# CENTIPEDE GAME DEVELOPMENT

**Lynch Mwaniki (1043475)** **Madimetja Sethosa (1076467)**

**School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa**

**ELEN3009A: SOFTWARE DEVELOPMENT II**

**Abstract:** This project presents the design and implementation of an arcade game in the fixed shooter genre written in C++17 and using SFML v2.5.0 framework in an Object Orientated design. Design decisions are discussed including the Inheritance model used. The code structure and responsibilities of the different layers is outlined, along with the roles classes played in the design and the interactions between game objects. The role of unit testing and the maintainability of the code is evaluated. The report gives a critical analysis of the implemented design detailing some of the flaws and proposes some alternative designs.

**Key words:** C++17

## 1. INTRODUCTION

Centipede is a fixed shooter arcade game released by Atari Inc. in June 1981. The project discussed in this report implemented a variant of the original game but with the same game mechanics. In the built game the Player moves in a confined area on the screen. The Player can move right, left, up or down in the area with the use of a keyboard. The Centipede train gets spawned from the top centre of the screen and other moving game enemies are spawned either from the left or right boundaries of the screen. The screen is populated with Mushrooms placed at random positions. Mushrooms act as obstacles for the Centipede and the Player. Scorpions move outside the Player's area and poison Mushrooms as they move across the screen. Centipedes that collide with poisoned Mushrooms dive directly into the Player's area. Spiders move in a randomized zigzag movement across the Player's area and eat Mushrooms in their path. Spiders and Centipedes attempt to kill the Player by colliding with the Player. The Player can shoot a projectile towards the top of the screen and destroys game enemies for points.

The game consists of five levels, each with an initial Centipede train, Scorpion, delayed Centipede heads and periodically generated Spiders. As the Player progresses through the levels, the number of Centipede heads increase and the frequency of the Spiders increase. The Player's weapon gets upgraded only once, when they advance to the next level, which is lost upon death. The Player loses if they die three times, and they win the game if they surpass all the game's levels before losing all their lives.

The report details the design of the code structure, the implementation, behaviour and testing of the code. A critical analysis of the design is given as well as recommendations that could improve the design.

## 2. CODE STRUCTURE

This section discusses the Domain Model and the different types of code structures investigated to be implemented in the project.

### *2.1 Domain Model*

The project's game-play is modelled as a collection of objects, visually represented, either placed stationary in a confined space or moving: into, around or out of a confined space. The game consists of various entities which are generated and contained within the game area. Table 1 lists the different game entities and their behaviour. There are some similarities between the entities, such as a position, can be poisoned and has a movement.

A requirement of the project was to have an object orientated solution, thus the design of the code structure was influenced by this requirement. The collection of similarities between entities led to the use of an objected orientated approach that uses inheritance. The objects in the game were split into two groups namely stationary entities and moving entities. Two pure interface classes were created, an *IEntity* and an *IMovingEntity*, to be used as base classes for the respective entity groups. All *IMovingEntity* and stationary objects inherit from *IEntity*, whilst moving game objects inherit from *IMovingEntity*. The implemented inheritance hierarchy can be seen in Figure 1. An Inheritance hierarchy was chosen to allow derived class objects to implement their own unique movements if any and their different states in order to achieve game mechanics.

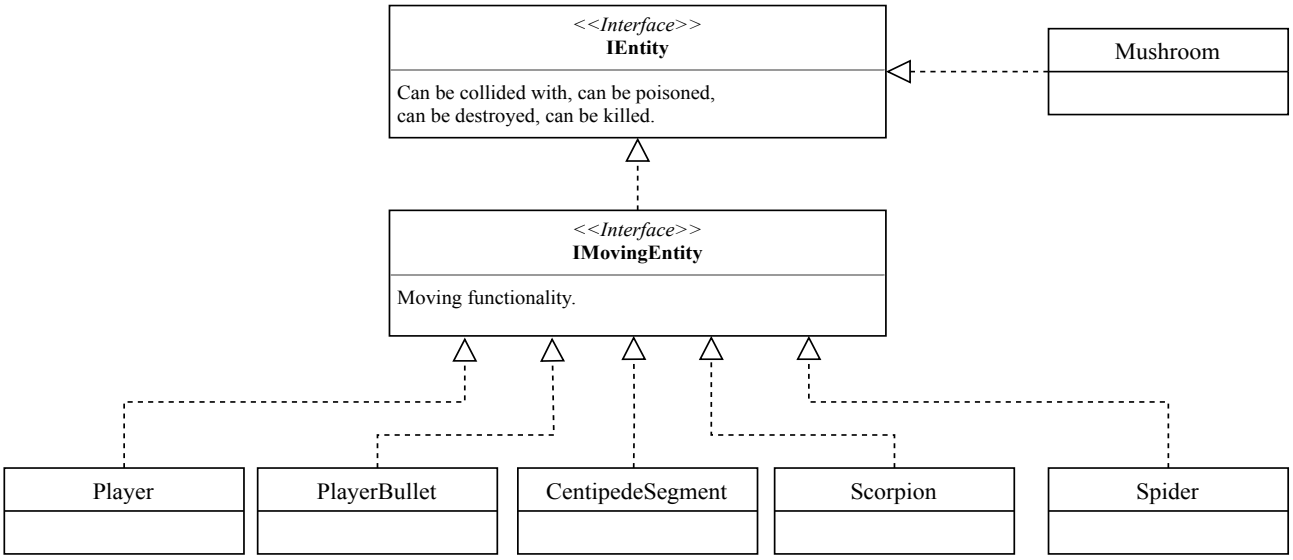


Figure 1: Inheritance Hierarchy Diagram

Table 1: Table Showing Entity Behaviour

| **Entity** | **Behaviour** |
|---|---|
| Player | Moves up, down, left or right in a confined area on the screen. |
| Player Bullet | Moves upwards from the Player's position. |
| Centipede | Moves left to right across the screen going downwards towards the Player's area. Moves upwards when screen bottom boundary is reached and stays in the Player's screen area. Goes down one row and changes direction when it collides with a Mushroom. When poisoned it dives straight down towards the Player's screen area. |
| Mushroom | Stationary. |
| Scorpion | Moves horizontally across the screen outside of the Player's screen area. |
| Spider | Moves randomly in a zig zag movement in the Player's screen area. |

### *2.2 Separation of Concerns*

The Separation of Concerns principle is used in programming to ensure that the computer program is separated into distinct sections, such that each section addresses a separate concern. The sections are known as the Presentation, Logic and Data Layer. The Presentation Layer is responsible for displaying the Graphical User Interface (GUI) and capturing inputs from the user. The Logic Layer handles the main functionality of the program such as calculations, logical decisions and data processing. The Data Layer is used for accessing resources such as text-files or images.

