Interactive Rendering of Characters with Cel Shading and Outlines

Justin Hung, Alex Riddle

Rensselaer Polytechnic Institute (RPI), Troy, NY

Abstract

In this project, we present basic methods of loading and rendering character models, cel shading (known otherwise as toon shading), and outlines. In particular, we discuss extending the traditional diffuse lighting technique to include cel shading using GLSL shaders, a method of rendering outlines using the OpenGL stencil, rim lighting (which often accompanies cel shading), and halftone shading. We also present an interactive Graphical User Interface (GUI) that allows users to expriment and interactively adjust properties of these rendering techniques in real time.

Keywords: Cel Shading, Rendering Outlines, Non-Photorealistic Rendering, Rim Lighting, Interactive GUI

Contents

Email addresses: hungj2@rpi.edu (Justin Hung), riddla@rpi.edu (Alex Riddle)

1. Introduction

Cel shading, known also as toon shading, is one of the simplest non-photorealistic rendering techniques used to give characters a cartoon look. This technique can be found everywhere from animated movies and TV shows to video games. Some recent examples of video games using this technique are The Legend of Zelda: Breath of the Wild (Nintendo, 2017), and Genshin Impact (miHoYo, 2020).

Cel shading is often used in conjunction with outlines and rim lighting, as in the above example of Link from Breath of the Wild (BOTW). We sought to implement these rendering techniques to come up with convincing renders of characters like Link that are very close to the original works, but with minimal code. We also implement these techniques in a dynamic way that can be altered in real time to allow for adjustment of the number of properties such as the number of cel shading zones and the outline thickness.

We believe that knowing how these rendering techniques work, as well as getting more practice with shaders and OpenGL in general, will be useful for those of us trying to enter the video game industry.

2. Related Works

[1] Raul Reyes Luque (2012) explains in "The Cel Shading Technique" the fundamentals of cel shading and outlines. He also includes an edge detection algorithm for creating the outlines and a color quantization algorithm.

[2] Abhishek Kumar (2014) describes in "Toon Shader Methods for Cartoon-like Rendering" three methods of toon shading. Method 1 is based on diffuse lighting, Method 2 uses a depth parameter for colors, and Method 3 uses a 2D texture to assign colors.

[3] Bert Freudenberg (2004) explains in "Real-Time Halftoning: Fast and Simple Stylized Shading." how to implement a system to create halftone screens from images that resemble pen-and-ink drawing styles in a fast way for real-time environments.

[4] Won-Ki Chung (2006) discusses in "Rendering Style: Toon Shading - Electrical Engineering and Computer Science" the concept of the cel shading technique and how it is used in various video game titles. such as Nintendo's 2002 title, The Legend of Zelda; The Wind Waker.

3. Loading and Texturing Character Models

3.1. The Wavefront OBJ File Format

In order to load character models and textures we chose to use the standardized OBJ and MTL file formats from Wavefront. Wavefront OBJ files are a type of file format for geometry definitions that was first developed by Wavefront Technologies around 1990. Today, OBJ files are still widely used and trivial to parse as they are text-based. Along with the OBJ file, there is the Wavefront MTL file format for the material definition, which is also simple and text-based. For the sake of our project, we decided to focus on these as they are the simplest and finding example models from sites like models-resource.com provides plentiful example models.

The OBJ file format consists of a list of "commands" such as **v** for vertices, **vn** for normals, **vt** for texture coordinates, and **f** for faces, among many others. These particular commands are used to form a single mesh. Larger, complex character models will often have groups and/or separate objects. These are denoted by the **g** and **o** commands, respectively, and are used to indicate when a new mesh is being constructed. Each mesh can be assigned a material with the **usemtl** command, which changes the active material for the current mesh. To render these complex models correctly, we will need to break up the model into its meshes and corresponding materials, so that we can update the OpenGL texture unit and shader properties for each mesh being rendered.

To facilitate this, we implemented a simple OBJ file parser with the following algorithm:

1. Push an empty mesh object to the list of meshes.
2. Collect the "RawFaces" of the current mesh, which consist of a list of indices for the vertex, texture coordinate, and normal (3 of each per face as we only support triangles).
3. If a new object or group is declared, go to step (1).
4. After all meshes are pased, iterate over all "RawFaces" and push the actual vertex, normal, and texture coordinate vectors to the corresponding mesh object, such that the indices become relative to that mesh.
5. Load any material files denoted by `mtllib` and map them to an index for meshes to reference.

