Hybrid Sketching:
A New Middle Ground Between 2- and 3-D

by

John Alex

Submitted to the Department of Architecture
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in the Field of Computer Graphics and Architectural Design

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author .....
Department of Architecture
October 15, 2004

Certified by .....
Julie Dorsey
Professor of Computer Science, Yale University
Thesis Supervisor

Certified by .....
William L. Porter
Norman B. and Muriel Leventhal Professor of Architecture and Planning
Thesis Supervisor

Accepted by .....
Stanford Anderson
Chairman, Architecture Department Committee of Graduate Students

Hybrid Sketching:

A New Middle Ground Between 2- and 3-D

by

John Alex

Submitted to the Department of Architecture
on October 15, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in the Field of Computer Graphics and Architectural Design

Abstract

This thesis investigates the geometric representation of ideas during the early stages of design. When a designer's ideas are still in gestation, the *exploration* of form is more important than its precise *specification*. Digital modelers facilitate such exploration, but only for forms built with discrete collections of high-level geometric primitives; we introduce techniques that operate on designers' medium of choice, 2-D sketches. Designers' explorations also shift between 2-D and 3-D, yet 3-D form must also be specified with these high-level primitives, requiring an entirely different mindset from 2-D sketching. We introduce a new approach to transform existing 2-D sketches directly into a new kind of sketch-like 3-D model. Finally, we present a novel sketching technique that removes the distinction between 2-D and 3-D altogether.

This thesis makes five contributions: point-dragging and curve-drawing techniques for editing sketches; two techniques to help designers bring 2-D sketches to 3-D; and a sketching interface that dissolves the boundaries between 2-D and 3-D representation. The first two contributions of this thesis introduce smooth exploration techniques that work on sketched form composed of strokes, in 2-D or 3-D. First, we present a technique, inspired by classical painting practices, whereby the designer can explore a range of curves with a single stroke. As the user draws near an existing curve, our technique automatically and interactively replaces sections of the old curve with the new one. Second, we present a method to enable smooth exploration of sketched form by point-dragging. The user constructs a high-level "proxy" description that can be used, somewhat like a skeleton, to deform a sketch independent of the internal stroke description.

Next, we leverage the proxy deformation capability to help the designer move directly from existing 2-D sketches to 3-D models. Our reconstruction techniques generate a novel kind of 3-D model which maintains the appearance and stroke structure of the original 2-D sketch. One technique transforms a single sketch with help from annotations by the designer; the other combines two sketches. Since these interfaces are user-guided, they can operate on ambiguous sketches, relying on the designer

to choose an interpretation.

Finally, we present an interface to build an even sparser, more suggestive, type of 3-D model, either from existing sketches or from scratch. "Camera planes" provide a complex 3-D scaffolding on which to hang sketches, which can still be drawn as rapidly and freely as before. A sparse set of 2-D sketches placed on planes provides a novel visualization of 3-D form, with enough information present to suggest 3-D shape, but enough *missing* that the designer can 'read into' the form, seeing multiple possibilities. This unspecified information - this empty space - can spur the designer on to new ideas.

Thesis Supervisor: Julie Dorsey
Title: Professor of Computer Science, Yale University

Thesis Supervisor: William L. Porter
Title: Norman B. and Muriel Leventhal Professor of Architecture and Planning

Author

John Alex

B.A., Computer Science

Brown University, 1999

Pixar, 1999-2000 (and during some breaks thereafter)

Toy Story 2, Monsters Inc., Finding Nemo

Committee

Julie Dorsey

Professor of Computer Science

Yale University

William L. Porter

Norman B. and Muriel Leventhal Professor of Architecture and Planning

MIT

Frédo Durand

Assistant Professor of Computer Science

MIT

John Fernandez

Assistant Professor of Architecture, Building Technology

MIT

Acknowledgments

I've relied on many, many people over the past four years. Thank you:

To my advisor, Professor Julie Dorsey, for giving me the luxury of studying architecture *and* computer graphics.

To Professor Bill Porter, who, over a weekly cappuccino and some mint Milanos, showed me how it's done - in architecture, in music, in life.

To Professor Frédo Durand, for repeatedly helping me when he didn't have to, not to mention providing all that wine, cheese, and outrageous Frenchness.

To Professor Seth Teller, who turned me on to research, helped me develop what is still the fastest raycaster around, made me feel at home in the Graphics Group, and watched out for me.

To Professor John Fernandez, for his consistently insightful critiques.

To Renée Caso and Britton Bradley, Holders of All The Answers.

To Schwartz/Silver Architects, for generously allowing me to scan many of their sketches.

To Yanni Loukissas and Axel Kilian, companions in the Architecture PhD program, for their infectious belief in the possibilities of computers in architectural design, and their subsequent willingness to be occasional guinea pigs.

To Tom Buehler, for making 18 videos for three paper submissions in one week.

To Barb Cutler, for stuffing 18 FedEx envelopes in one day.

To the MIT Graphics Group, for being more social than I could be.

To my friends, who made it fun and kept me sane:

To Kate Ruhl, who kept inviting me to do fun things.

To Marcel Utz, who taught me tensors after a bit of grappa.

To Chief Robert Sumner, for his social prowess and fantastic deformation transfer algorithm.

To Ken Giesecke, for his friendship and sense of humor.

To my family and surrogate family, who have been supporting me for a long, long time:

To Steve Boggs, who taught me trigonometry on a dinner napkin.

To Dolores Destefano, my oldest friend, for all those parties, whether in Brooklyn or, say, Damascus.

To Sascha Becker, for introducing me to computer graphics (*and* microwaveable slippers).

To Josh Carroll, for deleting his early recording of me saying 'I hate raytracing'; and, years later, informing me about pentimenti.

To Susan Morgan, who informed me, early and often, that everything would work out in the end; and was right, again.

To Norman Hildes-Heim, for his love of architecture and hatred of veneer.

To my sister Christina and my aunt Paula, for their love.

To my parents Paul and Linda for their love, their unwavering confidence in me, and clarity about what's really important.

Contents

1 Introduction **13**

1.1 Design Process 13

1.2 Affordances of Sketching and Digital Media 16

1.3 Philosophy 18

1.4 Overview 19

2 Related Work **21**

2.1 Geometry Creation: From Scratch 21

2.2 Geometry Manipulation 24

2.3 Transition from 2-D to 3-D 26

2.4 Sketch Rendering of 3-D Models 28

2.5 Image-Based Rendering 29

3 Pentimenti: Exploring Curves Through Overdraw **31**

3.1 Motivation 32

3.2 Related Work 32

3.2.1 Freehand Drawing 32

3.2.2 Editing by Point Dragging 34

3.2.3 Editing by Drawing Freehand Curves 35

3.3 Exposition 36

3.3.1 Visualization 37

3.3.2 Fine-grained, Immediate Subcurve Replacement 37

3.3.3 Extensions for More Curve Topologies, Topological Operations 42

3.3.4 Overall Algorithm . . . 47
3.4 Results . . . 47
3.5 Analysis . . . 48

4 Exploring Sketched Form by Dragging Control Points 54
4.1 Motivation . . . 56
4.2 Related Work . . . 56
4.2.1 Existing Sketch Practice . . . 56
4.2.2 Point-dragging . . . 57
4.2.3 Proxy Point-Dragging . . . 57
4.2.4 Proxy Construction . . . 58
4.3 Exposition . . . 59
4.3.1 Proxy Construction . . . 60
4.3.2 Proxy-based Deformation . . . 61
4.4 Results . . . 64
4.5 Analysis . . . 64
4.6 Appendix: Special Cases in Proxy Construction . . . 69
4.6.1 High-Valence Vertices . . . 69
4.6.2 Loops . . . 70
4.6.3 Endpoints . . . 70
4.6.4 Topological Gaps . . . 72

5 3-D Reconstruction from a Single Sketch 74
5.1 Related Work . . . 75
5.2 Exposition . . . 77
5.3 Results . . . 85
5.4 Analysis . . . 87
5.5 Appendix: Impossible Orthographic Axes . . . 89

6 3-D Reconstruction from Two Sketches 92
6.1 Differences from Reconstruction with a Single Sketch . . . 93

6.2 Related Work . . . 93
6.3 Exposition . . . 94
6.3.1 Incremental Reconstruction Using Topology . . . 94
6.3.2 Fast Specification of Correspondences Between Degenerate Features . . . 97
6.3.3 2-Curve Reconstruction . . . 98
6.4 Results . . . 102
6.5 Analysis . . . 103

7 Modeling with Camera Planes **106**
7.1 Motivation . . . 106
7.2 Related Work . . . 108
7.3 Exposition . . . 111
7.3.1 Slice . . . 112
7.3.2 Geometry . . . 113
7.3.3 Camera . . . 113
7.4 Results . . . 115
7.5 Analysis . . . 115

8 Conclusion **117**
8.1 Analysis . . . 117
8.2 Discussion . . . 120

A Prototype Details **122**

List of Figures

1-1 Stata Center artifacts 15
1-2 Decomposition of Meier house 17
1-3 Picture of Meier house 17
1-4 Overview 19

2-1 Gestural interface 23
2-2 B-Spline control points 25

3-1 Architecture sketches 33
3-2 A 19th century painting and its underdrawing 34
3-3 Baudel algorithm 35
3-4 Visualization of unified tolerance measure d 38
3-5 Mouse down handling 39
3-6 Mouse move handling 40
3-7 Traversal distance 41
3-8 Sharp, small corners 42
3-9 Editing a high-valence point 43
3-10 Integrating join mode 43
3-11 Forking diagram 44
3-12 Sewing up a fork 45
3-13 Forking and sewing 46
3-14 Mouse-down event handling 47
3-15 Mouse-move event handling 48
3-16 Logo sketch 49

3-17 Logo sketch — zoomed 49
3-18 3-D tent sketch 50
3-19 Duck sketch 50
3-20 Pants sketch 51

4-1 Comparison with direct manipulation 55
4-2 Proxy layers 57
4-3 Perceptual valence 60
4-4 Proxy construction 61
4-5 Proxy deformation 62
4-6 Curve deformation comparison 63
4-7 3-D manipulation 65
4-8 Aiding Point Selection 66
4-9 Attaching high-valence vertices to proxy edges 70
4-10 Handling degenerate proxy edges 70
4-11 Attaching endpoints to proxy edges 71
4-12 Bridge sets for topological gaps 72

5-1 3-D sketches as representation, in context 76
5-2 Full 1-view reconstruction 78
5-3 Bootstrapping 3-D axes 80
5-4 2-D axes that cannot be reconstructed in 3-D 81
5-5 Non-full rank constraints 84
5-6 Reconstructed sketch 1 85
5-7 Reconstructed sketch 2 85
5-8 Reconstructed sketch 3 86
5-9 Reconstructed sketch 4 86
5-10 Viewing right angles from various directions 90

6-1 Incremental two-view reconstruction workflow 96
6-2 Stereo triangulation 97

6-3 Degenerate features 98
6-4 Degeneracy-exploiting two-view reconstruction workflow 99
6-5 Two-curve reconstruction 100
6-6 Difficult two-curve case 101
6-7 Two-curve reconstruction — graph 101
6-8 Two-view result 102
6-9 Topology differences 103
6-10 Controlling 3-D with 2-D features 104

7-1 Billboard clouds 109
7-2 Representations 110
7-3 3-D visualization 111
7-4 Camera planes 112
7-5 3-D sketches on planes 114

A-1 Full screenshot of prototype 123

Chapter 1

Introduction

When designers' ideas are still in gestation, the exploration of form is more important than its precise specification. Digital modelers facilitate such exploration, but only for forms composed of discrete geometric primitives; we introduce techniques that operate on designers' medium of choice, 2-D sketches. Designers' explorations also shift between 2-D and 3-D, yet 3-D form must currently be assembled from discrete primitives, requiring an entirely different mindset from 2-D sketching. We introduce digital tools to transform existing 2-D sketches directly into a new kind of sketch-like 3-D model. Finally, we present a novel sketching technique which removes the distinction between 2-D and 3-D altogether.

1.1 Design Process

Since this thesis helps designers explore the space of forms, we first discuss why a designer would be searching this space, and current practice for doing so. A designer often begins with an abstract goal in mind - such as an emotion, evocative symbol, or appearance from a key view. In fact, one of the critical tasks in introductory design studios is to teach students not to create a initial form and develop it, but to create an idea about what the form will mean and explore different forms to express that idea. These "generative ideas" serve as a sort of litmus test throughout the design; that is, the designer continually questions whether a prospective form, however vaguely

imagined, successfully embodies these ideas. The ideas themselves undergo refinement as well, of course, as they are articulated. In fact, quite often a designer will come up with a form which fortuitously expresses something quite different than the concept she originally had in mind - quite different yet still desirable. Some forms created by the designer may thus not represent literal buildings, but simply be initial attempts at expressing these ideas in any way possible. In this thesis we do not attempt to model the designer's purpose or meaning behind a given form; we limit ourselves to aiding the designer in exploring form.

In practice, the dialogue between ideas and their execution in form is largely embodied by a series of physical and digital artifacts constructed along the way. This dialogue, internal to the architect or design team, is mostly hidden to the outside world. The artifacts of this design process, sketches and models, are generally thrown away, whether literally or in practice by being stored away with a sea of other sketches and models. Only a rare sketch or model is shown to the outside world, typically only for the express purpose of communicating some idea that survived through to the final design. This hiddenness might explain why modern tools do not, by and large, address designers' needs. To make this more concrete to non-architects, we show a few artifacts from an actual design process in Figure 1-1. In (a), an architect has sketched out a vague notion of a building's appearance. Notice how this is not nearly enough information to build a 3-D model from, but still a useful representation of how the building looks - and more importantly, how it feels. We might imagine the generative idea of this building being a feeling of a "free-for-all", buildings exploding and piling on top of each other. That idea is further explored in (b); here a 3-D model is used. Notice how the wooden block construction provides a much more helpful representation of the 3-D "massing" and layout of the building than the 2-D, while still remaining abstract - not at all the photorealistic 3-D we might expect. A more detailed sketch followed (c); later, more realistic and large-scale models (d) were developed, faithful to the original sketch but with more details worked out. These four artifacts are only a tiny sample of the huge number of sketches and models created by the designers of this building.

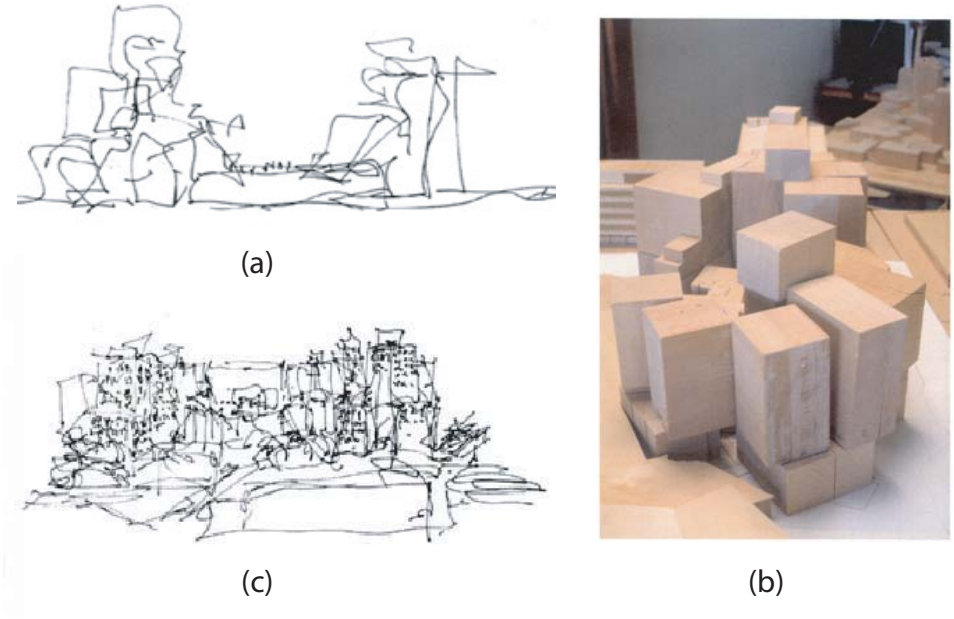

Figure 1-1: Artifacts from the design of the MIT Stata Center in Cambridge, MA by Frank O. Gehry and Associates. An initial design sketch (a) is followed by a massing model (b), which led to a more detailed sketch (c), and finally a large-scale model (d). This thesis brings the high-level editing and 3-D visualization capabilities of (b) and (d) to sketches (a and c), as well as introducing new representations in between 2-D and 3-D.

1.2 Affordances of Sketching and Digital Media

Each type of artifact constructed in the design process has its own affordances, leading to different patterns of use. Traditionally, 2-D sketching is used more for conceptual design - to *imagine* a form; while 3-D media such as small-scale physical or simple digital models are closer to the execution side - to *visualize* or otherwise sanity-check a given form, possibly making small changes. Sketches are used in the early phases of design, and CAD in the later phases. In this thesis we avoid this either/or situation, bringing the smooth high-level exploration and 3-D visualization tools of digital modelers directly to bear on 2-D sketches. Designers can more easily explore ideas when they have several alternatives in trading off precision, ease of construction, and suggestiveness.

Sketches are effective in specifying form for many reasons. The two most relevant to this discussion are their inherent simplicity and bottom-up nature. Unlike 3-D models, 2-D sketches are inherently limited in the amount of geometric information they can hold, quickly becoming too complex to understand and too detailed to edit. Designers are thus forced to distribute geometric information among many sketches. This distribution has many positive effects; for instance, it encourages the designer to segment the design problem, in effect dividing and conquering. Examples of how a designer might segment a building can be found in Clark and Pause[14], where 88 buildings are analyzed in several ways. We demonstrate one such decomposition in Figure 1-2; notice how simple each drawing is despite the complexity of these real-world buildings. The simplicity required in a sketch also makes designers much more careful about what is drawn; the level of specification tends toward the minimum necessary to describe the shape. This adaptive targeting of descriptive effort makes sketching very fast.

The second quality unique to sketching is its bottom-up use of primitives. That is, unordered sets of strokes are perceived to coalesce into larger shapes, or perhaps even several sets of possible shapes in an ambiguous drawing. A series of studies by Suwa et al. [72][73][74] has shown how designers exploit perceptual ambiguity, particularly

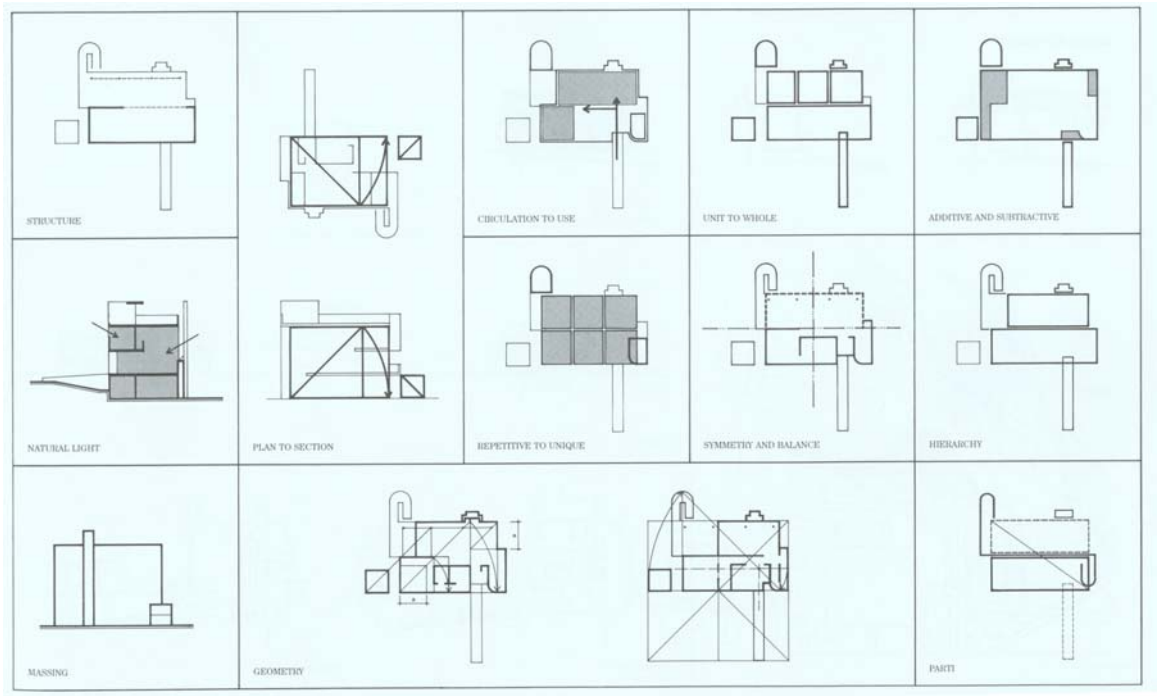

Figure 1-2: Several decompositions of a Richard Meier house (from Clark and Pause[14])

Figure 1-3: Picture of the Meier house from Figure 1-2.

seeing embedded shapes and subshapes, to generate new ideas, and how "fixating" on a particular interpretation stalls the development of ideas. The bottom-up accumulation of primitives also enables the targeting of descriptive effort mentioned previously: the designer can place low-level strokes where they are needed, building up detail in arbitrary ways that could be difficult to organize and express from the top down with higher-level primitives.

Although their use of a top-down approach makes digital tools less effective in modeling form (as will be described in Chapter 2), they provide two unique affordances not found in sketching. First, they enable smooth exploration of form. Most digital modeling packages (such as Maya [3] and Adobe Illustrator[1]) have a simple direct manipulation interface, where control points can be dragged around smoothly, with the results displayed continuously. With this interface, a large space of possible forms can be explored with a single mouse drag. Variations on this interface will be covered in Chapter 4. Secondly, digital tools provide 3-D visualization. With 2-D sketches, designers can only imagine what a collection of views, taken together, would look like in 3-D. With digital tools, 3-D models can be explored — they can easily be rotated, walked through, and zoomed into — more thoroughly than with physical 3-D models. Further, recent innovations in so-called "image-based rendering" have revealed that even a sparse set of 2-D images can be assembled onto a 3-D scaffold for 3-D visualization, where any artifacts due to missing information are considered acceptable due to the vagueness of the input. Both of these capabilities, smooth exploration and 3-D visualization, are extremely useful, but unfortunately cannot be currently deployed on representations made with sketching, the most powerful and flexible medium for creating forms.

1.3 Philosophy

Applying the affordances of digital tools to 2-D sketches carries a significant overhead, both in time and effort. Moving from sketches to smoothly editable 2-D digital geometry, or any kind of 3-D digital geometry, requires the user to re-formulate the

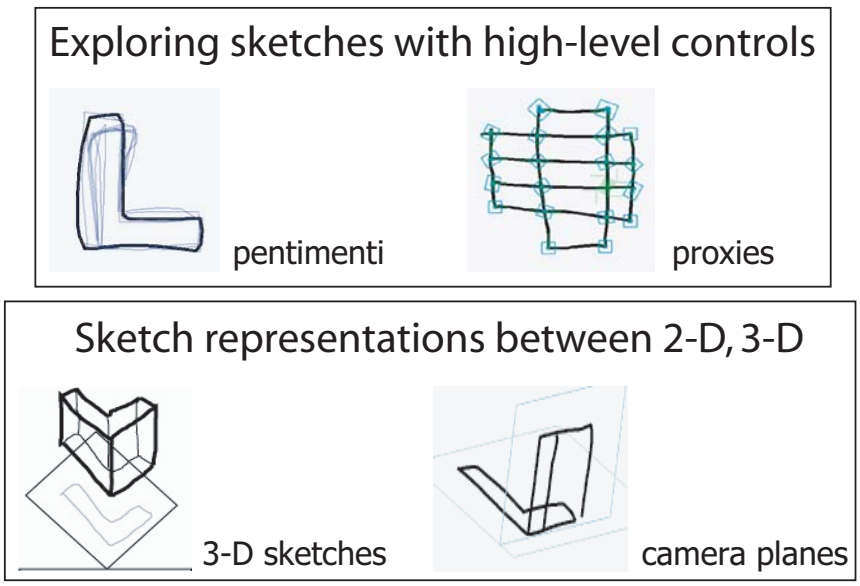

Figure 1-4: Our contributions. This thesis introduces new ways to explore and visualize form.

bottom-up collection of strokes in the sketch as a top-down collection of primitives, often resulting in an unavoidable simplification of the drawing (to a perfect, clean-line) in the process. The contribution of this thesis is to provide new tools to edit and visualize sketches directly, in the process removing the boundary between 2-D and 3-D modeling.

We support a wide variety of tools: to edit sketched curves as designers do with pencil and paper, by simply drawing over them, to manipulate unstructured sketches with a simple high-level structured representation, to directly create 3-D form by 2-D sketching, to visualize a collection of 2-D sketches as a 3-D form, to visualize 2-D and 3-D representations in common context, and to translate sketches from 2-D to 3-D. We use strokes both to encode 2-D sketches as well as 3-D models. In this thesis we do not handle images, although they are clearly a promising area for future work.

1.4 Overview

Figure 1-4 summarizes our contributions. We introduce two novel techniques to explore form with sketches. The first, "pentimenti," is based on sketching practice,

where the artist draws lines over an existing form to edit it. The second, "proxies," is based on digital practice, where the artist smoothly explores a range of form by dragging points. These techniques are described in Chapters 3 and 4, respectively.

Additionally, we introduce two novel representations of form which blur the boundaries between 2-D and 3-D. "3-D sketches" are fully 3-D objects but maintain the sketchy appearance of a source 2-D sketch; techniques for creating these sketches from one or two sketches are described in Chapters 5 and 6, respectively. "Camera planes" are a more speculative interface which poses the modeling of form as the accumulation of 2-D sketches on planes within a common 3-D context. The resulting representation combines the speed and suggestive ambiguity of 2-D sketching with the 3-D visualization capabilities of computers. This interface is described in Chapter 7.

Chapter 2

Related Work

In this thesis we aim to facilitate exploration of form with sketches, and to aid designers in moving between 2-D and 3-D. We synthesize existing work from several related areas: geometry creation, geometry manipulation, computer vision (including photogrammetry), non-photorealistic rendering, and image-based rendering. We discuss each in turn.

2.1 Geometry Creation: From Scratch

The most common modeling tools rely on the user to mentally decompose a desired form into appropriate 2-D or 3-D primitives, then instantiate those primitives to create the form. These primitives include basic shapes such as cubes, cylinders, and spheres, smooth curves and surfaces such as B-splines and subdivision surfaces, and unstructured representations such as polylines and triangle meshes. They differ in their complexity of shape and the number of internal degrees of freedom.

The most popular modeling tools require the user to specify all the degrees of freedom of these primitives to great precision. We list three categories, in decreasing order of popularity:

GUI and Direct Manipulation Typically the user will use menus to instantiate a standard primitive (such as a cube), then directly specify its key parameters

(such as length, width, and height) through dragging points, sometimes using multiple views to specify key dimensions along different 3-D axes. For control-point-based primitives such as polylines, B-spline curves, and NURBS surfaces, the user clicks on screen locations to add points. In some cases the user will type in numerical parameters specifying these primitive dimensions or even control point locations (X, Y, Z).

Procedural Modeling/Scripting With these text-based interfaces, the user types in an algorithm to construct the desired shape. Many commercial packages (such as Maya[3], AutoCAD[5], and FormZ[4]) provide custom scripting languages for this task. Scripting provides power users with great control and flexibility; however, it is an indirect, non-visual means of specifying form.

Relational Modeling Termed "parametric modeling" in commercial packages (such as CATIA [75] and Revit [6]), this interface enables the user to define every primitive dimension or control point as a function of some other quantity. Unlike the scripting interfaces described above, these relations remain the core representation, enabling the user to construct and manipulate geometries with complicated dependencies. Like scripting, however, relations are specified in an indirect, non-visual way.

Exhaustive control over the form comes at the price of more heavyweight specification; this cost is often unnecessary in early design stages where the form is not yet precisely known. Other interfaces require less information from the user, inferring some parameters:

Gestural Interfaces In gesture-based interfaces, the user draws strokes that the computer interprets as commands. In modeling tools, they have been used to instantiate and edit primitives, both in 2-D and 3-D, such as in Eggli et al.[24], SKETCH[83] (see Figure 2-1), Teddy[39], and Pereira et al.[56]. These have proven particularly effective when the shape of the gesture mimics the silhouette of the resulting primitive.

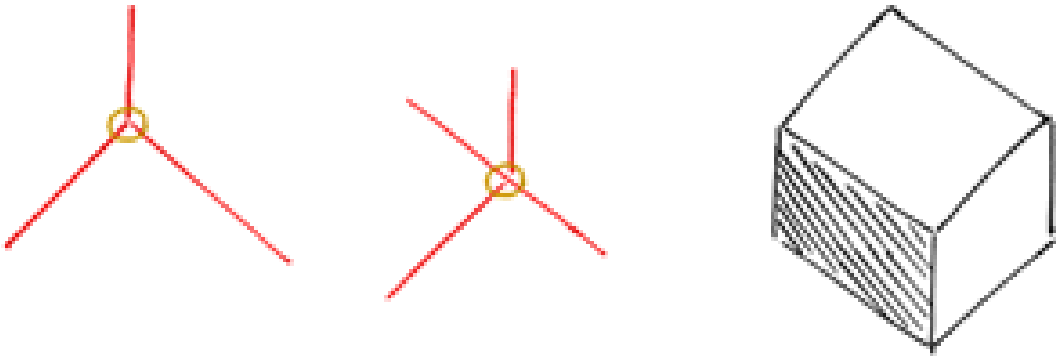

Figure 2-1: With gestural interfaces (SKETCH[83] is shown here), the user draws a 2-D symbol (either the left and center configuration), and the system creates a corresponding 3-D shape (right); the sketchy rendering style is entirely generated by the system, independent of the user's strokes.

Sketch Recognition These approaches handle input strokes more passively than gestural interfaces, often waiting for user confirmation to transform or otherwise act on their interpretations. This delay allows them to interpret more complex sequences or arrangements of strokes. These interpretations are highly domain-specific. Landay [42] presents a system for creating user interfaces which is unique for maintaining the original sketched gesture, turning each stroke into an active interface element. Saund [61] attempts to turn a set of edges into higher-level geometric primitives such as boxes or arcs; Shilman and Viola[64] group arbitrary strokes into higher-level objects. Preprocessing systems such as Shpitalni and Lipson [65] cluster endpoints, in effect linking isolated line segments into a graph, in preparation for 3-D reconstruction. Recently Hammond and Davis [34] have described a general language for interpreting stroke input based on context; this might prove effective for modeling purposes. "Beautification" systems such as Pavlidis and Van Wyk [55] attempt to recognize spatial relations and strengthen them, such as by straightening lines or making lines parallel, for aesthetic reasons.

Suggestive Interfaces "Suggestive" interfaces such as CHATEAU[38] and Tsang et al. [78] propose local geometry additions based on local geometric character-

istics of the current form.

While these interfaces greatly simplify the modeling process, their inferences can be wrong, requiring one of the more heavyweight interfaces to fix, and they lose some information in their sketched input, treating strokes purely as a set of abstract commands. Most importantly, they are still based on the user's mental preprocessing of the form into primitives. In contrast, direct drawing of polylines, a technique specific to 2-D, relies on the accumulation of lines to describe a shape. The emergent *perceptual* representation is thus generally disjoint from the unstructured digital representation. This strategy's efficiency allows a large space of forms to be rapidly explored simply by drawing various options rather than continually reworking an existing form. Our '3-D sketches' representation (created in Chapters 5 and 6) transfers this emergent representation from 2-D to 3-D. Tolba et al.[77] extends this technique to a spherical projective space, enabling "in-the-round" visualization and a vocabulary of 2-D transformations that appear to be 3-D. Our "modeling with planes" interface (described in Chapter 7) extends this idea to multiple projective spaces in a common 3-D context as well as enabling freehand drawing into 3-D space.