Because layer separation is important for a good design, the Presentation, Logic and Data layer have been carefully designed to ensure that separation of concerns is adhered to, which allows dependency decoupling. The layers communicate through objects which ensures that only the necessary information is exposed to the other layers while keeping the majority of layer's information hidden.

## 3. IMPLEMENTATION

The game was divided up into 3 layers in order to separate concerns as detailed in Section 2. Each of the classes implemented is can be grouped into one of the layers.

### *3.1 Data Layer*

The data layer has the responsibility of retaining information about the assets that will be required to render the logic of the game, as well as provide any data required at runtime. The assets needed in the game are sounds, textures and font styles. Textures are the most important items as they are needed to render objects on the interface. The *AssetManager* class was given the responsibility of retaining paths to the assets used. Further implementation was required to load the high score from a text file and to read the high score as well. This responsibility was given to the *HighScoreManager* class.

*3.1.1 AssetManager Class:* This class maps file paths on the disk to the relevant assets used by objects. This allowed the mapping of a sprite to a texture, sounds based on an object type and the ability to display text in the Presentation Layer. The textures used are in *.png* format and the sounds used are in *.wav* format.

*3.1.2 HighScoreManager Class:* This class simply reads the high score from a text file and writes the high score to a text file. It is controlled by the Logic Layer which then

provides the high score data to the Presentation Layer to be displayed on the interface.

*3.1.3 Position Class:* A class that holds an x and y screen position values. This class allows objects to create instances of their position and Axis Aligned Boundary Box's vertices instead of defining setters and getters for each class to retrieve the respective variables.

*3.1.4 Grid Class:* A class that holds information about the screen width and height. This information is taken in by the constructor and allows all the objects that use a grid to determine their movement behaviour or their functionality.

*3.1.5 Dimensions Structs:* Each of the child classes inherited from either *IEntity* or *IMovingEntity* consists of a struct that holds information about an object's height, width and movement speed. This information used by the object to construct its *BoundaryBox* and its movement function.

*3.1.6 Direction enum class:* A strongly typed enum class representing the directions game objects could be facing. Used in the game to indicate which direction game objects are facing. The directions are UP, DOWN, LEFT and RIGHT.

*3.1.7 ObjectType enum class:* A strongly typed enum class representing the type of game objects. The object types are PLAYER, PLAYER_LASER_BULLET, CENTIPEDE, MUSHROOM, SCORPION and SPIDER.

### *3.2 Logic Layer*

The Logic Layer has the responsibility of processing, making decisions and tells the other program components what to do and when to do it. This layer sends data and information from the Data layer to where it needs to go, usually the Presentation Layer. The Presentation Layer also sends data such as user input back to this layer for processing. The following logic classes were implemented in the game and are discussed.

*3.2.1 StopWatch Class:* This class is used to represent a timer. It is used to lock the screen frame rate which makes the game speed independent of CPU speed. Locking the frame rate limits the number of times updating, rendering and other functions in the game loop are called. This is done by quantifying the amount time which has passed per frame and using that to update entities movement or state.

*3.2.2 IEntity Class:* This class is a pure interface class which serves as a base class for the hierarchy. It contains pure virtual functions that generalizes commonality between entities in the game. The functions allow derived objects to get their Position, entity type, kill points, boundary box, remaining lives and poisoned status. The functions also allowed objects to change their alive status, reincarnate themselves and change their poisoned status.

*3.2.3 IMovingEntity Class:* This class is a pure interface class which inherits from *IEntity* class. It adds extra functionality for derived class objects to be moved.

*3.2.4 BoundaryBox Class:* This class is used to construct a rectangle shape. The rectangle shape or boundary box is constructed for each game object per frame based on an object's position and its rotation angle. This boundary box object is later used by collision detection. This class has five *Position* class objects. Four of them represent the location of vertices on the grid and the fifth one represents the object's position which is the centre of the boundary box.

*3.2.5 Mushroom Class:* This class inherits from the *IEntity* class. It is used to model a Mushroom object in the game. A mushroom is a stationary object in the game. It can be poisoned by a *Scorpion* and it dies either after being shot at four times or gets eaten by a *Spider*. It does not kill the *Player* if they collide with it.

*3.2.6 Scorpion Class:* This class represents a Scorpion. It inherits from the *IMovingEntity* class. A Scorpion moves horizontally across the screen. Its movement is restricted to outside of the Player's screen boundary. And when it collides with a *Mushroom* it changes the state of the *Mushroom* to poisoned. A Scorpion dies either when it is shot or when it reaches the screen's left or right boundaries.

*3.2.7 Spider Class:* This class represents a Spider. It inherits from the *IMovingEntity* class. A Spider moves in a randomized zigzag movement across the screen. Its movement is restricted to the Player's screen boundary. It eats *Mushroom* in its path and the *Player* loses a life when they collide. A Spider dies either when it is shot or when it reaches the screen's left or right boundaries.

*3.2.8 CentipedeSegment Class:* This class represents a Centipede segment. It inherits from the *IMovingEntity* class. A Centipede segment can either be a head or a just a body. if the segment is a head, it moves left and right, and moves down every time it collides with a mushroom or hits the screen's left or right boundary. If it collides with poisoned mushroom, it dives towards the player's screen boundary and then proceeds normally. The rest of the body segments follows the head. It advances towards the player's screen boundary and moves up and down the boundary once it hits the bottom boundary of the screen. A centipede segment dies either when it is shot or collides with the player.

*3.2.9 MushroomFactory Class:* This class represents a factory for Mushrooms. It creates Mushrooms at random positions on the screen excluding the player's screen area. The mushroom factory makes use of a grid to avoid

mushrooms from overlapping. The factory can also create a mushroom at a given position by first mapping it to the respective cell on the grid.

*3.2.10 GameEngine Class:* This class contains functionality to instantiate game objects, tell game objects to move, check for collisions and update the game's high score. This class will be controlled by the *Logic* class to set the game's behaviour.

*3.2.11 PlayerBullet Class:* This class represents a projectile that a *Weapon* can shoot. It inherits from the *IMovingEntity* class. It gets created at the position when it was fired. It moves upwards towards the top screen boundary and dies if it reaches it. It can collide with a *Mushroom*, *CentipedeSegment*, *Scorpion* and *Spider*.

*3.2.12 Weapon Class:* This class represents a weapon that can shoot *PlayerBullet* objects. The *PlayerBullet* objects are generated at a position given to the weapon's shoot function. It is used by the *Player* to shoot *PlayerBullet* objects in the game. It makes use of the *StopWatch* class to create a delay between the generation of bullets. A weapon can be upgraded once, this operation cuts the reload time of the *Weapon* object by half its original reload time delay. A *Weapon* object also has functionality to reset an upgrade.

