The design, implementation and critical analysis of the arcade game Duel Invaders using an Object-Orientated Design approach in C++.

Fortune Iga 1390898, Sinawo Dlulisa 1617250

School of Electrical and Information Engineering, University of Witwatersrand, Johannesburg, 2050, South Africa.

Abstract

The project presents the design, implementations and critical analysis of the arcade game, Duel Invaders. The design is implemented in C++ using an SFML v2.5.0 framework with a Doctest v1.2.9 testing Framework. The game domain is modelled as cartesian plane which provides a mathematical description of the problem and provides a mathematical approach to describe and model the game dynamics such as movement and collisions. The design model is a composition-based model that utilises the object-oriented software design approach. The implementation and testing of the game objects are shown, showing the designed game meets basic functionality, two minor features and two major features. Furthermore, a critical analysis of the design is presented. Future improvements such as information hiding and a more extensive functionality of the designed game, in accordance with the mechanics of the game are suggested.

1. Introduction

The purpose of this report is to convey the design and implementation of the arcade game Duel Invaders using an object-orientated design approach in C++. The report details the problem analysis, design model and code structure, design implementation, unit-testing, critical analysis, future improvements and optimisation of the game.

Section 1 shows a brief introduction. Section 2 shows the problem breakdown and analysis, showing the breakdown of the problem for easier implementation and execution. Section 3 shows the design approach and decisions taken, showing the model of the design. Section 4 shows the implementation of the design, revealing the separation of concerns in the model. Section 5 shows the overall game flow of the implemented design. Section 6 discusses the testing framework used to analyse the functionality of each object of the implemented design. Lastly, section 7 presents a critical analysis of the implemented design as well as future recommendations and optimisations of the design. A conclusion summarising the whole project is presented.

2. Problem Analysis and breakdown

Duel invaders is a two-player variation of the arcade game space invaders. The game mechanics involve two players shooting against armies of aliens. The game is won when the players shoot all the aliens. Conversely, lost when aliens shoot both players, aliens reach either of the players or the players shoot each other [1].

Adhering to object-orientated design principles the game can be broken down into smaller constituent parts, which can be modelled as objects in software design. The players, aliens and bullets are the main objects of the game. In software design however, these can be further broken down into smaller objects and modules which in turn allow for visualisation of the problem and an in-depth understanding. Analogous to the game player, a player in software design can further be broken down into an object that has a weapon and can move. This allows for easier time management and implementation of the design which lead to a quality product.

The interactive nature of the game provides an easy and understandable visualisation of the game and the problem. It also provides an interactive learning approach which induces interest in software design and development. Furthermore, it encourages and aids in learning fundamental software principles such as visualisation and object-orientated design.

The game domain is analysed and visualised as a two-dimensional plane; each object modelled as a point having a location represented by rectangular cartesian co-ordinates. This allows for the problem domain to be analysed mathematically which simplifies the logic flow and execution of the designed game. Furthermore, this allows for conceptually challenging concepts such as collision detection or bullet movement to be accurately and easily modelled and implemented in a reasonable manner. Hence the fundamental logic of the designed game relies on mathematical manipulations. See figure 1 below showing the modelling of the game domain.

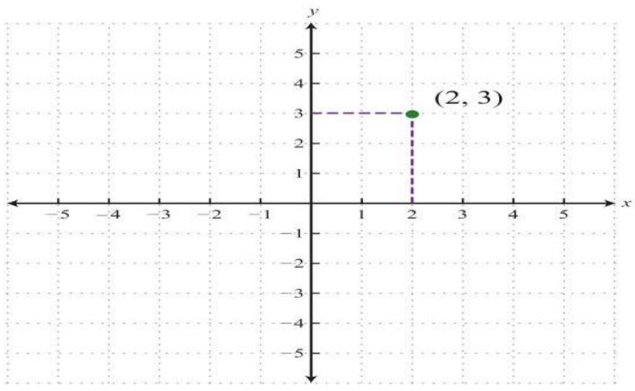


Figure 1: Game domain model.