2.2 Geometry Manipulation

Digital modelers primarily edit form by clicking and dragging 2-D points around; the results are displayed continuously. With this interface, a large space of possible forms can be explored with a single mouse drag. The meaning of these points, and thus the space of explorable forms, varies among interfaces as described below:

Control Points In the simplest direct manipulation interface used in most modeling packages (such as Maya [3] and Adobe Illustrator [1]), the points are those intrinsic to the geometric representation. With b-splines, NURBS, and subdivision surfaces, the control points are close to, but not on, the curve or surface (such as in Figure 2-2). With polylines, basic triangle meshes, and interpolating splines, the points lie on the curve or surface. With multiresolution representations (described in Finkelstein and Salesin[26]) different control points work

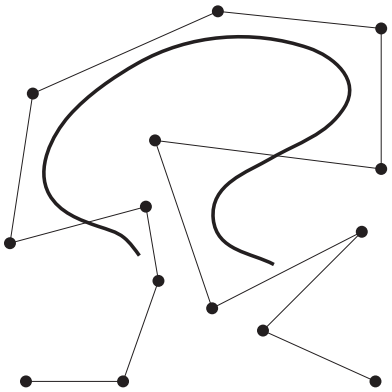

Figure 2-2: Some representations, such as the B-spline shown here, have control points which do not lie exactly on the final curve. The distribution of control points is still determined by the curve, however, and not under the user's control.

at different scales, which enables the independent exploration of fine detail and large-scale structures. As shown in Figure 4-1, control-point dragging requires that the control points correspond with features the user wishes to control — a situation much more common with the clean geometry created in later design stages than with freehand polyline sketches.

Point Constraints One could also consider a number of constraint-based surfaces to have point-dragging interfaces. In these works, the user is dragging point location constraints; the system then does extensive computation to find a constrained shape to fit those points. An interface for modeling with smoothness constraints is described in Gleicher and Witkin[29] and Welch and Witkin[80], while Moreton and Séquin[53] present a technique for generating such surfaces. Modeling based on physical processes such as the movement of cloth (supported in commercial modelers such as Maya[3] and described in Baraff and Witkin[7]) allows users to interact with shapes as they do in the real world — for example, shirts fold and wrinkle as the user pulls points on them. This approach requires that the rest of the surface not under the user's direct control be described with constraint equations; these constraints are better known later in the design process than the early phases this thesis concentrates on.

Deformation Parameters Commercial modeling software such as Maya [3] support many types of specialized deformations such as smoothing, stretching, and twisting. Here point dragging is used to specify parameters (such as the angle of twisting) rather than to move control points.

Proxy Control Points With proxy-based methods, the user drags control points on a scaffolding or skeleton geometry. Standard proxies include skeletons, such as with so-called "skinning" methods (Mohr et al.[52]), as well as 3-D lattice grids, as in "free-form deformation" (Sederburg and Parry [62]). Manipulating the skeleton causes the body surface to deform, while warping the lattice grid causes a model embedded inside to similarly warp. We describe these methods in more detail in Chapter 4.

In Chapter 4 we describe a technique, most similar to the "proxy control points" section above, to enable high-level control of sketched strokes through dragging of control points.

An entirely different way of editing geometry is through overdraw, as in Baudel[9]. This work relies on editing through line drawing rather than point dragging; essentially the user edits curves by redrawing parts of them. Our "pentimenti" (described in Chapter 3) technique extends this interface to support the common sketching practice of tracing back and forth over a line, as well as handling multiple, possibly intersecting, polylines.

2.3 Transition from 2-D to 3-D

A different way to create 3-D geometry is by bootstrapping from 2-D geometry. This approach fits with designers' workflow since they typically have made many 2-D sketches before moving to a 3-D model. This process is actually an iterative one: even after building a 3-D model, a designer will typically return to 2-D sketches, and then build another 3-D model, and so forth. While designers create both physical and digital 3-D models, we will concentrate solely on digital 3-D models.

The current practice for creating a 3-D model from a sketch is unfortunately brute-force: to start from scratch with the 3-D geometry creation and manipulation tools described above. With such tools, 2-D source material can be leveraged as a background to trace over. While time-consuming, this approach is also extremely flexible, able to handle arbitrarily inaccurate and ambiguous sketches.

The computer vision community has extensively studied the inference of 3-D form from 2-D information. We first confine our discussion to reconstruction from a single 2-D view. Shape-from-contour approaches attempt to replicate our human ability to automatically infer 3-D information from a single, unannotated view. Geometrically speaking, they recover depth coordinates of 2-D contours — although it is unclear whether human beings' 3-D perception is equivalent to recovering full depth information (Lowe[48]). These systems are reviewed in [16]; two recent papers not covered in that survey are Lipson and Shpitalni[45] and Shesh and Chen[63]. As humans apparently do, they make assumptions about 3-D properties from observed 2-D characteristics and create a best-guess 3-D model. Unfortunately the set of assumptions used by human beings are not yet well understood, even under the assumptions in state-of-the-art work of clean-line, closed polyhedral stimuli, nor are they easily parameterized for user intervention. Other systems (Mitani et al.[51] and Thorne et al.[76]) explicitly use a template — assuming strokes are drawn in a particular order, and connected in a specific way. This assumption, although strict, allows greater success in reconstruction and ease in providing a user interface. Noting that sketches are typically line-based, we do not treat the similarly automated "shape from shading" approaches (see Brooks and Horn[12] and Bourguignon et al.[11]).

The literature in computer vision on the problem of 3-D reconstruction from two or more 2-D views is vast. However, the nature of our problem deviates from the assumptions in these works in important ways. We use input strokes instead of images; 2-D sketches do not exhibit the same content or metric accuracy as photographic imagery; and the views might not be consistent projections of a 3-D model. An additional challenge posed by sketching practice is the pervasive use of degenerate views where 3-D edges are parallel to the viewing direction and thus project to a

single point.

A middle ground between brute force and automated approach is occupied by so-called "photogrammetry" or "image metrology" methods. These work with images instead of the 2-D contours of "shape from contours" methods; and rather than transform the images directly, they rely on the user to construct proxy 2-D geometry and annotate them with 3-D information (An exception is Oh et al. [54], where the user directly "paints" depth onto the image). Typically the resulting proxy 3-D model provides a collection of surfaces which the image can be projected onto, becoming a set of texture maps for the model. In the "Façade" system [18] the user builds a coarse 3-D model and identifies image edges; the system fits the model to the images, and (given multiple images) provides fine 3-D detail through pixelwise image correspondences. Liebowitz et al.[44] outline a modeling algorithm based on serial user-guided reconstruction of adjacent planar quadrilaterals in photographic images, where the user annotates each polygon in turn by specifying its vanishing points to the system, which recovers its placement and orientation relative to the previous plane. The "Tour into the Picture" interface [37] allows the user to reconstruct simple scenes with one-point perspective, where the user specifies a ground plane and several vertical billboards. Most similar to our own work, Poulin et al.[57], Shum et al.[66], and PhotoModeler[25] each present a vocabulary of point, line, and plane relations with which to annotate the image; they also support multi-image input, where the user can specify correspondences between images. Their mathematical formulations are linear and thus easy to solve.

In Chapters 5 and 6 we describe user-guided reconstruction approaches inspired by the photogrammetry works but taking messy stroke-based input. The chapters cover approaches given one and two source sketches, respectively.

2.4 Sketch Rendering of 3-D Models

The so-called "Non-Photorealistic Rendering" or "NPR" field addresses a problem that is complementary to our goals: to generate sketch-like 2-D images from 3-D

models. This is in a sense the inverse of one of the problems that we address, which is to add three-dimensionality to a 2-D sketch. These works attempt to support more abstract uses of images, including simulation of traditional media such as charcoal (Durand et al.[23]) or pen and ink rendering (Winkenbach and Salesin[81]), or simulation of artistic procedures to satisfy a variety of pictorial goals such as highlighting contours (Decarlo et al.[19]) or clarity in technical illustration (Gooch et al.[30]). User interfaces to create more general effects include Kalnins et al.[40] and Grabli et al.[31]. The most related usage of NPR is in the SKETCH interface[83], cited above as a gestural interface. In that work, the user draws a gesture with strokes, which the system translates into a clean 3-D primitive and renders — that is, translates back to 2-D — in a sketchy style using NPR techniques. The original strokes drawn by the user are lost; in contrast, our work maintains the user‘s strokes as we transition to 3-D, whether by direct translation (as in Chapters 5 and 6) or by placing them on planes (as in Chapter 7). The advantage of this approach is that it maintains any extra information in the stroke — whether aesthetic or geometric — beyond its identifying a particular primitive.

2.5 Image-Based Rendering

Various works have transformed 3-D models into sparse 2-D representations for interactive rendering or simplification. Billboard clouds [20] re-express a polygonal description as a set of transparent planes. Image-based rendering (introduced in McMillan and Bishop [49]) creates arbitrary views of 3-D models from an incomplete set of images taken from known viewpoints. “Imposters” are 2-D billboards containing the appearance of a 3-D scene from a given viewpoint (acting like the background sets in a theater production), used to speed up rendering (as in Decoret et al.[21]). Our “modeling with planes” technique (Chapter 7) performs the opposite task: building a 3-D model by accumulating a sparse set of 2-D sketches. We exploit an insight from these works: 3-D form can be well-represented by surprisingly few planes, and human viewers can be forgiving of holes and other artifacts created by

this approximation. Rather than reproducing high-fidelity 3-D form, however, we use 2-D representation to enable designers to construct and visualize sketchy, incomplete 3-D forms — additionally taking advantage of a designer's ability to use missing information as a creative catalyst.

Chapter 3

Pentimenti: Exploring Curves Through Overdraw

In this chapter we describe a technique for editing sketches that is inspired by sketching practice, where the user creates a series of discrete alternatives through freehand drawing; in the next chapter we will describe another method to edit sketches that is based on the smooth point-dragging provided by digital tools. With pencil and paper, designers can explore alternatives simply by drawing over previous strokes. We introduce a digital interface premised on the same idea but allowing the exploration of a entire *space* of curves rather than individual alternatives. If the user draws a new curve near an existing one with our technique, the new curve replaces a corresponding segment on the old one. Performing this operation in real-time, as the user draws, allows the user to trace back and forth with a single stroke to explore multiple curves. Not only can the user explore a wider set of curves with one stroke, but the drawing remains clean, without the layering of sketches. This layering can be useful as a reference, however, so we allow users to have it both ways by maintaining a history layer of previous curves, which users can hide at will. Our technique not only supports replacement of old strokes but additionally can extend them or change their topology (such as creating a fork in the curve or eliminating such a fork).

3.1 Motivation

Sketchers of real-world scenes sometimes choose to maintain a very clean sketch (such as in Figure 3-1a) and never revise a line. This strategy gives clarity to sketches and encourages care in the artist. When sketching an imagined form, designers will often trace over a line while thinking about it, as in Figure 3-1c. By confining it to a single element, as in Figure 3-1b, the sketcher can even exploit the resulting clutter as a visual device to highlight the element's importance in the sketch. Exploring several versions of a curve extends to painting practice. Evidence of the earlier underdrawings (see Figure 3-2) is known in the painting world as "pentimenti," which derives from the Italian "pentire," which derives from the Latin "pæntire": "repentance," "contrition," or "regret." We have thus chosen to name our interface "pentimenti" — that is, an interactive, real-time repentance system for designers.

Our system also borrows elements from digital tools, which modify rather than accumulate drawing elements. Current techniques force the user to choose between continuous exploration of a small amount of variation (by dragging control points) or explicit specification of a single large change (by drawing a new part of the curve). Our interface synthesizes these elements, allowing continuous exploration of large changes.

3.2 Related Work

3.2.1 Freehand Drawing

Freehand drawing, whether with pencil and paper or digital image-editing programs such as Adobe Photoshop [2], simply place new strokes on top of existing ones, relying on the user's perception to interpret that stroke's function — such as whether it is a new line, or darkening an existing line, or superseding a similar line. Completely redrawing a sketch is particularly effective for large changes, but is less efficient for small edits, where much of the redrawing effort is simply reproducing the original shape. Drawing *over* a sketch supports both small and large changes effectively —

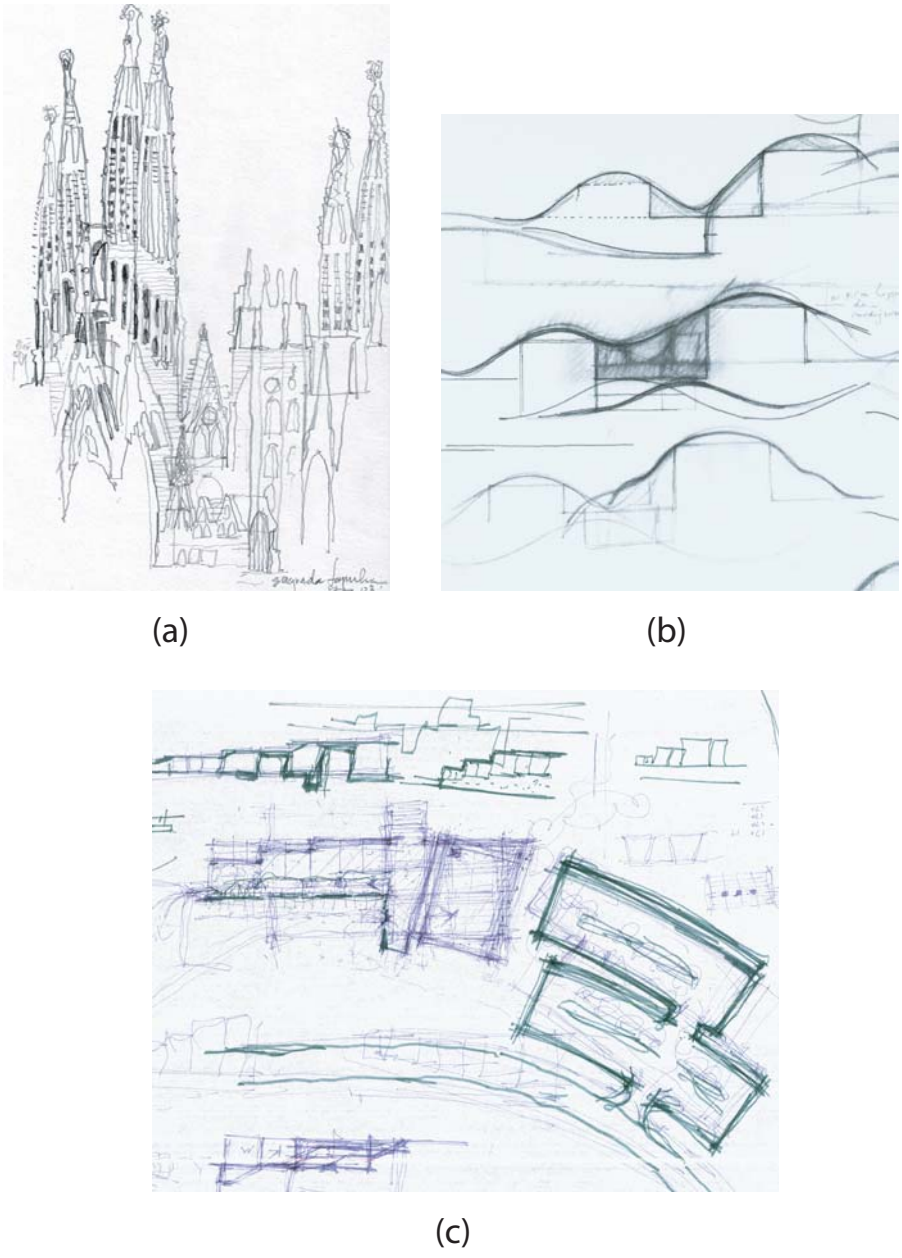

Figure 3-1: Clean-line sketch (a) of an existing building: The Sagrada Familia church in Barcelona, Spain, was designed primarily by Antonio Gaudí. Construction began at the end of the 19th century and continues today. Sketch by Yanni Loukissas, 2002. In (b), sketches by Herman Hertzberger [36] for a theater. Note how most lines were only drawn once, but the key curves were repeatedly drawn over. In (c), messier sketches of an imagined building. These were made by Schwartz/Silver Architects during the design phase for a competition for a conference center in Woods Hole, MA.

Figure 3-2: A 19th century painting, "The Derby Day," by William Frith, and its pencil underdrawing, seen as a mosaic of infrared images. [79]

the user just draws the new parts of the desired form over the old one — but only for a limited number of alternatives, since the sketch quickly becomes too messy to understand. Our technique supports the advantages of both; the user can make changes efficiently as with overdrawing but maintains the cleanliness of a complete redraw.

An additional advantage of freehand drawing over current digital editing tools is that older attempts are available as a reference; our system maintains this perceptual phenomenon by continuing to draw superseded strokes, but treats them as background imagery only, not live geometry. With digital editors such as Photoshop [2] or Genau and Kramer[28], older strokes could be similarly maintained on lower layers; however, the demotion of strokes to lower layers would have to be done by hand. Since the edited curve is capable of taking on many different positions over the course of a single stroke, it would be quite difficult for the user (we imagine a designer with pen in one hand and keyboard under the other) to manually snapshot every possible curve position.

3.2.2 Editing by Point Dragging

Digital editing tools provide less freedom than freehand sketching, but more control. The most common interface, point-dragging methods (covered in Section 2.2), provide continuous feedback, allowing the user to explore many positions with a single drag,

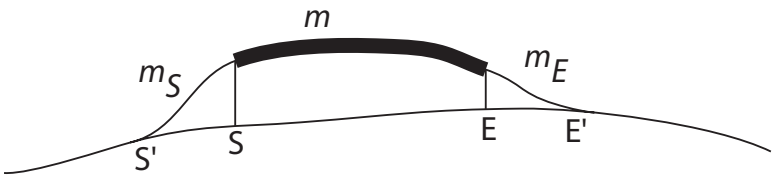

Figure 3-3: Baudel[9] algorithm: the user draws editing stroke m; when the curve is complete, the system finds the closest corresponding subcurve SE and replaces it with m; to maintain tangent continuity, the system also replaces adjacent segments S'S and EE' with automatically calculated segments m_S and m_E, respectively.

but they affect only a fixed section of the curve. The pentimenti interface will explore a higher-dimensional space of curves.

3.2.3 Editing by Drawing Freehand Curves

Most similar to our work is the curve-replacement interface of Baudel [9]. This interface has four modes: for creating new curves, editing existing curves, deleting curves, and linking endpoints of existing curves. The edit mode is of most interest: the user draws a new curve, and upon mouse up, the system finds a corresponding section of some nearby, existing curve and replaces it with the new curve. If there are multiple existing curves nearby, the system asks the user to disambiguate, then does the replacement. An additional, adjacent section of the existing curve is also modified to maintain tangent continuity between the old and new parts of the curve, and the final curve is re-smoothed; see Figure 3-3.

The section of original curve to be replaced is found through a optimization process which seeks to minimize the distance between evenly spaced points (where the spacing on each curve is a fraction of its total arclength) on each curve. The authors note that their optimization formulation could have several local minima when replacing a curve with a loop in it.

An interface similar to the above is commercially available in Adobe Illustrator [1]; although we do not know the algorithms used, we can guess at them. In this work, the "create" and "edit" functions of Baudel[9] have been integrated; that is, if no curve is nearby to the newly drawn curve, it is considered a "create" operation.

Only the currently selected curve can be edited, removing the possible ambiguity if several curves are nearby. From our experiments it appears that they use a similar curve-to-curve distance metric when finding the subcurve to replace.

3.3 Exposition

The primary contribution of our work is the real-time exploration of many curves with just one stroke. This is made possible through several novel techniques centered around real-time editing and feedback, discussed in Section 3.3.1:

- While the user draws a stroke, our system continuously modifies the source curve accordingly. From the user's point of view, this enables tracing back and forth, trying several curve locations with a single stroke. From an implementation point of view, local updating greatly simplifies our algorithm and makes it easy to establish fine-grained correspondences between the source and target curve. This, in turn, makes it easy to transfer attributes (such as opacity or color) from the original curve.

- We simplify several important constants in our algorithm to one scalar value d, and introduce both a real-time visualization of that value's effect (enabling the user to predict the system's behavior) and a simple interface for the user to change that value between strokes, effectively shifting the preference of the system between editing and creating new lines. Previous systems did not visualize their tolerances nor expose them to user control.[1]

- We integrate an automatic visualization of old strokes into our algorithm. This greatly enhances the usability of the system, in effect automatically maintaining a visual record of previously attempted curves.

A secondary contribution is seamless support for creating and editing curves with more complex topology, discussed in Section 3.3.2:

[1] One exception is Adobe Illustrator[1] where the user can set a cutoff distance (in pixels) from the selected curve that decides whether a new curve is created or the selected curve is edited. This setting is done numerically in a dialog box, as opposed to visually in context, as in our system.

- As in Adobe Illustrator[1], we integrate "create" and "edit" modes. Additionally, we integrate the "join" mode of Baudel[9].
- We extend our algorithm to handle curves with vertices with valence greater than two — essentially, tree-like or web-like structures.
- We add two additional operations: forking off a new edge from a vertex and eliminating such forks through "sewing." Tolerances for these operations are also folded into our single visualization.

We cover each contribution in turn, then present pseudocode linking them all in section 3.3.4.

3.3.1 Visualization

The visualization of d and an interface to control it is shown in Figure 3-4. We will describe how d is used in the ensuing exposition.

As the user moves the mouse over the scene, we highlight the closest edge, thus removing any ambiguity as to which curve will be edited.

In order to provide the visual layering effect as in freehand sketching, we simply record a separate "background stroke" as the user draws (shown in Figure 3-4). These strokes are persistent and are rendered underneath active geometry but are not themselves a live part of the model. As with strokes drawn on paper, they can never be moved again. We also provide controls for the user to vary the opacity of these strokes or delete them altogether.

3.3.2 Fine-grained, Immediate Subcurve Replacement

By interactively editing the curve at every mouse move event, our system decomposes a potentially complex matching problem into a series of much simpler, smaller edits. The matching problem in Baudel[9] involves searching the entire original curve for the best matching series of line segments to an input curve (in Figure 3-3, their algorithm must determine the location of curve segment SE). In our formulation, the

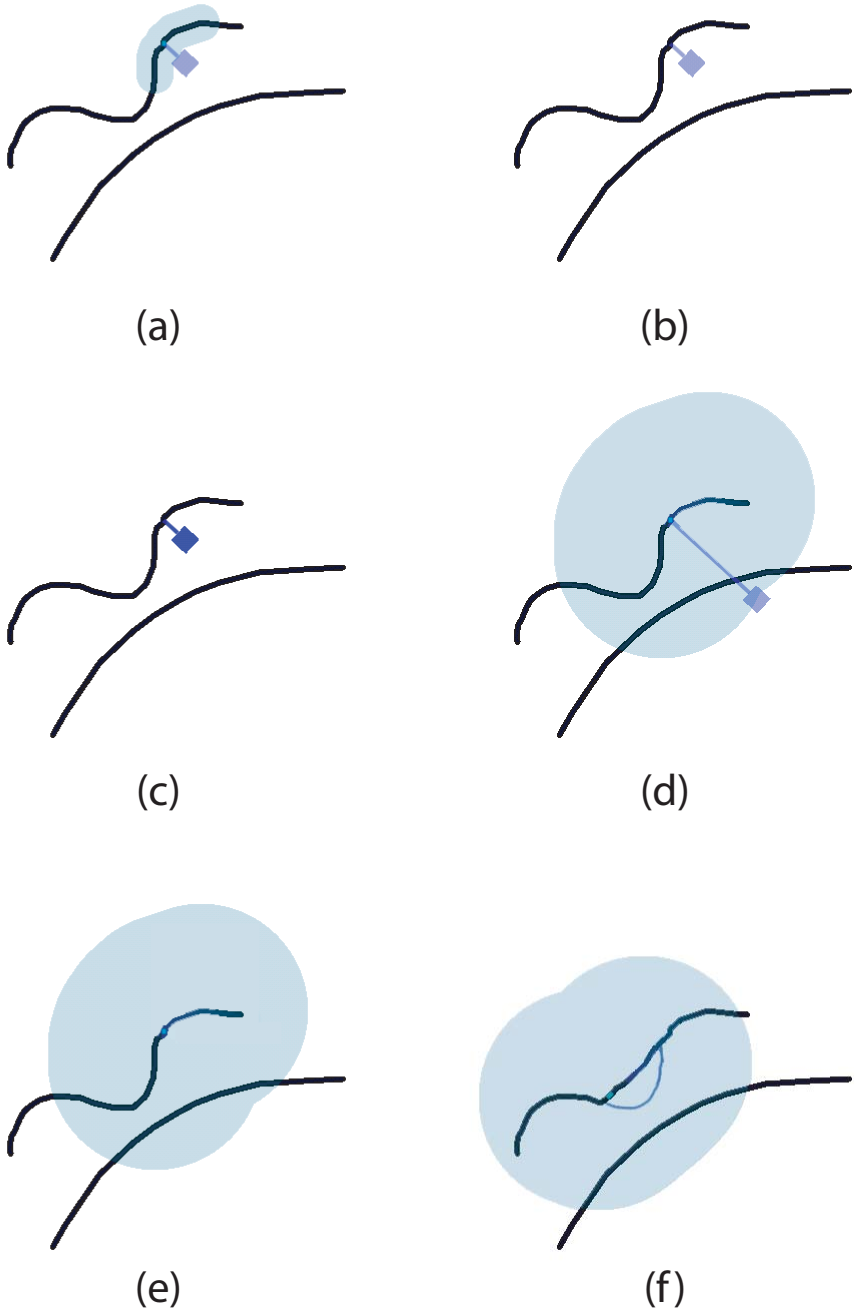

Figure 3-4: Visualization of unified tolerance measure d. The user hovers within d of the curve (a), goes more than d away from the curve (b), hovers over the widget (c), drags the widget, increasing d (d), then begins to actually draw over the curve (e), editing a segment and leaving a history trace of the old (curvier) curve underneath (f).

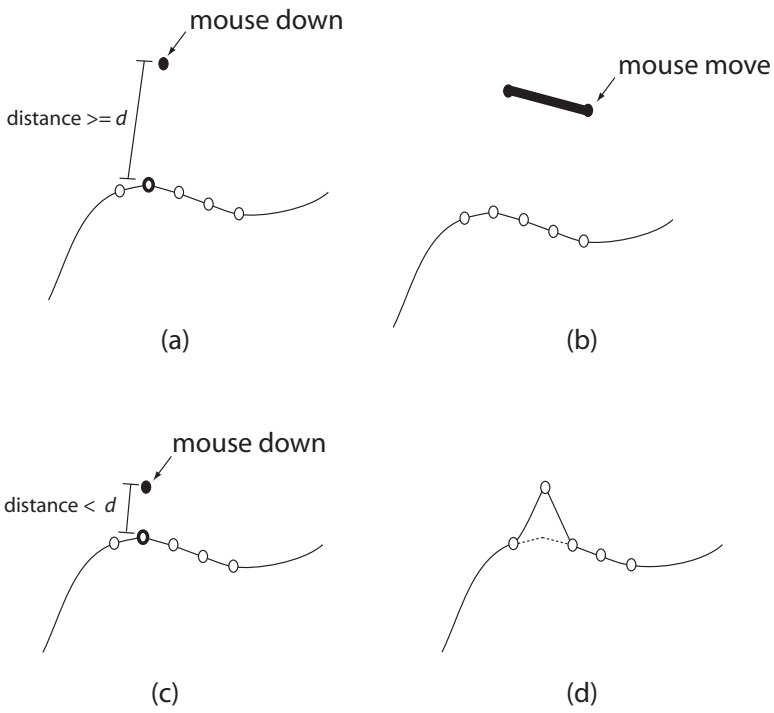

Figure 3-5: Fine-grained editing: mouse down event. The user starts a drag; if that drag is further than d away from any curve(a), the system begins a new curve (b). Otherwise, the system finds the closest curve point (c) and moves it to the drag point (d).

mouse down event establishes a current point on the source curve that subsequent mouse moves maintain (See Figure 3-5). Further, since we handle each mouse move immediately as it comes in, we have only one new segment to match at a time. We can thus simply walk along the source curve, from the current vertex, looking for a closer vertex to the latest drag point (See Figure 3-6). All vertices encountered along this path are projected onto the new edge. In order to prevent degenerate edges, if two adjacent vertices are projected to within a pixel of each other, the system merges them. Additionally, the distance d is used to bound the traversal search; this can often be helpful for specifying the scope of the edits (See Figure 3-7).

A drawback of our system is with small-angled corners (less than 60 degrees) — where it is ambiguous whether the user wishes to edit the curve or create a sharp corner. Smaller d were much more effective in these situations; we were also able to use the "sewing" method of section 3.3.3 to fix up unexpected behavior (see Figure

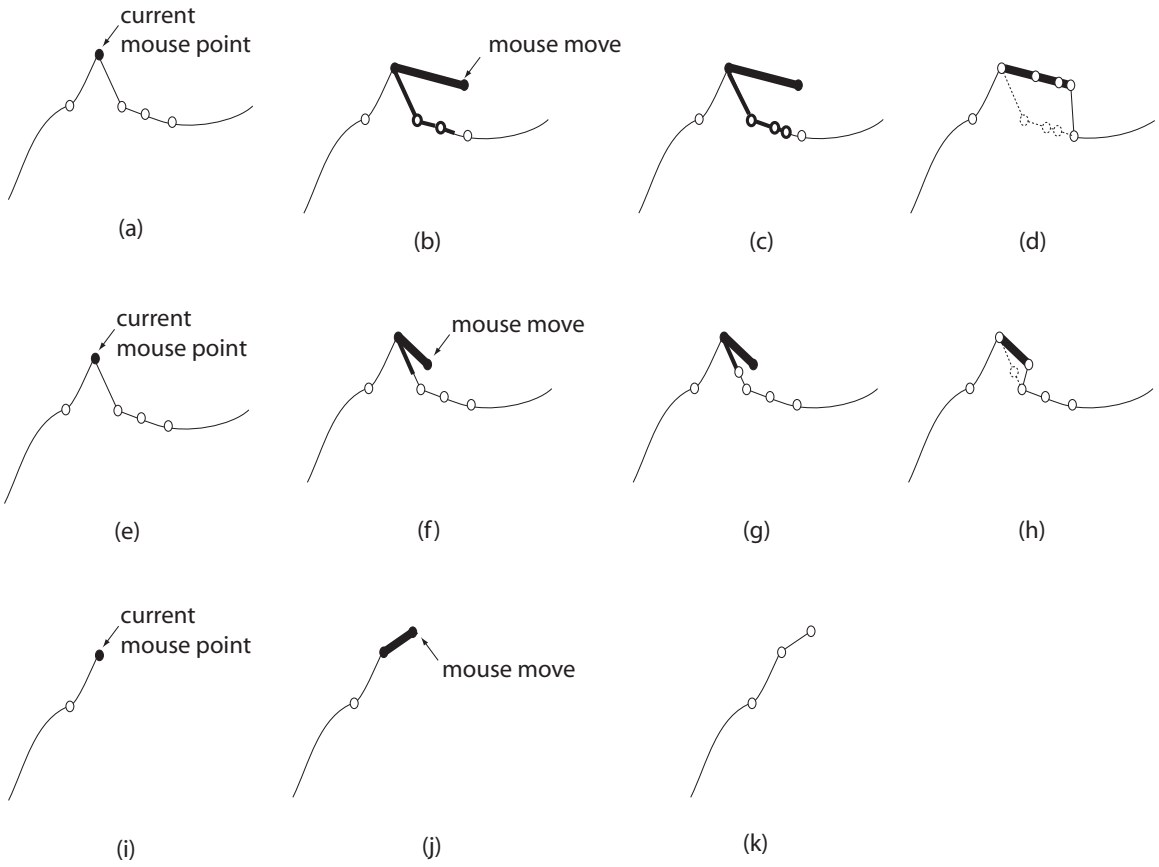

Figure 3-6: Fine-grained editing: drag event. The system begins with a current point on the source polyline from the previous move or mouse down event(a), and the new screen location that the mouse/pen has been dragged to. It searches along the polyline for the nearest point to the new mouse point(b). If the nearest point is a pixel or more from than the nearest vertex, the corresponding edge is split (c) and will be projected onto the new edge. Any intermediate vertices traversed along the way are also projected onto the edge(d). In the same start situation (e), the user draws a different stroke(f); no intermediate vertices were traversed, but since the nearest point was more than a pixel from any vertex, the edge is still split (g) and projected (h). In the special case where the current point was an endpoint (valence 1) (i), and no intermediate vertices were traversed nor edges split (j), the system extends the curve to the new mouse position (k).

