Design, Implementation and Analysis of Arcade Game Coded in C++17

Tebogo Nkomondo (1124458)
Oratile Marope(1057107)
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa

***Abstract*—This document presents the work of two students in building and designing a working 2D centipede game. The document details the design methodology followed in the implementation of the game, the functionality achieved, the dependency and hierarchy of class objects, challenges encountered and future recommendations. The document also gives a broad overview of the game logic tests performed. The overall functionality achieved for the game may be summarized as composing of all basic functionality, four minor features and a single major feature. The game consists of a splash screen that informs the user of playing instructions and game over splash screen that inform the user whether they have won or lost, and displays the point score achieved.**

I. Introduction

This project is aimed at introducing students to important software design principles such as objected- oriented coding (inheritance, composition and polymorphism techniques), performing unit tests, and the importance thereof, and separation of concerns through the design and implementation of a 2D centipede arcade game rendered on a windows platform. The game is to be coded strictly in C++ and should make use of the SFML library for animation and display purposes.
This project is also meant to expose students to the advantages and use of virtual control in software collaboration projects.

II. Background

The game's interface can be divided into the game mechanics, requirements, success criteria and constraints and assumptions.

A. Game Mechanics

Centipede is a 2D shooter based game that is composed of a player, a centipede object, spiders and mushrooms. The player is located in a small area at the bottom of the screen and its main function is to fight off a collection of enemies in the form of a centipede train and periodically appearing spiders. The game terminates when the player comes into contact with any on these enemy objects.
The centipede moves through the screen in a predefined algorithm, but there exits a field of randomly placed stationary mushrooms in the game who upon collision with the centipede will reverse its direction and make it hard for the player to anticipate its movement. If the head is shot then the segment preceding the head becomes the new head. If an interior segment (not the head or tail segments) is hit then the centipede splits into two and the first segment of the rear part becomes a new head and both of the smaller centipedes make their way down the screen independently.
Mushroom objects disappear off the screen when shot by the player's bullets.

B. Requirements

The functionality required for the game is as follows:

1) Basic Functionality: The following basic functionality is required:

- In response to user input the player is able to move left and right in one line at the bottom of the screen.
- The player is able to shoot lasers that move vertically upwards towards.
- The centipede train is able to move left and right and advances down the screen each time it hits the boundary of the screen.
- When the centipede is hit by a bullet it splits into two new centipede trains at the segment that was hit and the two trains follow each other.
- The game ends when the entire centipede train has been eliminated or when the player and centipede collide.

2) Minor enhanced functionality: The following minor enhancement features can be implemented:

- There is a field of randomly placed Mushrooms that upon collision with the centipede cause the centipede to behave the same way it does at the screen boundaries.
- The player can move up and down in a small section at the bottom of the screen.
- Scores are displayed and high scores can be saved from one game to the next.
- The player has more than one life.

3) Major enhanced functionality:

- Centipede segments that are shot turn into mushrooms and behave the same way as the randonly placed mushrooms.
- The game has spiders which appear periodically and move within the player movement area.
- Player and Spider collisions result in the game ending. Spiders also destroy any mushrooms they collide with.

C. Success Criteria

A primary requirement of the project is to achieve all listed basic functionality of the game given in section II B. The game is to be coded in C++ using object oriented programming

practices that encapsulates good programming practice and obeys the DRY (Do Not Repeat Yourself) principle.
The game is to make use of SFML (Simple and Fast Multimedia Library) to present sprite entities of different game objects to assist in presenting game dynamics. Unit Testing for each of the game's objects and classes must also be completed and presented. These tests must be performed using doc-test.

D. *Constrains and Assumptions*

The game needs to be coded in ANSI/ISCO C++17 using the SFML 2.5.0 library. No other versions may be used. The game must operate and run on a windows platform with maximum screen dimensions of 1920×1080. The game must not use openGL or any other libraries or frameworks that are built on top of SFML.