*3.2.13 Player Class:* This class represents a Player. It inherits from the *IMovingEntity* class. The player's movement is restricted to a certain screen area. Its movement and shooting is controlled through polling keyboard input using SFML. When the Player shoots, it tells the *Weapon* class to generate a *PlayerBullet*. This class keeps track of the number of lives of the player and is responsible for decrementing the number of lives and updating the alive status of the Player. A Player loses a life when it collides with *CentipedeSegment* or *Spider*. When a Player reincarnates their upgraded *Weapon* gets reset.

*3.2.14 SeparatingAxisTheorem Class:* This class implements the Separating Axis Theorem (SAT) algorithm [1]. This is used later in *CollisionDetection* to determine if any overlap occurs between two *BoundaryBox* objects. A further explanation is present in Section 4.5.

*3.2.15 SpatialHash Class:* This class implements a Spatial Hash [2]. This is used later in Collision Detection to speed up collision detection by reducing the number of checks that have to be made. A further explanation is present in Section 4.5.

*3.2.16 CollisionDetection Class:* This class is responsible for detecting all the collisions between different entities in the game. It finds a pair of entities that have collided and either moves them out of collision or calls their respective *eliminated()* functions. This might set an object to dead depending on the number of lives an object has left. A more detailed explanation of the implementation is provided in Section 4.5.

*3.2.17 Logic Class:* This class acts as the controller of the game. It makes use of the public interface of the *GameEngine* class to set the game's behaviour. It interacts with the *Presentaion* to continuously render the game's logic. It checks for game end conditions such as when the *Player*'s lives are depleted or if they have completed all the levels. It ensures the game ends when either conditions are meet. A further explanation is present in Section 4.1.

*3.2.18 EnemyFactory Class:* This class had the responsibility of generating *Spiders*, *Scorpions*, a Centipede train using *CentipedeSegments* and *CentipedeSegments* defined as Centipede Heads. The generation of *Spiders* and Centipede Heads is done periodically throughout the game. A further explanation is present in Section 4.4.

### *3.3 Presentation Layer*

This layer has the responsibility of polling for user input and does the visual rendering of the game objects to display something meaningful to the user. SFML is used as the graphics library in this layer.

*3.3.1 Splashscreen Class:* This class renders the game's Splash screen. It has instructions on how to play the game and it exists till the user presses the Enter key.

*3.3.2 Presentation Class:* This class is responsible for drawing all the game objects on the screen, playing the sounds of moving game objects and playing the sound when the Player shoots a bullet. The *Logic* class provides it with a vector of *IEntity* objects that are looped through. It retrieves the object type and maps it to the correct texture from the *AssetManager* and for those with animated movement or different states, it crops the texture using a *SpriteSheet* object.

*3.3.3 SpriteSheet Class:* This class is used by the *Presentation* class to crop textures that have multiple images in them to render the different states of an object or to iterate through images to render an animated movement. It has a pointer to a texture for each object with animated movement and is given information detailing the number of rows and columns in a texture based on the amount of images present.

*3.3.4 GameOverScreen Class:* This class renders the game's Game Over screen. It gets displayed when the *Logic* class has determined that the Player has lost all their lives. It renders the Player's score and the game's current high score to the user.

*3.3.5 GameWonScreen Class:* This class renders the game's Winner screen. It gets displayed when the *Logic* class has determined that the Player has completed all the

game's levels. It renders the Player's score and the game's current high score to the user.

## 4. GAME BEHAVIOUR

This section details the way in which the game behaves using the code structures that were implemented. The Game Loop, User Input, Mushroom generation, Enemy generation, Collision Detection and Enemy Movement are the topics discussed in this section.

### *4.1 Game Loop*

The game operations are managed in the active game loop found in the *Logic* class. The sequence of operations is summarized in Figure 2. It can be seen that the game loop follows the process of polling for user input, creating the required entities, moving game objects, check for collisions, removing dead entities, rendering game entities and finally updating the screen state.

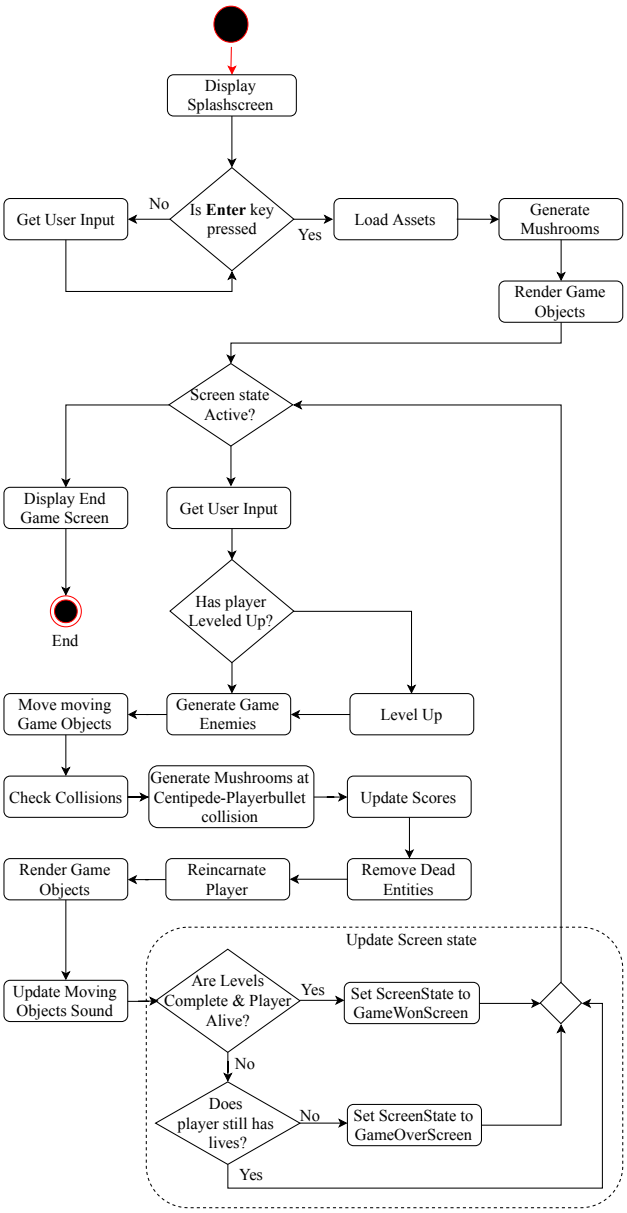


Figure 2: Game loop

The game loop was implemented in conjunction with a timer from the *StopWatch* class. This allowed for the decoupling of the game speed from the processor speed to ensure that the game ran at the same speed on different computer hardware configurations.

The key design in allowing the game loop to work as intended, was through the use of object conversations that allowed the *Logic* class to access object's public interfaces which were required to run the game. A class diagram of the game's main classes can be seen in Appendix A Figure 7. A detailed visual illustration of the object conversations can be viewed in Appendix B Figure 8.

