DESIGN AND IMPLEMENTATION OF DEFENDER:FINAL BATTLE

Sheena Philip 545819, Moolisa Tlali 564988

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

Abstract: A "defender-like" arcade game using object oriented principles is created in C++ with the addition of the SFML as a graphics library. The aim of the game is for the SpaceFighter to shoot down all the Landers. A SpaceFighter is able to move horizontally and vertically and only shoot horizontally. Landers teleport onto the screen at various times and shoot directly at the SpaceFighter. The implementation of the game meets all the basic criteria. It is robust and efficient since all unit testing is passed. Each key entity has a class created around it which forms the backbone of the design. This allows for encapsulation and protection of functionality. A strong point of the design is the collition class which is adaptable to any rectangular objects that are passed into it. This type of class allows for easy expansion and change of the game.Seperation between the presentation and logic layer is not completely successful. Whilst SFML performs none of the actual logic, classes combine the two layers into one in order to mimic changes in position of the various entities.There is redundancy within the move funcions of the various classes and the Game class, which controls the whole game is considered monolothic. Inheritance and better sepeartion of concerns could help improve the game in the future.

Key words: Object orientated Programing(OOP)

1. Introduction

This report details the process of designing and implementing a "Defender-like" arcade game using an object orientated solution in C++. The game has to make use of Simple and Fast Multimedia Library(SFML Version 2.1) for grapahics and user input handling purposes. The Game involves a Spacefighter shooting down all the Landers that teleport around it, without being shot down by a Lander.

Section 2 discusses the project specifications, assumptions and background to conceptualise the problem. Section 3 discusses key abstraction and domain modelling. Section 4 describes the design of the game on a micro and macro level. Section 5 critically analyses the design implemented and Section 6 provides future improvements.

2. Project Specifications

The requirement of the project is to implement an object oriented program using C++ and SFML graphics library. The solution must mirror a two dimensional Spacefighter battle and adhere to all of the basic functionality as stated in Section 5.1 of the project brief. Unit testing must be performed on the code to test it's accuracy. Furthermore a technical reference manual generated using Doxygen, a User Manual and Test Report must be included.

2.1 Project Constraints

- The maximum resolution of the screen is constrained to 1600 by 900 pixels
- The graphics library is limited to SFML
- The game has to be coded in C++
- The game must be able to run on a Windows platform
- The time to complete the project of all the versions is 7 weeks

2.2 Project Assumptions

To arrive at simplified solution, it is assumed that the player will control the game with a standard keyboard, as the solution cannot intepret input from a mouse or a joystick. It is also assumed that all the entities of the game are rectangular in shape.

2.3 Success Criteria

The Defender: Final battle is considered a success if it is designed and implemented using an (OOP), with separation between logic and presentation layers. The game must also meets all basic functionality as specified in Section 5.1 of the project brief. An acceptable technical reference manual, unit testing, user manual and version history should compliment the game.

2.4 Existing Solutions

The original Defender was released by Williams Electonics in 1981. It was seen as a huge breakthrough in video games as it's audio visuals and its interactive gameplay experience were merrited by critics. The original game is set on an alien planet where the player navigates a spaceship that can either fly left or right.The spaceship must destroy all aliens that come towards it, whilst protecting the astronauts from being abducted [1]. Several remakes and sequels were created such as Defender 2000, Defender 2002 etc which mainly involved advances in graphics from 2D to 3D and a third person view point. The implemented solution is a simplified version of the original Defender with scroll capabilities, confined space and the number of entities is reduced to four.

3. Key abstraction and domain modelling

The game entities that have to exist include the player's ship,the player's laser, Landers and Lander missiles. It was therefore decided to model each of these entities as a class in order to encapsulate all the data and functionality each object brings to the game. The responsibilities of each object are given in Table 1. These four game objects became the fundamental building blocks of the design and were the basis of the object orientated design solution.

