual components. The third part of the post-questionnaire, the recall test, asks subjects to show what gestures they would use for six different operations. The fourth part of the post-questionnaire was adapted from the Perceived Usefulness portion of Davis's questionnaire for user acceptance [Dav89] and asks whether subjects would use MathPad^2 in their work. After subjects answer the post-questionnaire, the experimenter reviews it with them to make sure their answers are clear and to elaborate further on any specific parts of MathPad^2.

4.4. Mathematical Expression Recognition

An important part of MathPad^2's user interface is that users can write down mathematical expressions as if they were using pencil and paper. Thus, mathematical expression recognition accuracy is an important part of the overall user experience. MathPad^2 uses a writer-dependent mathematical expression recognizer [LaV05] that includes a mathematical symbol recognizer and a mathematical expression parsing system. Each test subject had to provide handwriting samples to train the recognizer and this task took 50 minutes per subject. Note that subjects were given rest periods to ensure they did not get tired during training. Before completing the MathPad^2 tasks, we also had subjects write down symbols and a set of mathematical expressions to test the recognizer's accuracy Overall, the recognizer recognized symbols correctly 95.1% of the time with a standard deviation of 2.65%. The parsing component of our mathematical expression recognizer made correct parsing decisions 90.8% of the time with standard deviation of 4.47%. More detailed results on the mathematical expression recognition evaluation can be found in [LaV05].

4.5. Results and Discussion

4.5.1. Task Performance Results

For the first three tasks, subjects were able to write and recognize all of the mathematical expressions fairly easily. In some cases, they had to use the correction user interface to fix recognition errors, generally getting MathPad^2 to recognize their expressions on the second or third attempt. 27 out of 28 graphing operations (four per subject) were made on the first attempt. Subjects also had to change the domain of a graph; they all completed this operation on the first attempt. 12 out of 14 equation-solving operations (two per subject) were made on the first attempt. The other two equation solves were correctly performed on the second attempt. 34 out of 35 expression evaluations (five per subject) were made on the first attempt. One subject, however, did have difficulty in getting MathPad^2 to recognize $\frac{d^2y}{dx^2}$ and even after multiple attempts was not able to evaluate the expression.

All seven subjects were able to complete tasks 4–6 making the dynamic illustrations. Subjects also had no difficulty in making the drawings for each task and only once did a subject have trouble making a composite drawing element. In the Bouncing Ball task, 12 out of 14 associations were made on the first attempt and 8 of them were done implicitly. Three subjects did have difficulty in getting MathPad^{2}to recognize the required mathematical specification for the Bouncing Ball task and, after multiple attempts (about 10 minutes), were given prewritten expressions. The difficulty was not in symbol recognition, but in expression parsing. Two of these subjects had parsing decision accuracies below 90% in the mathematical expression test while the other subject's accuracy was 92%. This result provides evidence indicating that higher parsing decision accuracy is needed. In the Spring task, 56 out of 63 nails (seven per subject) were made on the first attempt. Most of the remaining nails were made on the second attempt. However, one subject required several attempts to make the necessary nails and had to recreate the drawing after inadvertently erasing part of it when erasing an incorrectly recognized nail. Subjects had to make one association in this task, and all seven were made on the first attempt explicitly. For the 2D motion task, subjects made 12 out of 14 associations on the first attempt with all of them made implicitly. One subject did had some difficulty with the implicit associations and needed several attempts to make them correctly.

Overall, subjects did well on all six tasks, considering they had no hands-on training beforehand. Their first attempt performances are summarized in Table 1. Subjects hand no difficulty in making a lasso and tap to recognize mathematical expressions or in using the scribble erase gesture. In only one case did a subject not complete part of a task and this was due to MathPad^2's inability to recognize an expression correctly. Subjects made 160 out of the 175 gestural operations correctly (91.4%) on their first attempt. This number is high considering that subjects had not practiced any of the gestural commands. One subject did have some difficulty with implicit associations due to problems with making taps. The greatest problem subjects had with the six tasks was obtaining correctly recognized expressions in certain situations. That three out of the seven subjects required prewritten mathematics for the Bouncing Ball task shows that the mathematical expression recognizer needs improvement.

4.5.2. Post-Questionnaire Results

Overall Reaction. Table 2 summarizes subject's overall reaction to MathPad^{2}and shows that they had a positive reaction to the prototype. When subjects were asked why they chose their rankings, most asserted that MathPad^{2}works well, is easy to use, and would be very useful for students in a classroom setting and/or doing homework problems. One subject was "amazed at the application's power". Two subjects claimed MathPad^{2}was easy to use but could be frustrating when it had trouble recognizing their handwriting; this frustration explains why the second and third rankings in Table 2 are slightly below the first and fourth rankings.

Ease of Use. Subjects rated different parts of the MathPad^{2}user interface from 1 (easy) to 7 (hard). Table 3 summarizes these results and shows that subjects found the tasks they had to perform easy to do. Subjects gave recognizing expressions the highest average ranking, indicating the fact that some users had trouble getting MathPad^{2}to recognize their handwriting. When asked about their ranking, they stated that the gesture for recognizing mathematical expressions (i.e., lasso and tap) was easy to do, but the results of the recognition operation led them to choose a higher ranking on the easy (1) to hard (7) scale.

First Attempt Gesture Performance Summary			
	Completed	Total	Percentage
Graphing:	27	28	96%
Equation Solving:	12	14	86%
Exp. Evaluation:	34	35	97%
Nails:	56	63	88%
Associations:	31	35	89%
Total:	160	175	91.4%

Table 1: *A breakdown of test subjects' first attempt gesture performance.*

Overall Reaction to MathPad^2		
	Mean	Std. Deviation
Terrible=1, Wonderful = 7	6.42	0.54
Difficult=1, Easy=7	5.57	0.98
Frustrating=1, Satisfying=7	5.57	1.13
Dull=1, Stimulating=7	6.14	0.38

Table 2: *Subjects' average ratings of their overall reaction to MathPad^{2}on a scale from 1 to 7.*

MathPad^{2}User Interface Ease of Use		
	Mean	Std. Deviation
Writing Mathematics	1.43	0.97
Recognizing Mathematics	2.57	1.81
Graphing Functions	1.0	0.0
Solving Equations	1.0	0.0
Evaluating Expressions	1.0	0.0
Grouping Drawing Elements	1.57	0.79
Making Associations	1.71	0.76
Making Nails	1.57	0.59

Table 3: *Subjects' average ratings of ease of use for different components of the MathPad^{2}user interface (scale: 1=easy, 7=hard).*

Association Preference. All seven subjects preferred explicit associations, claiming they were easier to remember and simpler and faster to perform. However, they did say that when associations need to be made with a drawing element and a large set of mathematical expressions, the implicit method is more appropriate. We can thus conclude that both association methods have their place in mathematical sketching.

