University of the Witwatersrand, Johannesburg

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

Software Development II: ELEN3009

Centipede++

Yisrael Zagnoev (1947758)

Benjamin Palay (1815593)

28th October 2021

Abstract: This report discusses and analyses the design and implementation of a 2D game called Centipede++. The game is coded using Object-Oriented Programming (OOP) in C++ and utilises the SFML library for its Graphical User Interface (GUI). The high-level design decisions, code structure, key abstractions, dynamic object behaviours and complex algorithms are explained. Basic functionality of the game is achieved and tested, along with several additions, both major and minor. Although there are many ways to improve its design, the game is implemented successfully.

1. Introduction

The objective of this project is to design, implement and test a solution for the Centipede++ game using object-oriented programming. The game is comprised of several interacting objects, including bullets, a bug player, a centipede and mushrooms. The bug can shoot bullets and destroy mushrooms as well as segments of the centipede. The centipede moves left and right, reversing direction and moving down when it hits a mushroom or reaches the edge of the screen. The game ends when either the entire centipede is destroyed, or the centipede collides with the bug. Some of the design constraints include making use solely of the C++ programming language as well using the SFML 2.5.1 library. Moreover, the game must run on the windows platform. The success criteria are comprised of writing a code which runs a keyboard-driven version of Centipede with some additional features from the game millipede. This report will provide a broad description of the design of the object-oriented code as well as a comprehensive explanation of each class. The code is then analyzed, and the restraints and shortcomings of the code are discussed. Moreover, possible improvements to enhance the code are considered. Lastly, the testing of the code is explained and examined.

2. Design

2.1 Domain Model

The domain model represents the essential entities of the system and their relationships. In figure 1 below, the main classes and their relationships are shown. For simplicity, only the types of objects the classes are composed of and their main functionality in terms of what they do for the game are shown.

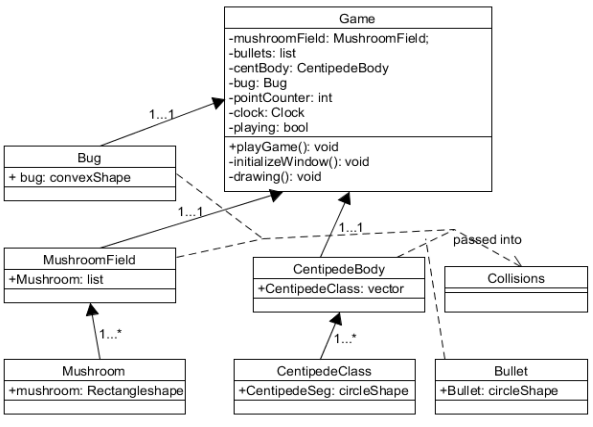

Figure 1: Domain Model diagram showing the main functions and interactions for the game.

The game was modelled to have an overarching Game class in which the window can be created and the game run from within. Inside a playGame loop, events are polled, different objects are created and drawn, and collisions are checked. The separate objects can interact within the Game class. There are two main layers, namely the presentation layer and the logic layer. The presentation layer renders objects and texts in the window, and performs all the user interface functionality. The logic layer handles the objects and their functionality. It also controls interactions between objects. There is no explicit data layer as memory does not need to be stored in that sense.

1.2.1 Classes and their Hierarchies

2.2.1 Classes for game objects

The Bug class contains code for constructing a bug as a built in SFML convexShape. Included in the class is a function which allows for the bug to move around using the arrow keys on the keyboard.

The Bullet class is used to construct a bullet as a circle shape at a given position. In the game this is the bug's position. The bullet moves in an upward direction when shot, until it reaches the top boundary of the screen, at which point it is deleted.

