Representation of Interwoven Surfaces in $2\frac{1}{2}$D Drawing

Keith Wiley Lance R. Williams

Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131

September 9, 2005

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 $2\frac{1}{2}$D drawing from top to bottom. A $2\frac{1}{2}$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.

In this paper we describe 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.

1 Introduction

Drawing programs originated with Sutherland's seminal PhD thesis in 1963, in which many recognizable components of modern drawing programs were already present (see Sutherland [18]). Since then, a number of refinements have been made to the general design of drawing programs, aided primarily by hardware innovations such as the mouse. In 1984, Apple released *LisaDraw 3.0* (see Craig [2]), which despite its age, is effectively a modern drawing program. For the last twenty years, research on drawing programs has focused on refinements to the existing approach rather than the development of entirely new approaches. For example, *CorelDRAW 12*, a professional drawing program, boasts features such as a *Smart Drawing tool*, which allows a user to freehand draw approximate shapes that are recognized and fitted to stock shapes such as ellipses, and *Dynamic Guides*, which are temporary guides for aligning graphic objects to one another (see Corel [1]). These changes are incremental extensions of the standard approach and do not represent quantum leaps in capability.

One function of a drawing program is to allow the creation and manipulation of drawings of overlapping surfaces, which we simply call *$2\frac{1}{2}$D scenes*. A $2\frac{1}{2}$D 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. Using existing programs, a drawing can easily be created in which multiple surfaces overlap in various *subregions* (a contiguous area of a surface). When multiple surfaces overlap, the program must have a means of representing which surface is on top for each overlapping pair of subregions. 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). This restriction precludes many common drawings which a user may wish to construct (Fig. 2). Because such programs do not span the full space of possible $2\frac{1}{2}$D 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 $2\frac{1}{2}$D scenes by using a representation that makes weaker assumptions about the drawing. This generality makes *Druid* a more versatile drawing tool.

2 Drawing Program User-Interactions

Many drawing programs provide tools for creating and editing splines. One such program, *MacPowerUser's iDraw*, [8], bases all objects on splines. Rectangles, polygons, even text are all represented using splines. This provides a consistent interface for editing the various kinds of objects in *iDraw*. Similarly, *Druid* is based entirely on splines. Like [7] and [17], *Druid* uses B-splines. We chose to use B-splines instead of Bezier splines because they are simpler to implement. All boundaries are B-splines, including shapes that may be approximated by splines, such as hand-drawn curves, rectangles, and text. The assumption that all boundaries are splines lends a uniformity to a

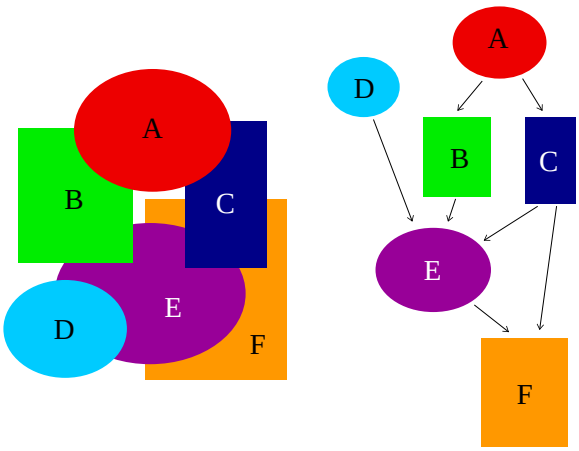

Figure 1: The classic approach to representing relative surface depths is to assign the surfaces to distinct layers. This implicitly imposes a DAG on the surfaces such that no subset of surfaces can interweave because this would require a cycle in the graph.

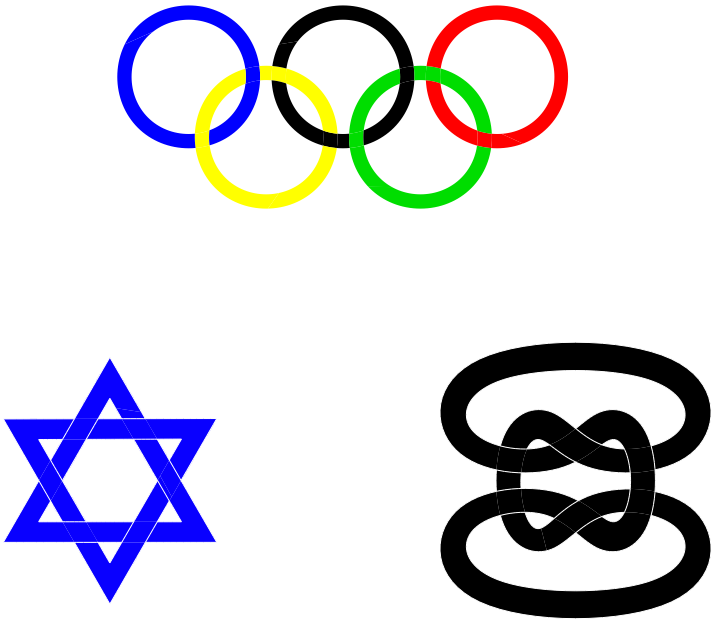

Figure 2: *Druid* permits the construction of drawings of interwoven surfaces, such as those shown here. a) Olympic rings, b) Star of David, c) Square knot

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 of two overlapping surface subregions

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.

2.1 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 [10]). The term was originally coined by Gibson [3], who defined the affordances of an object as the action possibilities of that object relative to a particular actor. 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 [13, 14], 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 is because 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 depicted by a drawing program is a $2\frac{1}{2}$D 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 or their subregions.

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 a set of affordances isomorphic to those of real surfaces should be provided by an effective drawing program. This amounts to a visual analogy between the program's usage and the thing depicted, in our case, $2\frac{1}{2}$D scenes. One example is translating a surface by "grabbing" it with a hand-shaped cursor and then dragging it in the desired direction. This 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 edit complex $2\frac{1}{2}$D 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.

2.2 Direct Manipulation Interfaces

A concept that is closely related to affordances is *direct manipulation interfaces* (see Norman and Draper [15]). 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. This is how Sutherland implemented Sketchpad [18]. Today, most computers use a mouse interface to move a cursor.

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 may seem obvious now, but it originated with the WYSIWYG [1] concept that came out of Xerox PARC in the 1970s (see Myers [12]). Raisamo and Raiha applied the idea of direct manipulation interfaces to the problem of aligning objects in a drawing [16]. 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 $2\frac{1}{2}$D scenes. Stated differently, the user interface is isomorphic to the $2\frac{1}{2}$D scene the program represents and manipulates. By constrast, 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.

2.3 Contstraint-based Interfaces

Constraint-based interfaces have recently 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 [4]). *Druid's* operation is also based on constraints. First, *Druid* constrains the user to constructing topologically valid $2\frac{1}{2}$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 $2\frac{1}{2}$D scene and its depiction in a drawing is not the first attempt to design a drawing tool based on a physical analogy. Gleicher [5] 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.

3 Spoofs

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. This is accomplished in existing drawing programs by constructing 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 and 4). Spoofs represent non-generic configurations, where various elements of a drawing are precisely aligned in order to create the illusion of interwoven surfaces.

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

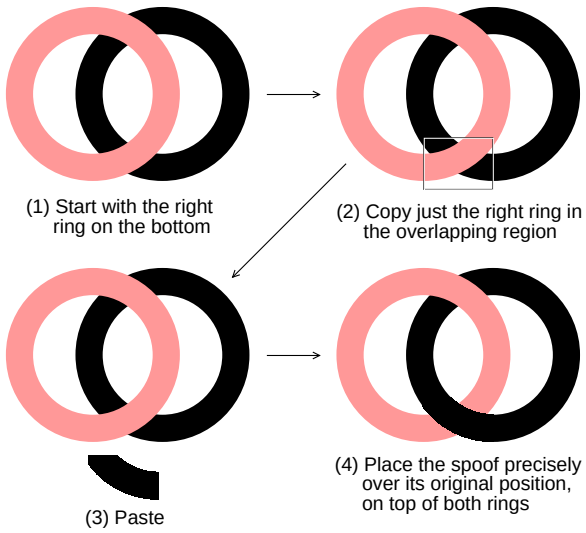

Figure 3: A spoof is a process by which the illusion of interwoven surfaces can be constructed in a layered system. The underyling representation does not match the final rendered image. See also Fig. 4.

