Painterly Rendering for WebGL

Andy Hanson*
Scott Todd†

Figure 1: *The Utah teapot rendered using our system.*

Abstract

TODO.

Keywords: radiosity, global illumination, constant time

1 Introduction

Non-photorealistic rendering (NPR) aims to bridge the gap between the creativity and expressivity of traditional art and the interactive and automated world of digital computer graphics. Various NPR techniques have applications in games, films, architecture, and scientific visualization. Painterly rendering is a subset of NPR that tries to create images that have a painted appearance, as if the image was created from a sequence of brush strokes on a canvas.

Painterly rendering research typically focuses on brush stroke properties, methods of user interaction, potential aesthetic styles, rendering techniques, and other methods of approximating natural artwork.

Many past painterly algorithms [Hertzmann 1998], [Lu et al. 2010] have first generated a "real" image using other methods, then attempted to draw it as a painting. This brings with it the computational cost of fully rendering the scene twice, as well as a "shower door" visual effect where 2D strokes do not move perfectly in sync with 3D geometry.

Like others [Meier 1996], we instead keep a scene of brush strokes. There is no "real" rendering but the paint, and brush strokes store their own color and calculate their own light.

*email: hansoa2@rpi.edu
†email: todds@rpi.edu

(Link to GitHub and hosted url)

1.1 Related Work

The use of particle systems to create painted images was explored in "Painterly Rendering for Animation" [Meier 1996]. Paint particles are distributed onto 3D objects and undergo the same transormation. During the rendering phase, a reference picture rendering is first created using traditional methods. Then every particle is sorted by depth, and particles are drawn back-to-front. Particles take on the color of the reference picture, or a solid color may be used per-particle and the reference picture ignored. The color is applied to a stroke texture (called the "brush image"), which is finally applied to the screen at the particle's (projected) location.

In [Hertzmann 1998], 2D input images are painted over by layers of spline brush strokes. Strokes are placed along areas of high gradient on a source image blurred proportionally to the current brush size. In this manner, progressive emphasis is placed on regions of high visual interest. An intuitive collection of parameters is chosen such that a user may easily select between a spectrum of visual styles. Parameter sets are presented for "Impressionist", "Expressionist", "Colorist Wash", and "Pointillist" styles.

[Kalnins et al. 2002] explores interactive techniques that allow artists to create non-photorealistic renderings of 3D models. Artists are given control over brush styles, paper textures, basecoats, background styles, and the position and styling of each stroke. Their system uses information supplied from one or more viewpoints to render the models at any new viewpoint while preserving the desired aesthetic.

2 Painterly Rendering System

2.1 Algorithm Overview

We represent brush strokes as particles within particle systems. (The particles will be referred to as "strokes".) Each object in the scene is represented as any number of layers; each layer has its own particle system.

Our system creates stroke meshes out of three.js geometries. Other

inputs will include layer, color, and lighting information. The system outputs an image to a WebGL canvas.

1. At load time: create a "stroke mesh" particle system for each layer on each object [Section 2.2]
2. At run time: every frame [Section 2.3]:
   (a) Render a depth image
   (b) For each object, for each layer from bottom to top, for each stroke
      i. Compute the stroke properties in the vertex shader:
         A. Compute zQuality (closeness to desired depth) and discard strokes with negative zQuality [Section 2.3.1]
         B. Calculate diffuse and specular lighting and discard strokes with specular less than the minimum [Section 2.3.2]
         C. Calculate the "drawing gradient" at the stroke's position, which provides the stroke's orientation and curvature [Section 2.3.3]
      ii. In the fragment shader, render an oriented, curved, colored version of the stroke texture according to the above parameters

2.2 Stroke Mesh Creation

2.2.1 Stroke Selection

Rather than storing models directly as stroke systems, our system loads three.js JSON object files. These are transformed into stroke systems in the client using a method similar to that of [Meier 1996]. The total number of particles to be placed on the mesh is a parameter. Then for each face on the mesh, the number of particles on that face is in proportion to its area; in other words, it is that face's fraction of the total area, times the total number of particles. Random Barycentric coordinates are selected, and position, normal, and u/v values are interpolated between the face's vertices.

Particles of a single layer are drawn in the same order every time, so if particles are generated one face at a time in this way, rendered images will show an undesirable ordering of particles. So, a stroke particles are shuffled once during the loading phase.

