Dynamic Planar Map Illustration

Paul Asente
Adobe Systems, Inc.

Mike Schuster
Adobe Systems, Inc.

Teri Pettit
Adobe Systems, Inc.

Abstract

There are many types of illustrations that are easier to create in planar-map-based illustration systems than in the more common stacking-based systems. One weakness shared by all existing planar-map-based systems is that the editability of the drawing is severely hampered once coloring has begun. The paths that define the areas to be filled become divided wherever they intersect, making it difficult or impossible to edit them as a whole.

Live Paint is a new metaphor that allows planar-map-based coloring while maintaining all the original paths unchanged. When a user makes a change, the regions and edges defined by the new paths take on fill and stroke attributes from the previous regions and edges. This results in greater editing flexibility and ease of use. Live Paint uses a set of heuristics to match each region and edge in a changed illustration with a region or edge in the previous version, a task that is more difficult than it at first appears. It then transfers fill and stroke attributes accordingly.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Graphics editors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques;

Keywords: Vector illustration, Graphics editor, Planar map, Gap detection, Dynamic, Recoloring

1 Introduction

Vector illustration systems fall into two classes: stacking (sometimes called 2 1/2-D) and planar map. In a system that uses the stacking metaphor, an illustration consists of a set of paths with fill and stroke attributes. These paths have a stacking order, and a path higher in the stacking order obscures the paths that are below it where they overlap. In a system that uses the planar map metaphor, paths have no fill or stroke attributes, and they are treated as being in a single plane with no stacking order. Fills apply to regions bounded by the paths and strokes apply to edges along paths between intersections.

The vast majority of systems use the stacking metaphor. It is common in everything from dedicated illustration programs like Adobe Illustrator [Adobe 2005] to drawing subsystems embedded in other programs like Microsoft Office [Young and Halvorson 2003]. Stacked illustration is easy to implement, and since the paths do not interact other than to obscure each other, each path can be edited independently from the others. There are, however, many kinds of illustrations that are difficult to create in the stacking metaphor. These include illustrations where the elements interact in a non-stacked way, such as weaves, knotwork, or linked rings, illustrations that do not have any underlying structure, like hand-drawn cartoons, and illustrations where the regions to be filled are bounded by several unrelated paths. Examples of such illustrations can be found throughout this paper.

Planar map illustration is much rarer. The first system to use it was MapSketch [Baudelaire and Gangnet 1989], followed by TicTacToon [Fekete et al. 1995], which was specifically dedicated to animation. Adobe Illustrator incorporated limited planar facility with its Pathfinder functionality, and Adobe Flash [Macromedia 2005] built its entire editing system around it. Planar map illustration has opposite strengths and weaknesses from stacking illustration. It is easy to draw elements that interact in non-stacked way, and easy to draw illustrations without structure. However, the systems are more difficult to implement because the algorithms to construct the regions and edges from a set of paths are complex. More importantly, while the illustrations are easy to color, they are difficult to edit. Paths become broken up at intersections and are no longer editable as a whole.

Figure 1 shows various results one gets in different planar systems in response to selecting the center bar of the illustration and pulling it down. Different systems give different results, sometimes depending upon how the bar was selected. All previous systems either break up the paths at the intersections, or, like TicTacToon, do not permit paths to be edited at all. Our system, Live Paint, is able to maintain the paths without breaking them up, leading to the final picture.

A user colors a Live Paint illustration using the planar map metaphor, applying fills to areas defined by paths and strokes to edges along paths. However, Live Paint does not break the paths up at intersections; the paths continue to exist in their original forms,

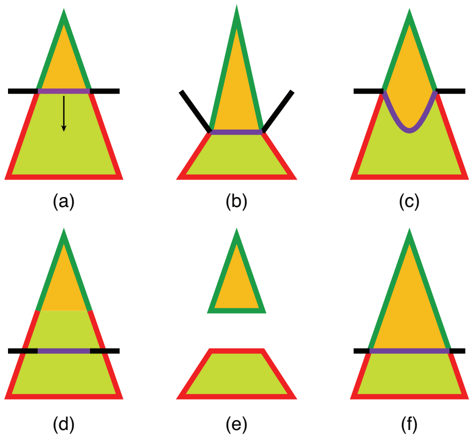


Figure 1: Various results of editing. (a) Original illustration (b) MapSketch (c) Adobe Flash (d) Adobe Flash (e) Adobe Illustrator Pathfinder (f) Live Paint

ACM Reference Format
Asente, P., Schuster, M., Pettit, T. 2007. Dynamic Planar Map Illustration. *ACM Trans. Graph. 26,* 3, Article 30 (July 2007), 10 pages. DOI = 10.1145/1239451.1239481 http://doi.acm.org/10.1145/1239451.1239481.

and the user manipulates these paths to edit the illustration. Live Paint maintains the fills and strokes on the regions and edges throughout the editing process. If the user does want the results generated by previous systems, they can still be achieved by adding control points at intersections or cutting paths.

Figure 2 shows two cases of coloring and editing a simple Live Paint illustration. The first column shows the paths of the illustration, and the second shows the initial colored results. These results would be difficult to achieve in a system that used the stacking metaphor, but they are easy to do in any planar map system. Live Paint takes the process a step further, allowing the user to modify the underlying paths and have the results appear as expected.

A subset of the need addressed by Live Paint was solved by Wiley and Williams [2006]. Their Druid system allows a user to manipulate interwoven 2 1/2-dimensional surfaces, like knots and interlocked rings, and to edit these surfaces to maintain apparent local stacking. It does not, however, address the problem of editing planar drawings that do not represent interwoven surfaces, like the first example in Figure 2.

It is difficult to convey the power of Live Paint through static figures. Any illustration drawn with Live Paint could equally well have been drawn with other planar map systems, assuming that the artist knew beforehand exactly what the final illustration should be. The advantage of Live Paint is that it allows the artist to easily modify a composition after applying color. Figure 3 shows a more complex example involving a portion of a cubist-style illustration of a sunny sky. The top picture shows the paths, and the middle shows the result after coloring. The bottom shows how Live Paint allows the artist to adjust the composition of the illustration after coloring without having to recolor or redraw, something not previously possible.

2 Correctness

Baudelaire and Gangnet refer to the desirability of this type of editing in the close of their 1989 paper, but suggest no way to achieve it. There are two possible reasons this hasn't been done before.

One is performance. Creating a planar map from a set of paths is $O(p \log p + n)$ in p the number of paths and n the number of intersections. While n can in theory be very large, the kinds of illustrations users actually create have only a few intersections per path, making this term effectively linear in p.

