A Triangle Mesh Approach to Painting

Mark D. Benjamin* Princeton Unviersity
Adam Finkelstein† Princeton University
Stephen Diverdi‡ Google

Abstract

Vector graphics have a compact representation, are easy to manipulate, and have infinite resolution. Traditionally, vector graphics have required artists to master a new skill in order to effectively utilize them. Recent work has converted raster graphics to vector graphics, allowed multi-resolution raster painting, and created new ways to represent vector graphics. However, none of these have solved the problem of letting an artist use standard and intuitive raster painting techniques to generate vector graphics. Our algorithm, TrianglePainter, converts a user's strokes into a triangle mesh vector representation. TrianglePainter lets artists focus on the art, while it handles the vectorization in the background.

CR Categories:

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 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 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 slightly 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.

*e-mail:mdbenjam@princeton.edu
†e-mail:af@cs.princeton.edu
‡e-mail:diverdi@google.com

Furthermore, this representation 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, not in pixels.

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. 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 Lvy 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 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.

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

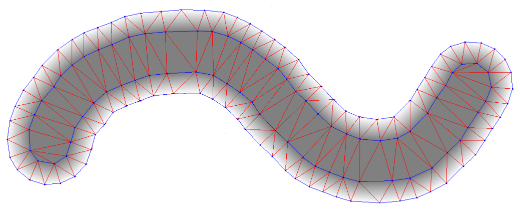


Figure 1: *An example triangulation. Blue lines are boundary edges. Black lines form the triangle mesh with the blue lines.*

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 is difficult and does not help most artists easily create.

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 use 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.

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, the triangles are not.

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 they may never cross a boundary edges.

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.

A color arc has three components: a start vector, an end vector, 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 vector to reach it and counter-clockwise from the ending vector. The point takes on the color of the arc in which the vector from the point to the centroid resides.

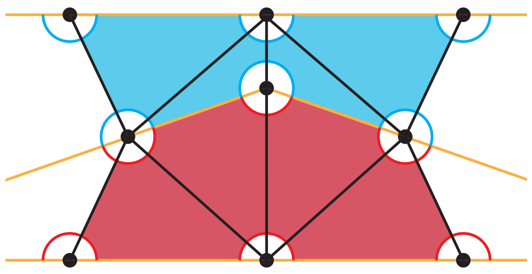


Figure 2: *An example of color arcs. The orange lines are boundary edges. Black lines form the triangle mesh with the boundary edges. Each point is given a color arc which defines its color in that range. Note that the points in the middle row have different colors on top and bottom. This gives blue triangles on top and red on bottom with a hard boundary in between.*

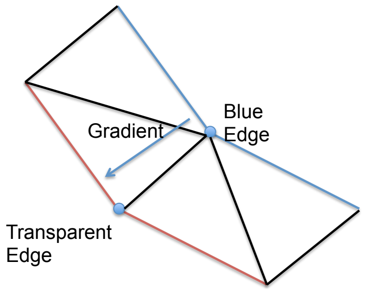


Figure 3: *Place Holder.*

The color arcs for every point 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 line up 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 solid. 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's easy to create regions containing either a solid color or linear gradient. A region will have a solid color 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.

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. Second, to obtain the contours the stroke is rasterized in black and marching squares is run 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 to perform 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 into a triangle mesh. Yet, a somewhat more interesting case is converting a soft brush 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 fed into Triangle and a mesh is returned. This mesh represents the soft stroke where no triangle crosses the boundary between the soft and hard parts 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.

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

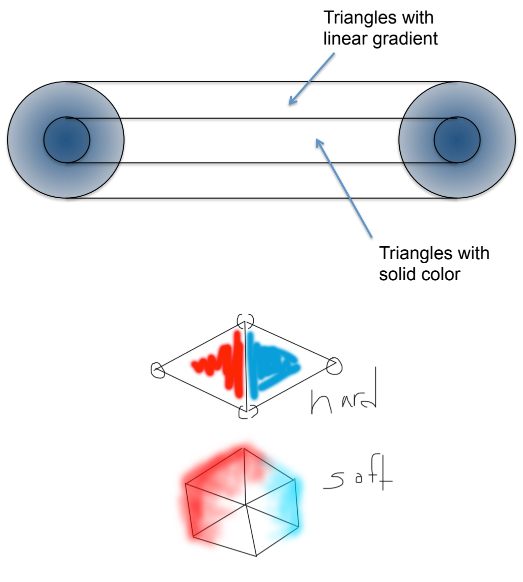


Figure 6: *Place Holder.*

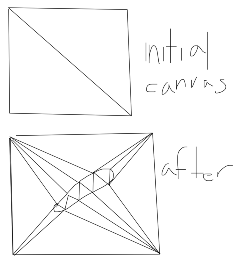


Figure 7: *Place Holder.*

boundaries pose for the algorithm, as well as a technique to reduce the number of triangles that are retriangulated on the canvas.

