NPR.js
: A JavaScript library for Interactive Non-Photorealistic Rendering in WebGL.

Michael Price
ScM Thesis Project Documentation

Background and Motivation

Non-photorealistic rendering (NPR) is the area of computer graphics that is concerned with rendering images in styles other than photorealism, such as a variety of art-based styles. Stylized rendering in general may have a number of perceived advantages, the most obvious being the potential for a diversity of visual appeal. Other motivations include visual clarity (illustrative diagrams are frequently simpler to comprehend than photographs or photorealistically rendered models), or practical asset limitation (applying a visual style other than photorealism may reduce the need for detailed textures and geometry).

We present a library of utilities for applying art-based NPR effects to 3d models in WebGL. The goals of the project are to provide a reasonable variety of customizable effects, as well as utilities for authoring and prototyping effects and assets for these effects.

Our work is motivated primarily by the visual possibilities of art-based rendering for digital art. Of particular interest is the application of NPR techniques to real-time digital art such as games, but increasingly including other forms such as installation art, interactive visualization for performance, and web-based work involving any or all of the aforementioned.

Recent developments in web standards (HTML5) allow for dynamic multimedia content using only standard HTML and JavaScript that previously required the use of browser plugins such as Adobe Flash. This has given rise to a large number of multimedia web experiments and applications. The integration of these technologies into the web standard serves a purpose further than just allowing a theoretically greater dissemination of such content to devices without common plugins. By embedding this functionality into the fabric of the web, HTML5 multimedia can interact with the DOM (Document Object Model) directly, which in turn would allow for a wide range of possibilities for the future of web content.

The HTML5 specification adds a canvas element which operates as a context for arbitrary procedural graphics. The canvas element offers contexts for 2d graphics, as well as a context for OpenGL programming. The JavaScript API for OpenGL through which this is possible is called WebGL [WGL], and is the basis for our project implementation.

The specific effects and algorithms implemented are based heavily on previous work in the field. Existing work is discussed in each implementation section.

Challenges/Limitations

The chosen platform provides both one of the most exciting aspects of the project and the most challenging. 3D graphics development of this kind in WebGL faces two main hurdles: the performance issues inherent in using an interpreted language, and the specific limitations of using WebGL (versus the more complete desktop OpenGL). These are discussed in the following sections.

JavaScript Performance

The first challenge is the performance implication of using JavaScript in the web browser as the primary platform. JavaScript implementations across browsers have made dramatic gains in performance in recent years (and this has facilitated such HTML5 developments as WebGL). However, there remains a considerable difference between what is feasible real time processing with native code and what is feasible in the browser through Javascript.

Google has recently released version 1.0 of its Native Client (NaCl) SDK, a framework for running sandboxed portable native code in the browser [NACL]. This would likely increase performance for some tasks, but has the distinct disadvantage that it runs only in the newest releases of Google's Chrome browser, and is disabled by default at this time. WebGL, on the other hand, has support in most modern browsers (with the notable exception of Internet Explorer) and an active developer community, and so remains the choice for implementation.

Instead of seeking to run native code, the main strategy used for performance is to leverage the capabilities of the GPU as much as possible, and modify existing algorithms to use the GPU for tasks previously done on the CPU. One example of such a modification can be found in [HS04], where the authors lessen the need for expensive CPU depth sorting of surface-bound particles by performing a depth test in the fragment shader. The particles whose depth does not match the corresponding value from a depth pass texture are interpreted as occluded, and their pixels are discarded. Such modifications may result in a compromise of functionality: the focus of our library is determining the best feasible WebGL approximation to the original algorithm.

OpenGL ES Limitations

The second challenge posed by WebGL development is the limitation of newer hardware capabilities. WebGL is an implementation of the OpenGL ES 2.0 API, designed for low-power embedded systems. As such, it lacks access to some new capabilities of consumer graphics hardware. Notably absent is the Geometry Shader, or the capability to generate additional geometry on the GPU based on the vertex input. This could be used to generate geometry for strokes or graftals given single sample points, which would greatly decrease the memory footprint and amount of computation on the client side. Another important feature unsupported by WebGL is instanced drawing, in which a piece of geometry is specified once and drawn an arbitrary number of times in a single draw call. As draw calls are expensive, it is critical that heavily instanced geometry be handled this way.

Furthermore, WebGL lacks support for rendering to multiple render targets. This requires reference passes for deferred rendering to be drawn in separate draw calls, using different shaders. Ultimately this is not as critical a feature, but where multiple deferred passes are used, performance suffers from its absence. Where possible, information that would otherwise be rendered to reference passes is instead computed directly in the shaders that would be reading it.

Implementation

The following sections discuss our implementation. In the next section, a technique of creating instanced geometry that is common to most of our implemented algorithms is discussed. In the sections following, each implemented algorithm is discussed in some detail.

Particle-based instanced geometry techniques

A common technique across many of the implemented algorithms is the usage of uniformly sampled random points and normals across the surface of a mesh to position pieces of stylized geometry. These pieces of geometry may be textured quads (as in the case of painterly rendering, or particle-based lapped texture splats for hatching) or arbitrary 2d shapes (as in the graftal case). Regardless of the style-specific differences, they share the property of being instanced across the surface of the model and benefit from a common implementation.

There are a number of possible approaches to implementing instanced geometry in general. On newer hardware that allows for the usage of a geometry shader stage, one might specify single sample points in a Vertex Buffer Object. These could each be transformed into the desired geometry in hardware, minimizing the amount of memory required to specify the stylized details. One might also use another feature of modern graphics hardware, instanced drawing, wherein a geometry buffer specified once may be repeated a given number of times in a single draw call. Each instance is given a separate index so that it may be transformed separately. Unfortunately, neither of these features is exposed in OpenGL ES, and both are therefore unavailable for use in WebGL.

A possibility for the billboarded quad case involves the use of hardware point sprites. Specified as a separate drawing mode, screen-aligned hardware point sprites may be given a specified size in the vertex shader program, and textured in the fragment shader. This allows each billboard to be specified as a single point for minimal memory consumption. The most obvious limitation of this approach is that the hardware point sprites do not directly support rotation or non-uniform scaling. Both of these limitations can, however, be worked around in the fragment shader, as both scaling and rotation transformations can be applied before the texture lookup in the fragment shader at a small computational cost. Such rotation requires padding within the point sprite to accommodate the longest diagonal of the rendered area. In practice, however, the maximum point size for hardware point sprites is capped differently across devices. On our main machine used for development, a laptop with a high-end graphics card (Nvidia GeForce

GTX 460M), this maximum value is 64 pixels. Particularly with padding to allow for rotation, this value is small enough to limit practical use.

