IMPLEMENTATION OF DUAL INVADERS ARCADE GAME USING C++ AND SFML

Tshepo Monene (1737389) and Goitseona Mohajane (1027296)

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

Abstract: The concept of Object-orientated programming in ANSI/ISO C++ was used to model an arcade game called Dual Invaders.The designed game makes use of a graphical library called Simple and Fast Multimedia Library (SFML) was used. The design was modelled using two important concepts of OOP, namely, *Abstraction* **and** *Encapsulation***. The designed game can only be played in two player mode using the keyboard. Critical evaluation and future recommendations which can be incorporated on the current design are discussed in this project.**

Key words: OOP,C++,SFML

Space Invaders is a Japanese video game created by Tomohiro Nishikado in 1978. The game was manufactured and sold by a Japanese company called Taito which specialises in video games, toys , arcade cabinets and game centers. It was the first fixed shooter game and set a template for the shoot 'em up genre. The game was originally controlled using buttons to move left, move right and shooting[1].

The aim of this project is to design, implement and write a keyboard-driven, two-player version of Space Invaders called Dual Invaders in groups of two.The implemented game is primarily based on the same game mechanics as Space Invaders.Unlike Space Invaders,the implemented game is played on a two-player mode and the player is able to move up and down. In order to achieve this functionality, an object-oriented programming (OOP) approach in C++ and a graphical library called Simple and Fast Multimedia Library (SFML) were used.In this report, the background, design approach and critical evaluation of the solution will be outlined. This report further provides possible improvements that can be incorporated in the existing solution.

1. BACKGROUND

1.1 Project outcomes

On completion of this project both students should be able to[2]:

- Perform and object-oriented programming decomposition/ analysis of a software problem which involves a variety of interacting objects.
- Design and implement an object-oriented solution in C++.
- Understand and use existing software libraries in conjunction with their own code.
- Provide a test suite verifying the correctness of their software.
- Provide the requisite documentation for a software project, including an automatically generated technical reference manual.

1.2 Project constrains

The following project constrains apply[2]:

1. *Code Implementation*:
   - The game needs to be implemented using ANSI/ISO C++ using SFML 2.5.0 library. Earlier versions of SFML may not be used.The game must run on the Windows platform.
2. *Graphics*:
   - The game must display correctly on a screen with a resolution of 1920 $\times$ 1080 pixels.This implies, when the game Window is maximised it may not exceed this resolution.
   - OpenGL may not be used.

1.3 Success Criteria

The project is considered successful if the following conditions are met:

- The game is implemented using object-oriented programming in C++.
- Basic functionality is achieved(as outlined in [2]).
- A technical reference manual, declaration, project report and published release notes on GitHub are provided.

2. DESIGN MODELLING

2.1 Separation of layers

It is good to separate a software program into different layers. This allows the developer to write a code that is easy to test, maintain and also makes it easy for other developers to understand the code. The layered pattern method was used to implement this game. Figure 1 shows the three different layers employed in this design. Each layer in the design provides services to the higher layers. The presentation layer contains the graphical design of the program and makes it possible for the user to interact with the game. The presentation layer does not contain any logic code that does not require user input.

The logic layer provides an abstraction so that the

presentation layer does not know about the data layer. This is good because the presentation layer can be changed without changing the data layer. The logic layer manipulates the data stored in the data layer and passes the information to the presentation layer.

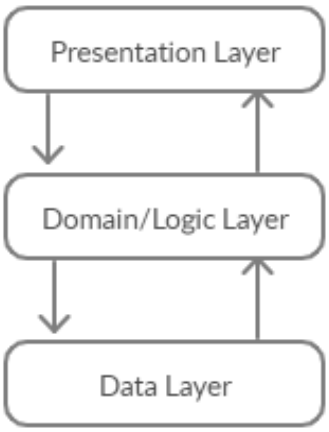

Figure 1 : Different Layer of the design

2.2 Object-Orientated Programming Approach

An OOP approach was used to model the logic layer of the design. The logic layer is the most important layer of the design as it is responsible for manipulating data. The logic layer was modelled in terms of objects. Each objects data members and methods are encapsulated into their appropriate units/ classes. This is achieved through abstraction. Different class objects interact with each other using access specifiers inside the respective classes. This method allows the developer to let the user/client know what the program does and not how the functionality was implemented. Table 1 shows different objects and their behaviours.

