Michael Caldwell
Abdulrahman Alshahrani
CS 354 Final Project Report
Misfit Toys I

Gooch Shading

In this project, we aimed to create non-photorealistic shading. Specifically, we wanted to make a Technical Illustration Shading Model. To achieve this, we needed to implement Tone-based Shading and Silhouetting.

Glitter

We originally planned to do the project on the ray tracing A1 codebase. After some research, we then tried to move over to the WebGL codebase from A4, but on Professor Abraham's suggestion, we looked into using OpenGL itself. We followed this learnOpenGL tutorial by Joey de Vries. This required a lot of boilerplate code, and Joey linked the project Glitter by user Polytonic as a starting place. Glitter is essentially the bare minimum needed to render a 2D window with OpenGL. What's nice is that Glitter already comes with many useful libraries like GLFW and Assimp.

From here we built out files to load and compile shaders, load and parse wavefront (.obj) models, and finally a few things like a movable camera. Most of these were based heavily on the source code provided by Joey, with modifications for Gooch rendering.

Getting the environment working took a lot of effort, but after reading through tutorials, as well as just testing things out, we have become much more comfortable with OpenGL, and Visual Studio.

Tone-based Shading

This part of the project is about creating a Tone-based shader based on the original paper by Gooch. We followed the same equations 2 and 3 from the paper. However, for the light direction, $\hat{\mathbf{l}}$, we decided to always make it the camera's right axis. This way, we can always see the full spectrum of shades on the object from all camera angles.

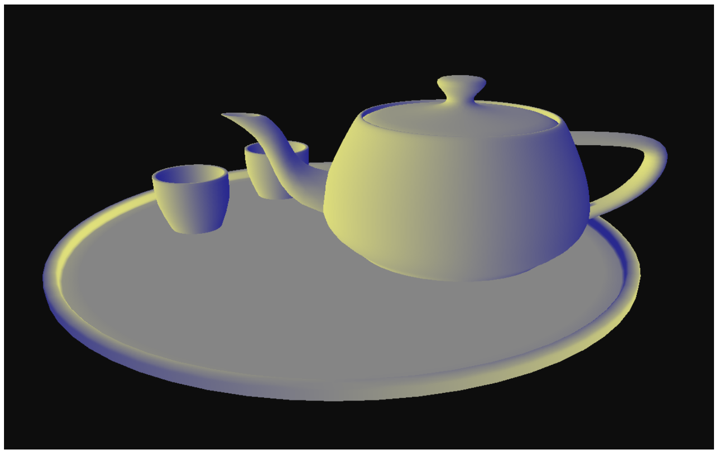
Here are Equations 2 and 3 we used from the paper:

$$I = \left(\frac{1+\hat{I}\cdot\hat{n}}{2}\right)k_{cool} + \left(1 - \frac{1+\hat{I}\cdot\hat{n}}{2}\right)k_{warm}$$

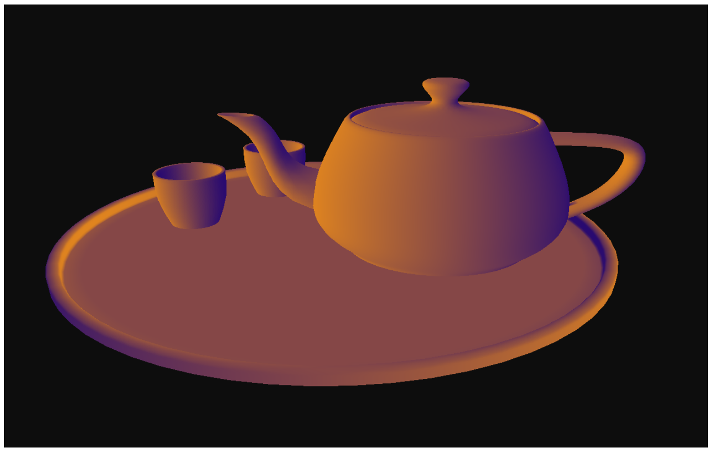
$$k_{cool} = k_{blue} + \alpha k_d$$
$$k_{warm} = k_{yellow} + \beta k_d$$

We achieved this by modifying the original Glitter project by adding a uniform for the 5 variables in the equation (**k_blue**: cool tone color, **k_yellow**: warm tone color, **alpha**: cool strength, **beta**: warm strength, **k_d**: object base color) and the camera's right axis, and then we modified the fragment shader to implement the equation.

Pictures



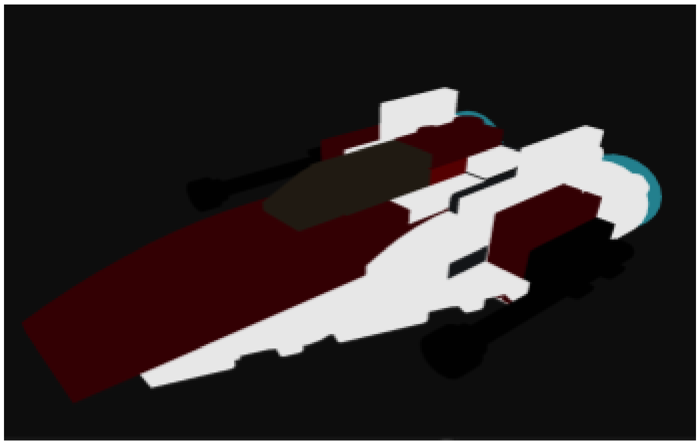
A render of a white teapot object with blue and yellow being the cool and warm colors, respectively.