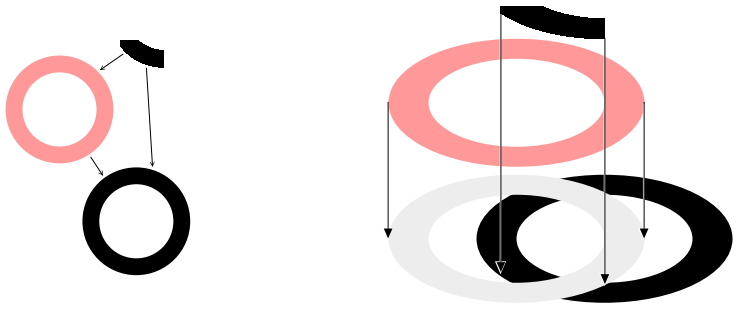

Figure 4: The figure on the left shows the DAG for the three components of the spoof in Fig. 3. While the illusion of interwoven surface has been created, the underlying representation is still a partial ordering, as required by existing drawing programs. The figure on the right shows an oblique view of the canvas, with the layers vertically spread apart to illustrate the spoof's construction.

One might ask, if spoofs are a sufficient method for creating rendered images of interwoven surfaces, what difference does it make if the underlying representation of the drawing does not correspond to the $2\frac{1}{2}$D scene which is perceived? Our answer lies in an analysis not of the *capability* of spoofs (spoofs are fully capable of solving the problem of creating images which will be perceived as interwoven surfaces), but in the unnaturalness and labor intensiveness of spoofs. 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.

4 Stages in Drawing Program Evolution

Druid represents a new kind of drawing program that is more powerful than existing programs. In this section we 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 DAG's 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 [19]). *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. 5).

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 subregions of the surfaces. 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 $2\frac{1}{2}$D scenes (see Williams [21]).

One might ask what advantages Stage 3 drawing programs have relative to Stage 2 drawing programs? One problem is that *Real-Draw's* approach is merely an incremental improvement over the Stage 1 representation. The result is that the actual interface for manipulating surfaces in *Real-Draw* is awkward and counterintuitive. It relies on the use of the push-back object, which both defines the region in which layers are to be reordered and provides the interface for manipulating the ordering. Because the push-back object does not reflect the way humans perceive and reason

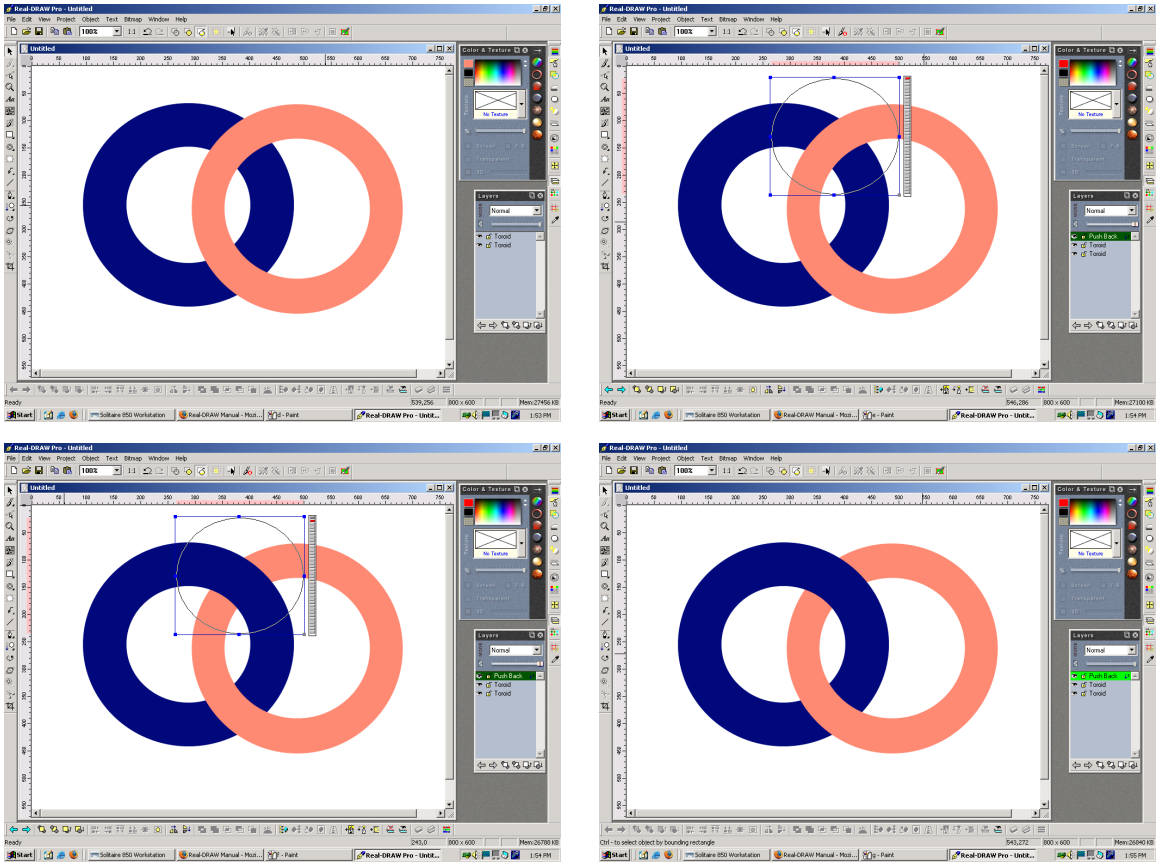

Figure 5: *Real-Draw* provides a *push-back* tool, which allows the user to define a region of the canvas and then manipulate the ordering of the layers within that region. The sequence shown here illustrates how this is accomplished, in the order left to right, top to bottom.

about surfaces, it is not a natural representation for overlapping surfaces, *i.e.*, it does not possess *natural* affordances (affordances that are isomorphic between the drawing and the surfaces being depicted).

Additionally, because the push-back object is a drawing 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 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 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 $2\frac{1}{2}$D scenes. There are two situations where this can arise. The first occurs when the user attempts to create a surface that overlaps itself. Each surface in *Real-Draw* is represented by a single label in the global surface DAG. A push-back only allows the reordering of layers based on labels in the DAG. Consequently, two overlapping subregions of the same surface will have the same global label, and that label will only occur once in the DAG. Since the label only occurs once in the DAG, and since the push-back tool allows manipulations of surface labels, not actual subregions, there is no way to represent a self-overlapping surface. The primary cause of this problem is that the push-back presents an interface for editing local DAG's, but the surface labels remain properties of a global DAG. This is a fundamental deficiency in *Real-Draw's* representation (Fig. 6).

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 (depending on the exact transparency model used) the appearance of a region where multiple surfaces overlap.

In theory, the push-back could be expanded in its power to allow a more comprehensive manipulation of the surface depths. However, the more serious problem of self-overlapping surfaces would remain unaddressed.

Druid's Stage 3 approach is more powerful in both of these regards since it naturally represents both self-overlapping surfaces and transparent surfaces with ease.

5 Labeling Scheme

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 regions which comprise the interiors of a set of surfaces. In constrast, a Stage 3 program represents the *boundaries* of surfaces and is not concerned with the regions interior to a surface until the final rendering step. The reason for this focus on boundaries rather than interiors is that depth changes always occur at surface boundaries, not interiors.

