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

Keith Wiley
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131 USA
505-277-7836
505-277-6927 (fax)
kwiley@cs.unm.edu

Lance R. Williams
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131 USA
505-277-3914
505-277-6927 (fax)
williams@cs.unm.edu

February 16, 2007

Abstract

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

This paper describes a novel approach which only requires local depth ordering of segments of the boundaries of surfaces in a drawing rather than a global depth relation between entire surfaces. Our program, *Druid*, 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.

Keywords: J.6.a Computer-aided design

Author Bios

Keith Wiley recieved the BA degree in psychology from the University of Maryland in 1997 and the MS and PhD degrees in computer science from the University of New Mexico in Albuquerque in 2003 and 2006, respectively. He is currently a lecturer in the Computer Science Department at the University of New Mexico in Albuquerque. His principal research interests are image processing, CCD imaging, and evolutionary algorithms and simulations.

Lance R. Williams recieved the BS degree in computer science from Pennsylvania State University in 1985 and the MS and PhD degrees in computer science from the University of Massachusetts at Amherst in 1988 and 1994, respectively. He spent four years as a post-doctoral scientist at NEC Research Institute. He is now an associate professor in the Computer Science Department at the University of New Mexico in Albuquerque. His principal research interests are human visual information processing, perceptual organization for artificial vision systems, and neural computation.

1 Introduction

1.1 Summary of Existing Drawing Programs

Drawing programs originated with Sutherland's seminal Ph.D. thesis in 1963, in which many recognizable components of modern drawing programs were already present (see Sutherland [23] and Sutherland [24]). Since then, a number of refinements have been made to the general design of drawing programs, aided primarily by increased computing power and hardware and software innovations such as the mouse and graphical user interface toolkits. In 1984, Apple released *LisaDraw 3.0* (see Craig [9]), which despite its age, still serves as a model of a modern drawing program. It uses a tool palette that is similar to the tool palettes of modern drawing programs and which provides similar functionality. However, its method of representating the relative depths of surfaces has not been improved upon in twenty years, despite its intrinsic weaknesses. For the last twenty years, research on drawing programs has focused on areas other than the nature of the surface representation. For example, a considerable amount of research has focused on methods for constructing objects and manipulating their shape, but once constructed, those objects are assigned depths in a conventional layer-based drawing representation. One area in which considerable progress has been made is the intelligent interpretation of *sketches*, *i.e.*, imprecise hand drawings which are analyzed to discover salient features. For example, *CorelDRAW 12*, a professional drawing program, provides a *smart drawing tool*, which allows a user to freehand draw approximate shapes that are recognized and fitted to stock shapes such as ellipses (see Corel [8]). Barla, *et al.* describe a method of line drawing simplification which either removes or merges extraneous lines without losing the salient features of the drawing (see Barla *et al.* [4]). Gangnet *et al.* describe a method for closing gaps in freehand curves, which is important for paint-fill algorithms (see Gangnet *et al.* [11]). In addition, some work has focused on interpreting two-dimensional sketches as three-dimensional shapes. Both Karpenko [14] and Cordier and Seo [7] describe systems which convert hand-drawn sketches into volumetric models. In addition, Raisamo [20] describes a novel way to construct shapes in which a block is chiseled, similar to three-dimensional sculpting, rather than defined using explicit boundaries such as splines. While the previous work on drawing programs is significant and valuable, it all relies on a layer-based representation for the relative depths of surfaces, and this representation has serious limitations.

One function of a drawing program is to allow the construction and manipulation of drawings of overlapping surfaces, which we simply call *2½D scenes*. A 2½D scene is a scene of surfaces that is fundamentally two-dimensional, but which also represents the relative depths of those surfaces in the third dimension, *i.e.*, the absolute position of a surface in the third dimension is not indicated, only the relative depths of pairs of surfaces are. Using existing programs, a drawing can easily be constructed in which multiple surfaces partially overlap. When this occurs, the program must have a means of representing which surface is on top wherever two surfaces overlap. Existing drawing programs solve this problem by representing drawings as a set of distinct layers where each surface resides in a single layer. For any given pair of surfaces, the one that resides in the upper (or shallower) layer is assigned a smaller depth index and appears above wherever those two surfaces overlap. Consequently, the use of layers imposes a partial ordering, or a directed acyclic graph (DAG), on the surfaces such that no subset of surfaces can interweave (Fig. 1). This restriction precludes many common drawings which a user may wish to construct (Fig. 2), specifically, it those drawings that contain *interwoven surfaces*, *i.e.*, pairs of surfaces for which each surface is above the other somewhere in the scene. Because such programs do not span the full space of possible 2½D scenes, they therefore impose limitations on the drawings that a user can create.

Our research uses a more general representation as the basis for a more powerful drawing tool, called *Druid* (see Wiley and Williams [27]). *Druid* eliminates the assumption that surfaces cannot interweave. It therefore spans a larger space of 2½D scenes by using a representation that makes weaker assumptions about the drawing. This generality makes *Druid* a more versatile drawing tool.

1.2 Previous Research

It is important to realize that *Druid* is not merely a Celtic knot tool. Many researchers have studied the construction of Celtic knots (*e.g.*, Cromwell [10] and Scharein [22]) and several programs are specifically designed to facilitate the construction of knots in general (*e.g.*, [1, 3, 6, 22, 29]). *Druid*, on the other hand, permits the construction of much more general scenes in which the surfaces can be any orientable two-manifold with boundary. In particular, there exist scenes of interwoven surfaces which are not knots, *e.g.*, Fig. 2 (*a* and *b*). Neither conventional drawing programs nor programs for constructing knots can represent such scenes.

Perhaps the research that most closely resembles *Druid* is that by Baudelaire and Gangnet [5], which relies on *planar maps* as the organizing principle for a drawing tool and *map-sketching* as a means to construct 2½D scenes. Planar maps permit the construction of overlapping curve segments, which are subsequently pruned using a process where some edges are erased. Each face of the graph (an edge-bounded region of the surface boundary graph) can be assigned an independent color in the drawing. With a proper erasure of edges and coloring of faces, an illusion of interwoven surfaces can be achieved. Baudelaire and Gangnet's planar maps are quite similar to the best method available in *Adobe Illustrator* [2] for constructing 2½D scenes[1] in which the user converts a series of overlapping curves into a planarized graph in which the faces can be assigned independent colors. As such, the Baudelaire method is simply the *Illustrator* method with an additional (and unnecessary) edge-erasing interaction, although their research predates the implementation of this feature in *Illustrator*. We describe this planar map method for constructing interwoven scenes in detail in Section 3.3.

A more significant deficiency of the planar map method is that it does not possess the natural *affordances* of 2½D scenes. Affordances are the natural uses an object suggests for itself (see Norman [17, 18]). For example, a mouse affords translational movement and clicking. 2½D scenes afford 2½D manipulations such as being stretched, translated, cut into smaller surfaces, having holes cut in them, and being lifted above

[1]This is the best method that is *natively* available in *Adobe Illustrator*. There are third-party plugins that facilitate the construction of a specific *subset* of interwoven scenes, *i.e.*, Celtic knots (see Artlandia [3]). However, these systems do not span the full space of 2½D scenes and do not provide natural *affordances*.

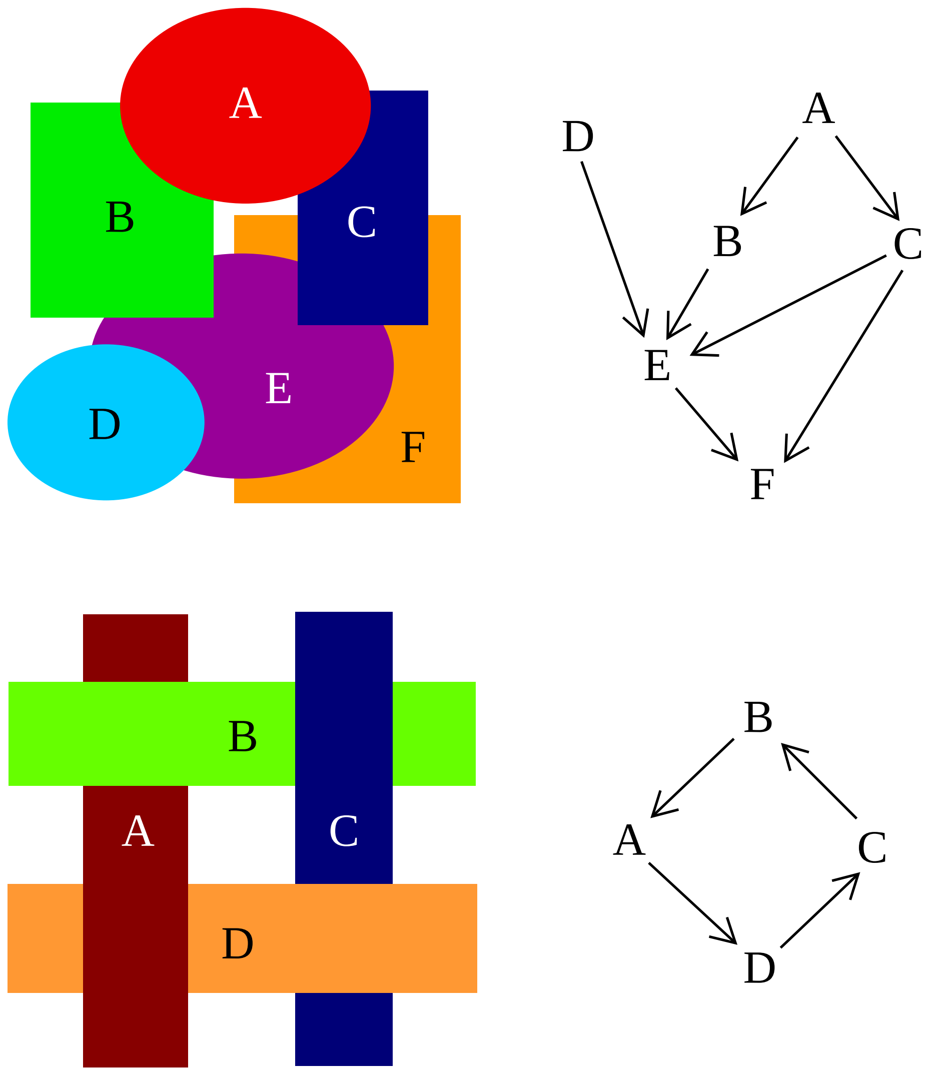

Figure 1: The classic approach to representing relative surface depths is to assign the surfaces to distinct layers (top left). It follows that the surface relative depth relation is a directed acyclic graph (DAG). No subset of surfaces can interweave because this would require a cycle in the graph (top right). This approach precludes interwoven drawings (bottom left) in which the surface relative depth relation has cycles (bottom right).

or pushed beneath one another in portentially interwoven arrangements. We believe that a 2½D drawing program should have a user interface with affordances isomorphic to those of 2½D scenes.