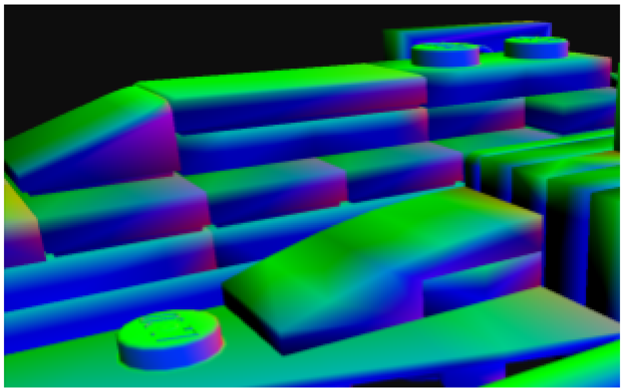
Another render of the same teapot but we changed its base color to red. As you can see, the tone spectrum almost matches the expected final tone spectrum in Figure 2 of the Gooch paper.

Silhouetting

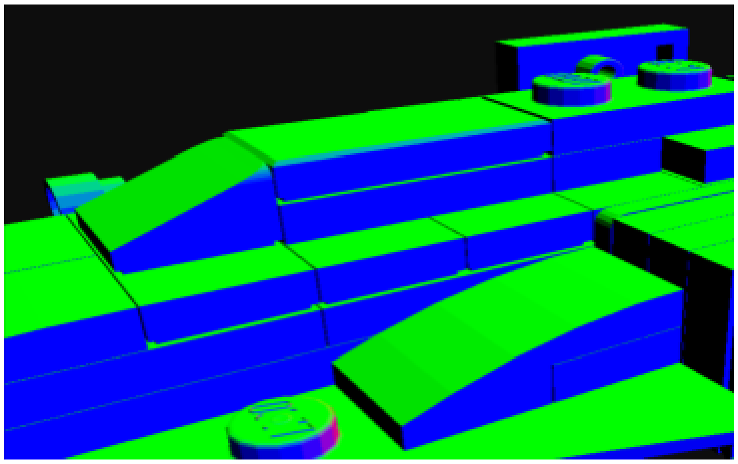
Silhouetting was where a large part of working time was spent. We originally started by trying to use the geometry shader to create outline primitives. This involved modifying the model loader to store adjacent lists for each poly. While we did get this working after implementing this Fast Adjacent Triangle algorithm, we found that many of the models we were using didn't have perfectly overlapping vertices, or were composed of submeshes, so we pivoted to post processing based edge detection.


Diffuse shaded Lego A-wing for reference

Crease Detection

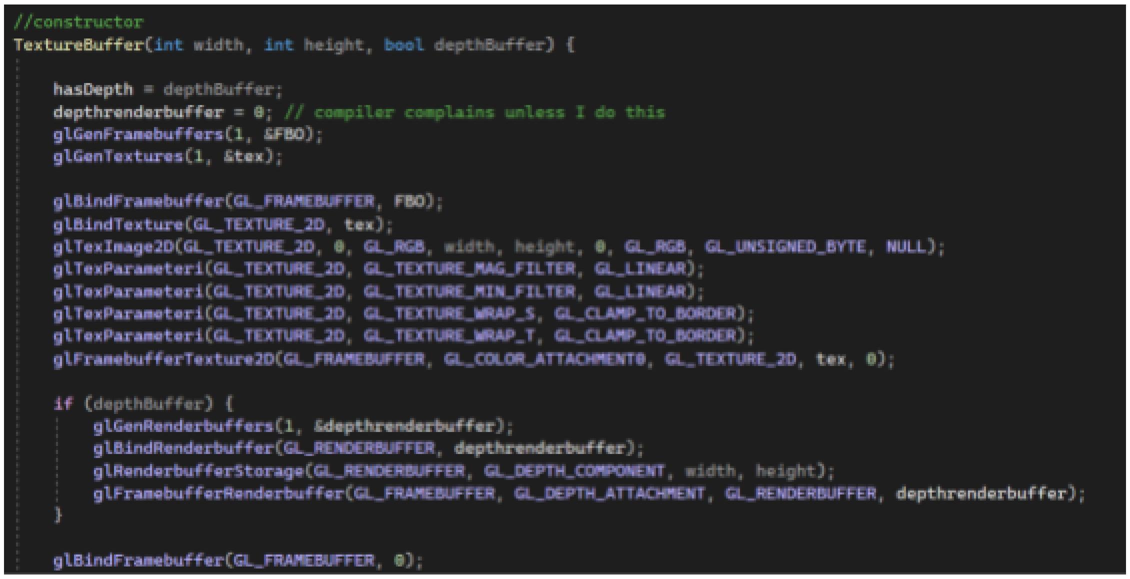

Model with per vertex normals

We planned to use face normals to detect angle changes between nearby surfaces to determine where any creases might be on an object. Unfortunately, our models only had per vertex normals, which resulted in the rounding of edges. So we changed our model loader once again. We now stored up the positions of each vertex for a given triangle, and used these to calculate a surface normal. This gave us the sharp edges that would work better for sobel edge detection.


