The Design and Implementation of an arcade based game: Defender:SA

Joshua Schwark (2118091) – Brent Butkow (2438311)
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa

***Abstract*—This paper presents the design, implementation, and testing of a computer-based game, Destoyer:SA, which is based on the 1980's arcade game *Defender*. The implementation strategy utilized Object-Orientated design along with a modular approach, organizing the code into distinct data, logic, and presentation layers. The Presentation layer was created through the use of of the SFML multimedia library. Inheritance was implemented through the use of three Abstract Bases classes: GameEntity, EnemyEntity and ProjectileEntity. GameEntity models the concept of any entity that exists within the game, EnemyEntity models the concept of entities that oppose the user while playing the game, and ProjectileEntity models any entity that acts as a projectile, throughout the game. A total of 123 test cases are implemented, in order to thoroughly test all of the game functionality and logic. Related tests are grouped into thirteen test suites which allow for easy sorting through tests. The game is considered a success, as all minor features and three out of four major features are implemented, along with good coding principles and practices being used. The game could be improved by implementing increased functionality and improving the inheritance hierarchy to minimize code redundancy.**

Keywords— C++, development, Object-Oriented Programming, SFML, unit tests

I. Introduction

This project centres on the development of a "Defender-like" video game using C++ and the SFML (Simple and Fast Multimedia Library) framework. It applies object-oriented programming principles, collaboration, and documentation practices as well as the separation of concerns through the structure of the code.

Object-oriented decomposition and analysis to dissect the software problem, emphasizing entity interactions and dependencies will be employed. Subsequently, object-oriented design principles will be applied to create a solution that promotes maintainability and code reusability. This includes an inheritance hierarchy implemented throughout the project. The integration of external software libraries, exemplified by SFML, highlights adeptness in incorporating existing resources to expedite development. A comprehensive test suite will validate the software's correctness and reliability.

Furthermore, this project underscores the significance of effective teamwork in software development. Collaborative efforts have been essential in achieving our goals, emphasizing our ability to function succinctly in a small team which is pivotal in modern software development practice. An adapted agile project management approach is followed in the project, and the software Trello is used to assist with the organisation of tasks.

This report explains and analyses the process of developing the "Defender-like" game, encompassing design choices, encountered challenges, accomplishments, and insights. It explains the game's functionality, the code structure and code layering used as well as the extensive testing done. Furthermore, the dynamic interactions used within the game are explained and analysed. The report thenthen critically evaluates the code and provides areas in which the code could be improved.

II. Problem Definition

The problem that the group is tasked with is to develop a video game using C++ coding language along with the SFML library. This video game should resemble and take inspiration from the 1980's arcade game "Defender".

A. Defender: SA Game Play

Defender:SA, based on the arcade game *Defender*, is a video game that is developed using ANSI/ISO C++ 17. This game is playable on a 1600 x 900 screen.

The game consists of a Player that can move left, right, up, and down on the screen. Enemies inhibit the Player by dropping/shooting projectiles. These projectiles would destroy the Player. The Player has three shields that make it completely immune to any Projectile. The player can shoot PlayerLasers at the Enemies. Each enemy shot and destroyed would award the Player 10 or 15 points.

There are two types of Enemies, both with different types of movement and with different Projectiles. The Lander entity (represented by a minibus taxi) moves towards the Player and move away from the Player when the Player faces them. The Lander fires LanderLasers at the Player, and if the LanderLaser collides with the Player, the player is destroyed. The Bomber entity (represented by a "Eishkom" bomber) moves more randomly around the screen until it reaches a certain target. The Bomber drops Mines (represented by potholes) randomly around the screen which do not move. If a Player collides with either the Bomber or the Mine it is destroyed.

Asteroid showers fall from the top of the screen at an angle. Asteroids (represented as stop signs) destroy everything in their path. These fall at random times and spawn 4-8 Asteroids at a time. Humanoids (represented as money stacks) exist and walk around the ground, and must be protected by the player.

Landers try to capture and kidnap Humanoids. If a Lander successfully captures a Humanoid and flies it to the top of the screen the Player would lose 80 points. The player can save the Humanoid by shooting the Lander that

is capturing the Humanoid and then setting the Humanoid back on the ground. The player also loses 80 points if the Humanoids die by being shot or falling to the ground.

The Player has limited fuel. If the fuel runs out the Player would fall out the sky. Players can replenish their fuel by collecting FuelTanks which are scattered along the bottom of the screen. The Player will level up when all enemies are destroyed. The number of enemies increases when the level increases. Player shields are replenished on each level increase.

A minimap exists to show the player all entities currently existing on the map, and a city skyline is visible in the background, to indicate the player's movement.

Once defeated, the user can either exit the game or restart and play another round. Each round, the high score will be visible, and will be updated if the player's score is greater than the previous high score.

B. Success Criteria

The project is deemed successful if all of the above functionality is implemented. Additionally, high quality code is implemented using good coding principles and significant separation of concerns such as data, logic, and presentation layers. Implementation of coding principles, such as inheritance and role modelling are required.

Effective unit tests which verify the behaviour, movement, and logic of the game objects are needed to verify effective code development and function.

C. Assumptions and constrains

The game was constrained to be played on a maximum screen size of 1600 x 900 pixels and on a Windows platform.

The game is required to be coded in ANSI/ISO C++ and use the SFML 2.6.0 library. The SFML library used is assumed to be thoroughly tested and therefore the need for further testing would be obsolete.

III. Code Structure and Implementation

A. Architecture Layers

The code is divided into three distinct layers:

- Presentation Layer
- Application Logic Layer
- Data Layer

The software architecture adheres to the separation of concerns principle, ensuring distinct layers for enhanced dependency decoupling. This separation facilitates the independent manipulation and maintenance of layers without impacting one another. Specifically, the graphics library, SFML, can be replaced seamlessly without any modifications to the application layer. The presentation layer is managed through the WindowUI class, the data Layer is overseen by the FileManager class, and the application logic layer is regulated through the Game class.

B. Domain Model

In the domain model of the game, various objects known as Entities interact uniquely within the game environment. These entities include:

- Player
- PlayerLaser
- Lander
- LanderLaser
- Bomber
- Mine
- Humanoid
- Asteroid
- FuelTank

Each of these entities has distinct properties and behaviors. They are essential components that need to be rendered on the screen, are capable of collision with other entities, and possess unique characteristics. Notably, all entities, excluding FuelTank and Mine, have the ability to move. Players, Landers, and Bombers are equipped with shooting capabilities. In terms of control, the player is manipulated by the user, whereas all other entities operate autonomously under the program's control.