Instead, we choose a method that is suboptimal with regard to memory usage, but is otherwise performant and of high quality. We copy the geometry once for each instance to be drawn and store the concatenated result in a Vertex Buffer. As WebGL does not support drawing with quad primitives, we have the further memory inefficiency that each billboard must be specified as two triangles, a total of six vertices per billboard instance. As the purpose of this large buffer is to allow instanced drawing, each instance must also contain a sequential index, and this index must be passed with each vertex of the instance. All per-instance attributes other than this index are stored in a floating-point texture, making use of the OES_TEXTURE_FLOAT extension which is currently supported on many WebGL implementations (such as Google Chrome and Mozilla Firefox). To minimize the size of the geometry buffer, the geometry is specified as 2d points in billboard space, with the instance index stored as the z-component. The object space position (as well as the normal) for each instance is then stored in the attribute texture. This attribute-texture technique for instanced drawing is crucial to our implementation, as it allows for an arbitrary number of attributes to be stored and referenced in the vertex shader. If each attribute were stored as a geometry buffer, it would have had to be needlessly repeated for each vertex. Similarly, were each attribute encoded into a separate texture, we would quickly exceed the hardware limit for vertex texture units (4, on our primary development machine). As such, the library includes a utility function for encoding an arbitrary list of three or four dimensional vectors into a power-of-two sized texture of three or four floating point channels respectively.

Some other benefits of this approach not available in hardware point sprites are that it extends nicely to arbitrary graftal geometry, and that the geometry is no longer required to be screen-aligned. Additionally, because the buffer geometry is specified only in billboard space, the same buffers may be used to apply the same effect to different models. Ideally, future implementations of WebGL will feature support for hardware instanced drawing, which will drastically alleviate the memory usage of the technique in general.

Implemented Algorithms

The following sections provide information on the specific effects implemented. Each of the four effects is heavily based on an existing published implementation. Relatively high-level overviews are given, with more detail provided as it relates to where our system diverges from the reference implementations.

Painterly Rendering

Our implementation of painterly rendering is based on [BM96]. The technique ultimately results in an image made up of similarly shaped brush strokes that animate smoothly with a changing viewpoint. A general explanation of the algorithm is as follows.

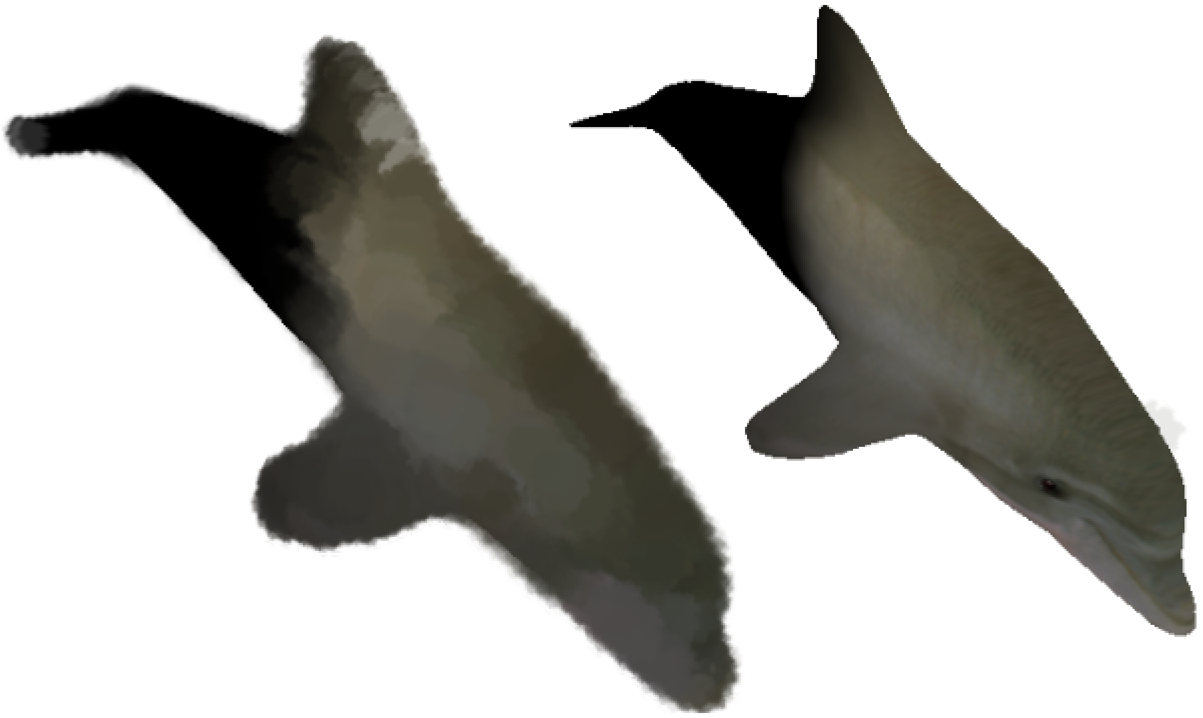
The scene is rendered once with multiple deferred rendering passes. In particular, a normal pass is generated, along with a color pass, which functions like a target image for the render. The color pass includes lighting and texture. It is also possible to specify passes for billboard size, opacity, or conceivably other attributes. From there, the geometry of the scene is rendered using instanced billboards that are statically positioned relative to the model. Each billboard has an alpha channel defined by a brush image, The color for the billboard is sampled from the target color pass, the entire billboard is colored according to the color pass color behind its center in screen space. Similarly, the orientation of the billboard (rotation within the screen plane) is defined by the normal pass. The billboards are then rendered with alpha blending and potentially depth sorting to create the final image. In the case where the billboards are not dense enough to cover the models entirely, a version of the color pass may be composited behind the brush strokes to prevent the background color from showing through.

Our implementation uses only the target color pass and a depth pass as deferred rendering passes. As the per-stroke texture reads are more efficiently done in the vertex shader once per billboard instance, rather than once per instance-pixel in the fragment shader, it is imperative that we remain below the hardware limit of four vertex shader texture units. The two passes constitute two of these textures, and the instance attribute texture for the billboards constitutes a third. The fourth unit is left free for further customization of any of the properties based on values that cannot be computed in the vertex shader, because they depend on either rendered images or values otherwise determined in screen space (i.e. a user painting on attributes directly). Because the normal information for each billboard is stored in the attribute texture directly, we have no need for a separate normal pass. Likewise, billboard size is currently determined by a uniform value and attenuated in hardware as any other geometry.