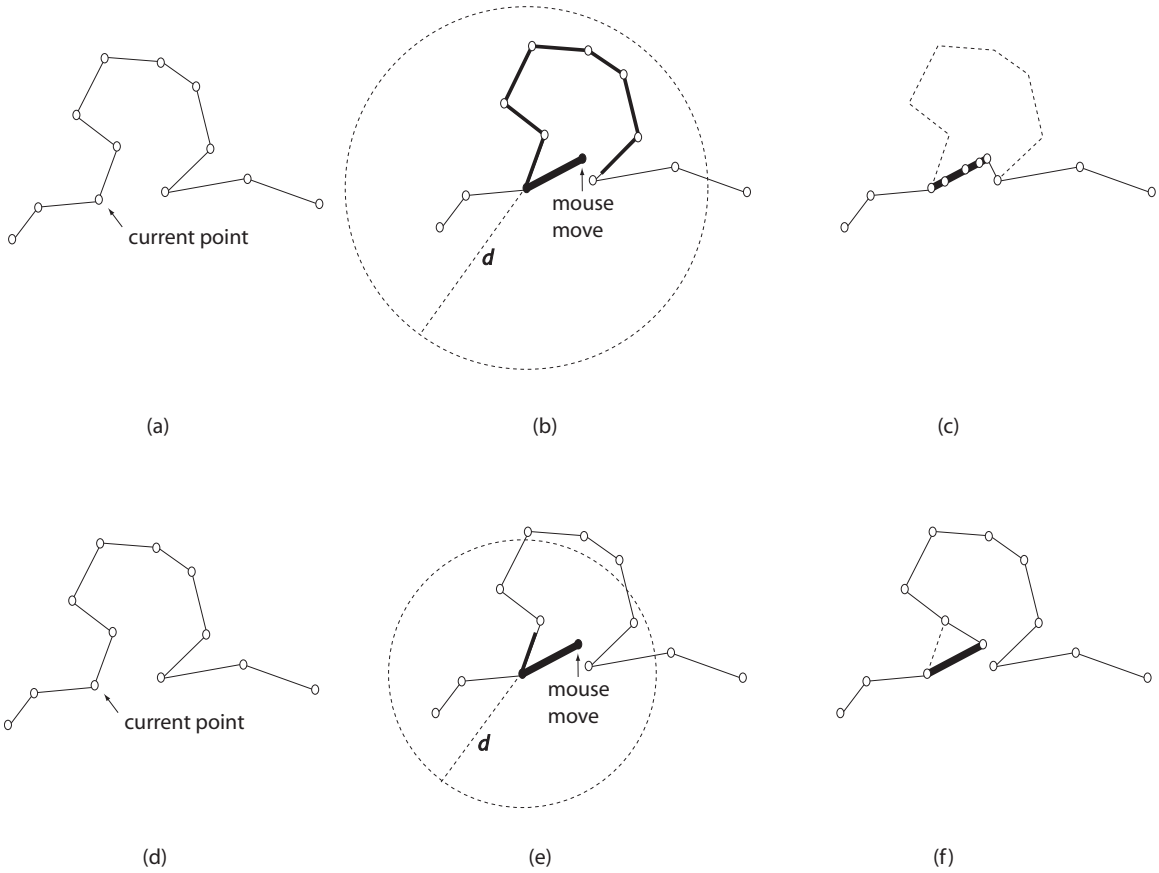

Figure 3-7: Bump collapsing using traversal distance. In situation (a), the user moves the mouse, and after finding the nearest point with d (b), connects to it, in effect collapsing the bump in the curve (c). The user may have merely wished to modify edges along the bump, however, and we support that choice by bounding the traversal with user-specified distance d. Starting from the same situation (d) and dragging out the same edge but this time with a smaller d in place (e), the closest point in range is simply on the adjacent edge, and thus the mouse move has the effect of editing a more local part of the curve (f), as desired.

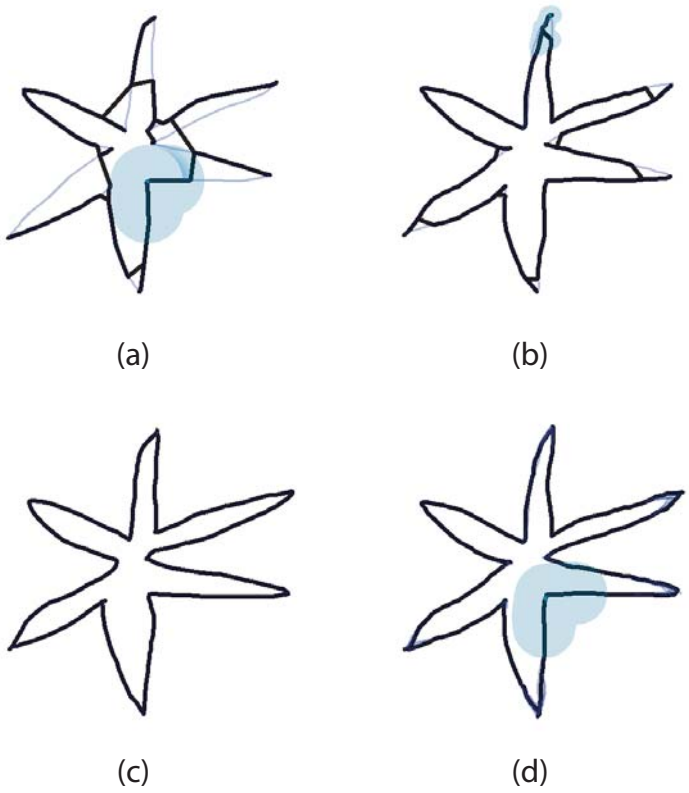

Figure 3-8: Sharp, small corners. We attempted to draw a star-like shape with a large d (a), a smaller d (b), and $d == 0$ (c). With a large d, many parts of the stroke were interpreted as edits; at the other extreme, with $d == 0$, none were. We took the model from (b) and, using a larger d, sewed the corners back up, resulting in (d).

3-8).

3.3.3 Extensions for More Curve Topologies, Topological Operations

In this section we extend our algorithm to handle vertices with valence 3 or more; in other words, we support T- or cross-like intersections, or more generally tree- or web-like topology (see Figure 3-9). To add basic editing capabilities, we simply extend our traversal algorithm to branch at high-valence vertices and search all possible paths; this is essentially a standard enumeration of vertices in a graph. While Baudel [9] does not mention this case, adding such functionality to their algorithm would be more

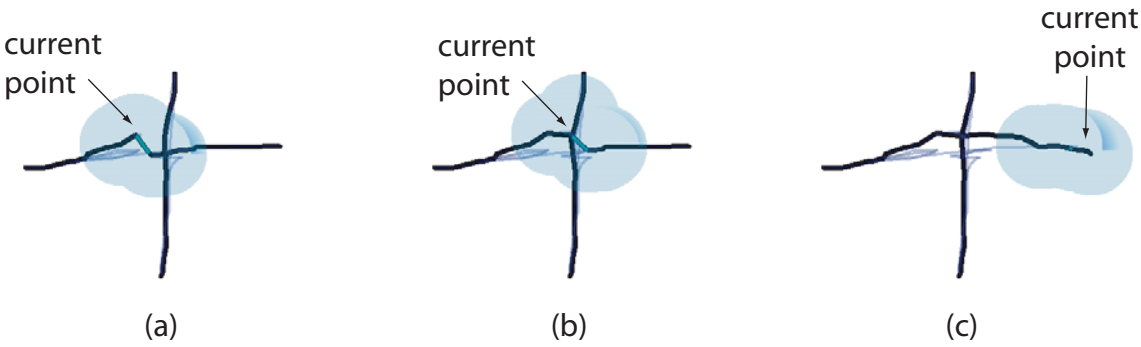

Figure 3-9: Editing a high-valence point. The user approaches the cross intersection(a), goes through (b), and finishes the stroke (c). As the user went near the cross, the closest-point algorithm was searching all branches of the edge-vertex graph.

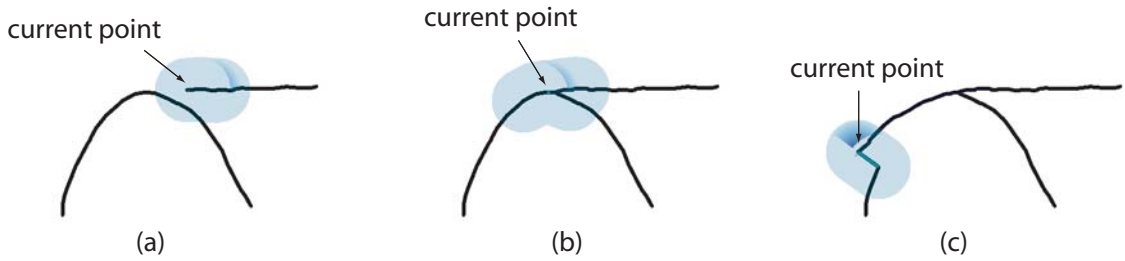

Figure 3-10: Integrating join mode. The user approaches another curve (a), intersects it (b), and continues on, editing the intersected curve (c).

difficult, requiring the enumeration of every distinct path in order to evaluate their curve-to-curve objective function, as opposed to the enumeration of every distinct vertex.

Just as the system automatically collapses a degenerate edge by merging its component vertices, it also automatically merges two outgoing edges that are roughly parallel (defined as one of the far vertices of an outgoing edge lying within a pixel of another outgoing edge).

We also integrate the "join" mode of Baudel [9] by detecting intersections as the user draws. Since we support high-valence vertices, intersections can result either in the closing of curves (the original intent of "join" mode) or in the creation of T-junctions (or, with higher-valence vertices, equivalent higher-valence junctions). In practice, we have found that intersection is used rarely compared to the curve-moving

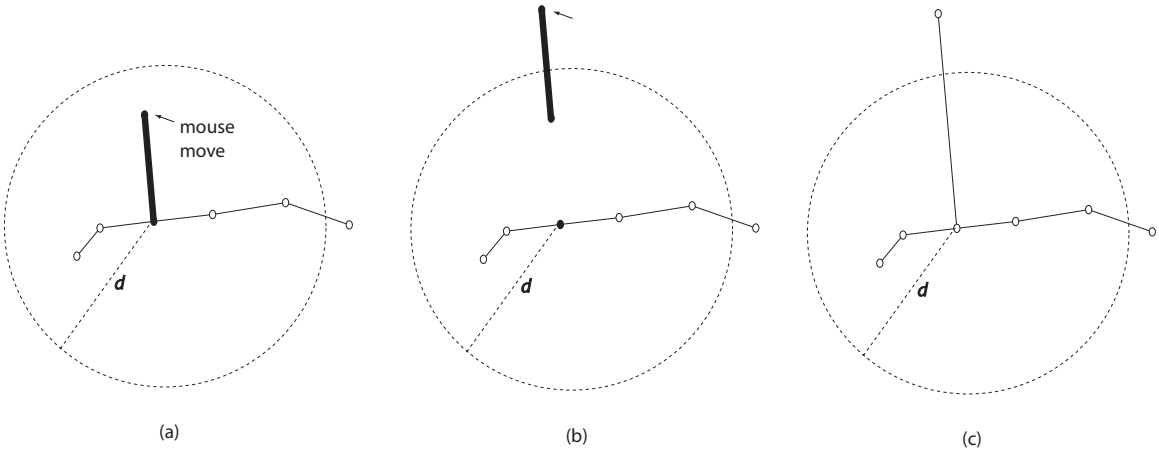

Figure 3-11: Forking diagram. As the user draws perpendicular to the curve in (a), nothing initially occurs since the nearest point on the original curve has not changed, nor has the user left the radius d of that point. When the user leaves the radius (b), a new segment is created (c).

functionality,[2] and, more importantly, not undoable by overdrawing, unlike most of the other operations we support. With these considerations in mind, we chose to restrict our search for intersections to the situations where the user is extending an existing curve (as in sequence i-k of Figure 3-6). In these circumstances, we search all segments in all curves for intersections with the newly minted edge. If we find an intersection, we split both edges at the intersection point and merge them. See Figure 3-10.

We round out our support for high-valence vertices by integrating the creation of forks (described in Figure 3-11) and the inverse of that operation, which we will call sewing (described in Figure 3-12). Our interface again re-uses the distance d, this time to decide whether to treat new strokes as editing strokes or forks. Forks can also be eliminated using essentially a variation on the bump-removal stroke shown in Figure 3-7. The user can thus perform a wide variety of editing tasks with a single continuous motion (see Figure 3-13).

[2]When making such claims, we face the standard problem with new tools: users initially rely on prior experience to use them, possibly not exploiting their features.

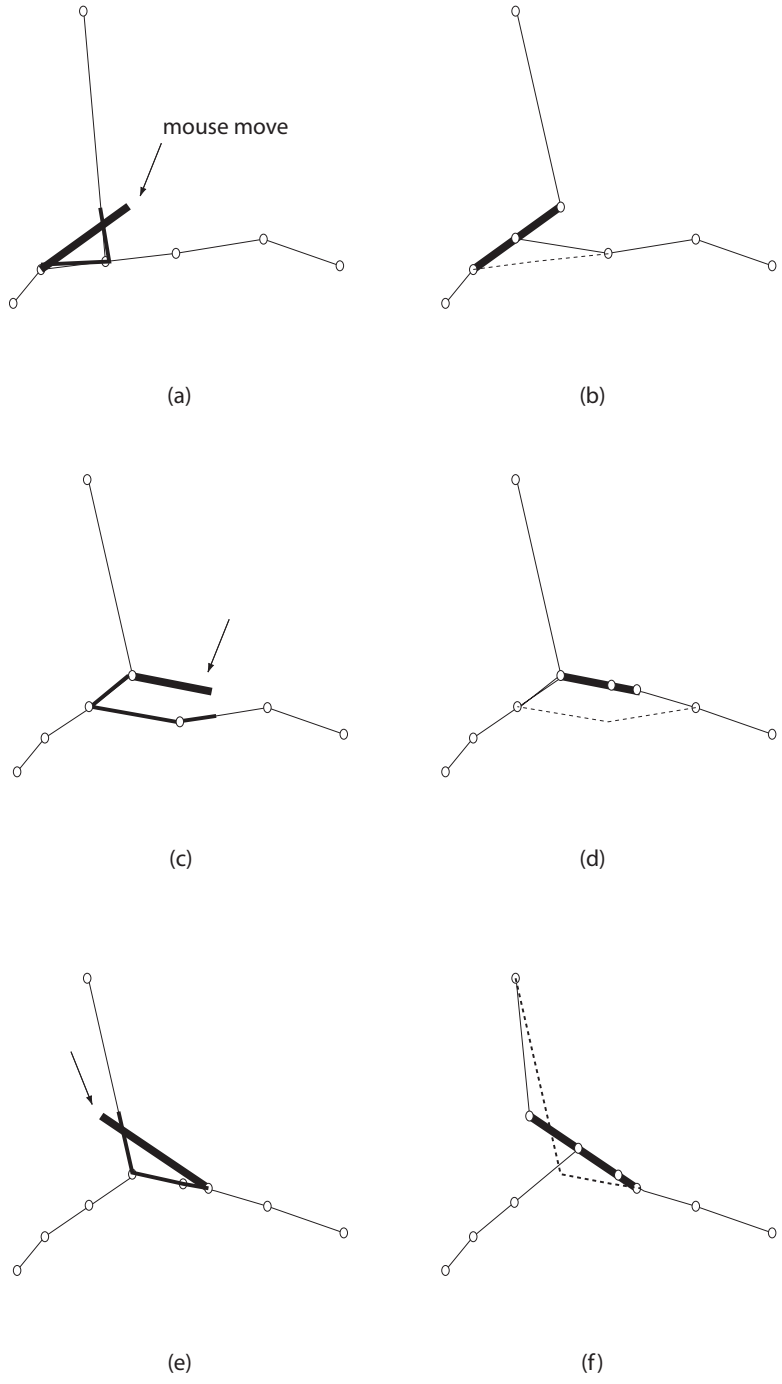

Figure 3-12: Sewing up a fork. The user essentially executes a series of bump-removal operations by zig-zagging up the fork. The individual steps are shown here; a screenshot of the complete process is in Figure 3-13

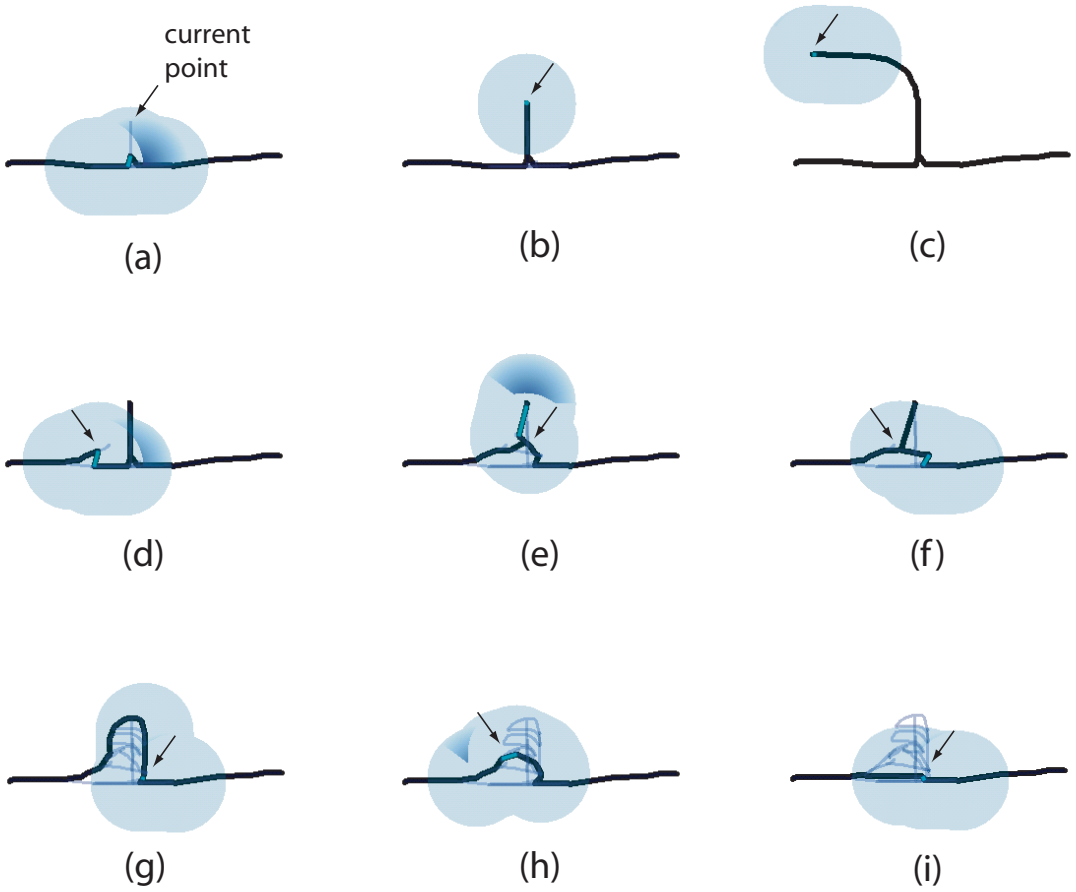

Figure 3-13: Forking and sewing. The user begins to drawing perpendicular to the curve; since the new closest point is identical to the current closest point, no action is taken (a), until the user is more than d away from the curve, at which point the curve creates a new edge, forking (b). The user can then continue on extending the curve (c). Conversely, the user can remove a fork by "sewing" it up with the trivalent vertex (d-i). The user approaches the edge (d), draws past it the first time (e), causing the first sewing operation, draws back the other way (f), then continues in this fashion until the edge is gone (g). The user then collapses the given bump in the curve similar to the fashion shown in Figure 3-7 in (h) and (i).

3.3.4 Overall Algorithm

We present pseudocode for the overall event handling code in Figures 3-14 and 3-15.

Mouse down
Start Point Establish current point p_c with algorithm in Figure 3-5.

Figure 3-14: Mouse-down event handling

3.4 Results

We tested our prototype system on a Tablet PC, which features a pressure-sensitive stylus with a pen tip at one end and an eraser at the other. We utilized the eraser to erase subcurves; as with our overall system, we implemented erasing as a small-scale operation, erasing the edges under and near the cursor, as opposed to the curve-level erase mode in Baudel [9].

Three architecture students used our prototype. They were able to pick up tracing back and forth immediately, as it is similar to known sketching behavior, but had to be taught the fork/sew behavior (we assume because there is no analog in other editing contexts). The students preferred pentimenti to dragging control points or the subcurve-replacement methods, although still preferring classic freehand sketching over all alternatives for initial curve *creation*.

One such student used our interface to design a logo, shown in Figure 3-16. The letters were first blocked out with rectangular shapes in order to establish correct proportions; these shapes were then edited into curves. The entire creation and editing sequence was done solely with our pentimenti interface. This student also built a simple 3-D model (Figure 3-18) using many of the tools in this thesis. The pentimenti interface in particular was used to freehand draw and edit the curves in the model, as described in the caption.

A novice artist drew two scenes from real life with the interface (Figures 3-19 and 3-20), and found it very helpful for finding the correct proportions and composition. Notice that pentimenti allows the clean-line ideal of real-world sketches to be main-

Mouse move

Traverse Enumerate all vertices and edges within radius d of p_c. Among them, find the new nearest point p_n to the drag point p_d (p_n may lie on an edge) and the shortest path between p_c and p_n within that radius d (such as in Figures 3-7 and 3-9).

Split Edge If p_n lies on an edge and is more than one pixel from any adjacent vertex, split the edge at that point (as in Figure 3-6); otherwise snap p_n to the closest vertex.

Project Project the split vertex and traversed vertices from the two previous steps (if any exist) onto the line segment between p_c and p_d (also shown in Figure 3-6).

Cleanup If any projected vertices are now less than one pixel from a neighbor, merge the two vertices. If any edges of those vertices are roughly parallel (defined in section 3.3.3), merge the two edges.

Extend If p_n is an endpoint (valence 1), create a new edge from p_n to p_d (as in Figure 3-6) and set p_n equal to p_d.

Fork If $p_n == p_c$, p_n is of valence 2 or more, and p_d is greater than d pixels away from p_n, create a new edge from p_n to p_d (as in Figure 3-11) and set p_n equal to p_d.

Intersect If a new edge was created, search all existing curves for an intersection. If one is found, split the intersecting line segments and merge the split vertices (as in Figure 3-10).

Update Set p_c equal to p_n.

Figure 3-15: Mouse-move event handling

tained simply by turning off the visualization of previous strokes, despite the artist's many overdraw attempts.

3.5 Analysis

Design Implications

The initial results are encouraging: for sketching real-life scenes, particularly for novice artists, the approach's utility is clear. The utility for architects, however, is less clear. The deep issue, which we face throughout this thesis, is integrating this

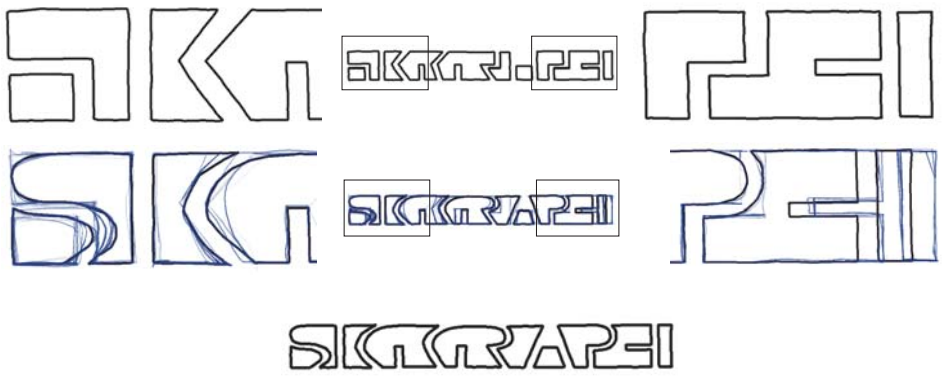

Figure 3-16: Designing a 'SIGGRAPH' logo. The top image shows the original blocking-out of the figures; the middle shows the final version with curved letters; the bottom is the logo with the history visualization removed. Zoomed-in versions are shown in Figure 3-17.

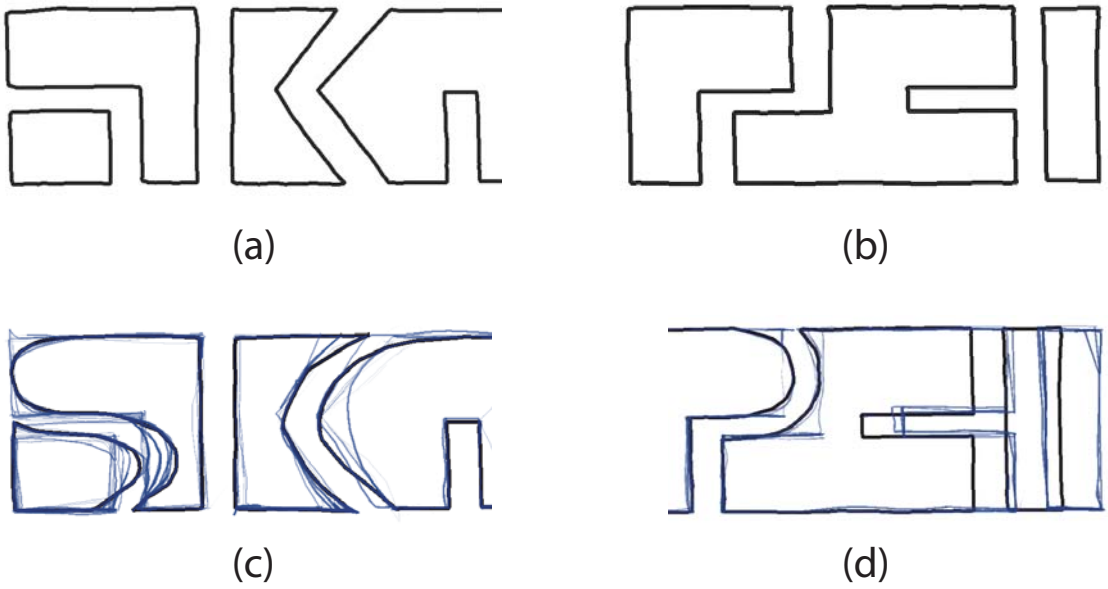

Figure 3-17: Zoomed-in versions of the boxed areas in Figure 3-16.

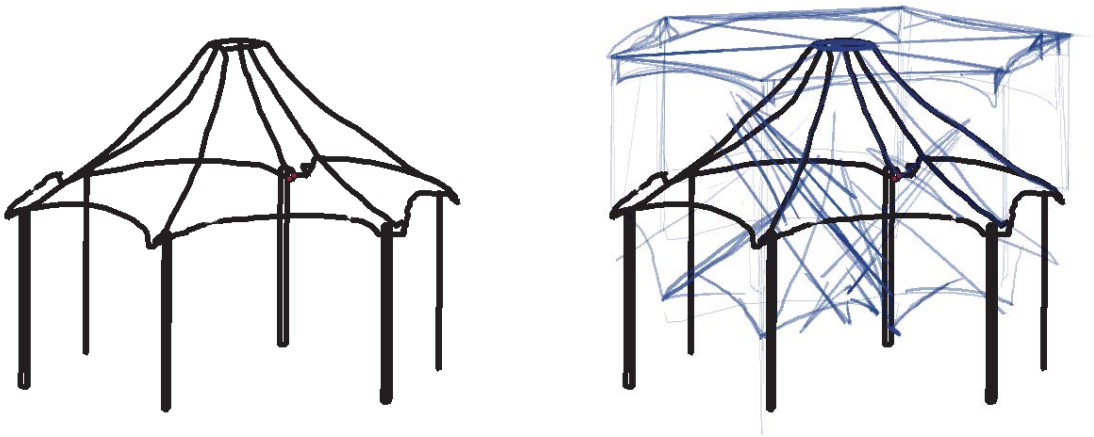

Figure 3-18: A 3-D tent edited with pentimenti. The tent is shown on the left, with history on the right. The history shows how the top part of the tent was originally drawn and edited on a plane before being brought down and placed in 3-D. It also shows the extensive work currently required to edit 3-D curves with our interface because projecting onto a plane and editing gives good results from the current view but not from others.

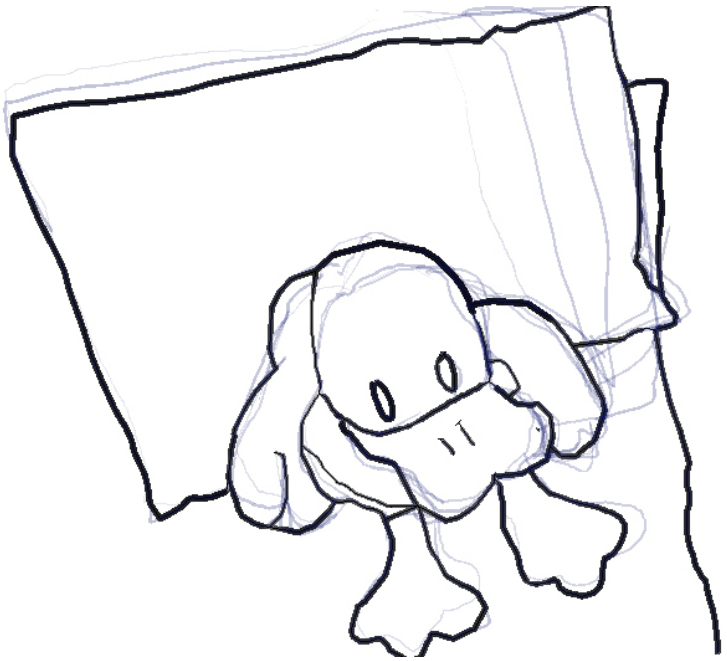

Figure 3-19: Sketch of duck, with pillow, made by a novice artist. Note the several strokes used to re-size the pillow, as well as the many strokes fixing the proportions of the foreshortened duck's head.

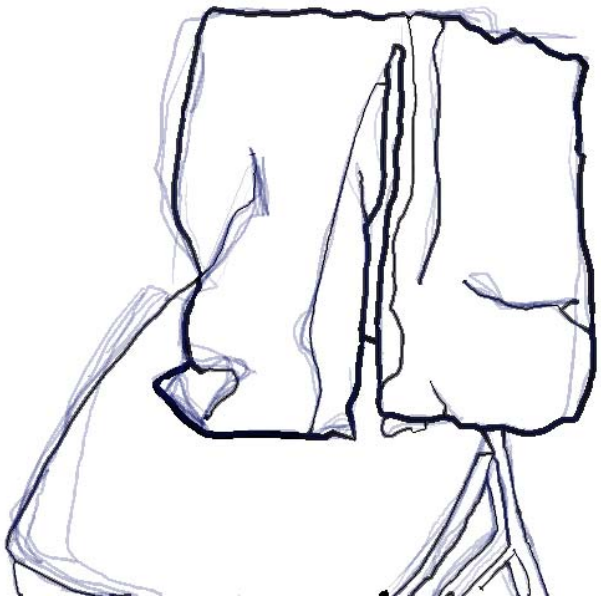

Figure 3-20: Sketch of pants on a chair, made by a novice artist. Note how the interface enabled the artist to try many curves in order to find the right one.

technique into design practice. Existing design practice is based on hundreds of years of tradition; further, each individual designer spends years picking and choosing from this tradition in developing their own practice. To be appropriated by architects, pentimenti needs this same integrative effort.