### *4.2 User Input*

The keyboard input from the user is captured through the *Presentation* class which uses functionality in SFML to poll for events. The input captured is sent to the *Logic* layer for it to be processed. The game loop timer helps control the sensitivity of the user's input. It does so by ensuring that *Player* does not move too quickly across the screen while an arrow key is pressed. The *Player* shoot key, Space bar, is also de-bounced to avoid the creation of *PlayerBullet* objects while holding down the key.

### *4.3 Mushroom Generation*

Mushroom generation was performed in the *Mushroom-Factory* class . This was performed by dividing the grid of size $592 \times 640$ pixels into cells of the same size as the *Mushroom* texture which has a size of $16 \times 16$ pixels. This gave a result of 37 columns and 40 rows. The number of rows and columns were used to get the total number of cells. The cell ID allocation can be viewed in the left of Figure 3.

Each cell is paired with a bool (initially false) indicating whether the cell is occupied or not as seen in the right of Figure 3. The pairs are then inserted into a map/hash-table. For populating the grid with mushrooms at random positions, the process is randomized by choosing a random column between 0 and 37, and a random row between 0 and 31. The rows are limited to 31 to avoid initially putting mushrooms in the player's screen boundary. The row and column are then mapped to its respective cell ID to check if it is occupied. The row-column to cell ID mapping is performed using Equation 1.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | ••• | 36 | 37 |
| 38 | 39 | 40 | 41 | ••• | 73 | 74 |
| 75 | 76 | 77 | 78 | ••• | 110 | 111 |
| 112 | 113 | 114 | 115 | ••• | 147 | 148 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋱ | ⋮ | ⋮ |
| 1074 | 1075 | 1076 | 1077 | ••• | 1109 | 1110 |
| 1111 | 1112 | 1113 | 1114 | ••• | 1146 | 1147 |

| | |
|---|---|
| 1 | false |
| 2 | false |
| 3 | false |
| 4 | false |
| ⋮ | ⋮ |
| 1146 | false |
| 1147 | false |

Figure 3: Left:Grid cells ID, Right: Cell ID mapping

$$cell\_ID = maxColums \cdot (row + 1) + (column + 1) \quad (1)$$

If the cell is not occupied, the row and column are then mapped into a position on the grid. The $x$ and $y$ position, which represent the centre of the cell on the Grid are calculated using Equations 2. The $x$ and $y$ position is then used to create a Mushroom. A maximum number of a 109 Mushrooms are then created.

$$\begin{aligned} xPos &= round(column \cdot 16 + 8.0) \\ yPos &= round(row \cdot 16 + 24.0) \end{aligned} \quad (2)$$

The creation of a Mushroom given at a position also makes use of grid cells. Given a position ($x$ and $y$), the row and column are calculated by manipulating Equations 2. The calculated row and column are rounded to the nearest integer. The position on the grid is then calculated using Equations 2. This is to make sure that the Mushroom is created at the centre of a cell. This functionality is used for generating Mushrooms at the positions of dead centipede segments.

### *4.4 Enemy Generation*

The enemy generation is controlled by the *EnemyFactory* which dictates when, where and how many enemies are generated for each level of the game. There are four types of enemy objects which are: A normal *Centipede* (1 head and 9 body segments), *Centipede* heads, a *Scorpion* and a *Spider*. The normal *Centipede* and the *Scorpion* are generated at the beginning of each and every level of the game. The *Spider* is also generated in every level but only after some time has passed since the level started. The *Centipede* heads are generated from Level 2 onwards, the number of *Centipede* heads per level can be viewed in Table 2.

Table 2: Table Showing the number of Centipede Heads generated per level.

| | **Number of Centipede Heads Generated** |
|---|---|
| **Level 1** | 0 |
| **Level 2** | 2 |
| **Level 3** | 3 |
| **Level 4** | 3 |
| **Level 5** | 3 |

The generation of the *Spider* and the *Centipede* heads is delayed in each level. The *Centipede* heads are only generated once per level where as the *Spider* is generated periodically. The delays can be viewed in Table 3.

Table 3: Table Showing delay of periodically generated game objects

| | **Delay (seconds)** | | | | |
|---|---|---|---|---|---|
| **Game Object** | **Level 1** | **Level 2** | **Level 3** | **Level 4** | **Level 5** |
| Spider | 15 | 12 | 9 | 6 | 3 |
| Centipede Heads | | 7.5 | 7.5 | 7.5 | 7.5 |

#### *4.4.1 Centipede Train Generation:*

The Centipede train was generated using *CentipedeSegment* objects and defining their body type as either a head or body using a strongly typed enum. Their initial direction, either facing left or right, was randomly generated. The Centipede train was then pushed into the respective game object vectors. Vectors in C++ make use of contiguous memory thus it was observed that the Centipede train segments would always be adjacent to one another in memory even after the removal of dead segments. It was decided to use these properties of a vector to manage a Centipede train in the game.

### *4.5 Collision Detection*

Collision detection is an important component in the functionality of the game as most of the game features are dependent on being able to register collisions between game entities. Due to the generation of many *Mushroom* objects in the game, this increases the number of collisions that need to be checked. Research was done to accurately detect collisions and create a computationally efficient algorithm.

The collision detection algorithm chosen was the SAT algorithm [1]. This algorithm checks for overlap between two convex polygons. The general idea of the algorithm is, if there exists an axis along which the projections of the two polygons do not overlap then the two polygons do not overlap. The algorithm implemented generates 4 axes to be checked for overlap, two from each polygon. The algorithm varies in complexity because if the first axis it checks finds an overlap then it has to iterate to the next one and check for overlap. The polygons chosen were rectangles which are generated by the *BoundaryBox* class.

In order to prevent the checking of every game object against every other game object, a Spatial Hash was implemented in the *SpatialHash* class [2]. A Spatial Hash aims to reduce collision checks by dividing the screen into cells and keeping track of which cells objects are located in. This data is stored in a hash table which can be used to query the nearby objects around the object of interest. The benefits of this is dependent on choosing a cell size that's not too large otherwise too many collisions checks will occur. Figure 4 shows a high level visualization of the structure of the *SpatialHash* class. The cell size chosen was $80 \times 80$ on the screen grid size of $592 \times 640$.

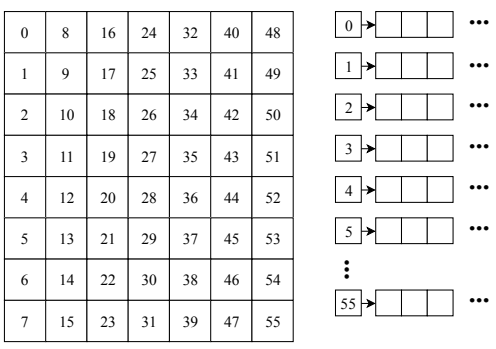


Figure 4: Spatial Hash visualization: Left: Spatial Hash Grid, Right: Hash Table

An object's position and boundary box vertices are all mapped to a cell using Equation 3, where x and y are screen positions.

$$cell\_ID = x \cdot (GridHeight \div CellSize) + y \qquad (3)$$

If an object's position or boundary box can be found in multiple cells it will be placed into the respective cells in the hash table where the overlaps occur. This reduces the collision checks from $O(n^2)$ to $O(n)$, as an object only checks for collision with objects in the cells it is located in. The algorithm's complexity is further reduced by only iterating through movable objects. Figure 5 shows how the game objects interact with each other by illustrating the collision relationships between them.

