Designing, Implementing and Analyzing an Arcade Game Coded in C++

ELEN3009A - Software Development II - 2021
Robin Jonker (1827572) & Tristan Lilford (1843691)

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

Abstract: This report is a breakdown of the design, implementation, and analysis of Centipede: Dragon Ball Z Edition, a 2D arcade game that is based on Centipede with added functionality from Millipede. The design was implemented with a focus on separate concerns, also known as layering, with a clear separation of the logic layer and presentation layer, using object-oriented programming, inheritance with polymorphism, and composition. The inheritance was implemented using two interface classes at the top of the hierarchy chain controlling all game objects. The game is implemented in C++17 with Simple and Fast Multimedia Library (SFML) used for presenting the game. The project meets all basic functionality, contains all minor feature enhancements along with extra minor features added, and 4 major feature enhancements. The implementation of the game does feature some flaws such as inefficient collision detection and violations to the DRY (don't repeat yourself) principle. Overall, Centipede: Dragon Ball Z Edition is an enjoyable game, with exciting graphical and gameplay additions, that follows many good coding practices such as an object-oriented design, encapsulation of data, separation of layers resulting in structured code that is easy to maintain and expand on functionality.

1. INTRODUCTION

This report documents the design, implementation and analysis of an object-oriented software program coded in C++17 with SFML. The program is a game based on "Centipede" with added features from "Millipede" and advanced graphics in a "Dragon Ball Z" theme. The sections of this report look at background information, the structure of the code, the implementation of classes, the control of the code, the unit testing, analysis, and a conclusion.

2. BACKGROUND INFORMATION

The project is initially described by the game mechanics, the feature and project requirements, and constraints.

2.1 Game Mechanics

Centipede: Dragon Ball Z Edition is a game rendered in a window with resolutions of 800 x 600 pixels. The game has a theme from the popular anime show "Dragon Ball Z". The main framework of the game is that the player is tasked with shooting the moving centipede that is progressing towards the player. If the player destroys the whole centipede, then the player wins. If the centipede hits the player, then the player loses. All features and functionality will be broken down below.

2.2 Requirements

The following is considered basic functionality:

- Game Objects Present: Player, Centipede, and Shooting.
- Player moving left and right. Centipede moving left and right and moving a row down with edge of screen collision.

- Centipede splitting in two on collision with bullet that was shot. Preceding segment of centipede becomes the new head forming a centipede “train”.
- Game ends with player and centipede collision or entire centipede train elimination.

The following is considered minor feature enhancements:

- Randomly placed mushroom field. If the head of the centipede train collides with a mushroom, then it moves down one row and switches direction. The rest of the train follows suit.
- Graphics are good (not composed of simple shapes such as rectangles and triangles).
- The player is also able to move up and down in a small area at the bottom of the screen. The centipede train reverses direction vertically and moves upwards if it hits the edge of the screen on the bottom row.
- Scoring system and display. High scores are saved from one game to the next and can be viewed.
- The player has more than one life and the remaining lives are depicted on the screen.

The following is considered major feature enhancements:

- Centipede segments which are shot turn into mushrooms. A consequence of this is that when an interior segment is shot, the new centipede which is formed from the rear piece will immediately collide with the mushroom resulting from the shot segment and reverse direction. Mushrooms can be eliminated by four of the player’s shots.
- The game has spiders. These appear periodically and move in a random zigzag/vertical fashion across the player movement area. They occasionally eat some of the mushrooms. If they hit the player, then the player loses a life. They are eliminated if shot by the player.
- The game has scorpions. These never appear in the player movement area. Scorpions move horizontally across the screen and poison every mushroom they touch (this must be shown graphically). A centipede which bumps into a poisoned mushroom dives straight down towards the player movement area, and then it returns to normal behaviour upon reaching it.
- The game has fleas. These drop vertically and disappear upon touching the bottom of the screen, occasionally leaving a trail of mushrooms in their path when only a few mushrooms are in the player movement area.
- The game has DDT bombs. This feature is adopted from Millipede. These are stationary and appear at random times and in random positions. A maximum of four bombs may be on the screen at any one time. A bomb is blown up if shot, destroying all enemies (centipedes, spiders, etc.) and mushrooms within the blast radius.

Unit gaming testing is required using the doctest framework. This includes testing of object movement and collisions of game objects. Game logic should also be tested.

2.3 Project Constraints

The game must be coded in ANSI/ISO C++ using SFML 2.5.1. The game must run on the Windows platform. The game window must not exceed 1600 x 900 pixels. OpenGL may not be used. No addition libraries are allowed on top of SFML.

