# Computer Graphics Coursework – Self Assessment Document

**Name:** Adam Wilton **ID number:** *23621661*

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 | Used vector and matrix objects in the maths.cpp file to translate, scale a matrix (Line 3-13) and find magnitude or dot product of vector (Line 31-39) using the glm types |
| LO2: Application compiles and runs without alterations to the source code of CMake file. |  |
| LO3: Implementation of shaders to apply appropriate textures to objects. | Added texture to model class (coursework.cpp Line 172) which then creates and binds the texture and then creates the texture buffer once given the UV coordinates of the texture and binds it to the VAO. The UV buffer is then sent to the shaders. The vertex shader then calculates the coordinates of the fragment (vertexShader.glsl Line 38-42) and the fragment shader retrieve the normal of the texture at that coordinate (fragmentShader.glsl Line 47). |
| 52, 55, 58 | LO1: Basic use of translation, rotation and scaling transformations. | Objects are translated, scaled and rotated in the world space, calculated in that order (Coursework.cpp Line (295-297, 299-300, 315-316) |
| LO1: Implementation of glm library functions for calculating view and projection matrices. | Implemented Own Functions as shown later |
| LO2: 3D virtual world has been created using instances of a single object type. | Used instances of multiple object types as shown later |
| LO3: Use of shaders to apply dynamic lighting from point light sources | Sent the light sources to the shaders where the fragment colour is found based of the point light properties (fragmentShader.glsl Line 64-66, 80-109) |
| 62, 65, 68 | LO1: Implementation of students own functions for calculating view and projection matrices. | Calculated the view matrix by calculating the front, right and up values using the target and eye values passed. The view matrix is then given by multiplying the rotation matrix for the camera to point down the z-axis with the translation matrix of the eye which is then returned by the function (maths.cpp Line 59-65). Projection matrix calculated by calculating the top and right variables and then inputting them into the matrix to find the transpose along with the near, top and far variables. |
| LO2: 3D world created using multiple object types. | My 3D world consists of teapots, walls, floors and Suzanne by using the object struct (coursework.cpp Line 35-42) to loop through all the types and implement them into the world (coursework.cpp Line 292-338) |
| LO2: Users can navigate the virtual world using keyboard and mouse inputs. | Uses glfwGetKey function to detect input of key press and moves the camera eye position in the direction of movement press (coursework.cpp Line 354-381) and takes mouse input using glfwGetCursorPos functions and sets it to the middle of screen then adjusts camera’s yaw and pitch to look left, right, up and down (coursework.cpp Line 383-396) |
| LO3: Use of shaders to apply dynamic lighting from different types of light sources. | Functions for calculating spotlight which takes in direction and angle of spotlight to calculate if object is within its light (fragmentShader.glsl Line 112-147) and a directional light that doesn’t take a position but always has a direction of light, like a sun (fragmentShader.glsl Line 150-173) |
| 72 75, 78 | LO1: Implementation of students own functions to replace glm functions (e.g., glm::length(), glm::dot(), glm::cross() etc.). | (All in maths.cpp) Implemented function for magnitude (Line 31-34), dot product (Line 36-39), cross product (Line 41-44), normalising a vector (Line 46-50) and transposing a matrix (Line 52-55) |
| LO1: Implementation of quaternions to calculate rotation matrix. | Rotated using quaternions by using complex numbers to derive a rotation matrix (maths.cpp Line 104-124) |
| 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.). | Player can control whether teapots rotate, change in scale and whether the lights flash red and blue or the lights just turn green by going into different corners of the space (coursework Line 282-286, Line 301-306 and light.cpp Line 51-68) |
| LO3: Appropriate implementation of normal and specular maps. | Wall and floor textures have a normal and specular map (coursework.cpp Line192-193 and Line 212-213) which are sent to the fragment shader where the specular and normal lights are calculated (fragmentShader.glsl Line 91 and Line 96-100) |
| 85, 90, 100 | LO1: Use of quaternions to calculate view matrix. | View matrix calculated using orientation quaternion matrix which is calculated using the quaternion class (camera.cpp Line 68-73) |
| LO1: Use of SLERP to smooth out changes in camera direction. | Orientation of camera is given by using SLERP method (maths.cpp Line 127-154) which takes 2 quaternions and a constant of rate. The method then calculates the angle to move and checks the 2 quaternions aren’t already together and stops it from going the long way around. It then calculates and returns a quaternion in between the two that were passed. |
| LO2: Implementation of a third person camera with the ability to switch between first and third period view. | Third person camera view matrix calculated by translating by an offset before calculating view matrix (camera.cpp Line 68-73) and can switch between first and third person using 1 and 2 keys (coursework.cpp Line 372-376). |
| 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.). | Position of camera is locked onto a y level of 0 so it cannot fly everywhere unless the player has pressed space initiating a jump (camera.cpp Line 34-36 and Line 41-51) |
| LO3: Use of shaders to apply parameter driven effects within the scene, e.g., light properties controlled using camera/character position. | Lights change colour based on position of player (light.cpp Line 51-68) |