The game employs a Cartesian coordinate system to represent the 2D world within the game. The game's world has a limited height, while its width is unlimited but circular. This design allows infinite sideways movement, with extensive rightward movement equivalent to approaching from the left. Additionally, the game is structured into levels, where a specific number of enemies must be defeated to progress. As the player advances in levels, the game's difficulty intensifies.

C. Presentation Layer Class Structure

The Presentation Layer of the software is structured through three main classes: **Window**, **KeyboardInput**, and **Clock**. These classes utilize the SFML graphics library to manage user input, display game content on the screen, and regulate the timing of the code execution.

1) Clock Class

The **Clock** class serves the purpose of storing both the total elapsed time of the game and the time difference between each iteration of the game loop. These time parameters enable functions to have specific effects for defined durations and subsequently modify their operations after the set time has elapsed. This class thus provides information about the exact framerate of the game at any moment, ensuring precise movement and entity spawning., despite the computer hardware used.

2) KeyboardInput Class

The **KeyboardInput** class is responsible for tracking user inputs. Utilizing SFML, it polls user events, processes the inputs, and converts them into valid results from the **UserInput** enumeration. This approach allows the application layer to discern user inputs without requiring direct access to the SFML class. The class returns a key map, indicating the specific keys being utilized at any given time during the game.

3) WindowUI Class

The WindowUI class manages all graphical displays within the game. It interfaces with the data layer to load textures and fonts, as well as load and save scores. Moreover, it retrieves essential information from the data layer about each entity, enabling the rendering of these entities. The class dynamically adapts the displayed screen based on the game's state, such as showing the start or

game-over screen. During gameplay, the WindowUI class associates textures with each entity, positions them correctly, and renders them on the screen, whislt flipping the direction of the sprite if needed, based on the direction of the entity. Additionally, it provides crucial player information such as score, remaining shields, and fuel levels. The class displays the player's current level and notifies the player when they level up. Although the window displays only a portion of the entire game, it includes a minimap depicting the location of all existing entities at any given time. The window features a scrolling background, which indicates the player's movements. As the player moves, the screen follows, adjusting the positions of all other entities accordingly.

D. Application Logic Layer Class Structure

The Application Logic Layer of the game is orchestrated through several key classes: **Game**, **EntityCreator**, **EntityManager**, **EntityDestroyer**, **HumanKidnapping,** and **ScoreManager**. These classes operate in conjunction, allowing for the seamless functioning of all game entities, which are created as children of the **GameEntity** class. **GameEntity** models all of the entities that can be shown on screen and that can interact with each other.

1) Game Class

The **Game** class serves as the core orchestrator of all game processes. It not only establishes and executes the game loop but also manages game states, entity creation, movement, collision detection, unique interactions, and entity destruction. This class controls the overall flow of the game, including resetting after a defeat. The game's state is dynamically influenced by user inputs and in-game events. It interacts with the **WindowUI** class from the Presentation Layer, to provide the necessary information for screen display. The Game class achieves this by utilizing composition with the **EntityCreator**, **EntityManager**, **EntityDestroyer** and **HumanKidnapping** classes.

2) EntityCreator Class

The **EntityCreator** class is dedicated to the creation of game entities. Player entities are initialized at the game's start and upon reset. Projectile entities are generated based on their parent entity's location and direction when the parent entity attempts to fire. Various creation events utilize random numbers, such as the spawning of Humanoids and FuelTanks on the ground within the visible screen area. Additionally, random asteroid showers can occur, resulting in multiple asteroids descending from above the screen. Enemies, including Landers and Bombers, are created within a random region around the player when the number of created entities is less than the maximum enemy count. Leveling up increases the maximum enemy spawn limit and enemy spawn rates.

3) EntityManager Class

The **EntityManager** class assumes responsibility for collision checking between entities and determining if an entity is within the visible screen area. It includes methods for detecting overlapping entities and checking collisions between entities based on their hitboxes. Projectile entities that are off-screen are flagged for destruction. The class also handles collisions between various entity types, ensuring that their hitboxes overlap and they are allowed to collide. Furthermore, it verifies if the player is colliding with a FuelTank, determining whether the player's fuel should reset or not.

4) EntityDestroyer Class

The **EntityDestroyer** class assumes the crucial role of managing entity deletions. It iterates through all existing entities, disregarding those lacking a destruction flag, and removes entities flagged for destruction. This class keeps track of defeated enemies and calculates the score changes resulting from entity destruction. The class possesses the ability to delete entities of specific types, for instance, the destruction of all mine entities upon player leveling up. It also clears all existing entities when the game is reset, ensuring the independence of each gaming session from the previous ones.

5) HumanoidKidnapping Class

The **HumanoidKidnapping** class is exclusively dedicated to the interaction between humanoids, players, and landers, focusing on the kidnapping and saving of humanoids. By utilizing composition with the **EntityManager** class, it can detect collisions between entities. This class checks collisions between landers and humanoids, indicating kidnappings. Additionally, it identifies situations where falling humanoids are saved by the player. When a lander successfully kidnaps a humanoid and reaches the top of the screen, both entities are destroyed. Similarly, when a player carrying a humanoid touches the ground, the humanoid is released and resumes normal movement. The class also ensures that landers periodically move towards humanoids to attempt kidnappings, even if they are far away. However, landers will only target humanoids currently visible on the screen.

6) ScoreManager Class

The **ScoreManager** class manages score storage, ordering, and high score display. It interacts with the data layer, utilizing composition with the **FileManager** class, to read scores from a text file. These scores are sorted, and the highest score is presented as the current high score. At the end of each game, the user's score is sent to this class, which adds it to the existing scores and updates the text file. Additionally, the class has the capability to provide a complete leaderboard of the top scores, although this functionality is not implemented in the current code.

7) GameEntity Class

The **GameEntity** class serves as an Abstract Base Class (ABC), acting as a template for all entities in the game. It defines essential attributes and behaviors shared by all entities, promoting a cohesive and organized entity structure. The class includes properties such as position, size, type, movement speed, movement angle and the direction it is facing. The size parameter determines the entity's hitbox, while the type property informs other functions about the entity's specific identity. Each entity also stores the score the user earns upon defeating it and a flag indicating whether the entity should be destroyed. The class provides a virtual method to retrieve the tilt angle, utilized by the **WindowUI** class for rotation determination.

The **GameEntity** class facilitates circular wrapping of an entity's position, ensuring a seamless game world. It contains pure virtual functions to be overridden by its derived classes, enabling unique movement and firing

functions for each entity. Additionally, it incorporates common getters for information such as position, size, and type, adhering to the Do Not Repeat (DRY) principle by eliminating redundant functions in child classes. A static vector within the class maintains all active entities in the game, enabling convenient management and access.