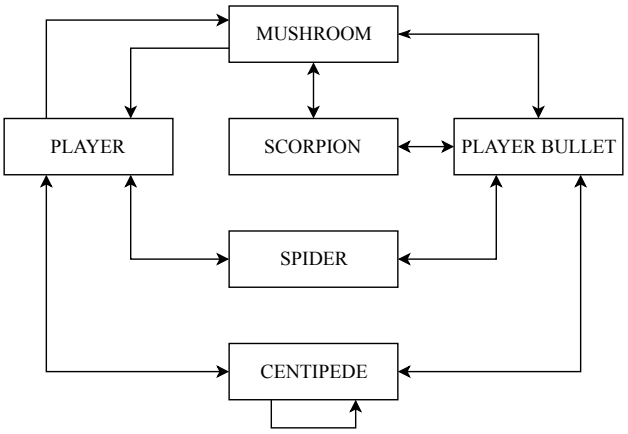


Figure 5: Illustration of collision relationships between objects.

### 4.6 *Moving Entities Movement*

The implemented moving game objects each have different movement styles which are implemented in their respective classes. The following sections discuss the different movement implemented in each object.

*4.6.1 Player's Movement:* The Player's movement depends on the user's input. The inputs are then processed to updated the direction of the player. The move function is then called, and the player moves in the direction corresponding to the inputs provided that the player does not go out of its screen boundary.

*4.6.2 PlayerBullet's Movement:* The *PlayerBullet* is created when a player shoots. It moves vertically upwards when ever its move function is called.

*4.6.3 Scorpion's Movement:* The *Scorpion* object is created either at the left or right boundary of the screen. Its movement is horizontal across the screen. When it is created at the left boundary it moves towards the right boundary and vice versa.

*4.6.4 Centipede's Movement:* The *CentipedeSegment* moves left and right as it advances towards the player's screen boundary. It moves down one row and changes direction either when it hits a boundary, a mushroom or another *CentipedeSegment*. The *CentipedeSegment* moves up one row if it has hit the bottom boundary and moves up and down the player's screen boundary. When a Centipede train collides with a poisoned mushroom, it dives straight to the player's area and continues with its normal movement upon reaching it.

*4.6.5 Spider's Movement:* The *Spider*'s movement is in a randomized zigzag fashion across the player movement area. The randomized zigzag movement is achieved by generating a random point, then calculating the slope. The *Spider* then moves towards the point using the slope, when the point is reached, a new point and slope are calculated.

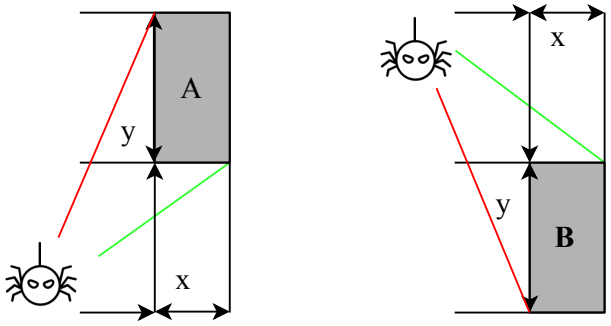


Figure 6: Spider movement : Left:Spider moving up, Right:Spider moving down

Figure 6 shows the region in which a random point is to be chosen from when the *Spider* moves up and down. Since the point is randomized, the x and y range is limited to make sure that there is never an infinite slope. The range for the up movement is depicted in Figure 6 on the left by the region marked A and for the down movement this is depicted on the right by the region marked B. The red lines indicate the maximum slope and the green lines the minimum slope.

## 5. CODE TESTING

A Test Driven Development (TDD) approach was used in developing the game. For most of the classes created, especially the ones inherited from the *IMovingEntity* class, tests were made to check the functionality of the derived objects from different class if they are behaving as intended. This approach was used due to the differences in how each of the game objects that inherit from *IMovingEntity* move. The tests were implemented using version 1.2.9 of the Doctest framework for C++. The public interfaces of the classes were tested to ensure that they behave as expected.

Table 4 indicates the number of unit tests performed and the number of assertions for each tested class. The tests performed on some of the game's classes passed the expected conditions. This indicates that the classes worked as intended.

Table 4: Unit tests summary

| TESTED CLASSES | TESTS | ASSERTIONS |
|---|---|---|
| EnemyFactory | 6 | 6 |
| PlayerBullet | 3 | 4 |
| GameEngine | 5 | 19 |
| Grid | 1 | 2 |
| Mushroom | 4 | 4 |
| Player | 6 | 11 |
| AssetManager | 7 | 14 |
| BoundaryBox | 4 | 24 |
| CentipedeSegment | 6 | 14 |
| CollisionHandler | 12 | 48 |
| HighScoreManger | 1 | 1 |
| MushroomFactory | 2 | 2 |
| Position | 3 | 5 |
| Scorpion | 3 | 5 |
| SeparatingAxisTheorem | 3 | 4 |
| SpatialHash | 3 | 5 |
| Spider | 3 | 4 |
| Weapon | 3 | 4 |
| **Total** | **75** | **176** |

### *5.1 TestGrid*

The tests performed on the grid were to check if a *Grid* object can be created with default values, and its width and height can be retrieved through its getter functions.

### *5.2 TestPosition*

The tests performed on the *Position* were to test the getters and setters of the x and y members of the position. Furthermore, the equality and the subtraction overloaded operators were also tested.

### *5.3 TestBoundaryBox*

The *BoundaryBox* object was tested to check if the boundary box is created with the right parameters and it can be rotated clock-wise and anti-clock-wise whilst retaining its size and centre position.

### *5.4 TestPlayer*

The first test performed on the *Player* object was to check if the player can be created at given default position on the grid, and its position can be retrieved. The player's movement was also tested to check if the player is able to move in the given direction and its position changes respectively. The other test include checking if the boundary box is created correctly around the player, the player's score can be set and retrieved. The player's remaining lives can be retrieved and the player is able to shoot.

### *5.5 TestCentipede*

The *CentipedeSegment* object was tested to check if the segment can be created at a given position. The movement of the segment was also tested to check if the segment is able to move up, down, left and right. Furthermore, the Boundary movement was tested to check if the segment moves down one row either when it hit the left or right boundary, and also moves up when it hits the bottom boundary.

### *5.6 TestMushroom*

The *Mushroom* object was tested to check if its getters and setters function properly. This includes creating a mushroom at a specified position and is position can be retrieved. The mushroom can be poisoned. The number of remaining lives can be retrieved and the mushroom is not alive after being shot four times.

