# COSC 342 Assignment 3 – Render Engine

## Report for Roland Locke, ID #5493858

## Introduction

This report details the creation and testing of a basic render engine. The structure of this report is as follows: A process that describes what features are included in the render engine, how they were implemented, and some brief yet informal experiments around some problems that were encountered. The second section formally tests some the implementations described in the process section, and the discussions and conclusions draw things together.

### Process:

In this section the process of creating the engine will be discussed, what features were added, how they were added, and what issues were encountered along the way. As well as some short discussion on the issues, problem solving, and resolutions.

The first addition made to the render engine was passing .mtl file data to the fragment shader and applying them during run time as Phong Shading and Blinn-Phong reflection model. This mainly involved changes to the mtlShader.frag file. After ensuring that the data was being passed as uniforms from the constructor and setter methods of MTLShader.cpp, the fragment shader needed to apply these material properties. Using data passed from the vertex about vertex positions in relation to the camera, the fragment shader uses the data to compute the diffuse component, angular and specular components for each fragment.

One more material property attempted to be implemented was the specular exponent, or d component in the .mtl file. This is essentially the size of the specular component highlights on the surface. The file material.cpp didn't have a getter or setter function for this material component so those were added, and the value was passed to the fragment shader uniform values as ns. However, when passing it to the fragment shader, the model appeared completely white, as if it had flat shading with a white color. Some experiments were performed to see what values of ns would produce the expected results. See *figure 1.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ns Value or ‘shininess’ Value on Scene Objects | | | | |
| Ns = 600.0 | Ns = 6.0 | Ns = -10.0 | Ns = -40.0 | Value passed from .mtl file |
| A person in a suit  Description automatically generated | A person in a suit  Description automatically generated |  | A white silhouette of a person  Description automatically generated | A silhouette of a person  Description automatically generated |

Figure 1

Values that went into the negatives seemed to make the object turn completely white, but no positive number created the same results as the value passed from the .mtl file. In the end the problem could not be resolved so, the ns value was fixed to 6.0 in the fragment shader.

Using the material components passes to the fragment shader, the Phong and Blinn Phong illumination models were implemented. One issue encountered during this process was a sort of cross that sat directly on the center of the screen with different lighting in each quadrant. See *figure 2.*

A close-up of a person's suit

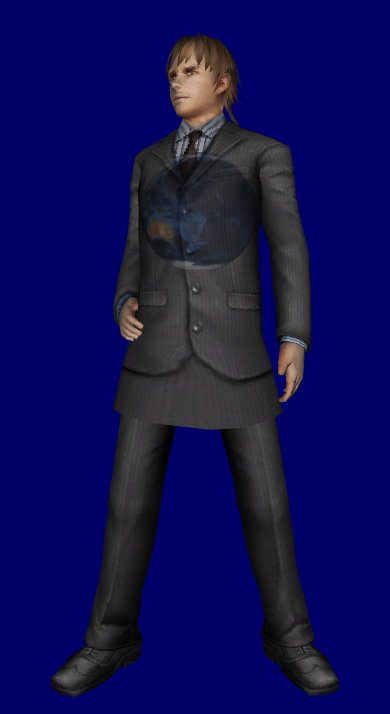
Description automatically generatedA close up of a suit

Description automatically generated

Figure 2

The issue causing this was the Blinn-Phong math involving the Halfway vector which wasn’t being computed correctly. Now, the program uses the normalize function to calculate this vector, which solved the issue.

Supporting transparent materials in the engine, was the second goal. This required the enabling of the blend function in OpenGL and only small tweaks to the fragment shader accounting for the alpha values of the textures. No issues occurred in the implementation of this, however the functionality does have a limitation. If an object with a transparent texture is passed to the engine before one that appears behind it in the scene, the transparent texture will not accurately reflect the occluded objects in the scene space. See figure 3.

A person in a suit

Description automatically generated

The Earth model was passed to engine before the model of the person.

The Model of the person was passed to engine before the model of Earth.

Figure 3

Note that the model of Earth in the second image does not display the person behind it, instead it displays the blue background of the scene through the person. This issue is similar to multiple rendering passes to create post processing effects; however, this issue is not caused by conflicting info from multiple render passes. The order which objects are passed to the engine must relate to the order in which they are rendered. Therefore, if the Earth model was rendered before the model of the person, then the transparency of the earth will not reveal the person.

Finally, Render-To-Texture approach was used to implement a special effect shader. This was achieved by looking at previous lab code and involved the introduction of postEffect.frag, passthrough.vert, PostProcessingShader files, and a basic quad rendering file. These work together to create the glassy effect seen in the renderer. This is organized into a function that can be enabled during run time.

The final engine can display multiple textured models in a scene, apply lighting, and apply a basic postprocessor effect to the screen. Next, this report will test some of the engine’s functionality.

### Experiment:

This section details some brief experiments conducted on the engine. First, testing the how fast our engine can create frames in the scene based on the number of triangles in the scene. The hypothesis being: the more triangles rendered in the scene the lower the ms/frame (milliseconds per frame) will be. The measurement of milliseconds per frame is calculated in the render loop and is reported almost exactly every second during runtime. The number of triangles for each scene was calculated with Microsoft’s 3d Viewer software. The experiment was conducted by running the program for 10 seconds and recording the average ms/frame for each of those periods. The test was conducted with three models, each with a different triangle count. The test was then repeated using the ‘glassy’ effect included in the render.

|  |  |  |  |
| --- | --- | --- | --- |
| Average Millisecond(ms)/Frame on Model Triangle Count | | | |
| Model Triangle Count | 896 | 5,004 | 36,700 |
| Average ms/Frame | 0.439963 | 0.511755 | 0.511695 |
| Average ms/Frame with ‘glassy’ Effect on | 0.614523 | 0.745225 | 0.7925101 |

Figure 4

As seen in figure 4, as model’s triangle count increased, so did the average frame time. The same is experienced with the ‘glassy’ Post Effects enabled.

### Discussion/Conclusions:

The results from this experiment show that the more triangles in a scene, the more the performance suffers. The post process effect suffers a similar performance hit, however the average ms/frame time is worse across the board as pos processing requires two render steps pre frame rather than one. Due to this performance impact, further iterations of this engine should include Occlusion culling for assets as well as Lod (level of detail) for objects at further distances. Further improvements to performance could involve ‘baking’ the lighting to the models before runtime. Effectively minimizing all lighting calculations when the engine is running.

With its current functionality, the render engine is capable of loading and displaying simple models with textures and basic shading. It allows basic parameters to be tweaked in the render and shader processes, and helps facilitate experiments with models, textures, and lighting. Improvements would enhance this engine, but in its current state it is still an important tool.