**Module Seven Project: OpenGL Scene**

Joseph Kim

Southern New Hampshire University

CS-330-T1028 Computer Graphic and Visualization 21EW1

Dr. Scott Gray

October 13, 2021

**Module Seven Project: OpenGL Scene**

In the project for CS-330 Computer Graphics and Visualization, a scene containing three-dimensional objects was developed using OpenGL. Through the use of OpenGL, objects were created, lighting and textures were applied, and control of the virtual environment was given to the user through the use of a keyboard and mouse input devices.

The scene contains six unique objects when including the desk that the objects rest on. The desk, airpod, and watch were constructed similarly, therefore; the development choices for these objects will be summarized together. However, it is important to note that each of these objects applied unique matrices to apply transformation, rotation, and scaling as well as textures from unique sources in order to properly portray said objects. All of these objects were implemented by constructing a cube primitive shape. Each of the objects are made up of a length, width, and height component, therefore; it is necessary to create a cube to properly portray the objects in three dimensions. A cube was constructed within a mesh struct which contained a vertex attribute object, vertex buffer object, and the number of indices. The vertex attributes defined the position of each point necessary to construct a cube, the texture coordinates to properly map the texture image to the cube, and the normal vertices to the surfaces that are utilized in the Phong lighting model. Once a cube was constructed and created within a mesh struct, the proper texture was loaded and bound to the appropriate texture unit and sent to the shader program. Matrices required for the Phong lighting model as well as matrices required for the model, view, and projection for the scene were sent to the shader as uniforms and the objects were rendered. The pencil object utilized source code found from Github user Michalbb1 to render the cylinder primitive shape and a pyramid primitive shape to portray the tip. In order to construct a cylinder, the code used additional header and cpp files that defined a mesh and vertex buffer object. However, the steps to render a cylinder were similar to the cube. Textures, lighting calculations, and model, view, projection matrices were send to the shader program as uniforms and the object was rendered. The pyramid shape for the pencil’s tip was created similarly within a pyramid mesh and textures, lighting calculations, and model, view, projection matrices were sent to the shader program as uniforms to render it. The glasses object was created using the same cylinder application logic as the pencil as well as the same cube primitive shape logic as a cube. The object required two cylinders for the glasses as well as three cubes for the frames and nose bridge in order to properly construct it. Unique textures, lighting calculations, and model, view, projection matrices were applied, and the object was rendered. The final object was a sticky note dispenser in the shape of a driver golf club head. In order to construct the base of the golf club, a sphere application logic was used from the same Github user Michalbb1. However, the logic was slightly modified to construct a half sphere instead of a full sphere. In order to construct the face, a cube was created and placed in front of the half sphere. A cylinder was also created to portray the insert for the golf shaft. Finally, a yellow cube was created in order to represent the sticky note protruding from the top of the golf club head. For each component, unique textures, lighting calculations, and model, view, projection matrices were applied, and the object was rendered.

The scene employs a keyboard and mouse input device to navigate the three-dimensional scene. In order to allow a user to control the camera present in the scene, the GLFW library’s provided functions were used to implement the logic. The function glfwGetKey(GLFWwindow \*window, int key) was used to bind a specific key and tie it the camera’s position using the camera’s function ProcessKeyboard(Camera\_Movement direction, float deltaTime). The camera object contains enumerations for the different directions that it can move. Therefore the logic ties a specific key to a specific movement of the camera. The table below shows the direction a user can move the camera with the press of a specific key:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Camera Movement Direction | Forward | Backward | Left | Right | Up | Down |
| Key | “w” | “s” | “a” | “d” | “e” | “q” |

In addition to camera movement, the user can switch the context between perspective and orthogonal view by pressing the “p” key. In addition to keyboard input, the user can change the view angle of the scene with mouse movement and alter the speed at which the camera moves with the mouse scroll bar. First, the location of the mouse cursor has to be determined and a function called UMousePositionCallback() was implemented to determine the x and y position of the mouse cursor on the screen. Then, within the camera object, the function ProcessMouseMovement() was implemented that altered the yaw component according to the x position and the pitch component according to the y position. The resulting pitch and yaw components were then used in the updateCameraVectors() function that created a new vector to view the scene. In order to utilize the mouse’s scroll bar, the position of the scroll bar first had to be determined with the function UMouseScrollCallback(). This function determined the offset in the x and y position and then called the camera object’s ProcessMouseScroll() function, which only passed in the y offset as an argument. The ProcessMouseScroll() function then altered the camera object’s data member, MovementSpeed, to change to speed at which keyboard inputs could move the camera around the scene.

Many of the libraries used such as GLFW, GLAD, and the provided header and source files provide functions for the program that create a modularity. However, a few custom functions were also created in order to ensure that the code was more modular and organized. The creation of each objects mesh was converted into a function. For example, the logic to create vertex buffer objects and their attributes and vertex attribute pointers were separated into a function based on the object and its primitive shapes. This allowed for the function to be called from main, instead of having to create each object’s mesh within it. In order to load textures from a resource, two functions were created, UCreateTextureRepeat() and UCreateTextureClamp(). These functions handled the loading of textures with either repeat or clamp texture wrapping. As most objects fell into two categories when computing lighting components, the functions setShaderUniforms() and setFaceShaderUniforms() were created to set objects with normal lighting calculations and objects with a shinier lighting calculations. Within the functions, the appropriate shader was selected, the material’s shininess was set, and each light source’s ambient, diffuse, and specular component was set. As the scene contains one directional light, four point-lights, and one spotlight, each light source’s ambient, diffuse, and specular components as well as the attenuation values were set. In addition, the view and projection matrices were set within the functions. To modularize the code within the main function, all of the shaders for each object were created first. Then all of the meshes were created for each object by calling the appropriate create mesh functions. Then all of the textures were loaded by calling the appropriate create texture functions. Then each texture unit was assigned to the appropriate shaders. Within the render loop, after setting the necessary variables for input processing, the logic for each object followed the same pattern. First the lighting uniforms were set by calling the appropriate shader uniforms function, each diffuse and specular maps were bound, the appropriate vertex attribute object was selected, the model matrix was calculated and sent to the shader, and the object was rendered.