# Computer Graphics Coursework – Self Assessment Document

**Name:** *Louis* Michaelides **ID number:** *22468581*

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 | I used GLM’s glm::vec3 to define positions, rotations, and scales in my Object struct, enabling easy 3D manipulation. For transformations, I created glm::mat4 translation, rotation, and scaling matrices using custom Maths functions, then combined them as model = translate \* rotate \* scale to represent each object’s transformation in world space.  Example:  //Object structure struct Object  { glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f);  glm::vec3 rotation = glm::vec3(0.0f, 1.0f, 0.0f);  glm::vec3 scale = glm::vec3(1.0f, 1.0f, 1.0f); float angle = 0.0f; std::string name; }; |
| LO2: Application compiles and runs without alterations to the source code of CMake file. | Paste a screenshot of your application below |
| LO3: Implementation of shaders to apply appropriate textures to objects. | I implemented GLSL vertex and fragment shaders to handle UV mapping and apply textures correctly to 3D models in the scene |
| 52, 55, 58 | LO1: Basic use of translation, rotation and scaling transformations. | I implemented translation, rotation, and scaling within the rendering loop using a custom Object structure, where each object holds its own position, rotation axis, angle, and scale.   I stored multiple cube instances (floor, walls, ceiling) in a std::vector<Object> and manually positioned them in world space.   Transformations were applied using glm::translate(), glm::rotate(), and glm::scale() in that order to build the model matrix, which was passed as a uniform to the vertex shader to render each cube with its unique transformation. |
| LO1: Implementation of glm library functions for calculating view and projection matrices. | I used GLM to calculate the camera’s view and projection matrices in a custom Camera class storing position, target, up vector, and FOV. The view matrix is computed with glm::lookAt() for dynamic first-person movement, while the projection matrix uses glm::perspective() with chosen FOV, aspect ratio, and clipping planes for realistic depth. Both matrices update every frame and are passed as uniforms to the shader to render the scene from the camera’s perspective. |
| LO2: 3D virtual world has been created using instances of a single object type. | I created the 3D virtual world by adding multiple copies of a single object type, creating my basic room. |
| LO3: Use of shaders to apply dynamic lighting from point light sources |  |
| 62, 65, 68 | LO1: Implementation of students own functions for calculating view and projection matrices. | I implemented my own functions to calculate the view and projection matrices without relying on GLM’s built in lookAt or perspective functions.   For the view matrix, I manually computed the cameras forward, right, and up vectors, then constructed the matrix using dot and cross products. For the projection matrix, I used the field of view, aspect ratio, and near/far planes to build the perspective matrix from scratch using the standard mathematical formula.   This demonstrates my understanding of the underlying maths behind camera transformations.  Example:  **calculateViewMatrix**  glm::mat4 Camera::calculateViewMatrix(const glm::vec3& eye, const glm::vec3& target, const glm::vec3& up) {  glm::vec3 f = glm::normalize(target - eye);  glm::vec3 r = glm::normalize(glm::cross(f, up));  glm::vec3 u = glm::cross(r, f);  glm::mat4 view(1.0f);  view[0][0] = r.x; view[1][0] = r.y; view[2][0] = r.z;  view[0][1] = u.x; view[1][1] = u.y; view[2][1] = u.z;  view[0][2] = -f.x; view[1][2] = -f.y; view[2][2] = -f.z;  view[3][0] = -glm::dot(r, eye);  view[3][1] = -glm::dot(u, eye);  view[3][2] = glm::dot(f, eye);  return view;  }  **calculateProjectionMatrix**  glm::mat4 Camera::calculateProjectionMatrix(float fov, float aspect, float near, float far) {  float tanHalfFOV = tan(fov / 2.0f);  glm::mat4 proj(0.0f);  proj[0][0] = 1.0f / (aspect \* tanHalfFOV);  proj[1][1] = 1.0f / (tanHalfFOV);  proj[2][2] = -(far + near) / (far - near);  proj[2][3] = -1.0f;  proj[3][2] = -(2.0f \* far \* near) / (far - near);  return proj;  } |
| LO2: 3D world created using multiple object types. |  |
| LO2: Users can navigate the virtual world using keyboard and mouse inputs. | I have successfully been able to transverse my 3D world using two input functions: keyboardInput() and mouseInput().   * With the keyboard, I control movement using the W, A, S, and D keys. The camera moves in the direction of its front and right vectors, scaled by deltaTime and a sprintMulti if I'm holding Left Shift.(Sprint function) * With the mouse, I track the cursor's movement relative to the center of the screen, reset it each frame, and use the offset to adjust the camera’s yaw and pitch. I then recalculate the camera’s direction vectors with calculateCameraVectors(), which lets me freely look around. |
| LO3: Use of shaders to apply dynamic lighting from different types of light sources. |  |
| 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. |  |
| 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.). |  |
| LO3: Appropriate implementation of normal and specular maps. |  |
| 85, 90, 100 | LO1: Use of quaternions to calculate view matrix. | I implemented quaternion-based camera orientation by converting Euler angles (pitch and yaw) into a quaternion. I applied spherical linear interpolation (SLERP) to smoothly update the camera’s orientation. Then, I constructed the view matrix by multiplying the quaternion’s rotation matrix with a translation matrix based on the camera position. Finally, I extracted the camera’s right, up, and front vectors from the view matrix to use for movement and rendering.  This approach allowed me to calculate the view matrix using quaternions effectively.  Example:  // Convert Euler angles (pitch, yaw) to a new quaternion orientation  Quaternion newOrientation(-pitch, yaw);  // Smoothly interpolate between current and new orientation using SLERP  orientation = Maths::SLERP(orientation, newOrientation, 0.2f);  // Build the view matrix by combining rotation (quaternion) and translation (camera position)  view = orientation.matrix() \* Maths::translate(-eye);  // Extract camera basis vectors from the view matrix for movement and rendering  right = glm::vec3(view[0][0], view[1][0], view[2][0]);  up = glm::vec3(view[0][1], view[1][1], view[2][1]);  front = -glm::vec3(view[0][2], view[1][2], view[2][2]); |
| LO1: Use of SLERP to smooth out changes in camera direction. | I used SLERP (Spherical Linear Interpolation) to smoothly interpolate between the camera’s current orientation and a new target orientation. This prevents sudden jumps and creates fluid camera movement.  Example:  // Calculate target orientation quaternion from Euler angles  Quaternion newOrientation(-pitch, yaw);  // Smoothly interpolate between current and target orientation  orientation = Maths::SLERP(orientation, newOrientation, 0.2f);  // Update view matrix with interpolated orientation  view = orientation.matrix() \* Maths::translate(-eye); |
| LO2: Implementation of a third person camera with the ability to switch between first and third period 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.). |  |
| LO3: Use of shaders to apply parameter driven effects within the scene, e.g., light properties controlled using camera/character position. |  |