Use of Crossing-State Equivalence Classes for Rapid Relabeling of Knot-Diagrams Representing 2½ D Scenes

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

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

April 19, 2006

Abstract

In previous research, we demonstrated a sophisticated computer-assisted drawing program called *Druid*, which permits easy construction of *2½D scenes*. A 2½D scene is a representation of surfaces that is fundamentally two-dimensional, but which also represents the relative depths of those surfaces in the third dimension. This paper greatly improves *Druid's* efficiency by exploitating a topological constraint on 2½D scenes which we call a *crossing-state equivalence class*. This paper describes this constraint and how it is used by *Druid*.

1 Introduction

Conventional drawing programs [1, 2, 4, 5] rely on the use of layers for ordering the surfaces in a drawing from top to bottom. This approach 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. Our research focuses on the development of a novel representation for drawings 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.

2 Overview of this Paper

In this paper, we first describe an important limitation of conventional drawing programs. We then describe a novel representation for surfaces which has significant advantages when compared to those used in existing drawing programs and explain how our program, *Druid*, uses this representation (see Wiley and Williams [7]).

We then describe a previously unrealized topological constraint on 2½D scenes. This constraint is termed a *crossing-state equivalence class*. Exploitation of crossing-state equivalence classes permits us to significantly improve *Druid's* performance, thus increasing the complexity of drawings that a user can construct. To simplify our exposition, where it is important to distinguish between the version of *Druid* described in [7] and the version of *Druid* described in this paper, we use *Druid (*OLD*)* and *Druid (*NEW*)*, respectively.

3 Computer-Assisted Drawing

One function of a drawing program is to allow the creation and manipulation of drawings of overlapping surfaces, which we call *2½D scenes*. A 2½D scene is a representation of surfaces that is fundamentally two-dimensional, but which also represents the relative depths of those surfaces in the third dimension. Our program, called *Druid*, permits the construction of interwoven 2½D scenes [7]. To accomplish this, *Druid* uses *labeled knot-diagrams* to represent surfaces (see Williams [8]).

Using existing programs, a drawing can easily be created in which multiple surfaces overlap in various *regions*. When multiple surfaces overlap, the program must have a means of representing which surface is on top for each overlapping pair of regions. Existing drawing programs solve this problem by representing drawings as a set of 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 implies that the surfaces relative depth relation is a directed acyclic graph (DAG). No subset of surfaces can interweave because this would require a cycle in the graph representing the relative depth relation. (Fig. 1). Because the representation employed by these programs does not span the full space of 2½D scenes, they preclude many common drawings which a user may wish to construct (Fig. 2).

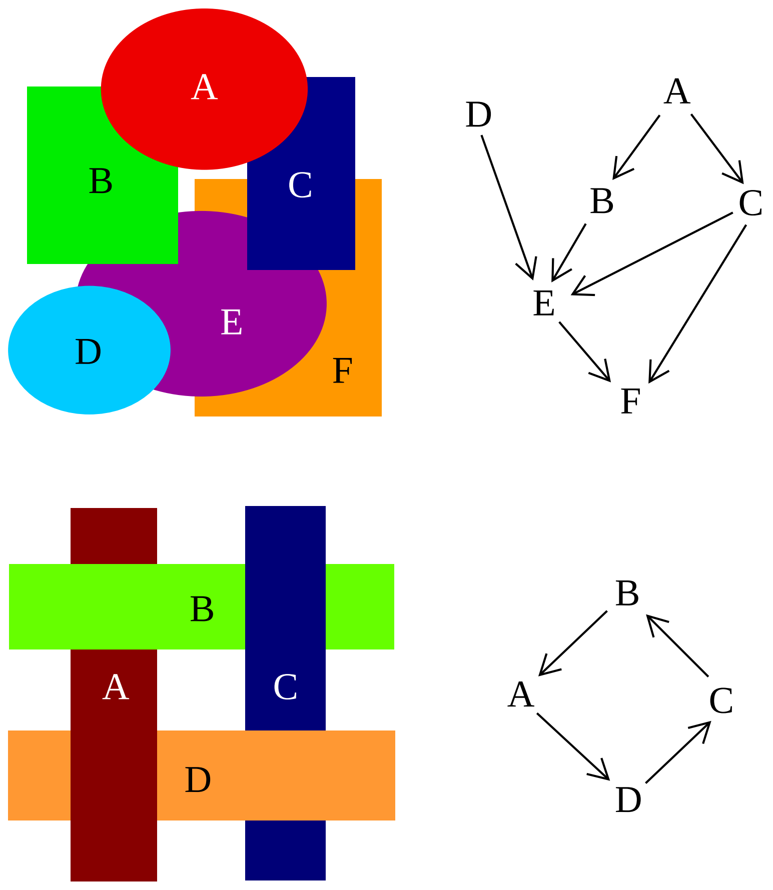

Figure 1: The classic approach to representing relative surface depths is to assign the surfaces to distinct layers (top left). It follows that the surface relative depth relation is a directed acyclic graph (DAG). No subset of surfaces can interweave because this would require a cycle in the graph (top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

