Non-photorealistic Rendering for Cartoon Artstyle

ETHAN SCHEELK, Macalester College, USA

Non-photorealistic rendering (NPR) can be used to recreate certain hand-drawn artistic styles, often called "cartoon" shading and in less frequent cases cel-shading. What are the challenges to implementing NPR versus physically-based rendering in the modern day? The methods to find and render edge outlines are implemented in OpenGL and GLSL and discussed.

CCS Concepts: • **Computing methodologies** → **Non-photorealistic rendering**.

1 INTRODUCTION

A graphic artist for video games or rendered video for movies, shorts, etc. may make the stylistic choice of non-photorealistic rendering (NPR), such as to emulate a hand-drawn or even a painted style [Hertzmann and Perlin 2000]. These NPR methods include but are not limited to cel-shading, limited color shades, strokes or coloring to recreate brush strokes, and the bleeding of colors [Decaudin 1996; Mitchell et al. 2002].

This paper focuses on "natural" edge outlines. These are those edges from object outlines alongside areas within the mesh which would naturally feature accent lines when drawn as a cartoon. For example, as shown in **Figure 1**, the outline of the rabbit and the box are drawn in thick outlines while also featuring less significant internal lines a cartoonist or comic book line artist would typically add.

We have also implemented a discretized light shading model. Notice in **Figure 1** the discrete, discontinuous steps in darkness/lightness on the rabbit.

These edges can be found by performing edge detection algorithms on the normal and depth maps. These edges are then layered on to a final image which is shaded with a limited color palette.

These effects were were written in OpenGL and in the OpenGL Shader Language (GLSL).

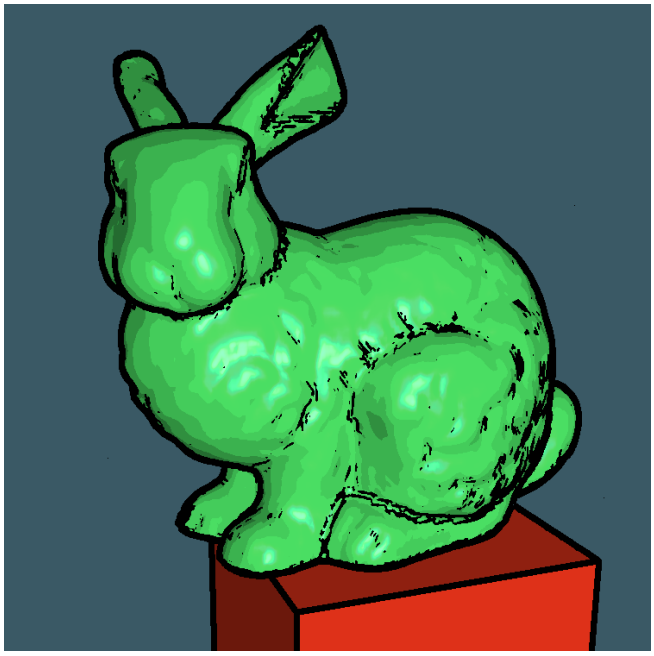

Fig. 1. The final result of our cartoon graphics pipeline.

2 RELATED WORK

The primary inspiration for this paper is the time-staying capability of cartoon graphics within animation and video games. The methods we implement are highly inspired by the toon-graphic pipeline from a 1996 paper on the topic [Decaudin 1996]. The author of this paper discusses his implementation of mutli-pass rendering in order to find the depth and normal maps, find edges with a convolution, then adding them together. Additionally, the author implements shadows and occlusion lighting. The author used this method to implement a cel-shaded cartoon appearance in an animated movie. On a consumer-grade workstation, each frame took six minutes to render.

There are other papers that use very similar techniques and suggest more artistic potential fromd detecting edges [Hertzmann 1999].

3 METHODS

We will discuss our methods for implementing this graphics pipeline in OpenGL. We use a basic Blinn-Phong shader model modified to sample material shade from an artist-created texture, allowing for fast customization of the discrete shading levels.

Then, we will discuss multi-pass rendering, an essential technique to our rendering. With multi-pass rendering, the scene is rendered multiple times to record and track different effects. We render and show the normal map, depth map, and find the edges of each of those with an edge-detection convolution.

