Painting with Triangles

Mark D. Benjamin
Princeton University

Stephen DiVerdi
Google

Adam Finkelstein
Princeton University

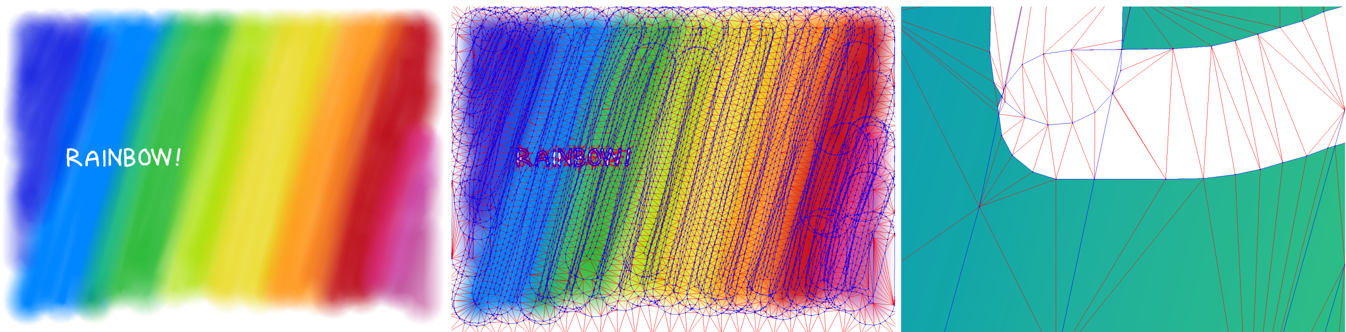


Figure 1: Left: *A vector painting made in our system exhibits both smooth gradients painted by soft brushes as well as infinitely sharp boundaries painted by hard brushes.* Middle: *The painting is represented by a planar triangle mesh.* Right: *Closeup showing the bottom of the letter B reveals both the sharp boundaries on the white region and a smooth gradient from blue to green in the background.*

Abstract

Although vector graphics offer a number of benefits, conventional vector painting programs offer only limited support for the traditional painting metaphor. We propose a new algorithm that translates a user's mouse motion into a triangle mesh representation. This triangle mesh can then be composited onto a canvas containing an existing mesh representation of earlier strokes. This representation allows the algorithm to render solid colors and linear gradients. It also enables painting at any resolution. This paradigm allows artists to create complex, multi-scale drawings with gradients and sharp features while avoiding pixel sampling artifacts.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities

Keywords: triangle mesh, digital painting, vector graphics

1 Introduction

All computer painting programs must store data through rasterization or vectorization. However, traditional computer painting programs take inputs of the same type as the stored data. For example, programs like Adobe Photoshop and GIMP allow a user to paint how he would on a physical medium. By letting the mouse act as a brush, a user may color individual pixels on the screen. By contrast, programs like Adobe Illustrator and Inkscape let a user paint using geometric primitives such as lines and Bézier curves. The program then stores these geometric primitives in order to render the drawing. Although using vector graphics is likely less intuitive, it has some distinct advantages including a compact representation, infinite resolution, and easier manipulation.

In this paper we introduce a system which uses a different representation than traditional raster or vector graphics. The system uses a triangle mesh to store the data of the painting. Since triangles are a geometric primitive used in vector graphics this approach achieves the same advantages traditional vector graphics have. However, the representation allows the user to paint with vector graphics without dealing directly with the underlying implementation.

In order to paint in our program the user simply drags his mouse on the screen to make strokes akin to the process in a raster graphics program. Upon mouse-up the stroke is converted to our underlying triangle mesh representation. In this way a user may paint using vector graphics without worrying about the representation.

This procedure allows painting at any scale. A user can make large strokes while zoomed out and then zoom in to make fine details. All of this can happen without loss of quality since the data is stored using triangles instead of pixels, which can represent sharp boundaries at any orientation. Furthermore, the use of smooth shading when rendering the triangles allows for vector representation of gradient effects. We take advantage of this feature to create soft, airbrush-like strokes, which are difficult to create in vector drawing software.

To transform a user's mouse inputs to a triangle mesh, our system uses a combination of rasterization and marching squares to find stroke contours. From the contour it performs a Delaunay triangulation and then merges this triangulation of the new stroke with the triangle mesh of the existing canvas. This process yields a new canvas with the combined strokes.

The system reduces the barrier to creating vector graphics by enabling a painting-like interface, with support for sharp and smooth edges. This in turn allows artists to focus on the painting and not the way in which they paint, while still getting the benefits of vector graphics. We demonstrate our method by creating digital paintings with a range of effects and resolutions, including zoom factors of up to 500,000:1 which implies an effective resolution equivalent to an image with more than 10^{17} pixels.