### *5.7 TestSpider*

The *Spider* object was tested to check if it can be created at a random position at the left and right boundary of the *Player's* screen boundary. The other tests were to check if its getters and setters function properly. The move function was also tested to check if the movement of the *Spider* object is randomized. Finally, it was also tested to check if the spider dies ones it goes beyond the left or right boundary.

### *5.8 TestScorpion*

The *Scorpion* object was tested to check if it can be created at a random position of its allocated range either on the left or right boundary of the screen. The getters and setters were tested to check if the position and the direction of the scorpion can be retrieved. The movement of the scorpion was also tested to make sure that it moves in the right direction. Finally, it was also tested to check if the scorpion dies when it moves beyond either the left or right boundary.

### *5.9 TestMushroomFactory*

The *MushroomFactory* object was tested to check if it is able to generate mushrooms at random positions on the screen. The other test performed was to check if the factory is able to create a mushroom at a given position.

### *5.10 TestSpatiaHashing*

A *SpatialHash* object was tested to check if objects were placed at in the correct cells defined in the grid constructed. A test was performed to determine that objects in different cells would not be retrieved when nearby objects were requested. A test was also performed to see if objects in the same cell as the object of interest were retrieved successfully. A test case to determine whether an object on a cell's boundary gets placed into the cells it overlaps was also performed. These tests were necessary as moving objects continuously move on the grid and their cell locations change.

### *5.11 TestSeparatingAxisTheorem*

A *SeparatingAxisTheorem* object was tested to check if it can detect overlap between two *BoundaryBox* objects. A test was also performed to determine the Minimum Translation Vector to move two *BoundaryBox* objects out of overlap.

### 5.12 TestHighScoreManager

A *HighScoreManager* object was simply tested to see if it can retrieve the high score from a textfile and write the high score to a textfile.

### 5.13 TestPlayerBullet

The *PlayerBullet* object was tested to check it can be created from at a given position, and move vertically from its initial position towards the top boundary of the screen, and when it moves beyond the top boundary it dies.

### 5.14 TestWeapon

A *Weapon* object was tested to determine if a *PlayerBullet* can be created after a delay, reload time, implemented in the *Weapon* object and at the position given. A test was also performed to determine if a *Weapon* can be upgraded which decreases its reload time delay.

### 5.15 TestAssetManager

An *AssetManager* object was tested to determine if the correct paths were assigned to the correct game asset. Tests were also performed to determine if the correct number of images found in a game object's texture were stored correctly. The textures used in the game had multiple images in one texture to represent the different states of an object. The number of images needed to be saved in order to crop the texture to get the different states.

### 5.16 TestCollisionHandler

A *CollisiionHandler* object was tested to determine if collision detection between the entity relationships from Figure 5 performed correctly. All the defined relationships were tested. The respective objects were created and pushed into a vector and passed to the *CollisionHandler* object. The movable objects were moved to cause a collision and the objects were checked if they had lost a life, moved out of the collision or changed the state of the object they collided with.

### 5.17 TestGameEngine

A *GameEngine* object initialization was tested to ensure the game logic started with the correct parameters such as starting on level one. It was tested further to determine if game enemies can be successfully generated and moved in the game's logic. Tests were also performed to determine if keyboard press events could successfully move a *Player* object and instruct the *Player* object to shoot *PlayerBullet* objects.

### 5.18 TestEnemyFactory

The *EnemyFactory* object was tested to ensure that the creation of *Centipede* segments, a *Spider* and *Scorpion* was successful. In order to test *Spider* and *CentipedeSegment* (of type head) generation, the delays were set to zero which allowed the objects to be generated. Tests were also performed to ensure periodically generated enemies were not created before the delay had elapsed. Each of the tests passed which indicated that the class behaved as expected and enemies would be generated at the correct intervals.

## 6. GAME FUNCTIONALITY

The project solution meets the project specifications for basic functionality and usability of game mechanics. The Player is able to move up and down in a small confined area of the screen. The Mushrooms are initially randomly generated on the screen outside of the Player's area. The Player's remaining lives, score and the game's high score are depicted at the top of the game screen. The Centipede moves left and right while advancing down one row and changes direction upon reaching the boundary or colliding with a Mushroom. The Scorpion moves across the screen and never in the Player's area, it also poisons every Mushroom it comes into contact with (shown graphically). A Centipede that collides with a poisoned Mushroom dives straight down towards the Player's movement area, it then returns to its normal behaviour upon reaching it. The shot Centipede segment is turned into a Mushroom. Spiders appear periodically and move in a random zigzag fashion across the Player's movement area, and they eat the Mushrooms. This meets the criteria for all four minor features and three of the major feature enhancements. Unit testing is implemented for all basic movement on all moving game entities and all logic-related and data-related game classes. Pre and post game splash-screens provide information and analysis of the game.

## 7. MAINTAINABILITY

Software requires ongoing maintenance throughout its life cycle thus maintainability is an aspect that demands effort from the developer. The addition of new features or the need to fix bugs is expected throughout a software's life span. Code that is maintainable has the property of orthogonality, which means that changes to code should be easy to do and does not affect other components negatively. The project code presented is classed based and modular and follows the DRY principle. The classes discussed in the Implementation section split functionality into short pieces of code implemented in different functions. It can be observed from the code structure that changes made to the Inheritance Hierarchy have to ripple down to the derived classes that implement them. This affects the maintainability of the code as multiple class's implementations have to be changed when functionality is added, removed or changed at the top of the hierarchy. The separation of concerns in the code does simplify program maintainability. Unit testing is implemented for all basic movement on all moving game entities and the logic-related and data-related game classes. Pre and post game splash screens provide information and analysis of the game. Player input is respected, decoupling keyboard input events from frame rate updates.

## 8. CRITICAL ANALYSIS AND RECOMMENDATIONS

This section details the functionality achieved by the program and the flaws present. The implementation

presented meets the specified success criteria for the project, but some design decisions warrant discussion and evaluation.

### *8.1 Design and Implementation*

The separation of concerns is successfully implemented in the project. The presentation layer can easily be changed and this will not affect the logic or data layers.

The separation of objects into *IEntity* and *IMovingEntity* was considered a success. The use of pure interface classes allowed for the implementation of different movements for derived class objects thus preventing tight coupling. The use of inheritance allowed for polymorphism to be leveraged when updating the movement of all moving entities stored in a vector.

The use of the C++17 standard allowed for the use of smart pointers to be leveraged for automatic memory management, lambda functions and auto keyword for automatic type deduction and to ensure variables are initialised when created. Copying of objects was minimised by passing them as a constant reference to functions, this improved processing time.

A object orientated solution was provided that makes use of a modular design. This allows for the code to be maintainable and scalable.