Our system, *Druid*, represents a $2\frac{1}{2}$D scene as a *labeled knot-diagram* (see Williams [21]). 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. 7). Williams extended ordinary knot-diagrams to include

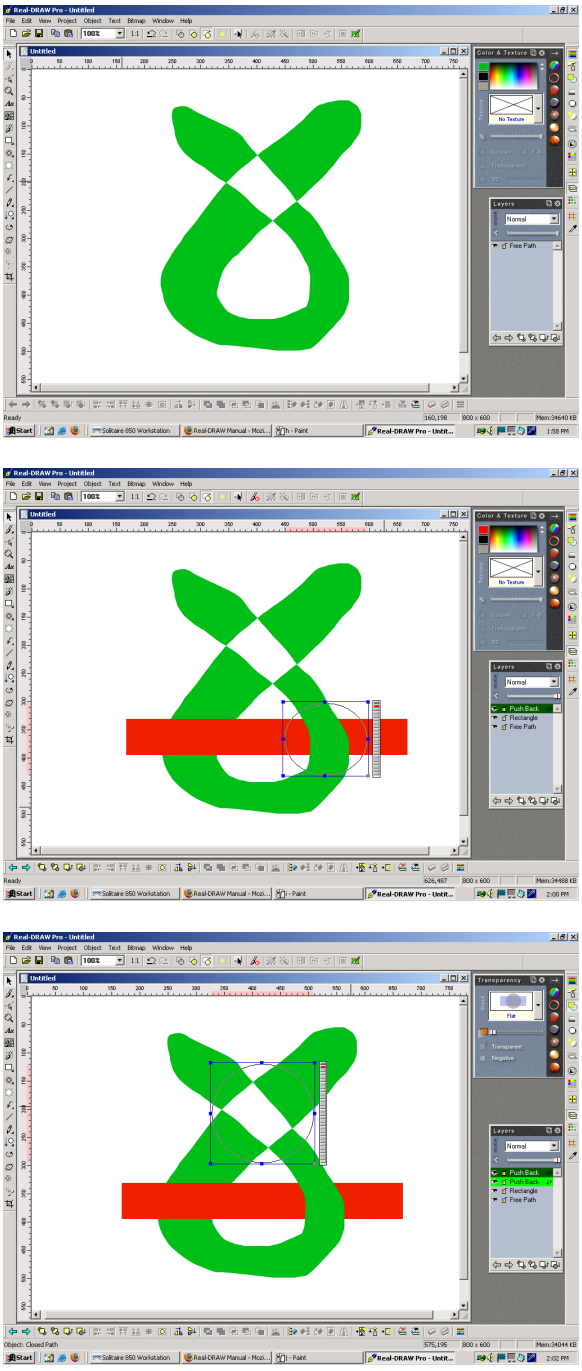

Figure 6: *Real-Draw* cannot properly represent self-overlapping surfaces. Since a surface has only one label in the surface layer list, there is no way to manipulate the depth ordering of multiple sub-regions belonging to a single surface. For this reason, *Real-Draw* cannot represent self-overlapping surfaces. The overlapping region is rendered with empty space, as shown here.

a *sign of occlusion* for every boundary and a *depth index* for every boundary segment (Fig. 8). 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.

This paper 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 [6]), 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. 9). 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 translated into images (the process for which is described later) in which the interiors of surfaces are filled with solid color.

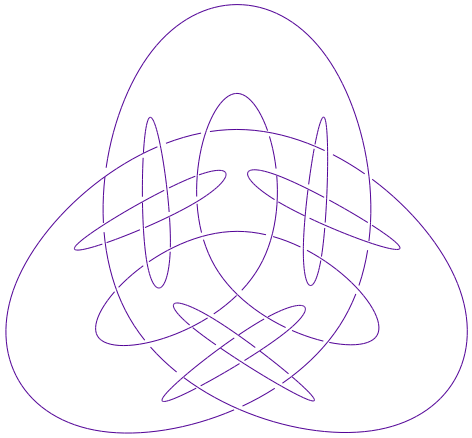

Figure 7: A *knot-diagram* is a projection of a set of closed curves onto a plane together with indications which show which curve is on top at every crossing.

6 Demonstration of *Druid*

Fig. 10 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 *solids* (*A* - *F*) or in a counter-clockwise order to create *holes* (*H*). Crossings can be clicked to reverse the relative depths of overlapping subregions (*G* and *I*). We call this interaction a *flip*.

Note that there is a natural logic to the operations in Fig. 10. For example, to alter the detpth ordering of various overlapping subregions, 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

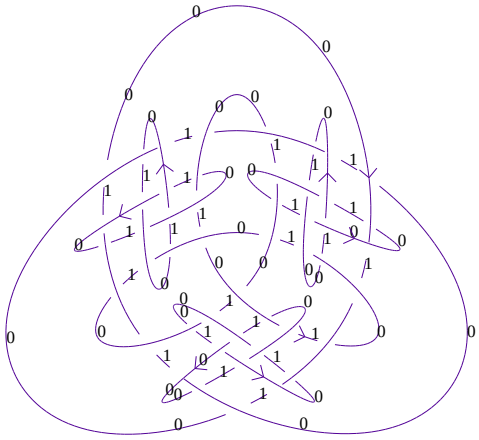

Figure 8: A *labeled knot-diagram* (see Williams [21]) is a knot-diagram with a sign of occlusion for every boundary and a depth index for every boundary segment. Arrows show the signs of occlusion for the boundaries, always denoting a surface bounded to the right of a boundary with respect to travel along the boundary in the direction of the arrow. Some depth indices of depth zero have been omitted for clarity.

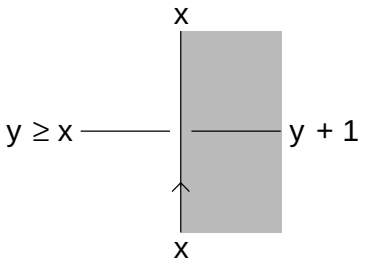

Figure 9: The *labeling scheme* (see Williams [21]) is a simple set of constraints on the depths of the four boundary segments that meet at a crossing. If every crossing in a labeling honors the labeling scheme then the labeling is a *legal labeling* and can be rendered. x is the depth of the upper boundary. 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.

consists of searching the space of legal labelings for a labeling which satisfies the constraint represented by the new crossing state. 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 I would be quite tedious. Worse yet, to invert the relative depth ordering within a subregion, 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 subregion and would have to be maintained if the user were to move the various surfaces around.

7 Labeled Knot-Diagram Spaces

Druid must search through a space of possible labeling assignments in order to find a legal labeling when a user-initiated change occurs. In this section we describe this search space.

Given an unlabeled drawing, there exists a set of possible legal labelings that can be assigned to that drawing. When the user causes a change to the drawing, the drawing becomes illegal in a localized area as a result of the change. The task for *Druid* is to fix the illegal part of the drawing by searching the labeling space for a new legal labeling.

The organization of the search process is motivated by the primary goal of finding the *minimum-difference labeling* with respect to the labeling prior to the change. 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 $L(t \in T)$:

$$C(t \in T) \subseteq L(t \in T) \tag{1}$$

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. 11) because signs of occlusion are specified by the user when boundaries are created (recall the convention that clockwise