III. Code Structure

A. *Design Architecture*

This section details the overall design layout of the game, how the game classes interact together and their dependencies. A description of the layer of consent that each of the classes belong to, together with the functions and responsibilities of each class is given in the *separation of concerns* section below.

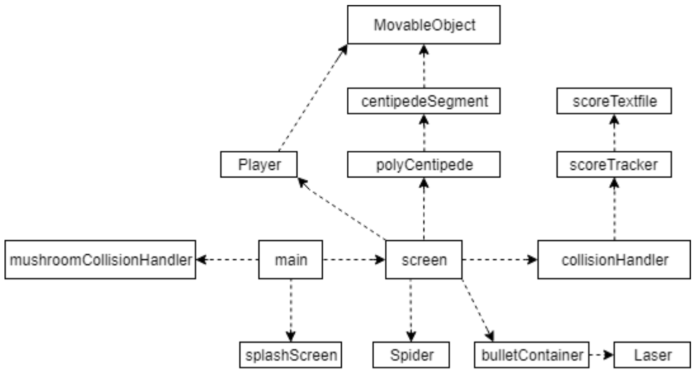

Fig. 1. Dependency diagram of classes

Figure 1 below shows a figure illustrating the dependency and interaction of the the classes. The classes in the game are linked by inheritance and composition.

The *player* and *centipedeSegment* are the only two classes that inherit from the base *movableObject* class. The rest of the classes are linked through composition.

B. *Separation of Concerns*

According to the separation of concerns design principle, the presentation, logic and data layers have to be separated. A concern is a set of information that affects the behaviour of a computer program. For this reason it was regarded essential to separate the concern of individual classes and limit their functionality to deal only with problems that concern them. These classes can then be seperated into three layers, namely the presentation, logic and data layer. The presentation layer is concerned with the Graphical User Interface and enables the user to interact with the game through the user of keyboard input. The logic layer is concerned with performing logical decisions and arithmetic calculations. The data layer is concerned with integrating resources of different types such as images with the existing game interface. The game layer separation design principle is summarized in figure below.

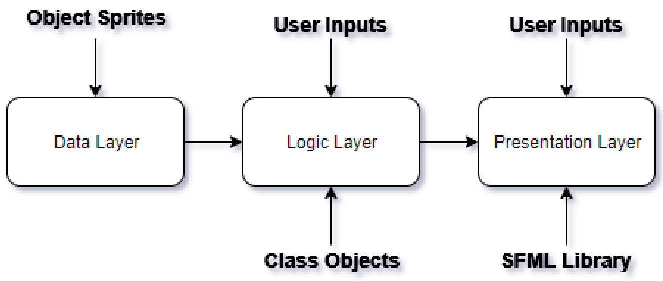

Fig. 2. Separation of layers interaction diagram

The benefits of separation of concerns are

- Debuging becomes easier because the files containing the errors are separate hence errors are also easily isolated.
- Allows for different people to easily divide work among themselves by working on different layers.
- Code is easily extensible and scalable since the code is in different layers.

IV. Implementation

The following section discusses how each implemented class obeys the separation of consent principle and lists the layer that each class forms part of.

A. *Data Layer*

1) SpriteGetter Class: This class is responsible for keeping track of all of the sprite files which are used to represent the game objects. Other classes use composition to load the sprites from this class.

2) scoreTextfile Class: This class is responsible for updating the currentScore and highScore text files which store the current score achieved by the player and the highest score ever recorded by a player. This class allows for the high score to be stored and displayed from one game session to the next.

3) PolyCentipede: a class that is used to create a vector of centipede segments to form a centipede train. The class also initializes each of the centipede's segments positions within the vector. This class forms part of the data layer because it is concerned with the storage of centipede segments into a vector.

B. *Logic Layer*

1) movableObject class: This is the base class from which moving objects The centipede and player inherit their moving abilities. A movable object is always aware of its current position (x, and y Coordinates) and the direction (up, down, left or right) that it should move in. This is what the *movableObject* class provides to all moving objects. The class hence obeys the separation of concerns principle, as it has limited responsibilities and functions that are related to moving a object on the screen.

