Druid : Representation of Interwoven Surfaces in $2^1/_2$D Drawing

by

Keith Wiley

B.A., Psychology, University of Maryland College Park, 1997
M.S., Computer Science, University of New Mexico, 2003

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2006

©2006, Keith Wiley

Dedication

For my parents, who taught me to love science.

For Aleta, who inspired me.

For Angie, who always believed in me.

Acknowledgments

To begin, I would like to thank my committee: Lance R. Williams (my advisor), George Luger, Ed Angel, and Michael Cook. Their commitment of time, suggestions, and feedback improved my experimentation and exposition in many ways. Lance quickly recognized my strengths while helping me overcome my weaknesses. His assistance was critical in familiarizing myself with the background work that lay the foundation for my work. Aside from Ph.D. research, Lance and I have also had many discussions on topics either unrelated or only peripherally related to my work. I will always value our brainstorming sessions on topics as diverse as vision, perception, image processing, and topology.

Lynne Jacobson, the computer science department's graduate advisor, was of great help in straightening out a number of technical problems pertaining to my transcript and other various paperwork. Thank you.

I would like to acknowledge the importance of having a circle of supportive friends, especially fellow graduate students who can empathize as no others with the tribulations of getting a Ph.D. The most influential of these friends are Terry Van Belle, Ben Andrews, Jen Andrews, Aaron Clauset, James Horey, and Alison Boyer. Many others rightfully belong on such a list as well.

I must also briefly acknowledge my lifelong friends from high school: Jen, Stacey, and Anne. Few friendships last so long. Cherish them.

My family is of immense importance to me. My dad has always been proud of my individual accomplishments while simultaneously encouraging me to take each project one step further. My mom was exuberant the from the day I started grad school to the day I finished. My sister, Aleta, holds a special place in my heart. She has always been my muse, my single greatest inspiration. In her I see all the things I wish I was. She is dedicated, hard-working, insightful, and frightfully clever. She has always looked up to me. What she may not realize is how much I have always looked up to her.

Finally, I conclude with my thanks to Angie. She has exhibited unfailing confidence in my eventual success and has stood by me through the difficult times. Most importantly, she was simply there, day after day, helping me press forward. Thank you Angie.

Druid : Representation of Interwoven Surfaces in $2^1/_2$D Drawing

by

Keith Wiley

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2006

Druid :

Representation of Interwoven Surfaces in $2^1/_2$D Drawing

by

Keith Wiley

B.A., Psychology, University of Maryland College Park, 1997
M.S., Computer Science, University of New Mexico, 2003

Ph.D., Computer Science, University of New Mexico, 2006

Abstract

The state-of-the-art in computer drawing programs is based on a number of concepts that are over two decades old. One such concept is the use of layers for ordering the surfaces in a *$2^1/_2$D* drawing from top to bottom. A $2^1/_2$D drawing is a drawing that depicts surfaces in a fundamentally two-dimensional way, but also represents the relative depths of those surfaces in the third dimension. Unfortunately, the current approach based on layers unnecessarily imposes a partial ordering on the depths of the surfaces and prevents the user from creating a large class of potential drawings, *e.g.,* of Celtic knots and interwoven surfaces.

The first half of this dissertation describes a novel approach which only requires local depth ordering of segments of the boundaries of surfaces in a drawing rather than a global depth relation between entire surfaces. Our program provides an intuitive user interface with a fast learning curve that allows a novice to create complex drawings of interwoven surfaces that would be extremely difficult and time-consuming to create with standard drawing programs.

The second half of this dissertation describes a previously unrealized topological property of $2\frac{1}{2}$D scenes. Knowledge of this property makes possible the design of algorithms for manipulating $2\frac{1}{2}$D representations in a way that is isomorphic to elemental $2\frac{1}{2}$D scene changes. Furthermore, this property can be exploited to vastly improve the performance of a $2\frac{1}{2}$D scene editor.

Contents

List of Figures

Glossary

affordances The set of interactions that an object or a user interface permits. For example, a doorknob *affords* rotation while a door on hinges affords pushing or pulling.

boundary The one-dimensional closed perimeter of a surface. In *Druid*, boundaries are defined using closed B-splines.

boundary segment The contiguous region of a boundary lying between two crossings on that boundary. If a boundary has no crossings, the entire boundary is a single boundary segment.

boundary segment depth The depth index assigned to a particular boundary segment, where the depth corresponds to the number of surfaces that lie between the viewer and that segment. A segment that lies on top of the drawing has a depth of zero.

constraint-propagation A method of graph-labeling by which certain combinatorially complex problems can be reduced to unique solutions. When one vertex of a graph is labeled, this constrains adjacent vertices, which in turn propagate their own constraints deeper into the graph. By means of this process, it is often the case that an apparently ambiguous labeling problem can be reduced to a single consistent labeling.

crossing A location where two boundaries intersect.

crossing-flip The user-interaction in which a use clicks on a crossing to invert the crossing's state. This interaction is used to invert the relative depth ordering of the two surfaces that overlap at the crossing.

crossing-projection The process of computing the new locations of the crossings of a drawing after a boundary is moved.

crossing-state An indication of which surface is above and which is below at the crossing of two boundaries.

crossing-state equivalence class A group of crossings that are constrained to flip as a unit.

cut A line connecting two different locations on one (or two) boundaries. When connecting two boundaries, a cut joins the two boundaries into a single boundary. When connecting two locations on the same boundary, a cut separates the boundary's surface into two abutting surfaces.

cut-chain An alternating sequence of boundaries and cuts that joins the two boundaries at each end.

Druid A vector-based drawing program which uses an interwoven representation and provides a user interface which has affordances that are isomorphic to those of idealized physical surfaces.

Druid (*SEARCH*) A mode of functionality in which *Druid* finds a new labeling by performing a tree search.

Druid (*CSEC SEARCH*) A mode of functionality in which *Druid* uses *crossing-state equivalence classes* to vastly reduce the size of the search space during a labeling tree search.

Druid (*DIRECT*) A mode of functionality in which *Druid* uses *crossing-state equivalence classes* to directly relabel a drawing following a *crossing-flip* without performing a tree search.

finite surface A surface bounded by a user-specified exterior boundary and which therefore has a finite area.

fully connected boundary A boundary which is joined by cuts or cut-chains to every other boundary bounding the same surface.

general 2½D scene A 2½D scene in which surfaces may have multiple boundary components and may self-overlap.

holeless 2½D scene A 2½D scene in which all surfaces are defined by a single boundary component (such surfaces do not contain *holes*), but in which surfaces may self-overlap.

infinite surface A surface with no user-specified exterior boundary but which has an implicit exterior boundary at an infinite distance. Thus, it has an infinite area.

interwoven surfaces A pair of surfaces where one surface is above the other in one location but is below the other in a different location. Interwoven surfaces cannot be represented using a layered representation because there is no global depth order relation.

knot-diagram A projection of a set of closed curves onto a plane together with indications which show which curve is on top at every crossing.

labeled knot-diagram A knot-diagram with a sign of occlusion for every boundary and a depth index for every boundary segment.

labeling The state of a particular labeled knot-diagram consisting of a unique assignment of signs of occlusion for every boundary, crossing-states for every crossing, and boundary segment depths for every boundary segment.

labeling scheme A set of localized constraints on the depths of the four segments that meet at a crossing. The labeling scheme must be satisfied at every crossing in a drawing in order for the drawing to be rendered.

labeling search A tree search that *Druid* automatically performs after topology-changing events occur. The purpose of the labeling search is to find a legal labeling for the current drawing so that it can be rendered.

layered 2½D scene A 2½D scene in which the global relative surface depth relation conforms to a directed acyclic graph.

legal labeling A labeling in which all crossings satisfies the labeling scheme. Legal labelings can be rendered.

non-self-overlapping 2½D scene A 2½D scene in which no surface self-overlaps, but in which surfaces may have multiple boundary components, *i.e.*, may contain holes.

partially connected boundary A boundary which is joined by cuts or cut-chains to some of the other boundaries bounding the same surface.

rendering The process of generating a final image of a 2½D scene. In a rendered image a color is assigned to each region that represents the color and opacity properties of the surfaces covering that region.

region A contiguous area of the canvas that is bounded by boundary segments, *i.e.*, a disjoint partitioning of the canvas.

sign of occlusion The designation of which side of a boundary is bounding, *i.e.*, which side of a boundary the surface lies on.

simple 2½D scene A 2½D scene in which all surfaces are defined by single boundary components that do not cross themselves, *i.e.*, Jordon curves.

slice A line connecting a location on a boundary to a point in the interior of the bounded surface.

spoof A configuration of surfaces which does not interweave, but which is constructed such that the rendered image presents the illusion of interwoven surfaces.

surface A two-manifold with boundary.

2½D scene A scene of surfaces which is fundamentally two-dimensional but which also represents the relative depths of surfaces in the third dimension, *i.e.*, the absolute position of a surface in the third dimension is not indicated, but the relative depths of pairs of surfaces are.

unconnected boundary A boundary which is not joined by cuts or cut-chains to any other boundary bounding the same surface.

Chapter 1

Introduction

1.1 Drawing Programs

Drawing programs originated with Sutherland's seminal Ph.D. thesis in 1963, in which many recognizable components of modern drawing programs were already present (see Sutherland [40] and Sutherland [41]). Since then, a number of refinements have been made to the general design of drawing programs, aided primarily by increased computing power and hardware innovations such as the mouse. In 1984, Apple released *LisaDraw 3.0* (see Craig [14]), which despite its age, is effectively a modern drawing program. It uses a tool palette that is similar to the tool palettes of modern drawing programs and provides similar functionality for the construction and manipulation of shapes to that of modern drawing programs. Most importantly, its underlying representation for determining the relative depths of surfaces has not been improved upon in twenty years, despite the weaknesses of that representation. For the last twenty years, research on drawing programs has focused on areas other than the underlying representation. For example, a considerable amount of research has focused on methods for constructing objects and manipulating their shape, but once constructed, those objects are assigned to depths in a conventional layer-based drawing representation. One area in which considerable progress has been made is the intelligent interpretation of *sketches*, *i.e.*, imprecise hand drawings which are analyzed to discover salient features. For example, *CorelDRAW 12*, a professional drawing program, provides a *smart drawing tool*, which allows a user to freehand draw approximate shapes that are recognized and fitted to stock shapes such as ellipses (see Corel [13]). Barla, et al. describe a method of line drawing simplification which either removes or merges extraneous lines without losing defining features of the drawing (see Barla et al. [8]). Gangnet et al. describe a method for closing gaps in freehand curves, which is important for paint-fill algorithms (see Gangnet et al. [16]). There has also been a fair amount of work on aligning the relative positions of objects on the canvas, such as *CorelDRAW 12*'s *dynamic guides*, which are temporary guides for aligning graphic objects to each another (see Corel [13]) and Raisamo and Raiha's work on object alignment using direct manipulation interfaces (see Raisamo and Raiha [35, 36]). Direct manipulation interfaces are discussed in Section 2.3. In addition, Raisamo describes a novel way to construct shapes in which a block is chiseled, similar to three-dimensional sculpting, rather than defined using explicit boundaries such as geometric primitives and splines (see Raisamo [37]). While the previous work on drawing programs is significant and valuable, it all relies on a layer-based representation for the relative depths of surfaces, and this representation has serious limitations.

One function of a drawing program is to allow the construction and manipulation of drawings of overlapping surfaces, which we simply call *2½D scenes*. A 2½D scene is a scene of surfaces that is fundamentally two-dimensional, but which also represents the relative depths of those surfaces in the third dimension, *i.e.*, the absolute position of a surface in the third dimension is not indicated, but the relative depths of pairs of surfaces are. Using existing programs, a drawing can easily be constructed in which multiple surfaces partially overlap. When multiple surfaces overlap, the program must have a means of representing which surface is on top for each area where two surfaces overlap. Existing drawing programs solve this problem by representing drawings as a set of distinct layers where each surface resides in a single layer. For any given pair of surfaces, the one that resides in the upper (or shallower) layer is assigned a smaller depth index and appears above wherever those two surfaces overlap. Consequently, the use of layers imposes a partial ordering, or a directed acyclic graph (DAG), on the surfaces such that no subset of surfaces can interweave (Fig. 1.1). This restriction precludes many common drawings which a user may wish to construct (Fig. 1.2). Specifically, a layered representation precludes the construction of scenes that contain *interwoven surfaces, i.e.*, pairs of surfaces for which each surface is above the other somewhere in the scene. Because such programs do not span the full space of possible 2½D scenes, they therefore impose limitations on the drawings that a user can create.

Our research uses a more general representation as the basis for a more powerful drawing tool, called *Druid*. *Druid* eliminates the assumption that surfaces cannot interweave. It therefore spans a larger space of 2½D scenes by using a representation that makes weaker assumptions about the drawing. This generality makes *Druid* a more versatile drawing tool.

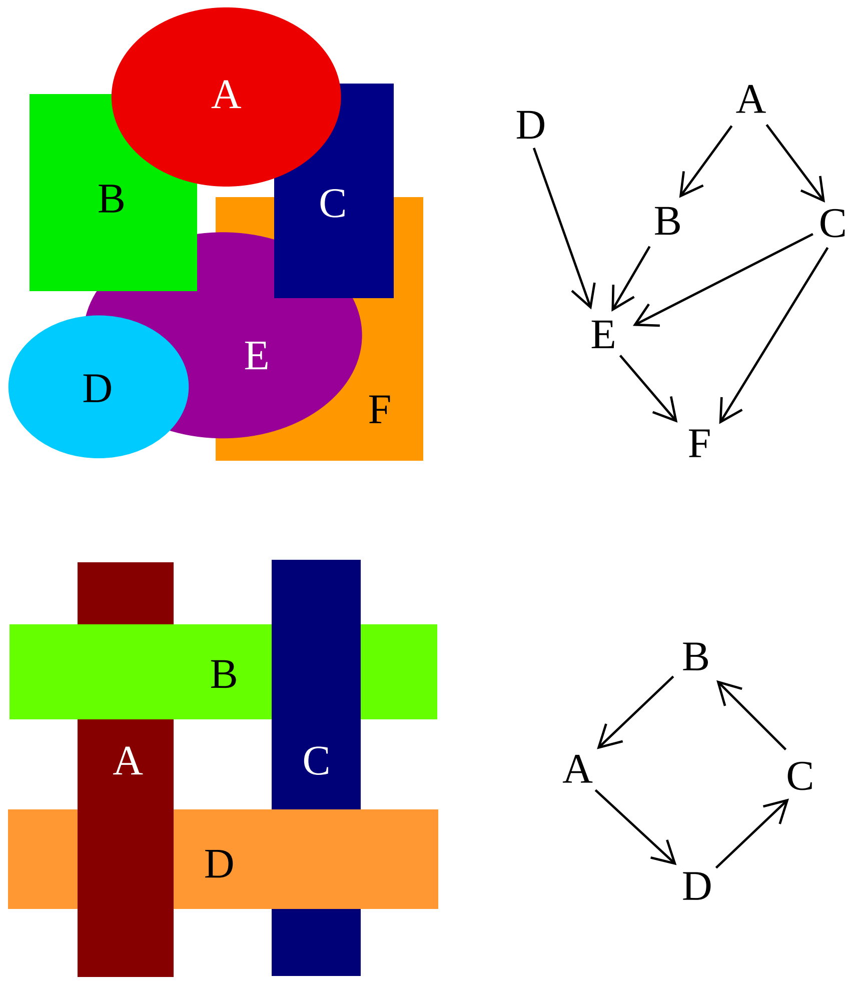

Figure 1.1: The classic approach to representing relative surface depths is to assign the surfaces to distinct layers (top left). It follows that the surface relative depth relation is a directed acyclic graph (DAG). No subset of surfaces can interweave because this would require a cycle in the graph (top right). This approach precludes interwoven drawings (bottom left) in which the surface relative depth relation has cycles (bottom right).

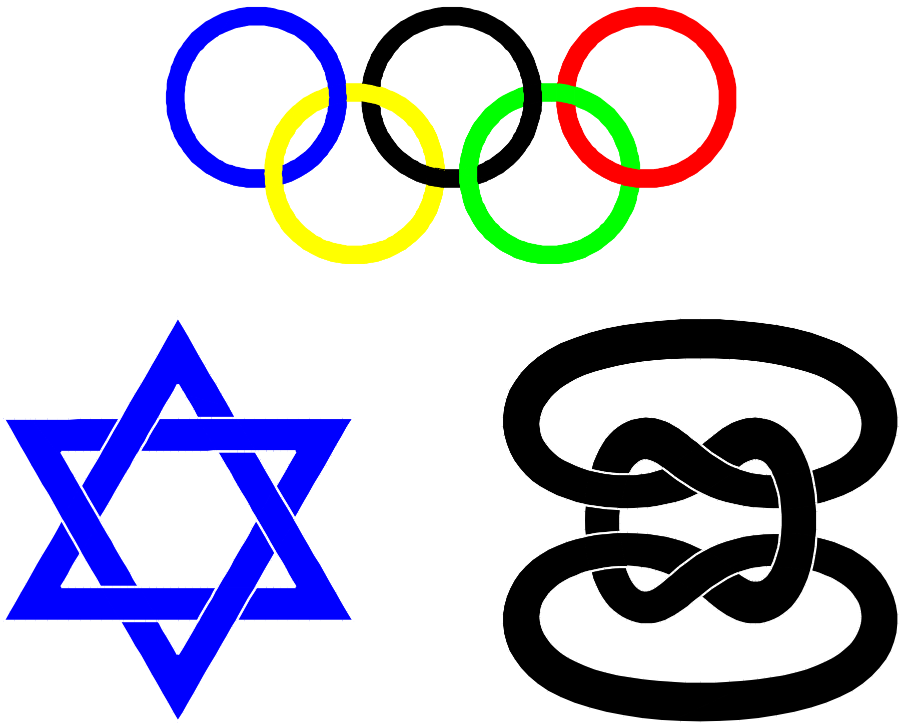

Figure 1.2: *Druid* permits the construction of drawings of interwoven surfaces, such as those shown here.

1.2 $2^1/_2$D Scenes

The term *$2^1/_2$D* is used in different ways in the literature. In particular, it is used differently when describing three-dimensional modeling and when describing the psychophysics of vision. For the purposes of our research we use the definition coined by Marr (see Marr [25]). According to Marr, a $2^1/_2$D sketch is a representation of surfaces that is fundamentally two-dimensional, but which also represents surface orientations and the relative depths of those surfaces in the third dimension.

1.3 Comparison Between Drawing and Three-Dimensional Modeling

Since the purpose of a 2½D drawing program is to construct drawings that contain some information about the third dimension, one might assume that 2½D drawing is effectively three-dimensional modeling, in which a description of a three-dimensional model is constructed in a three-dimensional domain [6, 7]. *Druid* has strong similarities to three-dimensional modeling programs in that the objects it represents are embedded in three-dimensional space.[1] However, there are inherent differences between the two construction methods. Since *Druid's* domain is confined to a plane, its representation is naturally manipulated using a two-dimensional user interface, such as a computer display and a mouse. Three-dimensional modeling programs produce truly three-dimensional representations which are then generally viewed from multiple viewpoints on a two-dimensional display and manipulated with a two-dimensional input device, *e.g.*, a tablet or a mouse. In an informal experiment, we observed that the time required to construct and alter 2½D scenes using *Druid* can be considerably shorter than the time required to construct and alter corresponding three-dimensional models. For example, we observed that the construction time for the drawing shown in Fig. 1.3 (top left) is about two minutes when using *Druid*, but is upwards of an hour or more when using a three-dimensional modeling program. Thus, *Druid* serves a useful function within its domain since it allows a user to accomplish his goal more quickly than he could in a three-dimensional domain.

One might ask if the reliance on a two-dimensional interface is the only reason three-dimensional modeling programs perform poorly relative to *Druid* when constructing 2½D scenes. There are devices which permit a user to manipulate and view three-dimensional computer models in three-dimensions. For example, stereoscopic displays provide three-

[1]More specifically, *Druid* constrains the user to the construction of topologically valid orientable two-manifolds with boundary embedded in $\mathbb{R}^3$ such that their projection onto $\mathbb{R}^2$ can have multiplicity greater than one but is everywhere nonsingular, *i.e.*, an *immersion* (see Williams [44]).

dimensional observation, and three-dimensional input devices (such as the Body Language User Interface (BLUI) [10]) permit three-dimensional manipulation. Although these systems are likely to assist in the construction of actual three-dimensional models, we remain unconvinced that 2½D scenes can be constructed or manipulated with precision or speed comparable to that of *Druid*. One major problem with using three-dimensional modeling to achieve 2½D results is that three-dimensional interfaces assume the third dimension is continuous. The continuity of the third dimension loses the useful property of 2½D scenes that the third dimension becomes discrete after projection onto a plane, *i.e.*, surfaces in 2½D scenes cannot interpenetrate and therefore reside at integer depth indices with respect to one another. This property is very useful and *Druid* exploits it to vastly reduce the number of operations a user must perform in order to construct an intended drawing.

1.4 Previous Research

It is important to realize that *Druid* is not merely a Celtic knot tool. Many researchers have studied the construction and organization of Celtic knots (see Cromwell [15] and Scharein [39]) and several programs are specifically designed to facilitate the construction of images of celtic knots and mathematical knots in general (see [1, 5, 12, 39, 46]). Such systems, however, are limited to knot constructions, *i.e.*, interwoven cords. *Druid*, on the other hand, permits the construction of much more general scenes in which the surfaces can be any orientable two-manifold with boundary. In particular, there exist scenes of interwoven surfaces which are not scenes of knots, *e.g.*, Fig. 1.3 (top left and top center). Neither conventional drawing programs nor knotwork programs can represent such scenes.

Perhaps the research that most closely resembles *Druid* is that by Baudelaire and Gangnet [9], which relies on *planar maps* as the organizing principle for a drawing tool and *map-sketching* as a means to construct 2½D scenes. Planar maps permit the construction of overlapping curve segments, which are subsequently pruned using map-sketching, during which some edges are erased. Each face of the graph (an edge-bounded region of the surface boundary graph) can be assigned an independent color in the drawing. With a proper erasure of edges and coloring of faces, the illusion of interwoven surfaces can be achieved. It is interesting to note that edge erasure is not required for this method to work and is therefore superfluous, *i.e.*, faces can be colored to achieve the appearance of interwoven surfaces without erasing any edges. Baudelaire and Gangnet's planar maps are quite similar to the best method available in *Adobe Illustrator* [2] for constructing 2½D scenes[2] in which the user converts a series of overlapping curves into a planarized graph in which the faces can be assigned independent colors. As such, the Baudelaire method is simply the *Illustrator* method with an additional (and unnecessary) edge-erasing interaction, although their research predates the implementation of this feature in *Illustrator*. We describe this planar map method for constructing interwoven scenes in detail in Section 3.1.2.

The planar map method does not possess the natural *affordances* of 2½D scenes. Affordances are the natural manipulations an object suggests for itself (see Norman [33, 34]). For example, a mouse affords translational movement and clicking. 2½D scenes afford 2½D manipulations such as surface construction and inversion of the relative depth ordering of surfaces. A 2½D drawing program should have a user interface whose affordances are isomorphic with the affordances of 2½D scenes. We discuss affordances in detail in chapters 2 and 3. The edit-distances required to achieve elemental drawing changes can be quite large when using the methods employed by conventional drawing programs.

[2]This is the best method that is *natively* available in *Adobe Illustrator*. There are third-party plugins that facilitate the construction of a specific *subset* of interwoven scenes, *i.e.*, Celtic knots (see Artlandia [5]). However, these systems do not span the full space of 2½D scenes and do not provide natural *affordances*.

1.5 Thesis Overview

This document describes the novel aspects of *Druid*, *i.e.*, those aspects of *Druid* which distinguish it from existing drawing programs. This level of description should be sufficient for understanding *Druid* since those aspects of *Druid* which are not described, *e.g.*, common utilities such as the use of B-splines or the Macintosh graphics toolbox, are common knowledge and are described in other reference material [4, 3].

1.5.1 Drawing Programs

One of the main goals of this document is to describe how *Druid* works in enough detail to allow a careful reader to reimplement it. Some of the components of *Druid* are common to most drawing programs, *e.g.*, The Macintosh graphics toolbox and B-Splines [4, 3]. We do not make an effort to describe those aspects of *Druid* which are common to most drawing programs, but instead focus our efforts on those aspects unique to *Druid*. Chapter 2 describes the user-interactions that a drawing program like *Druid* must provide. This chapter also describes important concepts from the field of human computer interaction (HCI), including *affordances*, *direct manipulation interfaces*, and *constraint-based interfaces*, and describes how these concepts apply to *Druid's* design.

1.5.2 Comparison Between Conventional Drawing Programs and *Druid*

While *Druid* enables the construction of scenes of interwoven surfaces, similar images can be constructed using conventional drawing programs. Consequently, one may wonder what additional benefits *Druid* offers. Chapter 3 describes how images of interwoven surfaces can be constructed using conventional drawing programs and illustrates the difficulties that are inherent in using these programs. This chapter also describes three stages in the evolution of drawing programs, starting with the most basic, *i.e.*, those that rely strictly on layers, and culminating with *Druid*, which can naturally represent any 2½D scene. This chapter should make it clear that despite the feasibility of using these programs to construct scenes of interwoven surfaces, there remain the unresolved issues of naturalness and ease of use. Finally, this chapter presents a sequence of *Druid* screen captures that illustrate the construction of a complex interwoven surface and describe what the user is doing at each step.

1.5.3 *Druid's* Representation: Labeled Knot-Diagrams

Druid represents scenes of surfaces in a fundamentally different way than conventional drawing programs. Understanding this representation is central to an appreciation of how *Druid* differs from conventional drawing programs both in terms of naturalness and ease of use. Chapter 4 describes *Druid's* representation, which we call a *labeled knot-diagram*, and describes the *labeling scheme*, a set of constraints which legally labeled knot-diagrams satisfy. To minimize the possibility of confusion about the origin of (or justification for) the labeling scheme, this chapter also motivates the labeling scheme using a real scene of physical surfaces, *i.e.*, a photograph of overlapping sheets of paper.

1.5.4 Labeled Knot-Diagram Spaces

Druid does a lot of work automatically, without any user intervention, thus reducing the cognitive burden on the user. A simpler program might merely be a labeled knot-diagram editor. The user of such a program would assign crossing-states, signs of occlusion, and boundary segment depths manually in order to construct a legal labeled knot-diagram. However, such a program would be tedious to use and would offer the wrong affordances, *i.e.*, interactions in such a program would not correlate with elemental 2½D scene changes. Instead, *Druid* automatically assigns a legal labeling following a user-interaction. To do this, *Druid* must search through the space of all possible labelings to find a legal labeling. In particular, it must find the *best possible* legal labeling for a given situation, *i.e.*, the *minimum-difference* labeling with respect to the labeling that precedes to the search. Chapter 5 describes the *labeling space*, the space of labeled knot-diagram that exist for a given drawing, *i.e.*, the space of configurations that a particular labeled knot-diagram can assume. This chapter also describes the *search space*, a subset of the labeling space which *Druid* must search through to find a new labeling. Lastly, this chapter describes the graph distance between labelings in the labeling space.

1.5.5 Editing 2½D Scenes

The user-interactions that are used to manipulate 2½D scenes in *Druid* can be classified into two categories, those which preserve the current labeling, *e.g.*, dragging a boundary without causing a topological change, and those which require *Druid* to find a new labeling, *e.g.*, dragging a boundary such that crossings are created or destroyed, or such that the relative positions of crossings on a boundary change. It is important to understand how these categories differ in order to understand how *Druid* operates in each situation. Chapter 6 describes these user-interactions and the two categories. This chapter also describes *crossing-projection* which is *Druid's* technique for maintaining the current labeling without the need to rediscover crossings as the user modifies the locations and shapes of boundaries. Finally, this chapter describes two techniques that are used to improve *Druid's* performance during many user-interactions. We call the first technique *delayed response*. It helps produces smooth feedback during continuous changes. Interactions come in two forms, which are distinct categories mentioned previously. They can be *continuous* or *discontinuous*. For example, dragging a boundary is continuous, while flipping a crossing is discontinuous. During continuous interactions, we want the fastest turnaround times possible in order to achieve a smooth animation-like effect. Delayed response is a method for maximizing the likelihood of fast smooth response times during continuous user-interactions. We call the second technique *minimum acceptable mouse distance*. If the mouse moves too far in a single step, there is a risk that *Druid* will suffer from errors in its crossing-projection algorithm. Minimum acceptable mouse distance is a technique for minimizing the possibility of these errors.

1.5.6 Finding a Legal Labeling

User-interactions which require a new labeling require that *Druid* actually find a new labeling. It does this by searching the search space corresponding to the current knot-diagram for a new legal labeling. This search can be difficult to perform quickly because the search space can be very large and the fraction of the labelings in the search space that are legal can be very small. Chapter 7 describes how the labeling search is performed. Since the search space can be very large, *Druid* uses a number of heuristics to increase the likelihood that the search will terminate quickly and with a good solution. Finally, this chapter describes the various heuristics that are used in the search and provides experimental results that demonstrate the effectiveness of these heuristics and the effectiveness of the search overall.

1.5.7 Boundary Grouping With Cuts

There are two specific problems that can occur when a surface is bounded by multiple boundaries. The first problem is that *Druid* must know which boundaries to drag as a unit when the user drags a surface. The second problem is that when *Druid* cannot tell which boundaries bound the same surface the labeling is ambiguous. Thus, it is useful to have information about boundary groups that represent a more abstract unit, *i.e.*, a surface. *Druid* is capable of automatically identifying surfaces with multiple boundaries without any intervention from the user. Chapter 8 describes how *Druid* does this by using *cuts*, *i.e.*, paths connecting points on two boundaries in a way that is analogous to a scissor cut through an actual surface.

1.5.8 Rendering

Druid can function in two viewing modes, *knot-diagram mode* and *rendering mode*. In knot-diagram mode, boundaries are illustrated with closed curves, signs of occlusion are illustrated with either hash marks or arrows, and occluded boundary segments are illustrated with a dimmed or faded color. The interiors of surfaces are not colored at all, thus surfaces are illustrated only by their boundaries. In knot-diagram mode *Druid* displays a schematic view of the 2½D scene that the user is constructing, not a high quality rendered image. It is analogous to working in *wireframe* mode in a three-dimensional modeling program, in which polygonal three-dimensional models are illustrated only by the edges of their polygons. The output that rendering mode corresponds to a completed piece of artwork. Surfaces are illustrated as actual surfaces would appear, *i.e.*, with solid areas of color filling their interior. Chapter 9 describes how rendering is performed. One important aspect of rendering is calculating the correct fill-color for every *region* of the canvas, *i.e.*, a contiguous area of the canvas disjointly partitioned by boundary segments. *Druid* calculates a region's color by using *slices*, which are a path connecting a point on a boundary to a point within the bounded surface. This chapter describes how slices are used to calculate a region's color.

1.5.9 Crossing-State Equivalence Classes

Druid's search, as described in Chapter 7, suffers from a potentially serious problem. The search space can be prohibitively large and the search can therefore take an unacceptable amount of time to terminate. We have discovered a property of 2½D scenes which we call the crossing-state equivalence class rule which can be exploited in two different ways to vastly improve *Druid's* performance. The first way is to use the crossing-state equivalence class rule as a further search heuristic in a way that exponentially decreases the size of the search space. The second way to improve *Druid's* performance is to directly deduce the correct labeling that follows a user-interaction without performing a search at all. This method is the fastest possible way to relabel a drawing, and performs significantly faster than even the labeling search regardless of which heuristics are applied to the search process. Unfortunately, this second method can only be applied in certain situations, principal among these the case of a crossing-flip interaction. *Druid* must rely on the labeling search in most other situations. Chapter 10 describes a complexity hierarchy of $2^1/_2$D scenes, describes how transformations between these complexities are performed, and then describes the crossing-state equivalence class rule, which applies to drawings of a specific kind of complexity called *simple scenes*. Chapters 11, 12, and 13 then describe how equivalence classes are found and how they are exploited by *Druid*.

1.5.10 Finding Crossing-State Equivalence Classes

In order to exploit crossing-state equivalence classes, they must first be found for a particular drawing. Chapter 11 describes how *Druid* finds crossing-state equivalence classes on a labeled knot-diagram. Later, experimental results on the performance of finding crossing-state equivalence classes are presented. These results demonstrate that equivalence classes can be found very quickly, and thus the need to initially find them does not hinder their use.

1.5.11 Equivalence Class Independence

Once one thoroughly understands the crossing-state equivalence class rule, it may seem like the space of possible configurations for a particular drawing has been fully described, *i.e.*, it may seem like the number of configurations that a drawing can assume directly corresponds to the number of possible equivalence class state instantiations for that drawing. This assumption is premature however. Chapter 12 describes the concept of crossing-state equivalence class *independence*. The dependence or lack there-of between pairs of equivalence classes dictates whether certain equivalence class flips are *atomic* or *nonatomic*. Nonatomic flips correspond to dependent sets of equivalence classes and preclude some equivalence class state instantiations from being possible, *i.e.*, for some equivalence class state instantiations, no legal labeling is possible. Finally, this chapter describes nonatomic and atomic equivalence class flips and how they govern which equivalence class configurations a labeling can legally assume.

1.5.12 Exploiting Crossing-State Equivalence Classes

As stated, there are two ways in which equivalence classes can be exploited to improve *Druid's* performance. First, they can be used to reduce the search space size. Second, following a crossing-flip interaction, *Druid* does not have to perform a search for a new legal labeling since a much faster and more efficient approach is to directly deduce the new labeling. Chapter 13 describes both methods, followed by experimental results that demonstrate how *Druid's* performance is improved by exploiting crossing-state equivalence classes for relabeling.

1.5.13 Future Work

Druid, as described in this document, is a completed application. It contains all the necessary functionality to construct interwoven $2^1/_2$D scenes. However, there are a few interesting directions in which this work could be extended. In Chapter 14, we present three areas of possible future work: labeling with crossing-state equivalence classes, locking and kinematic interactions, and constructing scenes containing *occluding contours*. Currently, crossing-state equivalence classes can only be found on a legally labeled figure.

Consequently, they are cannot be found prior to labeling. The effect is that *Druid* cannot fully exploit them as a search heuristic. Therefore, one useful extension of our work would be to devise a means for finding crossing-state equivalence classes on an unlabeled figure. Locking and kinematic interactions are interactions in which *Druid* could conceivably prevent the user from breaking the current topology by *locking* a user-interaction at the point at which the topology is about to break and then producing translations and rotations that are analogous to those of real physical objects, *e.g.*, dragging a chain. Occluding contours are the locus of points where a curved surface's tangent plane is viewed edge-on. Although *Druid* does not currently represent occluding contours it could be extended to include them. This would permit the construction of a much larger class of scenes.

1.5.14 Conclusions

Chapter 15 first presents general conclusions of our research such as *Druid's* novel representation and the affordances of its user interface. Following these general conclusions, a more specific list of the most important discoveries made during our work is presented.

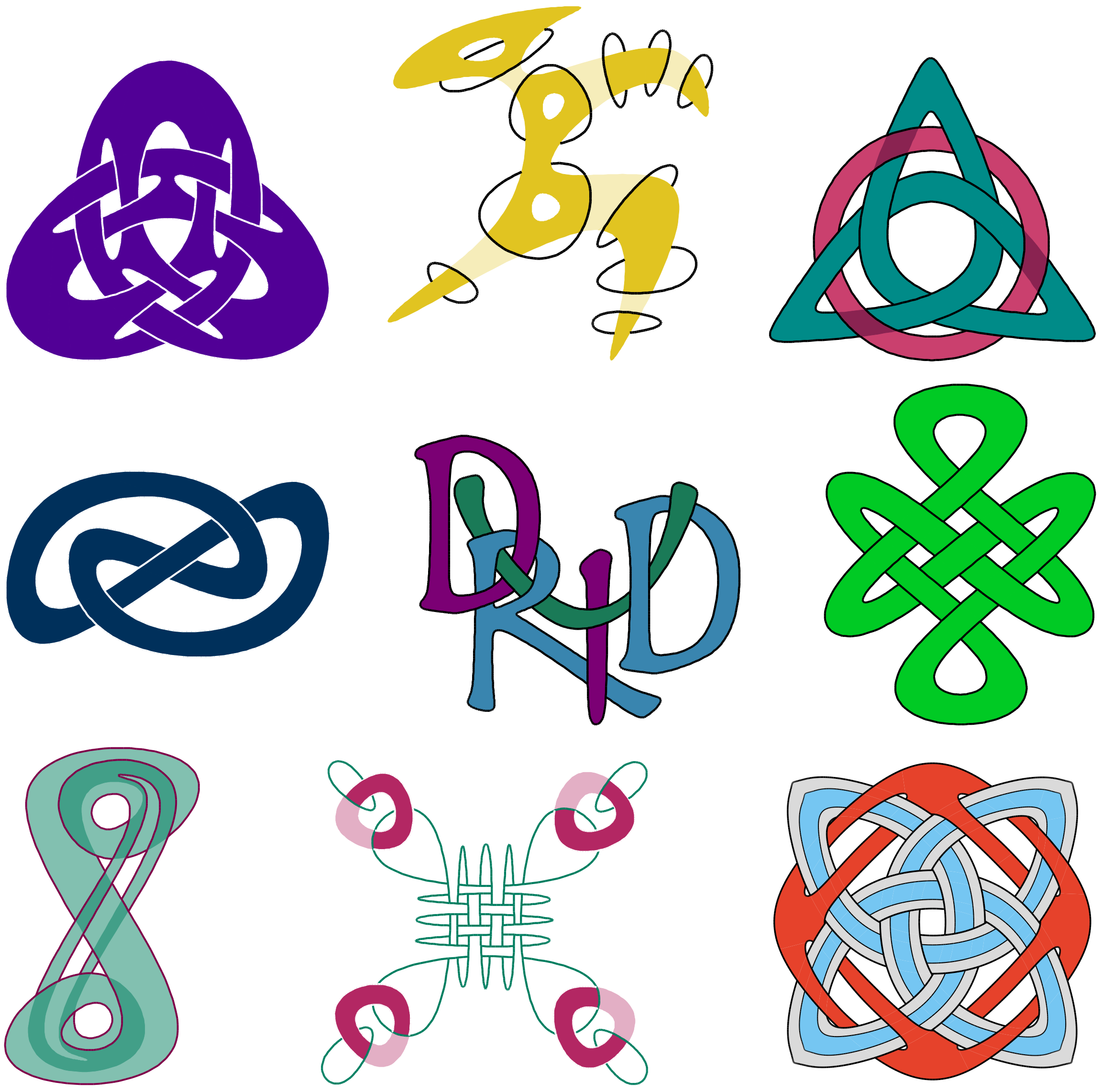

Figure 1.3: Examples of artwork created with *Druid*. The construction and maintenance of these drawings is simple and straightforward.

Chapter 2

Drawing Programs

While much research in computer science does not specifically describe a tangible software system, *i.e.*, the design and workings of an existing computer program, this research does exactly that. Our research focuses not merely on the development of new and *potentially* useful theories, but on the design and implementation of an *actual* computer program, which we call *Druid*. *Druid* is a new kind of vector-based drawing program quite unlike those of the past. In order to understand what makes *Druid* special against the backdrop of a large number of existing drawing programs, one must first be familiar with the user-interactions that drawing programs provide. Only then, can one understand how *Druid* works and what its advantages are over existing drawing programs. Thus, in this chapter we present the crucial user interactions that drawing program must provide. We discuss the concept of *affordances* and how they relate to the design of a drawing program's user interface. Affordances are the actions an object most naturally suggests for itself and they are an important aspect of the design of a drawing program's user interface. In addition to affordances, we discuss a related topic called *direct manipulation interfaces*, which refers to user interfaces in which the user interacts directly with the thing being depicted rather than interacting with it through intermediate tools, *e.g.*, menus, sliders, and buttons. Lastly, we describe *constraint-based interfaces*, which are interfaces in which hard constraints on the available options at any given moment guide the user's interactions. Constraint-based interfaces focus the user's actions and attention on the outcomes that are possible, and excludes those which are currently impossible.

2.1 Drawing Program User-Interactions

Many drawing programs provide tools for creating and manipulating splines. One such program, *MacPowerUser's iDraw* [23], bases all objects on splines. Rectangles, polygons, even text are all represented using splines. This provides a consistent interface for manipulating the various kinds of objects in *iDraw*. Similarly, *Druid* is based entirely on splines. Like [21] and [38], *Druid* uses B-splines. All boundaries are B-splines, including shapes that can be approximated by splines, such as hand-drawn curves, rectangles, and text. The assumption that all boundaries are splines lends a uniformity to a drawing program's interface that makes it easier for a user to understand, while simulaneously simplifying the programmer's job. There are a number of user interactions that a spline-based drawing program should permit. These interactions include:

- Create a new boundary
- Delete a boundary
- Smoothly reshape a boundary
- Drag a surface (drag all of its boundaries)
- Add or remove spline control points
- Increase or decrease spline degree
- Reverse a boundary's *sign of occlusion* (discussed later)
- Reverse the depth ordering where two surfaces overlap.

The interface of a program should not only provide a method for each of these interactions; these methods must be *user-friendly*, that is, simple to understand and easy to use.

Unfortunately, the only previous attempt to circumvent the partial ordering limitation, *MediaChance*'s *Real-Draw Pro 3*, forces the user to use a complex and confusing interface.

2.2 Software Affordances

A software application's interface possesses a specific set of *affordances*. The term affordance is debated at great length in the literature. There are multiple interpretations for this term and how it should be applied to design (see McGrenere [28]). Gibson originally coined the term[17]. He defined the affordances of an object as the action possibilities of that object relative to a particular *actor*, the person who is interacting with the object in question. Gibson claimed that affordances are independent of the actor's past experience and accumulated knowledge, but are dependent on the specific actor, *e.g.*, a table affords sitting for a cat, but not for an elephant. Norman adopted the same term in [33, 34], but uses it slightly differently. According to Norman, affordances are only those actions that are perceived, not possible actions that are not perceived. Additionally, perceived actions that are not actually possible are still affordances according to Norman. He defines affordances as the clues an object offers about how it can be used, and believes affordances can be dependent on an actor's experience. In this way different people might perceive different affordances for the same object based on their individual past experiences.

There is a general agreement that affordances are difficult to define with respect to software. This observation results from the fact that affordances are usually defined with respect to physical qualities of material objects. With this difficulty in mind, we define the affordances of a user interface as the ways in which the user can interact with the screen's imagery, since for the most part existing interfaces present a single two dimensional image to the user with which to interact. This interaction usually involves a keyboard and a mouse. In the case of drawing programs, use of the keyboard is generally minimized because this violates the notion of *direct manipulation interfaces* (discussed below). Therefore, the affordances of drawing programs mainly consist of clicking on and dragging various features of the visual presentation with a mouse and associated cursor. In the case of drawing programs, this means clicking and dragging on the control points of the various splines in the drawing.

In our design for *Druid*, we attempt to model our interface on a real physical system and then to offer the affordances characteristic of that system. The physical system that a drawing program depicts is a 2½D scene which the user can manipulate in ways that are appropriate for such scenes, *e.g.*, altering the shape and placement of surfaces, and altering the relative depths of the surfaces in areas of overlap.

Real physical surfaces possess certain natural affordances. They can be stretched, translated, cut into smaller surfaces, have holes cut in them, and lifted above or pushed beneath one another in potentially interwoven arrangements. They can also be colored or be made transparent. In some cases they can be glued to other surfaces to form larger surfaces. We believe that an effective drawing program should provide a set of affordances that is isomorphic to those of real surfaces. This amounts to a visual analogy between the program's usage and the thing depicted, in our case, 2½D scenes. One example is translating a surface by "grabbing" it with a hand-shaped cursor and then dragging it in the desired direction, which is directly analogous to how a graphic designer might move pieces of paper around on a drafting board. Unfortunately, many drawing programs do not offer such a set of isomorphic affordances. It is our belief that while some programs, such as *Real-Draw*, attempt to solve the problems posed in this research, they do so through interfaces with unnatural affordances which make the programs complicated and non-intuitive to use. *Druid's* interface possesses affordances which are isomorphic to the affordances of the physical surfaces which are depicted. As such, it is simpler to use, while at the same time, is more powerful than existing drawing programs in its capability to create and manipulate complex 2½D scenes. Demonstrating that *Druid's* affordances are more natural and therefore superior to the affordances of other drawing programs is a major focus of this research.

2.3 Direct Manipulation Interfaces

A concept that is closely related to affordances is *direct manipulation interfaces* (see Norman [34]). A direct manipulation interface is an interface which allows the user to interact with the depicted object using the most direct method possible given the I/O devices that are available. The least direct interface for drawings would be one in which the user types textual commands to manipulate a drawing. A more direct interface would use a light-pen or touch-sensitive screen, analogous to real pencil and paper. For example, Sutherland's *Sketchpad* [41] used a light-pen for user input. Today, most computers use a mouse interface.

Given this limitation, a direct manipulation device is an interface in which mouse and cursor movements are used to directly manipulate the elements of a drawing, as opposed to using a set of menus, buttons, and sliders. The hand-dragging example, mentioned previously, is a good example of a direct manipulation interface. Direct manipulation interfaces are popular because they minimize the user's effort by allowing him to interact directly with the drawing being edited rather than through an intermediate interface. The benefits of direct manipulation interfaces explain why most drawing programs rely heavily on a mouse and only weakly on a keyboard.

Direct manipulation interfaces have been used in drawing programs in the past, such as the original drawing program, *Sketchpad*. The basic premise of a drawing program as a tool which shows the drawing as it is being constructed might seem obvious now, but it originated with the WYSIWYG [1] concept that came out of Xerox PARC in the 1970s (see Myers [32]). Raisamo and Raiha applied the idea of direct manipulation interfaces to the

[1]"what you see is what you get"

problem of aligning objects in a drawing (see Raisamo and Raiha [35, 36] and Raisamo [37]). They demonstrated that such an approach provided significant improvements over established methods such as alignment commands and gravity active points.

There is a close correspondence between direct manipulation interfaces and good affordances because direct manipulation interfaces create a one-to-one correspondence between edit-distances and distances in the representation space of $2^1/_2$D scenes. Stated differently, the user interface is isomorphic to the $2^1/_2$D scene the program represents and manipulates. By contrast, a keyboard and command-line interface for a drawing program generally requires numerous keystrokes, each prone to error, in order to accomplish very simple tasks in terms of representation distance.

We have attempted to incorporate the notions of good affordances and direct manipulation interfaces in the design of *Druid*. Most user-interactions are performed by interacting directly with the drawing in ways that are intended to make sense even to a novice user.

2.4 Constraint-Based Interfaces

Constraint-based interfaces have become popular in drawing programs. The most common application of constraints in drawing programs is gravity-snapping, where the cursor (and any object linked to the cursor) snaps to grid points for the purpose of keeping objects aligned. A slightly different approach snaps objects to other objects rather than an underlying grid (see Gleicher [18]). *Druid's* operation is also based on constraints. First, *Druid* constrains the user to constructing topologically valid $2^1/_2$D scenes. Second, a user's interaction with a drawing takes the form of a constraint, indicating the user's intent, that guides the search process for a new legal labeling. This search is described in later sections.

The correspondence we describe between a $2^1/_2$D scene and its depiction in a drawing is not the first attempt to design a drawing tool based on a physical analogy. Gleicher [19] describes a method for manipulating boundaries in a drawing by applying physical forces that push and pull against the boundaries. Treating drawings like physical systems is useful because it allows the user to apply intuitive understanding of physics and topology to the process of creating a drawing.

Chapter 3

Comparison Between Conventional Drawing Programs and *Druid*

This document describes how our system, *Druid*, represents and permits the construction and manipulation of interwoven 2½D scenes. However, it is already possible to construct images of interwoven surfaces using conventional drawing programs. Consequently, one might wonder what purpose *Druid* serves. In this chapter we describe some methods by which images of interwoven surfaces can be constructed using conventional drawing programs. In doing so, and in a followup discussion, we will show that it is difficult and unnatural to construct interwoven scenes with conventional drawing programs, *i.e.*, our argument with interwoven surfaces in conventional drawing programs reaches beyond the issue of *feasability* (conventional drawing programs are perfectly feasible in regard to images of interwoven surface) to the more fundamental issues of *naturalness* and *efficiency*. *Druid* is more than an augmentation to the conventional representation, it is a quantum leap in the design and representation of 2½D scenes. To illustrate this point, we present the evolution of drawing program representations in three stages: global layers, localized layers, and a complete lack of dependency on layers. The first stage represents most conventional drawing programs. The second stage represents a single case we found amongst conventional drawing programs, *MediaChance*'s *Real-Draw Pro-3*. The third stage represents *Druid*. Finally, we present a sequence of *Druid* screen captures that illustrate the construction of a complex interwoven surface and describe what the user is doing at each step.

3.1 Constructing Images of Interwoven Surfaces in Conventional Drawing Programs

With considerable effort, it *is* possible to create images with existing drawing programs that depict interwoven surfaces. However, the underlying drawing representation in such cases is not, and cannot be, truly interwoven. Three ways of achieving this effect are:

1. Spoofs
2. Painting planarized graphs
3. Local DAG manipulation.

In this chapter we describe each of these methods in detail.

3.1.1 Spoofs

One method for constructing an image of interwoven surfaces without using an interwoven representation is to construct one set of surfaces which has the appearance of a completely different set of surfaces. We call this sort of illusion a *spoof* (Figs. 3.1 and 3.2). Spoofs represent non-generic configurations, where various elements of a drawing are precisely aligned in order to create the illusion of interwoven surfaces.

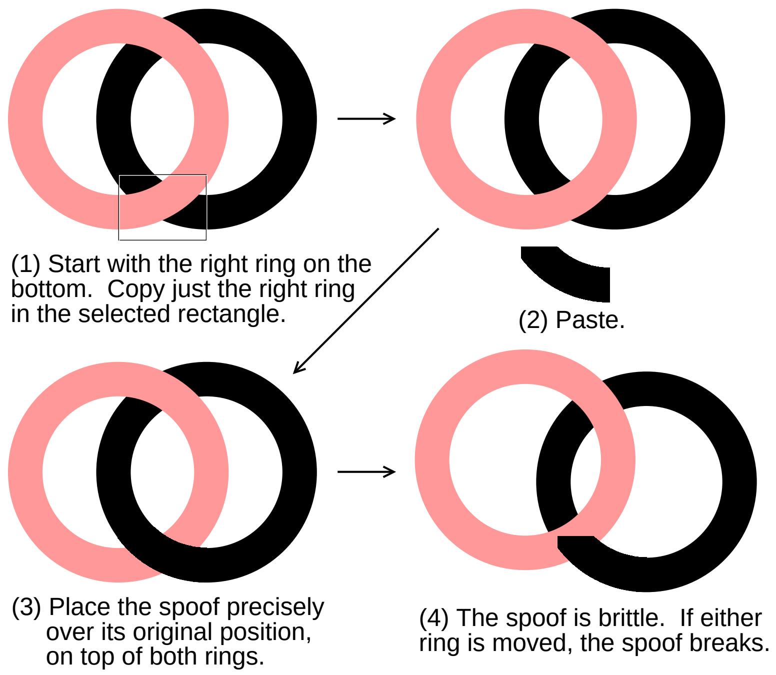
(1) Start with the right ring on the bottom. Copy just the right ring in the selected rectangle.
(2) Paste.
(3) Place the spoof precisely over its original position, on top of both rings.
(4) The spoof is brittle. If either ring is moved, the spoof breaks.

Figure 3.1: A spoof is a process by which the illusion of interwoven surfaces is constructed in a layered system. The underyling representation does not match the final rendered image. See also Fig. 3.2.

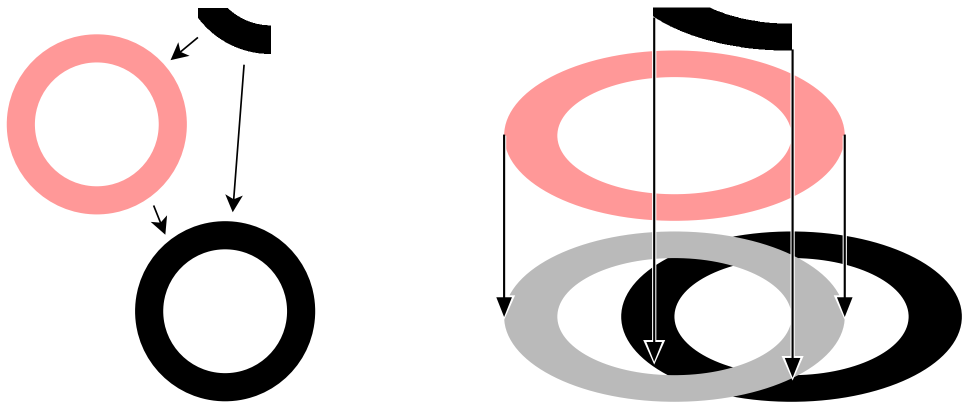

Figure 3.2: The figure on the left shows the DAG for the three components of the spoof in Fig. 3.1. While the illusion of interwoven surface has been created, the underlying representation is still a partial ordering, as required by existing drawing programs. The figure on the right shows an oblique view of the canvas, with the layers vertically spread apart to illustrate the spoof's construction.