3. CODE STRUCTURE

A breakdown of the different types of setups and structures of code implemented in the project.

3.1 Architecture

The focus with the development of the game is the use of object-orientated programming. An object-orientated approach involves the use of inheritance in structuring the code. There are different layers of code each responsible for different tasks.

The *Main* class controls and sees the *Game* class. The *Game* class controls and sees all game objects. All the different game objects are independent of each other and is solely reliant on the controlling *Game* class. There are resource classes involved in concerns external to the game such as loading of files. A flow chart to breakdown the flow and effect of each class is shown in section 5.

Having this structure allows the *Game* class implementation to alter and effect how game objects behave solely from its control with the game objects acting as a body that requires information from the brain. All game object entities are similar in their construction requiring specifications from the *Game* class such as their size, position and speed for the objects that move. This architecture allowed the feature enhancement of being able to select varying degrees of game difficulty as these specifications are not built into each game object but rather controlled from within the *Game* class. This structure of coding with inheritance allows for future changes and improvements to occur with ease.

3.2 Separation of Concerns

This design principle involves keeping different layers of the program separate. The **Start** layer is responsible for starting the game. The **Control** layer is responsible for all user-inputs and controlling the flow of the program. The **Object** layer is responsible for each individual objects' creation and their individual functions. The **Resource** layer is responsible for allowing the program to access files such as images and sounds.

The reasons for structuring the program in such a way is:

- Allows for direct control of each game session.
- Allows for a centered logic and control class which allows changes in game operation to be very easy.
- Allows for independent creation of objects.

- Allows for easier structure and control of desired resources.

  This all allows for the program to be very interchangeable, easily maintainable, and scalable.

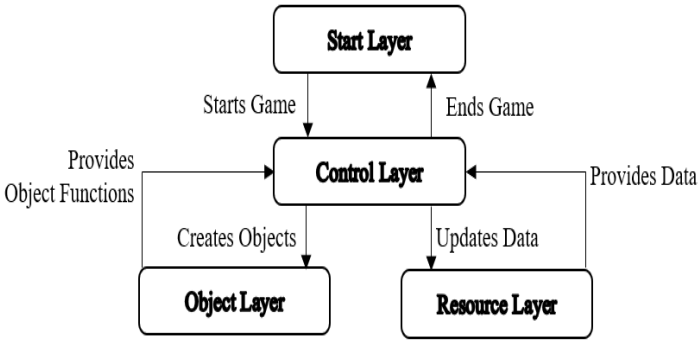


Figure 1: Separation of Concerns

4. IMPLEMENTATION OF CLASSES

The code is structured into four different layers. Each class can be classified into one of these layers.

4.1 Start Layer

- *Main* class: Its current role is the creation of the *Game* object and for initializing the game to start at the *Menu* function within *Game*. The reason for *Main*'s setup as it is, is largely to allow for future upgrades.
    - Having individual control of each game session will allow for multiple game session creations and control, enabling multiplayer, as an enhancement in the future.

4.2 Control Layer

- *Game* class: It acts as the brain of the whole game. It oversees the flow of the game and for controlling all the game objects. In section 5 a flow chart showing the logic of the *Game* class functions are shown. It handles the presentation of the different *Menu*'s at their required times and the presentation of the different game objects. It handles all the logic behind calling different functions from the different game objects, and in such a way the collision as well that enables the game to play.
    - Further improvements in this layer that will improve maintainability in the future is the separation of the *Game* class into separate classes for presentation, flow, and collision instead of currently having the functions within one class.

4.3 Object Layer

- *Player* class: Setup of the *Player* game object. A constructor requiring game dimensions, size, and speed from the *Game* object to create the player, which is an SFML rectangle.

  All player movement functions are created within this class. Functions such as setting the number of lives the player has remaining and retrieving the current player position are present too.

  As we have multiple sprites depending on the action of the player the size of the hitbox of the player also alters (an example is when the player moves upwards a different sprite is

present and therefore the size of the rectangle will be different too).

- *Centipede* class & *CentipedeSegments* class*:* Setup of each individual object that combines with other *CentipedeSegments* to create the whole *Centipede* "train".
  Similarly, to the *Player* object each segment requires game dimensions, size, position, and speed. The logic behind the classification of a segment being a head or a body segment of the Centipede is present.
  All segment movement functions are created and retrieving each segments position is present to allow for collisions.
  A feature that increases difficulty that keeps the *Centipede* within the player area at the bottom of the screen is present.