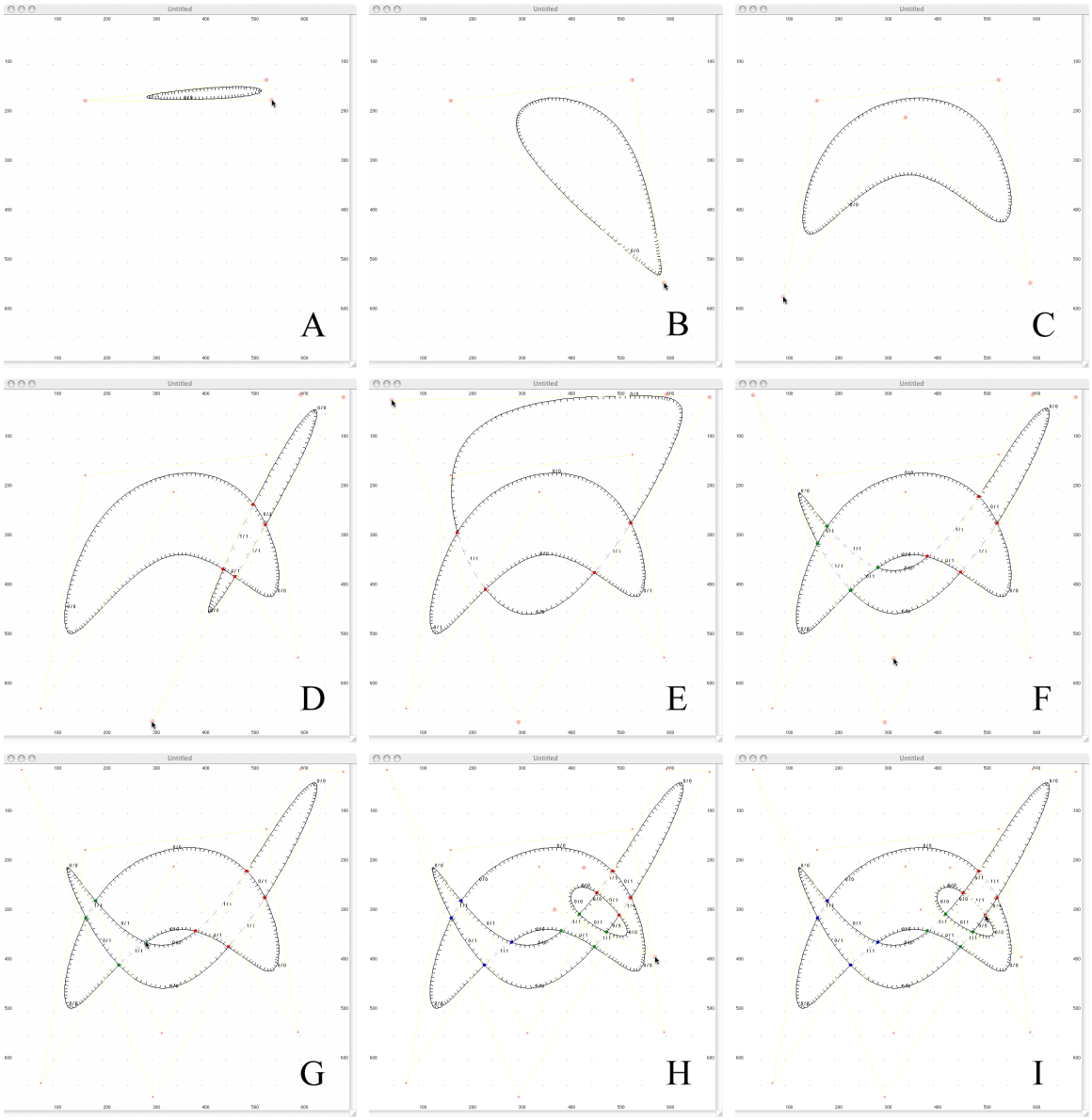

Figure 10: Demonstration of *Druid*. Spline control points are defined in either a clockwise order to create solids (*A* - *F*) or in a counter-clockwise order to create holes (*H*). Crossings are clicked to flip overlapping surface subregions (*G* and *I*).

construction of a boundary defines a solid and counter-clockwise construction of a boundary defines a hole).

For any individual $t \in T$ there exists $L(t)$ the space of possible labelings for that specific drawing topology (Fig. 12). $C \subseteq L$ is the subset of *consistent* (legal) labelings within L, *i.e.*, labelings in which all elements of the knot-diagram satisfy the labeling scheme.

Both L and C are connected graphs (Fig. 13). 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, $C_\Delta = 1$.

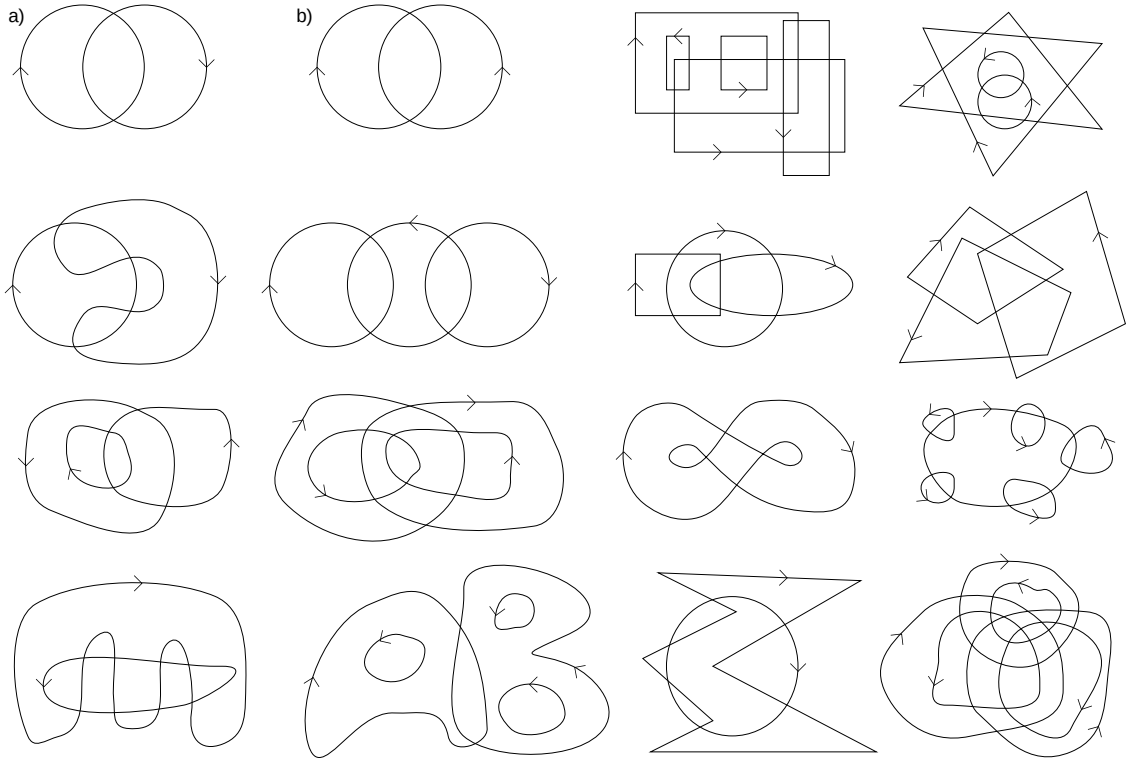

Figure 11: The space of all drawing topologies, T, is infinite. Some instances are shown here. Note that signs of occlusion are specified and distinguish otherwise identical topologies (*a* and *b*), but crossing states and depth indices are not specified.

7.1 Graph Distance Between Labelings

In a *connected graph* (a graph in which there exists a path between every pair of vertices) the *graph distance* for a pair of vertices is the number of edges that lie along the minimum length path between the two vertices. The root of the search tree is a consistent labeling, a member of $C(t \in T)$, *e.g.*, node