2 Drawing Programs

2.1 Drawing Program User-Interactions

Many drawing programs provide tools for creating and manipulating splines. One such program, *MacPowerUser's iDraw* [15], bases all objects on splines. Rectangles, polygons, and text are all represented using splines. This provides a consistent interface for manipulating the various kinds of objects. Like *iDraw*, *Druid* is based entirely on splines and like both *ivtools idraw* [13] and *Xfig* [21], *Druid* uses B-splines. All boundaries are B-splines, including shapes that can be approximated by splines, such as hand-drawn curves, rectangles, and text. The assumption that all boundaries are splines lends a uniformity to a drawing program's interface that makes it easier for a user to understand, while simulaneously simplifying the programmer's job. There are a number of user-interactions that a spline-based drawing program should permit. These interactions include:

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

The interface of a program should not only provide a method for each of these interactions; these methods must be *user-friendly*, that is, simple to understand and easy to use. Unfortunately, the only previous attempt to circumvent the partial ordering limitation, *MediaChance*'s *Real-Draw Pro 3* (see Voska [25]), forces the user to use a complex and confusing interface.

2.2 Software Affordances

A software application's interface possesses a specific set of *affordances*. According to Norman, affordances are the clues an object offers about how it can be used (see Norman [17, 18]). There is a general agreement that affordances are difficult to define with respect to software. This observation results from the fact that

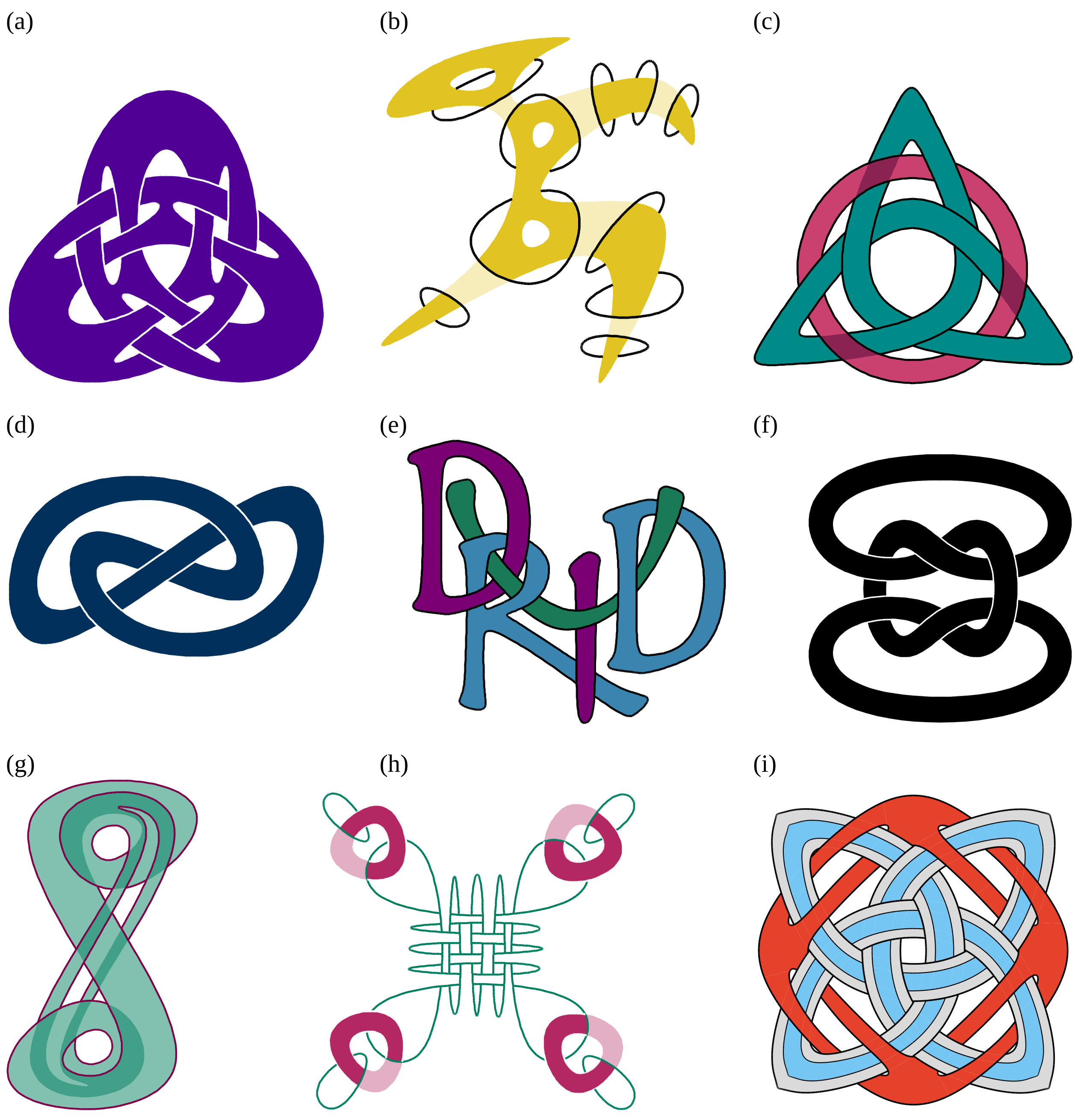

Figure 2: Examples of artwork created with *Druid*. The construction and maintenance of these drawings is simple and straightforward. Note that some of these drawings are mathematical knots (*c*, *d*, *e*, *f*) while the rest are not (*a*, *b*, *g*, *h*, *i*).

affordances are usually defined with respect to physical properties 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 two dimensional image to the user with which to interact. This interaction usually involves a keyboard and a mouse. However, in the case of drawing programs, use of the keyboard is generally minimized because this violates the notion of *direct manipulation interfaces*. 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 (see Norman [18]). Therefore, the affordances of drawing programs mainly consist of clicking on and dragging various features, *e.g.*, spline control points, of the display with a mouse and cursor.

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

Real physical surfaces possess certain natural affordances. They can be stretched, translated, cut into smaller surfaces, have holes cut in them, and lifted above or pushed beneath one another in potentially interwoven arrangements. They can also be colored or be made transparent. In some cases they can be glued to other surfaces to form larger surfaces. We believe that an effective drawing program should provide a set of affordances that is isomorphic to those of real surfaces. This amounts to a visual analogy between the program's usage and the thing depicted, in our case, 2½D scenes. One example is translating a surface by "grabbing" it with a hand-shaped cursor and then dragging it in the desired direction, which is directly analogous to how a graphic designer might move pieces of paper around on a drafting board. Unfortunately, many drawing programs do not possess isomorphic affordances. It is our belief that while some programs, such as *Real-Draw* (see Voska [25]), attempt to solve the problems posed in this research, they do so using interfaces with unnatural affordances which make the programs complicated and non-intuitive to use. *Druid's* interface possesses affordances which are isomorphic to the affordances of the physical surfaces which are depicted. As such, it is simpler to use, while at the same time, is more powerful than existing drawing programs in its capability to create and manipulate complex 2½D scenes.

3 Comparison Between Conventional Drawing Programs and *Druid*

3.1 Constructing Images of Interwoven Surfaces in Conventional Drawing Programs

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

1. Spoofs

2. Painting planarized graphs
3. Local DAG manipulation.

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

3.2 Spoofs

One method for constructing an image of interwoven surfaces without using an interwoven representation is to construct one set of surfaces which has the appearance of a completely different set of surfaces. We call this sort of illusion a *spoof* (Fig. 3). Spoofs represent non-generic configurations, where various elements of a drawing are precisely aligned in order to create the illusion of interwoven surfaces.

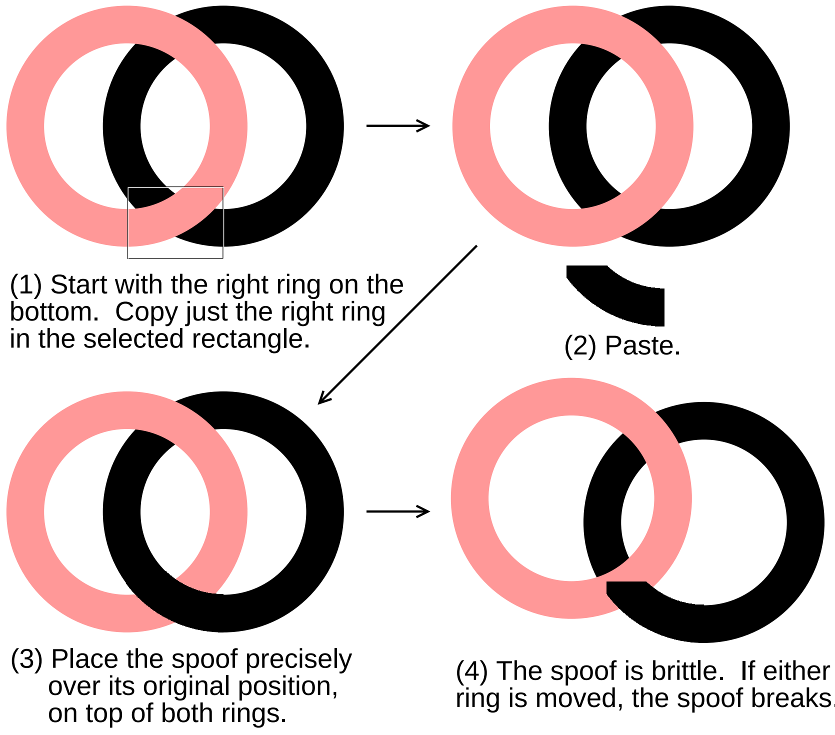
(1) Start with the right ring on the bottom. Copy just the right ring in the selected rectangle.
(2) Paste.
(3) Place the spoof precisely over its original position, on top of both rings.
(4) The spoof is brittle. If either ring is moved, the spoof breaks.

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

3.3 Painting Planarized Graphs

Another method for constructing an image that has the appearance of interwoven surfaces is *painting planarized graphs*, which was briefly discussed in Section 1.2. This method consists of converting a drawing of boundaries that represent overlapping surfaces into a planar graph where vertices represent boundary crossings and edges represent boundary segments (Fig. 4, top left). Once the drawing has been converted to a planar graph, each face of the graph can be independently *painted* (or *filled*) with a color of the user's choosing (Fig. 4, bottom right). With a proper assignment of paint colors to the faces of the graph, an image can be constructed which has the appearance of interwoven surfaces (Fig. 4, bottom right).

