Jacob Senior

10 December 2023

CS 330 Comp Graphic and Visualization

Professor Montalvo

Module 7 Project Submission: Reflection

In this project, I recreated a photographed scene that consisted of four objects on a countertop. To recreate these objects, I used primitive shapes including a cube, a cone, cylinders, a plane, a sphere, and a torus. These primitives were scaled, rotated, and translated to represent the captured objects. I incorporated a phong lighting model to mimic realistic lighting. The Phong lighting model includes ambient, diffuse, and specular components. These components together help accurately depict shadows and reflections. Initially, I included all components of the Phong lighting model exclusively in the light components. However, that does not help portray a realistic lighting environment with multiple objects since different objects react to light differently. To remedy this, I included a specular texture for materials along with the diffuse texture. The diffuse texture is used to show the color of the object while the specular texture shows what parts of the object should reflect light using the given specular intensity. This can be seen on the battery object in my scene. The diffuse texture consists of the grey and black color on the cube while the specular texture uses a scratched metal texture to reflect light. While this lighting model works well, it requires well-crafted textures to give the desired effect. This was a challenge that I encountered when working on the project. I struggled to find textures that accurately represented my objects and lacked the artistic knowledge and software to create specular maps that would provide the intended effect. Another challenge I encountered was with the meshes for the primitive shapes. In the meshes.cpp file there are multiple errors in the vertices and normal coordinates which create visual artifacts. I tried to repair some of these errors but could not find the cause of all of them. Despite these errors, I think the lighting model and textures help make the scene feel much more realistic.

The user is able to use multiple inputs which allow them to explore the scene from different perspectives. “W”, “A”, “S”, and “D” are used to move the camera forwards, left, backwards, and right respectively in relation to the camera’s orientation “Q” and “E” are used to move the camera up and down regardless of the camera’s orientation. The mouse cursor is used to change the camera’s orientation up, down, left, and right. The mouse scroll wheel can be used to adjust the speed at which the camera travels. Scrolling up increases speed while scrolling down decreases speed. Finally, the user can press “P” to toggle between orthographic and perspective views while running the program. Orthographic view provides a 2D projection of the scene while perspective view provides a 3D projection of the scene.

Several custom functions were used to make the code modular and organized. All mesh information for the primitive shapes is stored in the meshes source and header files. These files are used to generate the vertex, normal, and texture coordinates for each type of object. To further this modular approach, I created functions for each type of primitive shape that would draw the shape given a scale, rotation, and translation for the object. I used functions for each object that would draw each component of the object using the above-mentioned functions. One way I could have increased the reusability of these object functions is to create the objects so that they can be scaled, rotated, and translated as a single unit. Currently, the object is created by orienting each component independently and the components are unable to be transformed altogether. The encapsulation of the camera object in the camera.h file makes the camera object very portable. The file contains functions to get and update the camera’s vectors as well as process movement inputs for the camera. Within my fragment shader I used two custom functions for calculating the different types of lighting. CalcPointLight takes a point light and calculates the color output with the given point light. CalcSpotLight takes a spotlight and calculates the color output with the given spotlight. Not only can this function be reused in future programs, but allows for scalability if we wanted to add more of these types of lights to our scene. Finally, the UCreateTexture and UDestroyTexture functions make it very easy to import textures to our program and these functions can be easily reused in future programs. Most of the other functions used in this program are primarily for organization purposes. Using custom functions is important when programming to make our code more modular and organized. By doing so, we can increase the reusability of our code also increasing our efficiency during maintenance.