OpenGL Render Engine

Joseph George

COSC 342

Notes on Functionality

- Program is in standard render mode by default
- Pressing key `2` switches rendering to RGB Normalization Effect
- Pressing key `3` switches rendering to Black and White Effect
- Pressing key `1` switches program back to standard render mode
- Transparency specification for objects are enabled by default
- Inside of XCode, the following command line argument needs to be added before running the program in order to render the custom object file provided in the assignment submission: "sinon.obj"

1 Object Model and Material Loading

`Objloader.cpp` loads an object file using the `assimp` library. Contained within the source code of `Objloader.cpp` is a for loop implemented in order to store the geometries of the child meshes. Accessing `mNumMeshes` from the created `aiScene` instance, iteration through the data structure containing the meshes, storing indexed vertices, indexed UVs, and indexed normals was enabled. Finally, a `Mesh` instance was created, and using this instance, vertices, UVs, normals, indices, and the correct material index was set for each mesh and then added to the geometry.

Next, all of the materials were loaded using a for loop which went through the number of materials in the `aiScene` instance. Inside of this loop, K_d, K_a, K_s, N_s, and d were retrieved for each material, and were consequently set for the given material along with the texture name.

Inside of `Group.cpp`, all meshes were rendered instead of only the first one. Shaders for all of the meshes were set up by using a for loop, where for each mesh, an instance of `MTLShader` was created, where the diffuse, ambient, specular, opacity, and shader name were set for each mesh, as well as the texture.

2 Shader Implementation for Material

Inside of `MTLShader.cpp`, the implementation of the material setters for the diffuse component, ambient component, specular component, and opacity were completed by specifying a name for each of them which could be passed to and then used as `uniform` variables in the fragment shader. These set methods were used in the previously mentioned `Group.cpp` to get the material properties for each material of a given mesh.

3 Phong Shading and Blinn-Phong Reflection

Phong shading was used in the `mtlShader.vert` file and calculated on a per fragment basis. Given the `vertexPosition`, `vertexUV`, and `vertexNormals` for all different executions of the shader, the `UV, posWorldSpace, normalCameraspace, eyeDirectionCameraspace,` and `lightDirectionCameraspace` were interpolated for each fragment.

The interpolated values from the vertex shader were then passed to the fragment shader, `mtlShader.frag`. The fragment shader outputted a 4-dimensional vector utilizing the uniform values which were derived from the material files, including the diffuse, ambient, and specular color components, as well as the exponent n. The fragment shader used the Blinn-Phong reflection model to calculate the final result of the color values for each fragment, by the calculation of a halfway vector H, which was calculated to be the normalization of the light source L added to the viewer vector V (or more specifically, `eyeDirectionCameraspace`). The final `rgb` values were calculated by adding together the ambient, diffuse, and specular components (I_a, I_d, I_s), after performing the necessary calculations to get these values using the standard Blinn-Phong reflection method.

4 Special Effects Shaders

In order to enable custom effects shaders for rendering, a few things were added into `renderApp.cpp` and the fragment shader. Inside of `renderApp.cpp`, a vector of Group pointers were added. This allowed for access to the `setRenderMode()` method for any amount of objects passed as arguments to the program. A series of boolean statements were then added inside of the render loop, which allowed for the user of the program to press 1 switching the render mode to standard mode, 2 to switch into RGB normalization effect, and 3 to switch into Black and White effect.

Inside of the fragment shader, a uniform float variable was added whose value determined the render effect to take place through a series of boolean statements inside of `main()`. By default, the standard render mode is calcucated when the program starts up, and after switching to any other render modes, the user can simply press 1 to get back to the standard mode.

5 Testing

The first test of the program consisted of rendering the two object files provided without any special effects. After seeing that the objects did not render properly, I found out that there was an error in `MTLShader.cpp` where the transparency effect was not being properly passed to the fragment shader. After fixing this error, I found that the object files rendered successfully.

After adding a single special effect in the fragment shader, I tested the two object files again, first checking that the program rendered the objects in standard mode, and then using the specified keyboard keys, switched back and forth between standard render mode and the given special effect until I was satisfied that the special effect was working. I did this for the next special effect as well, switching between all effects and standard mode to make sure everything was working.

Finally, I used the above methods to test the program with an additional object which I produced by exporting an object and material file from Blender, specifying to include the UVs and normals in the export. After seeing that the program and the special effects shaders implemented worked with the two object files provided, as well as my own object file, I was satisfied that the program was written correctly.

After a copious amount of debugging at the beginning of the project as well as extensive testing, I've come to the conclusion that there are no flaws in my program that I am currently aware of.