Advanced Real-Time Cel Shading Techniques in OpenGL

Adam Hutchins
Sean Kim

Cel shading, also known as toon shading, is a non-photorealistic rending technique that has been used in many animations and video games. The first video game to popularize the technique was *Jet Set Radio*, published in June of 2000. Since then it has been used in well over 100 videogames; however, the popularity of true cel shading – where meshes have outlines and surfaces are shaded, not textured, seems to have waned in recent years. Instead, most modern cartoon-like games, like *Borderlands* or *The Walking Dead*, use traditional texturing with textures that are manually shaded and outlined by an artist.

This project explores advanced methods of true toon shading, hopefully showing that it can still produce visually appealing effects. It starts with basic shading, which can be seen as sampling a one-dimensional texture, and follows the ideas presented in the paper *X-toon: An Extended Toon Shader* by Barla et al. to expand this to a two-dimensional texture. This allows view-dependent effects to be used in the shader such as depth of field, highlighting, and near-silhouette lighting. Secondly, shaders that make a less realistic and more stylized highlight as discussed in *Stylized Highlights for Cartoon Rendering and Animation* by Anjyo et al are investigated. Finally, the two effects can be combined together to form a toon shader that produces customizable stylized highlights.

Basic Cel Shading

An early form of cel shading was first described in 1996 by Philippe Decaudin in his paper *Cartoon-Looking Rendering of 3D-Scenes*. Since then, the method has become quite simple. First of all, outline edges are generated on the 3D object. This can be implemented on the CPU by checking for edges that join one triangle that is facing the camera and another that is not, similar to how silhouette edges are generated for creating shadow polygons. If an edge is determined to require an outline, it is drawn by a call to OpenGL. Obviously, this is expensive for models with many polygons, such as the 40,000 polygon model used here. It also caused a problem which can be seen in Figure 1.

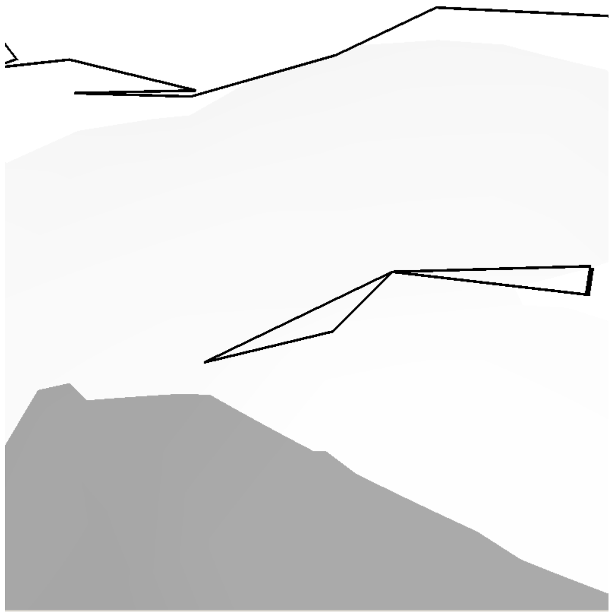
Figure 1: Poor selection of outline edges from the simple software method.

Thankfully, outline edges can easily be drawn using graphics hardware. First of all, culling is enabled on the front faces of the model, as well as line mode for the back faces. A wire mesh is then drawn with thick black lines. Finally, the code draws the front-facing polygons as normal. The result can be seen in Figure 2. This works very well, but it also has some minor issues. For example, imperfect meshes may have outline edges in incorrect places. The 40,000 polygon Stanford bunny model used here has a few such areas. This problem can be solved by careful work of a mesh creator, and is considered beyond the scope of this paper.

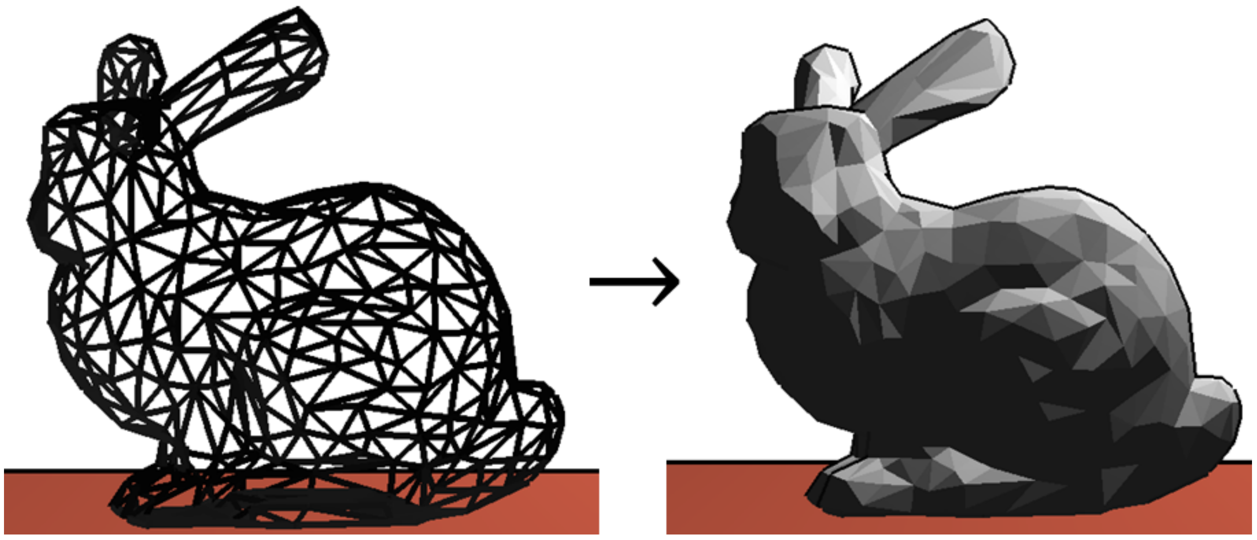
Figure 2: A simple rendering of the Stanford bunny with outline edges drawn by the GPU. A model with only 1,000 polygons is shown to show the method more clearly.

