Hand Drawn Rendering

Matthew McMullan and Ian Ooi

May 3, 2012

Abstract

There are many photo-realistic, or near photo-realistic games and engines available, and the video game industry in general has a fairly strong focus on realistic and semi-realistic 3D graphics, while the more artistic, non-realistic styles have been more limited to 2D games.

This paper describes our engine, which seeks to render scenes with a non-photo-realistic, "hand drawn" style. Cel-shading is used, as well as different methods of drawing outlines on objects to produce a "toon" look. To shade the scene, a cross-hatching effect is applied. Combined, the effects can be applied in real-time.

1 Introduction

Video games do not often include non-photorealistically rendered 3D graphics, especially with more stylized effects such as a hand-drawn look. As photorealism of 3D graphics improves, it becomes increasingly difficult for smaller, independent game developers to produce 3D titles with strong visuals, due to their lower polygon budgets and general inability to compete with large, well-established publishers and studios and their available resources.

An alternative to trying to compete with the larger publishers is to use non-photorealistic graphics. Outside of independent, small studios, relatively untried styles like many non-photorealistic effects are too large a risk to take on. For independent developers however, this difference can make a game stand out from the rest of the market and allow it to compete.

The hatching effects indicated by "Real-Time Hatching" by Praun et al., provide an interesting, distinctive, style which would fit well in an independent development environment. Since it is implemented as a shader, it may be modified and swapped in and out for other effects without major changes to the engine itself, allowing the same engine to also use cel-shading, cartoon outlines, or even combinations of these effects. Other effects could also be applied in the future.

2 Prior Work

A method of cross hatching was described by, Praun et al in their paper Real-Time Hatching, which shaded an object using textures which had the hatch marks drawn. Several different sizes of image were used, organized into a *tonal art map*, where different textures were used, with varied average lightnesses. Each darker texture in the TAM contains the same lines as the lighter textures, and each larger texture contains the same lines as the smaller textures in order to have continuity when blending between the textures. The different sizes of texture are used in a mipmap so as to have continuity in the size of the lines as the camera moves closer or farther away from objects in the scene.

Textures in the TAMs are generated by drawing numerous lines and choosing the "best fitting" one, based on a fitness function, and then only drawing that one into the final image. To determine the fitness, given a candidate stroke s_i, the average tones T_l and T_l^i are determined at level l of the TAM. The darkness the stroke would add is then expressed as:

$$\sum_l \left(T_l^i - T_l\right) \tag{1}$$

These fitnesses are also calculated in an image pyramid to achieve greater uniformity, such that a stroke that is near others in a finer level of the pyramid will actually overlap, giving instead:

$$\sum_{p,l} \left(T_{pl}^i - T_{pl}\right) \tag{2}$$

At a certain threshold, vertical lines are drawn as well as horizontal. The marks vary in size, as well as orientation. Different patterns such as stippling may also be used in the TAMs.

To apply these to the model, a method of blending is used. Lighting values are obtained for each vertex, and the individual TAM images are blended across the surface using a 6-way blending scheme. [4, 6]

Lapped textures are used to apply the TAM to arbitrary surfaces to avoid extraneous artist input. [3]

Cel-shading is a well known effect, with numerous methods available to produce similar results. We found little published work relating to methods to cel-shade objects in a scene, but general research and intuition indicated the use of step functions or thresholds to limit the number of colors in an image and to force other shades to a few values, either through calculation or the use of pre-calculation of desired colors and color indexing.

Philippe Decaudin shows some methods of drawing outlines, using a Sobel filter depth information, to find the silhouette, and again on the normal map to produce continuous cartoon outlines for models. An undescribed method of cel-shading is also used, as well as a method of producing proper shadows for this style [2].

Decaudin's method requires the use of a Sobel filter, which we implemented. Again, it was difficult to find publications relating to the creation of the Sobel filter, but descriptions of its use and implementation are relatively easy to find. Chris Wyman's website contained a short description [7] which we utilized for completing our implementation.

3 OGRE

In order to expedite our project, we choose to use the OGRE 3D rendering engine. OGRE handles our scene management, imports models, handles textures and geometry, and sends the geometry to the GPU. This allowed us to focus on the effects and allows for modularity, so that we can swap out effects with ease.