Surface normal rendering

Postprocessing textures

Our approach to post processing was to use framebuffers and fullscreen polygons. We wrote specific shaders for each step of the post processing pipeline. This involved creating dedicated framebuffers that wrote their color values to a blank texture. This black texture could then be mapped onto a polygon that filled NDC space. From here, we had a 2D image, to use for sobel filtering.

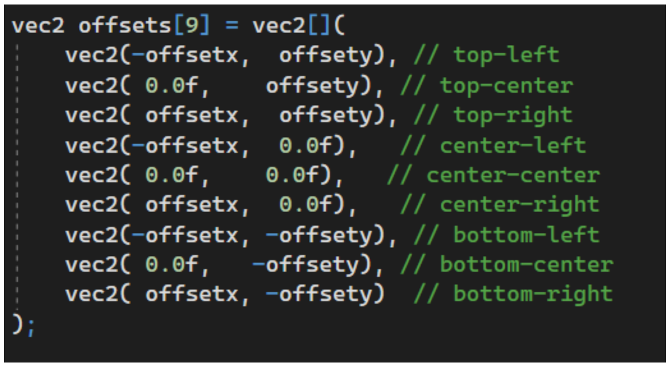


```
//constructor
TextureBuffer(int width, int height, bool depthBuffer) {

    hasDepth = depthBuffer;
    depthrenderbuffer = 0; // compiler complains unless I do this
    glGenFramebuffers(1, &FBO);
    glGenTextures(1, &tex);

    glBindFramebuffer(GL_FRAMEBUFFER, FBO);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex, 0);

    if (depthBuffer) {
        glGenRenderbuffers(1, &depthrenderbuffer);
        glBindRenderbuffer(GL_RENDERBUFFER, depthrenderbuffer);
        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthrenderbuffer);
    }

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