Table 1 : Different Objects and their respective behaviours

Object	**Behaviour**
Player	*Move left, right, shoot and collide*
Enemy	*Move left, right and collide*
Bullet	*Move up, down and collide*

3. DESIGN IMPLEMENTATION

The subsection to follow define and explain the classes in each layer and their responsibilities.

3.1 The Presentation layer

The presentation layer makes it possible for the client/user to interact with the code.This part of the code does not contain any logic code in it that does not require user input. It is also responsible for drawing and displaying information on the screen. This allows the user to interact with the code. As depicted in figure 2, the user/client is able to execute the functionality presented.

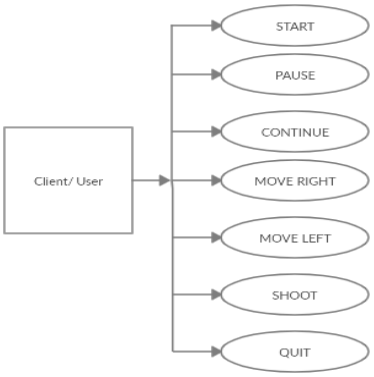

Figure 2 : Client Interaction

Keyboard Handler Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members and functions which operate on this data members are wrapped around inside this class. Other classes can only use the class object(worker of the class) to call the member functions to operate on the data members. Only public member functions are shown to the other classes.

As mentioned above, this is a keyboard controlled game and this class is responsible for constantly checking if any key is pressed by the user. If any key is pressed, this class returns an enumerated type called INSTRUCTION or DIRECTION depending on what the user has pressed. This was done to separate the logic layer from the presentation layer. This class depends entirely on SFML. SFML provides a class called *Keyboard* with has a member function called *isKeyPressed*, this function is responsible for checking if a certain key is pressed on the keyboard. When a certain key is pressed. this function returns a true Boolean variable and false otherwise.

Game Sprites Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members and functions which operate on the data members are wrapped around inside this class. The object of this class allows other classes to access the data members through the member function of this class.

In order for the user to be able to interact with the game, good graphics to bring the objects into realization are required. This class is responsibility for loading textures for each sprite, create and return the sprites. This class depends entirely on SFML. It makes use of the *Texture* class to load textures and *Sprite* class to create the sprites.

Text Handler Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members used to store the texts and the functions which create and draw the texts are encapsulated inside this class. The object of this class is responsible for calling the member functions which draws the texts.

In order for the user to play the game they need to know the instructions. The Text Handler class responsibility is to draw the instructions on the window. It does this by loading the required fonts, creating the string text and drawing these texts on the window. It does not display anything on the window. This class depends entirely on SFML. It uses the, *Text* class to create the text and the *Window* class to draw the text.

Object Bounds Class: This class is responsible for returning a struct variable called *dimensions*. Dimensions contains two float variable, namely the w(width) and h(height). This class is a helper class as it stores and return each sprites dimensions.

Screen Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members used to store the relevant information and the function members used to display certain entities on the window are wrapped inside this class. Other classes use the object of this class to call the function responsible for displaying entities on the window.

This class responsibility is to retrieve all that need to be displayed on the screen. It interacts with other logic classes that are responsible for updating the positions and state of game entities. This class is entirely dependent on SFML. It uses the *Window* class and its member function called *Display* to display game entities on the screen.

3.2 The Domain/Logic layer

This is the most important layer of the design and it is modelled using OOP in C++. This layer is responsible for manipulation and processing of data. The following classes were used to model the logic layer of the design.