This is a promising basis for an editor, but the constant factors are large. Until recently, any system that created a planar map after each editing operation would have been far too slow to be useful in practice. However today's processors let Live Paint be nearly instantaneous for simple illustrations, and to respond in about a second for illustrations containing a thousand paths.

A more important reason is that it is challenging to come up with a set of rules that capture what users would consider to be the right answer. It is quite easy to come up with ideas that seem as if they should work, but for most of them it is just as easy to come up with examples where they do not.

For example, one might think that moving a path should move the regions that border it in the same general way. Figure 4 shows that sometimes this works, and sometimes it doesn't.

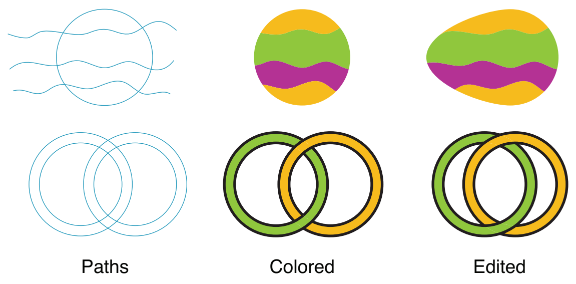


Figure 2: Coloring and editing Live Paint illustrations.

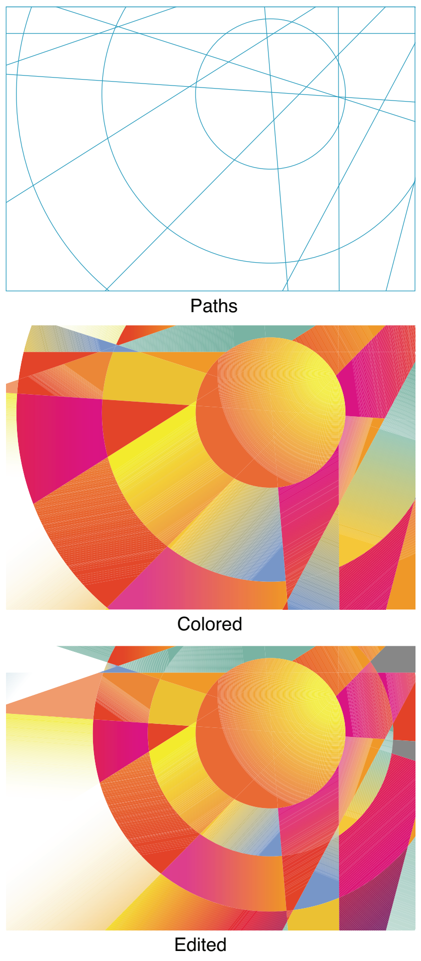


Figure 3: Coloring and editing a more complex illustration.

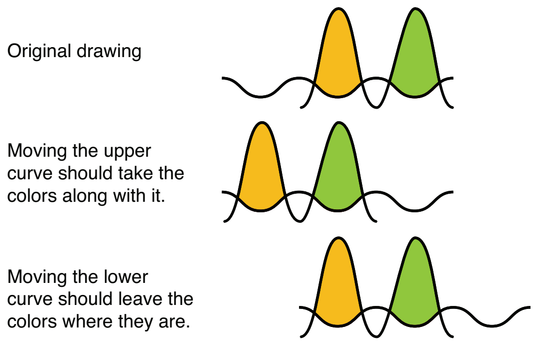

Original drawing
Moving the upper curve should take the colors along with it.
Moving the lower curve should leave the colors where they are.

Figure 4: Should regions move with paths?

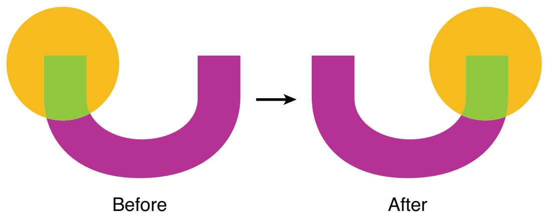


Figure 5: A bad case for dynamic flood-fill.

A dynamic flood fill algorithm constantly updates the filled regions as the user changes the paths. It would work in many cases, but it has problems. Not all editing operations are incremental; operations like path reflection or deleting a vertex from a polygon can make a path change instantaneously. Dynamic flood fill can also make things hard on a user—there may be no way to move a path from one spot to another without affecting intervening paths, and an edit like that shown in Figure 5 would require the user to carefully move a path to avoid losing a color. Finally, it provides no guidance in deciding how newly-created regions should be filled. These problems are sufficiently difficult that we chose not to base a system on dynamic flood fill.

However, dynamic flood fill does provide a conceptual framework for deciding what the correct result should be. If dynamic flood fill maintains the illustration's topology during an operation, neither creating nor destroying regions, then we will define the flood fill result as correct. Live Paint uses a set of heuristics that efficiently produce the same results as dynamic flood fill in nearly all topology-preserving editing operations, and applies them in all cases. User testing has shown that it produces the expected result in nearly all editing operations.

3 System Architecture

An illustration consists of a set of geometric paths that are given unique identifiers that persist across changes to the path. Each path has a non-decreasing parameterization from its beginning to end, and is characterized as open or closed depending on whether its ending point coincides with its starting point. In our system a path is an end-to-end connected set of Bézier curve segments, and the parameterization is a real number with integer part the ordinal number of the segment and fraction the Bézier parameterization along the segment. The set of paths in an illustration is collectively called the illustration's *geometry*. The user can select and modify the paths in the geometry, remove paths from the geometry, and add new paths to the geometry.

The geometry of an illustration divides the plane into a set of non-intersecting *regions*. Each region has a set of visual attributes such as solid colors, repeating patterns, gradient colors, or transparency. The attributes for a region are called the region's *fill*. The user interface of the system provides several ways for the user to modify the fill of a region, including a paint bucket tool that applies a fill to a region that the user clicks in.

The intersections of the paths divide each path into a set of non-overlapping *edges*. Each edge has a set of visual attributes such as color, stroke width, and dash pattern. The attributes for an edge are called the edge's *stroke*. The user interface of the system provides several ways for the user to modify the stroke of an edge, including a paint brush tool that applies a stroke to an edge that the user clicks on. Unlike regions, which normally do not have a close association with a particular geometry path, each edge can be associated with one or more paths—usually one, but with more if there are places where multiple paths coincide.

Vertices are path endpoints and places where paths intersect. A vertex exists at every intersection, even where exactly two paths intersect at their endpoints. Such a vertex has two edges, one on either side.