Polygon shading is also simple with modern graphics hardware and software. The implementation defines basic GLSL vertex shaders, which do calculations such as the normal and position of a pixel, and fragment shaders, which do the bulk of the coloring work. First, the fragment shader calculates the Lambertian shading at each point, $\vec{n} \cdot \vec{l}$, where $\vec{n}$ is the unit normal at a point on the mesh, and $\vec{l}$ is the unit direction to the light; therefore $-1 \leq \vec{n} \cdot \vec{l} \leq 1$. Areas where $-1 \leq \vec{n} \cdot \vec{l} \leq 0$ are all shaded the same color because they will all be shadowed from the light. Values of $\vec{n} \cdot \vec{l} > 0$ can be used to choose a color along discreet increments, from a dark color to a lighter color. This is what makes the rendering look "cartoonish." For example, for a simple black, white, and grey shader, the fragment code looks like:

```
if (nl > 0.5)
      color = vec4(1.0,1.0,1.0,1.0);
else if (nl > 0.0)
      color = vec4(0.33,0.33,0.33,1.0);
else
      color = vec4(0.0,0.0,0.0,1.0);
```

Although this is procedural shading, it can be thought of as sampling a simple one-dimensional texture based on $\vec{n} \cdot \vec{l}$, as shown in Figure 3.

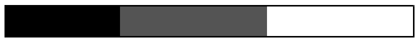
Figure 3: A black and white one-dimensional texture, with pixel positions from 0 to 1.

Shadow polygons were used to create shadows from a single point light. In areas that are in shadow, the color returned by the shader needs to be the darkest possible color. In order to accomplish this, an attribute variable was passed into GLSL from the C++ rendering code. In retrospect, a uniform variable (which is similar, but not allowed to be set per vertex) could have been used to prevent unnecessary instructions in the vertex shader. Results are good, as can be seen in Figure 4, except on one PC used, where some polygons near silhouette edges tended to flicker. This can only be seen when the scene is animated, and may be due to a small error in the shadow rendering code.

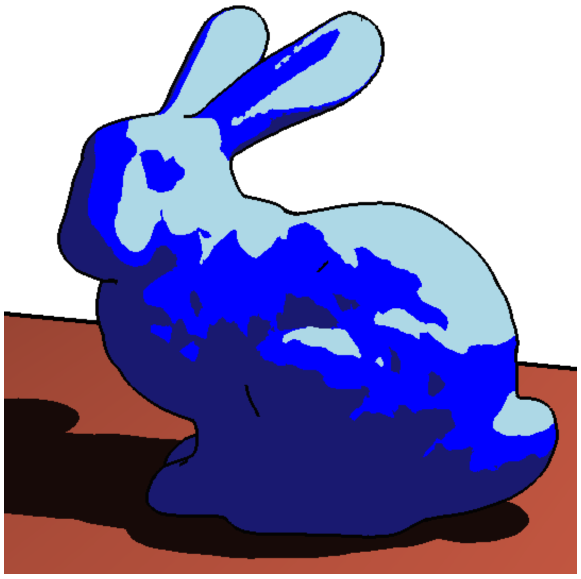

Figure 4: Self-shadowing with a basic toon shader.

X-Toon Shading

One problem with traditional cel shading is that the shading is not view dependent, which prevents effects like specular highlighting and backlighting, which are frequently seen in cartoons. To allow these effects to be rendered in a scene that still looks toon-shaded, Barla et al. propose adding a *level of abstraction* to the traditional toon shading model. This can be represented by expanding the one-dimensional texture into two dimensions. This texture can then be sampled at a point $(\vec{n} \cdot \vec{l}, D)$. Barla et al. show how using different methods to select D can generate the desired effects.

It can be surprisingly difficult to import simple textures for sampling into OpenGL. A great deal of time was spent staring at screens similar to those shown in Figure 5.