2 Related Work

Previous work has focused on converting images into vector graphics to take advantage of the efficient storage, easy editing, and infinite resolution it provides. For example the method of Lecot

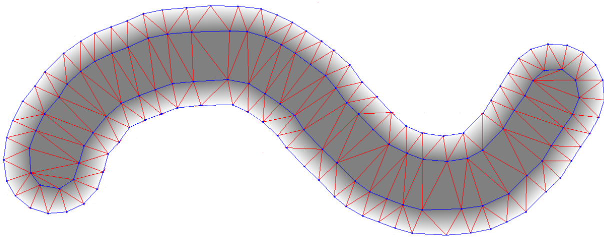


Figure 2: *Example smooth stroke triangulation. Blue segments are boundary edges, and small blue squares denote mesh vertices.*

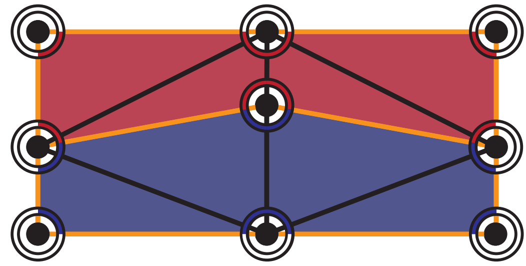


Figure 3: *Color arcs. Orange lines are boundary edges, and together with black lines form the triangle mesh. Each point is given one or more color arcs which store the color of triangle corners in that angular range. Points in the middle row have red arcs on top and blue below, yielding triangles with those colors, separated by a hard boundary.*

and Lévy [2006] minimizes an energy function to segment an image into a number of regions that are bounded by cubic splines and filled with solid colors or gradients. A different approach by Lai et al. [2009] automatically creates gradient meshes with support for holes. Our work builds on the representation introduced by Liao et al. [2012], who showed that a photo can be converted into a triangle mesh. They demonstrated that natural images can be concisely represented by triangles, using mesh simplification, and that various image editing operations can be applied in this representation, including an abstraction technique similar to that of DeCarlo and Santella [2002]. In principle, one could create a digital painting in a conventional program and then use one of these approaches to convert to a vector representation. In contrast our paper shows how to paint directly in this representation, offering several advantages including efficiency and the ability to handle imagery with enormously higher effective resolution.

Other work has proposed novel vector graphic data structures. For example Frisken et al. [2000] introduced Adaptively Sampled Distance Fields (ADFs), which specify a signed distance function to a surface in 3D (or a curve in the plane). Rendering the shape requires sampling the function to determine whether or not a pixel is on the shape. A quadtree accompanies the distance function to specify where higher sampling rates are necessary. The method of Bremer et al. [2001] further describes how ADFs can be created and used. However, it remains unclear how ADFs might be used in a general painting program where strokes with solid colors and gradients are composited on top of one another.

There have been a few projects that enable painting directly with vector data. Most similar is the work of DiVerdi et al. [2012; 2013], which uses many polygonal paint splats to create a watercolor-like effect. These arbitrary polygons are costly to render however, and "smooth" effects are only created via many overlapping transparent polygons, which results in a very large amount of geometry. The method of Ando and Tsuruno [2010] also uses a dynamic vector representation for their 2D fluid simulation to create marbling patterns stored as Bézier silhouettes. They developed a simulation that adaptively updates the Bézier to maintain nice contours, but the complexity of the document quickly grows too large to compute interactively. Finally, Asente and Carr [2013] implemented a feature in Adobe Illustrator to create contour gradients from shapes, which fill the shapes with small patches of linear gradients to create a bevel-like effect. Contour gradients can be used to control opacity to create soft strokes, but each stroke must be stored as a separate (aggregate) object rather than a single flattened canvas representation.

Another line of research has explored entirely new ways of painting. Orzan et al. [2008] introduced *diffusion curves* – a new painting technique where artists specify edges and colors for those edges. Their system then solves a Poisson equation to diffuse the colors between boundary conditions specified at the edges. Similarly the method of McCann and Pollard [2008] lets artists paint in the gradient domain. Both of these approaches yield creative new aesthetic ranges beyond what is straightforward in the traditional pixel-based painting model. Nevertheless, they represent a new painting metaphor for the artist, rather than a vectorized version of the conventional digital painting approach familiar to artists.

Work has also been done in multi-resolution painting using raster graphics. Early work with Pad++ by Bederson and Hollan [1994] explored the possibility of having an infinite resolution canvas. However, rather than a painting program, this work focused on creating an interface builder for developers. The method of Berman et al. [1994] stores image information in a quadtree so that different parts of the image can have different levels of detail. This means the image effectively has an infinite level of detail, yet since it is stored in raster form there are still limitations. For example a stroke painted at coarse detail will still look blurry when zoomed in. The strokes are limited to the resolution they are painted at. Similarly the method of Carr and Hart [2004] also allows the development of multi-resolution images using rasters, and as such suffers from the same limitations. One way to reduce blurriness in raster images of sharp features is proposed by Ramanarayanan et al. [2004]. By seprately storing where important boundaries are, their system can avoid excessive interpolation. However, such a method requires creating such a data structure with those stored boundaries. Another multi-resolution approach by Perlin and Velho [1995] uses procedural textures to avoid any fundamental limit on resolution. Although useful, generating procedural textures can be difficult for artists.

Our work builds on these findings. However, our goal is to enable artists to create vector graphics through standard painting techniques without having to consider the underlying representation.

3 The Canvas Model

The data necessary to render the painting resides in the canvas. However, the canvas contains more than just a triangle mesh, and understanding the underlying data structures is important for understanding the rest of the algorithm.