An issue specific to pentimenti, however, is whether it solves a true architectural design problem. That is, pentimenti seem most effective in exploring a series of small changes. This sort of refinement seems more appropriate to later stages of design. One response would be to better adapt it to early stages of design, perhaps by producing more feedback beyond literally changing the curve. Some designers, such as Frank Gehry, like to model with unusual materials such as velvet because of the helpful constraints imposed by its physical makeup — helpful both as an approximation of constraints imposed by real- world building materials, and as a way to constrain the range of exploration to an interesting set of shapes. Computers can simulate real-world materials, or provide some non-physically-based behavior; perhaps, when the user draws a new curve, if the sketch reacted or "pushed back" in the way real-world

materials do, that could be a useful design aid above and beyond traditional sketching practice. Another aid to early design would be to extrapolate from a series of changes made with pentimenti – presenting a range of novel forms, in effect exploring the space the user has implicitly defined by overdrawing.

A second response would be to consider using pentimenti in these later design stages. Again the idea of editing a constrained representation that "pushes back" could prove useful; here, the user could freehand sketch small changes to a clean-line drawing; when the system applies the user's changes to the source geometry, they are cleaned up into straight lines and precise angles before replacement.

Lower-level Improvements

Other interesting extensions include allowing the user to edit the sweep of a curve with freehand strokes while maintaining its detail, perhaps by using a multiresolution representation as in Finkelstein and Salesin[26]. Using a simple B-spline representation would also provide additional smoothness for free.

We do not currently exploit our system's ability to maintain attributes of replaced curve segments. Perhaps by maintaining depth we could edit 3-D curves, including surfaces by editing their silhouettes. Editing curves in 3-D through overdraw is not as easy as it first appears, however. Fleisch et al. [27] extended Baudel's algorithm to a virtual reality setup with 3-D input devices and found that it was helpful to limit the resulting changes to a *2-D* plane, while Karpenko et al. [41] extended it to a multi-view interface (redrawing the curve from several views) and discovered that while the concept of multi-view sketching is intuitive, in practice it is quite difficult.

Although the history strokes are currently inaccessible, it might be useful to let the user revive particular strokes, perhaps assembling a new stroke out of the pieces of old ones or making snapshots of particular moments in time, allowing the user to pursue edits on any of them.

Several reviewers have suggested a more gradual editing approach, where the latest stroke edits are averaged with earlier edits, albeit with later strokes having higher weights. This would keep the curve smooth, at the cost of making the user

draw a specific curve repeatedly until the actual curve replicates what the user is drawing.

Chapter 4

Exploring Sketched Form by Dragging Control Points

In the previous chapter we described a technique for editing sketches that is inspired by sketching practice, where the user creates a series of discrete alternatives through freehand drawing. In this chapter we describe another method to edit sketches that is based on the smooth point-dragging provided by digital tools. Sketching is effective because of its bottom-up construction method, where strokes coalesce to form larger strokes, and because a designer can target descriptive effort where needed, building up details in ways difficult to capture with higher-level shapes. This approach also promotes ambiguities in how the strokes combine, stirring new design ideas. In contrast, digital tools rely on high-level primitives with relatively few parameters which can be exhaustively explored. In order to bring the advantages of smooth exploration to sketching, where the degrees of freedom in these low-level strokes are far too numerous to control, we introduce a proxy construction interface where the user creates desired control points and the system constructs a skeleton with those points. Dragging the skeleton's control points cause corresponding changes to the sketch strokes.

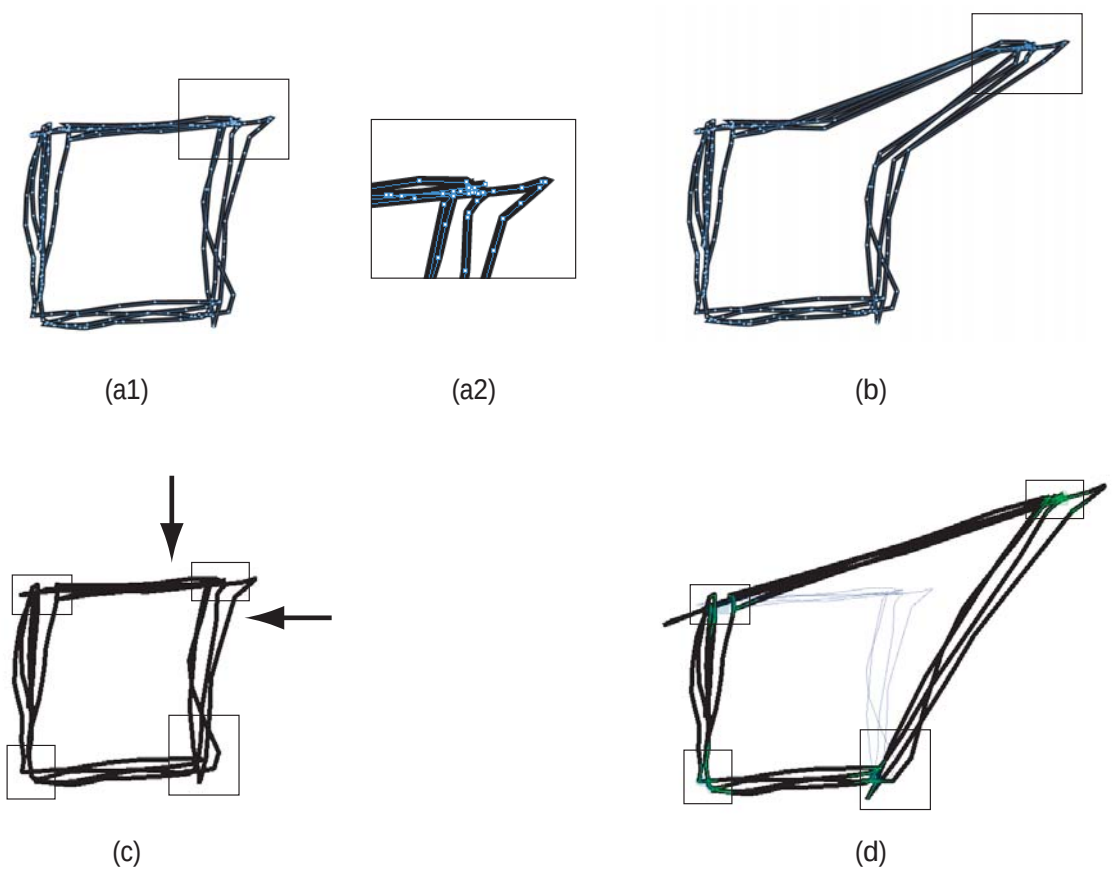

Figure 4-1: The user has drawn and overdrawn a box with a single stroke. We first show direct manipulation of control points(a1) with a commercial package. When we move a group of control points in the corner (a2), the rest of the control points stay put, resulting in (b). With our interface, the user defines control points (the boxes in (c)) and then can drag them (d), resulting in intuitive motion. To mimic the deformation produced by our system with standard tools, a determined user would have to manually translate a large number of point groups, or perhaps attempt to use some mix of translation, rotation, and scaling of various point selections — a time-consuming task that precludes exploration of different drag positions. The user of our system could mimic the deformation in (b) simply by making two additional selections on the edges where the points should stay fixed (shown as arrows in (c)).
.

4.1 Motivation

In the previous chapter we discussed sketching-based techniques for editing form. Those techniques support an iterative design process where the designer sketches a form, evaluates it — finding new possibilities — and then creates a new form based on those perceived possibilities. The key creative element of this process is not the specification (the freehand sketching) but the *perception* — seeing new ideas suggested by the form. Digital tools support a different, more proactive design process: the designer drags control points around, evaluating not a single option at a time but a continuum of forms. The proxy approach described in this chapter brings this continuous technique to sketches, leveraging their existing speed and expressiveness. In later chapters, we will describe still other novel ways to visualize and "see into" sketches.

4.2 Related Work

In this section we first describe editing in existing sketching practice, then smooth editing techniques in digital tools based on point-dragging. We are most interested in techniques where the user drags points on proxy skeletons which then deform the model. We discuss techniques for creating such proxies. The contribution we describe in this chapter is a simple user interface to rapidly construct such proxies, whose control points then can be dragged by the user to explore form.

4.2.1 Existing Sketch Practice

Standard pencil and paper provide a powerful means for discrete exploration of form. Designers can redraw a new sketch, or trace over an old one; rarely a direct copying operation, both allow the designer to propagate only desirable properties in a flexible way. Designers can also overwrite with new strokes or erase to destroy old strokes. This strategy is effective for large changes; the digital methods we will describe next are more effective for exploring of a space of forms using small changes.

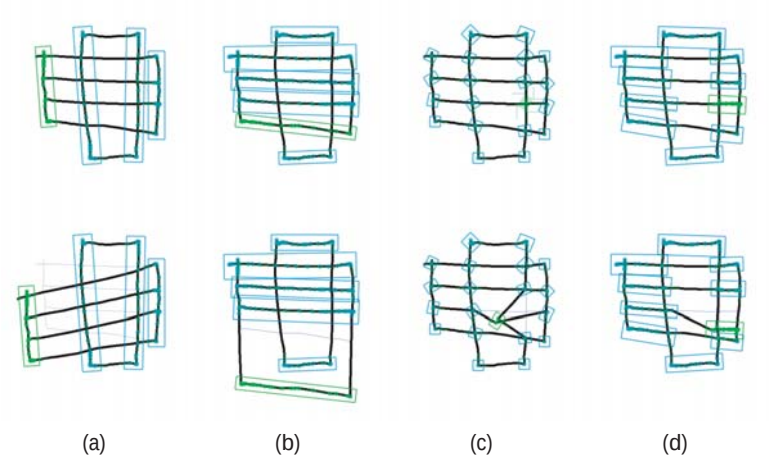

Figure 4-2: This sketch has at least four decompositions; with our approach, the user can specify any of them (top row) and deform the sketch accordingly (bottom row).

4.2.2 Point-dragging

We describe digital tools for point dragging in Section 2.2. With these interfaces, a large space of possible forms can be explored with a single mouse drag. Although effective for higher-level primitives, this approach does not scale well to sketches composed of low-level polylines, which have many degrees of freedom (see Figure 4-1).

4.2.3 Proxy Point-Dragging

With proxy-based methods, the user creates a simpler proxy geometry which acts as a scaffolding for the model. Modifying the proxy causes the actual model geometry to similarly deform. These methods include multi-resolution modeling, free-form deformation, and WIRES.

In multi-resolution modeling, the proxy is not a separately defined geometry; instead, the system stores the model as a combination of low- and high-resolution components, and the user can choose to deform control points corresponding to any level, as in Zorin et al. [84]. This method does not allow the user to specify what features to control.

In free-form deformation [62], the system re-expresses the actual model coordinates

relative to proxy coordinates. The form of this proxy differs between modeling and animation applications. In modeling, the proxy is typically a 3-D grid of points. In animation, the proxy is usually a skeleton of connected linear segments or "bones," and often the model coordinates are expressed as linear combinations of points relative to the bones (see Lewis et al.[43]). An added complication of animation skeletons is that the bones need to maintain a constant length; control points (such as hand or feet position) thus cannot be directly moved, but an "inverse kinematic" or "IK" solution must first be derived. This solution is a subject of ongoing research (one recent paper is Grochow et al.[33]). Our approach does not enforce this length constraint, resulting in a simpler internal formulation, but requiring the user to enforce this (or any other) constraint manually. Precise enforcement of constraints, however, is less important in the exploratory phases of design.

The WIRES interface [69] is different from free-form deformation in its internals, but based on a similar idea. Unlike other modeling interfaces, however, wires are meant to be applied locally — used at the scale of an individual muscle bulge, for example.

The advantage of the FFD and WIRES approaches is their independence from the intrinsic control points of the geometry. However, the construction of the proxy geometry remains indirect and heavyweight; additionally, these techniques do not "play well with others"; that is, other deformation techniques cannot be used in concert with them, as they typically will invalidate the mapping of proxy to source geometry. Additionally, they do not layer; architects often create forms which can be decomposed (and thus edited) in several meaningful ways (as in Figure 4-2), but these techniques support only one such decomposition — again, manipulating the form with one skeleton invalidates the other skeletons. Our approach does not have these limitations.

4.2.4 Proxy Construction

Automatic approaches to proxy construction involve creating a skeleton from the medial axis (see Yoshizawa et al.[82] and Liu et al.[46]). Multi-resolution approaches

also automatically extract the different resolution control meshes as described in the previous section. In our case, we need the user to tell us which features are important to control.

With manual approaches, the user constructs the proxy skeleton just like any other 3-D geometry; then, so-called ''skinning" methods initially place model points in appropriate skeleton subspaces (see Lewis et al. [43]). Typically model points are re-expressed as weighted combinations of points in multiple subspaces. Finally, the user can tweak that mapping and add a variety of additional local deformation controls (Maya[3] is a commercial package with many such controls).

Our approach has both manual and automatic components. The user manually specifies desired control points; the system creates the proxy automatically using stroke connectivity; the user can then fix up the proxy if the stroke connectivity does not give the desired result.

4.3 Exposition

In this section we describe the algorithms underlaying our system. We assume that the user has already drawn a sketch with polyline strokes. The user then uses our interface to construct a "proxy" geometry, which in this case is a simpler edge-vertex graph, over the sketch. This proxy is automatically connected to the actual model polylines such that when the user deforms the proxy, the model is similarly deformed.

In the default case, no vertices in the underlying stroke representation will have valence more than two, even though perceptually the valence is typically higher. This is because standard stroke input is composed of disconnected polylines — a single polyline per stroke; see Figure 4-3. If other geometry manipulation tools are available (such as a join operation), higher-valence vertices in the internal representation are certainly possible, and we describe their handling later.

We now describe the interface and algorithms for constructing the proxy, and for deforming model vertices as the proxy deforms.

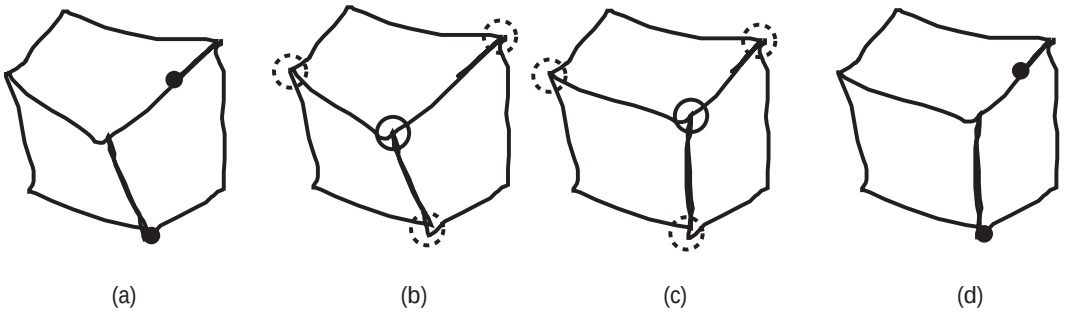

Figure 4-3: In this sketch the user has drawn the cube with one stroke. Although perceptually there are four trivalent vertices, in the internal stroke representation there is simply a single polyline, with two endpoints shown(a). Although the user may circle these perceived vertices with our tool(b) and drag them (c), the internal representation remains a single polyline (d).

4.3.1 Proxy Construction

The user selects desired control points by simply selecting groups of polyline vertices; each group becomes a control point in the proxy (see Figure 4-4). We shall call each group a "proxy vertex." Proxy edges are constructed from these proxy vertices by transferring the existing connectedness of their component model vertices. That is, for every path along the actual polyline geometry that connects pairs of model vertices belonging to different proxy vertices, an associated edge in the skeleton is constructed and bound to the interior vertices on the path. Note that with this construction, the skeleton topology potentially changes every time a new proxy vertex is constructed or the stroke topology is changed by any other modeling operation such as erasing an edge or merging two vertices. In our implementation, the proxy edges are discovered lazily and locally; that is, only when a proxy vertex is actually moved does it search for adjacent proxy vertices and discover appropriate proxy edges.

Gregory et al.[32] uses a similar interface to our own in order to efficiently specify corresponding edge chains for a morphing application; that is, the user specifies chain endpoints and the system uses connectedness information to find the interior chain edges. Our approach is more general, as it allows multiple edge chains to exist between a given pair of proxy vertices, and for proxy vertices to effectively unite unconnected model vertices. Our geometric representation is also somewhat different: we are

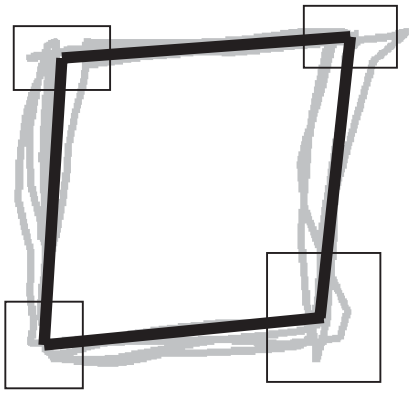

Figure 4-4: Selecting the vertices (boxes) causes the system to construct proxy edges (dark lines). These edges were inferred from the stroke connectivity in the original sketch geometry.

working with sketched polylines, while they are working with surface boundaries.

We discuss special cases handled by the proxy construction in Section 4.6. The nature of the binding of proxy edges to model vertices, and their subsequent deformations, are described in the next section.

4.3.2 Proxy-based Deformation

With our interface, the user drags a widget centered around the proxy vertex (located at the centroid of the proxy vertex's associated model vertices) and translates it in 3-D. The movement of the proxy vertices defines a movement in the proxy edges, which then move their associated polyline vertices (as shown in Figure 4-1). This movement could be accomplished in several ways; our prototype implements two (see Figure 4-5).

One method, essentially equivalent to skeleton-space deformation described in Lewis et al.[43], is to define all the polyline points relative to a coordinate system

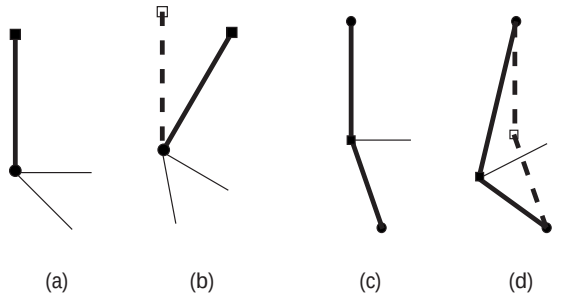

Figure 4-5: In one of our deformation methods, two arbitrary vectors (thin lines) are generated orthogonal to the proxy edge (thick line), creating a 3-D coordinate basis for stroke points(a). When user drags a control point (the square endpoint), the rotation of the proxy edge is propagated to the other basis vectors(b), and the new basis applied to associated polyline vertices. In our other method, the two outgoing edges from the dragged control point (the square point in (c)) form two directions of the basis; a third basis vector (thin line) is generated as the vector orthogonal to those two. After the user drags the control point, we recompute the new normal vector, defining the transformed coordinate system for stroke vertex positions(d).

that contains the original proxy edge as an axis. We then rotate the proxy edge to its new position, and find the corresponding new world-space positions. This is a rigid transformation. Lewis et al. note the undesirable effects of such transformations when bending an elbow or twisting an arm; in our experience these artifacts are less problematic, probably because our models are composed of lines instead of collapsible surfaces or volumes. More problematic is loss of continuity caused by ignoring curvature across joint boundaries (see Figure 4-6).

Our second method preserves curvature better, but currently only can be applied to for proxy vertices with two outgoing edges that are not collinear. In this method, a transformation is composed that essentially warps the two outgoing proxy edges from their original direction to their new direction; that is, it expresses the original polyline points in a basis composed of the original two proxy edge directions and their normal, then converts the points back to world space with a different basis, this one composed of the new proxy edge directions and their new normal. This is generally a shear operation. In practice the results look very similar to the skeleton-space deformation, successfully maintaining sharp edges or curved lines, except that the collapsing problem is eliminated — see Figure 4-6. The difference is that one

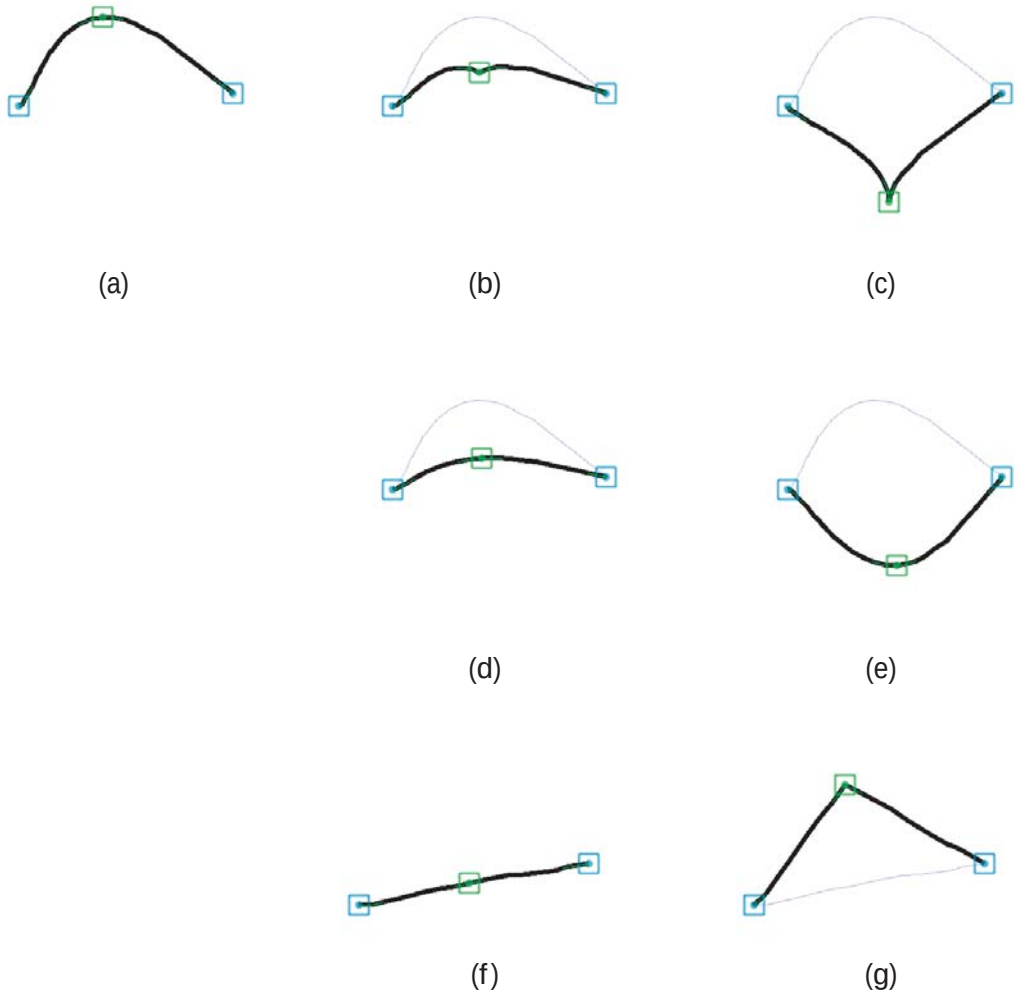

Figure 4-6: The user drags the middle vertex in (a). With our first deformation method, as the user drags out, a new kink in the curve is created (b and c). With our second deformation method, the curve remains smooth (d and e). However, the second method cannot handle nearly linear situations as in (f). Our first method, in such a situation, creates the corner as in (g).

introduces perceptual features (a corner in the line), and one does not. Both can be useful.

4.4 Results

The most immediate application of this algorithm is to allow the user to simply drag proxy vertices around to edit a model. We show 2-D examples in Figures 4-1 and 4-2, and a 3-D example with sharp edges in Figure 4-7. The technique also appears to work well for curves, particularly the second deformation algorithm as shown in Figure 4-6.

In practice, in order to avoid visual artifacts where edge vertices move rigidly with a corner vertex, having a minimal number of model vertices within a given proxy vertex are desirable. Helping the user make these selections has thus proven key to getting good-looking results. Our system has several selection modes, including basic "add" and "subtract" functionality. A "trace" mode allows the user to make selections along the original strokes. A more unique mode, useful for selecting point features, helps the user select minimally by taking all the vertices within a given lasso and sorting them into connected subsets. One vertex from each subset, the closest to the lasso center, is selected (see Figure 4-8).

We have implemented a simple "layering" feature, where each layer holds a set of proxy vertices. The user can build up multiple sets of key vertices to manipulate the sketch at different scales or with overlapping shapes, as shown in Figure 4-2.

A further application of this technique, 3-D reconstruction of sketched drawings, will be discussed in Chapters 5 and 6.

4.5 Analysis

Design Implications

Our system has proven quite successful and intuitive for simple direct manipulation of sketches. However, as with the pentimenti technique of the previous chapter, our

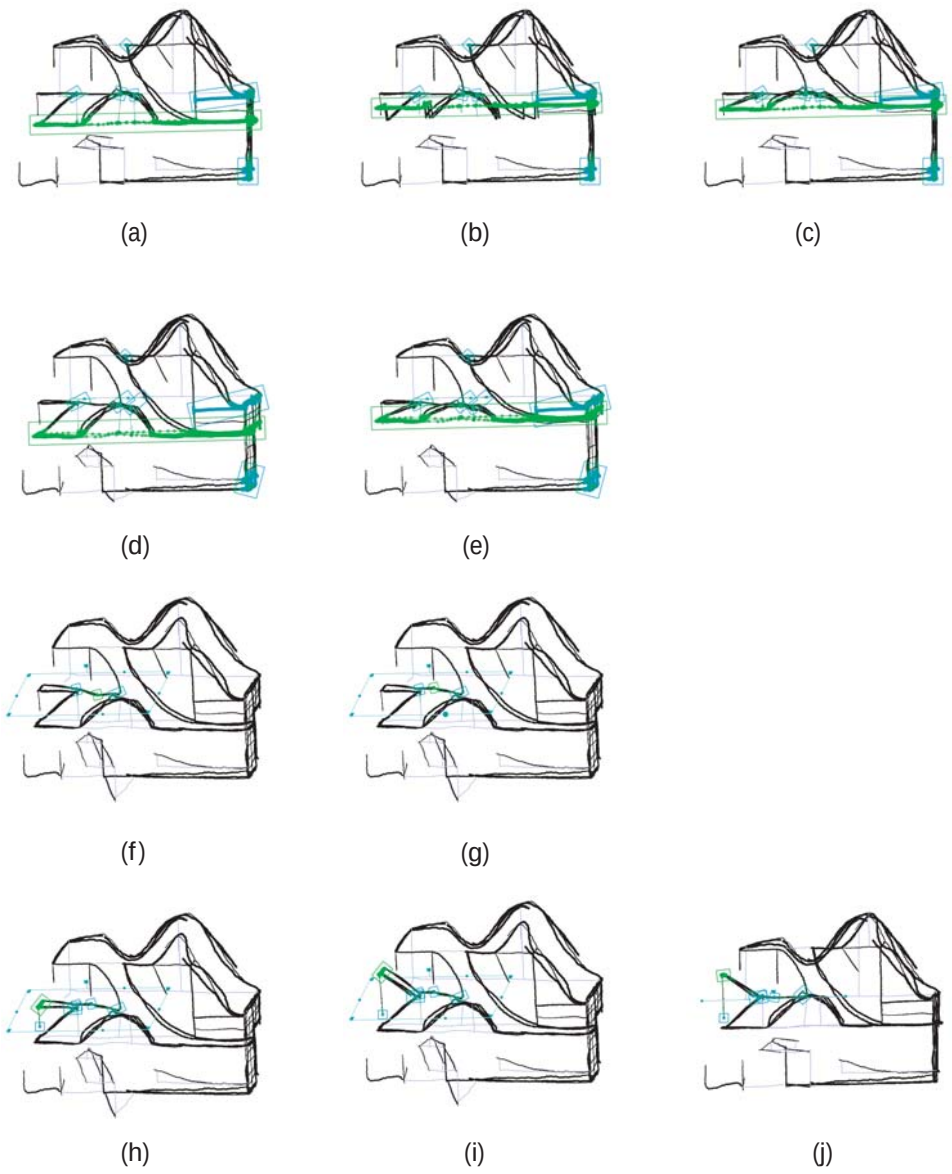

Figure 4-7: The user has made several selections (a) from a side view of the 3-D model described in Figure 5-8. Dragging the central horizontal selection using traditional techniques would result in an undesirable result (b). Using our technique (c), the curves have been naturally scaled down and straight lines maintained. We show the same deformation from a slightly oblique camera angle in (d) and (e), demonstrating that it is in full 3-D. In (f), we show a finer-scale deformation where the user is dragging a point along the ground plane (shown as the rectangle in the middle-left of the image), resulting in (g). The user then plans to raise an edge along the plane's normal, first by making new selections (h), then by dragging the edge (i). We show the result from another camera view in (j).

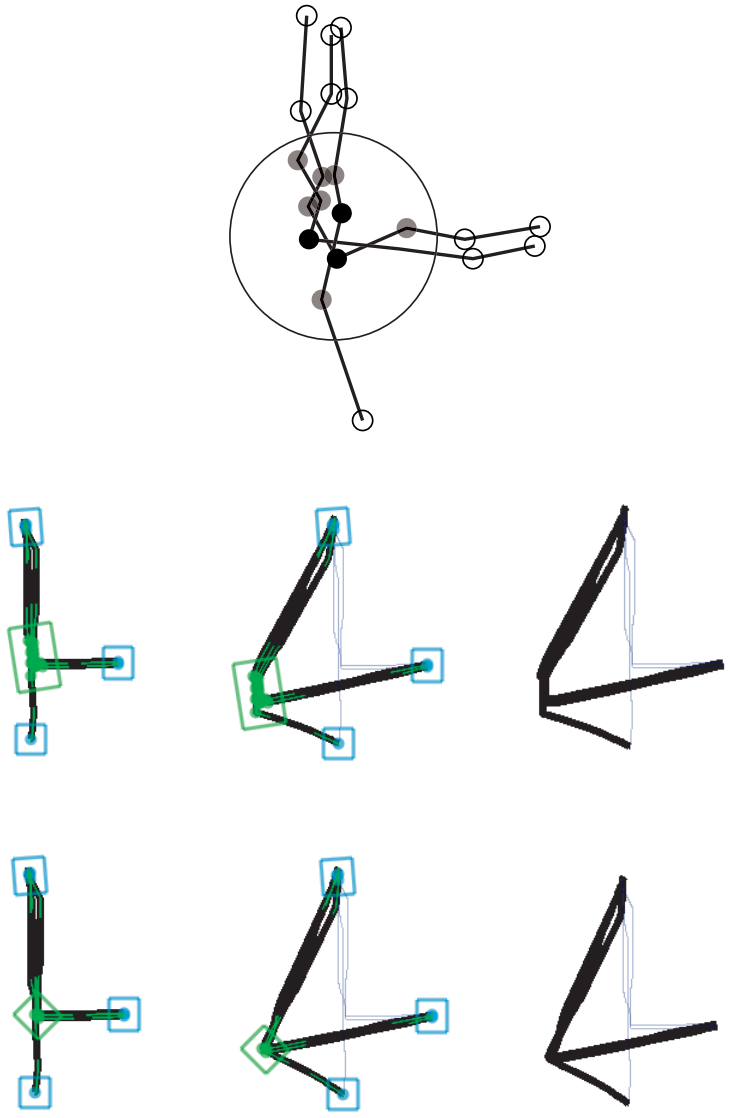

Figure 4-8: The user selects a corner feature (top). The system groups the vertices within the selection (shown in grey) into sets of connected vertices. From each set, the system chooses the one closest to the selection center. If the system had instead chosen all vertices inside the circle(middle), then those vertices would have been translated rigidly together when user drags them, causing visual artifacts. Using a selection made with this algorithm(bottom), the visual artifacts are gone.

technique is most effective for exploring small changes, which is less appropriate for early design. One strategy to explore larger changes would involve a more "hands-off" interface, where the system auto-selects key points and shows some variations based on moving those key points around; the user could then choose desirable options to pursue. This would effectively execute many local changes at once, and present a large number of discrete alternatives from the continuous space of variations. Another strategy would be to present higher-level primitives than edges and vertices (such as planes or cuboid volumes) for the user to manipulate. A similar capability is available with our existing prototype when the user makes selections from a degenerate view such as the plan or section, selecting walls and manipulating them as points. Even so, the 2-D nature of the interface would preclude exploring a large space of form with one mouse drag.