An illustration's set of regions with their fills and edges with their strokes make up the *result art* of the illustration. The result art is what is the user sees on the display.

A *planar map* is a data structure with a set of paths and the regions and edges defined by those paths. The planar map implementation in Live Paint is based on the work of Bentley and Ottmann [1979] as modified by Hobby [1999] to snap intersections to a fixed grid. Curved geometry paths are closely approximated by polylines before being added to the map.

The planar map supports normal graph operations like enumerating the edges that surround a region. Additional operations, like enumerating the edges along a path, enumerating the paths that include an edge, and enumerating the regions inside a closed path, relate the graph to the defining geometry. The map also maintains the stroke for each edge and the fill for each region.

At any one time, there exists the geometry, consisting of the paths that the user can select and modify, the result art, consisting of the regions and edges that the user can see and paint, and a planar map behind the scenes that ties the two together.

When the user makes a change to the geometry, the following steps occur, as shown in Figure 6:

1. A new geometry is constructed, called the *current geometry*.
2. These paths are used to create a new planar map, called the *current planar map*. The regions and edges in the current planar map do not yet have fills and strokes associated with them.
3. The previous geometry, current geometry, previous planar map, current planar map, and a description of what kind of changes were made are passed to a paint assignment module.
4. The paint assignment module assigns fills and strokes to the regions and edges in the current planar map. This is called *painting* the map.
5. New result art is created from the current, painted, planar map.
6. The current geometry replaces the old geometry, the current result art replaces the old result art, and the current planar map replaces the old planar map.

The system is now ready for the next change.

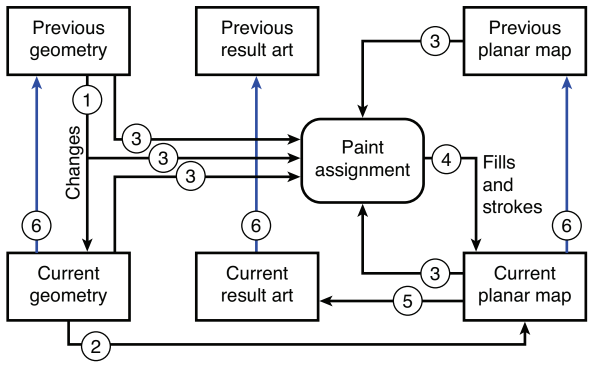


Figure 6: System architecture

It should be noted that this cycle has no memory built in; the result depends only upon the state when the user makes the change and the change itself. For example, if the intersection of two closed paths has a fill different from the outsides, the intersection will not regain that color if the user pulls the paths apart and then re-overlaps them as two separate operations. This leads to improved predictability. The user does not need to remember what the illustration looked like before or how it got to its current state.

The cycle when the user changes the fill of a region or the stroke of an edge is simpler. In this case Live Paint associates the new fill or stroke with the region or edge in the planar map, and updates the result art accordingly.

In order to facilitate operations like finding regions and edges that have been split by a newly added path, or finding regions and edges that have been merged by a deleted path, newly added paths are tagged as such when added to the current map, and deleted paths remain in the current map, but are marked as deleted. The region and edge operations for the map have the option to either combine or treat separately regions and edges that are separated only by deleted paths. Added paths lose their special tag and deleted paths go away completely in the next iteration.

The paint assignment module has two parts, a fill assignment module and a stroke assignment module. For each region or edge in the current map, Live Paint uses a set of heuristics to find the region or edge in the previous map that most closely matches the current one, and transfers the fill or stroke to the current from the old.

It might seem that general planar graph matching algorithms [Alt et al, 2003, Neuhaus and Bunke, 2005] might be useful in matching up the current and previous maps, but they are excessive. Live Paint maintains path identifiers throughout the editing process, which make it quick and straightforward to match up portions of an illustration that do match. These portions have regions bordered by the same paths and edges defined by the same intersections. The matching process at the same time identifies the portions of the graph that don't match exactly, and characterizes them in a way that gives a useful description of how they are similar and different.

It must be said that the problem of fill and stroke assignment is inherently ill-posed. Figure 7 shows a simple illustration with a circle that is bisected by a line segment and has different fills on the two sides. (The figures in this paper all use thin blue lines to show the paths in the geometry. These paths may or may not be visible in the result art, depending upon whether the user has assigned strokes to them.) If the line segment is moved to the right, there is no way to know which fill to give the undivided circle. In some cases different users could perform the exact same edit on an illustration and expect different results. The goal of Live Paint cannot be to always produce the expected result; instead it is to produce a reasonable result. The user should not be left wondering why the result came out as it did; even if it is not the desired result, the user should be able to understand why it was a reasonable choice.

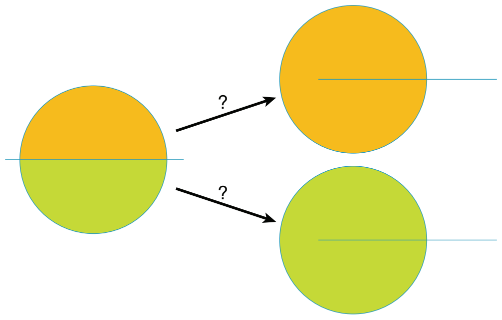


Figure 7: An ill-posed case.

Although this paper usually refers to a single Live Paint illustration, such an illustration can be embedded within a larger document that combines multiple Live Paint illustrations as well as traditional stacked shapes. Each Live Paint illustration is treated as a unit and can be stacked with other elements of the document, possibly with transparency. For example, a logo done with Live Paint could be placed on top of the shirt of a figure that was also drawn with Live Paint. The logo and figure would not interact; each could be edited and colored without affecting the other.

4 Fill Assignment

Live Paint uses four different ways to assign fills to the new map: simple assignment, stable assignment, closed path assignment, and general assignment.

4.1 Simple Fill Assignment

Simple fill assignment applies when a user's change consists only of reparameterizing, adding, extending, deleting, and truncating paths. This is an optimization that allows Live Paint to avoid the more complex assignment methods.

If the change consists of changing the parameterization of some paths in the geometry, no change is made to the fills. Every region in the current map takes the fill of the corresponding previous region.The most common case of parameterization change is subdividing one Bézier path segment into two.

If the change consists of adding new paths to the geometry, or extending existing paths at either end without changing their existing parts, then some regions in the previous map may have been divided into multiple regions in the current map. Each divided region takes the fill of the previous undivided region. Regions not divided by the addition take the fill of the corresponding previous region.