3. Code structure and design modelling

The aim of the design process is functionality, simplicity and efficiency. As such, the designed model and objects are created to align with the vision of the design while adhering to an object-orientated approach. This becomes the governing principle and the backbone of the design process to realise the final system. To achieve this the problem is analysed and is broken down into smaller components (objects). This provides a level of abstraction which shows the dependencies and interactions of the objects and how they contribute to the wholistic final product.

Aligning with the principle of the design, a composition-based model of the game is designed. Various factors are considered in designing the model.

3.1 *Problem analysis and modelling*

A breakdown and analysis of the problem shows the building blocks that ultimately form the final product. In the context of this project designing duel invaders, a game object such as a cannon contains bullets while a higher-level object such as a player contains a cannon. This analysis shows the 'has a' relationship of a composition design approach. Furthermore, this adequately describes the problem and thus an accurate model can be implemented which contribute to a high-quality output of the project.

3.2 *Flexible design approach*

A composition-based model allows for a flexible design and implementation due to the nature of the model as each object contains a lower-level object. Consequently, a higher-level object can be redesigned and improved without compromising other objects in the system. This allows for a flexible design that can deal with variances and changes in the responsibilities of each object of the whole system. In the context of this project a player object can have a cannon and in the later stages of the design have a shield or explosive, while an alien object can only have a cannon. To implement such variations a shield object is created and can be added in the player object while an alien object remains constant. This is different from an inheritance polymorphic hierarchy, where a derived class would inherit all the actions of the base class regardless whether they are needed (Rebuffed bequest). This allows for a flexible design while inheritance needs the designer to forecast the structure of the design. Changes to the structure affect all derived objects.

3.3 *Object-orientated design*

Lastly, this deign model epitomises the notion of an object-orientated design. Each object is a standalone object and does not need the linking of other objects to function. This clearly shows the actions and responsibilities of each object and how they contribute to the final system. Furthermore, this allows for an extensive implementation and easier unit-testing which lead to a high-quality output. See figure 2 below showing the dependency diagram of the system.

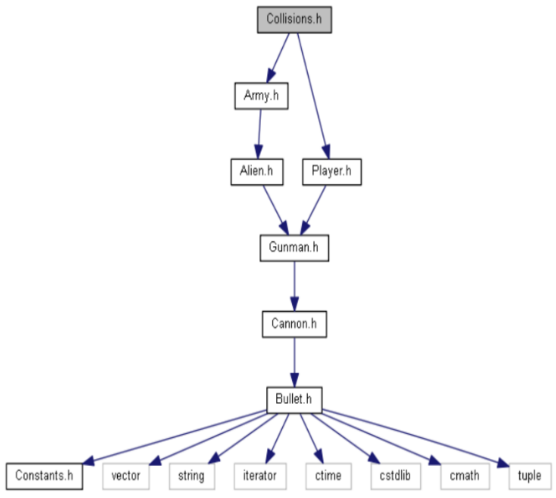


Figure 2: System dependency diagram

4. Implementation

The program is implemented in layers in order to maintain the separation of concerns principle [1]. These layers include data, logic and presentation layers. Implementing the design model, a framework is designed which outline the knowledge and responsibilities of each object and how it interacts with the hierarchical structure. This framework with the design model provides a guideline of implementation and testing of each object to meet the design and produce a coherent, high-quality system. See Appendix A for the game implementation and unit testing framework.

The classes of the designed game are contained in different layers according to the role they play in the design. See figure 3 below which shows class and layer relationships.