3.1.2 Painting Planarized Graphs

Another method for constructing an image that has the appearance of interwoven surfaces is *painting planarized graphs*, which was briefly discussed in Section 1.4. This method consists of converting a drawing of boundaries that represent overlapping surfaces (Fig. 3.3, top left) into a planar graph where vertices represent boundary crossings and edges represent boundary segments (Fig. 3.3, top right). Once the drawing has been converted to a planar graph, each face of the graph can be independently *painted* (or *filled*) with a color of the user's choosing (Fig. 3.3, center left). With a proper assignment of paint colors to the faces of the graph, an image can be constructed which has the appearance of interwoven surfaces (Fig. 3.3, bottom right).

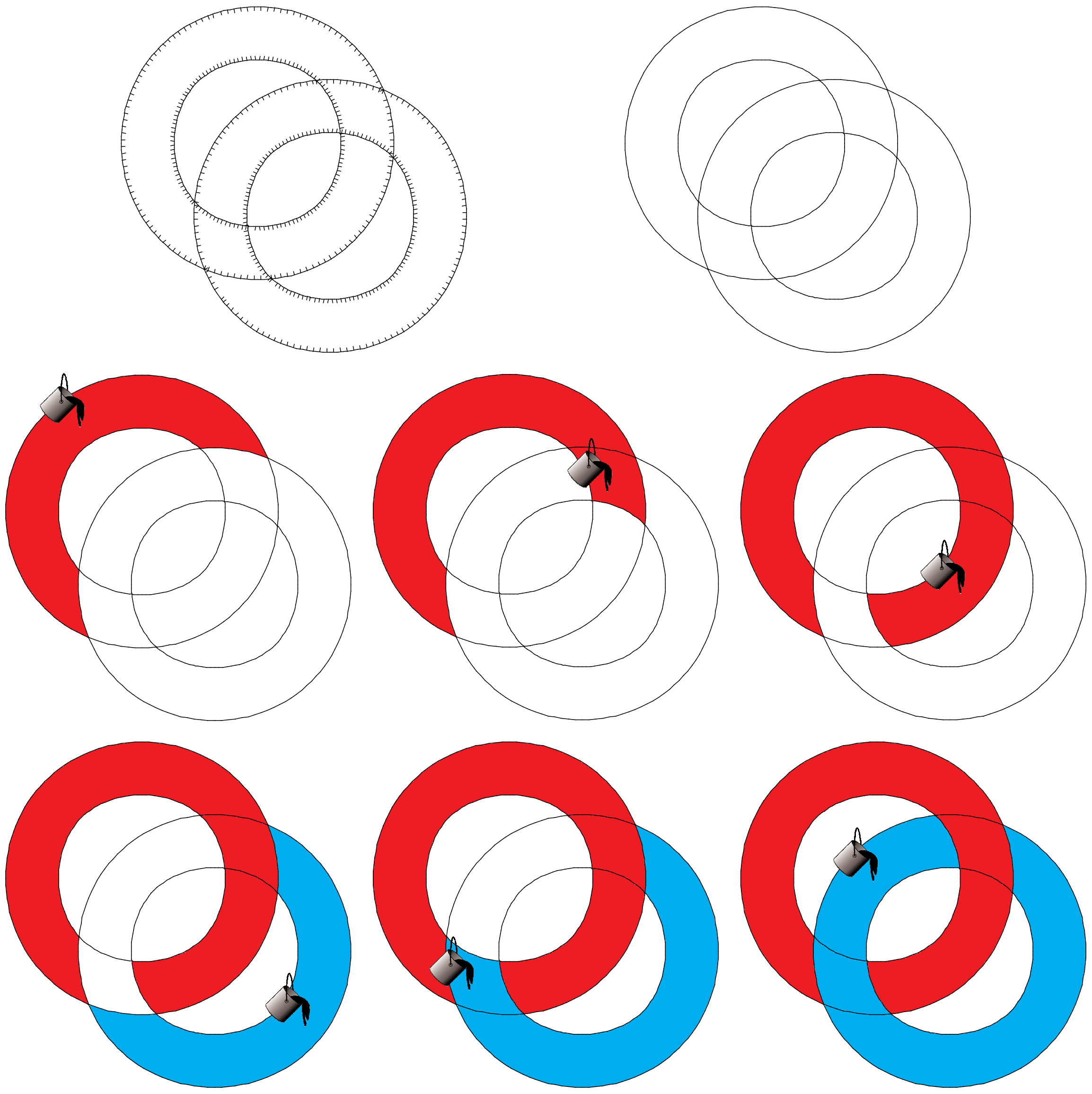

Figure 3.3: Painting a planarized graph begins by representing a set of boundaries (top left) as a planar graph (top right). The graph can be colored by painting faces of the graph with a fill color (center left). By carefull assignment of paint colors to the faces of the graph, an image can be constructed which has the appearance of interwoven surfaces (bottom right).

3.1.3 Local DAG Manipulation

To our knowledge, the only program that makes a purposeful attempt to solve the program of representing interwoven surfaces is *MediaChance*'s *Real-Draw Pro-3* (see Voska [42]). *Real-Draw* starts out with a basic layered representation, but then provides a special tool called the *push-back*. This tool lets the user define a region of the canvas where the partial ordering can be locally altered. The layer that resides at depth zero within the selected region can be pushed down to an arbitrary depth, placing it beneath some or all of the (previously) deeper layers. Although the depth ordering of surfaces below the surface that is at depth zero by default cannot be altered, this operation is sufficient to create most kinds of interwoven images (Fig. 3.4).

3.2 Affordances of Conventional Drawing Programs

One might ask, if spoofs, painting planarized graphs, and local DAG manipulation are sufficient methods for creating rendered images of interwoven surfaces, what difference does it make if the underlying representation of the drawing does not correspond to the $2^1/_2$D scene which is perceived? Our answer lies in an analysis not of the *capability* of these methods (these methods are fully capable of creating images which are perceived as interwoven surfaces), but in the unnaturalness and labor intensiveness of these methods. Spoofs are tedious to construct, requiring many steps to be performed with precision. Furthermore, they are brittle because once a spoof has been constructed, any alterations to the drawing will require the spoof to be redone. Likewise, planarizing a drawing then painting the faces of the graph independently requires many intermediate steps which must be performed in a precise manner in order to achieve the desired effect.

The push-back tool used by *Real-Draw* is the easiest method of the three listed above, but it is not ideal. One problem with *Real-Draw's* approach is that it is merely an incre-

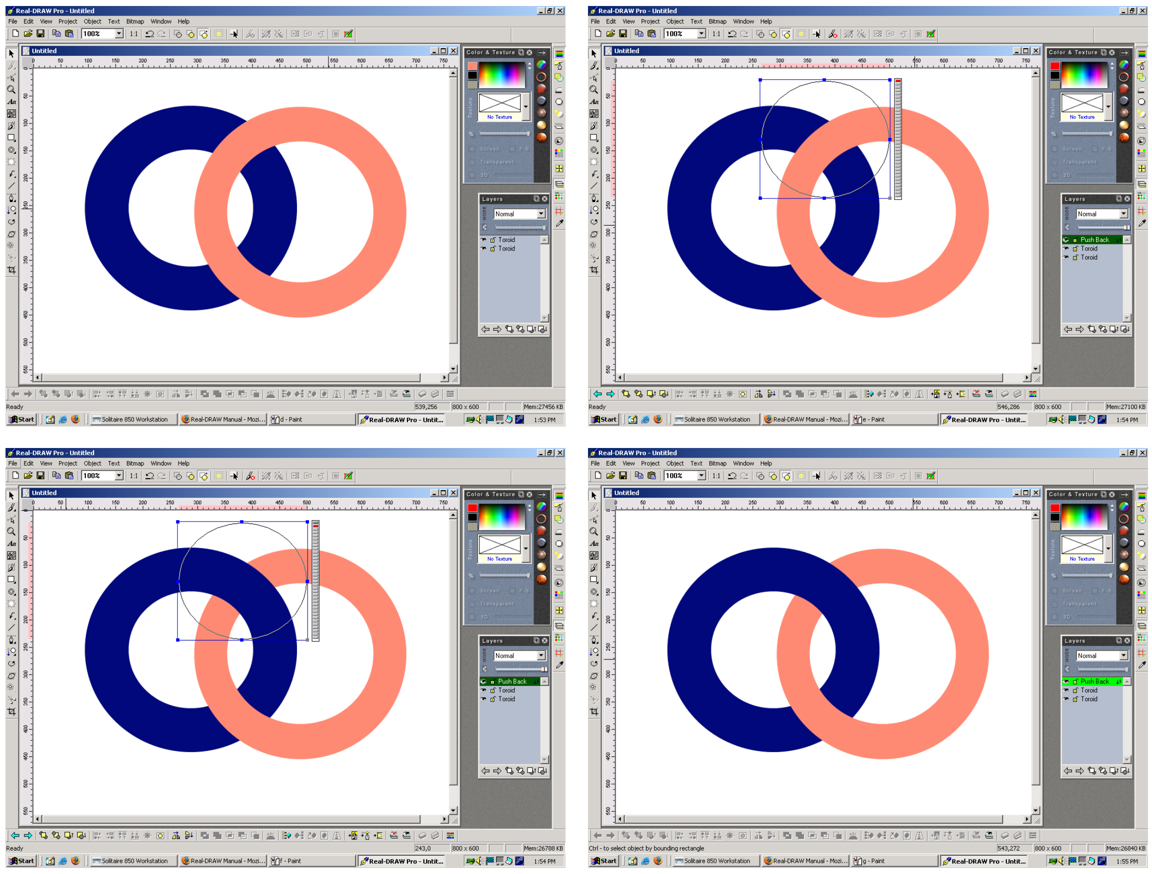

Figure 3.4: *Real-Draw* provides a *push-back* tool, which allows the user to define a region of the canvas and then manipulate the ordering of the layers within that region. The sequence shown here illustrates how this manipulation is accomplished, in the order left to right, top to bottom.

mental improvement over a global layering system. Consequently, the actual interface for manipulating surfaces in *Real-Draw* is awkward and counterintuitive. It relies on the use of a new kind of object on the canvas, the *push-back object*. Because the push-back object does not reflect the way humans perceive and reason about surfaces, it is not a natural representation for overlapping surfaces, *i.e.*, it does not possess *natural* affordances (see Section 2.2), *i.e.*, affordances that are isomorphic between the drawing and the surfaces being depicted.

Additionally, because the push-back object is an object on the canvas, it must be kept properly aligned with the surfaces it is associated with. If the user adjusts the locations of surfaces that are associated with a push-back, the user must also adjust the location and scale of the push-back object to make sure it still encompasses the relevant region of the canvas. Furthermore, the introduction of new surfaces into an existing push-back object's region requires that the old push-back object be replaced. We believe that a labeled knot-diagram-based representation, offering better affordances, and which makes fewer demands on the user, is a better solution.

More significantly, *Real-Draw* does not span the space of 2½D scenes. There are two situations where this can arise. The first situation occurs when the user attempts to create a surface that overlaps itself. A single label is used to represent each surface in the global surface DAG, even when that surface overlaps itself. Since a push-back only allows the reordering of layers based on labels in the DAG, there is no way for a push-back to represent or manipulate the multiple coverings implicit in a self-overlapping surface. The primary cause of this problem is that the push-back presents an interface for manipulating local DAGs, but the surface labels remain properties of a global DAG. The reliance on an underlying DAG is a fundamental deficiency in *Real-Draw's* representation (Fig. 3.5).

An additional problem with *Real-Draw* is that since the push-back object only allows the depth zero object to be pushed down, there is no way to manipulate the ordering of deeper layers. If surfaces are opaque this does not matter since only the depth zero surface will be visible. However, if surfaces are transparent, then the ordering of deeper layers might affect the appearance of a region where multiple surfaces overlap (depending on the exact transparency model used).

In theory, the basic idea of a push-back object could be elaborated to allow a more comprehensive manipulation of the surface depths. However, the more serious problem of self-overlapping surfaces would remain unaddressed.

In contrast, *Druid's* user interface has affordances which are isomorphic to those of idealized physical surfaces, *i.e.*, elemental scene changes are achieved through minimal or straight-forward user-interactions, *e.g.*, a single mouse click. Consequently, interacting with *Druid* is much more natural and intuitive than interacting with conventional drawing programs. The user's experience when using *Druid* is that of interacting with real surfaces, not an imitation of surfaces.

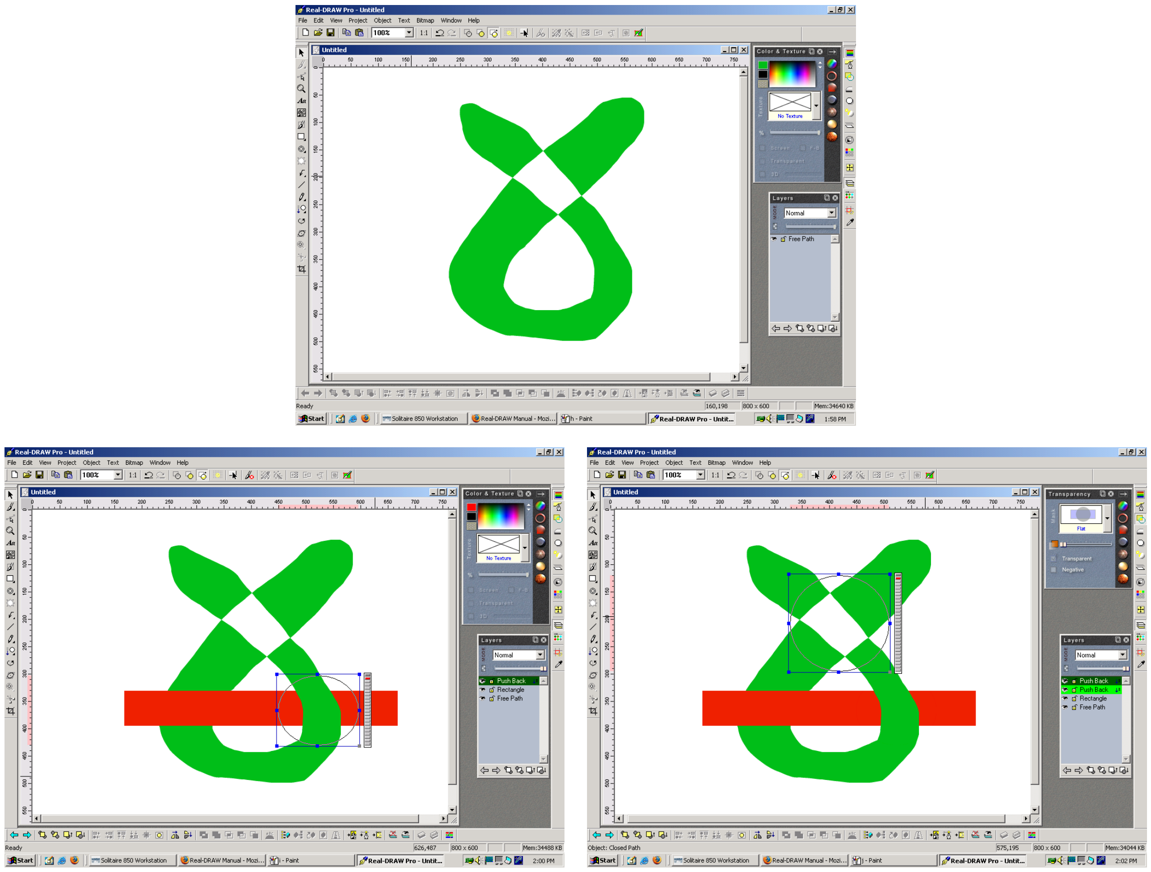

Figure 3.5: *Real-Draw* cannot properly represent self-overlapping surfaces. Since a surface has only one label in the surface layer list, there is no way to manipulate the relative depths where a single surface covers a region at multiple depths. For this reason, *Real-Draw* cannot represent self-overlapping surfaces. The overlapping region is rendered with empty space, as shown here.

3.3 Stages in Drawing Program Evolution

Druid represents a new kind of drawing program that is more powerful than existing programs. It is possible to describe a progression of drawing program functionalities. This progression classifies drawing representations in three stages of increasing generality:

1. Drawing representations which assume a *global* DAG on the surfaces
2. Drawing representations which allow different DAGs in different regions of the canvas
3. *Labeled knot-diagram* based representations.

Stage 1 consists of programs based on representations that can be described as layers of constant depth, and includes virtually all existing drawing programs.

Stage 2 consists of programs based on drawing representations which still rely on a partial ordering of the surfaces, but which allow the user to define special regions of the canvas where the partial ordering will differ from the default DAG. To our knowledge, the only program in this category is *MediaChance*'s *Real-Draw Pro-3* (see Voska [42]), which was described in Section 3.1.3.

Stage 3 consists of drawing programs based on representations that do not rely on any notion of a partial ordering of the surfaces. Such a representation contains only localized information about the depths of various surfaces in each region. We do not know if *Druid's* representation is the best representation of this type, but it has been proven that *Druid's* representation is sufficient to represent the full space of $2^1/_2$D scenes (see Williams [44]).

3.4 Demonstration of *Druid*

Up to this point, this chapter has illuminated the weaknesses inherent in attempting to construct interwoven 2½D scenes with conventional drawing programs. In this section we offer a demonstration of how *Druid* is used for the purpose of comparison. The best way to appreciate *Druid's* power is to see someone use it. In print, we can most effectively do this by showing a sequence of screen captures, much as we have done in the previous sections.

Fig. 3.6 demonstrates how *Druid* is used. *Druid* uses closed B-splines to represent the boundaries of surfaces. Spline control points are defined in either a clockwise order to create internally bounding boundaries (*A*) or in a counter-clockwise order to create externally bounding boundaries (*B* and *D*). Crossings can be clicked to reverse the relative depths of areas where surfaces overlap (*C* and *E*). We call this interaction a *crossing-flip*. Whenever the current labeling is *legal*, *i.e.*, whenever all crossings satisfy the labeling scheme, the drawing can be *rendered* (*F*).

Note that there is a natural logic to the operations in Fig. 3.6. For example, to alter the depth ordering of various overlapping surfaces, the user merely clicks on a crossing to invert its crossing-state. *Druid* then does all of the computation necessary to keep the labeling legal. This computation consists of searching the space of legal labelings for a labeling which satisfies the constraint that the new crossing-state represents. Compare this mode of interaction with either the spoof approach associated with Stage 1 drawing programs or with the push-back approach associated with Stage 2 drawing programs, *i.e.*, *Real-Draw*. Construction of a spoof that appears like *E* would be quite tedious. Worse yet, to invert the relative depth ordering within an overlapping area, the spoof would have to be completely rebuilt. If one were to use *Real-Draw*, push-back objects would have to be explicitly created for each desired overlap and would have to be maintained if the user were to move the various surfaces around.

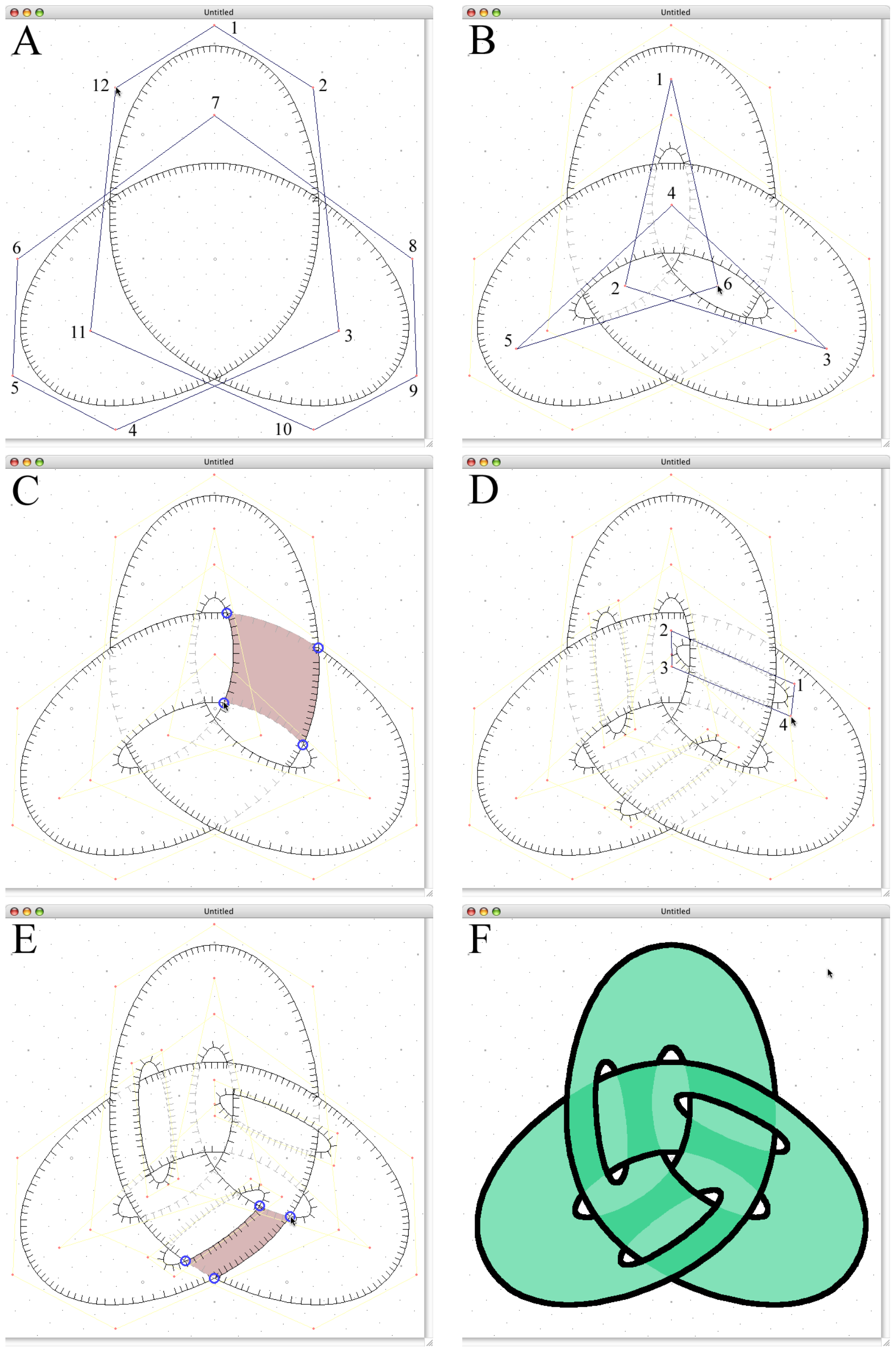

Figure 3.6: Demonstration of *Druid*. Spline control points are defined in either a clockwise order to create internally bounding boundaries (*A*, numbers denote control point order) or in a counter-clockwise order to create externally bounding boundaries (*B* and *D*). Crossings are clicked to flip overlapping surface regions (*C* and *E*). Whenever the drawing is legally labeled (*B* - *E*), the figure can be rendered (*F* renders *E*). In this example, the surface has been made partially transparent.

Chapter 4

Druid's Representation: Labeled Knot-Diagrams

In Chapter 3 we described three methods by which images of interwoven surfaces could be constructed using conventional drawing programs. We then described how these methods can be improved upon by using a more general representation, *i.e.*, one which makes no assumption of layers. At the end of Chapter 3 we demonstrated our system, *Druid*. In this chapter we describe *Druid's* novel representation for scenes of surfaces called a *labeled knot-diagram*. We then describe a set of natural constraints called the *labeling scheme* on the ways in which surface boundaries appear in 2½D scenes. To alleviate potential confusion about the justification for the labeling scheme, we motivate it with a pictorial example consisting of a photograph of physical surfaces, *i.e.*, a scene of overlapping pieces of paper. In the Chapter 7 we will describe how *Druid* assigns a legal labeling to a particular drawing in order to present the user with a scene that represents a set of idealized physical surfaces.

4.1 Labeled Knot-Diagrams

In order to build a Stage 3 drawing tool, it is necessary to develop a fundamentally new approach for the representation of drawings. Existing drawing programs represent a drawing as a set of surface boundaries which reside in distinct layers, *i.e.*a DAG, thus precluding an interwoven arrangement. In contrast, a Stage 3 program represents the boundaries of surfaces in a localized way that does not assume surfaces are arranged into a DAG. A Stage 3 program relies on the fact that local depth changes always occur at surface boundary crossings.

Our system, *Druid*, represents a 2½D scene as a *labeled knot-diagram* (see Williams [44]). A *knot-diagram* is a projection of a set of closed curves onto a plane and indicates which curve is above wherever two curves intersect (Fig. 4.1). Williams extended ordinary knot-diagrams to include a *sign of occlusion* for every boundary and a *depth index* for every boundary segment (Fig. 4.1). The sign of occlusion is illustrated with an arrow and denotes a bounded surface to the right with respect to a traversal along the boundary in the arrow's direction.

4.2 Labeling Scheme

Chapter 7 describes the algorithm *Druid* uses to assign a labeling to a knot-diagram. The process of assigning a labeling is similar to Huffman's *scene-labeling* (see [20]), in which he developed a system for labeling the edges of a scene of stacked blocks. In *Druid's* case, the labeling consists of signs of occlusion, crossing-states, and segment depth indices. The *labeling scheme* is a set of local constraints on the relative depths of the four boundary segments that meet at a crossing (Fig. 4.2). There are four rules of the labeling scheme:

1. The upper boundary must have the same depth on both sides of the crossing.

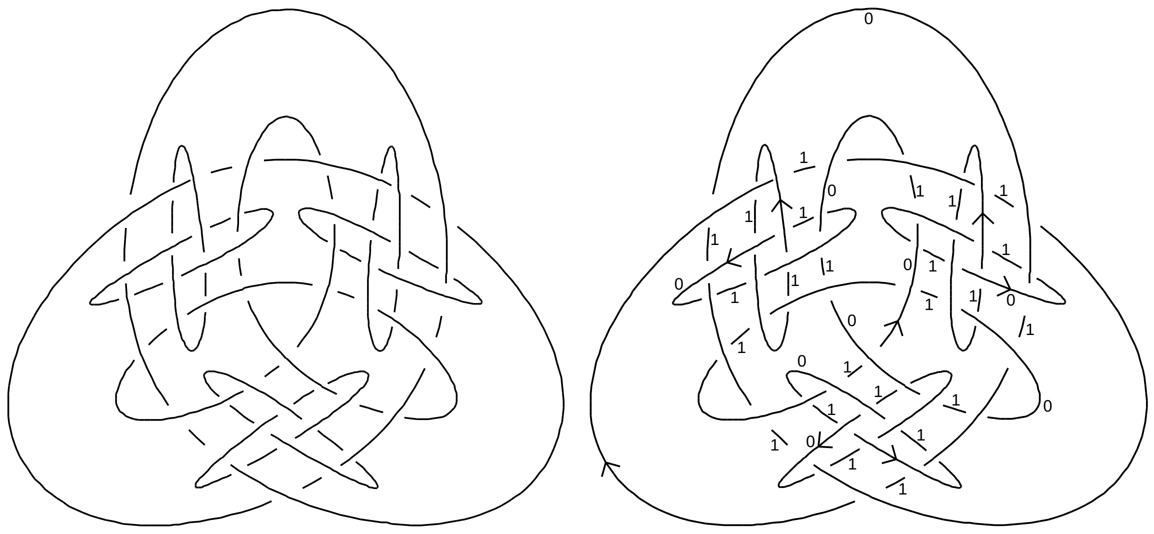

Figure 4.1: A *knot-diagram* (left) is a projection of a set of closed curves onto a plane together with indications which show which curve is on top at every crossing. A *labeled knot-diagram* (right, see Williams [44]) is a knot-diagram with a sign of occlusion for every boundary and a depth index for every boundary segment. Arrows show the signs of occlusion for the boundaries, always denoting a surface bounded to the right of a boundary with respect to travel along the boundary in the direction of the arrow. The sign of occlusion can also be denoted using hash marks. Some depth indices of depth zero have been omitted for clarity.

2. The lower boundary must differ in depth by exactly one across the crossing.
3. The lower boundary must be deeper on the occluding side of the upper boundary.
4. The lower boundary must be no shallower than the upper boundary on the unoccluding side of the upper boundary.

If every crossing in a labeled knot-diagram satisfies the labeling scheme, the labeling is a *legal labeling* and accurately represents a scene of topologically valid surfaces. Legal labelings can be *rendered*, *i.e.*, translated into images (the process for which is described later) in which the interiors of surfaces are filled with solid color.

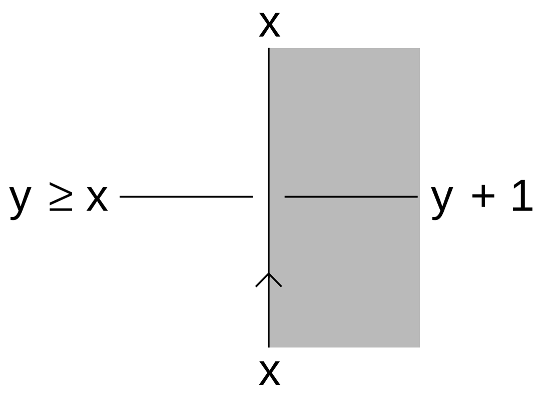

Figure 4.2: The *labeling scheme* (see Williams [44]) is a simple set of constraints on the depths of the four boundary segments that meet at a crossing. If every crossing in a labeling satisfies the labeling scheme then the labeling is a *legal labeling* and can be rendered. *x* is the depth of the upper boundary. The upper boundary must have the same depth on both sides of the crossing. *y* is the depth of the unoccluded half of the lower boundary. The lower boundary must have a depth of *y + 1* in the occluded region (shaded), as defined by the upper boundary's sign of occlusion. Finally, the lower boundary must reside beneath the upper boundary. Thus, *y* must be greater than or equal to *x*.

4.3 Motivating the Labeling Scheme

The labeling scheme may seem counter-intuitive at first glance. For example, it might be confusing that y is permitted to be equal to x in Fig. 4.2 if the lower boundary must reside below the upper boundary. The labeling scheme is a natural consequence of the ways in which the boundaries of two surfaces can cross. To assist in the visualization and comprehension of the labeling scheme, it is helpful to motivate it using a set of real surfaces as an example. Fig. 4.3 (left) shows a photograph of three pieces of paper in an overlapping arrangement. Fig. 4.3 (right) shows the labeled knot-diagram that describes this scene overlayed on the original photograph. From the information provided in Fig. 4.3 (right) it is possible to motivate the rules of the labeling scheme. In the crossing on the left side of the photograph, the two boundaries cross with no intervening surface between them. Such a crossing corresponds to a situation in which $y = x$. In the crossing in the center of the photograph, there is an intervening surface between the two crossing boundaries. Such a crossing corresponds to a situation in which $y > x$.

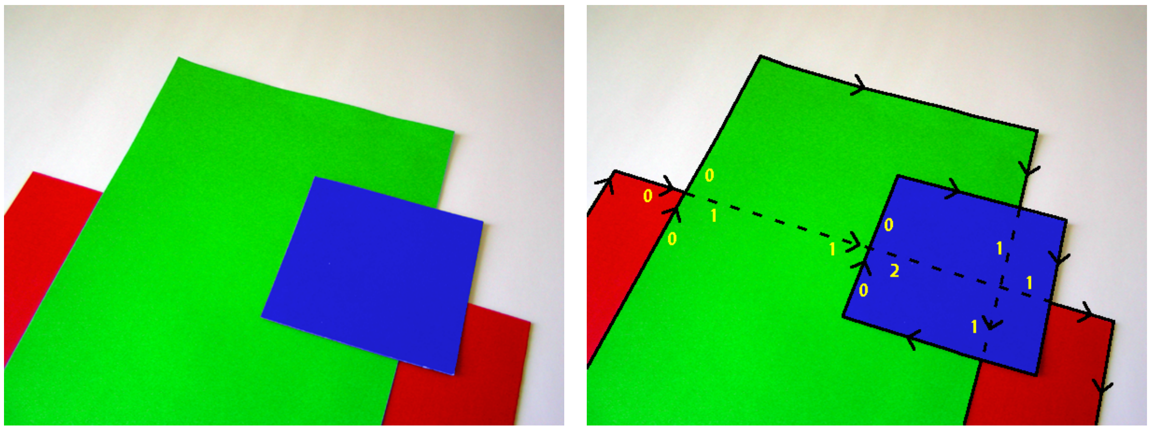

Figure 4.3: The labeling scheme can be motivated by studying a scene of real surfaces. The left figure shows a photograph of overlapping pieces of paper. The right figure shows the labeled knot-diagram for this scene overlayed on the original photograph. The rules of the labeling scheme (Fig. 4.2) follow from a study of the relative depths of boundary segments at crossings.

Chapter 5

Labeled Knot-Diagram Spaces

For a given knot-diagram, there are numerous possible labelings. Only a small subset of those will be legal, *i.e.*, will conform to the labeling scheme at all crossings and thus describe a set of plausible physical surfaces. A simpler program than *Druid* might let the user manually assign elements to a labeling in the hopes of successfully constructing a legal labeling. However, not only would such an approach be tremendously burdensome on the user, it would not possess the correct affordances, *i.e.*, user interactions would not correspond to elemental scene changes. *Druid* is much more than a labeled knot-diagram editor. It is an intelligent 2½D scene editor. It understands the semantics of 2½D scenes, *i.e.*, it can distinguish legal labelings from illegal labelings by using the labeling scheme as a mechanism for interpreting a potential labeling.

Druid automatically produces a legal labeling for the user whenever possible. This happens despite the fact that there are a large number of illegal labelings for a given figure. It maintains a legal labeling by automatically finding a new legal labeling without any prompting from the user whenever a labeling invalidating change occurs, *i.e.*, a topological change such as the creation of new crossings. The space through which *Druid* searches for a new labeling is called the *search space*.

In the previous chapters we described *Druid's* representation, labeled knot-diagrams, and we described a constraint on 2½D scenes, the labeling scheme. We then demonstrated how *Druid* is used with a sequence of screen captures. In this chapter we describe the search space. Later in this chapter we describe the depth ranges of a labeled knot-diagram. This information must be determined before the labeling search can proceed. Finally, we present the *relaxed labeling scheme*, similar to the original labeling scheme, which is used to determine the depths ranges for a labeled knot-diagram.

5.1 The Search Space

Druid must search through a space of possible labeling assignments in order to find a legal labeling when a user-initiated change occurs. Given an unlabeled drawing, there exists a set of possible legal labelings that can be assigned to that drawing. When the user interacts with the drawing, the interaction may invalidate the current labeling. The task for *Druid* is to fix the illegal part of the drawing by searching the labeling space for a new legal labeling.

The primary goal of the search is to find the *minimum-difference labeling* with respect to the labeling prior to the change. Therefore, this goal motivates the organization of the search process. The user communicates his intent by specifying a single constraint on the new labeling. *Druid* then deduces the remaining constraints by searching for a legal labeling that satifies the user's explicit constraint. In this way, *Druid* deduces the user's intentions automatically, thereby minimizing the user's effort.

The size of the search space $L(t \in T)$ corresponds to the number of distinct labeling assignments that are possible for the underlying knot-diagram, regardless of whether those labelings are legal or illegal. The space of consistent labelings $C(t \in T)$ is a proper subset of the space of all labelings $L(t \in T)$:

$$C(t \in T) \subseteq L(t \in T) \tag{5.1}$$

where:

1. T is the infinite space of all drawing topologies
2. $t \in T$ is a specific instance of T
3. $L(t \in T)$ is the space of possible labelings given t
4. $C(t \in T)$ is the subset of L that is consistent (legal).

T consists of all possible knot-diagrams with unspecified crossing-states and no depth indices, but with specified signs of occlusion. In other words, each $t \in T$ is a unique partial labeling, meaning that some elements are labeled and others are not. As stated, signs of occlusion are specified in instances of T, but crossing-states and segment depths remain unspecified (Fig. 5.1) because the user specifies signs of occlusion when he creates boundaries.

For any individual $t \in T$ there exists $L(t)$ the space of possible labelings for that specific drawing topology (Fig. 5.2). $C \subseteq L$ is the subset of *consistent* (legal) labelings within L, *i.e.*, labelings in which all elements of the knot-diagram satisfy the labeling scheme and consequently which can be rendered (Fig. 5.2, right column).

Both L and C are connected graphs (Fig. 5.3). In L, edges connect pairs of labelings that differ by a labeling distance $L_\Delta = 1$, *i.e.*, a single crossing-state or segment depth. In C, edges connect pairs of consistent, legal labelings that can be transformed into one another through a single user interface action such as a mouse click, *i.e.*, pairs of legal labelings that have an edit-distance, $C_\Delta = 1$.

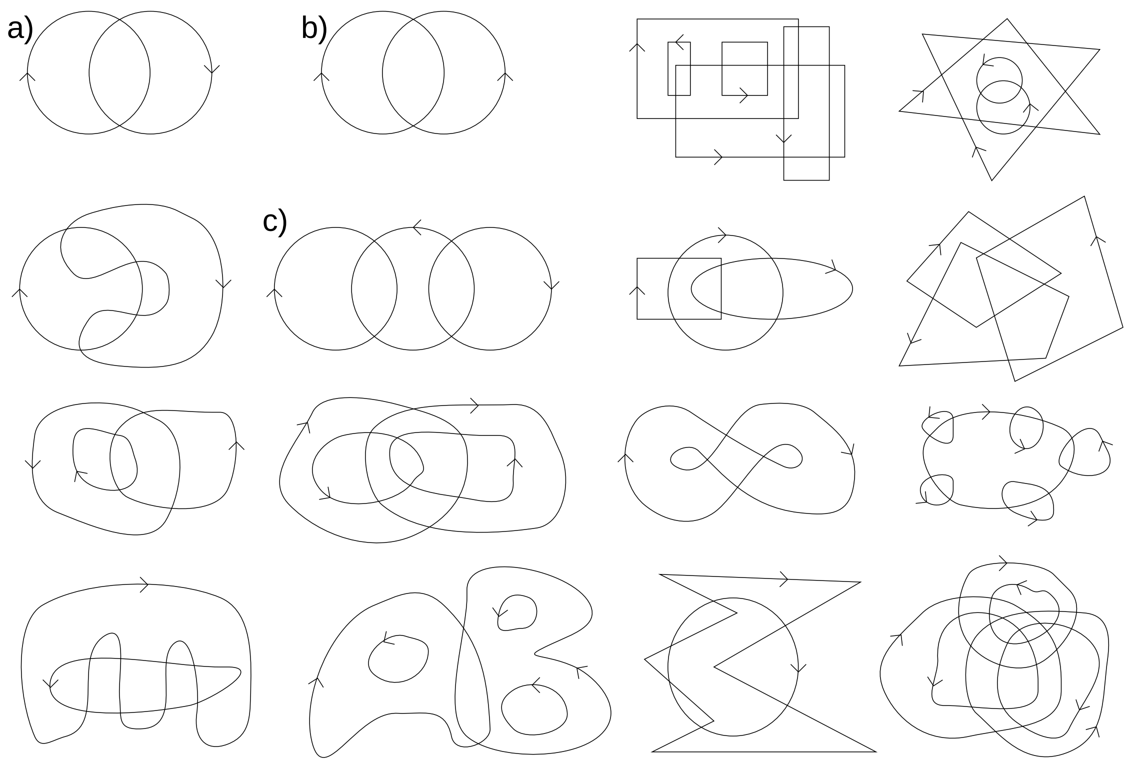

Figure 5.1: The space of all drawing topologies, *T*, is infinite. Some instances are shown here. Note that signs of occlusion are specified and distinguish otherwise identical topologies (*a* and *b*), but crossing-states and depth indices are not specified.

5.2 Graph Distance Between Labelings

The root of the search tree is a consistent labeling, a member of $C(t \in T)$, *e.g.*, node *v* in Fig. 5.3. The goal of a minimum-difference search originating from *v* is to find the consistent labeling with the minimum graph distance in *C* from *v*, *e.g.*, *u* or *x* in Fig. 5.3 (for both of which $C_\Delta = 1$ and $L_\Delta = 2$).

We can compute the size of $L(t \in T)$ and an upperbound on the size of $C(t \in T)$ by calculating the number of unique assignments of labeling features to a particular knot-diagram topology. Since crossing-states take on two values their number of combinations is 2^R for *R* crossings. This represents the size of *L*. Although a labeled knot-diagram

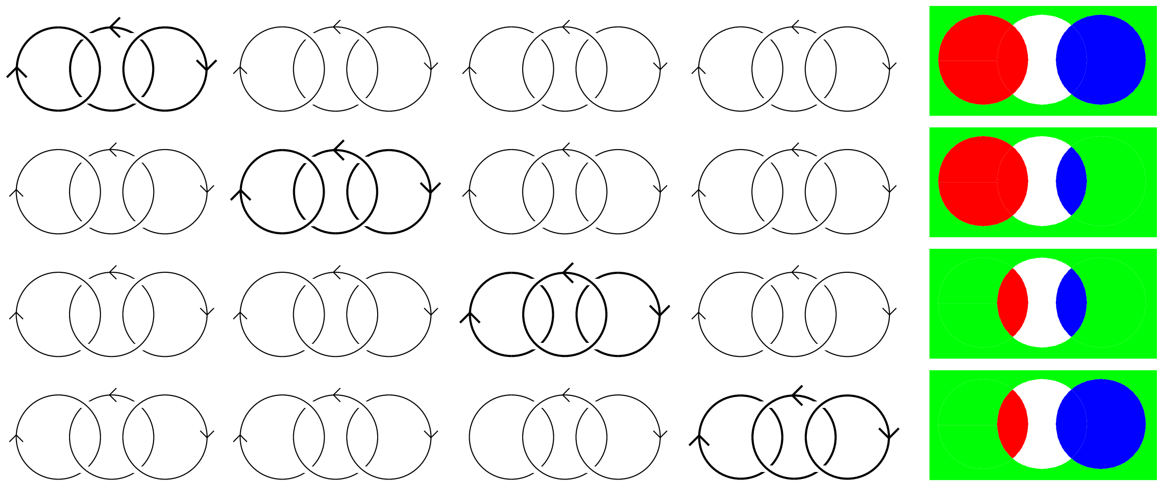

Figure 5.2: The space of labelings L for a particular topology, $t \in T$. The labeling space consists of all combinations of crossing-states for that topology. This figure shows $L(t \in T)$ corresponding to item c from Fig. 5.1. The four possible legal labelings are shown in bold with their corresponding renderings shown in the right column. They represent $C(t \in T) \subseteq L(t \in T)$. In this figure, horizontally and vertically adjacent labelings have a crossing-state difference of 1, *i.e.*, they are connected by an edge in L. Diagonally adjacent legal labelings (bold) can be transformed into one another with an edit-distance of 1, *i.e.*, they are connected by an edge in C. The observation that adjacent pairs of legal labelings on the diagonal have a difference of 1 is the same as the observation that adjacent pairs of the the four renderings in the right column have a difference of 1. Note that the labeling space is periodic in both dimensions, so the top row is adjacent to the bottom row, etc.

includes signs of occlusion and boundary segment depths, we don't include either of these variables in the search space. The user designated the signs of occlusion when he created the boundaries and boundary segment depths uniquely follow from the crossing-states, so neither is relevant to the search. Since L is a superset of C, the size of L, *i.e.*, 2^R, represents an upper bound on the size of C. Note that while boundary segment depths are easy to determine on a legally-labeled figure, they are actually impossible to determine on an illegally labeled figure. In fact, we define an illegal crossing-state instantiation as one for which there is no boundary segment depth assignment which satisfies the labeling scheme at all crossings in the drawing.

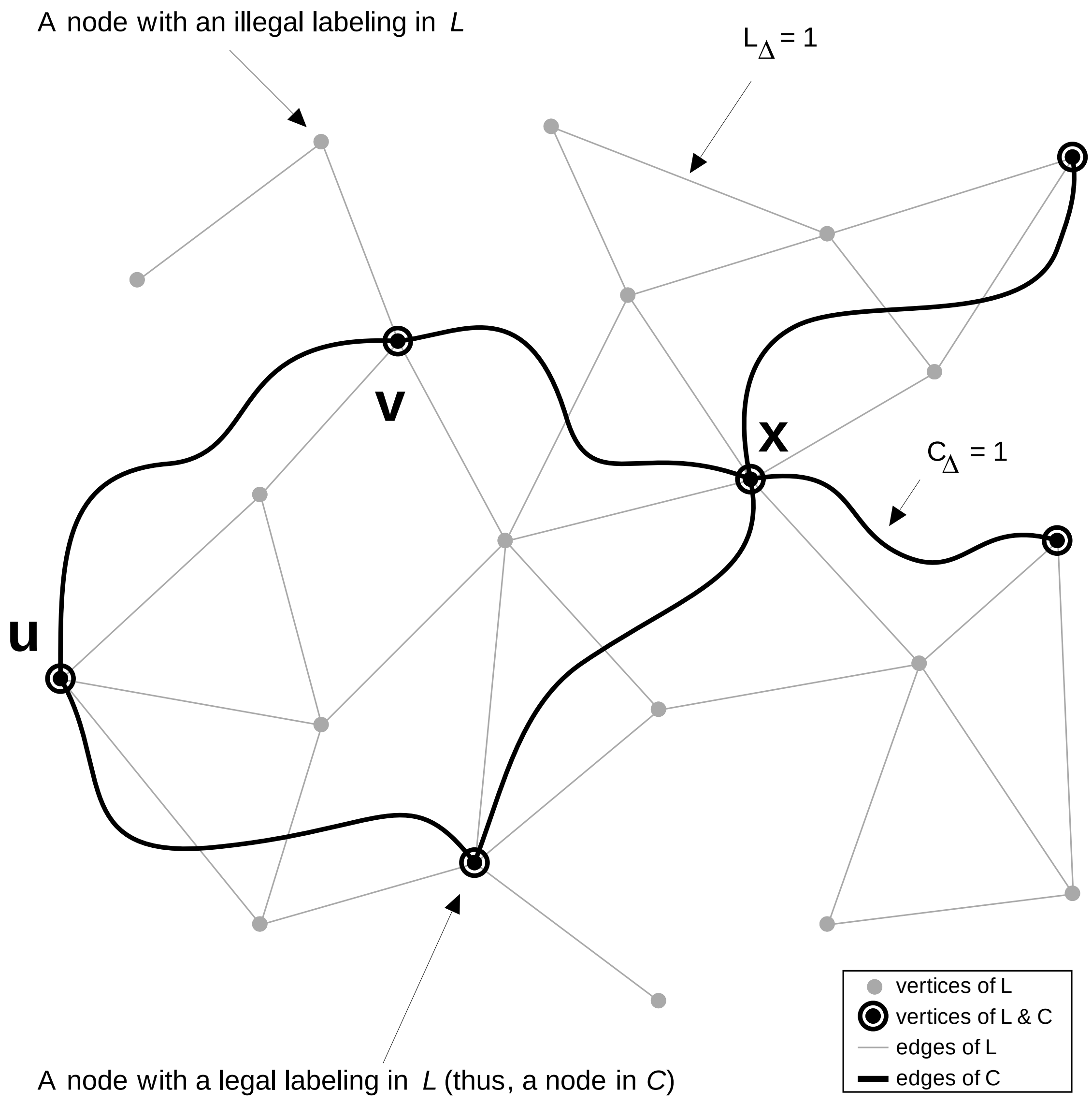
A node with an illegal labeling in L
$L_\Delta = 1$
v
x
$C_\Delta = 1$
u
A node with a legal labeling in L (thus, a node in C)
● vertices of L
◉ vertices of L & C
edges of L
edges of C

Figure 5.3: $L(t \in T)$ is a connected graph where nodes are labeled figures and edges denote labeling distance $L_\Delta = 1$. $C(t \in T) \subseteq$ L contains the consistent, legal labelings within L. $C(t \in T)$ is also a connected graph. Edges of C (shown as curves) denote user interface edit-distances of 1, *e.g.*, a single mouse click.

For drawings of a moderate size *L* can be extremely large given that we want *Druid* to perform fast enough to not annoy the user. Fast feedback is an important aspect of direct manipulation interfaces. Norman and Draper argue that fast feedback reduces the user's awareness of the computer as a barrier between themselves and the drawing. In our case, this contributes to the user's perception that he is interacting with real surfaces (see Norman [34]).

In Fig. 5.4, the size of *L* for each of the three drawings from left to right is one, four, and sixty-four respectively, while the size of *C* for the three drawings is 1, 2, and 6 respectively. This figure illustrates the fact that the size of *L* increases rapidly relative to the complexity of the drawing and that the ratio of the size of *C* to the size of *L* drops off quickly relative to the complexity of the drawing. Since *L* represents the potential search space and *C* represents acceptable solutions to the search, there is an inherent challenge in performing the search quickly. Performing the search quickly is crucial since it must be done fairly often and should occur with minimal inconvenience for the user. The primary focus of our research has been to find methods which can perform the search fast enough to not annoy the user, namely turnaround times of less than a second.

5.3 Calculating the Depth Ranges for a Labeled Knot-Diagram

The depth ranges that a boundary can assume need to be calculated before the figure can be labeled. Therefore, an efficient algorithm for finding the depth ranges is required. To solve this problem we have framed the task of finding the depth ranges for all segments for a particular topology as a new kind of labeling problem, similar to the labeling problem described at the beginning of Section 5.1. The challenge is to label a knot-diagram in the fashion shown in Fig. 5.4, where crossing-states remain unspecified but all segments have a range of possible depths associated with them. This *relaxed labeling problem* is similar to the original labeling problem where the goal is to assign a single index to each segment. In the original labeling problem the index is the actual depth of the segment in a topologically valid labeling. In a relaxed labeling the index is the *depth range* for a segment, *i.e.*, the maximum depth that a segment can assume among all topologically valid labelings.

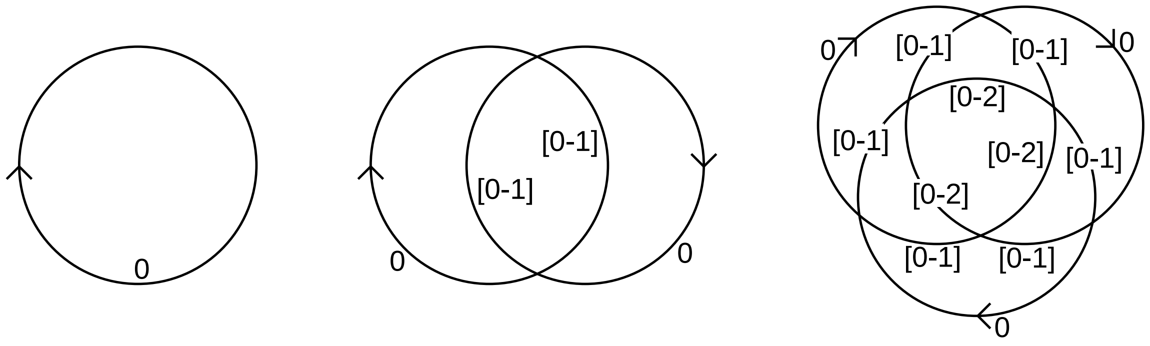

Figure 5.4: For a particular labeled knot-diagram, each boundary segment has a range of possible depths that it can assume, depending on how many surface coverings it overlaps. These figures show three simple examples with the depth ranges for each boundary segment indicated.

The constraints of the relaxed labeling scheme in Fig. 5.5 can be stated as follows:

1. Segment *B* must have a depth range that is one deeper than segment *A*.
2. Segment *C* must have a depth range that is one deeper than segment *D*.
3. Segments *A* and *D* must have the same depth range (and likewise for segments *B* and *C*).

