The Design, Implementation and Analysis of an Arcade Game Coded in C++

ELEN3009 - Software Development II
Kavilan Nair (1076342) & Iordan Tchaparov (1068874)
17/10/2017

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

Abstract: This document provides an in-depth explanation to the design, implementation and analysis of Cosmic Journey, a 2D arcade game that is based on the popular "*Gyruss*". The design is aimed to specifically separate concerns, otherwise known as layering, through the use of object-oriented programming, inheritance with polymorphism, and composition. The inheritance was implemented using two pure interface classes as the base classes for all game objects. The game is implemented in C++14 with the presentation being done through the use of Simple and Fast Multimedia Library (SFML). The project meets all basic functionality and contains 3 minor and 2 major features. The implementation of the game does feature some flaws such as an inefficient collision detection algorithm and a few violations of the Don't Repeat Yourself (DRY) principle. Overall, Cosmic Journey is an enjoyable game that conforms to many good coding practices such as an object-oriented design, encapsulation of data, separation of layers resulting in a code base that is easy to maintain and expand on functionality.

Key words: Object-Oriented, Separation of Concerns, Doctest, C++14, SFML

1. INTRODUCTION

This report documents the design, implementation and analysis of an object-oriented software program that was coded in C++14. The program is based off the incredibly popular arcade game *Gyruss* which was released in 1983 and this is from where it draws inspiration for its functionality. The sections in this report deal with the background information to the project, the design of the structure of the code, implementation of the structure, the behaviour of the implemented code, testing of the code and the critical evaluation of its design with recommended improvements which can be incorporated for a more successful design.

2. BACKGROUND INFORMATION

The project is first contextualized with respect to the game mechanics, requirements, success criteria, and constraints and assumptions.

2.1 Game Mechanics

Cosmic Journey is a fixed shooter game coded in C++14 that is based on the popular arcade game *Gyruss* and is rendered in a window with a resolution of 800x600 pixels. The game has a space theme and features a player controlled ship whose goal is to destroy a set amount of enemies within a life limit and while attempting to achieve the highest score possible. The player ship has movement in either a clockwise or anti-clockwise direction on the screen and can fire bullets that travel towards the center of the screen. There are different entities in the game such as normal enemy spaceships, satellites, laser generators and asteroids, each spawning in the center and then moving outwards. The normal enemies and satellites shoot out at the player ship while the laser generators have a laser arc in between them that kills the player if they collide, much like the asteroids.

2.2 Requirements

The following basic functionality is required:

- The movement of the player ship is controlled by user input and it is restricted to moving in a circular fashion on the game screen.
- The player can fire bullets towards the center which is controlled by user input. Enemies that collide with the bullets will be destroyed.
- The enemies spawn from the center of the circle and move radially outwards, firing bullets at the player. The player is destroyed if these bullets or enemies collide with them.
- Enemies that reach the circumference of the circle need to be re-spawned in the center and to move along a new radial path.
- The game ends when all the enemies are killed by the player or the player is killed.

The following enhanced functionality can be included:

- There are invincible asteroids that spawn in the center and quickly move out radially, whereby the player will be destroyed if they collide.
- A scoring system is incorporated which has both the current user score and the all time high-score.
- The player has three lives to destroy all enemies. A life is lost when the player collides with any enemies or enemy bullets.
- Satellite enemies spawn in groups of three which gyrate and shoot at the player. Killing all three upgrades the player bullets to shoot two at a time.
- Laser generator groups spawn which have arcs of laser fields, destroying the player if touched.

2.3 *Success Criteria*

For this project to be considered a success, all the basic and enhanced functionality listed in section 2.2 must be implemented in C++14 using good programming practices. The game must must make use of object-oriented structures and inheritance while conforming to the constraints of section 2.4. The game developed should work with Simple and Fast Multimedia Library (SFML) to present sprites representing the different entities in the game. It should run smoothly and fairly, giving each user the same experience and opportunity to win the game regardless of computer specifications. The classes that make up the game must also be tested to establish that they are functioning correctly.

