A Triangle Mesh Approach to Painting

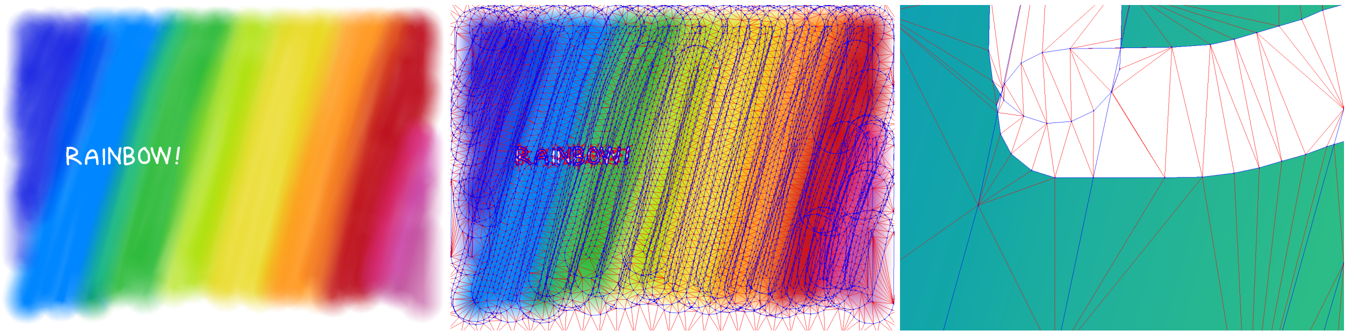


Figure 1: *A vector painting made in TrianglePaint that exhibits smooth gradients painted by soft brushes, as well as infinitely sharp boundaries painted by hard brushes, together in a single flat representation. The triangle mesh is shown in the middle, and the right image is zoomed in on the bottom of the letter B, to show the sharp boundaries and smooth gradients in detail.*

Abstract

Although vector graphics program have a number of advantages, current vector painting programs do not allow an artist to use traditional painting techniques. 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 the previous strokes. This representation allows the algorithm to render solid colors and linear gradients. It also enables painting at any resolution. This gives artists the opportunity to create complex, multi-scale drawings with gradients while avoiding artifacts.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Paint systems I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

Keywords: triangle mesh, 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 TrianglePainter which uses a different representation than traditional raster or vector graphics. TrianglePainter 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 usually very difficult to create in vector drawing software.

To transform a user's mouse inputs to a triangle mesh, TrianglePainter 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.

TrianglePainter 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.

2 Related Works

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 [Lecot 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. Yet another example is [Liao et al. 2012] which converts an image into a triangle mesh. All three of these can convert images into a vector representation, yet none of them allow a user to create a painting in the vector medium.

Other work has proposed novel vector graphic data structures. For example [Frisken et al. 2000] defines Adaptively Sampled Distance

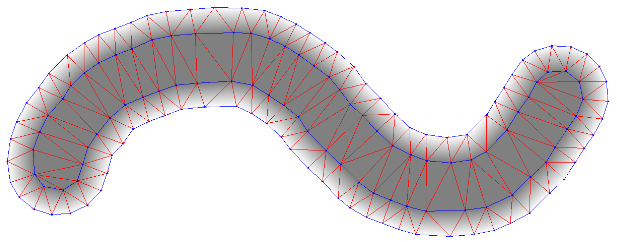


Figure 2: *An example triangulation of a smooth stroke. Blue lines are boundary edges. Black points form the triangle mesh with the blue lines.*

Fields (ADF). ADFs specify a signed distance function to a surface. Rendering the shape requires sampling the function to determine whether or not a pixel is on the surface. A quadtree accompanies the distance function to specify where higher sampling rates are necessary. [Bremer et al. 2001] further describes how ADFs can be created and used. However, it is 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. [Ando and Tsuruno 2010] also used 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 series of works have created entirely new ways of painting. Diffusion curves as specified in [Orzan et al. 2008] describe a new painting technique where the 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 [McCann and Pollard 2008] lets artists paint in the gradient domain. Both of these approaches yield fantastic works of art, yet they are really new mediums for artists to explore rather than a vectorized version of conventional painting techniques.

Work has also been done in multi-resolution painting using raster graphics. [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. [Carr and Hart 2004] also allows the development of multi-resolution images using rasters, and as such suffers from the same limitations. 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 deal with the underlying representation. We start from the triangle mesh structure described in [Liao et al. 2012] to build a painting program where the triangle mesh is created while the artists lays down strokes.

