# Computer Graphics Coursework – Self Assessment Document

**Name:** *Kofoworola Abiola* **ID number:** *22492419*

Complete the self-assessment grid below by writing a short explanation of how you have satisfied the requirement and how it has implemented in your code.

|  |  |  |
| --- | --- | --- |
| **Learning outcome** | **Mark** | **Weighted mark** |
| 1. Use appropriate mathematical tools (40%) |  | 0 |
| 2. Develop a 3D graphics application (30%) |  | 0 |
| 3. Write shader code (30%) |  | 0 |
|  | Total | 0 |

Your mark for each Learning Outcome (LO) is the highest mark achieved based on the criteria specified in the self-assessment grid. Note that you will need to have satisfied all criteria at the lower mark bands to be awarded marks in the higher mark bands, e.g., to get a mark in the 70 - 80 band for a learning outcome you will have needed to have satisfied all criteria in the 40 – 50 and 50 – 60 mark bands.

## Learning Outcomes:

**LO1** Select and use appropriate mathematical tools for constructing and manipulating geometry in 3D space.

**LO2** Develop an interactive 3D graphics application using an industry-standard API.

**LO3** Write shader code for the programmable pipeline on modern graphics hardware using an industry standard shader language.

## Self-assessment Grid

|  |  |  |
| --- | --- | --- |
| **Mark** | **Criterion** | **Comments (state how and where you have achieved the criterion)** |
| 42, 45, 48 | LO1: Basic use of vector and matrix objects | The basic use of vectors and matrices is predominant in my program. The use of vector objects can be seen in the object data structure in line 16 in coursework.cpp. The vectors are used to create default objects position, rotation and scaling. Another use of a vector object can be seen in line 216 in coursework.cpp. In line 216, a vector Object called objects is created and populated with cube objects positioned at different locations in the 3D world space.  The use of matrices can be seen in maths.cpp. Matrix objects are used in the methods to calculate translation, rotation and scaling from line 4 in maths.cpp. The use of matrices is also seen in maths.hpp to define the matrix function in the Quaternion class. |
| LO2: Application compiles and runs without alterations to the source code of CMake file. | A group of wooden crates  AI-generated content may be incorrect.  Figure 1  The image above is a screenshot of a successful run of my initial program without alterations to the source code of my CMake file. |
| LO3: Implementation of shaders to apply appropriate textures to objects. | The implementation of shaders to apply textures to objects has been used appropriately in my program. From line 159 - 161 in coursework.cpp, the LoadShaders() method in shader.hpp is called to load and compile the vertex and fragment shaders. The glUseProgram() function is used to activate the shader program. In vertexShader.glsl, the uv coordinates of the textures is outputted and passed to correspond with the object vertices.  In the fragmentShader.glsl, the UV input is outputted from vertexShader.glsl. The sampler2D uniforms are used to declare the texture types to allow appropriate application of textures to objects in the world space. |
| 52, 55, 58 | LO1: Basic use of translation, rotation and scaling transformations. | In my program, the basic use of translation, rotation, and scaling transformation is used appropriately in maths.hpp, maths.cpp and coursework.cpp. In maths.hpp, rotation, transformation, and scaling transformations matrices are declared with matrix objects as methods. In maths.cpp, these methods are calculated to be used in the program. The use of transformations in coursework.cpp can be seen in line 342 – 345 where the model matrix is calculated. |
| LO1: Implementation of glm library functions for calculating view and projection matrices. | The implementation of the glm library functions for calculating view and projection matrices is calculated in camera.cpp. The view matrix is calculated using the lookAt() glm function with eye, front and worldUp vector inputs. The projection matrix is calculated using the glm perspective() function with fov, aspect, near and far vector inputs. |
| LO2: 3D virtual world has been created using instances of a single object type. | A 3D virtual world created using instances of a single object type can be seen in coursework.cpp. From line 16 – 23, the Object data structure has been created and declared to represent a generic 3D object in the scene with position, rotation, scale, angle and name properties. As seen from line 201 – 226, the Object struct allows the cube model to be defined and load once with multiple instances of the cube created with different positions. |
| LO3: Use of shaders to apply dynamic lighting from point light sources | The use of shaders to apply dynamic lighting from point light sources can explicitly be seen in vertexShader.glsl, fragmentShader.glsl, lightVertexShader.glsl and lightFragmentShader.glsl. In vertexShader.glsl, the world space position of the fragment and normal vector for lighting calculations are declared. In fragmentShader.glsl, the calculation for the point light source prototype is declared and calculated. lightVertexShader.glsl and lightFragmentShader.glsl also includes inputs, outputs and uniforms for the point light sources.  In coursework.cpp, line 182 declared a Light object to manage multiple point light sources. Line 336 in coursework.cpp, sends the point light source properties to the shader program using light.cpp toShader() function. In line 368, the point light sources are rendered as small spheres using light.cpp draw() function to visually indicate their positions in the world space. |