Lower-level Improvements

Possible extensions of the existing method to handle lower-level concerns include:

Feature identification Identifying features is a time-consuming task; while it cannot be completely automated (due to the ambiguity of drawings and number of useful alternatives), it could be at least computer-*aided*. A large and active body of computer vision research (one recent paper is Mikolajczyk and Schmid[50]) attempts to automatically find image features (also known as "descriptors," "interest points," or "corners") that persist between different views. These algorithms — which are generally applied to real-world images — might be of great use with sketches. We also might be able to exploit stroke-based techniques such as Shpitalni and Lipson[65]; however, automatically finding corners and, particularly, identifying which 2-D intersections are truly 3-D intersections is a chicken-and-egg problem: it generally requires knowing the 3-D structure — which requires already having the corners and intersections identified.

With larger sketches, it is often very time-consuming to "lock down" the appropriate features; typically the user drags a proxy vertex, finds that it causes

unexpected and undesired motion, undoes the drag, circles an additional feature, and tries again. This workflow could be made more efficient, perhaps with a better visualization of the consequences of moving a proxy vertex.

Proxy vertex positioning We would also like to generalize our approach to allow arbitrary positioning of the proxies relative to the original model, rather than requiring proxy vertices to lie at the centroid of their respective model vertices. This is particularly useful when drawn edges stop short of the perceived vertex, leaving no convenient model vertices nearby. We would also like to allow arbitrary correspondence between proxy and model — this is similar to the "skinning" task in animation (described by Lewis et al. [43]) and is also required for the deformation transfer method of Sumner and Popović [71]. However, our technique has the added constraint that any user tweaking of the skeleton would have to be integrated with our automatic proxy construction method, so that the system could automatically adapt the skeleton if the user modifies the model with other tools.

Surfaces We would like to extend proxies beyond wireframes — to surfaces and drawings of shadows and materials. This may require different selection techniques, or a different kind of proxy (perhaps 2-D area-based instead of 1-D line-based).

Images We would like to extend our feature-based construction approach to the image-based object framework of Barrett and Cheney[8] or the more automated version of Saund [61]. Directly working with images would allow architects to directly work with scanned pencil and paper (or napkin!) sketches.

Complexity With complex sketches, there are often many T junctions which are more difficult to control, such as in Figure 4-9. Achieving the correct behavior in these cases will probably require exposing additional options to the user.

A similar problem when scaling to larger sketches is the visualization of the proxy itself, which can often overwhelm the sketch (see Figure 4-2). More

subtle visual indicators would help the designer keep the focus on the sketch and not its deformation controls.

2-D/3-D In a 3-D setting, our deformation formulation can give unexpected results. The user, when moving a proxy vertex in a direction normal to the film plane, often expects that the interior points will stay in the same 2-D location, which they generally do not. Taking the camera into account in the deformation might provide more intuitive control.

Deformation We described two deformation algorithms; we need some way to determine when the curve-preserving one won't work. When either is applicable, the user should be presented with the option of specifying whether a sharp or smooth feature is desired.

4.6 Appendix: Special Cases in Proxy Construction

4.6.1 High-Valence Vertices

We now consider vertices with valences higher than two, such as in a T formation (see Figure 4-9). Although such vertices are perceptually common, they would be non-existent in the underlying representation if the user simply draws with a polyline tool. However, other modeling tools could create such vertices, as will our "bridge sets" described below. In this situation, an intermediate vertex could lie on several proxy edges at the same time. One of our early "push" implementations handled this by computing the potential destination for each path separately, then placing the vertex at the average of those locations; however, we found the results generally undesirable and unpredictable, as usually only one of the paths was the perceptually desirable one. Our current implementation simply treats all high-valence vertices as proxy vertices.

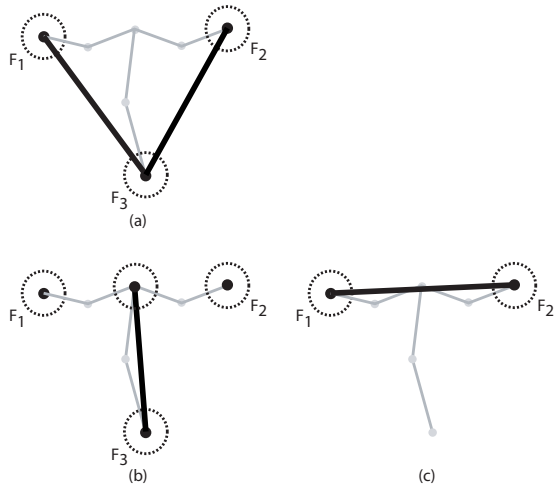

Figure 4-9: The problem of high-valence vertices. Selecting just the endpoints of a "T", and using the averaging-of-all-paths scheme described below, results in the inferred skeleton shown in (a), which does not represent the geometry well. The user would probably like some combination of (b) and (c), but no single proxy skeleton can get both at the same time. Our current implementation treats high-valence vertices as proxy vertices, interpreting situation (a) as equivalent to (b).

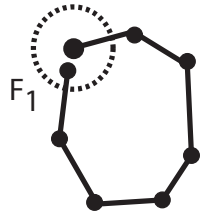

Figure 4-10: The special case of a proxy edge connecting a proxy vertex to itself.

4.6.2 Loops

Note that a proxy edge could potentially connect a proxy vertex to itself (see Figure 4-10). The "push" step of our system handles this special case by rigidly moving the intermediate stroke vertices with the proxy vertex.

4.6.3 Endpoints

We can think of five plausible strategies to deal with endpoints that are topologically connected to a proxy vertices, but are not themselves members of proxy vertices (see Figure 4-11); they are described in the figure caption.

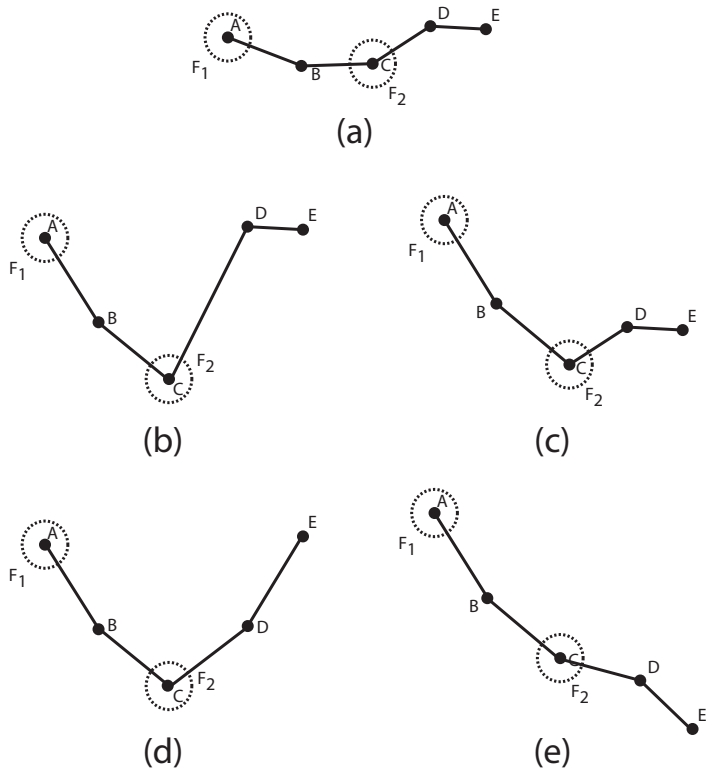

Figure 4-11: The special case of endpoints. In (a) proxy vertex F_2 is topologically linked to endpoint E. Upon motion of F_2, what to do with vertices D and E? We could leave them alone (b), translate them rigidly with the proxy vertex (c), consider the endpoint to be a proxy vertex itself (d), or treat it as an extrapolation (e) of some other incoming proxy edge of F_2; our implementation uses the most angularly consistent proxy edge of F_2, in this case A-C. We believe (e) to be most useful, but all are plausible.

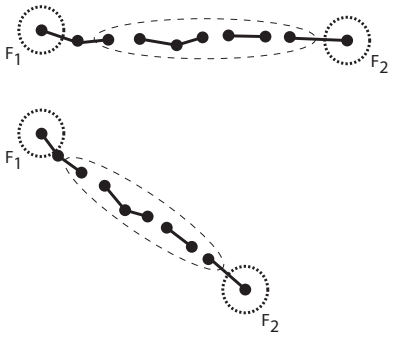

Figure 4-12: Bridge sets for topological gaps. Top: the user-defined bridge set bridges the gaps in the dotted line, effectively connecting proxy vertices F_1 and F_2. Bottom: after the user has moved F_2. Due to the bridge sets, the perceptually intermediate vertices between F_1 and F_2 are deformed appropriately.

4.6.4 Topological Gaps

Proxy vertices have a dual role: they specify endpoints of proxy edges and, implicitly, bridge topological gaps in the strokes (as in bridging the gap between strokes of the dotted line in Figure 4-12). Sometimes, such as with dotted lines, it is desirable to adjust the stroke connectivity (i.e. connect the dots) for deformation purposes without changing the actual model connectivity. We thus introduce the notion of a "bridge set," which we define as set of vertices that our algorithm will consider connected when determining which proxy vertices are connected, but otherwise is ignored.

We have two implementations of a bridge algorithm. The first, which works when precise ordering of vertices along the paths is not important, treats the bridge set as a single vertex. When any vertex in a bridge set is encountered, the entire set is added to the path, and all edges between bridge vertices and out-of-bridge vertices are traversed. The second maintains an ordering. In this algorithm, all the vertices in a bridge set are divided into connected subsets. Akin to Kruskal's algorithm for constructing a minimum spanning tree, we create a virtual edge connecting the subsets with the closest pair of vertices, and recurse until there is only one subset.[1] Searching

[1]To avoid the ambiguities and confusion of high-valence vertices as much as possible, we do not connect subsets using vertices of valence 2 or more unless there is no other alternative.

for paths between proxy vertices then uses these virtual edges. In practice, we have found the behavior of the second one easier to understand and predict, particularly when bridge sets and sets of proxy vertices overlap.

Chapter 5

3-D Reconstruction from a Single Sketch

In this chapter we present a system to construct a 3-D model directly from a sketch, relying on a user-guided annotation system to disambiguate the reconstruction. The result is a new kind of sketch-like representation with the same stroke composition and similar appearance to the original. This "3-D sketch" retains a similar ambiguity and suggestiveness as the 2-D original but now can be viewed and manipulated in full 3-D. The user first simplifies the sketch with our proxy deformation system from Chapter 4; this simplifies the problem to finding appropriate 3-D coordinates for the proxy vertices. Our approach then helps the user exploit axis-aligned structure in the proxy geometry. The user aids the system in identifying axially related features, in some cases sketching additional axial lines to connect features; the system then uses this information to reconstruct the full 3-D coordinates of the proxy vertices, which our proxy technique uses to deform the rest of the strokes, bringing the full sketch into 3-D. In the next chapter we extend this technique to work with two input sketches, which will change the problem from one of depth specification to one of correspondence.

5.1 Related Work

We assume that the user has made an orthographic drawing with polyline strokes depicting a 3-D scene. We first review existing work in transforming such drawings into 3-D.

A manual approach would be to start from scratch with the 3-D geometry creation and manipulation tools described in Chapter 2. In practice, this is the standard practice for creating a 3-D model from a sketch. With this approach, 2-D sketches can be leveraged as a background to trace over. While time-consuming, this approach is also extremely flexible, able to handle arbitrarily inaccurate and ambiguous sketches.

Automatic and semi-automatic methods are reviewed in Section 2.3. Our main contribution is a photogrammetry-like approach to reconstructing sketches. Our approach has several differences from existing algorithms. First, photogrammetry algorithms use image input; our interface, concentrating on line drawings, uses a stroke-based representation. With drawings made on digital tablets, stroke information is trivially available; they can also be extracted from images (albeit again a user-guided process) with edge-detection filters such as Canny[13] or higher-level user interfaces such as Simhon and Dudek[68]. Second, in photogrammetry, the user creates separate geometry, either tracing 2-D geometry over the image or assembling higher-level 3-D geometric primitives as in Façade. Our approach exploits the existing sketch, using our user interface from Chapter 4 to turn an arbitrary sea of strokes into a simple proxy edge-vertex graph (i.e. into clean contours with sparse control points); that graph will be our source 2-D geometry. In our examples, the proxy geometry typically is one to two orders of magnitude simpler than the source stroke geometry; the ratio is driven even lower by denser stroke sampling or by more use of overdraw by the user, both of which cause more stroke vertices to be mapped to a given proxy edge. Once we determine 3-D locations for points in that graph, we can then use the proxy to *transform* the original sketch into 3-D. Note that, in our case, moving vertices must carry along messy sketch strokes rather than simple lines or planes. Third, the 2-D location of vertices might also be modified by the reconstruction if

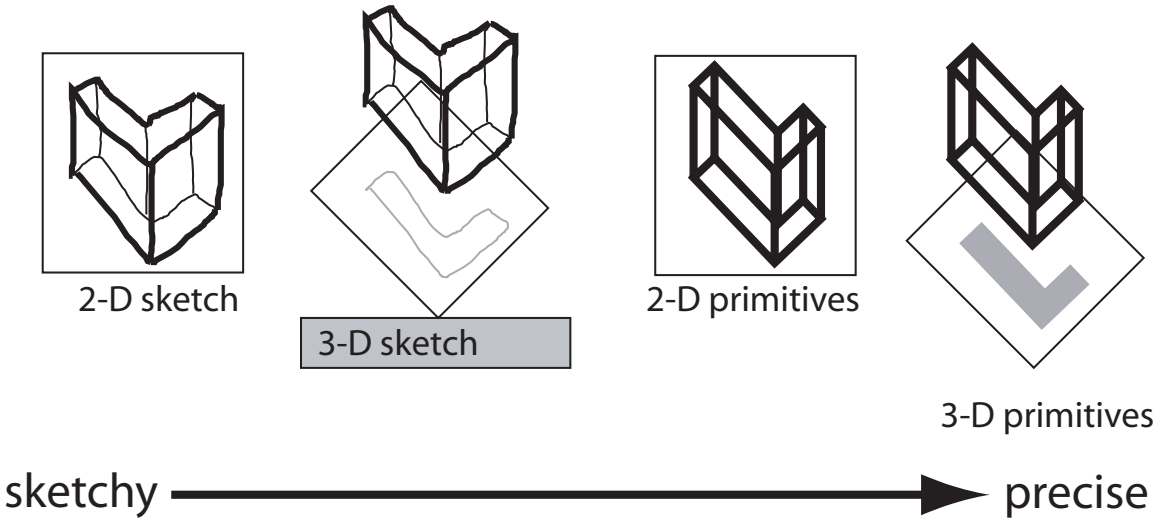

Figure 5-1: 3-D sketches in the context of other representations: more definite than a 2-D sketch of a 3-D object but more evocative than a clean-line drawing or 3-D model.

the sketch was not a precise projection of the desired shape. Such movement is not handled by shape-from-contours or photogrammetry works with the exception of Ros and Thomas[60], which moves 2-D contour vertices as a preprocessing step in order to support a particular type of planar reconstruction.

Our second contribution is a simpler annotation interface than previous works, which presented a variety of constraint types to the user. We use a single type of constraint — giving an edge a known 3-D direction — and we leverage freehand sketching to annotate the drawing such that the user can describe any spatial relation between proxy vertices with a small number of directions. This constraint system is simple, intuitive, and especially appropriate to architectural models, which have few 3-D directions.

We place this new representation relative to existing representations in Figure 5-1. A 3-D sketch is more definite than a 2-D sketch of a 3-D object, but more evocative than a clean-line drawing or 3-D model.

5.2 Exposition

Our system takes a 2-D edge-vertex graph as input; this could be a set of overlapping polylines, sketchy or clean-line, whose intersections may or may not represented in the geometry. The output will be the same graph, with vertices now in an appropriate 3-D position. Our reconstruction approach leverages the idea that knowing 3-D stroke directions is sufficient to disambiguate 3-D position up to a global translation. While specification of general 3-D direction is difficult, we simplify the problem by choosing a key orthonormal basis frame, and expressing all other directions relative to that frame. This is particularly effective when the form contains only axis-aligned edges, or more generally very few 3-D directions — a condition which tends to hold for architectural buildings. The system automatically labels edges according to the closest axis in the given basis frame; the user can then override this classification, possibly drawing new axial strokes to disambiguate certain cases.

The user has two primary tasks in our interface: first, to circle key feature points with our proxy interface from Chapter 4; second, to identify axial relations between those proxy vertices. A secondary task is to annotate the drawing with additional axial strokes relating proxy vertices that are not already axially related in the drawing.

We first discuss the workflow of our method as illustrated in Figure 5-2. We will discuss the crucial steps in more detail in the rest of this chapter.

1. The input is a freehand drawing, recorded as polyline strokes. Alternately image data could be converted into an edge-vertex representation with an edge-detection algorithm such as Canny [13] and imported.

2. First, the user identifies key points with the proxy interface of Chapter 4. Second, the user identifies a canonical coordinate frame of orthogonal directions by identifying a trivalent vertex with orthogonal outgoing edges; the system determines their 3-D directions using a technique described below. Third, the user annotates the drawing with additional axial strokes, connecting proxy vertices not related by the canonical frame directions.

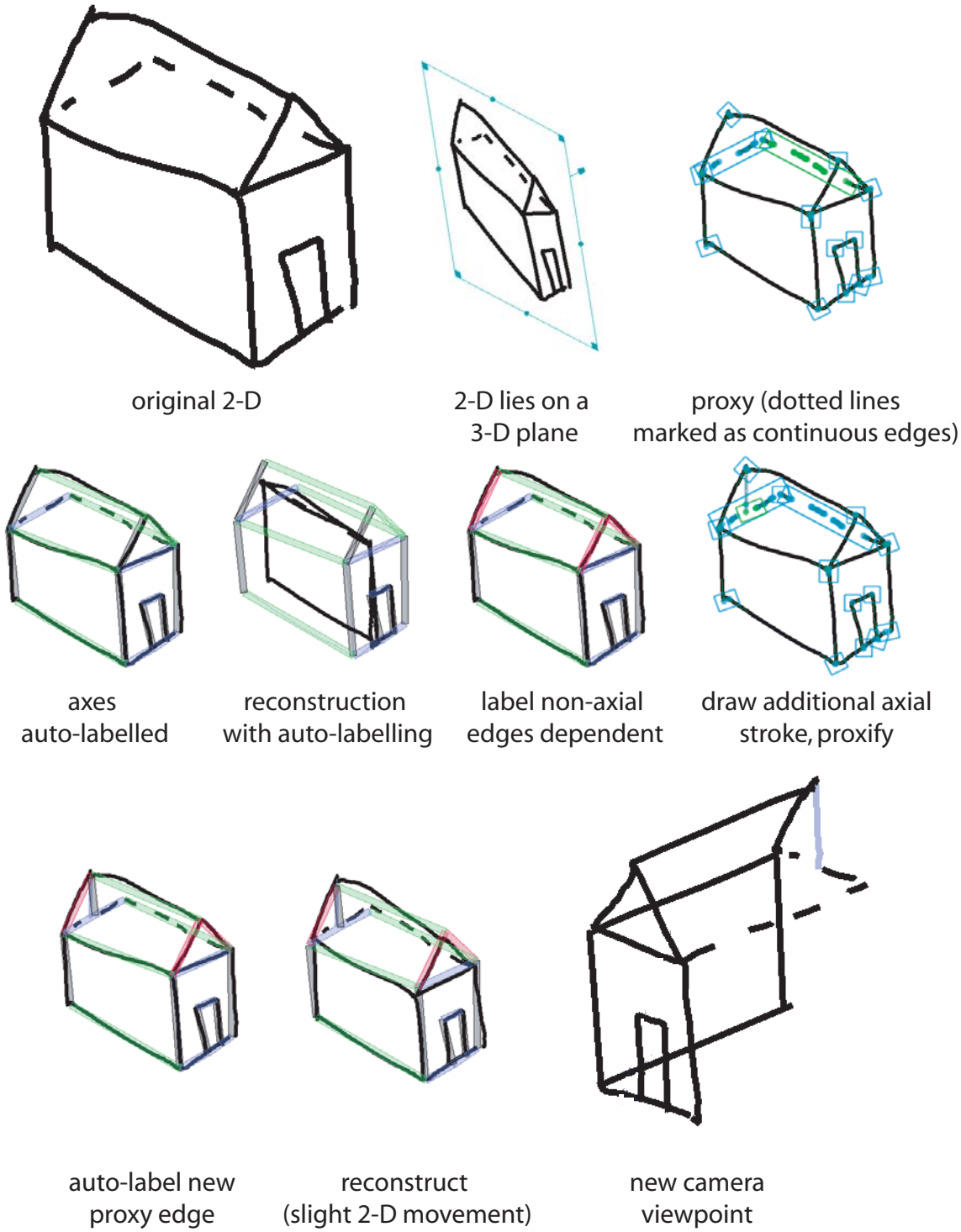

Figure 5-2: Full single-view reconstruction process. The model had 379 vertices, 14 proxy vertices, and 20 proxy edges; reconstruction took 18 msec.

3. The system automatically labels all the proxy edges as being in a particular known 3-D direction. For each proxy edge, our algorithm simply selects the canonical axis with the closest 2-D slope. Each edge is color-coded by its inferred axis as in Figure 5-2d.

4. The user overrides the automatic labelling. The user can cycle through other directions by clicking on the edge. In particular, the user typically identifies many edges as dependent — that is, the relative position of their proxy vertices can already be inferred by traversing other axial paths. Note that our system does not automatically classify any edges as dependent, so the user will certainly have to hand-classify such edges. We tried an approach where the system classified edges as dependent if they were sufficiently far away from any canonical axis, but found it to be rarely successful.

5. The system solves for 3-D vertex positions and moves all proxy vertices, using a solver described below. The strokes connected by the proxy vertices are moved correspondingly by the algorithm described in Chapter 4.

We now discuss generating 3-D orthogonal directions from a set of three 2-D projected edges and the 3-D solver.

Generating 3-D Directions In order to generate useful 3-D line slopes, we present a simple method to generate a set of orthogonal 3-D directions by simply drawing their projections. These initial directions can be used to construct new directions, as shown in Figure 5-3.

We assume that the user is drawing with no skew and a unit aspect ratio. We first turn our screen-space vectors into eye-space by generating four screen-space points corresponding to a center point and three points along the given canonical vectors. We project these points onto an arbitrary film plane, then project the vectors back onto our camera's up and right vectors; these will be our eye-space x and y coordinates, respectively. We can solve for eye-space z using simply the knowledge that the vectors are orthogonal:

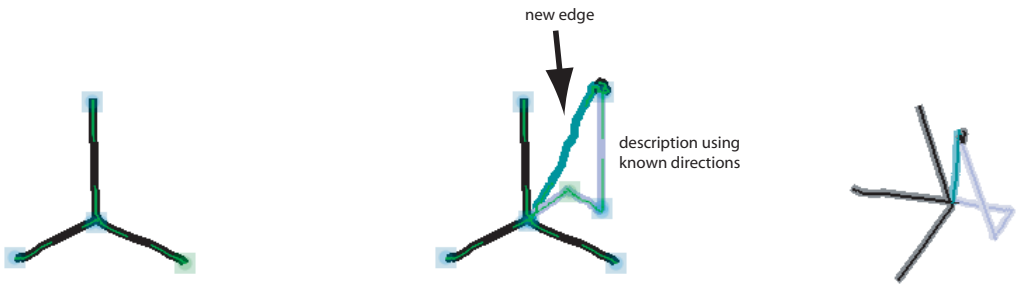

Figure 5-3: The user selects a trivalent vertex with orthogonal edges (left), which our system can reconstruct to 3-D. The user then freehand draws an edge with a new direction (middle), using the known axes to specify its endpoint. After reconstructing this figure (shown rotated on the right), the new edge direction can be used to label other 2-D edges.

$$x \cdot y = 0 \tag{5.1}$$

$$y \cdot z = 0 \tag{5.2}$$

$$x \cdot z = 0 \tag{5.3}$$

Note that the length of the vectors does not matter; projecting them onto the arbitrary film plane was simply a device to obtain the correct ratio between right and up vector components.

Expressing the known eye-space x and y components as two-dimensional vectors x_2D, y_2D, and z_2D, and the unknown z components as scalars x_z, y_z, and z_z, we can further rewrite this as

$$x_{2d} \cdot y_{2d} + x_z y_z = 0 \tag{5.4}$$

$$y_{2d} \cdot z_{2d} + y_z z_z = 0 \tag{5.5}$$

$$x_{2d} \cdot z_{2d} + x_z z_z = 0 \tag{5.6}$$

Substituting and rewriting, we get

$$x_z^2 = -\frac{(x_{2d} \cdot y_{2d})(x_{2d} \cdot z_{2d})}{y_{2d} \cdot z_{2d}} \tag{5.7}$$

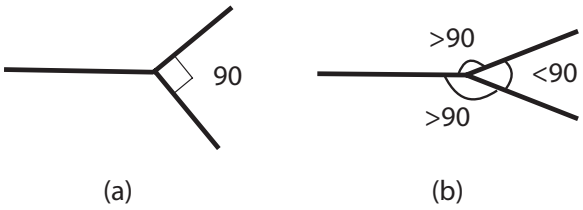

Figure 5-4: Relations of 2-D axes that cannot be reconstructed as orthogonal 3-D axes under orthographic projection, assuming unit aspect ratio and no skew: one or more right angles (a), or one angle (and only one) being less than 90 (b).

$$y_z^2 = -\frac{(x_{2d} \cdot y_{2d})(y_{2d} \cdot z_{2d})}{x_{2d} \cdot z_{2d}} \tag{5.8}$$

$$z_z^2 = -\frac{(x_{2d} \cdot z_{2d})(y_{2d} \cdot z_{2d})}{x_{2d} \cdot y_{2d}} \tag{5.9}$$

We cannot evaluate these depth values as real numbers if the fractions have 0 denominators or their squares are negative. Both of these situations have simple 2-D interpretations (see Figure 5-4). Having a 0 denominator means that a pair of 2-D vectors are already at right angles to each other on the film plane; this necessarily makes the third vector equal to the view vector and thus degenerate and inconsistent with the observed non-degenerate vector. In practice, we have never seen an exactly zero denominator occur. However, very small dot products result in very large z values and thus significantly more depth variation than probably expected. In our system, the user can recover from this by anisotropically scaling the resulting model or by adjusting the canonical 2-D axes and re-solving. A more proactive interface might try to automatically rotate the canonical axes further away from 90 degrees, or suggest the user do so. Having a negative square corresponds to one of the angles between vectors being less than 90 (and thus having a positive dot product), and the other two being greater (and thus having negative dot products). We can understand why this is an impossibility by analyzing what happens to a 90-degree angle under various viewing directions, which we do at the end of this chapter in section 5.5.

Given positive squares, we must be careful in choosing the sign when taking the square roots of these numbers in order to satisfy our original equations 5.1,5.2,5.3; the sign of x_z determines the proper sign of y_z and z_z, for example. There are thus

two valid answers (one for each sign of x_z), corresponding to the famous Necker Cube ambiguity, which the user can flip between. This gives us closed-form calculation of our orthonormal basis vectors. A derivation for perspective scenes based on vanishing points and an entirely different type of analysis can be found in Liebowitz et al.[44].

Reconstruction Given labeled edges, we first determine if the model is sufficiently labeled with edges of known orientation; imagine, for example, if the user had labeled every edge as dependent, the system would have no information from which to construct the depth. The necessary and sufficient condition for our purposes is that we can traverse from one vertex to all others using only edges with known directions. To confirm this, we simply perform a breadth-first or depth-first enumeration of the graph, not traversing dependent edges. If all proxy vertices are not reached, the user has not given us adequate information. We then help the user further by highlighting the edges which cannot be crossed: these are edges that connect vertices within the traversed set to vertices outside the set. The user can then choose to reclassify edges or draw additional disambiguating strokes to bridge the gaps, select appropriate features on them, and classify those edges.

Given a labeling that we know is unambiguous, we now turn to the actual 3-D reconstruction. We formulate our goal as a constrained minimization problem. The variables we solve for will be $p_0 ... p_{n-1}$ where n is the number of proxy vertices and p_i is the centroid of proxy vertex i; and $d_0 ... d_{m-1}$ where m is the number of edges labeled with known direction (we shall call them 'axial edges'), and d_j is the length of axial edge j. We have two types of constraints. We call "soft constraints" those constraints we would like to respect as much as possible, while "hard constraints" must be satisfied. In this application, the "soft constraints" are that the vertices should move as little as possible in screen-space; mathematically, that the distance of the solved screen-space locations of the proxy vertices to their original locations is 0. The hard constraints are that the solved proxy vertices lie along the appropriate 3-D axial vector from their neighboring proxy vertex centroids.

To find the optimum, we use the classic Lagrange equation for constrained linear

minimization:

$$\nabla f + \lambda \nabla g = 0 \tag{5.10}$$

where f expresses the soft constraint equation while g is a vector of the hard constraint relations, all defined, without loss of generality, to equal 0. The Lagrange equation says, essentially, that we have reached a minimum since moving along the direction to a lower value (∇f) would require changing the value of the hard constraints (∇g) away from 0. We can express f as

$$\sum_{i=1}^{n} \|p_{i_{2D}} - pOrig_{i_{2D}}\|^2 \tag{5.11}$$

where $p_{i_{2D}}$ is the 2-dimensional vector composed of eye-ray right and up components that we are solving for point i, while $pOrig_{i_{2D}}$ is the corresponding original location of point i, also in eye coordinates. The norm here is the L2 norm. For each axial edge e_j, we have 3 corresponding hard constraint equations, one for each dimension in 3-D; thus there are $3m$ hard constraint equations g_k:

$$(p_s - p_t) - a_j d_j = 0 \tag{5.12}$$

p_s and p_t are the unknown variables corresponding to the two 3-D points related by the edge, while a_j is the (known) 3-D axis corresponding to the labeling of the edge. All 3-D coordinates are expressed in eye-space. d_j is an unknown scalar, the distance along the axis that we will be solving for.

The gradient of f, ∇f, is conveniently linear in all its components relative to the points (and 0 relative to the d_i variables):

$$\frac{\delta f}{\delta p_i} = 2 * \|p_{i_{2D}} - pOrig_{i_{2D}}\|^2 \tag{5.13}$$

The gradient of each g constraint, ∇g, is also conveniently linear relative to each variable: for the partial derivatives $\frac{\delta g_k}{\delta p_i}$ it equals 1 or -1 if p_i is implicated in the constraint, or 0 otherwise. Similarly, $\frac{\delta g_k}{\delta d_j}$ equals the appropriate x, y, z component of

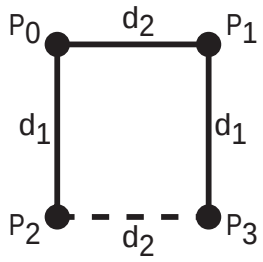

Figure 5-5: A
case where the constraints are not linearly independent. First consider all constraints but the dotted line: P2/P0 and P3/P1 are related by some direction d1, while P0/P1 are related by some direction d2. These constraints are linearly independent. They also determine that P2 and P3 are coplanar (the plane basis vectors being d1 and d2) — thus there are only two degrees of freedom left in their relative location. Adding the fourth constraint (the dotted line) that they are connected by some multiple of the 3-D direction vector d2 adds three constraints where only two are needed, causing the constraint set to no longer be linearly independent.

the appropriate edge axis if d_j is implicated in the constraint, or 0 otherwise.