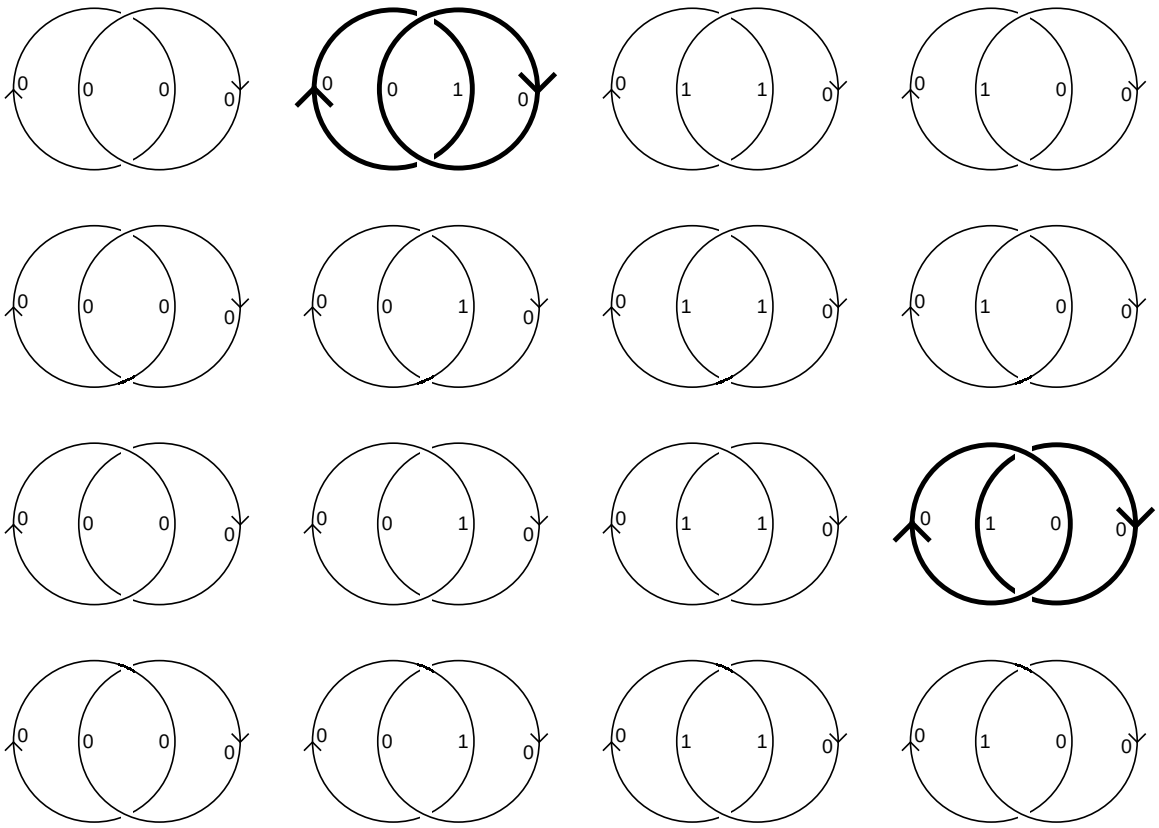

Figure 12: The space of labelings L for a particular topology, $t \in T$, consists of all combinations of crossing-states and depth indices for that topology. This figure shows $L(t \in T)$ corresponding to item *a* from Fig. 11. The two possible legal labelings are shown in bold and represent $C(t \in T) \subseteq L(t \in T)$.

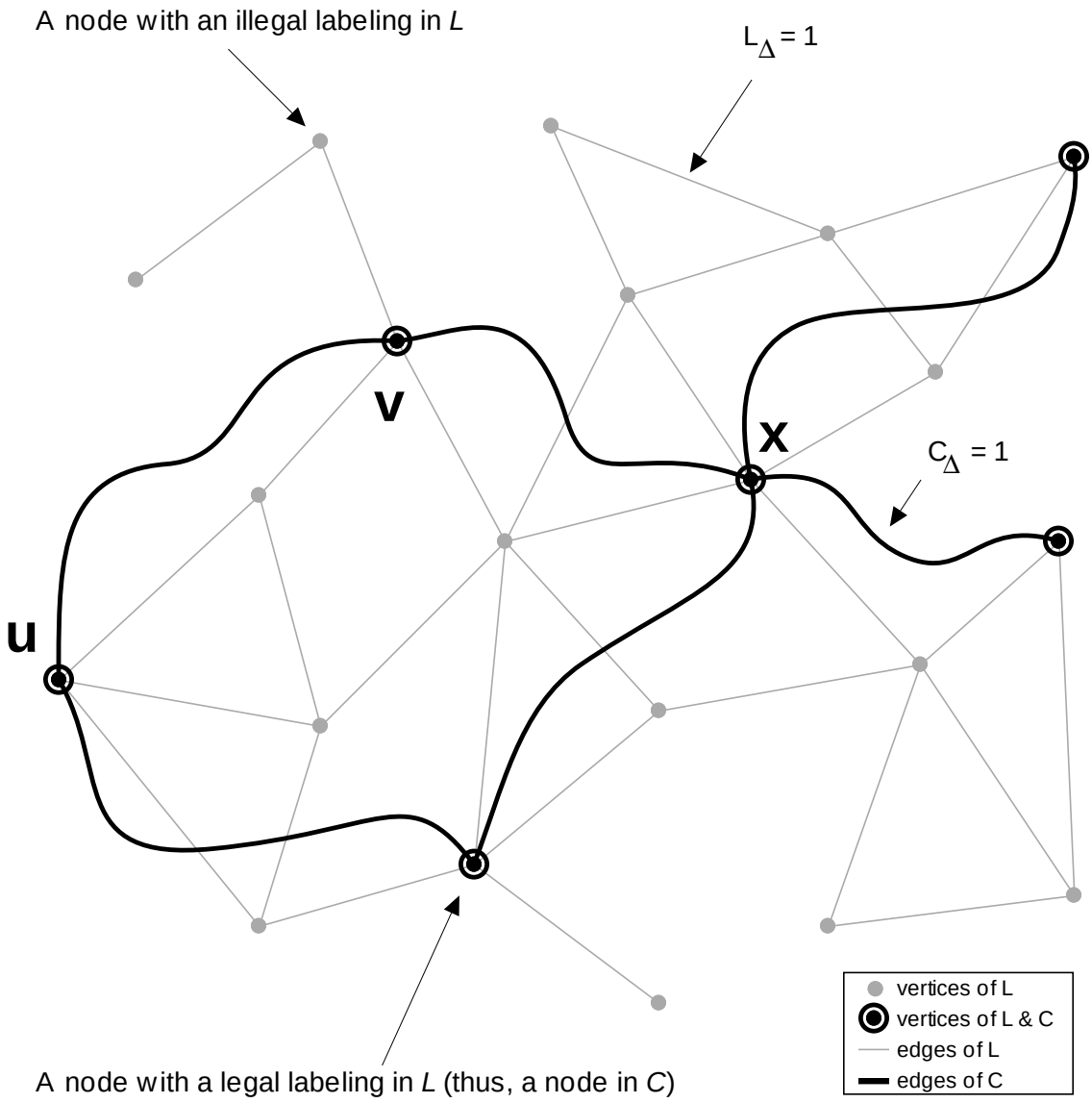

Figure 13: $L(t \in T)$ is a connected graph where nodes are labeled figures and edges denote labeling distance $L_\Delta = 1$. $C(t \in T) \subseteq$ L contains the consistent, legal labelings within L. $C(t \in T)$ is also a connected graph. Edges of C (shown as curves) denote user interface edit distances of 1, *e.g.*, a single mouse click.

v in Fig. 13. 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. 13 (for both of which C_Δ = 1 and L_Δ = 2).

By simply multiplying the number of distinct assignments to elements of the knot-diagram, we can compute the size of $L(t \in T)$ and an upperbound on the size of $C(t \in T)$. Since crossing-states take on two values their number of combinations is 2^R for R crossings. The number of combinations of depth assignments is more complicated to calculate. Given an unlabeled figure, every boundary segment has a maximum range of possible depths (Fig. 14). All boundary segments can theoretically lie at depth zero since they can always be lifted to depth zero by the user. Additionally, it is always true that if two depths are possible, then all intermediate depths are possible. Therefore, the range of depths for a boundary segment is limited to [*0 - d_i*] for segment i with a maximum depth of d_i. The number of combinations of depth assignments is the cumulative product of the depth ranges for all boundary segments. Thus, the size of L is:

$$2^R \cdot \prod_{i=1}^{S} d_i \tag{2}$$

for R crossings, S segments, and where d_i is the maximum depth for segment i. Since L is a superset of C, this value represents an upper bound on the size of C. For drawings of a moderate size this value 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. This contributes to the user's perception that he is interacting with real surfaces (see Norman [15]).

In Fig. 14, the size of L for each of the three drawings from left to right is 1, 16, and 110592 respectively. This figure illustrates the fact that the size of L scales explosively relative to the complexity of the drawing. Since L represents the potential search space, 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.

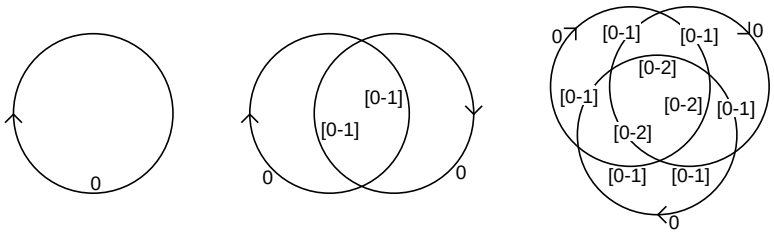

Figure 14: For a particular labeled knot-diagram, each boundary segment has a range of possible depths that it can assume, depending on how many surface subregions it overlaps. These figures show three simple examples with the depth ranges for each boundary segment indicated.