Table 1 : Game objects and their responsibilities

Object	**Responsibilities**
SpaceFighter	Controlled by player through keyboard
	Moves vertically and horizontally
	Can shoot horizontally
Lander	Sets position of Lander on screen
	Moves vertically,horizontally and diagonally
	Shoots vertically,horizontally and diagonally towards spaceship
Laser	Sets position of laser to Spaceship position
	Sets direction of laser to Spaceship direction
	Moves horizontally
Missile	Sets position of Missile to position of Lander
	Determines path between Lander and Spaceship and moves on that path

The design model is centred around the entities of the games as pillers/building blocks of the game. The Game class is then set up as a link and control class which utilises "worker" classes (Coliisions, detect collisions and create) which perform higher level functionality by making use of the entities.

4. Design

The design of the game is discussed below. The individual classes are first described and then an overview is performed to analyse the relationships between the various classes

4.1 Class descriptions

The following describe individual class responsibilities.

4.1.1 SpaceFighter,Lander,Laser and Missile Class: The four classes, SpaceFighter, Lander, Laser and Missile, share very similar responsibilities. The main objective of each of the classes is to manage their respective object. In SpaceFighter class the constructor initialises the SpaceFighter's object to the position (750,400), whilst in the other three classes the objects are initiated to an X and Y value that are passed as arguments into the constructor. In the case of the Missile and Laser, the X and Y co-ordinate will be equivalent to the Lander and SpaceFighter's X and Y coordinate respectfully.

Each Class has a move function. The move function takes in an argument that describes how the object must move. It then changes the X and Y coordinate respectively by adding or subtracting a set value (speed) from each value. The Missile class has additional functionality in its move function, since a automated target based shooting system has been implemented. The Missile's move function uses the Lander Position and the SpaceFighter position to determine a straight path between the two entities. This involves calculating the gradient between the two points. The X,Y or both co-ordinates are then multiplied by the gradient and the speed depending on how the object must move.

In addition each class also has a function that returns the updated position to the Game class, so that the corresponding sprite can replicate that movement.

4.1.2 Create: The responsibility of class create is to create Lander, Laser and Missile objects and sprite objects and populate their respective vectors. The creation of these entities is time limited.

4.1.3 Keyboard: The responsibility of this class is to poll for when keys on the keyboard have been pressed and when they have been released. If a certain key is pressed, a boolean variable is set and returned to Game. It also polls for when the window has been closed and actually closes the window.

4.1.4 Display: The purpose of this class is to display all the SpaceFighter,Lander,Laser and Missile sprites on the screen.

4.1.5 Detect Collision: This class acts as the intermediary class between Game and Collision. Its' responsibility is to run through and match all the objects that need to be passed into the class Collision. This involves the following combinations:Lander and SpaceFighter ,Missile and SpaceFighter and Lander and Laser. If a collision is returned for the first and second combination, a function sets the game status to finished. If a collision is returned for the third combination, a function deletes the associated objects and sets the status of the game to finished if all the Landers have been destroyed.

4.1.6 Collision: The purpose of this class is to actually detect whether a collision between two objects has occurred. It assumes each entity as rectangular in shape and therefore each entity has a width and a height. The advantage of this is that the function is adaptable for any combination of objects. The algorithm is based on detecting whether any of the edges of the entities have overlapped.

4.1.7 Common: The common class acts as a class which stores all the variables that are common between various classes. This includes the screen size values (the boundaries),sprite size values for various entitities and all ennumeration declarations that are used.This gives the developer the ability to manipulate variables that are constant throughout the framework in one class to suit their prefered gaming experience.

4.1.8 Game class: The Game class is the central class to the whole design and acts as the control centre for the whole game. Other than calling and linking all the various Classes, its main functionality is to update the SpaceFighter,Landers,Missiles and Laser.

4.2 Relationship between classes

The relationship between classes form the most important part of the design aesthetic of a game. It is even more important in Object-orientated programming for the purposes of separating logic from presentation.The design of the game is as follows:

The Game class is what controls the entire game and in essence represents one run through of the game (game loop).

