C++ CENTEPEDE GAME DESIGN AND IMPLEMENTATION USING SFML FRAME-WORK

ELEN3009 SOFTWARE DEVELOPMENT II

Bophelo Ntsemi (1932442) and Masilo Sekhele (1839578)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: The report presents the design and implementation of the Centipede arcade game. The game code structure follows an Object Orientated Programming and is assembled using CODEBLOCKS C++17 IDE compiler version and SFML V2.5.0 for Presentation framework. The choice of using the "Inheritance model" for the designated game code structure is discussed, Including the roles that the 3-tier (Data, Logic, and Presentation layers) along with the classes provide in constructing the game interacting objects. The significant act of Unit Testing the functionalities of classes is evaluated, the Don't Repeat Yourself (DRY) principle is emphasized in the design and the objects lifetime management is evaluated. The overall report provides the analysis of the designated game structure and furthermore outlines the existing flaws withing the game and proposes some future recommendations and alternative solution that can be incorporated into the current design for a better version.

Keywords: DRY principle, SFML, C++17, Object oriented programming, Inheritance, Unit Testing

1. INTRODUCTION

With diligent research, Centipede arcade game is found to be one of the first and most popular video arcade games, known for its vibrant mushroom patches, aggressive centipedes, bouncing spiders, scorpions, and fleas. Most sources reveal that "it was first released in 1981 by Atari and has since become a modern classic". The objective of this work is to develop this game in the machine language C++, rebuilt it using OOP and SFML libraries to get it as like the original Centipede game as possible. Object-oriented programming has two key principles, which are Abstraction and Encapsulation which were used to model the design.

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. The report structure is divided into sections. The following section will go through some of the important aspects of the project and the centipede gameplay as it appeared in 1981. **Section 3** discusses the layers and thereafter **section 4** addresses the game implementation of classes. Hence, **section 5** describes the game's behaviour in terms of the coding structures that were implemented while **section 6** goes over the tests that were created to see if the individual classes' functionality works as expected. In **Section 7**, it is addressed that the implemented game meets the project specifications required and **Section 8** discusses the importance of maintaining code and how the bugs were fixed following software principles. Thereafter, recommendations to improve the overall design are made in **Section 9**. The final section concludes all the preceding discussions.

2. BACKGROUND

2.1 Functionality of the game

A player, bullet, centipede, mushrooms, life indicator to display how many lives the player has left, and points indicator to show the total scores that player has gained are the important parts in the Centipede game development. In the designated game, the Player moves in a constrained region on the screen. With the help of a keyboard, the player can move right, left, up, and down within the specified region above. All moving game enemies emerge from the left or right or top limits of the screen, while the Centipede train starts from the top centre of the screen. Mushrooms are scattered across the screen in random positions. The Centipede and the Player movements are both limited by mushrooms.

Scorpions travel beyond the Player's region, poisoning the Mushrooms. When centipede clash with poisoned mushrooms, they jump right into the Player's zone. Spiders consume Mushrooms in their route as they travel in a zigzag pattern through the Player's territory. Fleas moves down

the screen leaving a trail of mushroom in their path. By clashing with the Player, spiders, Fleas, and centipedes attempt to kill the player. The player can fire a projectile at the top of the screen, which will destroy game enemies and earn points.

There are five stages in the game, each containing an early Centipede train. The quantity of Centipede heads and the frequency of Spiders grow as the Player proceeds through the stages. When the player advances to the next level, their weapon is upgraded once only, and it is lost when they die. If the player dies three times, the game is over, hence, the player wins after completing all 5 stages.

2.2 Requirements and Constraints for Design

It is required for the design to follow OOP. This requires the use of class objects that communicate to perform the desired functionality. Separation of Concerns, DRY principle, shall be followed and Writing of Readable Code. Use of C++17 functionality, Doctest framework for code-testing and maintainability.

The following project constrains apply [2]:

1. Code Implementation:

- The game needs to be implemented using ANSI/ISO C++ using SFML 2.5.0 library. Earlier versions of SFML may not be used.
- The game must run on the Windows platform.

2. Graphics:

- The game must display correctly on a screen with a resolution of 1600×900 pixels. This implies, when the game Window is maximised, it may not exceed this resolution.
- OpenGL may not be used

2.3 Success Criteria

A successful design is one that meets both the functional and aesthetic criteria. This involves a well-designed and well-executed low-level implementation with an interesting and thoroughly front-end for the user to engage with. The project is also considered successful if the following conditions are met:

- The game is implemented using object-oriented programming in C++.
- Basic functionality is achieved (as outlined in [2]).
- A technical reference manual, declaration, project report and published release notes on GitHub are provided