If the change consists of removing open paths from the geometry, or truncating existing paths at either end without changing the remaining parts, then some regions in the previous map may have been merged into a single region in the current map. The merged regions take the fill of whichever previous region was largest. Regions not merged by the operation take the fill of the corresponding previous region. Note that this is not done for removing closed paths; the results of doing so are in some cases not intuitive as shown in Figure 8. Closed path assignment handles the deletion of closed paths.

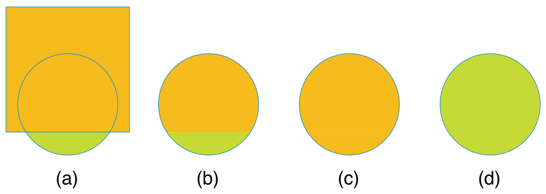


Figure 8: Deleting a closed path. (a) Original illustration. (b) After deleting square but before merging. (c) Result using largest area rule. (d) Likely desired result.

4.2 Region Context

If a change is too complex to be handled by simple assignment, Live Paint gathers information about the regions in the previous and current maps. This information is called the *context* of the region, and contains

- A list of every path that borders the region, and for each path:
  - The parameterization values at the start and end of the portion of the path bordering the region.
  - Whether the region is to the left of the path, the right of the path, or on both sides of the path. “Left” and “right” here are relative to the path, considered in the direction of increasing parameterization. They do not refer to the orientation of the path as viewed by the user. A region is on both sides only when an open path extends into a region but does not divide the region in two.
  - If the path is closed, whether the region is inside or outside the path.

  A region may be bordered by multiple parts of the same path; in that case Live Paint collects an instance of this information for each bordering portion.
- A list of every closed path that encloses the region, even if that path does not directly border the region

The list of the bordering paths and the geometric relationship of the region to these paths is called the *simple context* for the region. The simple context plus the parameterization information is called the *extended context* for the region. The lists of previous and current regions are ordered lexicographically by extended context, allowing efficient matching between previous and current regions with the same context. The details of the lexicographic comparison function are unimportant; all that matters is that it defines an arbitrary total ordering that is shared by both the current and previous set of regions.

4.3 Stable assignment

Stable assignment assigns fills to areas of the illustration where nothing has changed. For each current region, if there is a previous region with identical extended context and identical path inclusion, the current region takes the fill of that previous region. This guarantees no fill changes in unchanging parts of the illustration, and usually removes most regions from the more complex assignment methods that follow.

4.4 Closed Path Assignment

If there are regions not yet assigned, and at least one of these regions is within a closed path, Live Paint performs closed path assignment. The goal of closed path assignment is to preserve familiar editing results on parts of the illustration that look as if they are made from traditional filled, stacked closed paths. This involves maintaining two invariants across edits involving closed paths. Overlapping paths that appear to have a particular stacking order should maintain that apparent stacking order, and if a path appears to have a fill, all regions in that path should have that fill unless they are also within a path that appears higher in the stacking order.

Paths that do not overlap have no apparent stacking order relative to each other. For predictability, Live Paint adds the invariant that a path with an apparent fill that is moved or changed to overlap a path that it did not overlap before should now appear to be above that path.

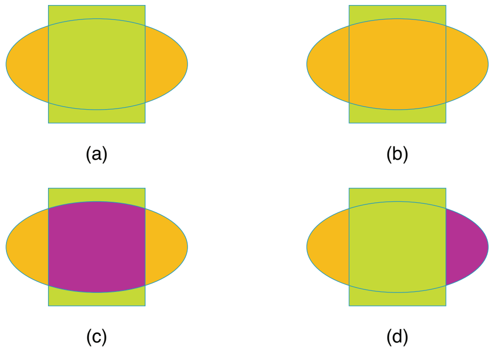


Figure 9: Apparent fills and stacking orders. (a) The ellipse appears orange, the rectangle green, and the rectangle appears to be above the ellipse. (b) Same apparent fills, but the ellipse appears to be above the rectangle. (c) Same apparent fills, but there is no apparent stacking order. (d) The ellipse has no apparent fill but the rectangle does, and the rectangle appears to be above the ellipse.

It must be emphasized that the fill and stacking order are only apparent, not actual. They can change or be eliminated when the user assigns new fills to regions, as shown in Figure 9. This figure also shows that not every pair of overlapping paths appear to have a stacking order, and that not every closed path has an apparent fill.

Closed path assignment begins by analyzing the closed paths in the previous map to find apparent fill colors and stacking relationships. This alternates between finding paths that have an apparent color and finding paths that are apparently below those just determined to have a color, and is fully described in Appendix A.

The apparent stacking relation is transitive only if all paths involved have a region in common. It is possible, and useful, to have non-transitively overlapping paths, as shown in Figure 10. This non-transitive overlapping will be maintained as long as the paths are not modified to overlap each other in new ways.

After determining apparent fills and stacking order, Live Paint examines each current region that is inside at least one closed path and that does not yet have a fill. If all closed paths that contain the region have apparent fills, Live Paint checks to see if one of these closed paths appears to be the top path. If there is exactly one enclosing path with no other enclosing path apparently above it, this is the apparent top path. If there is more than one such path, the apparent top path exists only if the potential top paths were previously disjoint, and at least one of them has been modified by the current change. In that case, Live Paint chooses (arbitrarily) one of the modified paths to be the apparent top path.

If there is an apparent top path, its apparent fill becomes the fill of the region. The test for multiple paths being previously disjoint and one path being modified makes modified paths appear to be above paths that they did not previously overlap.

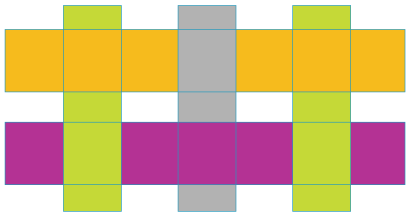


Figure 10: Non-transitive stacking

4.5 Context Assignment

If there are regions not yet assigned, Live Paint makes a final pass through the regions doing context assignment. This takes each current region and finds the previous region whose context most closely matches, and transfers the fill from that previous region to the current region. As a reminder, a region's simple context describes how it relates geometrically to the paths that border it, and its extended context is the simple context plus the parameterization values of the beginning and end of each bordering path segment.

For every unassigned current region, Live Paint looks at all previous regions that have the same simple context. There are three possibilities:

If all previous regions with the same simple context have the same fill, including when there is only one such previous region, then the current region takes that fill.