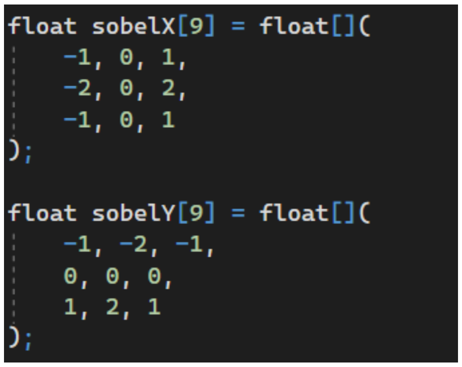
Class to create a framebuffer object with associated texture and renderbuffer

Normal Based Edge detection with Sobel Filters

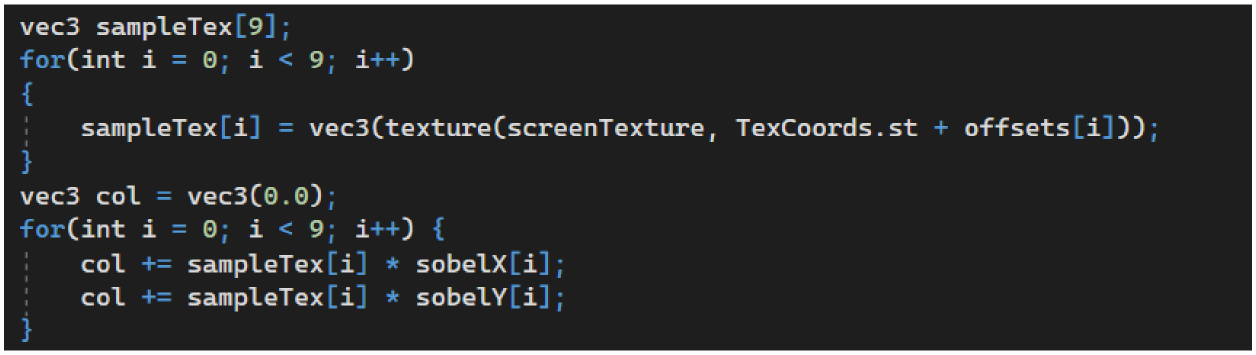
Now that we had a 2D image, we could effectively sample pixels from the texture, and do convolutions. We chose the Sobel operator for this. We wrote yet another shader to apply the sobel filtering, and paste the output on another fullscreen polygon. As for the shader, since we had a texture of the previous render pass, we could simply use an array of offsets, combined with a sobel kernel for our convolutions.



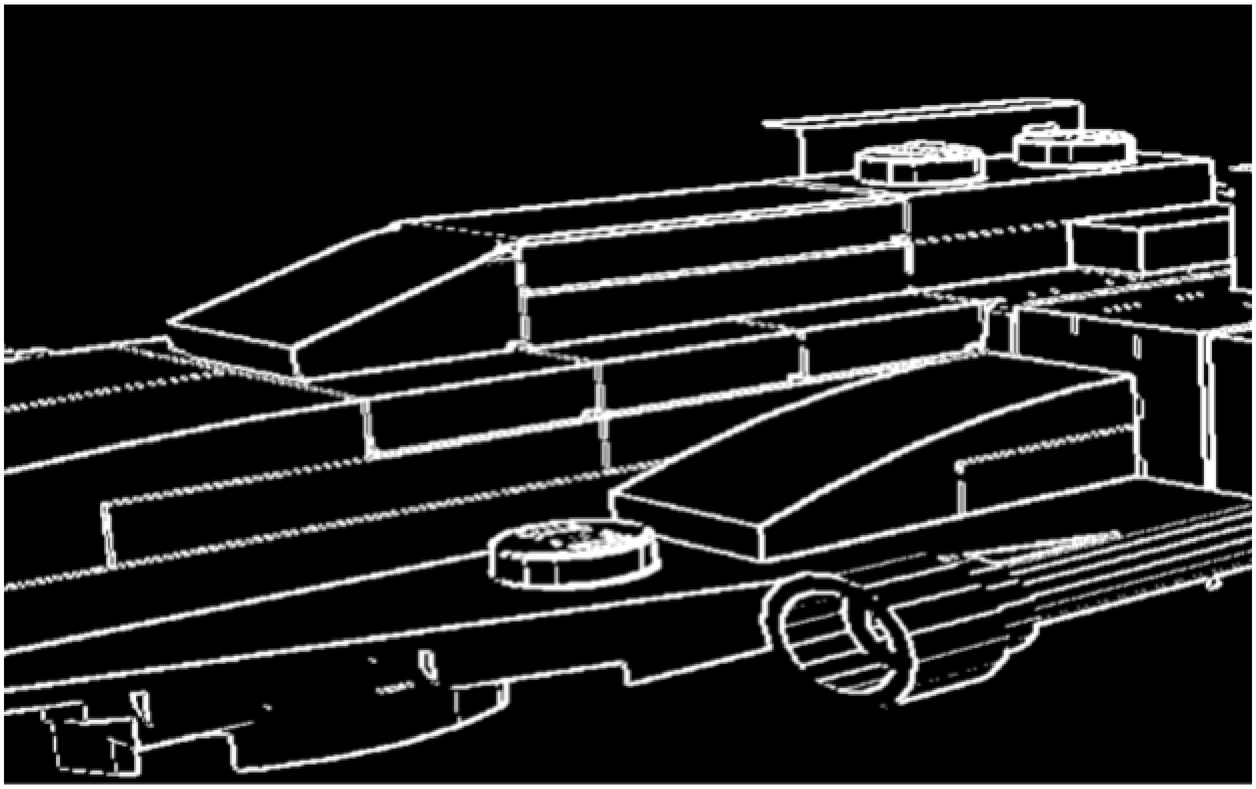
```
vec2 offsets[9] = vec2[](
    vec2(-offsetx,  offsety), // top-left
    vec2( 0.0f,    offsety), // top-center
    vec2( offsetx,  offsety), // top-right
    vec2(-offsetx,  0.0f),   // center-left
    vec2( 0.0f,    0.0f),   // center-center
    vec2( offsetx,  0.0f),   // center-right
    vec2(-offsetx, -offsety), // bottom-left
    vec2( 0.0f,   -offsety), // bottom-center
    vec2( offsetx, -offsety)  // bottom-right
);
```