We each compiled the source on our systems, and after we managed to set up the engine, we created a simple scene and used our various textures and shaders to produce effects.

To test frame rates, we used the built-in frame counter that OGRE provides. There was an issue in the library code however, that caused the frame rate counter to display incorrect values. Accordingly, we implemented a fix so as to be able to test our frame rates and ensure they are real-time. The bug in OGRE was found in its file SdkTrays.h. The fixed version is included for convenience.

4 Cel-Shading

To cel-shade our scene, we attempted two methods. First, we tried calculating, for each pixel, a new color, limiting each channel to a finite number of shades. Given a desired number n_s of shades per channel and the original color $C_{text} = (R, G, B)$, we calculate the new, cel-shaded color as:

$$C_{cel} = \frac{(\lfloor (C_{text} * n_s) + 0.5 \rfloor)}{n_s} \tag{3}$$

This method, when applied to our ogre head mesh, produces results as seen in Figure 2(a). Some undesirable artifacts can be seen. There is some distortion in the color as an effect of modifying the RGB values, as well as loss of detail due to too quick a drop-off in brightness, resulting in large parts of the model being colored black, or in our case, a dark grey due to limitations we placed on how dark the color can be. The edges of the cels are also poorly defined, with some patchiness caused by irregularities in the original texture of the model. The patchiness can be solved by only applying the technique to simply colored models, instead of using noisy textures. To attempt to correct the lost detail and distortions in color, we convert the colors to the HSL color representation as seen in §4.1 and apply the same method, but only to the saturation and lightness channels before converting back to RGB using the method in §4.2 and applying the color to the fragment. This causes desaturation as well as changes in brightness, producing the effect of losing color as the shade gets darker. It also preserves texture detail, which may be desirable depending on circumstance. For a more noticeable effect, smaller values of n_s are used.

4.1 HSL Color Representations

The HSL color representation contains three components, hue, saturation and lightness, which represent points in the RGB color scheme. Hue is calculated by first finding chroma. Chroma (C) in turn is calculated as follows:

$$C = M - m \tag{4}$$

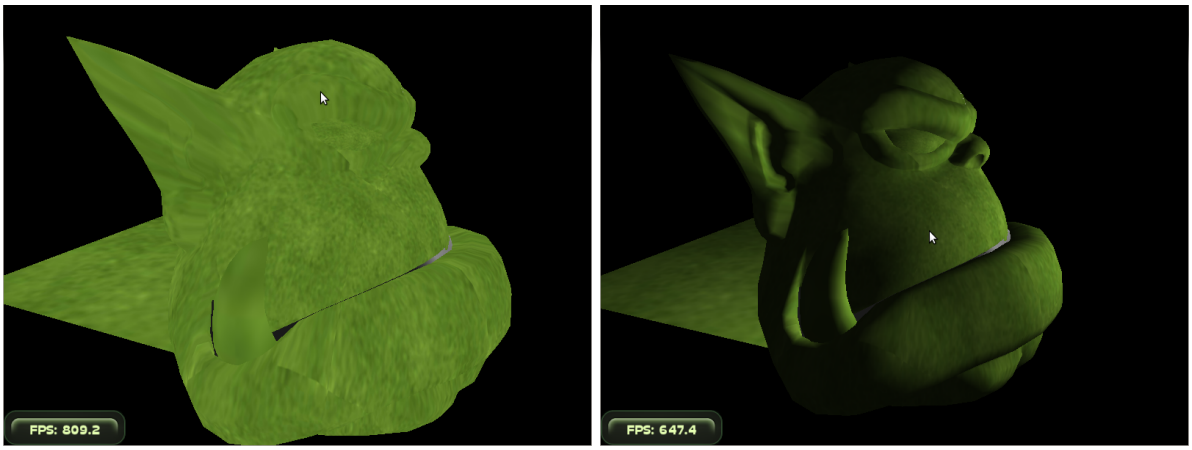
where

$$M = max(R, G, B) \tag{5}$$

$$m = min(R, G, B) \tag{6}$$

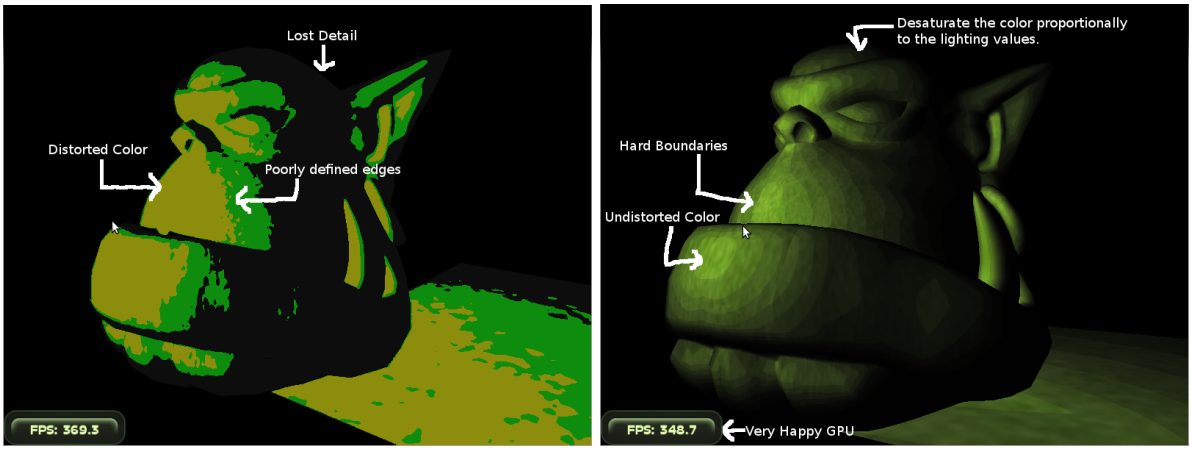
and R, G, and B are the red, green, and blue values of the color respectively. The hue is then defined by a piecewise function:

$$H' = \begin{cases} 0, & \text{if } C = 0 \\ \frac{G-B}{C} mod 6, & \text{if } M = R \\ \frac{B-R}{C} + 2, & \text{if } M = G \\ \frac{R-G}{C} + 4, & \text{if } M = B \end{cases} \tag{7}$$



(a) (b)

Figure 1: A simple texture mapping can be seen in 1(a). Our per-pixel lighting.



(a) (b)

Figure 2: Our original cel-shading 2(a) and improved cel-shading 2(b).

To calculate the lightness L, the M and m values are used from before:

$$L = \frac{1}{2}(M + m) \tag{8}$$

Finally, the saturation value is obtained through the following:

$$S = \frac{C}{1 - |2L - 1|} \tag{9}$$

4.2 Converting from HSL to RGB

To convert HSL to RGB, assuming $H \in [0°, 360°)$, $S_{HSL} \in [0, 1]$, and $L \in [0, 1]$, the first step is to again calculate chroma.

$$C = (1 - |2L - 1|) \times S_{HSL} \tag{10}$$

The hue H in our method was unmodified, and therefore required no conversion to find H'. Another value X is necessary to calculate colors.

$$(R', G', B') = \begin{cases} (0, 0, 0) & \text{if } H \text{ is undefined} \\ (C, X, 0) & \text{if } 0 \leq H' < 1 \\ (X, C, 0) & \text{if } 1 \leq H' < 2 \\ (0, C, X) & \text{if } 2 \leq H' < 3 \\ (0, X, C) & \text{if } 3 \leq H' < 4 \\ (X, 0, C) & \text{if } 4 \leq H' < 5 \\ (C, 0, X) & \text{if } 5 \leq H' < 6 \end{cases} \tag{11}$$

R, G and B are then found by adding a factor m to each term, where m is based on lightness. [1]

$$m = L - \frac{1}{2}C \tag{12}$$

$$(R, G, B) = (R' + m, G' + m, B' + m) \tag{13}$$

5 Cartoon Outlines

We produce cartoon outlines through two different methods. First, we apply a Sobel filter as a post-processing effect, resulting in the lines seen in Figure 3(a). A few issues are present, such as some extra lines drawn on the tip of the model's tusk, and the incomplete lines around the eyes.