8) Player Class

The **Player** class, inheriting from **GameEntity**, represents the entity controlled by the player. It responds to user inputs to determine movement, incorporating momentum that allows a gradual acceleration to a maximum speed. The player possesses three shields, each providing invincibility for five seconds when activated. Fuel consumption occurs with movement inputs, and when fuel is depleted, the player ceases user-directed movement and descends until it hits the ground, leading to the end of the game. The player can reset fuel by collecting a fuel tank and reset shield count by leveling up. The class contains a flag that records if a humanoid is being saved, such that two humanoids cannot be saved at the same time.

9) Humanoid Class

The **Humanoid** class inherits from the GameEntity class and contain three separate movement mechanics. When not kidnapped, humanoids roam the ground. When kidnapped, they move beneath the kidnapper based on the kidnapper's size. Upon the kidnapper's destruction, the humanoid inherits the kidnapper's momentum and starts falling. When saved by a player, the humanoid resumes ground-level movement.

10) FuelTank Class

The FuelTank class inherits from the GameEntity class and represents stationary entities at ground level. Fuel tanks do not move and set their destruction flag to true after existing for 10 seconds.

11) EnemyEntity Class

The **EnemyEntity** class is an ABS that inherits from the **GameEntity** class. It serves as the blueprint for enemy entities in the game. It encapsulates functionality related to enemy movement and firing, featuring pure virtual functions that guide movement and firing behaviors for its derived classes. This class also implements helper functions to calculate angle differences between the entity and its target, ensure the entity's position is within bounds, and compute distances to other entities. These functions are implemented to adhere to the Do Not Repeat (DRY) principle.

12) Lander Class

The **Lander** class inherits from **EnemyEntity** and represents the most common enemy type within the game. Landers exhibit two distinct movement mechanics. By default, they maintain a specific distance from the player, adjusting their position based on the player's movements while periodically firing lasers at the player. The lander's movement incorporates momentum, providing a realistic experience. When instructed to target a humanoid, the lander ceases firing and moves directly toward the humanoid. When off of the visible screen, the Lander's also cease to fire. Upon kidnapping a humanoid, it rises vertically in an attempt to kidnap the humanoid ato above the screen. If the player accidentally destroys the humanoid, the lander reverts to its default behavior of targeting the player.

13) Bomber Class

The **Bomber** class, inheriting from **EnemyEntity**, represents another enemy type. Bombers execute sinusoidal movements toward a specific target position. Upon reaching the target, the Bomber deploys a mine and selects a new target location. The new target position is determined relative to the player's location at the moment the previous mine is deployed. Once a target is established, the Bomber disregards the player until it needs to select a new target position, focusing solely on moving towards its chosen target.

14) ProjectileEntity Class

The ProjectileEntity class, an Abstract Base Class (ABC) inheriting from GameEntity, serves as the foundational template for all projectile entities in the game. It outlines the essential behavior for projectile movement, implemented through the move function. Projectiles travel in straight lines determined by their angles and move a specific distance dictated by their speed. The class also incorporates the tiltAngle defined in the GameEntity class, allowing projectiles to face the direction of their movement when being rendered by WindowUI.

15) PlayerLaser Class

The **PlayerLaser** class, an inherited class of **ProjectileEntity**, represents the laser projectiles fired by the player. PlayerLasers move horizontally at an exceptionally high speed, allowing swift and precise targeting of entities.

16) LanderLaser Class

The **LanderLaser** class, a derived class of **ProjectileEntity**, represents the laser projectiles fired by Landers. The angle of the **LanderLaser** is determined as the angle between the parent lander entity and the player entity at the moment of its creation. **LanderLasers** move relatively slowly, allowing players to dodge them.

17) Mine Class

The **Mine** class, a derived class of **ProjectileEntity**, represents the explosive projectiles fired by the Bomber. Mines have no speed and do not move, remaining in a fixed position until they are destroyed.

18) Asteroid Class

The **Asteroid** class, a derived class of **ProjectileEntity**, represents the asteroids of an asteroid shower. Asteroids are set at specific angles ($-3\pi/8$ or $-5\pi/8$) depending on their direction of movement, ensuring consistent downward diagonal movement from the top to the bottom of the screen.

E. Data Layer Class Structure

The Data Layer is managed by the **FileManager** class, which utilizes both built-in file openers and the SFML class to handle various file operations.

1) FileManager Class

The **FileManager** class is responsible for opening images and converting them into SFML textures, opening fonts and saving them as SFML fonts, as well as handling

the reading and writing operations for text files. When given a string representing the file name, the class validates the file's existence and throws an error if the specified file is not found. Upon successful file retrieval, it returns the content of the file.

F. Enumeration types

Enumeration types, denoted as Enums, are extensively employed in the codebase to represent well-defined sets of constants. This systematic approach not only enhances code comprehension but also significantly reduces the likelihood of human error. Three key Enumeration types are utilized within the code:

1) EntityList Enum

The EntityList enum comprehensively enumerates each entity existing within the game. By predefining interactions for each entity, this enumeration facilitates efficient entity creation and collision detection.

2) GameState Enum

The GameState enum enumerates all possible states the game can assume. This enumeration is pivotal in allowing the WindowUI class to dynamically display the appropriate screen based on the specific game state dictated by the logic layer.

3) UserInput Enum

The **UserInput** enum meticulously lists all user inputs utilized within the game. This enumeration enables the logic layer to discern the user's intentions effectively. Importantly, unused keys are systematically ignored and not processed, enhancing the overall efficiency of the system.

IV. Dynamic Game Behavior

The dynamic game behaviour and processes are explained through the Main Game Loop, the collision between entities and the polling for user inputs.

A. Main Game Loop

The main game loop is controlled by the **Game** class. In the main game loop, the state of the game is found, by polling for user inputs in the **KeyboardInput** class which is called in the **WindowUI** class. This allows the user to either start playing the game or to view the information screen. Once the user is playing the game, entities begin to spawn randomly according to a time seed. Following that, the entities are updated. This includes the movement of all entities, the firing of all entities, the humanoid kidnapping interactions and the Player's refuelling. The entity collisions are checked, which sets entities to be destroyed and can affect the game state. Said entities are then removed from the game. The screen is then rendered so that all the entities are drawn onto the screens in their correct positions. The process repeats until the game state changes to either **Pause** or **Game Over.** In a **Game Over** state the players score is displayed, and the player is given the option to exit the game or restart, while the user's score is saved into a text file. When the game is over, all entities are removed using **entityDestroyer_.removeAllEntities()**.