Then, we add the edges together using a filter and threshold function. Finally, we layer them on top of the Blinn-Phong lighting to create the final scene.

3.1 Lighting

Blinn-Phong shading is ideal for a less realistic art style. We use it as it is typically described [Decaudin 1996]. However, we make some slight adjustments. Those adjustments are firstly to add an exponential specular term, which controls the "roughness" of the material. Secondly, we alter it slightly to sample from a texture for the discretized lighting. The lighting is implemented within a

Author's address: Ethan Scheelk, escheelk@macalester.edu, Macalester College, 1600 Grand Avenue, Saint Paul, Minnesota, USA, 55105.

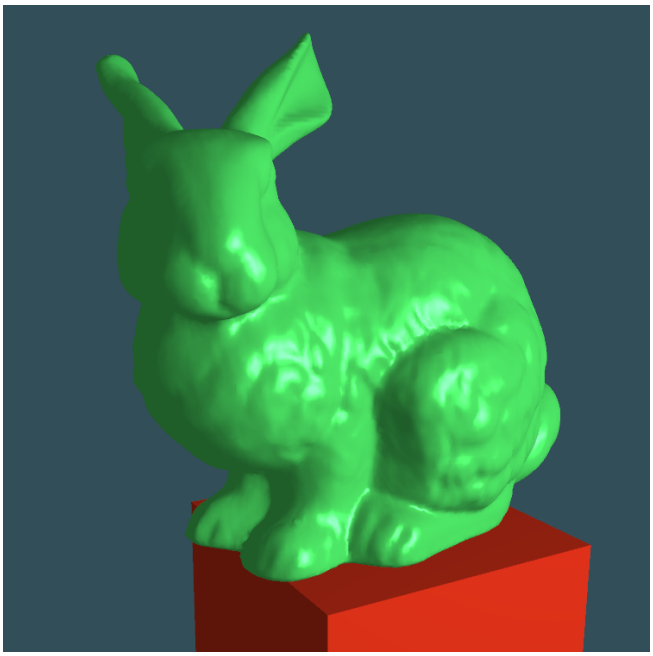

Fig. 2. Rabbit with basic Blinn-Phong rendering.

fragment shader. The Blinn-Phong method of lighting is as follows:

$$\begin{aligned}\text{color} = {} & I_{\text{Global ambient light}} * K_{\text{Material ambient coefficient}} \\ & + \sum_{\text{Source}}^{\text{All Lights}} \Big(I_{\text{ambient}} * K_{\text{Material Ambient Coefficient}} \\ & + \max(\mathbf{L} \cdot \mathbf{N}, 0) * I_{\text{diffuse}} * K_{\text{Material Diffuse Coefficient}} \\ & + \max((\mathbf{H} \cdot \mathbf{N})^x, 0) * I_{\text{specular}} * K_{\text{Material Specular Coefficient}} \Big),\end{aligned}$$

where L is the vector from the fragment to the light source, N is the interpolated normal vector from the vertex shader, H is the halfway vector between L and the vector to the eye, and x is some chosen specular exponent that controls for the "smoothness" of the reflection.

Material color parameters are passed as uniform variables before calling the shader. As shown in **Figure 2**, lighting varies smoothly and there are subtle specular highlights based on light and camera positions.

3.2 Limited Color Palette

The limited color palette can be implemented simply within a Blinn-Phong shader by replacing the usual zero (dark) to one (bright) with a texture call within the shader. For example, **Figure 3** shows a potential "lighting ramp" that can give the effect of toon lighting to an object.

In our GLSL fragment shader, this is a simple change: `NdotL` becomes `texture(diffuseRamp, vec2(NdotL, 0.0))`. Likewise, for the specular highlight, `pow(HdotN, x)` becomes `texture(specularRamp, vec2(pow(HdotN, x), 0.0))`.

Fig. 3. Toon lighting ramp

Fig. 4. Rabbit with toon lighting color ramp.

Here, `diffuseRamp` and `specularRamp`, for diffuse and specular lighting respectively, are of type `sampler2D`, which takes a texture passed into the shader. This can be seen in **Figure 4**. Observe the discretized and discontinuous color shading.

Different ramps can be given for the diffuse and specular coloring.

