CSC 413 Term Project Documentation

Summer Semester 2024

Student name: Binrong Zhu

Student ID: 923654521

Class : CSC413-02

GitHub Repository Link:

https://github.com/csc413-SFSU-SU2024/csc413-tankgame-Pandaz00.git

Details

Each student must write documentation for their term project. This document will cover the entire term project from beginning to end.

Your documentation MUST contain the following sections:

1. Title page containing
    1. Student's Name
    2. Class, Semester
    3. A Link to your repository.
2. Introduction
    1. Project Overview (the focus of the term project)
    2. Introduction of the Tank game (general idea)
3. Development environment.
    1. The version of Java Used
    2. IDE Used
    3. Were there any special libraries or special resources where you got them from and how to install them.
4. How to build or import your game in the IDE you used.
    1. Note saying things, like hitting the play button and/or clicking import project, is not enough. You need to explain how to import and/or build the game.
    2. List what Commands that were running when building the JAR. Or Steps taken to build jar.
        1. These can be the steps done either at the command line or in IntelliJ.
    3. List commands needed to run the built jar
        1. These can be the steps done either at the command line or in IntelliJ.
    4. How to run your game. As well as the rules and controls of the game.
    5. Assumptions Made when designing and implementing your game.
    6. Tank Game Class Diagram
    7. Class Descriptions of classes implemented in the Tank Game
        1. No need to over-explain but state the purpose of each class.
    8. Self-reflection on the Development process during the term project
    9. Project Conclusion.

When completing this documentation please make sure you are concise but that each section contains enough information. Point penalties will be added for insufficient information or missing sections.

The final documentation ***<u>MUST</u>*** be submitted in PDF FORM. Please submit your final documentation in PDF form to Canvas by the deadline at the heading of this section (note the repetition, must be important). ***<u>Submitting documentation that is not a PDF will cause a 10-point penalty.</u>***

1 Introduction

1.1 Project Overview

This program is a simple tank game program, written in Java language, running in the JDK environment. This game is in two-player mode, at the beginning of the game, the two sides of user respectively through the keyboard W, A, S, and D keys and up, and down, the left and right keys to manipulate the tank move, turn, press the space bar, and enter key shooting, and the other tank to engage, until the other tank can win. This procedure includes 25 classes, music pictures, and other game resources.

1.2 Introduction to my tank game

1.2.1 Start Game

The main function class of this program is the Game class. After compiling and running this class directly, the following interface will appear:

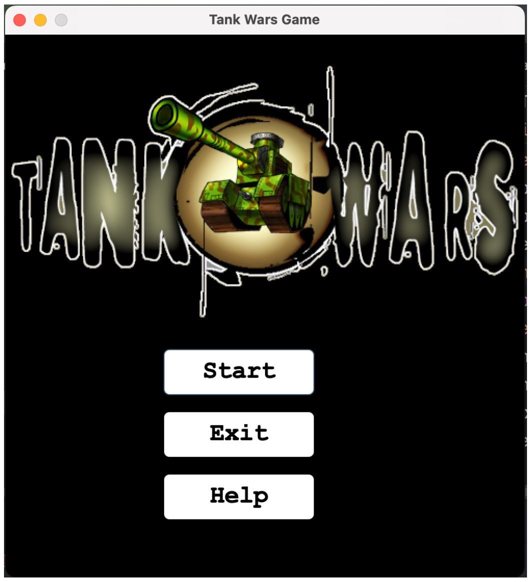
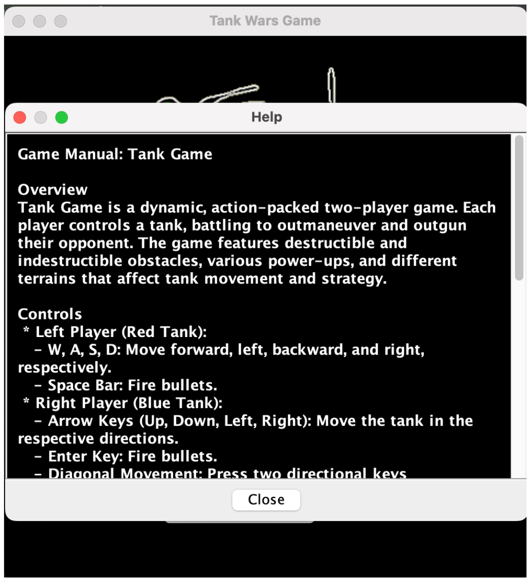


Image 01 – Start Page



Game Manual: Tank Game

Overview
Tank Game is a dynamic, action-packed two-player game. Each player controls a tank, battling to outmaneuver and outgun their opponent. The game features destructible and indestructible obstacles, various power-ups, and different terrains that affect tank movement and strategy.