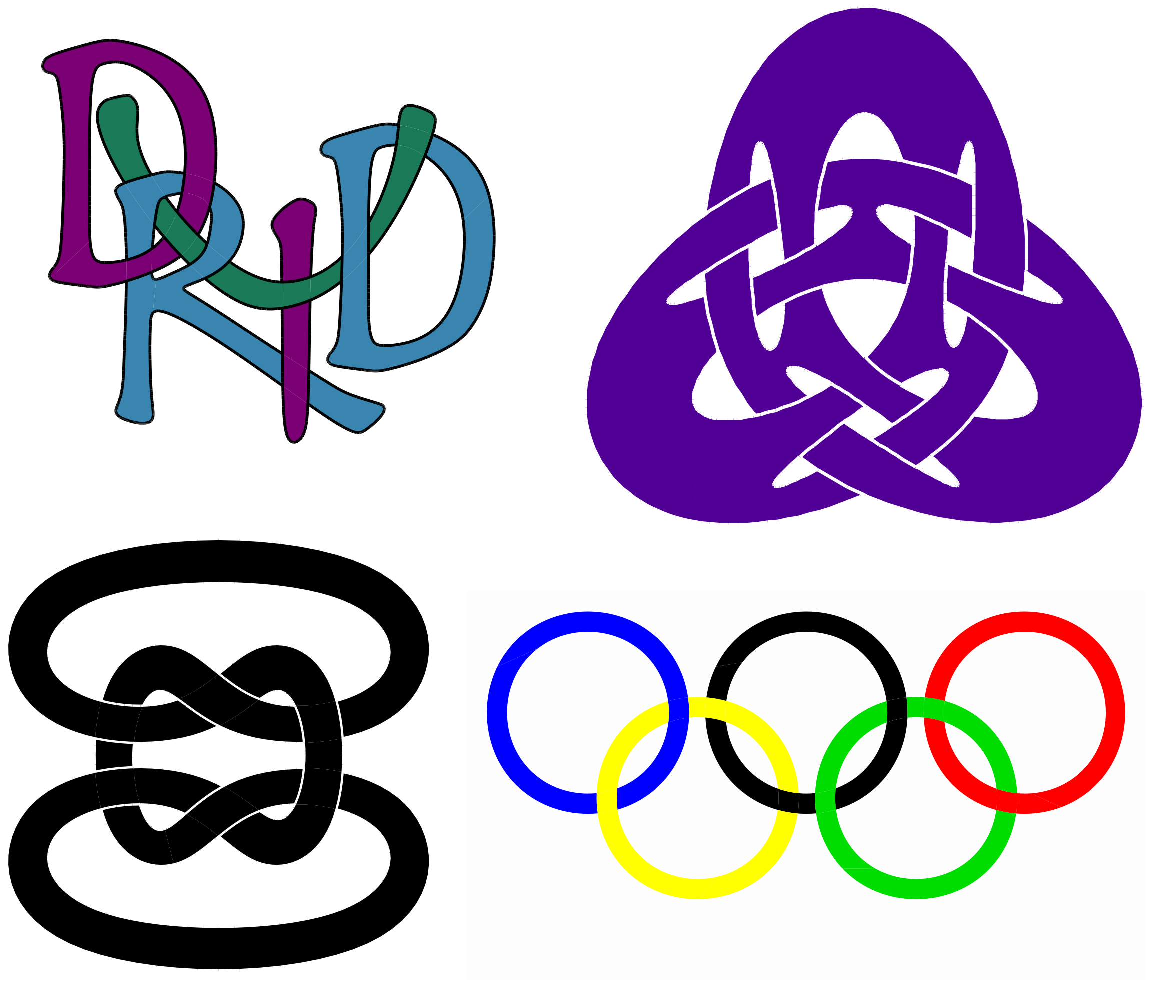

Figure 2: *Druid* permits the construction of drawings of interwoven surfaces, such as those shown here.

4 Labeled Knot-Diagrams

Druid differs from conventional drawing programs in that it permits the construction of interwoven scenes. In order to build such a tool, it was 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, *Druid* represents the *boundaries* of surfaces and is not concerned with the regions interior to a surface until the final rendering step.

Druid represents a 2$^1/_2$D scene as a *labeled knot-diagram* [8]. A *knot-diagram* is a projection of a set of closed curves onto a plane and indicates which curve is above wherever two intersect (Fig. 3, top). Williams extended ordinary knot-diagrams to include a *sign of occlusion* for every boundary and a *depth index* for every boundary segment (Fig. 3, bottom). The sign of occlusion can be illustrated with an arrow denoting a bounded surface to the right with respect to a traversal of the boundary in the arrow's direction. Alternatively, it can be denoted using a series of hash marks on the occluding side of the boundary.

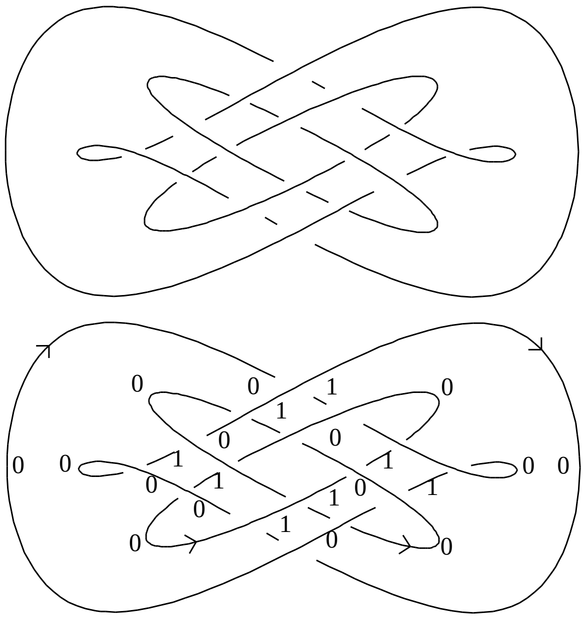

Figure 3: A *knot-diagram* (top) is a projection of a set of closed curves onto a plane together with indications of which is on top at every crossing. A *labeled knot-diagram* (bottom, see Williams [8]) is a knot-diagram with a sign of occlusion for every boundary and a depth index for every boundary segment. Arrows show the signs of occlusion for the boundaries, always denoting a surface bounded to the right of a boundary with respect to a traversal of the boundary in the direction of the arrow. The sign of occlusion can also be illustrated with a series of hash marks on the occluding side of the boundary.

Druid uses a combination of branch-and-bound search and constraint propagation (see Waltz [6]) to assign a labeling to a knot-diagram. This is called *labeling* a figure. A problem closely related to labeling is *relabeling*, in which one labeled figure is transformed into another related figure satisfying an additional constraint. When relabeling, *Druid (*OLD*)* used a highly optimized tree-search of the space of possible labelings to find the best labeling, *i.e.*, the *minimum-difference labeling* with respect to the labeling that existed prior to the search [7].

The process of labeling a knot-diagram is analogous to Huffman's *scene-labeling* (see Huffman [3]), in which he developed a system for labeling the edges of a scene of stacked blocks. In *Druid*, the labeling consists of signs-of-occlusion, crossing-states, and segment depth indices. The *labeling scheme* is a set of local constraints on the relative depths of the four boundary segments that meet at a crossing (Fig. 4). If every crossing in a labeled knot-diagram satisfies the labeling scheme, the labeling is a *legal labeling* and represents a scene of topologically valid surfaces. Legal labelings can be rendered, *i.e.*, translated into images in which the interiors of surfaces are filled with solid color.

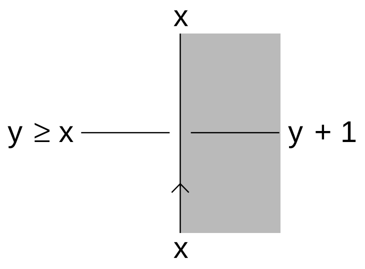

Figure 4: The *labeling scheme* (see Williams [8]) is a set of constraints on the depths of the four boundary segments that meet at a crossing. If every crossing in a labeling honors the labeling scheme then the labeling is *legal* and can be rendered. The upper boundary must have the same depth, *x*, on both sides of the crossing. If the lower boundary is at depth *y* in the unoccluded region, then it must have a depth of *y + 1* in the occluded region (shown 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*.

*Druid (*OLD*)* supported a user interaction which we termed a *crossing flip*. A crossing flip is an interaction in which the user inverts the relative ordering of two surfaces within a region of overlap. Following a crossing flip, *Druid* must quickly relabel the drawing. A fast response time is crucial to the quality of the user's experience, so *Druid* must relabel the drawing as quickly as possible. *Druid (*OLD*)* would relabel the knot-diagram by searching for the new labeling that most likely matches the user's intent. However, we have subsequently discovered a property of 2½D scenes which we call the *crossing-state equivalence class rule* which states that a labeled knot-diagram contains sets of crossings, *i.e.*, *crossing-state equivalence classes*, which are constrained to flip as units during any relabeling. *Druid (*NEW*)'s* method of relabeling exploits the crossing-state equivalence class rule to directly deduce the new labeling without performing a search. Consequently, the new method is much faster than the old method.

5 Demonstration of *Druid*

Fig. 5 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*, numbers denote control point order) or in a counter-clockwise order to create *holes* (*B* and *D*). Crossings are clicked to perform a crossing flip (*C* and *E*). Whenever the drawing is legal (*B*-*E*) it can be *rendered* (*F* renders *E*).