3.1 Triangle Mesh

When a user paints, he generates points that are stored on the canvas. These points are connected to form a triangle mesh. In every triangle the vertices are colored to either produce a solid color or a linear gradient. In fact a given point that is part of multiple triangles can take on different colors in each triangle. Note

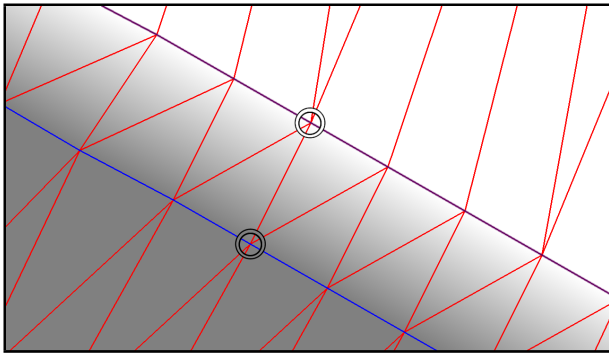


Figure 4: *A close look at a gradient stroke. The blue edges are boundaries where the points have gray color arcs as shown. The purple edges are boundaries where the points have white color arcs. This gives triangles connecting the two boundaries a linear gradient from gray to white.*

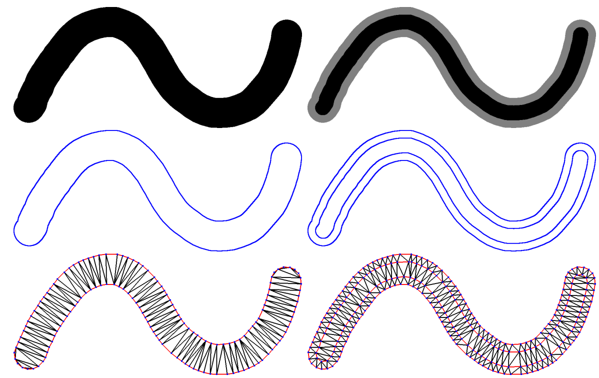


Figure 5: *Left: A hard stroke is rasterized, the contour is found, and a triangle mesh is generated. Right: A soft stroke is rasterized in two different shades of gray, two contours are found, a triangle mesh is generated with a boundary separating the inner and outer parts of the stroke.*

that although the points are persistent as the canvas is updated, the triangles are not. That is, triangulation of the mesh points is not unique, and different, equivalent triangulations may be used throughout the painting process. Thus, we refer to them as "mesh points" rather than "vertices" (which implies a particular graph structure).

3.2 Boundary Edges

Although the triangles may change over time, they must abide by constraints we call *boundary edges*, each of which connects a pair of points and specifies that no triangle can cross it. Like mesh points, boundary edges are persistent. So as new triangles are created by the triangulation algorithm, they may never cross a boundary edge. This partially restricts the set of valid triangulations of the points. Figure 2 shows boundary edges for a soft stroke.

3.3 Color Arcs

Since triangles are not persistent they cannot store color data. Storing colors in points works well when the color is the same for all triangles connected to that point. However, when a point lies on a sharp boundary it may take on different colors in triangles on either side of the boundary. The need for the same point to take on different colors in different triangles lead to the development of color arcs (see Figure 3).

A color arc has three components: a start direction, an end direction, and a color. Determining the color of a vertex in a given triangle requires two steps. First, the vector from the point to the centroid of the triangle is calculated. Second, the color arc is found which contains this vector. Containing the vector means one must turn clockwise from the starting direction to reach it and counter-clockwise from the ending direction. The point takes on the color of the arc in which the vector from the point to the centroid resides.

A point may have more than one color arc, and they must satisfy three conditions. First, they must be disjoint. Second, they must cover all 2π radians around the point. These first two conditions ensure that for any triangle a point will take on exactly one color. Third, the start and end vectors must align with boundary edges. This condition makes all colors and gradients emanate from boundaries.

3.4 Rendering

Once the triangle mesh has been generated it is simple to render. In each triangle the vertices take on a given color based on their color arcs and the centroid of the triangle. If the three vertices in the triangle take on the same color value then the triangle will be flat shaded. If instead the vertices take on different color values then there will be a gradient over the triangle.

Since the conditions on color arcs make colors emanate from boundaries it is easy to create regions (e.g. a brush stroke boundary) containing either a solid color or linear gradient. A region will have a solid color (e.g. a hard brush stroke) if the boundaries that surround it all take on the same color. If instead a region is surrounded by two boundaries, one which has a given color and the other which has a transparent version of that color then there will be a linear gradient in that region between the two boundaries (e.g. a soft brush stroke). Figure 4 shows the color arcs on such a soft stroke.

4 Triangulating Strokes

Here we consider how to convert a user's cursor motion into a triangle-mesh representation of their stroke. This is done in two steps. First, the stroke is rasterized and the contour is extracted from the raster. Second, the contour is converted into points and boundaries and triangulated.

4.1 Converting Strokes to Contours

A set of contours define the outline of a stroke. For simply connected strokes, those without holes, one contour describes the whole stroke. However, strokes with holes, such as a circle or figure-eight, will have more than one contour. Converting a user's mouse motion into a set of contours requires two steps. First, the user's mouse movements are captured by a set of polygons which can be rendered to yield a rasterized version of the stroke. This rasterization happens at the resolution of the current window. Therefore whatever the artist sees on the screen is the basis for the vectorization. Second, to obtain the contours the stroke is rasterized in black and marching squares [Lorensen and Cline 1987] is used to extract iso-contours of 50% gray. This returns a very dense set of points, which we reduce through a pruning process to get a sparse but accurate representation of the contour. The pruning procedure

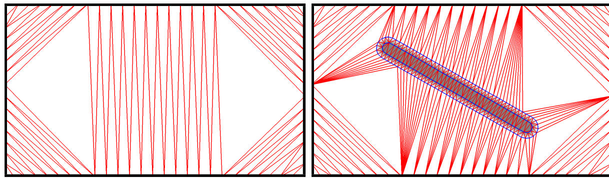


Figure 6: *The figure on the left shows an empty canvas with a number of points and boundaries defined along the edges to provide the initial triangulation. The figure on the right shows how a stroke is composited onto the empty canvas. Blue edges are boundary edges. Red edges form the triangle mesh with the blue edges.*

walks around the contour to form an approximating polygon P, greedily eliding vertices whenever they lie within a small threshold distance of P. We use a threshold of 0.5 pixels at the current zoom level, which works well empirically.