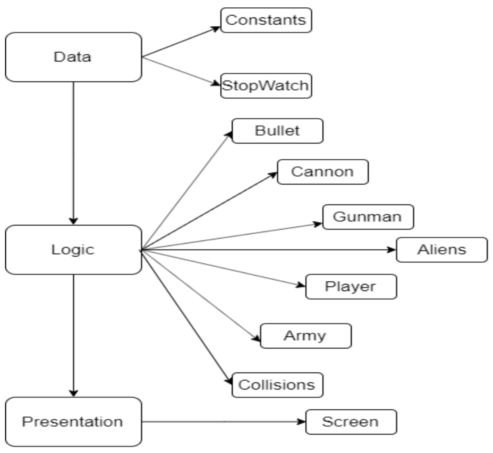


Figure 3: Class-Layer relationships

4.1 *Data Layer*

The data layer contains the *Constants* class, which has a struct with all constant values such as screen dimensions and object sizes. Information from this layer is collected and used in other layers. A stopwatch is also contained in this layer which serves as a timer.

4.2 *Logic Layer*

The logic layer is responsible for processing information and decision making. This layer acts as a link between the data and presentation layers. Within this layer exists different levels of abstraction, where the lowest level is the bullet class. The layer serves as the engine and intelligence of the game. This layer is composed of the game objects are shown in the dependency diagram in figure 2.

4.2.1 *Bullet class*

The bullet class provides the program with an object which encapsulates the idea of a bullet. This component is the lowest-level object present in the system structure. It serves as a fundamental object that will receive instructions and is used extensively by higher-level objects. A bullet in the game Duel invaders is a key requirement whereby, without it the game ceases to exist. All game entities and mechanics involve around a bullet and are directly linked to a bullet. Therefore, fittingly this abstraction of a bullet is the most fundamental building block or object in the entire design.

Due to its importance the bullets actions and responsibilities are carefully designed and implemented. As shown on the framework the main knowledge and action of a bullet is knowing and setting its current position. This is achieved by having to setter function which takes in the desired co-ordinates as parameters and assigns the current co-ordinates of the bullet object. Two getter functions are created to retrieve the co-ordinates of the bullet.

4.2.2 Cannon class

The next level of on the hierarchy structure is the cannon class. This class abstracts the notion of a weapon of the game, hence contains bullet objects. The storage container of the bullets is a vector from the standard library. Outlined by the framework the actions of this class include movement of the cannon object in all directions, shooting bullets, moving the shot bullets and deleting bullets from the container of bullets.

To implement the movement of the cannon, using the game domain mathematical model a movement translates to the adding or subtraction of the speed constant from the x or y co-ordinates. In the case of moving right, the x co-ordinate is increased by the speed constant and subtracted for movement to the left. This method applies for up and down direction on the y co-ordinate.

To implement the shooting functionality, a bullet object is created and initialised with co-ordinates that represent to the top or bottom of the cannon object and added to the existing bullets of the cannon.

Furthermore, to implement the movement of a bullet, the existing setter bullet functions are used by cannon class by specifying the location the bullet must move to from its current position. The movement of the bullet is one-directional in the vertical axis therefore, only the y co-ordinate changes.

Lastly, the removal of a bullet is achieved by the deletion of an element in the bullet vector.

4.2.3 Gunman class

A gunman class is the next level in the hierarchical structure, this class adds intelligence and restrictions of the cannon class. In accordance to the framework, this classes actions include movement, orientation and boundary checking. Most of the functionality has already been implemented in the internal object which is the cannon.

This class sets the orientation of the cannon class by a variable that is up or down represented by 1 or -1. This orientation is important because allows the gunman object to be versatile and adaptable, hence it is used by both the player and alien classes in the subsequent hierarchal levels. Lastly, restrictions are imposed on the movement functionality in accordance to the game specifications.

4.2.4 Player class

The player class is used as an interface between the presentation and logic layers. Its objects are represented by the player icons used to play the game. This class has a gunman object and extensively uses its implementation.

4.2.5 Alien class

Like the player class and alien class has a gunman and therefore extensively uses it. All the functionality stipulated in the framework is implemented in the lower-level classes.