3. CODE STRUCTURE

3.1 Domain Model

The interconnections between classes make up the logic layer. The vast bulk of them are game-related items. Individual game objects have behaviours that are distinct to them, as well as behaviours that are inherited by all or part of the game objects. These objects are thus identified as Entities. Therefore, the play of Centipede is modelled as a collection of visually displayed objects that are either stationary in a constrained space or move into, around, or out of it. Entities are formed and contained within the gaming area. The game entities and their behaviour are as listed below:

- Player- Player is generated at the bottom mid of the frame. Moves left, right, up and down in a box. Shoots bullets, collides with centipede.
- Centipede- Moves across the screen from left to right, descending towards the Player's area. When the screen bottom border is reached, it moves higher and stays in the Player's screen area. When it collides with a Mushroom, it moves down one row and switches direction. It dives straight down towards the Player's screen area when poisoned.
- Bullet-Shoots upwards from the Player’s position, it can collide with mushrooms, centipede, flea, spider and scorpion and they lose live.
- Spider- Moves randomly in a zig zag movement in the Player’s screen area.
- Scorpion- Moves horizontally across the screen outside of the Player’s screen area.
- Flea- Moves vertically, descending from top to bottom of the screen limits. It spawns mushrooms on its trail as it moves.
- Mushrooms- Stationary on the screen frame

To satisfy desired gameplay dynamics, all the above entities must be able to handle collisions with other entities.

The entityBase and entityMovementBase were established as pure interface classes to be used as basis classes for the corresponding entity groupings. entityMovement Base of all moving game objects and entityBase is the parent of all stationary objects. which these are classified by the hierarchy diagram shown in figure 1 below.

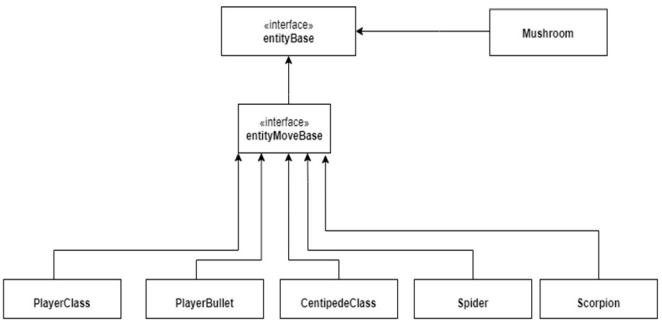


Figure 1: Inheritance hierarchy diagram of entityBase and entityMovementBase

3.2 Separation of Concerns

Layer separation has been a key design focus; therefore, the data, application logic, and presentation layers have all been carefully built to guarantee that the separation of concerns principle is followed, allowing for dependency decoupling. Layer communication is accomplished by object conversations, which ensures that only relevant information is shared with other layers while the remainder of layer data is kept private. Each layer is represented by a class, with the resourceManager class representing the data layer, LogicLayer Class representing the application logic layer, and the PresentationLayer Class representing the presentation layer.

As much as practical, the code is organized to maintain a clean separation of layers or concerns. The presentation layer, logic layer, and data layer make up the structure. The link between the layers is depicted in Figure 2. The separation is intended to make any of the layers, particularly the display and data layers, easily interchangeable. It also makes it possible for code to be readable, reusable, maintainable, and testable.

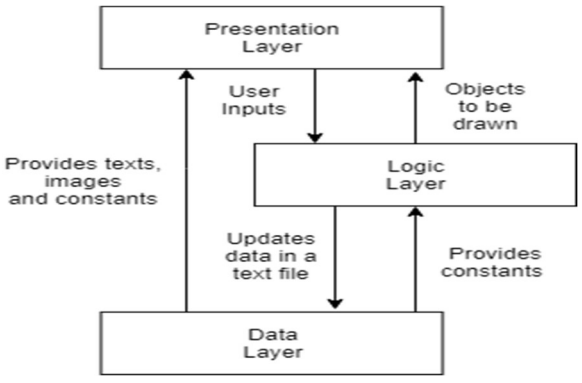


Figure 2: Layer overview

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 making, and data processing. The Data Layer is used for accessing resources such as text-files, images, and audio.