7.2 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 discussed in Section 5. The challenge is to label a knot-diagram in the fashion shown in Fig. 14, 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.

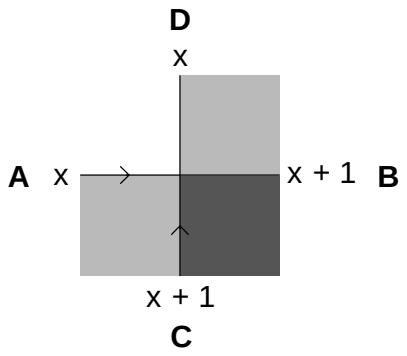

Figure 15: The *relaxed labeling scheme* is a set of constraints on the depth ranges of the four boundary segments that meet at a crossing. Each boundary occludes the half-plane on its right. Thus, for the two boundaries meeting at a crossing, one half of each boundary will lie in the *unoccluded* half-plane 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. Segments *B* and *C* are potentially occluded. The 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.

The constraints of the relaxed labeling scheme in Fig. 15 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 equations which can be solved by Gauss-Seidel or Jacobi iteration:

$$\begin{bmatrix} -1 & 0 & 1 & 0 & \dots \\ 0 & 1 & 0 & -1 & \dots \\ 0 & 1 & 1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} A \\ C \\ B \\ D \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \tag{3}$$

where A, B, C, and D are the segment depth ranges for a particular crossing and the right column vector according to Fig. 15.

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 our system of equations. Additionally, iterative methods are not always guaranteed to converge to a solution or convergence may 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.

8 Editing $2\frac{1}{2}$D Scenes

Note that *Druid* is much more than just an editor for labeled knot-diagrams. In a simpler program, a user might manually edit crossing-states and segment depths without constraint. There are two major problems with this approach. The first is that many of the knot-diagrams a user might create would be illegal and could not be rendered. Thus, the burden would fall upon the user to carefully manage the knot-diagram at all times. Transformations of the labeled knot-diagram would be incredibly tedious for the user to accomplish. The second problem is that the interface for such a program would provide the wrong affordances. The affordances of such a program would not be isomorphic with those of $2\frac{1}{2}$D scenes. This lack of isomorphic affordances would not give the user the experience that he is editing a $2\frac{1}{2}$D scene, but rather the experience that he is managing the details of a labeled knot-diagram. To summarize, *Druid* is actually an editor for $2\frac{1}{2}$D scenes, not for labeled knot-diagrams.

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. This is not necessarily a good design. There are situatinos where the user's intention may require navigating a large representation distance to achieve a small $2\frac{1}{2}$D 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 $2\frac{1}{2}$D scene transformation of size one, *i.e.*, an *elemental scene transformation*. However, in order to perform such an edit 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 edited 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 $2\frac{1}{2}$D 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 $2\frac{1}{2}$D 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 editing a $2\frac{1}{2}$D scene, including interwoven scenes, rather than the experience of editing one scene with 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 $2\frac{1}{2}$D scene transformations that makes *Druid* so novel.

9 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 $2\frac{1}{2}$D scenes. Consequently, the user has the experience of interacting with a $2\frac{1}{2}$D scene instead of editing a labeled knot-diagram. Several possible user interactions were listed in Section 2. 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 constrast, *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 these two kinds of interactions are handled by *Druid*.

9.1 Labeling-Preserving Interactions

Ideally, *Druid* should preserve the labeling during user interactions whenever possible because this is necessary to provide 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 editing a drawing because if it did, it would prevent him from being able to construct the $2\frac{1}{2}$D scene he has in mind. Labeling changes should only occur as the result of explicit constraints specified by the user. 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. 16). 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 analyze and follow 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. 16, where the user has dragged the lower boundary in a diagonal direction toward the upper-right in six discrete timesteps. Note that

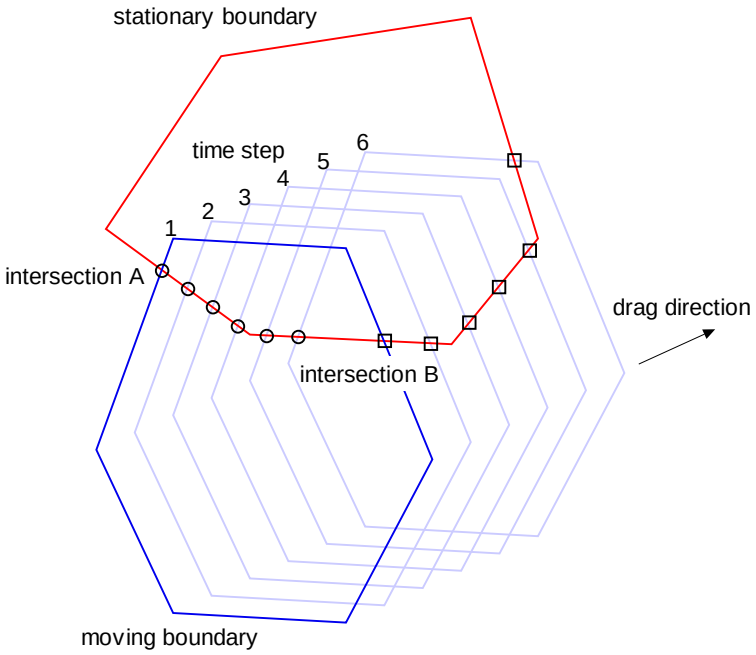

Figure 16: Dragging one boundary over another does not always alter the topology of the drawing. In such cases, it is best to preserve the crossings by predicting their new locations rather than deleting crossings and rediscovering them from scratch. In the above figure, the topology of the knot-diagram does not change over time. Only the locations of the two crossings change.

the 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. 16 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. 16, 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. 17. 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. 16, crossing B switches segments on both of its associated boundaries in a single jump. Fig. 17 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 are affected by the movement are immediately projected to their new locations. Since Fig. 16 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.

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 cross after a drag or reshape is performed, then projecting the crossing is simply a matter of updating the coordinates for the crossing. This is a trivial case 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. 17 illustrates how a single timestep is processed for a single crossing. It illustrates a case in which the projection will be complicated to perform because the crossings having 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

