DESIGN AND IMPLEMENTATION OF A 2D CENTIPEDE ARCADE GAME IN C++

Boikanyo Radiokana (1386807), Elias Sepuru (1427726)

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

Abstract: The document presents the solution to an Object Oriented Design of a 2D arcade game, *Centipede Revived!* written in `C++17`. The SFLM version 2.5.0 is used for presentation while the doctest framework is used for testing purposes. The solution meets all the basic functionality, minor features as well as two major feature requirements. The code structure makes use of inheritance and polymorphism in attempt to providing a solution that has a clean separation of layers as much as possible. However the solution has flaws due to inter collisions between centipedes. A high level description of the domain model, code structure and dynamic collaborations is presented. Recommendations for improving the solution are also provided.

Key words: Object Oriented Design, Separation of Concerns, Unit tests(Doctest FrameWork), SFML

1. INTRODUCTION

This paper presents the design, implementation, testing and analysis of a C++ Object Oriented Programming (OOP) solution to a 2D arcade game. The solution is based on the game Centipede, released by Atari in 1981. OOP forms part of the critical aspects of the solution to ensure that other programmers are able to modify the solution whilst all class invariants are preserved and the interface is maintained. For graphics and user interface, SFML version 2.5.0 is used. The correctness of the solution is verified through unit testing with the doctest framework. The game is designed to run on a PC with a windows OS.

The following sections provide a high level description of how the domain is modelled, an overview of the code structure as well as the description of key abstractions. The complexity of the dynamic behaviour and collaboration of objects is also discussed.

2. BACKGROUND

The solution to the 2D arcade game is *Centipede Revived!* which is based on the original Atari Centipede game. The word *Revived* highlights the use of improved graphics and user interface. The game is coded in C++17 and has a screen resolution of 540×640 pixels. The game has six main entities; the spaceship, lazershots, centipede, spider, scorpion and mushrooms. The aim of this game is for the player to destroy enemy entities(centipede,spider or scorpion) within a field of mushrooms. The player controls the spaceship and is given three lives at the beginning of the game. A life is lost if the spaceship collides with one of the enemy entities. The player advances to the next level if the centipede is destroyed. The game has infinitely many levels with increasing difficulty. For every enemy that is shot including a mushroom, points a rewarded.

2.1 Requirements

- Solution must be an object oriented design.
- A unit test using doctest framework must be implemented to verify the correctness of the game logic.
- A technical reference manual that is automatically generated.
- Clean separation of the presentation, data and logic layer.
- Basic functionality [1]
- A splashscreen with playing instructions.
- SFML version 2.5.0 must be used.

2.2 Constraints

- The screen resolution may not exceed 1920×1080 pixels.
- OpenGL may not be used.
- No other frameworks or libraries built on SFML may be used.

2.3 Success Criteria

The project is deemed successful provided that it adheres to all the specified requirements and constraints listed above as well as the use of good coding principles and practices. These include a clean separation of concerns , information hiding and not violating the DRY principle.

3. DOMAIN MODEL

A conceptual model which describes the *Centipede Revived!* entities and their various relationships is shown in figure 7 in the Appendix. This model captures the key concepts within the game. The diagram consists of several types of relationships as well as the multiplicity that exists between the game entities.

3.1 Domain Model Relations

Following is a brief description of relationships that exit within the model:

- Inheritance

  The `GameObject` class shares an "is-a" relationship with the `Scorpion`, `CentiSegment`, Spaceship, LazerShots, Spider and Mushroom classes. This implies that all the aforementioned classes are derived from the GameObject class. A similar notion is realised between the GameObjectContainer and the Centipede and MushroomField classes.

- Composition

  An ”owns-a” relationship exits between the Centipede and CentiSegments class similar to that of the Spaceship and LazerShots class. This means that the Centipede class cannot stand by itself which also applies to the LazerShots class. This kind of relationships also exist throughout the model.

- Aggregation

  The CollisionHandler class shares an ”has-a” relationship with the Score class, implying that the Score class can exist without the CollisionHandler class.