3.2. Loading textures

To load the textures for each material we utilize a tiny, header-only library called `stb_image`, which is in the public domain and is available from GitHub. This library supports most image formats that we can use for easy loading of textures of any format. When we load these textures into OpenGL we ensure that the *u* and *v* coordinates are set to repeat to handle the cases where texture coordinates are negative (which is often the case). For texture mapping, we simply tell OpenGL to wrap the S and T coordinate values using `GL_REPEAT` to handle negative texture coordinates. We then simply bind the current texture to the `TEXTURE0` texture unit, and add a uniform sampler variable inside the fragment shader. To render the texture on the model, we use the `texture2d()` GLSL function to sample fragments of the texture map using the new sampler variable.

3.3. Rendering the model with OpenGL

On the OpenGL side, we use our .obj loader to divide the mesh into separate buffers for the vertices, normals, and texture coordinates. This is opposed to having all of these next to each other contiguously. Rendering then looks like the following:

```
glDrawElementsBaseVertex(GL_TRIANGLES,
 mesh.numIndices, GL_UNSIGNED_INT,
 (void*)(sizeof(unsigned int) * mesh.baseIndex),
 mesh.baseIndex);
```

where `mesh.baseIndex` refers to the base index of where the current mesh vertex attributes are in the OpenGL buffers. The base indices are now needed since each mesh is relatively indexed; i.e., the index buffers for each mesh all start at 0, hence why we also need to add `baseIndex` at the end to tell OpenGL to add the value of `baseIndex` to each corresponding value.

This approach allows us to selectively render each mesh of the larger model in a clean way, allowing us to switch the material and texture sampler in the shader between each render. This is required to render the more complex models with different materials and textures.

4. Cel Shading

4.1. Overview of cel shading

This method of shading divides a model's surface into (usually two, but sometimes more) discrete zones based on how illuminated each spot on the surface is and lights each zone uniformly. This technique is used to approximate the style of old fashioned painted animation cels where distinct colors were used to convey value and depth.

Figure 1: A comparison of diffuse lighting (left) and cel shading (right) on Link from BOTW

The level of shading, or the "cel factor," corresponds to the dot product of the light factor and the surface normal for a given pixel. (Five different shading regions depending on the sampled surface luminance can be shown in the following two images.)

4.2. Extending diffuse lighting to cel shading

In typical diffuse lighting, the graph of light intensity given the angle between the light vector and the surface normal is the following smooth curve from the $cos(x)$ function:

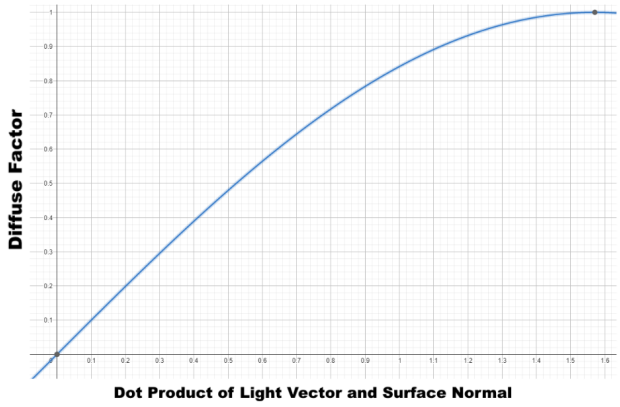


Figure 2: Diffuse lighting using traditional cos(x)

With cel shading, however, we simply divide this curve into discrete "zones" of shade, such as the following:

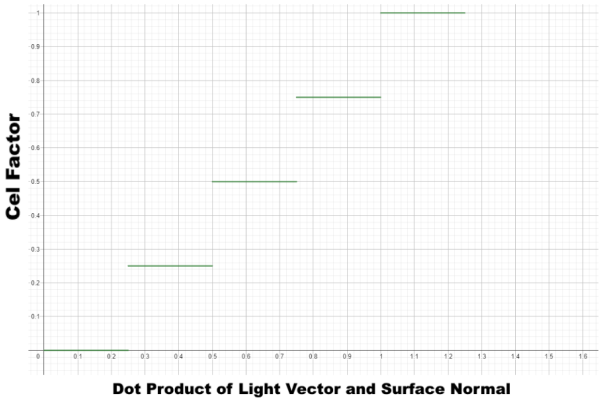


Figure 3: Diffuse lighting with steps/zones (cel shading)

If we already have diffuse lighting implemented in our fragment shader, then we can modify our diffuse factor to achieve this effect in a fairly straightforward way:

```
float celFactor = 1.0f / celSteps;
diffuseFactor = ceil(diffuseFactor * celSteps)
 * celFactor;
```

Note that the use of `ceil` means that characters will be rendered generally brighter as we round up to the nearest step/zone. `floor` could be used to yield the opposite effect; characters would generally be rendered darker.