The solution to the relaxed labeling problem can be formulated as a system of linear equa-

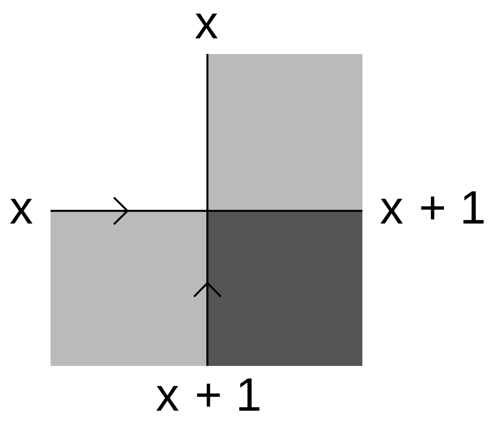

Figure 5.5: The *relaxed labeling scheme* is a set of constraints on the depth ranges of the four boundary segments that meet at a crossing. Each boundary occludes the half-plane on its right. Thus, for the two boundaries meeting at a crossing, one half of each boundary will lie in the *unoccluded* half-plane of the opposing boundary, and the other half will lie in the *potentially occluded* half-plane of the opposing boundary. The shaded regions in the figure show the potentially occluded half-planes of each boundary. The two segments with a depth range of x are unoccluded. The two segments with a depth range of $x+1$ are potentially occluded. The relaxed labeling scheme requires that the potentially occluded segments of a crossing have a depth range that is one greater than the depth range of the unoccluded segments and that the two boundaries have the same depth ranges as each other.

tions which can be solved by Gauss-Seidel or Jacobi iteration:

$$\begin{bmatrix} -1 & 0 & 1 & 0 & \dots \\ 0 & 1 & 0 & -1 & \dots \\ 0 & 1 & 1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} A \\ C \\ B \\ D \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \tag{5.2}$$

where A, B, C, and D are the segment depth ranges for a particular crossing and the right column vector according to Fig. 5.5.

Unfortunately, generalized equation solving methods such as Gauss-Seidel or Jacobi iteration do not take advantage of constraints specific to the problem at hand, *e.g.*, once a potential depth range for a segment has been found, an intelligent algorithm would only permit deeper depth ranges in subsequent iterations and never revert to shallower depth ranges. This constraint results from the fact that any shallower depth range is naturally a subset of a deeper depth range. Therefore, once a depth range has been found, there is no reason to ever consider shallower depth ranges, only deeper depth ranges. Neither Gauss-Seidel nor Jacobi iteration takes advantage of this fact. Consequently, these general purpose methods are not necessarily an efficient way to solve this system of equations. Additionally, iterative methods are not always guaranteed to converge to a solution or convergence might be quite slow. For these reasons, we have devised an iterative algorithm that determines the depth ranges by acting directly on the knot-diagram in a series of iterative passes, starting from depth zero and accumulating maximum depths for the segments incrementally. The algorithm terminates when the depth ranges converge on their deepest possible values.

Chapter 6

Editing 2½D Scenes

In this chapter we begin by comparing the concepts of scene editors and drawing representation editors. We describe how conventional drawing programs are representation editors while *Druid* is in a sense we will expand on in detail, actually a *scene* editor. Following this discussion we describe how *Druid* handles *labeling preserving interactions*, defined below.

In previous chapters we have described labeled knot-diagrams, legal labelings, and the search space. In Chapter 7 we will describe the method *Druid* uses to search through the search space for a legal labeling. Before we do that, however, it is necessary to distinguish between those user-interactions which require a search for a new legal labeling and those which do not. The interactions presented in Section 2.1 can be classified into two categories: *labeling preserving interactions* and *interactions requiring relabeling*. Labeling preserving interactions must be performed such that crossing-states are preserved throughout their course. This requirement precludes the naive method of deletion and rediscovery of the crossing-states. Therefore, *Druid* uses a process we call *crossing-projection* to calculate the new location of a crossing after it has moved. We describe crossing-projection in this chapter.

Later in this chapter we describe two methods which *Druid* uses to improve its performance and reduce the risk of crossing-projection errors. These two methods are *delayed response* and *minimum acceptable mouse distance*. Interactions come in two forms, different from the two classifications listed above. They can be continuous or discontinuous. Dragging a boundary is continuous for example, while flipping a crossing is discontinuous. During continuous interactions, we want the fastest turnaround times possible in order to achieve smooth animation-like feedback instead of stuttered frame-by-frame feedback. Delayed response is a method for maximizing the likelihood of fast smooth response times during continuous user-interactions. Minimum acceptable mouse distance is a method for minimizing crossing-projection errors. If the mouse moves too far in a single step, there is a risk that *Druid* will suffer from errors in its crossing-projection algorithm. Minimum acceptable mouse distance helps *Druid* minimize the risk of these errors by imposing a minimum distance between perceived mouse interrupt locations.

6.1 2½D Scene Editors vs. Drawing Representation Editors

At the beginning of Chapter 5 we described how *Druid* is much more than merely a labeled knot-diagram editor, it is actually a 2½D scene editor. Unlike *Druid*, *Real-Draw* essentially is an editor for its internal representation. The internal representation of *Real-Draw* is a global DAG with localized regions specifying local DAG changes. The manipulations the user performs in *Real-Draw* equate *edit-distances* of exactly one with *representation distances* of exactly one. An edit-distance is the total number of mouse clicks and key presses necessary to transform one instance of a program's representation into another. A representation distance is the number of edges along the shortest path connecting two representations in the graph of all representations, given a graph of representations in which the presence of an edge signifies an elemental transformation between two representations.

An elemental transformation on a representation is a transformation in which a single parameter of a representation is altered. Since *Real-Draw* equates edit-distances of one with representation distances of one, the user is directly manipulating *Real-Draw's* representation. Direct manipulation of the representation not necessarily a good design. There are situations where the user's intention may require navigating a large representation distance to achieve a small 2½D scene transformation distance, thus requiring the user the manually transform the starting representation to the desired representation through several intermediate representations. This long sequence of steps is tedious and distracts the user from his larger goals. For example, if the user wants to invert the ordering of a pair of overlapping surfaces, that goal corresponds to a 2½D scene transformation of size one, *i.e.*, an *elemental scene transformation*. However, in order to achieve such a manipulation with *Real-Draw*, the user must perform several steps. The push-back tool must be selected, the push-back object must be created and carefully placed in the proper location, and it is possible the push-back's default reordering must be adjusted to properly alter the depths within the push-back's region. Therefore, the actions of the user do not correspond to transformations of the scene, the user's actions correspond to transformations of the representation. Consequently, the affordances of *Real-Draw* are not isomorphic with those of 2½D scenes. Rather, they are isomorphic with transformations of *Real-Draw's* internal representation, which is of no actual interest to the user. To summarize, *Real-Draw* represents a genuine improvement over spoofs because it requires far fewer and less delicate steps than construction of a spoof requires, but it still requires more steps than would be necessary using a program which provides the natural affordances of 2½D scenes.

Druid's power derives from its ability to quickly search through the space of labeled knot-diagrams for legal labelings. It presents the user with the experience of directly interacting with a 2½D scene, including interwoven scenes, rather than the experience of interacting with one scene that has the appearance of a desired scene, *i.e.*, creating a spoof. The reason the user has such a qualitatively different experience when using *Druid* is that elemental scene transformations can be accomplished with single mouse clicks. It is this isomorphism between editing operations and 2½D scene transformations that makes *Druid* so novel.

6.2 User Interactions

All drawing programs are, at some level, editors for their underlying representation. However, more so than with other drawing programs, *Druid's* interface is abstracted away from its internal representation. It is, in effect, an editor for 2½D scenes. Consequently, the user has the experience of interacting with a 2½D scene instead of assigning labels to a labeled knot-diagram. Several possible user interactions were listed in Section 2.1. The effects of these interactions on the labeled knot-diagram can be grouped into two major categories:

- Labeling-preserving interactions
- Interactions requiring relabeling.

Labeling-preserving interactions are interactions in which the topology of the labeled knot-diagram does not change. In contrast, *interactions requiring relabeling* are of the following types:

- Drags or reshapes that create new crossings
- Drags or reshapes that delete existing crossings
- Drags or reshapes that change the order of existing crossings along the boundaries
- Change of the crossing-state of a crossing (*flipping* a crossing)
- Change of the sign occlusion of a boundary (flipping a sign of occlusion).

In the following sections we describe how *Druid* handles these two kinds of interactions.

6.2.1 Labeling-Preserving Interactions

Ideally, *Druid* should preserve the labeling during user interactions whenever possible because doing so provides a sense of continuity for the user. In other words, we assume that the user does not want the labeling to change arbitrarily while he is altering a drawing because if it did, it would prevent him from being able to construct the 2$^1/_2$D scene he has in mind. Labeling changes should only occur as the result of explicit constraints that the user specifies. At all other times, the labeling must be preserved so the user can maintain control over the drawing process.

When one boundary is dragged over another boundary, the crossings involved in both boundaries will move. The goal of preserving the crossings' states during such interactions precludes the naive method of deletion and rediscovery of crossings since such an approach would destroy the crossing-states. The effect of destroying the crossing-states is that the knot-diagram would have to be relabeled before the labeling would be legal again. While relabeling can be performed fairly quickly, efficiency is not the only concern. It is also crucial that the labeling following a non-topology-altering interaction match the labeling prior to the interaction. Because there is no way of guaranteeing that the new crossing-states (those discovered after relabeling) will match the old crossing-states, the naive method is infeasible. The only alternative is to avoid the relabeling whenever possible.

For some interactions, *e.g.*, drags and reshapes which do not alter the topology, the labeling can be preserved by projecting crossings along the paths they follow on the boundaries (Fig. 6.1). This method is more complicated to implement than the naive method, but it is the key to *Druid's* responsiveness.

The process of projecting crossings to their new locations during a move or reshape of a boundary is called *crossing-projection*. The algorithm that performs crossing-projection is simple in its outline but more complicated in its details. The goal is to explicitly complete

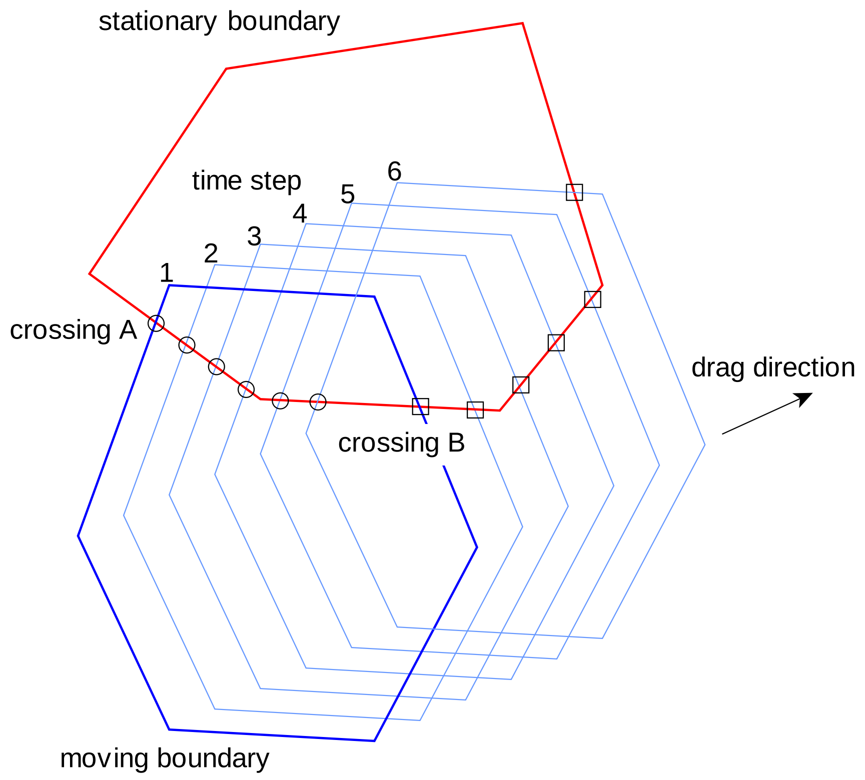

Figure 6.1: Dragging one boundary over another does not always alter the topology of the drawing. In such cases, it is best to preserve the crossings by predicting their new locations rather than deleting crossings and rediscovering them from scratch. In the above figure, the topology of the knot-diagram does not change over time. Only the locations of the two crossings change.

the path that a crossing follows around a boundary. This algorithm is relatively complex because there are a number of special cases that must be properly detected and handled, *e.g.*, the disappearance of a crossing, which requires relabeling.

The algorithm is illustrated using a detailed example in Fig. 6.1, where the user has dragged the lower boundary in a diagonal direction toward the upper-right in six discrete timesteps. Observe that the boundary moves discretely, from one location to another, and not continuously. These discrete jumps result from two factors. The first is that there is a latency between mouse-generated hardware interrupts, during which the mouse will drag a shape an unspecified distance. The second factor is the pixelization of the canvas that the user is drawing on. B-spline control points can only be dragged to and from integer pixel coordinates, and therefore exist in a discrete space. For these reasons, the crossing-projection algorithm assumes discrete timesteps and crossing locations.

Fig. 6.1 shows a straight-line *drag* consisting of six timesteps. Between Timesteps 4 and 5, crossing A moves from one segment of the stationary boundary to an adjacent segment of the same boundary. Likewise, crossing B switches segments on the stationary boundary between Timesteps 2 and 3, and again between Timesteps 5 and 6. Additionally, crossing B switches segments on the moving boundary between Timesteps 5 and 6. Although not illustrated in Fig. 6.1, it is in fact possible for a crossing to traverse more than one segment during a single timestep, especially when the segments are very short, or when the direction of motion is nearly orthogonal to the segment's orientation. The process by which a crossing is projected over a distance of many segments in a single timestep is shown in detail in Fig. 6.2. As a further example of the difficulty of the task of predicting the motion of crossings, notice that between Timesteps 5 and 6 in Fig. 6.1, crossing B switches segments on both of its associated boundaries in a single jump. Fig. 6.2 illustrates this case in greater detail. *Druid* must be able to handle all these cases.

The crossing-projection algorithm is invoked after each timestep. All crossings that the boundary's movement affects are immediately projected to their new locations. Since Fig. 6.1 illustrates six distinct timesteps, it represents six invocations of the crossing-projection algorithm for each of the two crossings shown. The algorithm requires that the positions of all boundaries be known both before and after the motion. First, the users drags a boundary. Second, *Druid* detects that a change has occurred and calculates the new location of the boundary. Third, the crossing-projection algorithm is performed to project the crossings to their new locations.

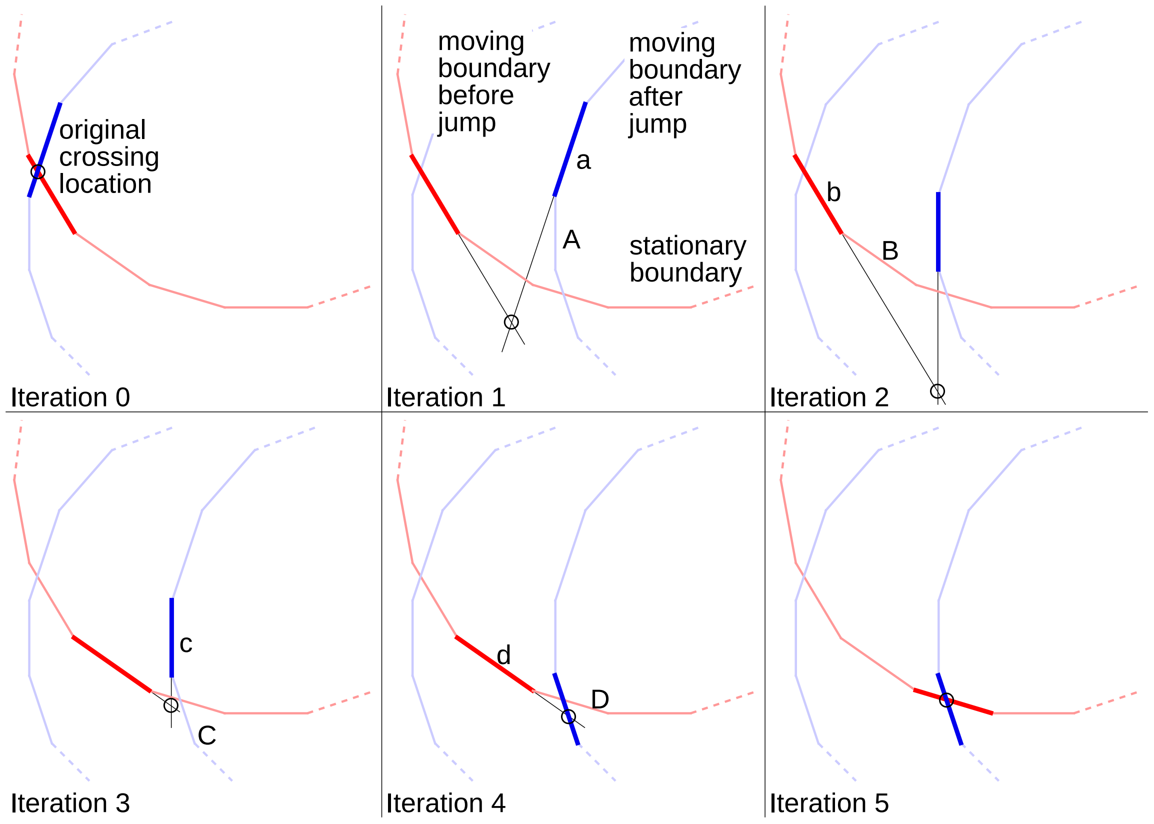

Figure 6.2: The sequence of drawings shown here illustrate successive iterations of the crossing-projection algorithm. Thick line segments show the segment pair associated with the crossing at the beginning of each iteration. Circles show the intersections of the lines containing the segments. Lowercase labeled segments (a-d) show the segment that will be switched out of the pair assignment at the end of that iteration. Uppercase labeled segments (A-D) show the segment that will be switched into the pair assignment at the end of an iteration. After a timestep, the algorithm tests the original segments assigned to the crossing (shown in Iteration 1). In the example illustrated, the segments no longer intersect. Since the error (the distance past the end of each segment to the point of intersection of the lines containing the segment pair) is greater for the moving boundary at the beginning of Iteration 1, the moving boundary segment assigned to the crossing's segment-pair assignment will be switched from *a* to *A*. The process continues until a pair of segments is found that actually intersect, as shown at the beginning of Iteration 5.

crossing-projection is performed by looping over all boundaries that have just been altered in an outer loop, and then looping over all the crossings of each altered boundary in an inner loop. Each crossing is individually projected to its new location.

crossing-projection is trivial if a crossing remains on its two assigned boundary segments after a boundary moves to its new location. If a crossing's two boundary segments still intersect after a drag or reshape is performed, then projecting the crossing is simply a matter of updating the coordinates for the crossing. Handling such a situation is trivial because *Druid* does not have to determine which segments the crossing has moved to, since the crossing remains on the original segments. If the two original segments of the crossing no longer intersect after the boundary is moved, then the more complex crossing-projection algorithm described below must be performed, in which a new pair of segments is found which intersect at the new boundary location.

Fig. 6.2 illustrates how a single crossing is processed for a single timestep. It illustrates a case in which the projection will be complicated to perform because the crossings have moved many segments on both boundaries. Given a pair of segments belonging to a crossing that no longer intersect after a boundary has been moved or reshaped (shown at the beginning of Iteration 1 as a pair of bold segments), the algorithm enters a loop. Each iteration of this loop updates the segment-pair assignment for the crossing by reassigning one of the two segments for the crossing and preserving the other. The loop terminates when the presently assigned pair of segments intersect.

Observe that between iterations in Fig. 6.2, one segment is retained while the other segment switches to an adjacent segment of its boundary. The decision about which segment to retain is made by measuring the two errors (the distance past the end of each segment to the intersection of the lines containing the segment pair) of the crossing with respect to the two segments. The error for one segment is the distance between the crossing and the nearest end of that segment. Notice that at the beginning of Iteration 1, the error is less for the stationary boundary but that at the beginning of Iteration 2 the error is less for the moving boundary. In each iteration, the segment with less error is retained and the segment with the greater error is replaced with the adjacent segment of its boundary in the direction closest to the point of intersection.

In some iterations, the crossing will only be in error on one of the two segments, *e.g.*, Iteration 4. In such cases, the error for the segment that contains the crossing is zero. Consequently, that segment is retained and the other segment is switched, *e.g.*, as shown between Iterations 4 and 5.

The loop repeats until the pair of segments actually intersect, as shown at the beginning of Iteration 5. The new location of the crossing is then calculated and the crossing has been successfully projected to its new location.

6.2.2 User Interactions Requiring Relabeling

While some user interactions do not require relabeling, other interactions cause changes to the knot-diagram's topology, and therefore, require a search for a new legal labeling. Such interactions include:

- Drags or reshapes that create new crossings
- Drags or reshapes that delete existing crossings
- Drags or reshapes that change the order of existing crossings along the boundaries
- Change of the crossing-state of a crossing (*flipping* a crossing)
- Change of the sign occlusion of an boundary (flipping a sign of occlusion).

Devising an algorithm that can find the minimum-difference labeling quickly is difficult because the search space can be extremely large relative to the complexity of the drawing. The crossing-projection method described above helps to avoid unnecessary search, but at other times, such as those just listed, a search is required. The labeling search is described in Chapter 7.

6.3 Delayed Response

Druid should handle reasonably complex drawings with acceptably fast turnaround times for as many user interactions as possible. However, some tasks are computationally expensive and consequently difficult to complete within acceptable turnaround times. Some tasks that *Druid* must perform include:

- Manipulation of B-splines, *e.g.*, addition, deletion and translation of control points (this will require recalculating the piecewise linear approximation of the splines)
- Redrawing the screen when *Druid's* view is in labeled knot-diagram mode
- Redrawing the screen when *Druid's* view is in rendering mode
- Tracking and projecting crossings as B-splines are manipulated
- Searching for equivalence classes on a labeled knot-diagram
- Assignment of a legal labeling to a knot-diagram (this might require a labeling search)
- Searching for cuts and grouping boundaries that bound the same surface.

The ideal turnaround time should be fast enough that the user perceives continuous changes, *e.g.*, drags and reshapes, as a smooth animation instead of as a series of discontinuous images since. Typically, smooth animation is achieved with a minimum framerate of between 0.03s to 0.1s of a second.

We are aware of at least two methods for decreasing the turnaround time during computationally expensive continuous changes. The first method is to operate on a simpler representation during a continuous change and then revert to the default level of complexity as soon as the continuous change is completed. One program that does this is *MetaCreations's Bryce 2*, a three-dimensional modeling program [29]. Bryce uses a typical *wireframe* method of visualization for most interactive tasks and only renders a scene when the user specifically directs it to do so. When the user rotates or moves the camera, the wireframe complexity used to illustrate the various models is simplified, *i.e.*, the wireframe models are constructed with fewer vertices and edges. Once the camera stops moving, the wireframe instantly reverts to the default level of wireframe complexity.

Another method for decreasing the turnaround time during computationally expensive continuous changes is what we call *delayed response*. Delayed response consists of classifying various tasks into two classes based on priority:

1. Critical priority
2. Default priority.

Critical priority is a subset of *default priority*, *i.e.*, all tasks of critical priority are also of default priority. The default behavior is to perform all tasks at all times. However, during times of continuous change, only tasks assigned critical priority are performed. By not performing more expensive but less crucial tasks during continuous changes, a program's turnaround time can be reduced.

One set of programs which uses delayed response is the *Myth* series of three-dimensional games by *Bungie* [11], in which texture smoothing on the three-dimensional terrain is disabled while the user is moving or rotating the camera, presumably to reduce the turnaround time, and is reenabled after a brief period of camera inactivity. *Druid* uses delayed response to decrease turnaround times during continuous changes as well.

The tasks listed above can be grouped into two major categories depending on their relationship to the knot-diagram and to the labeling:

1. Knot-diagram tasks (critical priority):
    - Manipulation of B-splines, *e.g.*, addition, deletion and translation of control points (this will require recalculating the piecewise linear approximation of the splines)

- Redrawing the screen when *Druid's* view is in labeled knot-diagram mode
- Redrawing the screen when *Druid's* view is in rendering mode
- Tracking and projecting crossings as B-splines are manipulated.

2. Labeling tasks (default priority):

   - Searching for equivalence classes on a labeled knot-diagram
   - Assignment of a legal labeling to a knot-diagram (this might require a labeling search)
   - Searching for cuts and grouping boundaries that bound the same surface.

In general, *knot-diagram tasks* are both less expensive and more crucial than *labeling tasks*, *i.e.*, knot-diagram tasks are of critical priority. They are more crucial because the knot-diagram cannot be presented to the user, regardless of whether it is legally labeled, if it is not maintained at all times.

Thus, *Druid* performs knot-diagram tasks with higher priority than labeling tasks. Whenever the user makes the slightest change to a boundary, *Druid* fully performs all knot-diagram tasks before returning control to the user. Since *Druid* fully performs the necessary computation before allowing the user to proceed, these tasks must be done extremely quickly or else the user's experience will be discontinuous.

Labeling tasks are not performed until the user has stopped moving the mouse for a specified window of time, *i.e.*, they are of default priority. We use 0.25s as this window of time. Thus, as the user smoothly drags a B-spline control point across the display, *Druid* continuously updates the piecewise linear calculation that estimates the spline's shape, continuously projects any moving crossings to their new locations, and continuously updates the display by redrawing the labeled knot-diagram in whatever viewing mode the user is currently using. However, *Druid* makes no attempt to label the knot-diagram until the user stops moving the mouse for 0.25s. The need for a new labeling is closely associated with most labeling tasks. Therefore, the task of assigning a new labeling broadly encompasses other tasks such as finding equivalence classes, searching for cuts, etc. The user's experience is that the labeling is not being maintained during the mouse motion, and that after stopping the mouse motion the labeling does not immediately update. However, very soon after stopping the mouse motion, the user sees the knot-diagram change to reflect a new labeling.

It is interesting to note that *Druid* attempts to update the display continuously, *i.e.*, as a critical knot-diagram task, regardless of whether the view is in labeled knot-diagram mode or in rendering mode. One potential problem is that while *Druid* can generally redraw the display in labeled knot-diagram mode quickly, rendering can take a significant amount of time. Thus, when performing continuous changes on a complex drawing in rendering mode, the user's experience can be unsatisfactory. Even very small mouse change can cause *Druid* to perform time-consuming rendering computations before updating the display and returning control to the user. Consequently, the most efficient way to use *Druid* is to do as much editing as possible in labeled knot-diagram mode and to not perform continuous manipulations in rendering mode. This point applies mainly to continuous changes. Discontinuous changes (*e.g.*, crossing-flips) perform reasonably well in rendering mode.

6.4 Minimum Acceptable Mouse Distance

Although delayed response isolates the most egregious tasks from continuous interactivity, knot-diagram tasks often cannot update fast enough for large mouse motions when the drawing is complex. If the user attempts to translate a spline control point or an entire B-spline a long distance very quickly, then the mouse positions that are reported by adjacent mouse interrupts might be far apart because *Druid* spent the intervening time performing knot-diagram tasks. When *Druid* must project crossings for a single large step, there is a reasonably high risk that it could make a mistake and lose track of the crossings. Such an error can be disastrous. If the crossings are lost, the drawing is effectively ruined and cannot be recovered. More seriously, this error can be difficult to detect, *i.e.*, *Druid* might not even realize that it has incorrectly projected crossings to their new locations. Therefore, crossing-projection errors are one of the most detrimental errors that can occur. To minimize the risk of this kind of error, we use a technique called *minimum acceptable mouse distance* as a buffer between the user's induced mouse movements and the mouse movements that *Druid's* mouse handling routines receive.

When a new mouse interrupt is delivered to *Druid*, it calculates the mouse distance with respect to the previous mouse interrupt. If the delta is within the *minimum acceptable mouse distance*, then *Druid* acts on the new mouse location directly. However, if the mouse distance exceeds the minimum acceptable mouse distance, then *Druid* creates a dummy mouse location that is projected from the previous mouse location toward the current mouse location by a distance of the minimum acceptable mouse distance. *Druid* then proceeds to act on the dummy mouse location (Fig. 6.3). *Druid* keeps track of whether such a *minimum delta exceeded event* has occurred. If so, then immediately after completing a series of knot-diagram tasks, *Druid* attempts to perform all knot-diagram tasks again, but it uses the most recent dummy mouse location as the location from which to project toward the current mouse location. In this way, the user can move the mouse quickly across the display without the risk of incurring errors in crossing-projection and *Druid* will catch up as quickly as possible. While *Druid* is continuously updating dummy mouse motions, we say it is in a *catch up phase*. During a catch up phase the user sees the control point he is altering (and by extension the shape of the spline being altered) smoothly moving across the display until it catches up with the mouse's current location. If he is translating an entire boundary, then during the catchup phase he sees the entire boundary moving steadily across the display toward the current mouse location.

If the user performs new mouse motions before a catch up phase is completed, *Druid* simply starts using the most recent actual mouse position as the point toward which to project dummy mouse locations (Fig. 6.3, *b*, *d*, and *e*). Thus, *Druid* will follow the mouse around the display at a pace which minimizes the risk of errors.

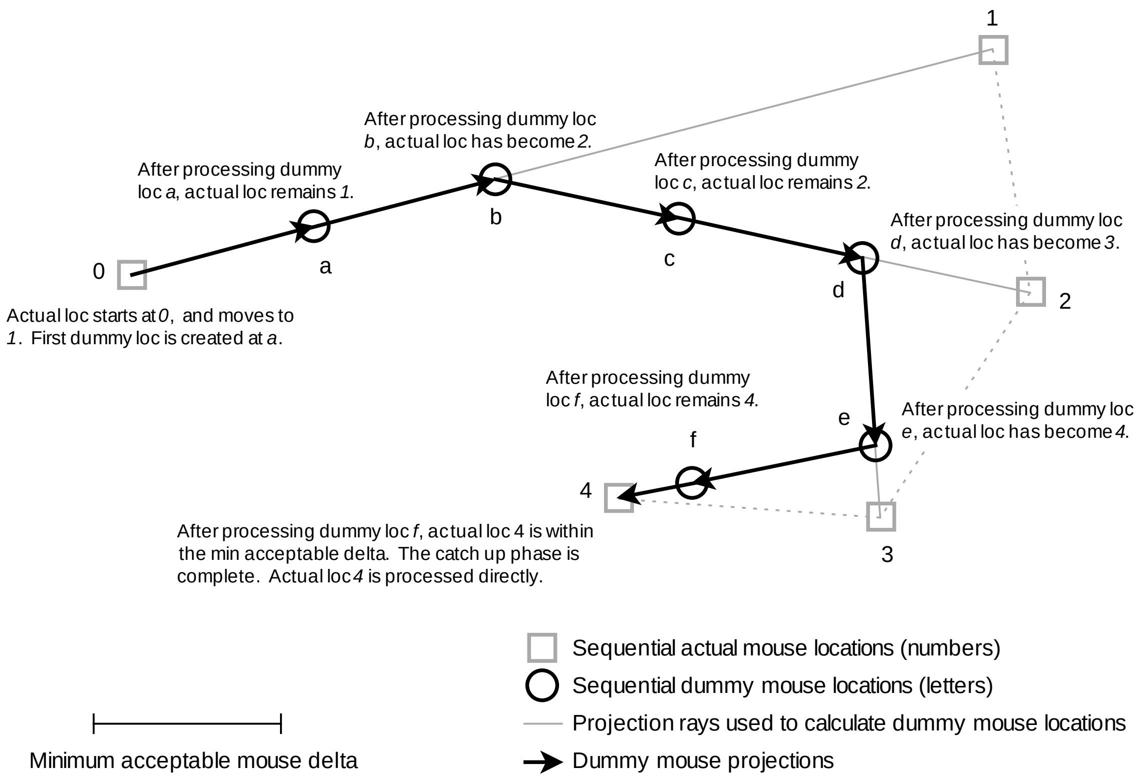
1
After processing dummy loc *b*, actual loc has become *2*.
After processing dummy loc *a*, actual loc remains *1*.
After processing dummy loc *c*, actual loc remains *2*.
After processing dummy loc *d*, actual loc has become *3*.
0
a
b
c
d
2
Actual loc starts at *0*, and moves to *1*. First dummy loc is created at *a*.
After processing dummy loc *f*, actual loc remains *4*.
e
After processing dummy loc *e*, actual loc has become *4*.
f
4
3
After processing dummy loc *f*, actual loc 4 is within the min acceptable delta. The catch up phase is complete. Actual loc *4* is processed directly.
Sequential actual mouse locations (numbers)
Sequential dummy mouse locations (letters)
Projection rays used to calculate dummy mouse locations
Dummy mouse projections
Minimum acceptable mouse delta

Figure 6.3: An example of how *Druid* uses dummy mouse locations as a buffer between actual mouse locations and its mouse handling routines. When the mouse moves too far in a single interrupt, *e.g.*, between actual mouse locations *0* and *1*, *Druid* enters a *catch up phase* in which it projects toward the mouse's actual location in a series of jumps of acceptable distance, creating dummy mouse locations along the way which *Druid's* mouse handling routines use. The catch up phase ends when the actual mouse location is within the minimum acceptable mouse distance of the most recent dummy mouse location. This happens between *f* and *4*.

Chapter 7

Finding a Legal Labeling

When the user performs an interaction that results in a topological change, *Druid* must assign a new legal labeling to the figure. In Section 5.1 we described the search space that *Druid* must search through to find a legal labeling. *Druid* must search through this space for a minimum-difference legal labeling as quickly as possible, so as to produce the most pleasant user experience possible. In this chapter we describe how this search is performed in the case of a *crossing-flip* user-interaction, *i.e.*, the user clicks on a crossing to flip its crossing-state, and in doing so, inverts the depth order of the two surfaces associated with that crossing. Later, we describe the heuristics that *Druid* employs in an attempt to complete the search in a reasonable time. Finally, we present experimental results which demonstrate the effectiveness of each of these heuristics and results which demonstrate the performance of the search relative to a drawing's complexity.

7.1 Crossing-Flip User Interaction

When the user clicks on a crossing to flip its crossing-state, he is imposing a constraint on the drawing that is inconsistent with the current labeling. *Druid* then attempts to find a legal labeling that satisfies the user's constraint. This will often require changing other elements of the labeling, such as the crossing-states of other crossings or the depths assigned to various boundary segments. Thus, the search solves a *constraint-satisfaction* problem initiated by the change in crossing-state that the user specified.

7.2 Overview of the Search

The following is a list of the major data structures and variables that the search process requires:

- A list of *touched boundaries*. Touched boundaries are boundaries that are crossed while traversing other boundaries.
- The depth ranges for all boundary segments.
- The *start* segment for each boundary (an arbitrary segment of each boundary, from which enumeration begins).
- The best solution (so far). This solution will be revised throughout the search as better solutions are found.

The search takes the form of a *constraint-propagation* process, similar to Waltz filtering (see Waltz [43]). Waltz's research illustrates how certain highly combinatorially complex systems can be reduced in complexity to uniquely determined solutions through a process called constraint-propagation. This term refers to the exploitation of local constraints on adjacent vertices in a graph. Thus, the assignment of a label to one vertex of a graph constrains adjacent vertices, which in turn propagate their own constraints deeper into the graph. By means of this process, it is often the case that a seemingly ambiguous system can be reduced to a single consistent labeling.

7.3 Boundary Traversal During the Search

Throughout *Druid's* search process, constraints are instantiated incrementally as the search process explores the labeling through *boundary traversal*, described below. As constraints propagate, each constraint imposes additional constraints on elements of the labeling. Thus, constraints propagate through the graph, vastly reducing the size of the combinatorial search space. We will illustrate the specific nature of the constraint-propagation later.

The search is performed by a set of *boundary traversals*. A boundary traversal begins at an arbitrary location on a boundary with a predetermined depth and visits each segment on the boundary, until it returns to the starting location. As a boundary is traversed, the traversal depth is altered where and in the way which is appropriate. Where the traversal goes under an intersecting boundary, the depth is incremented by one. Where it comes out from under an intersecting boundary, the depth is decremented by one. The purpose of performing a boundary traversal is to test the *zero integration rule* for the traversal. This rule states that a complete traversal must change depth by a net sum of zero, *i.e.*, for every step a boundary goes down (as a result of going under a surface at a crossing) the traversal must come back up again before the traversal is complete. This rule guarantees that a traversal ends at the same depth that it started at, which in turn is a requirement of a legal labeling.

The decision about which boundaries are traversed during the search is managed using the *touched boundary list*, a list of boundaries that have been crossed during the traversal of other boundaries. Since traversals visit every segment of a boundary, it is guaranteed that all boundaries that intersect a traversed boundary will be visited. As boundaries are touched, the new boundary is appended to the end of the touched boundary list. The touched boundary list is initially seeded with all boundaries that are illegal, *i.e.*, all boundaries that have at least one illegal crossing. In the case of a crossing-flip, the flipped crossing will initially always be illegal. Therefore, the touched boundary list is seeded with the two boundaries that intersect at the flipped crossing. When all boundaries on the touched boundary list have been legally traversed (a legal traversal satisfies the zero integration rule) a leaf in the search tree has been reached. This corresponds to a potential solution.

Note that once a boundary traversal has begun, the depth changes that occur during the traversal are uniquely determined. There are only three situations which can occur when a traversal reaches a crossing:

- If the boundary being traversed is on top at the crossing, the depth is not changed.
- If the boundary being traversed goes under at the crossing, the depth increases by one.
- If the boundary being traversed comes up at the crossing, the depth decreases by one.

Thus, the depths around the boundary are determined uniquely during a traversal, as the constraint propagates. In the absence of constraint, the full combinatorics of possible segment depth assignments for a particular boundary is the cumulative product of the depth ranges for all the segments for that boundary, which can be a very large number. However, when propagating constraints around a boundary incrementally, using the crossings as constraints on the depths of adjacent segments, the number of possible segment depth assignments is no longer combinatoric at all; there is, in fact, only one possible segment-depth assignment for the entire boundary. This vast reduction in the complexity of labeling a particular boundary is a crucial aspect of the search algorithm because it would take a prohibitive amount of time to search the full space of labelings.

While the differences in depth around a boundary are uniquely determined, the depth of the initial segment of the traversal is constrained, but not uniquely. The depth of the initial segment must be within the depth range for the initial segment, as shown in Fig. 5.4, but since any depth in the range might be the depth of the optimum solution, all depths in the depth range of the initial segment must be tested. Testing every depth in the depth range is accomplished by wrapping each boundary traversal within a loop that restarts the traversal with each of the possible depths for the initial segment.

At this point, we have described how a single boundary traversal is effected, and we have specified that the goal of a boundary traversal is to test the boundary with the zero integration rule. A failure of this test excludes the labeling that produces the illegal traversal from the set of potential solutions. We have described how the touched boundary list is initialized with all boundaries that are illegal at the beginning of a search and how the touched boundary list grows during the search as a result of a traversal visiting other boundaries at crossings. Finally, we have described how each single boundary's traversal is repeated multiple times, once for each of the possible depths of the initial segment. Next we will describe how the boundary traversal process is wrapped into a larger tree-search which performs multiple boundary traversals during the course of the search for a minimum-difference labeling.

7.4 Structuring the Search

The search for a minimum-difference labeling is a depth first search. There are two kinds of branch points in the search tree. The first kind was described at the end of Section 7.3; each of the possible depths for the initial segment of a traversal is the root of a distinct subtree which must be searched. The other branch point occurs when a boundary traversal arrives at a crossing. Each of the two crossing-states is the root of a distinct subtree. One subtree maintains the existing crossing-state, the other flips the crossing-state.

The following is a description of the entire backtracking search process. An empty touched boundary list is created. This list is initially seeded with all illegal boundaries, *i.e.*, the two boundaries associated with the crossing that the user flipped. The next boundary to be traversed is then chosen from the front of the touched boundary list. This boundary is then traversed repeatedly within a loop which enumerates each of the possible depths for the initial segment of that boundary.

As the traversal visits each crossing of the traversed boundary, the crossed boundaries are appended to the touched boundary list. When the traversal is completed, if the traversal satisfies the zero integration rule, the next boundary on the touched boundary list is chosen and the traversal process continues with the next boundary on the touched boundary list. If the traversal ends illegally the traversal backtracks to the last unflipped crossing in the traversal, flips that crossing, and continues traversing forward. Thus, all combinations of crossing-states for a boundary's crossings will be enumerated during the course of the search process.

If a traversal ends legally and there are no more boundaries on the touched boundary list, the search has reached a leaf, or a potential search solution, *i.e.*, a legally labeled figure. The solution is assigned a score based on its difference from the labeling that preceded the search and then the search continues by backtracking along the traversal in reverse, exploring other subtrees.

Each time a traversal enumerates all combinations of crossing-states for all crossings on the boundary, a single traversal is completed and the depth range iteration for that traversal continues to the next depth in the depth range. When the entire depth range has been enumerated, the next boundary on the touched boundary list is selected. When the entire tree has been fully explored, the search is completed. Of all the legal labelings, the one with the lowest difference relative to the labeling prior to the search is accepted. The new labeling is then displayed and the user can initiate a new interaction.

The accumulation of differences between labelings that the backtracking search process discovers and the prior labeling occurs in only two ways:

- When a crossing is flipped, the difference is incremented by one.
- When a segment depth does not match its original depth, the difference is incremented by one.

There are other potential *difference scoring functions* that could be employed. For example, the numerical difference between a segment depth and its original depth could be used, so that the accumulated difference for a particular segment might be greater than one. However, experiments have suggested that an alternate scoring function is unnecessary.

7.4.1 Branch-and-Bound

The goal of the search is to find a minimum-difference labeling with respect to the labeling that existed prior to the interaction. The difference between a candidate labeling found during the search and the prior labeling is accumulated, one δ at a time, during the search process as the boundary-traversal algorithm branches; either preserving features of the labeling ($\delta = 0$) or altering them ($\delta = 1$). The sum of all δ's between two labelings is termed L_Δ. Branching occurs when the boundary traversal arrives at crossings and as the starting depths of a traversal are enumerated. Notice that the accumulated δ can never decrease and that it is possible to exploit this fact during the search to improve the search performance by bounding the search.

If there were some way of knowing that the accumulated δ at any given point in the process had surpassed the L_Δ of the optimum solution, then there would be no need to continue exploring the subtree beneath that point in the search tree for potential labelings.

At the outset of the search, there is no way to exploit such knowledge because we cannot know in advance the L_Δ of the actual minimum-difference labeling. However, as the search proceeds, candidate labelings will be found. They might not be the optimum, but their L_Δ will be known, and although we cannot stop the search at the first legal labeling found because there is no way of knowing if it is the optimum, we can use its L_Δ as a bound on the subsequent search.

This process allows us to steadily increase the efficiency of the search as the search progresses. The search begins with an infinite bound. When a solution is found, the bound is tightened to that solution's L_Δ and the remaining search becomes more efficient as a result. Later, if a solution with an even lower L_Δ is found, the bound is tightened further, making the remaining search even more efficient. This process can quickly prune enormous parts of the search space from consideration, thereby vastly increasing the efficiency of the search.

7.5 Improving the Search

Because of the large size of the search space, searching for the minimum-difference labeling might take a considerable amount of time. In the worst case the entire space might need to be explored. Therefore, we employ a number of heuristics in an effort to find the minimum-difference labeling quickly enough to provide the user with a reasonable turnaround time. For the most part, these heuristics are not independent of one another, but work in tandem, each heuristic increasing the effectiveness of the others. The three heuristics *Druid* uses are:

- Choosing good boundary traversal starting segments
- Ordering the search to produce tight search bounds based on minimum-difference earlier rather than later
- Terminating the search process after a maximum permitted search time has transpired.

We discuss each of these heuristics in the next sections.

7.5.1 Choosing Good Boundary Traversal Starting Segments

Each time an untraversed boundary is selected from the touched-boundary list, that boundary is traversed within a loop over the possible starting depths for the segment where the traversal began. Naively, we might choose the starting segment arbitrarily, *e.g.*, by simply choosing the first segment in the segment list. However, since there is no way to know in advance which starting segment depth will yield the minimum-difference labeling, we have no choice but to enumerate all the possible depths for the starting segment. This enumeration increases the search time. If we choose a starting segment wisely, the search time can be greatly decreased.

Fig. 5.4 illustrates how we can choose a starting segment wisely. When a traversal starting location must be chosen, it is highly advantageous to choose a segment on the boundary with the minimum depth range for the entire boundary as the starting segment. This method for choosing a traversal starting segment can be easily accomplished if the depth ranges for all segments have been calculated in advance.

Experiments with this feature enabled and disabled have demonstrated significant gains in performance. Additionally, the benefit of this method appears to rise with the complexity of the drawing. Therefore, this feature is absolutely crucial when editing large complex drawings. In Section 7.6.1 we show experimental results that demonstrate the gains this heuristic can provide.

7.5.2 Ordering the Search

Bounding the search works best if we can find a good solution, *e.g.*, a solution with a tight bound, early in the search. By guessing that certain labelings have a better chance of being the optimum, we can order the search so that those labelings are explored first.

There are a number of criteria we can use to judge whether or not a potential labeling is a good candidate for early expansion. The purpose for formulating the problem as a search for the minimum-difference labeling is, as stated in Section 5.1, to allow *Druid* to anticipate the user's intentions. We assume that the user wants the minimum necessary change to occur. Similarly, we assume that the user expects most changes to occur within a relatively small region surrounding the location of the user specified constraint. This region is termed the *area of interest*.

If we order the search so that regions of the search space within the area of interest are explored first, then we can effectively enumerate all possible labelings which differ from the prior labeling only within the area of interest before considering any labelings which differ from the prior labeling outside of this area. Ordering the search in this manner has two benefits. First, it most likely reflects the user's intent. Second, the number of potential changes in a compact area is significantly smaller than the number of potential changes in the entire drawing, so we will find any solutions where changes are restricted to that area much more rapidly than we would find solutions where changes are restricted to the larger area.

One way to order the search so as to explore changes within the area of interest before changes outside it is to perform a breadth first search of the potential labelings. This would be feasible if the topology of the search tree is structured such that graph distances with respect to the user's area of interest in the knot-diagram correspond to depths in the search tree. In such a tree, exploring high levels of the tree before exploring low levels of the tree would cause labeling changes within the area of interest to be explored first. As it happens, the search tree is already organized in exactly this fashion. To appreciate this, one should keep in mind the distinction between the search space, which is a connected graph of labelings in which edges correspond to pairs of labelings differing by exactly one (Fig. 5.3), and the search tree, which represents the labelings the search algorithm explores. Since upper levels of the search tree contain labelings with a low L_{Δ}, they are within a bounded neighborhood of the area of interest.

Notice that although breadth first search has advantages, the algorithm described in Section 7.4 is a depth first search because subtrees are expanded regardless of the crossings' distances from the area of interest. Ordering the search so that crossings inside the area of interest are expanded before crossings outside the area of interest would be a better approach, and could be implemented using breadth first search.

Unfortunately, implementation of breadth first search requires a queue of partially labeled knot-diagrams, *i.e.*, the internal vertices of the search tree correspond to partial labelings, in which some aspects of the labeling are resolved and others are illegal or unresolved. Maintaining numerous partial labelings during the search would be costly, in terms of both storage and time. Thus, the overhead required to implement a breadth first search might be so costly that any benefits derived from using it would be negligible. In the worst case, the overhead required could incur a net cost instead of a net benefit.

Instead of using breadth first search, we can achieve many of the same advantages of breadth first search by executing a depth first search within an *iterative deepening* loop, which is commonly used in game tree search algorithms (see Marsland and Campbell [26]). We do this by calculating, in advance, the graph distance between every crossing in the knot-diagram and the crossing in the center of the user's area of interest. We then perform the depth first search described above in a loop where the search horizon increases by one after every iteration. As a boundary is traversed to test the zero integration rule, the traversal will potentially wander outside the area of interest. Without iterative deepening, all branches in the search tree would be expanded in the order they are encountered during a traversal. However, when wrapped within an iterative deepening loop with an increasing horizon, no crossings beyond the current horizon will be expanded. For example, in the first iteration of the loop, only the immediate neighbors of the crossing the user flipped are expanded. As a result, the traversal of a boundary in the initial loop will take linear time with respect to the number of crossings along a boundary.

It should be noted that of the two kinds of branch points mentioned above (crossing-states and boundary traversal starting depths), iterative deepening only constrains one. While *Druid* uses iterative deepening to constrain crossing-state branch points to crossings that lie within the iterative deepening horizon, it always selects traversal starting segments by choosing the segment of a boundary with the minimum depth range of all the segments on that boundary regardless of whether the chosen boundary segment lies within the iterative deepening horizon. Despite this inconsistency, applying iterative deepening to crossing-state branch points has demonstrated significant performance gains.

Not only does iterative deepening expand subtrees corresponding to crossings within the area of interest first and which are therefore more likely related to the user's goal, but it also finds labelings with small L_Δ's which provide stronger bounds early, which increases the effectiveness of the branch and bound search. Since there are only a small number of crossings within the area of interest, any solution that is found within that area will have a small L_Δ. Finding strong bounds early pays off heavily in terms of search efficiency.

7.5.3 Terminating the Search Process by Employing a Timeout

Even with all of the heuristics described so far, there exist drawings which are too complex to label efficiently. In such cases, the optimal solution is often successfully found early in the search, but does not provide a tight enough bound to truncate enough of the search space to guarantee that the search finishes quickly. This scenario is particularly unfortunate in light of the fact that the optimal solution has in fact been found early in the search process. Thus, the time spent finishing the search is effectively wasted (although there is no way of knowing this *a priori*).

Our solution is to employ two possible timeouts:

1. A very brief timeout if a solution has already been found during the search process
2. A longer timeout if no solution has been found yet during the search process.

Our values for these two timeouts have been 0.1s for the first timeout and 5.0s for the second timeout. We chose these values to satisfy the two design goals of providing fast response time if a solution is available and providing reasonable response time even if no solution is available. If a search has found a solution, the earlier timeout will be used, thus providing a faster response to the user. However, if no solution has been found, the search will continue up to the second timeout, thus raising the likelihood that a solution will be found before the search times out and *Druid* gives up. If no solution is found after the second timeout, *Druid* abandons the search and fails to return a solution to the user.

Note that the use of timeouts often means that it is not guaranteed that *Druid* will find the optimal solution (or any solution at all). However, in our experience, this almost never occurs. Iterative deepening is highly effective as a means for finding the correct solution, *i.e.*, the true minimum-difference solution, early in the search process.

The discussion so far has focused on the *crossing-flip* user-interaction, in which a user intentionally alters the depth ordering of overlapping areas of two surfaces. There are more complex interactions as well, as described in Section 6.2.2. These interactions are often more complex than flips because they can invalidate significant portions of the knot-diagram. Without going into the detail of how such situations are handled, it suffices to say that the search for a minimum-difference legal labeling in such situations is basically similar to the method described above. The management of the knot-diagram and the calculation of the knot-diagram L_Δ's is more complex, but the basic approach is the same.

7.6 Performance Improvement of Each Heuristic

This section presents experimental results of the benefits that can be achieved by employing the first two heuristics listed in Section 7.5, *i.e.*, choosing good boundary traversal starting segments and iterative deepening.

7.6.1 Choosing Good Boundary Traversal Starting Segments

In Section 7.5.1 we described a heuristic which consists of choosing a good boundary traversal starting segment. This heuristic calculates the maximum possible depth of all boundary segments in advance. When a boundary is traversed, the boundary segment of that boundary with the shallowest maximum possible depth is used as the boundary traversal starting segment. This heuristic minimizes the number of traversals which must be attempted for the boundary.

Since the naive method is arbitrary, it could choose a good starting segment or a bad starting segment as the traversal starting segment. Consequently, it is difficult to precisely measure *Druid's* performance when using the naive method. Instead, we have performed tests that compare the best and worst cases. In the best case, *Druid* chooses the boundary segment with the shallowest possible depth as the starting segment. This method corresponds to having this particular heuristic enabled. In the worst case, *Druid* deliberately chooses the boundary segment with the deepest possible depth as the starting segment. Since the naive method might choose the segment with the deepest possible depth, the worst case represents an upper bound on the naive method's running time.

Fig. 7.1 shows the drawing on which the running time tests for this heuristic were performed. This drawing is somewhat contrived to demonstrate the effect of the tradeoff between the best and worst cases. It is constructed so as to maximize the range of possible depths for the boundary segments of the cigar shaped surface. At the ends, the cigar has a depth range of one, *i.e.*, it can only assume one depth, zero, which is the depth range that will be enumerated for the cigar boundary when *Druid* employs the best case method. At the center, the cigar has a depth range of seven since the cigar overlaps seven disks, which is the depth range that will be enumerated for the cigar when *Druid* employs the worst cast method. Likewise, for the outermost disk, whose crossings with the cigar are marked in the right figure, the depth range in the best case is one and in the worst case is two.