4.2 Converting Contours to Triangles

Now that we have the contour we need to create a triangulation for the stroke. First we consider the case of a hard brush, which contains a single contour (Figure 5-left). Points on the contour describe the shape of the stroke, while the adjoining segments describe its boundary. A triangle mesh must contain all the points and stay within the boundaries. For example, a concave contour like the outline of the letter C should not contain triangles inside the concavity, since such triangles would be outside of the boundary. These boundaries will prove to be crucial when compositing a stroke onto the canvas. We perform constrained Delaunay triangulation on the given set of points and boundaries using Shewchuk's implementation [2002], and the output is a mesh that approximately represents the drawn stroke.

4.3 Soft Strokes

The above procedure converts a hard brush stroke into a triangle mesh. A more challenging case is converting a soft brush stroke with linear gradient into a triangle mesh. A soft stroke has two components, an inner hard stroke and an outer gradient stroke. Since the triangles in the inner and outer parts of the stroke must be colored differently, none of the triangles may cross the boundary between the hard and soft regions.

This representation needs a slightly different procedure. The user's mouse motion is captured in two sets of polygons. The first set describes the outer soft stroke and the second set describes the inner hard stroke. Next the outer stroke polygons are rendered in 50% gray and the inner stroke is rendered in black. This means the boundary of the outer stroke is the 25% gray iso-contour and the boundary of the inner stroke is the 75% gray iso-contour.

Again, using marching squares these contours are extracted and then pruned. The boundaries and points derived from the contours are triangulated as in Section 4.2. The resulting mesh represents the soft stroke where no triangle crosses the boundary between the soft and hard regions of the stroke. Points on the contour of the inner stroke will have color arcs with the stroke color, while the points on the contour of the outer stroke will have transparent color arcs. This gives all triangles connecting the inner and outer strokes a linear gradient as seen in Figure 4. Figure 5-right shows the process for triangulating soft strokes.

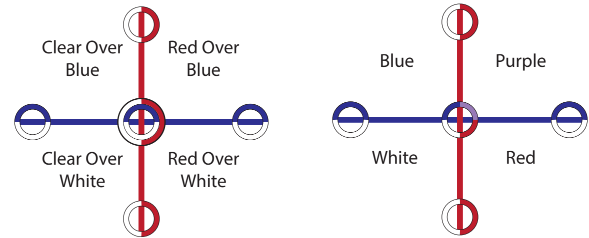


Figure 7: *Left: A red stroke is composited over a blue stroke. An intersection point is inserted in the center and four color arcs must be determined. Right: The four colors are resolved as the four combinations of the red stroke's color arcs composited over the blue stroke's color arcs.*

5 Compositing Strokes

Once the stroke has been converted to triangles by the above procedure it is necessary to composite the stroke onto the canvas. This is accomplished in two steps. First, the points and boundaries from the triangulation of the stroke are added to the canvas and a new triangulation is found. Second, the color arcs for the old and new points are updated. We also discuss the challenge intersecting boundaries pose for the algorithm, as well as a technique to reduce the number of triangles that are retriangulated on the canvas.

5.1 Adding Points to the Canvas

Every canvas begins with an initial set of points distributed along the edges. These points have boundaries connecting them. These points are originally part of a number of triangles which divide the canvas.

In the simplest approach, the points and boundaries from the new stroke are added to the list of points and boundaries already stored on the canvas. These points and boundaries are all triangulated using constrained Delaunay triangulation as in Section 4.2, to produce a new mesh (Figure 6). The new mesh contains all of the points of the old and new strokes. Since boundary edges form constraints in the triangulation, stroke edges are preserved. This means that no triangle will cross the boundary of a stroke. Without this constraint, it would be impossible to correctly color the triangles since they would straddle color arcs.

5.2 Determining Colors

Once a new mesh is formed the color arcs of the new and old points must be computed. All color compositing is done with regular RGB alpha blending. The color arcs for new points depend on two values, the color of the stroke and the color of the canvas at that point. The points on a hard stroke will have two color arcs. One arc describes the inside of the stroke, which takes the color of the stroke composited over the color of the canvas at that point. The other arc describes the outside of the stroke, which is just the color of the canvas at that point. The points on a soft stroke will each have one color arc. The color arc for points on the inner contour is the color of the stroke composited over the color of the canvas at that point. The color arc for points on the outer contour is the color of canvas at that point (because the outer edge of the soft stroke is completely transparent). This arrangement produces a linear gradient across the triangles separating the inner and outer contours.