2.4 *Constraints & Assumptions*

The game display is limited to a resolution of 1920x1080 pixels (but can be smaller) and OpenGL may not be used. In addition, no libraries built on top of SFML may be used and SFML version 2.3.2 or higher must be used. All user input will come form keyboard strokes and not come from either a controller or joystick.

3. CODE STRUCTURE

The following introduces the different types of structures and concepts that were investigated to be implemented in the project.

3.1 *Design Architecture*

The requirement that the project needed to have an object-oriented implementation heavily influenced the structure of the code. To adhere to this requirement, all game entities and their behaviours are listed in Table 1 alongside. It illustrates that there are many common behaviours between the entities. This allows objects to be characterized by entity type, position, size, movement and shooting. An object oriented approach that involves inheritance is therefore suitable for the game. The entities are split into entities that can move and entities that can move and shoot. Two pure interface classes were designed as the base class for all entities namely `IMovingEntity` and `IShootingMovingEntity`. All moving game entities and `IShootingMovingEntity` inherit from `IMovingEntity` while all entities that can shoot inherit from `IShootingMovingEntity`. The design of the inheritance hierarchy to be implemented is shown in Figure 4 in Appendix A. Interface inheritance was deemed more favourable over implementation inheritance as it avoids tight coupling between the base and derived classes and allows for the derived classes to implement their own unique shooting and movement functions.

Table 1 : Entities and their behaviours

Entity	**Behaviour**
Player	Moves circularly around the circumference of a circle. Shoots Player Bullets.
Enemy	Moves radially outwards from center of circle to circumference. Shoots Enemy Bullets.
Satellite	Moves radially outwards. Gyrates. Shoots Enemy Bullets.
Laser Generator	Moves radially outwards from center of circle. Shoots Laser Fields.
Player Bullet	Moves radially inwards from Player position to circle center.
Enemy Bullet	Moves radially outwards from Enemy position to circle circumference.
Laser Field	Moves outwards from center of circle. Oscillates between three positions on each frame.
Asteroid	Moves radially outwards from center to circumference of circle.

3.2 *Separation of Concerns*

Separation of concerns is a design principle that involves keeping the presentation, logic and data layers separate. The presentation layer displays the Graphical User Interface (GUI) and is responsible for interacting with the user and capturing input. The logic layer handles the functionality of the application which includes calculations, logical decisions and data processing. The data layer allows the software to access resources such as images or text-files. Enumeration data types are not attributed to any specific layer.

The benefits of Separation of Concerns [1] include the following:

- A well implemented design allows multiple people to work on separate aspects of the program.
- Designing the logic layer to be independent of the presentation layer allows the same logic layer code to support various multimedia libraries.
- Allows for easier porting of the software between platforms as the logic layer can be ported and another platform specific presentation layer be used.
- Decoupling the layers allows for independent implementation and testing which results in a high quality code base that is maintainable, extensible and scalable.

Even though the three layers will be separate, they require a connection to send and receive data be-

tween the layers. There are many design patterns that are used to separate layers such as the Model-View-Controller (MVC) and the Model-View-Presenter (MVP) [2]. The *Model* is the interface to the data that the program needs access to. The *View* is responsible for displaying the content to the user. The *Presenter* acts as an intermediary that allows for the decoupling of the layers. All the logical decisions and overall control is implemented in the *Presenter*.

The separation of concerns implemented in this project was inspired by the MVP design pattern where the Logic class represents the *Presenter* and the Presentation class represents the *View*. The `ResourceManager` and the `HighScoreManager` classes represent the data layer and represent the *Model*. Figure 1 illustrates how the Separation of Layers was implemented.

