THE DESIGN AND IMPLEMENTATION OF DUEL INVADERS IN C++

Tshepo Chokoe (1705890) and Ntladi Mohajane (1599953)

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

Abstract: This report documents the design and implementation of a fixed shooter arcade game that is written in C++17 and SFML 2.5.0. Design decisions are examined with relevant justifications. Furthermore, the overall structure of the program is also presented together with a discussion of all the significant classes that were implemented. Lastly, the testing procedure of the program is also outlined on the main part of the report.

Key words:

1. INTRODUCTION

Space invaders is the first fixed shooter arcade game that was invented by Tomohiro Nishikado. It was produced and released by Taito Corp. in 1978. The purpose of this technical report is to present the implementation of a two-player version of Space Invaders called Duel Invaders that is subject to the same game mechanics. The game that is modelled in this report comprises of two alien amardas that are facing in opposite directions and two player cannons of which one is at the top edge of the screen, while the other is at the bottom edge of the screen. The two alien armadas are initially positioned back-to-back in the middle of the screen and move vertically towards the opposite ends of the screen while shooting at the player cannons (players). The main goal is to conquer the alien armadas as they attempt to reach the opposite ends of the screen. This is achieved by making the laser cannons to shoot the aliens with their bullets whilst avoiding their shots thus preventing an invasion.

The shooting of the aliens is made to be random to achieve more challenging game-play. Furthermore, the laser cannons can also obtain power ups as players advance through the game which enables them to destroy more than one alien at a time. However, if a player is shot, they lose a life as well as any power ups they may have gained up until that point. Players can move to either edge of the screen to help each other to destroy the aliens. Lastly, the game can also be played in single player where both cannons can be controlled in one of two different modes. The modes in question are mirrored controlled mode and inverted controlled mode.

This report also discusses a high level conceptual model of the domain of the problem, followed by an overview of the code structure. Moreover, a detailed examination of the responsibilities of the implemented classes together with a critique of the final functionality attained and the object-oriented design.

2. CONCEPTUAL MODEL OF THE DOMAIN

2.1 Defining the Model Space

The game can be modeled as a series of two-dimensional rectangular game objects, each of which is placed within a fixed rectangular play grid. The origin of the grid is defined at the top-left corner, while the horizontal and vertical axes are defined at the top and left edge of the grid respectively.

2.2 Defining the Game Objects

The game objects are all capable of interacting with one another and the highest level of abstraction that the game objects can be modeled by, is described by the IEntity class. Objects from this class have two major attributes that allow interactions between them to take place.

The first attribute is the position. Position describes the center of a game object and is modeled as a point on the play grid. The position of all game objects is calculated relative to the origin of the grid.

The second attribute is the hitbox. This attribute describes how much space the object occupies on the on the grid. It is calculated based on the position of the object's vertices. If the hitboxes of two objects overlap, a collision has occurred and appropriate action must be taken. Figure 1 shows an illustration of the play grid populated with game objects.

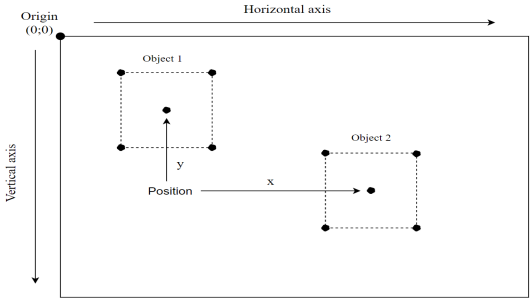

Figure 1: An Illustration of the play grid

In addition, game objects also have knowledge of other parameters about themselves. One of these parameters include the object's orientation. This is a binary value that describes the object's orientation relative to the grid.

Moreover, objects also have knowledge of the grid's dimensions, their own dimensions, the type of object which they are as well as how many points their worth. Furthermore, objects also have knowledge of how many hit-points they possess. These parameters are all abstracted away into a parameters attribute within the IEntity class.