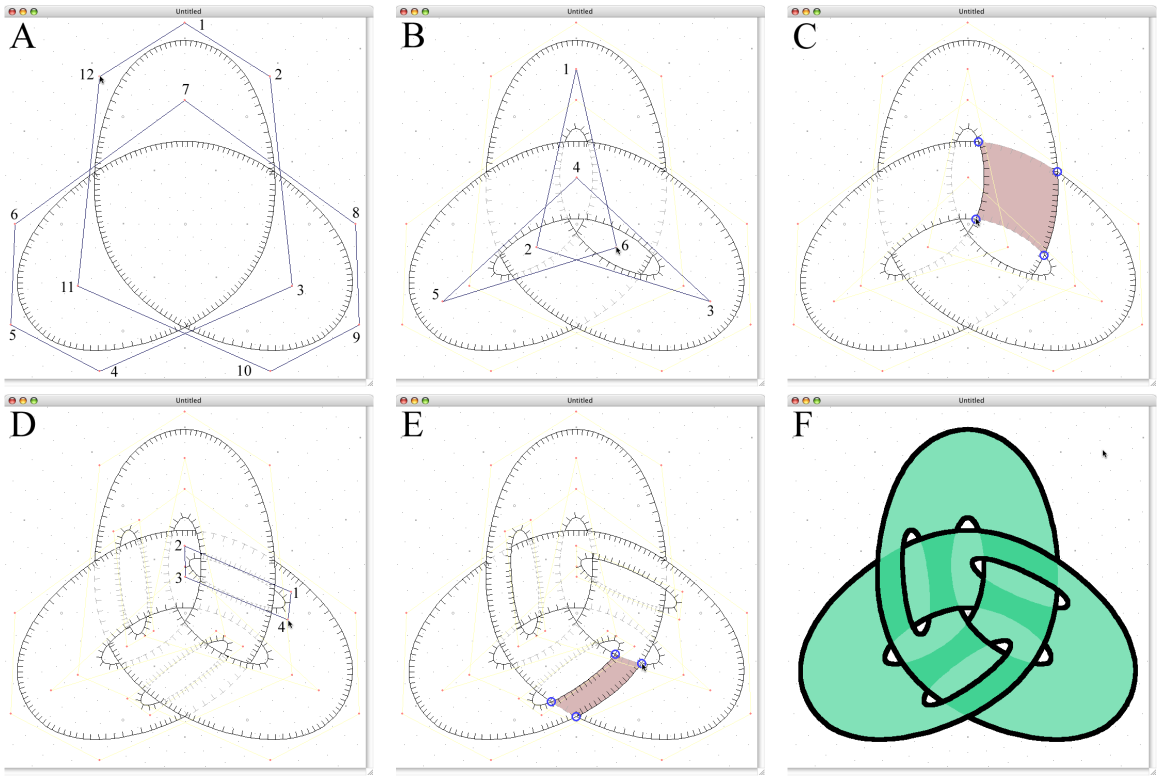

Figure 5: Demonstration of *Druid*. Spline control points are defined in either a clockwise order to create solids (*A*, numbers denote control point order) or in a counter-clockwise order to create holes (*B* and *D*). Crossings are clicked to flip overlapping surface regions (*C* and *E*). Whenever 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.

Note that there is a natural logic to the operations illustrated in Fig. 5. For example, to alter the depth ordering of various overlapping regions, 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. To achieve a similar transformation in other drawing programs, the user would have to perform less natural actions [1, 2, 4, 5].

6 Finding a Legal Labeling

In *Druid (*OLD*)*, user interactions that caused changes to the knot-diagram's topology required a search for a new legal labeling. The new labeling was the *minimum-difference labeling* with respect to the labeling that preceded the user's interaction. We wanted this behavior because we believe that the minimum-difference labeling is the most likely labeling to match the user's intent. Devising an algorithm to find the minimum-difference labeling quickly is difficult because the search space may be extremely large relative to the complexity of the drawing.

When the user interacts with the drawing and invalidates the current labeling, *Druid* must find a minimum-difference legal labeling as quickly as possible. *Druid* can find the minimum-difference labeling using either of two methods. *Druid (*OLD*)* performed a search of the space of all possible labelings. In contrast, *Druid (*NEW*)* directly deduces the result of a crossing flip by flipping all of the crossings in the equivalence class containing the crossing as a unit and propagating the necessary depth changes through the knot-diagram.

When the user clicks on a crossing to flip its crossing-state, the user imposes a constraint that is inconsistent with the present labeling. This user specified change will result in additional non-specified changes in the labeling, *e.g.*, to the states of other crossings and/or the depths of boundary segments. The search takes the form of a *constraint-propagation* process similar to Waltz filtering (see Waltz [6]). 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.

7 Crossing-State Equivalence Classes

Previously, *Druid* performed a search whenever a new labeling was required [7]. Despite a number of optimizations intended to speed up the search, *Druid* remained inherently limited in the complexity of drawings that it could handle. Drawings exceeding a certain degree of complexity required unacceptably long search times. In this paper, we describe a new constraint on 2½D scenes which when exploited, improves *Druid's* performance significantly. Consequently, users can construct much more complex drawings than they could previously.

The remainder of this paper describes a topological property of 2½D scenes which we call the *crossing-state equivalence class rule*. This property can be exploited by *Druid* to relabel a labeled knot-diagram without performing a search.

8 Definition of Key Concepts

Fig. 6 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 into disjoint *regions* separated by boundary segments. In Fig. 6, the regions of the canvas are labeled with letters. We observe that every region is covered by zero or more surfaces (numbered in Fig. 6). For example, region *k* is covered by surfaces *1* and *3* while region *m* is covered by surfaces *1*, *2*, and *3*.

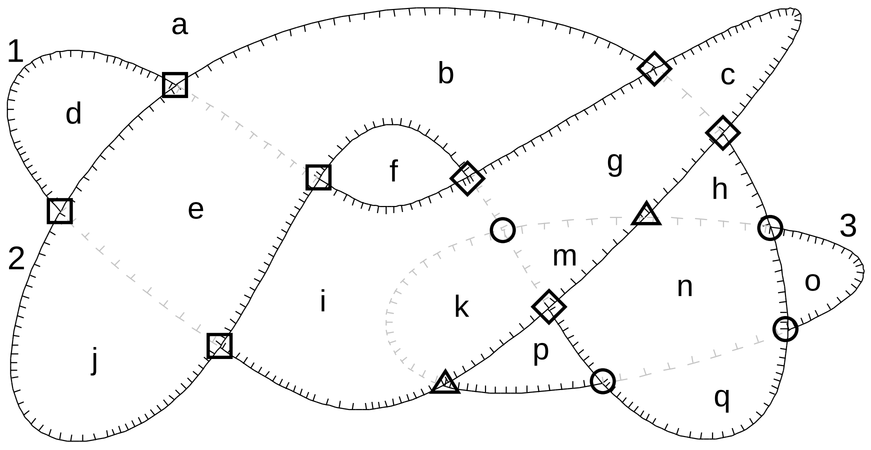

Figure 6: An interwoven 2½D scene. Regions are labeled with letters, surfaces with numbers, and crossing-state equivalence classes with shapes.

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. 6, {*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. 6, 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.

9 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*. Two steps are required to reduce a general 2½D scene to a simple 2½D scene. First, any surface with multiple boundary components (a surface containing holes) must be converted into a surface with a single boundary component. Second, any self-overlapping surfaces must be converted into a set of non-self-overlapping surfaces.