This process can be seen in the flow chart in Figure 2, which shows the logical flow of the main game loop. Figure 3 shows the extensive utilization of object conversations to incorporate the necessary functionality. The figure highlights the intricate interconnections between various elements within the game, underscoring the complexity of interactions that contribute to the game's overall dynamics and gameplay experience.

B. Collision Detection

The method used sets a hitbox around all the entities. The positions of the hitboxes are compared using the **hitboxOverlap()** function, and overlaps are found. The **isColliding()** function is then checked if those entity should be allowed to collide. **IsColliding()** is called for every pair of entities every game loop, and every entity that is colliding has its destroy flah set to true. Two entities are said to have an overlapping hitbox if the difference between their x positions is smaller than half of the sum of their widths, and the difference between their y positions is less than half of the sum of their heights.

IsCollding() is non-symmetrical, allowing one entity to collide with another, without the reverse being true. Such as an Asteroid destroying a LanderLaser, without the LanderLaser destroying the Asteroid.

When the player's shield is activated, the player is immune to collisions, and is thus invincible, while allowing other entities to collide, meaning that the other entity will be destroyed while the player will remain unscathed.

C. User Input Polling

The user inputs are polled in the **KeyboardInput** class which is in the presentation layer. To maintain a separation of the presentation and logic layers, the user input is converted to a **UserInput** enumerator variable. Once the input has been converted, it is passed to the **Player** object to control the movement, shooting, and setting of the shield. The user can hold down keys to rapidly fire and continuously move in any direction. The input is also used by the **Game** class in order to control the state of the game, allowing the game to know when to pause, unpause, start, restart, show the information screen, or exit.

V. Code Testing

Defender: SA underwent comprehensive testing using the Doctest 2.4.11 unit testing framework. The development followed the practice of test-driven coding, a methodology that ensures bug-free and efficient code by writing tests for a function before implementing the function itself. The tests were written according to the Arange Act Assert (AAA) methodology, with each test written according to the FIRST guideline: Each test must be fast, isolated, repeatable, self-validating, and timely. The extent of testing varied for different classes and functions, depending on their complexity and the presence of edge cases. Consequently, the number of tests created for each class differed, as indicated in Table 1.

All application layer functions, with the exception of **startGame()**, underwent testing. Similarly, all data layer functions, except for **setFile()**, were rigorously tested. However, only tests for the **KeyboardInput** class were conducted for presentation layer functions.

To facilitate efficient testing and improve the comprehension of tests, the tests were organized into 13 test suites. Each suite focused on specific classes or concepts, enhancing the clarity and manageability of the

testing process. These test suites encompassed the following concepts and classes:

- Testing EntityCreator
- Tetsing Entitymanager
- Esting EntityDestroyer
- Testing KeyboardInput
- Testing the functions related to the player refuelling
- Testing the player's shield
- Testing the firing function of the Player, Lander and Bomber
- Testing the base functions of the three ABC's, GameEntity, EnemyEntity and ProjectileEntity
- Testing HumanoidKidnapping and all functions related to humanoid kidnapping
- Testing the movement of the player
- Testing the movement of all autonomous entities
- Testing Scoremanager
- Testing FileManager

TABLE I. NUMBER OF TEST CASES AND ASSERTIONS PER CLASS AND LAYER.

Layer	Tested Class	Number of Test Cases	Number of Assertions
Application	Asteroid	1	3
	Bomber	3	8
	EnemyEntity	1	2
	EntityCreator	15	65
	EntityDestroyer	4	11
	EntityManager	23	62
	FuelTank	2	5
	GameEntity	11	32
	Humanoid	3	9
	HumanoidKidnapping	21	47
	Lander	8	12
	LanderLaser	1	2
	Mine	1	2
	Player	18	36
	PlayerLaser	1	6
	ProjectileEntity	2	6
	Scoremanager	2	2
Presentation	KeyboardInput	4	4
Data	FileManager	2	4
	TOTAL:	123	318

A. Non-tested Functionality

In the Data Layer, the **setFile()** function of the **Filemanager** class was intentionally left untested. This decision was made because testing this function would lead to a permanent addition of a score to the score text file, potentially invalidating the file's accuracy if automated tests were performed. To avoid compromising the integrity of the score file, this function was excluded from testing procedures.

In the Presentation Layer, no visual functions were subjected to testing. This exclusion stemmed from the assumption that the **SFML** class, being an established and widely used library, was thoroughly tested and reliable. Furthermore, testing functions within the **WindowUI** class proved challenging due to their direct impact on the render window's display. Automated tests typically rely on observable changes, which are difficult to achieve in graphical user interfaces. As a result, these functions were not automated but were manually tested instead. Manual tests ensured that functions altering the display on the render window behaved as expected.

In th**e Application Logic Layer,** the **Game.startGame()** function remained untested. This decision was motivated by the encapsulation and variable hiding principles. The **Game** class did not expose any public variables or public getters, making it difficult to perform automated tests without compromising encapsulation. Manual tests were conducted through continuous gameplay, ensuring that the **startGame()** function operated as intended in practice.

B. Hard To Test Functionality

The **EntityCreator** class presented challenges for testing due to its reliance on random numbers during entity creation. To address this, the class was modified to accept a vector of random numbers, with a default value indicating the generation of random numbers. During non-testing situations, the default vector implied the use of random numbers for entity creation. However, when a specific vector of random numbers was provided (not using the default vector), the class utilized those predetermined numbers instead of generating random ones. This approach allowed the class to use random numbers during normal gameplay while providing a deterministic set of results for testing purposes, ensuring reliable and reproducible tests.

C. Implemented Tests

1) FileManager Testing

The data layer **FileManager** class underwent testing to validate its behavior when given both invalid and valid file paths. When an invalid file path was provided, the class was verified to throw an error, halting the code execution. Conversely, when a valid string was entered, the class was tested to ensure smooth operation without errors.

2) KeyboardInput Testing

The presentation layer **KeyboardInput** class was rigorously tested to ensure accurate processing of SFML events into entries in the **UserInput** enum. Non-keyboard events are tested to be ignored, as well as key-presses for non-used keys. Pressing a modifier key such as shift or alt does not impact the key being registered. Pressing a used key sets the corresponding map entry to true, while releasing a used key sets the corresponding map entry to false.

3) EntityCreator Testing

The **EntityCreator** class was extensively tested to guarantee that projectiles were created with correct orientations relative to their parent entities, either at their centers or fronts based on the projectile type. Additionally, the class was assessed under scenarios where the number

of enemies defeated equaled the maximum allowed, ensuring the correct increase in the user's level. Thorough testing involved generating different sets of random numbers to verify that entities spawned at the correct rate and positions. The class was also examined to confirm that enemies were not created if the maximum number of enemies had already been reached.