Correction User Interface. Five out of the seven subjects tested found the correction user interface helped them. The two subjects who said no claimed that the alternate lists gave them no help in correcting recognition errors. One subject wanted more choices to appear in the alternate lists, especially in the equation alternate list.

Positive and Negative UI Aspects. Most subjects identified the most positive aspect as its ability to quickly make drawings move as described by mathematical equations. Two subjects claimed that solving equations was one of the user interface's most positive aspect. One subject thought that the best part of MathPad^2's user interface was the scribble erase command; another subject said the user interface's simplicity was its most positive aspect. Three subjects stated that getting MathPad^2 to recognize certain symbols and equations correctly was the most negative aspect of the user interface. Two subjects stated that the lack of interactive feedback for implicit associations was a significant drawback, and one subject stated that a negative aspect was the time necessary to get used to the gestural commands. Finally, two subjects said that MathPad^2's user interface had no negative aspects.

Overall Ease of Use. On average, subjects gave MathPad^2 a 1.86 (1 equals easy and 7 equals hard) with a standard deviation of 0.69. When they were asked to explain their ratings, two dominant themes emerged. First, subjects found the interface easy to use and remember, but were in some cases frustrated by problems in mathematical expression recognition. However, the subjects who had trouble with recognition all felt it would improve with more practice. Those subjects were also asked if they would still use MathPad^2 in spite of their recognition problems; they all said they could deal with these problems because of the functionality MathPad^2 would give them. Second, subjects felt the interface was easy to use once it was explained, a result that helps to validate our demonstration-based teaching protocol.

Gesture Recall Test. Subject were asked how to invoke gestural commands for graphing, solving equations, evaluating expressions, recognizing a mathematical expression, making nails, and making implicit associations. This part of the questionnaire took place about 5 to 10 minutes after they used MathPad^2. Subjects answered 38 out of the 42 recall questions correctly (six per subject) for a recall rate of 90%. Of the four questions subjects answered incorrectly, three subjects missed the equation solving gesture (squiggle) and one missed the expression evaluation gesture (equal and tap). The 90% recall rate indicates that subjects had little difficulty remembering MathPad^2 gestures except for the equation solving gesture. Even though three out of the seven subjects forgot the equation solving gesture, they still claimed it was easy to use based on their mean ranking in Table 3.

Likely Usage. Table 4 summarizes subjects' ratings on the different "perceived usefulness" statements, on a scale of 1 (unlikely) to 7 (likely). Most subjects would use MathPad^2 in their work. When asked to explain their ratings, four subjects stated that the application would help them to do their classwork and obtain a better understanding of problems and concepts. However, there was no consensus on whether MathPad^2 would speed their understanding of these problems and concepts. One subject said that the ability to quickly solve equations and make graphs would be very beneficial. Two subjects said they did not think they would use MathPad^2 in its current form in their work (explaining the high standard deviations in Table 4). Both of these subjects work in theoretical physics, one in optics and the other in modern physics. However, one of these subject stated she would have used MathPad^2 during beginning physics classes while the other stated he would use MathPad^2 if it had support for light ray and optics diagrams. Finally, all seven subjects felt the application would be a good tool for teachers of introductory mathematics and physics classes.

MathPad^2 Perceived Usefulness		
	Mean	Std. Deviation
Accomplish Tasks Faster	5.14	1.95
Improve Performance	4.71	2.36
Increase Productivity	5.0	1.91
Enhance Effectiveness	5.14	2.04
Easier To Do Work	5.57	1.90
Useful In Work	5.42	2.37

Table 4: *Subjects' average ratings of the perceived usefulness of MathPad^2 in their work (scale: 1=unlikely, 7=likely).*

4.5.3. Discussion

The results of our initial MathPad^2 usability study suggest that, based on our evaluation criteria, the MathPad^2 user interface is, in general, intuitive with subjects picking up the interface with relative ease. With only minimal training, most gestures are easy to remember and use. However, if we examine the first attempt task performance results (Table 1) in conjunction with the recall test from our post-questionnaire, we see that the equation solving gesture has the lowest first attempt accuracy and was the most difficult to remember. This indicates that this gesture is not as intuitive as the others. Additionally, if we look deeper into users' preferences for making associations, we see that they preferred explicit associations and of the four associations that were not made on their first attempt, all four were implicit. Again, this result suggests that explicit associations are more intuitive than implicit ones. First attempt performance for making nails was also a bit lower than expected, but we feel this might have been an implementation issue. In terms of perceived utility, subjects think the application is a powerful tool that beginning physics and mathematics students could use to help solve problems and better understand scientific concepts.

Most subjects performed the tasks with little trouble, while a few had some difficulty, stemming primarily from problems with mathematical expression recognition. However, these subjects also said they were willing to accept these recognition problems, given what MathPad^{2}can offer them. This result is somewhat contrary to our expectations about the negative impact of our mathematical expression recognizer on MathPad^{2}usability. Nevertheless, we need better mathematical expression recognition that will perform robustly across a larger user population. Although these results do not tell us how much more accurate the recognizer needs to be, its clear that a mean accuracy of 90.8% for making correct parsing decisions is too low. A better correction user interface could also go a long way to helping with users' frustrations when incorrect recognitions occur. In addition, more interactive feedback is needed for implicit associations, and the equation solving gesture should be redesigned.

Although the results of our initial evaluation are positive, we recognize it can be argued that there are two limitations with our study. First, we only used seven test subjects. We could have had more subjects, but we felt that seven was appropriate for an initial evaluation of MathPad^{2}and its gestural interface, given one of our main goals was to determine whether larger studies were needed. Second, we did not compare MathPad^2's user interface with any other interface metaphors. Although this could be considered a limitation, our goal in this evaluation was to determine how well users could use the MathPad^{2}interface, not whether it was better than any other interface. For this work, we feel our experimental design was suited to answering our intended questions. However, as we perform future usability tests to gain a deeper understanding of the benefits of mathematical sketching, we will need more comparative experimental designs with larger subject numbers.