Player Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members used to store the each players coordinates and the member functions used to update and return these coordinates are wrapped inside this class. The object of this class is responsible for calling the member functions responsible for updating and returning the coordinates.

This class is responsible for the players positions. It updates the players position depending on the user input and returns the updated position. This applies for both players of game. The players position is updated using *Equation 1* and *Equation 2* , where the speed determines by how much the player should move in the chosen direction.

- moving the player to the right

$$Xposition = Xposition + speed \tag{1}$$

- moving the player to the Left

$$Xposition = Xposition - speed \tag{2}$$

Enemy Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members used to store the enemy coordinates stored in a 2D vector of floats and the members functions used to move, update and return these coordinates are wrapped inside this class. The object of this class is responsible for calling the member functions responsible for moving, updating and returning the vector containing the respective coordinates. This applies for both the enemies at the top and those at the bottom.

This game consists of two groups of aliens moving in a formation. Each group has twenty four aliens. These aliens are divided in three rows and each row has eight aliens. The top half aliens move towards the top of the screen. The bottom half aliens move towards the bottom of the screen. Both groups start of from the left and move towards the right. Once the right screen boundary is reached, the top group moves one row up and the bottom group moves one row down and both groups start moving to the left. This formation and direction continues provided that the game over condition is not met.

This class is responsible for setting the correct initial coordinates and moving each group of aliens in the right formation and direction. It uses *Equation* 1 and *Equation* 2 to move each aliens in the formation to the right and left respectively. Each aliens X-position is constantly checked to see if it is less than the boundary. If any alien has reached the boundary, all aliens are moved one row up or down for the top group and bottom group respectively. Both groups continue in the opposite direction of the boundary that was reached.

Equation 3 is used to move the top formation up and *Equation 4* is used to move the bottom formation down. The top and bottom aliens Y-Position is decreased and increased respectively. This is done by traversing the vector which contains each aliens coordinates.

- moving the alien to the top

$$Yposition = Yposition - speed \tag{3}$$

- moving the alien bottom

$$Yposition = Yposition + speed \tag{4}$$

Bullet Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*.The data members used to store the bullets coordinates and the members functions used to move, update and return these coordinates are wrapped inside this class. The object of this class is responsible for calling the member functions responsible for moving, updating and returning coordinates of the bullet. This applies for both the top and bottom player bullets.

Initially the players are aligned such that the bottom and top player are facing each other. As the game continues, both players will move in a direction dictated by the user. Furthermore both players can shoot at any time and any position validated by the Player Class. The Bullet Class is responsible for setting the bullets coordinates to the respective coordinates of the player who shot. When a shooting event is called, this class gets the players coordinates at that instant. These coordinates are returned to the respective bullet vector holding the bullets coordinates of that player.

An enumeration type called IDENTIFIER determines which player shot so that the right coordinates of the right player can be updated. This class is also responsible for moving the bullet in a vertical direction. *Equation* 3 and *Equation* 4 are used to move the bottom and top players bullets respectively.

Collision Class: This class models two OOP concepts, namely *Abstraction* and *Encapsulation*. The data members and member functions used to perform arithmetic operations on the data members are wrapped inside this class. The object of this class is responsible for calling members functions of this class to give information on whether collisions have been detected or not.

The responsibility of this class is to check if any two entities have collided with each other. This class achieves its functionality by using *Equation 5* which is known as the distance formula. The distance formula is used to determine the distance between the two entities using their x and y coordinates. The distance between the two entities is compared to the sum of the radius of the two entities. Since squares were used to model the entities, the width of each entity was divided by two and approximated to be the radius of that entity. If the distance between the two entities was found to be less than the sum of the radius of both entities then a collision was detected(see figure 3).

$$Distance = \sqrt{|(X_1 - X_2)^2 + (Y_1 - Y_2)^2|} \quad (5)$$

Figure 3 models the approach used to determine the collision.