- Dependency

  The SplashScreen and Animate class depend on the Display class which means that a change in the definition of the Display class will affect the two classes.

The game is modelled in such a way that it has a user-friendly interface that makes it easier for the player to navigate their way through the game. The first thing that the player comes across is a splash screen with buttons; start button and help button. The start button is used to begin the game and the help button shows the window with all instructions of how to play the game. The help window also has a back button that allows the player to return back to the opening screen. At any given time in during the game, the player is allowed to exit the game by simply pressing the escape button on the keyboard. A pause and resume option is available. Figure 1 shows a flow diagram of how the userinterface works.

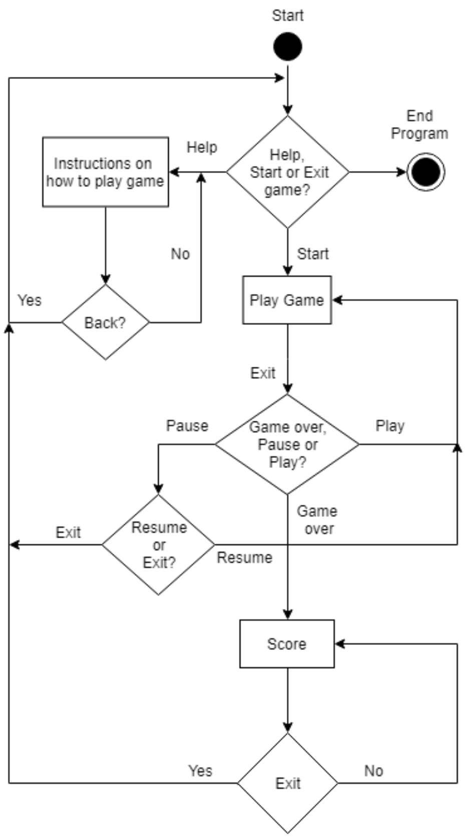

Figure 1 : Flow diagram of the user interface

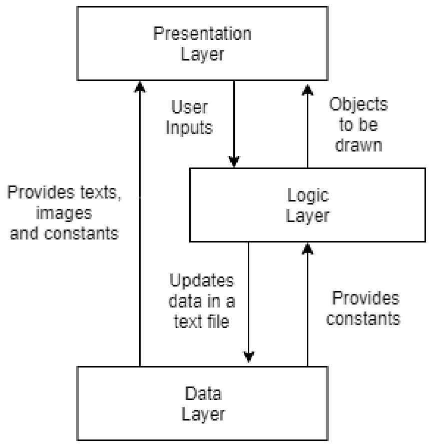

Figure 2 : *Centipede Revived!* Layers

4. CODE STRUCTURE

The code is structured in such a way it adheres to a clean separation of layers or concerns as much as possible. The structure is made up of the presentation layer, logic layer and data layer. The relationship between the layers is illustrated in figure 2 below.

The presentation layer serves as a user interface responsible for drawing and displaying the game objects and handle the user inputs. The logic layer handles all the mathematical computations responsible for the behaviour of game objects. All the game resources are stored in the data layer which contains all images, file names, constants and text files needed for both the presentation and logic layer.

The separation is done in attempt to make any of the layers, especially the presentation and data layer to be easily substitutable. It also allows code to be readable, reusable, maintainable and easy to test.

The code structure also exploits inheritance and polymorphism. This allows for dynamic binding and significant reduction of code and non-violation of the DRY principle.

5. IMPLEMENTATION

This section describes the implementation of all the classes used to construct the *Centipede Revived!* game. The classes are categorised into the three layers mentioned in section 4. The responsibilities of each class, hierarchies within the structure and internal representation of complex member functions are discussed in detail.

5.1 Data Layer

5.1.1 Constants

This is a header file with no implementation. It consists of only constants, in particular, dimensions and positions of all game objects and texts that are displayed on the splash screen. The inclusion of such an interface makes code readable and easy to maintain if changes need to be made.

5.1.2 GameFiles

The `GameFiles` class is responsible for providing data to some of the classes in the presentation layer. Some of the stored data includes game entity textures and game fonts. The class is also responsible for storing and retrieving the top five high scores in the game.