The use of Spatial Hashing and the Separating Axis Theorem as the collision detection algorithm, allowed for efficient collisions checks which did not impact the game speed greatly. The only disadvantage with the Spatial Hash is that it has to be rebuilt with every frame. The SAT algorithm is limited due the fact it was implemented to work only with Rectangles, thus adding objects with more complex shapes may result in the collision detection failing.

Despite attempts being made to make the game run at the same speed, it was observed that on faster hardware some lag was present as the Centipede segments moved on the screen. This is an issue that needs to be looked at further.

The use of vectors to contain game objects is not efficient during deletion operations. This could be solved by using a list which allows for efficient insertion and deletion. Search algorithms can then be used to look for objects, which could potentially improve the game's performance.

The inheritance hierarchy is flawed due to the fact that the functions *poison()* and *isPoisoned()* are inherited by derived class objects when only Mushrooms and Centipedes can be poisoned. This could be solved by defining and implementing poison functionality for the other game objects.

### *8.2 Testing*

A test case is created for majority of the classes within the game code with various levels of testing being performed on the class depending on the scale of its functionality. The main aspects of the entities functionality is the movement, which was tested thoroughly in each instance. This is because the the functionality varied for several instances. When testing that an entity can not exceed the screen boundaries, for instance, a test is to be performed on all four boundaries to ensure the functionality was successful. Getters and Setters found within each of the classes were also not given a test but were instead tested implicitly within the other test cases in order to obtain an integer, double, float or boolean values for comparison. The unit tests manage to successfully isolate objects and test them separately. The test cases together indicate that the derived objects can be able to manage and control themselves without external help, this means the objects correctly adhere to the encapsulation principle. Since all the tests are automated, the user input is not tested.

*IEntity* and *IMovingEntity* are conceptual classes as they contain no implemented functionality. These classes are not given their own test case but are instead tested in various other classes where their functionality has been instantiated. The lack of any Presentation Layer testing being present was due to the interface class only containing SFML functionality. By retesting each of the SFML functions the DRY principle is violated, as an assumption can be made that testing has been performed previously.

The *Logic* class is also not tested due to its core functionality being implemented in private member functions. The majority of the functionality implemented within the Logic class had been tested in the various classes it calls.

### *8.3 Type Deductions*

The implemented code suffers from type deductions which are used to make code decisions. This is evident in the *CollisionHandler* class which uses type deduction to access functionality only implemented in *CentipedeSegment* class to tell a segment to become a new head, to change a head's direction once it collides with a Mushroom, and to update segments following the head with the collision point. It would have better to implement a class that would have managed the centipede train explicitly and performed functionality such as splitting and updating segments following a head with collisions points. The *changeDirection()* function could have been added to the *IMovingEntity* class and derived classes would have added implementation to it or leave the implementation blank. Type deduction is also present in the *Presentation* class which uses it to determine the correct texture to display for each object being rendered.

### *8.4 Centipede Train Management*

When a Centipede train is generated it is pushed into two vectors containing shared pointers, one of type *IEntity*, and the other of type *IMovingEntity* used in the *GameEngine* class. From the observations made in Centipede Train Generation in Section 4.4, the vectors more specifically, the one of type *IMovingEntity*, was used to manage the Centipede Train. Functionality such as splitting the train was achieved in *CollisionHandler* class by making use of type deductions to access functionality implemented in *CentipedeSegment* class to change a segment to become a

new head. This allowed for an easier implementation but it gives unnecessary functionality to the *CollisionHandler* class which could be implemented in class that solely manages a Centipede Train. It was also observed that sometimes Centipede segments lose track of their heads after a collision with bullet or when diving into the Player's area. This bug was not fixed entirely and thus needs to be looked at further.

### *8.5 Presentation Class responsibilities*

*Presentation* concerns are isolated to the *Presentation* Class which makes it have many responsibilities, thus making it a monolithic class. This could be resolved by creating a *ScoundProcessor* class to process the sounds of the different game objects. The drawing functionality could have been pulled from the *Presention* class and given to a helper class *DisplayManager* that would be solely responsible for drawing the game objects on the screen.

## 9. CONCLUSION

The solution presented meets the basic specifications for the project's functionality and usability. It implements all minor features and adds three major features. It makes use of inheritance and polymorphism to successfully model the different game objects. This allowed for the implementation of different movements for the different types of objects. The game handles interactions between entities well, detects collisions accurately and successfully translates player input into shooting and movement. The game can be won and does not have any major graphical or logical bugs but has some flaws that could be addressed in future using the recommendations provided. Despite the present flaws the game provides an enjoyable game play experience.

## REFERENCES


[1] (2018) 2D Rotated Rectangle Collision. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem–gamedev-169

[2] M. Eitz and G. Lixu, "Hierarchical spatial hashing for real-time collision detection," pp. 61–70, June 2007.