4.2.6 Army class

This class abstracts the idea of a group of aliens. An army contains soldiers. Similarly, here an army contains aliens. The responsibilities of this class are shown on the Framework and include knowledge of the number of aliens, responsible for the automated movement of aliens and deletes an alien in the group of aliens.

The implementation of the movement is determined empirically and is controlled by Boolean variables that ensue the correct movement pattern. The implementation of number of aliens and deletion of aliens is implemented using the standard library vector functions.

4.2.7 Collision detection

The proper functioning of the game depends on good collision detection. As such, a collisions class is created in the logic layer of the program to identify and act on all occurring collisions. Two objects are said to be colliding if the absolute difference of their x-coordinates and the absolute difference of their y-coordinates are less than half their combined lengths. This class receives player and army objects as parameters and checks collisions between their components. It identifies two different types of collisions and handles them differently. The first type of collision is a collision that requires the involved objects to be deleted without ending the game. These include collisions between alien objects and the player bullets. Also include collisions between alien bullets and player bullets. The class uses the *RemoveBullet* and *RemoveAlien* functions of the player and army classes to delete the colliding objects. The second type of collision is one that requires the game to end immediately. Such collisions include those between bullet objects and players. Also, between alien and player objects. If a second type collision is detected the class return a bool variable instructing the game to end.

4.3 Presentation Layer

The presentation layer is the interface between the program and the user. It receives external user input and uses the appropriate logic functions to act on them. It then makes use of SFML to form graphic representations of the logic layer responses. The presentation layer contains one class called Screen. The Screen class has functions that enable it to manage user inputs in the form of SFML events, and use the necessary player and army functions to draw the results of these actions

5. Game flow

The game flow is managed in the screen class of the presentation layer. The *viewScreen* function which runs a loop that only breaks when the window is closed is called after the screen constructor is initialized. Within this loop, the program polls a function called *event_manager* that analyses different user inputs and sets appropriate private bool variables. When the user starts the game, the *event_manager* starts the timer which is used to control the shooting of the armies. Before the user starts playing, a splash screen is shown which welcomes the user and gives the game instructions. When the game is over, the screen displays a splash screen showing whether the user has won or lost. When the user presses the assigned key to start the game, a function is called which draws the player, alien and bullet graphics, while responding to user inputs and logic functions. A flow diagram illustrating the method of this function is shown is figure 4 below.