4) EntityManager Testing

The **EntityManager** class was validated to ensure that the **onScreen()** function functioned as intended, returning true only if an entity would be visible on the screen. Collision detection was thoroughly tested to ensure that entities' hitboxes correctly collided with each other, and the collisions between all sets of possible entities were tested to testif said entities are allowed to collide with each other. Special attention was given to the collision between the player and a **FuelTank**, ensuring that refueling worked as expected.

5) EntityDestroyer Testing

The **EntityDestroyer** class was tested rigorously to verify its ability to remove all destroyed entities and return the correct values. Additionally, the class was assessed for its capability to selectively destroy entities of a specific type, as well as its ability to destroy all entities, regardless of their types.

6) HumanoidKidnapping Testing

The **HumanoidKidnapping** class was tested to confirm that landers could successfully kidnap humanoids and players could save them. Special scenarios were tested, including situations where landers carrying humanoids died, causing the humanoid to drop and potentially be destroyed if not saved by the player in time. The class was also examined to ensure that players could only save one humanoid at a time, and a humanoid could not be kidnapped by two landers simultaneously. The behaviour of landers targeting humanoids was tested, and their subsequent actions when the humanoid that is kidnapped is destroyed, was tested as well.

7) ScoreManager Testing

The **ScoreManager** class underwent tests to add new scores to the leaderboard, verifying the accuracy of the resulting leaderboard. Additionally, the class was assessed to confirm that the largest score on the leaderboard was indeed given as the high score.

8) GameEntity Testing

The **GameEntity** class was thoroughly tested through its common base functions. These tests ensured that correct size, position, type, defeat score, and tilt angle values were returned as expected.

9) Player Testing

The **Player** class was validated under various keypress scenarios to ensure correct player movement, firing behavior, and shield activation. Special functionality, such as the invincibility during shield activation, was tested. The refueling mechanism was assessed to confirm the player's fuel could be replenished, and the level-up function was validated to ensure the player's shield count reset upon leveling up.

10) Humanoid Testing

The **Humanoid** class was validated to ensure that it moves around the ground, until kidnapped by a lander, at which point it moves with the Lander. When said lander is destroyed, the humanoid will retain the momentum of the lander and start to fall, until it is saved by the player, or destroyed.

11) FuelTank Testing

The **FuelTank** class was tested to ensure that it does not move as the game progresses. It was further tested to ensure that it destroys itself after existing for 10 seconds, if it had not been destroyed prior.

12) EnemyEntity Testing

The **EnemyEntity** class was tested to ensure that the **update()** functions returns true when the enemy intends to fire, and false otherwise. Further tests were not necessary, as the functionality was either tested in its parent class or children classes.

13) Lander Testing

The **Lander** class was tested to confirm that landers move toward the player when far away and move away when close. Momentum behaviour was tested to ensure the lander can not change its direction immediately. Lander firing behaviour was also tested to fire periodically, unless targeting or kidnapping a humanoid. The lander is unable to fire when it is not on the visible screen.

14) Bomber Testing

The **Bomber** class was tested to ensure it targets a position close to the player. Once a target is decided, the bomber moves toward the target, fires a mine, and chooses a new target upon reaching the previous one. The class was tested to confirm it did not fire mines when it had not reached its target.

15) ProjectileEntity Testing

The **ProjectileEntity** class was tested to ensure that the **update()** functions moves the projectile a specified distance in the direction determined by its type and angle. Further tests were not necessary, as the functionality was either tested in its parent class or children classes.

16) PlayerLaser Testing

The **PlayerLaser** class was tested to confirm it always moved horizontally, with its direction determined by the player's orientation, and that it moved the correct distance.

17) LanderLaser Testing

The **LanderLaser** class was tested to guarantee it always moved at an angle equal to the angle between its parent entity and the player, and that it moved the correct distance.

18) Mine Testing

The **Mine** class, was tested to ensure that it remains stationary as the game progresses.

19) Asteroid Testing

The **Asteroid** class was tested to ensure that it always downward diagonally, with its angle being determined by its direction, and that it moves the correct distance.

VI. Critical Analysis

A. Game Functionality

In this section an analysis of the functionality of the game is presented.

1) Successes

All the functionality that is required to achieve a grade of excellent is implemented. All basic features, all minor features and three out of four major features are implemented. The game contains all the required entities as listed in Section IV, B, and all entities collide with each other appropriately.

The player object is capable of movement in all directions, and can fire a laser and turn on its shield. Landers and Bombers exist, both aiming to destroy the player. Humanoids exist and landers try to kidnap them, while the player must try to save them. Asteroids occasionally fall down around the screen, and the player must avoid or shoot them. If the player moves too much without refuelling, it will fall out of the sky and the game will end. A minimap is available that shows the location of all entities, and the user's score and the previous high score are visible during gameplay. The graphics used are good and follow an original theme, with all entities specifically designed to fit the theme. The game has a scrolling background which wraps around, ensuring the limited size of the game world, without needing walls that stip the player's movement in either horizontal direction.

2) Weakness and Issues

A weakness in the game is that hitboxes of the entities do not follow the exact contour of the entity. Hitboxes are set as rectangles around the entities, and this could lead to inaccuracies when firing lasers at some entities. An issue is also that some game functionality can be masked when the parameters are set to high, and the game is too difficult.

The feature of "pods and swarmers" is not implemented in the game. This could be seen as a weakness as the game does not include all of the major features. It was not a requirement to include all major features however it would have been useful to increase the game difficulty at higher levels.

The lack of animations could also be a seen weakness. Animations were not required in the game, but they would have bettered the user experience in the game.

3) Trade-offs

A trade-off is made to not implement the "pods and swarmers" feature as this added complexity could have made the game less enjoyable for the user as it would have made the game more difficult. Time constraints were also a factor when deciding to not implement this feature. Another trade-off made is not implementing a full leaderboard. The current scores are all stored, and this feature would be implementable in future. A trade-off is made to use more time for other aspects of the project.

B. Code Design and Quality

This section provides an in-depth analysis of the code's design and quality.

1) Successes

The code that is implemented makes use of modern C++ features where possible. The code places heavy emphasis on object-oriented programming and the *Do Not repeat Yourself* (DRY) principle. The code has been written defensively in order to produce better, more readable and less error-prone code.

To manage entities a vector of shared pointers is used. The use of smart pointers protects the code against possible resource or memory leaks [ref]. Iterator-based loops are used over index-based loops where possible.