3.3 Multi-pass Rendering

As a preliminary to our work to finding the edges in the scene, we must first render and save other pixel information first. Though methods exist to find edges in 3-Dimensional space [Hertzmann 1999; Markosian et al. 1997], we chose to work purely within screen coordinates and with fragments. Edges in 3D require extra tests to gauge visibility and 2D edges are likely just as effective and are easier to find.

Written in our C++ file we have a function `renderPass()`, shown in Algorithm 1. Before each call of `renderPass()`, we must bind the relevant framebuffer. A framebuffer acts as a render location alternative to the screen, where the rendered image is saved to a connected texture instead. It is useful to think of this as an artist using several canvasses, each with their own layer of information. A later step in the process might reference a previous canvas, so the artist can take a look back at what was done before.

Algorithm 1 Render Pass

```
function RenderPass(Shader)
    Clear OpenGL Color Buffers
    Bind Uniform Variables to Shader
    Enable Shader
    for each model do
        Set Color
        Draw Model
    end for
    Draw anything else
```

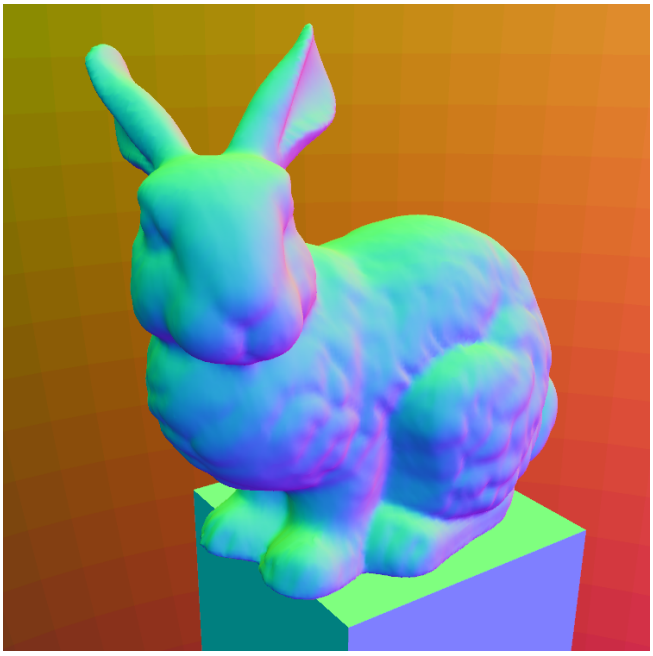

Fig. 5. The rabbit with color mapped to the normal vector at each location.

In this case we have multiple framebuffers, each with its associated texture. In order to produce our final result, we have framebuffers for each of the following results:

(1) Normal Map,
(2) Normal Map Edges,
(3) Depth Map,
(4) Depth Map Edges, and
(5) Combined Edges.

The following sections show how we fill these framebuffers.

3.4 Filling the Framebuffers

3.4.1 Normal Map. The Normal Map is a representation of the normal for each fragment saved instead as the color of the pixel. Given normal vector N, the color to map is $0.5 * N + 0.5$. Negative numbers will be placed in the first half of each range $(0, 0.5)$ and the positive numbers will be mapped to $(0.5, 1)$.

Fig. 6. The depth map of the rabbit, where objects that are farther away from the camera are colored brighter. The near plane is 0.1 and the far plane is 20.0

So, x, y, z components will instead be equivalently saved as r, g, b. Something that is green is pointing up, for example. Consider **Figure 5** for an example. The rabbit is inside an inside-out sphere.

First we bind the normal frame buffer and its associated texture. Then we run a render pass to save to the texture.

3.4.2 Depth Map. During the screen transformation, x, y, z coordinates are transformed to being screenspace coordinates, where the z component becomes equivalent to the depth to each vertex, which is interpolated for each fragment as well. In GLSL, the fragment position is accessible from the global variable `gl_FragCoord`. These coordinates have not been converted to normalized device coordinates, $(-1.0, 1.0)$.

Additionally, due to the perspective stretching of the frustum, depth is non-linear.

We convert depth to linear, where $z = gl\_FragCoord.z * 2.0 - 1.0$. Let d_{near} be the distance from the camera to the near plane and d_{far} the distance from the camera to the far plane. Then,

$$\text{linearized depth} = \frac{2.0 * d_{\text{near}} * d_{\text{far}}}{d_{\text{near}} + d_{\text{far}} - z * (d_{\text{far}} - d_{\text{near}})}.$$

Finally, we get the depth = linearized depth$/d_{\text{far}}$.