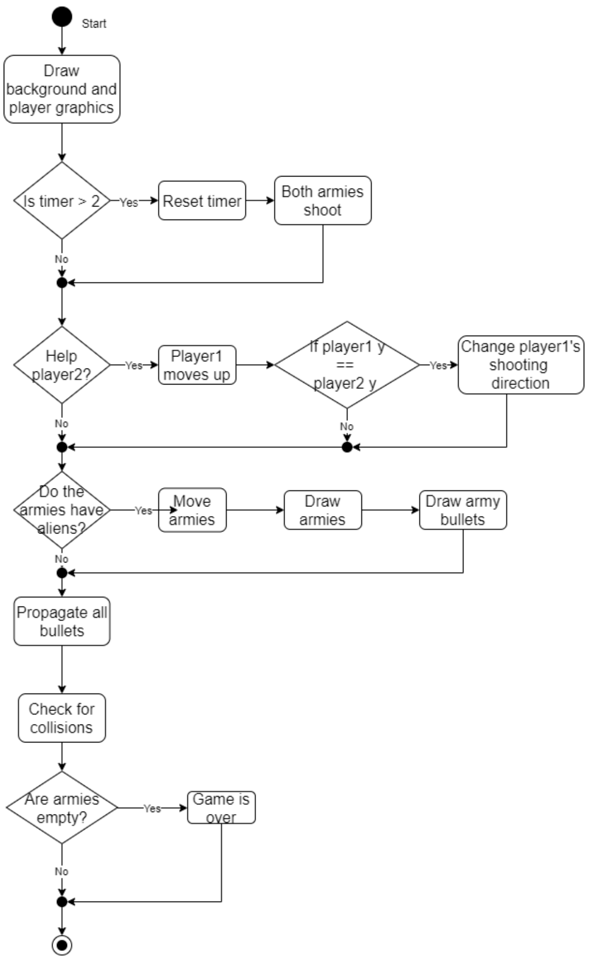


Figure 4: Flowchart of the designed game

6. Game Unit Testing

Testing is a key cog in the software design pipeline. It serves as a quality insurance of each individual object and the overall system. It is essential in achieving a quality product. Unit testing is individually testing each unit or component of the system, namely an object or class separately from the whole system. This simplifies the testing process as only a small portion of the system is tested at a time rather than the whole system once which is difficult to implement and prone to ineffective and erroneous tests. Furthermore, unit testing epitomises the iterative nature of the engineering design process as it shows the developer missing key aspects of an object and possible improvements thus re-implementing the class until a high-quality level is met.

A test-driven development (TDD) approach is a method of implementing unit testing whereby the tests are written first and code written to make the tests pass. Although this method is effective and useful as it provides a framework in which the objects are implemented with, it is not used in the development of duel invaders. Conversely, the unit-testing approach is designing and implementing each object followed by testing that object to meet the test criteria.

The testing framework used is Doctest and the Mingw compiler.

Due to the structure of the design model of duel invaders extensive tests are implemented on the lower-level objects such as the bullet class while higher-level objects have fewer tests since the methods used have already been tested in the low-level hierarchical structure. Furthermore, tests ensure the framework depicted by the table below are met. The framework describes knowledge, actions and responsibilities of each game object and the interaction of the objects.

6.1.1 Testing of bullets class.

To achieve the specifications in the framework, with the use of a parameterised constructor, the test bullet object is initialised with co-ordinates and then tested to ensure co-ordinates of the bullet object remain consistent throughout the lifetime of the object. The initialisation of the co-ordinates with the constructor mimics the instructions given by the cannon class when the bullet and cannon classes are integrated. The manual set function which sets the co-ordinates is also tested to determine the bullet maintains the co-ordinates throughout its lifetime unless instructed by the cannon class.

6.1.2 Testing of cannon class.

Adhering to unit-testing and object-orientated programming the cannon class is tested separately followed by the integration of the cannon class with the bullet class.

6.1.2.1 Individual cannon class

The cannon class has actions of movement and knowledge as an individual entity, see framework and implementation section for in-depth analysis. Similarly, with the bullet class a parameterised constructor is used to create the test object with the specified co-ordinates and checked whether the test object maintains the co-coordinates throughout its lifetime. The cannon class must be able to move in all directions, to test this functionality, after specifying arbitrary initial co-ordinates the cannon is moved in the right, left, up and down directions by utilising its functionality. To verify this test, the speed of the cannon is set and known and therefore, after each movement in a direction the deviation must be a factor of the speed for the test to be succeed.

6.1.2.2 Integrated cannon class with bullet class

The integration of the cannon class with a bullet object to be deemed a success according to the framework stipulated, must be able to hold bullets, shoot, correct movement of the shot bullets according to the game mechanics and the deletion of bullets that go out of scope according to the game specifications.

To test the shooting functionality, a test cannon object is created with default co-ordinates following by shooting of a bullet. To deem the test a success the number of bullets created and present in the container(vector) of bullets must be consistent with the number of times the function is called.

To test the movement of bullets after being shot, a test cannon object is created and shot. Thereafter, the functionality of moving the bullet is invoked. To deem this functionality a success the moved bullet can only increment by the factor of the speed of the bullet object. Furthermore, this functionality must move in a one-dimensional manner according to the mechanics of the game.

Lastly, the deletion of a bullet functionality is tested by deliberately creating a cannon object at the specified boundary and shooting a bullet. To deem this test a success is two-fold, firstly, the bullet container must exhibit a unit decrease once a bullet goes out of scope. Secondly, the final co-ordinates of the bullet must not exceed the dimensions of the boundary.