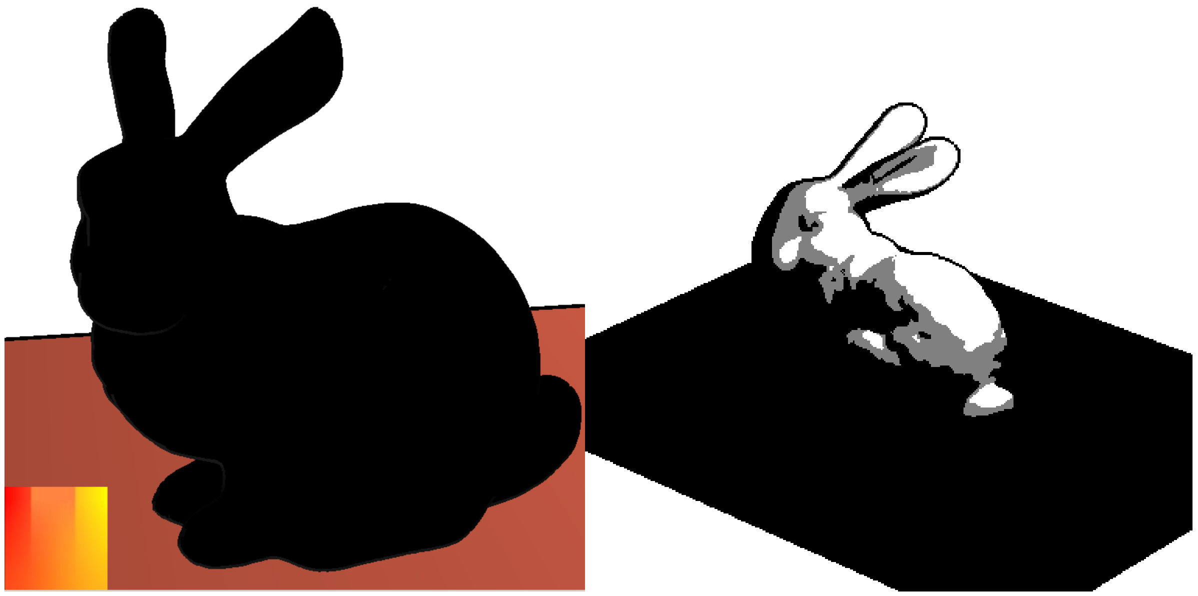

Figure 5: Texture not loading properly into OpenGL, and texture loading and being applied to all surfaces (even though the shader was not).

Initially, the texture was only available in the shader when stored as a mipmap, a specific kind of texture designed to be efficient for displaying traditional textures at different ranges. Although this may not be ideal for sampling purposes, it works. Later it was found that in order to load it as a simple texture for sampling, several texture parameters set for mipmapping had to be changed. Now the code correctly loads and samples a simple 2D texture.

The first effect that this new dimension can create is a simple depth of field effect. This can be generated by selecting D based upon the pixel's distance to the camera. In cartoons, objects in the background are frequently rendered with little to no detail. Barla et al. suggested picking D according to the following rule:

$$D = 1 - \log(\frac{z}{z_{\min}})/\log(\frac{z_{max}}{z_{\min}})$$

The authors also propose other ways to pick D for a slightly different depth effect. For example, it's possible to simulate a blurring effect for both objects too close to the eye and too far away from it for a more "realistic" depth of field effect. However, here we pick D according to this rule for purposes of a simple demonstration. Implementing this in a GLSL shader yields good results, as shown in Figure 6.

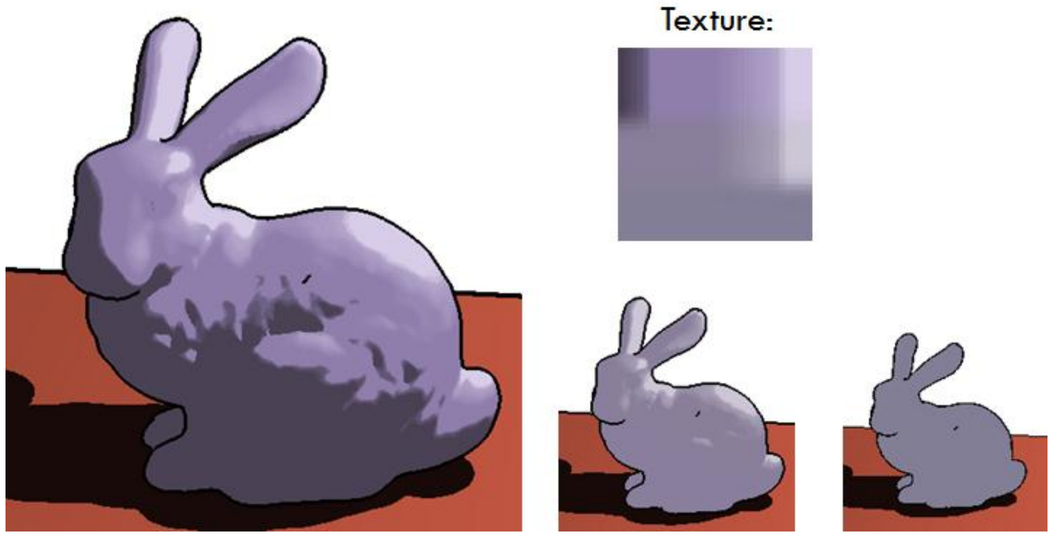

Figure 6: Depth rendering of the Stanford bunny, using the texture shown above the image. As the camera gets further away from the mesh, pixels from lower areas of the texture are sampled, causing a blurring effect.