We then solve the resulting matrix equation 5.10 with a standard linear solver (our prototype uses LU decomposition followed by back-substitution). One implementation detail is that the system will be underconstrained if the set of constraints are not linearly independent. This occurs in the common case when there are several axial paths between two vertices, as shown in Figure 5-5. Our solution has been to find an orthonormal basis (using standard QR decomposition, see Strang[70]) for the gradient rows of ∇g before solving.

If the user makes inconsistent requests — such as requiring that a given point be related to another point by the x axis along one edge and by the z axis along a different incoming edge — the system will return a distance of 0 for some edges, effectively collapsing them (see middle of Figure 5-2). Our user interface detects this condition and highlights the appropriate edges, enabling the user to go back and rethink their request.

If any solved proxy vertex is planar with two of its neighboring vertices, we project their intermediate stroke vertices onto the corresponding plane. This is primarily useful for proxy edges that correspond to sketched curves; the curves become planar.

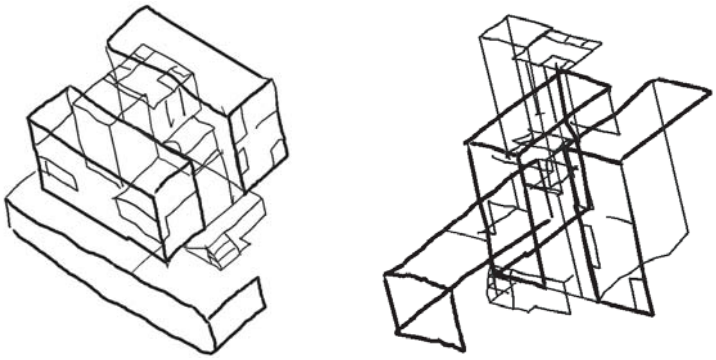

Figure 5-6: reconstructed architectural sketch — 1576 model vertices, 111 proxy vertices, 145 proxy edges, 20 sec — original view on left, rotated on right.

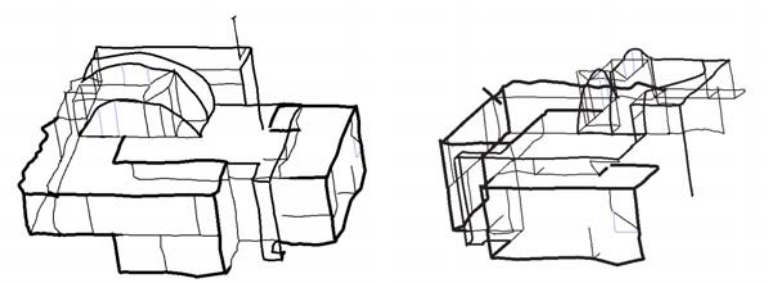

Figure 5-7: another reconstructed architectural sketch — 1668 model vertices, 103 proxy vertices, 152 proxy edges, 16 sec — original view on left, rotated on right.

This can be seen in Figure 5-9.

5.3 Results

We demonstrate four reconstructed sketches, in Figures 5-6, 5-7, 5-8, and 5-9. The first three were freehand drawn based on existing architectural sketches by Herman Hertzberger [36]; the fourth was drawn based on a screenshot of a 3-D digital model seen on the Web. Drawing orthographic sketches is a standard tool of the architectural trade. However, each designer utilizes them to different degrees during the design process. The architect shown here uses them more often than most. Our experience is that hand-drawn orthographic sketches are relatively accurate in the relative 2-D positions of vertices, probably because such drawings are simple to construct and

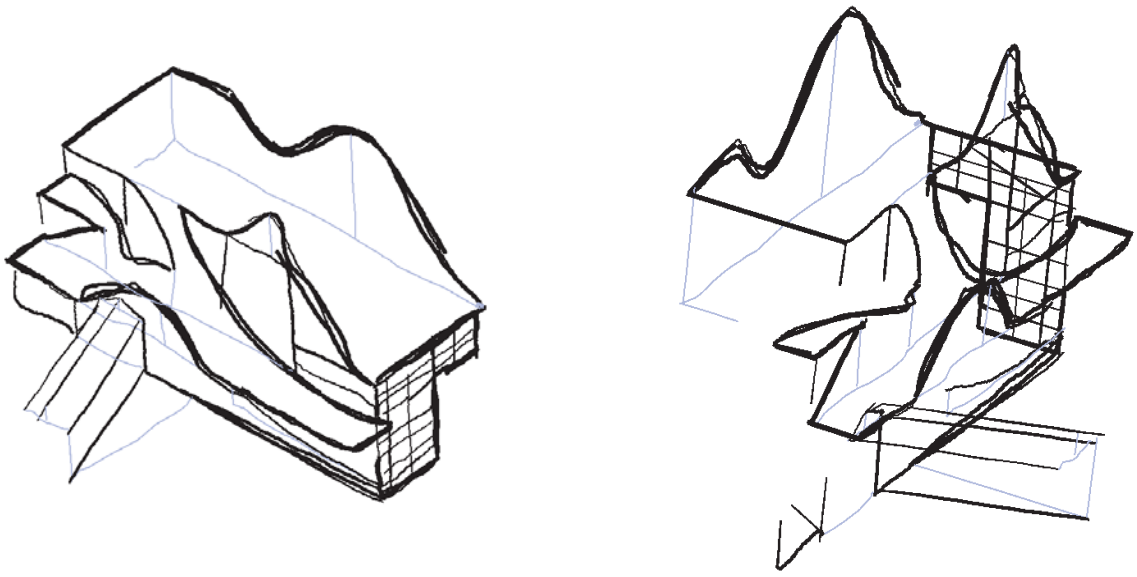

Figure 5-8: A third reconstructed architectural sketch — 7953 model vertices, 78 proxy vertices, 112 proxy edges to be formed, 5 sec. The largest 2-D vertex displacement caused by the reconstruction in the original was 26 pixels.

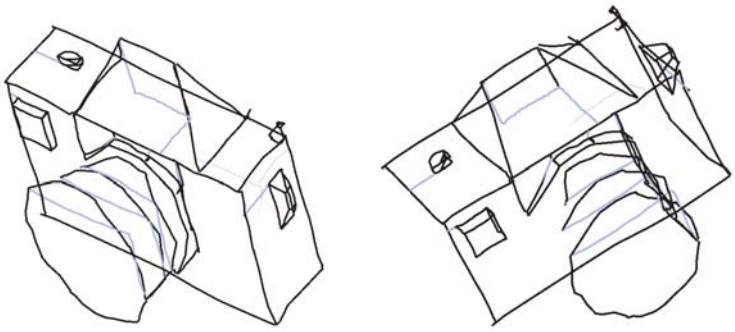

Figure 5-9: reconstructed sketch of a camera — 1430 model vertices, 68 proxy vertices, 95 proxy edges, 3 sec — original view on left, rotated on right.

validate by eye.

After turning it into 3-D, the user will probably want to edit and explore it in 3-D — perhaps with the manipulation tools of Chapter 4 — and perhaps add further 2-D forms to turn into 3-D. Our prototype supports this incremental workflow. For radical changes, the designer might simply draw a new orthographic sketch and reconstruct it.

5.4 Analysis

We have demonstrated an interface where the user can embed a simpler proxy geometry on top of their model, then specify simple axial relations between proxy vertices. This interface is both easy for the user to specify and easy and fast for the computer to solve. Unlike previous methods based on inference, the result is predictable.

Design Implications

Integrating this approach into an existing design process remains the high-level challenge. Particular to 3-D sketches are the questions of how to make a series of these 3-D sketches, since the designer does not enjoy the space provided by plentifully available sketch paper. Perhaps the models could be placed in virtual 3-D space in some useful way. An alternate approach is to use the 3-D reconstruction only as a visualization tool, and continue to use 2-D sketching as a primary exploration tool — such as allowing the rotated 3-D model to be used as a basis for new 2-D sketches. Along this line, the system could attempt to automatically reconstruct a sketch while it is drawn, selecting points and assigning axial labels; even if wrong, the visualization could be useful and our manual interface could be used to further correct the system.

A more incremental workflow would reduce solving times and make fixups easier. In particular, being able to assemble multiple submodels that have been reconstructed separately — finding a best overall fit while respecting the original constraints — would be helpful.

Lower-level improvements

We have not attempted to optimize the solver and thus believe there is significant room for speed improvement — for example, by using a solver that exploits matrix sparsity. Larger models would probably be constructed in stages; for those situations, incremental solvers like conjugate gradient, which can take advantage of variables being initialized close to their final values, might prove faster.

The primary difficulty with the interface was coping with unexpected results, typically due to inconsistent axial requests (as in Figure 5-2d). Some careful analysis on the user's part was required to untangle such requests; automatically fixing such problems, or helping the user diagnose them, would be helpful.

Selection of proxy vertices is time-consuming; adding automatic aids to the process, as described in Chapter 4, would be helpful.

Although axial lines suit architectural models very well, they are an inconvenient annotation system for many other classes of objects, particularly curved objects. Adding different types of annotation — creating a vocabulary of image-space depth cues — would be helpful; at the same time, this will probably make our optimization function non-linear, making it slower to converge and, perhaps, more difficult to guarantee convergence. For curves, drawing a second "shadow" curve as in Cohen et al.[15] might be a useful extension. Drawing surface contours is a fundamentally limited vocabulary, particularly for describing smooth surfaces where the surface outline changes with viewpoint. An annotation language for such surfaces, perhaps using "Teddy"[39] as a starting point, is necessary. Perhaps isocontours (of depth, or surface normal) would be useful, or a "shape-from-shading" approach as in Liu and Huang[47].

Another intriguing possibility is to treat the annotation system as a way to perform full-on geometric substitution like gesture-based modelers do; for example, turning a silhouette into a smooth blobby surface as in Teddy[39]. Unlike these existing works, this substitution could be designed to maintain the property that the reconstructed 3-D model looks like the original sketch strokes as much as possible. It should also allow a range of substitutions, from low-level edges like our interface currently supports to

higher-level primitives such as cubes or blobby surfaces.

The segmentation problem needs further exploration, particularly for a semi-automatic interface to extract strokes from images. Often some of the sketch is not really 3-D geometry (such as text annotation) and should not be reconstructed. Attaching proxies directly to images (as in morphing work like Beier and Neely[10]) might be another interesting research path.

We currently assume that the "wiggles" in sketched strokes represent 3-D information, which often they do not. It might be helpful to represent those wiggles as 2-D offsets from projections of straight 3-D lines or smooth 3-D curves, as in Kalnins et al.[40].

Extending the reconstruction to different camera types would also be useful. Perspective cameras are of course one option; we suspect that perspective drawings do not have as much foreshortening as they ought to, and additional user input will be needed to correct for that. Another camera, less explored in graphics, would be the oblique, where essentially the model is sheared such that all three axes maintain their world-space 3-D length ratios under projection. Oblique drawings are common in architecture and industrial design. For the perspective camera, the 3-D reconstruction step would likely become nonlinear; for the oblique, however, it might not.

5.5 Appendix: Impossible Orthographic Axes

In this section we demonstrate why the 2-D configurations in Figure 5-4 are impossible. We will show that the view direction determines whether we see the projected angle as greater than or less than 90 degrees. Turning this relation around, we will then infer qualities of our viewing direction given the three projected angles, and show how this analysis leads to a contradiction if one, and only one, of the projected angles is smaller than 90 degrees (see Figure 5-10).

In part (a) of the figure, we are viewing a right angle caused by two orthogonal line segments (x1 and x2) head-on. We characterize all possible view directions relative to this segment pair by the sign of their x1 and x2 components. This grouping

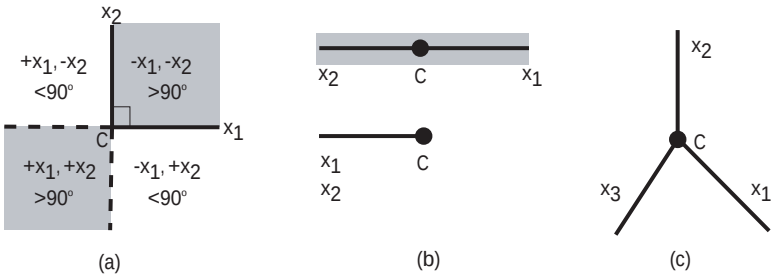

Figure 5-10: Analyzing view directions relative to the right angle made by C, x1, and x2. Consider the sign of x1 and x2 components of all view directions, and the resulting projected angle(a). When the signs are the same, the projected angle gets bigger; at most (b, top), going to 180 degrees when the viewpoint is in the plane of x1 and x2. When the signs are opposite, the angle shrinks to 0 in the plane (b, bottom). From this analysis we can show how the case where only one projected angle is less than 90 degrees (c) leads to contradictory inferences about viewing directions and thus is impossible to image with an orthographic camera.

corresponds to the magnitude of the projected angle in a fairly intuitive manner. If the sign of x1 and x2 components are the same, the segments will form a projected angle greater than 90 degrees, regardless of the x3 component. Similarly, when their signs are different, the projected angle will be appear to be less than 90 degrees. For more intuition, consider the set of view directions with no x3 component - that is, viewing the right angle from a direction composed only of x1 and x2 component. In this case, all three projected points will be collinear, but their order will be different (see Figure 5-10b). In the same-sign zones (top of b in the figure), the center point in 3-D projected to the center of the projected 2-D endpoints: that is, the projected angle will be 180 degrees. In the opposite-sign zones (bottom of b in the figure), the points will still be collinear, but both endpoints will be on the same side of the center point — that is, the projected angle will be 0 degrees. As we move our view direction out of the plane from any of the zones, the projected angle will again approach ninety, hitting it in the limit (i.e. when x1 and x2 components go to zero). Note that we can disregard the extraordinary points in this space — where the x1 or x2 components have no sign because they are zero, which includes the view this diagram is drawn from — because they result in degenerate projections: the third axis projects to either a point (going directly in and out of the film plane) or a line parallel to one of the

other axes' projections.

Given the above knowledge, we now consider a situation (as in Figure 5-10c) where only one of the projected angles (x1-x3) is less than 90. Taking each pair of orthogonal lines in turn, we will attempt to infer the x1, x2, and x3 components (we will call them c1, c2, and c3 respectively) of our view direction, and in so doing find a contradiction. Since x1 and x3 project to less than 90 degrees, we know c1 and c3 have different signs. Since x1 and x2 project to an angle greater than 90 degrees, we know c1 and c2 have the same sign; by the same logic, c2 and c3 have the same sign — but by the transitive property this means that c1 and c3 have the same sign, contradicting our first observation. Similar analysis of the other cases — that is, where zero or two of the angles project to less than 90 degrees — leads to no such contradiction.

Chapter 6

3-D Reconstruction from Two Sketches

In this chapter we extend the theme of reconstruction from the previous chapter to handle the case where the user would like to reconstruct a 3-D model from two input sketches. This case is common in architectural design, where the user may simultaneously work out designs in plan, section, perspective, and/or isometric views, or some combination thereof. As in the previous chapter, we use the proxy interface to simplify the input sketches. The problem here then becomes one of correspondence between proxy vertices in each sketch, rather than the depth annotation required with a single input sketch. We introduce a user-assisted technique for making these correspondences, creating a reconstructed 3-D model in the process. As with the output produced by the one-view approach, this 3-D model retains the sketched appearance of the 2-D originals. Although this problem has been studied in computer vision for photographic images, sketches introduce additional challenges. Particularly, they use degenerate views where 3-D edges are parallel to the viewing direction and thus project to a single point. Such a degenerate feature point must be matched with multiple feature points in the other view. We provide an efficient interface to specify a large set of such one-to-many correspondences. Additionally, the sketches might not properly correspond yet. We introduce a novel algorithm to reconstruct a 3-D curve from two 2-D projections that do not precisely correspond.

6.1 Differences from Reconstruction with a Single Sketch

Having an extra drawing changes our reconstruction task in an important way: rather than having too little information, now we have too much! The problem is that, for each source feature in one drawing, we simply needed a 1-D quantity, the depth coordinate, but instead, we now have a 2-D coordinate (the corresponding location of the same point in the second drawing). This extra degree of freedom is a potential (and very real) source of inconsistency. To handle it, our system will choose a "best-fit" 3-D point, spreading the error among both drawings. The same problem applies to the edges between features and again we will construct a best-fit approximation, this time with a novel curve-matching algorithm. Another way to think about this, more consonant with the graphics literature, is that we will first "morph" the 2-D sketches so that they are consistent, in effect removing the extra degree of freedom, then perform standard disparity stereo (i.e., triangulation-based) reconstruction.

There is another sense in which having an additional drawing could put us in the situation of having too little information: namely, we do not know the relative camera positions of the two drawings. In our system, we count on the user to perform "camera calibration" — i.e. place and orient the planes appropriately. Since it is common to make drawings that are at 90-degree angles to each other, hand-calibration is generally not difficult. However, for arbitrary angles, we could deploy automatic methods to assist the user.

Instead of specifying axial relations as in the previous chapter, our user specifies correspondences between pairs of features in separate sketches. In this chapter we present two novel user interfaces to speed this process.

6.2 Related Work

While reconstructing a 3-D model from two views is a classic issue in computer vision, our situation differs in that 2-D sketches do not exhibit the metric accuracy

of photographic imagery and the two views might not be consistent projections of a 3-D model. This inconsistency (and the use of degenerate views as described above) hinders automatic identification and matching of features, which requires us to resort to user assistance. On the other hand, our use of sketches allows us to exploit stroke topology to facilitate this matching.

6.3 Exposition

In this section we present two novel user interfaces to speed this process: the first exploits feature topology to offer a small set of possible points adjacent to the current reconstructed point and presents this choice in 3-D context, in effect allowing the user to reconstruct by iteratively traversing the 3-D model; the second additionally helps the user to exploit degenerate projections to make large numbers of correspondences at once. Unlike automatic methods, our method is robust to arbitrary error in the 2-D projections and feature selection (since the user can pause the reconstruction at any point and manually fix up the features) and can handle degenerate views (relying on the user to disambiguate which points on the same epipolar plane actually correspond; and allowing the user to handle the difficult case of detecting occluded feature points). The framework also allows automated algorithms — to identify features, to calibrate cameras, to find correspondences, or to suggest good adjacent correspondences — to be deployed for cases where their assumptions hold.

6.3.1 Incremental Reconstruction Using Topology

In our system, the user begins with two planes, assembled in virtual 3-D space, each with a sketch on them. The user first selects key features in each sketch using the proxy interface of Chapter 4. The reconstruction process is seeded with an initial correspondence made by the user that indicates a proxy vertex in one sketch that corresponds to a proxy vertex in the other sketch. This initial correspondence reconstructs an initial 3-D point.

From this initial 3-D location, our interface presents a small set of outgoing edges

and adjacent points in 3-D. The user selects one of these adjacent points, causing the intermediate edge to be reconstructed; the interface then switches to present all outgoing edges from that point, and so on. In effect, the user reconstructs by iteratively traversing the 3-D model. Since this is a local and incremental operation, the user needed not have established perfectly corresponding feature topology across the drawings; in the middle of our process, the user can pause to fix up features. In this section we first describe how each step in the process (see Figure 6-1) is accomplished.

Reconstructing a 3-D Point from a Single Correspondence

Given a pair of proxy vertices that are known to correspond, the system constructs 3-D feature by shooting eye rays out from each proxy vertex (see Figure 6-2). The points of closest approach on each ray determine a line segment — that segment is the extra degree of freedom mentioned at the introduction to this chapter (This line segment is shown in Figure 6-5). We place the 3-D centroid at the midpoint and copy all the vertices from each 2-D proxy vertex into the new 3-D feature, maintaining their relative position from the centroid. We thus minimize the world-space error for both views. For two orthographic cameras, this has the desirable property of also minimizing the projected error in both views; for the case of perspective cameras, this minimization requires solving for the roots of a degree-6 polynomial (see Hartley and Zisserman [35]). In practice, we have not observed the error to be perceptually significant with perspective cameras.

System Reconstructs and Offers Local Options

Given a current 3-D proxy vertex, the system looks at the corresponding proxy vertices from its 2-D sources. Each 2-D proxy vertex has some set of adjacent proxy vertices; the system enumerates every combination of those vertices — moving to an adjacent vertex in one drawing but not the other, or moving to adjacent vertices in both drawings. In order to display the reconstructed edges between these vertices, a 2-curve reconstruction algorithm is also required, which we will describe in section

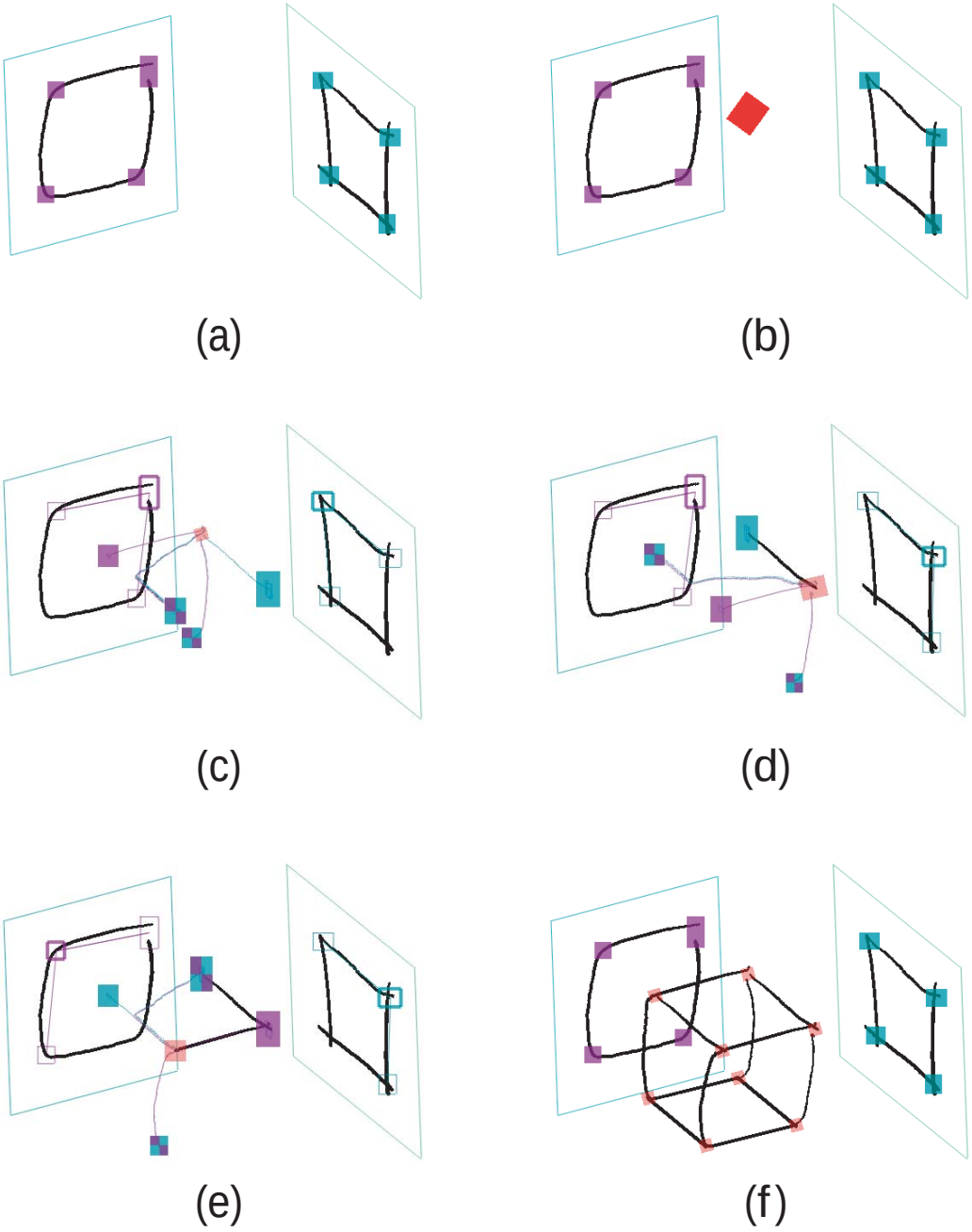

Figure 6-1: Incremental two-view reconstruction. The user selects features in both sketches (a), then establishes a single initial feature correspondence(b). From there the system presents local options: in this model, options corresponding to edges along the top face, and one option corresponding to the diagonal (c). The user selects one of those options, causing another set of options, relative to the new feature, to be displayed (d). The user chooses another option (e). By incrementally walking around the model in this manner, the user reconstructs the original shape (f).

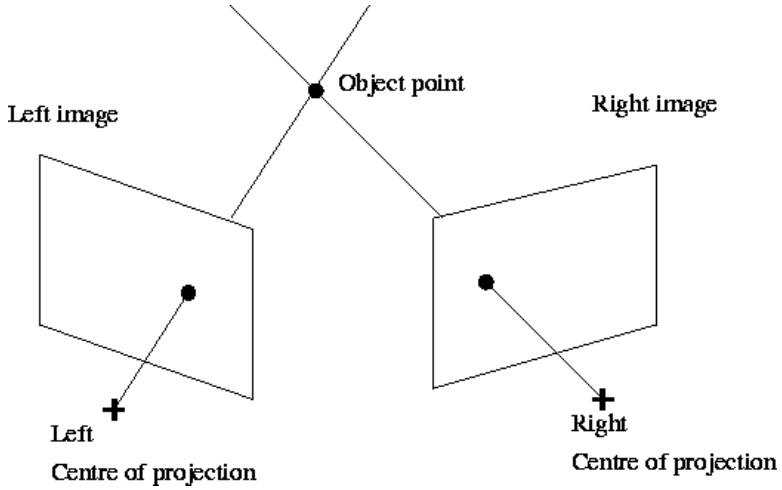

Figure 6-2: Stereo triangulation. Given two 2-D points that are known to correspond, we can easily determine a 3-D point with simple triangulation. If the rays emanating from the 2-D points do not meet, we can find the points of closest approach along each ray, and use the average of those points as the 3-D location.

6.3.3. The system displays all potential edges up to a constant error threshold; the user can then request display of more edges if the desired one has been filtered out. Our error function is defined as the length of the line segment between the points of closest approach on each eye ray (see Figure 6-5) divided by the Euclidean distance of the prospective 3-D centroid from the source 3-D centroid.

6.3.2 Fast Specification of Correspondences Between Degenerate Features

Most computer vision systems make the assumption of non-degenerate views; that is, that no two 3-D features will project to the same 2-D location. Architectural drawing practice typically violates this assumption by using axis-aligned orthographic views. As shown in Figure 6-3, use of such views for a simple cube results in every corner feature being imaged to the same location as another feature.

We allow the user to exploit degenerate relations to make large numbers of correspondences at once. That is, if we consider a single epipolar plane, where all the points project onto a single epipolar line in each drawing, when multiple features lie

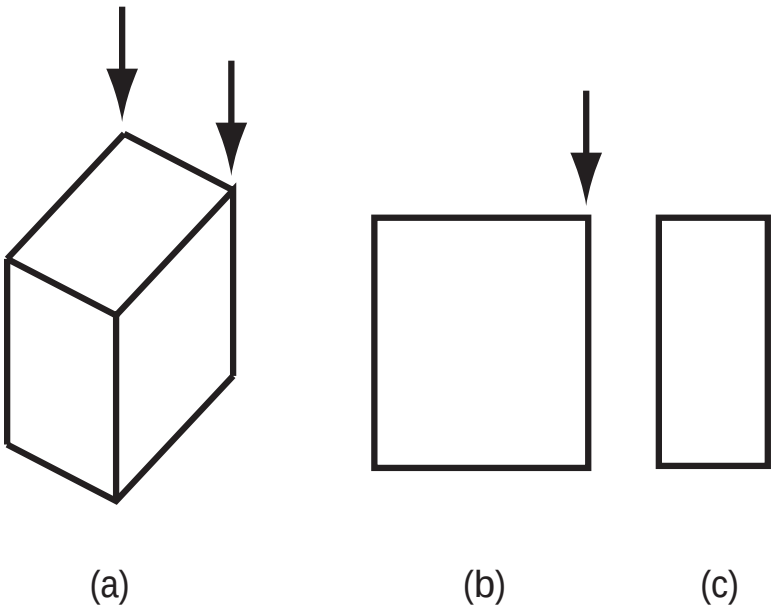

Figure 6-3: Drawing the 3-D cube (a) with two axis-aligned views (b) and (c) results in every corner feature being degenerate — that is, being projected to the same location as another corner feature. The arrows indicate one of these corner pairs which image to the same 2-D corner.

on that plane, there are a combinatorial number of possible reconstructions. The user specifies desired alignments, and we reconstruct all possible 3-D features (see Figure 6-4) using the same techniques as in section 6.3.1.

6.3.3 2-Curve Reconstruction

In this section we describe an algorithm that, given two curve projections drawn from known cameras (see Figure 6-5), automatically computes a 3-D curve that best-fits those projections. Our algorithm works with polylines. As in Cohen et al.[15], our algorithm traverses both curve's vertices in order, producing a list of vertex pairs. Each item in this list is then turned into a best-fit 3-D point; connecting these points in the list order creates the 3-D polyline. Our formulation is more flexible than previous work in allowing backtracking at any time (i.e. if the current pair includes point c_i, the next pair can include point c_{i+1} or c_{i-1}) and, optionally, degenerate projections (where the next pair includes c_i again); an additional advantage is that

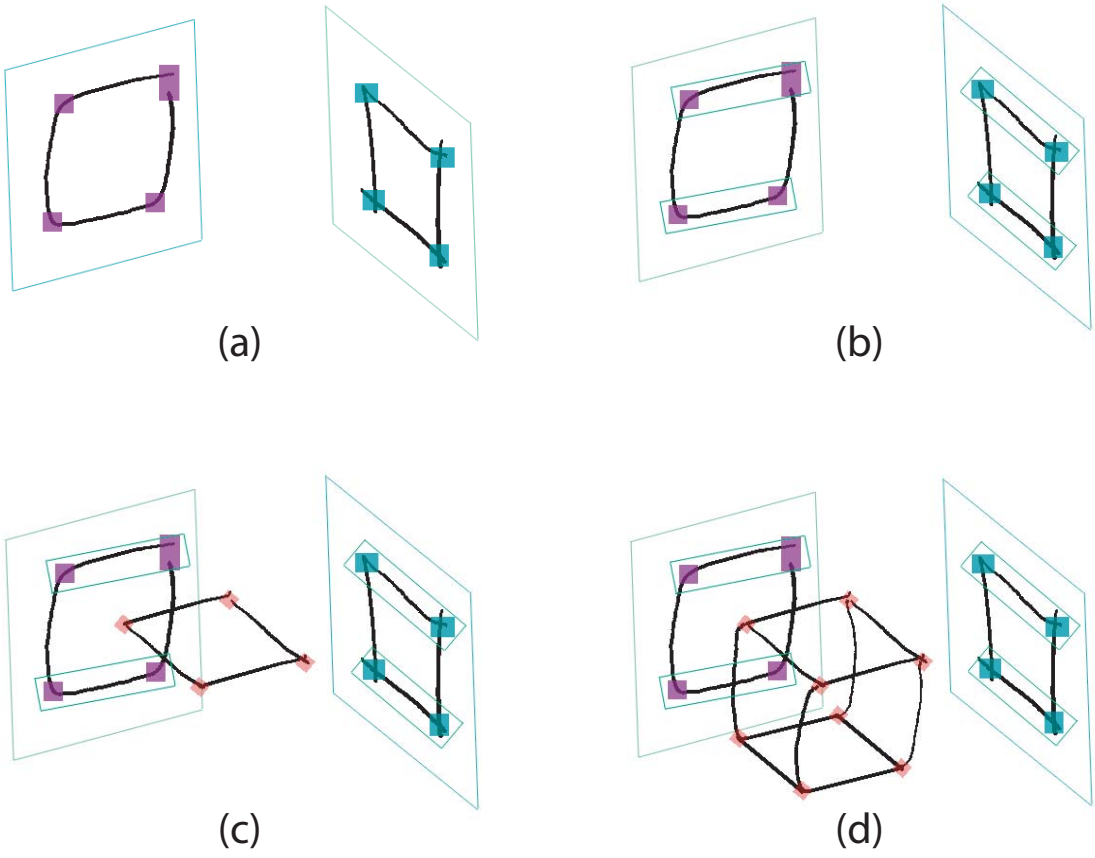

Figure 6-4: Degeneracy-exploiting two-view reconstruction. The user draws two sketches on planes and labels key features (a). The user groups proxy vertices that lie on the same epipolar plane (b). The user then associates these vertices, causing all combinations of 2-D features to be combined into 3-D features (c). The user does the same with another set of proxy vertices, causing more 3-D features, and their appropriate connections with existing 3-D features, to be instantiated (d).