- *Shoot* class: Setup of the *Shoot* object. Like the other objects, requiring a position, size, and speed from the *Game* class upon creation of the SFML rectangle.
  Movement and position retrieval functions are also present.

- *Mushroom* class & *MushroomField* class: Setup of each individual *Mushroom* within a *MushroomField*. A *Mushroom* of specified size and position is created as a SFML rectangle upon construction when called from *Game*. Setting and getting the health of the *Mushroom* is functions present too as after 4 shots the *Mushroom* gets destroyed.
  The *MushroomField* is an object constructed with game dimensions, size and spawn chance required from the *Game* class. The spawn chance is used in a function when setting the field whereby a random number is modulated by the chance to randomly set the number of mushrooms in the field and create them.

- *Spider* class: Setup of the *Spider* object that requires spawn position, size, and speed when construction is called from the *Game* class to create the SFML rectangle. Similarly, to the other game objects, movement functions – particularly diagonal movement – and positional retrieval functions are present.

- *Flea* class: Setup of the *Flea* object that requires spawn position, size, and speed when construction is called from the *Game* class to create the SFML rectangle. A movement function and positional retrieval function is present too.

- *DDT* class & *DDTField* class: Setup of a particular *DDT* bomb placed randomly within a *DDTField*. Requiring the size and position from the Game class upon construction. Functions to retrieve information, destroy, and set if the bomb is exploding is present within the *DDT* class.
  The *DDTField* is a combination of functions controlling the deletion and resetting of the field of bombs. The random spawning and

setting of position of bombs within the field is also present.

4.4 Resource Layer

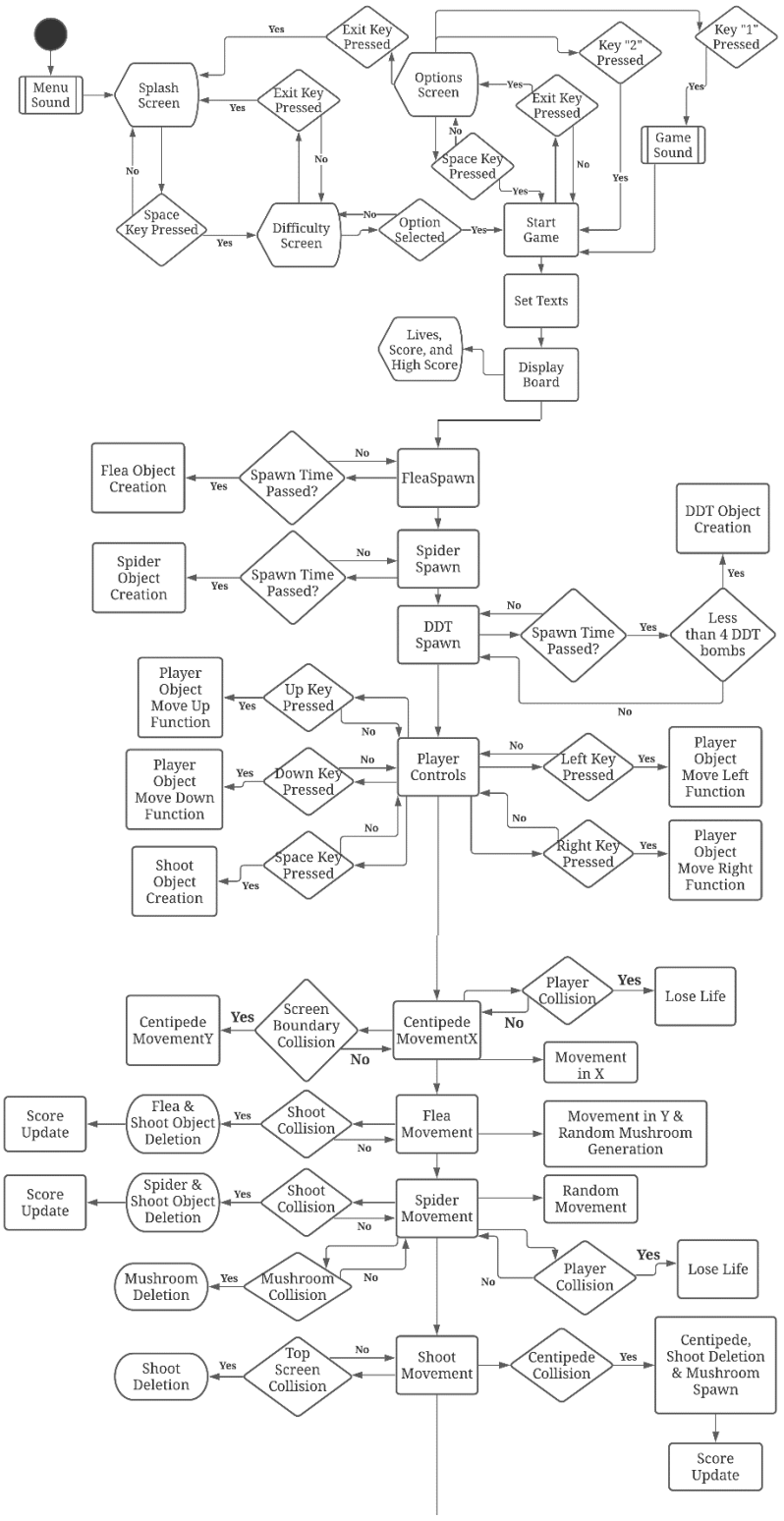
- *GameTextures* class: Upon creation the class loads in all the required images of the different sprites and creates the SFML sprites for each object. Functions the enable the *Game* class to access and mutate the current sprite selections within the *GameTextures* is present.