```
float sobelX[9] = float[](
    -1, 0, 1,
    -2, 0, 2,
    -1, 0, 1
);

float sobelY[9] = float[](
    -1, -2, -1,
    0, 0, 0,
    1, 2, 1
);
```



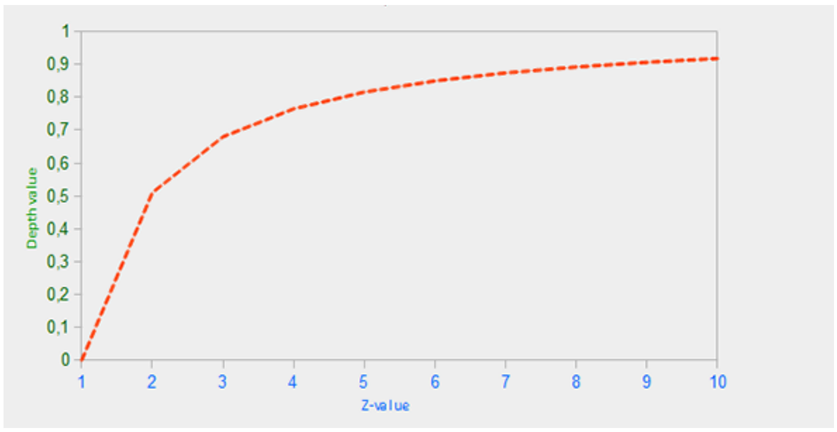
```
vec3 sampleTex[9];
for(int i = 0; i < 9; i++)
{
    sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));
}
vec3 col = vec3(0.0);
for(int i = 0; i < 9; i++) {
    col += sampleTex[i] * sobelX[i];
    col += sampleTex[i] * sobelY[i];
}
```

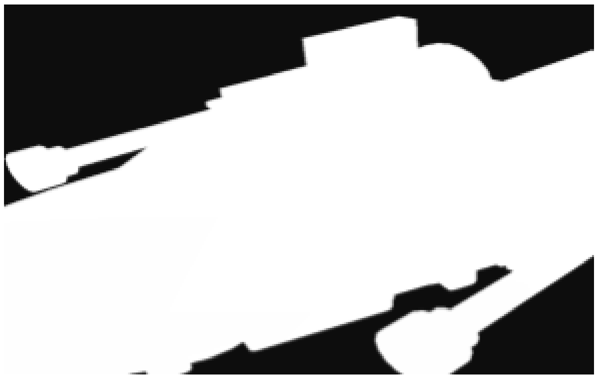


The final result of normal based crease detection

Outline Edge Detection

For our outline we decided to use the depth map after rendering the model once. This followed mostly the same process, rendering a texture, and pasting it on a fullscreen quad. The only notable difference was that we had to linearize our depth values. By default, there was high precision for closer values, and very low precision for far away values. This distribution would make it very hard for sobel to detect outlines, unless you were very close to the model. Thus, we scaled our gl_fragcoords.z to get a linear distribution. Again, thanks for Joey, this graph is his, as well as the information about scaling the depth values.