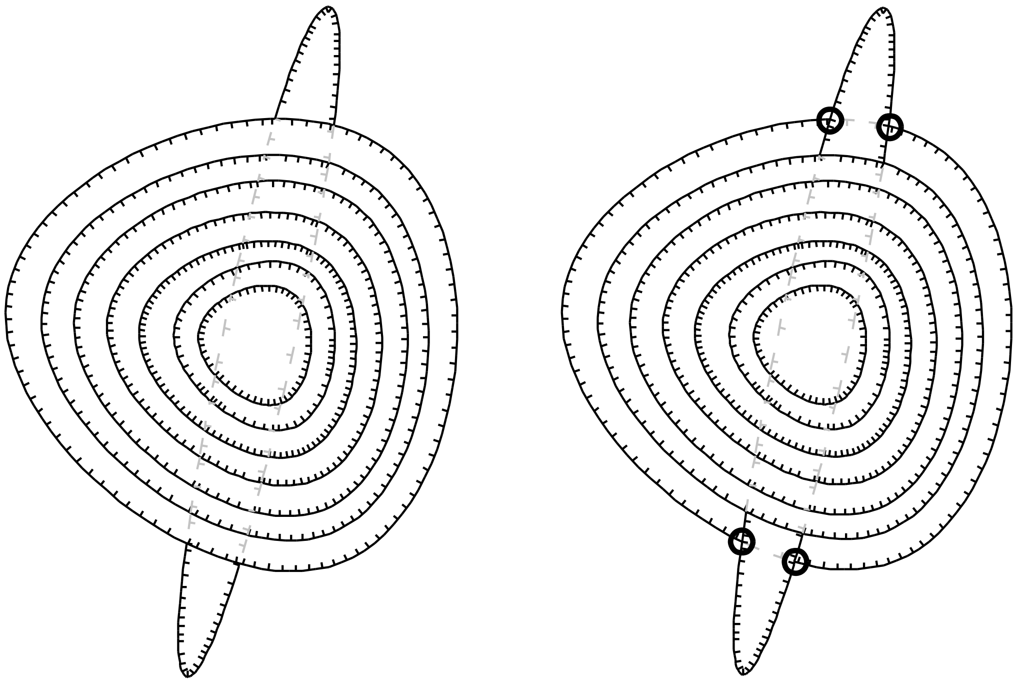

Figure 7.1: A drawing before (left) and after (right) a crossing-flip. The crossings corresponding to the flip are marked with circles in the right figure.

Figs. 7.2 and 7.3 show plots of the running time and number of nodes visited for the worst case (left) and best case (right) scenarios. We observe that the mean running time for the best case scenario is five times faster than the mean running time for the worst case scenario and the number of nodes visited for the best case scenario is seven times smaller than for the worst case scenarios. These differences represent the difference in performance between the best and worst cases.

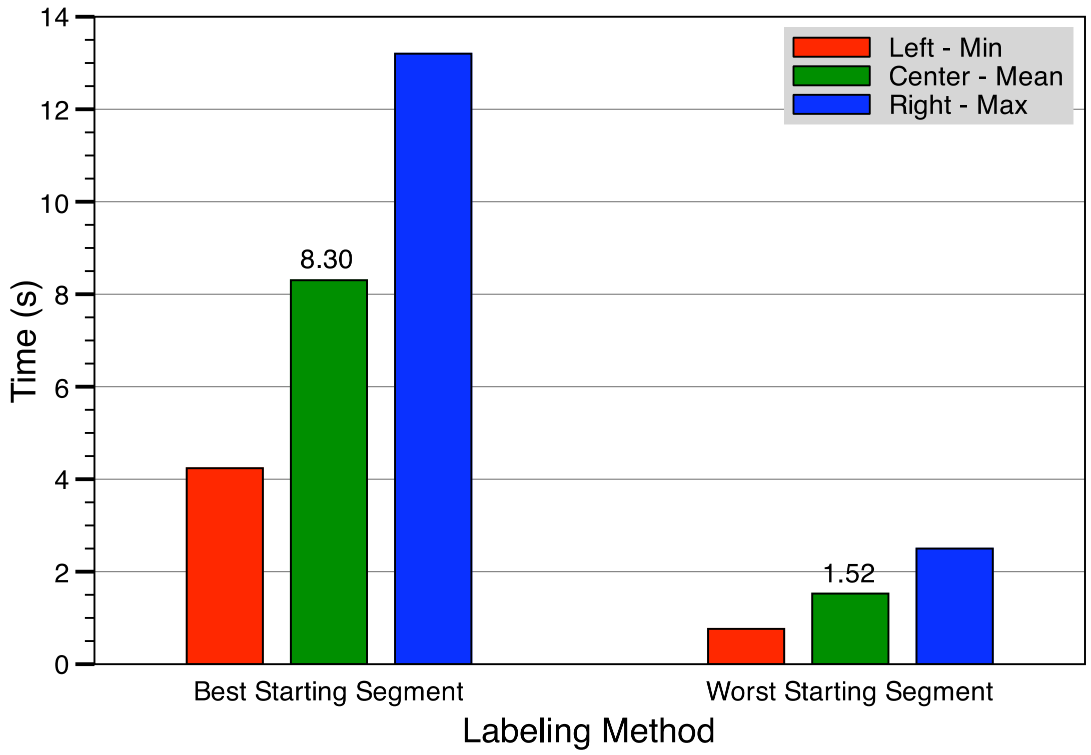

Figure 7.2: Running times for the best starting segment vs. worst starting segment applied to the flip shown in Fig. 7.1. We observe that best case scenario provides an improvement over the worst case scenario by a factor of five. Precise mean running times are given above the bar representing mean running times.

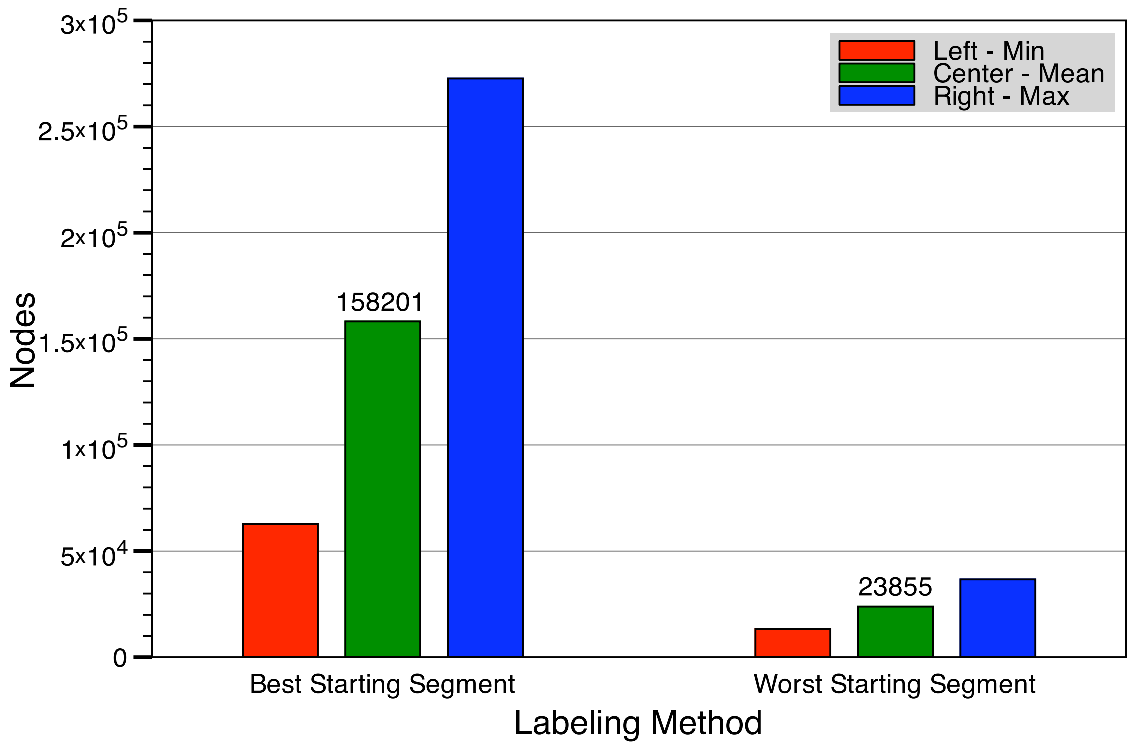

Figure 7.3: Search nodes visited for the best starting segment vs. worst starting segment applied to the flip shown in Fig. 7.1. We observe that the best case scenario provides an improvement over the worst case scenario by a factor of seven. Precise mean search nodes visited are given above the bar representing mean search nodes visited.

7.6.2 Ordering the Search

Bounding the search works best if we can find a good solution, *e.g.*, a solution with a tight bound, early in the search. By guessing that certain labelings have a better chance of being the optimum, we can order the search so that those labelings are explored first.

As discussed in Section 7.5.2, we explore the *area of interest* surrounding the user's induced change first. This ordering of the search is accomplished by projecting the knot-diagram onto a planar graph and calculating the graph distance from every crossing in the drawing to any crossing within the area of interest. We then perform branch-and-bound search within an iterative deepening loop such that only crossings within the iterative deepening horizon are considered for expansion.

Fig. 7.4 shows the drawing we used to test the effectiveness of iterative deepening. In some cases, different crossings will correspond to *topologically identical* features of the drawing, *i.e.*, different crossings can correspond to features that occur in multiple places in a single drawing. In Fig. 7.4, the crossings are topologically identical since the figure is highly symmetrical. Thus, the tests were performed on all crossings and the resulting data was merged. The set of crossings corresponding to one crossing-flip has been marked with circles.

Figs. 7.5 and 7.6 show plots of the running time and number of nodes visited with iterative deepening disabled (left) and enabled (right). We observe that the mean running time with iterative deepening is 170 times faster than without it and that the number of search nodes visited with iterative deepening is thirty times smaller than without it. Note that without iterative deepening, *Druid* failed to find a solution within the 120 second timeout in six percent of the tests.

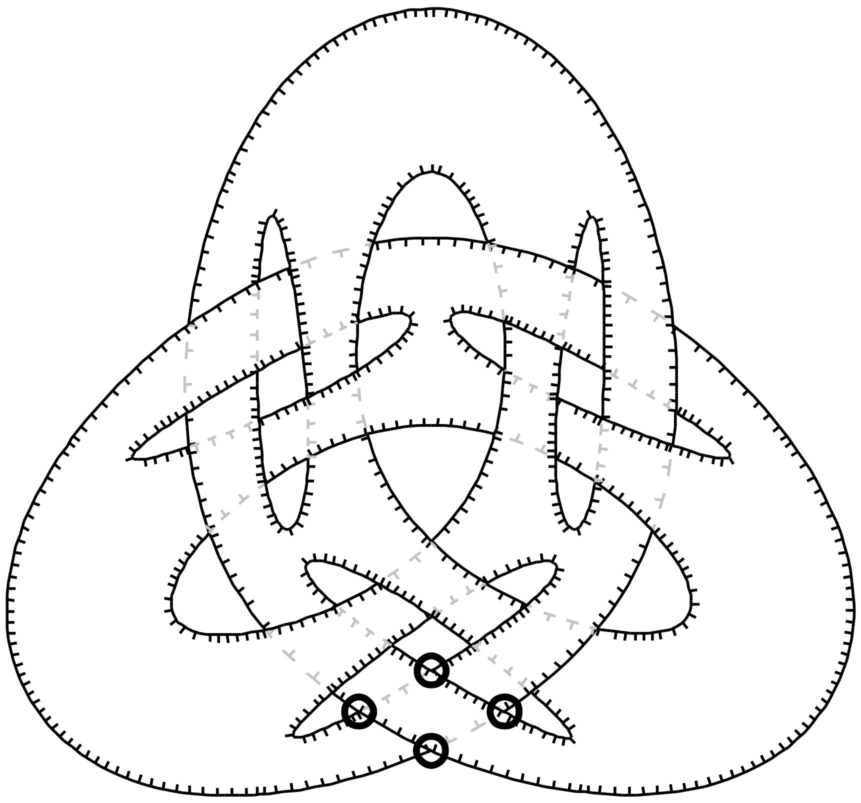

Figure 7.4: The drawing we used to produce the plots shown in Figs. 7.5 and 7.6. Since all crossing-flips are *topologically identical* (see text), the tests were performed on all crossings and then merged in the plot. The four crossings correspond to one crossing-flip are marked with circles.

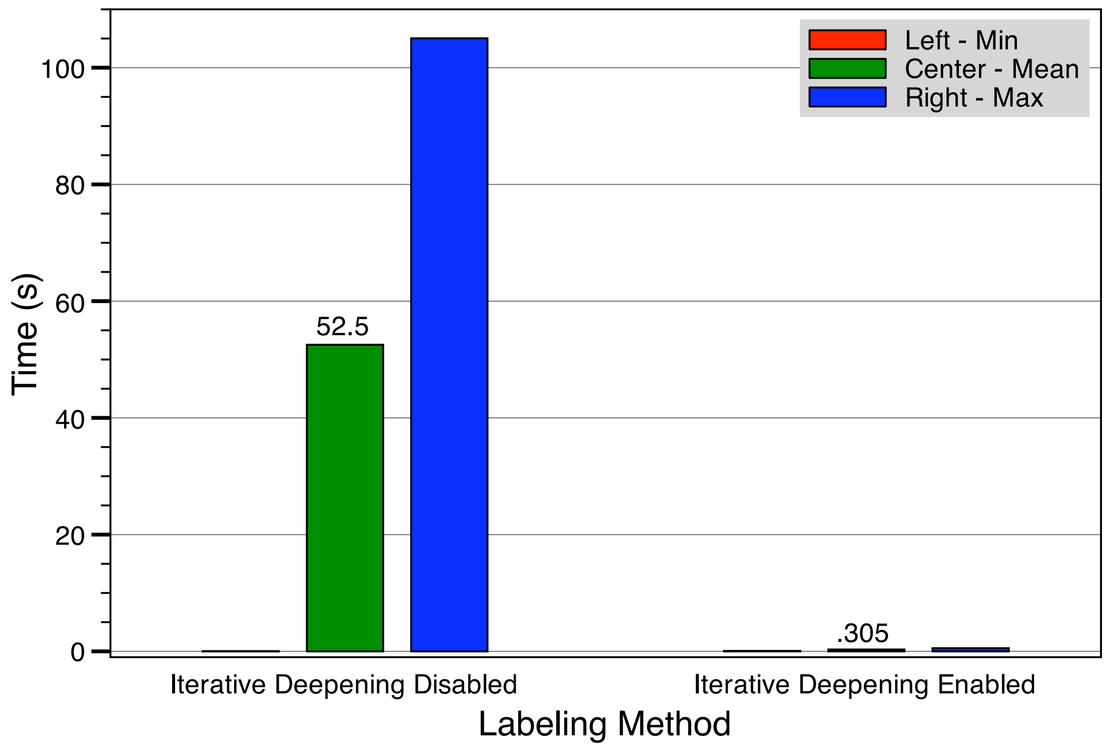

Figure 7.5: Running times for iterative deepening disabled and enabled applied to the flips shown in Fig. 7.4. We observe that the use of iterative deepening improves performance by a factor of 170. Precise mean running times are given above the bar representing mean running times.

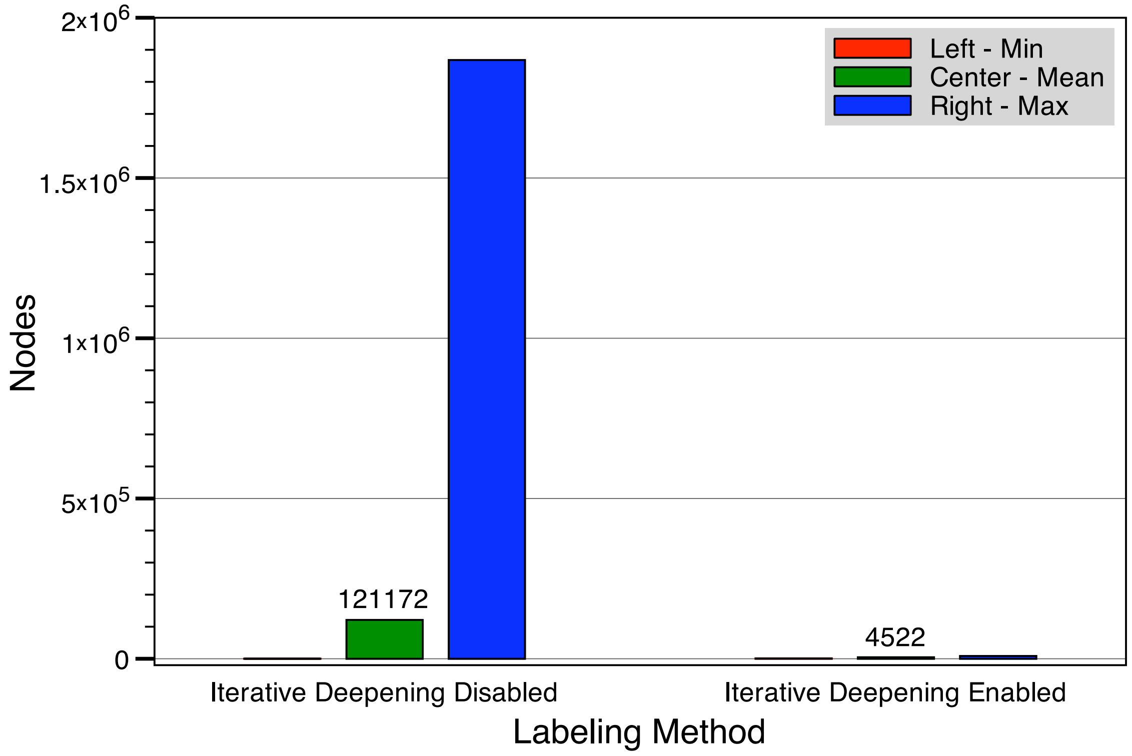

Figure 7.6: Search nodes visited for iterative deepening disabled and enabled applied to the flips shown in Fig. 7.4. We observe that the use of iterative deepening improves performance by a factor of thirty. Precise mean search nodes visited are given above the bar representing mean search nodes visited.

7.7 Search Performance

Section 7.6 demonstrated the benefits that can be achieved using each heuristic. This section presents experimental results that measure how the performance of the overall search process scales with drawing complexity. It has been shown that similar scene labeling problems, such as scenes of trihedral blocks considered by Huffman, are NP-complete (see Huffman [20] and Kirousis and Papadimitriou [22]). However, scene labeling is one of the few NP-complete problems for which a practical and fast algorithm (for typical scenes) has been found. We refer to Waltz's method of constraint-propagation (see Waltz [43]). Although Waltz filtering requires exponential time in the worst case, it performs much better in the average case. Likewise, *Druid* successfully labels its representation quickly using a form of Waltz filtering. In the following experiments we attempt to characterize *Druid's* labeling performance as a function of drawing complexity. We can measure drawing complexity in two ways:

1. Total number of crossings
2. Maximum region depth.

The first measure of complexity, total number of crossings, is a measure of the complexity of the underlying knot-diagram. The second measure of complexity, maximum region depth, is analogous to *depth complexity* (see McReynolds and Blythe [27]), which is a common metric for analyzing the complexity of rasterizing a three-dimensional polygonal scenes during three-dimensional rendering.

We conducted two kinds of labeling search experiments:

1. Randomized labeling
2. Incremental labeling.

Randomized labeling attempts to find a legal labeling for a particular drawing by starting from an initialized drawing, *i.e.*, one in which all crossing-states have been randomized and all boundary segment depths have been set to zero. Incremental labeling is more representative of the way in which *Druid* is actually used. As a drawing is altered over time, incremental labeling preserves the current labeling and uses the current labeling as the starting location in the search for a new minimum-difference legal labeling.

We conducted both experiments on two different sets of drawings. Each set of drawings is carefully designed such that its complexity can be incrementally and methodically increased by adding one more surface to the drawing in a predetermined fashion. This procedure gives us a progression of drawing complexities against which we can observe the performance of the labeling search. Whenever a new surface is added to the drawing, the drawing is labeled using both of the methods listed above, *i.e.*, it is randomly labeled by randomizing the labeling immediately after adding the surface, and it is incrementally labeled using the current labeling as the starting location in the search space.

7.7.1 Labeling Search Experiment Test 1

Fig. 7.7 (*a*) shows a drawing of a single cigar-shaped surface. To obtain a progression of drawings, we iteratively added new "cigars", one by one (Fig. 7.7, *b*, *c*). We did this nineteen times, which resulted in twenty different drawings (including the first drawing) (Fig. 7.7, *d*).

At any step during the iterative addition of new surfaces, the drawing's complexity can be described using the two measures of complexity listed above, total number of crossings and maximum region depth. For Test 1, the number of crossings is linear in the number of surfaces since two crossings are added for each additional surface. Likewise, the maximum region depth is constant since no more than two surfaces ever cover a region.

Labeling Search Experiment Test 1 Results

Figs. 7.8, 7.9, and 7.10 show plots of the labeling search performance relative to the first measure of complexity, the total number of crossings. We observe, based on the plots shown in Figs. 7.8 and 7.9, that the randomized method produces a performance which scales exponentially in the number of crossings. In contrast, based on the plot shown in Fig. 7.10, we observe that the performance of the incremental method scales linearly in the number of crossings. Note that these results do not describe *Druid's* behavior on all drawings. Rather, they serve to illustrate *Druid's* performance on drawings with characteristics similar to the drawings used in Test 1, *i.e.*, in which the number of crossings is linear in the number of surfaces. In the next experiment, we will show that *Druid's* performance can be quite different for drawings of different degrees of complexity.

We described two methods for measuring the complexity of a drawing, number of crossings and maximum region depth. However, since the maximum region depth is constant for Test 1, we have not provided experimental results for that measure of complexity. We do, however, provide such results in Test 2.

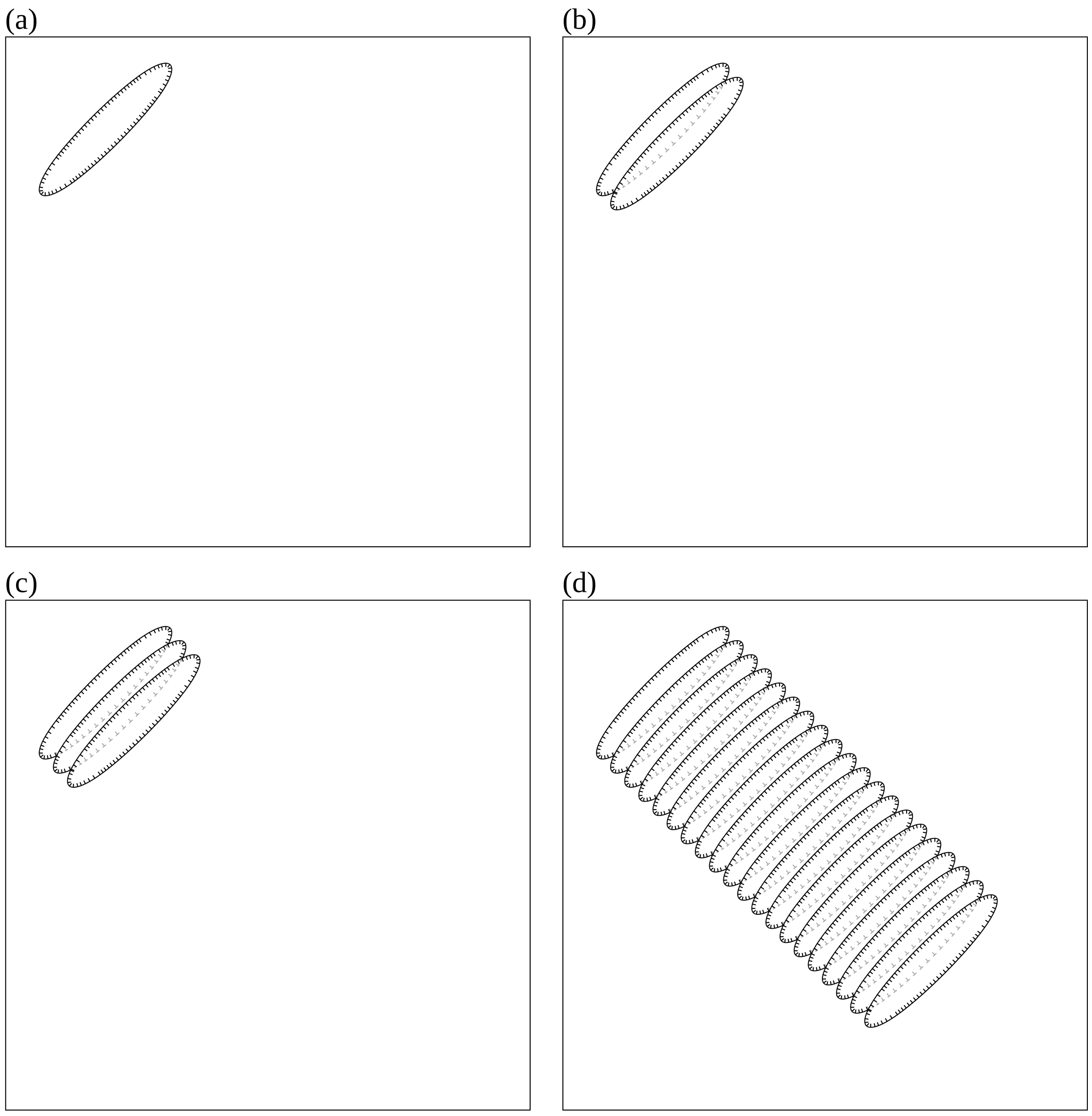

Figure 7.7: For the first labeling search experiment, we constructed a drawing of a single cigar-shaped surface (*a*). This drawing can be iteratively modified by adding identical cigars to the drawing, thus smoothly increasing the drawing's complexity (*b*, *c*). We performed this process nineteen times, resulting in twenty total drawings (including the first drawing) (*d*).

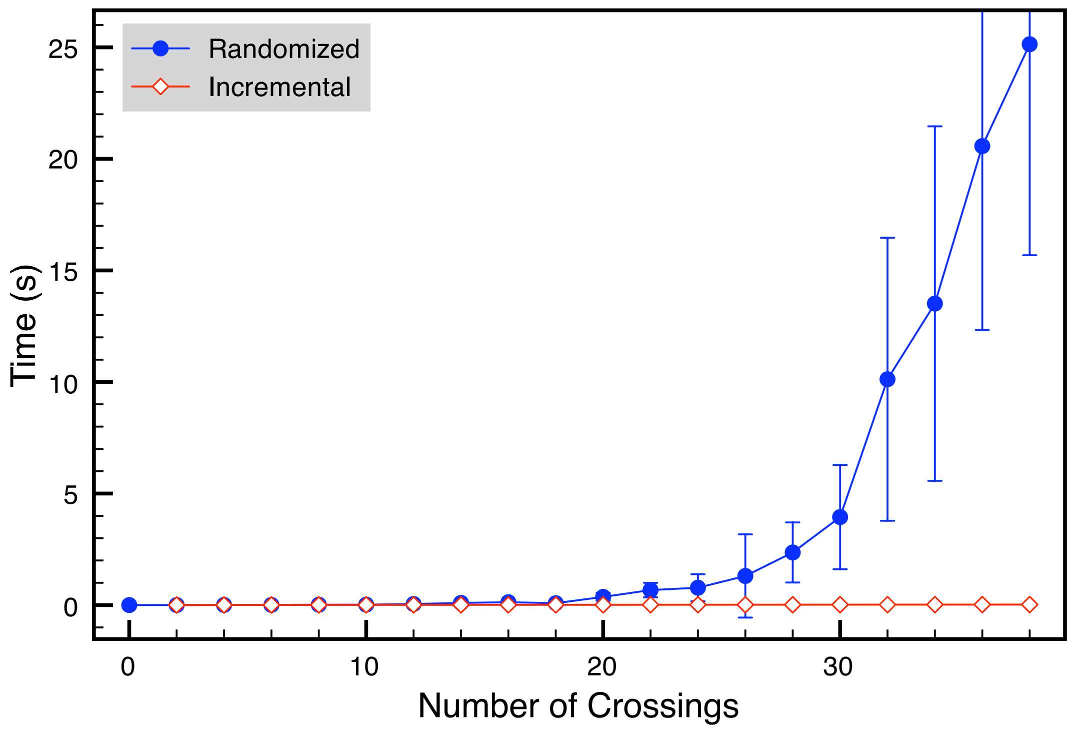

Figure 7.8: Labeling search time vs. number of crossings. We observe that the randomized method performs worse than linear in the number of crossings. Results for the incremental method are difficult to interpret on this plot. Figs. 7.9 and 7.10 more precisely characterize the performance of each labeling search method. Errors bars show 95% confidence interval.

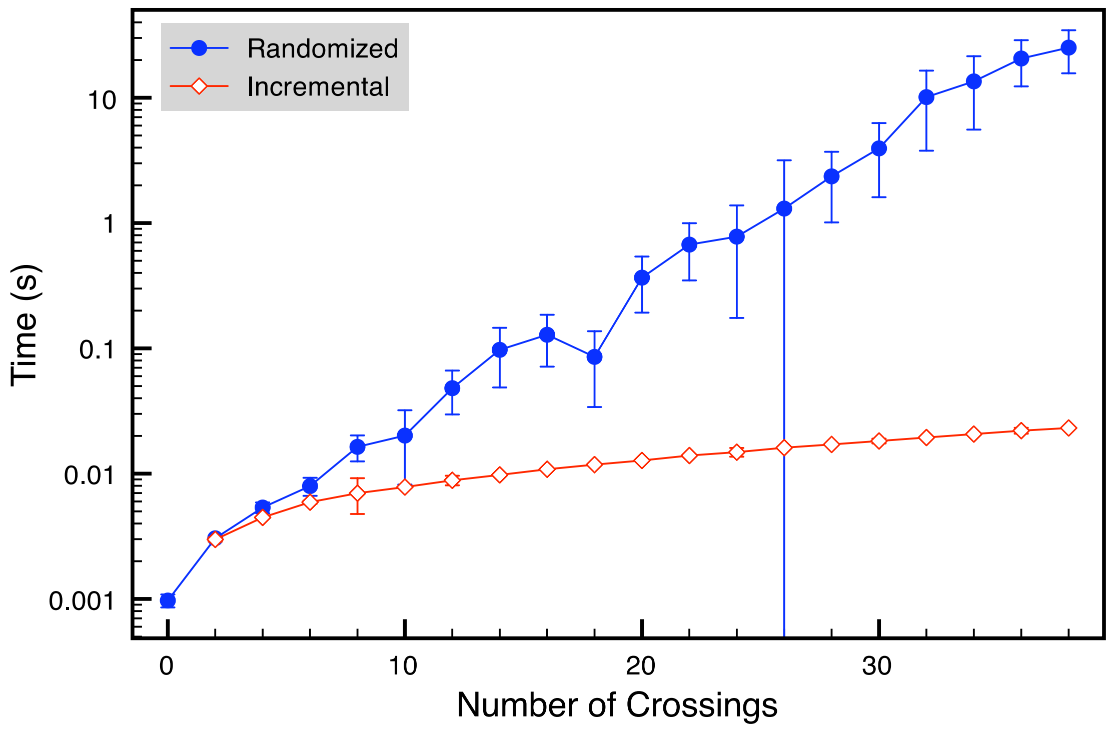

Figure 7.9: Labeling search time vs. number of crossings with a logarithmic y-axis. We observe that the randomized method performs exponentially in the number of crossings.

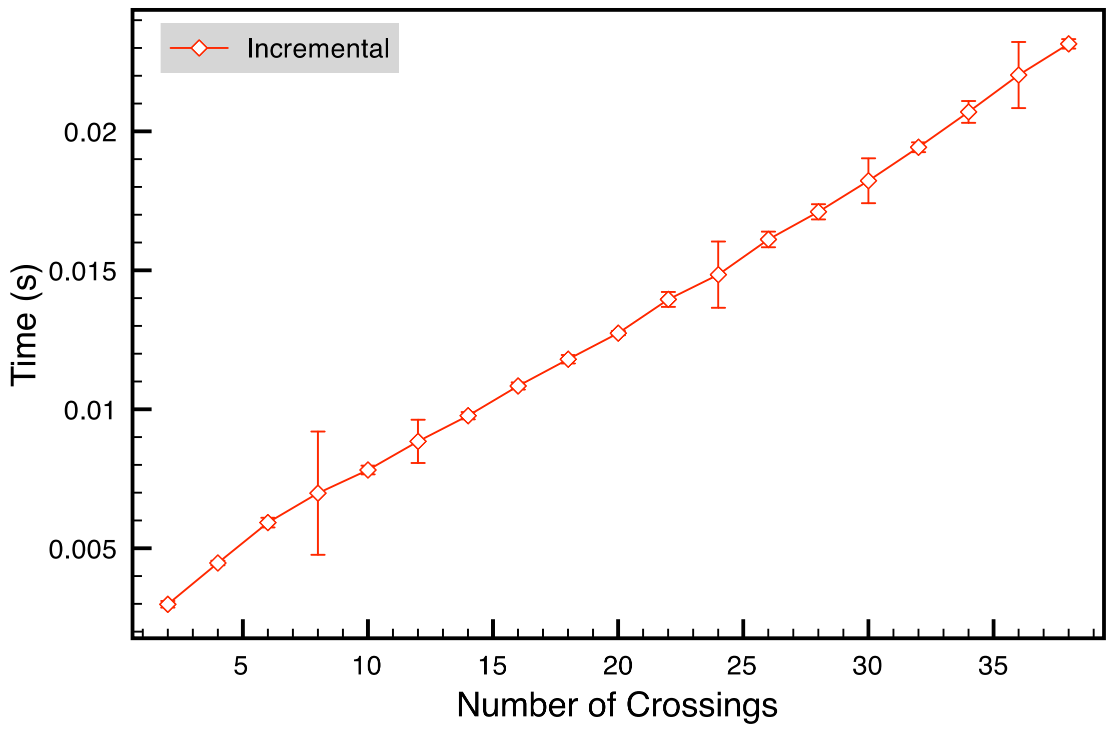

Figure 7.10: Labeling search time vs. number of crossings, incremental method. We observe that the incremental method performs linear in the number of crossings.

7.7.2 Labeling Search Experiment Test 2

In Test 2, we performed the exact same sequence of modifications to a drawing, *i.e.*, we incrementally added a surface with a specially designed shape to the drawing in nineteen steps. At each step, we performed the same two experiments that were performed in Test 1, *i.e.*, randomized and incremental labeling. However, in Test 2, the shape that is used produces a set of drawings that has different characteristics than those of the drawings used in Test 1. Fig. 7.11 shows the drawings that were used for Test 2. In Test 1, the addition of a new shape always produced two new crossings. In Test 2, the addition of a new shape produces two *more* crossings than were produced in the previous step, *i.e.*, the number of crossings that is added at each step increases from step to step. Since the derivative of the number of crossings relative to the number of surfaces is linear for Test 2, the number of crossings is quadratic in the number of surfaces. Likewise, in Test 1 the maximum region depth was constant, whereas in Test 2 the central region is covered by every surface in the drawing. Thus, the maximum region depth is linear in the number of surfaces for Test 2.

Labeling Search Experiment Test 2 Results

Figs. 7.12, 7.13, and 7.14 show plots of the labeling search performance relative to the number of crossings for Test 2. Note that only the first six data points of the randomized method are relevant. Subsequent data points represent labeling searches which timed out at 120 seconds without finding a solution. Consequently, the data beyond that point is meaningless. We observe based on the plots shown in Figs. 7.12 and 7.13 that the performance of the randomized method scales exponentially in the number of crossings. We observe based on the plots shown in Fig. 7.13 (in which the incremental method performs better than exponential) and Fig. 7.14 (in which the incremental method performs worse than linear) indicate that the incremental method produces a performance which scales polynomially in the number of crossings. Note that this result is different from Test 1, in which the performance of the incremental method performed linearly in the number of crossings. We conclude that for drawings of sufficient complexity, the labeling search performance can scale in a polynomial fashion, as shown.

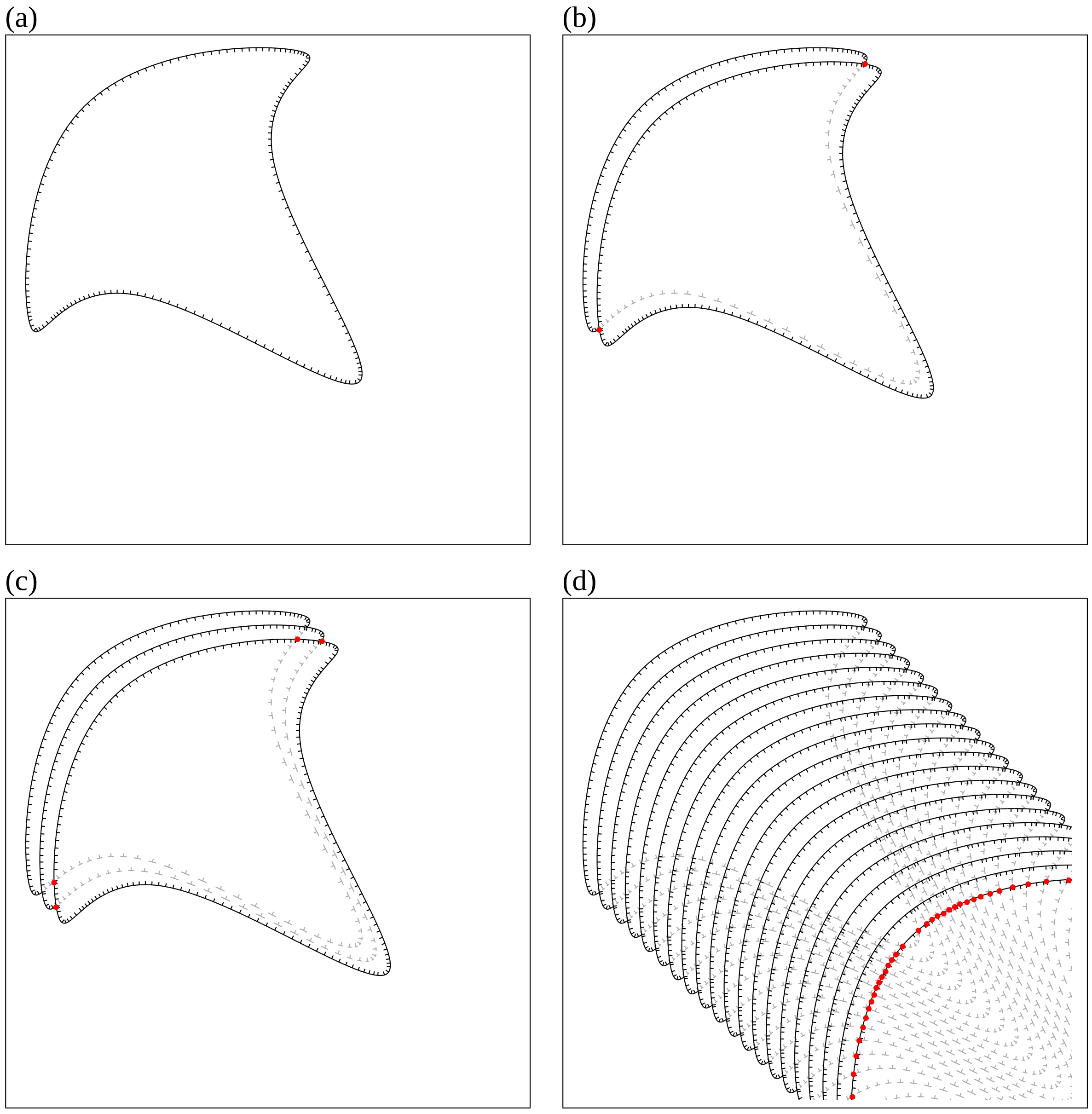

Figure 7.11: For the second labeling search experiment, we constructed the shape shown in *a*. Identical copies of this shape were incrementally added to the drawing (*b*, *c*). We performed this process nineteen times, resulting in twenty total drawings (including the first drawing) (*d*). In each figure above, the crossings that were created as a result of the addition of the most recent surface are marked with solid circles. Note that the number of crossings that are added for each surface is linear in the number of surfaces currently in the drawing.

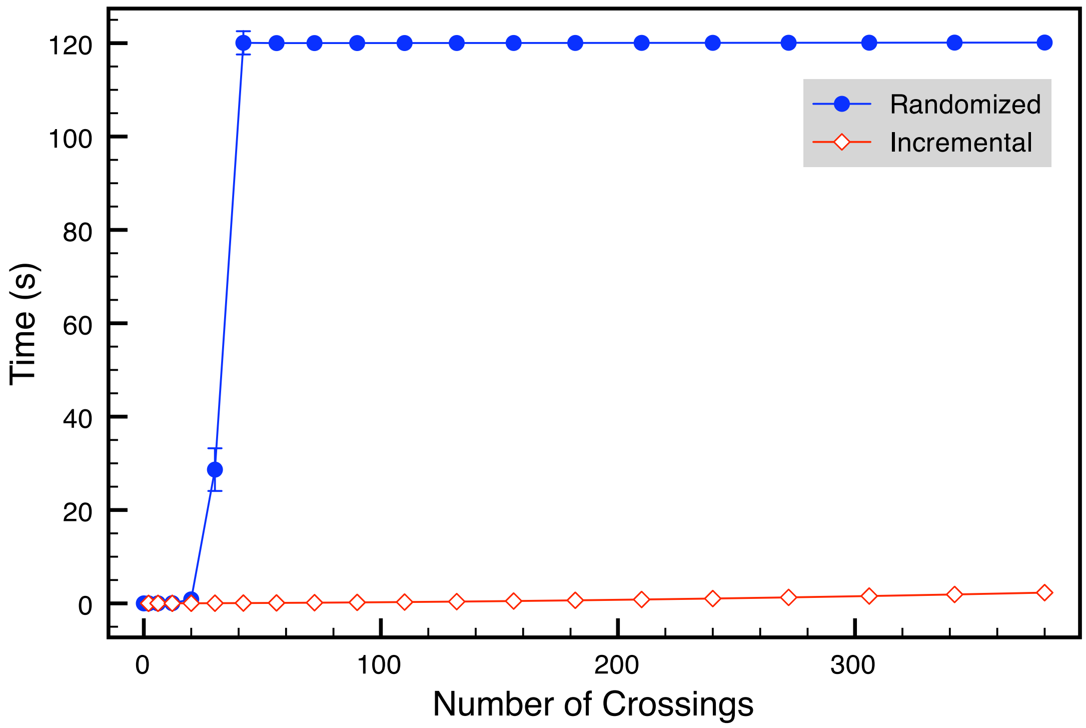

Figure 7.12: Labeling search time vs. number of crossings. Only the first six data points of the randomized method are relevant. Subsequent data points represent labeling searches which timed out at 120 seconds without finding a solution. Consequently, the data beyond that point is meaningless. We observe that the randomized method performs worse than linear in the number of crossings. Results for the incremental method are difficult to interpret on this plot. Figs. 7.13 and 7.14 more precisely characterize the performance of each labeling search method. Errors bars show 95% confidence interval.

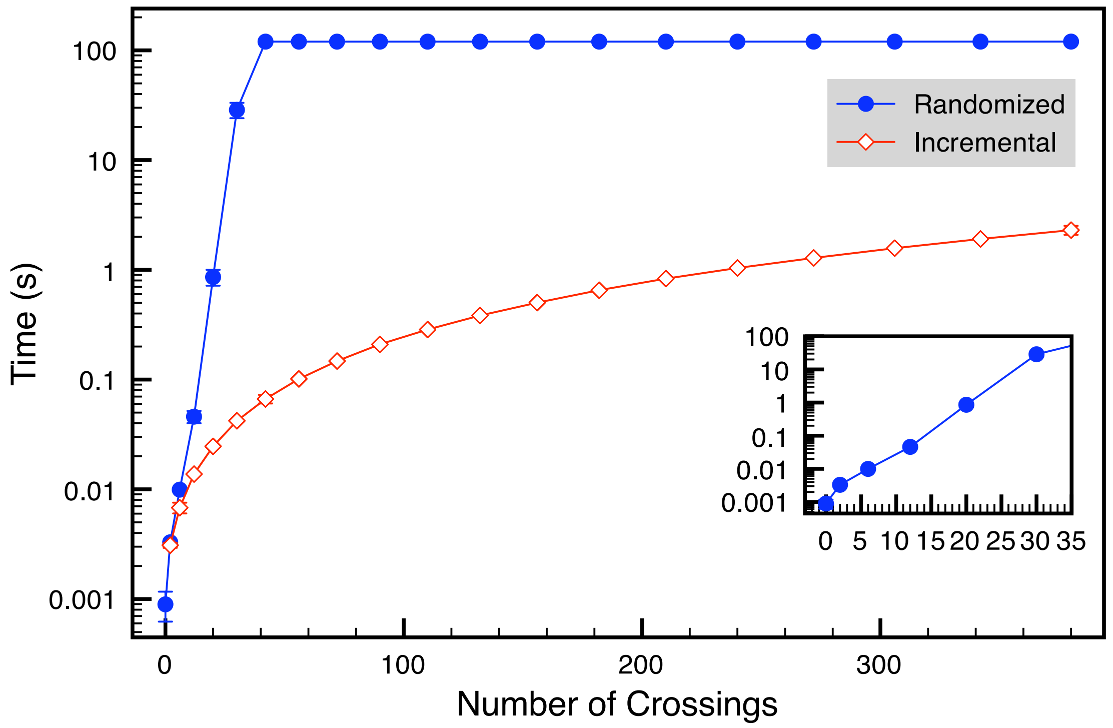

Figure 7.13: Labeling search time vs. number of crossings with a logarithmic y-axis. Only the first six data points of the randomized method are relevant. Subsequent data points represent labeling searches which timed out at 120 seconds without finding a solution. Consequently, the data beyond that point is meaningless. The inset plot shows only the first six data points of the randomized method, which more clearly illustrates that the plot of the randomized method is linear on a log plot. Thus, we conclude that the randomized method performs exponentially in the number of crossings and that the incremental method performs better than exponential in the number of crossings.

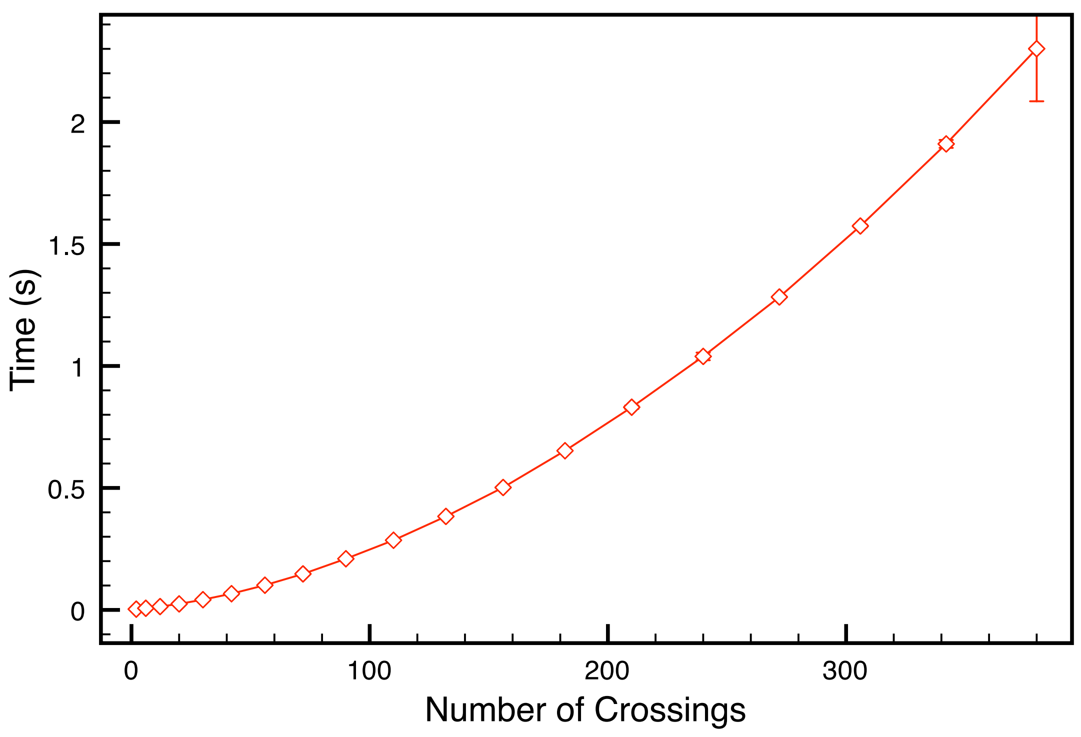

Figure 7.14: Labeling search time vs. number of crossings, incremental method. We observe that the incremental method performs worse than linear in the number of crossings. Since the incremental method performs better than exponential in the number of crossings (see Fig. 7.13), we observe that the incremental method performs polynomially in the number of crossings.

Figs. 7.15, 7.16, and 7.17 show plots of the labeling search performance relative to the maximum region depth for Test 2. We observe based on the plots shown in Figs. 7.15 and 7.16 that the performance of the randomized method scales exponentially in the maximum region depth. We observe based on the plots shown in Fig. 7.16 (in which the incremental method performs better than exponential) and Fig. 7.17 (in which the incremental method performs worse than linear) that the performance of the incremental method scales polynomially in the maximum region depth.

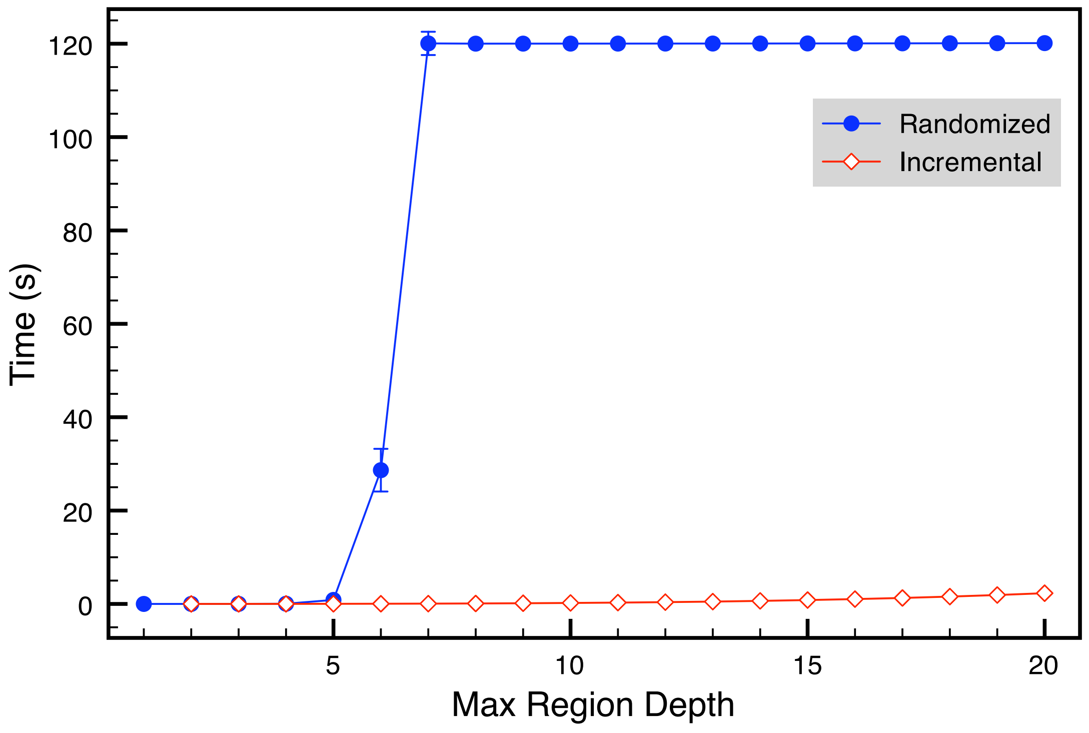

Figure 7.15: Labeling search time vs. maximum region depth. Only the first six data points of the randomized method are relevant. Subsequent data points represent labeling searches which timed out at 120 seconds without finding a solution. Consequently, the data beyond that point is meaningless. We observe that the randomized method performs worse than linear in the maximum region depth. The incremental method is difficult to analyze on this plot. Figs. 7.16 and 7.17 more precisely characterize the performance of each labeling search method. Errors bars show 95% confidence interval.