- *GameSounds* class: Upon creation the class loads in all the required sounds of the different game effects and creates the SFML sounds for each effect. Functions the enable the *Game* class to access the current sounds within the *GameSounds* is present. Not all the game effect sounds are in use within the *Game* class, however, the framework for everything is in place for easy enhancements in the future.

5. CONTROL OF CODE

A breakdown of the flow of the code with reference to the specific way it was implemented.

5.1 *Game* class flow. When the program is run initially the *Main* class creates a game object and then calls for the *Menu* function within the *Game* class to be run. The program runs in the *Start Game* loop, once it enters that point, until the program is closed, or the user wins/loses.



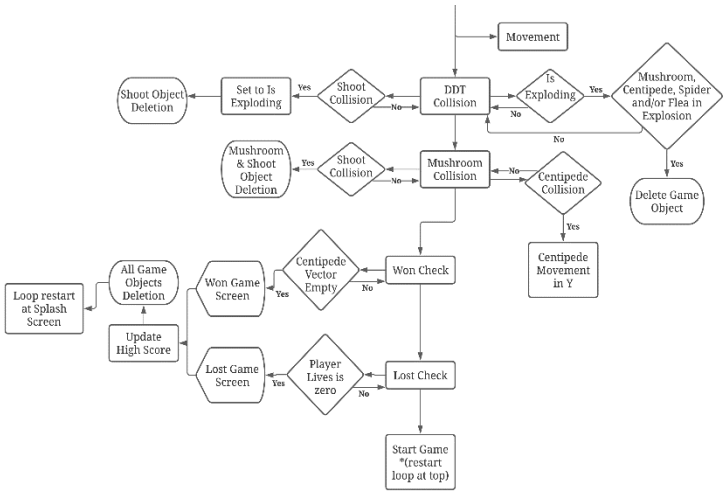


Figure 2 Flow Chart showing logic of Game loop

5.2 Features

The *Game* class is in control of the flow of the program and the using of these features. All the features present in the game will be broken down below.

5.2.1 Basic

All basic features that are required, as stated in section 2, has been met.

- Game objects, Player/Centipede movement, Centipede & Shoot collision, Game ending.

5.2.2 Minor

All minor features that are required, as stated in section 2, has been met.

- Mushroom Field object, Quality Graphics, Player/Centipede additional movement, Scoring/high score system, Player lives.

Additional minor features have also been added.

- Implemented the ability to select which difficulty you want to play the game at, adds a broader appeal to different gamer skill levels.
- Implemented a reload time to the shooting mechanic, improves the realism and difficulty of the game.
- Implemented the ability to gain extra lives if scoring thresholds are reached, adds an interesting dynamic to the gameplay.

5.2.3 Major

All major features that are required, as stated in section 2, has been met except for the scorpion game object feature.

- Centipede & Shoot collision results in mushroom spawning. Spider object. Flea object. DDT bombs object.

6. TESTING

The testing of the game was done using the doctest framework. A wide array of testing was done such as spawn testing, movement testing, event testing, and collision testing.

- Spawn Tests
    - Flea, Spider, and Mushroom Spawning
- Movement Tests
    - Player, Shoot, Centipede, Spider, and Flea movement tested in all directions
- Event Tests
    - Testing if the spider randomly eats a mushroom upon the mushroom and spider colliding.