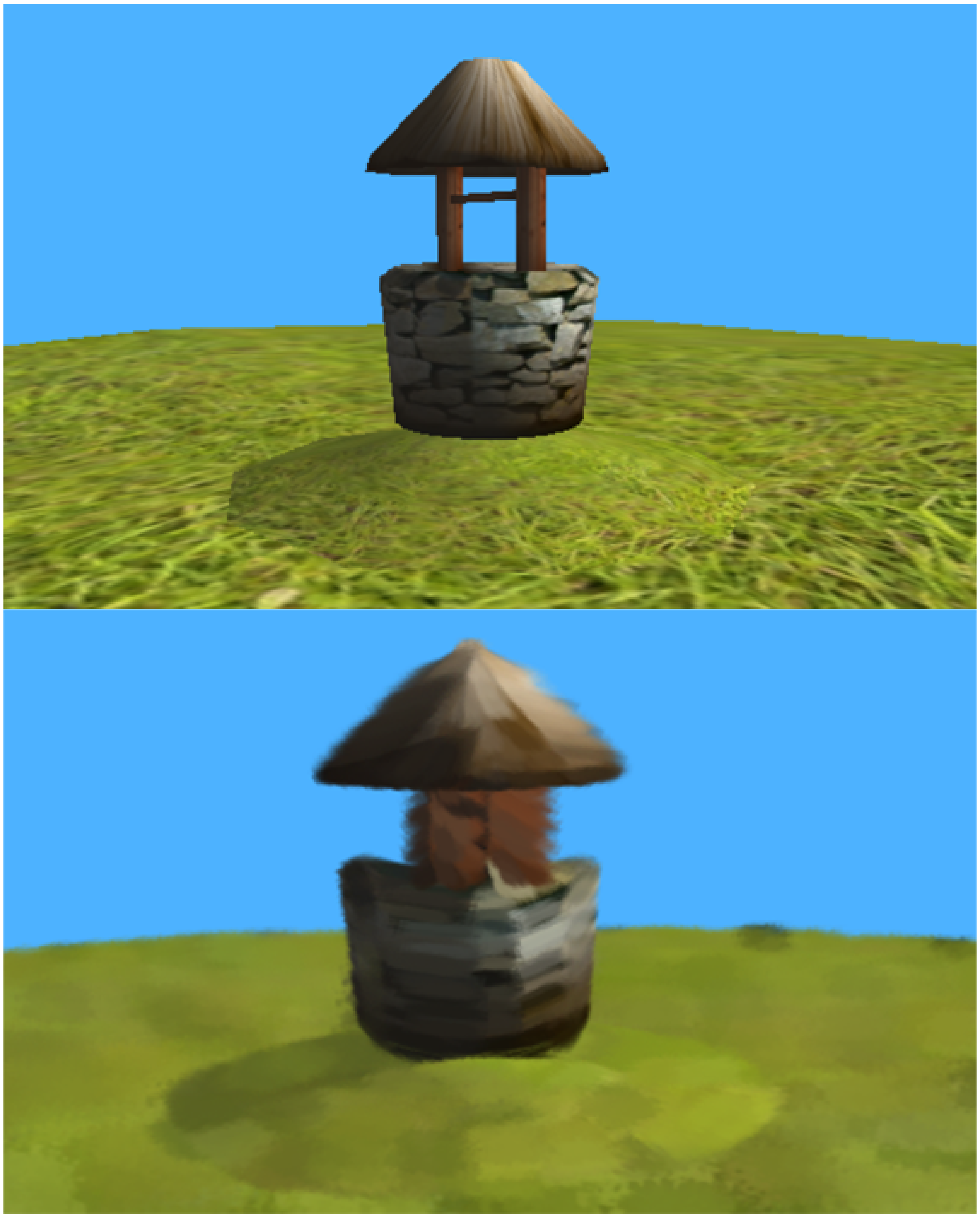
We draw with alpha blending using a premultiplied alpha channel for the brush texture, depth testing disabled, and no depth sorting (since it is quite detrimental to performance). With partially transparent brush strokes, this gives reasonable results. Because the draw order of the particles remains constant, disabled depth testing works to the benefit of temporal coherence: a particle drawn on top of another will always be drawn on top, eliminating popping artifacts. However, the lack of depth testing also presents problems with occluded surfaces, namely that strokes on the sides of objects facing away from the viewport will still be rendered. To prevent this, we disable drawing of back faces by comparing the normal of the point sample to the look vector of the camera, and prevent drawing at some minimum threshold. The tuning of this threshold is important. If the value too low, and a vector at the threshold faces away from the camera, the purpose of the test is somewhat defeated, as the billboards facing somewhat away will possibly be rendered on top. It too high (facing forward, or perpendicular), the image suffers from popping artifacts where billboards meet the threshold are suddenly removed. Ultimately the only way to eliminate these popping artifacts is to ensure a smooth transition from the visible to invisible state. This may be done either by shrinking the billboards down to a point, or by fading out the alpha channel. Both approaches gave usable results, but ultimately the alpha fading approach was more applicable to a general case because it appeared more seamless. Thus the alpha blending is enabled by default in the library's painterly billboard shader. The thresholds at which fading begins and full disappearance is attained are adjustable via uniform parameters. These will likely need to be adjusted on a per-object basis.

There is yet another occlusion problem that occurs due to lack of depth testing that is not solved by the simple facing ratio test. For all but the simplest meshes, and for any configuration of multiple meshes, it is possible for a front facing surface to be occluded by another front facing surface. The result is an undesirable artifact, where the color of all strokes looks correct (as they are sampled in screen space from the color pass) but upon movement it is apparent that there are two separate groups of strokes. If the orientation/normal pass is rendered as a separate pass and sampled, then the orientation of the strokes will conform to that of the visible object, and so the only artifact will be one of motion (which could conceivably be used for artistic effect, but is not desirable in the general case). In our implementation, with normals stored per-particle in object space, orientation of the offending particles conforms to that of the occluded object. To mitigate the effect for scenes with multiple meshes, a back-to-front drawing of primitives is beneficial, though not sufficient. In general, we follow the approach described in [HS04]. Particle depths are computed from clip space position, and this value is compared to the value stored in the depth buffer from the depth reference pass. If the geometrically determined depth for a billboard is greater than the depth stored in the depth pass buffer at the point corresponding to the screen-space position of the center of the billboard, then the billboard is treated as occluded and not rendered.

The largest performance bottleneck for this technique is the fill rate of the graphics processor. Because we are drawing with alpha blending and no depth testing, there is a great deal of overdraw. Each stroke is drawn to the screen in its entirety each frame, which degrades performance significantly if a large number of strokes are scaled to be large enough. In our experience, however, a large number of strokes can be drawn at a sufficient size with acceptable real time performance.



A dolphin with and without a painterly effect. Dolphin model and texture from http://thefree3dmodels.com


A grassy well scene rendered without and with a painterly effect.   Note that the texture maps provide color variation but not texture or detail.

