**Computer Graphics - Project Report Template**

|  |  |
| --- | --- |
| **Name:** | Minjuan Luo |
| **Student ID:** | 20313326 |
| **Declaration:** | I here by declare that this is my individual assignment and I did not show my work to anyone or make it available/accessible to anyone in any way |
| **YouTube link:** | Awaiting link |

|  |
| --- |
| **Required feature 1: 3-dimensional objects and views** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Main Program: Handles the creation and rendering of 3D objects. This involves setting up vertex data, possibly using vertex buffer objects (VBOs), and defining object geometries.*  *Vertex Shader: Transforms 3D coordinates into different spaces (model, view, and projection). It's crucial for viewing 3D objects from various perspectives.*  *Implementation: The code uses vertex attribute pointers and array buffers (glVertexAttribPointer, glBindBuffer) for defining the geometry of 3D objects. There's specific handling for textures and drawing arrays (glBindTexture, glDrawArrays).*  *Specific Logic: The code sets up vertex attributes for various properties like position, colour, texture coordinates, and normals. This is essential for defining the shape and appearance of 3D objects.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *OpenGL: A cross-language, cross-platform API for rendering 2D and 3D vector graphics. Used here for the core rendering engine.*  *GLM (OpenGL Mathematics): A header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. Used for matrix transformations and other mathematical operations related to 3D graphics.*  *FreeGLUT: An open-source alternative to the OpenGL Utility Toolkit (GLUT) library. Used for managing windows, inputs, and events.* |

|  |
| --- |
| **Required feature 2: User interaction and camera-control** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Main Program: Handles input events for camera control, such as keyboard and mouse inputs to move or rotate the camera.*  *Implementation: The code includes callback functions for keyboard and mouse events, manipulating the camera or view matrix.*  *Specific Logic: Callback functions are registered for keyboard and mouse events, which are then used to manipulate the camera or view matrix. This allows for interactive control over the camera's position and orientation in the 3D scene.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *GLFW: A library for OpenGL, Vulkan, and other graphics API development, providing a simple API for creating windows, contexts, and handling inputs. Used for managing user inputs and camera control mechanisms.*  *GLM: As above, for mathematical operations related to camera movement and view matrix calculations.* |

|  |
| --- |
| **Required feature 3: A Hierarchical animated Zombie/Robot etc. character or object relevant to the theme** |
| *Screenshot(s) of feature: DNF* |
| *Video timestamp: DNF* |
| *Describe how you implemented it:*  *Model Structure:*  *The model is split into a hierarchy of interconnected "bones" or nodes, each representing a part of the model. For example, in a human character, bones might represent limbs, the torso, the head, etc. Each bone has a transformation relative to its parent bone. The root bone typically represents the base of the hierarchy.*  *Animation:*  *Animations are created by defining keyframes, which specify the transformation (translation, rotation, scale) of each bone at specific points in time. The animation system interpolates between these keyframes to produce smooth motion.*  *Rendering:*  *During rendering, each bone's final transformation matrix is calculated by combining its local transformation with the transformation of its parent bone, all the way up to the root bone. This hierarchical transformation ensures that movements of parent bones naturally propagate to their children.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *OpenGL: Used for rendering the model.*  *GLM (OpenGL Mathematics): Provides the necessary matrix and vector operations for calculating transformations.*  *Assimp (Open Asset Import Library): Often used for importing animated models and their bone structures from various file formats.*  *Animation Libraries (e.g., FBX SDK, COLLADA): These libraries may be used for handling the animation data, especially if the animations are complex or involve advanced features like inverse kinematics.* |

|  |
| --- |
| **Required feature 4: A crowd of moving Zombies/Robots etc. characters or objects** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Main Program: Updates the position or transformation of models over time or in response to user interactions*  *Implementation: The code updates model matrices for moving models, possibly involving translation, rotation, or scaling transformations.*  *Specific Logic: The code likely includes updates to the model matrix for each object, incorporating translations, rotations, and scaling based on some logic (e.g., time-based animation, user input).* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *GLM: For all the mathematical calculations related to model movement and transformations.*  *OpenGL: For rendering the updated positions and orientations of the models.* |