Depth map with no scaling

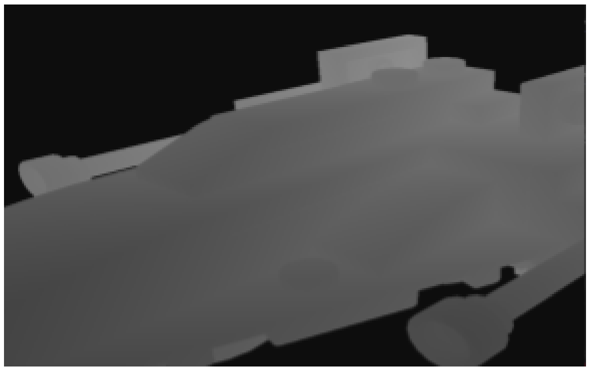
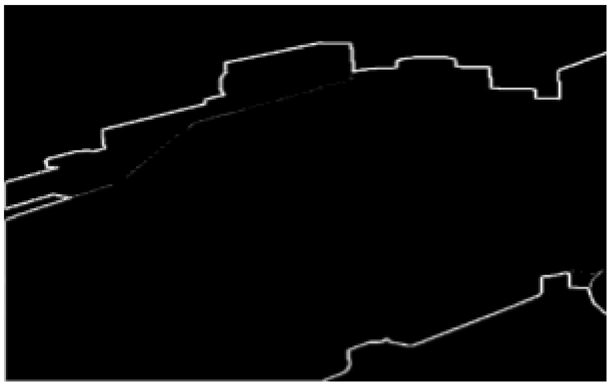


Image with linear Depth values

All that was left to do was apply the sobel operator one more time!


Depth based edge detection!

Combing the Textures

This proved to be the downfall of the project. While we could render different versions of the model, we struggled to combine them. You may have noticed a slight change every time we did a texture to fullscreen poly pass. With each iteration, the output image was slightly zoomed and stretched. I (Gus) tried for multiple days to debug this, but the answer still eludes me. Here are some of the things I did:

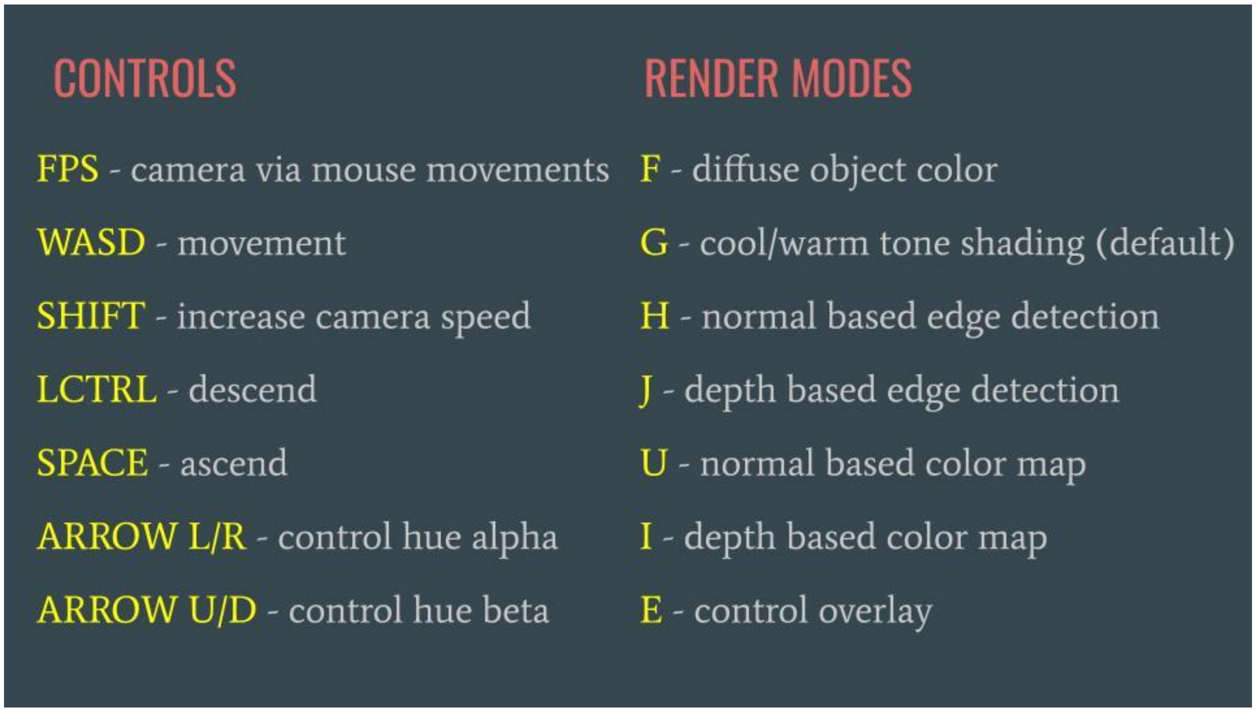
- Mess with the fullscreen polygons. From the tutorials, it was 2 triangles that covered from (-1,-1,0) to (1,1,0) in NDC space. I tried different z values with no output
- Mess with the texture. The textured image that was mapped onto the polygon was from texel (0,0) to texel (1,1) matching with the coordinates above respectively. I tried scaling the texture, scaling the frame buffer object that the texture was projected onto, shrinking the texture, literally every stretch and squeeze, but it seemed pixels were just being clipped!