The integration tests of the cannon class and bullet object serves as the foundation for most of the higher-level implementation, therefore, to prevent violating the DRY principle once this functionality is extensively tested and deemed successful it is not tested again in the higher-level. This epitomises the design properties of a composition-based model.

6.1.3 Testing of gunman class.

The gunman game object as explained in the implementation section is an abstraction of a game object that has intelligence, moves and shoots bullets it encapsulates the idea of an intelligent game object with a cannon. As such, it extensively uses the methods and implementation of a cannon object. Both and player and alien use the gunman class.

Similarly, as other game objects the gunman has actions and responsibilities stipulated in the framework and for the tests to be deemed successful the framework must be adhered to.

Since this object utilises the cannon object which has been tested, some of its functionality is therefore already tested, such as co-ordinates and movement. However, the gunman class has an orientation that a cannon class does not have. To test this functionality, a test gunman object is created by using its parameterised constructor. This constructor then re-assigns the internal cannon object and initialises its orientation. This orientation dictates the shooting direction of the gunman regardless of its position relative to the screen abstraction. Thus, this functionality is deemed a success when the shoot function is called, and the bullet co-ordinates decrease by bullet speed when orientation is up and increase when going down. This functionality is crucial in created a flexible design as both alien and player classes contain a gunman object.

The gunman class also has the responsibility of moving the internal cannon object in the right direction while not exceeding the screen boundary limits. For this functionality the mechanics of movement are not tested as that has already been tested in the cannon class, however, the range mechanism is tested. To test this, a gunman object is created thereafter, it is continuously m oved until the number of iterations of movement exceeds the screen boundaries. The co-ordinates of the gunman object are then subsequently checked and should not exceed the screen boundary limits.

6.1.4 Testing of player class.

The player class uses the gunman class as an internal object according to the design model and game framework. As such most of the functionality has already been implemented and tested in the gunman object.

The tested implementation of the player class is the initial position. The initial position is either top and on the middle of the screen or bottom and on the middle of the screen. To test this a tests player object is created and is initialised to the top. The co-ordinates of the top, middle location of the screen are determined empirically and used to verify the correctness with the player’s co-ordinates. This process is repeated for the location on the bottom of the screen. All movement and shooting functionality have been tested in the lower-level classes.

6.1.5 Testing of alien class.

The alien class is at the same level with the player class in the design model of the game, therefore the two classes have similarities and correlation. All movement and shooting functions are tested in the lower-level classes.

The test of this class is testing the creation of the alien test object.

6.1.6 Testing of army class.

To test if the correct number of aliens is present in an army, a test army object is created and the number of aliens present is checked against the known correct amount. Secondly, the removal of an alien is checked by removing an alien and then checking the number of remaining aliens in the army. Lastly, the index of the shooting alien tested to ensure that only an alien present within the army can shoot.

7. Critical analysis and future improvements

The designed game meets the specifications and requirements to be deemed a success. A critical analysis of the final design is presented with possible future improvements.

7.1 Information hiding

The design model allows for a flexible design and meets the requirements. However, due to the nature of the design, functionality of the imbedded objects deep in hierarchy structure are called by higher-level objects in consecutive order. An example is obtaining the co-ordinates of the bullet object. To achieve this, the function would be *playerObject.gunman().bullet-x-co-ordinate().* This is implemented in this manner to prevent the violation of the DRY principle, however it reveals the implementation of the class the client does not need to know. It reveals the secrets of the class. To solve this, encapsulation functions can be created in each class and in all levels of the hierarchy structure and thus the client code only uses the immediate functions of the class being used. This however creates redundancies and large classes that are big to understand, debug and are prone to errors and breakdown.