Next, calculating D in a different way, once again inside a GLSL shader, can be used to generate more realistic, but still cartoon-like highlights. In this method, D is calculated as follows:

$$D = |\vec{v} \cdot \vec{r}|^s$$

where $\vec{v}$ is the normalized view vector, $s \geq 1$ is a user-defined "shininess" value related to the material properties, and $\vec{r}$ is the normalized reflected light vector determined by

$$\vec{r} = \vec{l} - 2 \cdot (\vec{l} \cdot \vec{n}) \cdot \vec{n}$$

where $\vec{n}$ is the unit normal at the point. This yields the results shown in Figure 7.

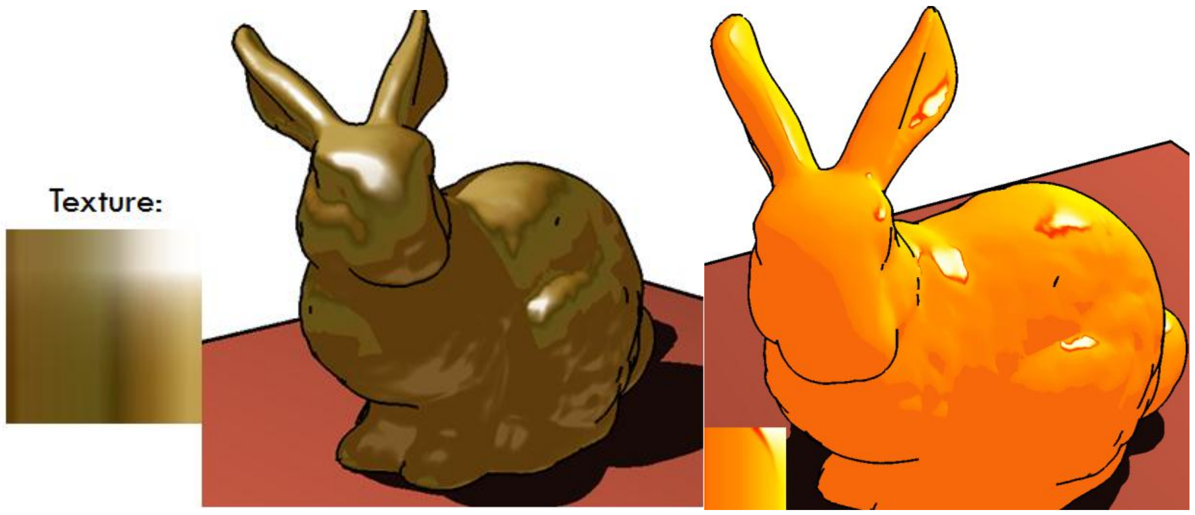

Figure 7: X-Toon textures used to generate specular highlights. Notice how the texture can be modified to customize the look of the highlights and shading for a more stylized look.

Finally, choosing D using the following equation

$$D = |\vec{n} \cdot \vec{v}|^r$$

where $r \geq 0$ can be adjusted to change the magnitude of the affect, and $\vec{n}$ and $\vec{v}$ are the same as defined before. This allows us to make an interesting effect on near-silhouette polygons that can be seen as simulating back-lighting on the object. Although it has little basis in reality, except perhaps for simulating a cartoon-like bloom effect, near-silhouette edge highlighting can generate some interesting and beautiful effects. See Figure 8 for results.

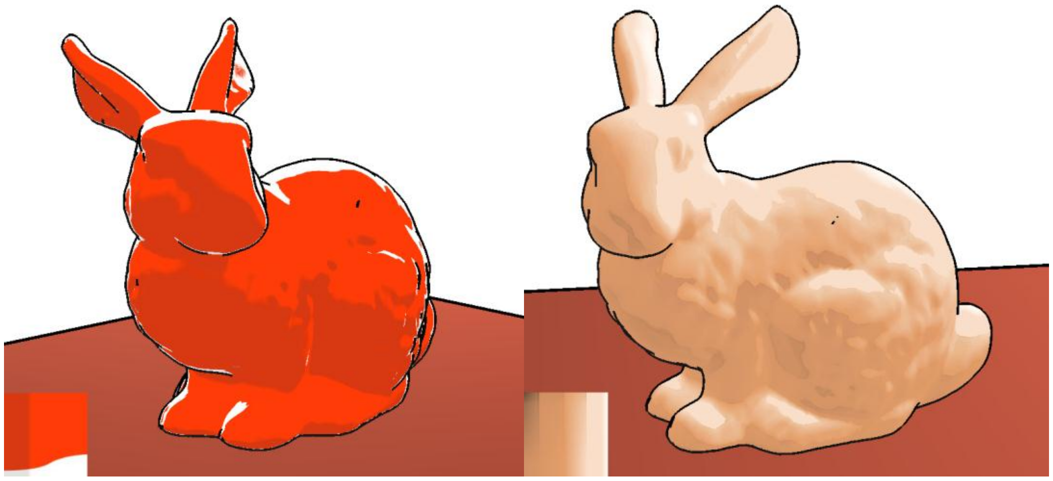

Figure 8: Near-silhouette edge highlighting effect. In the first image, edges are simply highlighted white as if a bright light were causing a bloom effect. In the second image, near-silhouette edges use a blurrier part of the texture, causing the edge to look fuzzy while the facing-size looks crisp.