Given the results of our evaluation, we plan to make improvements to MathPad^{2}by adding more functionality and improving the weaker points of the interface as well as improving the parsing component of our mathematical expression recognizer. Given the generally positive results of our evaluation, we are confident in pursuing further MathPad^{2}experimentation. Thus, we plan to explore the pedagogical benefits of MathPad^{2}in a summative evaluation where students will use MathPad^{2}as part of a mathematics or introductory physics course.

5. Conclusion

We have presented an initial evaluation of MathPad^2, a prototype application for making dynamic illustrations using the mathematical sketching paradigm, to test its intuitiveness and perceived utility. Our evaluation suggests that MathPad^2's user interface is generally intuitive, although some parts of the interface need to be reevaluated. Additionally, the MathPad^{2}application is perceived to be a powerful tool for exploring mathematics and physics concepts. Although some of our test subjects had some difficulty with getting the system to recognize their mathematical expressions, they still gave MathPad^{2}positive feedback and would use MathPad^{2}regardless of these issues because of its functionality. These results also support future MathPad^{2}development and longer term evaluations.

Acknowledgements

Special thanks to Robert Zeleznik and Andries van Dam for valuable discussions. This work is sponsored in part by a gift from Microsoft.

References

[Alv00] ALVARADO C.: *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Tech. rep., Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2000. 2

[Bor79] BORNING A.: *A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979. 2

[CDN88] CHIN J., DIEHL V. A., NORMAN K. L.: Development of an instrument measuring user satisfaction of the human-computer interface. In *Proceedings of the ACM Conference on Human Factors and Computing Systems (CHI'88)* (1988), pp. 213–218. 1, 5

[CT98] CHRISTIAN W., TITUS. A.: Developing web-based curricula using java physlets. *Computers in Physics 12*, 3 (May-June 1998), 227–232. 2

[DACP04] DAVIS J., AGRAWALA M., CHUANG E., POPOVIC Z.: A sketching interface for articulated figure animation. In *Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation* (2004), pp. 320–328. 2

[Dav89] DAVIS F. D.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly 13*, 3 (Sept. 1989), 319–340. 1, 5

[KGS04] KARA L. B., GENNARI L., STAHOVICH T. F.: A sketch-based interface for the design and analysis of simple vibratory mechanical systems. In *Proceedings of ASME International Design Engineering Technical Conferences* (2004). 2

[LaV05] LAVIOLA J.: *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*. PhD thesis, Brown University, 2005. 1, 2, 5

[LZ04] LAVIOLA J., ZELEZNIK R.: Mathpad^2: A system for the creation and exploration of mathematical sketches. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23*, 3 (August 2004), 432–440. 1, 2

[PBLP99] PICKERING J., BHUPHAIBOOL D., LAVIOLA J., POLLARD N.: *The Coach's Playbook*. Tech. rep., Master's Thesis, Department of Computer Science, Brown University, CS-99-08, May 1999. 2

Bibliography

Bibliography

[1] Adler, Arron and Randall Davis. Speech and sketching for multimodal design. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, pages 214–216. ACM Press, 2004.

[2] Aho, Alfred, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison Wesley Publishing Co., 1988.

[3] Alexe, A., L. Barthe, V. Gaildrat, and M.P. Cani, A Sketch-Based Modeling system using Convolution Surfaces, Proceedings of *Pacific Graphics 2005*, 2005.

[4] Alvarado, Christine J. *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design.* Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2000.

[5] Alvarado, Christine. Sketch recognition user interfaces: Guidelines for design and development. In *Making Pen-Based Interaction Intelligent and Natural*, pages 8–14, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[6] Alvarado, Christine and Randall Davis. Resolving ambiguities to create a natural sketch based interface. In *Proceedings. of IJCAI-2001*, August 2001.

[7] Alvarado, Christine and Randall Davis. Sketchread: A multi-domain sketch recognition engine. In *Proceedings of UIST 2004*, pages 23–32, New York, New York, October 24-27 2004. ACM Press.

[8] Alvarado, Christine and Randall Davis. Dynamically constructed bayes nets for multi-domain sketch understanding. In *Proceedings of IJCAI-05*, pages 1407–1412, San Francisco, California, August 1 2005.

[9] Alvarado, Christine and Lazzareschi, Michael. Properties of Real World Digital Logic Diagrams, *1st International Workshop on Pen-based Learning Technologies*, 2007.

[10] Anderson, Robert H. *Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics*. PhD Dissertation, Department of Applied Mathematics, Harvard University, 1968.

[11] Bahlmann, Claus, Bernard Haasdonk, and Hans Burkhardt. On-Line Handwriting Recognition with Support Vector Machines—A Kernel Approach. In *Proceedings of the Eighth International Workshop on Frontiers in Handwriting Recognition*, 49-54, 2002.

[12] Barzel, Ronen, John F. Hughes, and Daniel N. Wood. Plausible Motion Simulation for Computer Graphics Animation. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation'96*, Springer Verlag, 183-197, 1996.

[13] Belaid, Abdelwaheb and Jean-Paul Haton. A Syntactic Approach for Handwritten Formula Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(1):105-111, January 1984.

[14] Blostein, Dorothea and Ann Grbavec. Recognition of Mathematical Notation. In *Handbook on Optical Character Recognition and Document Image Analysis*, eds. P.S.P. Wang and H. Bunke, World Scientific Press, 557-582, 1997.

[15] Bourguignon, David, Marie-Paule Cani, and George Drettakis. Drawing for illustration and annotation in 3D. *Computer Graphics Forum*, 20(3):114–122, 2001.

[16] Cates, Sonya and Randall Davis. New approach to early sketch processing. In *Making Pen-Based Interaction Intelligent and Natural*, pages 29–34, Menlo Park, California, October 21-24 2004. AAAI Press.

[17] Chan, Kam-Fai and Dit-Yan Yeung. PenCalc: A Novel Application of On-Line Mathematical Expression Recognition Technology. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, 774-778, September 2001.