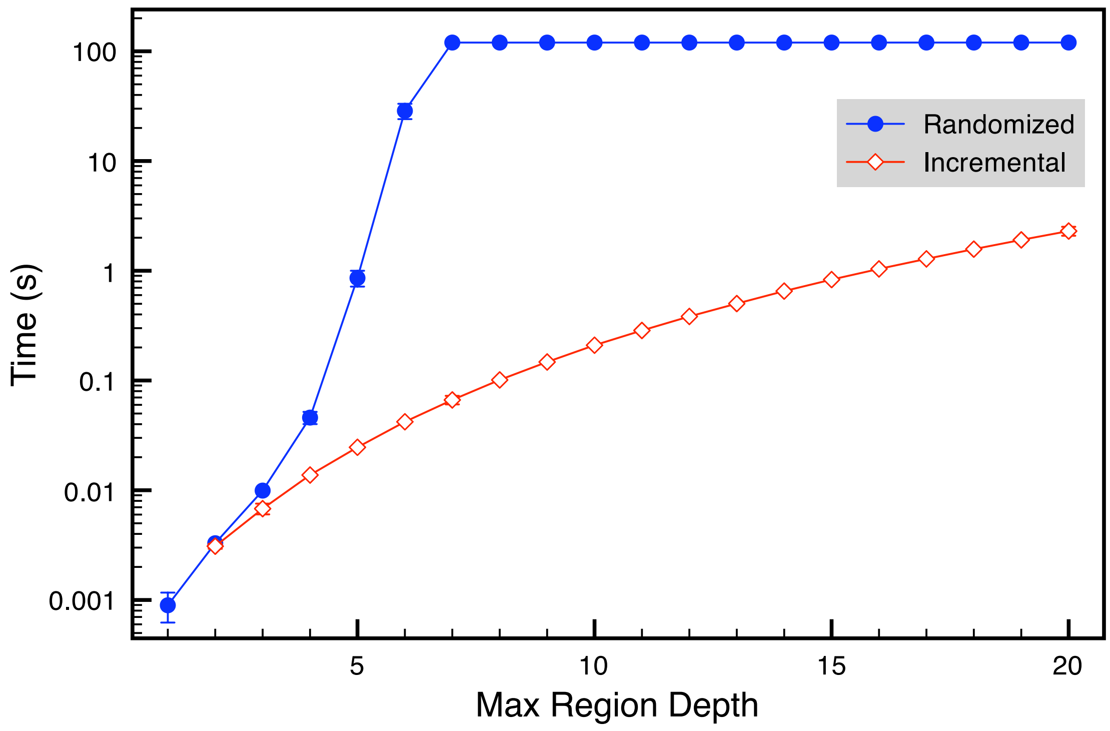

Figure 7.16: Labeling search time vs. maximum region depth with a logarithmic y-axis. Only the first six data points of the randomized method are relevant. Subsequent data points represent labeling searches which timed out at 120 seconds without finding a solution. Consequently, the data beyond that point is meaningless. The inset plot shows only the first six data points of the randomized method, which more clearly illustrates that the plot of the randomized method is linear on a log plot. Thus, we observe that the randomized method performs exponentially in the maximum region depth. We observe that the incremental method performs better than exponential in the maximum region depth.

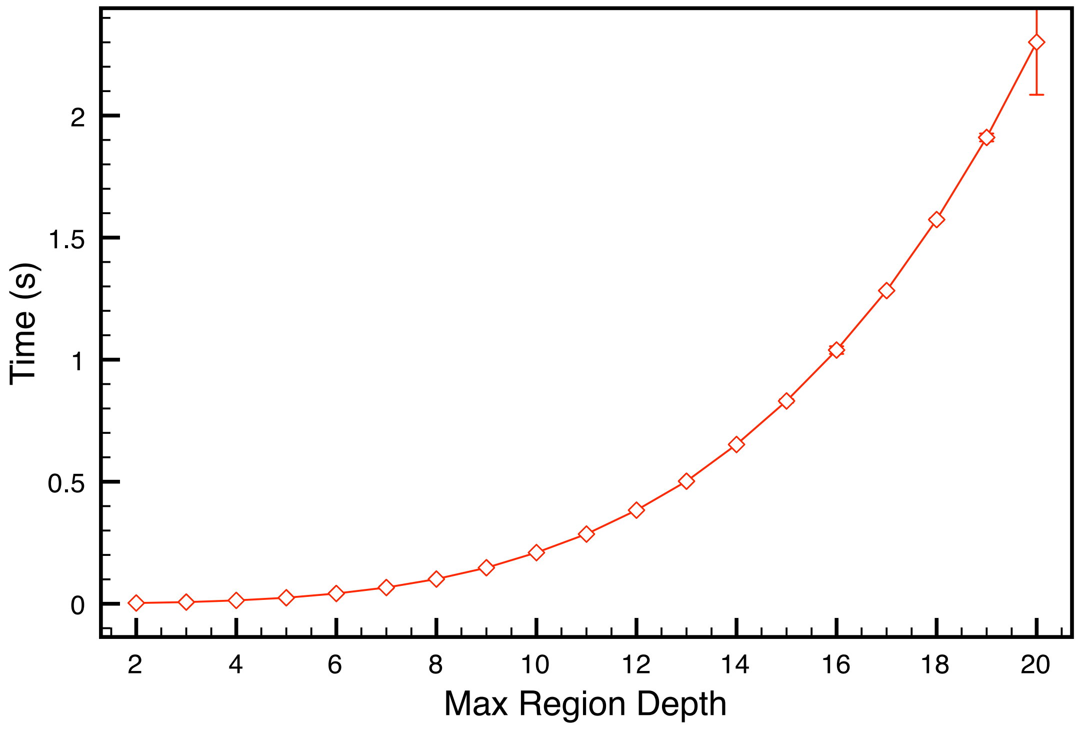

Figure 7.17: Labeling search time vs. maximum region depth, incremental method. We observe that the incremental method performs worse than linear in the maximum region depth. Since the incremental method performs better than exponential in the maximum region depth (see Fig. 7.13), we observe that the performance of the incremental method scales polynomially in the maximum region depth.

Chapter 8

Boundary Grouping With Cuts

To this point in the dissertation, we have described the central aspects of *Druid's* functionality. In particular, we have described its representation, *i.e.*, labeled knot-diagrams, the search space that *Druid* automatically searches through to find legal labelings when the user causes topological changes, and the search process that *Druid* uses to search through the this space. The next two chapters are somewhat orthogonal to the main thesis are still important. This chapter describes *cuts*, which *Druid* uses to group boundaries that bound a single surface. The next chapter discusses *rendering*, the process *Druid* uses to translate a labeled knot-diagram into an image with solid color fills for contiguous regions of the canvas.

The fact that a surface can be bounded by multiple boundaries can cause two kinds of problems for *Druid*. First, *Druid* must know which boundaries to translate as a unit when the user drags a surface across the canvas. Second, in some drawings there is a potential ambiguity as to which boundaries bound which surfaces. Consequently, it is possible that *Druid* will not assign a labeling to a knot-diagram that conforms to the user's intent. Both of these problems arise from the fact that *Druid* is not like other drawing programs in that the basic units *Druid* operates on are not contours or regions but surfaces. Therefore, *Druid* must use a *surface structure* at a higher level of abstraction than boundaries. In this chapter we describe how *Druid* automatically groups boundaries into surfaces by finding *cuts*, *i.e.*, paths joining two locations on two boundaries in a way that is analogous to a scissor cut through a surface. We briefly discuss the concept of manual cuts, *i.e.*, cuts that the user explicitly designates, and then describe how *Druid* automatically improves the *cleanness* of the cuts it has previously found for a drawing without degrading performance and negatively impacting the user's experience.

8.1 Two Kinds of Object Groups: Boundary Groups and Surface Groups

One feature that is common to most drawing programs is the ability to group objects together. Groups are usually used to perform transformations like translation, scaling, and rotation on a set of objects. Users who are familiar with other drawing programs might expect that the most obvious objects to group in *Druid* are individual boundaries since these are closely analogous to objects in other drawing programs. However, *Druid* is unlike conventional drawing programs in that the basic units that are operated on are *surfaces* and boundaries are used to represent surfaces.

Druid handles these two fundamental objects, boundaries and surfaces, differently with respect to grouping. With regard to surfaces, *Druid* only provides the notion of a temporary *selection* for the purpose of performing transformations on a group. It does not provide a more permanent form of surface grouping. Selection is a feature common to many kinds of programs, *e.g.*, drawing tools, text-editors, spreadsheets, and games. It is transient in nature and only exists while the user performs an action on the selection. In general, multiple simultaneous selections are not allowed, and selections are not stored as part of any persistent representation. Selections of surfaces are the only form of explicit user initiated grouping that *Druid* currently provides.

Druid automatically groups multiple boundaries into a *surface structure* without any intervention from the user. Boundary groups are not required for *Druid* to legally label a drawing. However, as in other drawing programs, *Druid* can use groups as a basis for translation of a surface with multiple boundaries. More importantly, *Druid* can use boundary groups to eliminate ambiguities about which surfaces boundaries bound. For example, in Fig. 8.1 (left), it is ambiguous whether boundary *B* bounds a surface below boundary *A*, above boundary *A*, or the same surface as boundary *A*. This ambiguity can potentially cause problems. If the user were to create a third boundary that overlaps the ambiguous surface of *A* and *B*, *Druid* might arbitrarily place the new surface *between* boundaries *A* and *B*. Clearly, if the user's intent is for boundaries *A* and *B* to bound the same surface, then such a placement violates the user's expectation about the effects of his interactions. Grouping boundaries can minimize this type of problem.

8.2 Boundary Grouping with Cuts

Druid automatically finds and maintains boundary groups without any input from the user. It does this by finding and maintaining *cuts*. A cut is analogous to a scissor cut through a surface joining two boundaries that bound a single surface. A cut converts two boundaries of a surface into a single contiguous boundary, as shown in Fig. 8.1 (center and right). After a cut is discovered it is used to merge the two boundaries' surfaces, *i.e.*, a boundary group. If a cut can be found between the two boundaries, as shown in Fig. 8.1, then the two boundaries demonstrably bound the same surface and can be grouped for later operations that require this particular ambiguity to be resolved.

Observe that the discovery of a cut between two boundaries effectively joins the two boundaries into a single closed boundary. Cuts effectively reduce the number of bound-

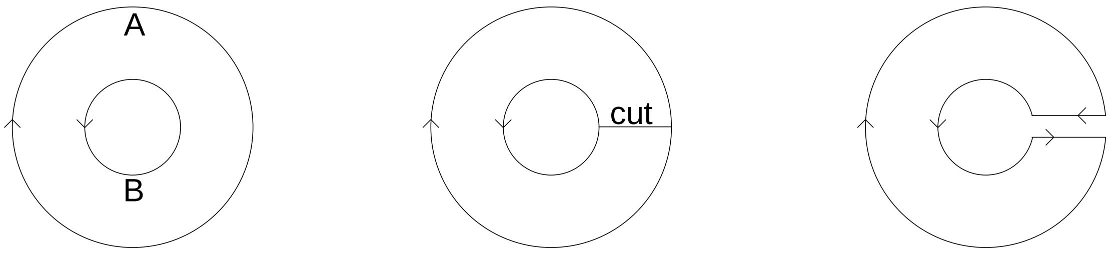

Figure 8.1: A *cut* is analogous to a scissor cut through a surface joining two boundaries of that surface. Cuts are used to group boundaries together for group transformations like translation, scaling, and rotation. Cuts also reduce ambiguities regarding the correct labeling for the knot-diagram.

aries in a drawing by one per cut. Consequently, they reduce the overall complexity of a drawing, thereby reducing the size of the search space and making the search significantly faster.

The inclusion of cuts in labeled knot-diagrams requires a more sophisticated labeling scheme which includes four new crossing types. In addition to the original boundary-to-boundary crossing type shown in Fig. 4.2, four new crossing types corresponding to all possible crossings involving cuts must also be satisfied in legal labelings (Fig. 8.2). The four new crossings types are:

- A boundary crosses a cut with the boundary above
- A boundary crosses a cut with the boundary below
- A cut crosses a cut
- A cut *ends* and attaches to a boundary, *i.e.*, a *T-junction*.

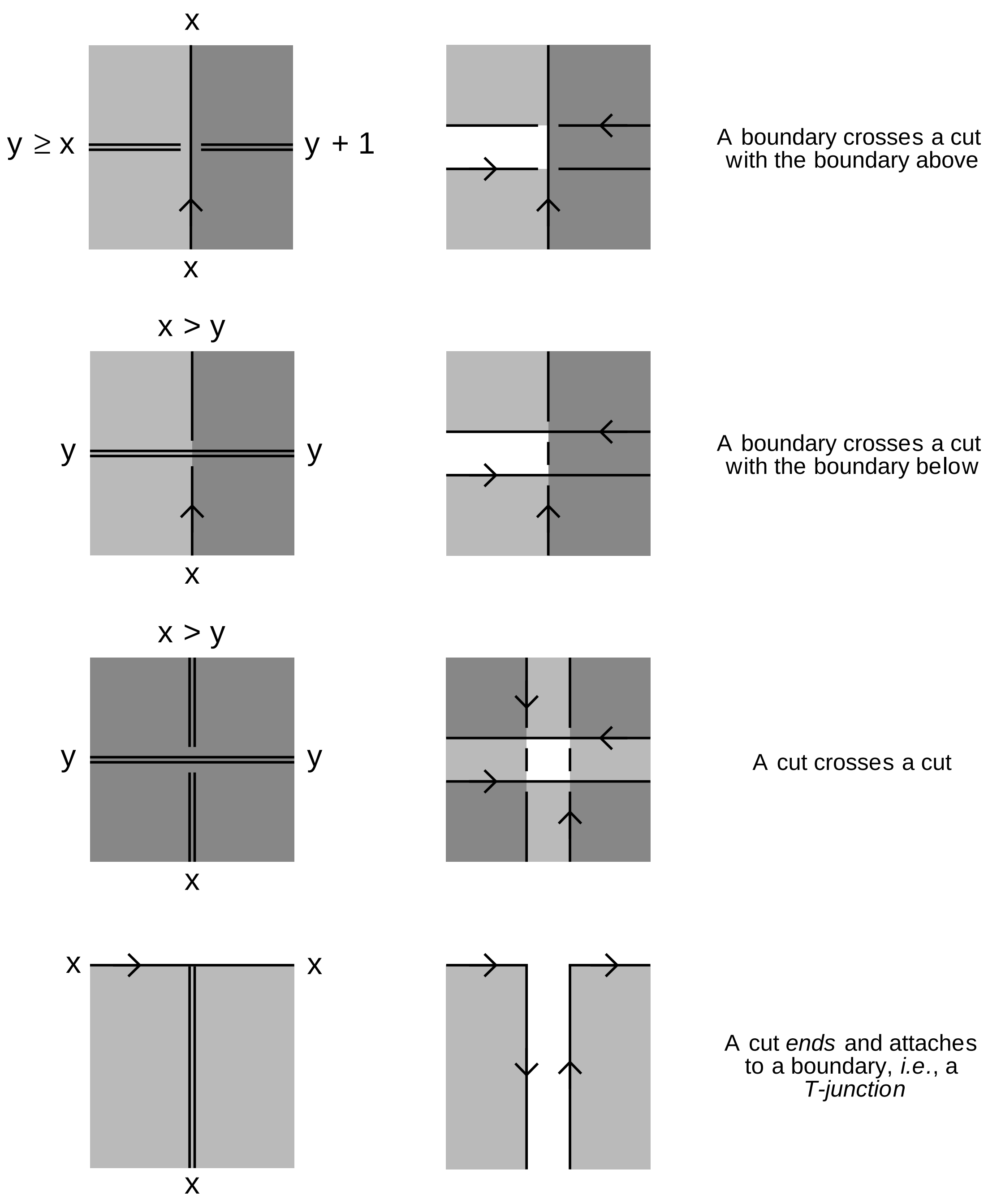
x
y ≥ x
y + 1
x
A boundary crosses a cut with the boundary above
x > y
y
y
x
A boundary crosses a cut with the boundary below
x > y
y
y
x
A cut crosses a cut
x
x
x
A cut *ends* and attaches to a boundary, *i.e.*, a *T-junction*

Figure 8.2: The inclusion of cuts in labeled knot-diagrams requires additional crossing types. In addition to the boundary-to-boundary crossing type shown in Fig. 4.2, there are four new crossing types involving cuts which can be present in a labeled knot-diagram. Each crossing type and its constraints are illustrated with a double line denoting the cut to emphasize that a cut is an atomic entity (left) and with a gap denoting the cut for increased clarity (right).

8.3 Finding Legal Cuts

A *cut* is a straight line segment joining a location on one boundary to a location on another boundary. Although a cut can theoretically join two locations on the same boundary (see Sections 10.2.2 and 14.1) we confine this discussion to cuts joining pairs of boundaries. A *legal* cut represents a physically plausible scissor-cut through a surface. Consequently, a cut can only be legal if the two boundaries it joins bound the same surface. Cuts do not have to be straight lines, but *Druid* restricts itself to straight line cuts because this simplifies *Druid's* implementation and is sufficient to achieve the goal described in Section 8.1, *i.e.*, the grouping of boundaries bounding the same surface. (for a description of a situation where curved cuts might be useful, see Section 14.1.).

A *cut-chain* is a sequence of alternating boundaries and cuts that joins two boundaries. In Fig. 8.3 (*a*) one cut-chain joins boundaries *4* and *1* through boundary *3* while a longer cut-chain connects boundaries *4* and *2* through boundaries *3* and *1*.

Boundaries can be classified according to their degree of *connectedness*:

- *Unconnected*: a boundary which is not joined by cuts or cut-chains to any other boundary bounding the same surface
- *Partially-connected*: a boundary which is joined by cuts or cut-chains to some of the other boundaries bounding the same surface
- *Fully-connected*: a boundary which is joined by cuts or cut-chains to every other boundary bounding the same surface.

The primary goal of the cut search is to ensure that all surfaces are defined by a single boundary component. This goal is accomplished by *fully-connecting* all boundaries.

It is important to note that *Druid* can never determine a boundary's connectedness. All *Druid* can do is assume that boundaries are not fully-connected and then attempt to find cuts for them. Likewise, after the cut search is completed, *Druid* assumes that all boundaries are fully-connected (although there is no way to verify their connectedness in practice). It is possible, although unlikely, that some boundaries might not be fully-connected following the cut search. We discuss this possibility later in Section 8.3.1 where we discuss the risk of false negatives during the search process.

There are two types of surfaces: *finite* and *infinite*. A finite surface has at least one user-defined exterior boundary which confines the surface to a finite area. An infinite surface has no explicit exterior boundary, and consequently has an infinite area. Despite the fact that the user has not constructed an exterior boundary in such a case, a boundary residing at an infinite distance is implicit and any user-created boundary, *e.g.*, a hole, belong to the infinite surface can be joined to it via a cut. Fig. 8.3 illustrates finite and infinite surfaces.

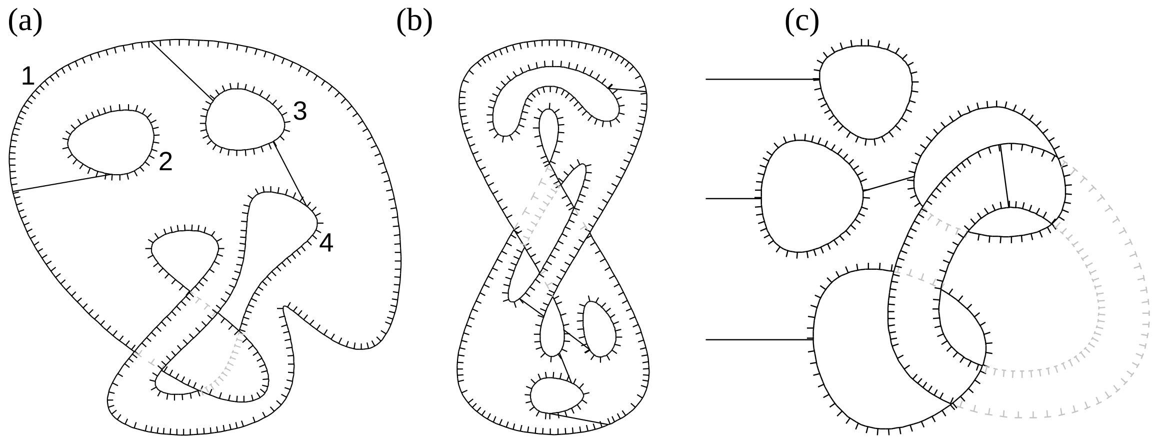

Figure 8.3: Examples of finite and infinite surfaces with cuts as indicated. (*a*) and (*b*) show finite surfaces. (*c*) shows two surfaces, one of which is infinite.

Since boundaries can be user-created or implicit, cuts can be classified into two corresponding types. A *finite cut* joins two boundaries which the user has explicitly constructed (Fig. 8.3, *a* and *b*). An *infinite cut* joins a user-created boundary to its implicit infinitely distant exterior boundary. In theory, infinite cuts might extend in any direction. However, for simplicity, *Druid* only considers horizontal infinite cuts extending to the left (Fig. 8.3, *c*). The cut search must be capable of finding both finite and infinite cuts. When a cut is found, the two associated boundaries' surfaces are merged into a single surface. The next section describes how *Druid* finds cuts.

8.3.1 The Cut Search Process

The cut search is performed for each boundary more or less independently of the cut search for other boundaries.[1] Thus, we can simplify our discussion of the cut search by describing how a cut search is performed for a single boundary. Performing the entire search is subsequently a matter of performing a similar search for all boundaries.

The following terms will aid our exposition of the processing of searching for cuts:

- *Primary boundary*: a boundary for which a cut search is performed
- *Test boundary*: a boundary which the cut search process attempts to join to the primary boundary with a cut
- *Potential cut*: a straight line joining an arbitrary location on the primary boundary to an arbitrary location on the test boundary (may be finite or infinite).

There are two steps in finding a cut. Step 1 searches a cache of previously discovered cuts (which we call *cached cuts*) in the hope of finding a cut involving the primary boundary. Reusing a cached cut is beneficial for two reasons. The first reason is that it obviates the need to search for a new cut, which is more time consuming than using a cached cut. The second reason is that the search for a new cut might suffer from *false negatives*. A false negative occurs when *Druid* fails to find a cut between two boundaries despite those boundaries being theoretically possible to join via a cut, *i.e.*, *cuttable*. When there is no legal cached cut for the primary boundary *Druid* proceeds to Step 2 which performs a randomized search for a new legal cut. We describe both steps next.

[1]The only way that cut searches can affect one another is if cut searches which occur earlier in the search process produce partially-connected boundaries which can then be used for cut-chains later in the search process.

8.3.2 Cached Cuts

Step 1 in finding a cut for a boundary is to determine whether there are any cached cuts for the boundary. Whenever *Druid* successfully finds a cut joining two boundaries during the randomized search in Step 2, the locations of the cut's attachment points on the two boundaries are cached. During Step 1, *Druid* tests any cached cuts associated with the boundary. In most cases, a cached cut will still be legal and thus will help the search terminate quickly by avoiding Step 2. Reconstructing a cut from a cached cut is much faster than randomly searching for a new cut. Additionally, a legal cached cut eliminates the risk of false negatives, which can only occur during Step 2. However, a cached cut is not guaranteed to be legal since changes to the drawing may have invalidated it. If a cached cut is no longer legal, *Druid* erases it.

It is possible for a boundary to have multiple cached cuts since *Druid* generates a cached cut for every cut currently in the drawing. Consider a boundary which is in the middle of a cut-chain. Since such a boundary has at least two cuts it consequently has at least two cached cuts. Thus, *Druid* must check every cached cut associated with the boundary before proceeding to Step 2.

If no legal cached cuts exist for the primary boundary, *Druid* proceeds to Step 2, in which random potential cuts are generated and tested. This randomized cut search process is described in the next section.

8.3.3 Finding a Cut for One Boundary

Step 2 in finding a cut is to perform a randomized search for a new legal cut. Potential cuts joining the primary boundary to various test boundaries are generated and then tested.

The cut search process is structured as three nested loops. The inner loop takes the primary boundary and a test boundary and enumerates a number of *cut attempts* for the boundary pair. A cut attempt generates a random potential cut between the two boundaries, *i.e.*, a straight line joining random locations on both of the boundaries, and then tests the potential cut's legality, a process which we describe in the next section.

The intermediate loop enumerates all boundaries, each of which is used as the test boundary in the inner loop. Thus, the intermediate and inner loops test all boundary pairs involving the primary boundary, searching for a legal cut between the primary boundary and each test boundary. If at any point a legal cut is found, the search is immediately terminated and the discovered cut is used to merge the two boundaries' surfaces.

There is no definitive way to know how many cut attempts to make between two boundaries. If too few are made, the risk of false negatives will be too high and *Druid* might not discover a legal cut for a pair of cuttable boundaries. However, if too many cut attempts are made, the search process will take a long time. In particular, *Druid* will waste a lot of time testing boundary pairs which are genuinely uncuttable, *i.e.*, boundary pairs which do not bound the same surface and consequently for which there is no legal cut to be found. In such cases the search will not end until the full number of cut attempts has been enumerated. Ideally, *Druid* would not spend much time attempting cuts between boundary pairs that are uncuttable, and yet it would find legal cuts quickly. We have designed the search process to increase of likelihood of this outcome with the use of the outer loop, which steadily increases the number of cut attempts in the inner loop.

Thus, on the first iteration of the outer loop, *Druid* makes very few cut attempts between the primary boundary and each test boundary.[2] If no legal cut is found, the outer loop repeats, and the number of cut attempts in the inner loop is increased. The entire search process continues until either a legal cut is found or the outer loop reaches its maximum cut attempt scaling factor and the search ends without a successful cut discovery.

[2]During the first iteration of the outer loop, *Druid* makes 1/32 of the number of cut attempts it will make in the last iteration of the outer loop. For a pair of boundaries defined ninety-six piecewise linear segments each, the first iteration of the out loop makes three cut attempts and the last iteration makes ninety-six cut attempts.

The following pseudo-code illustrates the structure of the search process. Lines 2, 3, and 6 correspond to the three nested loops.

```
1. FIND_LEGAL_CUT( $p$ )                               ▷ $p$: the primary boundary
2.   for $M \leftarrow 1/32$ to 1.0 step $M \leftarrow M \cdot 2$     ▷ $M$ increases exponentially
3.     for each $b \in B$                                ▷ $B$: the set of all boundaries
4.       if $b \neq p$                                   ▷ if $b$ is a valid test boundary
5.         $m \leftarrow$ sqrt($p$'s segs · $b$'s segs) · $M$      ▷ segs: the number of piecewise
                                                       ▷ linear segments forming a boundary
6.           for $i \leftarrow 1$ to $m$                      ▷ loop over cut attempts
7.             $c \leftarrow$ GENERATE_RANDOM_CUT($p$, $b$)
8.             if TEST_CUT($c$)
9.               return $c$                              ▷ $c$ is legal
10.  return NULL                                       ▷ no legal cut was found for $p$
```

We have not provided pseudo-code for the methods GENERATE_RANDOM_CUT() or TEST_CUT(). GENERATE_RANDOM_CUT() chooses a random piecewise linear segment on each of the two boundaries and uses the midpoint of each segment to form an endpoint of the potential cut. Alternatively, GENERATE_RANDOM_CUT() can also generate an infinite cut, *i.e.*, a cut from a location on the primary boundary to a location on an infinite boundary. Since it doesn't matter where the cut attaches to the infinite boundary, *Druid* simply chooses a location that makes the infinite cut horizontal. TEST_CUT(), which tests a cut's legality, is discussed in the next section.

8.3.4 Testing a Cut

A randomly generated potential cut must be tested for legality and it must satisfy a number of criteria in order to be legal. Because a cut is analogous to a scissor cut through an idealized physical surface, we can impose constraints on cuts that would be true of actual scissor cuts. One constraint is that both ends of the cut must attach to the interior (bounded) sides of the boundaries with respect to their signs of occlusion, *i.e.*, the cut must traverse through the bounded surface, not through empty space. Likewise, along a cut's full length, the cut can never exit and then reenter the surface because this would similarly correspond to cutting through empty space.

The *start location* of a cut is the attachment point of the cut to the primary boundary while the *end location* is the attachment point to the test boundary. *Cut weaving* is the process of assigning a depth at the start of a cut that matches the primary boundary's depth at the start location and then traversing the cut, assigning appropriate depth changes at each crossing as the traversal goes under and comes out from under surfaces. Cut weaving is similar to boundary traversal which is used during the labeling search described in Section 7.3. The crucial behavior associated with cut weaving, *i.e.*, the adjustment of the cut's depth as a traversal visits its crossings, is shown in the pseudo-code function UPDATE_DEPTH() in Section 11.2.2. A further constraint on legal cuts states that after weaving a cut, the cut's depth at the end location must match the test boundary's depth at the same location. In other words, in order to be legal, a woven cut must have legal crossings along its entire length, including the T-junctions at both ends (see Fig. 8.2). However, since the process of weaving implicity renders all crossings legal except the T-junction at the end location, the only crossing whose legality is in question after weaving is the final T-junction.

Fig. 8.4 illustrates how cut weaving is used to determine if a potential cut is legal. Both drawings show the same labeled knot-diagram. In the figure on the left, the cut is legal because the final T-junction (crossing *e*) is legal. In the figure on the right, the cut is illegal. Therefore boundary *Q* bounds the same surface as boundary *M*.

If a potential cut does not extend through empty space and weaves such that the attachment point at the end location forms a legal T-junction, then it is legal. A legal cut implies

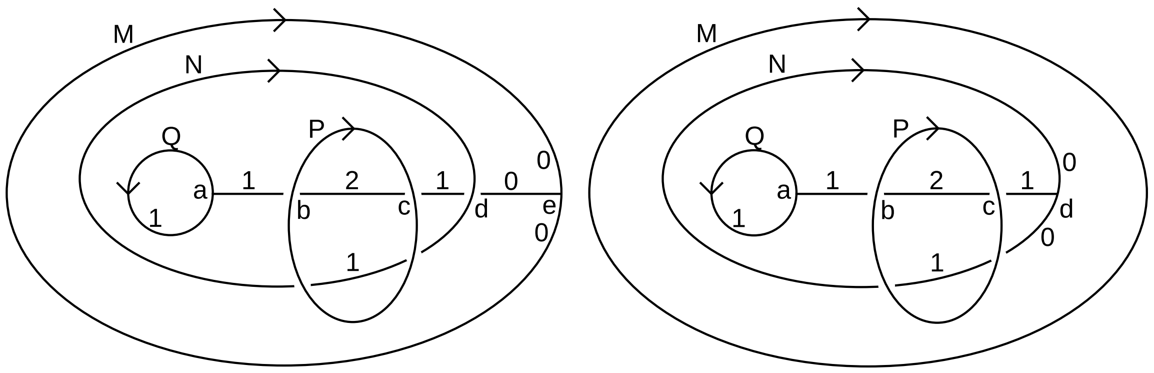

Figure 8.4: Using cut weaving to determine a potential cut's legality. To test a potential cut's legality, *Druid weaves* the cut. A depth is assigned at the start of the cut that matches the boundary's depth at the start location, *e.g.*, 1 for boundary *Q* in the drawings depicted above. The cut is then traversed, similar to boundary traversal (see Section 7.3), with appropriate depth changes assigned along the way. If the final T-junction (*e*, left and *d*, right) is a legal T-junction, then the cut is legal. The left cut is legal and the right cut is illegal.

that the two joined boundaries bound the same surface. It is impossible for a potential cut to be legal for two boundaries that do not bound the same surface.

8.4 Manual Cuts

Druid automatically finds cuts without any intervention from the user. However, in some situations it is useful for the user to *suggest* a cut. If a topological change invalidates an existing cut, *i.e.*, if the figure can no longer be legally labeled due to an impossible cut, that cut is deleted. However, new cuts cannot be found until after the labeling search because cut weaving relies on the boundary segment depths. Not having some cuts is generally acceptable since any missing cuts will be found after the labeling search and will be of great benefit during subsequent labeling searches and surface translations. However, a cut that is suggested *prior* to labeling can simplify the labeling search. A user suggests a cut by clicking on two boundaries to define the cut's endpoints. *Druid* then tests the potential cut. Some illegal suggested cuts can be rejected, but a suggested cut's legality cannot be definitively confirmed without the segment depths. Therefore, this method can suffer false positives in which an illegal suggested cut is accepted as legal and this will prevent *Druid* from finding a legal labeling. In summary, although this approach has serious limitations, the suggestion of a genuinely legal cut greatly aids the labeling search.

In the vast majority of cases a drawing can be constructed without any manual cuts and *Druid* will find the necessary cuts on its own. In fact, most users do not necessarily have to know that cuts are even part of *Druid's* representation. To emphasize this point, we often illustrate labeled knot-diagrams without showing the cuts because they are an element of the representation of which the user is generally unaware. However, if a greater emphasis were placed on detecting false positives, then perhaps manual cuts would be a genuinely useful feature of *Druid* for the purpose of improving the labeling search performance on complex drawings.

8.5 Improving Cuts

An *optimal* cut for a pair of boundariies is a cut with the minimum necessary number of crossings. There are two opposing goals when *Druid* attempts to find cuts. The first goal is to find cuts as quickly as possible while the second goal is to find optimal cuts, which necessarily takes longer because the first cut that is discovered may not be optimal. One might assume that the method of search described in Section 8.3.3 could be modified to reject legal cuts that are suboptimal. Thus, instead of terminating when the *first* cut is found, the search would not end until an optimal cut is found. There are two problems with such a search method however. The first problem is that there is currently no clear way to know if a cut is optimal. The second problem is that the search would take longer to run than the current method. We discuss these problems in detail next.

There is currently no known way to determine how many crossings an optimal *straight* cut must have. There is, however, a relatively simple method for determining many crossings an optimal *curved* cut must have. Fig. 8.5 illustrates examples of optimal straight and curved cuts for some drawings. Given the shortest chain of adjacent *regions* (contiguous areas of the canvas bounded by boundary segments) joining the two boundaries, the number of crossings an optimal curved cut must have is denoted by one less than the length of the region chain. For example, in Fig. 8.5 (*a* and *b*) boundaries *1* and *2* bound the same region, which corresponds to a region chain of length one. Therefore, those boundaries can be joined by an optimal curved cut with zero crossings. If the two boundaries bound two adjacent regions (a region chain of length two), as shown in Fig. 8.5 (*c*), those boundaries can be joined by an optimal curved cut with one crossing. However, *Druid* currently only uses straight cuts and we currently know of no method for ascertaining where or not a straight cut is optimal. If *Druid* were extended to use curved cuts, then this method would be useful for determining is a potential cut is optimal. From this point forward the term optimal cut will refer to optimal straight cuts.

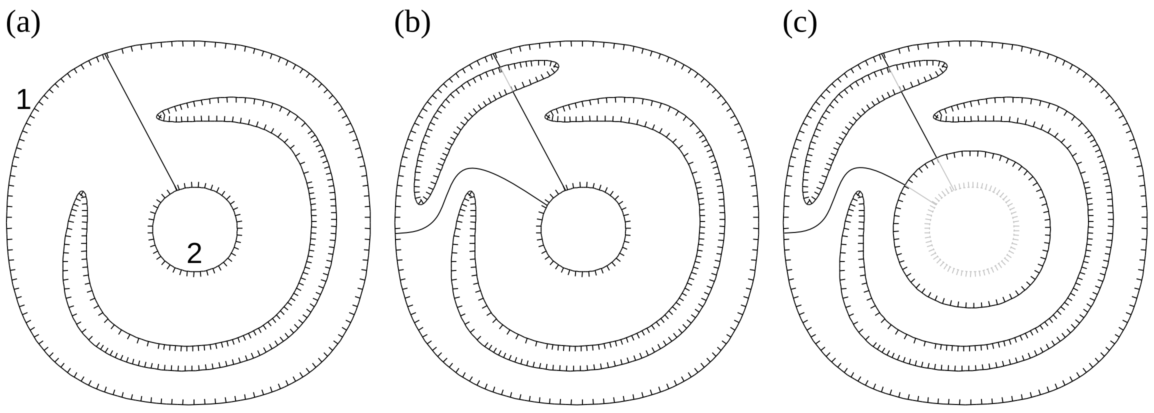

Figure 8.5: Examples of optimal cuts. In figure *a* an optimal straight cut has zero crossings. In figure *b* an optimal curved cut has zero crossings but an optimal straight cut has two crossings. In figure *c* an optimal curved cut has one crossing but an optimal straight cut has three crossings.

The only way to find an optimal cut is to perform an exhaustive search of all possible cuts between a pair of boundaries. In theory, an exhaustive search would never terminate since there are an infinite number of unique potential cuts that can join any two boundaries.

However, in Section 8.3.3 we explained that *Druid* only considers cuts with endpoints that are located at the midpoint of a piecewise linear segment forming a boundary.[3] Thus, *Druid* can only consider a finite number of endpoints for a cut and consequently can only consider a finite number of unique cuts. The number of unique cuts *Druid* can attempt between two boundaries is $S' \cdot S''$, where S' is the number of piecewise linear boundary segments on the primary boundary and S'' is the number of piecewise linear boundary segments on the test boundary.[4] Consequently, the maximum number of cut attempts to make in an exhaustive search is $S' \cdot S''$, although cuts will be generated randomly rather than being systematically enumerated. This exhaustive search can be terminated early if a cut with zero crossings is found (which is implicitly optimal), but in a situation where an optimal cut has at least one crossing, early termination is not possible.

A second problem with finding optimal cuts is that a cut search method that continues to find cuts after a successful cut has already been found will inherently take longer than the simpler method that terminates after the first legal cut is found. In fact, since an exhaustive search is required in order to find an optimal cut in a situation where an optimal cut has at least one crossing, searching for optimal cuts may take an extremely long time.

It should be noted that *Druid* does not have to find optimal cuts. Any legal cut suffices for operations that rely on cuts. However, while *Druid* might perform correctly with suboptimal cuts, its performance will be degraded. *Clean* cuts (those with few crossings) result in clean drawings. Two examples of operations for which the cleanness of a drawing can affect performance are the labeling search (see Chapter 7) and rendering (see Chapter 9). The labeling search performs slower for a less clean drawing for multiple reasons. The most serious problem is that additional crossings introduce additional crossing-state nodes

[3]Note that relying solely on the midpoints of piecewise segments means that *Druid* is *incapable* of finding an optimal cut in a situation where an optimal cut requires a cut endpoint that is not located at a piecewise midpoint. This problem could easily be alleviated by permitting *Druid* to consider cut endpoints at any location on a boundary. However, in practice, we have not found such an increase in complexity necessary.

[4]Note that $S' \cdot S''$ corresponds to line 5 from the pseudo-code in Section 8.3.3 with $M = 1.0$.

into the tree search, thus significantly increasing the the search space size. Rendering requires separate processing to determine a fill-color for each region of the canvas. Cuts with numerous crossings produce drawings with numerous regions.

Since fast turnaround times are crucial, but clean cuts are preferred, *Druid* attempts to improve cuts, but only in a way that minimizes its impact on the user's experience. During the cut search, *Druid* relies on the search method described in Section 8.3.1, which terminates for each boundary after the first legal cut is found. This method yields the fastest possible turnaround time. In order to improve cuts, *Druid* takes advantage of *idle time*, periods of inactivity on the user's part. When the user stops working with the drawing for a specified window of time, *e.g.*, 0.25s, *Druid* enters a *cut-improvement phase*. We assume that a user often pauses during the construction of a drawing to mentally plan the next steps, to study the current drawing, to simply take a break, etc. *Druid* attempts to take advantage of these pauses in the user's activity.

During the cut-improvement phase, *Druid* attempts to improve any cut with at least one crossing by continually attempting to find new cuts between the boundary pair that the original cut joins. The cut improvement for a single cut ends either when *Druid* finds a cut with zero crossings or when the maximum number of cut attempts has been reached. If an improved cut is found, it replaces the old cut.

In order to minimize the impact on the user's experience, the cut-improvement phase is aborted the instant a user begins a new interaction. The cut-improvement phase is simple to abort because *Druid* always relies on the current set of cuts. In this fashion, *Druid* steadily improves the cuts until they are optimal, but does so without affecting the user's experience. In fact, it improves cuts entirely without the user's awareness.

Chapter 9

Rendering

The ultimate goal when using *Druid* is to produce an image of a scene in which surfaces are displayed with their interiors filled with solid color (Fig. 9.1, lower left and lower right).[1] However, labeled knot-diagrams are most easily constructed and manipulated in a schematic style which does not represent a completed image. In this schematic style, which we call *labeled knot-diagram mode*, a labeled knot-diagram is displayed with closed curves denoting the boundaries of surfaces and with hash marks or arrows denoting the sign of occlusion (Fig. 9.1, top). No attempt is made to fill regions of the canvas with solid color. There are two important reasons to construct and manipulate a drawing using this mode of visualization. The first is that an image can be produced very quickly, thus improving the interactivity of the user's experience, *i.e.*, the task of redrawing the labeled knot-diagram on the display will have a minimal impact on the turnaround times for user interactions. The second reason is that a labeled knot-diagram can easily be visualized in

[1]To focus our research on the novel components of *Druid*, we have relied on simplified implementations of the less crucial components. For example, *Druid* uses fairly rudimentary curve-editing tools which make it difficult to construct complex shapes. Another way in which we have simplified *Druid* is the assignment of color to a surface, in which *Druid* is confined to the representation of surfaces with a single color throughout their interior. *Druid* could clearly be expanded to permit the assignment of more complex paint patterns to a surface, *e.g.*, color gradients, tiling patterns, or pictures.

this mode when the labeling is illegal. The boundary segments might have illegal depths, but they are simple to draw nonetheless.

Rendering is the process of translating a labeled knot-diagram into an image with solid color fills applied to the *regions* of the canvas (contiguous areas of the canvas disjointly partitioned by boundary segments). While visualizing a labeled knot-diagram in *rendering mode* is presumably the user's ultimate goal, it is not necessarily the best mode of visualization to use during the construction of the labeled knot-diagram. First, rendering is considerably slower than producing an image in labeled knot-diagram mode. Second, and more seriously, rendering cannot be performed on an illegal labeling. Thus, whenever a labeling is illegal, *Druid* shows a blank canvas in rendering mode.

This chapter describes how *Druid* renders a legally labeled knot-diagram. Specifically, we describe how *Druid* uses *slices*, which are similar to cuts (see Chapter 8), to determine which surfaces cover a region and their relative depth ordering within the region. This information must be determined in order to calculate a fill color for each region.

9.1 Overview

During most of a user's interaction with *Druid* the drawing is generally shown in *labeled knot-diagram mode*, a mode of visualization in which surface boundaries are displayed as closed curves, signs of occlusion are unambiguously indicated using either arrows or hash marks on the bounded side of the boundaries, and boundary segment depths greater than zero are dimmed to emphasize their occlusion. In such an visualization, the interiors of surfaces are not filled with solid color. Only the markings denoting the signs of occlusion indicate where the interiors of surfaces reside.

Rendering is the process by which a labeled knot-diagram (Fig. 9.1, top) is converted into an image in which contiguous bounded regions of the canvas partitioned along boundary segments are filled with designated colors. To render opaque surfaces (Fig. 9.1, bottom left), *Druid* only needs to find the depth zero surface for each region. However, to render transparent surfaces (Fig. 9.1, bottom right) it must find the full depth ordering of all surfaces for each region so a coloring model (see Metelli [30, 31]) can be applied. Since an opaque surface is simply a specific case of transparency (with an opacity factor of 1.0), *Druid* does not specifically perform opaque rendering. Instead, opaque rendering is a natural consequence of performing transparent rendering on surfaces with an opacity of 1.0.

The following steps must be performed in order to render a 2½D scene:

1. Collect all regions.
2. Calculate region colors.
3. Calculate suspected enclosing regions.
4. Draw all regions.

The next sections describe each step in detail.

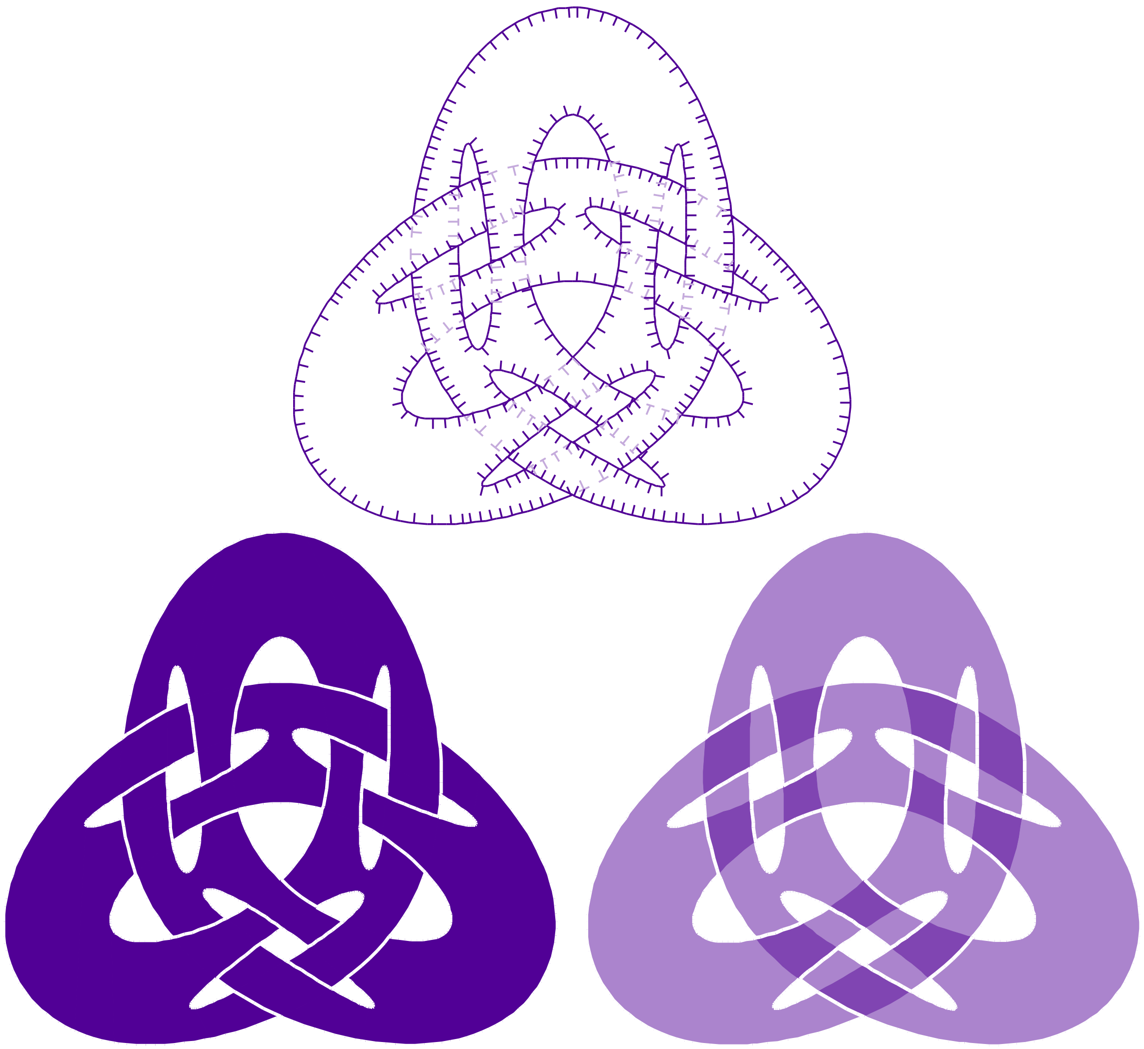

Figure 9.1: A labeled knot-diagram (top) can be rendered to produce an image in which regions are filled with solid color (bottom left, bottom right). The surfaces associated with each region must be determined so that a proper coloring for that region can be determined.

9.1.1 Collect All Regions

The full set of regions that result from a canvas partitioning must be collected before they can be rendered. If the drawing was projected to a planar graph with crossings representing nodes of the graph and boundary segments representing edges, then regions would be the faces of the graph. Therefore, the sequence of crossings encountered during such a traversal is a logical representation for a region. We define regions in a clockwise direction.

Druid finds a single region by traversing the region's border starting from an arbitrary crossing on the border. As the traversal visits new crossings, it always turns right. Thus, the traversal will eventually arrive back at the starting crossing, at which point the region's border will be defined and the proper sequence of crossings defining the border will have been gathered. It is important not to define any region more than once since multiple records of the same region could cause potential errors during rendering and other algorithms that rely on knowledge of the regions. Note that a boundary segment can be traversed in two different directions, each of which represents a unique *directed boundary segment*. For this reason, as *Druid* traverses a boundary segment it checks the corresponding *directed* boundary segment against a list of all directed boundary segment traversals that have occurred to this point in the region collection process. A boundary segment only bounds two regions, and a clockwise (right-turning) traversal of the two regions will traverse the shared boundary segment in different directions. Thus, once a boundary segment has been traversed in a particular direction, it can never be traversed in that direction again. If *Druid* finds the most recently traversed directed boundary segment in the list of previously traversed directed boundary segments, then the region has already been defined and *Druid* terminates the traversal and definition of the region currently being collected.

Druid finds all regions of the drawing by iterating over all crossings in the drawing, attempting a traversal in all possible directions away from the crossing. This must be done not only for boundary-to-boundary crossings, but also for boundary-to-cut crossings, cut-to-cut crossings, and boundary-cut-T-junctions. Note that there are situations in which it is important to find infinitely thin regions residing within the infinitely thin gap that a cut defines. Thus, traversals originating from boundary-to-cut crossings must be performed six times since the cut at the crossing can be traversed in four distinct ways and the boundary at the crossing can be traversed in two distinct ways. The cut can be traversed in two directions on the inside of the cut and in two directions on the outside of the cut whlie the boundary can be traversed in two directions. Likewise, traversals originating from cut-to-cut crossings must be performed eight times.

9.1.2 Calculate Region Colors

Transparency Coloring Model

Druid uses Metelli's episcotister model to determine region colors [30, 31]. This model is effectively the same as one of the more common transparency models used by modern graphics systems (see Angel [3]). Each surface has two properties, a color and an opacity. For a set of overlapping surfaces, the perceived color of the subset of surfaces below and including depth i is:

$$S_i = O_i \cdot C_i + (1 - O_i) \cdot S_{i+1} \tag{9.1}$$

where O_i is the opacity of the surface at depth i, C_i is the RGB color of the surface at depth i, and S_i is the perceived RGB color of the subset of surfaces below and including depth i. The background canvas has an implicit opacity of 1.0. Notice that for the set of surfaces below and including depth zero the perceived color S_0 is the perceived color of the region, *i.e.*, the RGB color that will be used to render the region. It is important to realize that this transparency model requires that the depth ordering of the surfaces be determined in advance.

Using Slices to Find Region Colors

To find the surfaces that cover a region and their depth ordering within the region, *Druid* uses a *slice*. A slice is similar to a cut except that instead of connecting two boundaries of a surface, it connects a boundary of a surface to an arbitrary location within the surface's interior (Fig. 9.2). To find the surfaces associated with a region, *Druid* finds slices that originate at a location inside the region and terminate on surrounding boundaries at various depths. This process establishes a depth-ordering for all of the surfaces associated with the region in question so that a coloring model can then be applied. For simplicity, *Druid* only uses straight horizontal slices that extend to the left from the *slice origin*, the point from which the slice originates. Although slices could extend in any direction, this assumption simplifies *Druid's* implementation.

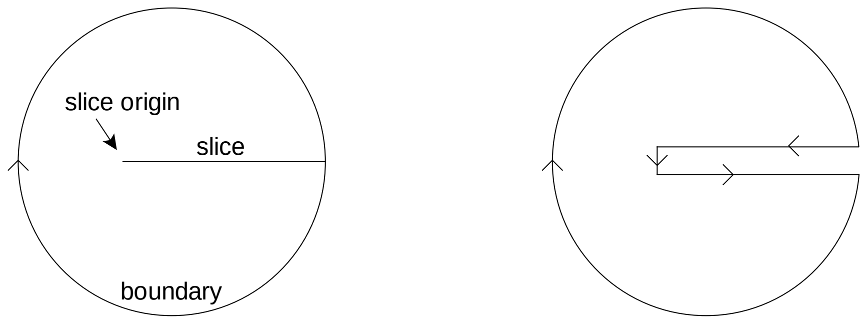

Figure 9.2: A slice connects a boundary to a location inside the bounded surface. Slices are used to find the depth ordering of the surfaces associated with a particular region. This must be done in order to render a scene.

For simplicity, *Druid* only considers slices that cross the region's boundary once. To cross the region's boundary a greater number of times the slice would have to exit and and reenter the region, which is analogous to a cut which exits and reenters a surface. Note that the sequence of points defining the region's boundary is known in advance of finding a slice origin, *i.e.*, every point of the piecewise linear boundary segments defining the region's boundary, *i.e.*, a polygon, was collected when the region was collected in Section 9.1.1. We can use the points defining the region's boundary to easily find a legal slice origin by testing at most two potential slice origins. These two points are determined using the point furthest left of the region's boundary (see Fig. 9.3).