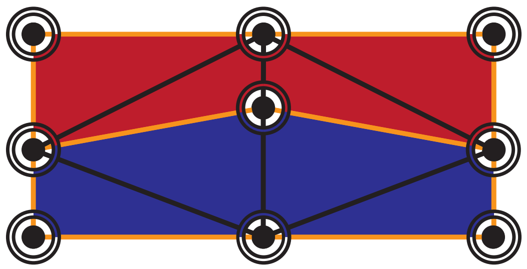


Figure 3: *An example of color arcs. The orange lines are boundary edges. Black lines form the triangle mesh with the boundary edges. Each point is given a color arc which defines its color in that range. Note that the points in the middle row have different colors on top and bottom. This gives blue triangles on top and red on bottom with a hard boundary in between.*

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 that although the points are persistent as the canvas is updated, the triangles are not. That is, the correct triangulation of the mesh points is not unique, and different equivalent triangulations may be used during the painting process.

3.2 Boundary Edges

Although the triangles may change over time, they must abide by constraints known as boundary edges. A boundary edge connects two points and specifies that no triangle can cross it. Like points, boundary edges are persistent. So as new triangles are created by the triangulation algorithm, they may never cross a boundary edges. This partially restricts the set of valid triangulations of the points. Figure 2 shows boundary edges in a soft brush 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

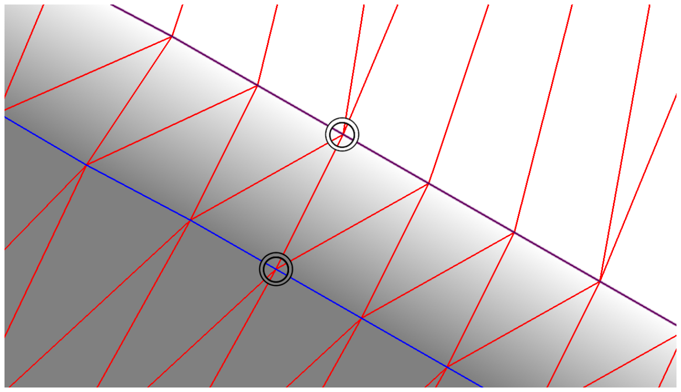


Figure 4: *A close look at a gradient stroke. The blue edges are boundaries where the points have gray color arcs as shown. The purple edges are boundaries where the points have transparent gray color arcs. This gives triangles connecting the two boundaries a linear gradient from gray to white.*

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.

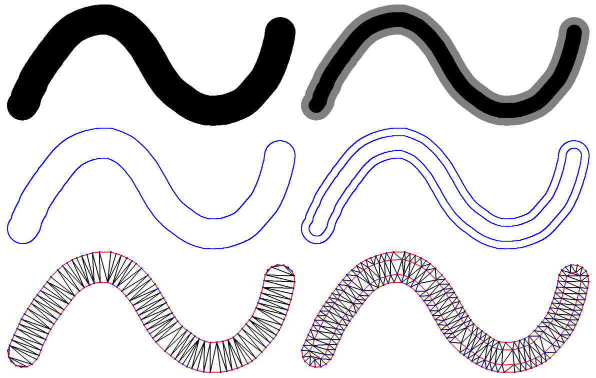


Figure 5: *Left: A hard stroke is rasterized, the contour is found, and a triangle mesh is generated. Right: A soft stroke is rasterized in two different shades of gray, two contours are found, a triangle mesh is generated with a boundary separating the inner and outer parts of the stroke.*

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. 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.

4.2 Converting Contours to Triangles

The contour contains two important pieces of information. First, it describes all of the points of the stroke. Second, segments connecting adjacent points describe the boundaries of the stroke. A triangle mesh must contain all the points and stay within the boundaries. For example, a concave contour should not contain triangles inside the concavity since such triangles would be outside of the boundary. Furthermore, these boundaries will later be crucial when compositing a stroke onto a canvas.

We utilized Jonathan Shewchuk's Triangle library [Shewchuk 2002] to perform constrained Delaunay triangulations on the given set of points and boundaries. The library outputs 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 interesting 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%

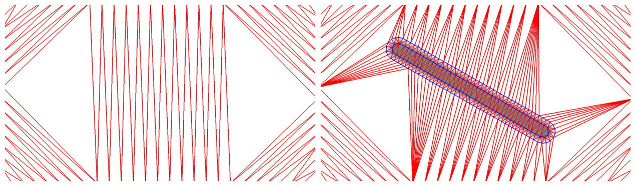


Figure 6: *The figure on the left shows an empty canvas with a number of points and boundaries defined along the edges to provide the initial triangulation. The figure on the right shows how a stroke is composited onto the empty canvas. Blue edges are boundary edges. Red edges form the triangle mesh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

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 given to Triangle and a mesh is returned. This 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 (again see Figure 4). Figure 5 shows the process of triangulating both hard and soft strokes.