If previous regions with the same simple context have different fills, the current region takes the fill of the one whose extended context most closely matches the current region's extended context, using the metric described below.

If there is no previous region with the same simple context, Live Paint considers each previous region that borders a path bordering the current region. The current region takes the fill of whichever of these regions has an extended context most closely matching the current region's extended context, using the metric described below. If there is no previous region that borders a path bordering the current region, the current region is left unfilled.

The matching algorithm gives a match quality to each potential previous region, and chooses the one with the highest quality. Match quality takes four things into account: hits, misses, boundary coverage, and parameterization distance.

- A hit can potentially occur for each path that borders both the current and previous region.
  - If the current and previous versions of the path are both closed, a hit occurs if
    - both the current and previous regions are inside the path, or
    - both the current and previous regions are outside the path.
  - If either the current or previous version of the path is open, a hit occurs if
    - the current region is to the right of the path and the previous region is either to the right of or on both sides of the path, or
    - the current region is to the left of the path and the previous region is either to the left of or on both sides of the path, or
    - the current region is on both sides of the path and the previous region is either to the left or right of the path.
- A miss can potentially occur for each path that borders both the current and previous region. For each such path, a miss occurs if no hit occurs, and it is not the case that both the current and previous regions are on both sides of the path.
- Boundary coverage is what fraction of the boundary of the current region is covered by the portions of paths that bounded the previous region, using path parameterization.
- Parameterization distance looks at each path that borders both the current and previous region and computes how far away the bordering portions along the path are from each other. If a path borders the current region from t_{1c} to t_{2c} and the previous region from t_{1p} to t_{2p}, the parameterization distance for that path is sqrt$((t_{1c} - t_{1p})^2 + (t_{2c} - t_{2p})^2)$. The parameterization distance for two regions is the sum of the parameterization distances for each bordering path.

Note that no hit or miss occurs for a path that borders only one of the previous and current regions, or for a path that has the previous and current regions on both sides. Such a path is ignored in computing match quality.

In comparing two match qualities, the better is the one with

- Fewest misses
- If the number of misses is the same, the most hits
- If the number of hits is the same, the largest coverage
- If the coverage is the same, the smallest parameterization distance.

Figure 11 shows a simple case of context assignment. The original illustration has two regions **a** and **b**, with the context descriptions shown. After moving the diagonal path, there are two regions **x** and **y**, with the descriptions shown. There is only one previous region that matches **x**, and one that matches **y**, giving the expected results.

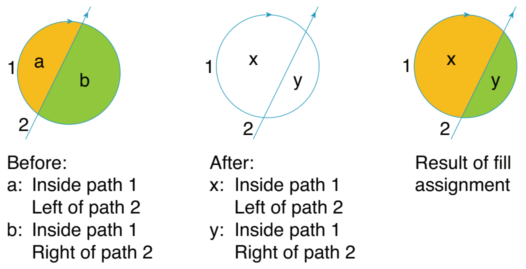


Figure 11: A simple case for context assignment.

Figure 12 shows a case where boundary coverage resolves an ambiguity. After moving the straight line, there are two regions, **x** and **z**, with the same simple context. Previous regions **a** and **c** match in simple context but have different fills, so Live Paint invokes the matching rules to find the best matches for regions **x** and **z**. When looking for a match for **x**, regions **a** and **b** have equal numbers of misses (0) and equal numbers of hits (2). Live Paint compares the boundary coverage of **x** relative to region **a** with its boundary coverage relative to region **c**. The coverage relative to region **a** (complete) is larger than that relative to **b** (none), so **x** takes region **a**'s fill. Similarly, **z** takes **c**'s fill. Intuitively, ambiguous cases are handled by taking regions that are as close as possible to the defining parts of the bounding paths.

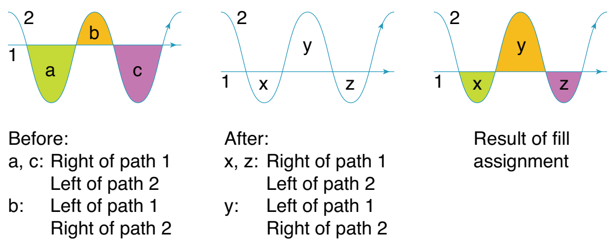


Figure 12: Fill assignment using boundary coverage

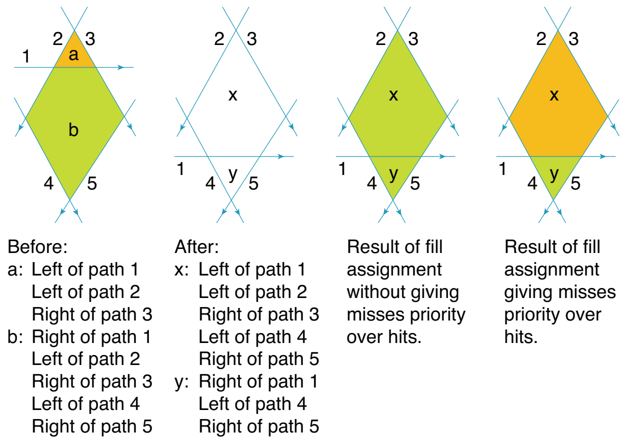


Figure 13: Effect of misses in region matching

Figure 13 demonstrates why misses are given priority over hits in region matching. After moving path 1, neither region **x** nor **y** has a direct context match, so Live Paint considers all regions that border the defining paths. Finding a match for region **y** is simple: region **a** has no hits and 1 miss, while region **b** has 3 hits and no misses. Region **b** is a better match. Finding a match for region **x** is more complicated. Region **a** has 3 hits and no misses, but region **b** has 4 hits and 1 miss. If misses were not given priority over hits, region **x** would take **b**'s fill. Giving misses priority results in the improved fill assignment of the final picture. Intuitively, giving misses priority prevents fills from jumping across paths.

Parameterization distance is used only when the boundary coverages match, most commonly when the boundary coverage is zero. This finds the closest matching region when the regions do not border any of the same portions of the paths.

As an optimization, Live Paint saves the context for each current region with the planar map. This can then be used directly as the context for the previous region on the next fill assignment operation.

5 Stroke Assignment

Assigning strokes to the result art is simpler than assigning fills. The appropriate result for most editing operations is to give each edge of the current path the stroke of some edge of the previous version of the path; it is rarely appropriate to introduce a stroke that was not present on the path before. This means that each path can be treated independently, only looking at the edges of its previous version to find strokes for the current version's edges.