The game loop is as follows: The window is set up when an object of the Game class is created. The SpaceFighter Sprite and object are then created. The main loop in then initiated at this point. Game calls the class keyboard which polls for keys on the keyboard being pressed. If any keys have been pressed, respective boolean variables are set(which are used later in the gameloop).

As part of the design, the Landers have to appear at various times on the screen. Game therefore checks whether a minimum time for Landers to be created has passed. If it has, Game calls the create class which creates a Lander object and corresponding Sprite, otherwise it continues in the game loop as normal. Game then calls Create class and passes to the class whether the space-bar(which indicates shooting) has been pressed. If the boolean is true, Create class creates a Laser and corresponding laser sprite.

Game then calls the SpaceFighter class, which updates the position of the SpaceFighter object depending on what movement boolean is set. It then gets the updated position from the SpaceFighter class, and sets the corresponding sprite to the value retrieved. Part of the design is that Landers have to move automatically on the screen. To achieve this, Game randomly generates a number which corresponds to a movement direction. Game then calls the Lander class and passes through the random number as an argument. The Lander class updates the position of the Lander object respectively. Game then retrieves this updated position from the Lander class and sets the corresponding sprite to the value retrieved. Game then calls the Laser class ,which updates the position of the Laser in whatever direction the SpaceFighter was in when the Laser was created. game then retrieves the updated position from the laser class and sets the corresponding sprite to the value.

The Lander has to automatically shoot at the SpaceFighter. In order for the SpaceFighter to have a chance of winning the game, the Lander only shoots at the SpaceFighter after a set time period. Game checks whether this interval has passed. If it has, it calls the Create class to create a Missile object and corresponding sprite. Game then calls the Missile class to update the position(if any have been created) of the Missile. If the Missile movement causes the Missile to move past the screen boundaries, the object is destroyed. Game then retrieves the updated position from the Missile class and sets the corresponding sprite to the value.

The Game class then calls the Detect Collision class which then calls the Collision class. In Detect collision class, if a collision is returned from the Collision class between a Lander and the SpaceFighter or the Missile and the SpaceFighter, the Display class is called, which will end the game. If a collision is returned between a Lander and a Laser, the Lander object is deleted. If all the 15 Landers are deleted, the Display class is called to end the game, otherwise Display is called to draw all the sprites onto the screen.The game loop is then run through again. This design loop can be seen in Figure 2,Appendix A

The dynamic relationships between the main classes are shown in Figure 1,Appendix A.

4.3 Distribution of classes between layers

The three-layer architecture is a software methodology of dividing the design into 3 layers:presentation,logic and data.The presentation layer is composed of receiving user input and displaying information onto the screen. The logic layer is composed of the manipulation of variables and the decision making of the design.

The data layer is an information storage layer.

For the purposes of this design the data layer is omitted. The main source of information is from the keys being pressed. Also only the current position of every object is saved (not previous positions). Therefore large amounts of data are not being stored and so the data layer is neglected.

Table 2 shows how all the classes in the design are separated between the logic and presentation layers.

Table 2 : Sepeartion of the classes between layers

Presentation	**Logic**	**Both**
Display	Collision	Create
Keyboard	Landers	DetectCollision
	Laser	Game
	Missiles	
	SpaceFighter	
	Common	

4.4 Critical Evaluation

A critical analysis of the design and its implementation is performed on the game. One of the positives that were noted is that all the basic functionality was achieved and when it was tested it was found that it performed as expected, indicating that the logic is robust.

A strong advantage of the implemented design resides in collisions class. The collision function of this class is not constrained to a specific case or entity of the game as long as the entity is rectangular in shape. This means the collision function can take in any two objects positions and determine whether a collision has occurred or not. The advantage of this is that it is a compact design, it is precise and if additions or changes are to be made to the game (for example the numbers of entities are expanded), this function would not have to be altered as the function is adaptable in nature.