|  |
| --- |
| **Required feature 5: texture-mapping your scene and/or characters using image files** |
| *Screenshot(s) of feature:* |
| *Video timestamp* |
| *Describe how you implemented it:*  *Fragment Shader: Applies textures to the fragments (pixels) of the 3D objects. Requires texture coordinates from the vertex data.*  *Main Program: Loads texture images and passes them to the GPU.*  *Implementation: Textures are bound and applied to objects using texture buffers and uniform variables in the shader.*  *Specific Logic: Textures are bound to texture units and applied to the fragments of the 3D objects. Texture coordinates are used to map the image correctly onto the object's surface.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *OpenGL: Used for its texture mapping capabilities, enabling the application of images to the surfaces of 3D models.*  *Image Loading Library (e.g., STB Image, FreeImage): While not directly mentioned in the snippet, these types of libraries are commonly used for loading and processing texture images.* |

|  |
| --- |
| **Required feature 6: implementation of the Phong Illumination model** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Fragment Shader: Calculates the lighting on each fragment based on the Phong reflection model*  *Implementation: The shader code calculates lighting based on ambient, diffuse, and specular components, considering light source properties and surface normals.*  *Specific Logic: The code calculates the ambient, diffuse, and specular components for each fragment based on light positions, surface normals, and viewer position. This determines how light interacts with the surfaces of objects.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *OpenGL: For implementing the lighting calculations in the shaders.*  *GLM: Used in the calculation of lighting effects based on the Phong model.* |

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

|  |
| --- |
| **Simulated dynamics or physics of any sort** |
| *Description/name of feature:* **Collision Detection** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Main Program: Implements algorithms to detect intersections between objects and the environment.*  *Implementation: The code includes functions for detecting collisions between objects, using bounding volumes and obstacle position for restriction.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *Custom Algorithms: The specific method of collision detection is often custom-developed, using basic principles of geometry and physics.*  *OpenGL and GLM: Used for rendering and mathematical calculations related to the positions and dimensions of objects for collision detection.* |

|  |
| --- |
| **Intelligent Characters** |
| *Description/name of feature:* **Zombie chasing Human** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*  *Main Program: Logic to make one model (e.g., an NPC) follow or chase another model or a set point.*  *Implementation: Logic is implemented to make one model follow or chase another model, adjusting its position and orientation.*  *Specific Logic: This feature involves algorithms where one model (like an NPC) dynamically changes its position and orientation to follow or chase another model or a target point in the scene.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *GLM: For vector calculations necessary in implementing chasing behaviour (e.g., calculating direction vectors).*  *OpenGL: For rendering the movement and updates of the chasing and target models.* |

|  |
| --- |
| **Great models that you made yourself** |
| *Description/name of feature:* **The giant monster and flying bats** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it: using blender to build model* |
| *Pseudocode: none* |
| *Credits (e.g., list source of any tools, libraries, assets used): Blender* |

|  |
| --- |
| **Procedurally generated terrain or meshes** |
| *Description/name of feature:* **plain grass terrain** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:*   1. *Perlin Noise Generation:*   *The code uses Perlin noise, a gradient noise function, to generate realistic terrain elevations. This is evident in the perlinNoise function.*  *Functions like lerp (linear interpolation), fade (a smoothing function), and grad (gradient calculation) support the noise generation.*   1. *Terrain Mesh Construction:*   *The generateTerrain function creates a grid of vertices, where each vertex's height (y-coordinate) is determined by the Perlin noise function. The terrain mesh is built by defining vertices for a grid and calculating their respective normals and texture coordinates.*   1. *OpenGL Integration:*   *Vertex Buffer Objects (VBOs) and a Vertex Array Object (VAO) are used to manage and render the terrain data. The terrain's vertex positions, normals, and texture coordinates are sent to the GPU via VBOs.* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):*  *Perlin Noise Algorithm: Developed by Ken Perlin, this algorithm is widely used in computer graphics for generating organic-looking textures and terrains.*  *OpenGL: Used for all the rendering tasks, including managing buffers and executing the rendering pipeline.*  *GLM (OpenGL Mathematics): Likely used for any mathematical operations not shown in the snippet, such as matrix transformations for the terrain model.*  *C++ Standard Library: For general-purpose functionalities like array management (permutation array).* |

|  |
| --- |
| **Advanced Texturing Effects** |
| *Description/name of feature:* **multi-texturing** |
| *Screenshot(s) of feature:* |
| *Video timestamp:* |
| *Describe how you implemented it:* |
| *Pseudocode:* |
| *Credits (e.g., list source of any tools, libraries, assets used):* |