Finally, this rendering technique can use any 2D texture as a shading source, although currently only very simple .ppm ASCII-encoded files are supported. This means that almost anyone with a simple image editor that supports .ppm files, such as Paint.NET with a downloadable extension, can create X-Toon textures. Although, as shown in Figure 9, just because an artist can use any image as a texture, doesn't mean that one should.

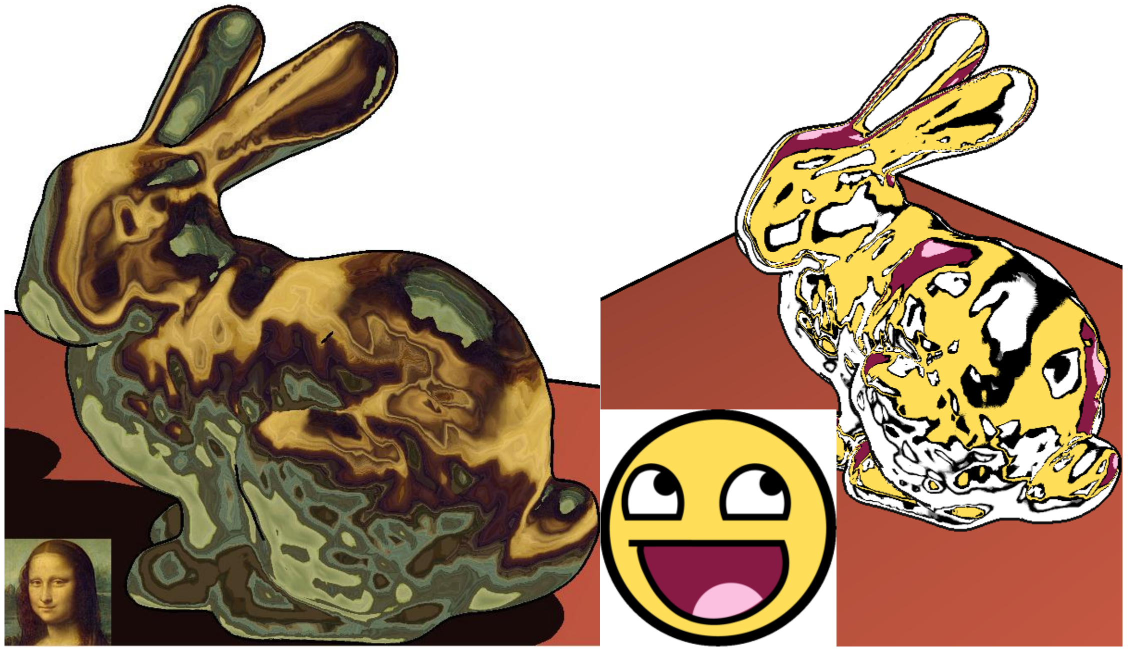

Figure 9: Strange (but expected) results when using the Mona Lisa as a highlighting texture and the "Awesome Face" as a backlighting texture.

To see more results, please look at this YouTube video that demonstrates most of the textures described in *X-Toon*.

Stylized Highlights

The model for highlights examined in this paper for different stylistic techniques involves using the Blinn method over the Phong specular highlights. Instead of using the light vector reflected across the normal of the point, the inspected vector is halfway between the reflected light vector and the vector pointing at the camera. If the dot product of this halfway vector, H, and the normal, N, at this point, P, is greater than 1 – ε, then the point is highlighted. An example rendering using this type of highlight can be seen below.

