Hao Dinh Mai
Paul Kim
CSE 165 - Object Oriented Programming
Final Project Report

Construction of a Survival Game using OpenGL

The mission statement for this project is to use the OpenGL library along with C++ object oriented programming language to compose and render a 2D graphical game. The construction methodology involves the creation of an engine consisting of a virtual environment with texture, shaders, vertexts and sprites in order to render a map along with characters within the game. This approach serves as a starting point, and allows for later game logic integration. Given that the standard graphic libraries are initially implemented as support for the actual game logic and functionality - from context, the game construction method suffices as the bottom-up approach. The game folder itself contains the classes that consist of the player's functionality, input motion for the game, description and logic of the map, the logic to how the player interacts with the map, the winning condition, and obstacles that the player must avoid to achieve victory.

The composition of the game engine relies heavily upon graphical libraries such as OpenGl, OpenGL Extension Wrangler (glew), OpenGl mathematics library (glm), OpenGl Direct media Layer, and PicoPNG. Each serves as a building block within the game engine to substantiate window rendering for the game, map creation, frame rates and frame transition management, loading in PNG files as part of the game, and keystroke input for maneuver functionality. More specifically, within the custom window files, it utilizes SDL2 to create a window compatible with OpenGL and set the window size in terms of length and width, and ensure that the buffer windows are properly swapped for smooth game play. The Camera2d file within the engine support for the scaling and transition of the player as they move about the game map, this file includes the OpenGL mathematical library that computes the view matrix, tracking the camera view of the game when the player is moving, changing orientation and transition about the map during gameplay - this file will prevent offscreen play.

To supplement the game with visuals in terms of rendering image and text files to the desired portrayal of characters and map layouts, the game engine will utilize the picoPNG library that intertwines with the file load , load image, texture, manage texture and the texture cache header and cpp files as a way to handle picture data junction into the game's graphic. First the data is imported into the memory of the system that will be broken down into binary which will be converted into the vectors and then shader components that are necessary for the construction of the characters and map. More specifically, within the load file implementation, it takes in files and loads them into the memory, converting it into binary which will later be converted into strings with reference to the file path from the managed texture file- this allows for the text file to be loaded into the game as the map. The text file will then be assigned pixel values and given textures as well as set parameters to define its size, shapes and color. The load image implementation corresponds to that of the load file class, yet will instead take in PNG files and convert them into pixel data. Once the pixel is stored within a vector, the picoPNG library will ensue to decode the pixel data and return the texture information. From the texture file implementation, it will then receive the decoded pixel data and convert it as OpenGL texture - these textures are assigned values and are bind - the procedure of setting the parameter to the texture while also subsampling the image derived from the texture by multiple scales so it corresponds to the initially assigned parameter to match the ratio of the OpenGL window. Once completed, the shader implementation will compile every morsel of textures and link it to the OpenGL window to visually render the desired image as some component of the game.

Specifically, the game consists of the PNG representing the character - the image that will be incorporated into the game and is managed by the bitmap 2d file which inherits the functionality of the file loader, load image, texture and texture cache to set the condition for the character and load the parameter with a visual representation into the OpenGL window. Within the bitmap2dbatch file, it contains a function to first set the vertex buffer ID to 0, to check if the buffer has been rendered, then set the source of the file by linking it to the project file sourcing the file path to the PNG location, and then ultimately draw the component into the OpenGL window. In order to incorporate complexity and definition into the SDL windows, the timing, vertex, and shader file adds to the engine with the implementation of the OpenGL Extension Wrangler and the direct media layer. The timing implementation accounts for the frame rates of the game within the SDL windows since its initial execution tracking each frame. The shader and vertex of the game are complements to one another, for the purpose of exacting specific shapes within the 2D game and specifying the constraints such as size or color attribute to how they are rendered to the window screen. The final implementation to the game that allows the user to execute keystrokes as means of controlling the game is the key file implementation, by employing the glm library will employ the standardize function implementation to assigned key value within the keyboard to the mechanics of the game, allowing the user to control the maneuver of the player within the game.