Separation of concerns is also prioritised when designing the code. The logic layer is completely separated from the SFML graphics library and presentation layer. This would be a major success as this would allow for the code to be reused easily with other graphics libraries, with only the presentation layer needing to be changed.

The implementation of the code uses many minimal classes which each have smaller and more focused responsibilities. Using classes that are large and monolithic is avoided to create ease of reuse in the code and avoid code smells. Private functions are used in all possible places to protect the class's information. Getter and setter functions are used to allow for communication between objects and to avoid public variables, however, the use of these getter and setter functions is minimised throughout the game to preserve the object-oriented design. Interdependence of classes is also avoided, which again emphasises the decoupling of the code.

The code was intentionally designed with maintainability and modularity in mind, allowing for easy integration of new features and interactions. New entities can be effortlessly created as child classes of **GameEntity**, **EnemyEntity**, or **ProjectileEntity**. The implementation process for these entities remains simple, with minimal updates required, primarily focused on the **EntityList** enum and the **EntityManager**'s **isColliding()** function.

Furthermore, the code exhibits a high degree of separation between different game functions, each encapsulated within its own class. This separation enables the addition of new game interactions by creating new classes and utilizing composition within the **Game** class. This modular approach facilitates the integration of new functionalities without needing extensive changes to the existing codebase. This showcases the code's flexibility and ability to accommodate future expansions, showcasing its robust and scalable design, and emphasising that the code design is considered successful.

The implemented inheritance hierarchy is considered a success of the code design. Inheritance is used to reduce the reuse of code throughout the game with entities that have similar functionality being grouped together. This hierarchy emphasised the use of the DRY principle. This hierarchy can be seen in Figure 1 below. Additionally, classes that manage all entities and load in all files further reduce the repetition of these functions. An emphasis was placed on interface inheritance, allowing an interface to be shared among children classes, however implementation inheritance was also used with functions that are simple, short, and shared among all entities.

2) Weakness and Issues

Creating a new entity would require additional code to be written for the **EntityList** strongly typed enumerator, along with the collision functions. A weakness could also be that hitboxes do not fit the contour of the entities.

The **Window** class is a very large class and contains many functions. Efforts were made to reduce the size of the class; however, functions could be further split up into smaller classes. There is also some repetition in this class due to the use of the SFML framework.

The error checking and throwing of errors is also not extensive in the code. More error checks could be implemented in the game to ensure a seamless experience for the user. Unprecedented errors occurring at run time could cause impairments for the user.

3) Trafe-offs

The major trade-offs seen in the game code would be that implementing hitboxes that fit the contour of entities against the time this would have taken to complete. The implementation of "pods and swarmer's" would against having increased time to refine the code is another trade-off. The refinement and production of good quality code is prioritised in the game over the implementation of features that were not required to give an excellent grade.

C. Unit Tests

This section discusses the critical analysis of the unit tests that are implemented.

1) Successes

The testing implemented is considered successful as all the movement and functionality is confirmed. Comprehensive testing across the game logic and functionality is achieved despite some functionality not being also to be tested. A total of 123 test cases indicates the broad coverage of the testing framework.

The ability to simulate a key press through the **UserInput** enumerator class allows seamless testing of all functions that require an input from the user, such as movement, firing and shield activation. This demonstrates the success of the testing allowing them to be automatic.

The use of the *Doctest* framework including the implementation of test suites is noted as a success criterion in the project. Test suites group related tests together and provide the ability to sort through the test cases according to the suite they are in. Individual suites can be run to avoid running all 123 tests each time.

2) Weaknesses and Issues

The difficulties of testing random number generation proved to be a weakness through the testing process of the game. Tests can be long and complicated to understand for an external developer to understand and examine. Test driven coding also proved difficult during the development of the game as when using a graphics library all feedback is visual and observable immediately.

Certain functions proved difficult to provide testing for, such as, the **Game** class, the **WindowUI** class, the **Clock** class and the saving of scores. The **WindowUI** class contains most of the User Interface of the project which would only be able to be tested visually. Saving the score cannot be tested as it would change the scores in the text file which would interfere with game play scores. The **Clock** function cannot be tested due to its continuous nature.

3) Trade-offs

The major trade-offs of the testing infrastructure are the implementation of tests for a large amount of the games logic which caused lengthy test cases. The large coverage of test cases allows for a greater understanding of the codes working.

Another trade-off in the code would be having a sophisticated separation of concerns proves a difficulty in testing certain functions in the presentation layer. The priority here lies in the separation of concerns over testing certain elements of the User Interface.

VII. Future Improvements

A. Game Functionality

The "pods and swarmers" feature could be implemented. This is the only major feature of the project that is not implemented in the game. This would increase the difficulty of the game.

Animations could also be added to the game when each entity is destroyed or an animation for when the players shield is activated. Flashing animations could be implemented for when the shield is about to end or when the player is low on fuel.

A leaderboard could be implemented where a user would have to enter their name and their score would be saved after each game. This base work for this functionality is implemented with scores already being saved after each game in a text file.

B. Code Design and Quality

The inheritance hierarchy of the code could be further improved to decrease the reuse of code throughout the game. The hitboxes of the entities could follow the contour of each entity. This would allow for more accurate game play.

C. Unit Tests

A mocking framework could be a future improvement for the implementation of tests in the game. This would require the implementation of third-party libraries. This mocking framework would allow increased rigour in the testing of functions that use random events as well as GUI interactions.

VIII. Conclusion

The design, implementation, and testing of the video game *Defender:SA* has been presented in this report. An explanation of each class and their respective functionality has been given. The game behaviour has been discussed along with a full description of the testing that has been implemented in the game. A critical analysis of the game was performed analysing the successes, weaknesses and trade-offs made. Through this critical analysis the game was deemed to be a success as all of the necessary functionality was implemented. The code was analysed as was also deemed to be a success and of a high quality. Notable features of the code would be the inheritance

hierarchy, the complete separation of concerns by decoupling layers in the code. The unit tests provided thorough evaluation of all movement and game logic. Future improvements of the game could include the addition of additional features, such as a leaderboard and “pods and swarmers”. The inheritance hierarchy and hitbox size could be improved as well as a mocking framework could be adopted to increase the effectiveness of the tests performed.