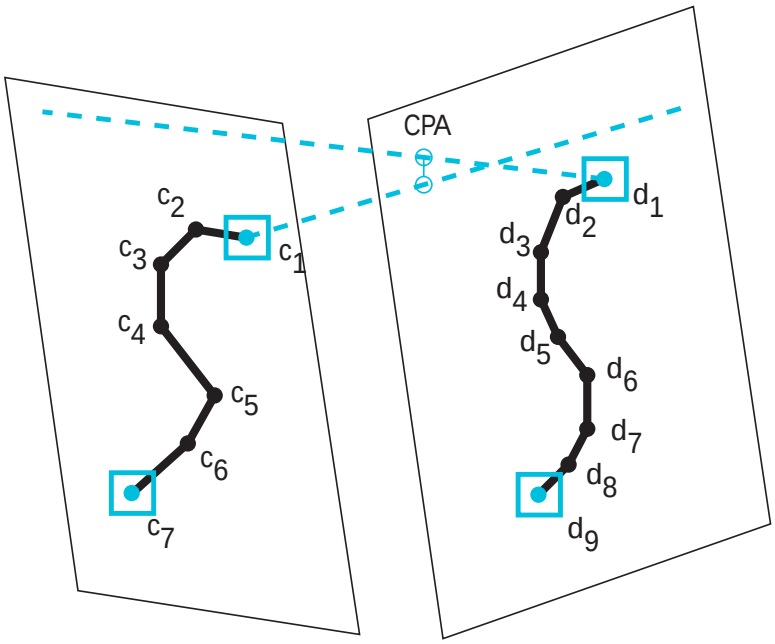

Figure 6-5: Our 3-D curve reconstruction algorithm is given two 2-D curves on two different projective planes, with known endpoint correspondence — here, $\{c_1, d_1\}$ and $\{c_7, d_9\}$. We incrementally traverse each curve's vertices, allowing backtracking, and minimizing the sum of per-pair errors, defined as the distance between the closest-points-of-approach, labeled CPA, along the camera rays through each point in the pair.

there are no parameters to tune. Figure 6-6 shows a case where the backtracking along one curve would be necessary.

We can model the space of all valid lists of vertex pairs as that of all possible paths through a weighted graph, then use Dijkstra's algorithm to efficiently find the optimal path. We define one curve c to have n vertices, and the other curve d to have m vertices. Each node in the graph corresponds to a candidate vertex pair $\{c_i, d_j\}$, where i is $[0...n)$ and j is $[0...m)$. We associate a weight with each node using an error value that we wish to minimize. The error function we use is the 3-D distance between closest points of approach (see Figure 6-5) of the eye rays through each vertex. Each graph node is connected to nodes containing adjacent vertices as shown in Figure 6-7. Dijkstra's algorithm works on graphs with directed, weighted edges (not weighted nodes), so each edge's weight is assigned to be the weight of their destination node.

We find the shortest path through the graph between the start point pair $\{c_0, d_0\}$

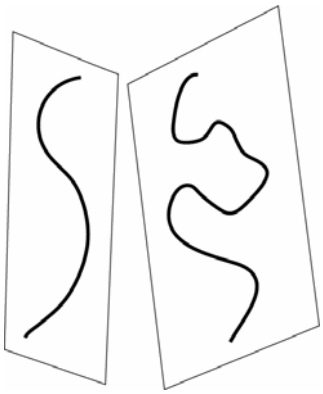

Figure 6-6: Difficult reconstruction case. In order to match up the curves, we have to traverse both up and down along the curve on the right while traversing down the curve on the left.

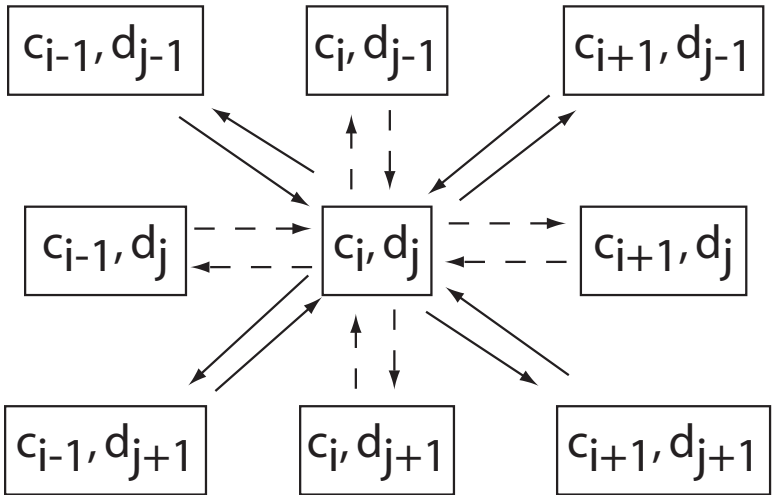

Figure 6-7: For a given node in our graph of vertex pairs, there are eight possible neighbors and sixteen possible directed edges. If we only allow the curve vertices to be traversed in forward order, we would only connect nodes along the large arrow as shown. If we allow backward traversal, we would include the rest of the diagonals. Finally, to allow degenerate edges, we would include the horizontal and vertical connections.

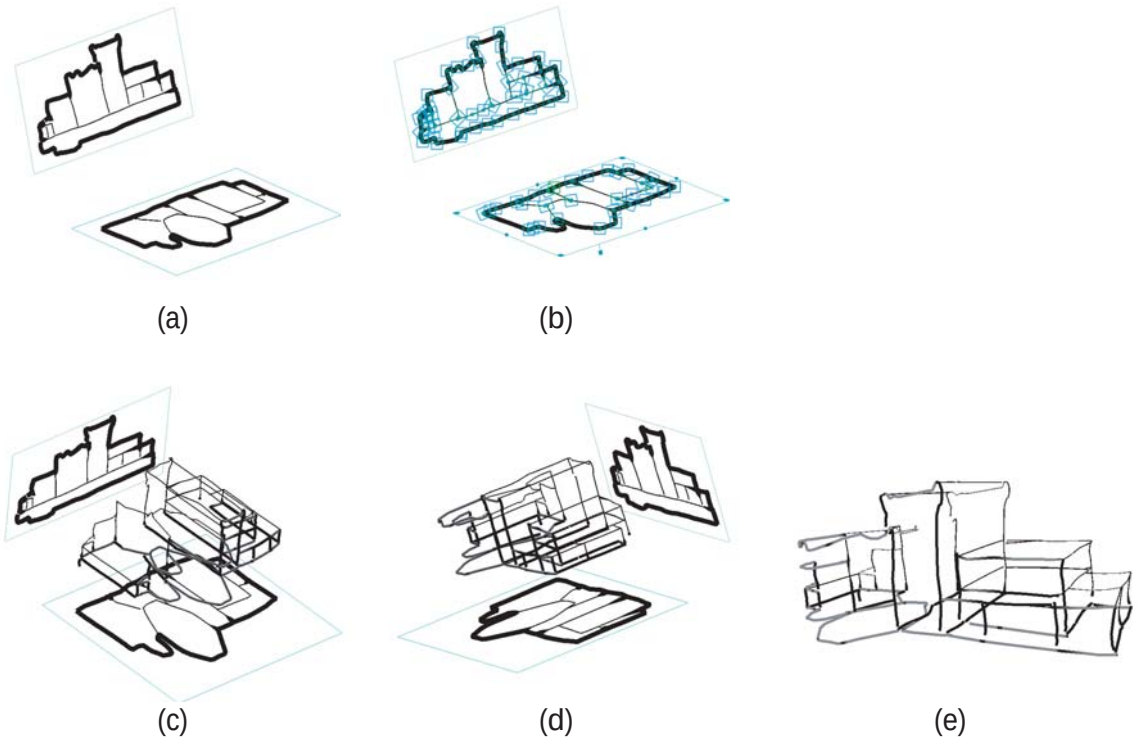

Figure 6-8: These two drawings (a) were copied freehand from [17] and placed on orthogonal planes. The user then circled key points (b). Using our reconstruction interface, a 3-D model was created (c)(d). Another view of the model, under perspective projection, is shown in (e).

and end point pair $\{c_{n-1}, d_{m-1}\}$. Each vertex pair is assigned a 3-D location midway between the CPA of each vertex's eye ray. We then construct a polyline from those 3-D points.

6.4 Results

We show a 3-D model reconstructed with our system in Figure 6-8. In some cases, we matched up features that weren't technically projections of the same 3-D feature but whose reconstructed 3-D location looked acceptable. Although incorrect correspondences are troublesome in automated approaches in computer vision, these user-guided "mistakes" seemed to work quite well. In retrospect, we were somewhat lucky with these particular sketches in that we were able to circle key points in each

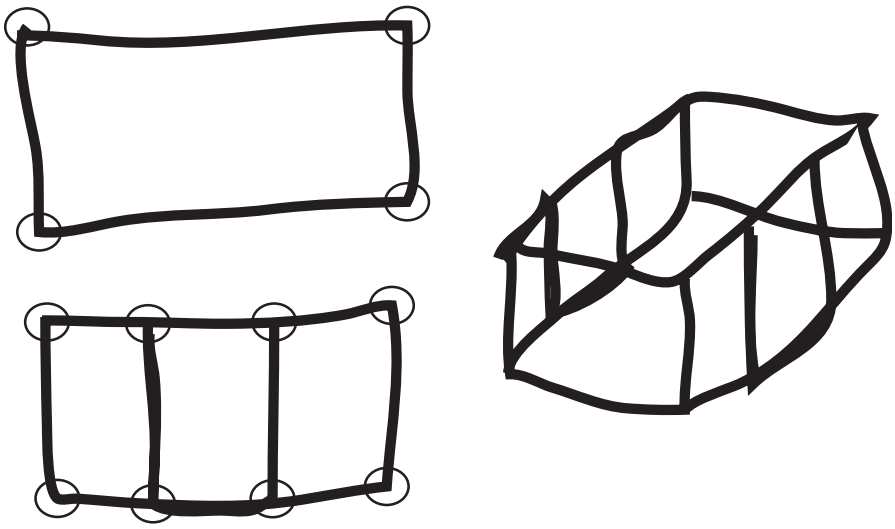

Figure 6-9: With topology differences between the source sketches (left), the system cannot successfully find the one-to-one mappings between the features in each sketch needed to generate the 3-D model (right). In the top left sketch, the system would have to generate additional features along the middle of the horizontal strokes.

sketch without any concern for finding matching features in the other sketch. We have since found this approach to be troublesome when the topology of the vertices is vastly different between sketches — i.e. one or the other of the sketches has many extra features (see Figure 6-9). Assisting the user in finding and correcting these conditions during the feature-identification phase would be helpful.

We also enhanced our manipulation interface to take into account the known links between 2-D and 3-D features, as shown in Figure 6-10. This further exploits the degeneracy of the original sketches by causing degenerate features to move together.

6.5 Analysis

Design Implications

Integrating this approach into an existing design process remains the high-level challenge. Our architecture students seemed most excited by the back and forth shown in Figure 6-10 where the 2-D sketches and 3-D model control each other. As we noted at the end of the previous chapter on one-view reconstruction, semi-automatic methods

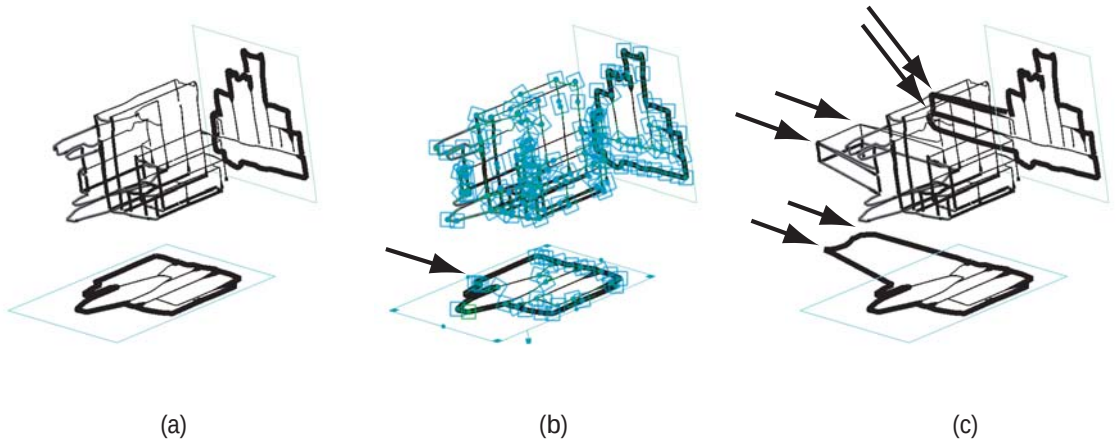

Figure 6-10: Linking 2-D and 3-D. Given the 3-D model from Figure 6-8 shown in (a), the user drags the selection from the 2-D plan as shown in (b). The result, shown in (c), is that the 2-D plan causes the 3-D model to move, which causes the corresponding feature in the 2-D elevation to move. Moving the 2-D elevation feature, in turn, causes the other 3-D corner feature to move — causing the other corner of the edge to move in the plan as well.

that allow designers to continue to work primarily in 2-D would be helpful. With two sketches, the system could skip 3-D visualization altogether and simply show how changes to one sketch apply to the other.

Lower-level Improvements

Given features in one drawing, we should be able to further automate the feature-finding in the second drawing, since we know that corresponding features should lie near the corresponding epipolar lines and have similar connectedness. Additionally, assisting the user in finding discrepancies would be helpful.

Currently, given two pairs of corresponding features, we create a 3-D curve for every pair of source 2-D curves (note that multiple sketched curves could connect a given pair of features). A simpler, and perhaps visually superior, approach would be to make a best-fit 3-D curve that fits all the curves, then treat the 2-D source curves as view-dependent appearance.

Our two-curve reconstruction algorithm could also be improved. The biggest difficulty is that the most accurate reconstruction may be much different from what the

user expected and that slight changes to the 2-D sketches could cause large changes in the best-fit curve. We probably need some sort of user interface to trade off projective error for 3-D smoothness. The error function itself could also be improved: we could use the 2-D distance between the original 2-D vertices and the projection of the best-fit reconstructed 3-D point. These distances are equivalent under orthographic projection, but can be quite different under perspective projection. The algorithm also does not scale well, finding correspondences in $O(nm)$ time in the worst case, where n is number of points in one curve, and m is the number of points in the other curve. The worst-case behavior occurs when the source curves are essentially lines defining an epipolar plane. In these cases, there exists an entire family of lines within the plane that would be valid reconstructions which the optimization struggles to maximize. Since we can identify these degenerate segments before the reconstruction, they could potentially be separated out of the optimization.

Chapter 7

Modeling with Camera Planes

We present a novel interface for creating suggestive 3-D models through 2-D freehand sketching. Thus far we have described 2-D sketches, and a 3-D equivalent. Now we present something in-between - a 3-D model with the same targeted descriptive power as a 2-D sketch. Unlike low-resolution 3-D geometry, the resulting models are not *smooth* where the user has not specified information — they are *empty*. These models are more descriptive than an individual plan and section, on the one hand, but less descriptive than a standard 3-D model or orthographic or perspective sketch, on the other. The basic widget in our system is a combination of a camera and a plane, where the plane is a film plane of the camera. "Camera planes" can be utilized in three ways: as descriptions of sectional cuts, as projections, and as approximations to nearly planar geometry. The key way to create models with these planes is through 2-D sketching, designers' preferred medium.

7.1 Motivation

In Chapter 1, we stated that designers work on 3-D ideas with a sequence of different media. We now explain specifically why using both 2-D and 3-D representations is advantageous.

First, 2-D drawings are not simply projections or slices of 3-D worlds. A more accurate relationship would be that of an oil painting to an equivalent sculpture.

Certainly there is geometric information in the painting that can be translated into full 3-D, and we would like to support that translation. However, this information is represented in a unique way: with a sort of visual shorthand or approximation (such as silhouette outlines or hatching) both easy to make with a drawing tools and tuned for human visual processing — a serendipitous combination other media still struggle to match, but also a challenge to bring into 3-D. Similarly, with their deviation from projective accuracy, use of line weight and shading, and use of abstraction, drawing elements cannot simply be given depth coordinates (discussed in Durand [22]). If they can be translated, it would be through completely different expressive conventions specific to 3-D, and even then probably not a matter of simple 1-to-1 substitution. Our system is meant to hold both these untranslated (and possibly untranslatable) 2-D artifacts and ensuing 3-D translations in a common context.

Second, a set of 2-D sketches are required to represent a single 3-D object. Again perhaps counterintuitively, this highlights a great advantage: the ability to divide and conquer. That is, sketches segment a design problem into a set of subproblems, each of which can be worked on independently, visualizing only the elements that matter, and whose consequences can be carried over into other sketches (and differences resolved) at a time of the designer's choosing. At any given time (and, in practice, most of the time), the sketched geometry may not correspond to a set of projections of a single 3-D model. This is another powerful argument for working in 2-D. At the same time, the designer *is* imagining a 3-D object, and when 3-D elements *are* resolved, we would like to capture that too. Our system spans this spectrum: the user can work on various planes as though they are separate 2-D objects; then assemble them, perhaps resolving incongruities; then, finally, translate them to 3-D.

A final, unique quality of 2-D is the disjunction between the mark-making tool and the perceived primitives. That is, unordered sets of strokes are perceived to coalesce into arbitrary shapes, or perhaps even ambiguous sets of possible shapes. This reliance on perception instead of the artifact's method of construction to describe the form gives the process an incredible freedom and flexibility. Combine this with the partial nature of any individual sketch, and the designer has an excellent free-association aid

— every drawn shape results in a set of new shapes defined by perceptual ambiguity. Sketches assembled on planes have a similar perceptual properties, lost in primitive-based digital modelers.

However, the advantages of 3-D representation are also compelling, as it most closely represents the designed artifact. It can be used for visualization, mathematical analysis, consistency checking, and in general as a reality check. 3-D representation has been extensively studied in computer graphics and continues to develop. However, most research concentrates on situations where the 3-D information is already well-known; our goal, in contrast, is to provide a spectrum of analysis and visualization options that gracefully scale with the amount of information (in our case, the amount of 2-D and 3-D geometry) the designer has.

Below, we introduce a new approach to modeling that involves the development and viewing of multiple 2-D concept drawings in a common context. The metaphor is similar to the stacks of vellums that 3-D animators use when drawing motions of sequences of character cells. Placing these vellums in a 3-D context suggests a 3-D object even before it is completely decided upon and ultimately computed. Successive drawing is used to fill in more detail and, with a set of novel editing and annotation techniques described in future chapters, refine a sketch into a resolved 3-D model. The designer can thus smoothly move between 2-D and 3-D representations without losing (or having to trace) earlier context.

7.2 Related Work

Our work is inspired by the image-based rendering methods described in Section 2.5. Those methods transformed 3-D models into sparse 2-D representations for interactive rendering or simplification (such as in Figure 7-1); a surprising result from these works were how few planes were required to represent complex, photorealistic 3-D form. Our technique performs the opposite task: building a 3-D model by accumulating a sparse set of 2-D sketches. Moreover, since we are not building photorealistic form, we can expect an even more radical simplification.

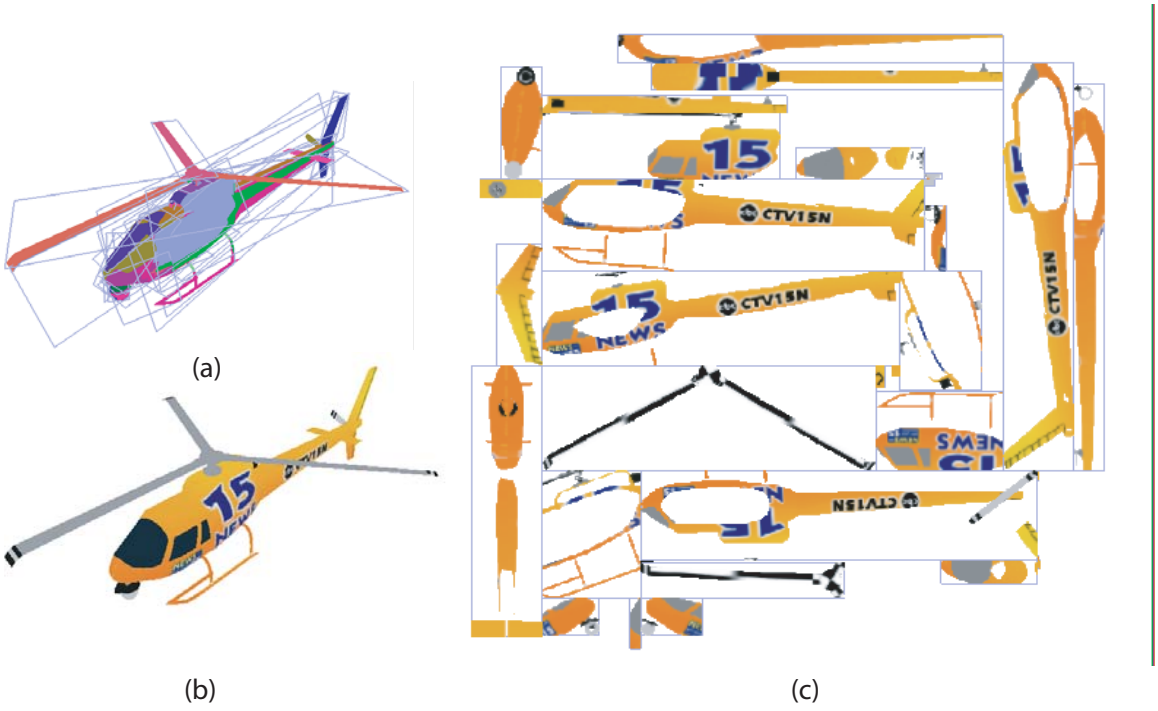

Figure 7-1: Our work is inspired by "billboard clouds," where the system decomposes a 3-D model (a) into planes (c), which it can then re-render from any viewpoint (b). This system can make good approximations with surprisingly few planes; similarly, the user of our system can create evocative 3-D models with few planes. Unlike their system, our goal is not a full 3-D model, but an assembly of sketches in 3-D that suggest, but do not precisely specify, a form to the designer.

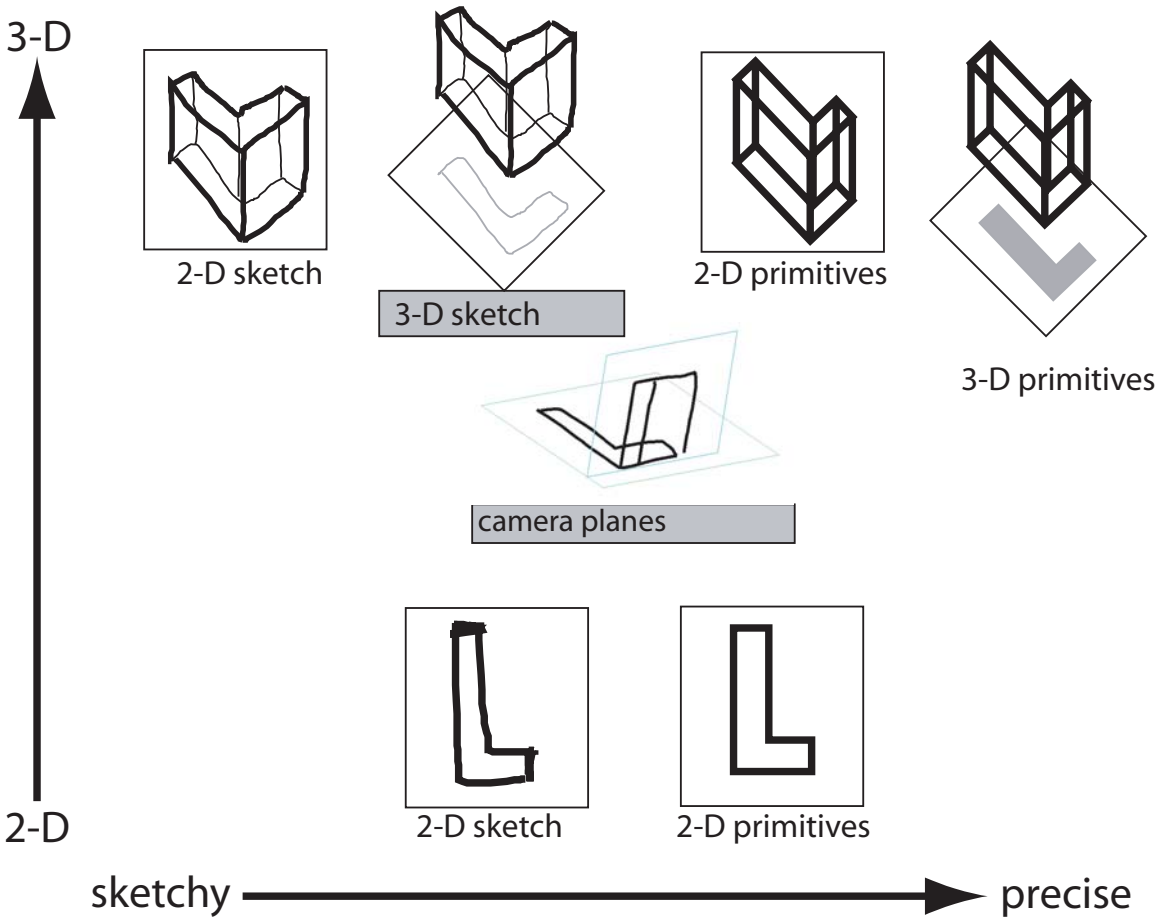

Figure 7-2: Camera planes in the context of other representations. The form shown by a camera planes representation has both 2-D and 3-D qualities, and can be made of sketches or primitives.

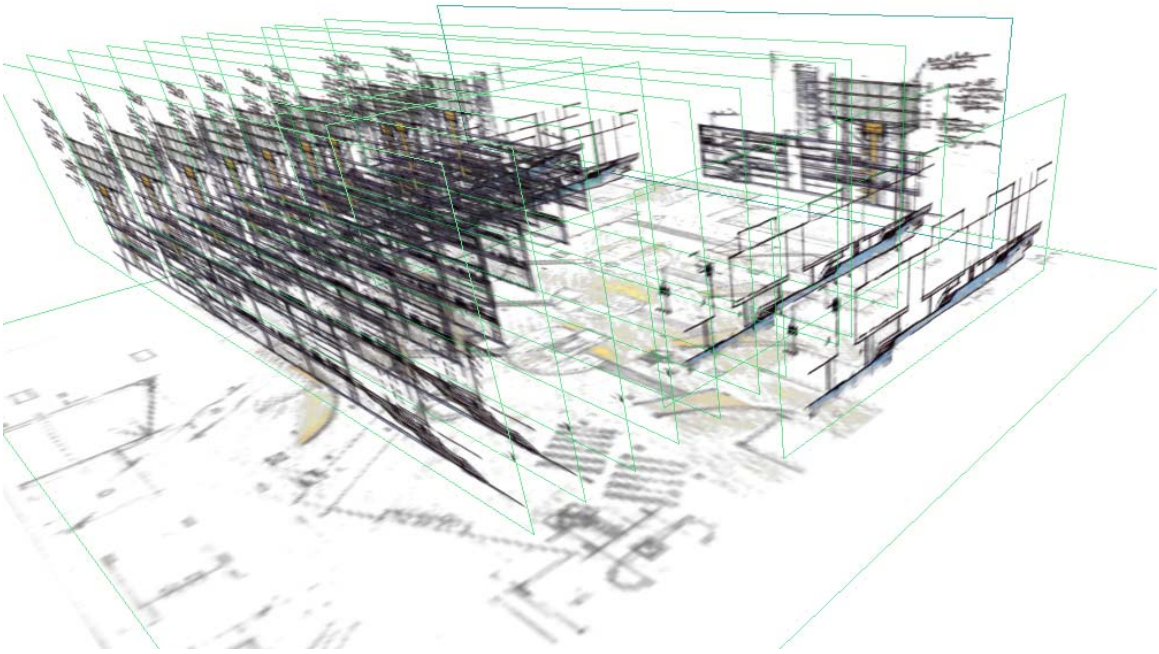

Figure 7-3: 3-D Visualization with 2-D Sketches. Note that these sketches have extra non-3-D information and do not necessarily correspond to a real 3-D model, but simply by placing them in the appropriate context, we can gain a sense of an as-yet-unknown un-worked-out 3-D model.

We place this new representation in context of existing works in Figure 7-2. The form shown by this representation has both 2-D and 3-D qualities, and can be made of sketches or higher-level geometric primitives.

7.3 Exposition

In our approach, arbitrary 3-D planes have associated images (imported) or strokes drawn on them. We provide a simple user interface to create and manipulate planes. The user can create a film plane using the current 3-D camera or copy an existing plane. Direct manipulation widgets then can be used to resize, translate, and rotate them, implicitly transforming the associated camera as well. See Figure 7-4.

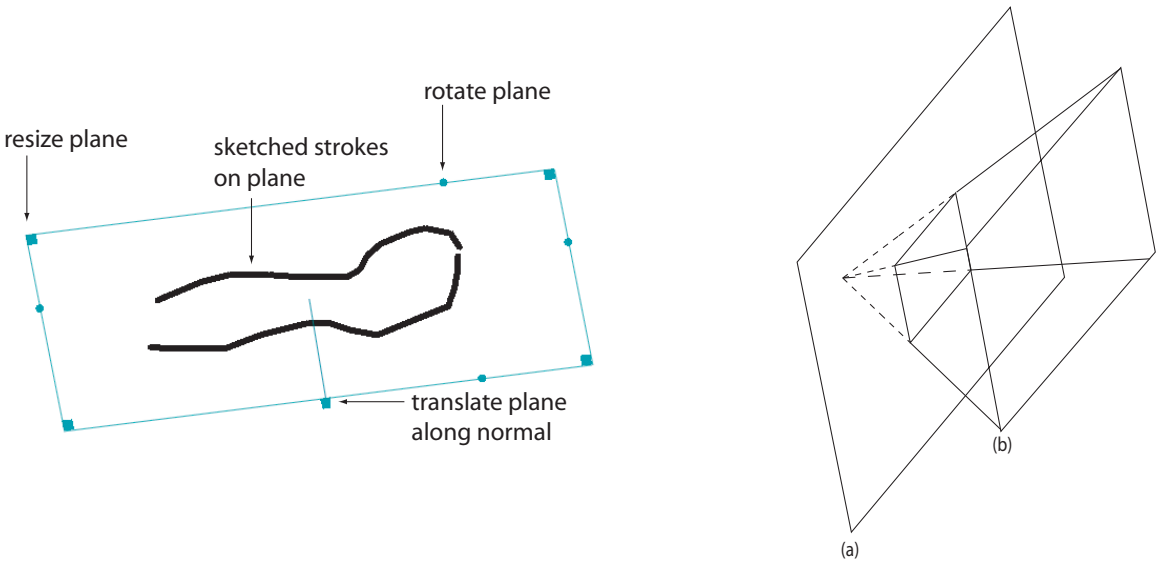

Figure 7-4: On the left, widgets to control the plane. These widgets become larger as the cursor approaches them. On the right, camera planes are a simple combination of a plane (a) and a camera, represented here by a perspective view frustum (b).

Our system provides a unified interface to all three interpretations of planes: slice, geometry, and camera. When initially assembling sketches in 3-D, the "slice" and "geometry" interpretations are most useful. When constructing 3-D geometry, the "camera" interpretation also comes into play.