- I tried directly rendering the framebuffer's color buffer to GLFW's default framebuffer (the one displayed to the window). This actually worked. I was able to get full resolution output with no zoom. However, when I tried to directly sample color values from the buffer on the next shader pass, the model would go entirely one color. Still not sure why that happened.

Still even with the zoom, we should be able to just scale the model texture so it is zoomed in, and combine it with the pixels from the edge detection textures right? I tried that too. Here is the output:

This monstrosity was also alive. I think something was wrong with the offsets for the reads in the texture buffers, because every frame I would get a new facefull of chaos. It was like watching TV static, but intensely bright, and in my dark room it caused me great pain. At this point, I was running out of time, so I had to call it quits.

Render Modes

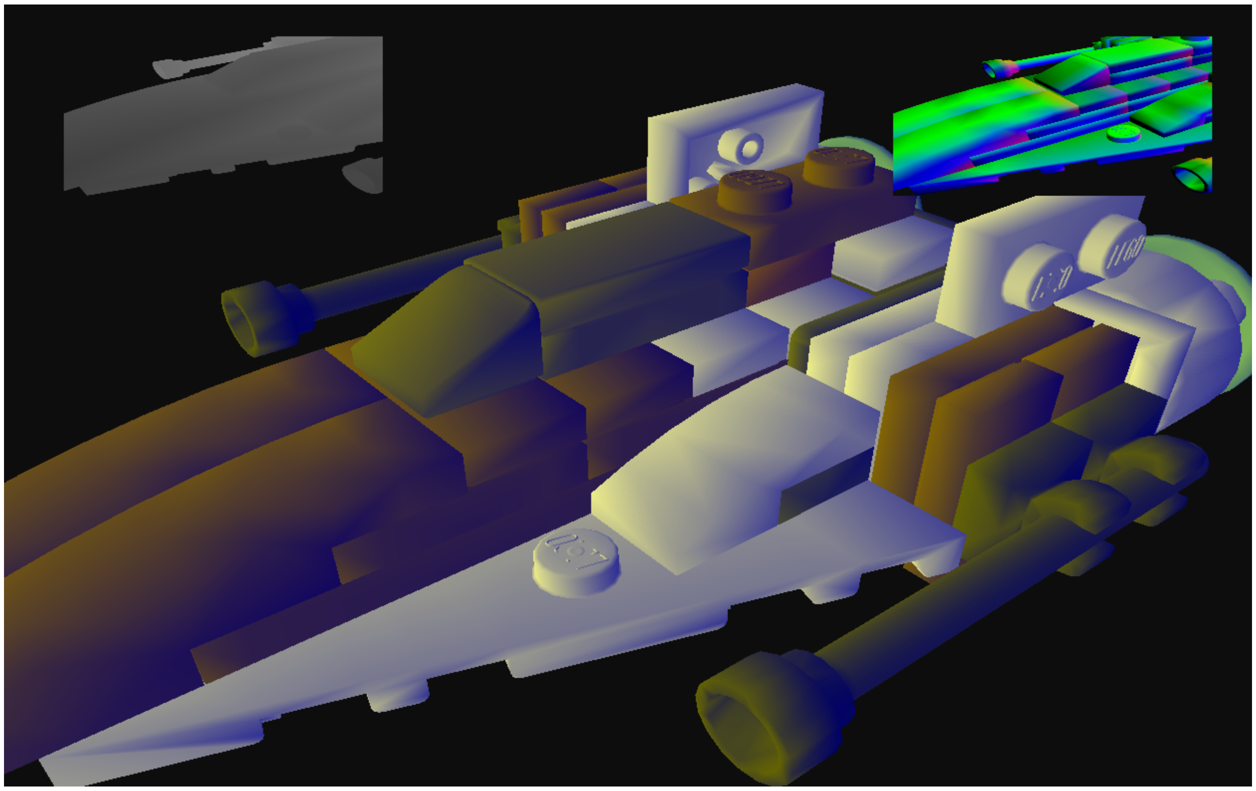
We spent the rest of the time adding a bit of user functionality. Arrow keys to control hue parameters for the tone shading, and buttons to toggle different render passes of the model, since we couldn't combine them.