We return the color vector in the depth fragment shader with this linearized normalized device coordinate depth in every r, g, b position, making it greyscale, as seen on **Figure 6**.

The choice of near and far plane will depend on the application. Linearizing depth helps retain information at farther distances, as normally those become compressed.

Depth is saved in its own texture via its framebuffer.

Table 1. The neighborhood of each fragment X.

A	B	C
D	X	E
F	G	H

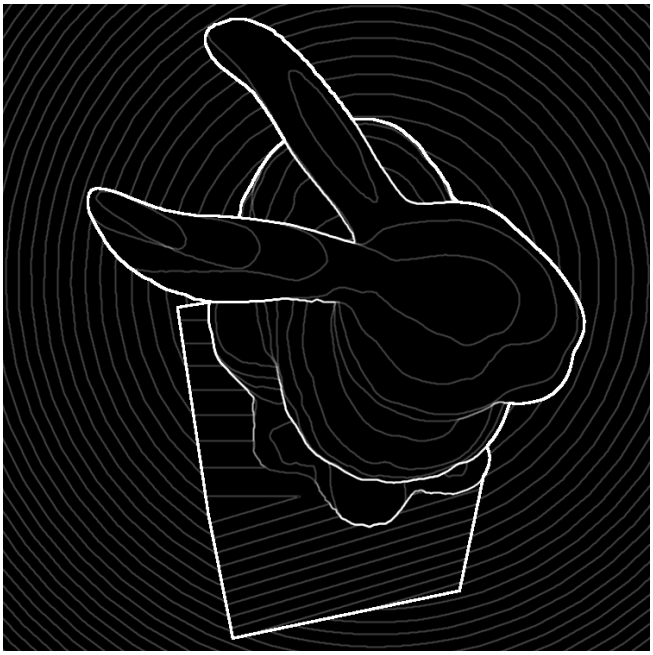

Fig. 7. The edges as calculated by the convolution on the depth map.

3.4.3 Edge Detection. The method of edge detection is an improved version of the Sobel filter [Decaudin 1996; Saito and Takahashi 1990]. This is also the same edge detection filter used in the game *Borderlands 2* by Gearbox Software.

The Sobel filter is a first order differential operator and will detect discontinuities of zero order, that is, discontinuous locations. For example, in the depth map, it will be discontinuous when there is a gap in distance between two objects [Decaudin 1996].

This differential operator can be performed with a convolution, where the values in neighboring pixel are used to calculate the gradient. This is performed in a depth edge fragment shader with a `sampler2D` of the depth texture that was previously saved.

The gradient,

$$g = \frac{1}{8}(|A - X| + 2|B - X| + |C - X| + 2|D - X| +$$

$$2|E - X| + |F - X| + 2|G - X| + |H - X|)$$

where $A - H$ are the 8 neighbor pixels of X, as shown in **Table 1**.

The calculated gradient g is then saved to a depth edges texture, as shown in **Figure 7**.

Due to large depth differences between the rabbit's ears and its body, a line is drawn there. The secondary lines also exist because there is a natural gradient in depth across the image. For demonstration and debugging purposes, the depth edges are multiplied by 128 to make the edges more visible.

Fig. 8. A case where the depth convolution fails to detect an edge.

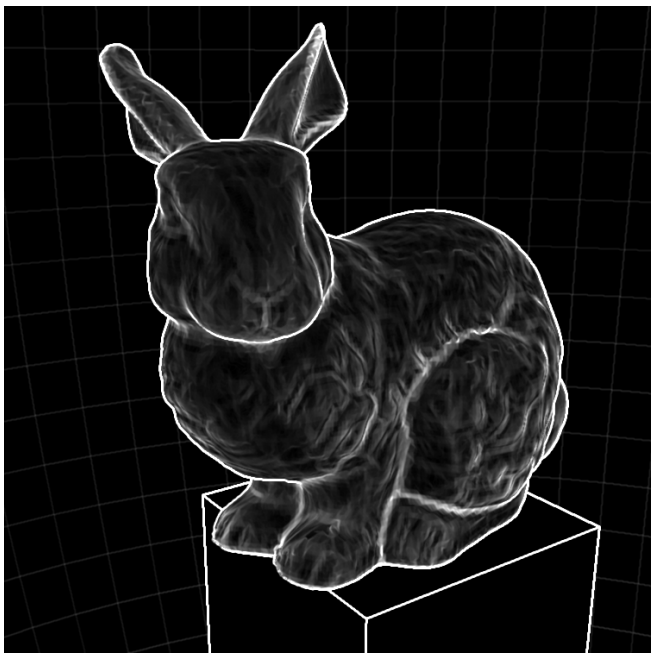

Fig. 9. The gradient of the normal map.