2.3 Defining Movable Objects

Although all the game objects that directly inherit from the IEntity class are capable of full interaction with one another, they are incapable of dynamic movement around the play grid as time changes. Therefore, a new class called IMovingEntity that inherits from IEntity is defined to overcome this issue.

Objects derived from the IMovingEntity class have the potential of incrementing or decrementing their horizontal or vertical position on the grid. However, the exact movement behaviour and increment step varies depending on the type of the object that is inherited. The behavior is implemented

In addition, movable objects also have the ability to determine if they are at either boundary of the play grid and respond accordingly with the appropriate action if necessary.

2.4 Defining the game object hierarchy

There are five main types of game objects that could exist on the play grid at any given time. These exists a Player object, an Alien object, a Barrier object, a Bullet object and a PlayerLife object. Of these objects, Player, Alien and Bullet are of type IMovingEntity, while Barrier and PlayerLife are of type IEntity.

In addition, there are also three additional objects that are inherited from the Bullet class. These are the LeftDiagonalBullet, RightDiagonalBullet and PiercerBullet objects.

A UML diagram illustrating the class hierarchy relationship between the various game objects can be seen in Appendix B.

3. CODE STRUCTURE

The code that implements the game model is divided into three main layers namely the Presentation layer, Logic layer and Data layer.

3.1 Presentation layer

The Presentation layer consist of a render window, as well as a class which contains data members that mimic the game objects. The render window is managed by the SFML library. The class which contains the data members prints the members onto the render window in order to display the model to the user using a GUI. The user responds with inputs which are relayed to the Logic layer.

The reason for the redundancy of having two classes that mimic each other is so that if a new library were to be used to display the render window or the Sprites which render onto it, the Logic layer of the program would not have to change.

3.2 Logic layer

The Logic layer is responsible for maintaining the game loop. It is also responsible for receiving inputs from the Presentation layer and relaying those inputs to the various Data classes, which model the game objects. These classes then respond with outputs that are processed by the Logic layer before being returned to the Presentation layer and displayed to the user.

The main input that the Logic layer receives are keyboard inputs that the user makes, in order to interact with the render window. The Logic layer responds by passing those inputs to the relevant Data classes which in turn return positions back to the Logic layer. These positions serve as the main output that will communicate the game model back to the user through the Presentation layer.

3.3 Data layer

The Data layer comprises of multiple classes that are meant to model the game objects. These classes may also consist of container classes that are used to model a large group of related game objects.

Inheritance is used heavily throughout this layer because all the game objects conform to an inheritance hierarchy that makes it easy to manipulate many of them at once.

Figure 3 below illustrates the relationship between these three classes.

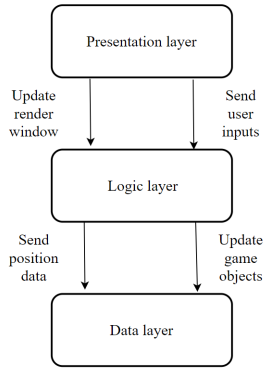

Figure 3: An illustration of how the different program layers communicate with on another.

4. VARIOUS IMPLEMENTED CLASSES AND THEIR RESPONSIBILITIES

This section outlines all the essential implemented class methods together with their responsibilities for the program.

4.1 Armada class

This class forms part of the data layer of the model. It is a container class that encapsulates the aliens away from the rest of the program. Furthermore, it returns a vector containing the positions of all aliens objects when queried by the Logic class. This class also ensures that all dead alien objects that populate the play grid are removed from model. Periodically removing dead entities from the game frees up both processing power and memory for other activities of the model to occur.

4.2 AlienBulletFDactory and PlayerBulletFactory classes

These classes are responsible for generating and managing the bullets produced by the Armadas and the Players respectively. Much like the Armada class, these classes are container classes that ensure that dead objects do not populate the screen. Additionally, they also ensure that the correct Bullet object is produced by the classes that contain them. The logic class does not have direct access to these classes because they are encapsulated away.