7.2 Further breakdown of the presentation layer

The presentation layer is separated from the other layers and the principle of separation of concern is met. However, this is implemented in one class called *Screen*. This creates a large class that contains extensive implementation of SFML. This leads to an error prone class that needs constant debugging. An improvement on this is to further breakdown this layer by modularizing the *Screen* class into smaller class that handle specific methods and responsibilities

7.3 Improvement of functionality

The designed game meets the functionality criteria that is stipulated by the brief, that is basic functionality, two minor and two major features. However, the functionality can be further improved such as implementing a scoring system for the user, multiple lives for the players, increase in difficulty as the game progresses and special weapons for the players as they progress through the stages. A design trade-off between functionality and complexity was taken and the implementation of this functionality is not implemented for this design

7.4 Graphical User Interface (GUI) Testing

The GUI is important in achieving a wholistic final solution. The designed game exhibits good graphics however, these graphics were tested experimentally and not using an extensive testing framework such as Doctest. This does not ensure longevity of the GUI and maintainability and may cause error further down the design pipeline.

8. Conclusion

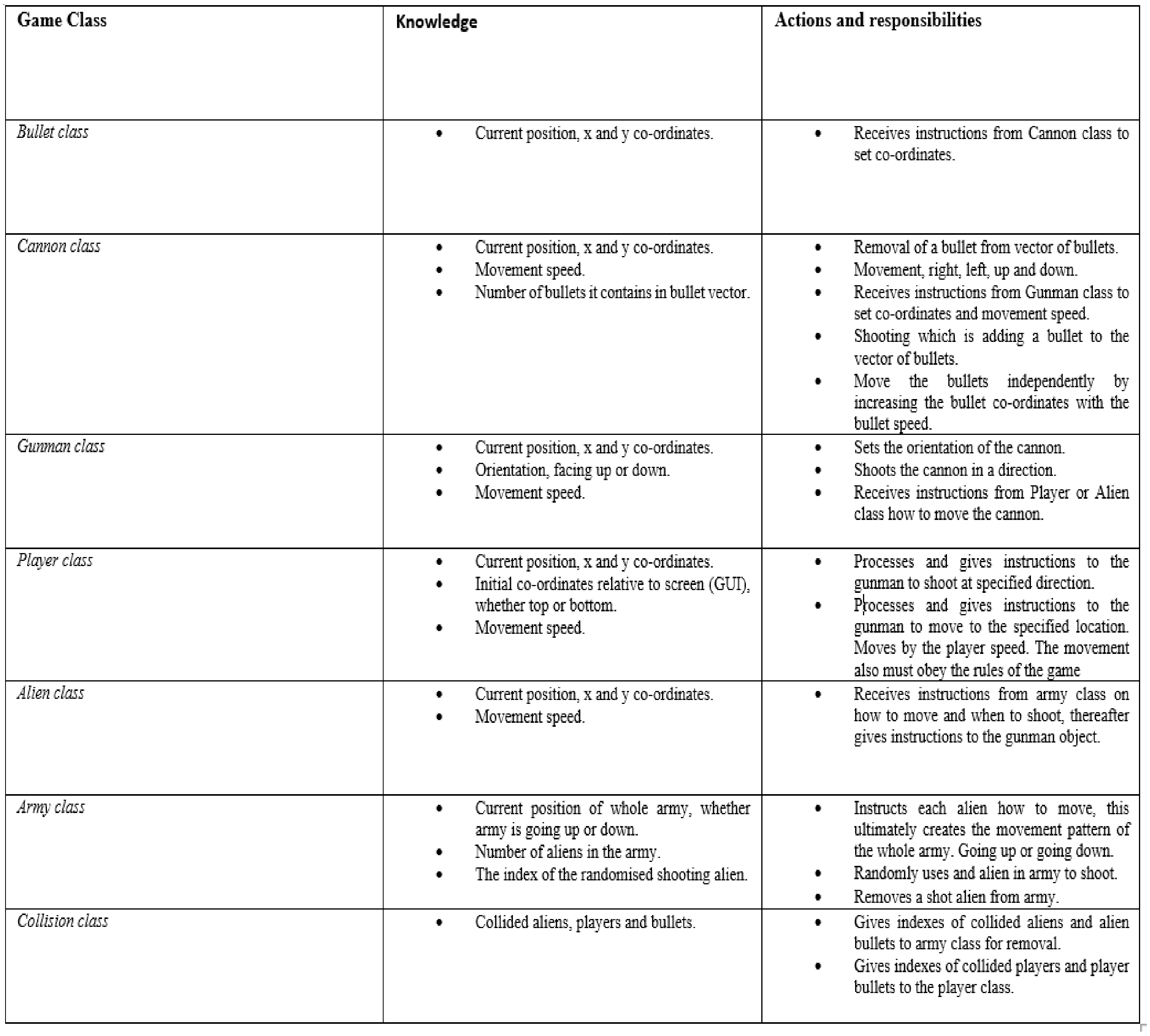
The game meets the requirements and success criteria and therefore deemed a success. The game contains the basic functionality, two minor features and two major features. The design model accurately models the problem while the game framework provides a quality assurance mechanism that ensures a quality product. Furthermore, the performance of the game exhibits no indication of major flaws or errors that compromise the design of the game. Improvements and optimisations are presented that suggest solutions to improve the designed game.