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. All of this data is given to Triangle to produce a new mesh (see figure 6). The new mesh contains all of the points of the old and new strokes. Since all of the boundaries are stored and given to Triangle, the stroke edges are preserved. This means that no triangle can cross the boundary of a stroke. Without this constraint, it would be impossible to correctly color the triangles since they would straddle color arcs. Once a new mesh is formed the color arcs of the new and old points must be computed.

5.2 Determining Colors

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. The color for this arc is the color of the stroke composited over the color of the canvas at that point. The other arc describes the

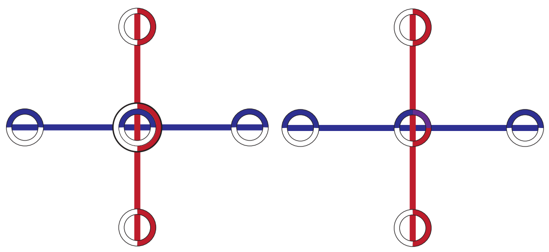


Figure 7: *Left: A blue stroke is composited over a red stroke. Point A is covered by the the blue stroke and point B is on top of the red stroke. Middle: New color arcs are determined. Both of A's color arcs are composited with the blue stroke which turns red to purple and transparent to blue. Similarly 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.*

outside of the stroke. The color for this arc 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 gives any triangle connecting inside and outside contour points a linear gradient from the stroke color composited over the canvas color to the canvas color.

The color arcs for old points remain the same unless part of the new stroke covers it. In this case the colors in each arc must be 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 Triangle is asked to triangulate a set of points with intersecting boundary edges, it must insert a point at the intersection, since without it at least one triangle would have to cross a boundary. These points are introduced with no color information, but 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, travel 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)

An example of how color arcs are determined for both intersection and regular points see figure 8.

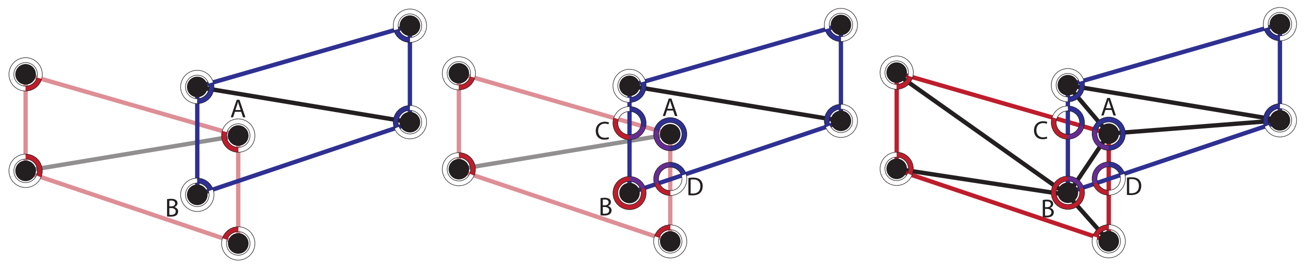


Figure 8: *Left: A blue stroke is composited over a red stroke. Point A is covered by the the blue stroke and point B is on top of the red stroke. Middle: New color arcs are determined. Both of A's color arcs are composited with the blue stroke which turns red to purple and transparent to blue. Similarly 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.*

5.4 Finding Modified Points

The above procedure has two important drawbacks. First, it processes many points that do not need consideration which takes a considerable amount of processing time. Second, since it acts on the whole canvas, local changes can have global effects. To avoid this problem the algorithm determines which triangles can remain 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 which 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. The new triangulation combined with the old triangles give a new triangle mesh representation of the entire canvas.

6 Discussion and Results

Paintings made with TrianglePainter can be seen in Figures 11, 1, 12 and 13. Here we first consider how the complexity of the algorithm grows both in time and geometry. The results from this analysis and our experience with TrianglePainter 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 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 stroke yielding a quadratic growth rate. However, in the zooming case each stroke rarely overlaps 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 the reasons described. 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.