We perform both surface conversions using *cuts* [7]. A cut is analogous to a scissor cut through a surface from one boundary to another. When two boundaries are connected by a cut, they are joined into a single boundary component (Fig. 7). Likewise, a self-overlapping surface with a single boundary component can be cut into multiple smaller surfaces which abut and such that no surface in the final scene self-overlaps (Fig. 8).

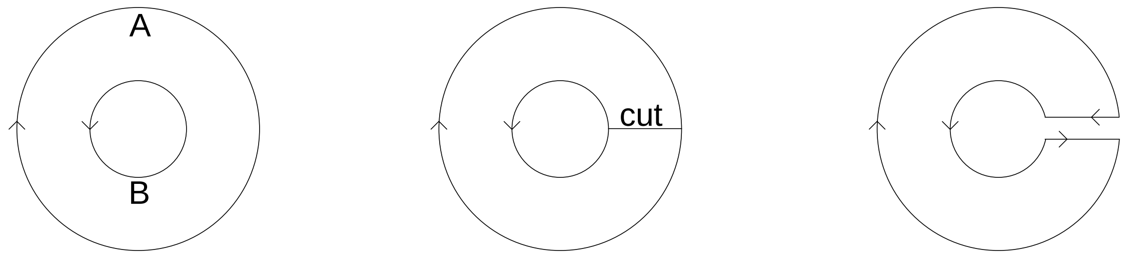

Figure 7: A cut connects two boundaries of a single surface into a single boundary for that surface.

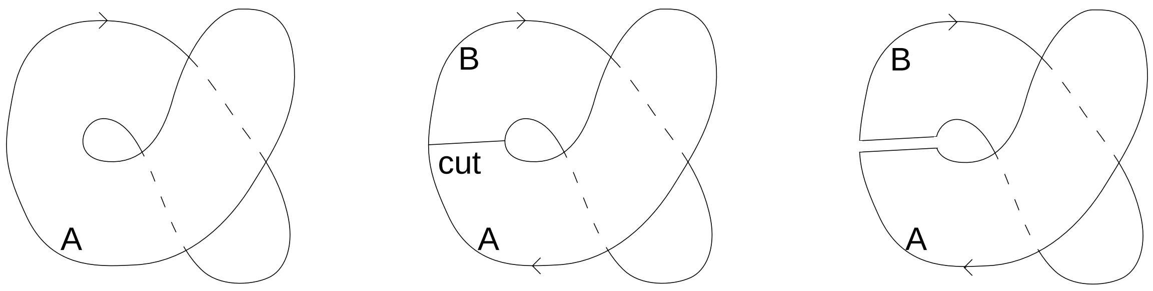

Figure 8: A cut connects two locations on the same boundary to break the boundary into two boundaries and the surface into two surfaces.

10 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 general 2½D scene can be reduced to a simple 2½D scene, this suffices to prove the theorem in the general case. We begin by observing the following:

- We observe that for every region there is a total depth ordering of the surfaces which cover that region.
- The total depth ordering of adjacent regions is identical except for the addition or deletion (depending on the sign of occlusion) of the surface whose boundary segment separates the two regions.

- It follows that the relative depth of two surfaces in adjacent regions remains the same if the boundary segment which divides the regions belongs to neither surface.
- It follows that the relative depth of two surfaces is constant within a shared superregion.
- The relative depth of the two surfaces whose boundaries intersect at a crossing is the same as the relative depth of those surfaces in the region they corner.

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

For example, in Fig. 6, 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.

11 Finding Equivalence Classes

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

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

Every crossing is associated with two *unoccluded* segments which cannot be occluded by the crossing regardless of the crossing-state, and two *potentially occluded* segments, one of which will be occluded and the other unoccluded depending on the crossing-state (Fig. 9).

In Fig. 10 two boundaries cross at *A*, the *traversal boundary* and the *crossing boundary*; this distinction is arbitrary. *Druid* searches for one of *A's* two neighbors by moving away from *A* along the potentially occluded segment of the traversal boundary. Before the traversal begins, *Druid* initializes the *target crossing boundary depth* with the crossing boundary's depth at *A*. During the traversal, the target crossing boundary depth is modified as the traversal goes under and comes out from under surfaces encountered at crossings. The traversal ends when it reaches the neighboring corner of the shared superregion. The neighboring corner is identified using the following criteria:

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

The first crossing the traversal finds that satisfies all three criteria is the first neighbor of *A*. By switching the role of traversal boundary and crossing boundary at *A*, the second neighbor of *A* is found.

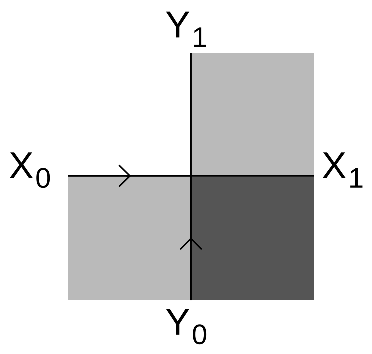

Figure 9: This figure shows a crossing with an unlabeled crossing-state. We observe that every crossing involves four boundary segments (X_0, X_1, Y_0, and Y_1). Two segments are always *unoccluded* regardless of the crossing-state (X_0 and Y_1) and two segments are *potentially occluded* (X_1 and Y_0). Only one of the two potentially occluded segments is actually occluded at any given time. Which of the two potentially occluded segments is actually occluded depends on the crossing-state that is ultimately assigned to the crossing.

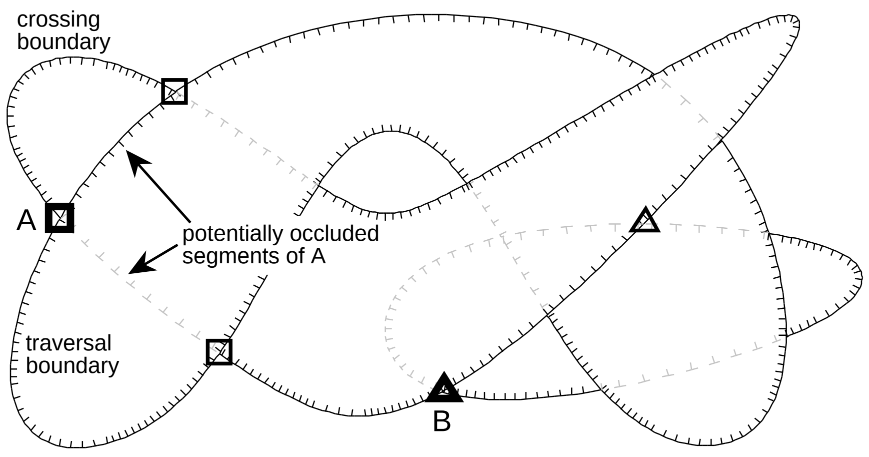

Figure 10: Crossing *A* (thick square) and its neighbors (thin squares). Crossing *B* (thick triangle) and its neighbor (thin triangle). A crossing's neighbors are the two corners of the crossing's shared superregion which precede and follow it on the border. Note that for crossing *B*, the neighbor that precedes it is the same as the neighbor that follows it, *i.e.*, it has only one neighbor (thin triangle).