7.3.1 Slice

The first way a designer can use our plane-based interface is to sketch 3-D slices. Treating planes as slices allows designers to assemble sketches into a 3-D visualization — without having decomposed them into consistent, or 3-D, primitives (see Figure 7-3). In effect, they are hand-crafting a sparse "billboard cloud" [20], a representation of 3-D shapes as agglomerations of images on planes used in real-time rendering. The ability of this representation to represent inconsistency is, for designers, a distinct advantage.

Users can annotate these drawings by simply drawing on the desired plane. They can also manipulate their sketched drawings as standard 2-D objects; line drawings in particular can be conveniently edited with techniques in this thesis: penti-

menti (Chapter 3) or feature-based proxies (Chapter 4). We have also implemented a small subset of image-editing tools found in commercial packages such as Adobe Photoshop[2].

7.3.2 Geometry

Planes can also be considered to be a simple approximation for geometry. In our example model (Figure 7-3), planes are used to approximate the planar sides of the boxcars. Although the images on those planes are technically "sections" or "slices" of the building, in this case we also deploy them as a wireframe outline of the surface. Note that this move also exploits the viewer's ability to conceptualize 3-D more abstractly, since these drawings do not match up with their perpendicular counterparts.

7.3.3 Camera

Since planes are associated with cameras, the user can also move to look through the plane's camera or push geometry off the plane along camera eye rays. This thesis provides two tools to accomplish this: the feature-based editing of Chapter 4 and 1-view reconstruction in Chapter 5). Our two-view reconstruction system (in Chapter 6) triangulates 3-D geometry using corresponding features on two planes, simultaneously taking advantage of the planes' camera position and geometry.

Given a set of assembled sketches as in Figure 7-3, the designer may also choose to draw a new view. In our system, the user simply instantiates a new film plane and begins drawing on it like a 3-D piece of tracing paper.

Finally, we can project 3-D geometry onto a plane using its camera. This allows 3-D geometry that was worked out from one view to be brought into a different 2-D sketch, and perhaps edited in 2-D before re-placing in 3-D.

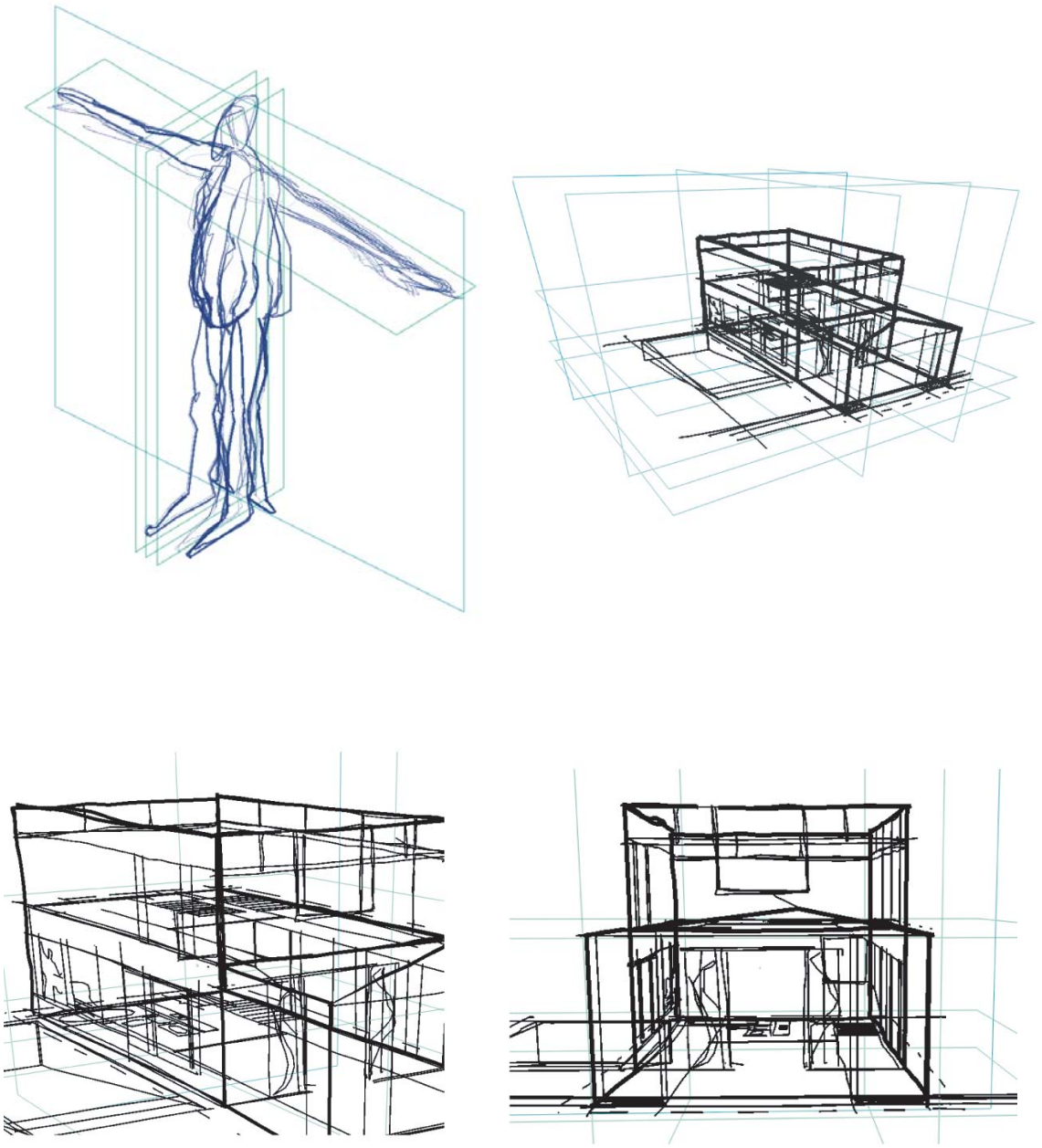

Figure 7-5: On the top left, a designer drew this simple human figure via several cross-sections drawn on planes. On the top right, an architectural student drew this simple house diagram via several cross-sections drawn on planes. Closeups on bottom row.

7.4 Results

A designer drew a human figure (Figure 7-5) with our system. Human form is traditionally difficult to model in 3-D with standard primitives and without any special knowledge in the modeling system, but our user was able to rapidly sketch a convincing 3-D figure with surprisingly few cross sections. The learning curve was quite shallow, as most of the modeling task involved familiar, high-speed sketching.

A designer also drew a simple house plan (Figure 7-5) with the planes interface. The planes map naturally to this architectural geometry, fulfilling their "slice" function. The designer also surprised us, however, by using the planes in other capacities (shown in the bottom row of the figure) without prompting or discussion. The designer drew a human figure on one of the outside walls — depicting how a person inside the building would be seen from the outside — thus treating the plane as a projective view. She also drew curving column elements next to the front door, using the drawing as a planar approximation of a shallow 3-D column.

7.5 Analysis

Design Implications

We are encouraged by our results to think that the planes are a simple interface for users to understand and exploit. The suggestiveness of the models, with so much missing information, is also promising.

Of all the contributions in this thesis, planes seem to be potentially the easiest to integrate into a design process, containing very little overhead over pure 2-D sketching. At the same time, architects will need to develop techniques to best exploit these new suggestive models. Perhaps there are standard configurations of planes that are useful; perhaps it is better to draw 2-D sketches and then import them rather than drawing directly on the 3-D planes. There are probably other useful camera views and planar cuts to use besides the traditional plan and elevation/section (particularly for 3-D curves and curved surfaces), but not every plane orientation and camera placement

is going to be a helpful or meaningful one. It would be helpful to aid the user in choosing these placements. We found the planes to be somewhat awkward for literally describing full 3-D form; some kind of transitioning technique is needed.

Lower-level Improvements

We believe there is significant promise in further extending our UI to allow a sketch or set of sketches to be projected onto arbitrary geometry. The methods of Shum et al. [67] could be used to support using multiple planes as geometry; and those of Reche and Drettakis [59] to handle visibility issues among images. Another useful representation could be the view-dependent rendering of Rademacher [58] — this would allow different views to control a single 3-D model without those views having to precisely agree.

Chapter 8

Conclusion

We have described many novel techniques to explore form, aimed at the early exploration of sketch ideas. Specifically, we have brought the advantages of smooth editing and 3-D visualization to traditional freehand sketches. We introduced a re-modeling interface, "pentimenti," for general curve editing (Chapter 3). We then presented a novel editing interface based on exploring a smooth continuum of form, with "proxies', in Chapter 4. This union allows new types of representation, and suggests new ways to design. First, the concept and construction of a "3-D sketch" was described in Chapters 5 and 6. Second, in Chapter 7, we introduced a user interface, camera planes, to place and edit 2-D and 3-D representations in a common context as well as bootstrap the "remodeling" of old representations by creating new ones. These contributions form a coherent system, where the user can sketch on planes, edit with pentimenti or the proxies, and then reconstruct full 3-D models with the "3-D sketch" creation tools, before returning to further sketching on planes.

We now discuss the whole system and how it could be developed in the future.

8.1 Analysis

In the previous chapters, we noted how each technique needs to be integrated with traditional tools. More user interaction — both with architects, and with other communities such as civil and mechanical engineers — would probably help us in

figuring this out. That said, there are still deep issues that will require more thought.

One set of issues concerns the interactions between multiple representations. This thesis raises the question of when full 3-D representation is needed: the camera planes technique pushes full 3-D to a later point in the design process, while 3-D sketches pull it earlier. A related question is how to accomplish the transition between these new representations and traditional media, whether physical or digital. Finally, how can a designer integrate multiple sketches, both in time (a versioning problem) and in space (an overlay problem) — how to visualize them in context, assemble new models from them, create and show connections between them, propagate changes — an entire vocabulary based on this premise seems possible, at least in theory. This integration differs from the more traditional computer graphics problem of assembling a form from multiple parts, such as assembling a form from some legs here and some arms elsewhere: here the parts might largely not match, nor is it desirable to spend time making them all fit exactly. Trying many messy assemblies would be more useful than a single clean one. An intriguing, and less conventional, example of this combination would be combining a sketch that describes the form of a building's components with that describing the form of its space — both the positive and negative space, also known as the "figure" and the "ground." Both are useful to architects, but cannot currently be designed separately — currently, one must simply be the inverse of the other, which is a necessary condition at the end of the process but not as useful a constraint when exploring the possibilities.

The idea of integrating multiple sketches in time seems particularly fruitful. Good questions include: how to work on a succession of 3-D models with 2-D tools, providing techniques both to start from scratch and to carry over pieces from earlier models; how to visualize a gallery of earlier forms, to discover new patterns and ideas from them; how to support parallel, non-linear iteration, where the user has several active ideas going at once, often with relations between them, in 2-D and in 3-D.

Another line of thought involves a more active, suggestive role for the computer in design; that is, to visualize or come up with alternatives, particularly as a background process while the architect is sketching or otherwise occupied. If the architect is

drawing a 2-D sketch, perhaps the computer could create 3-D models with similar projections; or, as discussed in Chapter 4, perform and show the results of several interesting deformations. It seems plausible that even simple heuristics with little design "smarts" embedded in them could seed an open-minded designer — one able to brainstorm even on bad suggestions — with many ideas.

Further analysis of sketches — perhaps analyzing a corpus of sketches rather than the individual artifacts computer scientists tend to work with — would hopefully reveal a better representation than unannotated images or strokes, or standard geometric primitives like spheres and cubes. Sketched form tends to perform several functions at once — as symbol, as approximate photorealism, as targeted indicators of geometric detail — and finding a representation to express all these elements, and perhaps separate them out for editing, would be excellent. One use for such an analysis would be the re-expression of a 2-D sketch in 3-D. The approaches we presented in Chapters 5 and 6 were predicated on the idea of *preserving* the 2-D appearance in the 3-D artifact; such preservation is not possible when the system has to generate new strokes (such as when rendering smooth surfaces with constantly changing contours), and perhaps not *desirable* either, as 2-D sketches tend to have viewpoint- and task-dependent highlighting and simplification (which also reduces visual clutter and complexity) which might be inappropriate for a 3-D model where multiple views are available. Some sort of description of these pictorial goals, suitable for retargeting to either 2-D or 3-D renderings, would be useful for designers.

A related issue to the representation of sketches is the simultaneous visualization of sketches with some sort of sketch interpretation. In our system this primarily involved the visualization of the proxy vertex selections; the problem was that the proxy visualization overwhelmed the sketch. It seems to us that the importance of each type of representation varies throughout the design process, so some sort of user control would be most appropriate, to vary the visual importance of each type of element as needed.

8.2 Discussion

Architecture and computer graphics (and computer vision) have much to gain from greater dialogue. Although modeling is of general interest to computer graphics researchers, the process developed by architects over hundreds of years is not an intuitive and obvious one to infer. The process of design largely remains hidden from them and from the public eye in general; a particularly glaring example are the thousands of sketches that are thrown out every day, or placed in storage, never to be seen again. Greater knowledge about this process could introduce interesting, deep problems to the modeling field. On the design side, computers have certainly revolutionized design for the better: the drudgery of copying drawings and the physical demands of making exact, clean drawings has largely been done away with. The results of design have also improved in some ways: the 3-D visualization powers of computers have supported entirely new types of construction (as in the work of Frank O. Gehry and Associates) as well as more environmentally efficient buildings made possible by mathematical analysis of 3-D digital models (as in the work of Norman Foster and Partners). Many architects are extremely interested in the possibilities of digital tools to further revolutionize not only construction but the design process itself. A conversation awaits.

Similarly, the computer vision and computer graphics communities, already beginning to talk, have much more to learn from each other. Computer graphics focuses on generative methods, on synthesis; computer vision focuses on inference and understanding. In design, these tasks form a virtuous circle: understanding aids synthesis which aids understanding, and so forth. Sketching itself demonstrates this process: viewing a new sketch typically results in a new understanding of the nascent form, which results in a new sketch. Additionally, the extreme compactness and efficiency of a sketch is due to the sketchers' understanding of the missing information that a viewer will be able to infer. Tools which assist designers with both tasks in a similarly integrated way would probably be very good for designers.

In the user-interface field, we believe an important challenge to researchers is not

simply to work toward "blank-screen" or "invisible" interfaces that automatically do what the user wants, but to recognize that such an interface will always be relying on assumptions that may not always be true. Typically interfaces take either a completely inference-free, "white-box" approach (as with the "direct manipulation" and "direct specification" interfaces of Chapter 2) or utilize a large number of inferences (as with the reconstruction work cited in Section 5.1) to present a very simple ''black-box" interface. Manual interfaces will always be necessary, as long as the assumptions inherent in more automatic interfaces do not always hold — in the worst case, the system will need fully manual input. It would be thus more valuable for a user interface to smoothly scale with the amount of assumptions. As such, the challenge is to articulate the assumptions inherent in models of human behavior in such a way that users can view and override them — and in the process, perhaps contributing to the understanding of said assumptions, reducing human drudgery even more. In this arena we believe our one-view reconstruction interface (Chapter 5) is most successful, presenting the user with a sequence of steps which have both an automatic "first-guess" component and an interface for manual override.

Appendix A

Prototype Details

All the features in this thesis were wrapped into one monster program, with plenty of different user interaction modes. We show a snapshot in Figure A-1.

Our prototype was written in a mixture of Tcl/Tk (for a portable, easily changed interface to GUI elements) and C++ with STL. STL is immensely handy but (unless you're careful, which I was not, at least at first) allocates many small memory blocks, which cause unpredictable stops and stutters in event handlers. The program could be used with a mouse or with a Tablet PC or Wacom tablet, both of which support much higher-resolution (100x!) pen input events, as well as providing pen pressure information. For the linear algebra we used the GNU Scientific Library (GSL), free and available on the Web. For graphics support we used basic OpenGL. The user strokes were drawn with polygons whose screen-space width was determined by the vertex's width but whose position was determined by the 3-D location of the vertex - an awkward combination that necessitated doing the projection in software; perhaps newer shader languages could do this in hardware. The code certainly runs on Windows, and should still compile on Linux and Solaris, although we have not tried for several months.

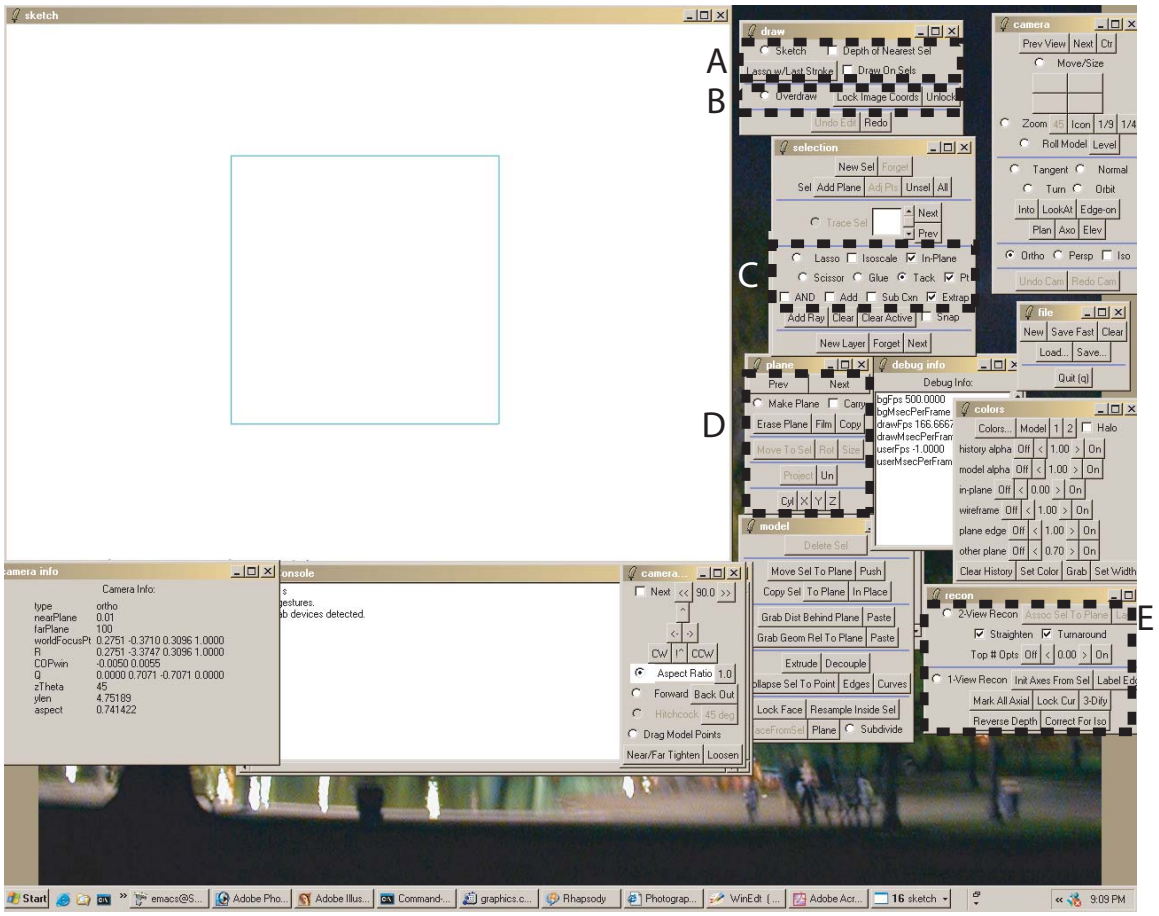

Figure A-1: A full screenshot of our prototype. The user's actual workspace, from where the rest of the screen captures in this thesis were taken, is the window in the upper left. In general, all of these windows would not be shown at the same time; the user would only keep open the necessary ones for the current task, as with the floating toolbars in Adobe Photoshop or Illustrator. The interface for freehand sketching on planes is shown in (A), with various options for adding the new stroke to existing proxy vertices (termed 'lasso' in the interface). The pentimenti interface is activated with the 'Overdraw' button in (B). The proxy interface is accessed through the widgets shown in (C); that window also contains other widgets to create proxy selections and layers. Widgets to cycle through the camera planes, copy and erase them, and other functions are presented in (D). The 1-view and 2-view reconstruction interfaces are in (E). This screenshot shows many extra buttons whose functionality did not make it into this thesis. Other windows, such as the 'camera' and 'file' windows, implement standard modeling functionality.

Bibliography

[1] Adobe. Illustrator 10.0. Commercially available software, 2004.

[2] Adobe. Photoshop 7.0. Commercially available software, 2004.

[3] Alias. Maya Unlimited 6.0. Commercially available software, 2004.

[4] Auto-des-sys. FormZ 4.2. Commercially available software, 2004.

[5] Autodesk. AutoCAD 2005. Commercially available software, 2004.

[6] Autodesk. Revit 6. Commercially available software, 2004.

[7] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54. ACM Press, 1998.

[8] William A. Barrett and Alan S. Cheney. Object-based image editing. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 777–784. ACM Press, 2002.

[9] Thomas Baudel. A mark-based interaction paradigm for free-hand drawing. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 185–192. ACM Press, 1994.

[10] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 35–42. ACM Press, 1992.

[11] David Bourguignon, Raphaelle Chaine, Marie-Paule Cani, and George Drettakis. Relief: A modeling by drawing tool. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 151–160. Eurographics Association, 2004.

[12] M. J. Brooks and B. K. P. Horn. *Shape from Shading*. MIT Press, Cambridge, MA, 1989.

[13] F J Canny. A computational approach to edge detection. *IEEE Trans PAMI*, 8(6):679–698, 1986.

[14] Roger H. Clark and Michael Pause. *Precedents in Architecture*. Wiley, April 1996.

[15] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. An interface for sketching 3d curves. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 17–21. ACM Press, 1999.

[16] Pedro Company, Ana Piquer, and Manuel Contero. On the evolution of geometrical reconstruction as a core technology to sketch-based modeling. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 97–106. Eurographics Association, 2004.

[17] Alexander Davis. Plan for vmi faculty residence. VMI Web site (http://www.vmi.edu/archives/images/ms27610.jpg), July 2004. Image taken on 5/21/04.

[18] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 11–20, August 1996.

[19] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Trans. Graph.*, 22(3):848–855, 2003.

[20] Xavier Décoret, Frédo Durand, François Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *Proceedings of the ACM Siggraph*. ACM Press, 2003.

[21] Xavier Decoret, François Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3), 1999.

[22] Frédo Durand. An invitation to discuss computer depiction. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 111–124. ACM Press, 2002.

[23] Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling strokes and high-level attributes for interactive traditional drawing. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 71–82. Springer-Verlag, 2001.

[24] Lynn Eggli, Beat D. Brüderlin, and Gershon Elber. Sketching as a solid modeling tool. In *Proceedings of the third ACM symposium on Solid modeling and applications*, pages 313–322. ACM Press, 1995.

[25] Eos. Photomodeler 5.0. Commercially available software, 2004.

[26] Adam Finkelstein and David H. Salesin. Multiresolution curves. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 261–268. ACM Press, 1994.

[27] Timo Fleisch, Florian Rechel, Pedro Santos, and André Stork. Constraint stroke-based oversketching for 3d curves. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 161–166. Eurographics Association, 2004.

[28] Andreas Genau and Axel Kramer. Translucent history. In *Conference companion on Human factors in computing systems*, pages 250–251. ACM Press, 1995.

[29] Michael Gleicher and Andrew Witkin. Differential manipulation. In *Graphics Interface '91*, pages 61–67, 1991.

[30] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452. ACM Press, 1998.

[31] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François Sillion. Programmable style for NPR line drawing. In *Rendering Techniques 2004 (Eurographics Symposium on Rendering)*. ACM Press, june 2004.

[32] Arthur Gregory, Andrei State, Ming Lin, Dinesh Manocha, and Mark Livingston. Feature-based surface decomposition for polyhedral morphing. Technical Report TR98-014, Department of Computer Science, University of North Carolina - Chapel Hill, April 14 1998.

[33] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *ACM Trans. Graph.*, 23(3):522–531, 2004.

[34] Tracy Hammond and Randall Davis. Automatically transforming symbolic shape descriptions for use in sketch recognition. In *Proceedings of the The Nineteenth National Conference on Artifical Intelligence (AAAI-04)*, July 2004.

[35] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521623049, 2000.

[36] Herman Hertzberger. *Studies by Herman Hertzberger*. 010, Rotterdam, Netherlands, 1996.

[37] Youichi Horry, Ken-Ichi Anjyo, and Kiyoshi Arai. Tour into the picture: using a spidery mesh interface to make animation from a single image. In *Proceedings*

of the 24th annual conference on Computer graphics and interactive techniques, pages 225–232. ACM Press/Addison-Wesley Publishing Co., 1997.

[38] Takeo Igarashi and John F. Hughes. A suggestive interface for 3d drawing. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 173–181. ACM Press, 2001.

[39] Takeo Igarashi, 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*, pages 409–416. ACM Press/Addison-Wesley Publishing Co., 1999.

[40] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. WYSIWYG NPR: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, July 2002.

[41] Olga Karpenko, John F. Hughes, and Ramesh Raskar. Epipolar methods for multi-view sketching. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 167–174. Eurographics Association, 2004.

[42] James A. Landay. SILK: sketching interfaces like krazy. In *Conference companion on Human factors in computing systems*, pages 398–399. ACM Press, 1996.

[43] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000.

[44] David Liebowitz, Antonio Criminisi, and Andrew Zisserman. Creating architectural models from images. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 39–50. The Eurographics Association and Blackwell Publishers, 1999.

[45] H. Lipson and M. Shpitalni. Correlation-based reconstruction of a 3d object from a single freehand sketch. In *Proceedings of the AAAI Spring Symposium Series–Sketch Understanding*, 2002.

[46] Pin-Chou Liu, Fu-Che Wu, Wan-Chun Ma, Rung-Huei Liang, and Ming Ouhyoung. Automatic animation skeleton construction using repulsive force field. In *Proceedings of Pacific Graphics 2003*, pages 409–413. IEEE, 2003.

[47] Sujin Liu and Zhiyong Huang. Interactive 3d modeling using only one image. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 49–54. ACM Press, 2000.

[48] David Lowe. Three-dimensional object recognition from single two-dimensional images. *Artificial Intelligence*, 31(3):355–395, 1987.

[49] Leonard McMillan and Gary Bishop. Plenoptic modeling: an image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46. ACM Press, 1995.

[50] K. Mikolajczyk and C. Schmid. An affine invariant interest point detector. In *ECCV (1)*, pages 128–142, 2002.

[51] Jun Mitani, Hiromasa Suzuki, and Fumihiko Kimura. *3D sketch: sketch-based model reconstruction and rendering*, pages 85–98. Kluwer Academic Publishers, 2002.

[52] Alex Mohr, Luke Tokheim, and Michael Gleicher. Direct manipulation of interactive character skins. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 27–30. ACM Press, 2003.

[53] Henry P. Moreton and Carlo H. Séquin. Functional optimization for fair surface design. In *Proceedings of the 19th annual conference on Computer graphics*, pages 167–176. ACM Press, 1992.

[54] Byong Mok Oh, Max Chen, Julie Dorsey, and Frédo Durand. Image-based modeling and photo editing. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 433–442. ACM Press, 2001.

[55] Theo Pavlidis and Christopher J. Van Wyk. An automatic beautifier for drawings and illustrations. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 225–234. ACM Press, 1985.

[56] João P. Pereira, Vasco A. Branco, Joaquim A. Jorge, Tiago D. Silva, Nelson F.and Cardoso, and F. Nunes Ferreira. Cascading recognizers for ambiguous calligraphic interaction. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 63–72. Eurographics Association, 2004.

[57] Pierre Poulin, Mathieu Ouimet, and Marie-Claude Frasson. Interactively modeling with photogrammetry. In *Proceedings of Eurographics Workshop on Rendering 98*, pages 93–104, June 1998.

[58] Paul Rademacher. View-dependent geometry. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 439–446. ACM Press/Addison-Wesley Publishing Co., 1999.

[59] Alex Reche and George Drettakis. View dependent layered projective texture maps. In J. Rokne, R. Klein, and W. Wang, editors, *Proceedings of Pacific Graphics 2003*, pages 492–296. IEEE Press, October 2003.

[60] L. Ros and F. Thomas. Correcting polyhedral projections for scene reconstruction. In *IEEE International Conference on Robotics and Automation*, May 2001.

[61] Eric Saund and Thomas P. Moran. A perceptually-supported sketch editor. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 175–184. ACM Press, 1994.

[62] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 151–160. ACM Press, 1986.

[63] Amit Shesh and Baoquan Chen. Smartpaper: An interactive and user friendly sketching system. In M.-P. Cani and M. Slater, editors, *Computer Graphics Forum (Eurographics '04)*, volume 23(3). The Eurographics Association and Blackwell Publishers, 2004.

[64] Michael Shilman and Paul Viola. Spatial recognition and grouping of text and graphics. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 91–96. Eurographics Association, 2004.

[65] M. Shpitalni and H. Lipson. Classification of sketch strokes and corner detection using conic sections and adaptive clustering. *Trans. of ASME J. of Mechanical Design*, 119(2):131–135, 1997.

[66] H. Y. Shum, M. Han, and R. Szeliski. Interactive construction of 3d models from panoramic mosaics. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, page 427. IEEE Computer Society, 1998.

[67] Heung-Yeung Shum, Jian Sun, Shuntaro Yamazaki, Yin Li, and Chi-Keung Tang. Pop-up light field: An interactive image-based modeling and rendering system. *ACM Trans. Graph.*, 23(2):143–162, 2004.

[68] Saul Simhon and Gregory Dudek. Pen stroke extraction and refinement using learned models. In J.F. Hughes and J.A. Jorge, editors, *Proceedings of the 2004 Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM-04)*, pages 73–80. Eurographics Association, 2004.

[69] Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414. ACM Press, 1998.

[70] Gilbert Strang. *Linear Algebra and its Applications*. Saunders College Publishing, Fort Worth, 3rd edition, 1988.

[71] Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. In *Proceedings of the 31st annual conference on Computer graphics and interactive techniques*, pages 399–405. ACM Press, 2004.

[72] Masaki Suwa, John Gero, and Terry Purcell. Unexpected discoveries and s-invention of design requirements: Important vehicles for a design process. *Design Studies*, 21(6):539–567, 2000.

[73] Masaki Suwa and Barbara Tversky. Constructive perception: A metacognitive skill for coordinating perception and conception. In *Cog Sci*, 2003.

[74] Masaki Suwa, Barbara Tversky, John Gero, and Terry Purcell. Seeing into sketches: regrouping parts encourages new interpretations. In J.S. Gero, B. Tversky, and T. Purcell, editors, *Visual and Spatial Reasoning in Design II*, pages 207–219. Academic Press, Australia, 2001.

[75] Dassault Systemes. Catia v5r13. Commercially available software, 2004.

[76] Matthew Thorne, David Burke, and Michiel van de Panne. Motion doodles: An interface for sketching character motion. In *Proceedings of the ACM Siggraph*. ACM Press, 2004.

[77] Osama Tolba, Julie Dorsey, and Leonard McMillan. A projective drawing system. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 25–34. ACM Press, 2001.

[78] Steve Tsang, Ravin Balakrishnan, Karan Singh, and Abhishek Ranjan. A suggestive interface for image guided 3d sketching. In *Proceedings of the 2004 conference on Human factors in computing systems*, pages 591–598. ACM Press, 2004.

[79] Victoria and Albert Museum. http://www.vam.ac.uk/exploring/collections/paintings/paintings_galleries/techniques/underdrawing/. Museum Web site, July 2004. Images taken on 7/22/04.

[80] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 247–256. ACM Press, 1994.

[81] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Computer Graphics*, 28(Annual Conference Series):91–100, 1994.

[82] Shin Yoshizawa, Alexander G. Belyaev, and Hans-Peter Seidel. Free-form skeleton-driven mesh deformations. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 247–253. ACM Press, 2003.

[83] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: an interface for sketching 3d scenes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 163–170. ACM Press, 1996.

[84] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive multiresolution mesh editing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 259–268. ACM Press/Addison-Wesley Publishing Co., 1997.