4.3 SeparatingAxisTheorem class

The SepartingAxisTheorem class is responsible for handling collisions between two game objects. This method is used because it can determine the occurrence of collisions more precisely than competing solutions such as circle detection algorithms. These methods were initially considered when deciding a strategy for collision detection. However, they were discarded because the corners of the objects may have not been taken into account when calculating collision detection. This is especially true if the objects are not moving directly towards each other. Moreover by using the hit box, the entire size of the object is taken account when calculating the collisions.

4.4 CollisionHandler class

This class is responsible for handling collisions for every game object present in the game. A SeparatingAxisTheorem object exists within this class which calculates the actual collisions. This class decides which of the game objects must be compared when determining collisions. By calculating collisions in this manner, objects which will never collide, such as upright and downright aliens are never compared. This reduces processing time and makes collision detection more efficient than if every object was compared to every other object.

5. TESTING

Only game objects of type IMovingEntity were considered when testing. More specifically, the Player object, the Alien object, PlayerBulletFactory and the AlienBulletFactory were tested. This is because when testing the player object, many of the methods tested were public methods inherited from the IEntity class. This implies that tests written for any object that inherit these methods will inherently pass. In addition, a single alien was tested rather than testing an entire Armada container because it is implied that if one passes, all of the same type will pass as well.

Moreover, the PlayerBulletFactory and the AlienBulletFactory were tested rather than a single bullet. This is because a bullet object must be able destroy itself when it reaches the edge of the play grid. In order for this to happen, the corresponding factories must overwrite the destroyed bullet since it cannot destroy itself.

6. CONCLUSION

The implemented game meets all the basic functionality specifications that were outlined in the project brief. Furthermore, the game adds five minor features together with all the major features. The interactions of the game entities are successfully handled and collisions are accurately detected. Additionally, player input can successfully be translated to movement and shooting. Reasonable victory conditions were implemented to establish a pleasurable gameplay experience.