12 Equivalence Class Independence

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

Fig. 11 shows a simple scene of three overlapping disks. This particular scene can be represented using a DAG, which will aid our discussion. Since there are three surfaces, and each surface is an element of a partially ordered set, there are only six possible DAGs that the surfaces of the drawing can assume: $1 \rightarrow 2 \rightarrow 3$, $1 \rightarrow 3 \rightarrow 2$, $2 \rightarrow 1 \rightarrow 3$, $2 \rightarrow 3 \rightarrow 1$, $3 \rightarrow 1 \rightarrow 2$, and $3 \rightarrow 2 \rightarrow 1$.

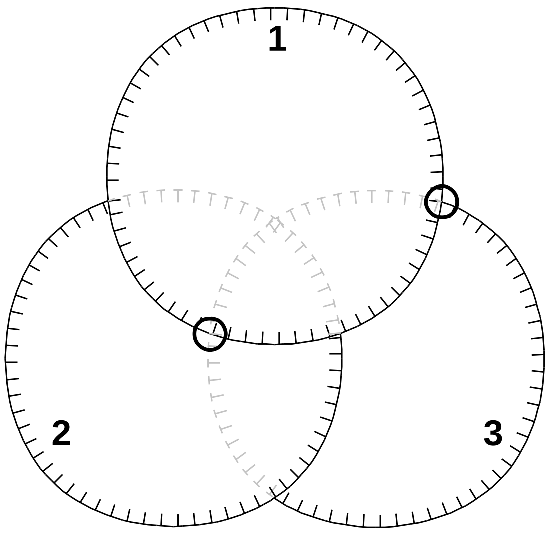

Figure 11: This figure shows a simple scene consisting of three overlapping disks. The surface relative depth relation for this drawing can be represented as a DAG. There are only six DAGs that can describe the relative depth relation for this drawing, but there are three equivalence classes, which naively suggests that there ought to be 2^3 (or eight) equivalence class instantiations for the drawing. This discrepancy is due to the fact that two of the equivalence class instantiations represent illegal labelings.

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

In Fig. 11, the marked equivalence class cannot be flipped without also flipping one of the other two equivalence classes so that the knot-diagram has a legal labeling. However, flipping either of the other two equivalence classes would constitute a valid solution to the problem, and each would produce a different result. The two possible results of flipping the marked equivalence class are shown in the two bottom drawings of Fig. 12. Flipping one of the equivalence classes results in the partial ordering 3 → 1 → 2, shown at the bottom left of Fig. 12. Flipping the other results in 2 → 3 → 1, shown at the bottom right.

13 Atomic vs. Nonatomic Crossing-State Equivalence Class Flips

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

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

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

We can avoid the ambiguity inherent in nonatomic flips by exploiting the fact that any nonatomic flip can be decomposed into a sequence of atomic flips, each of which is unambiguous, *i.e.*, there is only one way to interpret the user's intent. *Druid (*NEW*)* forces the user to perform a nonatomic flip by performing a sequence of atomic flips instead. Fig. 12 shows how this is done. An attempt to flip the marked equivalence class in the top drawing would be nonatomic, since the result cannot be legally labeled. There are two possible intended outcomes, each of which requires flipping one of the other equivalence classes in the drawing while leaving the third equivalence class unflipped. The two possible outcomes are shown at the bottom of the figure. *Druid* cannot know which outcome actually corresponds to the user's intent, and thus cannot perform the flip specified by the user without also producing a possibly unintended result. However, each outcome can be accomplished by a sequence of two atomic flips. The first atomic flip is shown in the smaller intermediate drawings. The second atomic flip corresponds to the equivalence class the user originally intended to flip, which will have become atomic as a result of the intermediate flip.

When the user attempts to perform a nonatomic flip, *Druid (*NEW*)* does not perform the flip, but

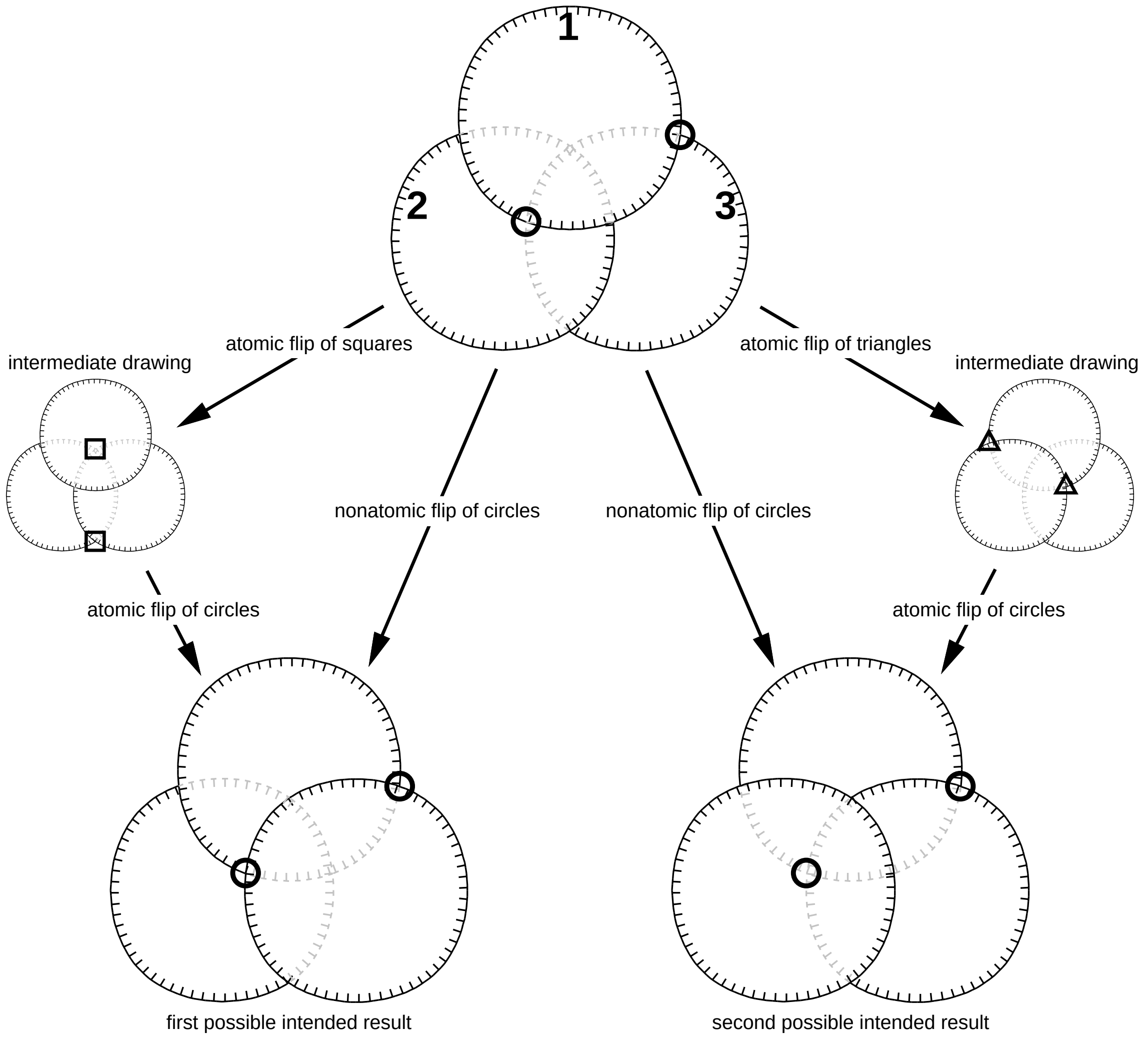

Figure 12: This figure shows the drawing from Fig. 11 at top, with a nonatomic equivalence class marked with circles. Flipping this equivalence class is a nonatomic flip since the result cannot be labeled without flipping other equivalence classes as well. The user's intent when attempting to flip this equivalence class must correspond to one of the two possible results shown at the bottom, but there is no way for *Druid* to tell which result is actually intended. However, each of the two results can be decomposed into a sequence of two atomic flips, the first of which are shown in the small intermediate drawings, and the second of which are shown below. The intermediate atomic flip will convert the desired nonatomic flip into an atomic flip, thus resolving the ambiguity.

rather helps the user choose a sequence of atomic flips which will yield the desired result. This is accomplished by displaying other equivalence classes which may need to be flipped as blinking on and off.