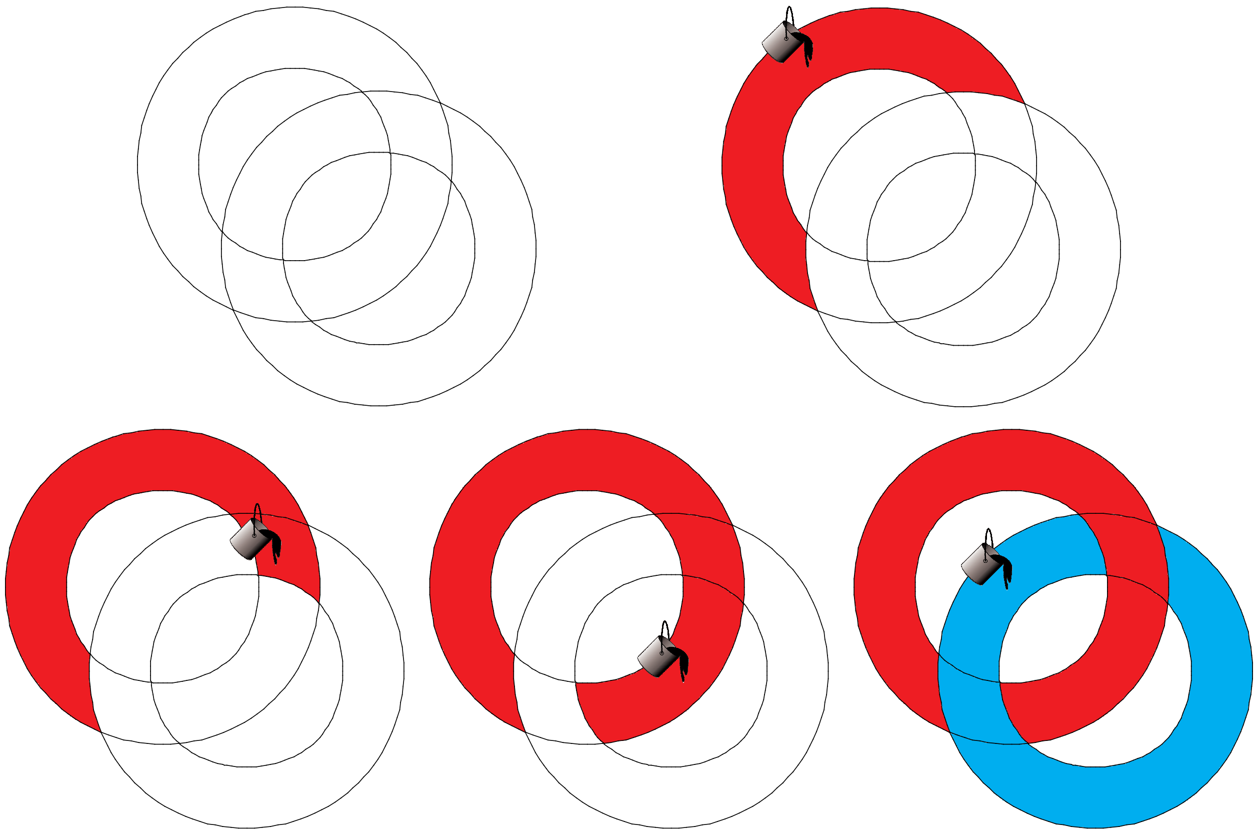

Figure 4: Painting a planarized graph begins by representing a set of boundaries as a planar graph (top left). The graph can be colored by painting faces of the graph with a fill color (top right). By careful assignment of paint colors to the faces of the graph, an image can be constructed which has the appearance of interwoven surfaces (bottom right).

3.4 Local DAG Manipulation

To our knowledge, the only program that makes a serious attempt to solve the problem of representing interwoven surfaces is *MediaChance*'s *Real-Draw Pro-3* (see Voska [25]). *Real-Draw* starts out with a basic layered representation, but then provides a special device called a *push-back object*. This device 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, top).

3.5 Affordances of Conventional Drawing Programs

One might ask, if spoofs, painting planarized graphs, and local DAG manipulation are sufficient methods for creating rendered images of interwoven surfaces, what difference does it make if the underlying representation of the drawing does not correspond to the 2½D scene which is perceived? Our answer lies in an analysis not of the *capability* of these methods (these methods are fully capable of creating images which are perceived as interwoven surfaces), but in the unnaturalness and labor intensiveness of these methods. Spoofs are tedious to construct, requiring many steps to be performed with precision. Furthermore, they are brittle because once a spoof has been constructed, any alterations to the drawing will require the spoof to be redone. Likewise, planarizing a drawing followed by painting the faces of the graph independently requires many intermediate steps which must be performed in a precise manner in order to achieve the desired effect. The point is not that layer-based systems are unnatural (they represent layered scenes perfectly), it is that they are not general, *i.e.*, there are 2½D scenes which cannot be represented using layers and whose appearance is difficult to simulate using a layer-based system.

The push-back tool used by *Real-Draw* (see Voska [25]) is the easiest of the three methods listed above, but it is not ideal. One problem with *Real-Draw's* approach is that it relies on the use of a new kind of object on the canvas, the push-back object. Because the push-back object does not operate in the way in which humans perceive and reason about surfaces, it does not possess the *natural* affordances of 2½D scenes (see Section 2.2).

Additionally, the presence of a push-back object on the canvas makes the user's job more tedious because 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 object, the user must also adjust the location and scale of the object to make sure it still encompasses the relevant region of the canvas. Furthermore, the introduction of new surfaces into an existing push-back object's region requires that the old push-back object be replaced. We believe that labeled knot-diagram-based representations (as used by *Druid*) offering better affordances, and which make fewer demands on the user, are a better solution.

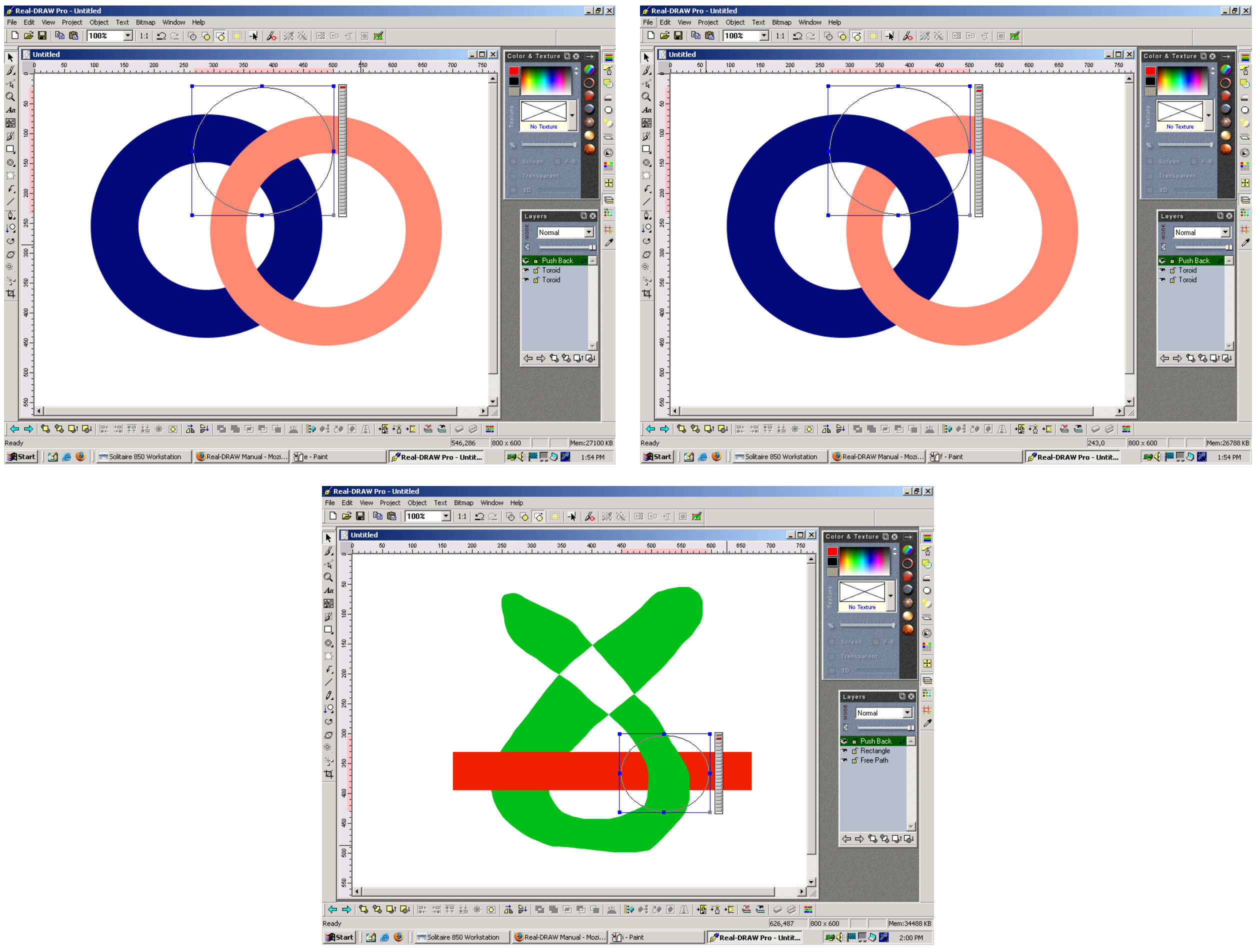

Figure 5: *Real-Draw* (see Voska [25]) provides a device called a *push-back* object, which allows the user to define a region of the canvas and then manipulate the ordering of the layers within that region. The top two figures illustrate how this manipulation is accomplished. Notice that *Real-Draw* cannot properly represent self-overlapping surfaces (bottom figure). Since a surface has only one label in the surface layer list, there is no way to manipulate the relative depths where a single surface covers a region at multiple depths. For this reason, *Real-Draw* cannot represent self-overlapping surfaces. The overlapping region is rendered with empty space, as shown in the bottom figure.

A possibly more significant drawback than possessing unnatural affordances is that *Real-Draw* does not span the space of 2½D scenes. There are two situations where this can arise. The first situation occurs when the user attempts to create a surface that overlaps itself. A single label is used to represent each surface in the global surface DAG, even when that surface overlaps itself. Since a push-back object only allows the reordering of layers based on labels in the DAG, there is no way for a push-back object to represent or manipulate self-overlapping surfaces. The primary cause of this problem is that the push-back object presents an interface for manipulating *local* DAGs, but the surface labels remain properties of a *global* DAG. In summary, the dependence on an underlying DAG is a fundamental deficiency in *Real-Draw's* representation (Fig. 5, bottom).

An additional problem with *Real-Draw* is that since the push-back object only allows the depth zero object to be pushed down, there is no way to manipulate the ordering of deeper layers. If surfaces are opaque this does not matter since only the depth zero surface will be visible. However, if surfaces are transparent, then the ordering of deeper layers might affect the appearance of a region where multiple surfaces overlap (depending on the exact transparency model used). In theory, the basic idea of a push-back object could be extended to allow a more comprehensive manipulation of the surface depths. However, the more serious problem of self-overlapping surfaces would remain unaddressed.

3.6 Demonstration of *Druid*

To this point, this section has illuminated the weaknesses inherent in constructing images of interwoven 2½D scenes with conventional drawing programs. In this section, we offer a demonstration of how *Druid* is used to perform this task for the purpose of comparison (see Fig. 6). Spline control points are defined in either a clockwise order to create solids (*A*) or in a counter-clockwise order to create holes (*B* and *D*). Crossings can be clicked to reverse the relative depths of areas where surfaces overlap (*C* and *E*). We call this interaction a *crossing-flip*. Whenever the current labeling is *legal*, *i.e.*, whenever all crossings satisfy the labeling scheme, the drawing can be *rendered* (*F*).

