**Team: Syntax\_Error**

**Engineering Computation 24-780**

**Date: 11/26**

**Beta-version Description**

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

The intro and exit screen is a crucial part of the user interface. This component is mainly responsible for starting a game, reading and writing to a .txt as the leaderboard, and allowing the user to exit the game entirely. Thus, the requirement for a user interface is solved with this component. To ensure robustness, the component is tested with various name lengths, backspacing with zero-length names, and limiting names that could be problematic, such as an empty name.

One difficulty of the component is to ensure a correct/expected name inputting after modification of other parts. When the user uses a and d to fire cannons in the main game, and when the user restarts another round, the a and d are stored in FsInKey() and will be pushed out to the name input. Although no crash happened, this function did not work as intended. After digging through potential bug points, I was able to eliminate the problem by emptying the FsInKey() buffer first before allowing the user to enter a name.

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

The sound management system was carefully developed to enhance the gaming experience by providing appropriate audio elements and ensuring their functional integration within the game. All music tracks for the intro, in-game background, and outro, as well as various sound effects, were sourced from Pixabay.com, ensuring high-quality, royalty-free audio assets. To maintain compatibility and optimal performance, the selected MP3 files were converted to WAV format, which offers better reliability for game development.

A comprehensive SoundManager class was implemented using the yssimplesound.h library. This class is responsible for loading, playing, and stopping different audio tracks efficiently. Key functionalities include managing background music transitions and triggering sound effects in response to in-game actions such as firing cannons or collisions. The system was designed to handle multiple audio channels, allowing for overlapping sounds without performance degradation.

Through careful selection of audio assets and robust technical implementation, the sound system effectively complements the visual components of the game, contributing significantly to its overall functionality and immersion. Extensive testing was conducted to verify that all audio elements operated as intended, demonstrating that the sound management system runs smoothly and achieves the desired functionality. This integration ensures that players experience a cohesive and engaging environment, where audio cues enhance gameplay and provide feedback for various actions within the game.

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

My code touches everything that has to do with the user. This includes, rendering the ship, firing the cannonballs, and moving the ship based on the driving angle, theta. The rendering is also reused for rendering enemy ships.

One challenge I ran into was rotating and translating the points of the ship. I learned later that using the transformation matrix would have simplified this issue greatly but instead took a more roundabout way by logging the hypotenuses of each point based on the origin of the ship and logging the angles of each point relative to the driving angle theta. Another challenge was making sure the cannons always fire at 90 degrees from the ship. After getting some help from the professor, I learned which version of plus/minus sine/cosine to use and was able to get my cannons consistently firing at 90 degrees to the direction of the ship, every time.

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

Obstacles and Enemy Ships are classes similar to the Friendly ships but with some key differences. The Enemy Ships are generated with a boolean parameter passed such that the Enemy Ship has a similar look and functionality to the Friendly Ship minus the user-controlled behavior, thus providing reusability in the Ship Class. The Enemy Ship instances even retain the cannon functionality to fire cannonballs at the Friendly Ship at predefined intervals. Both the Obstacles and Enemy Ships are randomly rendered to provide more robust gameplay while the Enemy Ships are scrolled across the screen to create a scene of sailing. The randomly generated number of Enemy Ships and Obstacles are both stored in a vector and rendered through a For loop in the main game’s While loop.

Building further gameplay mechanics, the main EnemyLogic class receives Enemy Ships, Obstacles, and CannonBalls. It handles the gameplay behavior of those elements when any of them collides with or is destroyed by the Friendly Ship. It does this by determining a coordinate proximity of the rendered elements and tracks their proximity in the game’s While loop.

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

Main loop contains Introduction window, which asks user for three different input options and audio manager, if clicking start, game window will pop up, and user input to control pirate ship against enemy ships, music and ship fire audio are managed through the main loop, and keep track of scores, if terminate condition meet, which is user ship crash onto the obstacle or hit by enemy ship, game outro window will pop up and ask user to exit or start new game. Current round score will be saved.

The main loop executes in two nested loops where the outer loop handles game sessions (start screen to game end) and the inner loop manages real-time gameplay, processing player input (movement and shooting), updating enemy states and projectiles, performing collision detection, managing game objects (ships, cannonballs, obstacles), handling rendering through OpenGL, controlling sound effects, maintaining score, and checking victory/defeat conditions.