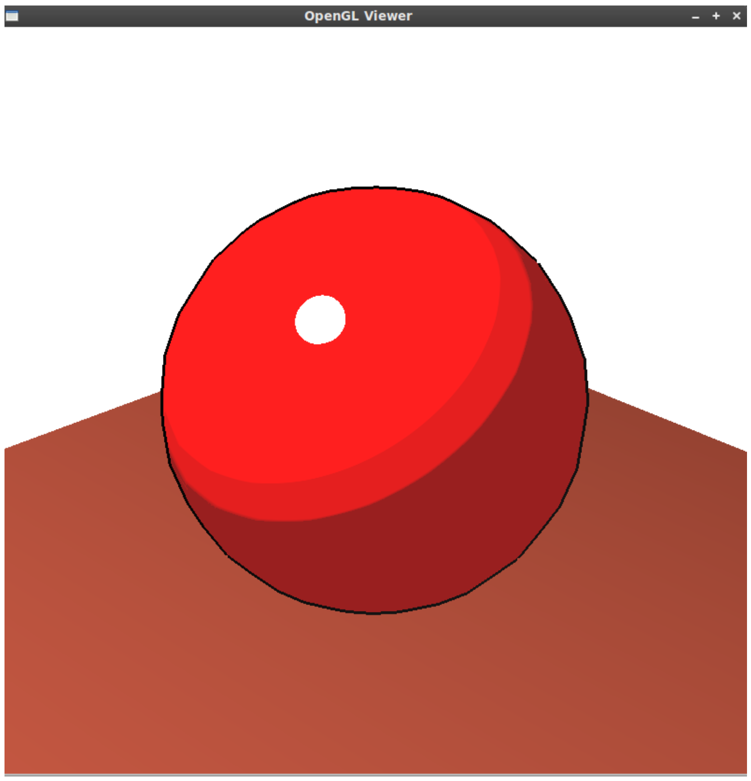

Figure 10: Basic specular highlight

There are many different operations that can be performed to modify this highlighted area. One of the simplest involves modifying the value of ε. The size of the highlighted area is directly proportional to the value of ε ($0 < \varepsilon \leq 1.0$).

The many other transformations are based in modifying H in some way. To modify this vector, the tangent plane to P has to first be calculated. Since we are given a point and the unit normal to that point, solving for the Cartesian equation ($Ax + By + Cz = D$) is trivial. Once this is calculated, a random point on the plane is chosen. The unit vector from P to this random point is designated as one of the linearly independent unit vectors, *du*. By taking the cross product of *du* and the normal, we can get the other unit vector *dv*.

Affine Transformations

To translate the position of the highlight on the object, given two real numbers, x and y, the function below can be used.

$$H' := H + x\boldsymbol{du} + y\boldsymbol{dv}$$

$$t(H) := \frac{H'}{||H'||}$$

Rotation can be achieved by rotating the *du* and *dv* vectors about the normal by a specified angle, followed by reevaluating the vector H using the following formula where du^r and dv^r represent the rotated axes.

$$H'' := H + (H \cdot \mathbf{du})\, du^r + (H \cdot \mathbf{dv})\, dv^r$$

$$r(H) := \frac{H''}{||H''||}$$

Directional scaling can be done in any direction, though it is defined in the du direction in the following segment. For a number x ($0 < x \leq 1.0$), the equation for this method is

$$H''' := H - x * (H \cdot \mathbf{du})\, \mathbf{du}$$

$$s(H) := \frac{H'''}{||\, H'''\, ||}$$

This function only scales the highlight in the *du* direction, by combining this functionality with rotation, the highlight can be effectively scaled in any direction.

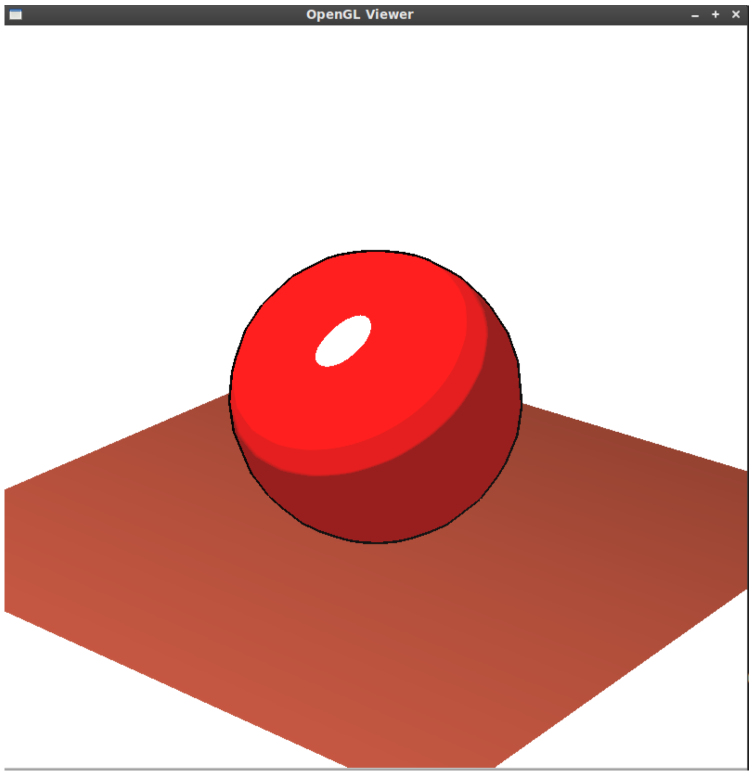

Figure 11: Directional scaling of highlight

Stylized Transformations

In addition to the affine transformations listed above, there are a few stylistic transformations that can add more variety and character to the highlights, suggesting more details than the simple affine transformations.