Note that there is a natural logic to the operations in Fig. 6. For example, to alter the depth ordering of various overlapping surfaces, the user merely clicks on a crossing to invert its crossing-state. *Druid* then does all of the computation necessary to keep the labeling legal. This computation consists of relabeling the figure so that the constraint represented by the new crossing-state is satisfied and the entire figure is legal. Compare this mode of interaction with either the spoof approach or with the local DAG manipulation approach, *i.e.*, the approach employed by *Real-Draw*. Construction of a spoof that appears like *E* would be quite tedious. Worse yet, to invert the relative depth ordering within an overlapping area, the spoof would have to be completely rebuilt. If one were to use *Real-Draw*, push-back objects would have to be explicitly created for each desired region of overlap and would have to be maintained if the user were to move the various surfaces around.

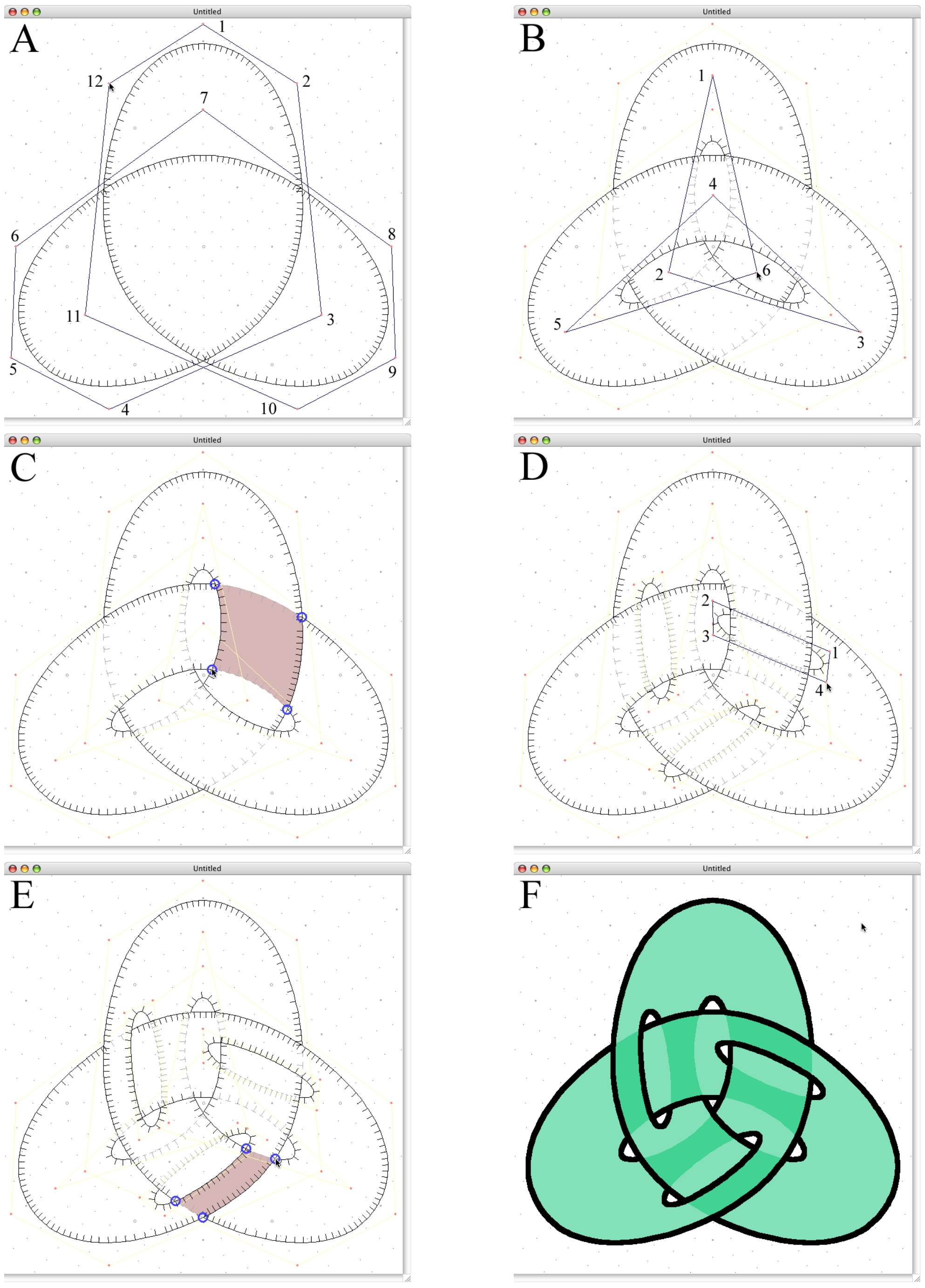

Figure 6: Demonstration of *Druid*. Spline control points are defined in either a clockwise order to create solids (*A*, numbers denote control point order) or in a counter-clockwise order to create holes (*B* and *D*). Crossings are clicked to flip overlapping surface regions (*C* and *E*). Whenever the drawing is legally labeled (*B* - *E*), the figure can be rendered (*F* renders *E*). In this example, the surface has been made partially transparent.

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

In Section 3 we described three methods by means of which images of interwoven surfaces could be constructed using conventional drawing programs. We then described how these methods could be improved upon by the use of a more general representation, *i.e.*, one which makes no assumption of layers. At the end of Section 3 we demonstrated our system, *Druid*. In this section we describe *Druid's* representation, the *labeled knot-diagram*. We then describe a set of natural constraints called the *labeling scheme*, which govern the ways in which surface boundaries appear in 2½D scenes.

4.1 Labeled Knot-Diagrams

In order to build a drawing tool that is capable of naturally representing interwoven surfaces, it is necessary to develop a fundamentally new representation for drawings. *Druid* represents the boundaries of surfaces in a localized way that does not assume surfaces' depths correspond to a DAG. It does this by exploiting the fact that local depth changes only occur at surface boundary crossings.

Our system, *Druid*, represents a 2½D scene as a *labeled knot-diagram* (see Williams [28]). 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, top left). Williams extended ordinary knot-diagrams to include a *sign of occlusion* for every boundary and a *depth index* for every boundary segment (Fig. 7, top right). 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. It is interesting to note that Williams applied knot-diagrams not to the construction of drawings representing 2½D scenes, but to the inverse problem of understanding existing 2½D scenes, *i.e.*, computer vision.

4.2 Labeling Scheme

The process of assigning a labeling to a knot-diagram is similar to Huffman's notion of *scene-labeling* (see Huffman [12]), 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. 7, bottom). There are four rules of the labeling scheme:

1. The upper boundary must have the same depth on both sides of the crossing.
2. The lower boundary must differ in depth by exactly one across the crossing.
3. The lower boundary must be deeper on the occluding side of the upper boundary.

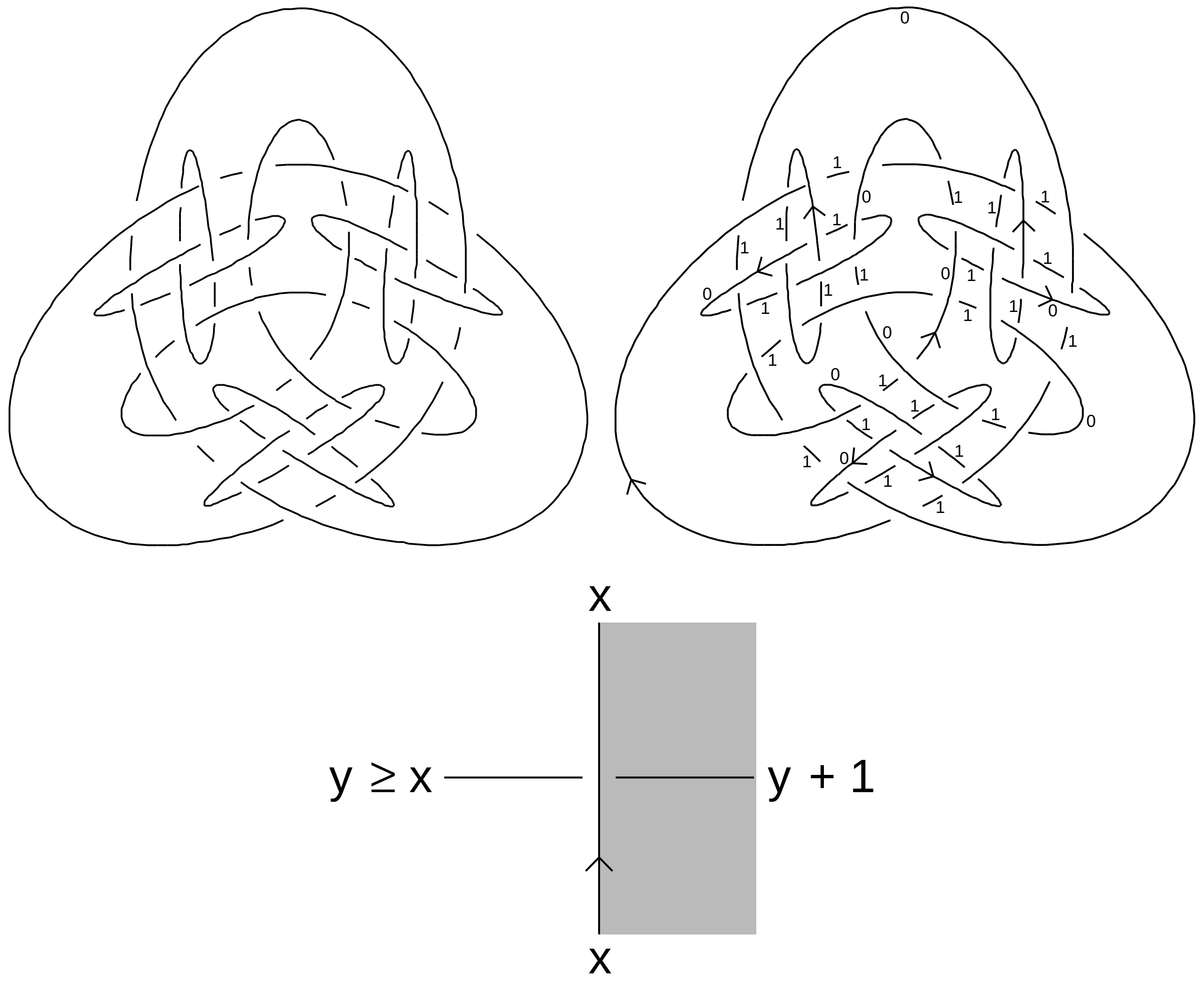

Figure 7: A *knot-diagram* (top left) is a projection of a set of closed curves onto a plane together with indications which show which curve is on top at every crossing. A *labeled knot-diagram* (top right, see Williams [28]) is a knot-diagram with a sign of occlusion for every boundary and a depth index for every boundary segment. Arrows show the signs of occlusion for the boundaries, always denoting a surface bounded to the right of a boundary with respect to travel along the boundary in the direction of the arrow. The sign of occlusion can also be denoted using hash marks. Some depth indices of depth zero have been omitted for clarity. The *labeling scheme* (bottom, see Williams [28]) is a simple set of constraints on the depths of the four boundary segments that meet at a crossing. If every crossing in a labeling satisfies the labeling scheme then the labeling is a *legal labeling* and can be rendered. Section 4.2 describes the constraints imposed by the labeling scheme.