Controls
 * Left Player (Red Tank):
   - W, A, S, D: Move forward, left, backward, and right, respectively.
   - Space Bar: Fire bullets.
 * Right Player (Blue Tank):
   - Arrow Keys (Up, Down, Left, Right): Move the tank in the respective directions.
   - Enter Key: Fire bullets.
   - Diagonal Movement: Press two directional keys

Image 02 – Introduce Text

The interface shown in the figure above is the program's main interface. Users can play or quit the game by clicking on the options in the figure or clicking Help to view the game instructions document, which contains a brief introduction, game environment, and operation instructions. Click Start to start the game. The game interface is shown as follows:

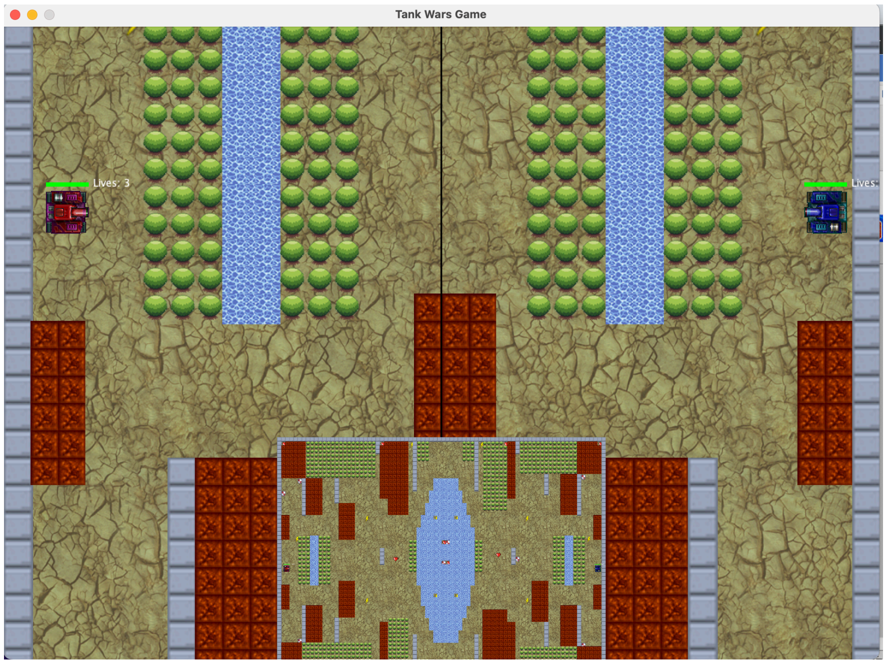


Image 03 – The Game Page

In the interface shown above, the left and right sides of the split screen are controlled by one player each, with the red side attacking from left to right and the blue side attacking from right to left. There is a mini-map located in the lower center of the screen, easy to view the overall situation. The red tank presses W/S/A/D to control the tank to move up and down, left and right and turn, and presses the space bar to shoot. Blue tank press up and down the left and right keys to control the tank's direction, and press the Enter key to shoot.

This game also has a supply, hit, above-the-tank reduced health, and heart-shaped items that can replenish health. There are also shield-shaped items that can be used to fend off enemy tanks for 8 seconds; Lightning-shaped items increase speed for 5 seconds.

2 Development Environment

2.1 OpenJDK 21.0.2 2024-01-16

OpenJDK Runtime Environment (build 21.0.2+13-58)

OpenJDK 64-Bit Server VM (build 21.0.2+13-58, mixed mode, sharing)

2.2. IDE: IntelliJ IDEA 2023.3.3 (Ultimate Edition)

2.3. Java Swing / Java AWT (Not special library)

3 How to Build/Import Your Project

3.1 Note saying things, like hitting the play button and/or clicking import project, is not enough. You need to explain how to import and/or build the game.

Building the game is not complicated. I have uploaded all the source code and resources to my GitHub. You can clone the repository from GitHub and open the game using IntelliJ. Alternatively, you can directly open the jar folder and run the game by clicking on “csc413-tankgame-Pandaz00.jar”. You can also run the game by navigating to the project folder in the terminal and using the command “java -jar csc413-tankgame-Pandaz00.jar”.

3.2 List what Commands that were running when building the JAR. Or Steps taken to build a jar. These can be the steps done either at the command line or in IntelliJ.