Fig. 9.3 shows how *Druid* finds a legal slice from a region. There are two boundary segments associated with the point furthest left of the region's boundary, the boundary segment that precedes the point furthest left and the boundary segment that follows it. By taking the midpoint of one of these boundary segments and then traveling to the right a very small distance, we can find a slice origin that is guaranteed to cross the region's boundary exactly once. However, both possible boundary segments are not guaranteed to produce a legal slice origin, *i.e.*, one that is located inside the region and crosses the boundary exactly once. Determining which of the two possible boundary segments will result in a legal slice origin is a simple matter of trying both and seeing which one works (Fig. 9.3).

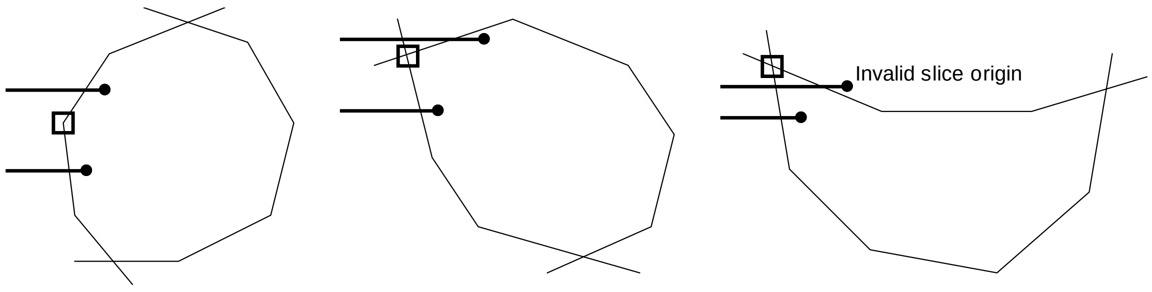

Figure 9.3: Slice origins for three sample regions defined by a polygon of piecewise linear boundary segments. Note that when a user constructs smooth curves, a large number of very small line segments generally define the polygon that represents a region, thus approximating a spline curve. These figures exaggerate the polygonal nature of a region to clarify how slice origins are found. The point furthest left on each region's boundary is marked with a square. The two potential slice origins for each region are marked with circles and their associated slices are marked with thick horizontal lines. The two potential slice origins are calculated by finding the midpoints of the boundary segments that precede and follow the point furthest left and then traveling to the right a small distance (exaggerated in these figures). In the left and center examples, both potential slice origins are legal and either can be accepted. In the right example, however, only one of the two potential slice origins is legal since the other potential slice origin lies outside the region.

Note that we cannot simply take the point furthest left and move a small distance to the right because the slice would then cross that point instead of one of the segments defining the region's border (see Fig. 9.3, right). Having a line intersect a point represents a nongeneric situation and is undesirable because the many algorithms that depend on slices can behave unpredictably if a piecewise linear line segment crosses a slice precisely at its endpoint.

Once a legal slice origin has been found for a region, the surfaces that cover that region and their depth ordering can be determined. Fig. 9.4 shows how the surface coverings for a region are found by repeatedly initiating traversals along the slice such that each subsequent traversal is initiated from an iteratively increasing depth. The first traversal starts at depth zero. As the slice is traversed, required depth changes are made as the traversal goes under and comes out from under surfaces at crossings along the slice. This process is called *slice weaving* the slice and is closely related to *cut weaving*, as described in Section 8.3.4. Cut and slice weaving are both related to boundary traversal (see Section 7.3 and Chapter 11). The slice is traversed, *i.e.*, woven, until a boundary-to-slice crossing is found where the crossing boundary's depth matches the slice's current traversal depth, *i.e.*, a legal T-junction (see Fig. 8.2). Once the slice traversal reaches a legal T-junction, the traversal is terminated and the surface whose boundary has been found is assigned a position in the region's depth ordering corresponding to the depth at which the traversal was initiated.

If a slice traversal fails to find a terminating boundary for its initiated depth, then there is no surface covering the region at that depth. Assuming the drawing is *vertically compact* (see Section 13.2), such an occurrence means no surfaces can possibly cover the region at a deeper depth than the depth that was just attempted, so the traversal loop is terminated. When the traversal loop ends, all surfaces covering the region have been identified and sorted based on the depth at which they cover the region. A coloring model can then be used to determine a single fill-color for the region.

In Fig. 9.4 the rendering in (*a*) shows a drawing of a transparent red circular surface above a green vertical cigar-shaped surface above a blue horizontal cigar-shaped surface. The three slices shown in the figure demonstrate how the depth covering is determined for the central region. *Druid* starts by finding a slice origin within the central region. It then repeatedly initiates slices from the slice origin, initiating the process with an increasing depth each time. The depth is initially zero, as shown in (*b*). The first crossing along the slice cannot form a legal T-junction, but the second crossing can. Therefore, the second boundary along the slice bounds the surface that covers the central region at depth zero, *i.e.*, the red circle. This process is then repeated at a deeper depth, *i.e.*, one, as shown in (*c*). The first boundary along the slice forms a legal T-junction. Thus, the green vertical cigar covers the central region at depth one. Likewise, as shown in (*d*), when starting at depth two, a legal T-junction is not found until the third crossing along the slice. Thus, the blue horizontal cigar covers the central region at depth two. *Druid* then attempts a slice at depth three (not shown). It fails to find a legal T-junction for that slice and therefore stops seeking surfaces that cover the central region. *Druid* now has the depth ordering for all surfaces covering the central region: red $\rightarrow$ green $\rightarrow$ blue. A coloring model can now be applied to calculate a single color for the region.

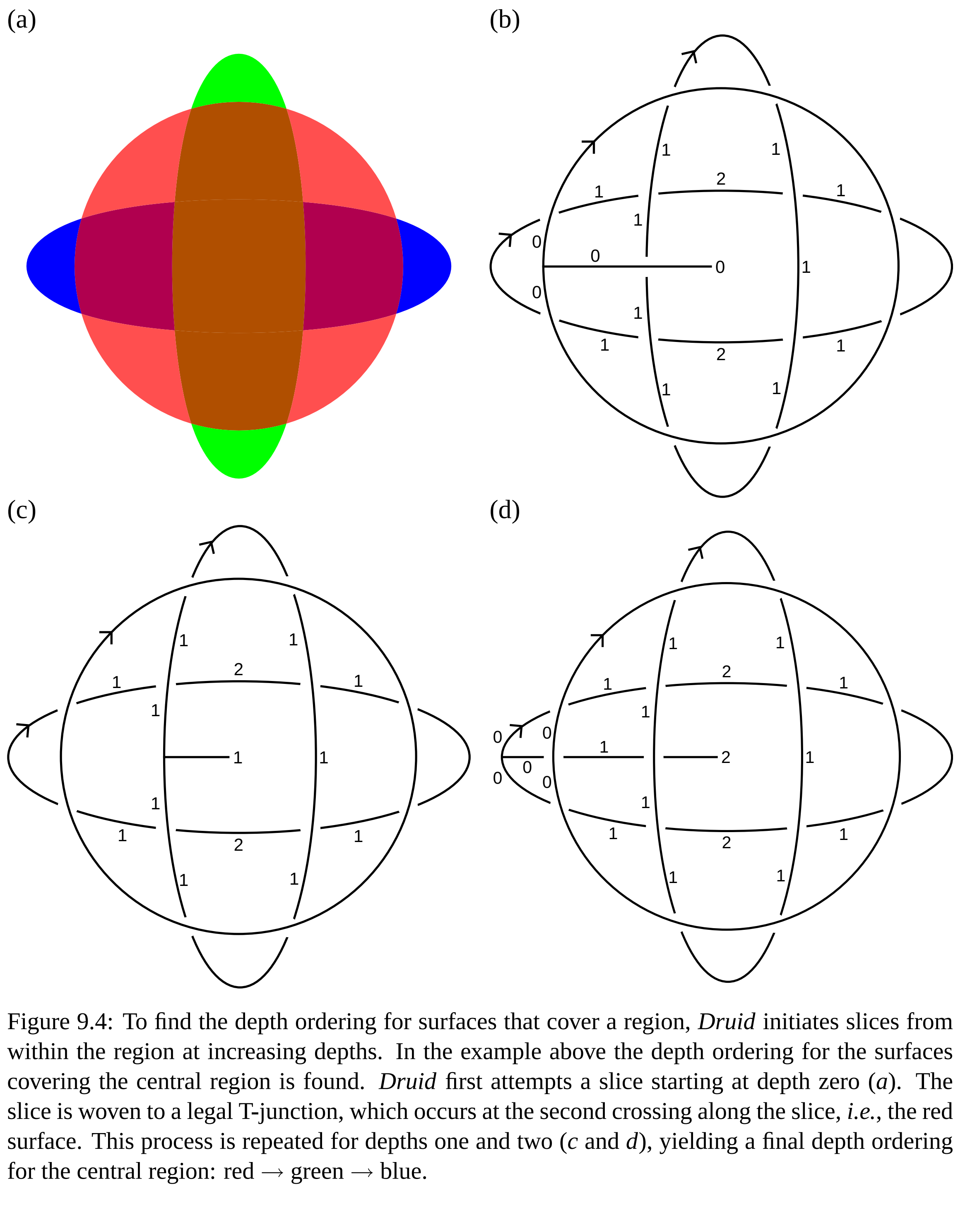

Figure 9.4: To find the depth ordering for surfaces that cover a region, *Druid* initiates slices from within the region at increasing depths. In the example above the depth ordering for the surfaces covering the central region is found. *Druid* first attempts a slice starting at depth zero (*a*). The slice is woven to a legal T-junction, which occurs at the second crossing along the slice, *i.e.*, the red surface. This process is repeated for depths one and two (*c* and *d*), yielding a final depth ordering for the central region: red → green → blue.

9.1.3 Calculate Suspected Enclosing Regions

Once all the regions and their fill colors have been determined, they are incrementally rendered one at a time. The effect of incremental rendering is that regions which are rendered later in the rendering process will erase any features of the drawing that overlap their area. Usually, this does not matter, because regions generally abut and do not overlap one another. However, there are circumstances under which one region may overlap another region, and one of the two regions must be rendered before the other to ensure that neither region improperly erases the other.

Consider a drawing which consists of multiple disconnected labeled knot-diagrams. Such a drawing is shown in Fig. 9.5, in which surface *1's* labeled knot-diagram is disconnected from surfaces *2* and *3's* labeled knot-diagram. Note that region *a* overlaps regions *b*, *c*, and *d* (region *a's* area comprises the entire interior of surface *1* since there are no crossings on surface *1's* boundary). Thus regions *b*, *c*, and *d* lie entirely within *a*, *i.e.*, they are *enclosed regions* of *a* and *a* is an *enclosing region* of *b*, *c*, and *d*. If the enclosed regions were rendered before *a*, then the subsequent rendering of *a* would effectively erase them. Thus, it is crucial to render *a* before rendering *b*, *c*, and *d*. In other words, it is crucial to render a region before any of its enclosed regions. Note that the importance of rendering the various regions in the correct order is unrelated to the actual depth ordering of the surfaces in the drawing because surfaces may be transparent, *i.e.*, it does not matter if surface *1* is above or below surface *2*; it is still important to render *a* before *b*.

Druid finds the *suspected enclosing regions* (regions which *might* be enclosing regions) for a region by finding those regions whose surface coverings are a subset of the region's surface covering. This information is gathered during slice weaving, as described in Section 9.1.2. The slices that are used for rendering the drawing in Fig. 9.5 are denoted by horizontal lines with their slice origins marked with a small disk. Notice that analysis of the crossings along the slice originating in *a* yields a surface covering of $\{1\}$, as labeled

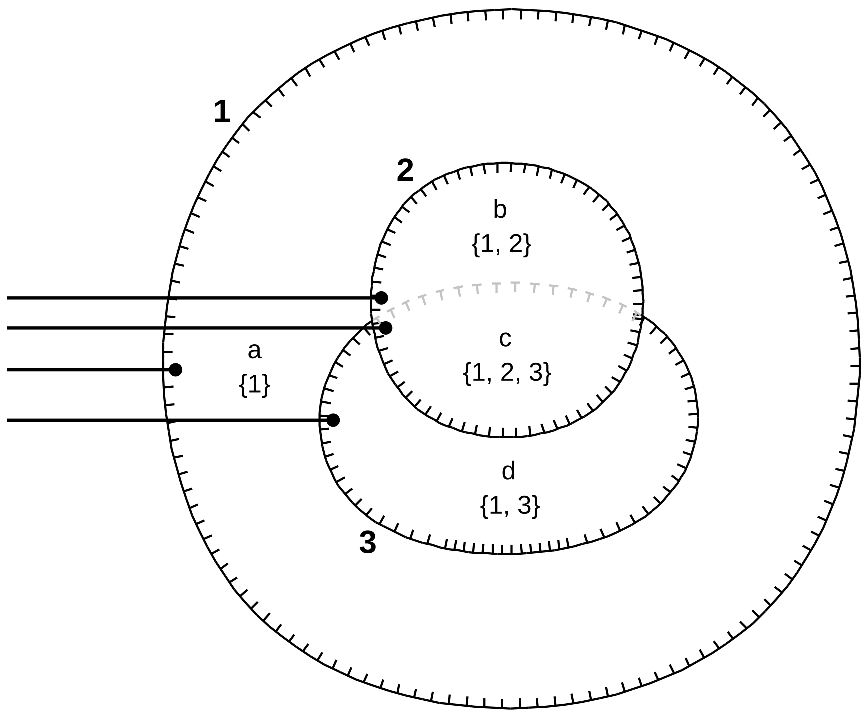

Figure 9.5: This drawing has two disconnected labeled knot-diagrams that partition into four total regions, marked with letters. Region *a* overlaps regions *b*, *c*, and *d*. Likewise, *a's* set of covering surfaces, $\{1\}$, is a subset of those for regions *b*, *c*, and *d*. The sets of surfaces covering each region are labeled in the figure. Because *a's* set of covering regions is a subset of the other three regions' sets of covering surfaces, *a* must be rendered before the other three regions.

in the figure, *i.e.*, analysis of *a's* slice misses the fact that surfaces *1* and *2* lie within its area and partially cover it. This mistake results from the fact that the slice origin lies just to the right of the point furthest left on region *a's* border (see Section 9.1.2 and Fig. 9.3). Analysis of *b's* slice discovers the set of covering surfaces $\{1, 2\}$. Since region *a's* surface covering is a subset of region *b's* surface covering, it *might* be crucial to render *a* before *b*. Thus, *a* is added to *b's* suspected enclosing regions list and *b* will not be rendered until all regions on its suspected enclosing regions list have been rendered.

The approach described above can produce false positives, in which one region is added to another region's suspected enclosing regions list when it is unnecessary to do so. For example, *b* and *d* would be added to *c's* suspected enclosing regions list as well as *a*, even though it is not crucial to render them before rendering *c*. Doing so is harmless however, so *Druid* makes no attempt to detect false positives.

9.1.4 Draw All Regions

Once the regions have been collected, their colors have been assigned, and their suspected enclosing regions lists have been established, the drawing can be rendered, one region at a time. *Druid* loops through all of the regions attempting to render each one. Any region for which a member of its suspected enclosing regions list has not yet been rendered is skipped. After attempting to render all regions, the loop is repeated on any regions that were skipped. Since the addition of one region to another region's suspected enclosing regions list depends on the nonsymmetric subset relation between the two lists, there is no risk that the suspected enclosing regions lists will contain cycles. The rendering is complete when no regions remain to be rendered.

This final stage in the rendering process, the drawing of the regions, could conceivably be performed by a dedicated graphics system such as a graphics card instead of being handled directly by *Druid*. Such an approach would consist of placing polygons in a three-dimensional scene using a system like OpenGL. One polygon would be placed in the scene for every surface covering of every region. Each polygon would be placed at a depth relative to the other polygons covering the same region that reflects the depth ordering for that region. The graphics system would then perform the necessary transparency calculation and produce the rendered image.

9.2 Exporting *Druid* Renderings

Although *Druid* is capable of representing and rendering interwoven surfaces, there remains the question of how drawings constructed with *Druid* can be exported to other applications. Most printers use *postscript* as a representation for vector-based graphics. Likewise, many computer systems use standard vector-based file formats, *e.g.*, *pdf*. In order to export a rendered *Druid* image to another environment, *Druid* converts the image to one of these standard formats. Notice that in our discussion of rendering, we have, in effect, described a simple method for converting labeled knot-diagrams to a planar format. Thus, once every region has been discovered and its rendering color established, the resulting rendering can be easily represented as a set of independent polygons, one polygon for each region. This conversion is similar to the planarized graph method described in sections 1.4 and 3.1.2 in which the drawing is converted into a planar graph and each face is assigned a color independently of the other regions in the drawing. Thus, the same planar graph that results from rendering is used both to produce an on screen visualization and to export *Druid* interwoven representation to other formats.

Chapter 10

Crossing-State Equivalence Classes

In Chapter 7 we described the basic search process which *Druid* uses to find a legal labeling. There is a potentially serious problem with the search as it has been described so far however. The search space size is exponential in the number of crossings in the figure and a drawing can easily have hundreds of crossings, resulting in an enormous search space. The heuristics described in Chapter 7 help *Druid* search a reasonably large search space quickly, but the search is inherently limited. There are two situations where the search can suffer prohibitively. The first is a simple case in which the search space is simply *so* large that the heuristics are insufficient to yield a search that terminates quickly. The second case occurs when the minimum-difference solution has a fairly high L_Δ. In such a case, minimum-difference solution's bound provides does not truncate enough of the search space to yield a search that terminates quickly, even if the search space size initially seems moderate, *i.e.*, the figure appears to be fairly simple.

During our research, we discovered a previously unrealized topological property of 2½D scenes called the *crossing-state equivalence class rule*. By building knowledge of this rule into *Druid*, we can effectively increase *Druid's* intelligence with respect to 2½D scenes, *i.e.*, *Druid* better understands the semantics of overlapping surfaces as a result of this rule. This improved understanding of 2½D scenes can be exploited to vastly improve *Druid's* performance. The crossing-state equivalence class rule can be exploited in two ways. First, since it represents a constraint on legal labelings, we can use it to eliminate labelings from consideration that violate this constraint. In other words, we can use it to vastly decrease the size of the search space, which we do by applying the crossing-state equivalence class rule as an additional form of constraint propagation during the search process. A second way to exploit this rule is to directly deduce the new labeling that follows some user interactions, which is must faster than a search regardless of any heuristics that are applied to the search process. The only user interaction we are aware of that can be performed using the second method is a crossing-flip.

This chapter describes a spectrum of drawing complexities, describes how transformations between these complexities are performed, and then describes the crossing-state equivalence class rule, which applies to drawings of a specific kind of complexity called *simple scenes*. Chapters 11, 12, and 13 then describe how equivalence classes are found and how they are exploited by *Druid*.

10.1 A Problem with the Labeling Search

The search as described in Chapter 7 has a problem. The search space size was established in Section 5.2 to be 2^C for a drawing with C crossings. A drawing may have hundreds of crossings however, which can result in a prohibitively large search space. Even when a drawing is relatively simple, there are cases where the minimum-difference solution has a fairly large L_Δ, and therefore does not provide a tight enough bound to guarantee that the search ends in a reasonable time. In the best case, the effect of a long search merely slows down *Druid's* performance, thus detracting from the quality of the user's experience. In the worst case, however, the search can timeout without finding a solution. *Druid* is then unable to perform the user's action, an effect which is detrimental to *Druid's* usefulness.

Consider the crossing-flip illustrated in Fig. 10.1 (top row). *Druid* fails to label this flip in under 120 seconds in fifty percent of trials despite the apparent simplicity of the figure. This problem is due to the fact that the minimum-difference solution has an L_Δ of sixteen, which is an insufficient bound the search tightly. An additional problem is that the search will have to explore realtively far from a user's clicked crossing to discover every crossing that must be flipped, and consequently, iterative deepening does not assist very well in the discovery of an early solution. Alternatively, consider the eight crossing-flips illustrated in Fig. 10.1 (bottom row). Despite the fact that each of these flips involve relatively small and topologically simple areas of the drawing, *i.e.*, areas involving few crossings, *Druid* takes thirty-five seconds on average to perform one of these flips and fails to perform them in under 120 seconds in two percent of trials.

We have discovered a previously unrealized topological constraint on 2½D scenes termed the *crossing-state equivalence class rule*. Exploitation of the crossing-state equivalence class rule permits us to significantly improve *Druid's* performance, thus increasing the complexity of drawings that a user can construct. There are two labeling tasks which *Druid* must perform: *labeling* and *relabeling*. Labeling is the task of assigning a labeling to an unlabeled or a partially labeled figure. Relabeling is the task of finding a new labeling following a crossing-flip. Crossing-state equivalence classes are applied to the two labeling tasks in very different ways. When applied to tree search labeling, they are used to vastly reduce the search space size by exploiting that fact that certain groups of crossings must flip as a unit. When used for relabeling, they are used to obviate the need for a search all together. Instead, the new labeling that results from a crossing-flip is directly deduced by applying the crossing-state equivalence class rule. The ways in which equivalence classes are exploited are described in greater detail in Chapter 13. To simplify our exposition, where it is important to distinguish the various labeling methods, we use *Druid (*SEARCH*)* to refer to a labeling search that makes no use of equivalence classes (as was described in Chapter 7), *Druid (*CSEC SEARCH*)* to refer to a labeling search that exploits equivalence classes, and *Druid (*DIRECT*)* to refer to the direct relabeling method.

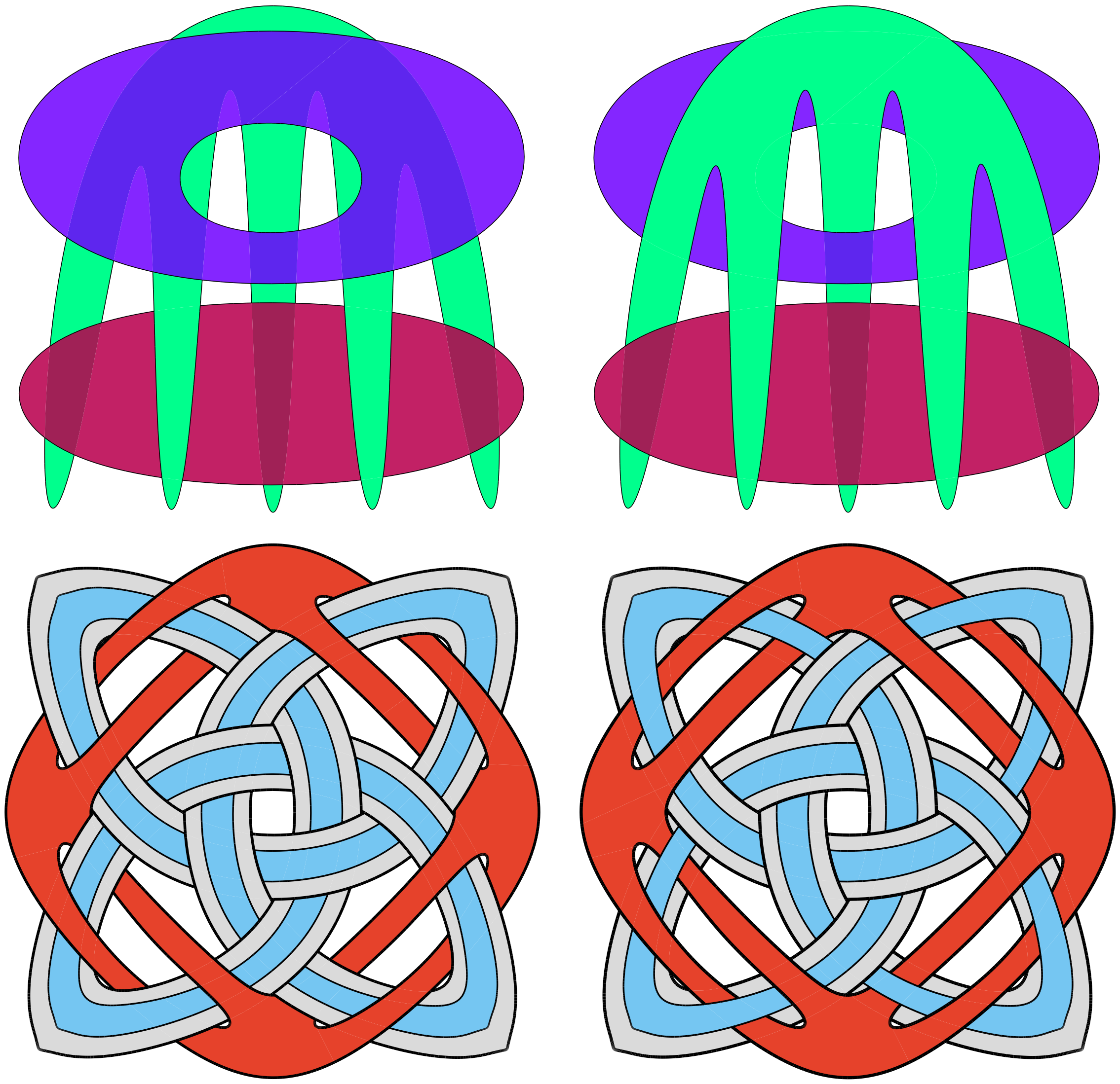

Figure 10.1: The labeling search described in Section 7 performs relatively poorly on the two crossing-flips shown here. In the top row a flip has been performed which requires many crossings to be flipped as a unit. In the bottom row, there are eight topologically identical flips, *i.e.*, many equivalence classes represent features that occur in multiple places in the drawing. Any one of these flips requires a significant search time to perform due to the complexity of the overall drawing.

10.2 Crossing-State Equivalence Classes

10.2.1 Definition of Key Concepts

Fig. 10.2 shows a 2½D scene of interwoven surfaces. A section of a boundary joining two crossings is termed a *boundary segment*. We observe that the canvas is partitioned by boundary segments into disjoint *regions*. In Fig. 10.2, the regions of the canvas are labeled with letters. We observe that zero or more surfaces (numbered in Fig. 10.2) cover every region and that at least one surface covers any region that lies within the interior of some surface. For example, surfaces *1* and *3* cover region *k* while surfaces *1*, *2*, and *3* cover region *m*.

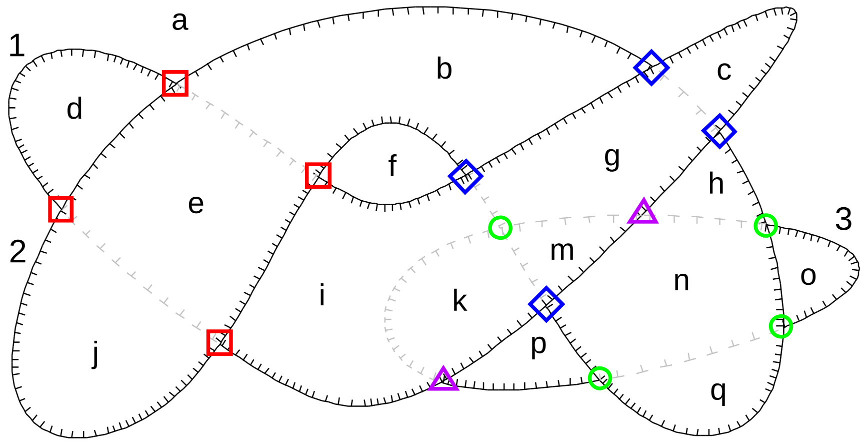

Figure 10.2: An interwoven 2½D scene. Regions are labeled with letters, surfaces with numbers, and crossing-state equivalence classes with shapes.

To define and prove the crossing-state equivalence class rule, we first define the following terms:

- A *superregion* is a set of contiguous regions covered by a single surface. For example, in Fig. 10.2, $\{b, g, h, n\}$ is a superregion of surface *2*.
- A *border* of a superregion is the set of boundary segments which define its perimeter.

- A *shared superregion* is the maximum superregion common to two surfaces, *e.g.*, $\{g, m\}$ is a shared superregion of surfaces *1* and *2*.
- A *corner* of a shared superregion is a crossing where adjacent boundary segments of the border belong to different surfaces. In Fig. 10.2, corners corresponding to the shared superregion $\{m, n\}$ common to surfaces *2* and *3* are marked with circles.

The corners of a shared superregion comprise the *crossing-state equivalence class* for that shared superregion. Notice that every crossing in a drawing is a corner of some shared superregion. Consequently, every crossing is a member of some crossing-state equivalence class.

10.2.2 Reducing General 2½D Scenes to Simple 2½D Scenes

Drawings can be divided into a number of complexity classes which can be placed on a spectrum of drawing complexities with drawings of maximum complexity (those with the loosest assumptions about a drawing's structure) at one end and drawings of minimum complexity at the other. Fig. 10.3 shows this spectrum. A *simple surface* is a surface with a single boundary component which does not intersect itself, *i.e.*, a *Jordon curve*. Two steps are required to reduce a general 2½D scene to a simple 2½D scene. First, any surface with multiple boundary components must be converted into a surface with a single boundary component. Second, any self-overlapping surfaces must be converted into a set of non-self-overlapping surfaces.

We perform both surface conversions using *cuts*. When a cut connects two boundaries, those boundaries are joined into a single boundary component (Fig. 8.1). Likewise, a self-overlapping surface with a single boundary component can be cut into multiple smaller surfaces which abut and such that no surface in the final scene self-overlaps (Fig. 10.4).

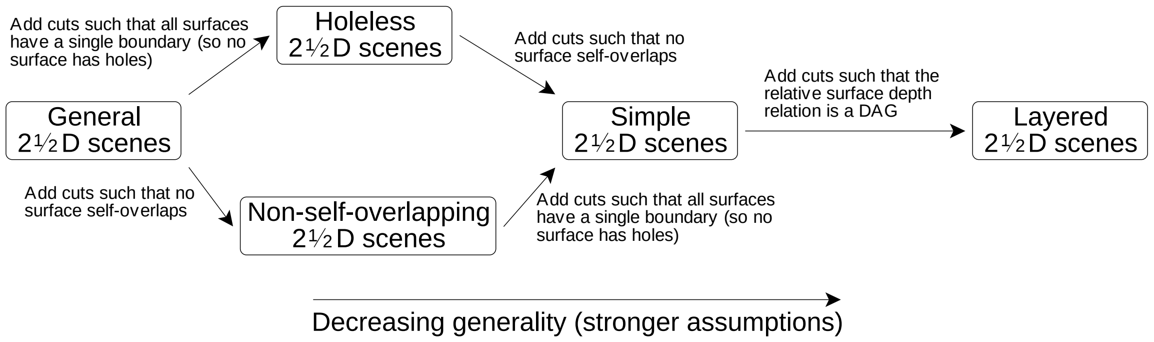
General 2½D scenes
Add cuts such that all surfaces have a single boundary (so no surface has holes)
Holeless 2½D scenes
Add cuts such that no surface self-overlaps
Simple 2½D scenes
Add cuts such that the relative surface depth relation is a DAG
Layered 2½D scenes
Add cuts such that no surface self-overlaps
Non-self-overlapping 2½D scenes
Add cuts such that all surfaces have a single boundary (so no surface has holes)
Decreasing generality (stronger assumptions)

Figure 10.3: Spectrum of levels of generality of 2½D scenes. Generality decreases from left to right. Scenes further to the right along this spectrum make stronger assumptions about the properties of a scene and those assumptions preclude more general scenes that can reside further to the left along the spectrum. The transition from simple scenes to layered scenes is analogous to *depth sort* rendering. The level of generality that *Druid* maintains is the *holeless* level, *i.e.*, *Druid* finds cuts to remove holes, but makes no attempt to convert self-overlapping surfaces into non-self-overlapping surfaces.

10.3 The Crossing-State Equivalence Class Rule

Let X and Y be the two surfaces whose boundaries intersect at a crossing. We observe that the crossing can only be in one of two states. Either surface X is above surface Y or surface Y is above surface X.

Theorem *All crossings in a crossing-state equivalence class must be in the same state.*

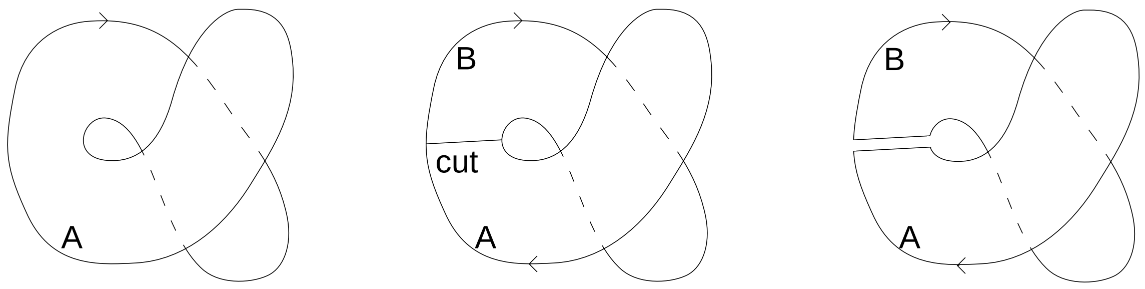

Figure 10.4: A cut connects two locations on the same boundary to break the boundary into two boundaries and the surface into two surfaces.

Proof We first prove the above theorem for simple surfaces. Because any general 2½D scene can be reduced to a simple 2½D scene, this suffices to prove the theorem in the general case. We begin by observing the following:

- We observe that for every region there is a total depth ordering of the surfaces which cover that region.
- The total depth ordering of adjacent regions is identical except for the addition or deletion (depending on the sign of occlusion) of the surface whose boundary segment separates the two regions.
- It follows that the relative depth of two surfaces in adjacent regions remains the same if the boundary segment which divides the regions belongs to neither surface.
- It follows that the relative depth of two surfaces is constant within a shared superregion.
- The relative depth of the two surfaces whose boundaries intersect at a crossing is the same as the relative depth of those surfaces in the region they corner.

Consequently, the relative depth ordering of two surfaces at every crossing in a crossing-state equivalence class must be the same. □

For example, in Fig. 10.2, consider the superregion $\{m, n\}$ shared by surfaces *2* and *3*. The only segment interior to the superregion is part of the boundary of surface *1*. Therefore, the relative depths of surfaces *2* and *3* cannot change along that boundary segment.

Chapter 11

Finding Crossing-State Equivalence Classes

The labeling search described in Chapter 7 suffers from a serious problem. It can take too long to terminate if the drawing is of sufficient complexity or if the minimum-difference solution has too great a L_Δ to provide a tight enough bounds. In Chapter 10 we described a topological property of $2^1/_2$D scenes which can be exploited to improved *Druid's* performance. In Chapter 13 we will describe how equivalence classes are exploited. First, however, we describe in this chapter how *Druid* finds the crossing-state equivalence classes of a drawing. We first describe this process using topological concepts. We then describe the process in more algorithmic terms. Later in this chapter we present experimental results that demonstrate the performance of the search for crossing-state equivalence classes.

11.1 Topological Description

Every crossing is the corner of some shared superregion that represents an area of overlap between two surfaces. A crossing's *neighbors* are the two corners of the crossing's shared superregion which precede and follow the crossing on the border of the shared superregion (Fig. 11.1). Equivalence classes represent the reflexive, symmetric, transitive closure of the crossing-state neighbor relation.

To find the equivalence classes for a legally labeled knot-diagram, *Druid* first searches for every crossing's two neighbors. Once the neighbors of all crossings have been found, equivalence classes can be constructed by computing the reflexive, symmetric, transitive closure of the neighbor relation.

Every crossing is associated with two *unoccluded* segments which the crossing cannot occlude regardless of the crossing-state, and two *potentially occluded* segments, one of which will be occluded and the other unoccluded depending on the crossing-state (Fig. 5.5).

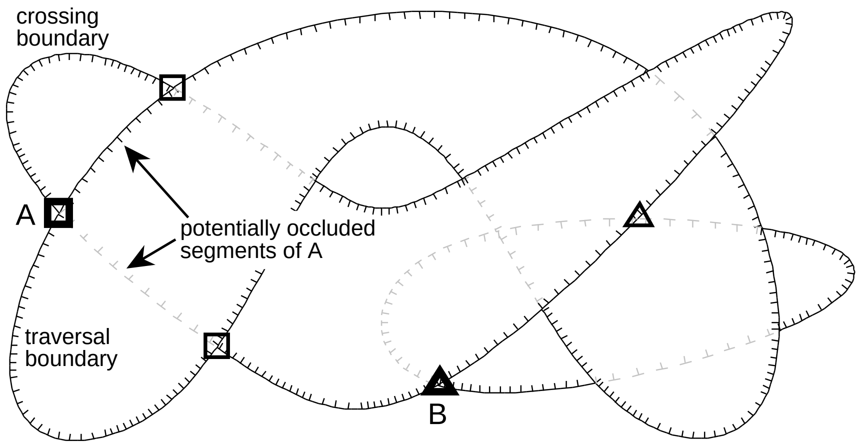

Figure 11.1: A crossing (thick square) and its *neighbors* (thin squares). A second crossing (thick triangle) and its *neighbor* (thin triangle). A crossing's neighbors are the two corners of the crossing's shared superregion which precede and follow it on the border. Note that for the crossing marked with a thick triangle, the neighbor that precedes it is the same as the neighbor that follows it, *i.e.*, it has only one neighbor, the thin triangle.

Call the crossing that *Druid* is seeking a *neighbor* for crossing *A*. Two boundaries cross at *A*, the *traversal boundary* and the *crossing boundary*. This distinction is arbitrary. *Druid* searches for one of *A's* two neighbors by traversing away from *A* along the traversal boundary in the direction of the potentially occluded segment on the traversal boundary. Before the traversal begins, *Druid* initializes the *target crossing boundary depth* with the crossing boundary's depth at *A*. During the traversal, the target crossing boundary depth is modified as the traversal goes under and comes out from under surfaces encountered at crossings. The traversal ends when it reaches the neighboring corner of the shared superregion. The neighboring corner is identified using the following criteria:

1. The boundaries of the same two surfaces that cross at *A* must also cross at the neighboring corner.
2. The traversal must arrive at the neighboring corner along one of that corner's potentially occluded segments.
3. The crossing boundary must be at the target crossing boundary depth at the neighboring corner, *i.e.*, it must have the same traversal-adjusted depth as the crossing boundary at *A*.

The first crossing the traversal finds that satisfies all three criteria is the first neighbor of *A*. By switching the role of traversal boundary and crossing boundary at *A*, the second neighbor of *A* is found. The next sections describe the process of finding crossing-state equivalence classes in greater detail.

11.2 Algorithmic Description

The search for equivalence classes on a legally labeled knot-diagram can be performed on a subset of the entire drawing, *i.e.*, if some set of crossings in the drawing require new assignment to equivalence classes, the following algorithm can be applied to that set without having to apply the algorithm to the drawing as a whole. We describe this process below.

The algorithm for finding equivalence classes on a legally labeled knot-diagram is performed with the following steps:

1. Prepare to find all crossing class neighbors in the drawing.
2. Find all crossing neighbors in the drawing.
3. Calculate the reflexive, symmetric, transitive closure of the crossing neighbor relation.

11.2.1 Preliminaries

The search for equivalence classes is initiated on a set of crossings that require assignment to equivalence classes. This set is called the *search list*. There are a number of reasons that a crossing may be on this list. The most common reasons are that the crossing was just created or one of the the boundary the crossing is on was just connected to another boundary through the construction of a new cut.

Before the search for equivalence classes can be performed, some necessary maintenance must be performed. First, any crossing on the same boundary as a crossing in the search list must itself be added to the search list. Second, any equivalence class containing a crossing on the search list must be destroyed before the replacement equivalence class is found.

At this point, the search can be performed on the search list.

11.2.2 Find All Crossing Neighbors in the Drawing

The first step in finding equivalence classes on a labeled knot-diagram is to find all of the crossing neighbors, *i.e.*, to find two neighbors for each crossing in the search list, one preceding the crossing and one following it on the shared superregion's border. Although we use the definition of a shared superregion to illustrate what an equivalence pair is, the actual algorithm for finding crossing neighbors does not require any knowledge of the shared superregions.

The search for one neighbor consists of a traversal away from the crossing along one of its two potentially occluded boundary segments to find the corner that should be paired with it. Note that the behavior of this algorithm, *i.e.*, traversing a boundary and tracking relevant depth changes as the traversal visits crossings, is closely related to the boundary traversal that is performed during the labeling search (see Section 7.3) and is also closely related to *cut weaving* and *slice weaving*, which we discuss in sections 8.3.4 and 9.1.2. The following pseudo-code illustrates the algorithm for finding crossing neighbors.

UPDATE_DEPTH() updates depth D of a boundary traversal according to the labeling scheme:

```
UPDATE_DEPTH($D$, $T$, $R$)        ▷ $D$: boundary depth, $T$: traversal boundary, $R$: traversal crossing
  **if** $T$ goes under at $R$
    $D \leftarrow D+1$
  **else if** $T$ comes out from under at $R$
    $D \leftarrow D-1$
  **else** $D$ is unchanged
```

FIND_NEW_CROSSING_NEIGHBORS() finds all necessary crossing neighbors by initiating boundary traversals from each crossing in the search list along the potentially occluded boundary segments:

```
FIND_NEW_CROSSING_NEIGHBORS($C$)        ▷ $C$: the search list
  **for each** $c \in C$
    **for** $b$ = 1 through 2        ▷ Two passes: left, right neighbors
```

```
if $b = 1$
  $B \leftarrow c$'s first potentially occluded boundary segment
else
  $B \leftarrow c$'s second potentially occluded boundary segment
$T \leftarrow B$'s boundary
$P \leftarrow$ the opposing boundary at $c$
$D \leftarrow$ depth of $P$'s unoccluded boundary segment at $c$
$F \leftarrow$ false
while $F$ = false                                          ▷ See note 1 below.
  $R \leftarrow$ the next crossing along the traversal
  if the opposing boundary at $R$ is not $P$
    UPDATE_DEPTH($D$, $T$, $R$)                            ▷ See note 2 below.
  else if $P$'s sign of occlusion at $R$ does not potentially
    occlude the boundary segment the traversal arrived on
    UPDATE_DEPTH($D$, $T$, $R$)                            ▷ See note 3 below.
  else if $P$'s unoccluded depth at $R$ is not $D$
    UPDATE_DEPTH($D$, $T$, $R$)                            ▷ See note 4 below.
  else
    Create the crossing neighbor pair ($c$, $R$).
    $F \leftarrow$ true                                     ▷ See note 5 below.
```

Notes

1. Traverse along *B* away from *c* until the next corner is found.
2. Continue with the traversal since corners must cross the same pair of boundaries and *R* crosses a different pair of boundaries than *c*.
3. Continue with the traversal since corners must corner the same quadrant, *i.e.*, the same shared superregion, and *R* corners a quadrant that differs from that of *c*.

4. Continue with the traversal since the shared superregion must be at the same relative depth at all of its corners since surfaces cannot interpenetrate.
5. The crossing is a corner of the correct quadrant involving the correct boundary at the correct depth, thus it is the corner being sought. Make the equivalence neighbor pair and continue to the next crossing $c \in C$.

We observe that since this algorithm is performed in a series of boundary traversals, surfaces must have a single boundary component if the algorithm is to perform correctly. However, surfaces may self-overlap, *i.e.*, the scene can be a holeless scene; it does not have to be a simple scene (see Fig. 10.3). Consequently, the drawing may contain cuts, and the boundary traversals must properly traverse boundaries that are connected by cuts. Thus, when the algorithm reassigns R, the next crossing along the traversal, the traversal must properly traverse the surface's single boundary by turning into and out of cuts at boundary-cut-T-junctions.

11.2.3 Calculate the Reflexive, Symmetric, Transitive Closure of the Crossing Neighbor Relation

Once all crossing neighbor pairs have been found, they must be grouped into equivalence classes. We observe that since each crossing is a member of two crossing neighbor pairs, and since the equivalence class is an equivalence relation, crossing neighbor pairs can be reflexively, symmetrically, transitively collected into larger sets of crossings, which we call equivalence classes. Once the crossings have been assigned to equivalence classes, the crossing neighbor pairs of no further use and are discarded.

11.3 Performance of the Search for Crossing-State Equivalence Classes

We want to exploit crossing-state equivalence classes because they decrease the search space size during labeling and eliminate the need to search during relabeling. However, they must be found in advance. Therefore, it is important to establish whether crossing-state equivalence classes can be found a timely fashion. In order to measure the performance of the search for crossing-state equivalence classes, we performed a series of experiments on a set of drawings of steadily increasing complexity (Fig. 11.2).

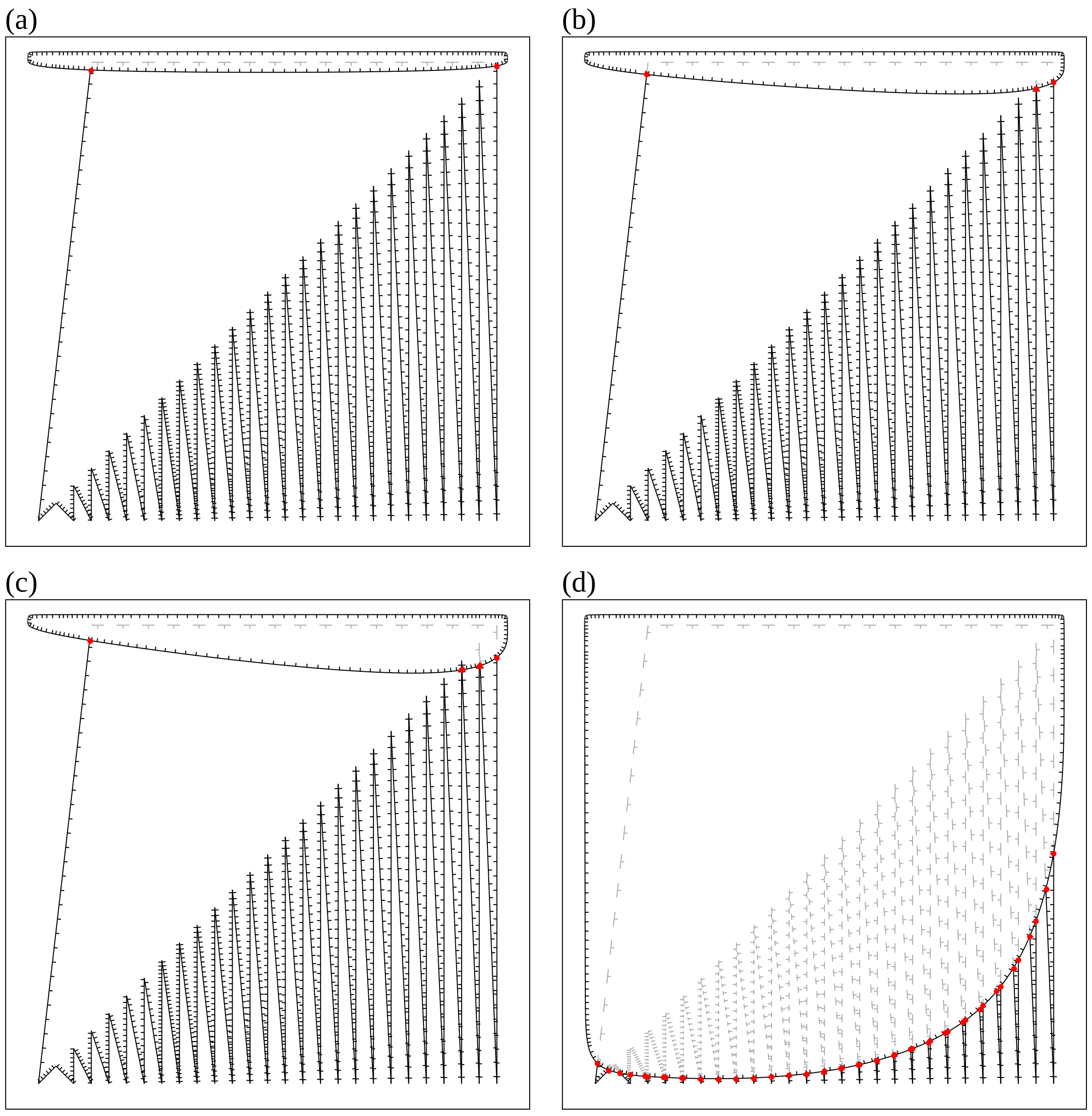

Figure 11.2: To test the performance of the search for crossing-state equivalence classes, we constructed a set of drawings that steadily increase in complexity. At any step in the progression, two new crossings are added to the figure. At all steps, the corresponding drawing has a single crossing-state equivalence class that contains every crossing in the drawing.

Figs. 11.3 and 11.4 show plots of the performance of the search for crossing-state equivalence classes relative to the number of crossings in the figure. All experiments were conducted on a 1.6 GHz PowerMac G5. We observe based on the plots that the performance scales polynomially in the number of crossings.

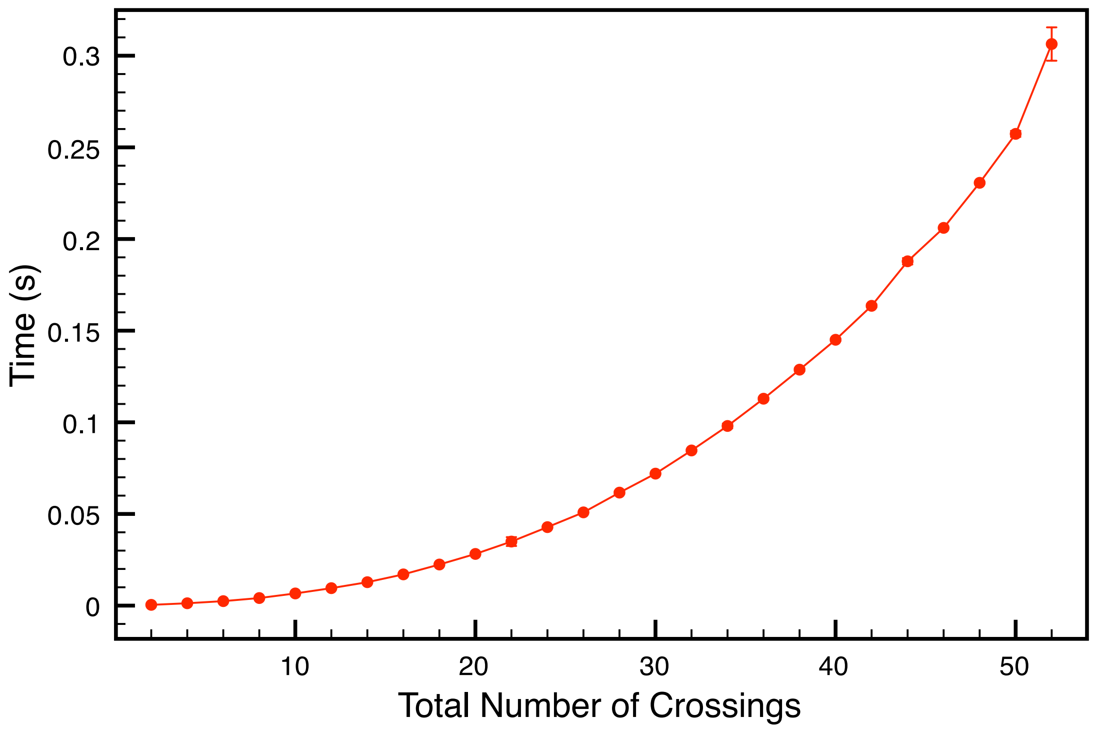

Figure 11.3: Search time for crossing-state equivalence classes vs. number of crossings. We observe that the performance is worse than linear in the number of crossings. Note that the actual time is very fast (.3s for fifty-two crossings).