4. The lower boundary must be no shallower than the upper boundary on the unoccluding side of the upper boundary.

With respect to the labeling scheme shown in Fig. 7 (bottom) the constraints are as follows: 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, *i.e.*, y must be greater than or equal to x.

If every crossing in a labeled knot-diagram satisfies the labeling scheme, the labeling is a *legal labeling* and accurately represents a scene of topologically valid surfaces. Legal labelings can be *rendered*, *i.e.*, translated into images (discussed in Section 11) in which the interiors of surfaces are filled with solid color.

While the labeling scheme is sufficient to describe all 2½D scenes, it does not make explicit all useful relationships between the elements of a drawing, *i.e.*, some elements of a drawing may not be entirely independent of other elements. Knowledge of such relationships could be used to improve *Druid's* performance. In Section 6, we describe one relationship between drawing elements which we call the *crossing-state equivalence class rule*. By exploiting this rule, *Druid* is able to quickly relabel drawings in a way which would otherwise be impossible. This method of relabeling is far superior to alternative methods, such as searching for a new labeling, which is described in the next section.

5 Finding a Legal Labeling

For a given knot-diagram there are numerous possible labelings. However, only a small subset of those will be legal, *i.e.*, will satisfy the labeling scheme at all crossings and therefore represent a topologically valid 2½D scene. *Druid* attempts to maintain a legal labeling at all times. The task of *labeling* a drawing consists of searching the space of all possible legal labelings for the labeling that most likely corresponds to the user's intent. However, in many situations, *Druid* does not need to discover a *new* labeling, it simply needs to *relabel* an existing drawing which has become illegal in a localized region due to a user-interaction, *e.g.*, a crossing-flip. Section 2.1 introduced a number of drawing program user-interactions. These can be grouped into three categories:

- Labeling-preserving interactions – drags or reshapes in which the topology of the knot-diagram does not change
- Interactions requiring *labeling* – interactions which create or destroy crossings, or which change the order of existing crossings along the boundaries, resulting in a topological change, *e.g.*, drags, reshapes, and sign of occlusion flips
- Interactions requiring *relabeling* – interactions which alter crossing-states or boundary segment depths, but preserve the current topology, *e.g.*, crossing-flips.

We will discuss labeling-preserving interactions in Section 7.2. In this section we describe how *Druid* handles interactions which require *labeling*; interactions which require *relabeling* are described in Section 9.2. Labeling and relabeling are almost always performed with respect to the labeling that existed prior to the interaction. In both cases, the primary goal is to find the *minimum-difference labeling* with respect to the prior labeling because we believe that this labeling will most likely match the user's intent. Both labeling and relabeling can be accomplished through a *labeling search*. However, we will show in Section 9.2 that relabeling can often be accomplished without resorting to a search, resulting in a considerable increase in efficiency.

5.1 Constraint-Propagation

The search takes the form of a *constraint-propagation* process similar to Waltz filtering (see Waltz [26]). Waltz's research illustrated how certain combinatorially complex graph-labeling problems can be reduced to unique solutions through a process called constraint-propagation. In a graph-labeling problem, when one vertex of a graph is labeled, this constrains adjacent vertices, which in turn propagate their own constraints deeper into the graph. By means of this process, it is often the case that an apparently ambiguous labeling problem can be reduced to a single consistent labeling.

5.2 Boundary Traversal During the Search

As the search process explores the labeling space, *Druid* propagates constraints through a set of *boundary traversals*. A boundary traversal begins at an arbitrary location on a boundary at a hypothesized depth and visits each segment on the boundary until it returns to the starting location. The purpose of performing a boundary traversal is to test the traversal's legality using the *zero integration rule*. This rule guarantees that a traversal ends at the same depth that it started at, which is a requirement of a legal labeling. Note that once a boundary traversal has begun, the depth changes that occur during the traversal are uniquely determined. It is in this way that depth constraints propagate around a boundary. 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 does not change.
- 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.

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. Consequently, all possible depths for the initial segment must be tested. This is accomplished by calling the boundary traversal process within a loop that enumerates the possible depths for the initial segment.

5.3 Structuring the Search

The primary goal of the search process is to find the *minimum-difference labeling* with respect to the prior labeling because we believe that this will most likely match the user's intent. The search is a depth-first search in which each of the possible depths for the initial segment of a traversal is the root of a distinct search tree. The search trees branch when a boundary reaches a crossing; each of the two crossing states is the root of a distinct subtree.

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. As boundaries are crossed, or *touched*, they are 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. The following is a description of the entire backtracking search process:

- An empty touched boundary list is seeded with all illegal boundaries. 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 a boundary, the crossed boundaries are appended to the touched boundary list. When all combinations of crossing-states for all crossings on the traversal boundary have been enumerated, a traversal is completed. If the traversal satisfies the zero integration rule, the traversal process restarts with the next boundary on the touched boundary list. If the traversal ends illegally the process backtracks to the last unflipped crossing, flips that crossing's state, and continues.
- 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 solution, *i.e.*, a legally labeled figure. The solution is assigned a score based on its difference from the prior labeling.
- When the entire depth range for the initial segment has been enumerated, the next boundary on the touched boundary list is selected. When the search is completed, the solution with the lowest difference relative to the prior labeling is accepted. The new labeling is then displayed and the user can initiate a new interaction.
- The difference between labelings discovered by the backtracking search process and the prior labeling increases whenever expanding a node in the search tree requires flipping a crossing.

5.4 Branch-and-Bound Tree Search

The search process takes the form of a branch-and-bound depth-first search in which expanding a node consists of either preserving a crossing's state or flipping it. As the search descends into the search tree, a total difference (L_Δ) with respect to the prior labeling is accumulated. Expanding a node adds a local δ of 0 (if the crossing-state is preserved) or 1 (if it is flipped) to the cumulative L_Δ. Since the L_Δ that is being accumulated during tree-descent can never decrease (δ can never be negative), we know that it represents a lower bound on the final L_Δ of all leaves in the subtree below the current node. Therefore, assuming that a solution has been found already, we can truncate the search at any internal search tree node where the current cumulative L_Δ equals this global bound. The search process then backtracks and tries a different path.

5.5 Optimizing the Search Through Iterative Deepening

To further optimize the search process, *Druid* attempts to confine changes to the drawing to the *area-of-interest*, *i.e.*, the area of the drawing near where the user is interacting. This strategy not only better reflects the user's intent, it also helps find a tighter bounds earlier. *Druid* confines changes to the area-of-interest by applying an iterative deepening loop. Given a drawing with a set of illegal crossings (*e.g.*, such as the drawing that directly precedes a labeling search), each legal crossing's graph distance in the knot-diagram to the nearest illegal crossing is calculated. To apply iterative deepening during the search process, *Druid* only considers flipping crossings whose graph distance lies within the current *horizon*. If the search terminates with no legal solutions, the horizon is increased and the search is repeated. In this way, *Druid* considers all combinations of crossing-flips within a steadily increasing horizon before considering any crossing-flips beyond the horizon.

A labeling search is required after interactions that alter the drawing's topology. However, when merely relabeling after a nontopological change, such as a crossing-flip, a labeling search is unnecessary. Instead, the minimum-difference labeling can be directly deduced. The method for directly deducing a new labeling is described later in this paper and makes use of a property of 2½D scenes which we call the *crossing-state equivalence class rule*. In fact, this rule can also be applied to the labeling search to vastly improve the efficiency of the search.

6 Crossing-State Equivalence Classes

In this section we describe a topological property of 2½D scenes which we call the *crossing-state equivalence class rule*. *Druid* uses this rule to relabel knot-diagrams represented 2½D scenes rapidly.

6.1 Definition of Key Concepts

Fig. 8 shows a 2½D scene of interwoven surfaces. A section of a boundary joining two crossings is termed a *boundary segment*. We observe that the canvas is partitioned by boundary segments into disjoint *regions*. In Fig. 8, the regions of the canvas are labeled with letters. We observe that zero or more surfaces (numbered in Fig. 8) cover every region. For example, surfaces *1* and *3* cover region *k* while surfaces *1*, *2*, and *3* cover region *m*.

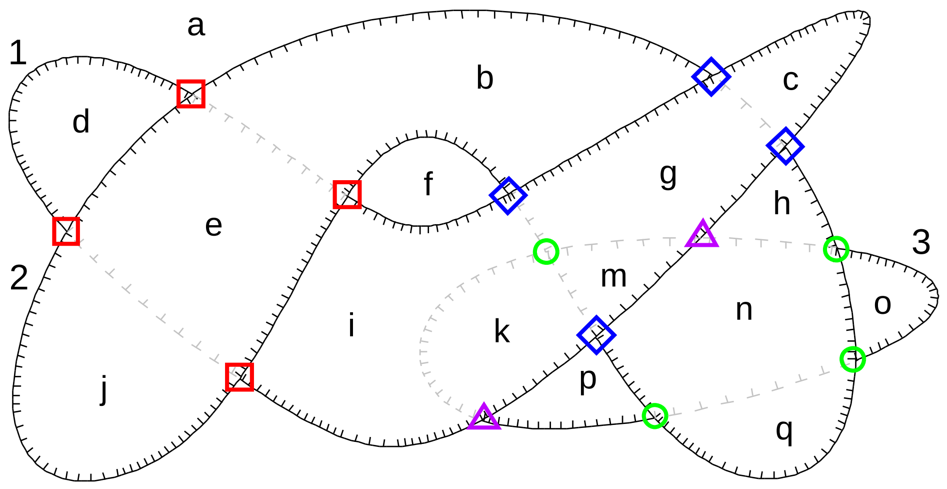

Figure 8: An interwoven 2½D scene. Regions are labeled with letters, surfaces with numbers, and crossing-state equivalence classes with shapes.

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

- A *superregion* is a set of contiguous regions covered by a single surface. For example, in Fig. 8, {*b*, *g*, *h*, *n*} is a superregion of surface *2*.
- A *border* of a superregion is the set of boundary segments which define its perimeter.
- A *shared superregion* is the maximum superregion common to two surfaces, *e.g.*, {*g*, *m*} is a shared superregion of surfaces *1* and *2*.
- A *corner* of a shared superregion is a crossing where adjacent boundary segments of the border belong to different surfaces. In Fig. 8, corners corresponding to the shared superregion {*m*, *n*} common to surfaces *2* and *3* are marked with circles.