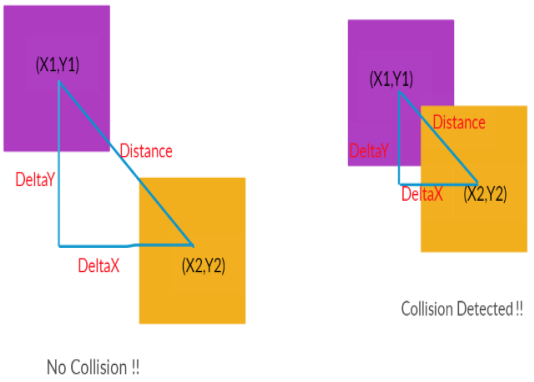

Figure 3 : Collision Detection

4. DYNAMIC OBJECT INTERACTION

The functionality of the game depends entirely on the objects in the design. *Figure* 4 models the interaction of different game objects. It can be seen from the figure that the user only sees what is displayed by the screen. This is good because the client has to know only about the presentation and not the logic.

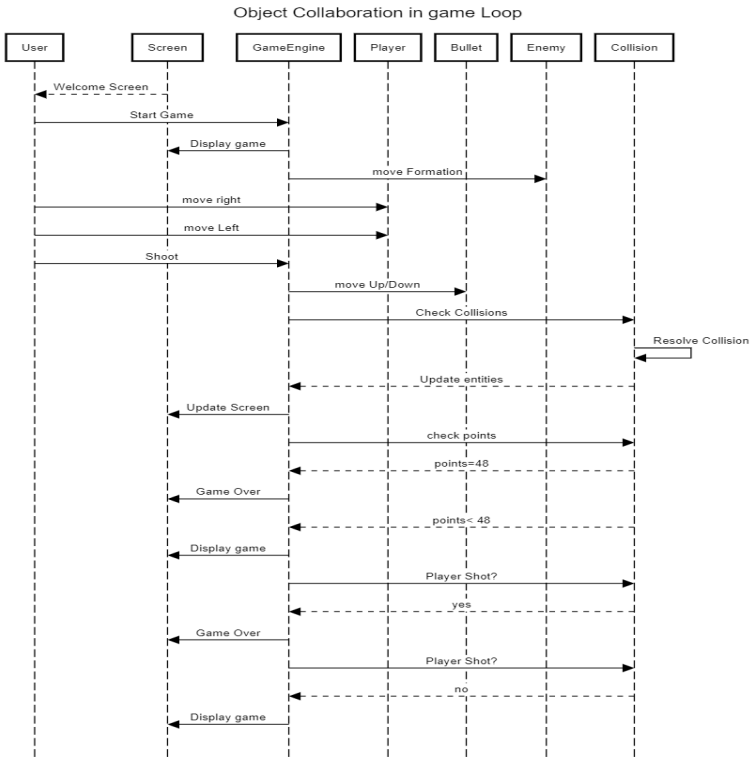

Figure 4 : Object collaboration

5. UNIT TESTING

In OOP a unit is a class and testing these individual classes is known as unit testing. This is done to uncover hidden bugs or defects that might exist in that particular class/unit. The platform used for the unit testing is the doctest C++ testing framework.

5.1 Testing Player Class

1. *Setting and updating the coordinates :*
   This class was tested to set the correct initial positions of the players. Furthermore it was tested for updating the players position when the coordinates are changed.
2. Keeping the player within screen bounds This class was tested to keep the players coordinates within the screen boundary.When a player is at any boundary, their coordinates cant be increased to take them off the screen.

5.2 Testing Enemy Class

1. *Setting and updating the coordinates :*
   This class was tested to set the correct initial positions of both the top and bottom aliens. Their respective movement and left-right boundary conditions were tested. The top and bottom boundary conditions for game over conditions were not tested.
2. *Keeping the formation and right direction :* This class was tested for the sideways movement and their vertical movement when a boundary is reached.