[18] Chan, Kam-Fai and Dit-Yan Yeung. Error Detection, Error Correction, and Performance Evaluation in On-Line Mathematical Expression Recognition. *Pattern Recognition*, 34(8):1671-1684, August 2001.

[19] Chan, Kam-Fai and Dit-Yan Yeung. An Efficient Syntactic Approach to Strctural Analysis of On-Line Handwritten Mathematical Expressions. *Pattern Recognition*, 33(3):375-384, March 2000.

[20] Chan, Kam-Fai and Dit-Yan Yeung. Mathematical Expression Recogition: A Survey. *Interational Journal on Document Analysis and Recognition*, 3(1):3-15, 2000.

[21] Chan, Kam-Fai and Dit-Yan Yeung. Elastic Structural Matching for On-Line Handwritten Alphanumeric Character Recognition. In *Proceedings of the Fourteenth International Conference on Pattern Recognition*, 1508-1511, 1998.

[22] Chan, Kam-Fai and Dit-Yan Yeung. A Efficient Syntactic Approach to Structural Analysis of On-Line Handwritten Mathematical Expressions. Technical Report HKUST-CS98-10, Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong, China, August 1998.

[23] Chang, Shi-Kuo. A Method for the Structural Analysis of Two-Dimensional Mathematical Expressions. *Information Sciences*, 2(3):253-272, 1970.

[24] Chou, P. Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar. In *Proceedings of SPIE Visual Communications and Image Processing IV*, 852-863, 1989.

[25] Connell, Scott D. and Anil K. Jain. Template-Based On-Line Character Recognition. *Pattern Recognition*, 34(1):1-14, January 2000.

[26] Cowans, Philip J. and Martin Szummer. A Graphical Model for Simultaneous Partitioning and Labeling, *Tenth International Workshop on Artificial Intelligence and Statistics*, 2005.

[27] Damm, Christian H., Klaus M. Hansen, and Michael Thomsen. Tool Support for Cooperative Object-Oriented Design: Gesture-Based Modeling on an Electronic Whiteboard. In *Proceedings of the 2000 SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 518-525, 2000.

[28] Davis, James, Maneesh Agrawala, Erika Chuang, Zoran Popovic, and David Salesin. A Sketching Interface for Articulated Figure Animation. In *Proceedings of the Eurographic/SIGGRAPH Symposium on Computer Animation*, 320-328, 2003.

[29] Davis, Richard C. and James A. Landay. Informal Animation Sketching: Requirements and Design. In *Proceedings of AAAI 2004 Fall Symposium on Making Pen-Based Interaction Intelligent and Natural*, Washington, D.C., October 21-24, 2004.

[30] Dimitriadis, Yannis A. and Juan López Coronado. Towards An Art-Based Mathematical Editor That Uses On-Line Handwritten Symbol Recognition. *Pattern Recognition*, 28(6):807-822, 1995.

[31] Day, A. M., J. R. Parks, and P. J. Pobgee. On-Line Written Input to Computers. *Machine Perception of Pictures and Patterns*, 233-240, 1972.

[32] Donahey, Alvin V. Character Recognition System and Method. United States Patent 3,996,557, 1976.

[33] Duda, Richard O., Peter E. Hart, and David G. Stork. *Pattern Classification*, John Wiley and Sons, 2001.

[34] Eisenstein, Jacob and Randall Davis. Natural gesture in descriptive monologues. In *Supplementary Proceedings of the ACM Symposium on User Interface Software and Techology (UIST'03)*, pages 69–70, New York, New York, November 2-5 2003. ACM Press.

[35] Eisenstein, Jacob and Randall Davis. Visual and linguistic information in gesture classification. In *International Conference on Multimodal Interfaces (ICMI'04)*, pages 113–120, New York, New York, October 14-15 2004. ACM Press.

[36] Fateman, Richard J., Taku Tokuyasu, Benjamin P. Berman, and Nicholas Mitchell. Optical Character Recognition and Parsing of Typeset Mathematics. *Journal of Visual Communication and Image Representation*, 7(1):2-15, 1996.

[37] Faure, Claudie and Zi Xiong Wang. Automatic Perception of the Structure of Handwritten Mathematical Expressions. In *Computer Processing of Handwriting*, eds. R. Plamondon, C. G. Leedham, World Scientific Publishing Company, 337-361, 1990.

[38] Feiner, Steven, David Salesin, and Thomas Banchoff. Dial: A Diagrammatic Animation Language. *IEEE Computer Graphics and Applications*, 2(7):43-54, 1982.

[39] Fukuda, Ryoji, Sou I, Fumikazu Tamari, Xie Ming, and Masakazu Suzuki. A Technique of Mathematical Expression Structure Analysis for the Handwriting Input System. In *Proceedings of the Fifth International Conference on Document Analysis and Recognition*, IEEE Press, 131-134, 1999.

[40] Forsberg, Andrew, Mark Dieterich, and Robert Zeleznik. The Music Notepad. *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 203-210, 1998.

[41] Freund, Yoav, and Robert E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119-139, August 1997.

[42] Gennari, Leslie, Levent Burak Kara, Thomas F. Stahovich, and Kenji Shimada. Combining geometry and domain knowledge to interpret hand-drawn diagrams. *Computers and Graphics* 29 (4) 2005.

[43] Giraud-Carrier, Christophe. A Note on the Utility of Incremental Learning. *AI Communications*, 13(4):215–223, December 2000.

[44] Grbavec, Ann, and Dorothea Blostein. Mathematics Recognition Using Graph Rewriting. In *Third International Conference on Document Analysis and Recognition*, 417-421, 1995.

[45] Groner, G. F. Real-Time Recognition of Handprinted Symbols. *Pattern Recognition*, ed. L.N. Kanal, 103-108, 1968.

[46] Gross, Mark D., and Ellen Yi-Luen Do. Ambiguous Intentions: A Paper-Like Interface for Creative Design. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 183-192, 1996.

[47] Gross, Mark D. Sketch-a-Sketch: A Dynamic Diagrammer. In *Proceedings of the IEEE Symposium on Visual Languages*, IEEE Press, 232-238, 1994.