The highlighted area can be split along both of its unit vectors using the following function, given two non-negative numbers *i* and *j*. The sign function simply returns -1 if the number is negative, and 1 otherwise.

$$H^{\wedge} := H - i * sign(H \cdot \mathbf{du}) * \mathbf{du} - j * sign(H \cdot \mathbf{dv}) * \mathbf{dv}$$

$$split(H) := \frac{H^{\wedge}}{||H^{\wedge}||}$$

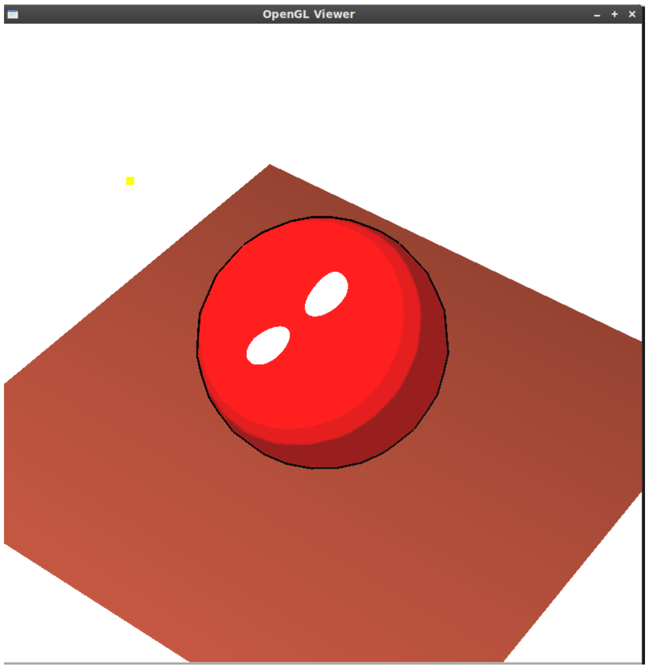

Figure 12: Combined split and directional scaling function.

The highlighted area can also be squared, creating hard edges. Given an integer exponent, n, and a positive number x (0.0 ≤ x ≤ 1.0), the function is calculated as follows

$$V' := (H \cdot du) * du + (H \cdot dv) * dv$$

$$V := \frac{V'}{||V||}$$

$$\theta := min(\, cos^{-1}(\, V \cdot du\,), cos^{-1}(\, V \cdot dv\,))$$

$$k := sin(\, 2\theta\,)^n$$

$$H^{\wedge\prime} := H - k * x * V$$

$$sqr(H) := \frac{H^{\wedge\prime}}{||\, H^{\wedge\prime}\, ||}$$

These different functions can all be combined to create a wealth of different effects.

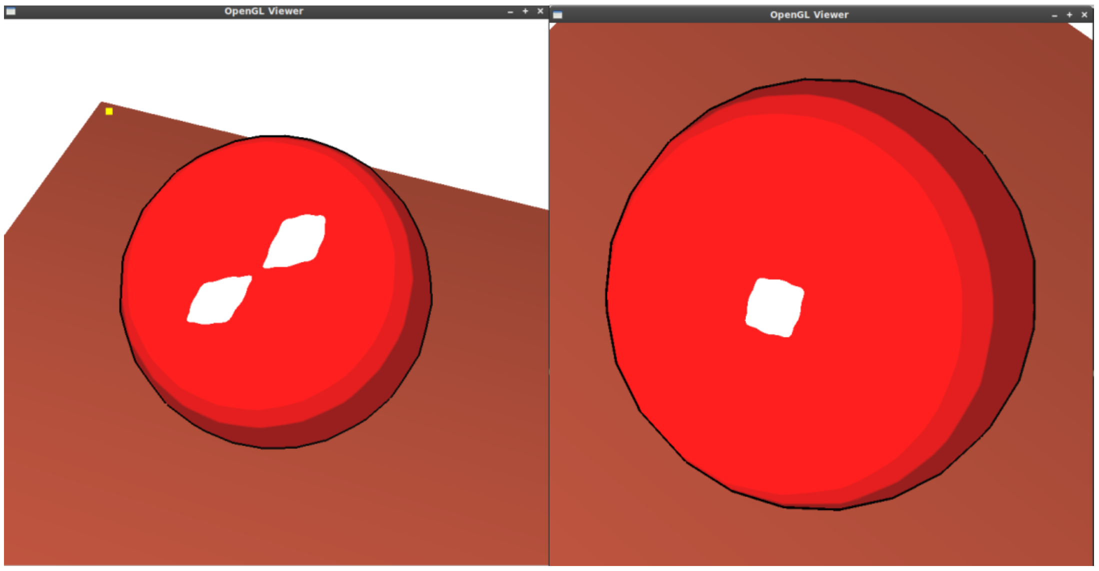

Figure 13: Left: Directional Scale, Split, and Squaring function. Right: Squaring function.