5.3 Testing Bullet Class

1. *Setting and updating the coordinates :* This class was tested to set the bullets coordinates to that of the respective player. When a player moves, the bullet class updates to the correct coordinates of that player.
2. Moving the bullet vertically This class was tested to move vertically from the horizontal position of the player that shot and maintaining that horizontal position.

5.4 Testing Collision Class

5.4.1 Bullet Player Collision

1. *Detecting the collision :* This class was tested to detect collision between a player and bullet. Collision detection were tested in three main areas: Middle of the screen, left and right boundary.

5.4.2 Bullet Alien Collision

1. *Detecting the collision :* This class was tested to detect collision between a bullet and an alien. Collision detection were tested in three main areas: Middle of the screen, left and right boundary.

5.4.3 Alien Player Collision

1. the collision: This class was tested to detect collision between a bullet and an alien. Collision detection were tested in three main areas: Middle of the screen, left and right boundary.

5.5 Classes That Were Not Tested

The above tested classes deal with the games logic. It is hard testing classes that depend on SFML(Presentation). Furthermore an assumption was taken that the SFML developers have conducted their own unit testing.

6. CRITICAL DESIGN EVALUATION AND FUTURE RECOMMENDATIONS

This section focuses on how the game functionality was achieved. It explains how objects of different classes interact with one another to achieve the games basic functionality.

6.1 Game Functionality

The designed solution met the basic functionality as outlined in [2]. In designing the game, efficiency was not the main priority thus the game can be classified as less efficient. The game functionality can be greatly improved by including the following:

- Reward system for killing aliens(points)
- The highest score achieved
- Different difficulty levels(stages)
- Sound enhancement(Impact detection and game music)
- Game obstacles
- Alien firing power

During the design process, trade-offs were made. Attaining basic functionality was the main priority and this allowed both developers to put more time on other parts of the design.

6.2 Rejected Design

The first design was a single player mode where the user controlled both players simultaneously. The moves of the second player were a mirror of the first player. In this mode shooting for one player resulted in both players shooting. The players movement and bullets were dependent of each other and happen simultaneously. Furthermore the bottom player could move vertically to join the top player and start shooting from the top side.

A major disadvantage of this design was that the game supported single player mode only. This resulted in the initial design being discontinued and another design was implemented. While the rejected design was manageable and functional, the new design had to change major parts of it. This changes resulted in certain logic classes implementation changing. Major classes that were changed are:

- Player Class
- Bullet Class
- Collision Class
- Keyboard Handler Class (include player two keys)

6.3 The Final Design

The final design is a two player mode, meaning it can be played by two users. The implemented design allows both user to execute the functionalities presented in *Figure* 2. The player is welcomed with a splash-screen showing all the instructions of the game. The game end as outlined in [2].

7. DESIGN DEPENDENCY ON SFML

A good design has good separation of layers. One way of measuring the quality of the design is by comparing the presentation layers dependence to the logic layer. This section focuses on classes that would need to change if a different graphics platform was used. In this design only presentation classes would change since logic knows nothing about the presentation. A majority of presentation classes would change due to separation of concern.

8. CONCLUSION

An arcade game called Dual Invaders was designed, implemented and written in C++ with an aid of a graphical library called SFML. The design was broken down into three layer, namely presentation, logic and data layer. The logic layer of the implemented design was tested using an automated testing framework called doctest. Two important concepts of OOP were employed to model the logic layer of the design. The implemented design achieved basic functionality however future recommendations to improve the design were discussed in this report.

REFERENCES

[1]S. Levitt, Project 2019- Dual Invaders, https:/ /wit-seie.github.io/software -dev-2/assessment/ elen3009-project-2019.

[2]Wikipedia. "Space Invaders.: http://en.wikipedia.org /wiki/Space$_i nvaders, Lastaccessed : 24September$2019.