**Project Reflection**

Ikechukwu Onuoha

Southern New Hampshire University

CS-330-J7086 Comp Graphic and Visualization 23EW1

Malcolm Wabara

October 15, 2023

**Justify development choices for your 3D scene. As you write, think about why you chose your selected objects. Also consider how you were able to program for the required functionality.**

3D graphics is central to computing applications like video games and virtual simulations. It offers a deeply immersive experience, letting users engage with environments much like they would in the real world. Crafting a 3D scene requires attention to numerous details, including object selection, depiction, interaction techniques, and the overall user experience.

I chose to create a 3D scene featuring a tabletop with legs and a bowl-shaped plate. These objects are common in real-world scenarios, making my scene relatable. I also Incorporated textures for the tabletop, legs, and plate to add realism to the scene by providing detailed visuals of the objects. The choice of shaders and lighting models (ambient, diffuse, specular) enhanced the visual quality of the scene by simulating how light interacts with different materials. This choice provide the user with the flexibility to switch between orthographic and perspective projections, offering different viewing experiences.

I was able to program for the required functionality by implementing vertex and fragment shaders, using texture mapping, programmed user navigation, input handling, enabling switching between orthographic vs. perspective projection,and employing realism, and aesthetics (LearnOpenGL - Textures, n.d.).To begin, I implemented vertex and fragment shaders to define how the objects in the scene were to be rendered and shaded. This choice allowed the customization of the lighting, materials, and textures applied to the objects.

// Create the shaders

GLuint VertexShaderID = glCreateShader(GL\_VERTEX\_SHADER);

GLuint FragmentShaderID = glCreateShader(GL\_FRAGMENT\_SHADER);

For texture mapping, I used it for enhancing realism, loading, and applying textures to the objects. This programming choice allowed simulation of the appearance of real materials and surfaces on the table, legs, and plate.

unsigned int tableTexture;

glGenTextures(1, &tableTexture);

glBindTexture(GL\_TEXTURE\_2D, tableTexture);

unsigned int legsTexture;

glGenTextures(1, &legsTexture);

glBindTexture(GL\_TEXTURE\_2D, legsTexture);

unsigned int plateTexture;

glGenTextures(1, &plateTexture);

glBindTexture(GL\_TEXTURE\_2D, plateTexture);

For user navigation, I programmed it by controlling a virtual camera. This choice made the scene interactive and user-friendly, allowing users to explore the 3D environment from different angles and perspectives.

float cameraSpeed = 2.5f \* glfwGetTime();

glfwSetTime(0.0);

if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)

cameraPos += cameraSpeed \* cameraFront;

if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)

cameraPos -= cameraSpeed \* cameraFront;

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)

cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)

cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (glfwGetKey(window, GLFW\_KEY\_E) == GLFW\_PRESS)

cameraPos += cameraSpeed \* cameraUp;

if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS)

cameraPos -= cameraSpeed \* cameraUp;

if (glfwGetKey(window, GLFW\_KEY\_P) == GLFW\_PRESS && glfwGetTime() > 0.2)

{

For input handling, I developed functions to respond to keyboard and mouse input. By doing so, I provided users with intuitive ways to control the camera's movement, view direction, and speed, enhancing their interaction with the scene.

/ Process user input

void processInput(GLFWwindow\* window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

Additionally, the option to switch between orthographic and perspective projections adds versatility to the scene. Users can choose the viewing style that suits their preferences, whether they want a realistic perspective or an orthogonal projection for specific tasks.

// Update projection matrix based on the chosen view (perspective or orthographic)

glm::mat4 projection;

if (orthographic) {

// Set up orthographic projection

projection = glm::ortho(-2.0f, 2.0f, -2.0f, 2.0f, 0.1f, 100.0f);

}

else {

// Set up perspective projection

projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);

Lastly, for realism and aesthetics, I created a custom plate geometry that simulates a bowl-like structure. This choice adds depth and visual interest to the scene, making it more engaging for users. I also incorporated lighting and material properties to create a realistic rendering. This includes ambient, diffuse, and specular lighting, as well as shininess properties for materials. These choices contribute to the scene's visual quality.

**Explain how a user can navigate your 3D scene. As you compose your thoughts, discuss how you set up to control the virtual camera for your 3D scene using different input devices.**

The primary tool for navigation in my 3D scene is the virtual camera. It is the user's eyes within the virtual world. The position and direction of this camera determines what the user sees and from which angle.

// Process user input

void processInput(GLFWwindow\* window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

For the keyboard input, the 'Escape' key prompts the window to close from the scene. Key W moves the camera forward while key S moves it backward; this is determined by the cameraFront vector. Key D moves the camera to the right while key A moves it to the left. Key E moves the camera upwards, while key Q moves it downwards. This movement uses the cameraUp vector, determining the up and down direction. Key P allows users to toggle between orthographic and perspective projections.