References

[1] S. Parkin, "The Space Invader," 17 October 2013. [Online]. Available: https://www.newyorker.com/tech/annals-of-technology/the-space-invader. [Accessed 27 September 2019].

Appendix A

Table 1: Designed game framework



Game Class	Knowledge	Actions and responsibilities
Bullet class	• Current position, x and y co-ordinates.	• Receives instructions from Cannon class to set co-ordinates.
Cannon class	• Current position, x and y co-ordinates. • Movement speed. • Number of bullets it contains in bullet vector.	• Removal of a bullet from vector of bullets. • Movement, right, left, up and down. • Receives instructions from Gunman class to set co-ordinates and movement speed. • Shooting which is adding a bullet to the vector of bullets. • Move the bullets independently by increasing the bullet co-ordinates with the bullet speed.
Gunman class	• Current position, x and y co-ordinates. • Orientation, facing up or down. • Movement speed.	• Sets the orientation of the cannon. • Shoots the cannon in a direction. • Receives instructions from Player or Alien class how to move the cannon.
Player class	• Current position, x and y co-ordinates. • Initial co-ordinates relative to screen (GUI), whether top or bottom. • Movement speed.	• Processes and gives instructions to the gunman to shoot at specified direction. • Processes and gives instructions to the gunman to move to the specified location. Moves by the player speed. The movement also must obey the rules of the game
Alien class	• Current position, x and y co-ordinates. • Movement speed.	• Receives instructions from army class on how to move and when to shoot, thereafter gives instructions to the gunman object.
Army class	• Current position of whole army, whether army is going up or down. • Number of aliens in the army. • The index of the randomised shooting alien.	• Instructs each alien how to move, this ultimately creates the movement pattern of the whole army. Going up or going down. • Randomly uses and alien in army to shoot. • Removes a shot alien from army.
Collision class	• Collided aliens, players and bullets.	• Gives indexes of collided aliens and alien bullets to army class for removal. • Gives indexes of collided players and player bullets to the player class.

Appendix B

Group Peer Assessment

Documented below is a group assessment of how the project was delegated among members as well as member feedback in each assessment field.

Table 1: Allocated project components for each member in the group

Member Name	Assigned Project Component
Sinawo Dlulisa	Report write up, front -end game design, code implementation and testing, technical reference manual.
Fortune Iga	Report write up, front -end game design, code implementation and testing.

Table 2: Group member feedback for required fields of assessment

Quality	Sinawo Dlulisa	Fortune Iga
Leadership	5	5
Depth of knowledge and understanding	5	5
Participation and contribution in group discussions	5	5
Responsibility and accountability	5	5
Final average score (out of 10)	5	5

Decrees of Approval:

[illegible] [illegible]

Fortune Iga Sinawo Dlulisa