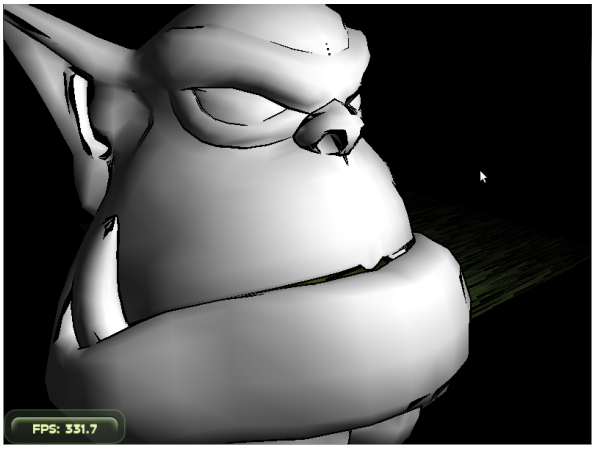
Our filter uses color information to detect the edges and draw outlines accordingly, due to limitations in the current version of OGRE which does not allow access to depth information. Converting to a depth-based filter would solve some of these issues, but for creases in the model, such as the incomplete lines around the eye, information about the normals in the area would be necessary, such as described in [2], where outlines are drawn using a combination of depth and normal map information. Since our current approach uses per-pixel lighting, we would need to modify our method to be able to access information about the normals. For depth, the next version of OGRE will support access to the depth information.

Due to the issues of the first method, we implemented an alternative method of drawing outlines. The geometry is drawn a second time, with the vertices offset from their original positions relative to the normal at that vertex. The anticlockwise faces, i.e. front faces, are culled such that only the back faces (clockwise faces) are rendered. The faces are drawn with a solid color, such as black, to produce the effect of an outline around the geometry. This method also fails to handle creases, such as round the eyes, but produces a much more continuous, tidy outline for the objects in the scene as demonstrated in figures 3(b) and 3(c). This method also has the advantage of working in object space as opposed to image space, allowing some objects to be rendered with outlines and others to not have outlines.

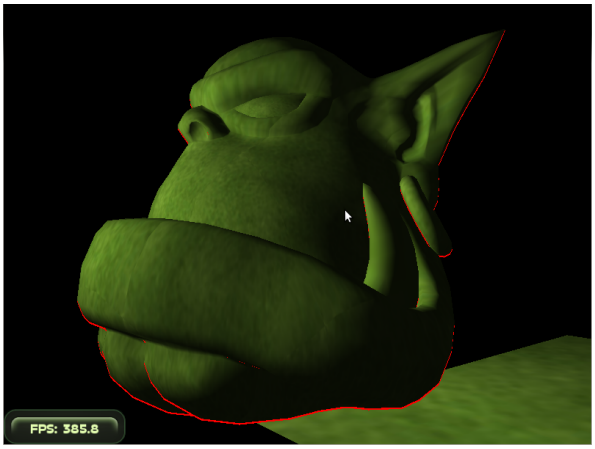
6 Hatching

We generate TAMs similar to the method described in [4], but did not use an image pyramid. This led to some loss of uniformity, which we attempted to combat with a few different stratified sampling methods. First, we attempted to restrict where our candidate lines are drawn, subdividing our image into a grid and at each iteration forcing the line into that region in either the horizontal or vertical direction. Due to the fact that we perform this step when drawing candidate lines, applying it to each separately, and only 1 of 1000 was chosen, the line placement is still non-uniform. Next, we tried applying the first scheme in both the horizontal and vertical directions simultaneously, but this resulted in some diagonal patterns in the line placement. Finally, we tried performing local averages to determine the fitness of a line in a limited, local area of the image, but met with little success.