# Appendices

## A CLASS DIAGRAM

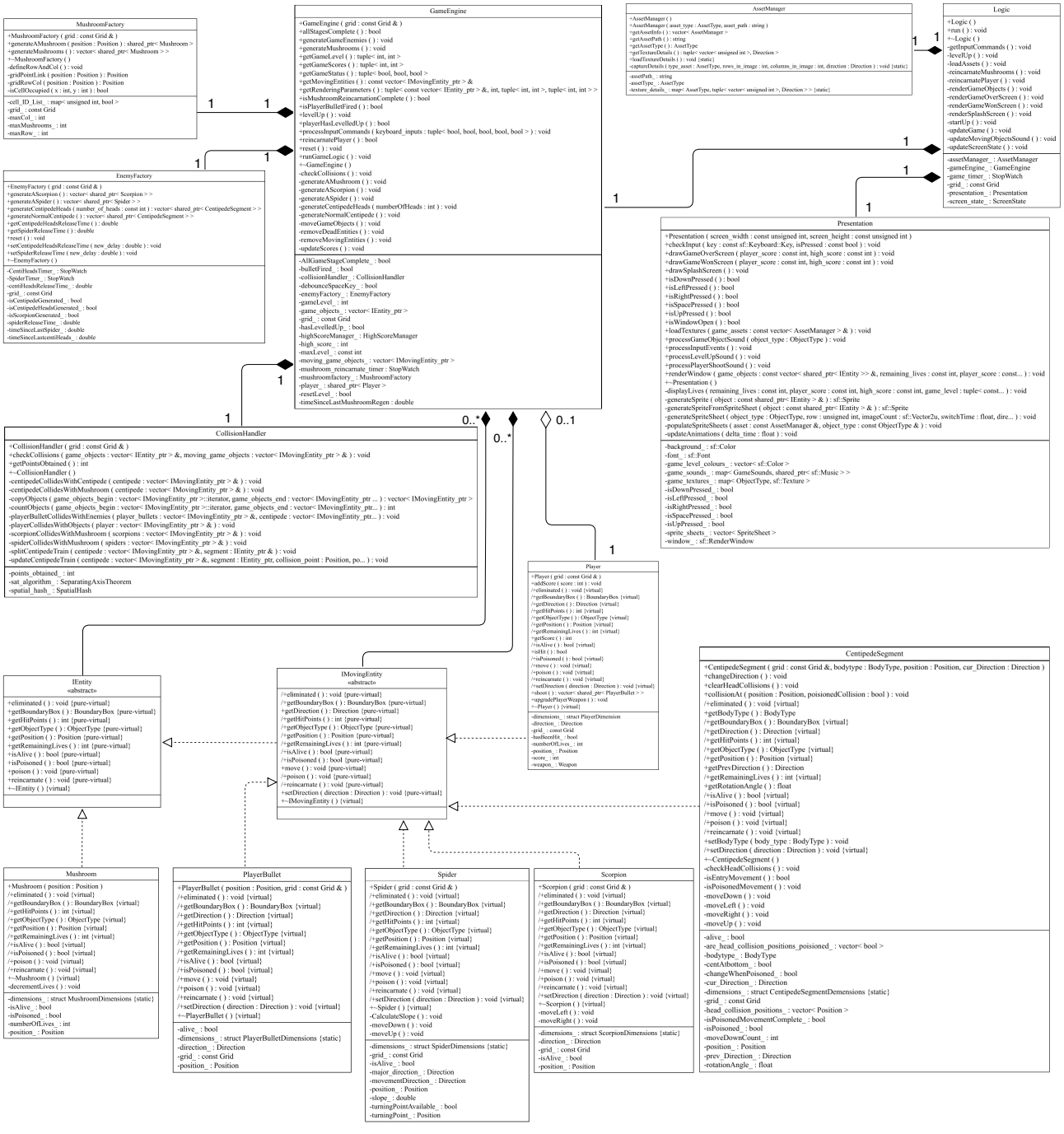


Figure 7: Class Diagram Showing Inter Class Functionality

# B SEQUENCE DIAGRAM

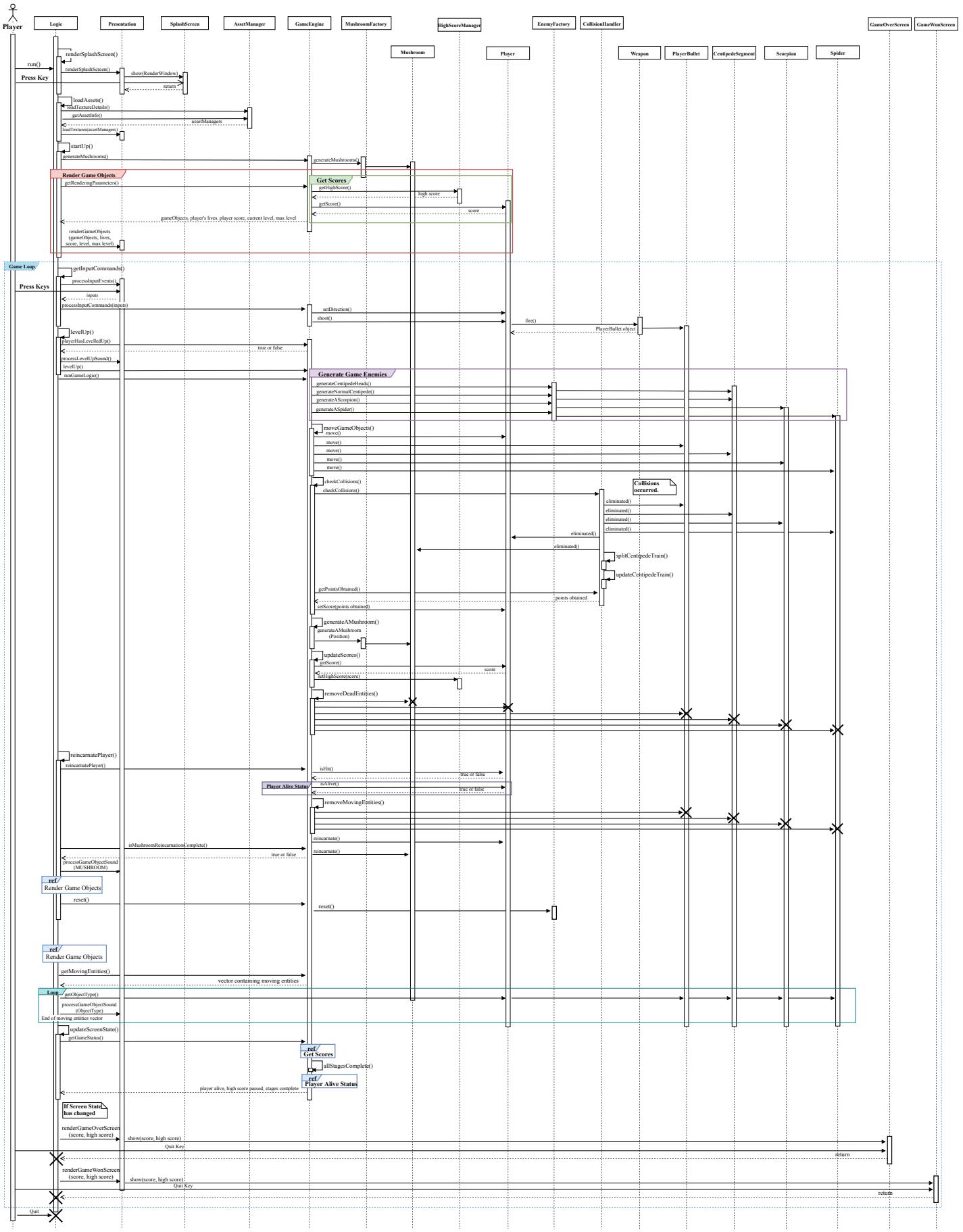


Figure 8: A Detailed Sequence Diagram

**Group Peer Assessment**

Documented below, is the group structure of how the project was delegated among members as well as member feedback in each field of assessment.

Table 5: Allocated project components for each member in group

| **Member Name** | **Assigned Project Components** |
|---|---|
| Lynch Mwaniki | Report Write-up, Front-end game design, Code Implementation & Testing, Technical Reference Manual |
| Madimetja Sethosa | Report Write-up, Back-end game design, Code Implementation & Testing. |

Table 6: Group member feedback for required fields of assessment

| | Lynch Mwaniki | Madimetja Sethosa |
|---|---|---|
| Group Leadership | 5.0 | 5.0 |
| Volume and Quality of Work Done | 5.0 | 5.0 |
| Depth of Knowledge and Understanding | 5.0 | 5.0 |
| Participation in and Contribution to Discussions | 5.0 | 5.0 |
| Other (if not mentioned above) | 5.0 | 5.0 |
| Final Average Score (Out of 10) | 5.0 | 5.0 |

Decrees of Group Member Approval:

Lynch Mwaniki

Madimetja Sethosa