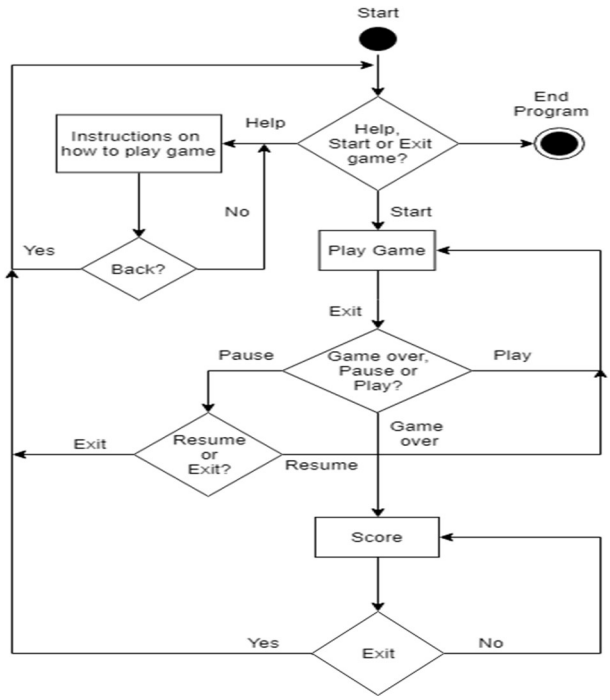


Figure 3: Overview of the presentation layer

4. GAME LAYERS AND IMPLEMENTATION

As described in Section 3, the game was divided into three layers to separate concerns. Each of the implemented classes can be classified into one of the layers. The responsibilities of each class, hierarchies within the structure and internal representation of complex member functions are discussed in detail.

4.1 Data layer

The data layer holds the task of retaining information about the assets that are used 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. The AssetManager class is given the task of retaining paths to the assets used. The loading of the high score from a text file and reading task was given to the HighScoreManager class.

4.1.1 resourceManagerClass: The required functionality of this class is implemented as a template class. It improves game performance by optimizing how the presentation layer's essential materials, such as textures, sounds, and text styles, are loaded. This is accomplished by loading all essential assets into memory before the game starts off, sparing the game from

needing to load assets later or frequently, actions that would burden the target system's performance. This resource loading procedure follows the DRY coding philosophy, which was a major consideration when creating the template class.

4.1.2 positionHandler: This class holds an x and y screen position values of the game objects. And it allows objects to create instances of their positions and Axis Aligned Boundary Box's vertices instead of defining setters and getters for each class to retrieve the respective variables.

4.1.3 ObjectsDirection 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, RIGHT and NONE. The NONE simple tells that an objects faces no specific direction *(it is in neutral*).

4.1.4 movingObjectsType enum class: A strongly typed enum class representing the type of game objects. The object types are PLAYER, PLAYER LASER BULLET, CENTIPEDE, MUSHROOM, SCORPION, SPIDER and FLEA.

4.1.5 HighScoreManager: This class reads/write the high score from/into a text file respectively. It is controlled by the Logic Layer which then provides the high score data to the Presentation Layer to be displayed on the interface.

4.1.6 Dimensions Structs: All the child classes inherited from either entityBase class or entityMovementBase class consists of a struct that holds information about an object's height, width, and movement speed. Object uses such information to construct their Boundary Box and its movement function.

4.1.7 ScreenParameters: This class holds information about the game screen width and height. This information is passed around classes and taken in by the constructor and it allows all the objects that use a game screen grid to determine their movement behaviour or their functionality.

4.2 Logic layer

4.2.1 PlayerClass: This class inherits from entityMovementBase class and represents the game's user-controlled entity. Its movement is directly controlled by keyboard interaction in the interface using SFML event polling, which is then converted to logic-based events that are processed in the presentation layer to allow the Player to move in the intended direction while tracking the distance traversed for proper interface rendering. The Player has the power to shoot a projectile in the direction of the previous movement. When the Player shoots, it tells the *Weapon* class to generate a *PlayerBullet*.

4.2.2 TimerClass: This class is critical to the game's functionality because time is utilized to quantify game activity. The movement of creatures, projectile movement, and the locking of the screen's frame rate are all examples of this. This is accomplished by measuring the amount of time that has passed per frame and using that time to scale entity movement and in-game activity speed.

4.2.3 BoundaryHandler: This class is used to make a rectangular shape. Based on an object's position and rotation angle, a rectangle shape or boundary is created for each game object per frame. Collision detection uses this border rectangle object afterwards. There are five Position class objects in this class. Four of them represent the grid's vertices, while the fifth represents the object's position, which is the boundary frame's centre.

4.2.4 MushroomManager: 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 screen 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.

4.2.5 MushroomClass: This class inherits from the entityBase class. It models a Mushroom object in the game which is a stationary object. 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. A mushroom is also spawn along the Flea's path as the Flea descends on the screen.

4.2.6 PlayerBullet: This class inherits from the entityMovementBase. It models a projectile that a Weapon can shoot. It gets created at the Player's position when it gets fired and moves upwards towards the top screen boundary, it then dies if it reaches top end of the screen or when it collides with a Mushroom, CentipedeSegment, Scorpion, Spider or Flea.

4.2.7 EnemyEntities: This class is responsible for generating Spiders, Scorpions, Fleas, a Centipede train using Centipede Segments. The Centipede Segments can be labelled as Centipede Head or Body.

The generation of Spiders, Fleas and Centipede Heads is done periodically throughout the game.

4.2.8 EntityBaseClass: This class represents the fundamental functionality of all objects in the game, including having a position, permitting collisions, and providing the entity's character data, which allows the interface to render the object. This allows the user to view a collision that has been rendered, but only after relevant logic has notified the interface via object conversations to do so. This class also provides additional logic features like as inflation and deflation, although it is not required that all derived classes use it.