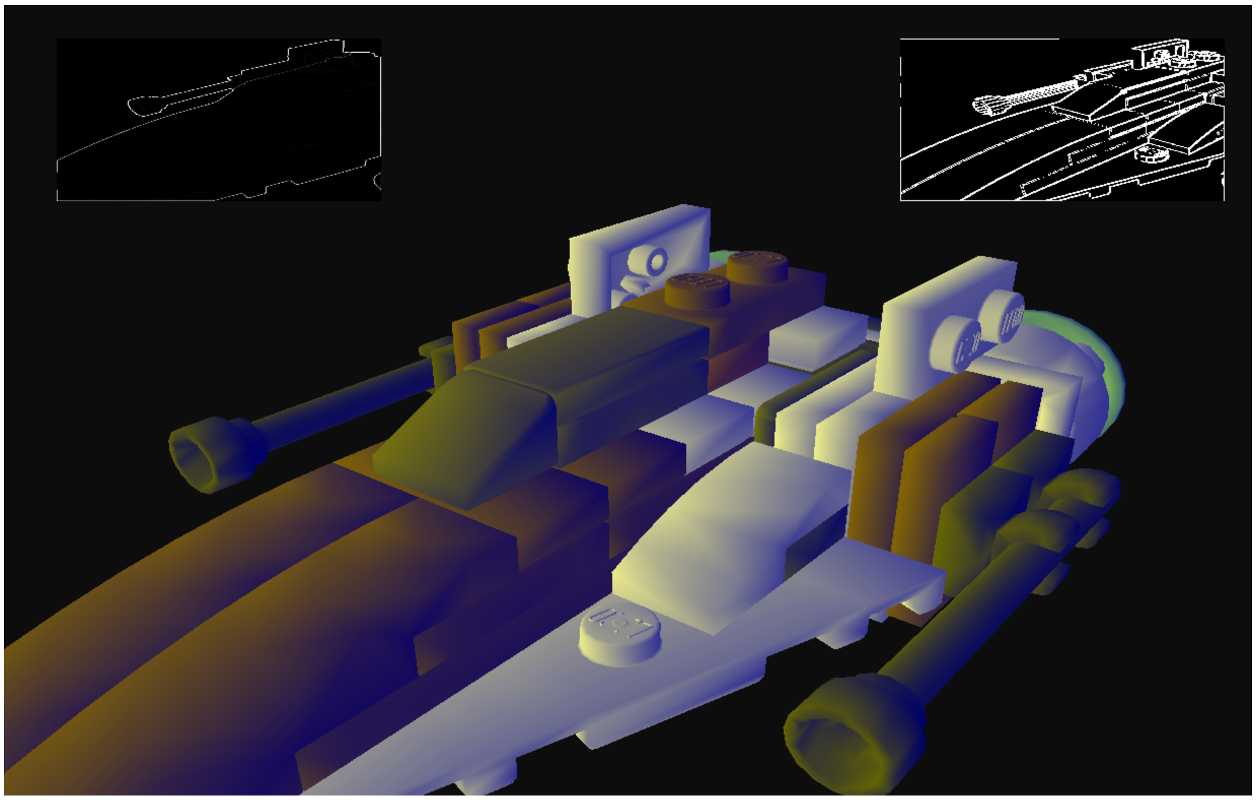


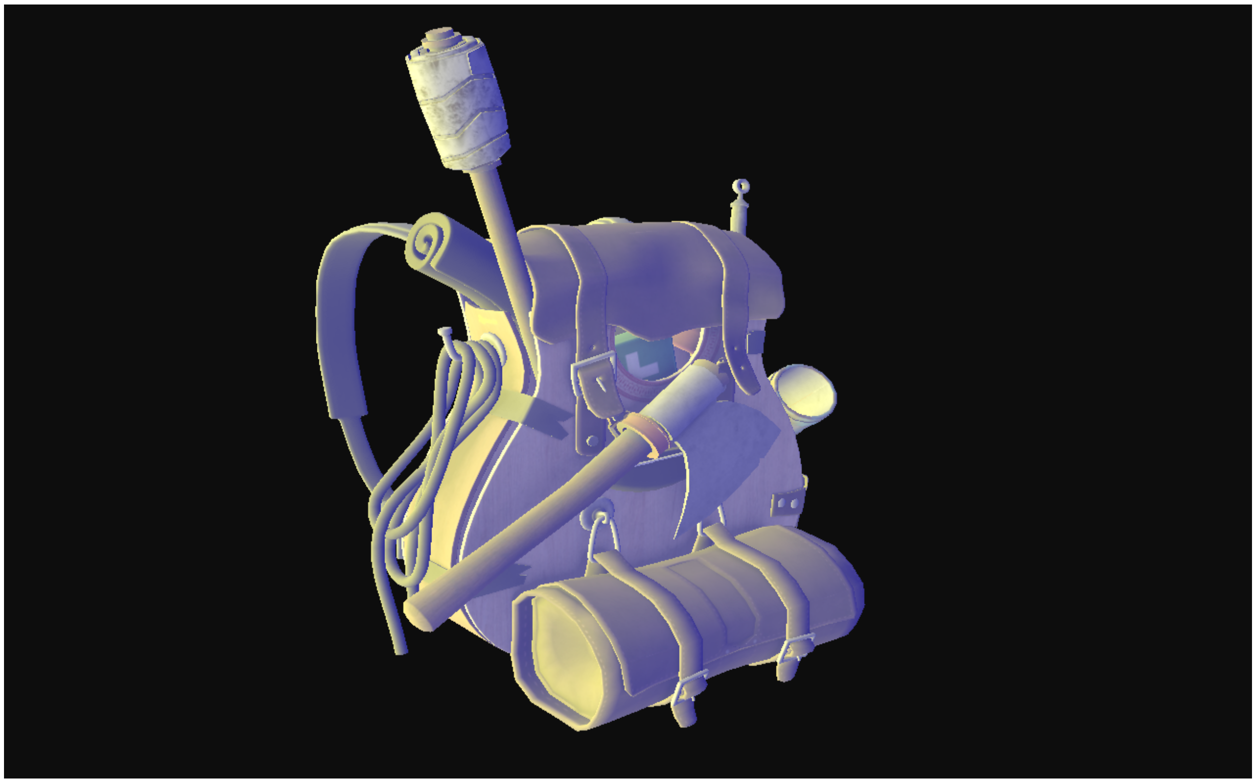
CONTROLS	RENDER MODES
FPS - camera via mouse movements	F - diffuse object color
WASD - movement	G - cool/warm tone shading (default)
SHIFT - increase camera speed	H - normal based edge detection
LCTRL - descend	J - depth based edge detection
SPACE - ascend	U - normal based color map
ARROW L/R - control hue alpha	I - depth based color map
ARROW U/D - control hue beta	E - control overlay

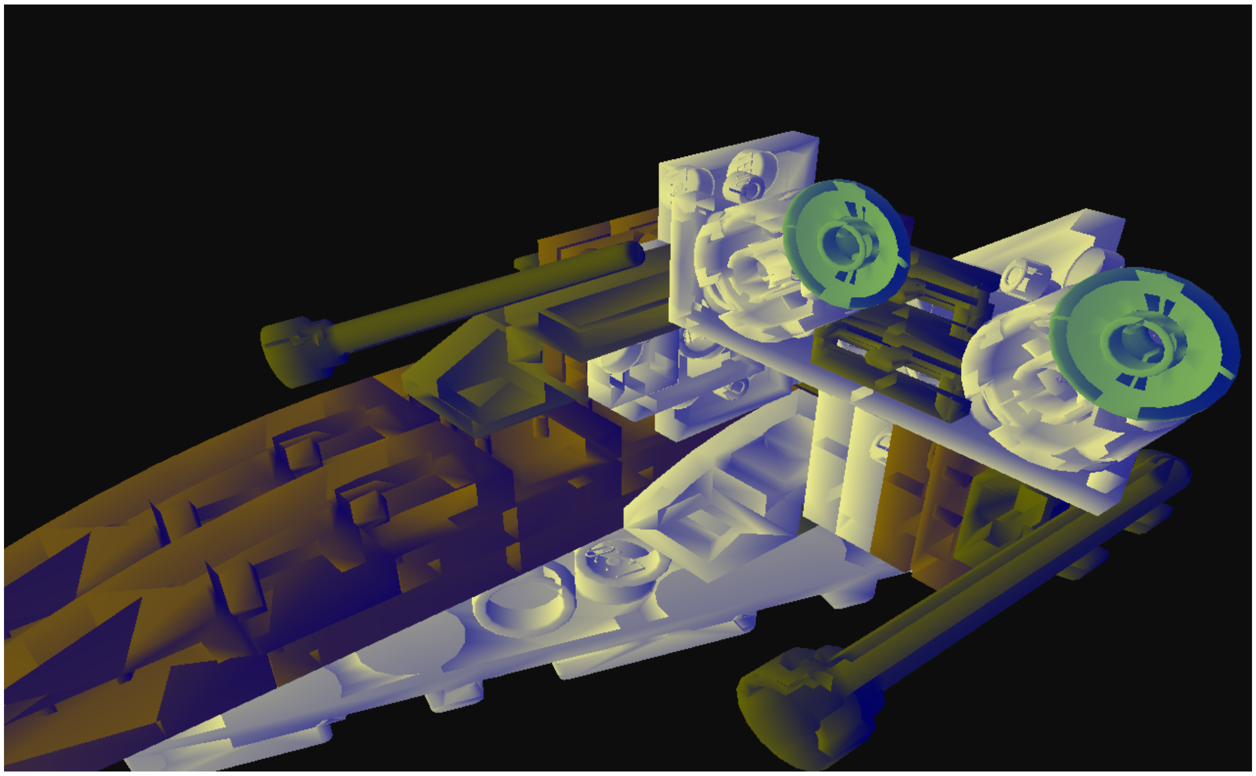
Closing Thoughts

Overall, this project was pretty fun! Getting deeply involved in OpenGL code was a challenge, and there were many hours spent pouring over documentation and online tutorials. We really wish that silhouetting worked out, and we had to scrap the metal textures for lack of time. But the experience alone was well worth it. Enjoy some extra images from the process!


First pass normal and depth textures, all in frame!


Second edge detection passes, showing it all off at once!


The tutorial came with this asset, it used a normal map, so our entire shading algorithm was thrown off.


Forgot to enable the depth buffer.