The CentipedeClass class creates individual centipede body parts, or segments, which are circle shapes. These segments have the status of being a head, tail or neither. This class has functions to move the centipede segments, determines the status of the different segments, and determines their direction. By knowing the size of the window, the Centipede 'move' functions know when it has reached a boundary. When the Centipede hits the side of the window while it is moving down, it reverses direction and moves down one row. If it reaches the bottom of the screen, it changes direction and moves upwards. When the Centipede hits the side of the window while it is moving up, it reverses direction and moves up one row. When it reaches the top of the screen once more, it reverses directions again and moves down. Furthermore, there are functions which treat a mushroom like the side of the window, in that the Centipede will move down (or up if moving up) and reverse direction after it hits a mushroom.

The CentipedeBody class constructs a centipede and can move the centipede. The centipede body is a vector composed of individual CentipedeClass objects which each represent a separate segment of the centipede body.

The Mushroom class – a Mushroom is constructed as a rectangle shape in this class and placed in a random area on the screen. This class also has a function which decreases the health of the mushroom when said mushroom is hit by a bullet. Moreover, it contains a function (getHealth()) which provides the number of times a mushroom can be hit before being destroyed (4) and it follows the number of times that the mushroom object it represents has been hit.

The MushroomField class – this class contains code for constructing a field of mushrooms randomly placed throughout the screen. The field is made up of mushroom objects from the Mushroom class which are placed into a list.

The Collisions class contains several functions that check for collisions and perform certain actions if objects have collided. Points are also added depending on what is shot and pointCounter is updated. Figure 2 below illustrates the general functionality of the Collisions class and how it works.

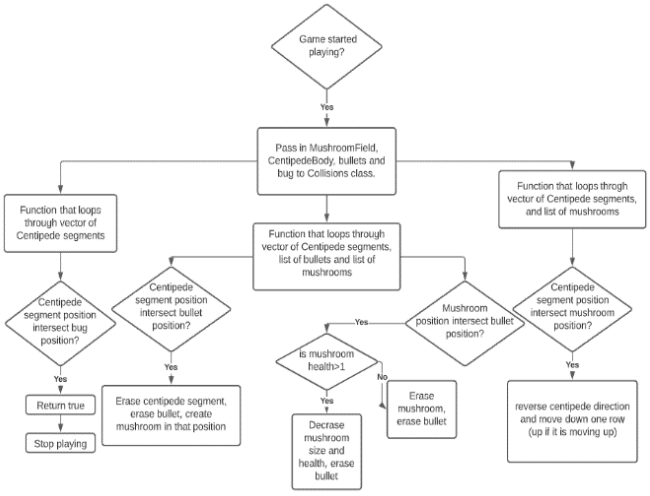

Figure 2: Flow Chart showing the functionality of the Collisions class.

Furthermore, when a Centipede segment has been destroyed, the preceding segment becomes a head.

2.2.2 The Game class, presentation and logic

The Game class is the overarching class where the presentation layer is held. It also constructs all of the different game objects. The logic and game play implementation are stored within the other classes, apart from the bullet class. For efficient shooting of one bullet at a time, the bullets had to be instantiated within the function that polls for events, and so bullets are shot when 'b' is pressed (or rather released).

The Game class constructor initializes texts, fonts, the point counter and the Boolean 'playing'. When the playGame function is called, the window is rendered with the splash screen explaining the game rules. Events are polled, and when the space bar is pressed the variable 'playing' is set to true, and the game begins. The play function, which essentially contains the logic and initializes object movement, is then called. Within this function, all the game objects are told to behave as they should, and the collisions class is instantiated. Each game object is then drawn on the screen in the 'drawing' function and then the window is displayed.

3. Discussion and Analysis

All of the basic functionality was achieved as well as three minor features and one major feature. Firstly, the basic functionality includes the following three objects: the player, bullets, and a centipede. Moreover, the movement of the player and the centipede is as specified by the basic rules of the game. Thus, the player moves left and right at the bottom of the screen and the centipede changes direction every time it hits a barrier in addition to starting to move downwards. Thirdly, the centipede separates into two if it is hit by a bullet. The segment preceding the hit segment becomes a head. Lastly, when the centipede is destroyed or the centipede collides with the player, the game ends.