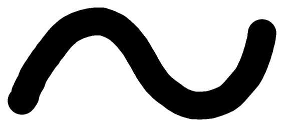
5.1 Adding Points to the Canvas

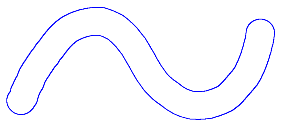
Every canvas begins with a number of points, along the edge of the screen. These points have boundaries connecting them. These points are originally part of a number of triangles which divide the canvas.

In the simplest approach the points and boundaries from the new stroke are added to the list of points and boundaries already stored. All of this data can be fed into Triangle to produce a new mesh. The new mesh contains all of the points of the old and new strokes. Since all of the boundaries are stored and fed into 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 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

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


(a) *The rasterized stroke.*


(b) *The contour of the stroke.*

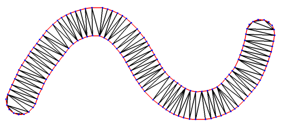
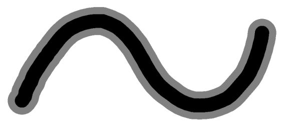
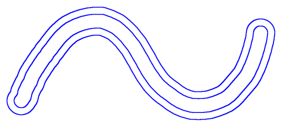

(c) *The triangulation of the contour*

Figure 4: *Triangulating a hard stroke.*


(a) *The rasterized stroke.*


(b) *The contour of the stroke.*

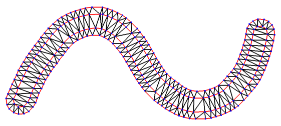

(c) *The triangulation of the contour*

Figure 5: *Triangulating a soft stroke.*

over the color of the canvas at that point. The other arc describes the 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 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. 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 overlapping boundaries 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 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 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.

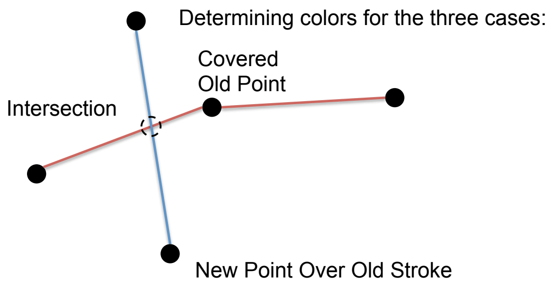

Determining colors for the three cases:
Covered
Old Point
Intersection
New Point Over Old Stroke

Figure 8: *Place Holder.*

5.4 Finding Modified Points

The above procedure has two important drawbacks. First, it processes many points that don't 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. 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 can-

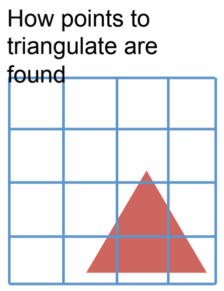


Figure 9: *Place Holder.*

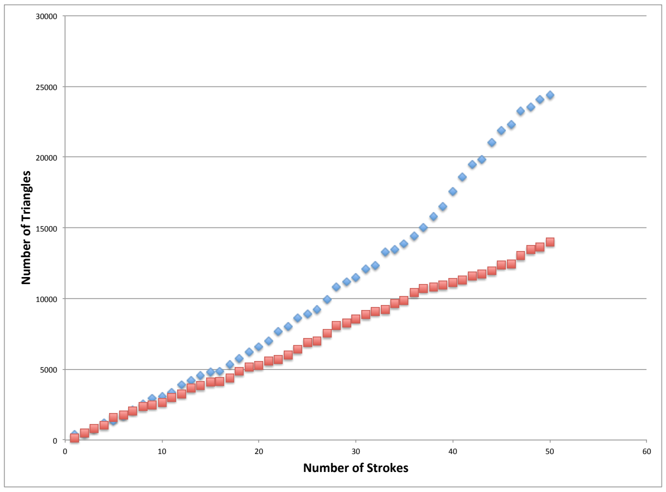


Figure 10: *The blue graph is the non-zooming case. The red graph is the zooming case.*

vas.

6 Discussion and Results

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 observations in three main areas.

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 zooming. 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.

As seen in figure 6 the zooming case grows linearly while the non-zooming appears to grow quadratically. This is expected since every new stroke has the chance to overlap all of the old strokes. This means each new stroke does not just add a constant amount of geometry. Due to intersections with previous strokes intersection points are added. This results in a linear increase in the amount of geometry. This results in a quadratic growth rate. However, in

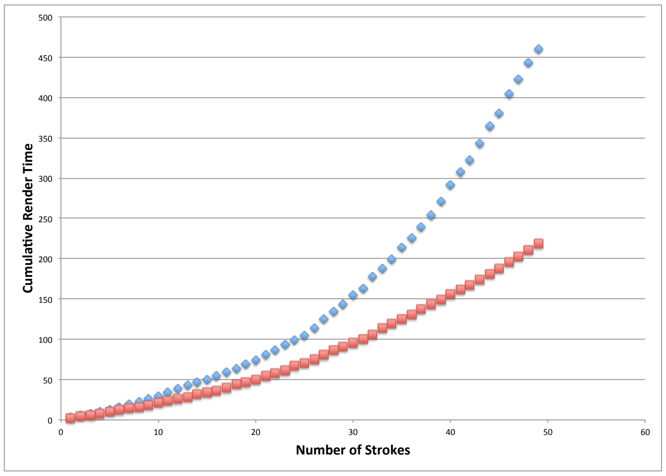


Figure 11: *The blue graph is the non-zooming case. The red graph is the zooming case.*

the zooming case each stroke rarely overlaps and therefore intersection points are rarely added. Instead the geometry increases by a constant which leads to a linear growth rate.