| 62, 65, 68 | LO1: Implementation of students own functions for calculating view and projection matrices. |  |
| LO2: 3D world created using multiple object types. | A 3D world created using multiple object types can be seen in coursework.cpp. The Object data structure created from line 16 – 23 allows the use of multiple object types using the name property. From line 216 – 311, multiple objects are added to the world space and identified with individual names, object models, textures and lighning properties. |
| LO2: Users can navigate the virtual world using keyboard and mouse inputs. | A user can navigate the virtual world space using keyboard and mouse inputs as seen in coursework.cpp. The keyboardInput() function from line 384 handles a user’s movement with the W, A, S, and D keys. The mouseInput() function from line 414 handles a user’s mouse movement to control the camera orientation. |
| LO3: Use of shaders to apply dynamic lighting from different types of light sources. | The use of shaders to apply dynamic lighting from different types of light sources can explicitly be seen in vertexShader.glsl, fragmentShader.glsl, lightVertexShader.glsl and lightFragmentShader.glsl. In vertexShader.glsl, the world space position of the fragment and normal vector for lighting calculations are declared. In fragmentShader.glsl, the calculations for each light source prototype are declared and calculated. lightVertexShader.glsl and lightFragmentShader.glsl also includes inputs, outputs and uniforms for the light sources.  In coursework.cpp, line 182 declared a Light object to manage different types of light sources. Line 336 in coursework.cpp, sends multiple light sources properties to the shader program using light.cpp toShader() function. In line 368, the different types of light sources are rendered as small spheres using light.cpp draw() function to visually indicate their positions in the world space. |
| 72 75, 78 | LO1: Implementation of students own functions to replace glm functions (e.g., glm::length(), glm::dot(), glm::cross() etc.). |  |
| LO1: Implementation of quaternions to calculate rotation matrix. | The implementation of quaternions to calculate the rotation matrix can be seen from line 57 - 78 in maths.cpp. A 4x4 matrix is constructed using the quaternion components to calculate and extract the rotation matrix directly.  In the quaternionCamera() function in camera.cpp, a quaternion is declared using pitch and yaw to represent the camera’s orientation based on the Euler angles. The camera vectors then updated based on the transformed view matrix, ensuring a smooth and accurate camera rotation. |
| LO2: Interactive dynamic aspects of the virtual word and controllable by the user (e.g., position of objects, location and function of light sources etc.). | An Interactive dynamic aspect of the virtual world controllable by the user can be seen in coursework.cpp. The user can control the camera using the **W, A, S, D keys** to move forward, backward, left, and right. The camera orientation can be adjusted using the mouse to look around. The camera is constrained within the room boundaries and objects using collision detection implemented in the checkCollision() and objectCollision() functions. |
| LO3: Appropriate implementation of normal and specular maps. | An appropriate implementation of normal and specular maps can be seen in fragmentShader.glsl. The normal and specular maps are applied to objects in coursework.cpp when the textures are applied. |
| 85, 90, 100 | LO1: Use of quaternions to calculate view matrix. | The use of quaternions to calculate view matrix can be seen in camera.cpp. The view matrix is calculated using the quaternion matrix and the translation matrix which ensures that both the rotation and position of the camera are incorporated. |
| LO1: Use of SLERP to smooth out changes in camera direction. | The implementation of Spherical Linear Interpolation (SLERP) is calculated in maths.cpp. In camera.cpp, the use of SLERP to smooth out changes in camera direction can be seen in the quaternionCamera() function. |
| LO2: Implementation of a third person camera with the ability to switch between first and third person view. |  |
| LO2: The position of the camera or character obeys the constraints of the physical space (e.g., can’t pass through objects, can’t hover in midair etc.). | The position of the camera or character to obey the constraints of the physical space using collision detection can be seen in coursework.cpp. Frome line 47 - 54, room bounds have been defined to represent the physical limits of the room. The checkCollision() function checks if the camera’s position is inside any of these bounds or colliding with any objects in the world space. The checkCollision() returns true if the camera tries to move inside a collision boundary or intersect an object. If collision is detected, the camera position is reset to the original position, effectively preventing the camera from passing through objects and hovering in midair. |
| LO3: Use of shaders to apply parameter driven effects within the scene, e.g., light properties controlled using camera/character position. |  |

# References

ambientCG.com (2025) *Free 3D Assets for Everyone and Everything.* Available at: <https://ambientcg.com/> (Accessed: 15 May 2025).