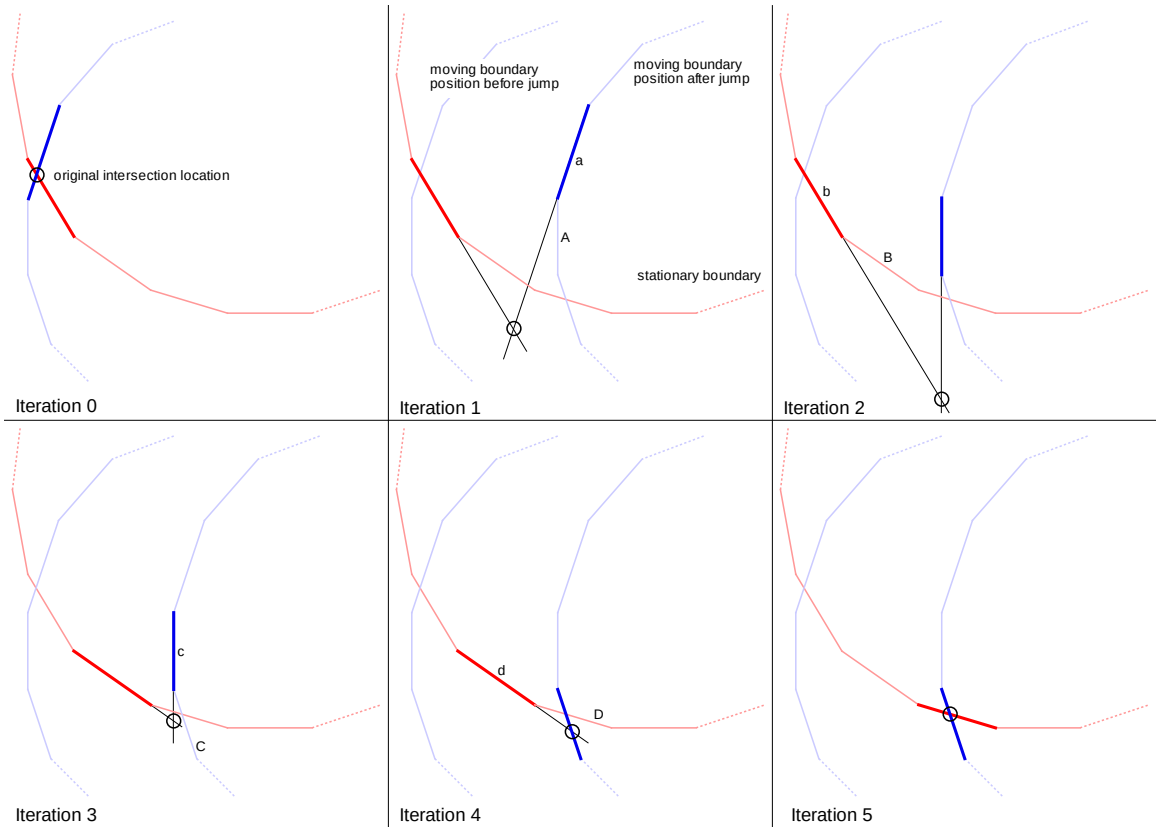

Figure 17: The sequence of drawings shown here illustrate successive iterations of the crossing projection algorithm. Thick line segments show the segment pair associated with the crossing at the beginning of each iteration. Circles show the intersections of the lines containing the segments. Lowercase 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.

the other. The loop terminates when the presently assigned pair of segments intersect.

Observe that between iterations in Fig. 17, one segment is retained while the other segment switches to an adjacent segment on 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 by 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.

9.2 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 may 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, this is not possible. The process for relabeling a labeled knot-diagram is described in the next section.

10 Finding a Legal Labeling

When the user causes a change that invalidates the current labeling, *Druid* must find a minimum difference legal labeling as quickly as possible. We will describe how this is accomplished 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 subregions associated with that crossing.

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

10.1 Overview of the Search

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

- 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 on each boundary, from which segment 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 [20]). 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, 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 system can be reduced to a single consistent labeling.

10.2 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 other 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 in 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. 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. This 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. 14, 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. This 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.

10.3 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 the previous section; 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 flipped by the user. 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 discovered by the backtracking search process 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.

11 Optimizing 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 have to be explored. Therefore, we employ a number of strategies 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 strategies are not independent of one another, but work in tandem, each strategy increasing the effectiveness of the others. The strategies are broken down into three major classifications:

- Choosing good boundary traversal starting segments
- Bounding the search using the current minimum difference to avoid a full enumeration of the search space
- Ordering the search to produce tight search bounds based on minimum difference earlier rather than later.

11.1 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. 14 illustrates how this can be accomplished. 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. While we do not presently have hard empirical data, we have informally observed that search times can often be decreased by up to a factor of ten for drawings of moderate complexity. 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.

11.2 Bounding the search

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 performance by bounding.

If there was 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.

11.3 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 may 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 was, as described previously, 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. 13), and the search tree, which represents the labelings explored by the search algorithm. 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.

So, although breadth first search has advantages, the algorithm described in Section 10 is a depth first search because subtrees are expanded regardless of the crossings' distances from the area of interest. Ordering the search such that those 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 [9]). 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 present 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), only one of these is constrained by iterative deepening. 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, informal observation of the performance that results from 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 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 Δ. Finding strong bounds early pays off heavily in terms of search efficiency.

The discussion so far has focused on the user-interaction of *flipping* a crossing, in which a user intentionally alters the layer ordering of overlapping subregions of two surfaces. There are more complex interactions as well, as described in Section 10. 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 Δ's is more complex, but the basic approach is the same.

12 Boundary Grouping With Cuts

One feature that is common to almost all drawing programs is the ability to group objects together. Groups are usually provided so that transformations like translation, scaling, and rotation can be applied to all of the members of a group. Users who are familiar with other drawing programs might expect that the most obvious objects to group in *Druid* would be individual boundaries since these are closely analogous to objects in other drawing programs. However, *Druid* also possesses a higher level of abstraction in which *surfaces* are the central objects, not individual boundaries. The use of surfaces as an organizing concept in *Druid* potentially conflicts with the user's expectation that group members be boundaries since a user may inadvertently create a group consisting of some boundaries from one surface and some from another. *Druid* handles these two kinds of objects, boundaries and surfaces, differently with respect to grouping. The multiple boundaries of a single surface are automatically grouped by *Druid* without any intervention from the user. The purpose is to allow a surface to be manipulated easily, *e.g.*, to translate a surface across the canvas without individually dragging each boundary component of that surface to its new location. Additionally, the user can group multiple surfaces into a temporary *selection* for the purpose of performing group transformations on a set of surfaces. A selection is transient in nature, in that it only exists for as long as the user keeps the selection selected. There cannot be multiple simultaneous selections, and selections are not stored as part of any persistent representation, but exist rather as a temporary grouping outside the representation for the purpose of applying user actions, *e.g.*, transformations, to the surfaces in a selection.

Boundary groups are not required for *Druid* to legally label a drawing. As in other drawing programs, *Druid* can use groups as a basis for translation of a surface with multiple boundaries. However, the more important use is for eliminating ambiguities about which surfaces boundaries

belong to. For example, in Fig. 18 (left), there exists an ambiguity as to whether boundary B bounds a surface below boundary A, above boundary A, or is part of the same surface as boundary A. This ambiguity will potentially cause problems, *e.g.*, if a the user were to attempt to place a third boundary that overlaps the ambiguous surface of A and B. In such a situation, there is the possibility that the third surface might be placed *between* boundaries A and B. Clearly, if the user's intent is for boundaries A and B to be part of the same surface, then such a placement violates the user's expectation about the effects of his interactions. Grouping boundaries can minimize this kind of problem.

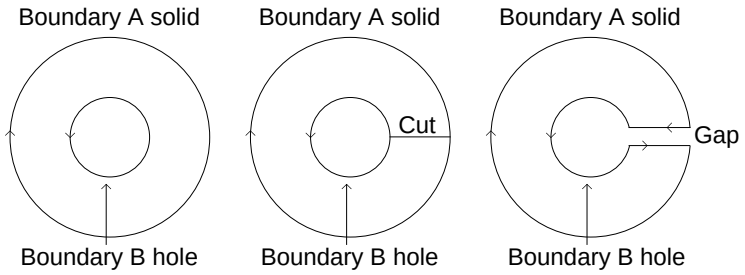

Figure 18: A *cut* can be thought of as a scissor cut through a surface connecting two boundaries of that surface. Cuts are used to group boundaries together for group transformations like translation, scaling, and rotation.

Groups are not created in the traditional way, through manual user grouping. Instead, *Druid* automatically finds and maintains groups without requiring any input from the user. It does this by finding and maintaining *cuts*. A cut can be thought of as a scissor cut through a surface connecting two boundaries that belong to a single surface. A cut converts two boundaries of a surface into a single boundary. See Fig. 18 (middle and right). After a cut is discovered it is used to create a boundary group. If a cut can be found between the two boundaries, as shown in Fig. 18, then the two boundaries are demonstrably part of the same surface and can be grouped for later operations that would otherwise require this particular ambiguity to be resolved.

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

13 Rendering

Rendering consists of converting the labeled knot-diagram (Fig. 19, left) to an image with solid fills for contiguous bounded regions of the canvas. To render opaque surfaces, we only need to find the depth zero surface for each region (Fig. 19, center). However, to render transparent surfaces we must find the full depth ordering of all surfaces for each region so that a transparent coloring model,

such as Metelli's *episcotister* model (see Metelli [11]), can be applied (Fig. 19, right).