Figure 7 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 a lot longer for the reasons described. It's not unexpected that even the zooming case grows non-linearly since even though the geometries involved aren't getting very complicated, the geometry of the entire canvas is increasing linearly and must be considered even if the entire canvas is not retriangulated.

All this suggests that a simplification is of great importance. The case that it is most likely to help is the non-zooming case with complex geometry. When lots of strokes overlap there is usually an 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

Something we've worked hard to achieve is program stability. Certain geometries are fatal for the Triangle library. The current instability in the program is normally caused by providing the library with such difficult geometries. To improve program stability would require addressing geometries such as vertices that are too close together. This would take more processing time and is something we have yet to explore.

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 is slow. On a blank canvas strokes take on the order of 1 second to process. However, on a canvas with lots of geometry a stroke that covers that geometry can take several seconds to render. Some of this time is due to poor implementation of the algorithm. However, there are lots of calculations to do with the old points affected by the new stroke and the points associated with the new stroke itself. Since these geometries can get arbitrarily complicated, these calculations can take an arbitrarily long. This brings us to the final point, scalability.

6.4 Simplification

No matter how good the implementation, the fact that paintings can get arbitrarily complicated means there must be some way to simplify the mesh. Although we have yet to implement such a simplification algorithm, we have a couple suggestions on how it might be done. In most situations where a lot of strokes are composited on top of 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. Furthermore, even with translucent or feathered strokes, enough composited strokes may produce complex geometry that does not add much to the output. Some of this geometry can be reduced without noticeable changes in the output. Such a simplification algorithm must be implemented to ensure a painter can continue to paint without the program slowing to an unusable state.

7 Conclusion

Vector graphics are useful for representing complex images with limited information while also enabling easy editing and infinite resolution. Previous work has focused on converting raster graphics to vector graphics, unique vector representations, and multi-scale raster graphics. However, until now no system has enabled artists to create an entire painting in vector graphics using standard brush techniques.

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 allows artists to create minimal works. 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 transforms, or other operations that utilize the underlying triangle mesh structure to make the artists' job easier.

Acknowledgments

References

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.

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 LVY, 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.

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.

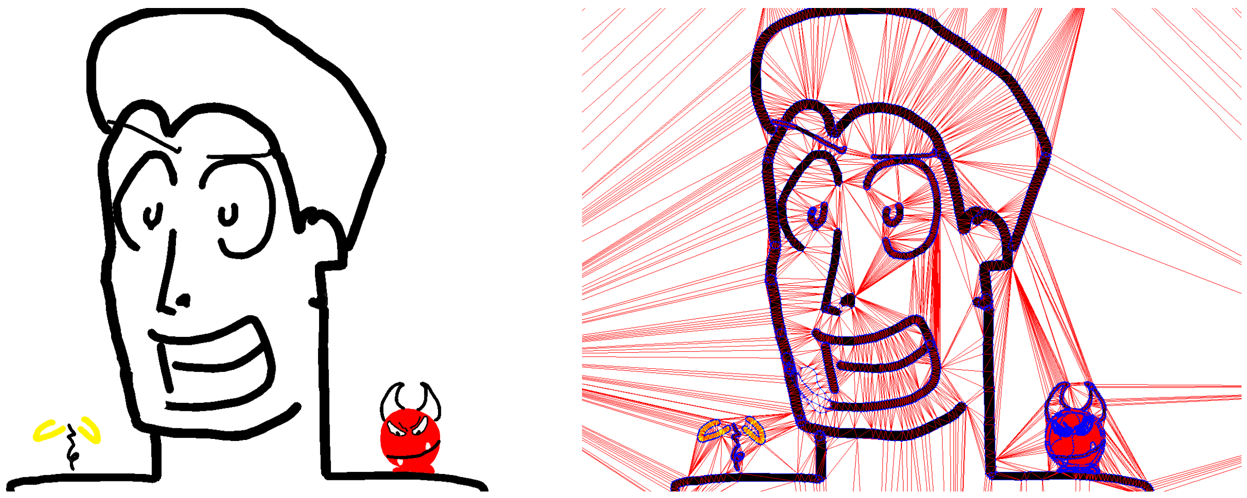


Figure 12: *Artist time 30 minutes. 9030 triangles.*