Another advantage of the design is that objects are only created when they are needed, stored in vectors and erased from these vectors when they go out of scope. This dynamic use of memory makes the code efficient.

In addition, defensive coding is employed as all constants in the game are made static so that the client code cannot alter their values. Strongly typed enumerations are also used to ensure that the entities do not move in a direction that is not predefined. The use of structs also add an adavantage of a single variable that can be used to hold two variables. This reduces the number of parameters that are passed in a function.

One thing that was noted was that there is a good separation of the key game objects and their implementation. It is observed that if new data or functionality is to be added to one of the key objects this would not affect any of the other object classes. This highlights the fact that encapsulation was performed successfully regarding the main object classes. It is also noted that the key object classes have a level of privacy since the Game class could tell the key object classes to move but it is unaware of how this movement is updated.

A weakness in the design is that SFML and the logic layer are not completely separated. Although three classes contain a mixture of SFML and logic, the SFML in essence does not perform any logical operations. The main reasons why these two overlap are because firstly the sprites positions are set to the objects position. Secondly if a collision is detected, the object and a corresponding sprite are also deleted. The problem with not having complete separation is that it is not be easy to exchange the SFML library for another graphics library. On the contrary, the Keyboard and Display class are completely SFML based and purely a presentation layer class. It therfore adds significantly to the separation of concerns.

A big drawback of the design is that there is redundancy in the move function of the Lander, SpaceFighter, Laser and Missiles classes. Inheritance should have been implemented where a parent class containing virtual move functions can be implemented and the main object classes can derive the necessary move functions.

While the Game class represents the game loop, it is performing too many responsibilities thus rendering it a monolithic class. A monolithic class is difficult to understand, maintain and to integrate with new code. Althogh functions have been used to separate concerns, this class is relatively long in comparison to the other classes.

According to the success criteria as mentioned in Section 2.3, the project is deemed a partial sucess. Basic functionality was achieved with an object oriented approach however complete separation between the presentation and logic layer was not achieved. A technical reference manual, version history, unit testing and test report successfully contribute to the overall understanding of the design.

5. Future Improvements

In future versions of the game, the first improvement would be to implement inheritance and polymorphism as previously mensioned. This will reduce redundancy and duplicated code as this will capture code for common functionality features such as the movement function. In effect, a parent class for all movable objects can be created from which all movable entities can be derived.

The length and responsibilities of the Game class can be reduced by creating separate classes that handle a specific functionality and ties together related behaviours. For example, one of the biggest responsibilities of the Game class is to update the state of all the main game objects. A separate class Update can be created to handle this functionality. This will not only reduce the size of Game but it will also add a level of abstraction and further encapsulation.

Moreover, an additional class can be created that handles the updates of the sprites. The advantages of this class are that firstly, it will completely separate SFML from the logic layer. Secondly, the solution will ensure that the source code will not have to be changed should a developer decide to use a different graphics library. Only Keyboard, Display and the new function that handles sprites will have to be altered.

A possible improvement to the design is to use pointers to create objects of the major classes. This implementation will create these objects on dynamic hip instead of on the stack. This implementation tradesoff a fast access of the objects for no limit on memory size. Additional features that can be included are a welcome screen, scoring, abillity for a rolling screen, addition of entities such as smart bombs, bombers, humanoids, minimap, and power-ups.

6. Conclusion

A "Defender-like" arcade game has been designed, implemented and tested. An object oriented programming solution using C++ and SFML graphics library has been produced.The basic functionality as mentioned in the project brief has been achieved. The seperation of logic and presentation was not completely successful. The project is therefore deemed a partial success. Future improvements such as inheritance,separating concerns in big classes should be implemented.