2) centipedeSegment: This class is responsible for governing how a single centipede segment object moves on the screen. The class is limited to outlining the algorithm that a centipede segment is to follow when coming into contact with mushrooms or when it hits the boundary of the screen.

3) singleMushroom: This class is responsible for setting the coordinates of a single mushroom on the screen. The class places these mushrooms randomly on the screen.

4) MushroomCollisionHandler: : This class deals solely with collisions that involve mushrooms on the screens. The class detects what objects collides with the mushrooms and sends booleans to the to the respects object classes to communicate a message that a collision has happened. The class is part of the logic layer.

5) Player Class: This class inherits from the MovableObject class and is responsible for giving the player object its properties. The class inherits the ability to move the player up, down, left and right and additionally it restricts the player movement to only a designated player movement area at the bottom of the screen.

6) Laser Class: This class is represents a single Laser/Bullet used in the game. Once fired the bullet moves vertically upwards until it exits the screen. The bullet's initial location is always set to be the position of the player at the moment that the bullet is shot through the use of the space bar key on the keyboard.

7) BulletContainer Class: This class is responsible for allowing users to shoot more than a single bullet over time. It uses a vector to store lasers which it gets from the Laser class through composition. Once bullets are located outside of the screen they are erased from the vector, this keeps the vector size small and hence less memory is used from the vector.

8) Spider Class: This class represents the spider object in the game. The spider appears periodically over time and moves in a zigzag motion within the player movement area. The spider appears in random locations every time that it appears in the screen.

9) collisionHandler Class: This class is responsible for handling the different game outcomes that occur as a result of collisions occurring between various objects. A collision between the player and centipede or player and spider will result in game termination and a lose message will be displayed on the screen along with the score for the current game session and the highest score ever recorded.

A collision between a bullet and a centipede segment, a spider or a mushroom will result in the score being updated(incremented by 1). If all of the centipede segments in the centipede train have been destroyed(collided with bullets) then the game will terminate and display a win message along with the scores.

C. Presentation Layer

1) SplashScreen class: This class is responsible for displaying the game controls to the user at the start of the game. When the game ends this class is responsible for displaying whether displaying whether the user has won or lost. This class also displays the current score of the player and the high score.

2) ScoreTracker class: This class is responsible for keeping track of the current score of the player. Once the game has been completed this class provides the SplashScreen class with the current score of the player and the high score.

Table 1 below gives a summary of the functionality that has been achieved for each game entity:

TABLE I
GAME ENTITIES AND THE FUNCTIONALITY ACHIEVED FOR EACH ENTITY

Entity	**Achieved Functionality**
Player	• Can move up, down, left and right in a small area at bottom of the screen. • Can shoot bullets • Player dies upon collision with centipede train or spider.
Bullets	• Once fired moves vertically upwards • Initial position = position of player at moment of shooting
Centipede Train	• Moves left and right. • Advances down the screen upon collision with a mushroom or screen boundary and reverses direction. • Moves up and down in a small section at the bottom of the screen. • Upon collision with a bullet, the centipede train splits into two and the trains follow each other.
Spider	• Moves diagonally across the player movement area • Appears periodically at the bottom of the screen. • Spider dies upon collision with a bullet.
Mushroom	• Randomly placed in the screen • Mushroom disappears from the screen upon collision with a bullet.

V. Game behaviour

The flow diagram in figure 3 shows the game behaviour from the start of the game to the end.