4.2.9 EntityMovementBaseClass: This class is a pure interface class which inherits from entityBase class. It adds extra functionality for derived class objects to be moved.

4.2.10 ScorpionClass: This class inherits from the entityMovementBase class. It models a Scorpion. 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.

4.2.11 Spider: This class models a Spider. It inherits from the entityMovementBase class. The Spide 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 when collide with the Player, the Player loses a life. A Spider dies either when it is shot or when it reaches the screen's left or right boundaries.

4.2.12 Weapon: This class denotes a weapon capable of firing PlayerBullet objects. The PlayerBullet objects are created when the weapon's shoot function is called at a specific position. In the game, the Player uses it to shoot PlayerBullet objects. To provide a pause between the creation of bullets, it uses the timerClass. Once a weapon has been upgraded, the reload time of the Weapon object is cut in half. Resetting an upgrade is also possible with a Weapon object.

4.2.13 CollisonManger: This class oversees all entity collision detections, telling the relevant entities if a collision does occur, and then allowing the appropriate actions to take place.

4.2.14 Flea: This class models a Flea. It inherits from the entityMovementBase class. The Flea moves vertically descending from the top boundary to the bottom boundary of the screen grid. The Flea spawn mushrooms in its path as it moves down. A Flea dies either when it is shot twice or when it reaches the screen's bottom boundaries. A player will loose live if collide with the Flea.

The collision detection includes the following 13 types:

- Mushroom & Centipede
- Mushroom & Bullet
- Mushroom & Spider
- Mushroom & Player
- Mushroom & Scorpion
- Bullet & Centipede
- Bullet & Spider
- Bullet & Flea
- Bullet & Scorpion
- Bullet & Mushroom
- Player & Centipede
- Player & Spider
- Player & Flea

4.2.15 LogicLayerClass: This class oversees setting up all the game's prerequisites, creating all the game's required entities, and eventually starting the game. The major purpose of this class is to interface with the Interface for the presentation layer to depict the game update in real time. When the game reaches the required end circumstances, the Logic class ensures that the game ends either when the Player's life have been depleted, resulting in a loss, or when all Enemies have been destroyed, resulting in a win.

4.2.16 CentipedeClass: This class models a Centipede segment. It inherits from the entityMovementBase 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.

4.2.17 GameEngineClass: This class can create game objects, order them to move, check for collisions, and update the game's high score. The game's behaviour will be controlled by this class, which will be controlled by the Logic class.

4.2.18 SplitAxis: This class implements the Separating Axis Theorem (SAT) algorithm [1]. This

is used later in CollisionDetection to determine if any overlap occurs between two Boundary Box objects of the type BoundaryHandler class.

4.1.19 KeyboardInputKeys Struct: This class holds the Boolean expressions which determine if the LEFT, RIGHT, UP, DOWN and Space key have been pressed or not.

4.1.20 GameStateClass: A strongly typed enum class representing different game screens state occurring within the logical layer. Object types are WELCOME_SPLASHSCREEN, PLAY_GAME, GAMEOVERSCREEN and GAMEWONSCREEN.

4.1.21 ActivityStates: A struct class that holds the Boolean expressions about the activities occurring in the GameEngine class.

4.3 Presentation Layer

This layer has the most important features of the three layers because it oversees handling all game entities, all in-game actions, and game looping to allow for dynamic gameplay as needed. The domain classes, which control objects and their interactions, and the logic classes, which manage game activities and allow the game to execute in a specified order, are the two portions of this layer. This layer is responsible for all user input polling through SFML events and rendering all game entities.

4.3.1 SplashScreenClass: This class is responsible for displaying a welcome screen, as well as the game instructions. It draws the splash screen on the window after configuring all the settings.

4.3.3 PresentationLayer: This class oversees drawing all the game items on the screen, as well as playing the sounds of moving game objects and the sound that plays when the Player fires a bullet. It receives a vector of entityBaseClass objects from the Logic class, which is looped over. It gets the object type from the resourceManagerClass and maps it to the relevant texture, then crops the texture with a Sprite object for those with animated movement or changing states.

4.3.4 gameOverScreen: This class renders the game over screen. It will only be displayed when the Logic class has determined that the Player has lost all lives. It renders the Player's score and the game's current high score to the user.

4.3.5 gameWonScreen: This class renders the game's Winner screen. It will only be displayed if the Logic class has determined that the Player has completed all the five game levels. It displays the Player's score and the game's current high score to the user.

5. GAME FUNCTIONALITY IMPLEMENTATION and the DYNAMIC BEHAVIOUR OF THE GAME

This section outlines the game's behaviour using the code structures. The Game Loop, User Input, Mushroom generation, Enemy generation, Collision Detection and Enemy Movement is discussed in this section.

