**Team: Syntax\_Error**

**Engineering Computation 24-780**

**Date: 11/19**

**Component Code Description Report**

**Intro/Exit Screen (*Sky Song*)**

The intro and exit screens are the GUI that handles the starting and ending of the game, recording the player names and scores, displaying them, and properly keep them in a .txt file. Since my component only happens at the beginning and the end of the game, I chose to not making it into a class since most functions of my component should be called only once at the beginning or end of the main loop.

The main functionality of the intro screen is to give user the option to start a new game, check leaderboard, or leave the game (exit the program completely). When the user clicks “starts new game”, the user is prompted to enter a player name. When the user clicks “Leaderboard”, all the player names and their scores are displayed on the screen. When the user clicks “Exit”, the program is terminated. The main functionality of the outro screen is to display the score of the current user at the end of the game.

The testing of my functions is mainly focused on data safety. Const-correctness is implemented for all functions and all kinds of possible inputs for player names are tested to ensure no crashes. Edge cases such as empty name or names that’s too long are checked and prevented as well. All different modules/functions in my component are self-contained in terms of FsPollDevice() and OpenGL drawing functions.

**Audio Management (*Jesse Barkley*)**

For our pirate shooter game, I was responsible for developing the audio system, which plays a key role in enhancing the overall gaming experience. I created the SoundManager class to manage all sound-related functionalities, including background music for the intro, game, and outro screens, as well as sound effects for actions like cannon firing and collisions. The class initializes by loading the necessary WAV files for each audio component, ensuring that all sounds are ready for use during gameplay.

To ensure the audio system works seamlessly, I implemented methods to play and stop each type of music and sound effect as needed. For example, the intro music plays automatically at the start of the game, transitions to the game music during active play, and switches to the outro music when the game ends. Sound effects are triggered in response to specific events, such as firing a cannon or a collision occurring, providing immediate audio feedback to the player.

Testing the SoundManager was a crucial part of my development process. I included a DebugTest method that systematically plays each sound to verify they load correctly and function as intended. Additionally, I integrated the audio system into the main game loop and conducted extensive testing by simulating various gameplay scenarios. This involved checking the responsiveness of sound effects to user inputs and ensuring that background music transitions smoothly without any delays or overlaps. Through this thorough testing and debugging, I was able to identify and resolve any issues, resulting in a reliable and immersive audio experience for our game.

**Game Objects Development 1: Ship/Ship Shooting/Ship Movement (*Nate Salazar*)**

My code is responsible for the playable pirate ship. This includes the movement of the ship and the firing of the cannons. Inherent in those responsibilities, my code renders both the pirate ship and the cannonballs. In the future my code will be responsible for checking collisions with foreign objects/enemy objects such as rocks and rival pirate ships.

Notably the cannonballs fire at 90 degrees offset from the direction the pirate ship is moving. This is to mimic pirate ships of old. The cannons move in a straight line from where they were fired and continue until they hit a foreign target or hit the edge of the screen. No additional physics are added to their movement. In gameplay it is currently planned that a single cannonball will destroy any foreign object (rock or rival pirate ship) removing said object from the game. It is possible that this will change in the future.

The code is organized into two primary classes: Pirate ship and cannon ball. The pirate ship class will be in charge of the movement and rendering of the pirate ship, although currently those have yet to be added to that class. Also in the future the pirate ship will be responsible for checking for collisions. The cannonball class is in charge of firing the cannonball and setting the trajectory of the cannonball, based on the angle of the pirate ship. The cannonball class will also be incharge of terminating itself once a cannon ball goes offscreen or hits a foreign object. Finally the cannonball class will be incharge of regulating the number of cannonballs that can be on screen at one time.

To test this code a “debugg” function was created that is called inside the main function. In this manner the main function is not responsible for running the pirate ship. Furthermore the classes allow me to isolate issues as I build in new member functions.

**Game Objects Development 2: Obstacles, Enemy Pirate Ships (*Jonathan Roberts*)**

I am responsible for randomly generating the obstacles and enemy pirate ships to scroll across the screen as the main pirate ship sails the sea forcing it to maneuver and shoot enemy ships to survive. The enemy pirate ship will be a subclass of the pirate ship parent class, and the rocks will have their own class. Each of those classes will have a constructor (self-explanatory) function, render function with randomized y coordinate to set the y distance on the window where the obstacle will scroll across, and class.Move() function that will change the x and y coordinates at a rate of speed that makes it look like the main pirate ship is sailing past them. I will also have a check collision function that registers when the main ship makes contact with the obstacles or an enemy cannonball.

The method of generating these obstacles has 2 parts. First, each obstacle type will have a vector initiated to hold them. Then, a for loop will create an instance of each class stored in the vector adjustable to how difficult we want to make the game (aka how many obstacles to create. In the game loop inside the main function, a for loop will loop over one of the vectors, create an obstacle or both (depending on difficulty), then move that obstacle across the screen, eliminating an instance as it crosses the screen or is shot by the enemy ship and generating a new one then pushing it to the back of the vector to replace it.

**Main Game Loop (*Zihao Liu*)**

The `main` function serves as the central control point for a naval battle game, integrating various modules such as UI, sound management, naval battle mechanics, enemy ship logic, and an end-screen system. It begins by initializing essential components like `SoundManager` for audio playback, `UIManager` for managing the user interface, `Cannonball` for handling the ship's cannons, `PirateShip` for rendering the player's ship, and `EnemyLogic` for enemy interactions. The game window is set up with a resolution of 800x600 pixels, and randomness is seeded using `srand()` to introduce variability, such as enemy ship placement.

The game starts with a UI-driven start screen, allowing the player to either begin the game or exit. During this phase, background music is played using the `SoundManager`. If the player decides to proceed, the program transitions into the main gameplay loop, where the player’s ship can move, fire cannonballs, and engage enemy ships. The loop continuously updates the game state, processes player inputs, and renders the necessary visual elements on-screen.

In the gameplay loop, key mechanics include controlling the player's ship movement and firing, ensuring the ship remains within screen boundaries, managing the movement and rendering of cannonballs, and interacting with enemy ships. Enemy interactions are handled by the `EnemyLogic` module, which updates their positions, checks for collisions with cannonballs, and determines their status. Victory is achieved when all enemy ships are destroyed, while defeat occurs if the player’s ship is hit or otherwise incapacitated.

The game concludes with an end screen indicating the player’s victory or defeat. This transition is managed by the `UIManager`, and appropriate sound effects are played using the `SoundManager`. This structured design ensures clear separation of responsibilities across modules, allowing for scalability and easier maintenance. Future enhancements, such as new levels or additional features, can be incorporated seamlessly into the existing framework.