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

1. Arbitrarily choose from among the various legal labelings consistent with the flip.
2. Allow the user to fix the states of some equivalence classes so that they may not be flipped.
3. Discover all possible results. Present them to the user in a table and ask the user to choose among them.

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

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

The third method listed above for handling nonatomic flips, showing all possible solutions and letting the user choose his preferred result, presents the wrong affordances (see Wiley and Williams [7]). We believe that *Druid* should present a legal labeling of a 2½D scene to the user, not a list of options from which to select. However, this method could be used to remedy the problem posed by the potential ambiguity of nonatomic flips. A more serious problem with this method is that there is no obvious bound on the number of legal labelings that might result from a nonatomic flip. In most cases, there will probably be relatively few options. However, there is no guarantee that *Druid* would not have to present a large number of legal labelings from which the user would be required to choose.

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

14 Relabeling Without Search

In the previous section, we showed that the crossing-states of the new labeling following an atomic flip are uniquely determined, and thus no search is necessary to discover the new crossing-states. It might seem necessary to perform a search to find the new boundary segment depths for the labeled knot-diagram, but this is not so. With two basic assumptions, boundary segment depths can be deduced directly from the crossing-states. The first assumption is that the labeled knot-diagram is *normalized*, *i.e.*, that the depth of the shallowest boundary segment in the entire labeled knot-diagram is exactly zero. The second assumption is that the labeled knot-diagram is *vertically compact*, *i.e.*, that the drawing is compacted in the depth dimension as much as possible subject to the constraints of the labeling scheme. With these two assumptions, boundary segment depths are uniquely determined by the crossing-states.

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

When a crossing is flipped, the depths of its two potentially occluded segments will always change and the depths of its two unoccluded segments will never change (see Fig. 9). 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 may not 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.

The relabeling method processes crossings in a FIFO queue. This queue is initially seeded with all crossings in the flipped equivalence class. For each crossing in the queue, *Druid* assigns new boundary segment depths to some of its four boundary segments in order to make the crossing legal. When members of the equivalence class are retrieved from the queue, new boundary segment depths are always assigned to the potentially occluded segments and never to the unoccluded segments, as described above. When crossings are retrieved from the queue that are not a member of the equivalence class, their boundary segment depths must be reassigned so that they are consistent with a labeling scheme.

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

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

If at any time the propagation process reaches a crossing that cannot be legally relabeled without changing the depth of some depth-constrained segment incident at the crossing, then the propagation process must be abandoned because the user's desired flip cannot be performed. Such a situation corresponds to an attempted nonatomic flip since continuing the propagation process would require that crossings which are not members of the user-flipped equivalence class be flipped.

15 Results

To perform a crossing flip, *Druid (*OLD*)* would perform a search to find a new labeling. In contrast, *Druid (*NEW*)* flips the equivalence class and deduces the boundary segment depths that result from the flip. *Druid (*NEW*)'s* method is considerably faster than *Druid (*OLD*)'s* method.

Fig. 13 shows a drawing of low complexity before an equivalence class flip is performed (top) and after two different equivalence classes have been flipped (bottom-left and bottom-right). The equivalence class that has been flipped in each case is marked with circles. The flip at bottom-left involves a fairly large equivalence class, consisting of sixteen crossings, while the flip at bottom-right involves a fairly small equivalence class, consisting of only four crossings. Figs. 14 and 15 show plots of the relabeling running times when each method is applied to each of the two flips illustrated in Fig. 13. Tests were performed on a 1.6 GHz G5 PowerMac.

Based on the plots shown in Figs. 14 and 15, we observe that *Druid (*NEW*)* performs well for both of the flips performed on this drawing. Average turnaround times in both cases were approximately 0.06 seconds, which is effectively instantaneous from the user's perspective.

We observe that the benefit of exploiting equivalence classes is significantly greater when applied to flips of large equivalence classes than when applied to flips of small equivalence classes. The mean running time plot in Fig. 14 shows that *Druid (*OLD*)* takes almost 4000 times as long as *Druid (*NEW*)* to complete the flip shown in the lower-left of Fig. 13. The cost for *Druid (*OLD*)* would have been even greater if the search had not been terminated after 120 seconds. Perhaps more typically, the mean running time plot in Fig. 15 shows that *Druid (*OLD*)* takes only 12 times as long to complete as *Druid (*NEW*)* for the flip shown in the lower-right. The equivalence class employed in the first flip contains only four times as many crossings as the equivalence class employed in the second flip (sixteen crossings vs. four crossings), yet the benefit of exploiting equivalence classes in the first flip is more than 300 times greater than in the second flip. Thus, the benefit of using equivalence classes to relabel scales much faster than linear with respect to the size of the equivalence classes.

In drawings possessing mirror and/or rotational symmetry, distinct equivalence classes will often correspond to topologically identical elements of the drawing, *i.e.*, distinct equivalence classes may represent similar elements that occur in multiple places. For example, in Fig. 13, the two equivalence classes formed by the overlap of the two outer "fingers" and the bottom ellipse are related