Hatching

Hatching is a technique by which levels of tone are indicated by the density of close parallel lines, but we have generalized it to any technique in which tone is indicated by the density of layered marks. The implementation of the hatching effect is inspired by the implementation in [PH01]. Praun divides the problem into the synthesis of Tonal Art Map (TAM) textures and the creation of surface parameterizations suitable for their application. The latter will be discussed first, and is approached using the technique of 'Lapped Textures' [PF00].

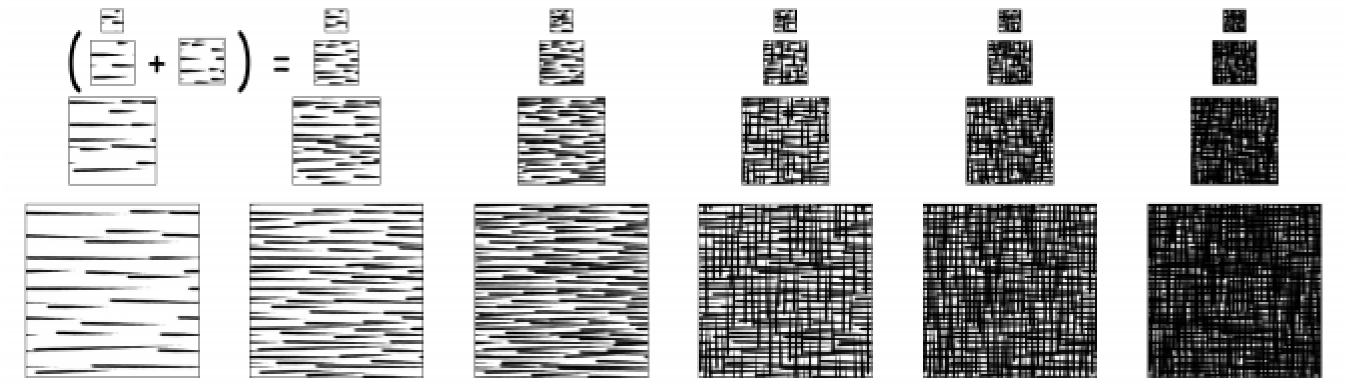
The lapped texture approach seeks to take irregular blob-shaped portions of a tileable and repeatable texture and paste them over the mesh until the mesh is entirely covered. The paper employs a sophisticated technique to optimize the patch placements and distortions. First, each triangle in the model is given a desired texture orientation that is computed from the local curvature of the mesh. Next, each lapped patch of triangles (enough geometry to apply one splatted blob) is 'grown' breadth-first from a central triangle. This is completed until enough patches are made to cover the model. Finally, the texture parameterization of each lapped patch is adjusted. The patch is rotated to align with the central triangle's desired texture orientation, and then the optimal texture coordinates for each point are formulated as a solution to least-squares optimization problem. With the error between the actual texture orientation and each triangle's desired orientation minimized, the resulting parameterization nicely curves the strokes along the curvature of the mesh. The resulting parameterization may be either converted to a texture atlas, or the patches may be drawn directly (necessitating a reformulation of the geometry as well as the drawing strategy).

Our implementation forgoes the more sophisticated techniques at a cost of some image quality. Our primary implementation simply assumes acceptable texture coordinates on the model, which provides reasonable results for models with largely continuous texture coordinates. Certainly nothing prevents a texture atlas created from an offline preprocessing lapped texture step from being used with our implementation, but utilities are not provided to do so.

We also implemented a secondary approach that takes the spirit of lapped textures, but utilizes the same particle-based approach as the painterly rendering technique. The main functionality that is missing from this approach is the texture coordinate optimization. At best, this approach gives results similar to lapped texture patches that are rotated to align with the center triangle but not warped to account for the orientations of each triangle in the patch. However, this is still a potential improvement for models with unsuitable texture coordinates, as it makes no assumptions about surface parameterization. The technique takes minimal preprocessing (akin to the painterly rendering technique), and works as follows:

Static object space position and normal samples are taken for the target model, and an attribute texture is made encoding this information. Geometry buffers identical to those made in the painterly rendering technique are made. Unlike for the painterly rendering technique, however, here no target color pass is used. Instead, lighting is computed directly in the shader, which currently uses a single directional light. A depth pass is still used to resolve the occlusion culling issues that arose for painterly rendering. The primary difference between the vertex shader for this hatching technique and for painterly rendering is that whereas in painterly rendering, the billboards were rotated in the screen plane to align with the screen projection of the normal (or its 90 degree rotation), in hatching we rotate the quads so that their normal is aligned with the object space normal in 3d space. That is to say, we rotate each quad so that it is flush with the surface of the mesh at its center. This is done without sending any additional information to the GPU; a rotation matrix is constructed directly from the object space surface normal in the attribute texture. The result of this technique is a crude approximation of lapped textures: the model is covered with splatted patches that approximate the shape of the underlying surface.

One limitation of this technique is that because each patch is planar, the tesselation of the result is dependent of the number and size of billboards used. A larger number of smaller billboards will use more memory but may be necessary to cover the model with an acceptable tessellation. Furthermore, areas of high curvature may be poorly approximated by the planar patches. What we may lose in quality, we make up for somewhat in convenience and portability: the technique can be applied to any mesh with vertices and normals regardless of texture parameterization, with minimal preprocessing and only a TAM texture and patch 'splat alpha' texture.



An example of a TAM. Figure from [PH01].

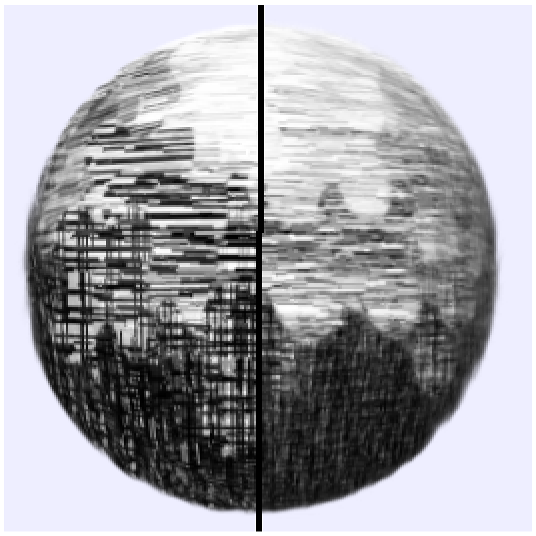
The second component of the algorithm is the Tonal Art Map texture synthesis. A TAM texture stores tileable swatches of stroke textures at a number of different tone levels. At render time, these tone textures are interpolated based on the value of the lighting at a point. Praun et al make the key observation that a given tone should be represented as a fixed density of marks across different screen space scales. Using a regular mipmapped texture of strokes will work at one optimal resolution, but as the textured object becomes smaller in screen space (due to changes in distance, scale, or facing ratio), the texture will blur out to an even gray. To combat this, an important property of the TAM is that each tone level contains custom mipmap levels consisting of fewer strokes. To maximize coherence, the following algorithm is used to synthesize the TAM texture for a given number of tone levels:

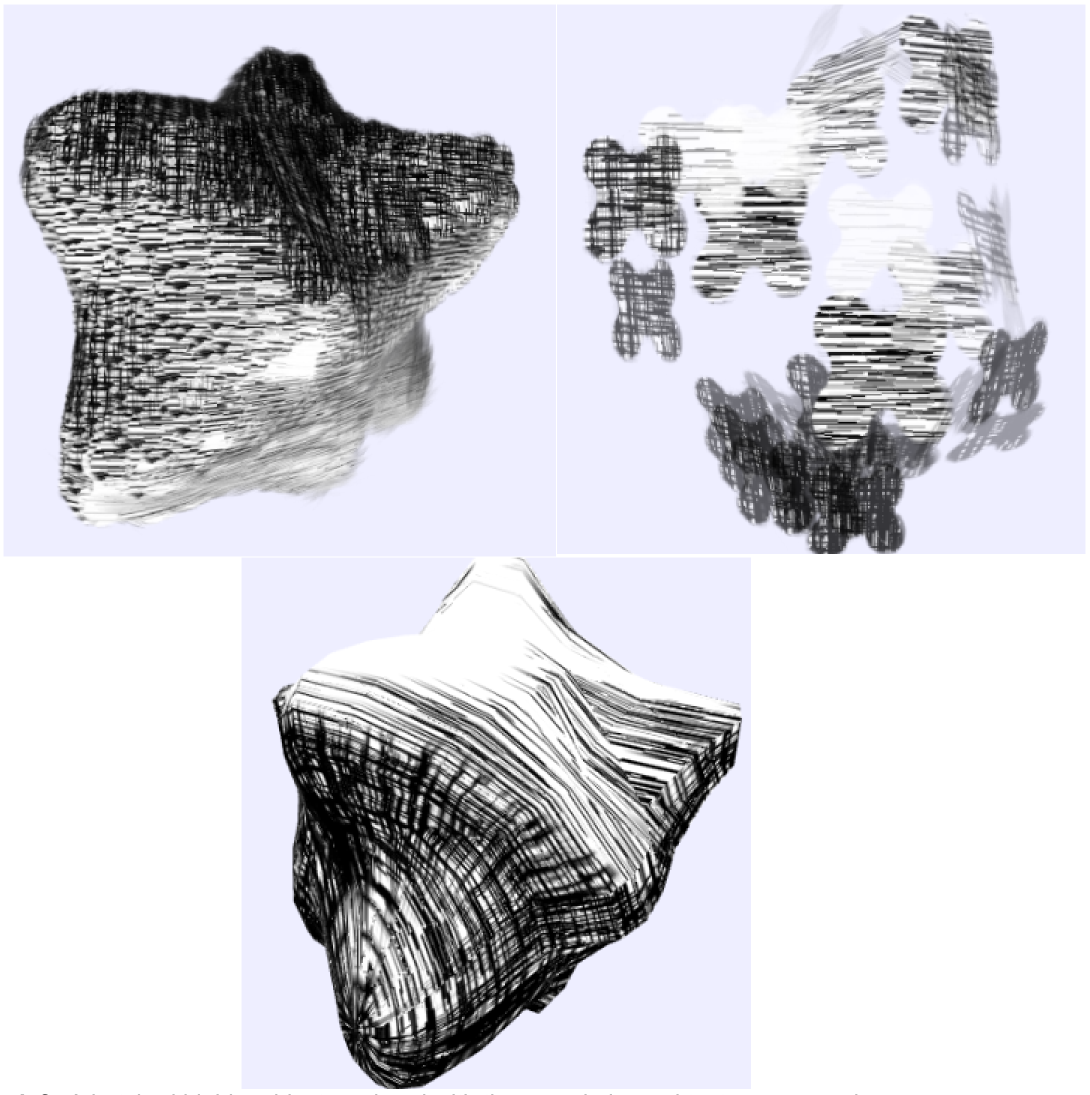
We begin with the lowest resolution level of the lightest tone. Strokes are added to the image and recorded until the average tone (computed as an arithmetic mean) of all pixels in the image passes the required threshold. Then we move to the next higher resolution, redrawing the (parametrically positioned) existing strokes in their corresponding positions and sizes on the larger texture. We then continue adding strokes until average tone of this patch is similarly over the required threshold. This continues until the highest resolution image is complete.

Next, we begin the next darker tone level, starting from where we left off by copying each tone image over to a new image. The process is then repeated for a darker tone threshold, and again until all of the resolutions of all tones have been completed.

Our implementation uses six TAM tone levels with four size levels (the smaller mipmap levels are the result of standard mipmapping from the lowest resolution level). We store the six single-channel tone textures in different channels of two different RGB textures, which are both bound before drawing.

Our library provides some utilities to aid creation of TAM textures. It supports loading a power-of-two texture given multiple mipmap images explicitly. In addition, the general 2d paint utilities built on top of the HTML5 Canvas API for the 2d graphics context contain representations for redrawable and transformable strokes. To demonstrate these utilities, one of the sample applications includes a system similar to that described in [WP02], in which TAM textures are created as a result of an interactive process. First, a user creates some example strokes in a drawing area. Afterwards, the system will choose from the example stroke list at random to synthesize the TAM texture. When it is complete, it packs the texture into the RGB channels of an image and displays it for saving.


A hatched sphere with and without the custom mipmaps.



Top left: A hatched blobby object rendered with the pseudo lapped texture approach.
Top right: The same configuration with a reduced number of point samples shows insufficient coverage.
Bottom: Rendered with the standard texture mapped approach. Where continuous texture coordinates are available, the result from this method is typically superior.