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

6.2 Reducing General 2½D Scenes to Simple 2½D Scenes

A *simple surface* is a surface with a single boundary component which does not intersect itself, *i.e.*, a *Jordon curve*. For every 2½D scene, there is a corresponding 2½D scene where all surfaces are simple. We call a 2½D scene comprised solely of simple surfaces a *simple 2½D scene*. The fact that any 2½D scene can be reduced to a simple 2½D scene is significant because our proof of the crossing-state equivalence class rule applies only to simple 2½D scenes. Two steps are required to reduce a general 2½D scene to a simple 2½D scene. First, any surface with multiple boundary components must be converted into a surface with a single boundary component. Second, any self-overlapping surfaces must be converted into a set of non-self-overlapping surfaces (see Fig. 9).

We perform both surface conversions using *cuts*. A cut can be thought of as a scissor cut through a surface connecting two boundaries that belong to a single surface. When a cut connects two boundaries, those boundaries are joined into a single boundary component (Fig. 9, top). Likewise, a self-overlapping surface with a single boundary component can be cut into multiple smaller surfaces which abut along cuts and such that no surface self-overlaps (Fig. 9, bottom).

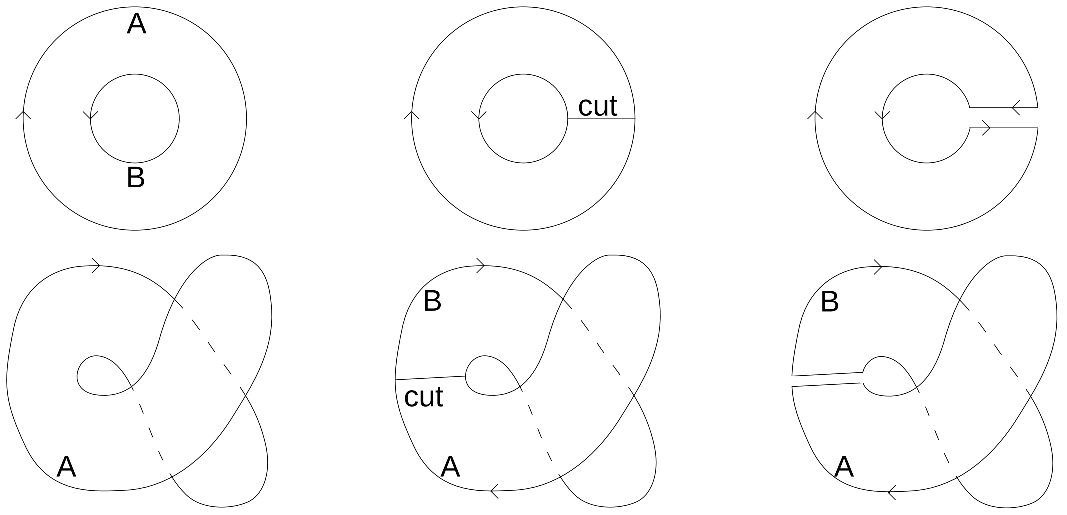

Figure 9: A *cut* is a line connecting two boundaries or connecting two locations on a single boundary. Any surface with multiple boundary components can be converted into a surface with a single boundary component by connecting them with a cut (top). Alternatively, any self-overlapping surface can be converted into a set of abutting non-self-overlapping surfaces using a cut (bottom).

6.3 The Crossing-State Equivalence Class Rule

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

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

Proof We first prove the above theorem for simple surfaces. Because any 2½D scene can be reduced to a simple 2½D scene (see Section 6.2), this suffices to prove the theorem in the general case. We begin by observing the following:

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

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

For example, in Fig. 8, consider the superregion $\{m, n\}$ shared by surfaces *2* and *3*. The only segment interior to the superregion is part of the boundary of surface *1*. Therefore, the relative depths of surfaces *2* and *3* cannot change along that boundary segment.

What the crossing-state equivalence class rule tells us is that the crossing-states of certain sets of crossings are entirely interdependent, *i.e.*, they represent a single variable in the labeled knot-diagram. We will show how *Druid* exploits this rule in Section 9.

7 Editing 2½D Scenes

7.1 2½D Scene Editors vs. Drawing Representation Editors

Unlike *Druid*, *Real-Draw* is essentially an editor for its internal representation, *i.e.*, a global DAG with localized regions specifying local DAG changes. The manipulations the user performs in *Real-Draw* equate *edit-distances* of one with *representation distances* of one. An edit-distance is the total number of mouse clicks and keystrokes necessary to transform one drawing representation into another. A representation distance is the minimum number of elemental transformations (changes in which a single parameter is altered) that are necessary to transform one representation into another. Editing the internal representation is not necessarily the best possible design for a drawing tool; there are situations where the user's intention may require navigating a large representation distance consisting of many elemental changes to achieve a small 2½D scene transformation. This long sequence of operations is tedious and distracts the user from his larger goals. The main problem is that the actions of the user do not correspond to transformations of the scene, they correspond to transformations of the drawing representation. Consequently, the affordances of *Real-Draw* are not isomorphic with those of 2½D scenes. To summarize, although *Real-Draw* represents a genuine improvement over spoofs, it still requires more operations than would be necessary using a program which provides the natural affordances of 2½D scenes.

In contrast, *Druid* provides an interface not for directly editing its internal representation, but instead of editing the thing its representation depicts, *i.e.*, 2½D scenes. *Druid's* power derives from its ability to quickly relabel a knot-diagram following a user-interaction. It presents the user with the experience of directly interacting with a 2½D scene (including scenes containing interwoven surfaces), rather than the experience of interacting with a scene that merely resembles the desired scene, *i.e.*, creating a spoof. The reason the user has such a qualitatively different experience when using *Druid* is that elemental transformations on 2½D scenes can be accomplished with single mouse clicks. It is this isomorphism between editing operations and 2½D scene transformations that makes *Druid* so novel.

Several possible user-interactions were listed in Section 2.1 and grouped into three categories in Section 5: *labeling-preserving interactions*, *interactions requiring relabeling*, and *interactions requiring relabeling*. In the next section we describe how *Druid* handles labeling-preserving interactions. We will describe how *Druid* handles interactions requiring labeling or relabeling in Section 9.

7.2 Labeling-Preserving Interactions

Ideally, *Druid* should preserve the labeling during user-interactions whenever possible because doing so provides a sense of continuity for the user. Labeling changes should only occur as the result of explicit constraints that the user specifies. At all other times, the labeling must be preserved so the user can maintain control over the drawing process.

When one boundary is dragged over another boundary, the crossings belonging to both boundaries will move. The goal of preserving the crossings' states during such interactions precludes a naive approach like deleting and rediscovering crossings since such an approach would destroy the crossing-states, thus invalidating the labeling. 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 newly assigned crossing-states will match the old crossing-states, the naive approach is infeasible. The only alternative is to avoid 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, a process termed *crossing-projection*. The goal is to explicitly compute the path that a crossing follows around a boundary. The algorithm is illustrated in Fig. 10, where the user has dragged one boundary to the right. Observe that the boundary moves in discrete jumps, 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 that spline control points only reside at pixel coordinates.

The crossing-projection algorithm is invoked each time *Druid* receives a mouse interrupt notifying it that a boundary's location or shape has been altered. All crossings that the boundary's movement affects are immediately projected to their new locations. Note that crossing-projection is performed individually on each affected crossing.

Crossing-projection is trivial if a crossing remains on the same two boundary segments after a boundary moves to its new location. If a crossing's two boundary segments still intersect after a drag or reshape is performed, then projecting the crossing is simply a matter of updating the coordinates for the crossing. If the two original segments of the crossing no longer intersect after the boundary is moved, then a more complex crossing-projection algorithm must be employed, in which a new pair of boundary segments which intersect are identified and the coordinates of the new point of intersection are calculated.

Fig. 10 illustrates how a single crossing is projected. It illustrates a case in which the projection process is complicated because the crossing has traversed many segments on both boundaries. Given a pair of segments belonging to a crossing that no longer intersect (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 boundary segments and preserving the other. The loop terminates when the currently assigned pair of boundary segments intersect.

Observe that between iterations in Fig. 10, one boundary segment is retained while the other boundary segment switches to the adjacent segment on its boundary that is closest to the current intersection. The decision about which segment to retain is made by measuring the two errors of the crossing with respect to the two segments. The error for a segment is the distance between the crossing and the nearest end of that segment. For example, at the beginning of Iteration 1, the error is less for the stationary boundary but at the beginning of Iteration 2 the error is less for the moving boundary.

The loop repeats until the pair of boundary 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.

Crossing-projection helps to avoid unnecessary relabeling, but when the knot-diagram's topology changes, relabeling is required. The relabeling method is described in Section 9.

8 Labeling Running Time

While equivalence classes can be exploited to rapidly relabel a previously labeled figure, there is also the issue of how much time is required to initially label a figure following a topological change. To address this