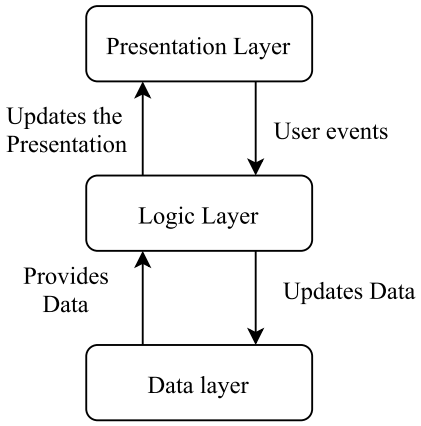

Figure 1 : Separation of Concerns

4. IMPLEMENTATION

The game has been shown that it can be divided up into three layers in section 3. Each class can be classified by one of these three layers. To simplify this section, the enumeration classes will be listed under the data layer even though they do not belong to this layer.

4.1 Data Layer

The data layer is described as any part of a program that gives some form of indication of a state or an input that can be transformed by logic to control another part of the program. This layer has the following classes:

4.1.1 ResourceManager Class: A class that maps the file path on the disk drive of relevant textures to an entity type. This allows mapping of the texture of to a sprite in the presentation layer based on a specific entity type. Textures are in the PNG format.

4.1.2 HighScoreManager Class: This class has the ability to read in the high-score from a text-file and write the new high-score to the file. This class is controlled by the logic layer, which then provides the presentation layer with the high-score data to display.

4.1.3 Direction Class: A strongly typed enumeration class that contains the different types of movement for the player: clockwise, anti-clockwise and hover. The hover direction means the player ship must stay stationary on the screen.

4.1.4 EntityType Class: A strongly typed enumeration class that contains the different types of entity objects in the game, such as the player, enemy, satellite, laser generator, player bullet, enemy bullet, laser field and asteroid.

4.1.5 GameState Class: A strongly typed enumeration class that contains the different states the game can have. This includes a state for presenting the splash screen, when the game is active and running, when the game has been lost and when the game has been won.

4.1.6 WeaponType Class: A strongly typed enumeration class that contains the different weapon types the player can have. There are only two types, single and double for one or two bullets fired at a time respectively.

4.1.7 Position Class: A class that defines the current location, starting location, angle and radius of every entity. This data class allows many objects to create an instance of their position instead of repeatedly defining the same variables and getter and setter functions in each of the classes.

4.1.8 Grid Class: A class that defines the size of the grid where the game will be played. By giving the constructor the width and height, the grid defines the center and the radius of the circle, storing the parameters so other objects can access them to determine their movement speed as a factor multiplied with the radius.

4.2 Logic Layer

The logic layer is described as the part of the program that does the processing, makes the decisions and tells other components of the program what to do and when to do it. This layer sends data and information to where it needs to go (usually presentation) by using the data layer (and sometimes the presentation layer in terms of user input), acting as a sort of “controller”. The following are the logic classes:

4.2.1 Asteroid Class: This class inherits from `IMovingEntity` and represents an asteroid that spawns at a random angle and moves radially outwards from the center of the screen. It has the property of being invincible, only being destroyed once it has reached the circumference of the circle. Colliding with the player results in the player being destroyed.

4.2.2 CollisionHandler Class: This class is responsible for handling all the collision detection between different entities in the program. By finding a matching pair of entities that have collided, it sets both objects as dead, which means they have been destroyed but still exist within a vector that is controlled by `Logic`. A more detailed explanation about the implementation is available in 5.3.

4.2.3 Enemy Class: This class inherits from the `IShootingMovingEntity` and represents the normal enemies within the game. The enemies spawn in the center of the grid and generate a random angle to travel along outward. Colliding with the player will cause both objects to be destroyed. Enemies who exit the circumference of the circle are destroyed but flagged for re-spawn, so that `EnemySpawner` can know to create another enemy for the one that has just passed through the circumference. Enemies can shoot `EnemyBullet` objects in front of them at faster speeds than that at which they travel.