For the implementation of the game logic into the game engine, there exist a total of the six headers and seven implementation files. Within the Entity header, it declares all the entities that exist within the game via the humans and the zombies classes. This header files contains a virtual method allowing for polymorphism to be implemented as the game proceeds; hence, the derived class of zombies and humans within the game will inherit the virtual base method and such implementation will be updated in corresponds to what happens within the game. The file also has accessors including the coordinate of the objects within the game as well as a set speed for each object. There also exists collision detection which will prevent the game from crashing when the player or component of the game come into contact with each other. This collision detection algorithm within this game consists of colliding with the title function - this function constructs a box around each entity and title within the game, if these constructed boxes overlap with each other, it infers that there has been a collision. This file also contains the declaration of the function to draw the characters and map with reference from the game engine file bitmap 2d batch files. The declaration of the player, initial starting point, their texture ID, the speed, and tracker for the account of the moment made by the user is also present within the entity file. Within the human header file, it will inherit the virtual function from its parent class evident within the entity implementation taking into account only the speed and frame rate - this class will serve as the declaration for the player file within the game, for the player who identified as being human within the game. The level header file will serve as a pathway to call the file of the map, set the winning condition of the game, and employ accessors to establish the player starting point, store the level data and the coordinates of both zombies and humans within the map. Also, there exists getter functions that checks and returns the appropriate condition for game play; for example it checks for the amount of human characters within the game, the tile position, the starting position and the win conditions. The player likewise inherits from the entity class that contains the derived virtual function of zombies and humans calling for the update status, the speed as well as accounting for the keystrokes and frames within the game. The zombie header consists of inheriting the virtual method of the entity class that requires update and accounts for speed and a special condition of getting close to humans - if the zombie touches the human, the player loses the game.

The Entity class holds the logic for collision detection between tiles, center of radius, and level collision. The collision check ensures that all four corners of the box-bound checker don't overlap each other, preventing the game from crashing. The file accounts for the distance between each object, their depth of collision, and the direction in which the objects within the game collide. There is also an implementation of what happens when a collision occurs; if both radius and tile checking return true, then the health will be less than zero, which will also return true, and will be further implemented as the player losing within the Game file.The level implementation files will account for all the players within the game, check for the path to render the player image, as well as the map by calling the bitmap 2d batch files to render all the textures and pixels to draw the game image, set the color attribute, and assign the condition for collision between the player and objects within the game. The Player files will be assigned the image of the PNG designed for it via sources from the managed texture file within the engine. It will inherit the virtual method, which will be updated during the Game file - which, in turn, if collides with the zombies within the game, the player will lose.The Zombie file implementation consists of pulling in the PNG within the managed texture file within the engine and assigning it speed and color attributes. It will also have the virtual method that will be updated as the game proceeds. Additionally, the Zombie class has functionality for finding the tile that is closest to the human by using a pointer to the Human class's location represented in terms of vector coordinates. Once the player starts the game, the Zombie function will execute by obtaining the coordinate value present within the Human class and begin to follow in that direction.

The main game function integrates all of the headers and implementation classes. It initiates the OpenGL window for game startup, the win condition within the level files, and the game loop which consistently checks for the win condition, updating the conditions for both zombies and humans which incorporate polymorphism into the game and declare a specific frame rate to keep the gameplay consistent.First, the main game calls for the construction of the OpenGL library to render the screen, the bitmap 2d files to draw the map and players within the engine, including the camera 2d file to track the location of the player, as well as the keystrokes to maneuver the player across the map.Then the code proceeds to update the virtual methods within the entity class that exist across the player and zombie class implementation by checking their collision status; thus, incorporating polymorphism. First, the code checks for the current position of the current zombie and player as well as keeping track of their speed by the constant delta time. With references, with pointers, the game will then check for the collision between the zombies and humans - this check will update the base virtual method within the entity class and reference it to the level file for the set parameters.If the player succeeds in arriving at the winning point represented in the text file, then the player has won the game. However, if collision detection between humans and zombies occurs, then the player has not fulfilled the winning condition, and the game will end.