5.1 Game Loop

The overall game operation is controlled inside the active game loop found in the LogicLayer class. Referred in Figure 4 is the operation sequence diagram, it outlines the game loop process of polling for user input, creating the required entities, moving game objects, checking for collisions, removing dead entities, rendering game entities, updating game levels, and finally updating the screen state.

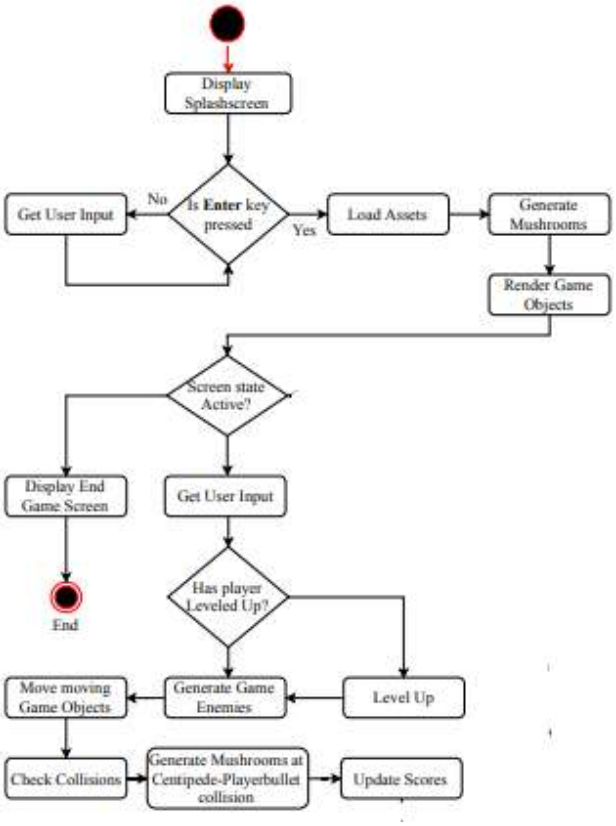


Figure 4: Logical game loop

The loop is designed and implemented in conjunction with a game timer from the Timer class that enforces the decoupling of the game speed from the processor speed to run at the same speed on different computer hardware configurations.

Referred in Appendix A Figure 6 is the class diagram of the game classes visual illustration of the object conversations. The game loop works as intended, making use of object conversations that allowed the Logic class to access only the object's public interfaces

which are required to run the respective game operations.

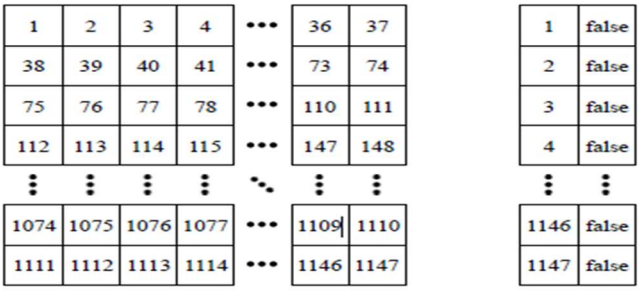
5.2 User Input

The keyboard user input gets captured through the Presentation class which make use of SFML functionalities to poll for events. The captured input then processed in Logic layer class. The timer will control the sensitivity of the user's input by enforcing the Player does not move quickly across the screen while an arrow key is pressed. The Space bar is de-bounced to eliminate the continuous creation of Player Bullet objects while holding down the key.

5.3 Mushroom Generation

Mushroom generation is performed in the Mushroom Manager class. Maximum of mushrooms can be generated. This is done by dividing the grid of size 592×640 pixels into cells of the same size as the Mushroom texture of the size of 16×16 pixels. Therefore, there are 37 columns and 40 rows. The number of rows and columns are used to compute for total number of cells. The visual diagram for the cell ID allocation is referenced in Figure 5. Each cell is paired with a bool (initially false) representing if the cell is occupied or not. The map/hash-table holds the cell pairs. Mushrooms are populated in the screen grid at random positions, The process is randomized by choosing a random column and row between 0 to 37, and 0 to 31 respectively. Rows are limited to 31 to restrict putting mushrooms in player's boundary. Rows and columns are mapped into corresponding cell ID to check if it is occupied or not. The equations below evaluate the mapping of row-column to cell ID (1), x and y cell centre positions on the screen grid (2).

To make sure that the Mushroom is created at the centre of a cell. The generation of Mushrooms at the positions of dead centipede segments and the path of the Flea make use of manipulating 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.



