A GPU-Based Approach to Non-Photorealistic Rendering in the Graphic Style of Mike Mignola

Christian Brown

University of Maryland Baltimore County

Abstract

The subject of Non-Photorealistic Rendering (NPR) is one which tends towards a certain, small set of targeted appearances. Work in NPR - in particular in real-time graphics - is generally aimed at a classic cel-animated, "cartoon" visual style - that is, solid black outlines with clamped ramp shading. While this look is often arresting and interesting, it is a narrow approach to NPR. Even within the bounds of a "cartoony" style, significant variations can be found.

Of particular interest is the art of illustrator Mike Mignola, popularized in his *Hellboy* comic book series. We present a method for rendering a pre-built model in the style of Mignola's illustration. GPU acceleration is used to funnel large numbers of geometry-based edges, without a huge performance drop. Vertex Buffer Objects (VBOs) are used to create strokes along the outside edge of geometry, and along the edges of shadows.

1. Introduction

NPR has long been used as a way to abstract or simplify the output of a renderer. While sometimes often purely aesthetic, as in real-time video games like *Okami* (Figure 1), it has also been used to simplify rendering, or to allow CG objects to blend in with real-time video footage.

One of the less explored realms of NPR is illustrative – that is, NPR in the style of an illustration or drawing. While "sketchy" renderers have been developed, these tend to be based more around replicating the materials used to create an image, and less around replicating a particular style of drawing. We aim to explore the stylistic capabilities of a GPU-based illustration renderer.

In particular, we attempted to mimic the style of graphic novel illustrator Mike Mignola. Mignola's *Hellboy* series is renowned for its extremely recognizable style, with high-contrast lighting, angular designs, and very textured line work. (Figure2) There is a lot of detail, in spite of this contrast. Very small, rough features – visible here in the jacket – is described with small, rough marks. The line between light and dark is almost ragged, as seen along Hellboy's arm and leg where it zigzags. These minor details are not approachable using traditional cel-shading techniques. In such an approach, this detail would have to be described texturally, such that it would only work from certain angles, or geometrically, which would increase the model complexity by orders of magnitude. These limitations combine to make a real-time approximation of Mignola's style intractable using these old techniques. New tools must be developed.

While many NPR solutions have been image-based, doing post-processing on 2D rendered images, we developed a geometrically-based solution. We attempted to use a combination of graftal-based detailing [Kowalski et al. 1999] and fin-extrusion "sketchy outlining" [Loviscach 2005] to do this. Graftals would be used to programmatically detail otherwise smooth surfaces or shadows, while the outlining will be used to draw illustrated edges. Graftals would also be used to modify the edge between light and darkness along a surface. Unfortunately, complexities with the Graftal model of detailing arose during the project. Drawing a line demarking shadow edges proved more difficult than originally thought as well, resulting in several unsuccessful attempts to generate shadow-bounding edges.

Figure 1: Still from the video game *Okami*.

Figure 2: Original illustration by Mike Mignola.

2. Related Work

Hertzmann and Zorin [2000] were interested in simulating artistic shading and hatching in rendered images. In "Illustrating Smooth Surfaces," they analyzed both the technical and stylistic approaches to shading used by artists – in particular, by line-artists who created engravings or woodcuts. By analyzing these pre-existing images, Hertzmann and Zorin were able to develop new approaches to shading, and development methods which, while not strictly realistic, made it much easier for viewers to interpret complex images. They determined that hatch curvature was less useful for describing small details, but instead developed techniques to overlay different directions of straight hatching to describe changing or curving surfaces. They also developed "Mach bands" and undercuts, techniques to enhance the contrast between overlapping but similarly lit areas on the same surface.

All of these techniques represent stylistic achievements. They take advantage of optical illusions or confusion of the viewer's eye to simulate otherwise complex images. Likewise, the use of non-realistic lighting techniques by Gooch et al. [1998] allowed for a far greater apparent dynamic range than a realistic lighting model would have.