4.2.4 EnemyBullet Class: This class inherits from `IMovingEntity` and represents a projectile that both normal enemies and satellites can shoot. A collision with the player will result in the object being destroyed and the player losing a life, causing them to respawn.

4.2.5 EnemySpawner Class: This class is used to control the spawn rate of all non-user entity objects as well as when enemies and satellites can shoot or not. In addition, the class also checks whether a satellite group or a laser generator group is currently spawned, and if one is, it will not spawn another until the existing one dies. It also performs checks for how many normal enemies need to be re-spawned and spawns the correct amount.

4.2.6 IMovingEntity Class: A pure interface class which serves as the base class of the inheritance hierarchy. It has pure virtual functions that allow the object to get its position and entity type, to move, check the alive status, set object as dead, re-spawn, and retrieve the hit radius of the object.

4.2.7 IShootingMovingEntity Class: Another pure interface class that inherits from the `IMovingEntity` class and gives it the added functionality of being able to shoot in addition to the inherited functions.

4.2.8 LaserField Class: This class inherits from `IMovingEntity` class and is used to represent the laser arc in between laser generators. This object moves radially outwards from the center of the grid while also changing its angle by 4°, making it oscillate between frames, creating an illusion of a moving laser field between the two laser generators. It dies once it reaches the circumference of the circle, killing the player if it collides with them.

4.2.9 LaserGenerator Class: This class inherits from `IShootingMovingEntity` and represents one of the two sides of the laser generator group. Either side has the same functionality as the opposite, only working at a different angle. This shoots `LaserField` objects that acts as the laser arc in between the generators. It dies at the circumference or when it collides with either player or player bullet.

4.2.10 Logic Class: This class serves as the controller of the game functioning and as the intermediary between the data and presentation layers. It is responsible for combining the logic of all the other classes by calling the relevant functions outside of itself in one loop to process the game. The main functioning of this class is by having a vector of all current alive `IMovingEntity` and then calling their functions in a loop. A further explanation of this is present in section 5.1.

4.2.11 Player Class: This class inherits from `IShootingMovingEntity` and represents the player. The movement and shooting is controlled through the polling of input from SFML. When it shoots, it creates `PlayerBullet` objects. This class is responsible for keeping track of the players lives and decrementing them when it collides with either an asteroid, enemy bullet, laser field, laser generator or normal enemy. If the lives are zero, it is set dead at which point `Logic` picks up that the player is dead and therefore the game is over.

4.2.12 PlayerBullet Class: This class inherits from `IMovingEntity` and represents a bullet that can fired by `Player`. It moves radially inwards from the circumference of the circle to the center of it, spawning at the location of the player when it was fired. The bullet can collide with an enemy, satellite or laser generator.

4.2.13 Satellite Class: This class inherits from `IShootingMovingEntity` and represents one of the satellites that spawn in groups of three. This class

has three phases: movement expansion from the center until they are a fixed distance of the radius, then setting a new origin and finally gyrating around this origin. During phase three, the satellites can shoot at `Player` if the random chance allows.

4.2.14 StopWatch Class: This class represents a timer that counts seconds which is independent on CPU speed. It is used to limit the number of times per second that the functions within the game loop runs, updating it once enough time has passed for a smoother graphical presentation with no screen lag or tear, and so that the processing and rendering speed is irrespective of processor type.

4.3 Presentation Layer

The presentation layer is described as the part that does the visual rendering of objects to display something meaningful to users. In this project, it will be all classes that utilize SFML.

4.3.1 GameOverScreen Class: This class renders the Game Over window on the screen when player loses all lives. It displays the user's current score as well as the high-score.

4.3.2 Presentation Class: This class is responsible for drawing all the sprites on the main game window. `Logic` provides it with vector of `IMovingEntity` that are looped through, using a function that extracts the entity type, maps the corresponding texture from `ResourceManager` to a sprite and draws it in the necessary position.

4.3.3 SplashScreen Class: This class renders the splash screen window of the game which has instructions on how to play. It exists until the user presses the space key.