IX. Bibliography

[1] “Avoid getters and setters whenever possible,” DEV Community. Accessed: Oct. 12, 2023. [Online]. Available: https://dev.to/scottshipp/avoid-getters-and-setters-whenever-possible-c8m

[2] T. Janssen, “Design Patterns Explained – Dependency Injection with Code Examples,” Stackify. Accessed: Oct. 06, 2023. [Online]. Available: https://stackify.com/dependency-injection/

[3] “Doxygen Manual: Features.” Accessed: Oct. 10, 2023. [Online]. Available: https://www.doxygen.nl/manual/features.html

[4] “Doxygen Manual: Special Commands.” Accessed: Oct. 05, 2023. [Online]. Available: https://www.doxygen.nl/manual/commands.html#cmdenum

[5] “SFML.” Accessed: Oct. 10, 2023. [Online]. Available: https://www.sfml-dev.org/

X. Appendices

A. *Table of Collision Management*

TABLE II. TABLE SHOWING THE COLLISIONS BETWEEN ENTITIES.

	Player	Player Laser	Lander	Lander Laser	Bomber	Mine	Asteroid	Humanoid	Fuel Tank
Player	O	O	X	X	X	X	X	O	O
Player Laser	O	O	X	X	X	O	X	X	O
Lander	X	X	O	O	O	O	X	O	O
Lander Laser	X	X	O	O	O	O	X	O	O
Bomber	X	X	O	O	O	O	X	O	O
Mine	X	O	O	O	O	O	O	O	O
Asteroid	X	X	X	O	X	O	O	O	O
Humanoid	O	X	O	O	O	O	O	O	O
FuelTank	O	O	O	O	O	O	O	O	O

X = left entity can collide with top entity, O = cannot collide

B. *Inheritance Hierearchy*

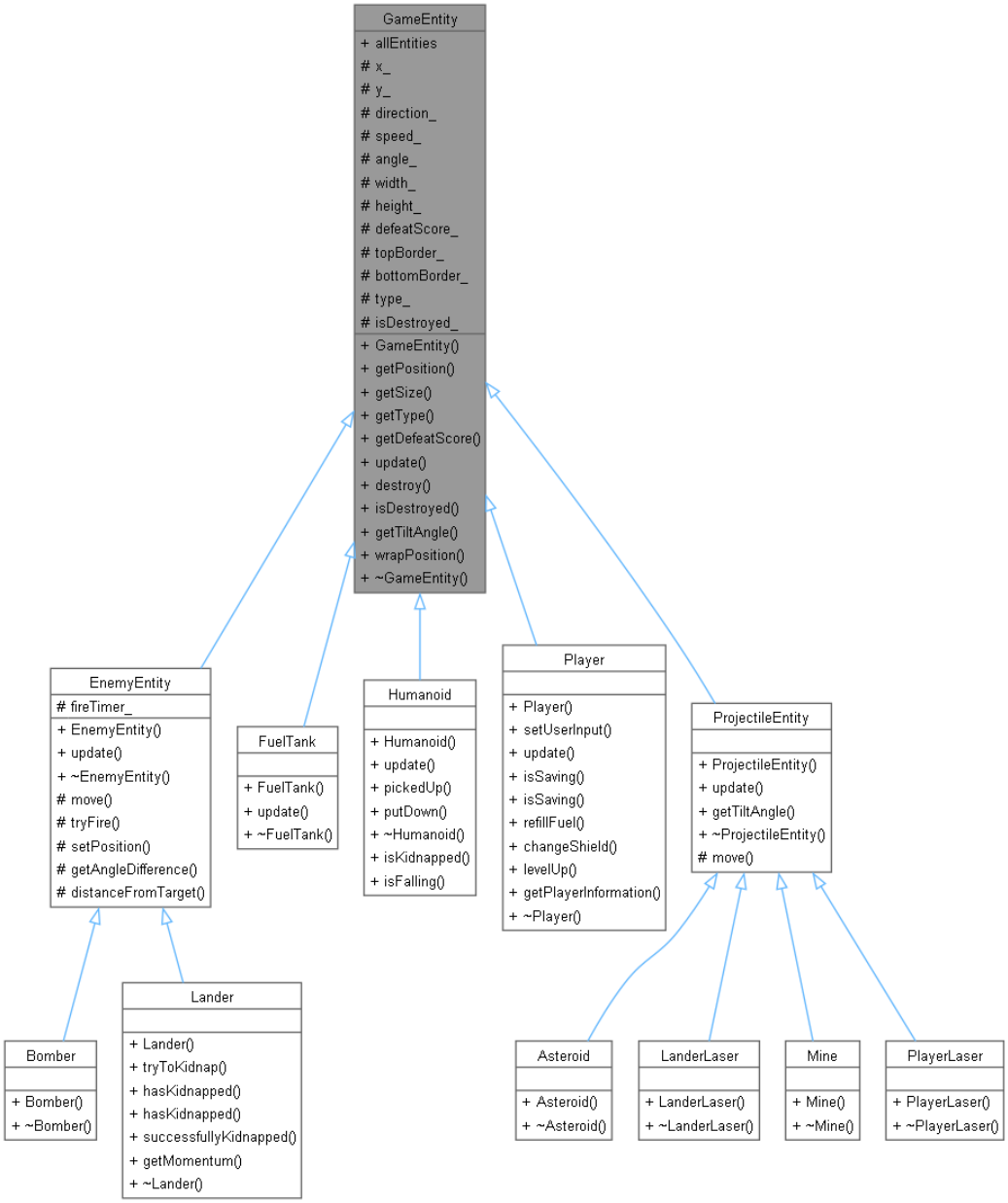

Fig. 1. A UML diagram showing the full inheritance hierarchy of the game.