However, there is a problem. Some edges do not get detected because there is not a large enough gradient. This is shown in **Figure 8**. The nearest corner and edges of the cube are not drawn.

In order to remedy this, we will also perform a convolution on the normal map.

The logic is identical, except g is calculated for x, y, and z. The texture lookup is equivalently on the normal map texture as well. Then, the color provided to the texture is the average of g_x, g_y and g_z. **Figure 9** shows the gradient of the normal map multiplied by 16 for visibility.

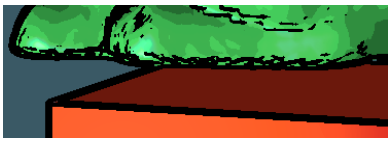

Fig. 10. The final result but only normal edges are considered in the final combine process. The result is splotchy and inconsistent around the silhouette.

Though the normal map does detect most of the edges, it proves to be too inconsistent to use alone without depth map edges. See **Figure 10**. The normal edges catch most of the desirable edges that the depth edges miss, as well as some other desirable mid-surface edges, such as around the rabbit's mouth as legs.

We now have two sets of edges that we will combine together.

3.4.4 Combining Edges. For filtering the edges, we found the maximum and minimum gradient, g_{max} and g_{min} respectively, in the 3x3 region around each pixel, and used the following equation:

$$p = \min\left(\left(\frac{g_{max} - g_{min}}{k_p}\right)^2, 1\right),$$

where $k_p \in (0, 1)$ is some coefficient that will alter the sensitivity for drawing an edge. A smaller number will add more lines. For our desired effect, we found $k_{\text{depth}} = 0.003$ and $k_{\text{normal}} = 0.0125$ to be best. See **Figure 12** for examples of how the choice of K impacts the final image.

The usage of g_{min} and g_{max} also results in a thickening of the lines by a few pixels.

Once we have found p for each edge type, we add them together. Further, we reject any fragment with value less than 0.75 and set it to be black.

This gives our combined edge images as seen in **Figure 11**. Even with our careful choice of K coefficients, some points in the background sphere are detected. However, when it comes to the final image, they are not noticeable.

We now have a robust silhouette alongside "messier" inside-the-object lines. There is the question of if the same internal line segments would still exist on different models that are perhaps smoother, or if they will become too jagged on a model with fewer polygons.

The last step now is to draw these edges on top of the regularly-colored scene.

3.5 Combining Everything Together

We now pass the combined edges as a texture into our Blinn-Phong shader as previously discussed. First, we retrieve the edge color at the same fragment coordinate. Next, if that edge brightness is nonzero, we instead set the final fragment color as black instead of the regular color.

Our final result is shown in **Figure 13**.

4 RESULTS

The result is good and we are content with the final appearance. The silhouette outlines are thick and smooth and we have internal outlines as well. The internal outlines were not an original desire from the paper that spawned this idea [Decaudin 1996], but they add some extra detail. However, these internal lines are not typically continuous or the smoothest. Around the bottom side of the leg, for example, the line is patchy. These results will also be dependent on the triangulation of the model.