This research combines to describe many situations in which stylization is not only aesthetically pleasing, but also conveys useful information. By studying artists rather than physics, these researchers found novel ways to render otherwise complicated subjects.

Several papers have described interactive methods of generating artistic renders. Salisbury et al. [1997] developed a technique for using a vector field and a vector-based collection of brush strokes to generate images that appeared hand-drawn. Their use of vector fields, albeit ones generated by an artist, were a precursor to the direction fields of Hertzmann and Zorin [2000]. The process of creating an image using this technique relied on having an artist place vectors along an image, and "color code" the image based on which stroke would be used to draw it in.

Kalnins et al. [2002] continued with the interactive approach, but shifted from placing brush strokes on an image to placing them on a 3D model. While not real-time, the combination of a surface shader and a lighting model provided a complete package for controlling the artistic appearance of an NPR system.

With the development of GPU-based rendering, complex variants on traditional rendering techniques were developed. McGuire and Hughes [2004] described a technique to use the GPU to determine edge-features and outlines, rendering them entirely using Vertex Buffer Objects. Loviscach's "Stylized Haloed Outlines on the GPU" [2005] described a VBO-based GPU rendering technique for outlined edges that also uses multiple passes and image-based jittering to create a very sketchy appearance. (Figure 3)

Finally, Kowalski et al. [1999] described a technique to approximate otherwise complicated appearances by stylizing and abstracting them using "graftals". These graftals are small, easily modified pieces which can be used to draw fur, tufts, or other surface detail on an otherwise simple model.

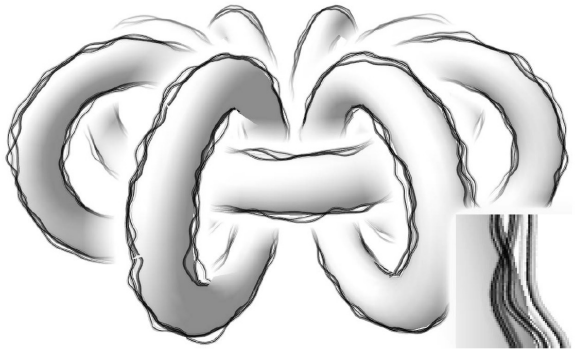

Figure 3: Haloed sketchy outlines. [Loviscach, 2005]

3. Implementation

We attempted to use four techniques to approximate Mignola's artistic style.

First, we outlined the models using the finning technique described by McGuire and Hughes [2004].

Second, we used high-contrast lighting to give the same inky shadows Mignola uses. This was done on the GPU, using a vertex shader. We attempted to use a texture to provide the impression of drawing in the fills with a marker, but were unable to orient the textures properly without more information on the GPU.

Third, we attempted to use graftals to simulate otherwise invisible surface detail – pock marks, bumps, etc. These were intended to be defined ahead of time, and will be present both in shaded and lit areas. They will particularly be used to define texture in large, well-lit areas.

Fourth, we attempted to use several techniques – most notably graftals and textured outlines – to create complex divisions between lit and shaded areas. The "zigzagging" shadows seen in Figure 2, as well as other shading techniques, were explored.

These techniques were applied to black and white images, mimicking the look of an un-inked illustration by Mignola.

3.1. Outlining

Outlining was executed using the Vertex Buffer Object based GPU-accelerated technique described by McGuire et al. [2004]. The model file is read into an Edge Mesh, in which information about an edge in the model is encoded into vertex position, color, and multitexture information.

In order to determine whether an edge in a mesh is an outline edge, the normal vectors of the two facing faces must be compared. If one face normal faces the camera and the other does not, the edge between them is an outline edge. While this technique only works with faces that are guaranteed to be coplanar, this includes any triangular mesh. We tested this technique on several OBJ files with success.

The relevant information can be encoded in several ways. While Loviscach encoded the information in to several floating-point values, rather than using the full 6-value method, this was not useful for our method. The four-value

technique only worked when only edge-outlines were being drawn. Since we were attempting to draw non-outline edges along shadow boundaries, we had to include all 6 vector values.