6.2 Stability

While the Triangle library 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

The current implementation uses Python and PyOpenGL. Python's performance is slower than a lower level implementation in C++ would be. The drawing of the canvas is quick 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

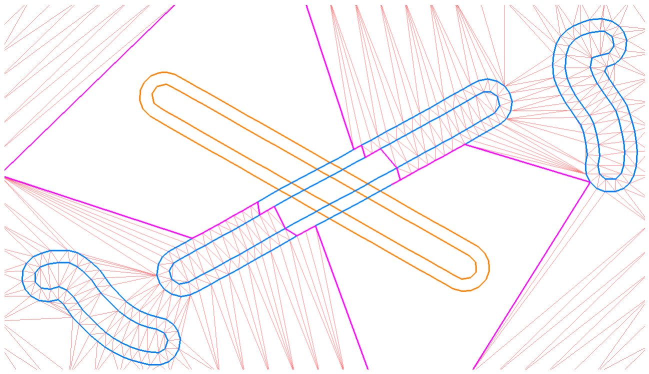
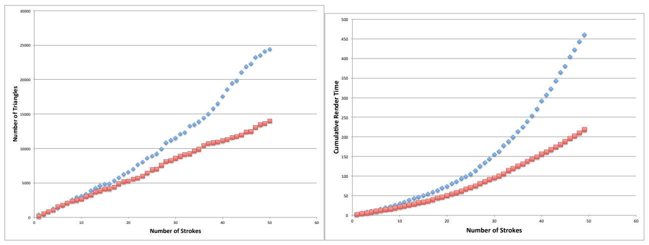


Figure 9: *An example of how TrianglePainter determines which triangles to save and which to retriangulate. Blue lines represent the boundaries of old strokes, orange lines are the boundaries of the new stroke, red lines are the triangles that will be reused, and purple lines define the area that needs 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.*

points associated with the new stroke itself. Since these geometries can become arbitrarily complicated (in pathological cases), these calculations can take an arbitrarily long. This brings us to the final point, simplification.

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.

7 Conclusion

TrianglePainter enables a user to paint like they would in a raster graphics program to create vector graphics. The core of TrianglePainter is a triangle mesh representation of the image. We have created an algorithm to convert a user's mouse motions into a triangle mesh, and then composite that mesh on the preexisting mesh containing the previous strokes.

The current implementation provides a baseline painting experience. Future work should focus on implementing a simplification algorithm to allow the complexity of the drawings to grow. Other improvements might focus on creating interesting manipulation tools, or other operations that utilize the underlying triangle mesh structure to make the artists' job easier.

Acknowledgments

Omitted for review.

References

ANDO, R., AND TSURUNO, R. 2010. Vector fluid: A vector graphics depiction of surface flow. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, ACM, New York, NY, USA, NPAR '10, 129–135.

ASENTE, P., AND CARR, N. 2013. Creating contour gradients using 3d bevels. In *Proceedings of the Symposium on Computational Aesthetics*, ACM, New York, NY, USA, CAE '13, 63–66.

BERMAN, D. F., BARTELL, J. T., AND SALESIN, D. H. 1994. Multiresolution painting and compositing. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '94, 85–90.

BREMER, P. T., PORUMBESCU, S. D., KUESTER, F., JOY, K., AND HAMANN, B. 2001. Virtual clay modeling using adaptive distance fields. In *Proceedings of the 2001 UC Davis Student Workshop on Computing, TR CSE-2001-7*, University of California, Davis, California, Davis, D. Keen, Ed.

CARR, N. A., AND HART, J. C. 2004. Painting detail. In *ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, SIGGRAPH '04, 845–852.

DIVERDI, S., KRISHNASWAMY, A., MĚCH, R., AND ITO, D. 2012. A lightweight, procedural, vector watercolor painting engine. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '12, 63–70.

DIVERDI, S., KRISHNASWAMY, A., MĚCH, R., AND ITO, D. 2013. Painting with polygons: A procedural watercolor engine. *IEEE Transactions on Visualization and Computer Graphics 19*, 5 (May), 723–735.