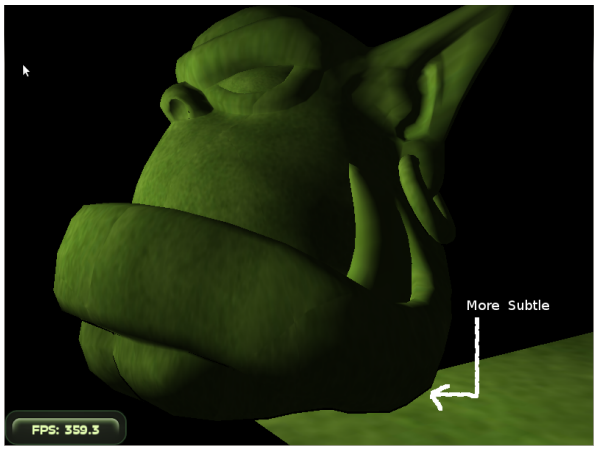
Currently, our implementation uses a per-pixel lighting model, as opposed to the per-vertex lighting used in [4]. This allows us to perform a somewhat simpler blending, between two different "brightnesses" of the TAM and two different sizes, where the sizes translate to levels in the mipmaps. Blending is handled in a Cg shader, which is called by an OGRE material file (which also applies a vertex shader). The blending is achieved by applying weights to the different textures and performing a weighted average to



(a)



(b)



(c)

Figure 3: The effect of our Sobel filter is visible in 3(a). Note the extra lines captured in the tip of the tusk, as well as around the ear, and parts of lines missing, especially at creases in the model, such as around the eye. Our alternative outlines are shown in 3(b), drawn in red for visibility and in black in 3(c).

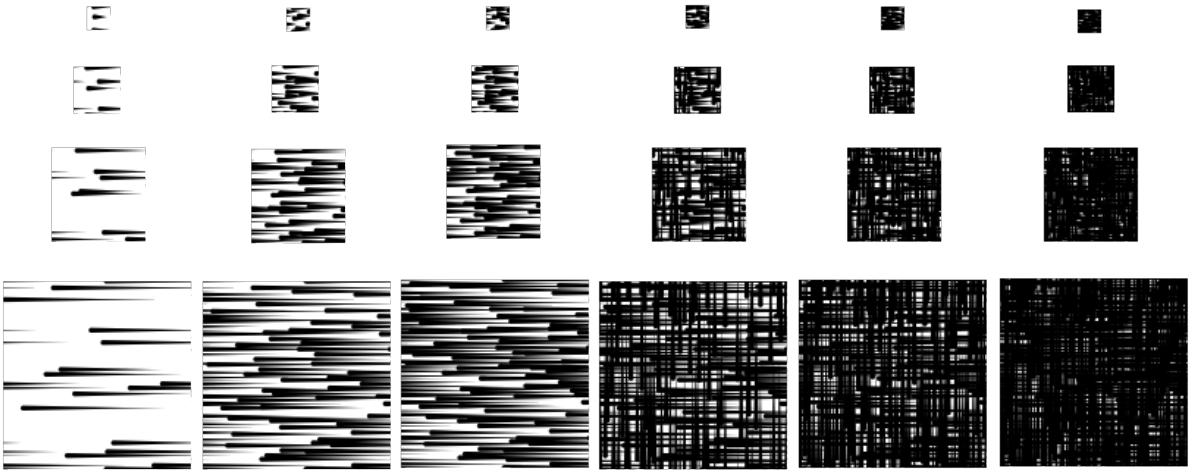


Figure 4: Our generated tonal art maps. Loss of uniformity is seen, due to lack of an image pyramid. To counteract this, we used a scheme of limiting the line locations based on a uniform grid, with a random location chosen within the grid cell. The lines are still non-uniform, but further improvements could be made by generating from the bottom-up and using the pyramid method. Additional levels of the TAM omitted for space reasons.

get a color value for the pixel. This color value is then multiplied by the color value of the object's texture (or face color) to allow for non-white objects.

To choose the image from the TAM with the desired brightness, we first calculate the diffuse fall off. This value indicates how the light value changes as the point in question moves farther from the light source. Given a light source at distance r from the point in question, diffuse fall off is calculated as:

$$DFO = min\left(\frac{10000.0}{r^2}, 1.0\right) \tag{14}$$

This value is then used to calculate a color $C_{bright.}$ as follows:

$$C_{bright.} = DFO * (L_{diff} * Lit_y + L_{spec.} * Lit_z + L_{amb.}) \tag{15}$$

The $C_{bright.}$ is then converted to HSL from RGB so we may extract lighting and saturation values as seen in §4.1 Once the color has been converted, the lightness is used to determine which two darkness levels of the TAM to blend, using thresholds as seen in Table 1. These thresholds are determined through a tree-structured series of if statements for efficiency. The distances of the current brightness from the low and high values of the range are then determined, and these weights are used to calculate a weighted average for each of the textures to blend at the pixel.