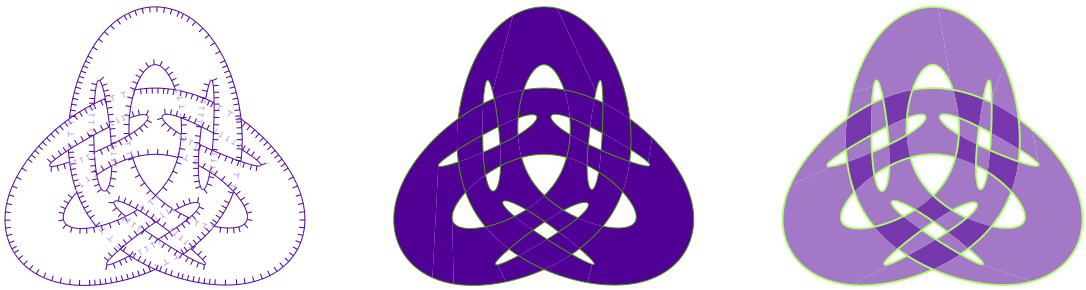

Figure 19: A labeled knot-diagram (left) can be rendered into a surface rendering such as those shown above (center, right). The surfaces associated with each region must be determined so that a proper coloring for that region can be assigned to the final image.

Rendering is the process by which a labeled knot-diagram (Fig. 19, left) is converted into an image in which contiguous bounded regions of the canvas are filled with designated colors. To render opaque surfaces (Fig. 19, center), *Druid* only needs to find the depth zero surface for each region. However, to render transparent surfaces (Fig. 19, right) it must find the full depth ordering of all surfaces for each region so the coloring model can be applied.

To find the surfaces associated with a region *Druid* uses a *slice*. A slice is similar to a cut except that instead of connecting two boundaries of a surface, a slice connects one boundary of a surface to an arbitrary location within the surface (Fig. 20). To find the surfaces associated with a region of the canvas, *Druid* finds slices within the region that originate at a location inside the region and terminate on surrounding boundaries at various depths, iterating from depth zero and increasing in depth until it fails to find a boundary for a slice that starts at a particular depth. 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.

14 Conclusions

All drawing programs must have a way to distinguish which surface is on top for any overlapping pair of subregions. 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 $2\frac{1}{2}$D scene, a drawing program should use a representation that permits the construction of any $2\frac{1}{2}$D scene. Unfortunately, the assumption adopted by most existing drawing programs is that surfaces reside in distinct layers. This assumption is not true of the space of all possible $2\frac{1}{2}$D scenes. Therefore, existing drawing programs cannot represent all $2\frac{1}{2}$D scenes. We have developed an innovative new drawing program with the following major capabilities:

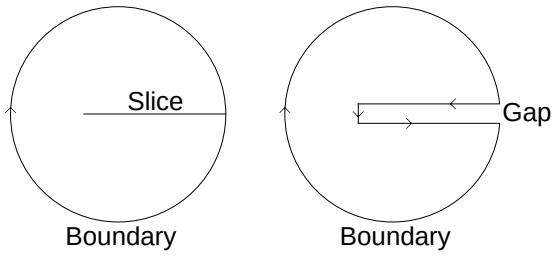

Figure 20: A slice connects a boundary to a location inside the bounded surface. Slices are used to find the depth ordering of the surfaces associated with a particular region. This must be done in order to render a scene.

- Naturally representing a more general class of drawings than other programs, *i.e.*, drawings in which surfaces may interweave
- Providing 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 subregion of the canvas for the purpose of rendering.

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* represents surfaces by their closed boundaries and 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 $2\frac{1}{2}$D scenes in that actions performed by the user are isomorphic to elemental transformations of $2\frac{1}{2}$D 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.

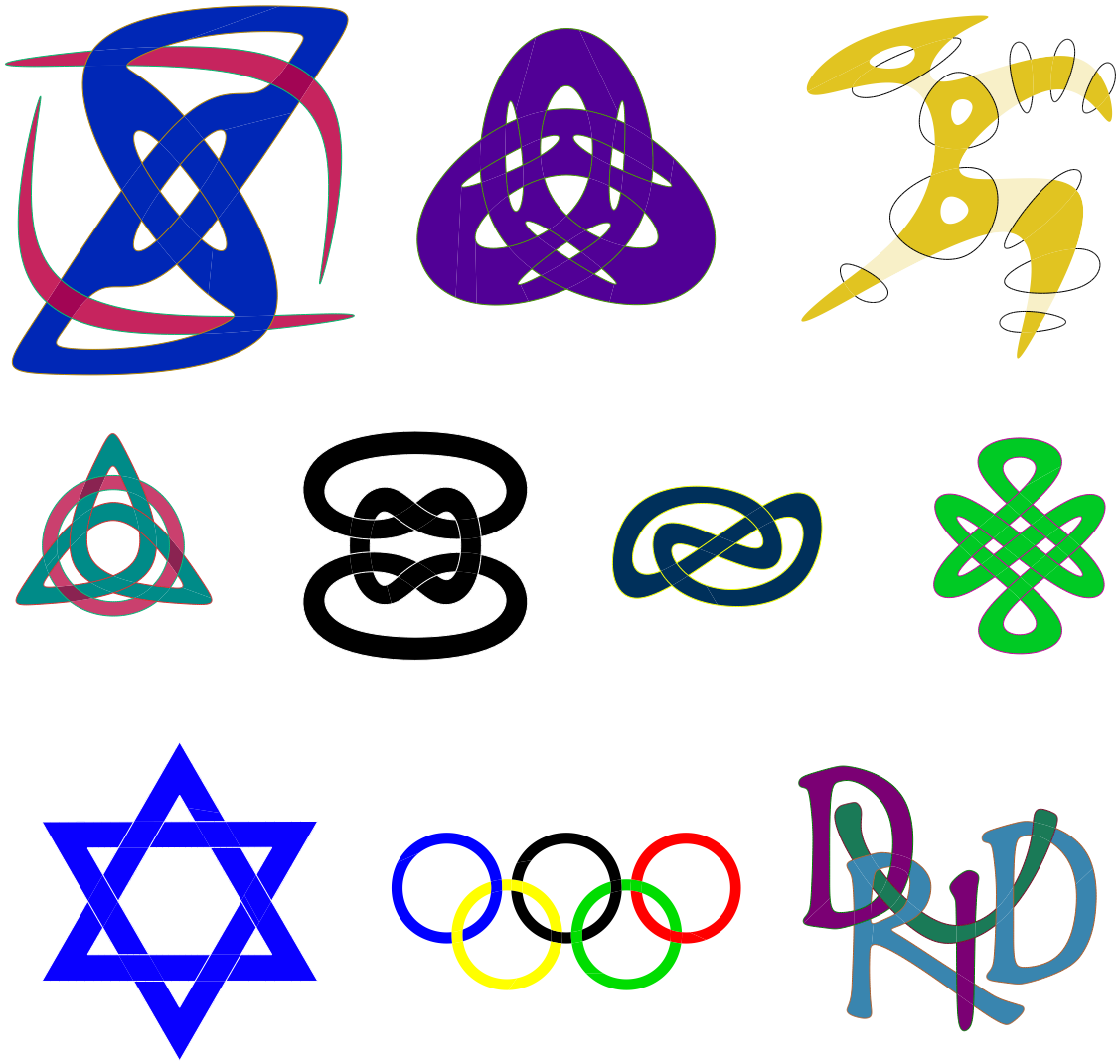

Figure 21: Shown above are several examples of artwork created with *Druid* . The construction and maintenance of these drawings is simple and straightforward.

References

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

[2] Craig, D., LisaDraw 3.0 manual, 1984.

[3] Gibson, J. J., *The Ecological Approach to Visual Perception*, Houghton Mifflin Co., Boston, MA, 1979.

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

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

[6] Huffman, D. A., Impossible Objects as Nonsense Sentences, *Machine Intelligence*, **6**, 1971.

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

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

[9] Marsland, T. A., and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *ACM Computing Surveys*, **14** (4), pp. 533-551, 1982.

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

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

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

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

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

[15] Norman, D. A., and S. W. Draper, *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[16] 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.
http://citeseer.ist.psu.edu/raisamo96techniques.html

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

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

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

[20] Waltz, D. L., Understanding line drawings of scenes with shadows, McGraw-Hill, New York, pp. 19-92, 1975.

[21] Williams, L. R., *Perceptual Completion of Occluded Surfaces*, PhD dissertation, Univ. of Massachusetts at Amherst, Amherst, MA, 1994.