Once I have cloned my GitHub repository to access all the necessary code and resources, the next step is to establish the Tank Game > Simple > Resources as a project dependency. This is essential because all background music, animations, images, and sound effects are housed within that repository. Additionally, setting the Tank Game folder as the Resource will allow us to retain all the previous work, ensuring that every resource I have developed can be utilized once the project is complete. By following this setup, we will be prepared to execute the game in the subsequent steps.

3.3 List commands needed to run the built jar. These can be the steps done at the command line or in IntelliJ.

Open IntelliJ and click on File in the navigation bar. Then, click on Project Structure, and under Project Settings, select Artifacts. Click the plus sign to add project files and specify the storage directory. Click Apply and exit. Return to the navigation bar, click Build, and then select Build Artifacts. This will generate a jar file in the directory you specified earlier. You can start the game by running the jar file. The jar file can be placed anywhere on your computer without affecting its ability to run. Additionally, you can run the game via the terminal using the command “java -jar filename.jar”.

3.4 How to run your game. As well as the rules and controls of the game.

As mentioned earlier, after downloading and opening the source code, locate the tank-wars package, click on launcher.java, and start it to open the game. Alternatively, you can find the jar folder and click on the jar file “csc413-tankgame-Pandaz00.jar” to start the game as well. Once you are on the main page, you can click on Help to view the game’s instructions for further operation. The game description is as follows：

Game Manual: Tank Game

Overview

Tank Game is a dynamic, action-packed two-player game. Each player controls a tank, battling to outmaneuver and outgun their opponent. The game features destructible and indestructible obstacles, various power-ups, and different terrains that affect tank movement and strategy.

Controls

* Left Player (Red Tank):

- W, A, S, D: Move forward, left, backward, and right, respectively.
- Space Bar: Fire bullets.

* Right Player (Blue Tank):

- Arrow Keys (Up, Down, Left, Right): Move the tank in the respective directions.
- Enter Key: Fire bullets.
- Diagonal Movement: Press two directional keys simultaneously to move diagonally.

Game Environment

* Red Walls (Brick Walls):

- These can block tank movements and can be destroyed by firing bullets at them.

* White Walls (Bulletproof Walls):

- Impenetrable by both tanks and bullets, effectively blocking passage.

* Water Terrains:

- Tanks slow down when entering water and return to normal speed once they exit.

* Jungles:

- Tanks become invisible to the opponent when inside the jungle, becoming visible again once they leave.

* Power-Ups:

  - Heart-Shaped Supply: Restores health.
  - Shield Supply: Grants invulnerability to damage for 8 seconds.
  - Lightning Bolt Supply: Temporarily boosts speed for 5 seconds.

3.5 Assumptions Made when designing and implementing your game.

3.5.1 Technical Assumptions

The game is assumed to run on mid-to-high-end desktop environments, using Java Swing for the UI design, with thread management ensuring the game interface remains responsive. Resource management is optimized with object pools and a resource manager to reduce the frequency of memory allocation.

3.5.2 Player Behavior Assumptions

It is assumed that players are familiar with classic tank games, so simple and intuitive controls like arrow keys and a shoot button are provided. The two-player mode encourages local multiplayer interaction, assuming players will enjoy this and want to replay the game.

3.5.3 Game Design Assumptions

The design assumes the inclusion of various obstacles and terrains (e.g., water, forests, destructible and indestructible walls) to add strategic depth. Different terrains affect tank visibility and movement speed, increasing the complexity of player decision-making.

3.5.5 Market Assumptions

The target market is assumed to favor retro-style games and local multiplayer experiences, hence the use of simple pixel graphics and basic sound effects. The game is designed to appeal to players who enjoy classic arcade games.

3.5.6 Game Balance Assumptions

It is assumed that players value balanced gameplay, so both tanks are designed with identical performance, and the game map and power-ups (such as shields, speed boosts, and health recovery) are carefully balanced to ensure fairness during gameplay.