5.2 Logic Layer

5.2.1 GameTypes

The GameTypes interface is responsible for maintaining the game's vocabulary. It contains three enumeration classes namely; the `ObjectID`, `State` and `Direction` classes. `ObjectID` is used to identify some of the game entities within the scope of the game. `State` is used to describe some of the game entities state of being (Alive or Dead). The `Direction` class is used as a reference to the direction of movement of a game entity. The interface also contains a `Vector2D` class responsible for the representation of the game entities in a 2D space.

5.2.2 GameObject

GameObject is an ordinary virtual super class which is used as a contract for a majority of the game objects. The class models most of the similarities that exist between game objects. Three pure virtual functions that must be implemented and overridden exist within the class. Figure 3 below illustrates the relationship between the `GameObject` super class and its sub-classes.

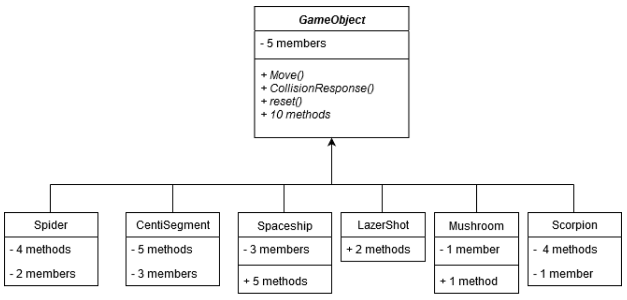

Figure 3 : Inheritance Hierarchy of GameObject class

The sub-classes have different definitions of the movement, reset and collision response hence the corresponding functions in the `GameObject` are pure virtual.

5.2.3 Spaceship

The `Spaceship` is derived from the `GameObject` class as seen in figure 3. It models the spaceship, which is the main entity within the game. The class is also responsible for keeping track of the number of lives that a player has. A life is lost if the spaceship collides with any of the enemy entities and the game ends when the number of lives associated to the spaceship is zero.

5.2.4 LazerShot

The `LazerShot` class, inherited from the `GameObject` class is responsible for controlling lazer shots when they are shot by the player. This class has a composition relatioship with the `Spaceship` class, this means that if the spaceship is destroyed it will also be destroyed. The class ensures that lazer shots are loaded to the spaceship and are set at the position of the player. When the player shoots, lazer shots are released in a vertical direction across the screen with the aim of destroying an enemy

5.2.5 CentiSegment

Inherited from the `GameObject` class, the `CentiSegment` class models the segment of the centipede. The class ensures that the centipede does not go over the screen boundaries. Left, right, up and down movement is also controlled by this class. The segment is either a head or body depending on the state of the segment. If the centipede segment either head or body collides with a mushroom or reaches the edges of the screen, it moves one row down and changes direction.

5.2.6 Mushroom

Mushrooms in this game are fixed objects. The `Mushroom` class is also derived from the `GameObject` class and it is responsible for placing a mushroom across the screen. The *move()* function is left undefined since mushroom objects are static. The class keeps track of the lifetime of a mushroom object. If a mushroom is shot by a lazer shot, it chips away and is completely destroyed after four shots.

5.2.7 *Spider*

The spider is an enemy object that moves in a random zig-zag motion within the bottom 1/3 of the screen. The `Spider` class is derived from the `GameObject` class. When a mushroom collides with a spider, it is eliminated. However, if the spider collides with the spaceship, the spaceship looses a life.

5.2.8 *GameObjectContainer*

The `GameObjectContainer` similar to the `GameObject` class is an ordinary virtual class that serves as a contract to all the sub-classes derived from it. It handles a collection of game objects. Figure 4 illustrates the relationship between the `GameObjectContainer` and its sub-classes.