The one exception to this occurs when a change makes paths that did not coincide in the previous geometry partially or completely coincide in the new geometry. In this case, there is at least one current edge that is part of both paths, and its stroke can only come from one of the previous paths' edges. However, the invariant in the previous paragraph can still hold true for one of the paths; in Live Paint it holds true for the modified path. Each path can therefore still be treated independently as long as unmodified paths have their edges assigned before modified paths. A newly coincident edge that is part of an unmodified path first takes a stroke from that path, and then assigning strokes to the modified path overrides that stroke with a new one. If a change modifies multiple paths simultaneously to make them coincident, the order is arbitrary and whichever path gets processed last keeps the invariant.

Stroke assignment is similar to fill assignment, but it uses vertices to match edges rather than edges to match regions. We will only sketch out the general method:

1. Handle simple cases, when all edges of the previous path had the same stroke or when neither the path nor any paths that intersect it have changed.
2. Characterize each vertex along the current and previous path with a description of which other paths intersect it, how they intersect it, and the parameterization values.
3. Associate each vertex in the current path with the vertex in the previous path that matches most closely.
4. Use the vertex matches at the ends of each current edge to choose a previous edge, and give the current edge the previous edge's stroke.

6 System Considerations

While fill and stroke assignment are the core of Live Paint, there are other pieces needed to make a complete editing system.

6.1 Gap Detection and Closure

Hand-drawn pictures often contain small gaps in outlines of regions. These gaps cause areas that the user intended to be separate regions instead become one region, and areas that should be enclosed become part of the outside world. Live Paint allows the user to fill illustrations with gaps, as shown in Figure 14, using a gap-detection algorithm similar to that described by Gangnet et al. [1994]. One extension is that Live Paint dynamically finds gaps each time the illustration changes, and gaps can move around as parts of the illustration change.

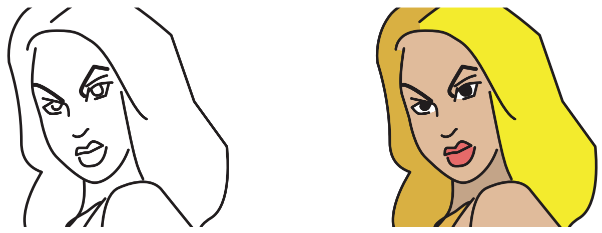


Figure 14: Filling a picture that has gaps.

When Live Paint finds gaps, it adds gap-closing segments to the geometry. Live Paint completely ignores these segments when doing region matching for fill assignment. They are not considered to be part of a region's context, they do not participate in coverage computation, and they are not part of the parameterization distance computation.

Gap segments never receive strokes; if the user wants to stroke a gap, he or she can draw a new path. However, gap segments do divide paths that they intersect into multiple edges. When doing stroke assignment on such a path, Live Paint treats all gap segments as equivalent in vertex matching. Any gap segment matches any other gap segment, and no gap segment matches any non-gap path.

6.2 Conversion

Live Paint allows the user to convert an illustration that uses stacking into a Live Paint illustration. First Live Paint creates an illustration with the same paths, with all regions having no fill and all edges having no stroke. It then runs the classic painter's algorithm by considering each path in the original illustration from lowest in the stacking order to highest, filling regions inside the path, stroking edges along the path, and removing the strokes from previously-stroked edges inside the path.

6.3 Persistent Representation

When saving an illustration, Live Paint annotates each path with a list of *fragments*, which are stretches along the path where the left-side fill, the right-side fill, and the stroke do not change. Each fragment describes the two fills and the stroke. To reconstruct an illustration, Live Paint creates a new planar map from the paths and uses the fragment descriptions to paint the regions and edges.

7 Validation and Problem Cases

Live Paint's results were validated by observing real users using it to create real illustrations. If a user got unanticipated results, we saved the illustration before the edit along with the edit and tried to enhance the assignment heuristics to produce the desired outcome. One example of this was the introduction of closed path assignment. Our test users quickly grasped the concept of planar editing, but they continued to expect closed paths to behave like they do in a traditional stacking editor. Figure 8 shows one case where the results differ. Adding closed path assignment allowed users to work as before when editing closed paths.

We also kept a large body of test cases that produced expected results. Whenever the assignment heuristics changed, we re-ran all these cases to detect the introduction of unexpected behavior.

We also ran tests with several experienced Live Paint users creating real-world illustrations. During these tests, we kept track of how often they felt that Live Paint gave unexpected results, and our correctness rate was approximately 98%. However, since these were experienced users, they knew how to avoid cases that give Live Paint problems. Other tests with inexperienced users showed higher error rates, but the incidence of corrective behavior (undoing or making fixes) was no higher than with traditional methods.

There are four categories that cause most Live Paint problems.

Stacked art made from non-closed paths: Live Paint's handling of apparently stacked illustrations so closely mimics traditional stacked semantics that users expect it to work this way when they create filled shapes that are defined by separate open paths, such as a rectangle defined by four intersecting lines. However, Live Paint can only apply stacking semantics to closed paths, giving rise to problems as shown in Figure 15(a). This deficiency accounts for the largest number of failures, and we know of no way to address it in the Live Paint heuristics. User interface techniques could, however, give additional information to the system to improve this.

Interwoven shapes: Interwoven shapes are closed paths that have locally stacked areas but no global stacking. This includes cases like linked rings, where two shapes are both above and below each other in different places, and cases like knotwork where a single path appears to overlap itself. Live Paint produces the expected results when a user makes small changes to these paths, but large

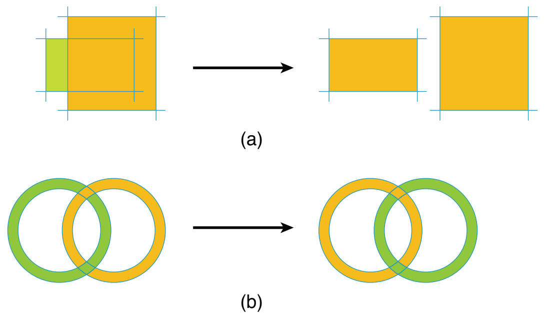


Figure 15: Problem cases: (a) Apparently stacked art from non-closed paths. (b) Interwoven shapes.

changes, like moving one ring to the other side of a ring it is linked with, often fail, as shown in Figure 15(b). Users usually learn that they can get more predicable results in these cases by making the change in several smaller steps. This failure could be addressed by augmenting the apparent stacking algorithms with the techniques described by Wiley and Williams [2006] to give Live Paint an explicit understanding of these overlaps so that it could maintain them.