(a) (b)

Figure 5: (a) cells ID, (b) Cell ID mapping

$$cell\ ID = maxColums \times (row+1)+(column+1) \quad (1)$$

$$xPos = round(column \times 16+8:0)$$

$$yPos = round(row \times 16+24:0) \quad (2)$$

5.4 Enemy Generation

The generation of enemy Entities is controlled by EnemyEntities class which dictates when, where and how many enemies are generated for each level of the game. Five types of enemy objects are created: A normal Centipede (1 head and 9 body segments), Centipede heads, a Scorpion, a Flea, and a Spider. The normal Centipede and the Scorpion are generated at the beginning of each level of the game. The Spider and Flea are generated in every level but only after a specified continuously delay of time since the level started. The Centipede heads are generated from Level 2 onwards. The generation of the Spider, flea and the Centipede heads is delayed in each level. The Centipede heads are only generated once per level whereas the Spider and Flea are generated periodically. Refer to Table 1 and 2 for delays and number of heads in each level.

The Centipede train is generated making use of Centipede Segments objects and defining their body types as either a Head or Body of a strongly typed enum class. Their initial direction is randomized to either left or right. The Centipede train is pushed into the vectors that holds stationary and moving game objects. Vectors in C++ make use of contiguous memory and based on diligent observed, the Centipede train segments would always be adjacent to one another in memory even after the removal of dead segments.

Table 1: delay of periodically generated game objects

GAME OBJECTS	**DELAYS (seconds)**				
	Level 1	Level 2	Level 3	Level 4	Level 5
Centipede Heads	-	7.5	7.5	7.5	7.5
Spider	15	12	9	6	3
Flea	22.5	22.5	22.5	22.5	22.5

Table 2: 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

5.5 Collision Detection

Major game features are dependent on CollisionManager class. The density of Mushrooms objects in the game, specify the number of collisions that need to be checked. The designated method of collision is an existing method which is encouraged and had been used before by senior students in the software development field. Diligent research and validation were conducted, and it was concluded that such method will accurately detect collisions and constructs a computationally efficient algorithm. The

chosen collision detection algorithm "SAT" algorithm [1] checks for overlap between two convex polygons. The logic 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 iterates to the next one and check for overlap. The polygons are rectangles generated by the BoundaryHandler class. To prevent the checking of every game object against every other game object, a Spatial Hash was implemented in the HashGrid class [2]. A Spatial Hash carry out the task of reducing collision detections by dividing the screen into cells and keeping track of cells objects where 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 are dependent on choosing a cell size that's not too large otherwise too many collisions checks will occur. An object's position and boundary box vertices are all mapped to a cell using Equation 3, where x and y are screen positions.

$$cell\ ID = x \times (GridHeight / CellSize) + y \quad (3)$$

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 $O(n2)$ 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. Appendix A shows the objects' collision relationship.

5.6 Moving Entities Movement

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

5.6.1 Player's Movement: The Player's movement depends on the user's input which is then processed to update 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.

5.6.2 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.

5.6.3 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 at the left and right screen grid boundary, a mushroom or another CentipedeSegment. The CentipedeSegment moves up one row if it 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.

5.6.4 PlayerBullet's Movement: The PlayerBullet is created when a player shoots. It moves vertically upwards.

5.6.5 Spider's Movement: The Spider's movement is in a randomized zigzag fashion across the player movement area. A random point is generated and used to calculate the next slope. The Spider then moves towards the next point using the calculated slope. when the point is reached, a new point and slope is calculated.

5.6.6 Flea's Movement: The Flea's movement is vertically. The Flea moves move towards the bottom boundary of screen from the top a randomized zigzag 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.

6. TESTS

6.2 *TestboundaryHandler:* The boundaryHandler object was checked to see if the boundary was constructed correctly and could be rotated clockwise and anti-clockwise while maintaining its size and center position.

6.3 *TestCentipede:* The Centipede object was put to the test to see if its mobility could move up, down, left, and right. It was also checked if the centipede stays within the boundary.

6.4 *TestPlayerClass:* Test the player's remaining lives can be retrieved, the player is able to shoot and can move at left, right, up, and down direction.

6.5 *TestWeapon:* Tested to see if Weapon object could form PlayerBullet after the timerClass is called, and if could reload time, and at the provided position. A test was also conducted to see if a weapon might be modified to reduce the reload time delay.

6.6 *TestMushroom:* Tested to see if object could create mushrooms at random locations on the screen. Also tested to see if it could generate a mushroom at a

specific location.

6.7 *TestSpider:* Tested to see if it could be formed at random locations around the left and right edges of the Player's screen.

6.8 *TestScorpion:* The getters and setters were put to the test to see if the scorpion's position and orientation could be retrieved. The scorpion's movement was also examined to ensure that it was moving in the appropriate direction and if the scorpion dies if it moved past the left or right barrier.

6.9 *TestpositionHandler:* Test the getters and setters of the x and y members of the position.

6.10 *TestHashGrid*: The grid tests were run to see if a Grid object could be constructed with default values and its width and height could be accessed using its getter functions.

6.11 TestPlayerBullet: The PlayerBullet object was tested to see if it could be formed at a specific location and then move vertically from there to the top limit of the screen, where it would die if it went beyond the top boundary.