3.6 Tank Game Class Diagram

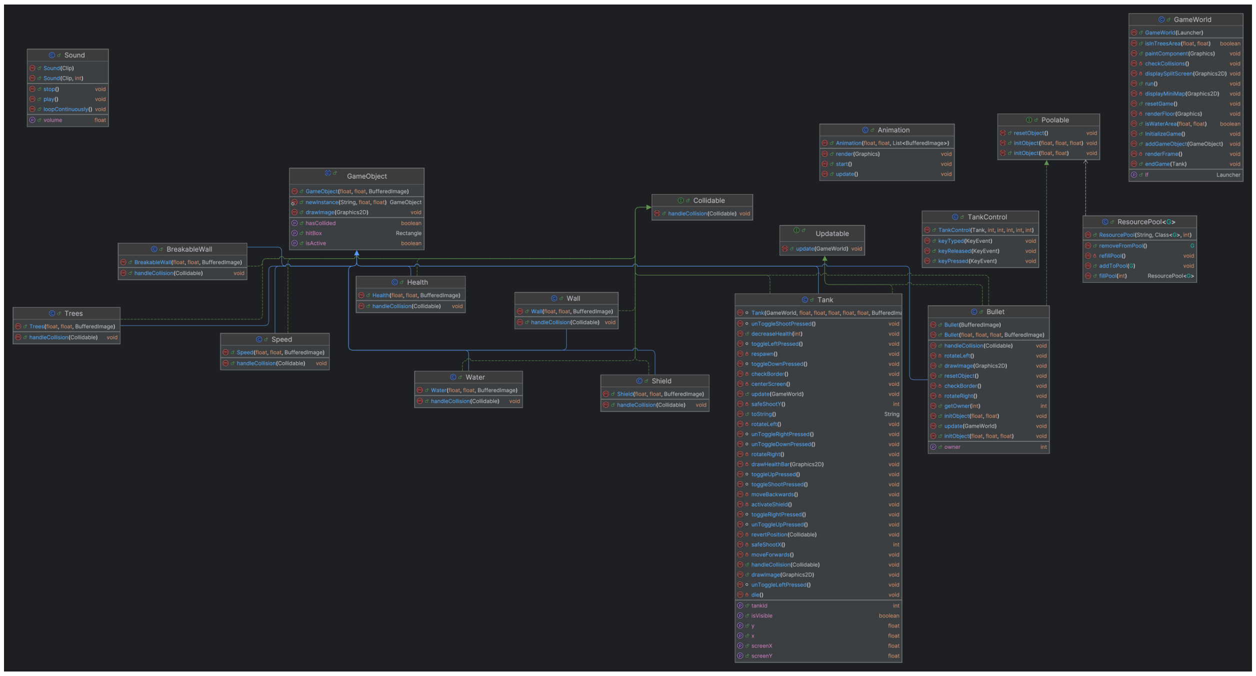


Image 04 – UML Image

3.7 Class Descriptions of classes implemented in the Tank Game

Animation: The Animation class manages and displays a sequence of images (frames) to create an animation effect in the game. It handles the position, timing, and progression of frames. The animation starts with the start method and updates frames based on a specified delay in the update method. The render method is responsible for drawing the current frame on the screen. The animation plays once through its sequence of frames and then stops.

BreakableWall: The BreakableWall class represents a wall in the game that can be broken upon collision. It extends the GameObject class and implements the Collidable interface, allowing it to interact with other objects in the game. When a collision occurs, the handleCollision method is triggered, though its specific behavior is not defined in this snippet. This class is likely used to create destructible obstacles within the game environment.

Bullet: The Bullet class represents a projectile in the game that is fired by tanks. It extends the GameObject class and implements the Poolable, Updatable, and Collidable interfaces. The class manages the bullet's movement based on its velocity and angle, and it handles collisions with other objects, such as breakable walls and enemy tanks. When a collision occurs, the bullet triggers effects like reducing a tank's health or breaking a wall. The Bullet class also supports object pooling, allowing bullets to be reused efficiently within the game.

Collidable: The Collidable interface defines a contract for any game object that can interact with other objects through collisions. It contains a single method, handleCollision(Collidable with), which must be implemented by any class that uses this interface. This method is intended to define the behavior that occurs when the object collides with another object in the game. By using this interface, different game objects like tanks, bullets, and walls can interact with each other in a standardized way when they collide.

GameObject: The GameObject class is an abstract base class for all game elements, providing common properties like position, image, and collision detection. It includes methods to manage the object's activity status, collision state, and rendering. Additionally, it has a factory method to create specific game objects based on a type identifier, making it a foundational class for building and managing game entities.

GameWorld: The GameWorld class is the main game engine that manages and renders the game environment in a Tank Wars game. It extends JPanel and implements Runnable, handling the game loop, including updating game objects, detecting collisions, and rendering the game frame. The class manages two tanks (t1 and t2), game objects, and animations, as well as handles the split-screen display and minimap rendering. It also includes logic for resetting the game, initializing resources, and determining the winner when the game ends.