2.2.2 Colors

To color particles, the user may select random hue, saturation, and luminance ranges for particles to be chosen from. This produces aesthetically pleasing representations of what are essentially "solid" colored objects.

We are also exploring the use of textures.

2.3 Stroke Rendering

Both the original triangle mesh and stroke meshes are needed for rendering. We render in two passes. First, we render the original geometry into a (floating point) depth texture. Then, we may render objects in any order, but render lower stroke layers before higher layers. Some higher-layer strokes will decide to discard themselves. (As WebGL does not yet support geometry shaders, discarding a stroke is done by shrinking it to 0 size and moving it offscreen.)

We also render object borders by fattening the original and drawing only triangles which face away from the camera. This is visually pleasing, but not necessary for our method.

2.3.1 zQuality

Good results may be achieved by sorting particles by depth, but at the cost of performance. Blending particles is also unsuccessful because the number of overlapping particles is unpredictable. So, within each layer, particles are always drawn in the same order and always draw on top of each other. This leads to occlusion issues when a layer overlaps itself, shown in Figure 2.

To mitigate this effect, we render in two passes. First, we render the original geometry into a floating point depth buffer texture.

In the first iteration of our algorithm, we discarded fragments whose depth values differ from the value stored in the depth buffer at that position. This method performs a sharp cut-off around the silhouette of each object, so it conflicted with the painterly rendering focus of our project.

This depth texture is passed as a uniform into the vertex shader for paint strokes. Each stroke measures its depth compared to the expected depth, and fades out when there is a discrepancy. (A simple on/off threshold would work, but strokes just showing up coming around the edge of an object would suddenly "pop" into existence.)

There is still another problem for strokes just at the edge of an object, which do not frames share its depth buffer. A slight fattening of the original mesh ensures that all of its strokes fit within its borders. We expanded the original mesh by pushing each vertex a small distance along the normal at that vertex.

2.3.2 Lighting

Each stroke is shaded the same throughout, using the light calculated at its center. We use the Phong equations. Light is calculated per stroke (in the vertex shader), rather than per-fragment. The total diffuse and specular lighting are calculated separately (see the section on specular fade in).

2.3.3 Use of Gradient

For each stroke, we calculate a "gradient" property which strokes will be drawn perpendicular to. We would like strokes to satisfy the following attributes: they should follow the contours of objects and not extend far beyond the edges; and they should circle around centers of light. To satisfy these, we calculate the "net gradient" of the image based on both position and lighting attributes. These two parts are called the "edge gradient" and "light gradient".

We calculate their contributions with this formula:

TODO.

2.3.4 Layering

As in [Hertzmann 1998], [Meier 1996], [Lu et al. 2010], we use multiple layers of brush strokes for each object. While the layer properties can be user-controller, we intend for the base layers to contain a smaller quantity of larger brush strokes and the top layers to contain a larger quantity of smaller brush strokes. The base layers capture rough details and cover the entirety of the original mesh, while the top layers show fine details and are expected to leave gaps between each other. We found that three layers of strokes was sufficient for the scenes that we tested.