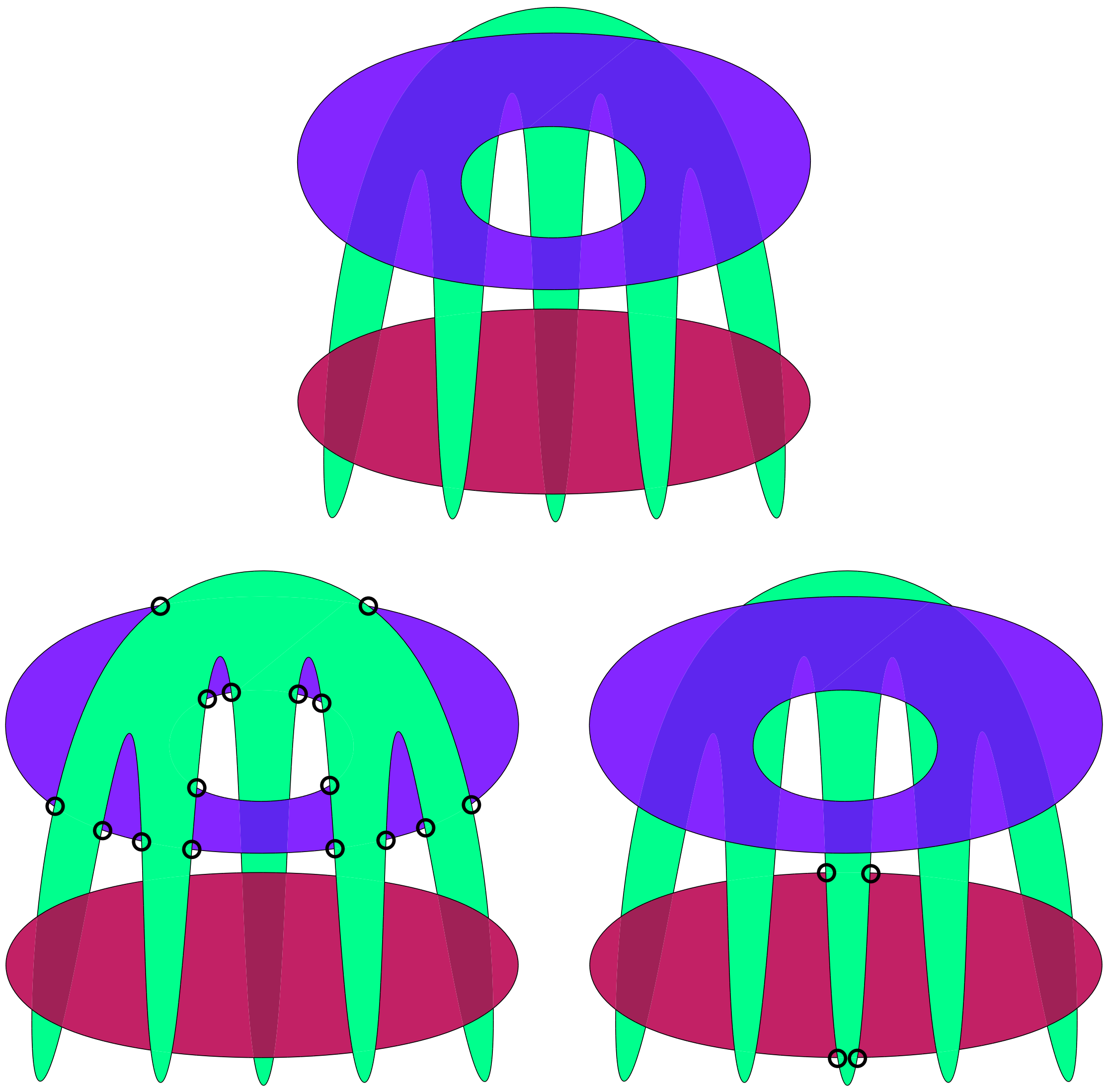

Figure 13: These figures show equivalence class flips for two different equivalence classes of the same drawing. The original drawing is shown at top. The results of performing the two equivalence class flips are shown at bottom with the members of the flipped equivalence classes marked with circles. In the lower-left figure, a fairly large equivalence class has been flipped. The equivalence class for the flipped shared superegion has sixteen crossings. In the lower-right figure, a fairly small equivalence class has been flipped.

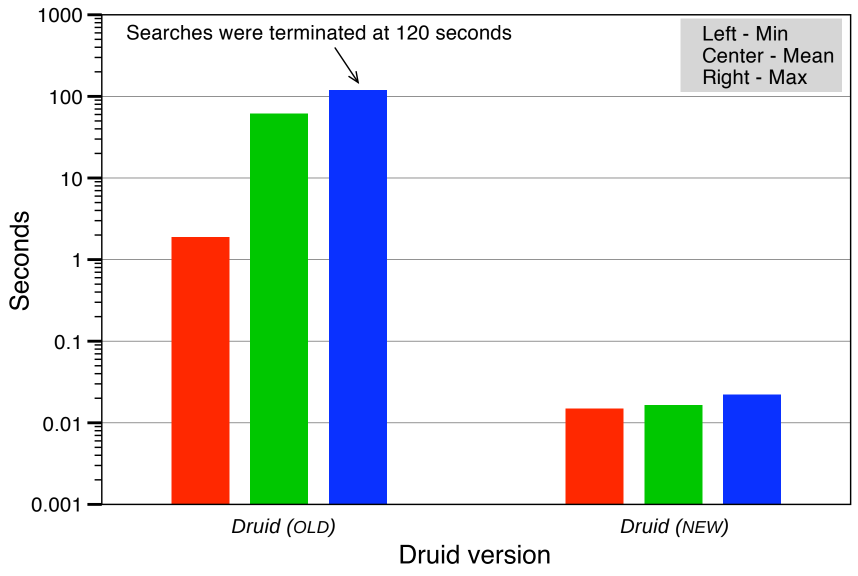

Figure 14: Running times for the two relabeling methods applied to the first flip shown in Fig. 13. Note that time is shown using a logarithmic scale.

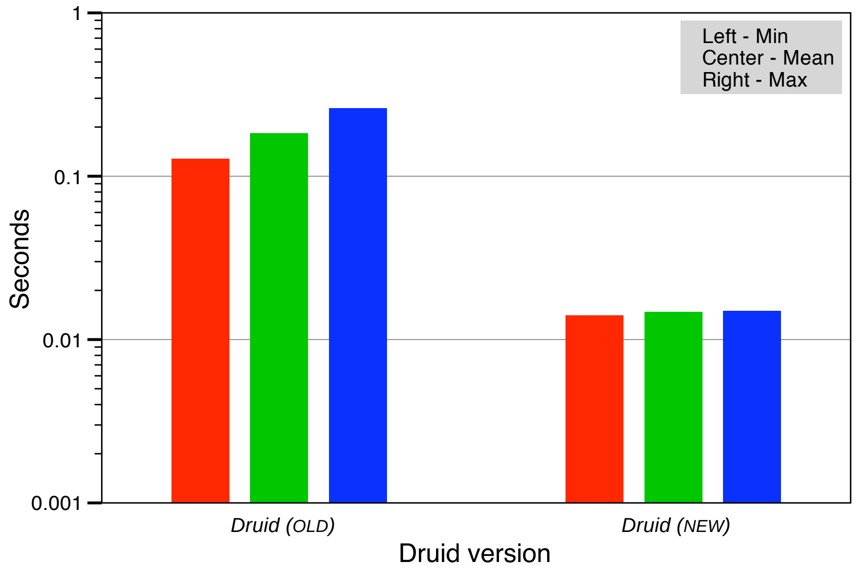

Figure 15: Running times for the two relabeling methods applied to the second flip shown in Fig. 13.

by a mirror symmetry. Fig. 16 shows a fairly complex drawing that contains numerous symmetries which result in numerous topologically identical equivalence classes. One set of topologically identical equivalence classes consists of the eight equivalence classes that have been flipped in the right figure. In order to measure the running times in such cases, we performed each of the eight flips illustrated in the right drawing individually and then combined the data to produce the plot shown in Fig. 17.