Health: The Health class represents a health power-up in the game. It extends the GameObject class and implements the Collidable interface, allowing it to interact with other objects in the game. When a tank collides with the Health object, it would typically increase the tank's health, although the specific collision behavior is not defined in this snippet. This class is used to place health-restoring items in the game world.

Poolable: The Poolable interface defines methods for objects that can be reused and managed in a pool to optimize performance. It includes methods for initializing an object with specific coordinates, optionally including an angle, and resetting the object to a default state. This interface is typically used for objects like bullets or other frequently created and destroyed items in a game, allowing them to be recycled efficiently rather than being constantly created and discarded.

ResourcePool: The ResourcePool class manages a pool of reusable game objects to optimize performance by minimizing the creation and destruction of frequently used objects. It allows objects to be removed from the pool when needed and returned to the pool after use. If the pool is empty, it automatically refills with new instances of the specified class. This class is particularly useful for managing resources like bullets or other objects that are frequently instantiated and discarded during the game.

Shield: The Shield class represents a shield power-up in the game. It extends the GameObject class and implements the Collidable interface, enabling it to interact with other objects. When a tank collides with the Shield, it would typically grant temporary protection, although the specific collision behavior is not defined in this snippet. This class is used to add shield items to the game world, enhancing gameplay by providing tanks with a defensive boost.

Sound: The Sound class manages audio playback within the game, providing control over sound clips. It allows for playing, stopping, and looping sounds, as well as adjusting the volume. The class supports continuous looping for background music and ensures that sounds are played from the beginning each time they are triggered. This class enhances the game's audio experience by managing sound effects and music efficiently.

Speed: The Speed class represents a speed power-up in the game. It extends the GameObject class and implements the Collidable interface, allowing it to interact with other objects. When a tank collides with the Speed object, it typically increases the tank's movement speed, though the specific collision behavior is not defined in this code snippet. This class is used to place speed-enhancing items in the game world, providing players with temporary boosts in agility.

Tank: The Tank class in the game controls the player's tank, handling movement, shooting, health management, and interactions with other game objects. It allows the tank to move, rotate, and shoot, while also managing health points and lives. The class handles collisions with walls and other objects and can apply power-ups like shields and speed boosts. The tank is visually represented on the screen, including its rotation and health bar.

TankControl: The TankControl class is responsible for handling player input by implementing the KeyListener interface. It maps specific keys to control the movement and actions of a tank, such as moving up, down, left, right, and shooting. When the corresponding keys are pressed or released, the class updates the tank's state by toggling the appropriate actions, allowing the player to control the tank in the game.

Trees: The Trees class represents tree objects in the game, which may act as obstacles or part of the environment. It extends the GameObject class and implements the Collidable interface, allowing it to interact with other objects in the game. Typically, trees might block the movement of tanks or other game elements, contributing to the game's terrain and strategy.

Updatable: The Updatable interface requires game objects to implement an update method, allowing them to be refreshed or changed during the game loop. This ensures that objects like tanks, bullets, or animations can update their state or behavior as the game runs.

Wall: The Wall class represents a solid, non-destructible barrier in the game. It extends the GameObject class and implements the Collidable interface. Walls typically block movement and projectiles, contributing to the game's strategic environment.

Water: The Water class represents water terrain in the game. It extends the GameObject class and implements the Collidable interface, allowing it to interact with other objects. Water typically affects the movement of tanks or other game elements, slowing them down or altering their behavior when they enter the water area.

EndGamePanel: A user interface panel displayed at the end of the game, offering options like restarting or exiting.

HelpDialog: The HelpDialog class creates a modal help window for the Tank Game, providing game instructions and controls. It displays detailed information about gameplay, controls, environments, and power-ups in a scrollable JTextArea. A "Close" button allows users to exit the dialog and return to the game. This class helps players quickly access essential game information.

StartMenuPanel: The initial menu is shown to players, providing options to start the game or exit.

GameConstants: The GameConstants class defines constant values used throughout the Tank Game for various screen dimensions. It includes constants for the game screen's width and height, the size of the game world, and the dimensions of the start and end menu screens. These constants ensure that the game's layout and scaling remain consistent across different parts of the game, making it easier to manage and adjust screen sizes when needed.

Launcher: The Launcher class serves as the central controller for the Tank Wars game, managing the game's user interface and switching between different screens such as the start menu, game screen, and end menu. It initializes and configures the main game window (JFrame) and uses a CardLayout to switch between various panels. The class also ensures the game runs on a separate thread to keep the GUI responsive. The Launcher handles the creation and management of key game components, like the GameWorld and UI panels, and provides methods to control the game flow, including starting, ending, and closing the game. The main method initializes resources and starts the game by displaying the start menu.