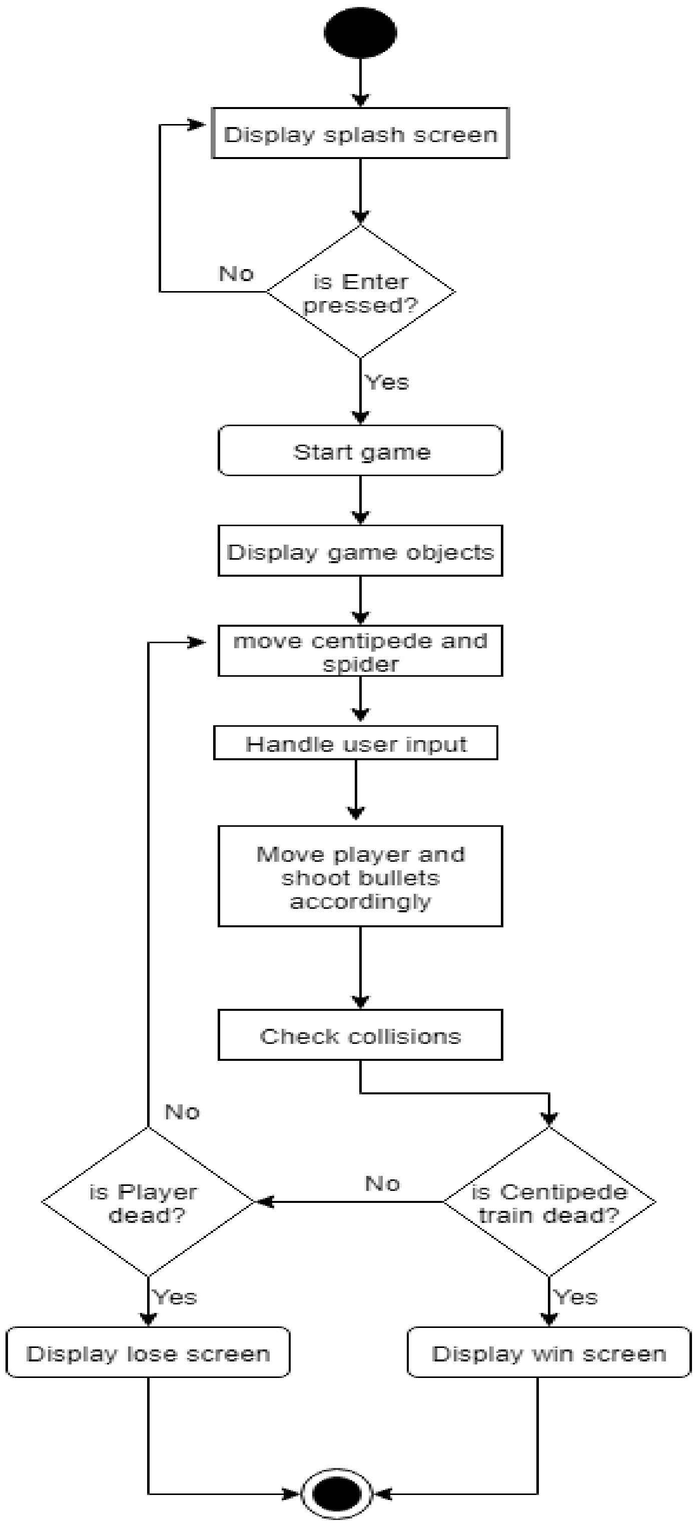

Fig. 3. Flow chart of game

The entire game runs withing a loop that is constantly checking that the window is still open, and performs polling to check whether the 'Enter' key has been pressed. If the key is not pressed, a splash screen displaying a user welcome message and the game's instructions is displayed. The logic of the game unfolds through a communication of different class objects that call each other to correctly determine the state of the game and record process (how many centipedes segments are being shot, which direction must the player move, has collisions between the game objects occurred) to correctly predict when to terminate the game. The game has two termination stages: when either the player or centipede is dead, and a score board is displayed to show how many centipedes segments and mushrooms were shot.

A. Movement

The movement of game objects is achieved through a manipulation of the object's x and y coordinates on the screen. When a object moves right, increments of a particular number are added to the objects x coordinate; and when the objects moves left, these increments are subtracted from the object's current x coordinate. The same is done for the object's y coordinate when moving up or down. These increments are what determines the objects' movement speed. The player and centipede move at the same speed in the game, while the bullet moves at twice their speed.