These 6 values are called v0, v1, v2, v3, n1, and n2. Each edge in the mesh must be described using all 6 of these values in the Edge Mesh. v0 and v1 are the positions of the endpoints of an edge, v2 is a third point on one face that uses the edge, and v3 is the position of the third point on the other face. n0 and n1 are the normal vectors of v0 and v1, respectively. The face normals of each of the faces sharing the edge, nA and nB, can be calculated using v0-4 and n0-1. (Figure 4)

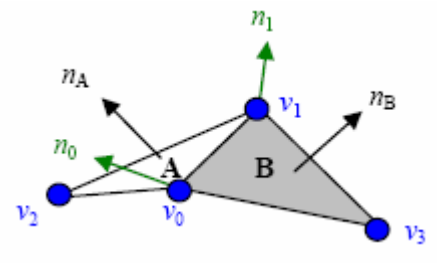

Figure 4: v0, v1, v2, v3, define the relevant vertices, while n0 and n1 define the relevant normals. nA and nB can be calculated in the vertex shader using cross products. [McGuire and Hughes, 2004]

Since OBJs are described by indexed vertices and faces, we must iterate across the faces and store the implicitly defined edges. Each edge will be defined twice (once by each face which shares it), so we must protect against edge duplication. Once the edges are extracted, we can define the Edge Mesh.

Since we used VBOs to pass edge information to the vertex program, we must encode edge information into a single vertex structure.

When using VBOs, each vertex is described using an array of floats – one array for each attribute of the vertex being described. We used these attributes to encode edge information – specifically, the MultiTexCoord attributes. We encoded v0 and v1 into the Position and Color attributes, and the remaining attributes into MultiTexCoord0-3. We called this collection of edge-describing attributes packed into a single vertex object an "Edge Vertex Object."

Using only a vertex shader and a fragment shader, we cannot create additional geometry on hardware. Thus, while each Edge Vertex Object contains all necessary information to create an edge quad, the edge shader can only do this if it receives for "vertices" that it can transform into the corners of the quad. This means each Edge Vertex Object (EVO) must be duplicated four times – one for each corner.

In order to make sure each EVO is transformed to the correct corner, we must include an indexing value. The first corner is index 0, the second corner is index 1, etc. This fourth value must be passed using an additional MultiTexCoord attribute.

The vertex shader uses these attributes to transform the vertices to the appropriate positions. To do this, use the following vectors and values for them:

s0 = (MVP * v0).xy

s1 = (MVP * v1).xy

p = normalize(<s0.y – s1.y, s1.x – s0.x>)

s0 and s1 are the screen-space endpoints for the edge, and p is a normalized vector perpendicular to s0 and s1. This means the final position on one vertex in the vertex shader is:

Position	=	s0 + p	if i = 0
		s1 + p	if i = 1
		s1 - p	if i = 2
		s0 - p	if i = 3

If the edge is not an outline, its position is transformed to behind the clipping plane. This also culls it from the fragment shader, preempting the need for any additional culling.

Stroke information and appearance can be passed as a texture, and applied to the strokes. This can be used for a few purposes. Most obviously, and as used previously, it can be used to define a stylized stroke appearance.

3.2. Shading

On close examination, in all but straight black-and-white outline illustration, Mignola's drawings have subtle gradients found in them. These gradients add a little depth to even the well-lit areas of the drawing. While in Mignola's illustration, these are often abstracted and simplified gradients, such simplification is outside the scope of this paper. To simulate the subtle shading, we simply implemented a clamped and modified shading algorithm.

For the large "pools of shadows" that cover the models, we simply monitored the intensity of light in the fragment shader. If intensity was below a constant, α, the fragment color was set to black. Otherwise, it was modified using the following equation:

Color = ((i*0.5)*(i*0.5)+0.75) * White

Where i was the intensity. Using this equation, the gradient range was narrowed and moved, from ranging from 0.5 to 1, to ranging from 0.75 to 1, and with a steeper falloff. This provided a subtler gradient than the default values.