After calculating the color from the texture and the color from the cel-shading, the final color of the pixel may be calculated as follows.

$$C_{out} = C_{TAM} * C_{cel\_shading} \tag{16}$$

7 Results and Discussion

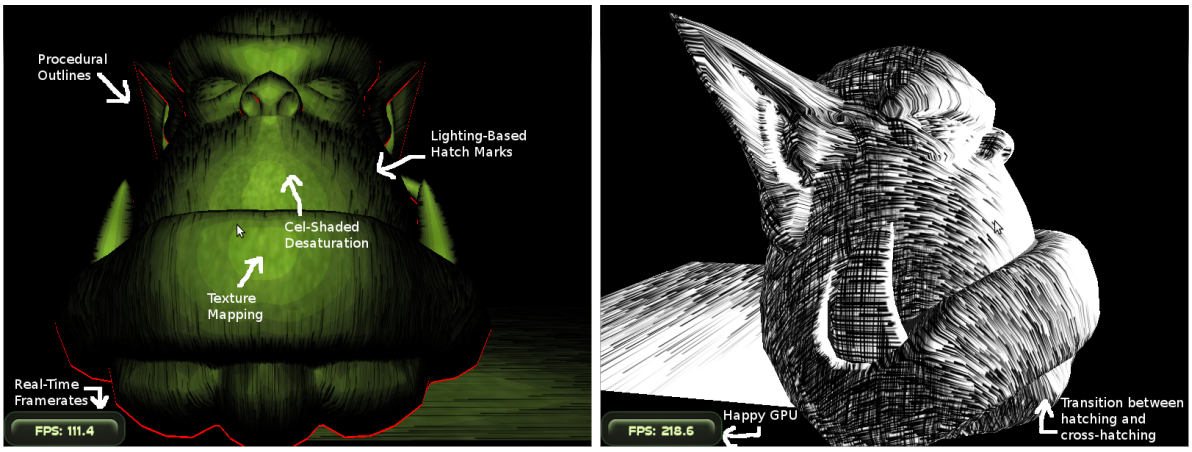
As can be seen in figure 5(a), we achieve a final scene, rendered in real time, with cel-shading, cartoon outlines, and cross hatched shading. We tried different combinations, which are not all shown, such as applying the hatching before cel-shading. Another notable change we tried were variations in how dark the object could be shaded, to vary how much of the shading was suggested by the cross hatching and how much was handled by darkening the object. Overall, the effects are very easy to modify and combine.

Our method overall is a fairly expensive effect in terms of calculation, though it still runs in real-time. Cross-hatching also relies heavily on the texture mapping, requiring either an automatic method of mapping the textures of the TAM to each model, or extensive user input. The hatching effects shade properly, and produce coherent, aesthetically pleasing results.

Our outlines are currently imperfect, failing to draw outlines for creases in the geometry, and in our Sobel filtering method, applying to every object in the screen as well as drawing lines where an outline should not be present. This is a result of our Sobel filter working off of the color information instead of the depth and normal information in the scene. Our

Low	High	TAM Level 1	TAM Level 2
0.0	0.0033	Black (0.5, 0.5, 0.5)	TAM 5
0.0033	0.1060	TAM 5	TAM 4
0.1060	0.2053	TAM 4	TAM 3
0.2053	0.4040	TAM 3	TAM 2
0.4040	0.5364	TAM 2	TAM 1
0.5364	0.7351	TAM 1	TAM 2
0.7351	0.93	TAM 1	TAM 0
0.93	1.0	TAM 0	White (1, 1, 1)

Table 1: The lightness thresholds and the corresponding levels of the TAM at the given brightness. The individual textures of the TAM are labeled in order according to their average brightness, with 5 being the darkest and 0 being the lightest.