There are three Minor Feature Enhancements. The first minor enhancement is a field of randomly placed mushrooms. This is implemented well as each the time the game is played, the mushrooms are laid out differently and randomly. If the centipede collides with a mushroom, the centipede moves down and changes the direction it was travelling. Secondly, the player does not only move left and right at the bottom of the screen but has an area in the bottom quarter of the screen in which it can also move up and down. The centipede also moves vertically after hitting the edge of the screen at the bottom. Lastly, there is a scoreboard in the top right section of the screen which follows the rules for scoring in the Centipede game – destroying a mushroom is one point, destroying the head of the centipede is 100 points, and destroying any other segment of the centipede body is 10 points. However, this feature is only partially implemented as the high score is not saved from one game to the next.

One Major Enhancement was used in the game. This is that when a centipede segment is shot, it turns into a mushroom. This results in all of the segments which follow said segment to turn around and move down as they hit a mushroom as soon as the shot segment is hit. After reaching the bottom of the screen, the centipede then moves vertically upwards.

However, there are several ways the game design and implementation can be improved. Sometimes there is a lag in the game, and this can cause the centipede segments to move closer together or further apart after hitting a mushroom or the screen boundary. Furthermore, if a mushroom is destroyed while a centipede is 'hitting' it, the segments that do not hit the mushroom continue moving forward in the same direction. One way these issues can be improved is by making a vector of vectors for the Centipede body. In other words, each separated part of the Centipede knows that is a meant to stay together. Then error checking can be added that repositions centipede segments that have moved too close or too far from the preceding segment.

In terms of overall design, there are many alternative ways of designing the game, many of which may be better in the long term and for adding new features. In this game design, there was no use of inheritance, and so code is repeated. One improvement could be to make a abstract base parent class called 'Entity' which contains all the repeated functionality, such as for positioning, and container construction. 'Entity' could have a child class called 'MovingEntity' which has additional

movement functions and its own children (derived classes), such as the bullets, bug and centipede. In this way there is a better hierarchy and less code repetition.

Another way the design can be improved is with the Collisions class. The Collisions class takes in entire containers of objects by reference and loops through each of them and their positions. It then modifies the containers from within the Collisions class, which is not ideal because many variables and functions in other classes have to be made public. One way this could be improved is by having functions that get the position of the objects in their respective classes. These positions could then be passed into the Collisions function, which simply returns true if the positions intersect. Thus, each class can modify itself from within instead of having the Collisions class modify them externally. This will also improve maintainability, scalability and transferability, as new objects can be added to the game without having to change the Collisions class inherently.

Finally, the presentation and logic layers could be further separated, and more classes could be made to break up the large ‘Game’ class. For example, there can be a Drawing class, a SplashScreen class and a GamePlay class. In this way, the Game class will initialize the texts, fonts and window, and the other classes will handle presentation and logic separately.

4. Testing

All game object movements and collisions were tested thoroughly. However, certain aspects of the game are not tested. These include the presentation layer as well as built in SFML functions, such as setting the fill colour of a shape or determining that the outline thickness was set as expected. These were not tested as it is assumed that the SFML library has been used and tested thoroughly before its release. Furthermore, user inputs were not tested as the testing needed to be automated, without input. Therefore, the Game class itself was not tested.

5. Conclusion

A game of Centipede++ was coded successfully using the C++ programming language whilst making use of the SFML library. In addition to the basic functionality of the game being achieved, enhancements based on the game Millipede were added to the game which greatly improved the functionality of the code. These include Major Feature 1, Minor Feature 1, Minor Feature 3, partial implementation of Minor Feature 4. Movement and collision testing for all game objects was implemented. However, there are ways in which the program can be made more efficient and reliable which were discussed in the report above.