Scribbly paths: If a path intersects itself many times, changes to it tend to produce unexpected results. This is because there are many regions and edges with similar or identical simple contexts. Editing such a path often creates a new region that most closely matches a distant region, causing fills to jump around surprisingly. Fortunately, these kinds of paths are rarely used. This could be addressed by making boundary coverage more important than hits or misses, but we found it difficult to do this without creating undesirable results in simpler cases.

Bars are not lines: While a user might expect a thin rectangle or other thin closed path to act similarly to an open path, Live Paint treats them very differently. An open path has a left side and a right side, but a long rectangle has an inside and an outside. Both "sides" of the rectangle are outside, and Live Paint treats them similarly. If the horizontal line in Figure 13 were replaced by a thin horizontal rectangle, there would be no misses to constrain context matching, and the result would be as shown in (c). This failure could be addressed by using the medial axis of thin shapes to define a left and right side of the shape. However, while this problem has caused much anxiety for the authors, it has never been observed happening to a real user.

8 Future Work

The major performance bottleneck with Live Paint is that each change requires reconstruction of the planar map and assigning fills and strokes to all regions and edges. Unlike most illustration systems, which typically respond in time proportional to the number of paths changed at one time, Live Paint responds in time proportional or worse to the total number of paths in the illustration. Although response time is quite acceptable for small- and medium-complexity illustrations, large illustrations with many thousands of paths can take seconds to respond to even small changes. Such cases normally only arise through wholesale conversion of complex illustrations to Live Paint. Users don't normally want to have thousands of potentially interacting paths; it is excessively confusing. When constructing a complex illustration, they usually divide it into simple, overlapping, non-interacting layers.

An incremental planar map implementation such as that described by Ganget et al. [1989] could improve performance by eliminating the need to reconstruct the map with each edit and by limiting fill and stroke assignment to the parts of the illustration that have actually changed. However, past experience by the authors found scalability problems when applying Gangnet's approach directly to maps containing even several hundred paths; the rounded approach described by Hobby [1999] performed better for maps of this size. Combining an incremental approach with rounding is challenging since any modification to the geometry can make rounding changes cascade throughout the map far beyond the area directly affected by the modification.

Since assigning fills involves picking the most highly-ranked region from several alternatives, a useful extension would be to create a user interface that presents ranked choices to the user in case an assignment is incorrect.

A problem with any planar system is that it can create tiny edges and sliver regions where paths do not intersect exactly as the user intended. The intersection rounding done by Live Paint makes the problem even more likely to occur. The system could be improved by merging these with larger adjacent edges and regions.

There is always room for improvement in any heuristic-based system. Since fill and stroke assignment are independent, it is possible, but rare, for a change to cause a stroke along the edge of a region to become detached from the region and end up elsewhere on the bounding path. Other kinds of heuristics are possible; for example, a change that preserves the number of regions should usually also preserve the number of regions with a particular fill.

Live Paint does not make use of continuous descriptions of user changes. There are cases where doing so might give better results, for example, using the trajectory when a user moves a dividing line out of a circle to decide which color to leave behind. However, using the trajectory in all cases gives rise to the problems described in Section 2 discussing dynamic flood fill. It is not clear whether using the information in some, but not all, cases would give a more predictable system.

In general, a great deal of care must be taken when introducing a new heuristic; it is quite easy to create rules that improve assignment in some situations but that make the assignment in other situations worse. We explored several promising directions that ended up having too many failure cases before we settled on the current set of rules.

Appendix A Finding Apparent Fills and Stacking Relationships

Live Paint determines the apparent fills and stacking relationships using an iterative algorithm.

1. Mark each region as active, meaning that it has not yet been associated with a path that has an apparent fill.
2. For each closed path, if all regions within it have the same fill, or all regions within it that are not also within another path have the same fill, assign that fill as the apparent fill of the path.
3. Repeat:
   a. For each path p that was just assigned an apparent fill, consider each closed path q that shares a region with p. If each active region that is inside both paths and that borders on p has p's apparent fill, and q contains at least one region outside p that borders on p and has a fill different from p's apparent fill, then path p is apparently above path q. Mark all regions that are within both paths as inactive. Place q on a list of potentially fillable paths. If this generates no potentially fillable paths, terminate.
   b. For each potentially fillable path, if all active regions within it have the same fill, or all active regions within it that are not also within another path have the same fill, assign that fill as the apparent fill of the path. If no path was assigned an apparent fill, terminate.
4. For each triple of paths a, b, and c, with a apparently above b, b apparently above c, and c not apparently above a, if there exists at least one region inside all of a, b, and c, then a is apparently above c.

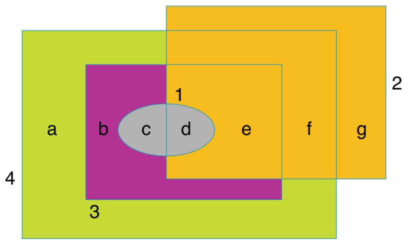


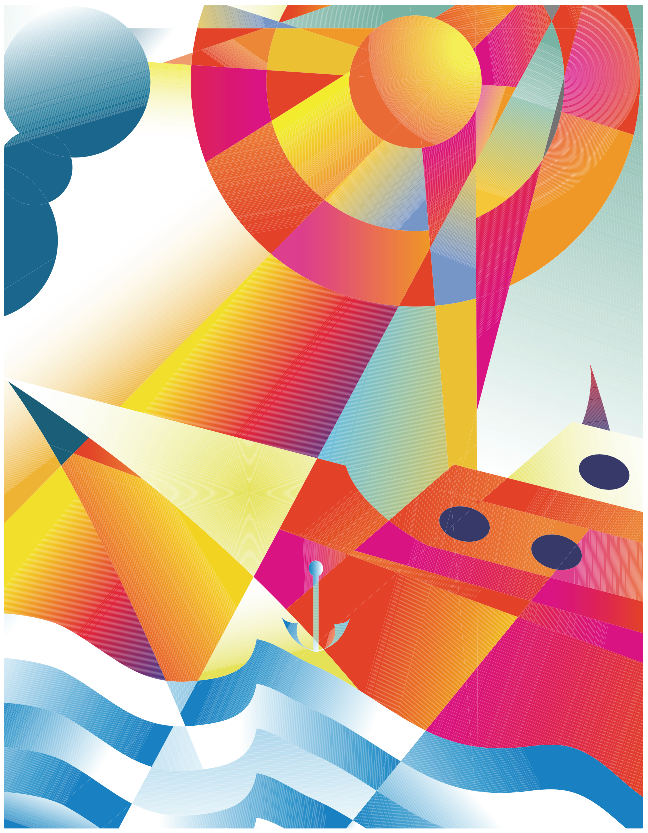
Figure 16: Apparently stacked closed paths