[48] Gross, Mark. Advanced Visual Interfaces in "Recognizing and Interpreting Diagrams in Design" Gross, M.D. In T. Catarci. M. Costabile, S. Levialdi, G. Santucci eds., *Advanced Visual Interfaces '94 (AVI '94)*, ACM Press, 1994.

[49] Guerfali, Wacef and Réjean Plamondon. Normalizing and Restoring On-Line Handwriting. *Pattern Recognition*, 26(3):419-431, March 1993.

[50] Guimbretière, François and Terry Winograd. FlowMenu: Combining Command, Text, and Data Entry. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2000)*, ACM Press, 213-216, 2000.

[51] Ha, J., R. Haralick, and I. Philips. Recursive X-Y Cut Using Bounding Boxes of Connected Components. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, 952-955, 1995.

[52] Hammond, Tracy and Randall Davis. Tahuti: A geometrical sketch recognition system for uml class diagrams. *AAAI Spring Symposium on Sketch Understanding*, pages 59–68, March 25-27 2002.

[53] Hammond, Tracy and Randall Davis. LADDER: A language to describe drawing, display, and editing in sketch recognition. *Proceedings of the 2003 Internaltional Joint Conference on Artificial Intelligence (IJCAI)*, pages 461–467, 2003.

[54] Hammond, Tracy and Randall Davis. LADDER, a sketching language for user interface developers. Elsevier, *Computers and Graphics* 28, pp.518-532, 2005.

[55] Hammond, Tracy and Randall Davis. Interactive Learning of Structural Shape Descriptions from Automatically Generated Near-miss Examples. In *Intelligent User Interfaces (IUI)*, pp.37-40. 2006.

[56] Hammond, Tracy and Randall Davis. Automatically Transforming Symbolic Shape Descriptions for Use in Sketch Recognition. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pp.450-456, 2004.

[57] Hanaki, Shin-Ichi and Takemi Yamazaki. On-Line Recognition of Handprinted Kanji Characters. *Pattern Recognition*, Vol. 12, 421-429, 1980.

[58] Hansen, Charles and Christopher Johnson (eds.). *The Visualization HandBook*, Elsevier Academic Press, 2005.

[59] Herot, C. Graphical Input Through Machine Recognition of Sketches, In *Proceedings of SIGGRAPH76*, 97-102, 1976.

[60] Hinckley, Ken, Patrick Baudish, Gonzalo Ramos, and François Guimbretière. Design and Analysis of Delimiters for Selection-Action Pen Gesture Phrases in *Scriboli.* In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2005)*, ACM Press, 2005.

[61] Hoffman, Donald D., *Visual Intelligence: How We Create What We See*, W. W. Norton & Company, Inc. 1998.

[62] Hong, J. and J. Landay, SATIN: A Toolkit for Informal Ink-based Applications, ACM Symposium on User Interface Software and Technology, *CHI Letters*, 2(2):63-72, 2000.

[63] Hse, Heloise and A. Richard Newton, Recognition and Beautification of Multi-Stroke Symbols in Digital Ink. *Computers & Graphics.* 2005.

[64] Hull, Jesse F. *Recognition of Mathematics Using a Trainable Context-Free Grammar.* Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1996.

[65] Igarashi, Takeo, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley, 409-416, 1999.

[66] Igarashi, Takeo, and John F. Hughes. A Suggestive Interface for 3D Drawing. In *Proceedings of the ACM Symposium On User Interface Software and Technology (UIST 2001)*, ACM Press, 173-181, 2001.

[67] Igarashi, T., T. Moscovich, and J. F. Hughes. Spatial Keyframing for Performance-Driven Animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 107–115, New York, NY, USA, 2005. ACM Press.

[68] Igarashi, Takeo and John F. Hughes. A Suggestive Interface for 3D Drawing. In *ACM Symposium on User Interface Software and Technology*, 2001.

[69] Igarashi, Takeo and John F. Hughes. Clothing Manipulation. *ACM Trans. Graph.*, 22(3):697–697, 2003.

[70] Igarashi, Takeo and John F. Hughes. Smooth Meshes for Sketch-Based Freeform Modeling. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 139–142, New York, NY, USA, 2003. ACM Press.

[71] Igarashi, Takeo, Rieko Kadobayashi, Kenji Mase, and Hidehiko Tanaka. Path Drawing for 3D Walkthrough. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 173–174, New York, NY, USA, 1998. ACM Press.

[72] Igarashi, Takeo, Sachiko Kawachiya, Hidehiko Tanaka, and Satoshi Matsuoka. Pegasus: A Drawing System for Rapid Geometric Design. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pages 24–25, New York, NY, USA, 1998. ACM Press.

[73] Igarashi, Takeo, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive Beautification: a Technique for Rapid Geometric Design. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 105–114, New York, NY, USA, 1997. ACM Press.

[74] Ijiri, Takashi, Shigeru Owada, Makoto Okabe, and Takeo Igarashi. Floral Diagrams and Inflorescences: Interactive Flower Modeling Using Botanical Structural Constraints. *ACM Trans. Graph.*, 24(3):720–726, 2005.

[75] Impedovo, S., B. Marangelli, and A. M. Fanelli. A Fourier Descriptor Set for Recognizing Nonstylized Numerals. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8(8):640-645, August 1978.

[76] Karpenko, O. and J. Hughes. Inferring 3d Free-Form Shapes From Contour Drawings. In *Siggraph 2005 Sketches Program*, 2005.

[77] Karpenko, O., J. Hughes, and R. Raskar. Free-Form Sketching with Variational Implicit Surfaces. In *Eurographics Computer Graphics Forum*, volume 21/3, pages 585–594, 2002.

[78] Karpenko, Olga, and John F. Hughes. SmoothSketch: 3D free-form shapes from complex sketches, Proceedings of *SIGGRAPH 2006*, 589-598, 2006.

[79] Kara, Levent Burak, Leslie Gennari, and Thomas F. Stahovich. A Sketch-Based Interface for the Design and Analysis of Simple Vibratory Mechanical Systems. In *Proceedings of ASME International Design Engineering Technical Conferences*, 2004.

[80] Kara, L. B. and T. F. Stahovich. Sim-U-Sketch: A Sketch-Based Interface for Simulink, In *Proceedings of Advanced Visual Interfaces*, 354-357, 2004.