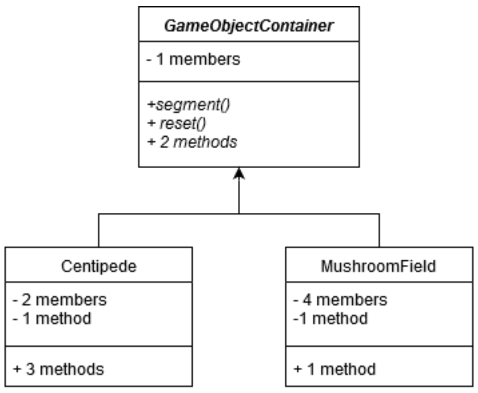

Figure 4 : Inheritance Hierarchy of GameObjectContainer class

5.2.9 *Centipede*

This class is derived from the `GameObjectContainer` class as seen in figure 4. It is a collection of `CentiSegment` objects and ensures that every centipede segment moves accordingly across the screen. The class also keeps track of how many segments are destroyed when shot and updates the status of the segment to dead or alive. If the body segment is shot it splits into two and the segment shot turns into a mushroom. The first segment of the rear part becomes the new head. If a head is shot it is destroyed and the segment before the head turns into a head. The reset member function reconstructs the centipede and sets it to its initial conditions when it is called.

5.2.10 *MushroomField*

The `MushroomField` class is also derived from `GameObjectContainer` as seen in figure 4 because it is a collection of `Mushroom` objects. These objects are scattered across the screen and placed at random positions for every new game and new level. The reset member function generates a new set of random positions for the `Mushroom` objects.

5.2.11 *Player*

The `Player` dictates what type of spaceship movement should be executed according to the player inputs. This class ensures that the spaceship moves within 1/3 of the screen and does not exceed boundaries. The spaceship can only be moved up, down, left or right depending on the player inputs. It also ensures that the spaceship does not go over a mushroom if it collides with it.

5.2.12 *Score*

The `Score` class is responsible for calculating what points should be awarded to the player if an enemy is shot. The points awarded vary with every type of enemy. The class is also responsible for updating the high scores through the `GameFiles` class. Table 1 below shows the point allocated to each enemy object:

Table 1 : Point allocated to each enemy

Enemy	**Points**
Scorpion	1000
Spider	300
Centipede head	100
Centipede body	10
Mushroom	1

5.2.13 *Timer*

The `Timer` class uses the system clock to keep track of when a specific code was executed and when it ended. This ensures timed calls to some of the game entities and functions.

5.2.14 *Collision*

The `Collison` class is responsible for detecting an overlap between positions of game entities. The class uses booleans to notify the `CollisionHandler` class of any collisions that are present.

5.2.15 *CollisionHandler*

The `CollisionHandler` class shares a composition relationship with the `Collison` class. The class responds to flags sent by the `Collision` class to execute a certain behaviour. The collaboration of the two collision classes is further explained in section 6.

5.2.16 Domain

The `Domain` class provides a platform for interaction of all the classes that fall under the logic layer. The class is responsible for updating game objects, resolving collision between game objects and handling the state of all game objects.

5.3 Presentation Layer

5.3.1 UserInputs

The `UserInputs` interface is responsible for handling and detecting user inputs. It contains an enumeration class with all the possible input keys that are defined within the game.

5.3.2 Animate

This class is responsible for drawing textures of the game objects on the screen. If the file of the texture cannot be accessed a `FILENOTFOUND` exception is thrown.

5.3.3 SplashScreen

The `SplashScreen` is responsible drawing the user interface. All the graphics of the game except for the game objects are handled in this class. The opening screen, help screen and game over screen are created in this class as well as the start, help and back buttons. When a player left clicks on a button with a mouse, the position of the mouse is checked to be within the area covered by the button. If it happens that the cursor is within the bounds of the button, a particular window is shown which corresponds to the button.

5.3.4 Display

The `Display` class is responsible for drawing all the objects drawn in the `Animation` and `SplasScreen` class. It also ensures that the window is cleared if needs be. It handles the window and all the events that happen in the window.

5.3.5 GameLoop

The `GameLoop` class provides a plaform where all the *Centipede Revived!* layers interact. It is considered as part of the presentation layer because it is where the final product is presented. This class ensures a smooth flow of the game and is responsible for all the game administration.