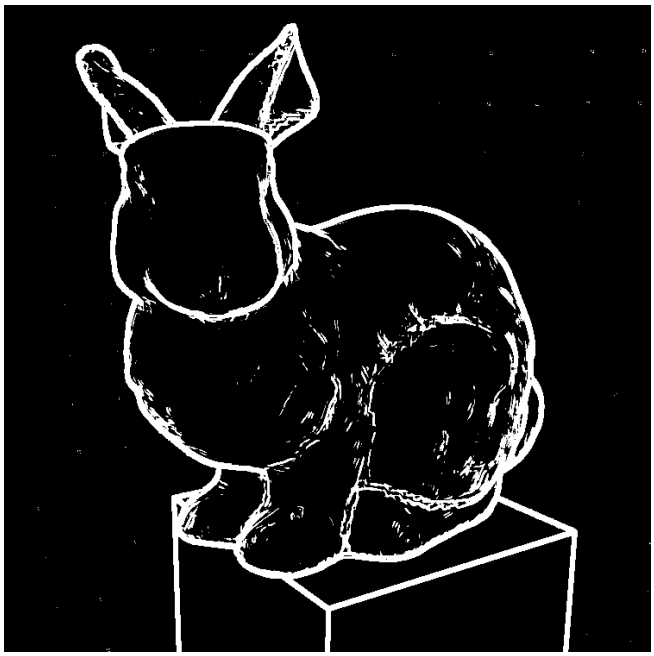

Fig. 11. The edges combined and filtered.

5 FUTURE WORK

We recognize that there is always more fun things to do, but here is what we are still interested in continuing to work on.

5.1 Deferred Rendering

We would like to implement deferred rendering to our graphics pipeline. This would group all possible rendered objects into their own buffer, wherein the same algorithms can be used to find edges. This would avoid speed varying linearly negative with the number of objects in the scene, since each object calls the shader for each render pass. The convolutions are not the fastest processes, so it would be best to avoid a scaling problem.

5.2 Edge Aliasing

Further, there are questions of the aliasing of the edges. Several sources recommend rendering on a larger resolution and scaling down [Decaudin 1996] or to perform a larger convolution. Additionally, there will be other antialiasing methods that could be used to reduce the number of jagged edges on the drawn scene.

5.3 Improve Edges Combine

The coefficients in the shader to combine edges approach being magic numbers. We arrived to these results by repeatedly guessing

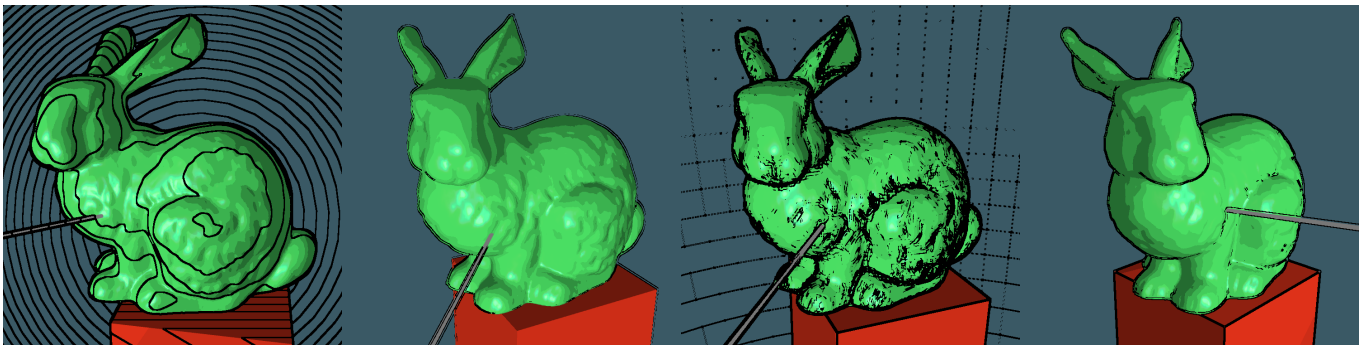

Fig. 12. Examples of poor choices of K. The first is K_{depth} of 0.001 and the second of 0.005. The third is K_{normal} of 0.005 and the fourth of 0.025.