Graftals

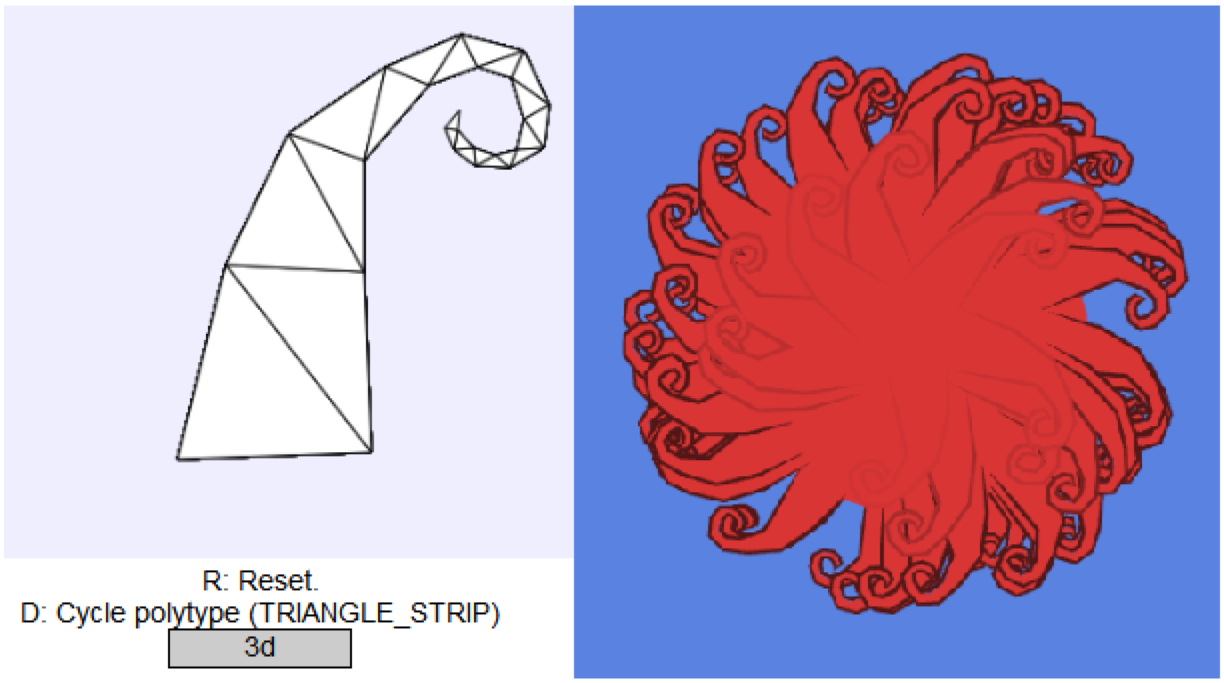
The graftal technique used in the library is based on the technique described in [KM99] to create cartoon rendering of fur, grass, and similar surfaces in a style inspired by the illustrations of Theodor Geisel (Dr. Seuss). The basic idea is to augment simple models with ‘fins’ of graftals: billboarded geometry of an arbitrary 2d shape with an arbitrary rotation in the screen plane that is placed on the surface of the mesh. When rendering exclusively with solid colors (or conceivably toon shading), the graftal fins appear as natural extensions of the underlying mesh, as bunches of fur, leaves, cloud-like tufts, or other protruding surface detail. The graftal protrusions are in general rendered as a solid color with a solid color outline. The authors use a system where the graftal fins may be fully outlined, partially outlined, or not outlined at all, mapping this variation to an attribute such as facing ratio.

Our implementation is a limited approach, omitting some sophisticated but costly parts of the pipeline such as screen-space density control and dynamic level of detail (though nothing precludes a simple LOD scheme from being implemented by combining different sets of model samples based on camera distance). The graftal technique is again a particle-based approach, and as such makes further use of our existing infrastructure for random mesh samples and attribute textures. It is similar to painterly rendering in that the particle geometry is aligned to the screen and oriented relative to the screen-space projection of the surface normal at the sampled point. It differs, however, in several ways. First, the graftal geometry is not a simple two-triangle quadrilateral, but instead an arbitrary two dimensional mesh. Second, the graftals as implemented require no texturing and no transparency. This works to our advantage in that it allows us to use hardware depth testing effectively to cull occluded graftals. As a result, we do not need a depth pass or any other deferred rendering pass to render. This is convenient, but does occasionally result in popping artifacts when one graftal moves closer to the camera than another.

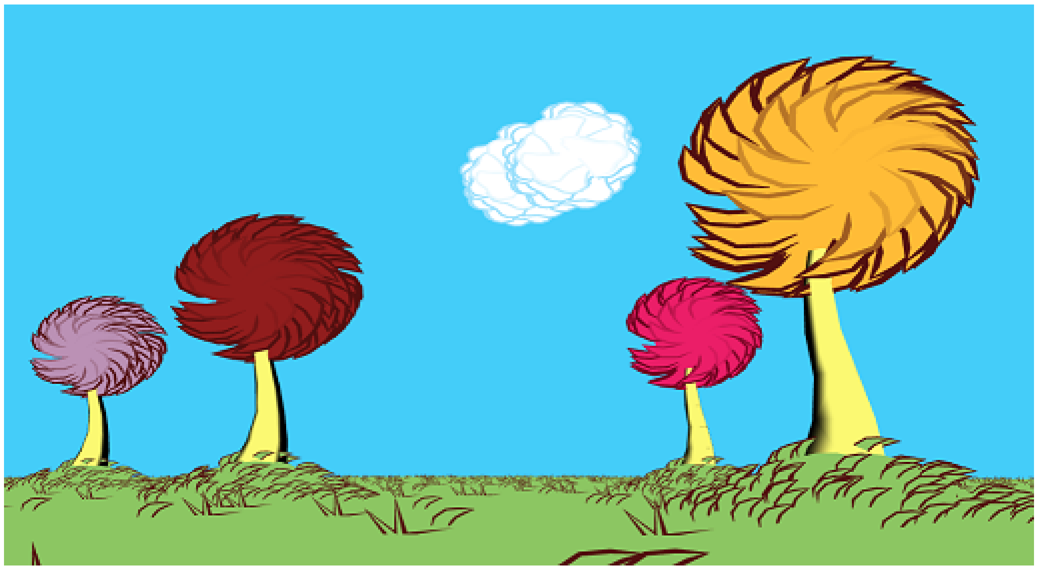
Our implementation has a few limitations in flexibility. First, we adhere to the convention that graftals near the contours of models (a low facing ratio) are meant to be drawn, and that graftals are oriented according to the surface normal. One could imagine a system with a wider range of options for graftal visibility and orientation, for example [KM99] allows for graftals that are always oriented downwards, to give the appearance of drooping fur. Furthermore, graftals that are not being rendered are simply faded to an opaque ‘fill’ color that is meant to be identical to the solid fill color of the underlying geometry. This works well for the supported case where the underlying geometry is a solid color, but would need to be adapted for any other case, such as toon shading. Finally, graftal outlines are not as dynamic as in the [KM99] implementation. Their implementation utilizes immediate mode drawing and so may easily decide to draw partial or complete outlines on a per-graftal basis. Because our implementation is dependent on storing static buffers of graftal geometry, such adjustments are less trivial and must be determined in the vertex shader. Each graftal is simply always drawn as fully outlined. This could be altered by using a different storage format for the edges that would allow portions of edges to be dynamically collapsed into degenerate geometry in the vertex shader, with ‘bottom’ and ‘top’ edges annotated dynamically using an additional vertex attribute.