6. DYNAMIC COLLABORATIONS

The dynamic behaviour and collaboration of the game entities is illustrated by a sequence diagram shown in the Appendix, figure 8.

6.1 Algorithms

Following is the algorithms used to control the automated game entities.

6.1.1 Centipede

Figure 5 demonstrates the algorithm used to control the centipede. The algorithm ensures that the centipede movement is limited to only the bottom 1/3 of the screen when reaches the bottom of the screen .

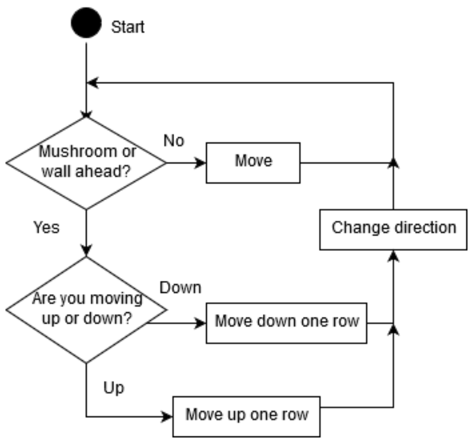

Figure 5 : Centipede Algorithm

6.1.2 Spider

The algorithm used to control the movement of the spider is similar to the one presented for the centipede. It ensures that the spider only moves within the bottom 1/3 of the screen. It uses a random number generator to move the spider across the screen. Different direction of movements are selected at random in accordance to the random number generated.

6.1.3 Collision Detection The collision detection method implemented in the game is the Axis-Aligned Bounding Box Algorithm (AABB) [2]. Figure 6 illustrates how the collision algorithm works.

The algorithm works as follows: When the absolute distance between the centres of the two game objects is less than half the sum of their sizes, the collision flag is set to true. The mathematical representation of the algorithm can be seen in equation 1.

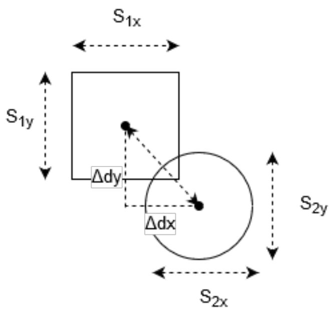

Figure 6 : Visual representation GameObject collision

$$flag = |\Delta dy| \leq \frac{1}{2}(S_{1y}+S_{2y})\&\&|\Delta dx| \leq \frac{1}{2}(S_{1x}+S_{2x}) \quad (1)$$

If the flag returns true the `CollisionHandler` will be signalled to handle a collision.

6.2 Key Dynamic Collaborations and Behaviour

The first key dynamic collaboration of the game is the relationship between the `Animation` class, `GameObject` class and `GameFiles` class. Their key interaction allows for seamless addition of `GameObject` entities. When a new `GameObject` entity is added, its corresponding sprite and presentation to the window is easily determined by using the `ObjectID` and polymorphism with little addition of code. The only addition of code will be of the corresponding texture file name.

The `Mushroom` and `Centipede` form the second key collaboration of the game. When a centipede segment(`CentiSegment`) is shot it turns into a mushroom making the previous segment a head. The new head will then collide with the mushroom and immediate action is taken to prevent the head from moving across the mushroom. The transformation of the centipede segment to a mushroom is performed with the assistance of the `Collision` and `CollisionHandler` class. When a `CentiSegment` is shot, its current position is sent immediately to the `MushroomField` class and a `Mushroom` object is created at that specific position at that instance.

The last key dynamic behaviour is the relationship between the `GameObject` and `ObjectID` class. `ObjectID` identifies some of the `GameObject` entities as an explosion if destroyed to the `Animate` class with the help if the `Collision` and `CollisionHandler` classes. This happens to the spaceship when it looses as life and some enemy entities. The explosion lasts for a second for the spaceship and half a second for the enemy entities. The `ObjectID` also plays a role in assigning the head of the `Centipede`. Initially the first segment is identified as the head by the `ObjectID`. When a segment of the `Centipede` is shot, its `ObjectID` changes to head and this change propagates to other classes that work with the `Centipede` class.