B. Collision Detection

Two mushrooms collision detections are performed in the game. A collision detection is performed when a centipede collides with a mushroom, or when a mushroom collides with a bullet. If a bullet collides with a mushroom, the mushroom is set to disappear from the screen. and the player's current score,to be displayed at the end of the game, is updated.

C. Object Interactions

The bullets have an initial position which is equal to the position of the player at the time that a bullet is shot. Upon collision with the bullets the mushroom, spider and centipede all die and the current score is incremented. When the player collides with a spider or centipede it dies and the splash screen displays game stats.

D. SFML dependence and use of idiomatic c++17

The player and centipede only use SFML for graphics and user input, their movement and the rest of their functions are independent of SFML functions such as $move$. For collisions the SFML function $getGlobalBounds()$ is heavily used. The only collision handling performed is for centipede-mushroom collisions. Iterators were used to traverse through the vector containers used for storing game objects. However more use of C++17 would have led the decreased code length in most instances.

E. User Inputs

The user can choose a when to start the game. Once the game is in progress the player can control the movementof the player entity. The player can also fire bullets from the player which will move upwards towards the player's enemies(i.e. the mushroom, spider and centipede). Table 2 below shows possible user input.

TABLE II
USER INPUT

Keyboard Input Key	**Resulting game action**
Enter	Start game
Left Arrow	Move player right
Right Arrow	Move player left
Up Arrow	Move player up
Down Arrow	Move player down
Space Bar	Shoot lasers

VI. UNIT-TESTING

Unit tests were performed for all game objects to assert the functionality and correct logic throughout the game. The table below shows the number of tests done and the number of failed tests for all game objects and classes.

A. Movable object testing

Only the classes that inherited their movement from this class's interface were tested since their movement functions are identical to the movement functions of this class. Detailed information about the tests which were performed on this class are therefore in the sections to follow.

B. Player Class tests

The player's movement tests were performed by placing the player object at various locations in the screen then calling the movement functions. After the movement functions were called the new location of the player is then tested by comparing it to the expected new location of the player. Tests for player movement at the player movement area boundaries were also performed the same way.

Tests were also performed to test if the proportions of the player object conformed to those which were set in the constructor of the Player class.

C. PolyCentipede tests

By performing movement tests on the poly centipede, the *centipedeSegment* and *movableObject* class functionality were also tested. This is because the *polyCentipede* class uses composition to create a vector of centipedeSegments that inherit their moving functions from the movableObject base class. Tests to assert whether the centipede train is able to move up, down. left and right were done. An additional test to test that centipede train never goes out of screen bounds was also implemented.

D. MushroomCollisionHandler tests

Test were performed on the *MushroomCollisionHandler* class to verify that all centipede segments successfully detect mushrooms, and that each centipede segment is able to move down and reverse its direction of movement after colliding with a mushroom.

E. Spider class tests

Tests were performed to test if the proportions of the spider object were the same as those that were set in the Spider class. Movement tests were also performed for the spider using the same methods used the section above for player object since both of these objects move within the same designated player movement area.

F. collisionHandler class tests

The collisionHandler class handles various collisions between game objects and the tests performed for these collisions are as follows:

1) Player-Spider collision tests: The Player and Spider were placed in locations on the screen where their bounds would intersect and tests were performed to check if this led to the player dying and game termination.

2) Player-centipede collision tests: The player and centipede's locations were also set so that their bounds intersected and tests were performed to check if this leads to the player dying and hence a game termination.

3) Bullet-Centipede collision tests: Tests were performed to check if centipede segments that collide with bullets die.

4) Bullet-Spider collision tests: Tests were performed to check if spiders that collide with bullets die.

5) Bullet-Mushroom collision tests: Tests were performed to test if mushrooms that are hit by a bullet die.

Table 3 gives a summary of tests performed on each object.