The outline buffers themselves are stored separately from the graftal bodies. These are drawn in a separate OpenGL call from the graftal bodies, using the same shader (the fade out color is kept the same and the main color set to the desired color for the outlines). Ultimately this approach is quite memory intensive, as the outline buffers are generally larger than those created for graftal bodies, which, in turn, are generally larger than the buffers used for the techniques requiring only square billboards. An implementation of instanced drawing for the WebGL specification would relieve this memory usage drastically.

In addition to the implementation of the basic graftal algorithm, the library provides utilities for the interactive creation of graftals. A simple 2d polygon class is provided, where points may be dynamically added and specified as triangles, triangle strips, or triangle fans. The polygons provide utilities for normalization and uploading to the GPU as instanced geometry buffers (in fact, the instanced billboard utilities for the other techniques rely on this framework). Furthermore, a simple JSON (JavaScript Object Notation) format is provided for the reuse and serialization of graftal styles. The format includes the list of points of the base polygon for the graftal (along with the primitive type), as well as parameters for color, scale, outline thickness, static rotation, number of instances, and position offsets for tweaking the appearance of graftals for individual models, etc.



A portion of our graftal editing interface. The polygon on the left is defined by clicking points in a triangle strip pattern, and is automatically instanced and uploaded to the GPU for drawing as graftals on the right.



Our WebGL rendition of the Truffula scene from [KM99].

Contour Edges

The final NPR effect supported by the library is that of stylized edge drawing. The problem of extracting edges for stylized rendering is one that has been approached from both geometric and image-based directions. Image-based approaches generally extract feature edges from a framebuffer or combination of framebuffers (such as depth and normal passes), perhaps by using a convolution kernel such as the Sobel. While this approach yields good results (and is in fact implemented as a post process pass in our library), it provides limited opportunity for parametric stylization. One cannot simply apply an oriented stroke texture to a framebuffer of detected edges. Thus, geometric object space approaches are useful for analytically identifying edges which can be used for a wider variety of effects. Object space techniques are, however, typically more computationally intensive.

[NM00] introduces an interesting hybrid screen space and object space approach where object space edges are found on the model and are later linked into coherent chains based on their screen space position. This allows for rendering the model as a smaller number of long, flowing strokes. This provides excellent results, but requires a good deal of dynamic processing on the CPU.

For our library, it is imperative that we provide a solution that runs entirely on the GPU, and is compatible with OpenGL ES functionality. Fortunately, [MH04] provide such an algorithm, and their approach inherently makes similar tradeoffs to the other techniques implemented by the library. In particular, some degree of quality, as well as a large amount of GPU memory, is sacrificed in order to render in real time on the GPU. This approach builds an 'edge mesh' data structure that contains information for every edge in the model, and sends it to the GPU once. The basic principle is similar to that used to render graftals near the edges of models: based on a decision made by the analysis of normals (facing ratio for the graftals), a decision of whether or not to render a given edge is made. The edge mesh information contains the vertex positions for each edge, as well as the vertices of the two adjoining triangles (assuming each edge is shared by no more than two triangles). From this information, it is simple to construct the face normals for the two triangles by taking the cross product of two edge vectors from the same triangle. This is performed in the vertex shader, and the resulting normals are used to identify whether the given edge should be rendered. The original implementation identifies a number of renderable edge scenarios (concave and convex creases, contours, border edges) but our system focuses on contour edges, with the other cases being left as simple extensions. A contour edge is detected when the two normals of the adjacent triangles point in opposite directions relative to the view vector, i.e. when the signs of the z components of the view space normals differ. While face normals in object space could be stored directly in the edge mesh rather than computed in the vertex shader, storing vertices theoretically allows for accurate normals to be computed after mesh distortions such as skinned animation.

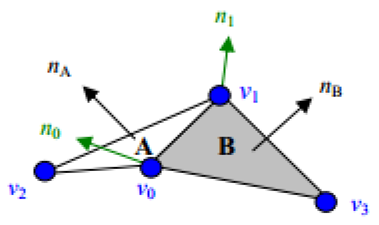
The full edge mesh of four vertices per edge is drawn in a single draw call. Where a renderable edge is detected, the drawn vertices are positioned to make a quadrilateral of arbitrary thickness in screen space. Each vertex is placed according to its index within the edge itself, the first two vertices are placed on the vertices of the underlying geometry, and the second two are displaced the desired thickness away in screen space. The paper implementation displaces the outer vertices in a direction perpendicular to the edge in screen space, filling the holes left between adjacent convex edges with triangular caps in a separate drawing pass, using different indices into the same edge geometry buffer as well as a modified placement strategy for edge caps in the vertex shader.

One of the main advantages of using this geometric technique is to allow for the arbitrary stylization of strokes via texture mapping, so texture parameterization is important. As the entire edge width generally maps to the entire texture height, the parameterization problem is reduced to finding the single horizontal texture coordinate. [MH04] presents two texture parameterizations, one in object space and one in screen space. The former favors temporal coherence over spatial coherence, and the latter the reverse.

The object space parameterization simply assigns to each edge a texture coordinate range proportional to the length of the edge. To mitigate obvious tiling, each edge is optionally given a static random value to offset into the texture space. This approach is dependent on mesh resolution, and works best for relatively low resolution meshes, with the short edges of high resolution meshes degenerating into noise.

The screen space parameterization computes a coordinate by comparing the screen space edge points with the screen space projection of the object space origin. For edges that are more vertical, the coordinate is a line-width and direction dependent constant factor, times the vertical distance of the point from the screen space origin. For horizontal edges, the horizontal distance is used.

Worth noting is the implementation by [HV09], an improvement on the technique that uses the Geometry Shader stage to generate the outline geometry dynamically. This approach greatly reduces preprocessing and memory consumption. They also make a further modification to the screen space texture parameterization. Instead of using a horizontal or vertical distance from the screen space origin, they use the angular distance in polar coordinates. This gives good results for edges that do not radiate outward from the origin To accommodate this degenerate case, the distance from the screen space origin is used as an additional scaling factor. The influence of this secondary scaling factor is adjustable via a uniform parameter. While we cannot use the improved Geometry Shader implementation of the pipeline, we do use this polar coordinate schema for texture parameterization.