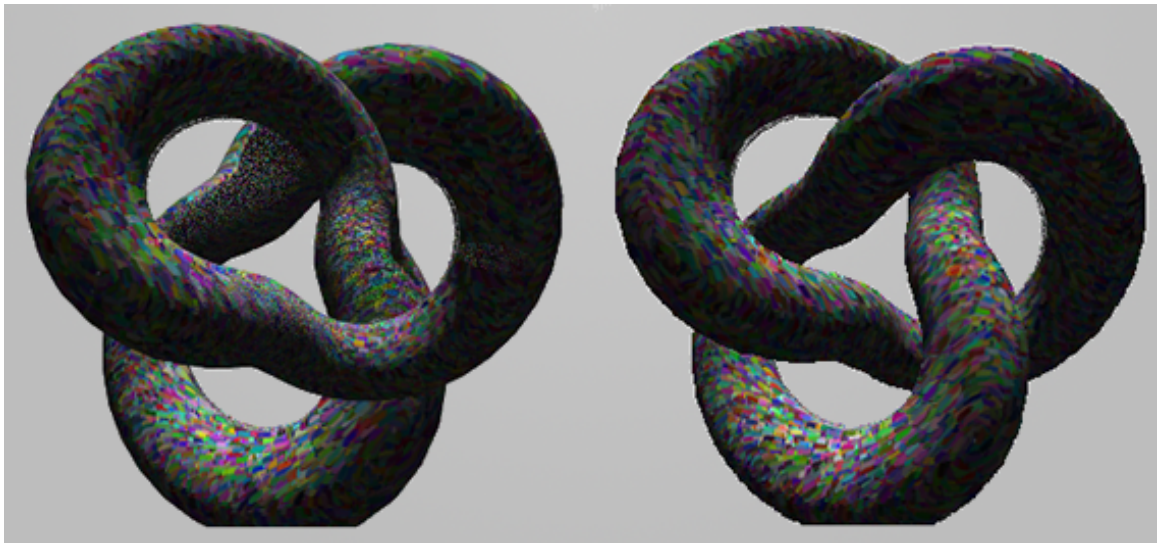

Figure 2: *The image on the right discards fragments whose depth values differ from the value stored in the depth buffer texture while the image on the left does not. Note the intersections and other artifacts on the left.*

Figure 3: *From left to right: no orientation or curvature, orientation with no curvature, orientation and exaggerated curvature (2.0).*

We support using a specular threshold and specular fade in to control the visibility of strokes within a layer. If the specular lighting contribution at a stroke does not exceed the minimum value, it is discarded. Similarly, if a stroke has low specular, we change its alpha value based on the specular fade in. This allows us to approximate the visual interest sampling in [Hertzmann 1998] and (other reference).

2.4 Parameters List

TODO.

3 Results

TODO.

4 Limitations

TODO.

5 Future Work

TODO.

6 Conclusion

TODO.

Acknowledgements

(TODO) Three.js

LaTeX: (TODO: write)

SIGGRAPH (TODO: LaTeX style)

References

HERTZMANN, A. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 453–460.

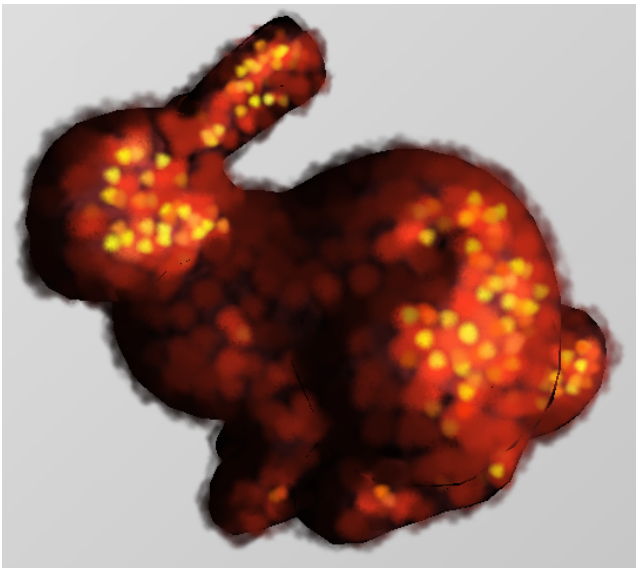

Figure 4: *The Stanford bunny rendered using three layers of brush strokes. The topmost layer uses a minimum specular cutoff of 0.95 and a specular fade-in of 0.63.*

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. Wysiwyg npr: Drawing strokes directly on 3d models. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '02, 755–762.

LU, J., SANDER, P. V., AND FINKELSTEIN, A. 2010. Interactive painterly stylization of images, videos and 3d animations. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '10, 127–134.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '96, 477–484.

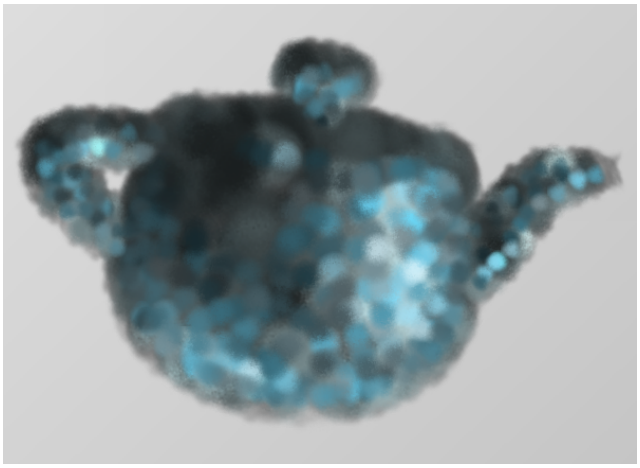

Figure 5: *The Utah teapot rendered using three layers of brush strokes. The topmost layer uses a specular fade-in of 0.03.*