7. UNIT TESTING

The implementation of most classes is done hand in hand with unit tests. Such a development procedure is called a Test Driven Development(TDD) and involves writing code gradually , testing it and adding more and improved code if the test passes [2] . Unit testing is performed in the development of *Centipede Revived!* in order to verify the correctness of most classes. The framework used to test the code is `doctest C++`. Table 2 below shows all the tested classes, number of tests performed and assertions.

Table 2 : Unit Tests

Class	**Tests**	**Assertions**
Spaceship	10	15
LazerShots	6	9
Centisegement	6	12
Centipede	1	1
Vector2D	4	7
Collision & CollisionHandler	8	15
Score	3	6
Mushroom	6	19
Spider	2	9
Timer	3	4
Scorpion	5	11
Total	**55**	**105**

As seen from table 2, most of the tested classes are sub-class of the `GameObject` class. Each of the derived `GameObject` entities are first tested if they are within their defined boundaries and screen boundaries. If the entites are not within the defined boundaries, an `ObjectOutOfBounds` exception is thrown. All the derived `GameObject` entities successfully threw an exception when set outside the defined boundaries.

Movement for all movable objects is also tested. The `GameObject` class has a pure virtual void move function that every sub-class inherits. Each derived class is tested for the its specific movement. All non-movable objects are set to throw a `NonMovable` exception if tested for movement. All movable objects were successfully tested.

`Collision` and `CollisionHandler` play a significant role in the interaction of game entities. The `Collision` class is tested if it can correctly detect collision between two game entities and if it can identify non colliding objects. The `CollisionHandler` class on the other hand is tested if it can correctly handle collision between game entities. Both classes successfully passed all the tests.

The classes in the presentation and data layers were not tested mainly because the classes deal with user inputs and graphics. It is complex to simulate such tests.

8. CRITICAL ANALYSIS

8.1 Strengths

The design and implementation of the Centipede game is successful in decoupling SFML from the game logic. The game entities interact on their own, independent of SFML. SFML is only used for graphics and visualisations. The exploitation of inheritance and polymorphism allows seamless addition of any `GameObject` entity. The use of `ObjectID` makes it easier for the `Animate` class to recognise which texture it must use from the `GameFiles` class to represent a certain entity. Most of the functions in the game classes are broken down into smaller, readable and maintainable code.

8.2 Weaknesses

For every addition of a game entity a new function for handling its collision has to be coded in the `CollisionHandler` class. This makes the `CollisionHandler` class long and more complex with new additions of game entities. This may in turn break the class and sometimes lead to a violation of the DRY principle. The class is structured this way because of the different relationships that exist between the game objects.

The `GameFile` class has no real behaiviour, it only provides textures to the `Animation` and `SplashScreen` class. The textures are loaded in the respective classes. This can be seen as a poor design according to [3].

The small move classes in all the automated game enemy entities violate the DRY principle. With respect to the `CollisionHandler` class and dynamic binding, an ordinary virtual class that could parent enemy entities can be created to exploit polymorphism even further.

Intercollisions between independent centipedes occurs which often leads to the centipedes clustering in one place, this might mislead the player.

9. Future Improvements

The game runs at different speeds on different machines. Possible future improvements could be implementing a frame rate independent game play [4]. This ensures good game performance and a smooth game play.

The `CollisionHandler` class can be shortened to take advantage of polynorphism if classes with similar characteristics are grouped by inheritance.

10. CONCLUSION

The object oriented solution to the 2D Centipede arcade game was successfully presented. The *centipede Revived!* game fulfills all the requirements presented and basic functionality, minor and a few major features were achieved The solution was designed in an attempt to adhere to good programming principles and practices as mush as possible such as non-violation of the DRY principle, readable code and a clean separation of concerns. All the game objects were thoroughly tested and all tests were successful. The solution was critically analysed and it was found that it has some flaws. However, future improvements and recommendations are provided in light of achieving an improved version of the solution.