6.12 *TestGameEngine:* The game logic was checked to ensure that it started with the necessary parameters, such as starting at stage one. It was also put to the test to see if game enemies could be generated and moved successfully inside the boundaries of the game. The ability of keyboard press events to move a Player object and direct the Player object to shoot PlayerBullet objects was also tested.

6.13 TestResourceManger: To see if the correct routes were assigned to the correct game asset, a resourceManager object was used. Tests were also carried out to see if the right number of images detected in the texture of a game object were stored correctly.

6.14 TestFlea: To see if the flea can be created at random x axis position withing the zero vertical position and can only move down. The Flea is tested if dies after shot twice.

7. MAINTENCE OF GAME CODE

In the implementation of the code, error catching was made to be minor, consisting primarily of error handling for logic and runtime issues in the process of loading both assets and high score files. As a result, an unanticipated error could have a negative impact on the game, stopping the player from playing at all or crashing the code.

The code presented is modular and follows the DRY principle. If changes are made to the Inheritance Hierarchy, they to ripple down to the derived classes that implement them. This affects the maintainability of the code as multiple class's implementations which must be changed when functionality is added, removed, or changed at the top of the hierarchy. The separation of concerns in the code simplifies 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 by decoupling keyboard input events from frame rate updates.

8. CRITICAL ANALYSIS AND RECOMMENDATIONS

9.1 Critical analysis

9.1.1 Strengths - The code follows good programming principles, such as the usage of type definitions to improve code readability and member-wise initialization to avoid re-initialization of data members because the compiler initializes all data members before executing the constructors. Passing objects and arguments by reference to modify original values improves code efficiency since it prevents the compiler from calling copy constructors [2].

The code's memory management is generally managed through the usage of C++17 smart pointers. This minimizes the number of raw pointer issues, which can lead to undefined behaviour or memory leaks [3]. Composition is used to make excellent use of reusability in the programming. This results in fewer, more manageable classes and avoids the DRY principle being broken.

Most classes are largely self-contained and encapsulated to allow them to collaborate without exchanging too much information. Getters and setters were used to give relevant information to other classes, limiting access to private information to the getter functions that were allowed.

9.1.2 Weaknesses - The code makes use of vectors which can be inefficient when deleting elements. For future development, vectors should be replaced with lists. The monolithic classes LogicLayer and entityBaseClass presentation can be recognized as needing refactoring.

9.2 Object-oriented programming

9.2.1 Strengths- The concept of inheritance is used to model the roles of entities. This is done to prevent DRY violations by capturing similar behaviour in parent classes. The domain and presentation layers are clearly separated. As a result, the code is easier to test and less prone to dependency-related problems. Clean separation also means that you can use another graphics library instead of SFML without having to change the domain.

9.2.2 Weaknesses- The usage of getters and setters in entities compromises encapsulation and exposes class information to the class users.

9.3 Flaws

The game centipede has an error tendency of randomly splitting the segments without being shot. This incident is suspected to be caused by the SplitAxis class which has functions responsible for checking the overlap between two objects.

The game sound effects were correctly implemented, but however there exist a bug due to the time processing between when the audio for player shoot was played and stopped which caused the program to crack and stops running.

9.3 Recommendations

9.3.1 Game functionality

Not all game features are included in the game and thus leaves room for improvement. This includes the game having no DDT bombs. A bomb which if shot up blows up, destroying all enemies (centipedes, spiders, fleas, etc.) and mushrooms within the blast radius. The issues can be implemented for future projects to ensure a successful game functionality making use of the existing entitybase class pure functions.

9.3.2 Code structure and Design

During the development of the game progressed, functionality took precedence over coding structure, and a broader domain model emerged. As a result, having more precise conceptual classes built to fulfil a more conceptual design rather than prioritized functionality would be desirable. The usage of inheritance might be better structured to avoid redundant member functions in derived classes and allow for minimal code reuse, while also guaranteeing that derived classes do not have access to member methods that are intended for them.

Depending on the machine, the game runs at varying speeds. Implementing a frame rate independent game play could be a possible future upgrade. This ensures a seamless game experience [4]. Overall performance could be enhanced by implementing multi-threading, which allows a CPU with multiple threads to take advantage of the software by running various processes on separate CPU threads at the same time [2].

9. CONCLUSION

The design, implementation, and testing of the C++ centipede arcade game code structure have been thoroughly detailed, with a focus on the use of Object-Oriented Programming and Inheritance throughout the code. The game meets all the requirements listed to achieve an exceptional level of game play while keeping to all the limits listed in Section 2.4. The code has been arranged to reduce the number of dependencies and demonstrates a well-designed separation of concerns through proper modelling of the presentation, application, and data layers. The solution was created with the goal of adhering as closely as possible to excellent programming concepts and practices, such as the DRY principle, understandable code, and a clear separation of concerns. The game objects were thoroughly tested and found to be functional. The solution was examined critically, and it was discovered to have some problems. However, to achieve a better version of the solution, future upgrades and recommendations are presented.