TABLE III
STATISTICS OF TESTS PERFORMED

Object tested on	**Number of tests performed**
Player Tests	7
Laser Tests	3
Spider Tests	6
Player-Spider collision Tests	8
Bullet-Spider collision Tests	4
Player-Centipede collision Tests	4
Centipede-Bullet collision Tests	3
Mushroom-Bullet collision Tets	2
PolyCentipede Test (Centipede train)	5
Mushroom-Centipede collision Tests	3
Total Tests	45

VII. Critical Analysis and Recommendations

This section presents the challenges encountered throughout the design of the game and recommended altercations that can be made to the code to increase the functionality achieved.

A. *Centipede and Mushroom Collisions*

It was noticed during the design phase of the game, that the implemented centipede and mushroom collision detection behaves differently when a centipede approaches a mushroom from the left than from the right. When approaching a mushroom from the right, a centipede train successfully moves down once and reverse its direction of motion (starts to move left), this not the same functionality achieved when the centipede train approaches a mushroom from the left. When a centipede train approaches a mushrooms from the left, the train moves down one row but continues to main its direction of motion (continue moving left). This is because the **moveCentipedeSegment** function inside the *centipedeSegment* class currently deals with all three events (when a centipede train hits the screen boundaries, when a centipede collides with a mushroom from the left, and when a centipede train collides with a mushroom from the right) that lead to a change of movement direction in the centipede train's projection.
A proposed solution to solve this is to code three separate algorithms that carter for each of the events that deal with changing the direction of motion of the centipede train.

B. *Bullet Mushroom collisions*

When shot with a bullet, mushroom objects disappear from the screen but are not removed from their actual location in memory. As a result, the centipede train still detects their location and moves down, then reverses direction upon detecting a mushroom. Numerous attempts of trying to solve this problem by moving the mushrooms out of the screen upon collision with the bullet, and deleting the mushrooms index that collided with the bullet did not work. The problem causing this is unknown, and a possible solution cannot be offered.

C. *Collision Handler*

The current implementation for collisionHandler contains the bulletCentipede and $bullet\_Mushroom\_Collisionfuntions$ which are very long functions and contain 3 nested loops. These functions also contain code that is similar in structure since they both check for collisions for objects that are stored inside of vectors. This is bad programming practise and traversing through vectors takes $O(n^2)$ time. For future improvement to reduce these functions a general function for checking if collisions have occurred between two game objects can be implemented separately from each of the functions and called inside of the functions. This will reduce the length of the functions and object they DRY principle.

The functions spiderBullet, bulletCentipede and bullet_Mushroom_Collision all contain a function call to update the currentScore and highScore this is another violation of the DRY principle. A possible future improvement of these functions is to add a function which deals with calling the functions in scoreTracker which update the currentScore and highScore. This will reduce the length of the functions and also obey the DRY principle.

D. *Moving Objects*

The movableObject class was implemented with the sole purpose of enabling objects that move to move. However currently only the player and the centipede game entities inherit their movement from the movableObject class because of poor design and use of inheritance and polymorphism. The spider and laser objects are also moving entities however they have their own move functions which is inefficient. For future improvement these classes should inherit and override their movement functions from the base movableObject class.

VIII. Conclusion

The final game only has one major feature that is the periodically appearing spiders. However the basic functionality has been achieved and various minor features were also achieved. The minor features include a score tracking system which compares current scores against scores that other game players have achieved. The product is a 2D smoothly running centipede game. The game and all of its source files occupy 26.1MB on hard disk. While running it uses 19.4MB RAM. The project was a success because most of the required functionality was met in the end however there is still room for improvement in the form of decreasing SFML dependence as this would make the game more versatile and easy to implement on various graphics libraries. The use of idiomatic C++17 can also be improved so that code becomes cleaner and more efficient.

References

[1] Prof. Stephen Levitt, Project 2018 Centipede, https://witseie.github.io/software-dev-2/assessment/elen3009-project-assessment-form.pdf, last accessed 07 October 2018