The color arcs for old points remain unchanged, unless part of the new stroke covers it. In this case the colors in each arc must be

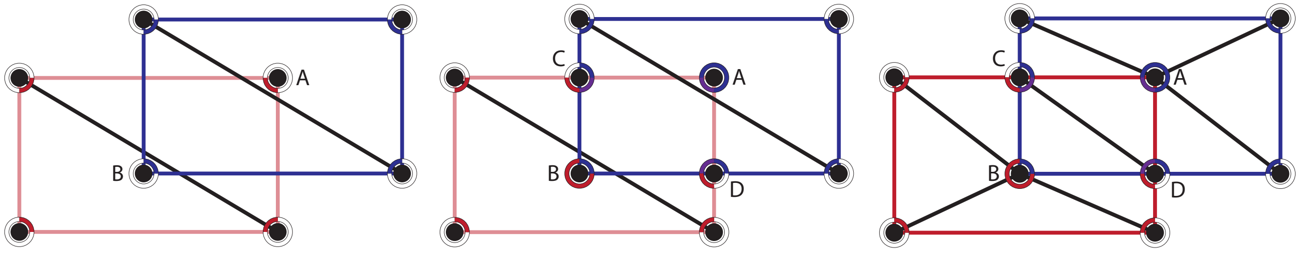


Figure 8: *Left: A blue stroke is composited over a red stroke. Point A is covered by the the blue stroke and point B is on top of the red stroke. Middle: New color arcs are determined. Both of A's color arcs are composited with the blue stroke which turns red to purple and transparent to blue. Similarly both of B's color arcs are composited on top of the red stroke which turns blue to purple and transparent to red. Two intersection points C and D are added. Their color arcs are determined by examining neighboring points. Right: A new triangle mesh is generated to preserve boundaries. Note that triangles in the intersection will be purple.*

replaced with the stroke color composited over the old color of the arc. If the new stroke is soft then the color at the location of the old point must be determined through bilinear interpolation before compositing.

5.3 Intersection Points

When the constrained Delaunay triangulation is computed for a set of points with intersecting boundary edges, it must insert a new point at the intersection of the edges (otherwise at least one triangle would have to cross a boundary). These points are introduced with no color information, whereas the final colors for all of the other points in the mesh have already been determined.

An intersection point lies on two boundaries and therefore has four boundary edges emanating from it. To determine the colors of an intersection points requires knowing the colors associated with other points on the boundaries. For both boundaries, we search both directions to the first point with color data. Note this may not be the first point on each side, since this intersection point may be adjacent to *other* intersection points also without color data. For both boundaries, the colors in the color arcs are linearly interpolated. This yields four average colors, two from the new stroke boundary and two from the old stroke boundary. The colors from the new stroke are composited over the colors from the old stroke to yield four new colors, one for each region defined by the boundary intersections (see Figure 7). For an example of how color arcs are determined for both intersection and regular points see Figure 8.

5.4 Finding Modified Points

As described so far, our retriangulation procedure has two substantial drawbacks. First, it processes many points that do not need consideration which is inefficient. Second, since it acts on the whole canvas, local modifications may have global effects. Instead our approach is to determine which triangles can remain unchanged and which triangles need to be included in the re-triangulation (see Figure 9). To accomplish this a static grid is generated and all of the cells that contain a triangle from the new stroke are marked. The set of triangles from the old triangulation that intersect these grid squares are found. The set of edges from these triangles that do not intersect these grid cells form a ring around the area of our new triangles. The edges in this ring become constraints in the triangulation. All of the points and constraint edges from the old triangulation are included as well as the points and constraint edges from the new triangulation. Triangulating these points and edges gives the new geometry for the modified area and has no effect on any other part of the canvas. Therefore, the old triangles from the rest of the canvas can be reused, and their union with the new triangle mesh represents the new canvas.

6 Results and Discussion

Paintings made with our system in a range of visual styles are shown in Figures 1 and 11–14. The images shown in Figure 12 reveal a drawing with an extreme zoom range up to 524,000:1, for an effective resolution equivalent to an image with more than 10^{17} pixels. At this zoom level our renderer begins to suffer from artifacts at the limits of single floating point precision; this could be ameliorated by switching to double-precision. Nevertheless, this range far exceeds limits imposed by existing vector graphics software such as Adobe Illustrator, which has a maximum zoom ratio of 64:1. Furthermore, our ability to make paint strokes of smooth gradients in a flat vector representation (e.g., Figure 1) is unique.

Next we consider how the complexity of the algorithm grows both in time and geometry. The results from this analysis and our experience with the system have lead to a number of observations.

6.1 Complexity

To find the growth rates of the algorithm we performed two experiments. First, the program drew fifty random strokes without zooming. Next, the program drew fifty random strokes with random zoom level for each stroke. Geometry grows fastest when many strokes are composited on one another. When not zooming, strokes must cover the same area of canvas over and over. This results in more geometry and therefore takes longer to render. The zooming case explores different areas of the canvas and therefore results in less overlapping geometry. This situation should result in less complex geometry and therefore take less time.

Figure 10a shows the zooming case grows linearly while the non-zooming case appears to grow quadratically. This is expected since in the non-zooming case every new stroke has the chance to overlap all of the old strokes. Therefore each new stroke does not just add a constant amount of geometry, but due to intersections with previous strokes extra intersection points are added. This results in a linear increase in the amount of geometry per new stroke, yielding a quadratic growth rate overall. However, in the zooming case each stroke rarely overlaps previous strokes, and therefore intersection points are rarely added. In this case the geometry increases by a constant amount per stroke, which yields a linear growth rate.