REFERENCES

[1] Williams.[Online].1980.[Accessed 4 October 2015]. Avaliable from: https://archive.org/details/arcade$_defender$

Appendix A: DIAGRAMS

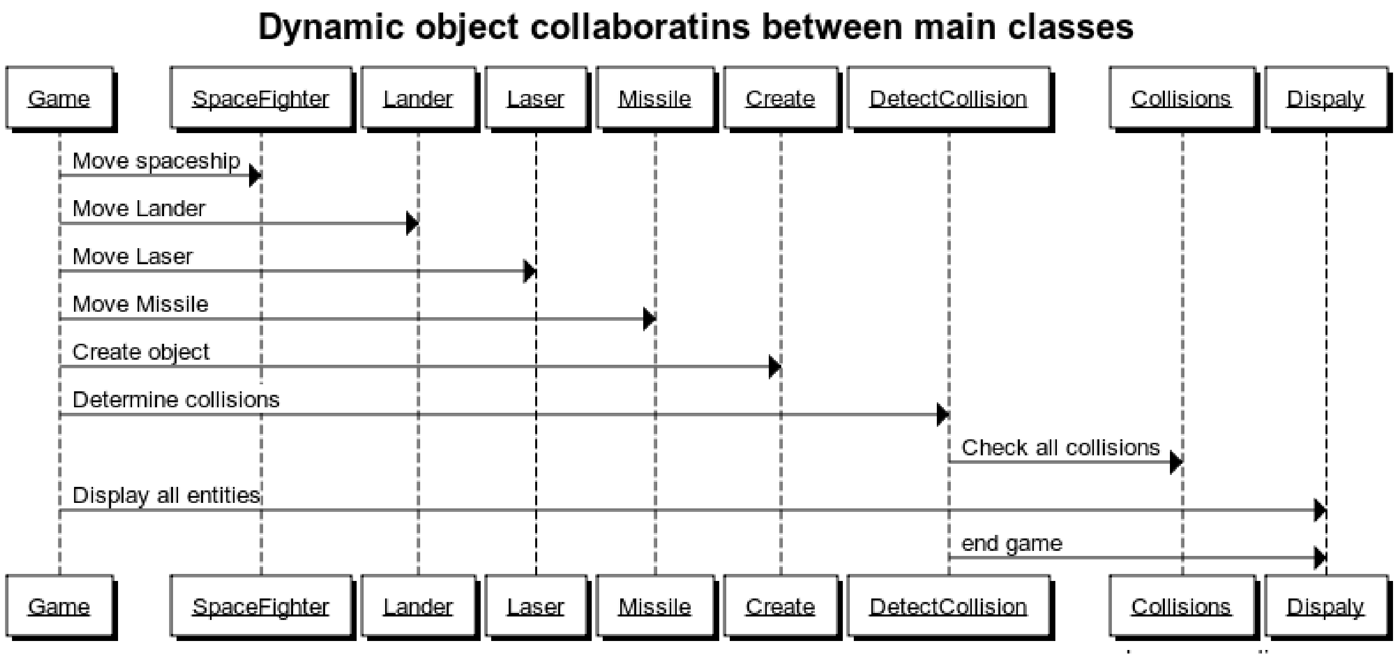
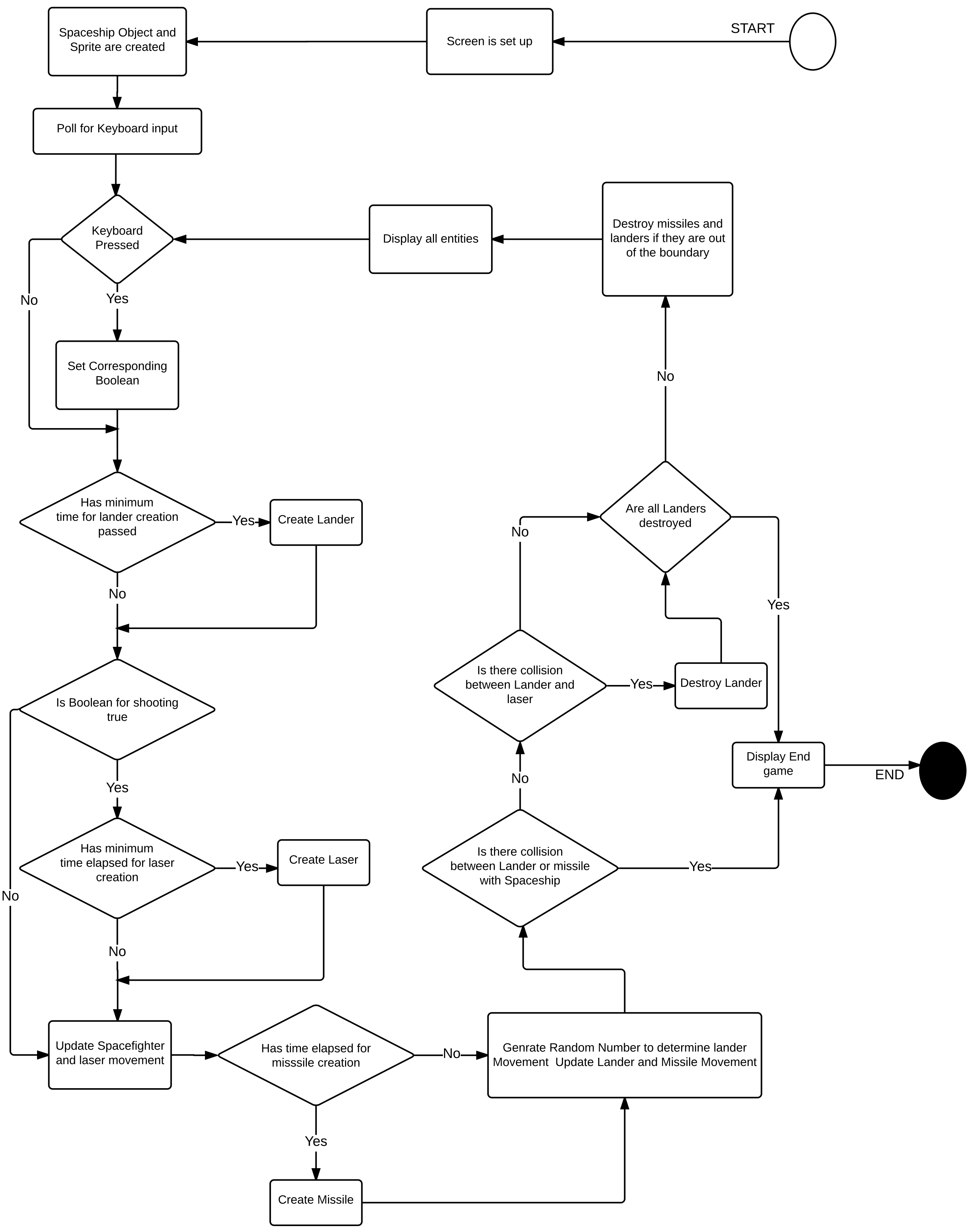


Figure 1 : Sequence diagram showing dynamic relationships between main classes

Flowchart(1).pdf



START
Screen is set up
Spaceship Object and Sprite are created
Poll for Keyboard input
Keyboard Pressed
No
Yes
Set Corresponding Boolean
Has minimum time for lander creation passed
Yes
Create Lander
No
Is Boolean for shooting true
Yes
Has minimum time elapsed for laser creation
Yes
Create Laser
No
No
Update Spacefighter and laser movement
Has time elapsed for misssile creation
No
Yes
Create Missile
Genrate Random Number to determine lander Movement Update Lander and Missile Movement
Is there collision between Lander or missile with Spaceship
Yes
No
Is there collision between Lander and laser
Yes
Destroy Lander
No
Are all Landers destroyed
Yes
No
Destroy missiles and landers if they are out of the boundary
Display all entities
Display End game
END

Figure 2 : Representation of a complete single run through the game loop