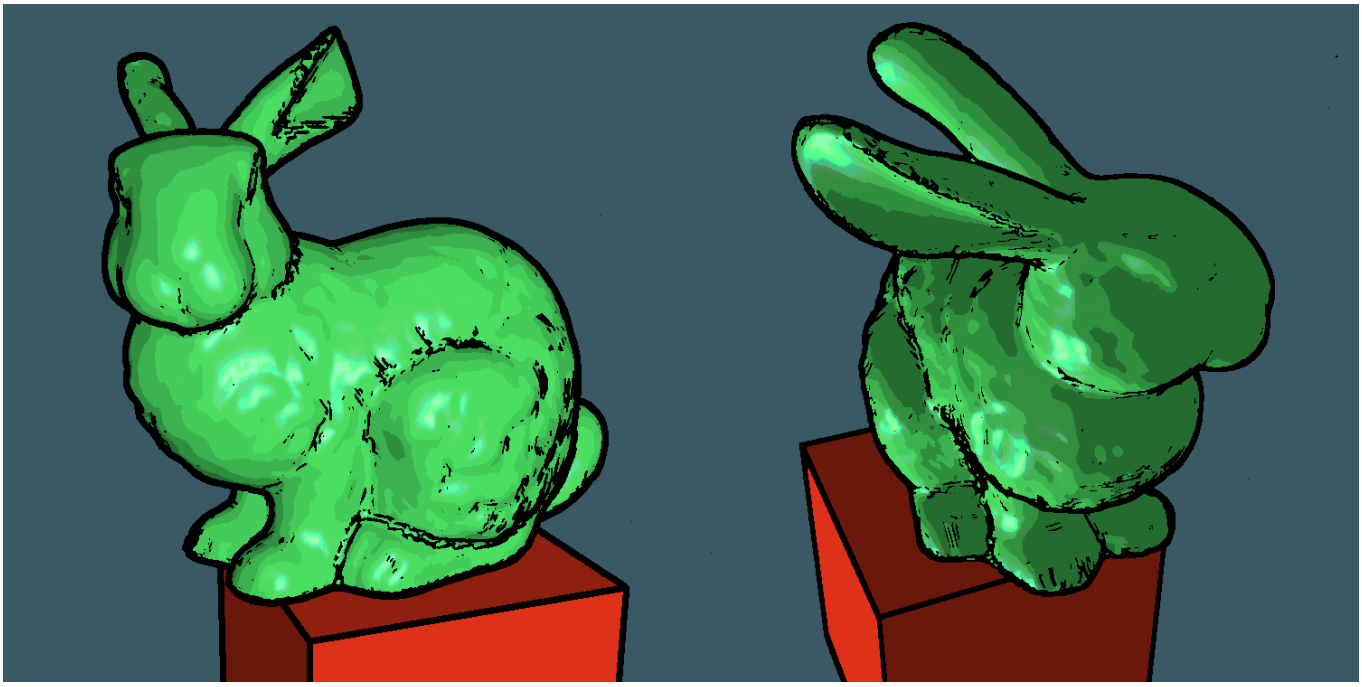

Fig. 13. The final result of our cartoon graphics pipeline from two different angles.

and checking. We would like to investigate alternative methods of combining edges.

Additionally, we would like algorithmically find the curves describing the edges [Hertzmann 1999]. This would allow us to apply the edges with variable thickness, paint texture, and other unique artistic qualities. Specifically, thick silhouette outlines with fainter internal outlining interests us.

5.4 Implement More Toon Graphic Effects

There are a variety of other staple graphic effects we could implement. These include but are not limited to shadows, light occlusion, and unique materials.

6 CONCLUSION

Through mild modification of previous techniques, we are able to outline inner-edge detail of objects within our scene alongside the silhouettes. Due to the advancement in computing power, this process is much faster than it used to be. The technique of calculating edges from depth and normal maps proves to be effective and adaptable. Further techniques can be used to enhance the appearance in more unique, artistic directions.

REFERENCES

Philippe Decaudin. 1996. *Cartoon Looking Rendering of 3D Scenes.* Research Report 2919. INRIA. http://phildec.users.sf.net/Research/RR-2919.php

Aaron Hertzmann. 1999. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In *SIGGRAPH 99.* ACM Press, Chapter Course Notes.

Aaron Hertzmann and Ken Perlin. 2000. Painterly Rendering for Video and Interaction. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering* (Annecy, France) *(NPAR '00).* Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/340916.340917

Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. 1997. Real-Time Nonphotorealistic Rendering. In *Proceedings of SIGGRAPH 97 (Computer Graphics Proceedings, Annual Conference Series).* 415–420.

Jason L. Mitchell, Chris Brennan, and Drew Card. 2002. Real-Time Image-Space Outlining for Non-Photorealistic Rendering *(SIGGRAPH '02).* Association for Computing Machinery, New York, NY, USA, 239. https://doi.org/10.1145/1242073.1242252

Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Comput. Graph.* 24, 4 (sep 1990), 197–206. https://doi.org/10.1145/97880.97901