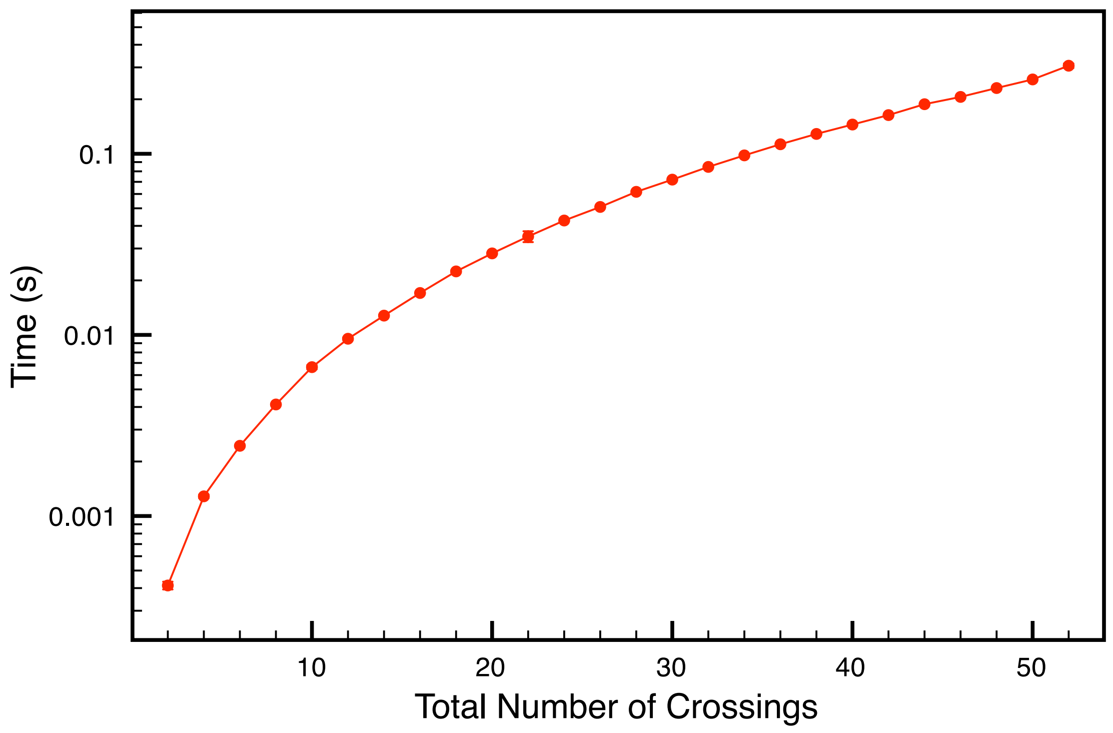

Figure 11.4: Search time for crossing-state equivalence classes vs. number of crossings with a logarithmic y-axis. We observe that the performance is better than exponential in the number of crossings. Since the performance is worse than linear in the number of crossings (see Fig. 11.3), we observe that the performance is polynomial in the number of crossings. Note that the actual time is very fast (.3s for fifty-two crossings).

Chapter 12

Equivalence Class Independence

In Chapter 10 we present a constraint on legal labelings, the crossing-state equivalence class rule. At first glance, this rule seems to embody the crucial aspects of a legal labeling, *i.e.*, it appears to be a *sufficient* condition for determining the legality of a labeling. This is not so. The equivalence class rule is a *necessary* but not sufficient condition of legal labelings. In other words, there are drawings for which some crossing-state equivalence class instantiations are impossible, *i.e.*, cannot be legally labeled. The reason some equivalence class instantiations can be impossible is that equivalence classes can be interdependent, *i.e.*, the state of one equivalence class can place constraints on the states of one or more other equivalence classes. One situation in which a lack of equivalence class independence can affect *Druid's* behavior is the crossing-flip interaction. If a crossing-flip results in a possible equivalence class configuration, *i.e.*, it can be legally labeled, we call that flip *atomic* since it corresponds to an elemental $2^1/_2$D scene change. However, if a crossing-flip results in an impossible equivalence class instantiation, then other equivalence classes must also be flipped in addition to the one the user explicitly flipped. Such flips are *nonatomic* and present problems for *Druid* due to inherent ambiguities in their interpretation. In other words, when the user attempts a nonatomic flip, there are generally multiple possible results and *Druid* cannot easily distinguish the user's intended result from the other results. Thus, when performing nonatomic flips, there is a risk that *Druid* will not produce the result that the user intends.

In this chapter we demonstrate equivalence class dependence, and *atomic* and *nonatomic* equivalence class flips. With the matter of equivalence class independence settled, we will then proceed to describe how *Druid* exploits equivalence classes in the case of atomic flips in Chapter 13.

12.1 Equivalence Class Independence

An important fact about equivalence class states is that, like crossing-states, they are not necessarily independent in all drawings. For a drawing with E equivalence classes, it may not be true that there exist 2^E equivalence class configurations for the drawing. 2^E simply represents an upper bound on the number of configurations a drawing can assume, *i.e.*, some instantiations of equivalence class states may be impossible, by which we mean that the corresponding knot-diagram cannot be legally labeled.

Fig. 12.1 shows a simple scene of three overlapping disks. This particular scene can be represented using a DAG, which will aid our discussion. Since there are three surfaces, and each surface is an element of a partially ordered set, there are only six possible DAGs that the surfaces of the drawing can assume: $1 \rightarrow 2 \rightarrow 3$, $1 \rightarrow 3 \rightarrow 2$, $2 \rightarrow 1 \rightarrow 3$, $2 \rightarrow 3 \rightarrow 1$, $3 \rightarrow 1 \rightarrow 2$, and $3 \rightarrow 2 \rightarrow 1$.

While the drawing can assume six possible configurations, it has three equivalence classes, which naively suggests that there are 2^3 (or eight) configurations. The two extra configurations correspond to equivalence class state instantiations which form a cycle rather than a DAG. One cycle is $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$. The other cycle is $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$.

In Fig. 12.1, the marked equivalence class cannot be flipped without flipping one of the other two equivalence classes as well in order to achieve a knot-diagram that can be legally

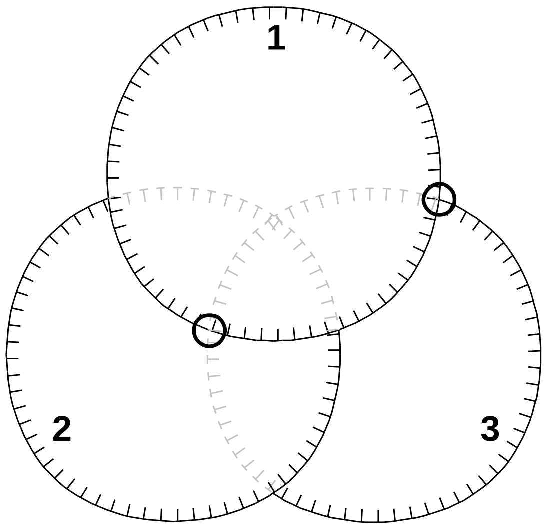

Figure 12.1: This figure shows a simple scene consisting of three overlapping disks. Notice that since this drawing can be represented as a DAG, it can be described by a partial ordering of the surfaces. There are only six DAGs and corresponding partial orderings that can describe this drawing, but there are three equivalence classes, which naively suggests that there ought to be 2^3 (or eight) equivalence class instantiations for the drawing. This discrepancy implies that two of the theoretical equivalence class instantiations are illegal, *i.e.*, they cannot be legally labeled.

labeled. However, flipping either of the other two equivalence classes could constitute a valid solution to the problem, and each would produce a different result. The two possible results of flipping the marked equivalence class are shown in the two bottom drawings of Fig. 12.2. Flipping one of the other equivalence classes would result in the partial ordering $3 \rightarrow 1 \rightarrow 2$, shown at the bottom left of Fig. 12.2, but flipping the other equivalence class would result in $2 \rightarrow 3 \rightarrow 1$, shown at the bottom right.

12.2 Atomic vs. Nonatomic Crossing-State Equivalence Class Flips

A *crossing-state equivalence class flip* is the method by which *Druid (*DIRECT*)* performs a crossing-flip interaction. To perform a crossing-state equivalence class flip, *Druid (*DIRECT*)* flips all members of the clicked crossing's equivalence class as a unit and then attempts to relabel the knot-diagram.

The crossing-state equivalence class rule might seem to imply that a crossing-flip user-interaction has a uniquely determined effect on the crossing-states of the knot-diagram, *i.e.*, every crossing in the equivalence class of the clicked crossing must be flipped, and no crossing in any other equivalence class need be flipped. However, as we have shown, it is not always possible to flip a single equivalence class without flipping other equivalence classes, *i.e.*, the result of flipping a single equivalence class can, in some cases, result in an illegal labeling. The user's ultimate intent must be to flip more than one equivalence class. Unfortunately, inferring which equivalence classes must be flipped in order to achieve the user's intent is impossible since there is no way to resolve the inherent ambiguity.

An *atomic* crossing-state equivalence class flip is one that can be performed independently of all other equivalence class flips in the drawing. Such a flip corresponds to an *atomic* change in a 2½D scene. If a flip results in an illegal labeling, then it can only be performed by flipping other equivalence classes as well. For this reason, we call such a flip *nonatomic*. Nonatomic flips can often be interpreted in multiple ways, *i.e.*, there may be multiple legal labelings that a nonatomic flip could correspond to depending on which other equivalence classes are flipped in addition to the equivalence class the user clicked on. This inherent ambiguity makes it impossible for *Druid* to deduce the user's intent, *i.e.*, *Druid* cannot know which of the multiple possibilities the user intends when the user performs a nonatomic flip.

We can avoid the ambiguity inherent in nonatomic flips by exploiting the fact that any nonatomic flip can be decomposed into a sequence of atomic flips, each of which is unambiguous, *i.e.*, there is only one way to interpret the user's intent. *Druid* forces the user to perform a nonatomic flip by performing a sequence of atomic flips instead. Fig. 12.2 shows how a nonatomic flip is broken down into a sequence of atomic flips. An attempt to flip the marked equivalence class in the top drawing would be nonatomic, since the result cannot be legally labeled. There are two possible intended outcomes, each of which requires flipping one of the other equivalence classes in the drawing while leaving the third equivalence class unflipped. The two possible outcomes are shown at the bottom of the figure. *Druid* cannot know which outcome actually corresponds to the user's intent, and thus cannot perform the user's specified flip without also producing a possibly unintended result. However, both outcomes can be accomplished by a sequence of two atomic flips. The first atomic flip is shown in the smaller intermediate drawings. The second atomic flip corresponds to the equivalence class the user originally intended to flip, which will have become atomic as a result of the intermediate flip.

When the user attempts to perform a nonatomic flip, *Druid (*DIRECT*)* does not perform the flip, but rather helps the user choose a sequence of atomic flips which will yield the desired result by displaying other equivalence classes which might need to be flipped as blinking on and off.

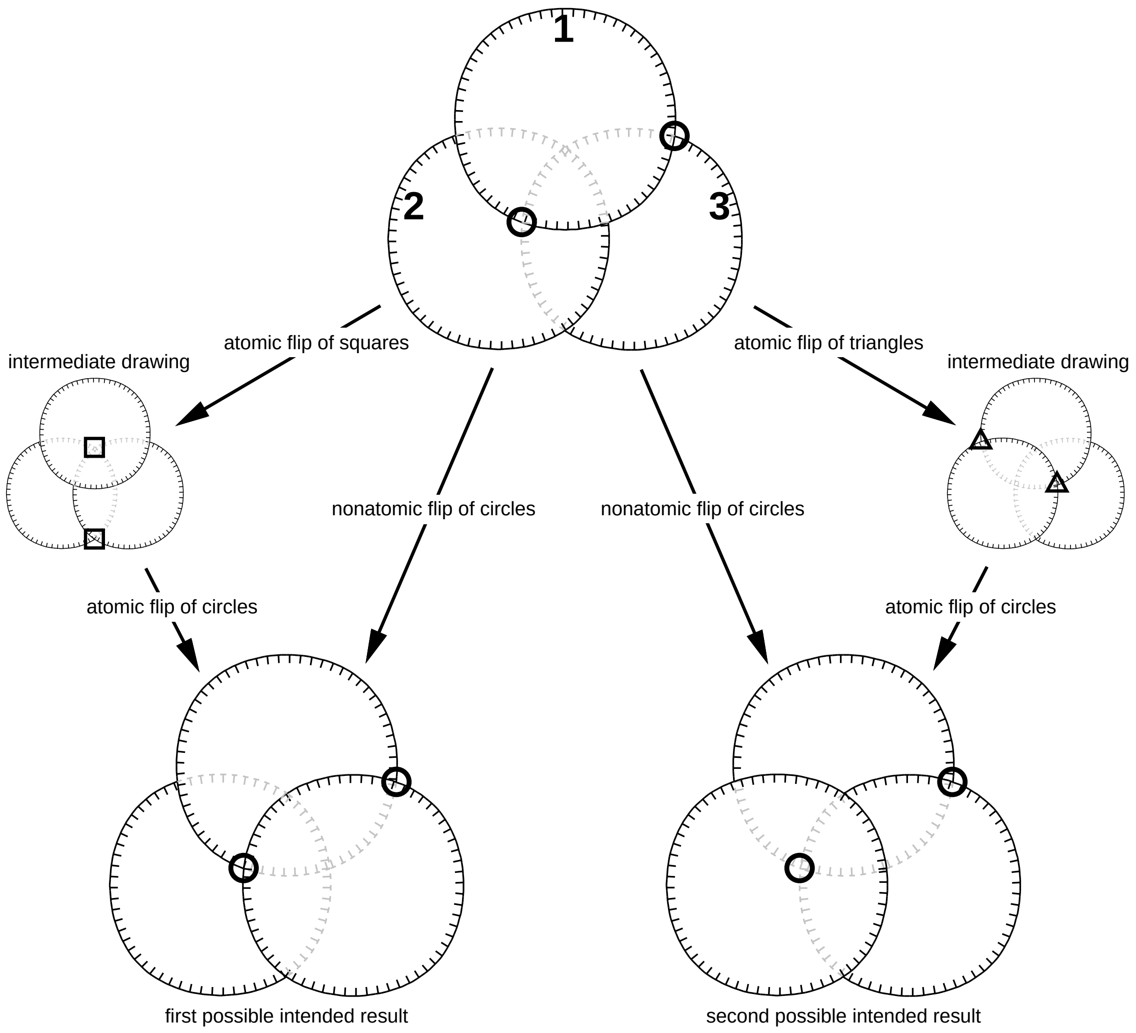

Figure 12.2: This figure shows the drawing from Fig. 12.1 at top, with a nonatomic equivalence class marked with circles. Flipping this equivalence class is a nonatomic flip since the result cannot be labeled without flipping other equivalence classes as well. The user's intent when attempting to flip this equivalence class must correspond to one of the two possible results shown at the bottom, but there is no way for *Druid* to tell which result is actually intended. However, each of the two results can be decomposed into a sequence of two atomic flips, the first of which are shown in the small intermediate drawings, and the second of which are shown below. The intermediate atomic flip will convert the desired nonatomic flip into an atomic flip, thus resolving the ambiguity.

Although *Druid (*DIRECT*)* currently employs the above method of resolving the ambiguity associated with nonatomic flips, there are alternate methods which could be used. The following lists some alternate methods for handling nonatomic flips:

1. Arbitrarily choose from among the various legal labelings consistent with the flip.
2. Allow the user to fix the states of some equivalence classes so that they cannot be flipped. Follow this user-specified constraint by performing Step 1 above.
3. Discover all possible results. Present them to the user in a table and ask the user to choose among them.

An interesting question is which of the proposed methods for handling nonatomic flips is best from the point of view of good user interface design. Our current method of prohibiting nonatomic flips is not necessarily the best approach. If *Druid* used the first method listed above, it would perform a nonatomic flip by arbitrarily choosing one of the various legal labelings that result from propagating the constraint. As a result, all clicks on a crossing would yield a change to the drawing, which is desirable since the user's intent clearly requires some kind of change to occur. This method might reduce the cognitive burden on the user since he would not have to manually navigate a sequence of atomic flips to achieve a nonatomic flip. On the other hand, this method might increase the cognitive burden on the user instead of decreasing it. If *Druid's* arbitrary result did not match the user's intent, then he would have to correct *Druid's* mistake. It is not clear how such corrections would be made since such corrections might not lead to the desired result. Without devising a method for the user to specify corrections to an incorrect nonatomic flip, this method cannot be used. The next method expands on this method in such a way that it becomes possible.

A second possibility is for *Druid* to choose arbitrarily, but to do so subject to a set of user-specified constraints. This method would require a new user-interaction in which the user constrains some equivalence classes to remain in their current state so that they cannot be flipped during a nonatomic flip. Initially the user would attempt a nonatomic flip without any constraints, *i.e.*, by using the first method described above. If *Druid's* arbitrary choice did not correspond to the user's intent, he would then undo the flip, reverting to the previous labeling, constrain some equivalence classes to their current state, and try the flip again.

The third method listed above for handling nonatomic flips, showing all possible solutions and letting the user choose his preferred result, presents the wrong affordances (see Section 2.2). We believe that *Druid* should present a legal labeling of a 2½D scene to the user, not a list of options from which to select. However, this method could be used to remedy the problem that the potential ambiguity of nonatomic flips poses. A more serious problem with this method is that there is no obvious bound on the number of legal labelings that might result from a nonatomic flip. In most cases, there will probably be relatively few options. However, there is no guarantee that *Druid* would not have to present a large number of legal labelings from which the user would be required to choose.

Due to the ambiguity of nonatomic flips and the potentially large number of possible solutions that might result, *Druid* currently does not permit nonatomic flips. Instead, it forces the user to perform a series of atomic flips. For this reason, in the remainder of this discussion, references to an equivalence class flip will assume that the flip in question is atomic.

Chapter 13

Exploiting Crossing-State Equivalence Classes

In Chapters 10, 11, and 12 we described a topological property of 2½D scenes called the crossing-state equivalence class rule, we described how *Druid* finds equivalence classes, and we showed that equivalence classes are not always independent and how equivalence class interdependence affects *Druid's* handling of equivalence class flip interaction. What remains to be explained about equivalence classes is precisely how *Druid* exploits them to improve its performance. In this chapter we describe how *Druid* actually exploits equivalence classes to greatly reduce the search space size in some cases and to obviate the need to search in other cases. Later in this chapter we present experimental results which demonstrate the improved performance that is gained by using equivalence classes for relabeling.

This chapter represents the culmination of our work on crossing-state equivalence classes. Equivalence classes represent a significant portion of our work because they embody a previously unrealized topological property of 2½D scenes. As such, their discovery contributes not only to *Druid's* performance, but to a greater understanding of the topology of scenes of surfaces and the relationships of surfaces to one another in the world. Not only are they applicable to a drawing program such as *Druid*, they may very well have applications to the field of computer vision, in which they could be incorporated into systems that attempt to intelligently interpret and understand visual scenes.

13.1 Applying Crossing-State Equivalence Classes to the Labeling Search

In Section 10.1, we described two labeling tasks: 1) labeling (a tree search of the space of possible labelings); and 2) relabeling (a transformation from one legal labeling to another following a crossing-flip). Relabeling can be accomplished without a search, as described later in Section 13.2. Labeling, however, requires a search for a new legal labeling. Crossing-state equivalence classes can be of great use when performing the tree search. However, they can only be partially applied to the search process. Since the method for finding crossing-state equivalence classes requires that the labeling be currently legally labeled, it is impossible to use equivalence classes to assist in the initial labeling of unlabeled elements of a figure. However, once a figure is legally labeled, we can find the crossing-state equivalence classes for the entire figure and use them to reduce the search space during subsequent labeling searches.

Consider the effect that crossing-state equivalence classes have on the search space. In Section 5.2, we showed that the naive search space has a size of 2^C for a drawing with C crossings. Equivalence classes can theoretically reduce the search space from 2^C to 2^E for a drawing with E equivalence classes, although the full reduction is often not possible, as explained below. 2^E represents a significant reduction in the search space because there are always far fewer equivalence classes in a drawing than there are crossings. E can be at most size $C/2$ since equivalence classes always come in even sizes and thus have a minimum size of two. Consequently, even in the worst case $2^E = 2^{C/2}$. Notice that this gain often cannot be entirely realized because *Druid* might not have complete knowledge of the equivalence class structure for all of the crossings prior to the search, in which case the search space will be a combination of equivalence class states for those crossings for which equivalence classes are currently known and individual crossing-states for the remaining crossings. However, in practice, even a partial application of equivalence classes can have a dramatic effect on the search space size. Once the search is complete and a legal labeling is found, equivalence classes can be found for all crossings in the drawing, and thus, can be applied to future labeling searches.

The drawing shown in Fig. 13.2 contains forty crossings. Thus, the search space for *Druid (*SEARCH*)* has size 2^{40}. *Druid (*CSEC SEARCH*)* exploits the fact that there are only seven equivalence classes in the figure. Thus, it can theoretically search a space of size 2^7, although as stated, the space may be somewhat larger depending on which elements of the drawing have not yet been labeled in any given situation.

The reduction in the search space is accomplished by virtue of the fact that the equivalence classes make explicit the same information that constraint-propagation would otherwise implicitly deduce. A node in the search can be expanded in one of two ways: either the corresponding crossing's state is maintained or it is flipped. If the crossing-state is maintained during the node's expansion, then that crossing and all members of the associated equivalence class are locked to their current state so that they cannot be flipped during the search of the subtree under that node. Alternatively, if the crossing-state is flipped during the node's expansion, then all other members of the associated equivalence class are immediately flipped as well, and all members are similarly locked to the new state. In this way, the crossing-state equivalence class rule is maintained at all times during the search.

13.2 Relabeling Without Search

In Chapter 12, we showed that the crossing-states of the new labeling following an atomic flip are uniquely determined, and thus no search is necessary for the crossing-states. It might seem necessary to perform a search to find the new boundary segment depths for the labeled knot-diagram, but with two basic assumptions, boundary segment depths can be deduced directly from the crossing-states. The first assumption is that the labeled knot-diagram is *normalized*, *i.e.*, that the depth of the shallowest boundary segment in the entire drawing is zero. The second assumption is that the labeled knot-diagram is *vertically compact*, *i.e.*, that the drawing is compacted in the depth dimension as much as the constraints of the labeling scheme will allow. With these two assumptions, the crossing-states uniquely determine the boundary segment depths.

It is preferable to confine the relabeling of boundary segment depths to an area local to the flipped equivalence class when the user flips a crossing because such behavior will scale better with the complexity of the drawing than relabeling all boundary segment depths in the drawing. Thus, *Druid (*DIRECT*)* propagates depth-changes through the knot-diagram away from the flipped crossings rather than globally relabeling all boundary segment depths.

We recall, based on Fig. 5.5, that a crossing always has two potentially occluded segments and two unoccluded segments. When a crossing is flipped, the depths of its two potentially occluded segments will always change and the depths of its two unoccluded segments will never change (see Fig. 13.1). Since the depths of the unoccluded segments do not change, the new depths for the potentially occluded segments can be deduced directly by applying the labeling scheme to the flipped crossing-state and the two unoccluded boundary segment depths. After deducing the new depth for a boundary segment, that boundary segment's depth is fixed and cannot be changed again during the propagation process. We say that such a boundary segment is *depth-constrained*. This constraint guarantees that the depth propagation process always converges.

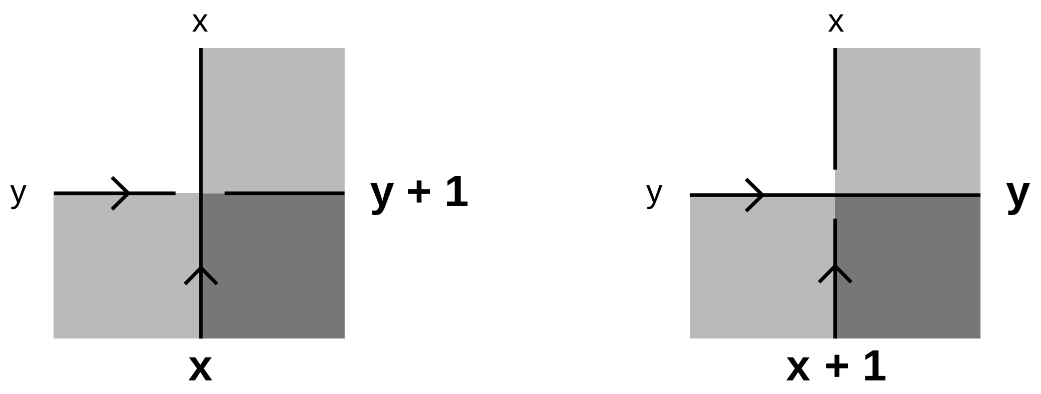

Figure 13.1: When a crossing is flipped, *e.g.*, transforming from the state shown at left to the state shown at right, the depths of its two potentially occluded segments will always change (large bold text) and the depths of its two unoccluded segments will never change (small plain text).

*Druid (*DIRECT*)'s* relabeling method processes crossings in a FIFO [1] queue. This queue is initially seeded with all crossings in the flipped equivalence class. For each crossing in the queue, *Druid* assigns new boundary segment depths to some of its four boundary segments in order to make the crossing legal. When members of the equivalence class are retrieved from the queue, new boundary segment depths are always assigned to the potentially occluded segments and never to the unoccluded segments, as described above. When crossings are retrieved from the queue that are not a member of the equivalence class, their boundary segment depths must be reassigned in a uniquely determined fashion that makes the crossing legal.

When the relabeling process assigns a new depth to a boundary segment, the propagation process must propagate across that boundary segment to the next crossing. Thus, the next crossing is added to the queue. The effect of processing the propagation in a FIFO queue is that changes occur near all members of the equivalence class equally early in the propagation process and then expand outward from crossings in the equivalence class.

[1]"first in, first out"

As the propagation traverses boundaries and reaches new crossings, some of the four boundary segments incident at those crossings will be depth-constrained, as described above. The one exception to this rule will be the members of the equivalence class. The propagation process did not add them to the queue. Instead, they were directly inserted into the queue as a result of the equivalence class flip. Therefore, they will not have any depth-constrained boundary segments. However, as described above, their new boundary segment depths will be uniquely determined. For all other crossings, the effect of the propagation process is that at least one boundary segment will be depth-constrained. The unconstrained depths of a crossing that the propagation process reaches are uniquely deduced by applying the labeling scheme to the crossing's state and the depth-constrained boundary segment depths.

If at any time the propagation process reaches a crossing that cannot be legally relabeled within the confines of its depth constraints, the propagation process must be abandoned and the user's desired flip cannot be performed. Such a situation corresponds to an attempted nonatomic flip since continuing the propagation process would require that crossings which are not members of the user-flipped equivalence class be flipped.

In Chapter 12 we described a possible alternative method that *Druid* could use to implement a nonatomic flip, *i.e.*, it could automatically flip other equivalence classes to find an arbitrary labeling consistent with the user-specified flip. The dependence between some equivalence classes specifies which equivalence classes *Druid* should consider flipping in order to complete the nonatomic flip. We can be confident that the user's intent does not require flipping other equivalence classes that are independent of the equivalence class they clicked on. However, equivalence classes that are dependent on the clicked equivalence class might require flipping. Thus, *Druid* could enumerate the space of possible equivalence class instantiations for the set of dependent equivalence classes, attempting the boundary segment depth propagation for each instantiation. The enumeration would be terminated when an instantiation that can be labeled is found. If the iteration were ordered such that instantiations that flip the fewest number of equivalence classes were attempted first, then the first solution found would be a minimum-difference labeling, and thus would be a preferable solution to the search for reasons we described at the beginning of Chapter 5.

Since the solution of the proposed method of performing nonatomic flips is relatively arbitrary, it could produce a result that does not match the user's intent. Thus, *Druid (*DIRECT*)* does not work this way. Instead, it requires the user to perform a series of atomic flips. Further research is required to decide which method is ultimately the best to use.

13.3 Performance of the Direct Crossing-State Equivalence Class Flip

13.3.1 Example Performance

The following experiments demonstrate the improvements gained by *Druid (*CSEC SEARCH*)* and *Druid (*DIRECT*)* over *Druid (*SEARCH*)*. Bear in mind, however, that *Druid (*CSEC SEARCH*)* is not used for crossing-flips in practice since *Druid (*DIRECT*)* generally performs the best. We have only provided data for *Druid (*CSEC SEARCH*)* to help to illustrate its effectiveness over *Druid (*SEARCH*)*. We tested *Druid's* performance under three circumstances:

- Flipping a small equivalence class on a simple figure
- Flipping a large equivalence class on a simple figure
- Flipping a small equivalence class on a complex figure.

Flipping a Small Equivalence Class on a Simple Figure

Fig. 13.2 shows a drawing of low complexity before an equivalence class flip is performed (top) and after two different equivalence classes have been flipped (bottom left and bottom right). The equivalence class that has been flipped in each case is marked with circles. The flip at bottom left involves a fairly small equivalence class, consisting of only four crossings, while the flip at bottom right involves a fairly large equivalence class, consisting of sixteen crossings. Tests were performed on a 1.6 GHz G5 PowerMac with labeling searches terminated after 120s if no solution was found. The plot in Fig. 13.3 shows the running times required to perform the flip illustrated in Fig. 13.2 (bottom left) for each of the three methods (*Druid (*SEARCH*)*, *Druid (*CSEC SEARCH*)*, and *Druid (*DIRECT*)*). The plot in Fig. 13.4 shows the running times required for each method to perform the flip illustrated in Fig. 13.2 (bottom right). Searches in which no solution was found are not included in the plots.

We observe that *Druid (*DIRECT*)* performs well for both of the flips that were attempted on this drawing, producing search times of approximately 0.003s. Note that when flipping the large equivalence class, *Druid (*SEARCH*)* fails to find a solution within 120s in fifty percent of trials. We observe that the benefit of exploiting equivalence classes is significantly greater for large equivalence classes. For example, the mean running time plot in Fig. 13.4 shows that *Druid (*DIRECT*)* is almost 1000 times faster than *Druid (*SEARCH*)* for the flip shown in the bottom right of Fig. 13.2. Alternatively, the mean running time plot in Fig. 13.3 shows that *Druid (*DIRECT*)* is only eighty-five times faster than *Druid (*DIRECT*)* for the flip shown in the bottom left. The equivalence class in the second flip contains only four times as many crossings as the equivalence class in the first flip (sixteen crossings vs. four crossings), yet the benefit of exploiting equivalence classes in the second flip is eleven times greater than in the first flip. Thus, the benefit of exploiting equivalence classes scales faster than linear in the size of the equivalence classes.

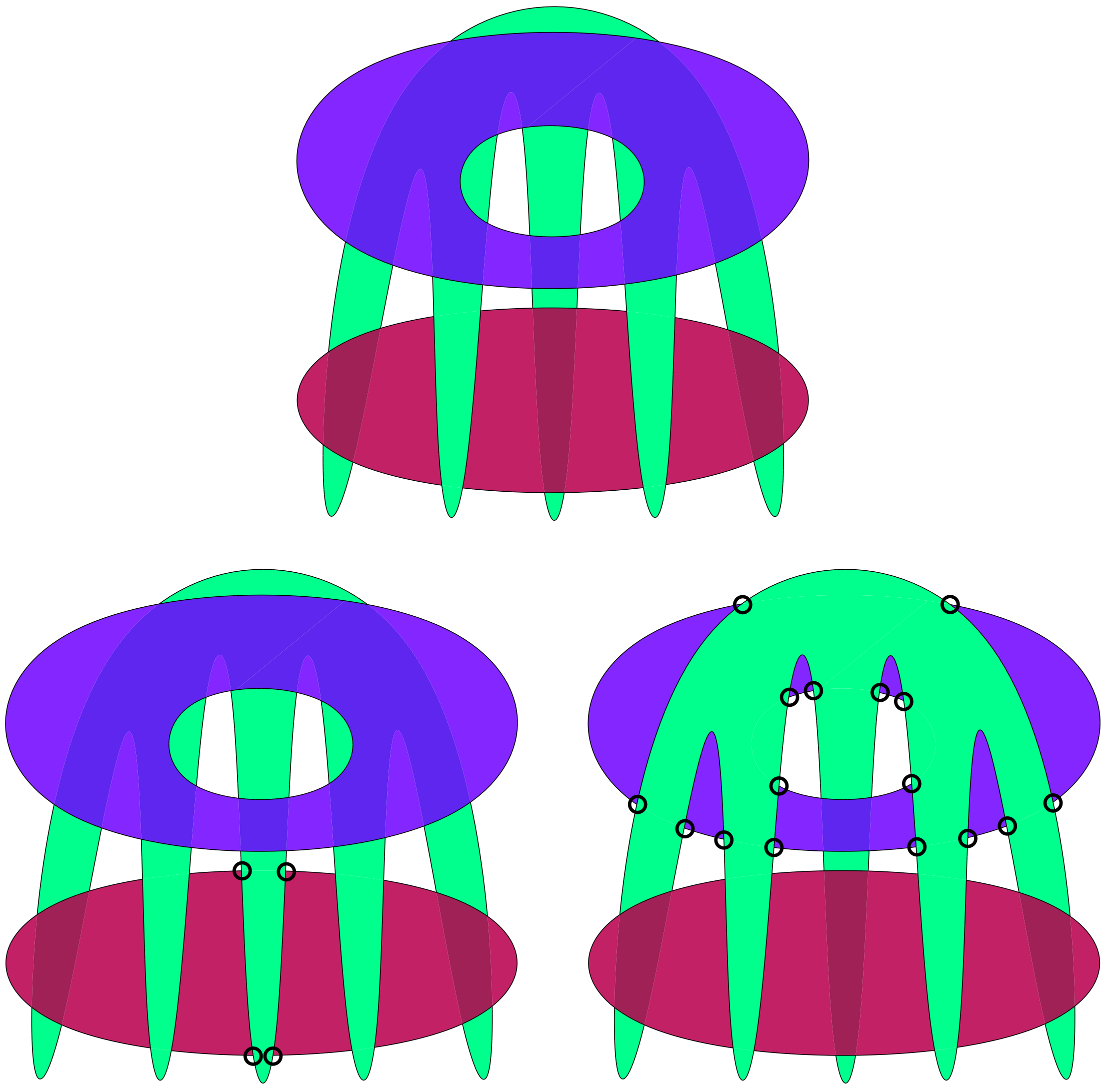

Figure 13.2: These figures show two equivalence class flips of two different equivalence classes applied to the same drawing. The original drawing is shown at top. The results of performing two different equivalence class flips are shown at bottom with the members of the flipped equivalence class marked with circles. In the bottom left figure, a fairly small equivalence class has been flipped. The equivalence class for the flipped shared superegion has four crossings. In the bottom right figure, a fairly large equivalence class has been flipped. The running time tests discussed in the text demonstrate that the benefit of exploiting equivalence classes is greater for larger equivalence classes.

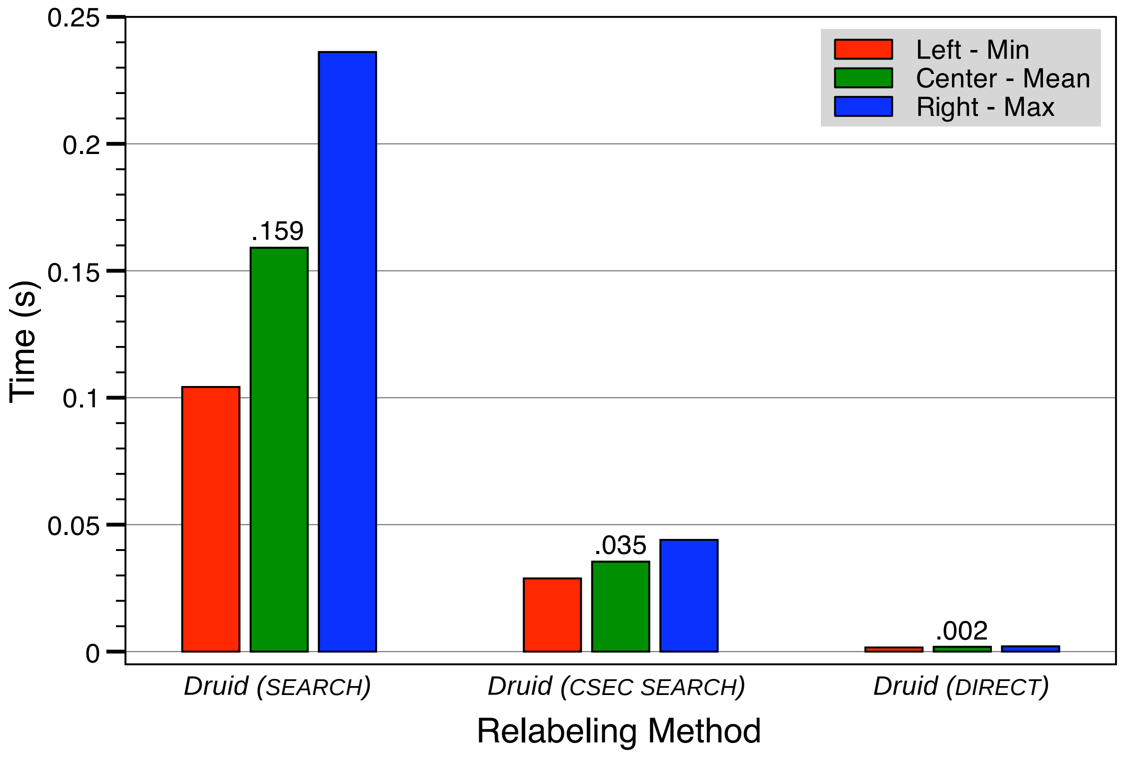

Figure 13.3: Running times for the three relabeling methods applied to the flip shown in Fig. 13.2 (bottom left).

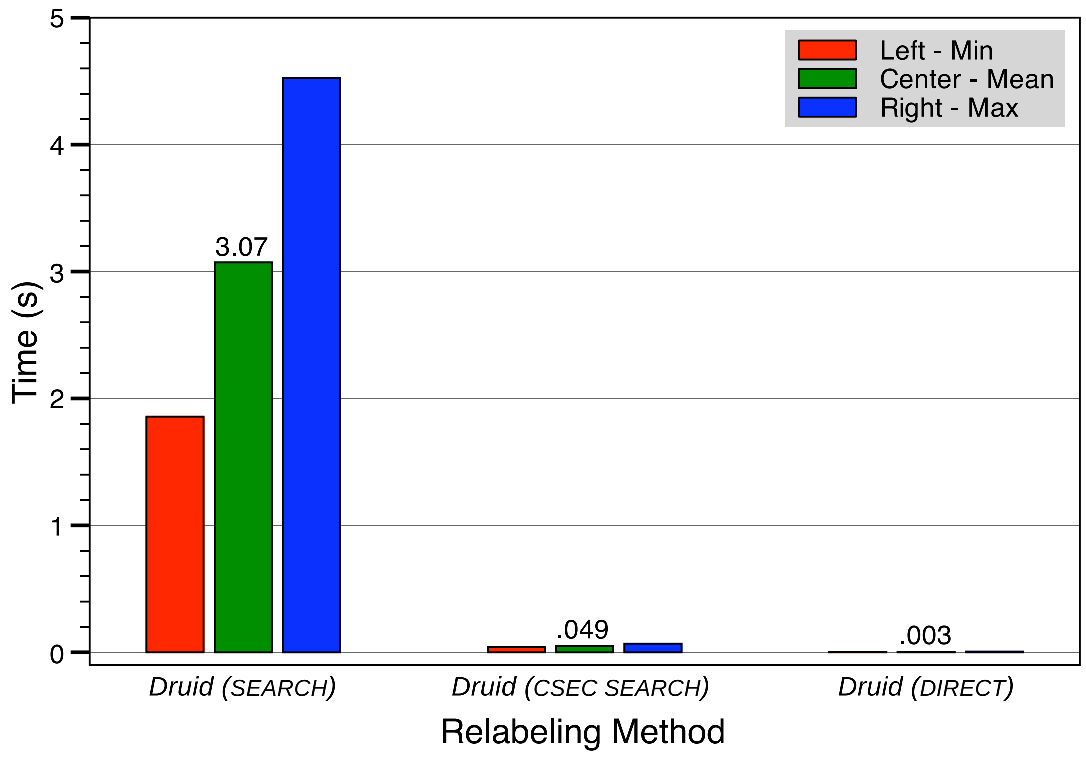

Figure 13.4: Running times for the three relabeling methods applied to the second flip shown in Fig. 13.2 (bottom right).

Flipping a Small Equivalence Class on a Complex Figure

Fig. 13.5 shows a drawing with many topologically identical equivalence classes, *i.e.*, many equivalence classes represent features that occur in multiple places in the drawing. One set of eight such equivalence classes is marked with circles in the right figure. The tests were performed by flipping each of the eight marked equivalence classes separately on the left figure and the resulting data was merged.

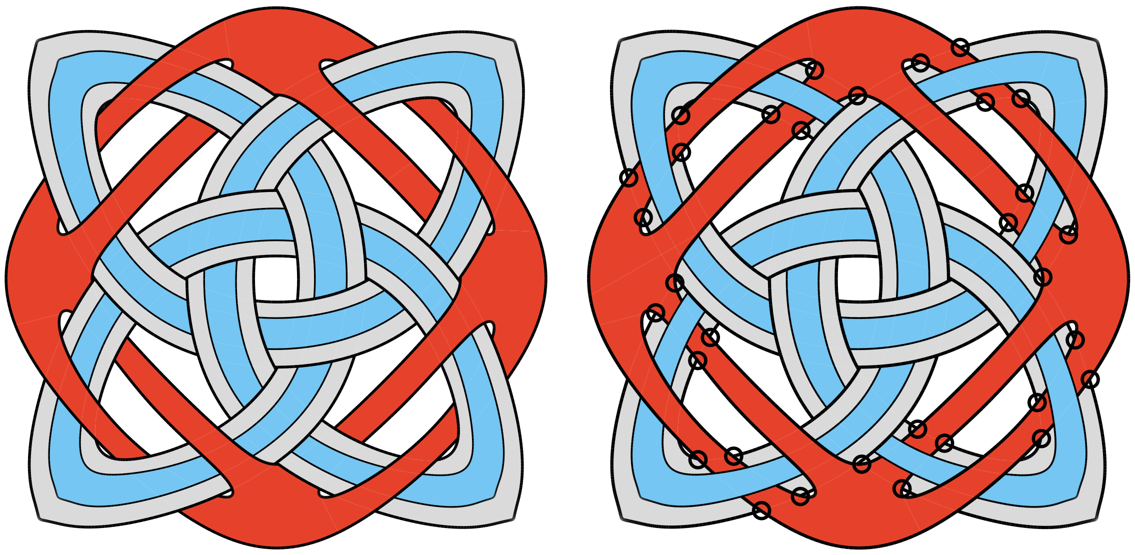
Figure 13.5: These figures show a drawing before (left) and after (right) a set of eight topologically identical equivalence class have been flipped, marked with circles. The plot shown in Fig. 13.6 represents a merge of the result of applying each of these flips separately to the left figure.

The plot shown in Fig. 13.6 shows the benefit of exploiting equivalence classes on the drawing shown in Fig. 13.5. We observe that *Druid (*SEARCH*)* does not exhibit acceptable turnaround times. In contrast, *Druid (*DIRECT*)* performed quite well with mean turnaround times about 1900 times faster than *Druid (*SEARCH*)*. Note that *Druid (*SEARCH*)* failed to find a solution within 120s in two percent of trials.

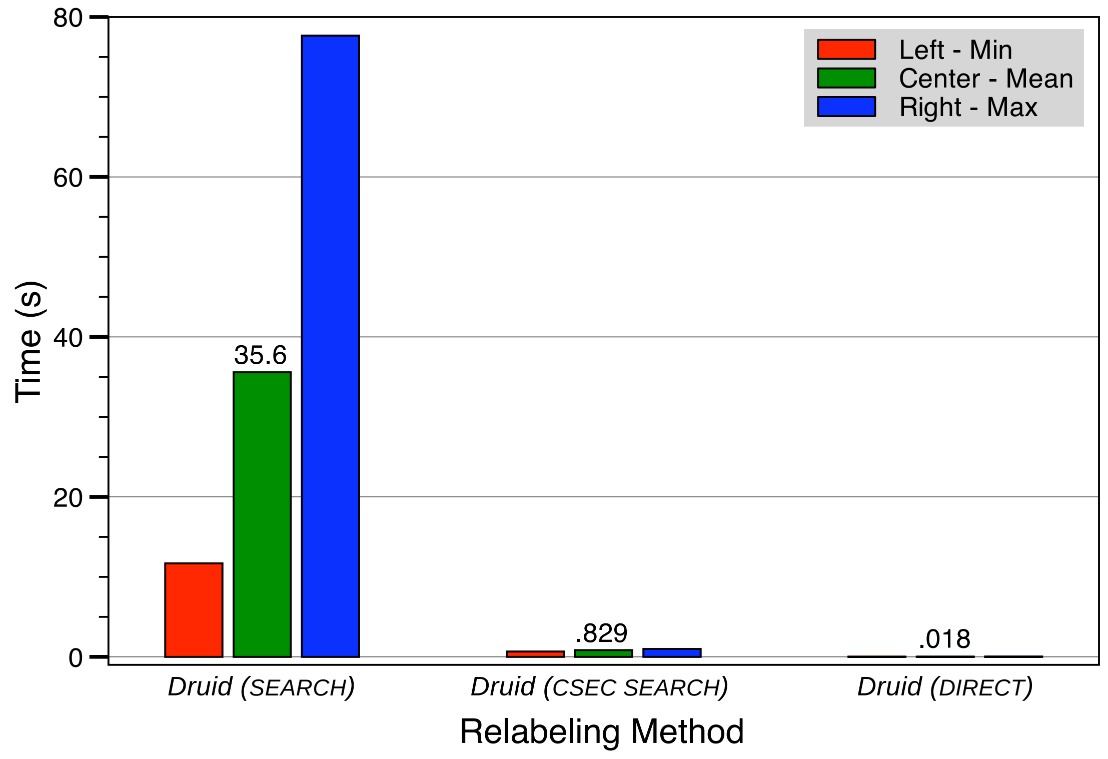

Figure 13.6: Running times for the three relabeling methods applied to the flips shown in Fig. 13.5. The data across all eight possible flips have been merged in this plot.

13.3.2 Performance Relative to Drawing Complexity

The experiment presented in this section illustrates the benefit of *Druid (*DIRECT*)* over *Druid (*SEARCH*)* relative to drawing complexity. Since *Druid (*DIRECT*)* consistently outperforms both *Druid (*SEARCH*)* and *Druid (*CSEC SEARCH*)*, it is used whenever possible, *i.e.*, following a crossing-flip interaction. In order to measure the performance relative to drawing complexity, we used the same set of drawings shown in Fig. 11.2. At each step, a flip of the single equivalence class in the figure was performed and its running time was measured relative to the complexity of the drawing. Figs. 13.7, 13.8, 13.9 show plots of the flip time relative to the equivalence class size for this experiment.

Based on the plots shown in Figs. 13.7 and 13.8, we observe that *Druid (*SEARCH*)* performs exponentially in the size of the equivalence class. Based on the plot shown in Fig. 13.9 we observe that *Druid (*DIRECT*)* performs exponentially in the size of the equivalence class as well. However, *Druid (*DIRECT*)'s* performance scales much better than *Druid (*SEARCH*)'s*, as the plot shown in Fig. 13.7 demonstrates. Most importantly, *Druid (*DIRECT*)* produces very fast actual turnaround times. It relabels a flip of an equivalence class that contains fifty-two crossings in .03s.

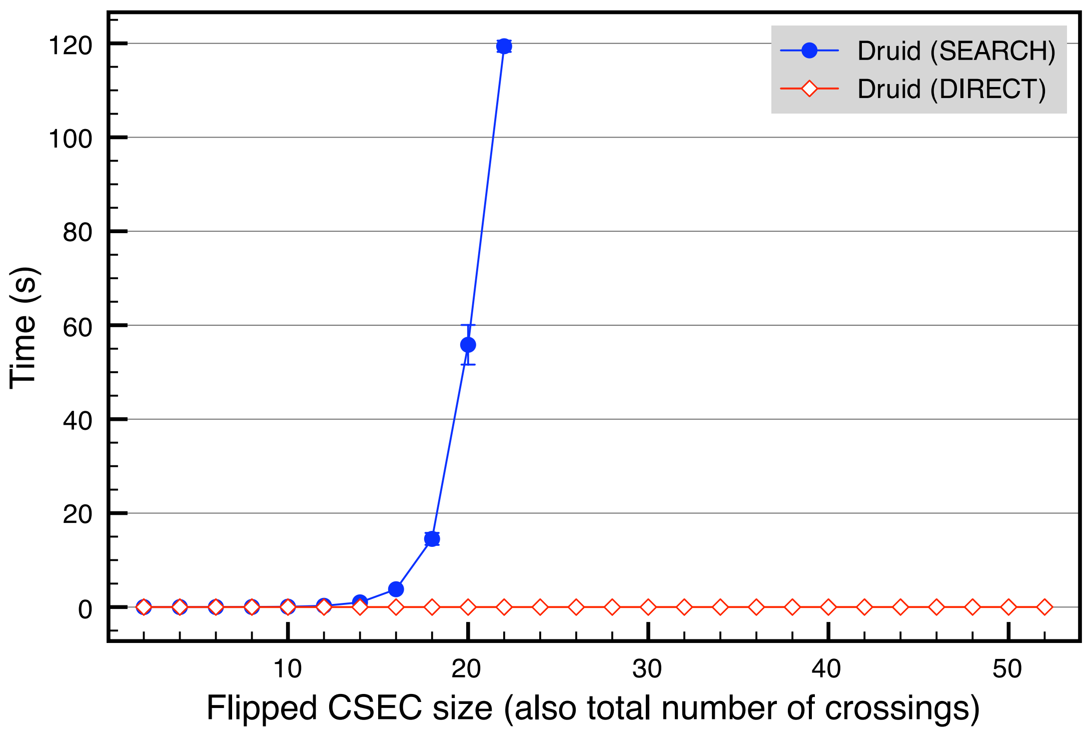

Figure 13.7: Crossing-State Equivalence Class Flip Time vs. Class Size. We observe that *Druid (*SEARCH*)* performs worse than linear relative to the equivalence class size. The performance of *Druid (*DIRECT*)* is difficult to analyze on this plot. Figs. 13.8 and 13.9 more precisely characterize the performance of each relabeling method. Errors bars show 95% confidence interval.

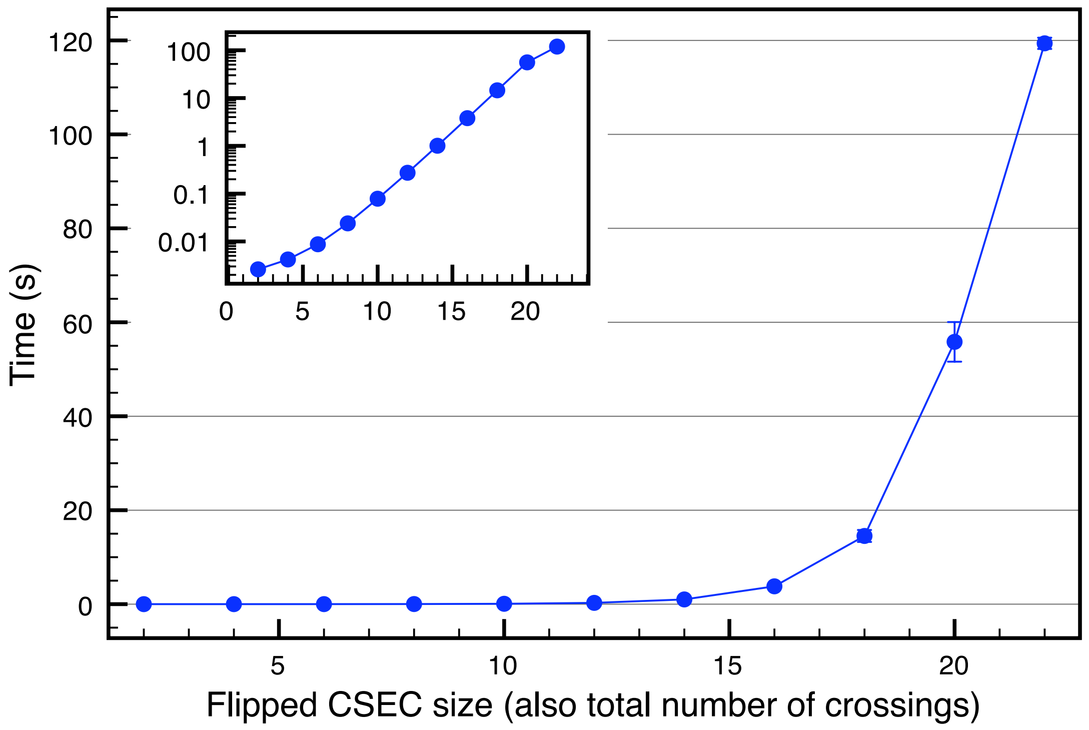

Figure 13.8: Crossing-State Equivalence Class Flip Time vs. Class Size for *Druid (*SEARCH*)*. The inset plot shows the same data plotted with a logarithmic y-axis. We observe that *Druid (*SEARCH*)* relabels the crossing-flips in this experiment exponentially in the size of the equivalence class.