Reference

[1] S. Prata. C++ Primer Plus. Pearson Education, Inc., 2012.

[1] S. Levitt. Laboratory 3 - Construction, static, and Smart Pointers. The University of the Witwatersrand, Johannesburg, 2016.

[3] Barbier, M. (2015) SFML blueprints. Page 175. United Kingdom: Packt Publishing.

[4] How-to Geek. “How to View and Improve Your Games’s Frame Per Second (FPS).”, 2018. hhttps://www.howtogeek.com/353730/how-to-view-and-improve-your-games-frame-per-second-Last accessed on 2021-10-24.

Appendix B.

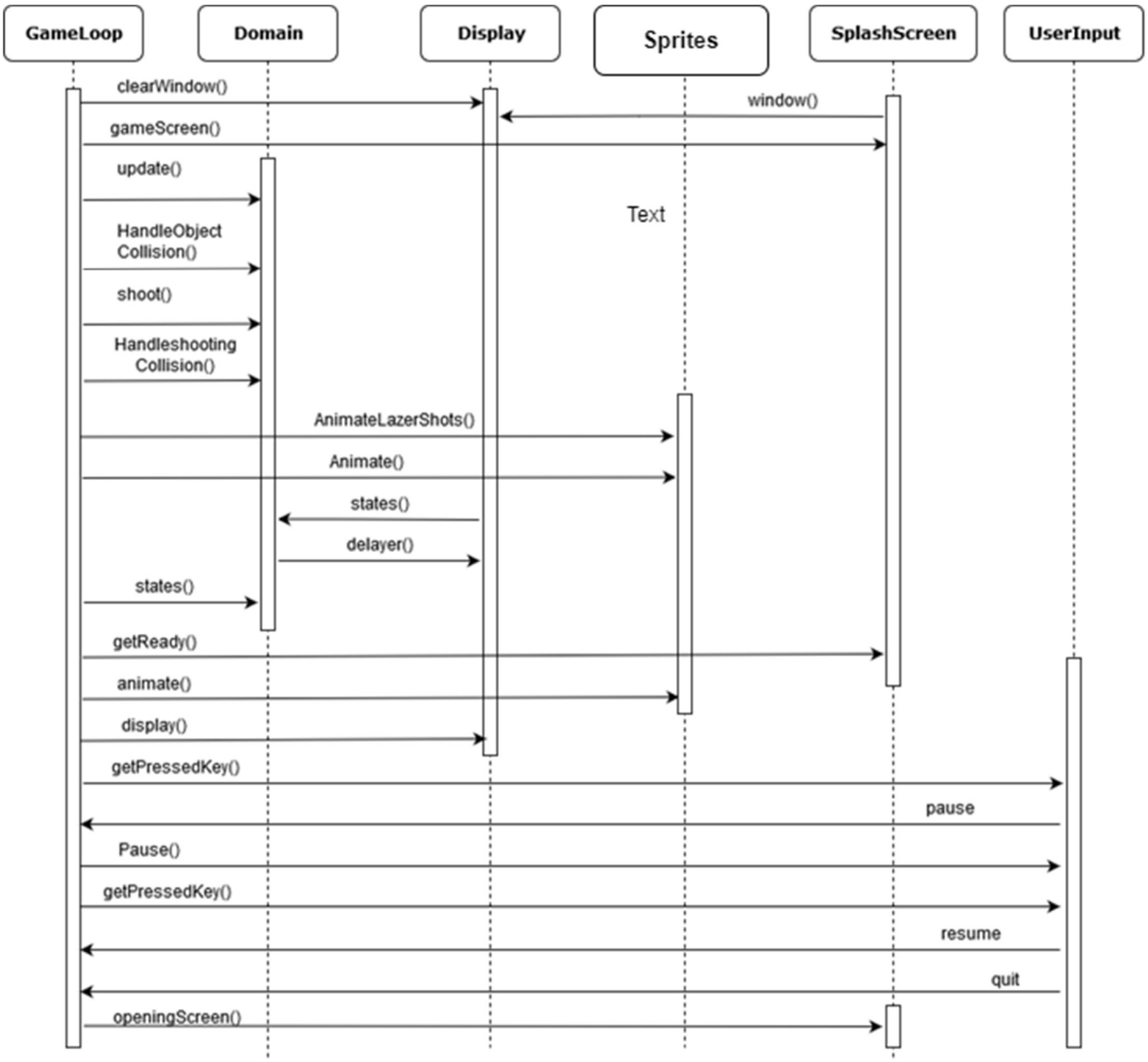


Figure 6: Sequence diagram for dynamic behaviour of major classes