**Computer Graphics - Project Report Template**

|  |  |
| --- | --- |
| **Name:** | Aaron Dinesh |
| **Student ID:** | 203326661 |
| **Declaration:** | See plagiarism declaration on the next page |
| **Youtube link:** |  |

|  |  |
| --- | --- |
| **Student Name:** | **Aaron Dinesh** |
| **Student ID Number:** | **20332661** |
| **Assessment Title:** | **Computer Graphics Final Project** |
| **Lecturer (s):** | **CAROL O'SULLIVAN** |
| **Date Submitted** | **22/12/23** |

I hereby declare that this assessment submission is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I hereby declare that I have not shared any part of this submission with any other student or person.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at:  <http://www.tcd.ie/calendar>

I have also completed the Online Tutorial on avoiding plagiarism ‘Ready, Steady, Write’, located at <http://tcd-ie.libguides.com/plagiarism/ready-steady-write>

I am aware that the module coordinator reserves the right to submit my exam to Turnitin and may follow up with further actions if required should I be found to have breached College policy on plagiarism

**A drawing of a person's head

Description automatically generated**

**Signed:**

|  |
| --- |
| **Required feature 1: 3-dimensional objects and views** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* Anywhere in the video (0:00) |
| *Describe how you implemented it:* The vertices are stored in a GLTF file along with their indices. The vertices are read into a Vertex Buffer Object (VBO). The indices are read into an Element Buffer Object (EBO). The EBO allows us to reduce the number of vertices in the VBO by allowing vertices to be referenced by one number. In the case that two triangles shared some number of vertices, we wouldn’t need to store all 6 vertices; we would only need to store 6 integer indices. Then the VBO and EBO could all be passed to the glDrawElements function. |
| *Pseudocode:*  VAO.Bind()  VBO <- readVertices(gltfFile)  EBO <- readIndices(gltfFile)  VAO.Bind()  glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0) |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Victor Gordon’s OpenGL Tutorial <https://youtu.be/45MIykWJ-C4?si=_ssoOqn0G-FiPKyE> 2. GLAD.h 3. GLFW.h |

|  |
| --- |
| **Required feature 2: User interaction and camera-control** |
| *Screenshot(s) of feature:*    |  | |
| *Video timestamp:*  Anywhere in the video (0:00) |
| *Describe how you implemented it:* I implemented a camera object that accepted the screen width and height and starting position upon construction. I then updated the camera matrix using the glm LookAt function. This allows me to create the view matrix that represents the camera looking at a particular object from a certain position. This matrix is then passed to the vertex shader. I then wrote a function that handles keyboard inputs and changes the camera’s position in 3D space. |
| *Pseudocode:*  *Camera = createCamera(WIDTH, HEIGHT, POSITION)*  *Camera.Handle\_Inputs()*  *camMat = Camera.Update\_Cam\_Matrix()*  *Camera.Upload\_Matrix\_to\_Shahder(camMat)* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Victor Gordon’s OpenGL Tutorial <https://youtu.be/45MIykWJ-C4?si=_ssoOqn0G-FiPKyE> 2. GLAD.h 3. GLFW.h |

|  |
| --- |
| **Required feature 3: A Hierarchical animated Zombie/Robot etc. character or object relevant to the theme** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* I used hierarchical animation on the helicopter. The reason this helicopter is in the scene is because I imagine this to be set in a zombie apocalypse in Dublin and that the military have deployed their helicopters to monitor the situation. So, I thought that a helicopter would be the best showcase of hierarchical animation since that is the only way to animate this properly.  The helicopter body and its rotor and two separate objects and are in a parent-child relationship, with the parent being the helicopter body and the rotor being the child. So whatever transformations I do to the parent get propagated to the child. This was achieved by having one matrix that defined a transformation for the parent, then the child would apply its own offset to define the transformation in its frame and would then apply it to itself. In this way the rotor and the body of the helicopter would always keep the same relative position to each other. |
| *Pseudocode:*  *//Define the transformation in the world frame (ie the frame of the helicopter)*  heliBodyTransform = someTransformation()  //Transform the helicopter transform matrix into the frame of the rotor  heliRotorTransform = to\_rotor\_frame(heliBodyTransformation, rotorTransform, offset)  //Apply the Transform  heliBody.applyTrans(heliBodyTransform)  heliRotor.applyTrans(heliRotorTransform)  //Compute the total global transformation for the rotor  def to\_rotor\_frame(heliBodyTransformation, rotorTransform, offset):  return heliBodyTransformation\*(translate(offset)\*rotorTransform); |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Hierarchical Transformations (Lecture 10) 2. GLAD.h 3. GLFW.h 4. GLM.h |