Figure 10b gives a similar result for the cumulative time strokes take to render. Both the zooming and non-zooming cases grow non-linearly over time. However, the non-zooming case takes longer for

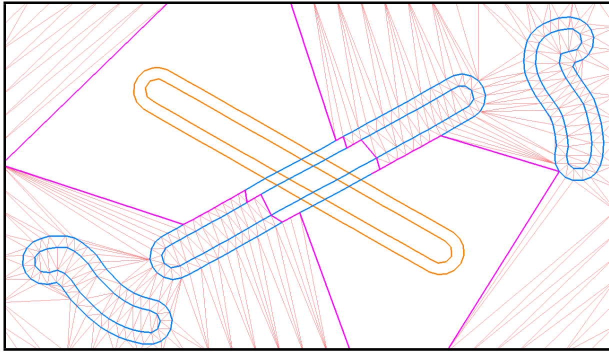
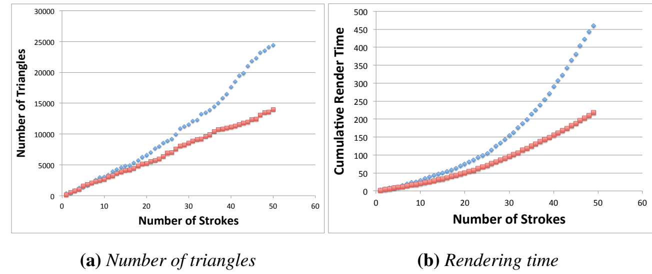


Figure 9: *An example of how our system determines which triangles to save and which to retriangulate. Blue lines represent the boundaries of old strokes, orange lines are the boundaries of the new stroke, red lines are the triangles that will be reused, and purple lines define the area that needs to be retriangulated. The purple lines are edges of triangles that intersect the new stroke. Since they interesect the new stroke they are no longer valid and must be retriangulated. All other triangles on the canvas can be reused.*



(a) *Number of triangles* **(b)** *Rendering time*

Figure 10: *For paintings of 50 random strokes, the left graph shows the number of triangles in the canvas as of stroke x, and the right graph shows the cumulative render time of the painting. The blue points are the non-zooming case, while the red points are the zooming case.*

the reasons described. For example, the first ten strokes in both cases take about 2 seconds to render, while the last ten strokes take about 20 seconds in the non-zooming case and 7 seconds in the zooming case. It is not unexpected that even the zooming case grows non-linearly since even though the geometries involved are not becoming very complicated, the geometry of the entire canvas is increasing linearly and must be considered even if the entire canvas is not re-triangulated.

This suggests that canvas simplification is of great importance. The case that it is most likely to be helped is the non-zooming case with complex geometry. When lots of strokes overlap there is usually extra geometry that does not add much detail to the drawing. By removing this extra geometry the algorithm should achieve a steady growth rate.

Beyond reducing the complexity of the geometry, the speed of the algorithm might be improved by utilizing the GPU more heavily. For example, Nehab and Hoppe [2008] demonstrate an approach for using GPUs to speed up rending of more traditional vector graphics.

6.2 Stability

While Shewchuk's implementation [2002] of Delaunay triangulation is impressively robust, there are still certain pathological geometries that can cause it to fail. Specifically, long skinny triangles and vertices that are too close together seem to trigger triangulation failures. Detecting these situations and fixing them, or even better modifying the geometry construction to avoid them, would solve this problem. For example, identifying if a new vertex is too close to an existing one, and if so, merging them. Another example is to subdivide long edges to reduce the incidence of skinny triangles. The impact these solutions would have on processing time is something that needs to be explored.

6.3 Performance

Performance is measured on a 1.8 GHz Core i5 MacBook Air with 8GB of RAM and an Intel HD 4000 graphics card. The current implementation uses Python and PyOpenGL. Python's performance is slower than a lower level implementation in C++ would be. The preview rasterization of a new stroke on the canvas is interactive with no lag. However, the actual process to take a stroke from rasterization to triangle representation on the canvas can be slow. On a blank canvas, strokes take less than one second to process. However, on a canvas with lots of geometry a stroke that overlaps that geometry can take several seconds to render. Some of this could be alleviated by a carefully optimized implementation. However, there are many calculations necessary for the old points affected by the new stroke and the points associated with the new stroke itself. Since these geometries can become arbitrarily complicated (in pathological cases), these calculations can take arbitrarily long.

6.4 Simplification

No matter how good the implementation, the fact that paintings can become arbitrarily complicated as more and more strokes are added means there must be some way to simplify the mesh. In most situations where many strokes overlap one another, most of the geometry is redundant. For example, when an opaque stroke is laid down on top of a complex geometry, that complex geometry no longer has any useful information since the outline of the opaque stroke completely defines it (all vertices are the same color). Furthermore, even with translucent or feathered strokes, a large enough number of composited strokes may produce complex geometry that does not add much to the visual appearance of the canvas. Some of this geometry can be reduced without noticeable changes. To accomplish this, our proposed simplification algorithm would identify a vertex that can be removed via an edge collapse by measuring the change the collapse would cause in the image. The change is computed as the magnitude of the color shift and the translation of a boundary. If these deltas are below a threshold, then the collapse can proceed. This process could be done when a stroke is added to the canvas, or continually in a background thread to avoid increasing apparent latency. Though we have yet to implement this simplification algorithm, it is important to ensure a painter can continue to iterate on a painting without the program becoming too slow. Finally we note that Liao et al. [2012] demonstrated that this kind of simplification can be effective at reducing mesh complexity, at least for natural imagery. In their case they were working on an offline process for the entire image, whereas our problem needs to be interactive on mouse-up (or as a background thread) but fortunately only needs to operate on regions of the mesh that have changed during recent painting.