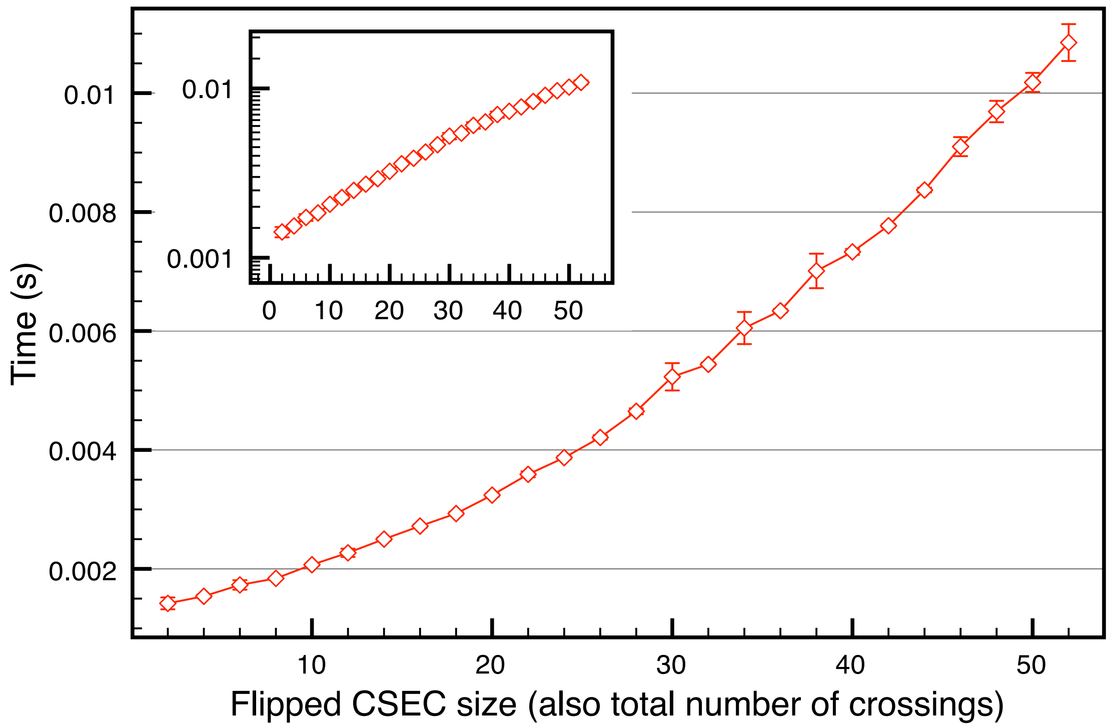

Figure 13.9: Crossing-State Equivalence Class Flip Time vs. Class Size for *Druid (*DIRECT*)*. The inset plot shows the same data plotted with a logarithmic y-axis. We observe that *Druid (*DIRECT*)* relabels the crossing-flips in this experiment exponentially in the size of the equivalence class. However, Fig. 13.7 shows that *Druid (*DIRECT*)'s* performance scales much better than *Druid (*SEARCH*)'s* performance and produces actual turnaround times that are well within our requirements for smooth performance. This plot demonstrates that *Druid (*DIRECT*)* is capable of relabeling a crossing-flip for an equivalence class of size fifty-two in .03s.

Chapter 14

Future Work

This document has presented the essential components of a new kind of drawing program called *Druid*. This program uses a representation called a labeled knot-diagram which naturally represents interwoven surfaces. In addition to developing *Druid's* representation, this document has described how *Druid* automatically maintains a legal labeling by searching through the space of possible labelings for a given figure. Finally, it has described a property of 2½D scenes called the crossing-state equivalence class rule which can be exploited to vastly improve *Druid's* performance. This concluded the description of *Druid's* representation and its core functionality.

The current version of *Druid*, *i.e.*, the version described up to this point, is a completed application. No features required for *Druid* to represent and manipulate 2½D scenes were omitted. However, there are a number of interesting directions in which this work could be extended. In this chapter we discuss three areas of possible future work:

- Labeling with crossing-state equivalence classes
- Locking crossings and kinematic interactions
- Occluding contours

14.1 Labeling with Crossing-State Equivalence Classes

In Section 13.1 we explained how *Druid* currently exploits crossing-state equivalence classes during the labeling search. Because equivalence classes can only be found on a legally labeled figure, *Druid* cannot use those equivalence classes to label the figure initially. At any given time, *Druid* knows the equivalence classes that were present in the drawing at the conclusion of the last labeling search but does not have knowledge of any equivalence classes created as a result of subsequent topological changes. However, even making limited use of equivalence classes during the labeling search is highly beneficial because doing so significantly reduces the size of the search space. The necessity of having a legal labeling before equivalence classes can be found can be problematic. For example, there are drawings for which the user must construct a complex configuration of surface boundaries which cannot be incrementally labeled because no intermediate configurations leading up to the goal configuration are legal. Consequently, *Druid* must label the final drawing without having any knowledge of the equivalence classes in advance. Fig. 13.5 shows such a drawing. This drawing consists of several boundaries including a pair of boundaries defining the blue band and a pair defining the gray band. Until these boundaries are placed in their final positions, the drawing cannot be legally labeled. Therefore, *Druid* must label the entire drawing without any knowledge of equivalence classes. Such a search can be prohibitively expensive because the search space may be very large. If *Druid* could find equivalence classes on an unlabeled figure, then it could fully apply equivalence classes to the labeling search, thus vastly decreasing the complexity of the search.

In Section 10.2.2 we described how holeless scenes can be converted into simple scenes by introducing cuts to split self-overlapping surfaces into sets of abutting nonself-overlapping surfaces. While this method of converting holeless scenes to simple scenes is valid in principal, we have yet to devise a practical method for finding the correct cuts to accomplish this transformation. A crucial step toward in design of a system which can find equivalence classes on an unlabeled figure is devising a method for finding the cuts which will convert a holeless scene into a simple scene. Our preliminary research on this conversion process suggests that such cuts might have to follow a curved path, whereas the cuts discussed previously in this document have been exclusively straight. Once a scene is simple, finding the equivalence classes is a fairly trivial matter.

14.2 Locking Crossings and Kinematic Interactions

Fig. 14.1 (top) shows a drawing of a clockwise boundary and a counter-clockwise boundary that have been grouped together into an annulus. The user can alter this drawing so that the two boundaries no longer bound the same surface (Fig. 14.1, bottom left). When the user attempts such a reshape, there are two behaviors that *Druid* could exhibit. The simpler behavior is to permit the interaction to continue, thus adding two new crossings, and attempting to find a new labeling in which the two boundaries no longer bound the same surface. An alternative behavior would *lock* the user's drag at the point where the two boundaries come into contact, thus preventing the two boundaries from being broken into two separate surfaces (Fig. 14.1, bottom right). Notice that either behavior might actually represent the user's intent. Furthermore, it is difficult to determine which behavior is actually desired in any given situation.

A second potential locking situation occurs when two different surfaces that are currently interlocked through holes or hooks (Fig. 14.2, top) are dragged past their interlocking position (Fig. 14.2, bottom left). *Druid* currently permits such interactions to occur and finds a new labeling for the resulting figure. Alternatively, the drag interaction could lock just prior to the point where the current labeling can no longer be maintained (Fig. 14.2, bottom right). There are at least two ways *Druid* could handle locked interlocking surfaces. The first way is for the drag to simply halt at the locked position. A second possibility is to treat *Druid's* idealized surfaces like physical surfaces subject to forces which propagate along kinematic chains formed by points of contact.

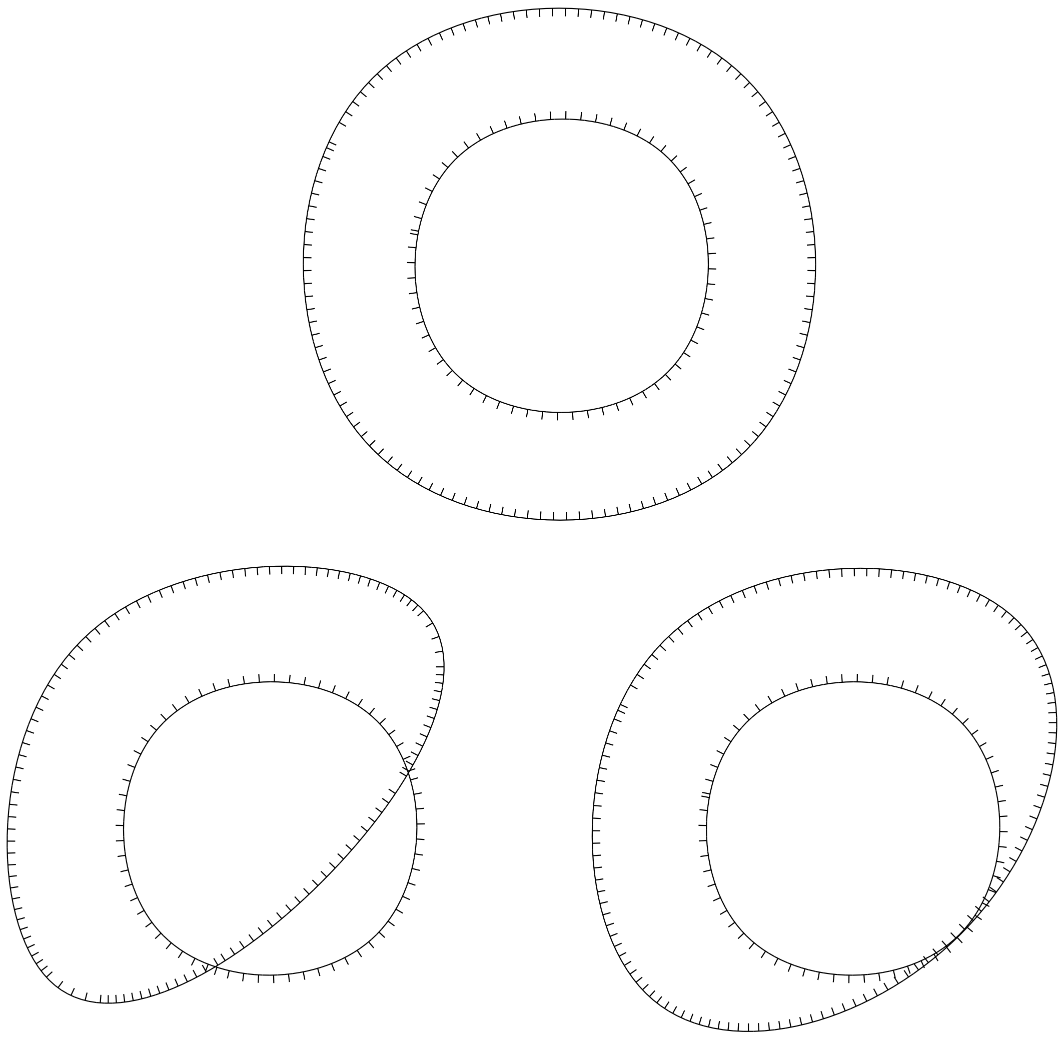

Figure 14.1: When two boundaries that currently bound the same surface (top) are altered so that they can longer bound the same surface (bottom left), *Druid* could *lock* the interaction at the point where the associated topological change occurs (bottom right), thus preserving the current labeling and the topology of the figure.

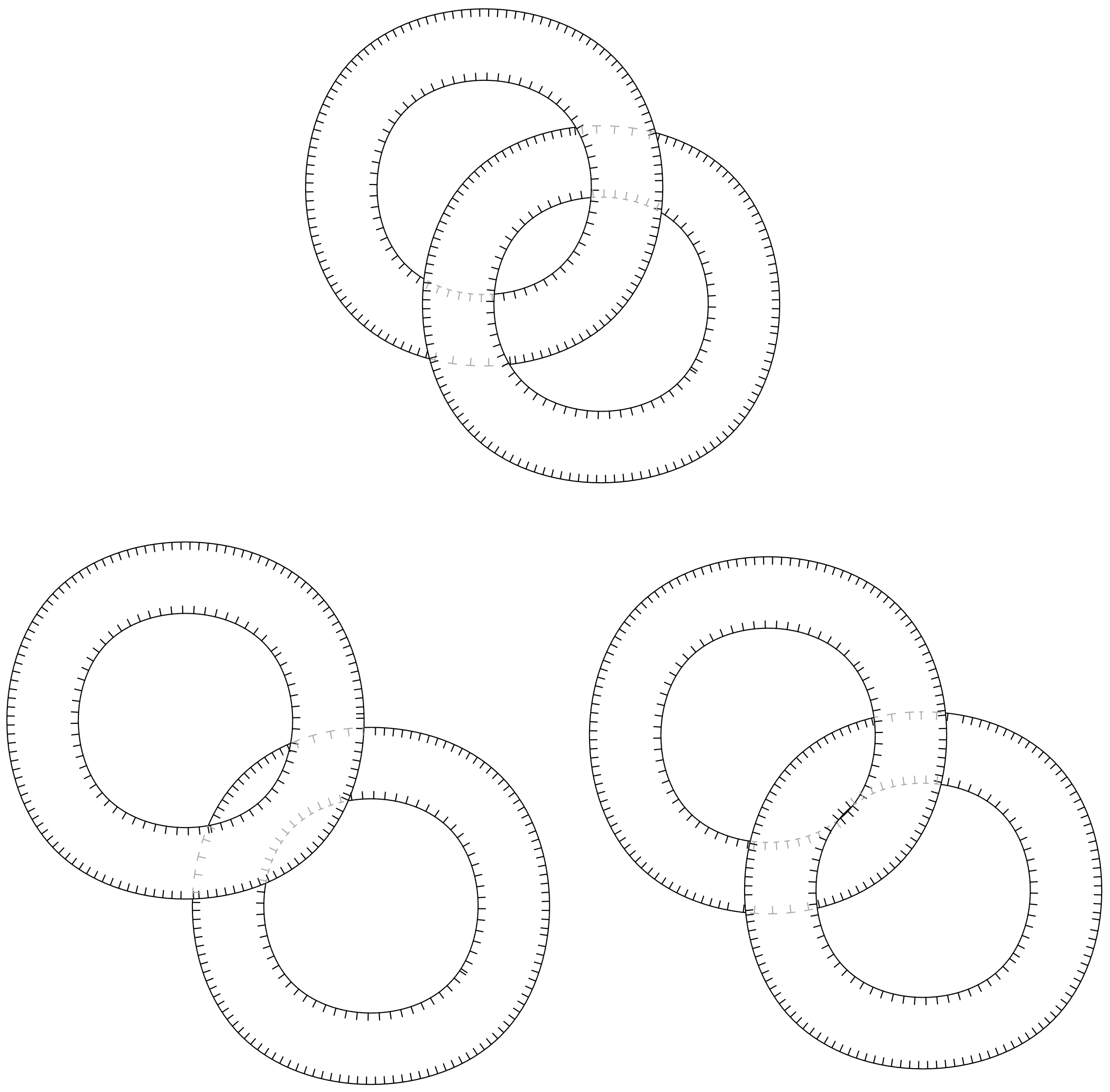

Figure 14.2: When two surfaces that are currently interlocked through holes (top) are altered so that they can longer maintain their interlocked configuration (bottom left), *Druid* could *lock* the interaction at the point where the associated topological change occurs (bottom right), thus preserving the current labeling and the topology of the figure. After the lock occurs, *Druid* could treat its idealized surfaces like physical surfaces subject to forces which propagate along kinematic chains formed by points of contact.

14.3 Occluding Contours

In the labeling scheme currently used by *Druid*, contours always indicate surface boundaries. This is the only type of contour *Druid* allows because we have assumed that 2½D scenes represent embeddings of surfaces in $\mathbb{R}^3$ which project onto $\mathbb{R}^2$ without singularity.[1] For this reason, all surfaces have been assumed to be effectively fronto-planar, *i.e.*, all normals of all surfaces are parallel to the viewing direction everywhere. If we permit curved surfaces, then surface normals can be oriented in arbitrary directions. The locus of points where surface normals are orthogonal to the viewing direction, *i.e.*, those points where tangent planes of the surfaces are viewed edge-on, form a new kind of contour (Fig. 14.3). This new kind of contour, called an *occluding contour*, is discussed in [20], [24], and [45]. Occluding contours have a sign of occlusion similar to that of ordinary boundaries which indicates which side of the occluding contour the surface resides on. However, in the case of an occluding contour, the surface resides at two distinct depths with respect to the *contour generator* (or *limb*).

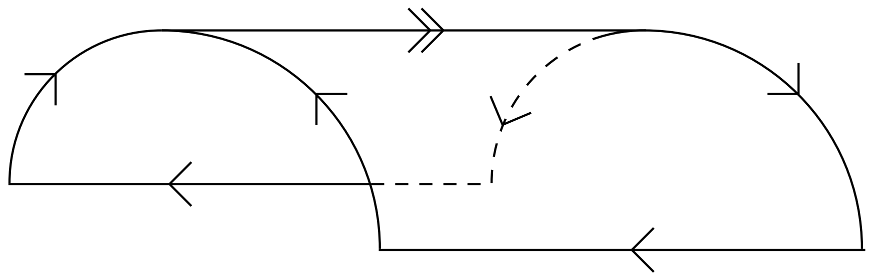

Figure 14.3: An *occluding contour* represents a locus of points where the surface normal is orthogonal to the viewing direction, *i.e.*, the tangent plane of the surface is viewed edge-on. Occluding contours have a sign of occlusion which designates which side of the boundary the surface lies on. Occluding contours are denoted with a double arrow that designates a curved surface to the right with respect to a traversal along the the *contour generator* (or *limb*).

Generalizing *Druid's* labeling scheme to include occluding contours would allow users

[1]A point in a projection of a surface embedded in $\mathbb{R}^3$ to its image in $\mathbb{R}^2$ is a *singularity* if two points from the same surface neighborhood project to a single point in the image.

of *Druid* to construct drawings representing a more general class of surfaces, *e.g.*, the drawing in Fig. 14.3 or a cylinder (Fig. 14.4, left). Additionally, occluding contours would permit the construction of scenes containing nonorientable surfaces, *e.g.*, the Mobius strip (Fig. 14.4, right).

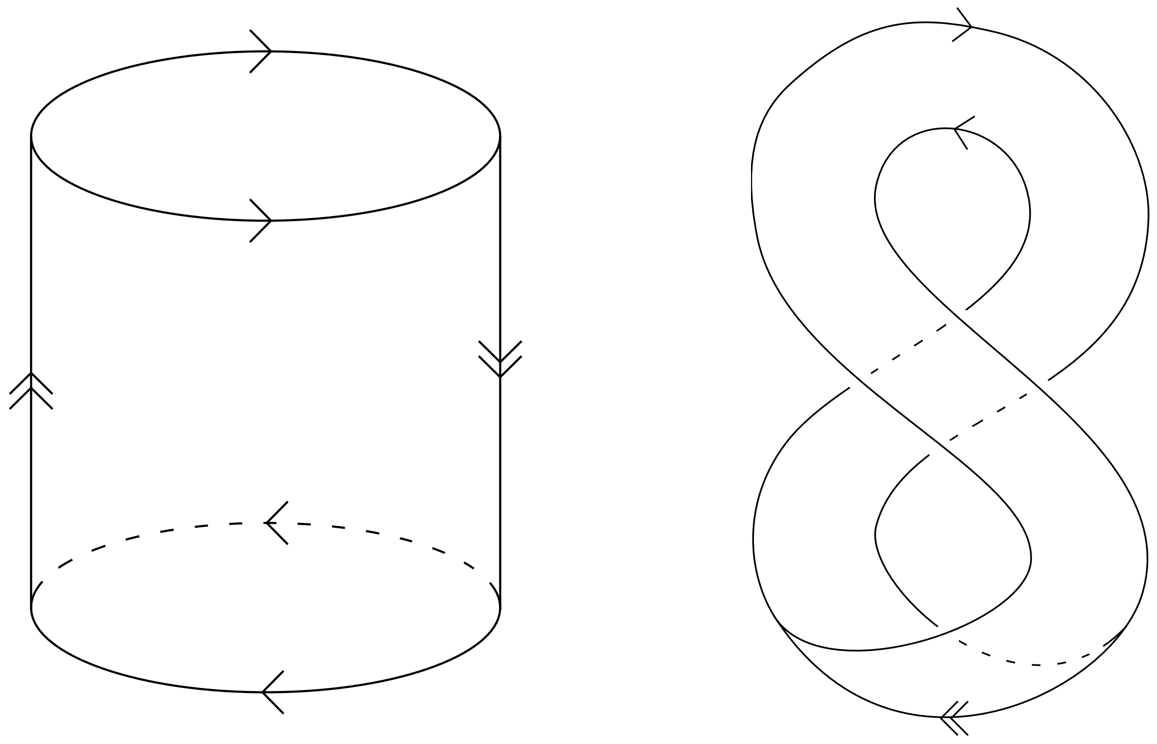
Figure 14.4: The inclusion of occluding contours in a labeling allows the construction of previously unrealizable scenes, such as cylinders (left). In addition, they allow the construction of scenes containing nonorientable surfaces such as the Mobius strip (right).

Occluding contours would also make possible construction of surfaces without boundary, *i.e.*, surfaces which themselves form the boundaries of three-dimensional solid objects, *i.e.*, three-manifolds with boundary. Consider a surface whose exterior contour consists of a single occluding contour of clockwise orientation. We call the surface defined by this contour a *container* (Fig. 14.5, *a*). A container can be a two-manifold *without* boundary (as shown in *a*). In contrast, *Druid* (to this point) has been restricted to two-manifolds *with* boundary. Other surfaces in the drawing can be assigned a depth that places them beneath the top layer of the container but above the lower layer of the container. Such surfaces would be inside the container (Fig. 14.5, *b*). If a hole were placed in the top layer of the container, *Druid* would reveal the surfaces contained within it (Fig. 14.5, *c*). Note that once a hole has been placed in a container it is no longer a two-manifold without boundary, it has become a two-manifold *with* boundary. For this reason, a container is not defined as a surface which is a two-manifold without boundary, but rather is defined as a surface whose exterior contour is a clockwise occluding contour. Containment can be ambiguous however. If the hole in the upper layer of the container (Fig. 14.5, *c*) were enlarged enough to surround one of the contained surfaces, it would be ambiguous whether the contained surface still resides inside the container or resides above it (*d*).

Adding occluding contours and containers to *Druid* raises a host of interesting questions. So far, boundaries have only been considered part of a group if a legal cut can connect them, thus designating that they bound a single surface. *Druid* treats these grouped boundaries in a consistent way, *i.e.*, when the user drags one boundary of a surface across the canvas, *Druid* automatically drags all other boundaries that are part of the same surface so that the surface does not change shape. If a container contains other surfaces, it is an open question or design issue whether the best behavior is to drag the contained surfaces together with the container which contains them. If the contained surfaces are not dragged with their container, then one possibility is for them to stay still until they collide with the occluding contour defining the posterior edge of the container. This collision would be similar to the simulation of the kinematic chain of interlocking annuli described in Section 14.2. Following this collision, the interior surfaces would be dragged along with the container, such that contained surfaces would tend to pile up along the posterior edge of the dragged container. An alternative (and quite likely preferable) behavior would require contained surfaces to always move as a unit within their container. This would preserve the relative positions of the surfaces in the drawing. However, this behavior would be sensitive to ambiguous situations such as the one described above (Fig. 14.5, *d*), in which it is ambiguous whether a surface is contained within a container that has had a hole cut in it.

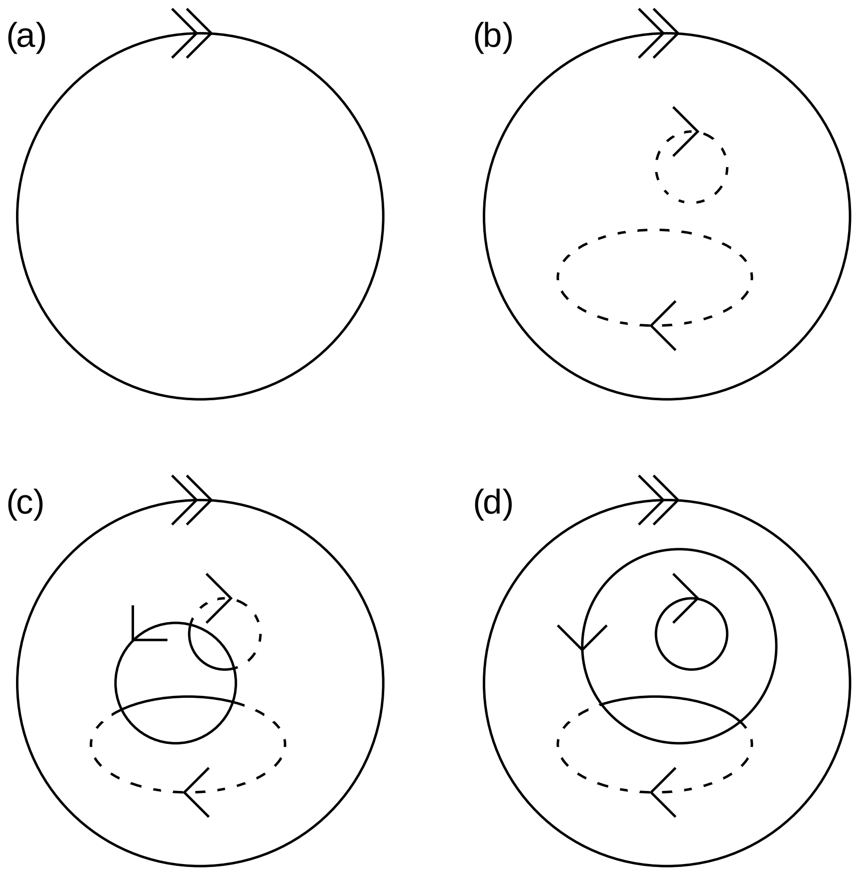

Figure 14.5: A surface defined by an exterior occluding contour is a *container* (*a*). Containers can contain other surfaces if those surfaces reside at a depth deeper than the upper layer of the container but shallower than the lower layer of the container (*b*). By placing holes in the upper layer, the user can view the contained surfaces (*c*). Containment can be ambiguous, however. This ambiguity is illustrated in (*d*), in which the smallest surface may reside within the container or may reside above it.

Another possible source of ambiguity is determining which boundaries and occluding contours belong to the same surface. Consider Fig. 14.5 (*b*, *c*, and *d*). Previously we described these figures as a container which contains two disks. In *c* and *d* a hole is cut in the top surface of the container. Another interpretation of these figures would perceive either of the clockwise boundaries, *e.g.*, the smallest boundary in the figure, not as a disk contained within the container, but instead as a hole in the *back* surface of the container.

Note that when a hole is placed in the back of a container it has a clockwise sign of occlusion whereas to this point holes have always been defined by a counter-clockwise sign of occlusion. Given an interpretation in which the smallest boundary is a hole in the back of the container, (*c*) and (*d*) represent figures in which a container has two holes in it, one partially (*c*) or completely (*d*) surrounding the other, thus permitting the viewer to view all the way through the container (like looking through a beach ball that has two holes cut into its opposite sides). To resolve this ambiguity we believe that an approach similar to cuts might be useful, in which various boundaries and occluding contours that belong to the same surface are connected to one another in some way. However, it is unclear precisely how such a system would work.

One important observation is that occluding contours can also represent creased surfaces rather than curved surfaces (see Fig. 14.6). With respect to a curved surface, the surface normal changes continuously and the occluding contour represents the locus of points where the tangent plane of the surface is viewed edge-on. However, in the case of a crease, the occluding contour represents the locus of points where the component of the surface normal in the viewing direction changes sign discontinuously. The contour generator represents a discontinuity in surface orientation, *i.e.*, the tangent plane is undefined. While these two types of occluding contours are distinct with respect to surfaces embedded in $\mathbb{R}^3$, they are topologically identical with respect to the projections of surfaces embedded in $\mathbb{R}^3$ onto $\mathbb{R}^2$, *i.e.*, they are topologically identical with respect to 2½D drawings. Indeed, in the *paneling construction* described in [45], occluding contours resulting from curved surfaces are converted into creases while maintaining the overall topology.

The inclusion of occluding contours requires a fairly elaborate extension to the basic labeling scheme (Fig. 14.7) which includes many new crossing types in addition to those already described (Figs. 4.2 and 8.2). As new kinds of contours are added to labeled knot-diagrams, the combinatorial complexity of crossing types and their associated labeling constraints grows quickly.

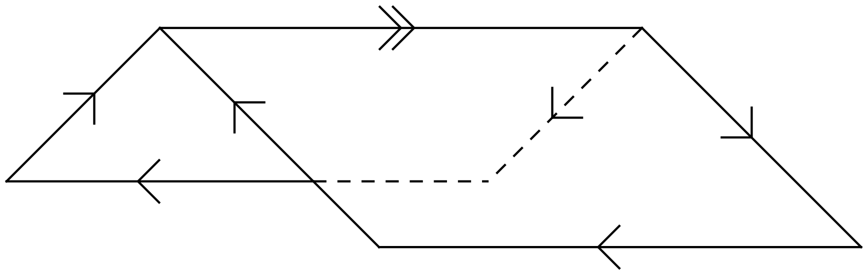

Figure 14.6: An *occluding contour* does not have to represent a curved surface, as shown in Fig. 14.3. It can also represent a crease, as shown above. While these two types of occluding contours are distinct with respect to three-dimensional space, they are topologically identical with respect to 2½D drawings.

This chapter has presented some topics of possible future work that could follow our work on *Druid*. As research on 2½D drawing progresses, drawing programs will likely become more powerful and intelligent, and they will offer representations and user interfaces of increasing generality, sophistication, and naturalness.

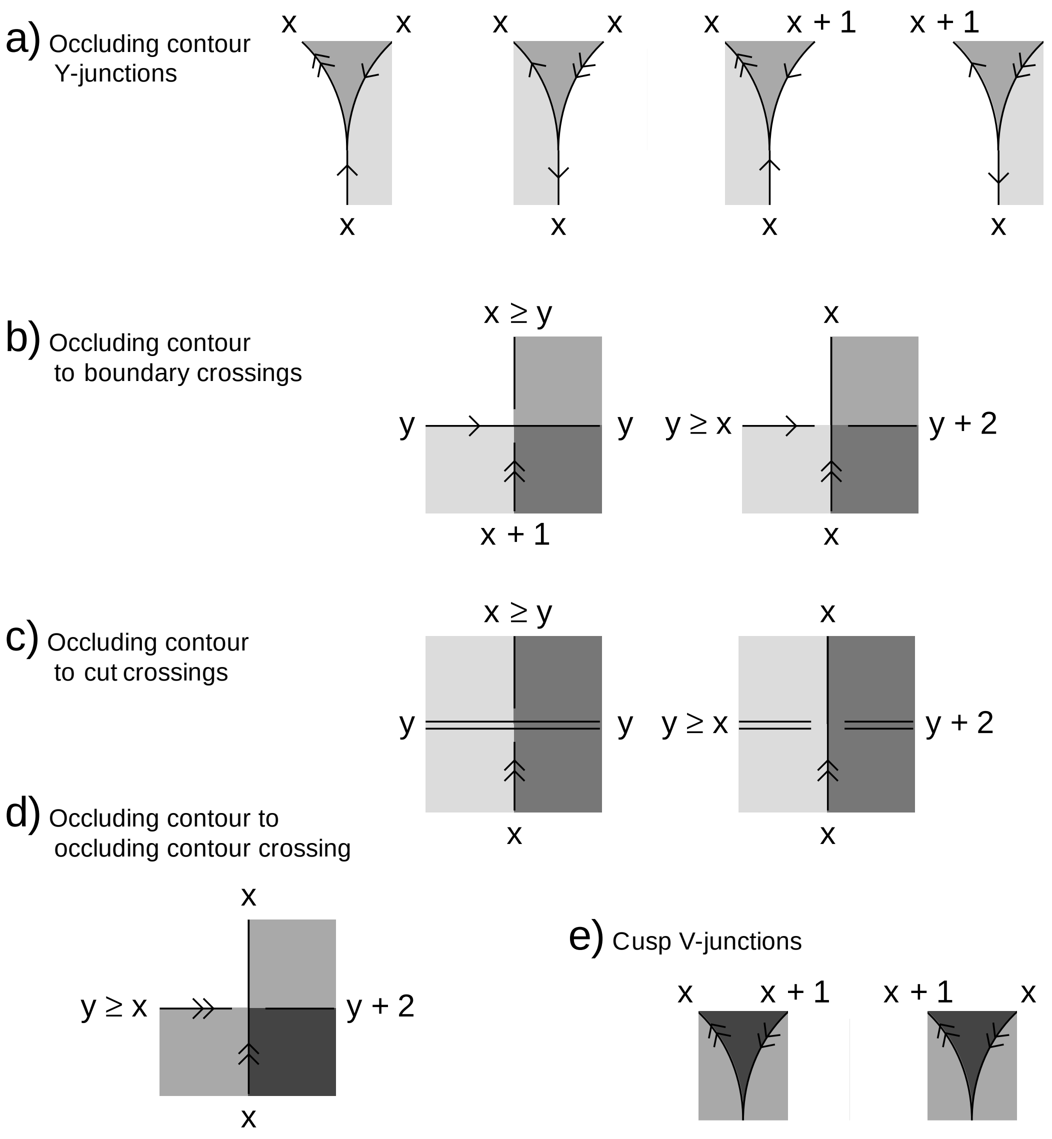
a) Occluding contour Y-junctions
x
x
x
x
x
x
x
x + 1
x
x + 1
x
x
b) Occluding contour to boundary crossings
x ≥ y
y
y
x + 1
x
y ≥ x
y + 2
x
c) Occluding contour to cut crossings
x ≥ y
y
y
x
x
y ≥ x
y + 2
x
d) Occluding contour to occluding contour crossing
x
y ≥ x
y + 2
x
e) Cusp V-junctions
x
x + 1
x + 1
x

Figure 14.7: Occluding contours require an elaborate extension to the basic labeling scheme. In addition to the crossing types shown in Fig. 4.2 and Fig. 8.2, the extended labeling scheme must also include the crossing types shown above.

Chapter 15

Conclusion

All drawing programs must have a way to distinguish which surface is on top anywhere that two surfaces overlap. Existing drawing programs solve this problem by assigning surfaces to distinct layers in depth. Consequently, interwoven sets of surfaces cannot be represented, thus precluding a large class of potential drawings. Since drawings should be able to depict any 2½D scene, a drawing program should use a representation that permits the construction of any 2½D scene. Unfortunately, the assumption that most existing drawing programs adopt is that surfaces reside in distinct layers. Since this assumption is not true of the space of all possible 2½D scenes, existing drawing programs cannot represent all 2½D scenes. We have developed an innovative new drawing program with the following major capabilities:

- Naturally represents a more general class of drawings than other programs, *i.e.*, drawings in which surfaces may interweave
- Provides user-interactions in the form of user specified constraints which are automatically propagated throughout the drawing to maintain topological validity of the representation.

Specific contributions of this work are as follows:

- Use of labeled knot-diagrams as the basis for a more general drawing tool capable of representing drawings of interwoven surfaces
- Development of a method for projection of the locations of crossings of surface boundary components after move and reshape interactions
- Development of a relaxation method for determining depth ranges for boundary segments in a labeled knot-diagram based representation
- Development of a branch-and-bound search method for efficiently finding minimum-difference labelings with respect to the labeling preceding a user action
- Introduction of the notion of cuts for representing surfaces with multiple boundary components and for reduction of the search space
- Introduction of the notion of slices for determining which surfaces contribute color to each region of the canvas for the purpose of rendering
- Discovery of a topological property of 2½D scenes which we call the crossing-state equivalence class rule
- Development of a relabeling method which exploits the crossing-state equivalence class rule to rapidly relabel a figure without a need for a labeling search.

Druid uses a novel surface representation which makes it possible to represent a more general class of drawings than is possible with existing drawing programs. *Druid* uses closed boundaries to represent surfaces. It only maintains local constraints on the ways in which boundaries can cross one another. This local constraint does not impose a global layering on the elements of the drawing and therefore permits the construction of scenes of interwoven surfaces.

Additionally, *Druid's* interface provides the natural affordances of 2½D scenes in that actions that the user performs are isomorphic to elemental transformations of 2½D scenes. Using *Druid* is easy because it operates in a way which is consistent with a user's intuition about real surfaces. Therefore, a user must learn relatively few new skills in order to start using *Druid*. *Druid's* affordances minimize the effort required of the user and decrease the time required to construct complex drawings.

Appendices

Appendix A

Relationship to Depth Sort

Depth sort is an algorithm commonly used to prepare for rendering in three-dimensional graphics applications. Given a set of polygons embedded in three-dimensional space, depth sort must assign an ordering to those polygons that reflects their relative positions in the depth-dimension with respect to the user's point of view. This depth-dimension sorting is an important step in rendering a three-dimensional scene because most polygon renderers use this ordering to subsequently prune occluded polygons, *i.e.*, hidden surface removal, and for rendering algorithms such as the painter's algorithm (see Angel [3]), which renders polygons in order from back to front so that shallower polygons are rendered on top of deeper polygons. There is a problem with sorting polygons along the depth-dimension however. In Section 10.2.2 we described a spectrum of drawing complexities (see Fig. 10.3). Polygons embedded in three-dimensional space correspond to simple scenes, *i.e.*, they might contain cyclic relative depth relationships between sets of polygons. The presence of a cyclic depth relation will prevent a partial ordering of the surfaces. Therefore, depth sort converts simple scenes to layered scenes by *splitting* some polygons into multiple abutting polygons such that the relative depth relation of the resulting set of polygons conforms to a DAG.

Druid does not need to perform the conversion from simple scenes to layered scenes and therefore does not need to utilize depth sort, because its representation is general enough to accommodate cyclic depth relationships, *i.e.*, *Druid* makes weaker assumptions about the arrangement of sets of surfaces than three-dimensional rendering tools make, such as the algorithms employed by graphics cards. *Druid* maintains a holeless representation and does not convert to a more tightly constrained representation for any of its methods. However, there exists a potential for great benefit in converting *Druid's* representation to a layered scene. The benefit is that *Druid's* rendering process could be implemented using a special purpose graphics card. This would free the primary CPU for other processing and *Druid's* performance might improve as a result. In our research, we have not attempted this conversion however, as it is somewhat orthogonal to our main research goal, the development of *Druid's* representation and the necessary methods for constructing and manipulating instances of that representation.

Appendix B

Description of Data Structures and Basic Organization

The purpose of this appendix is to more easily facilitate *Druid's* reimplementation. While this entire document suffices to describe *Druid* and the operations it performs, it may be confusing how one should organize the data structures involved in a reimplementation of *Druid*. This appendix describes the crucial data structures that *Druid* operates on but does not describe the operations that are performed on those data structures. Descriptions of the operations which *Druid* performs can be found in earlier chapters.

B.1 The Fundamental Drawing Program

A programmer who is knowledgeable in GUIs[1] should be able to implement a *fundamental drawing program* that can be expanded into *Druid*. A fundamental drawing program is one that presents a window to the user, detects and handles mouse events, interpolates curves for visualization on the computer's display, and otherwise generally permits the

[1]"graphical user interface"

construction of closed curves representing surface boundaries. We will assume that such a fundamental drawing program can be easily implemented for further expansion to *Druid*. In the next section we describe the data structures that *Druid* uses represent a $2^1/_2$D scene.

B.2 The Crucial Data Structures

Druid uses the following data structures:

- drawing
- general boundary
  - B-spline boundary
- surface
- cut
- cached cut
- general crossing
  - boundary-to-boundary crossing
  - boundary-to-cut crossing
  - cut-to-cut crossing
  - boundary-to-cut-T-junction crossing
- crossing-state equivalence class
- slice
- region

The data structure that contains all data describing a drawing is the *drawing structure*, which contains all of the boundaries, cuts, and crossings comprising a drawing. A drawing also contains cached cuts (see Section 8.3.2), crossing-state equivalence classes, and surfaces. The following is a basic description of the drawing structure (note that crossing-state equivalence class is shortened to CSEC):

DRAWING

vector<**General Boundary**>	boundaries
vector<**Cut**>	cuts
vector<**General Crossing**>	crossings
vector<**Cached Cut**>	cachedCuts
vector<**CSEC**>	csecs
vector<**Surface**>	surfaces

Although a boundary may *represent* a curve, *Druid* never uses a truly curved boundary. Instead it uses a boundary defined by a polygon, *i.e.*, a closed sequence of straight line segments. If those line segments are small enough and numerous enough, the polygon can reasonably approximate a curve. *Druid* uses straight line segments to define boundaries because this vastly simplifies the problem of finding crossings. The boundary can be defined in any way the programmer chooses, *e.g.*, splines, ellipses, polygons, or freehand drawn curves, but the underlying representation is still a polygon. It is important to distinguish between polygon segments and boundary segments. To simplify our exposition, we use *polygon segment* to refer to the line segments defining a boundary and *boundary segment* to refer to a contiguous portion of a boundary between crossings. Since a boundary's polygon can be defined using many different methods, we first describe the *general boundary structure* which is independent of the method used to define the boundary's polygon. A general boundary contains a sequence of points indicating the corners of a polygon. A general boundary also stores information about its boundary segments, namely each segment's current depth index and maximum attainable depth. The maximum attainable depth is necessary for segment depth enumeration (see Section 7.3). A general boundary also stores the set of crossings that occur on the boundary, preferably sorted with respect to a traversal of the boundary. Note that it is not necessary to explicitly store the boundary's sign of occlusion, which is determined by traversing the sequence of polygon corners in order. To invert the sign of occlusion, the sequence of polygon corners is simply reversed. The following is a basic description of the general boundary structure:

General Boundary

vector<point>	polygonCorners
vector<int>	segmentDepths
vector<int>	maxSegmentDepths
vector<General Crossing>	crossings

Since we use B-splines to define boundaries, the specific boundary structure *Druid* uses also includes a set of control points defining the B-splines that comprise a boundary. In an object-oriented framework, specific boundary definitions can be derived from the general boundary defined above. The following describes the *B-spline boundary structure*:

B-spline Boundary : derived from General Boundary

vector<point>	controlPoints

A *surface structure* stores information about the multiple boundaries that bound a surface. Therefore, a surface structure stores a set of related boundaries. Additionally, if a surface is infinite, the surface structure stores the infinitely distant depths the surface resides at. Note that it is possible for an infinite surface to reside at multiple infinite depths. Finally, a surface stores a color and opacity that describe the surface's appearance:

Surface

vector<General Boundary>	boundaries
vector<int>	infiniteDistanceDepths
real[3]	RGB color
real	opacity (between 0.0 and 1.0)

A *cut structure* stores information about the two boundaries joined by a cut. This information includes which two boundaries are joined and the locations on those boundaries where the cut attaches. For simplicity, cuts only attach to the midpoints of polygon segments, although this limitation could be generalized in future implementations. Therefore, a cut's attachment point is easily indicated by the segment index to which it attaches. Cuts have segments similar to boundary segments, and those segments have similar properties, *i.e.*, a depth index and a maximum depth. The cut structure also stores a list of its crossings, much like a boundary:

CUT

General Boundary	startBoundary
General Boundary	endBoundary
int	startAttachmentPolygonSegment
int	endAttachmentPolygonSegment
vector<**int**>	segmentDepths
vector<**int**>	maxSegmentDepths
vector<**General Crossing**>	crossings

A data structure that is related to cuts is the *cached cut structure* (see Section 8.3.2). A cached cut structure contains only the information necessary to construct a cut with specific attachment points to a specific pair of boundaries. Other data incidental to a cut, such as the cut's segments and crossings, can be recalculated after the the cut has been constructed:

CACHED CUT

General Boundary	startBoundary
General Boundary	endBoundary
int	startAttachmentPolygonSegment
int	endAttachmentPolygonSegment

A crossing stores information about the way in which pairs of boundaries, pairs of cuts, or boundaries and cuts cross one another. It is important to realize that each crossing type requires a unique data structure. We first describe a *general cut structure*, from which the specific cut structures are derived. Notice that while a cut's coordinate can always be calculated when needed from the boundaries or cuts it involves, it is more efficient to store the coordinate in the cut structure because it often needed for various operations, *e.g.*, defining the border of a region. A crossing's state may be *constrained* during the search process, which indicates that the subsequent search is not permitted to flip the crossing. Therefore, this constraint is stored in the crossing structure. Note that the T-junction crossing makes no use of the *state* or *constrained* variables in the general crossing structure since T-junctions have only one possible state.

GENERAL CROSSING

Point	location
bool	state
bool	constrained

BOUNDARY-TO-BOUNDARY CROSSING : DERIVED FROM GENERAL CROSSING

General Boundary	boundary1
General Boundary	boundary2
int	polygonSegment1
int	polygonSegment2

BOUNDARY-TO-CUT CROSSING : DERIVED FROM GENERAL CROSSING

General Boundary	boundary
Cut	cut
int	polygonSegment (on boundary)

CUT-TO-CUT CROSSING : DERIVED FROM GENERAL CROSSING

Cut	cut1
Cut	cut2

BOUNDARY-TO-CUT-T-JUNCTION CROSSING : DERIVED FROM GENERAL CROSSING

General Boundary	boundary
Cut	cut
int	polygonSegment (on boundary)
bool	atCutStart (a crossing at the cut's start or end?)

The *crossing-state equivalence class structure* describes a crossing-state equivalence class. This information basically consists of a set of crossings comprising the equivalence class:

CSEC

vector<**General Crossing**>	crossings

Slices are used during rendering. The *slice structure* is similar to the cut structure, except that it only stores an attachment to one boundary. The other end of the slice is located somewhere in the interior of the bounded surface and is therefore stored as a coordinate:

SLICE

General Boundary	boundary
int	attachmentPolygonSegment
point	sliceOrigin
vector<**int**>	segmentDepths
vector<**int**>	maxSegmentDepths
vector<**General Crossing**>	crossings

One of the fundamental data structures used during rendering is the *region structure*, which describes a region (a polygon) of the canvas and the surfaces that cover it, sorted in order by depth. Call the region in question *A*. The region structure for *A* also contains a list of other regions which are suspected to enclose *A*, and which must therefore be rendered prior to rendering *A* in the rendering process (see Section 9.1.3). Likewise, a region has a boolean which indicates whether it has been rendered yet. This boolean, say for region *A*, is queried by other regions whose suspected enclosing regions lists contain *A*. Finally, a region stores the color assigned to it, as calculated using a color model applied to the covering surfaces:

REGION

vector<point>	polygonCorners
vector<Surface>	coveringSurfaces (sorted by increasing depth)
vector<Region>	suspectedEnclosingRegions
bool	hasBeenRendered
real[3]	RGB color

The data structures described in this appendix should be helpful in the design and reimplementation of *Druid*. Descriptions of the various operations that are performed on these data structures are described throughout this document, *e.g.*, the labeling search (see Chapter 7), calculating the depth ranges for the boundary segments (see Section 5.3), crossing-projection (see Section 6.2.1), the cut search (see Section 8.3), the equivalence class search (see Chapter 11), the direct crossing-flip (see Section 13.2), and rendering (see Chapter 9).

References

[1] Knots3D, ©2006 Abbott, S.
http://www.abbott.demon.co.uk/knots.html

[2] Adobe Illustrator, ©2006 Adobe.
http://www.adobe.com/

[3] Angel, E. *Interactive Computer Graphics*. Addison-Wesley, 2006.

[4] Apple Developer Connection (ADC) reference library
http://developer.apple.com/

[5] SymmetryWorks Adobe Illustrator plugin, ©2006 Artlandia.
http://artlandia.com/products/SymmetryWorks/

[6] Autodesk and Alias Maya, ©2006 Autodesk.
http://usa.autodesk.com

[7] Autodesk 3ds Max, ©2006 Autodesk.
http://usa.autodesk.com

[8] Barla, P., J. Thollot, and F. Sillion, Geometric clustering for line drawing simplification, *Siggraph Technical Sketch: SIGGRAPH 2005*, ACM, 2005.

[9] Baudelaire, P., and M. Gangnet, Planar maps: An interaction paradigm for graphic design, *Proc. of CHI*, 1989.

[10] Brody, B., and C. Hartman, BLUI, a body language user interface for 3d gestural drawing, Report to Photonics West, 1999.
http://www.blui.org/papers/spiepaper.html

[11] Bungie's 3D games, Myth I, ©1997 Bungie, and Myth II, ©2000 Bungie.
http://www.bungie.net/

[12] Clanbadge's True Type fonts which represent square sections of a Celtic knotwork pattern, ©Clanbadge 2006.
http://www.publishingperfection.com/clanbadge/

[13] Coreldraw graphics suite upgrade matrix, 2003.
http://www.corel.com/content/pdf/cdgs12/CDGS_Version_to_Version_matrix.pdf

[14] Craig, D., LisaDraw 3.0 Manual, 1984.

[15] Cromwell, P. R. Celtic knotwork: Mathematical art, *The Mathematical Intelligencer*, **15** (1), pp. 36-47, 1993.

[16] Gangnet, M., J-M. Thong, and J-D. Fekete. Automatic gap closing for freehand drawing. *Siggraph Technical Sketch: SIGGRAPH 1994*, ACM, 1994.

[17] Gibson, J. J., *The ecological approach to visual perception*, Houghton Mifflin Co., Boston, MA, 1979.

[18] Gleicher, M., Briar: A constraint-based drawing program, *Proc. of CHI*, 1992.

[19] Gleicher, M., and A. Witkin, Differential manipulation, *Proc. of Graphics Interface*, Calgary, Alberta, pp. 61-67, 1991.

[20] Huffman, D. A., Impossible objects as nonsense sentences, *Machine Intelligence*, **6**, 1971.

[21] ivtools team, idraw man page.
http://www.ivtools.org/ivtools/idraw-README.txt

[22] Kirousis, L. M., and C. H. Papadimitriou, The complexity of recognizing polyhedral scenes, *Journal of Computer and System Sciences*, **37** (1) pp. 14-38, 1988.

[23] MacPowerUser team, iDraw 1.3.2 README, 2002. Available as part of the downloadable iDraw package.
http://www.macpoweruser.com/downloads.html

[24] Malik, J., Interpreting line drawings of curved objects, *International Journal of Computer Vision*, **1** (1), pp. 73-103, 1987.

[25] Marr, D. *Vision: A computational investigation into the human representation and processing of visual information*, Henry Holt & Company, 1982.

[26] Marsland, T. A., and M. Campbell, Parallel search of strongly ordered game trees, *ACM Computing Surveys*, **14** (4), pp. 533-551, 1982.

[27] McReynolds, T., and D. Blythe, Advanced Graphics Programming Techniques Using OpenGL, *Siggraph Course Notes: SIGGGRAPH 1998*, ACM, 1998.

[28] McGrenere, J., and W. Ho, Affordances: Clarifying and evolving a concept, *Graphics Interface*, pp. 179-186, May 2000.

[29] MetaCreations's Bryce 2, ©1996 MetaCreations.
http://www.metacreations.com/products/

[30] Metelli, F., The perception of transparency, *Scientific American*, **230** (4), pp. 90-98, 1974.

[31] Metelli, F., Stimulation and perception of transparency, *Psychological Research*, **47** (4), pp. 185-202, 1985.

[32] Myers, B. A., A brief history of human computer interaction technology, *ACM Interactions*, **5** (2), pp. 44-54, 1998.

[33] Norman, D. A., Affordance, conventions, and design, *Interactions*, pp. 38-43, 1999.

[34] Norman, D. A., *The Design of Everyday Things*, Basic Books, 2002.

[35] Raisamo, R., and K-J Räihä, Techniques for aligning objects in drawing programs, Technical Report, University of Tampere, Department of Computer Science, A-1996-5, 1996.

[36] Raisamo, R., and K-J Räihä, A new direct manipulation technique for aligning objects in drawing programs, *ACM Symposium on User Interface Software and Technology*, pp. 157-164, 1996.

[37] Raisamo, R., An alternative way of drawing, *Proc. of CHI*, 1999.

[38] Sato, T., and B. Smith, Xfig User Manual, 2002.
http://xfig.org/userman/

[39] Scharein, R. G. *Interactive Topological Drawing*. Ph.D. dissertation, University of British Columbia, 1998.

[40] Sutherland, I. E., Sketchpad: A man-machine graphical communication system, *Proc. of the 1963 Spring Joint Computer Conference, AFIPS*, **23** pp. 329-346, 1963.

[41] Sutherland, I. E., Sketchpad: A man-machine graphical communication system, Technical Report, Univ. of Cambridge, UCAM-CL-TR-574, Sept, 2003. (This technical report is a modern republication of Sutherland's 1963 doctoral dissertation.)

[42] Voska, R., Real-Draw Manual, pp. 67-72, 2003.
http://www.mediachance.com/files/RealDrawPDF.zip

[43] Waltz, D. L., Understanding line drawings of scenes with shadows, *The Psychology of Computer Vision*, McGraw-Hill, New York, pp. 19-92, 1975.

[44] Williams, L. R., *Perceptual completion of occluded surfaces*, Ph.D. dissertation, Univ. of Massachusetts at Amherst, Amherst, MA, 1994.

[45] Williams, L. R., Topological reconstruction of a smooth manifold-solid from its occluding contour, *International Journal of Computer Vision*, **23** (1), pp. 93-108, 1997.

[46] Celtic Knot Thingy (CKT), ©2006 Zongker, D.
http://isotropic.org/uw/knot/

Index