|  |
| --- |
| **Required feature 4: A crowd of moving Zombies/Robots etc.. characters or objects** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* I had two groups of zombies in my scene. These were actually 6 individual models of zombies that were grouped in 2 groups of 3. I used a struct to hold the objects, their shaders and various parameters for a group. Then I implemented a state machine to control the movement of the zombies as well as the direction in which they were facing. This was also animated in a hierarchical style where only one matrix was used to move the 3 zombies in each group, after their respective offsets have been applied. |
| *Pseudocode:*  *ZombieATransMat <- some\_transformation*  *ZombieBTransMat = offsetTrans(ZombieATransMat, zombieBOffset)*  *ZombieCTransMat = offsetTrans(ZombieATransMat, zombieCOffset)*  *ZombieA.applyTrans(ZombieATransMat)*  *ZombieB.applyTrans(ZombieBTransMat)*  *ZombieC.applyTrans(ZombieCTransMat)* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Hierarchical Transformations (Lecture 10) 2. GLAD.h 3. GLFW.h 4. GLM.h |

|  |
| --- |
| **Required feature 5: texture-mapping your scene and/or characters using image files** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* I read the locations of the diffuse colour texture and the metallic roughness texture from the GLTF file. Once these paths were read, I could load the textures using the STB Image library. These textures were then bound to a texture unit in OpenGL when they needed to be drawn. The UV mapping for these textures were also read from the GLTF file. These Texture UVs were stored along with the vertices in the VBO. |
| *Pseudocode:*  *imagePaths <- read\_GLTF\_Textures(gltfFile)*  *textures <- loadImages(imagePaths)*  *VBO <- readUVCoords(gltfFile)*  *//On Draw call do the following*  *VAO.Bind()*  *VBO.Bind()*  *//This binds the textures to the relevant texture units in the opengl context*  *Textures.Bind()* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. GLAD.h 2. GLFW.h 3. STB Image Library <https://github.com/nothings/stb/blob/master/stb_image.h> 4. Victor Gordon’s OpenGL Tutorial <https://youtu.be/45MIykWJ-C4?si=_ssoOqn0G-FiPKyE> |

|  |
| --- |
| **Required feature 6: implementation of the Phong Illumination model** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* This was completely done in the shader. Since it is running on the GPU computations are incredibly fast. To begin I calculated the diffuse contribution to the light by calculating the dot product between the light direction and the normals. This was bounded between 0 and infinity. An ambient light factor was added to this to account for various scattering effects that would make the scene brighter. The specular reflection factor was calculated using the dot product between the view direction and the reflection direction of the light. This was again bounded between 0 and infinity, and was then raised to a certain power “n”. This determined whether the specular reflection would appear more point-like (corresponding to higher n) or if it would appear more diffuse (corresponding to low n). The specular reflection factor was then multiplied by a chosen specular light amount to yield the final specular reflection.  The colour from base colour texture was multiplied with the sum of the diffuse and ambient lighting terms. The specular reflection was multiplied by a roughness map that was read in as a texture. The result of these two multiplications were multiplied together to yield the final colour at that fragment.  The result of the specular reflection can be seen on the lamp post as well as on the spire behind it as a white sheen. The diffuse and ambient terms is what allows the colour of the textures to appear bright on the screen. |
| *Pseudocode:*  *//All this code is implemented in the fragment shader*  *//This code is for directional light but it can be adapted for point and spotlight*  *ambientFactor <- 0.1f*  *lightDirection = Normalize(lightDirection)*  *diffuseFactor = max(dot(lightDirection,normal),0)*  *speculatLight <- 0.5f*  *viewDirection = Normalize(cameraPosition - currentPosition)*  *reflectDirection = reflect(-lightDirection, normal)*  *specularFactor = power(max(dot(viewDirection, reflectDirection),0),16)*  *totalLight = diffuse\_color\_from\_texture\*lightColor\*(diffuseFactor + ambientFactor) + roughness\_from\_texture\*specularFactor\*specularLight* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Victor Gordon’s OpenGL Tutorial <https://youtu.be/45MIykWJ-C4?si=_ssoOqn0G-FiPKyE> 2. OpenGL Documentation[*https://docs.gl/*](https://docs.gl/) |

**NOTE: Please insert another table for any additional Advanced Features.**

|  |
| --- |
| **Advanced feature 1** |
| *Description/name of feature:* Audio Soundtrack |
| *Screenshot(s) of feature:* Not applicable |
| *Video timestamp:* |
| *Describe how you implemented it:* Sound was implemented using the miniaudio.h library. This is a header only library that allows you to work with sound on various operating systems and machines. I began by setting up the various parameters such as the output format, the number of channels and the sample rate. Upon successful initialization, the audio device was started, which started the playback. |
| *Pseudocode:*  *audioDeviceParameters = loadParamters()*  *device = initDevice(audioDeviceParameters, soundFile)*  *if(device.successful){*  *device.start()*  *}* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. *Miniaudio.h* [*https://miniaud.io/docs/manual/index.html*](https://miniaud.io/docs/manual/index.html) |