Appendix

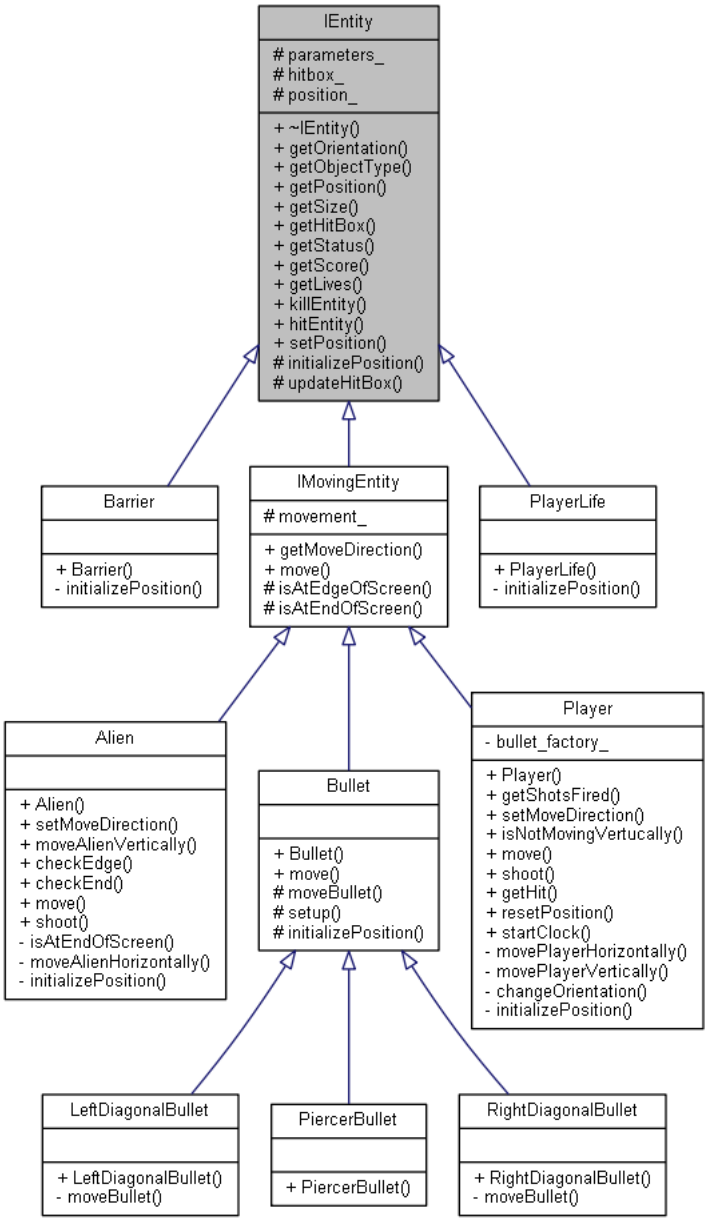

Figure 1A: UML inheritance diagram of all game objects

Appendix

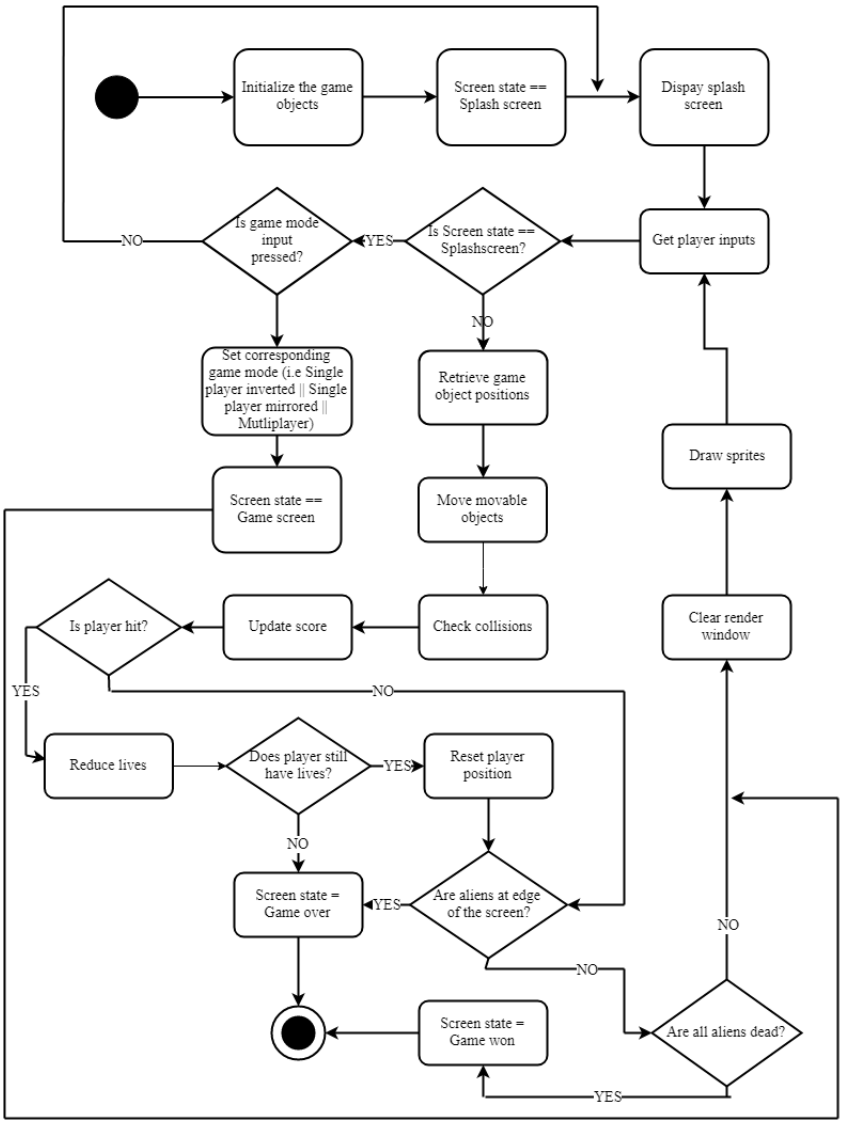

Figure 1B: Flow chart describing the class logic.