    - Testing if flea randomly spawns mushrooms while the flea moves.
- Collision Testing
    - Shoot collision with Mushroom, Centipede, Flea, and Spider is tested.
    - Centipede and Player collision with screen boundary tested.
    - Centipede colliding with a Mushroom is tested.
    - DDT bomb explosion
    - Player collision with Centipede and Spider.

36 Tests were done in total. Many tests cover occurrences that are of very similar nature, thus, to not violate the DRY principle, not every single possible test was checked for. Spawning, Movement, Special Events, and Collisions were all tested for to a large extent. Testing is a necessity as it allows one to determine that the code functionality is exact and accurate as predicted. This will ensure that when such functionality is applied to the operation program it performs in the correct manner, therefore, reducing the possibility of bugs and operational errors drastically.

7. ANALYSIS

A breakdown looking at successful aspects of the project and flaws currently present that can be fixed in the future.

7.1 Successes

A major success in this project was the ability to have met all basic and minor features, added our own additional features, and had 4 major feature enhancements.

The structure of our code to create all our game objects as independent classes that gets inherited by the controlling *Game* class. This allows for easy game debugging and easy altering of how the game plays as one class manages the rest instead of having fixed mechanics within each game object.

The addition of a resource layer where all files get loaded into improves the ability to apply sprites and backgrounds with ease. Currently the game is in a “Dragon Ball Z” theme however due to the way we set it up, any theme can be easily applied by altering only a few lines within one class.

7.2 Flaws

The implementation of the *Game* class is large. In the future the Control layer can be separated into a Logic and GUI layer. The *Game* class can also be reduced by adding an additional collision class within the Logic layer.

The repetitive uses of some functions such as the move function within the different game object classes violates the DRY principle. These functions are largely the same and can be solved by using an abstract base class that implements the function for those game object classes.

The high scoring system can be improved upon. If a final release is made and the game gets played on a single device, then storing the high score only within a single game session should be altered to permanent storage by loading in a text file allowing the high score to be saved from one game session to the next.

8. CONCLUSION

This report has documented the Designing, Implementing, and Analyzing of an Arcade Game, Centipede: Dragon Ball Z Edition, coded in C++ using SFML.

The design was focused on an object-orientated programming approach, proper use of inheritance, encapsulation of data and polymorphism.

The implementation of the code had great emphasis on good programming practices and having easy to follow code.

The functionality of the game was shown to be worthy of the excellent criteria by the vast features going above and beyond the required feature enhancements needed.

Vast testing on most mechanics of the game was done including tests for all object movements and game collisions.

All requirements and constraints were adhered to.

A game being flawless is largely objective and we have acknowledged flawed areas in our game with solutions to these flaws that can be implemented in the future.

Overall, Centipede: Dragon Ball Z Edition is an enjoyable game that we are proud of.

9. REFERENCES

- endtoend.ai. (2021). Atari Centipede Environment. [online] Available at: https://www.endtoend.ai/envs/gym/atari/centipede/ [Accessed 4 Nov. 2021].
- GeeksforGeeks. (2019). SFML Graphics Library | Quick Tutorial - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/sfml-graphics-library-quick-tutorial/ [Accessed 4 Nov. 2021].
- Tutorialspoint.com. (2021). C++ Inheritance. [online] Available at: https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm [Accessed 4 Nov. 2021].
- SeekPNG.com. (2019). Majin Boo Png - Majin Buu Png PNG Image | Transparent PNG Free Download on SeekPNG. [online] Available at: https://www.seekpng.com/ipng/u2w7e6w7w7e6i1o0_majin-boo-png-majin-buu-png/ [Accessed 4 Nov. 2021].
- Spacey, J. (2016). What is Separation Of Concerns? [online] Simplicable. Available at: https://simplicable.com/new/separation-of-concerns [Accessed 4 Nov. 2021].
- exscape (2009). How do I apply the DRY principle to iterators in C++? (iterator, const_iterator, reverse_iterator, const_reverse_iterator). [online] Stack Overflow. Available at:

https://stackoverflow.com/questions/1776641/how-do-i-apply-the-dry-principle-to-iterators-in-c-iterator-const-iterator [Accessed 4 Nov. 2021].

- Sfml-dev.org. (2021). sf::Image Class Reference (SFML / Learn / 2.5.1 Documentation). [online] Available at: https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Image.php [Accessed 4 Nov. 2021].

- EI (2020). Doxygen Tutorial: Getting Started Using Doxygen on Windows! [online] Embedded Inventor. Available at: https://embeddedinventor.com/doxygen-tutorial-getting-started-using-doxygen-on-windows/ [Accessed 4 Nov. 2021].