# Computer Graphics Coursework – Self Assessment Document

**Name:** *Your Name* **ID number:** *12345678*

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 GLM library (`glm::vec3`, `glm::mat4`) is used for all vector and matrix operations. This is evident throughout `coursework.cpp` for creating and managing camera position, view matrices, projection matrices, and model transformation matrices (e. g., `camera.Position`, `viewMat`, `projectionMat`, `modelMat`). |
| LO2: Application compiles and runs without alterations to the source code of CMake file. |  |
| LO3: Implementation of shaders to apply appropriate textures to objects. | Textures are loaded and applied to 3D models. Textures are correctly sampled in the fragment shader using UV coordinates supplied as vertex attributes. (`fragmentShader.glsl`, `common/texture.hpp`, `common/model.cpp`) |
| 52, 55, 58 | LO1: Basic use of translation, rotation and scaling transformations. | Transformations are used to position, orient, and scale objects in the 3D world. This is demonstrated in `coursework.cpp` where `glm::translate`, `glm::rotate`, and `glm::scale` are applied to the `modelMat` (now `instance.transform`) before rendering each object. The `Model` class uses GLM library functions for matrices and vectors, and these are used in `coursework.cpp` to position, orient, and scale objects (e.g., `glm::translate`, `glm::rotate`, `glm::scale` are used to construct the `instance.transform` matrix). The `Camera` class uses GLM for its position, front, up, and right vectors. |
| LO1: Implementation of glm library functions for calculating view and projection matrices. | View and projection matrices are calculated using GLM's `glm::lookAt` (within `camera.GetViewMatrix()`) and `glm::perspective` (in `coursework.cpp`'s main loop) |
| LO2: 3D virtual world has been created using instances of a single object type. | Initially, the world was demonstrated with instances of the `caixote.obj` (crate) model. (`coursework.cpp`) |
| LO3: Use of shaders to apply dynamic lighting from point light sources | Dynamic lighting from a point light source has been implemented. The fragment shader (`fragmentShader.glsl`) calculates diffuse lighting based on the dot product of the surface normal and the light direction. Uniforms for `lightPos`, `lightColor`, and `viewPos` are used to control the lighting parameters from the C++ code. (`fragmentShader.glsl`, `coursework.cpp`) |
| 62, 65, 68 | LO1: Implementation of students own functions for calculating view and projection matrices. | Currently using GLM's built-in `glm::lookAt` and `glm::perspective` functions for view and projection matrices. Custom functions have not yet been implemented. |
| LO2: 3D world created using multiple object types. | The scene now includes multiple distinct types of 3D objects. Specifically, two instances of a crate model (`caixote.obj`) and one instance of a blue cube model (`pilot.obj`) are loaded and rendered. (`coursework.cpp`) |
| LO2: Users can navigate the virtual world using keyboard and mouse inputs. | The camera (`common/camera.hpp`, `common/camera.cpp`) processes keyboard (WASD for movement, Space/Shift for up/down) and mouse (for looking around) inputs. This is implemented in `coursework.cpp` through `keyboardInput()` and `mouse\_callback()`. The `deltaTime` ensures frame-rate independent movement speed. |
| LO3: Use of shaders to apply dynamic lighting from different types of light sources. | Currently, the application uses a single point light source model in `fragmentShader.glsl`. While the \*foundations\* for lighting are there (FragPos, Normal, lightPos, lightColor, viewPos uniforms), it hasn't been extended to demonstrate \*different types\* of light sources (e.g., directional, spot) yet. This criterion requires showing flexibility in handling more than just one point light, which is not yet implemented. |
| 72 75, 78 | LO1: Implementation of students own functions to replace glm functions (e.g., glm::length(), glm::dot(), glm::cross() etc.). | GLM library functions are used for vector operations. Custom math functions for these operations have been implemented in `common/maths.hpp` and `common/maths.cpp` (normalize, dot, cross, length), and are used within the `Camera::updateCameraVectors()` method for calculating `Front`, `Right`, and `Up` vectors. |
| LO1: Implementation of quaternions to calculate rotation matrix. | Rotations are currently handled using GLM's Euler angle-based rotation within the camera and `glm::rotate` for model transformations. Quaternions are not yet implemented for rotation calculations |
| 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.). | The camera is fully controllable by the user. The light source's position can be controlled by the user via IJKL UO keys as implemented in `coursework.cpp`'s `keyboardInput` function, affecting the dynamic lighting on objects. Object positions are currently static after initialization. |
| LO3: Appropriate implementation of normal and specular maps. | Normal maps and specular maps are not yet implemented. The current fragment shader uses basic diffuse lighting based on vertex normals. |
| 85, 90, 100 | LO1: Use of quaternions to calculate view matrix. | The view matrix is calculated using GLM's `glm::lookAt` which internally uses a standard matrix approach, not explicitly quaternions controlled by student code for this purpose. |
| LO1: Use of SLERP to smooth out changes in camera direction. | Camera direction changes are directly updated from mouse input without SLERP (Spherical Linear Interpolation). |
| LO2: Implementation of a third person camera with the ability to switch between first and third period view. | Currently, a first-person camera is implemented. No third-person camera or switching mechanism exists |
| 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.). | No collision detection or physics constraints are implemented. The camera can freely pass through objects and move in any direction. |
| LO3: Use of shaders to apply parameter driven effects within the scene, e.g., light properties controlled using camera/character position. | Light properties (position) are controllable by the user via keyboard input, which is a parameter-driven effect. The light's effect on the scene changes based on its position relative to objects and the camera. (This is partially met by user-controlled light position). |