C. Flow chart of Game loop

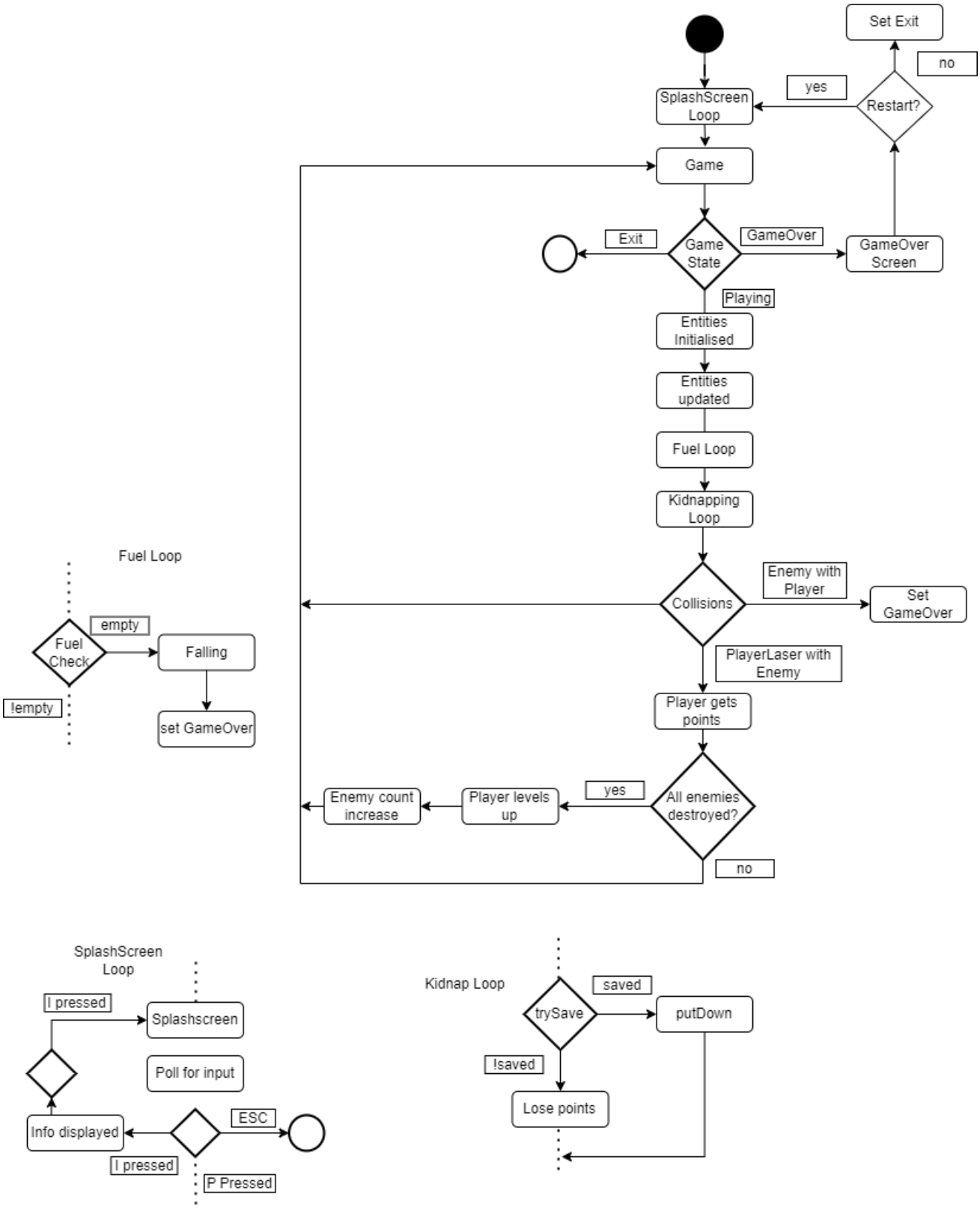

Fig. 2. Flow chart of the Game Loop

D. Sequency Diagram

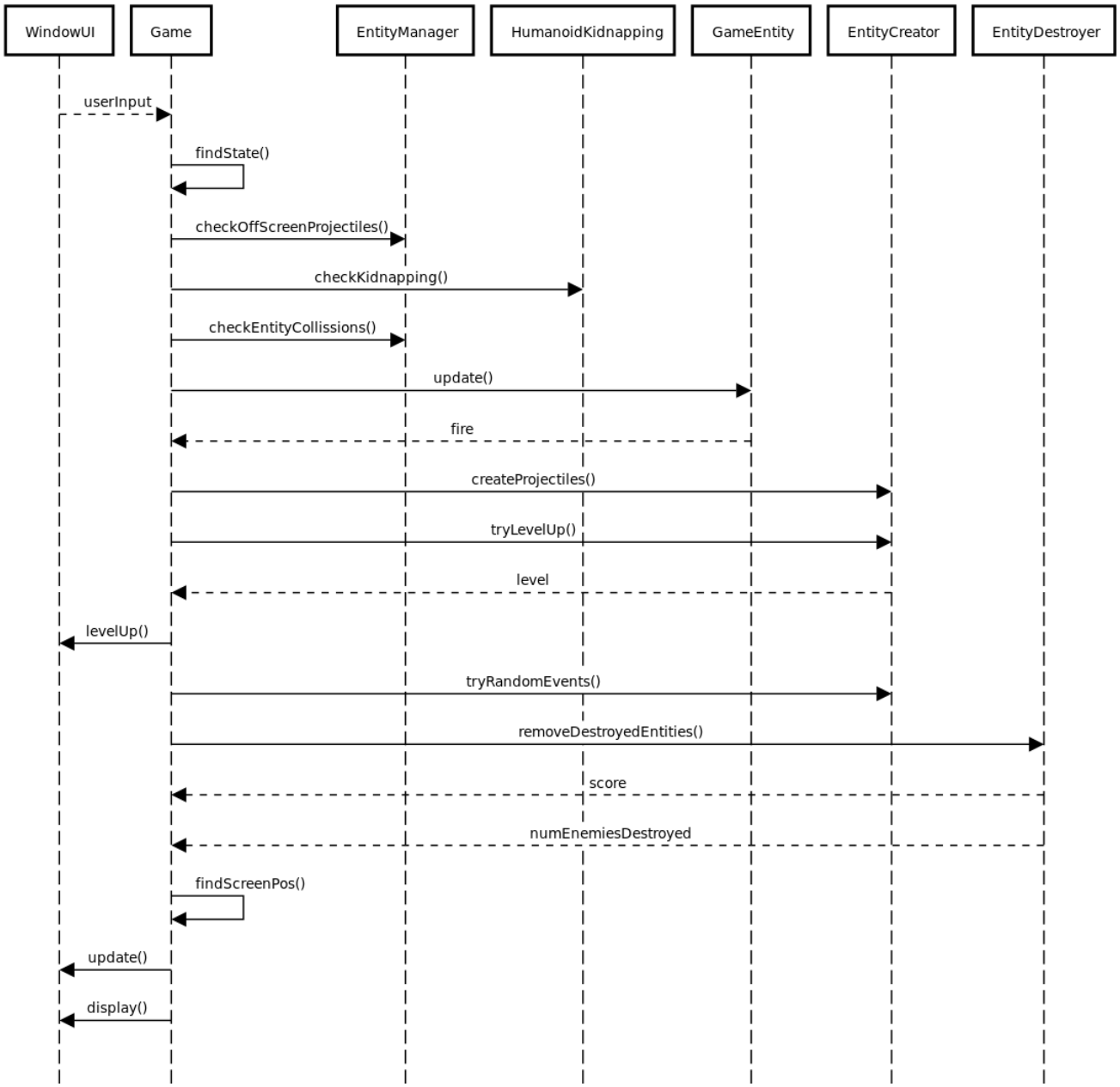

Fig. 3. Sequence Diagram showing the code structure.