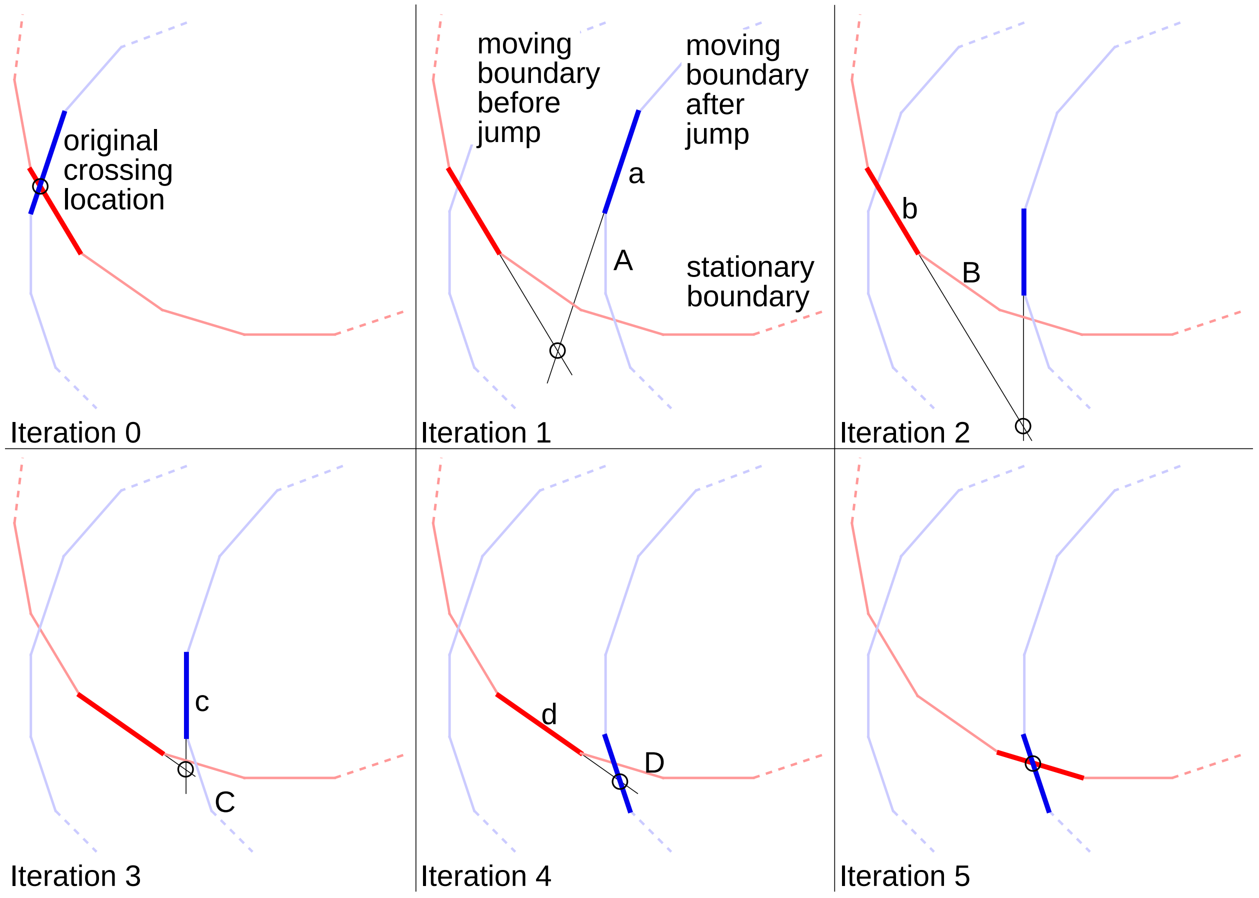

Figure 10: The sequence of drawings shown here illustrates successive iterations of the crossing-projection algorithm. Thick line segments show the boundary segment pair associated with the crossing at the beginning of each iteration. Circles show the intersections of the lines containing the boundary segments. Lowercase labels (*a* - *d*) indicate the boundary segment that will be switched out of the pair assignment at the end of that iteration. Uppercase labels (*A* - *D*) indicate the boundary segment that will be switched into the pair assignment at the end of an iteration. After a timestep, the algorithm tests the original boundary segments assigned to the crossing (shown in Iteration 1). In the example illustrated, the boundary 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 boundary segments is found that actually intersect, as shown at the beginning of Iteration 5.

question we performed an experiment in which we constructed a series of drawings of increasing complexity. The labeling time was then calculated relative to a particular drawing's complexity. To construct this series of drawings we repeatedly introduced a new surface into a drawing in a systematic manner. Fig. 11 shows the last drawing in a series of drawings used to measure labeling performance. We began with a figure consisting of only the upper left surface shown in Fig. 11 and steadily introduced new surfaces until the figure shown had been constructed. At each intermediate stage we measured the time required to label the figure after the introduction of the new surface. We conducted two experiments, one to measure randomized labeling and the other to measure incremental labeling. Randomized labeling is performed on a *zeroed* labeling, *i.e.*, one in which all boundary segment depths have been set to zero, all equivalence classes have been destroyed, and all crossing-states have been randomized. In contrast, incremental labeling preserves the prior labeling and its equivalence classes, and merely labels the new elements of a drawing, *i.e.*, new crossing-states and boundary segments, followed by discovery of new equivalence classes. We observe that incremental labeling is more characteristic of how *Druid* is actually used. Results for randomized labeling are only provided for the purpose of comparison.

Fig. 12 shows a plot of the labeling performance relative to the total number of crossings in the figure. We observe that the performance of the randomized method scales exponentially in the number of crossings. In contrast, the performance of the incremental method scales linearly. Note that these results will not necessarily be representative of *Druid's* behavior for all drawings. However, we believe that this experiment provides some indication of *Druid's* performance when labeling. Note that the number of trials conducted varies across the test set. We ran ten trials for every incremental test, but for the randomized tests, we increased the number of trials as the number of crossings increased. This was necessary because the labeling performance exhibited a high variance when there were many crossings. This high variance resulted from the fact that the labeling performance is highly dependent on the initial zeroed state. For some zeroed states the labeling process finds a solution quickly while for others, it takes a long time. In contrast, the incremental method performs precisely the same way every time, and therefore yields extremely low variances across the test set.

9 Exploiting Crossing-State Equivalence Classes

9.1 Exploiting Crossing-State Equivalence Classes as a Search Constraint

In Section 5 we described how *Druid* performs a branch-and-bound constraint-propagation search to find a new labeling following a topological change. Crossing-state equivalence classes provide an additional *non-local* constraint that can be exploited during the search process. Any partial labeling that occurs during the search which violates the equivalence class rule must be illegal and can therefore be immediately eliminated from consideration.

During the search process, crossing-states may be flipped. Without knowledge of the equivalence class rule,

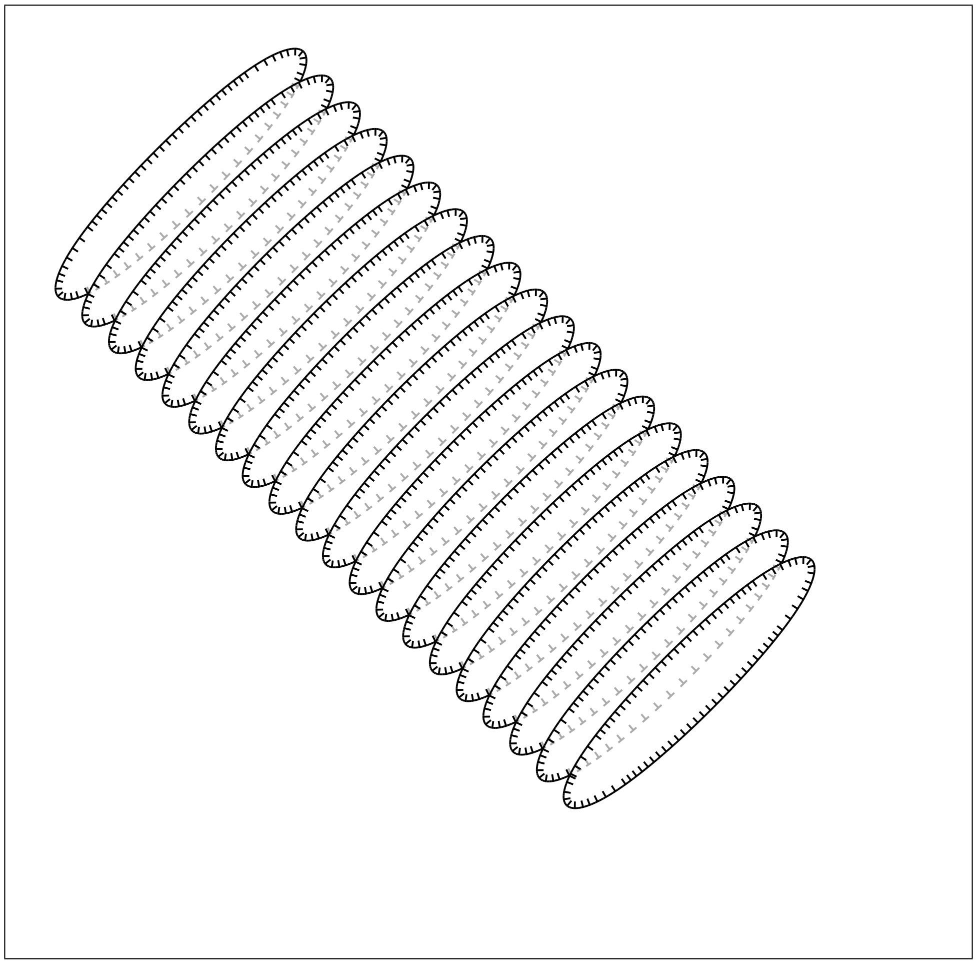

Figure 11: For the labeling running time experiment, we constructed a drawing of a single cigar-shaped surface (as if only the upper left surface were present in the figure shown). This drawing was then repeatedly modified by adding an identical cigar-shaped surface to the drawing, thus uniformly increasing the drawing's complexity. We performed this process nineteen times, resulting in twenty total drawings. The last drawing is shown.

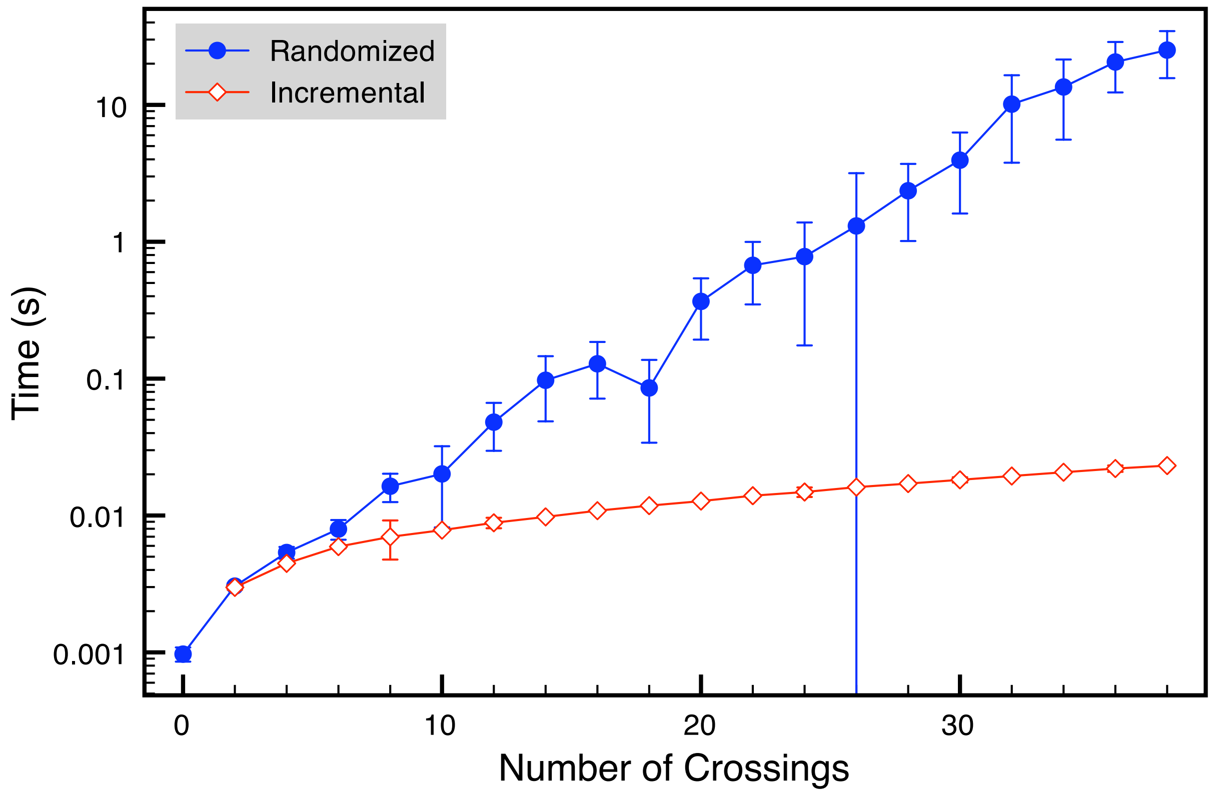

Figure 12: Labeling time vs. number of crossings. We observe that the randomized method performs worse than linear in the number of crossings but that the incremental method performs in a linear time in the number of crossings. Note that time is plotted on a log axis. The number of trials conducted varies across the test set. For all incremental tests, ten trials were conducted. For randomized tests with zero to eighteen crossings, ten trials were conducted, for those with twenty to thirty crossings, thirty trials were conducted, and for those with thirty-two to thirty-eight crossings, seventy trials were conducted. We increased the number of trials for the randomized tests in which there were many crossings because the variance tended to be very high in those experiments. This high variance was due to the large number of possible zeroed states the labeling process could start from, *i.e.*, a labeling process starting from one zeroed state might find a solution must faster than a labeling process starting from a different zeroed state. In contrast, the incremental method performs precisely the same way every time, and therefore yields extremely low variances across the test set. Errors bars show 95% confidence interval.