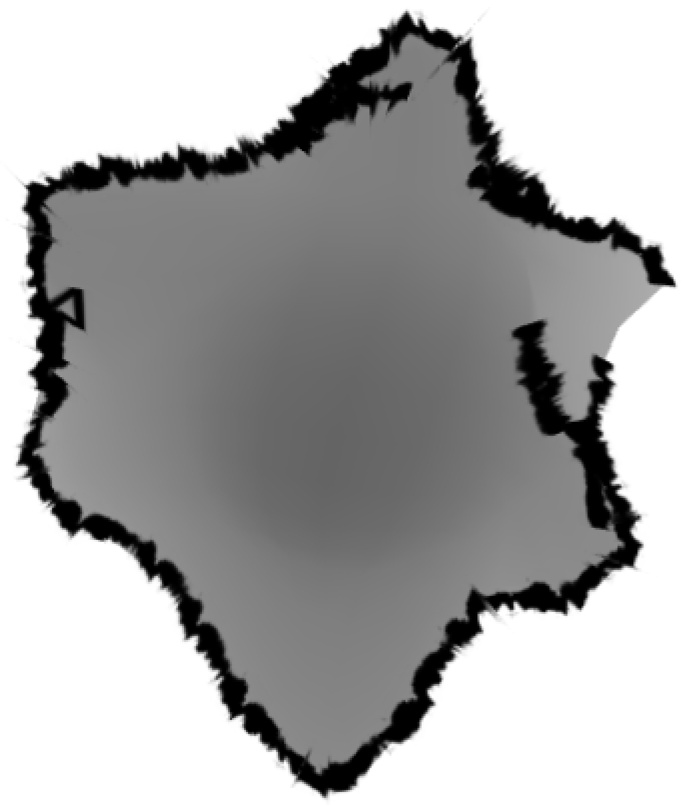
The values supplied for the edge from

v0 to v1. Figure from [MH04].

Our implementation makes a modest memory improvement over the original by using the library's attribute textures to full effect. The original constructs an 'edge mesh' entirely using vertex buffer data with many attributes, which results in repeated information whenever per-edge information is needed by each vertex in the edge mesh. Specifically, each vertex is supplied with a tuple <v0, v1, v2, v3, n0, n1, r, i> where v0 and v1 are the positions that define the actual edge, v2 and v3 are the remaining vertices of the triangles sharing this edge (if v2 = v3, there is only one such triangle), n0 and n1 are the mesh normals at vertices v0 and v1 respectively, r is the random offset used for the object-space texture parameterization (a single floating point value), and i is the index of the specific vertex relative to the single edge that it represents. This index ranges from 0-3, and is used in the Vertex Shader to determine which vertex of the rendered quadrilateral is being processed. All of these values except for i are constant for a given edge, and so are repeated exactly for each set of four edge vertices. This was a necessity at the time of the original implementation, as Vertex Shaders were not yet able to perform texture reads. We take advantage of this modern luxury, and instead store the edge mesh in a texture, with one entry per edge index that contains the above information, except for the index i. This is particularly beneficial when we note that due to the lack of a quad drawing mode in WebGL, we must draw the edges with two triangles, which uses an additional two edge vertices worth of memory per edge. With this scheme, our edge mesh VBOs are reduced to two values: the index i of the vertex within the edge, and the index j of the edge itself (used to index into the edge mesh attribute texture).

We also make a few other deviations from the original algorithm. When drawing edges, rather than displacing the outer vertices in a direction perpendicular to the edge and filling holes with triangular caps in a second pass, we draw in a single pass with less ideal edge geometry. We avoid gaps by displacing the vertices according to the screen projected normals of the vertices at the edges. We also use the screen space texture parameterization described in [HV09].

For drawing edge quads that begin at the edge and extend outward (rather than quads centered on the edge), it is somewhat suitable to draw with depth testing enabled. However, it is likely desirable to draw strokes centered on the edges with proper blending, and so the contours must be drawn after the geometry with depth testing disabled. This presents the familiar occlusion culling problem encountered with the particle based techniques - contour edges on the far side of the object may be rendered. This problem is again dealt with by rasterizing the depth buffer and performing a manual depth test in the Vertex Shader. For the fill-related particle techniques that faded off towards the edges, a single sample was sufficient. For edges, due to aliasing of the framebuffer, fetching a depth value at the screen space projection of the edge vertex is error-prone. We accommodate by instead sampling a small 3x3 window around the desired point and using the minimum depth.



A blobby object with stylized contours.

Results and Considerations for Future Work

Our resulting library provides reasonable implementations of the four main algorithms discussed. All of the techniques run in real time by offloading the majority of their work to the GPU. The techniques are packaged into shaders with accompanying utilities for creating the buffers and textures necessary for their use. Common functionality (that is not applicable to all WebGL applications) is factored out into library utilities, such as the creation of instanced billboard buffers and arbitrary attribute textures. The sample applications demonstrate the parameters for each of the algorithms, as well as provide examples of rendering full scenes with each of the effects.

Each of the techniques is customizable with a predefined set of parameters. While this is sufficient to provide a wide range of effects, the ultimate goal of a library to aid in the prototyping of new effects would benefit from a more modular structure. A modular structure for shader creation based on required attributes would be particularly helpful. Such a system would allow for the runtime creation of custom shader source code in a way that would facilitate experimentation with a part of the pipeline that is not currently customizable.

The library also contains general utilities for making WebGL applications, particularly for deferred rendering. However, as a number of existing libraries provide similar functionality and are widely used, a future implementation of the library should allow for simple interfacing with these existing frameworks.

Acknowledgements

Many thanks to thesis advisor Barbara Meier for her continuous feedback and guidance thoughout this project.

References

[BM96] Barbara J. Meier, Painterly rendering for animation, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, p.477-484, August 1996

[HS04] Michael Haller, Daniel Sperl. Real-time painterly rendering for MR applications. GRAPHITE '04 Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia.

[KM99] Michael A. Kowalski , Lee Markosian , J. D. Northrup , Lubomir Bourdev , Ronen Barzel , Loring S. Holden , John F. Hughes, Art-based rendering of fur, grass, and trees, Proceedings of the 26th annual conference on Computer graphics and interactive techniques, p.433-438, July 1999

[NM00] J. D. Northrup , Lee Markosian, Artistic silhouettes: a hybrid approach, Proceedings of the 1st international symposium on Non-photorealistic animation and rendering, p.31-37, June 05-07, 2000, Annecy, France

[PH01] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. Proceedings of SIGGRAPH 2001, pages 579–584, August 2001.

[WP02] Matthew Webb , Emil Praun , Adam Finkelstein , Hugues Hoppe, Fine tone control in hardware hatching, Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, June 03-05, 2002, Annecy, France

[PF00] Praun, E., Finkelstein, A., and Hoppe, H. Lapped Textures. Proceedings of SIGGRAPH 2000, 465–470.

[MH04] Morgan McGuire, John F. Hughes: Hardware-Determined Feature Edges. Brown University, Juni 2004

[HV09] P. Hermosilla, P.P. Vazquez: Single Pass GPU Stylized Edges. Universitat Politècnica de Catalunya, 2009

[WGL] WebGL Specification, The Khronos Group. http://www.khronos.org/registry/webgl/specs/latest/

[NACL] Native Client Technical Overview, Google. https://developers.google.com/native-client/overview