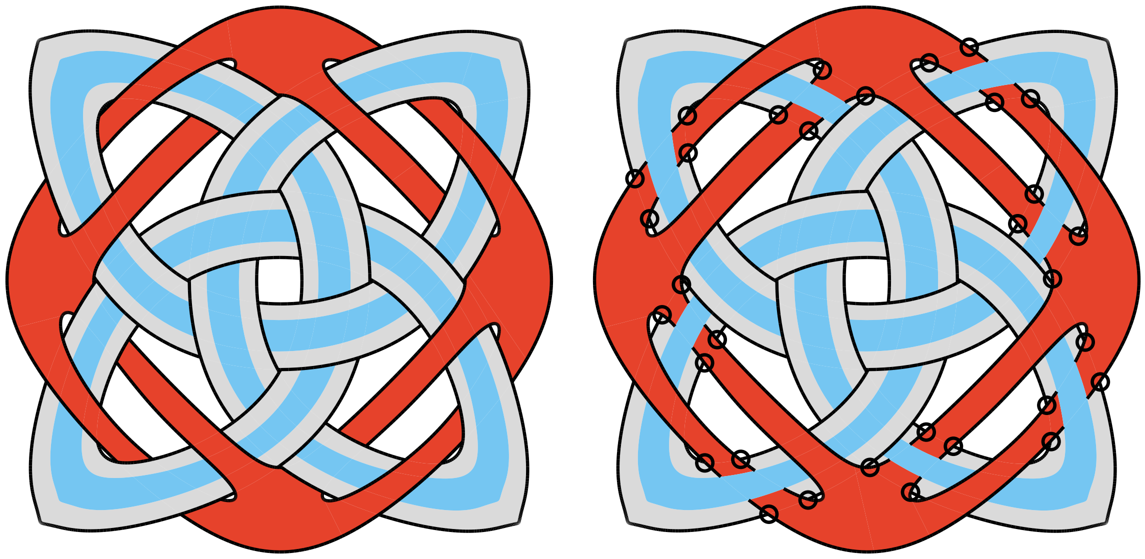

Figure 16: The right figure shows eight topologically identical equivalence class flips performed on the left figure, each marked with circles. The running time tests discussed in the text measure the average time required to perform a single equivalence class flip.

Fig. 17 shows the benefit of exploiting equivalence classes on the drawing shown in Fig. 16. We observe that *Druid (*OLD*)* does not exhibit acceptable turnaround times. We terminated the search after 120 seconds, and in many cases, as shown in the plot, the search failed to find any legal labeling within the alloted time. In contrast, *Druid (*NEW*)* performed quite well with mean turnaround times about 500 times faster than *Druid (*OLD*)* .

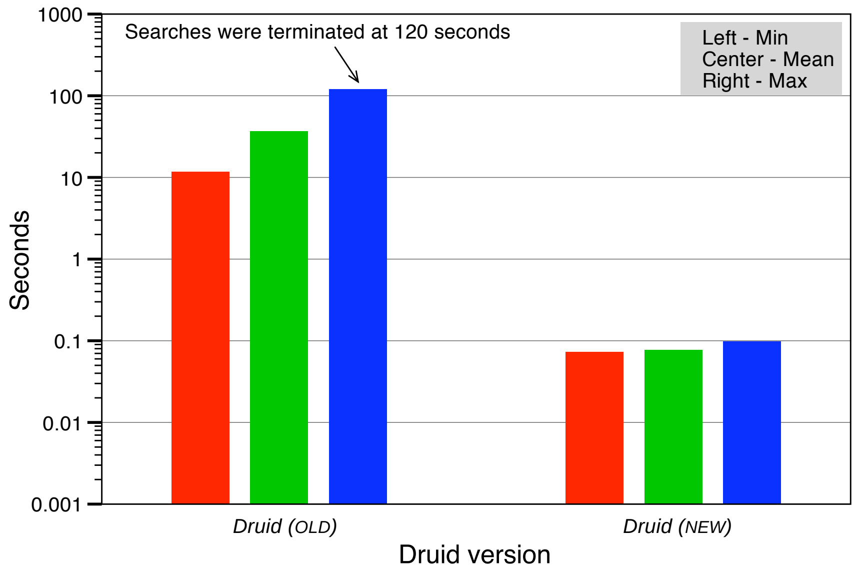

Figure 17: Running times for the two relabeling methods applied to each of the topologically identical flips shown in Fig. 16. The data across all eight possible flips have been combined in this plot.

16 Future Work: Using Crossing-State Equivalence Classes For Labeling

This paper has focused on how the crossing-state equivalence class rule can be used to relabel a labeled knot-diagram. However, there are occasions when an initial labeling must be found for a knot-diagram. Like relabeling, *labeling* can also potentially benefit from the constraint represented by crossing-state equivalence classes. *Druid (*NEW*)* does not take advantage of these constraints when labeling because we do not presently have a robust method for finding the equivalence classes of an unlabeled knot-diagram. However, if *Druid* knew the equivalence classes in advance, and if each equivalence class was initialized to an internally consistent state (all crossings in the equivalence class possess the same state), then there is a clear method for exploiting them during the search for a legal labeling that would vastly decrease the search space. Labeling would be performed using branch-and-bound search with constraint propagation. The proposed method would use equivalence classes as the basis for a new form of constraint propagation. Whenever the search expands a subtree that requires flipping a crossing, the proposed method of search would simultaneously flip all crossings in the associated equivalence class, thus respecting the crossing-state equivalence class rule.

This method for optimizing the search would vastly decrease the search space; the use of crossing-state equivalence classes reduces the search space from 2^C to 2^E for a drawing with C crossings and E equivalence classes. This is a significant reduction in the search space because there are always far fewer equivalence classes of a drawing than there are crossings. Indeed, E can be at most size $C/2$ since equivalence classes always come in even sizes. Consequently, even in the worst case $2^E = 2^{C/2}$.

Consider Fig. 13. There are forty crossings in the figure. Thus, the search space for the search method currently used by *Druid* would have size 2^{40}. The proposed method would exploit the fact that there are only seven equivalence classes in the figure. Thus, it would only search a space of size 2^7.

Since a search is performed on an unlabeled drawing, the method for finding the equivalence classes must be different than the method described earlier for labeled drawings. Since that method makes use of the boundary segment depths, it cannot be applied to an unlabeled figure.

In summary, at the present time, we do not have a proven method for finding equivalence classes on unlabeled figures. Future work on *Druid* will include devising a robust and proven method for accomplishing this task.

17 Conclusions

In earlier work, we developed a system called *Druid* which permits the construction of interwoven 2½D scenes. The new *Druid* system exploits a topological constraint on 2½D scenes which we call the crossing-state equivalence class rule, and consequently can relabel knot-diagrams much more rapidly than the old system. This vastly extends the complexity of drawings that users of *Druid* can construct. In our earlier work [7], we developed an innovative new drawing program. The original contributions of that work were:

- Use of labeled knot-diagrams as the basis for a more general drawing tool capable of representing drawings of interwoven surfaces
- Development of a branch and bound search method for efficiently finding minimum-difference labelings with respect to the labeling preceding a user action

The contributions new to the work described in this paper are as follows:

- Discovery of a topological property of $2^1\!/_2$D scenes, *i.e.*, the crossing-state equivalence class rule
- Development of a method for finding the crossing-state equivalence classes of a labeled knot-diagram
- Development of a method that uses the crossing-state equivalence class rule to relabel a labeled knot-diagram without the need for additional search, thus vastly decreasing the time required to perform a crossing flip user interaction.

References

[1] Adobe Illustrator, ©2005 Adobe.

[2] Coreldraw, ©2005 Corel.

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

[4] iDraw, ©2005 MacPowerUser.

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

[6] Waltz, D. L., Understanding Line Drawings of Scenes with Shadows, McGraw-Hill, New York, pp. 19-92, 1975.

[7] Wiley, K. and L. R. Williams, Representation of Interwoven Surfaces in 2 1/2 D Drawing, *Proc. of CHI*, 2006.

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