We experimented with using a texture to draw the black shadows, scanned from large fills in Mignola's drawings. While the textures looked extremely hand drawn and worked in still frames, we were unable to get them to animate properly without information on surface orientation.

3.3. Surface Graftals

We attempted to implement graftals, as described by Kowalski et a. [1999], but discovered several pitfalls which prevented them from being used.

The findings of Hertzmann and Zorin [2000], and the techniques they used to generate a "direction field" along the surface of the model, were similar to the problems we faced while trying to develop graftals. We had expected to be able to avoid the problems of surface-orientation by using an illustrative technique – the assumption being that since the

result was so stylized, no orientation of graftals would be necessary. Unfortunately, this was not the case.

Close examination of Mignola's work showed that the small marks used for surface details were not at all randomly oriented or distributed – while lacking depth, they all followed the 2D orientation of the surface they were placed along. This is most easily seen in Figure 4. The scale "graftals" placed along the tail of the mermaid are oriented in 2D to match the curve of the tail. This orientation means graftals must be generated based on surface geometry.

We then attempted to use an approach more directly related that that outlined by Hertzmann and Zorin. We tried generating graftals at edges, parallel to the screen but with one edge set to an edge on the mesh. Unfortunately, the orientation to the screen made self-intersection a significant problem. Without any time-consuming self-intersection prevention, there was no way to guarantee surface details wouldn't disappear into geometry – especially with randomly placed graftals.

Due to scope and time constraints, we were unable to attempt other solutions.

3.4. Shadow Outlining

As with surface graftals, shadow outlining proved more difficult than originally expected.

Originally, we had intended to use edge-aligned graftals, as described above, to provide variation along shadow edges. The self-intersection problems of surface-oriented graftals, however, were also a problem with shadow-bounding graftals.

We moved on to attempt to use textured strokes to define edges. While self-intersection was still an issue, it was more manageable with controlled stroke-width.

To determine if an edge was shadow-bounding, we simply used the same dot product used in the rendering equation to calculate light intensity at a point. As with the intensity clamping used for the shading, we used α as a cutoff. If the intensity on one edge-connected face was greater than α and the intensity on the other was not, that edge was a shadow-bounding edge. Intensity was calculated using the usual formula:

Intensity = dot(light direction, face normal)

The problems arose with actually finding a shadow-bounding edge. With an outlined edge, there is no interpolation between points – that is, the outline will always be made of discrete, straight line segments. But with shading, there is interpolation between vertices, and even texture lookups to modify the vertex normal if normal mapping is used. All of this additional normal information is only accessible to the fragment shader. Since we did all implementation of the stroke-generation on the vertex shader, there was no way to get shadow-boundaries smoother than the edges of the triangulated mesh. In some cases, depending on triangulation, this led to a zigzagging stroke down the front of the mesh, which did not actually follow the shading.

Without any sort of interpolation across triangles, there was no way to avoid this jaggedness.

4. Results

Edge stroking has been implemented on the GPU using the VBO solution described above. Any OBJ model can be loaded into an Edge Mesh, passed to the GPU in a VBO, and rendered with all edges as strokes. With outline-culling implemented, only the outline edges of the model are drawn. These strokes have UV coordinates defined, and can be drawn with textured strokes. (Figure 5)

The "fill" of the model is a separate render pass with a different GLSL shader program. The black shadows are shaded using the fragment shader, as described above in section 3.2. This shading is placed behind the strokes. (Figure 6)

Attempts to render surface-graftals failed, due to the difficulty of placing graftals along a surface without using any direction field detection algorithms and without self-intersection problems.

Attempts to render shadow-bounding strokes failed as well. While the narrowness of the strokes prevented any significant self-intersection, the inability to get any finer resolution than the triangle edges also led to zig-zagging shadow boundaries.