(a) (b)

Figure 5: Our results are shown in 5(a). Note the combination of all of the effects, and the real-time frame rates. In 5(b), we demonstrate cross-hatching applied to a plain, untextured model.

alternative method fails to capture the full range of outlines that would be aesthetically ideal.

8 Summary and Future Work

We presented a technique for producing real-time nonphotorealistic renderings that mimic a hand-drawn style. This technique made use of several, previously disjoint, techniques to produce components of the final rendering [4] [6] [2]. On top of the combination, we have modernized several of the components to achieve better effect, and to allow for a coherent look [4] [2]. Our results could be improved in a number of areas by applying this knowledge to expand upon our technique.

The TAM generation system can be improved by adding extra constraints to distribute the lines better. Our first improvement would be to generate the TAMs in reverse order. Currently, we generate the TAMs from smallest and least dense to largest and most dense. This allows lines to be added in the smaller levels that would be far from ideal in the larger and more dense levels. To do this, we would generate a number of possible lines on the largest level and select a subset of those that produces an optimal TAM. In reducing to smaller levels, ideal subsets would be extracted from the larger concrete set. When determining the placement of the lines in the larger levels, the smaller levels would be considered in the weighting function. This allows for lines that are poor choices on any level to get naturally weeded out of the system. This method would require a large number of samples, so to improve the efficiency, we would introduce a stratified sampling system that would guarantee that the initial pool of lines had a more uniform distribution across the surface. With this improvement, the number of lines to consider would be greatly reduced and so generation times would go down.

Another improvement to the hatching method involves the creation of different TAM styles. In previous work, other styles such as stippling and multicolor lines were generated to achieve a different aesthetic [4] [6]. Although these techniques were not specifically described in their respective papers, they could be implemented based on the core generation technique. We could also implement haphazard grayscale splotches. When multiplied with the output color in the final steps, this may achieve a somewhat paint-like look. Our current implementation does not allow for variable stroke widths. Implementing such an addition would allow for more control in any of the prosed TAM generation techniques.

There are several different styles for cel-shading that do not preserve the color of the underlying texture as well [2]. We could implement one of these techniques to see if it melds better with the hatched style. It is also a possibility that our current technique would look better with some of the proposed TAM styles, while a more traditional cel-shading implementation could work better with others.

It is also possible to remove some of the artist interaction from the process of mapping the TAMs to the surface. By implementing lapped textures [3] like the original hatching papers, we could reduce artist involvement primarily to specifying the principle curvature of the surface [4] [6]. Even this involvement could be eliminated by adapting techniques from newer papers to work as input to the lapped textures implementation [5].

As it stands, our implementation requires significant branching and several divisions in the fragment shader. Together, these take a significant proportion of the time spent rendering each frame. In order to use it in a production game, it may be necessary to optimize these elements to reduce branching and possibly offload more of the work to the vertex shader.

A final improvement would be to apply the Sobel filter to both a depth map and the normal map. This would catch depth discontinuities and highlight some more important surface features. This would eliminate the problem with our current filter that causes lines to be drawn at color discontinuities This could be added by extending the renderer to use multiple render targets in a way similar to differed shading. The color and alpha channel would be rendered to one image while the normal and depth would be rendered to another. The Sobel filter can be applied using the second image on the first to achieve the artistic strokes. This would be a significant improvement over our current offset geometry method.

References

[1] Max K. Agoston. *Computer Graphics and Geometric Modeling: Implementation and Algorithms*, pages 305–306. Springer, 2005.

[2] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[3] Praun, Finkelstein, and Hoppe. Lapped textures. In *SIGGRAPH*, 2001.

[4] Praun, Hoppe, Webb, and Finkelstein. Real-time hatching. In *SIGGRAPH*, 2001.

[5] Vergne, Barla, Granier, and Schlick. Apparent relief: a shape descriptor for stylized shading. In *NPAR*, 2008.

[6] Webb, Praun, Finkelstein, and Hoppe. Fine tone control in hardware hatching. In *NPAR*, 2002.

[7] Chris Wyman. Sobel filtering. 2008.