this method of search would spend a lot of time considering labelings that are guaranteed to be illegal due to a violation of the equivalence class rule. To use equivalence classes as a constraint, *Druid* flips equivalence classes as a unit during the search process. If expanding a node in the search tree requires flipping a crossing, all crossings in that crossing's equivalence class are immediately flipped before proceeding with the search process. In this manner, it is guaranteed that the partial labeling being considered conforms to the crossing-state equivalence class rule at all times during the search. In practice, this additional constraint provides significant gains to the search performance. In some situations, the search terminates 50 times faster as a result of this constraint.

9.2 Exploiting Crossing-State Equivalence Classes to Directly Deduce a New Labeling

Some user-interactions, such as crossing-flips, do not result in a topological change. Following such interactions, the minimum-difference labeling can be directly deduced without resorting to a labeling search. *Druid* directly deduces the new labeling by propagating boundary segment dept-changes through the knot-diagram away from the flipped equivalence class. In Section 5.5 we stated that it is preferable to confine the relabeling of boundary segment depths to the area-of-interest, the local area around a user-flipped equivalence class. Propagating depth-changes away from flipped crossings rather than globally relabeling all boundary segment depths will accomplish this goal.

Fig. 13 shows the two crossing-states for a single crossing. Observe that a crossing always has two *potentially occluded* segments which may or may not be occluded at the crossing by the other boundary, depending on the crossing-state (*C* and *D*), and two *unoccluded* segments which can never be occluded at the crossing by the other boundary (*A* and *B*). When a crossing is flipped, the depths of its two potentially occluded segments will always change and the depths of its two unoccluded segments will never change. Since the depths of the unoccluded segments do not change, the new depths for the potentially occluded segments can be deduced directly by applying the labeling scheme to the flipped crossing-state and the two unoccluded boundary segment depths. After deducing the new depth for a boundary segment, that boundary segment's depth is fixed and cannot be changed again during the propagation process. We say that such a boundary segment is *depth-constrained*. This constraint guarantees that the depth propagation process always converges.

Druid's relabeling method processes crossings in a FIFO [2] queue. This queue is initially seeded with all crossings in the flipped equivalence class. For each crossing in the queue, *Druid* assigns new boundary segment depths to some of its four boundary segments in order to make the crossing legal. As described above, the assignment of new boundary segment depths to the members of the equivalence class are uniquely determined.

When the relabeling process assigns a new depth to a boundary segment, the propagation process must propagate across that boundary segment to the next crossing. Thus, the next crossing is added to the queue.

[2] "first in, first out"

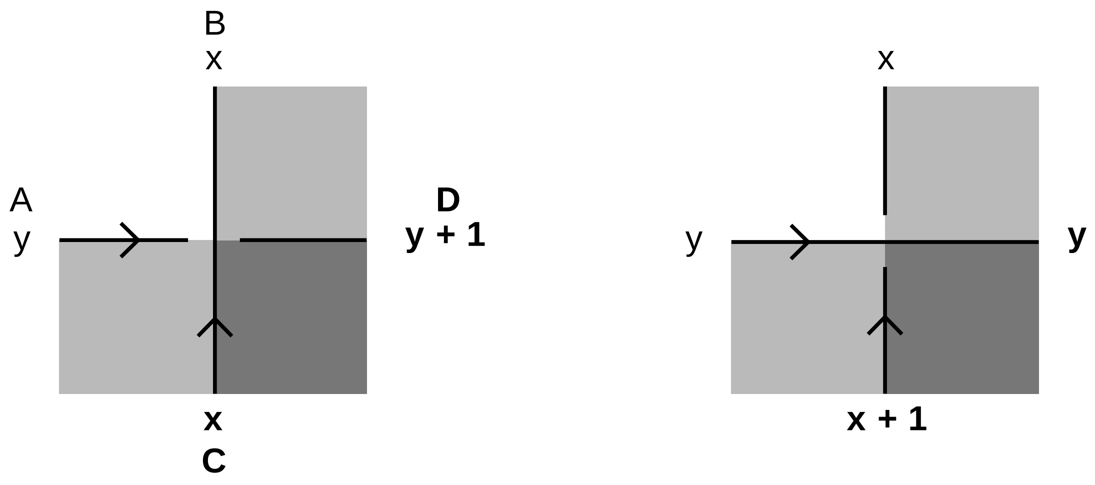

Figure 13: For the two boundaries meeting at a crossing, one half of each boundary will lie in the *unoccluded* half-plane of the opposing boundary (*A* and *B*), and the other half will lie in the *potentially occluded* half-plane of the opposing boundary (*C* and *D*). When a crossing is flipped, *e.g.*, transformed from the state shown at left to the state shown at right, the depths of its two potentially occluded segments will always change (labeled in bold) and the depths of its two unoccluded segments will never change.

The effect of processing the propagation in a FIFO queue is that changes occur near all members of the equivalence class equally early in the propagation process and then expand outward.

As new crossings are retrieved from the queue, some of the four boundary segments incident at those crossings will be depth-constrained, as described above. The one exception to this rule will be the members of the equivalence class, which will not have any depth-constrained boundary segments. However, as described above, their new boundary segment depths will be uniquely determined. For all other crossings, the effect of the propagation process is that at least one boundary segment will be depth-constrained. The unconstrained depths of a crossing that the propagation process reaches are uniquely deduced by applying the labeling scheme to the crossing's state and the depth-constrained boundary segment depths.

10 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. Although boundary groups are not required for *Druid* to legally label a drawing, they can provide a basis for translation of a surface with multiple boundaries, and more importantly, can be used to eliminate ambiguities about which surfaces boundaries belong to. For example, in Fig. 9 (top 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*. If a user were to attempt to place a third boundary that overlaps the ambiguous surface of *A* and *B*, 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 expectations about the effects of his interactions. Grouping boundaries can minimize this kind of problem.

Druid automatically finds and maintains boundary groups without requiring any input from the user. It does this by finding and maintaining *cuts* (see Section 6.2). If a cut can be found between the two boundaries, as shown in Fig. 9 (top), then the two boundaries are demonstrably part of the same surface and can be grouped.

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.

11 Rendering

Rendering consists of converting the labeled knot-diagram (Fig. 7, top right, and Fig. 6, *E*) 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. 2, *a*). 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 [16]), can be applied (Fig. 2, *g*, and Fig. 6, *F*).

12 Future Work

12.1 Labeling with Crossing-State Equivalence Classes

Because equivalence classes can only be found on a legally labeled figure, *Druid* cannot use those equivalence classes to label the figure initially. At any given time, *Druid* knows the equivalence classes that were present in the drawing at the conclusion of the previous labeling attempt but does not have knowledge of any equivalence classes created as a result of subsequent topological changes. However, even making limited use of equivalence classes during labeling is highly beneficial because doing so significantly reduces the complexity of the drawing. The necessity of having a legal labeling before equivalence classes can be found can be problematic. For example, there are drawings for which the user must construct a complex configuration of surface boundaries which cannot be incrementally labeled because no intermediate configurations leading up to the goal configuration are legal. Consequently, *Druid* must label the final drawing without having any knowledge of the equivalence classes in advance. Fig. 2 (*i*) shows such a drawing. Such a labeling process can be prohibitively expensive because the drawing complexity may be very large. If *Druid* could find equivalence classes on an unlabeled figure, then it could fully apply equivalence classes to the labeling process.

Finding equivalence classes for an unlabeled *simple* scene is relatively easy to do. We have already devised an algorithm that explores an unlabeled knot-diagram to discover adjacent corners of an equivalence class (see Fig. 8). By taking the transitive closure of a set of adjacent corners, an equivalence class can be established. Since this algorithm only works on simple scenes, finding equivalence classes on an unlabeled general scene is tantamount to converting that general scene to a simple scene. In Section 6.2 we described how general scenes can be converted into simple scenes by introducing cuts to split self-overlapping surfaces into sets of abutting non-self-overlapping surfaces. While this method of converting general scenes to simple scenes is valid in principal, we have yet to devise a practical method for finding the correct cuts to accomplish this transformation. A crucial step toward in design of a system which can find equivalence classes on an unlabeled figure is devising a method for finding the cuts which will convert a general scene into a simple scene. Our preliminary research on this conversion process suggests that such cuts might have to follow a curved path, whereas the cuts discussed previously have been exclusively straight.

13 Conclusion

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

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

Specific contributions of this work are as follows:

- Use of labeled knot-diagrams as the basis for a more general drawing tool capable of representing drawings of interwoven surfaces

- Development of a method for projection of the locations of crossings of surface boundary components after move and reshape interactions
- Introduction of the notion of cuts for representing surfaces with multiple boundary components and for reduction of the search space
- Introduction of the notion of slices for determining which surfaces contribute color to each region of the canvas for the purpose of rendering
- Discovery of a topological property of 2$^1/_2$D scenes which we call the crossing-state equivalence class rule
- Development of a relabeling method which exploits the crossing-state equivalence class rule to rapidly relabel a figure.

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

Additionally, *Druid's* interface provides the natural affordances of 2$^1/_2$D scenes in that actions that the user performs are isomorphic to elemental transformations of 2$^1/_2$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.

References

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

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

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

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

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

[6] Clanbadge's True Type fonts which represent square sections of a Celtic knotwork pattern, ©Clanbadge 2006.

http://www.publishingperfection.com/clanbadge/

[7] Cordier, F., and H. Seo, Free-Form Sketching of Self-Occluding Objects, *IEEE Computer Graphics and Applications*. 2007.

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

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

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

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

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

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

[14] Karpenko, O., SmoothSketch: 3D free-form shapes from complex sketches, *ACM SIGGRAPH*, 2006.

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

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

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

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

[19] 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.

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

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

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

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

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

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

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

[27] Wiley, K. and L. R. Williams, Representation of interwoven surfaces in 2 1/2 D drawing, *Proc. of CHI*, 2006.

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

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