ResourceManager: The ResourceManager class is responsible for loading, managing, and providing access to various game resources such as images, sounds, and animations. It uses maps to store sprites, sounds, and animation frames, which are loaded from the file system. The class includes methods to initialize these resources (loadAssets, initSprites, loadSounds, loadAnims) and retrieve them when needed (getSprite, getSound, getAnim). This ensures that all game assets are efficiently managed and can be easily accessed throughout the game, helping to keep the code organized and maintainable.

ResourcePools: The ResourcePools class manages a collection of object pools for different types of reusable game objects. It uses a map to store ResourcePool instances, each identified by a string key. The class provides methods to add a pool to the collection (addPool) and retrieve an object from a specific pool (getPoolInstance). This allows the game to efficiently manage resources like bullets or other frequently used objects by reusing instances instead of constantly creating and destroying them, which helps optimize performance.

3.8 Self-reflection on the Development process during the term project

In terms of technical implementation, I have realized the importance of code structure and performance optimization. In the early stages of the project, I focused primarily on functional implementation without giving much thought to the maintainability and scalability of the code. However, as the project progressed and the complexity of the game increased, I began to understand the significance of having a well-structured codebase and design patterns. For instance, by using the Object Pool pattern to manage game resources, such as bullets and sound effects, I was able to reduce the frequency of memory allocations and improve the game's performance. This experience has taught me the importance of planning resource management strategies early in the development process.

Regarding player behavior assumptions, I initially overlooked the diversity of user experiences. In the early versions, I assumed that all players would be familiar with the controls of classic tank games, so I implemented very simple keyboard controls. However, through user testing, I discovered that some players desired customizable key bindings or more modern control schemes. This realization has led me to understand the need for offering more flexible configuration options in future projects to accommodate diverse user needs.

In terms of game design, I initially underestimated the importance of diversity in map and terrain design. Although the game included various terrains and obstacles, such as water, forests, and destructible walls, their impact on gameplay did not meet my expectations. This prompted me to reassess the balance and strategic elements of map design, emphasizing the importance of thorough design and repeated testing to ensure gameplay quality.

From a market perspective, I initially believed that the retro-style pixel graphics and simple sound effects would appeal to fans of classic arcade games. However, as the project progressed, I realized that while retro aesthetics have their charm, they may not fully meet the expectations of all players, especially with the evolving market demands. In future projects, I plan to conduct more thorough market research and incorporate modern design trends to create games that better align with broader audience preferences.

Finally, regarding game balance, I came to understand that balance is not just about ensuring equal tank performance. It also involves considerations such as map layout, power-up distribution, and game pacing. Achieving satisfying gameplay requires careful tuning and ongoing testing. This experience has highlighted the critical importance of continuous iteration in the development process to maintain balance and deliver a fun and fair gaming experience.

Overall, this project has deepened my understanding of the entire development process, from design to implementation and optimization. Moving forward, I will place greater emphasis on early-stage planning, understanding user needs, designing for diversity, and continuously refining and testing to enhance the quality of my projects and ensure player satisfaction.

3.9 Project Conclusion

This project was incredibly meaningful and provided an excellent opportunity to comprehensively learn Java while enhancing various skills. I would like to express my sincere gratitude to Professor Souza for his invaluable guidance and support throughout the project. His insights were crucial in resolving the challenges I encountered, especially in addressing the collision detection issues that allowed for the successful implementation of more complex features.

Due to a lack of experience early on, I mistakenly set up new hit boxes in classes other than the GameObject class, rather than using setLocation. This caused the hit box detection areas to be too large, leading to immediate collision detection when the game started and also affecting the map display. Because of these collision issues, many features could not be completed initially. Fortunately, with Professor Souza’s guidance, I was able to resolve this problem, which paved the way for the successful implementation of more complex features later on. I will remember these lessons and strive to continue learning.

Throughout the development process, I learned the importance of structuring code efficiently, optimizing performance through techniques like object pooling, and continuously testing and refining gameplay elements. The project highlighted the need to accommodate diverse player preferences by offering customizable controls and considering various user experiences. Additionally, I recognized the significance of aligning game design with market trends, balancing gameplay through thoughtful map design, and ensuring fair power-up distribution. Overall, this project deepened my understanding of game development and taught me valuable lessons that will guide my future projects.