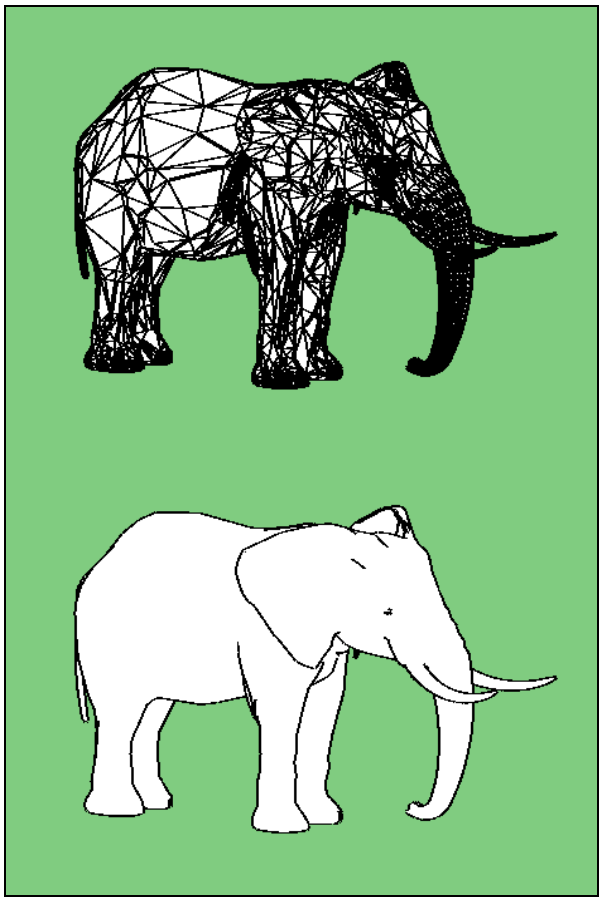

Figure 5: At top, all edges are drawn. At bottom, only outline edges are drawn.

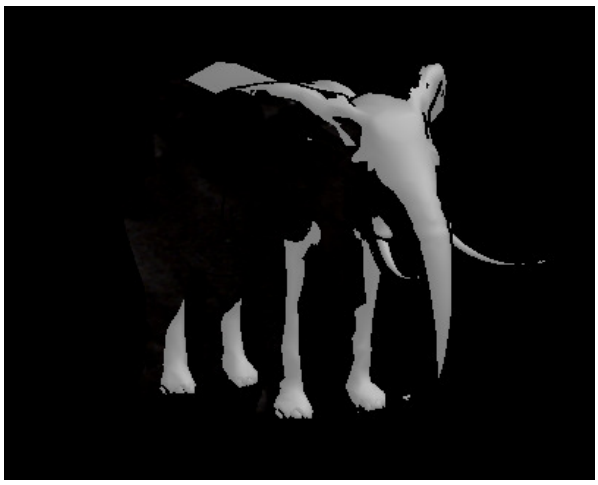

Figure 6: An elephant model is Shaded using a combination of stroked outlines and shaded fills. Gradients are used to include subtle shape cues.

In spite of the problems faced, we were able to develop an interesting-looking real-time shader which is comparable to the style of Mike Mignola's illustrations. We implemented an entirely GPU-based rendering process to outline and shade any OBJ model loaded into the program. While the use of an Edge Mesh leads to approximately 9x the number of faces as a non-outlined model [McGuire and Hughes, 2004], the use of VBOs provides an offsetting efficiency, which allows even a 100,000 triangle model to run in real-time on a GeForce 6600 256 MB graphics card using GLSL. We also were able to pinpoint issues and concerns in more artistically-based NPR techniques – specifically, the importance of both gradient fills and direction fields.

5. Future Work

The problems faced by the researchers pointed out several avenues of approach to further work in the field of stylistically-targeted NPR systems. First and foremost, a system to preprocess a model and develop a direction field in real-time is a must. Any sort of brush or stroke-based rendering, including our original idea of using surface-based graftals, would need direction information on a per-vertex basis.