5. Outlines Using the Stencil Buffer

5.1. The Scaling Probem

In concept, in order to create an outline for a 3D model, one would render a scaled up version of the model first as a solid color like black, then render the actual model on top of it. One of the immediate problems presented by simply scaling the original mesh proportionally is that a proportionally larger mesh will only be able to properly surround the original if the mesh is strictly convex. In the case of more detailed meshes with appendages like limbs or more cavernous details like skin wrinkles, parts of the original mesh will stick out from within the larger mesh.

(Bold and Brash (SA Style). SarkenTheHedgehog, 2019) (Edited) (1)

To solve this problem, the mesh must be scaled up disproportionately. That is, every face of the mesh should be extruded outward at a consistent distance away from the original position and scaled with respect to the normals of each of the face's vertices.

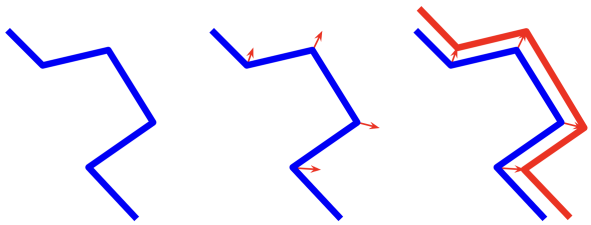


Figure 4: Disproportionate face extrusion

The simple solution to this problem that we employ involves the GLSL vertex shader. We can simply alter `gl_Position` to extrude out the vertices using the normal multiplied by the desired thickness, as can be seen below:

```
gl_Position = wvp *
  vec4(Position + Normal * thickness, 1.0);
```

5.2. Using the OpenGL Stencil Buffer

Now, the first step in our OpenGL rendering code for the outlines is to render the model as normal. But before we do this, we enable the stencil buffer with `GL_ALWAYS` such that, wherever we have fragments of our character rendered, the corresponding pixels are given stencil values of 1. The OpenGL code is as follows:

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xff);
glStencilMask(0xff);
RenderModel();
```

To demonstrate this, we will use Fox from Nintendo's Star Fox Adventures (2002), as the model is complex and works well for this method. At this point, the stencil buffer looks like the following (in the case of Fox):

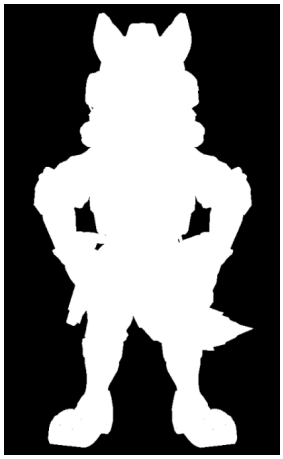


Figure 5: Stencil representation of Fox from Star Fox Adventures. Nintendo, 2002.

Next, we render the model again, but this time using the outline shader. We also change our stencil function to only pass where we have 0s in the stencil, i.e., where we do not have fragments from the first render of the model. This allows us to render only the scaled model and avoid rendering over the original model. We also disable the depth test here to ensure that we render the scaled model in front of the original model.

```
glStencilFunc(GL_NOT_EQUAL, 1, 0xff);
glStencilMask(0x00);
glDisable(GL_DEPTH_TEST);
RenderModel(m_outlineShader);
```

Now, if we fix our fragment shader for the outline to always return white, we get the following for Fox:

Figure 6: Stencil Outline render of Fox from Star Fox Adventures. Nintendo, 2002.

5.3. Coloring the Outline

Our method of coloring the outline is basic. We simply sample the texture in the fragment shader, invert the color, and darken it. This approach is unlikely to work well for darker textures, as you would be left with an even darker outline color. The fragment shader is as follows:

```
vec3 outlineColor = vec3(1.0f)
  - texture2D(gSampler, TexCoord0).xyz;
outlineColor *= 0.05f;
FragColor = vec4(OutlineColor, 1.0f);
```

To add contrast for darker textures, we could simply check if the texture started dark and increase the brightness, which lends itself to future work.

6. Rim Lighting

6.1. Overview of Rim Lighting

Rim lighting is a technique used in photography where the subject is illuminated from the back such that the very edges of the subject glow, but the majority of the actual subject remains in shadow. In the context of cel shading and similar rendering techniques, however, rim lighting is frequently used to supplement cel shading and provide extra definition and contour to otherwise flatly shaded models.

Figure 7: A comparison of a model without rim lighting (left) and a model with rim lighting (right)

6.2. *Implementation of Rim Lighting*

With typical diffuse lighting, the intensity of the light is at its peak when the light is perpendicular to the surface. Rim lighting is the opposite: triangles that are directly facing the viewer are dark, while triangles facing away are more illuminated. The normal lighting equation still applies. The GLSL fragment shader code is as follows:

```
float rimFactor = max(0.0f, 1.0f - dotProd);
color = DiffuseColor * pow(rimFactor, intensity);
```

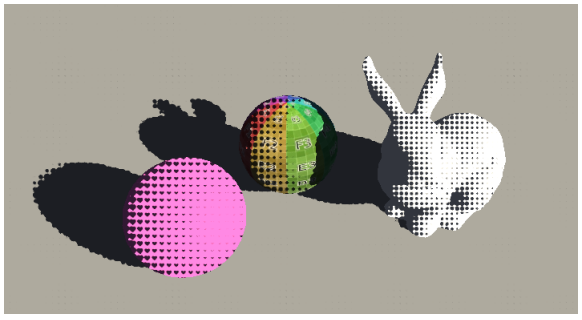
Intuitively, we "invert" the diffuse lighting formula in the sense that we make surfaces brighter as the light and normal vectors begin to tilt away from each other, hence the `1.0f - dotProd`. Then we raise the resulting factor to the specified rim lighting intensity to find the resulting additive color in the lighting equation.

7. Halftone Shading

7.1. *Overview of Halftone Shading*

Halftone shading is another rendering technique like cel shading. Halftone shading, like cel shading, typically uses two distinct levels of shadow intensity or color to convey depth. In contrast to cel shading, halftone shading is almost exclusively used in the printmaking industry when creating images comic books, newspapers, or other media that require images to be made with many small dots. Furthermore, halftoning is often used to create a "retro" look, as it particularly mimics the poor quality and limited color set of old printers.

In contrast to cel shading, traditionally, halftone shading zones are determined by sampling the entire image (with the model within it) at specified points on a grid. A circular zone surrounding each point on the grid is created, with a radius determined by the luminance sampled at that particular point (darker points get larger circles, while brighter points get smaller circles or a circle with a radius of 0). Then, only the pixels within the circles at each point are colored as the darker of the two tones.



(Ronja's Turotials - Halftone Shading. Ronja, 2019) (2)

7.2. *Our approximate implementation*

While we were not able to perfectly recreate this effect, we were able to approximate it using some basic techniques. The main drawback with this alternative is that the dots present are all uniformly sized. In this approximated version, there are three zones shaded by two tones: the most unlit region (region 1), the somewhat lit region (region 2), and the most lit region (region 3, which effectively equates to the absolute rim). Pixels in region 1 are effectively cel shaded with the existing technique. Pixels in region 2 are shaded only if they fall within a pattern of designated circles on a checkerboard grid. Pixels in region 3 are not shaded at all using the darker tone.

Figure 8: An approximated halftone shader

While this technically does not look the same way that most computer generated halftone shading does, we believe that our approximated version has the advantage of more closely mimicking actual printing dots, as shown below.

(10 Ways to Use Halftones in Photoshop, Retro Supply Co.) (3)

8. Interactive Graphical User Interface (GUI)

8.1. *Introduction to ImGui*

We use the simplistic "Dear ImGui" library (4), an immediate-mode user interface library that has a small code footprint and sees widespread use in the video game industry.

The term “immediate-mode” originates from the fact that the GUI is drawn immediately in a single frame. In our case, our App::Render function calls Gui::Render which calls the virtual OnGui functions of the base window objects. E.g., for a single frame, the following code may be executed to display a button:

```
if (ImGui::Button("Click Me!"))
  printf("Clicked!\n");
```

As soon as the button is clicked, the if statement activates. This simple way of creating UIs is extremely productive for experimental projects and debugging, such as in our case.

8.2. Overview of GUI in our application

The GUI can be toggled using the ’g’ key in our application. The GUI allows users to adjust the light color, ambient and diffuse intensity, as well as properties for cel shading, rim lighting, outlines, and halftone shading. Additional options include adjustments for the camera and scene where users can place objects and alter the camera movement speed as necessary.

8.3. Shader Properties Window

The shader properties window is where all shader properties can be tuned in real time. This allows for experiments and live debugging, as you are also able to recompile the shader code in real time and see any error messages (printed to console).

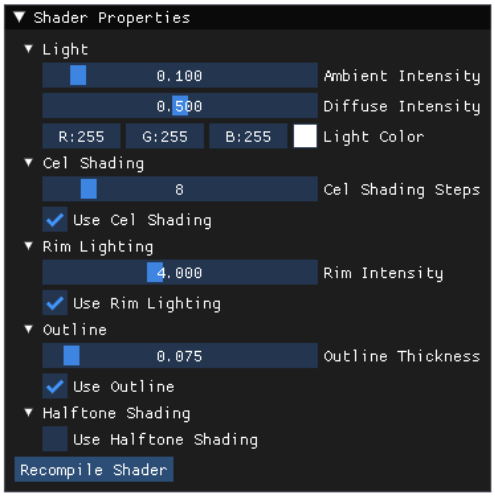


Figure 9: The shader properties window

9. Limitations and Future Work

Our method of outlines is very simple and does not work for models with sharp vertices. In particular, any vertices that create acute angles create jarring gaps (at the very least, this issue is less noticeable with obtuse angles).

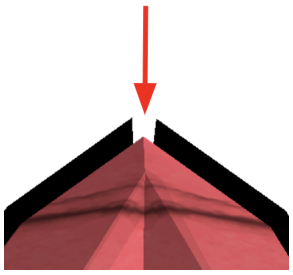


Figure 10: A gap in the outline at a sharp vertex

Our rim lighting is not quite accurate and appears to make characters brighter than they should be (currently, our rim lighting fades, similar to something lit via normal diffuse lighting, as another student pointed out during our presentation).

Figure 11: Bright Link from BOTW (Buggy Rim Lighting)

Our half tone shading is not fully accurate, but our approach is good enough to be used in most cases in various video games. We came up with a number of other unsuccessful methods of attempting this effect, such as the following, which more closely resembles a technique called ”dithering,” wherein every other pixel is either black or white to create the illusion of transparency:

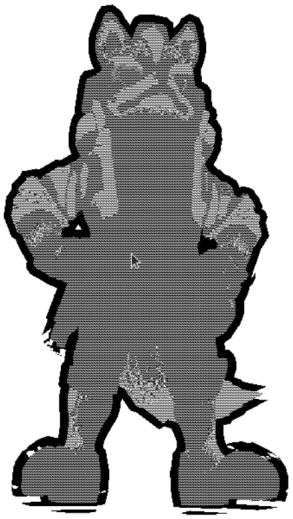


Figure 12: Failed halftone attempt that closely resembles dithering

10. Conclusions / Summary of Results

Cel shading itself is very simple to implement, but rendering outlines and rim lighting are more challenging to perfect. Rim lighting can be hard to get right, and may not be applicable in all scenarios but nonetheless provides an added level of detail to cel shaded models. Outlines need to be the correct color to provide enough contrast, and must extrude out from the model without artifacts (in our implementation we will often get artifacts from hard edges).

Our cel shading implementation is perfect, and we can render characters and get close results to the games from which these characters belong (e.g., Link from BOTW). Our simple vertex shader approach to outlines works for many models, except for those with hard edges, in which case the outline can very easily explode and create artifacts. Our rim lighting implementation appears buggy, and makes characters appear brighter than they should in most cases.

We have found that we can get decent results with these simple implementations for simpler characters, but there is much more work to be done to support a wider range of characters, particularly those that demand rim lighting and have hard edges.

11. Project Organization

This project took approximately 120 hours of work to complete across time spent researching, writing code, and writing this final report.

Member responsibilities:

- **Alex:** OpenGL renderer, model loading, controls, GUI, cel shading, stencil code for outlines, obj loader
- **Justin:** Rim lighting, halftone shading, model conversion, texture mapping, outline shader

12. References

References

[1] SarkenTheHedgehog. "Bold and Brash (SA Style)., https://www.deviantart.com/sarkenthehedgehog/art/Bold-and-Brash-SA-Style-823706977

[2] Ronja. "Halftone Shading." Ronja's Tutorials, 2 Mar. 2019, https://www.ronja-tutorials.com/post/040-halftone-shading/.

[3] RetroSupply Co. "10 Ways to Use Halftones in Photoshop." RetroSupply Co., https://www.retrosupply.co/blogs/tutorials/10-ways-to-use-halftones-photoshop.

[4] "An Introduction to the Dear Imgui Library - Blog - CONAN.IO." Conan C/C++ Package Manager Official Blog, 26 June 2019, https://blog.conan.io/2019/06/26/An-introduction-to-the-Dear-ImGui-library.html.

[5] "Simple GPU Outline Shaders." io7m.Com, https://io7m.com/documents/outline-glsl/d0e61.