7 Conclusion and Future Work

Our system enables a user to paint like they would in a raster graphics program to create vector graphics. The core element is a triangle mesh representation of the painting. We have created an

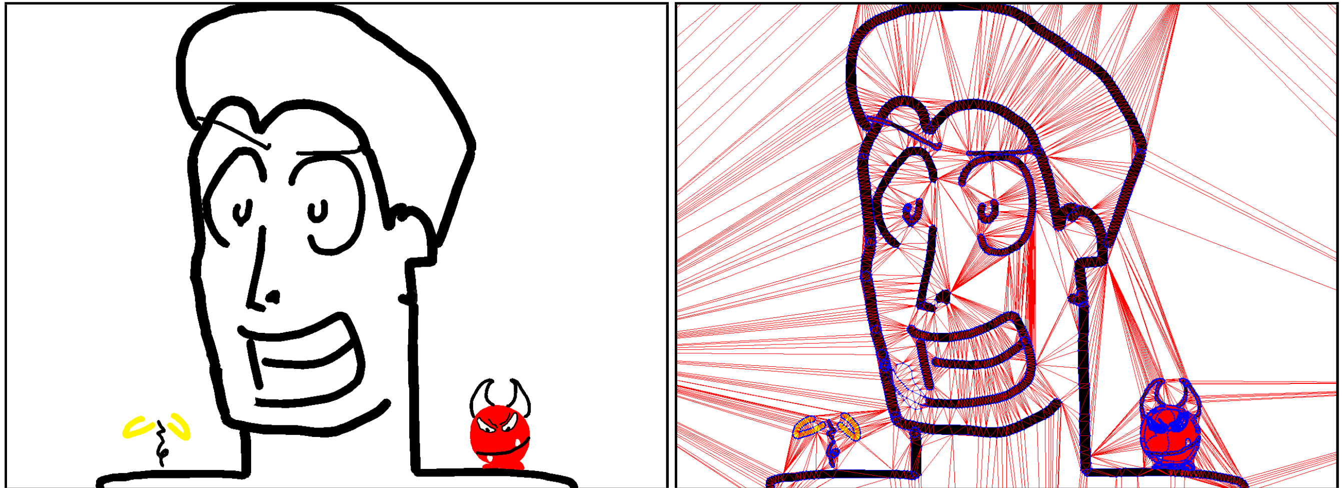


Figure 11: *This drawing uses about 70 strokes, resulting in 9030 triangles.*

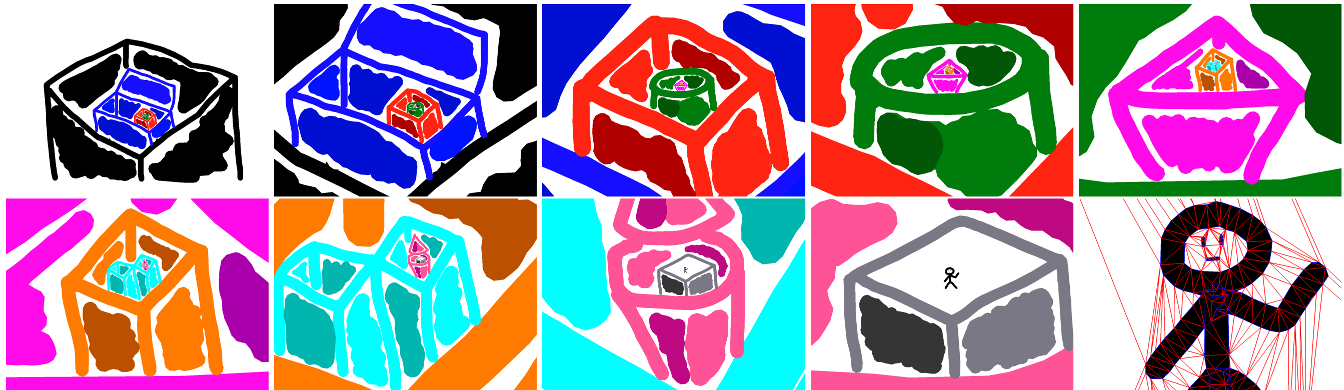


Figure 12: *This single painting shows an extreme zoom; the final closeup image is at zoom level 524,000:1.*