[81] Kara, L. B. and T. F. Stahovich. Hierarchical Parsing and Recognition of Hand-Sketched Diagrams. *17th ACM User Interface Software Technology (UIST)* 2004.

[82] Kara, L. B. and T. F. Stahovich (2005) An Image-Based, Trainable Symbol Recognizer for Hand-drawn Sketches. *Computers & Graphics* 29(4): 501-517 2005.

[83] Kerrick, David D. and Alan C. Bovik. Microprocessor-Based Recognition of Hand-printed Characters From a Tablet Input. *Pattern Recognition*, 21(5):525-537, May 1988.

[84] Kho, Youngihn and Michael Garland. Sketching Mesh Deformations. In *Symposium on Interactive 3D Graphics and Games 2005*, 2005.

[85] Kincaid, David and Ward Cheney. *Numerical Analysis* Second Edition. Brooks/Cole Publishing Company, 1996.

[86] Koschinski M., H.-J. Winkler, and M. Lang. Segmentation and Recognition of Symbols Within Handwritten Mathematical Expressions. In *1995 International Conference on Acoustics, Speech, Signal Processing*, 2439-2442, 1995.

[87] Kosmala, Andreas and Gerhard Rigoll. On-Line Handwritten Formula Recognition Using Statistical Methods. In *Proceedings of the International Conference on Pattern Recognition*, 1306-1308, 1998.

[88] Kurtenbach, Gordon and William Buxton. User Learning and Performance with Marking Menus. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, ACM Press, 258-264, 1994.

[89] LaFollette, Paul, James Korsh, and Raghvinder Sangwan. A Visual Interface for Effortless Animation of C/C++ Programs. *Journal of Visual Languages and Computing*, 11(1):27-48, 2000.

[90] Laleuf, Jean R., and Anne Morgan Spalter. A Component Repository for Learning Objects: A Progress Report. In *Proceedings of the First ACM/IEEE-CS Joint Conference on Digital Libraries*, ACM Press, 33-40, 2001.

[91] Landay, James A., and Brad A. Myers. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of the 1995 SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 43-50, 1995.

[92] LaViola, J. An Initial Evaluation of MathPad^2: A Tool for Creating Dynamic Mathematical Illustrations, To appear *Computers and Graphics*, 2007.

[93] LaViola, J., and Zeleznik, R. A Practical Approach to Writer-Dependent Symbol Recognition Using a Writer-Independent Recognizer, To appear *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007.

[94] LaViola, J. Advances in Mathematical Sketching: Moving Toward the Paradigms Full Potential, *IEEE Computer Graphics and Applications*, 27(1):38-48, January/February 2007.

[95] LaViola, J. An Initial Evaluation of a Pen-Based Tool for Creating Dynamic Mathematical Illustrations, Proceedings of the *Eurographics Workshop on Sketch-Based Interfaces and Modeling 2006*, 157-164, September 2006.

[96] LaViola, J. *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*, Ph.D. Dissertation, Brown University, Department of Computer Science, May 2005.

[97] LaViola, Joseph and Robert Zeleznik. MathPad^2: A System for the Creation and Exploration of Mathematical Sketches. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, 23(3):432-440, August 2004.

[98] Lavirotte, Stéphane and Löic Pottier. Optical Formula Recognition. In *Fourth International Conference on Document Analysis and Recognition*, IEEE Press, 357-361, 1997.

[99] Lee, WeeSan, Kara, L. B., and T. F. Stahovich (2007) An Efficient Graph-Based Symbol Recognizer. Accepted to *Computers & Graphics Special Issue on Sketch-Based Interfaces and Modeling*, 2007.

[100] Lee, Hsi-Jian, and Jiumn-Shine Wang. Design of a Mathematical Expression Recognition System. *Pattern Recognition Letters*, 18:289-298, 1997.

[101] Lee, Hsi-Jian and Jiumn-Shine Wang. Design of a Mathematical Expression Recognition System. In *Third International Conference on Document Analysis and Recognition*, IEEE Press, 1084-1087, 1995.

[102] Lee, Hsi-Jian, and Min-Chou Lee. Understanding Mathematical Expressions Using Procedure-Oriented Transformation. *Pattern Recognition*, 27(3):447-457, 1994.

[103] Lehmberg, Stefan, Hans-Jürgen Winkler, and Manfred Lang. A Soft-Design Approach for Symbol Segmentation Within Handwritten Mahthematical Expressions. In *1996 International Conference on Acoustics, Speech, and Signal Processing*, 3434-3437, 1996.

[104] Li, Xiaolin, and Dit-Yan Yeung. On-Line Handwritten Alphanumeric Character Recognition Using Dominant Points in Strokes. *Pattern Recognition*, 30(1):31-44, January 1997.

[105] Lin, James, Mark W. Newman, Jason I. Hong, and James A. Landay. DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. *CHI Letters*, 2(1):510-517, ACM Press, 2000.

[106] Lipson, H. and M. Shpitalni. Correlation-based reconstruction of a 3D object from a single freehand sketch, *2002 AAAI Spring Symposium on Sketch Understanding*, pp. 99-104, AAAI Press, Melno Park, CA, 2002.

[107] Lipson, H. and M. Shpitalni. Conceptual Design and Analysis by Sketching. In *AIDAM-97*, 1997.

[108] Lipson, H. and M. Shpitalni. Optimization-Based Reconstruction of a 3D Object From a Single Freehand Line Drawing, *Journal of Computer Aided Design*, Vol. 28 No. 8, pp. 651-663, 1996.

[109] Littin, Richard H. *Mathematical Expression Recognition: Parsing Pen/Tablet Input in Real-Time Using LR Techniques.* Master's Thesis, University of Waikato, Hamilton, New Zealand, 1995.

[110] Mankoff, Jennifer, Scott E. Hudson and Gregory D. Abowd, Interaction techniques for ambiguity resolution in recognition-based interfaces," In *Proceedings of UIST 2000*, 11-20, 2000.

[111] Martin, William J. A Fast Parsing Scheme for Hand-Printed Mathematical Expressions. Artificial Intelligence Memo No. 145, Massachusetts Institute of Technology, 1967.

[112] Marzinkewitsch, Reiner. Operating Computer Algebra Systems by Handprinted Input. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, 411-413, 1991.