REFERENCES

[1] Steven Levitt. "Project 2018 Centipede.", 2018. `https://witseie.github.io/software-dev-2/assessment/elen3009-project-2018.pdf`, Last accessed on 2018-06-08.

[2] M. H. A. H. A. Sulaiman, A. Bade. "Inner AABB for Distance Computation in Collision Detection." pp. 1–6, Oct. 2014.

[3] Steven Levitt. "Indicators of Poor Design (aka Code Smells).", 2018. `https://witseie.github.io/software-dev-2/guides/elen3009-code-smells.pdf`, Last accessed on 2018-07-08.

[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-f` Last accessed on 2018-06-10.

Appendix

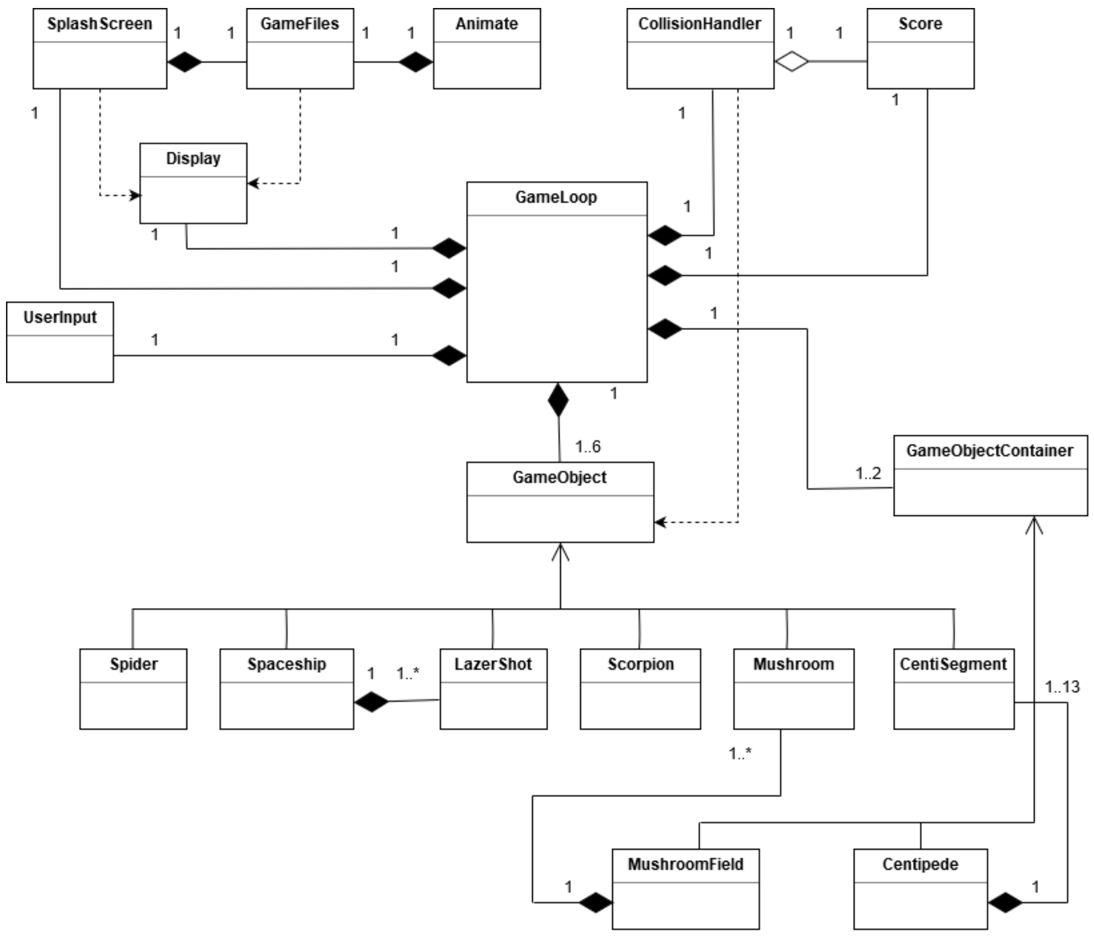

Figure 7 : Domain Model

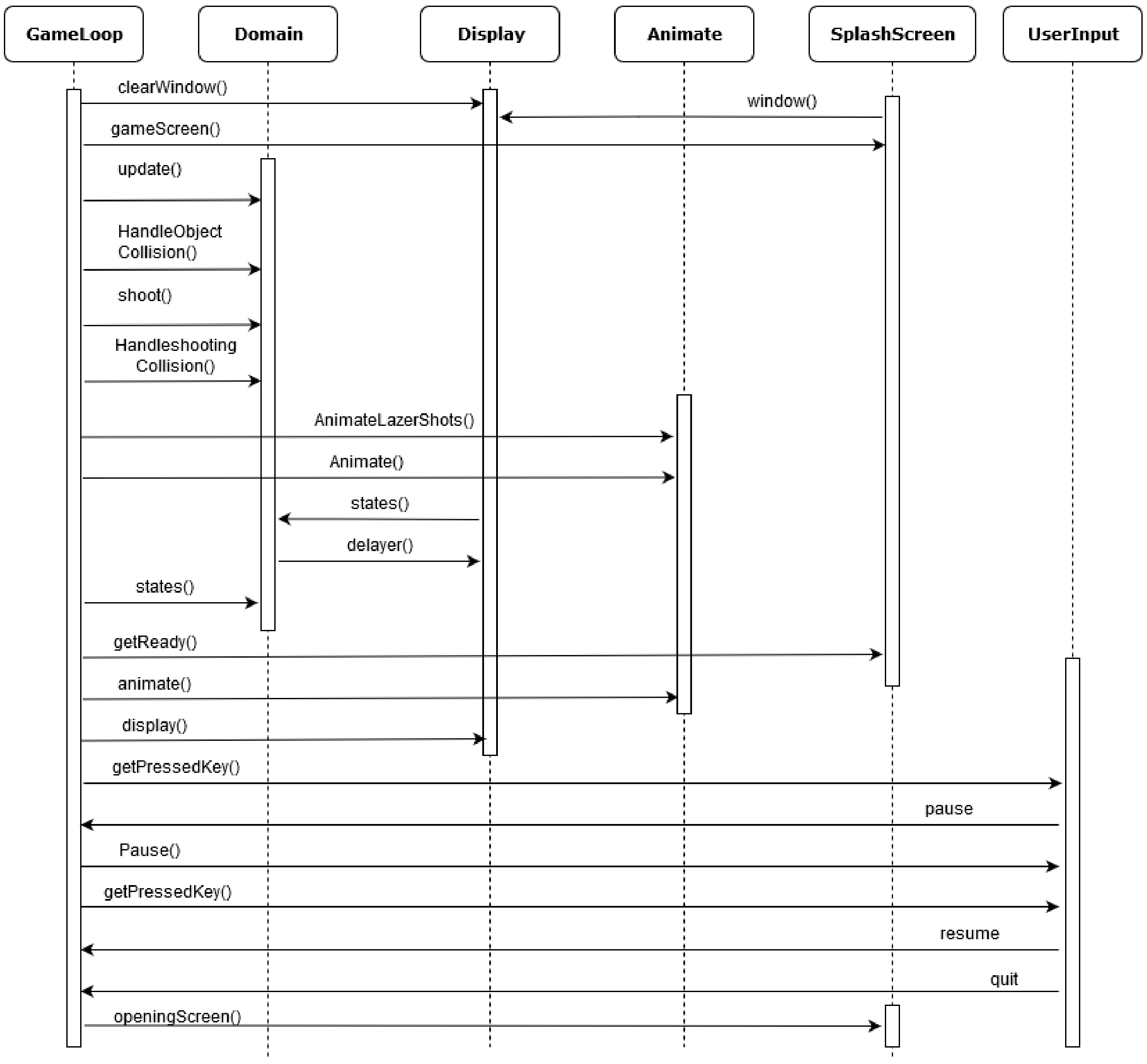

Figure 8 : Sequence diagram of GamePlay