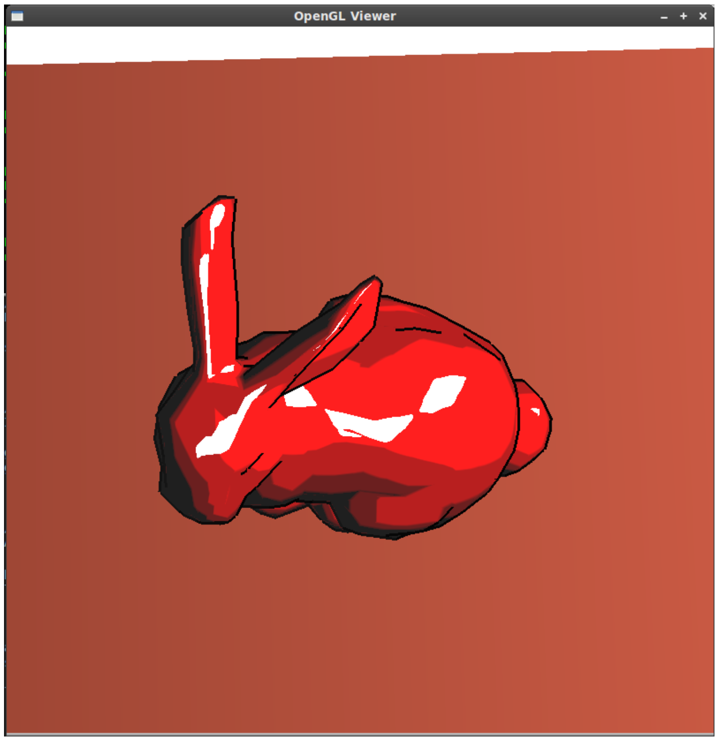

Figure 14: Stanford bunny with stylized highlights

X-Toon & Stylized Highlights Combined Effects

The calculation of D for use in the y direction for the texture look up is normally dependent on the view, but can be generated in any number of methods. By combining the stylized highlight functionality with the 2D texture map, interesting highlights and textures can be created. This is due to the fact that all of the modifications that the stylized functions perform only change an intermediate vector. The following function demonstrates how the two functions can be combined, given the modified vector, V, the normal to the point, N, and the exponent, s, to modify the value further.

$$D := |\, sin\,(\,V \cdot N\,)\,|^{S}$$

This combination of effects removes the extreme stylized look from the results shown below, but also allows for a range of different effects due to the combination of both techniques.

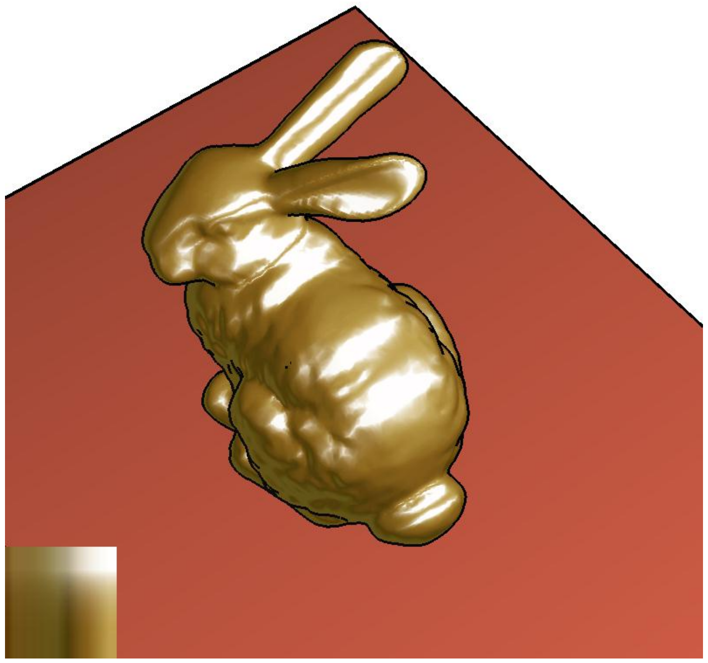

Figure 15: Combined effects, splitting and directional scaling of highlight area using 2D texture.

Results

Overall, the implementation runs smoothly. Currently, the test cases are lacking what models were used. The object files could have been chosen better, and if they had correct normal values, then there would have been many more examples. Everything runs smoothly in real time, as it was mostly implemented using GLSL, except for animated shadow polygons on a detailed model, which is not the focus of this paper. Using parallel processors or a shadowing method that does not rely on the CPU could alleviate this problem.

Unfortunately, two of the functions detailed above for the stylized highlights were not fully implemented. The square function certainly makes the edges straighter, but they do not scale properly, becoming oblique parallelograms rather than rectangles. The rotation function also does not work, there was difficulty in actually implementing rotation about a central vector, and the multiple attempts did not result in the desired effect.

Conclusion

Despite some problems with the stylized renderings, the overall goal of this paper has been achieved. These different effects make toon shading look vastly improved over older versions.

Over the course of this project, the combined workload was about 60 hours, split evenly between the two participants. Adam focused on the implementation of the 2D textures, and Sean implemented the different stylized highlights. Both participants worked equally on the initial cel shader, including the outlines and general purpose shading. Much of the time for both contributors was spent simply debugging the code.

References

Anjyo, Ken-ichi., and Hiramitsu, Katsuaki. 2003. Stylized highlights for cartoon rendering and animation. IEEE Computer Graphics and Applications 23, 4, 54–61.

Barla, P., Thollot, J., and Markosian, L. 2006. X-Toon: An Extended Toon Shader. In International Symposium on Non-Photorealistic Animation and Rendering.

Decaudin, Philippe. 1996. Cartoon-Looking Rendering of 3D-Scenes. Research Report INRIA #2919.