[113] Masry M., Kang D., Lipson H. A Pen-Based Freehand Sketching Interface for Progressive Construction of 3D Objects, *Journal of Computers and Graphics*, Volume 29, pp. 563-575, 2005.

[114] Masry M. and H. Lipson. A Sketch-BasedInterface for Iterative Design and Analysis of 3D Objects, *Proceedings of Eurographics workshop on Sketch-Based Interfaces*, Dublin, Ireland, pp. 109-118, Aug. 2005.

[115] Matsakis, Nicholas E. *Recognition of Handwritten Mathematical Expressions.* Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[116] McMillan, Leonard, Osama Tolba, and Julie Dorsey. Sketching with Projective 2D Strokes. In *ACM Symposium on User Interface Software and Technology*, pages 149–157, 1999.

[117] Miller, Erik G., and Paul A. Viola. Ambiguity and Constraint in Mathematical Expression Recognition. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 784-791, 1998.

[118] Moran, Thomas P., Patrick Chui, and William van Melle. Pen-based Interaction Techniques for Organizing Material on an Electronic Whiteboard. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 45-54, 1997.

[119] Moscovich, Tomer, and John F. Hughes. Animation Sketching: An Approach to Accessible Animation. Technical Report CS-04-03, Computer Science Department, Brown University, Providence, RI, February 2004.

[120] Mynatt, Elizabeth D., Takeo Igarashi, W. Keith Edwards, and Anthony LaMarca. Flatland: New Dimensions in Office Whiteboards. In *Proceedings of the 1999 SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 346-353, 1999.

[121] Nakayama, Y. A Prototype Pen-Input Mathematical Formula Editor. In *Proceedings of ED-MEDIA 93 - World Conference on Educational Multimedia and Hypermedia*, 400-407, 1993.

[122] Nealen, A., T. Igarashi, O.Sorkine and M. Alexa, FiberMesh: Designing Freeform Surfaces with 3D Curves, To appear, *SIGGRAPH 2007*, 2007.

[123] Nealen, Andruw, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. A Sketch-Based Interface for Detail-Preserving Mesh Editing. *ACM SIGGRAPH Transactions on graphics*, 24(3), 2005.

[124] Odaka, Kazumi, Hiroki Arakawa, and Isao Masuda. On-Line Recognition of Handwritten Characters by Approximating Each Stroke with Several Points. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-12(6):898-903, November/December 1982.

[125] Okamoto, M. and B. Miao. Recognition of Mathematical Expressions by Using Layout Structures of Symbols. In *Proceedings of the First International Conference on Documenent Analysis and Recognition*, 242-250, 1991.

[126] Oltmans, Michael, and Randall Davis. Naturally Conveyed Explanations of Device Behavior. In *Workshop on Perceptive User Interfaces*, 2001.

[127] Owada, S., F. Nielsen, K. Nakazawa, and T. Igarashi, A Sketching Interface for Modeling the Internal Structures of 3D Shapes, Proceedings of *Smart Graphics 2003*, 2003.

[128] Pavlidis, Ioannis, Rahul Singh, and Nikolaos P. Papanikolopoulos. On-Line Handwriting Recognition Using Physics-Based Shape Metamorphosis. *Pattern Recognition*, 31(11):1589-1600, November 1998.

[129] Pedersen, E., K. McCall, T. Moran, and F. Halasz, Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings, In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, 391-389, 1993.

[130] Pereira, Joao P., Vasco A. Branco, Joaquim A. Jorge, Nelson F. Silva, Tiago D. Cardoso, and F. Nunes Ferreira. Cascading Recognizers for Ambiguous Calligraphic Interaction. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2004.

[131] Pickering, Jeff, Dom Bhuphaibool, Joseph LaViola, and Nancy Pollard. The Coach's Playbook. Technical Report CS-99-08, Brown University, Department of Computer Science, Providence, RI, May 1999.

[132] Plamondon, Réjean and Sargur N. Srihari. On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):63-84, January 2000.

[133] Plamondon, R. and F. J. Maarse. An Evaluation of Motor Models of Handwriting, *IEEE Transactions on Systems, Man, and Cybernetics*,19(5):1060-1072, 1989.

[134] Powers, V. M. Pen Direction Sequences in Character Recognition, *Pattern Recognition* Vol. 5, 291-302, March 1973.

[135] Rubine, Dean. Specifying Gestures by Example. In *Proceedings of SIGGRAPH'91*, ACM Press, 329-337, 1991.

[136] Saund, Eric, David Fleet, Daniel Larner, and James Mahoney. Perceptually-Supported Image Editing of Text and Graphics. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2003)*, ACM Press, 183-192, 2003.

[137] Saund, Eric, and Edward Lank. Stylus Input and Editing Without Prior Selection of Mode. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2003)*, ACM Press, 213-217, 2003.

[138] Saund, E., J. Mahoney, D. Fleet, D. Larner, E. Lank. Perceptual Organization as a Foundation for Intelligent Sketch Editing. *AAAI Symposium on Sketch Understanding*, 2002.

[139] Scattolin, P. and A. Krzyzak. Weighted Elastic Matching Method for Recognition of Handwritten Numerals, In *Proceedings of Vision Interface'94*, 178-185, 1994.

[140] Schapire, Robert. A Brief Introduction to Boosting, In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1401-1406, 1999.

[141] Schmidt, R., Wyvill, B., Sousa, M.C., Jorge, J.A. ShapeShop: Sketch-Based Solid Modeling with BlobTrees, *Eurographics Workshop on Sketch-Based Interfaces and Modeling 2005*, 2005.

[142] Sezgin, Tevfik Metin and Randall Davis. Handling Overtraced Strokes in Hand-Drawn Sketches. In *Making Pen-Based Interaction Intelligent and Natural*, pages 141–144, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[143] Sezgin, Tevfik Metin and Randall Davis. Scale-Space Based Feature Point Detection for Digital Ink. In *Making Pen-Based Interaction Intelligent and Natural*, pages 145–151, Menlo Park, California, October 21-24 2004. AAAI Fall Symposium.

[144] Sezgin, Tevfik Metin and Randall Davis. Hmm-Based Efficient Sketch Recognition. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI'05)*, pages 281–283, New York, New York, January 9-12 2005. ACM Press.