Modification of the GPU outlining algorithm to include the ability to move edges to interpolate across triangles would be valuable as well. This would enable exploration of non-edge outlines which follow shadow boundaries, surface orientation, etc. without having jagged, zigzagging paths.

Exploration of other, non-physically-based shading models would benefit all NPR techniques greatly. Our studies of illustrations showed that shading often follows the 2D outline of an object, rather than its curvature or surface orientation. This flies in the face of traditional graphical lighting models. Any attempts to replicate this on a GPU would go a long way towards generating convincingly illustrative and artistic real-time NPR techniques.

The development of GPU-based geometry shaders will make the techniques used to create edge fins much simpler in the future. While the researchers did not have access to an NVIDIA 8800 graphics card, needed to use the new geometry shaders, future researchers could use the geometry shader to generate the additional vertices instead of passing in four copies of the EVO. This would cut the number of EVOs to one fourth of the number used in our version, making this method that much more efficient.

The use of VBOs precludes traditional vertex-based animation techniques. While the VBOs could be updated every frame, this would eliminate the efficiency offered. Several techniques have recently been used to "fake" animating VBOs, however, which could be applied to our techniques, allowing them to be used on changing models.

References

Jan Fischer, Dirk Bartz and Wolfgang Straßer, "Artistic Reality: Fast Brush Stroke Stylization for Augmented Reality," SIGGRAPH 2005.

Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen, "A Non-Photorealistic Lighting Model for Automatic Technical Illustration," SIGGRAPH 1998, pp. 447 – 452.

Michael Haller and Florian Landerl, "A Mediated Reality Environment using a Loose and Sketchy rendering technique," ISMAR 2005.

Aaron Hertzmann and Denis Zorin, "Illustrating Smooth Surfaces," SIGGRAPH 2000, pp. 517 - 526.

Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes and Adam Finkelstein, "WYSIWYG NPR: Drawing Strokes Directly on 3D Models," SIGGRAPH 2002.

Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden and John F. Hughes, "Art-Based Rendering of Fur, Grass, and Trees," SIGGRAPH 1999, pp. 433 - 438.

Jörn Loviscach, "Stylized Haloed Outlines on the GPU."

Morgan McGuire and John F. Hughes, "Hardware-determined edge features," 2004.

Michael P. Salisbury, Michael T. Wong, John F. Hughes and David H. Salesin, "Orientable Textures for Image-Based Pen-and-Ink Illustration," SIGGRAPH 1997, pp. 401 - 406.

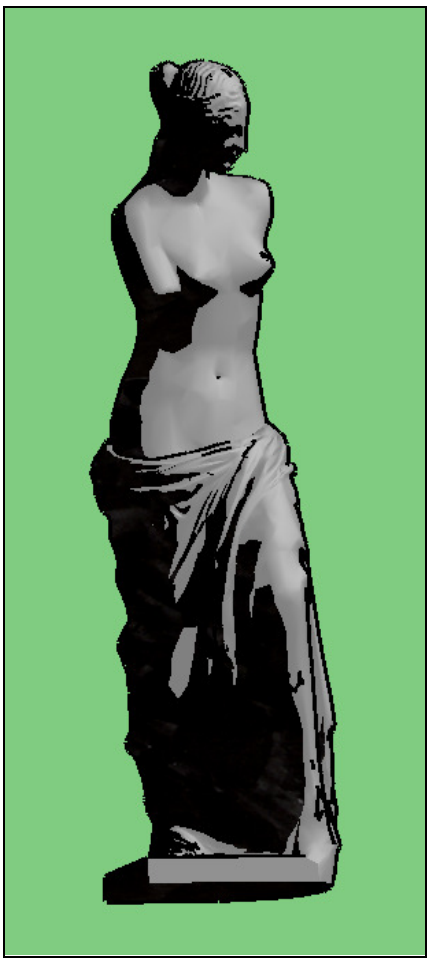

Figure 7: High-poly render of the Venus model using the Mignola shader.

Figure 8: High-poly render of the Ataneal model using the Mignola shader.