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

float cameraSpeed = 2.5f \* glfwGetTime();

glfwSetTime(0.0);

if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)

cameraPos += cameraSpeed \* cameraFront;

if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)

cameraPos -= cameraSpeed \* cameraFront;

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)

cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)

cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (glfwGetKey(window, GLFW\_KEY\_E) == GLFW\_PRESS)

cameraPos += cameraSpeed \* cameraUp;

if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS)

cameraPos -= cameraSpeed \* cameraUp;

if (glfwGetKey(window, GLFW\_KEY\_P) == GLFW\_PRESS && glfwGetTime() > 0.2)

{

orthographic = !orthographic;

glfwSetTime(0.0);

For the mouse input, the camera’s up and down movement and left and right movement is controlled using mouse’s x and positions. This helps the user to look around the scene.

// Callback function to handle mouse movement

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos)

For the scroll input, using the mouse scroll wheel to adjust the camera’s moement speed gives users a dynamic way to control how fast they navigate through the 3D scene.

// Callback function to handle mouse scroll

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)

**Explain the custom functions in your program that you are using to make your code more modular and organized. Ask yourself, what does the function you developed do and how is it reusable?**

**processInput(GLFWwindow\* window)`**

This function handles user input for camera movement and perspective/orthographic toggling. It makes the code more modular by encapsulating the input processing logic. This function processes user input, such as keyboard and mouse input, to control camera movement. It encapsulates the logic for camera movement based on user input, making the main loop cleaner and more organized.

**Reusability:** The primary concept behind this function is to process user inputs to control camera movement. In any 3D application or game, camera control is fundamental, so a function like this can be easily reused. For other projects, you'd need to ensure the camera object and controls are compatible.

`**mouse\_callback(GLFWwindow\* window, double xpos, double ypos**

This function manages mouse input to control camera pitch and yaw, enhancing code organization and reusability. It handles mouse movement events and updates the camera's orientation accordingly. It separates the mouse input handling logic from the main loop, improving code readability.

**Reusability:** Mouse-based camera control, like for pitch and yaw adjustments, is a common requirement in 3D environments and first-person games. The core logic for updating camera orientation based on mouse movement remains the same. The sensitivity or scale of the movement may need change depending on the desired user experience.

`**scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)`**

It adjusts the camera's movement speed based on mouse scroll input, promoting code modularity and clear separation of responsibilities. This callback function responds to mouse scroll events and adjusts the camera's movement speed. It isolates the logic for changing the camera speed based on mouse scrolling. it reusable.

**Reusability:** Adjusting parameters based on mouse scroll input isn't exclusive to camera movement speed. Zooming in/out and modifying other variables using scroll input are common patterns. In this project it adjusts camera speed but it can be adapted to control other parameters, like zoom levels or object sizes.

**key\_callback(GLFWwindow\* window, int key, int scancode, int action, int mods)`**

This function toggles between orthographic and perspective projections when the 'P' key is pressed, making the code more organized and maintaining a clear structure. This callback function handles keypress events, allowing users to toggle between perspective and orthographic projections. It separates the key input handling logic, making the main loop more organized.

**Reusability:** Key callbacks in graphics applications or games are standard. They allow the user to interact with the scene or control gameplay aspects. The underlying structure of capturing a key event and executing logic based on the key pressed is general. Modify it to handle different key presses or actions specific to another application. Add new toggles, control object properties, or introduce gameplay mechanics.

**Framebuffer Size Callback (framebuffer\_size\_callback)**

This callback function is responsible for handling window resizing events. It encapsulates the logic for setting the viewport dimensions when the window is resized.

**Reusability** - Window resizing is a common event any windowed application might need to handle, especially in graphical applications where the viewport must adjust accordingly. The core functionality of adjusting the viewport based on window size remains consistent across OpenGL applications. Integrate it into the window event to handle loops of any other OpenGL project to ensure graphics adjust to window resizes.

**Benefits of these functions to another program**

In game development, vertex definition and buffer management can be used in-game objects and characters. They ensure that complex 3D models, even with many vertices, can be efficiently loaded and manipulated in real-time. Texturing brings life into these models, giving them realistic appearances. Lighting plays an important role in setting the mood and atmosphere of game scenes, like in a horror game to the vibrant radiance in a fantasy realm. Transformations ensure that game objects can be dynamically moved, rotated, and scaled in response to game logic or player inputs. This encompasses everything like the motion of a player's avatar to the animation of a swinging sword or the movement of a vehicle.

Creating a 3D scene from a 2D image is a process that requires both artistic vision and technical proficiency. As highlighted, numerous decisions dictate the final appearance and interactivity of the scene, from object selection to texturing, and from lighting to user navigation. Through the modular approach adopted in the programming phase, the code remains organized, allowing for easy modifications and adaptations for future projects. The development and organization of such a project offer a glimpse into the intricacies of 3D graphics programming, revealing the blend of artistry and engineering that underpins such creations.

**References**

*LearnOpenGL - Textures*. (n.d.). Learnopengl.com. https://learnopengl.com/Getting-started/Textures