Figure 16 shows some closed paths that appear to have fills and stacking. Paths are identified with numbers and regions with letters. Ellipse 1 appears gray, rectangle 2 appears orange, rectangle 3 appears purple, and rectangle 4 appears green. The paths appear to be stacked in numeric order. The fill and stacking determination finds this out in this order:

1. Step 2 determines that path 1 is apparently gray because all regions within it (c and d) are gray. Path 2 is apparently orange because all regions within it that are not in another path (g) are orange. Path 4 is apparently green for the same reason (a).
2. Step 3a determines that path 1 appears to be above 2 and 3. After making the regions inside 1 (c and d) inactive, it determines that path 2 appears to be above 3 and 4. Regions inside 2 (e, f, and g) then become inactive.
3. Step 3b determines that path 3 is apparently purple because all its active regions (b) have the same fill).
4. Step 3a determines that path 3 appears to be above 4. b becomes inactive.
5. Step 3b finds no new apparent fill, so the iteration ends.
6. Step 4 determines path 1 appears to be above path 4 because 1 appears above 3, 3 appears above 4, and all three paths share a region (c).

Appendix B Performance

We timed Live Paint making single-path changes to three illustrations, two of them shown in Figure 17. "Cruise" is a fairly simple Live Paint illustration. "Building" is of moderate complexity. "Multi-cruise," not shown, is a very complex illustration made by creating multiple offset copies of the paths in "Cruise".



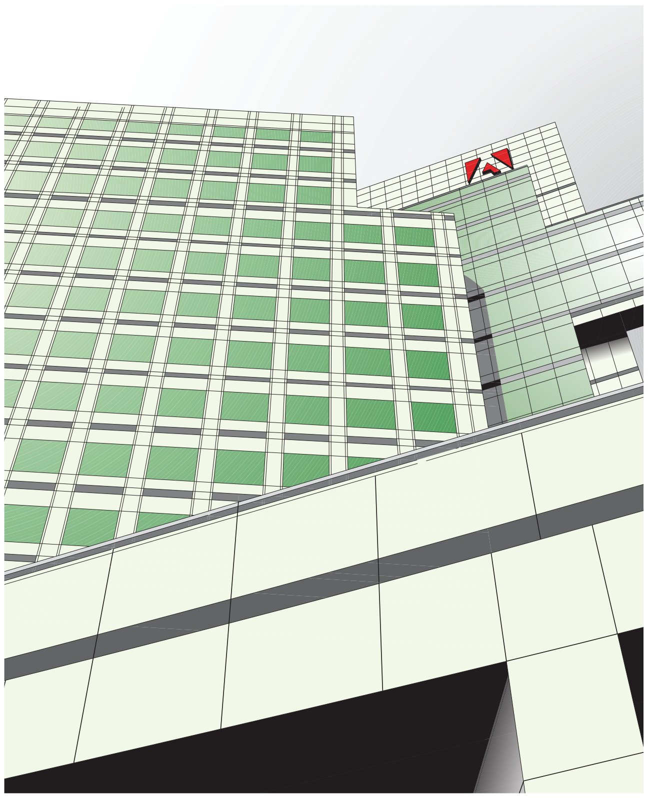


Figure 17: Sample illustrations

All performance figures were measured with a stopwatch using a 2.5 GHz Macintosh PowerPC G5, and include editor overhead to redraw the illustration after the change. Changes to "Cruise" were too fast to measure. "Building" has a high degree of structure and an unusually large number of regions for the number of paths. "Multi-cruise", with over a thousand paths and eight thousand regions, is essentially unusable from a user perspective because of its complexity. Its performance is not good, but it is reasonable.

Example	**# Paths**	**# Regions**	**Response**
Cruise	38	233	instantaneous
Building	210	2596	.4 seconds
Multi-cruise	1154	8388	1.1 seconds

It is also worth noting that the "Building" example could easily be split into 3 simpler, overlapping Live Paint illustrations. This would improve performance considerably and make the drawing easier to edit. Lines from one area of the building would not interfere with those in other areas. Most complex illustrations are like this; it is rare to find one that is not.

References

ADOBE SYSTEMS INC. 2005. *Adobe Illustrator CS2 User Guide*. Adobe Systems Inc.

ALT, H., EFRAT, A., ROTE, G., and WENK C. 2003. Matching Planar Maps. In *Journal of Algorithms*, 49, 2, 262-283.

BAUDELAIRE, P. and GANGNET, M. 1989. Planar Maps: An Interaction Paradigm for Graphic Design. In *CHI'89 Proceedings*, Addison-Wesley, 313-318.

BENTLEY, J. and OTTMANN, T. 1979. Algorithms for Reporting and Counting Geometric Intersections. In *IEEE Transactions on Computers*, C-28, 9, 643-647

FEKETE, J.-D., BIZOUARN, É., COURNARIE, É., GALAS, T., and TAILLEFER, F. 1995 TicTacToon: A paperless System for Professional 2D Animation. In *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, ACM, 79-89.

GANGNET, M., HERVÉ, J.-C., PUDET, T, and VAN THONG, J.-M. 1989. Incremental Computation of Planar Maps. In *Computer Graphics*, *23*, 4, ACM, 345-354.

GANGNET, M., VAN THONG, J.-M., and FEKETE, J.-D. 1994. Automatic Gap Closing for Freehand Drawing. ACM SIGGRAPH 94 Technical Sketch.

HOBBY, J. 1999. Practical Segment Intersection with Finite Precision Output. In *Computational Geometry 13*, Elsevier, 199-214.

MACROMEDIA, INC. 2005. *Macromedia Flash 8: Using Flash*. Macromedia, Inc.

NEUHAUS M. and BUNKE, H. 2004. An Error-tolerant Approximate Matching Algorithm for Attributed Planar Graphs and its Application to Fingerprint Classification. In *Proceedings of the 10th International Workshop on Structural and Syntactic Pattern Recognition*, Springer-Verlag LNCS 3138, 180-189.

WILEY, K. and WILLIAMS, L. 2006. Representation of Interwoven Surfaces in 2 1/2 D Drawing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 65-74.

YOUNG, M. and HALVORSON, M. 2003. *Microsoft Office System Inside Out—2003 Edition*. Microsoft Press.