|  |
| --- |
| **Advanced feature 2** |
| *Description/name of feature:* My Own Models |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* The models were inspired by models from sketch fab. These models were modelled in blender, with some of their textures changed to suit the theme. The helicopter rotors are a perfect example of this. I imagined this scene to be set in the future so I imagined the helicopter to have carbon fibre rotor blades for its strength and light weight properties. Hence the black colour in the texture.  I also modelled the spire from Dublin along with a custom metallic texture I made. I also added some noise and dirt to the spire, so it looks a bit old and run down.  In the first image you can see a building that I had created and then destroyed to make it look like it was rubble. Along with this I also placed some planks of wood at random angles, this I imagined to be some of the rafters of the building that collapsed. |
| *Pseudocode:* No pseudo code since this was all done in blender. |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. SketchFab 2. Material Inspiration <https://www.youtube.com/@RyanKingArt> |

|  |
| --- |
| **Advanced feature 3** |
| *Description/name of feature:* Realistic Fog |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* This was done in the shader. Here we gradually blend objects that are far away from the camera with the colour of the sky. In this way, we get a hazy effect, with far away objects gradually blending in with the sky colour and disappearing into the fog. Once we got the distance to the vertex, we could then plug this distance into a function that will return a visibility factor. Using this factor, we can then blend between the sky colour and the colour from the lighting calculations. |
| *Pseudocode:*  *//All this code was written in the shader.*  *distance = distance(vertexLocation - cameraPosition)*  *//We clamp the visibility factor between 0 and 1 since values outside that range*  *//don’t make sense in this context*  *visibilityFactor = clamp(-power(distance\*density, gradient), 0.0, 1.0)*  *finalColour = mix(vec4(skyColour,1), lightingColour, visibility)* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. GLAD.h 2. GLFW.h 3. Fog tutorial <https://youtu.be/qslBNLeSPUc?si=M2YydVi-UU_o75ss> |

|  |
| --- |
| **Advanced feature 4** |
| *Description/name of feature:* Blinn-Phong Illumination Model |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* Instead of just using the view direction of the reflection direction to compute the specular highlights, I computed the halfway vector between the view direction and the lighting direction. Using this halfway vector and the normals I could calculate the amount of specular reflection needed. This helps with giving “fuller” specular reflections as well as a more realistic reflection over all |
| *Pseudocode:*  *//This was implemented in the shader*  *//diffuse and ambient factor calculations remain the same.*  *halfwayVec = normalize(viewDirection+lightDirection)*  *specularFactor = power(max(dot(normal, halfwayVec),0), 16)*  *diffuse\_color\_from\_texture\*lightColor\*(diffuseFactor + ambientFactor) + roughness\_from\_texture\*specularFactor\*specularLight* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. Victor Gordon Tutorial on Blinn-Phong lighting[*https://www.youtube.com/watch?v=-NSBP5q8nNE*](https://www.youtube.com/watch?v=-NSBP5q8nNE) |

|  |
| --- |
| **Advanced feature 5** |
| *Description/name of feature:* Spotlight Attached to the Helicopter |
| *Screenshot(s) of feature:* (This is better seen in the video) |
| *Video timestamp:* |
| *Describe how you implemented it:* When loading the helicopter body model I computed the centre of the helicopter. Then when I applied any transformations to the helicopter body, I also applied the same transformations to the centre coordinates. Then at each draw call I set the light position to be the centre of the helicopter. In this way I could get a spot light that appears to be coming from underneath the helicopter as it travels in the world |
| *Pseudocode:*  *centreHelicopter = helicopterBody.computeCentre()*  *helicopterTrans = some\_transformation*  *helicopterBody.applyTrans(helicopterTrans)*  *centreHelicopter = helicopterTrans\*centreHelicopter*  *lightPosition = centreHelicopter*  *//Upload the light position to all the shaders at each draw call.* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. GLM.h |
| **Advanced feature 6** |
| *Description/name of feature:* Helicopter and Zombie State Machine |
| *Screenshot(s) of feature:*  Helicopter State Machine:    Zombie State Machine: |
| *Video timestamp:* |
| *Describe how you implemented it:* I created an enum that would contain all the states the helicopter or the zombies would be in. Then using a switch case as shown in the screenshots above I could run different bits of code for each of the different states the helicopter or the zombie could be in. Also within each state, when a certain condition was reached, I would switch to another state. The current state was stored inside a struct that stored all the relevant variables for the helicopter or the group of zombies. |
| *Pseudocode:*  *//This is the code for the helicopter, but the zombie code is largely the same*  *switch(helicopter.state)*  *case STATE\_1:*  *do\_something*  *if(condition\_for\_state\_2):*  *helicopter.state = STATE\_2*  *case STATE\_2:*  *do\_something\_else*  *if(conditition\_for\_state\_3):*  *helicopter.state = STATE\_3*  *//This pattern of code continues for however many states you have.* |
| *Credits (e.g., list source of any tools, libraries, assets used):*   1. GLM.h |

//Collision detection for helicopter and zombie

//Individually controlled models

//Attach the camera to the helicopter.