[145] Sezgin, Tevfik Metin, Thomas Stahovich, and Randall Davis. Sketch Based Interfaces: Early Processing for Sketch Understanding. *Workshop on Perceptive User Interfaces, Orlando FL*, 2001.

[146] Sezgin, Tevfik Metin and Randall Davis. Sketch Interpretation Using Multiscale Models of Temporal Patterns. *IEEE Computer Graphics and Applications*, 27(1), pp.28-37, January 2007.

[147] Shesh, Amit and Baoquan Chen. Smartpaper–An Interactive and User-Friendly Sketching System. In *Eurographics 2004*, 2004.

[148] Schneider, Philip J., and David H. Eberly. *Geometric Tools for Computer Graphics*, Morgan Kaufmann Publishers, 2003.

[149] Shilman, Michael and Paul Viola. Spatial Recognition and Grouping of Text and Graphics *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)*, 2004.

[150] Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Third Edition. Addison Wesley Publishing Company, 1998.

[151] Smithies, Steve, Kevin Novins, and James Arvo. A Handwriting-Based Equation Editor. In *Proceedings of Graphics Interface'99*, 84-91, 1999.

[152] Smithies, Steven R. *Freehand Formula Entry System.* Master's Thesis, Department of Computer Science, University of Otago, Dunedin, New Zealand, 1999.

[153] Stahovich, Thomas F. Segmentation of Pen Strokes Using Pen Speed. *AAAI Symposium on Making Pen-Based Interfaces Intelligent and Natural*, 2004.

[154] Stasko, John T. Using Direct Manipulation to Build Algorithm Animations by Demonstration. In *Proceedings of the ACM Conference on Human Factors and Computing Systems (CHI'91)*, ACM Press, 307-314, 1991.

[155] Stasko, John T. Animating Algorithms with XTANGO. *SIGACT News*, 23(2):67-71, 1992.

[156] Sutherland, I. SketchPad: A Man-Machine Graphical Communication System, In *Proceedings of AFIPS Spring Joint Computer Conference*, 329-346, 1963.

[157] Szummer, Martin and Yuan Qi. Contextual Recognition of Hand-drawn Diagrams with Conditional Random Fields *9th Intl. Workshop on Frontiers in Handwriting Recognition (IWFHR)* 32-37, 2004.

[158] Tappert, Charles C., Ching Y. Seun, and Toru Wakahara. The State of the Art in On-Line Handwriting Recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787-808, August 1990.

[159] Thorne, Matthew, David Burke, and Michiel van de Panne. Motion Doodles: An Interface for Sketching Character Motion. *ACM Trans. Graph.*, 23(3):424–431, 2004.

[160] Tolba, Osama, Julie Dorsey, and Leonard McMillan. A Projective Drawing System. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 25–34, March 2001.

[161] Turquin, Emmanuel, Marie-Paule Cani, and John Hughes. Sketching Garments for Virtual Characters. In John F. Hughes and Joaquim A. Jorge, editors, *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM)*. Eurographics, august 2004.

[162] Twaakyodo, Hashim M., and Masayuki Okamoto. Structural Analysis and Recognition of Mathematical Expressions. In *Third International Conference on Document Analysis and Recognition*, IEEE Press, 430-437, 1995.

[163] Veselova, Olya and Randall Davis. Perceptually Based Learning of Shape Descriptions. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 482–487, 2004.

[164] Vuokko, Vuori, Jorma Laaksonen, Erkki Oja, and Jari Kangas. On-Line Adaptation in Recognition of Handwritten Alphanumeric Characters. In *Proceedings of the Fifth International Conference on Document Analysis and Recognition*, 792-795, 1999.

[165] Wang, Xin, Michael Shilman, and Sashi Raghupathy. Parsing Ink Annotations on Heterogeneous Documents. *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)*, 2006.

[166] Warner, Simeon, Simon Catterall, and Edward Lipson. Java Simulations for Physics Education. *Concurrency: Practice and Experience*, 9(6):447-484, June 1997.

[167] Wehbi, H., H. Oulhadj, J. Lemoine, and E. Petit. Numerical Characters and Capital Letters Segmentation Recognition in Mixed Handwriting Context. In *Third International Conference on Document Analysis and Recognition*, IEEE Press, 878-881, 1995.

[168] Weinzapfel, G. and N. Negroponte. Architeture-By-Yourself: An Experiment with Computer Graphics for House Design, In *Proceedings of SIGGRAPH76*, 74-78, 1976.

[169] Winkler, Hans-Jürgen. Symbol Recognition in Handwritten Mathematical Formulas. In *International Workshop on Modern Modes of Man-Machine Communication*, 7/1-7/10, June 1994.

[170] Xuejun, Zhao, Lin Xinyu, Zheng Shengling, Pan Baochang, and Yuan Y. Tang. On-Line Recognition Handwritten Mathematical Symbols. In *Fourth International Conference on Document Analysis and Recognition*, IEEE Press, 645-648, 1997.

[171] Yang C., D. Sharon, and Mi. van de Panne. Sketch-based Modeling of Parameterized Objects. In *2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 63-72, 2005.

[172] Zanibbi, Richard, Dorothea Blostein, and James Cordy. Recognizing Mathematical Expressions Using Tree Transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(11):1-13, November 2002.

[173] Zanibbi, Richard, Dorthea Blostein, and James R. Cordy. Baseline Structure Analysis of Handwritten Mathematics Notation. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, 768-773, 2001.

[174] Zanibbi, Richard, Kevin Novins, James Arvo, and Katherine Zanibbi. Aiding Manipulation of Handwritten Mathematical Expressions Through Style-Preserving Morphs. In *Graphics Interface 2001*, 127-134, 2001.

[175] Zeleznik, Robert, Timothy Miller, Loring Holden, and Joseph LaViola. Fluid Inking: Using Punctuation to Allow Modeless Combination of Marking and Gesturing. Technical Report CS-04-11, Department of Computer Science, Brown University, Providence, RI, July 2004.

[176] Zeleznik, Robert C., Kenneth P. Henrdon, and John F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, 163-170, 1996.

[177] Zhao, Y., T. Sakuri, H. Sugiura, and T. Torii. A Methodology of Parsing Mathematical Notation for Mathematical Computation. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, 292-300, 1996.