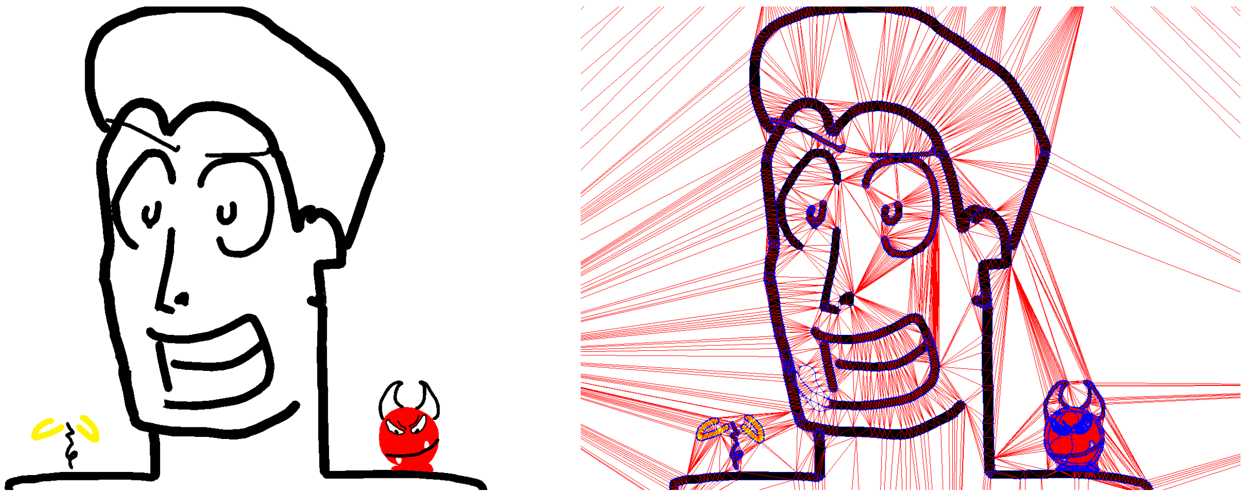


Figure 11: *A drawing by an artist that took 30 minutes. 9030 triangles.*

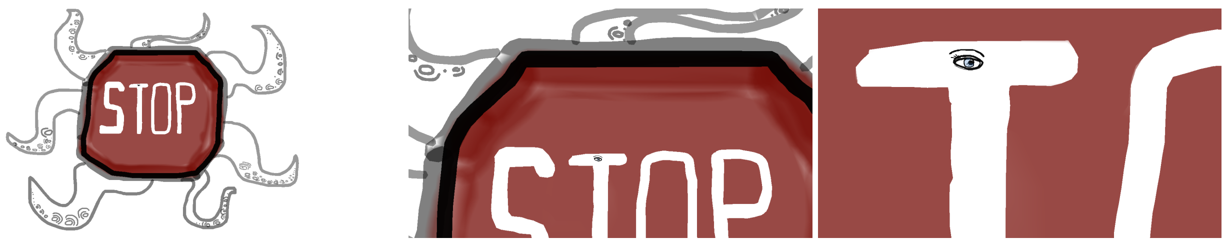


Figure 12: *Another painting by an artist, using soft and hard strokes at a variety of scales. Middle and right images are zoomed in.*

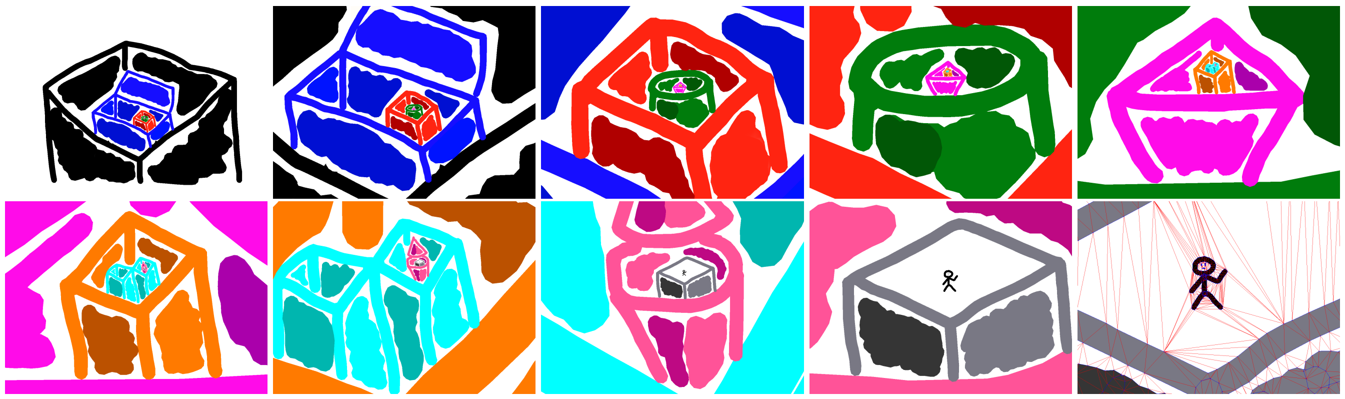


Figure 13: *A painting that shows extreme zoom levels. The final image is at zoom level 93,000.*

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 *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 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 Transactions on Visualization and Computer Graphics 18*, 11, 1858–1867.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '87, 163–169.

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

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 *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 153–160.

SHEWCHUK, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications 22*, 1–3 (May), 21–74.