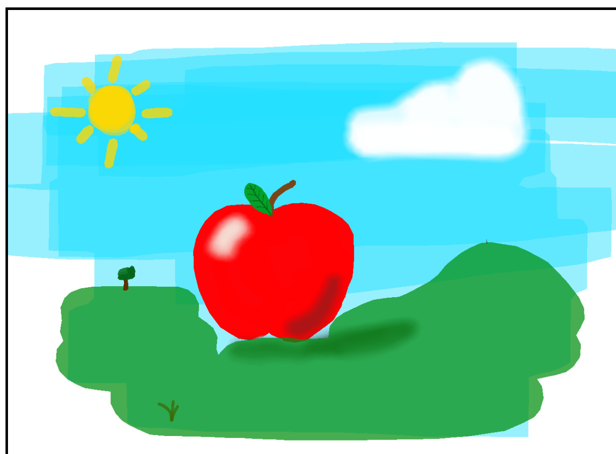


Figure 13: *Two more example paintings exploring different styles.*

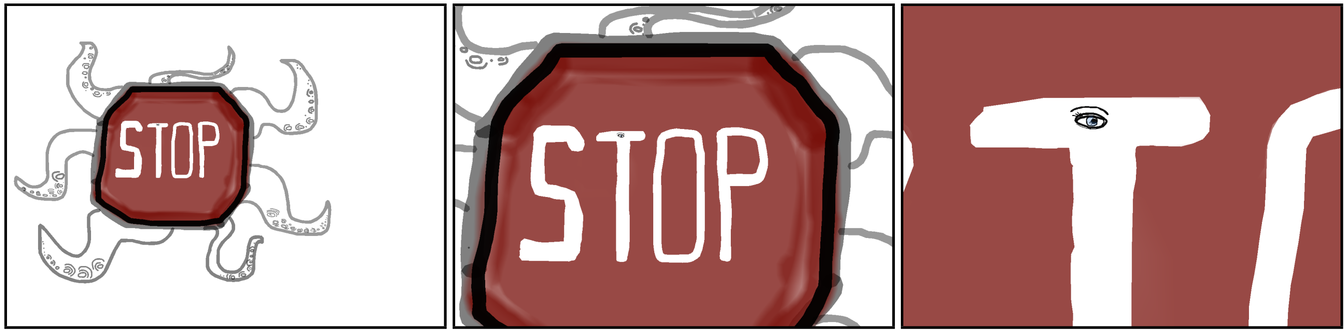


Figure 14: *This painting shows soft and hard strokes at a variety of scales. Middle and right images are zoomed in.*

algorithm to convert a user's mouse motions into a triangle mesh, and then composite it onto the preexisting mesh containing the previous strokes.

Aside from the **simplification** challenge described in Section 6.4, there are a number of other interesting areas for future work, including:

Curved boundaries. In our implementation, curve boundaries are represented as polylines. We believe it would be relatively straightforward to modify these boundaries to be represented by higher-order curves, for example a Catmull-Rom spline passing through the vertices of the mesh. This would offer higher-quality paths for the same sampling rate, but would require subdividing the path at rendering time based on the image resolution. Today such operations can typically be performed on graphics hardware with little impact on runtime performance.

Geometric operations. Because the underlying representation is vector in nature, it should robustly support warping and other geometric operations. It should also be easy to provide other kinds of useful tools to the artist like a "smoothing brush" which could locally adjust the geometry and/or the colors to soften a region. Finally, it would be interesting to offer this kind of painting interface on non-planar canvasses, for example allowing an artist to paint directly onto a manifold representing a shape in 3D.

References

ANDO, R., AND TSURUNO, R. 2010. Vector fluid: A vector graphics depiction of surface flow. In *Non-Photorealistic Animation and Rendering*, 129–135.

ASENTE, P., AND CARR, N. 2013. Creating contour gradients using 3D bevels. In *Computational Aesthetics*, 63–66.

BEDERSON, B. B., AND HOLLAN, J. D. 1994. Pad++: A zooming graphical interface for exploring alternate interface physics. In *User Interface Software and Technology*, 17–26.

BERMAN, D. F., BARTELL, J. T., AND SALESIN, D. H. 1994. Multiresolution painting and compositing. In *SIGGRAPH*, 85–90.

BREMER, P. T., PORUMBESCU, S. D., KUESTER, F., JOY, K., AND HAMANN, B. 2001. Virtual clay modeling using adaptive distance fields. Tech. Rep. CSE-2001-7, University of California, Davis.

CARR, N. A., AND HART, J. C. 2004. Painting detail. In *SIGGRAPH*, 845–852.

DECARLO, D., AND SANTELLA, A. 2002. Stylization and abstraction of photographs. *ACM Trans. Graph. 21*, 3 (July), 769–776.

DIVERDI, S., KRISHNASWAMY, A., MĚCH, R., AND ITO, D. 2012. A lightweight, procedural, vector watercolor painting engine. In *Interactive 3D Graphics and Games*, 63–70.

DIVERDI, S., KRISHNASWAMY, A., MĚCH, R., AND ITO, D. 2013. Painting with polygons: A procedural watercolor engine. *IEEE Trans. Vis. Comp. Graph. 19*, 5 (May), 723–735.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *SIGGRAPH*, 249–254.

LAI, Y.-K., HU, S.-M., AND MARTIN, R. R. 2009. Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Trans. Graph. 28*, 3 (July), 85:1–85:8.

LECOT, G., AND LÉVY, B. 2006. ARDECO: Automatic region detection and conversion. In *Eurographics Symposium on Rendering*.

LIAO, Z., HOPPE, H., FORSYTH, D., AND YU, Y. 2012. A subdivision-based representation for vector image editing. *IEEE Trans. Vis. Comp. Graph. 18*, 11, 1858–1867.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH*, 163–169.

MCCANN, J., AND POLLARD, N. S. 2008. Real-time gradient-domain painting. *ACM Trans. Graph. 27*, 3 (Aug.), 93:1–93:7.

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5 (Dec.), 135:1–135:10.

ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (Aug.), 92:1–92:8.

PERLIN, K., AND VELHO, L. 1995. Live paint: Painting with procedural multiscale textures. In *SIGGRAPH*, 153–160.

RAMANARAYANAN, G., BALA, K., AND WALTER, B. 2004. Feature-based textures. In *Eurographics Symposium on Rendering*, 265–274.

SHEWCHUK, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Comp. Geom. Theor. App. 22*, 1–3 (May), 21–74.