4.3.4 WinnerScreen Class: This class renders a Winner window with the points scored during the game alongside the high-score. It is only displayed if `Logic` has counted a total of 50 enemies killed.

5. GAME BEHAVIOUR

This section details the way in which the game functions with the implemented code structures. The game can be broken up into four main parts that need to be explained to gain and understanding of the flow of control.

5.1 Game Loop

The entire game runs within a loop. This game loop was implemented in conjunction with a timer to decouple the pace of the game from the processor speed, ensuring that the game runs smoothly and consistently on a variety of computers. The game loop is implemented in the `Logic` class and is simplified flow of control is illustrated in Figure 2. The game loop has the following the processes: polling for user input, spawning enemies (although in the implementation, a function for each of the different enemy entities exists), updating the position of all the entities and player on screen, shooting of entities, checking of collisions and checking whether the game is over or not. The loop has a delete function in between the checking of collisions and the checking of enemies that deletes dead objects from the vector.

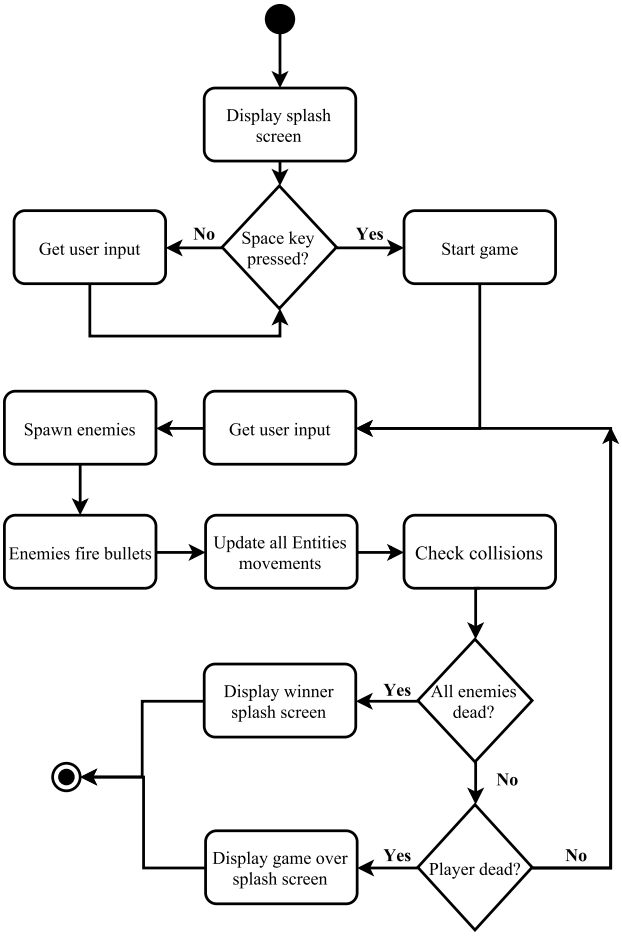

Figure 2 : Flow chart showing logic of the game loop

The game loop utilizes multiple classes to enable communication with each other, an overview of these conversations is illustrated in the sequence diagram in Figure 5 in Appendix B.

5.2 Movement

The player moves along the circumference of a circle while the other entities move radially inwards or outwards. Due to these characteristics, a combination of both rectangular and polar coordinate systems are used and stored in the `Position` class. The equations

that relate the rectangular and polar coordinates are shown below in Equation (1) and (2).

$$x = rcos(\theta) \tag{1}$$

$$y = rsin(\theta) \tag{2}$$

These equations allow for easy calculations of x and y coordinates according to the relevant angle and radius sizes.

5.3 Collision Detection

There are many different entities that can exist in the game at any given point. However not all entities can collide with each other. The entity collision relationship is shown in Figure 3. It is important to use an efficient and accurate collision detection algorithm as the vector of objects grows larger, requiring more time to process all valid collisions.

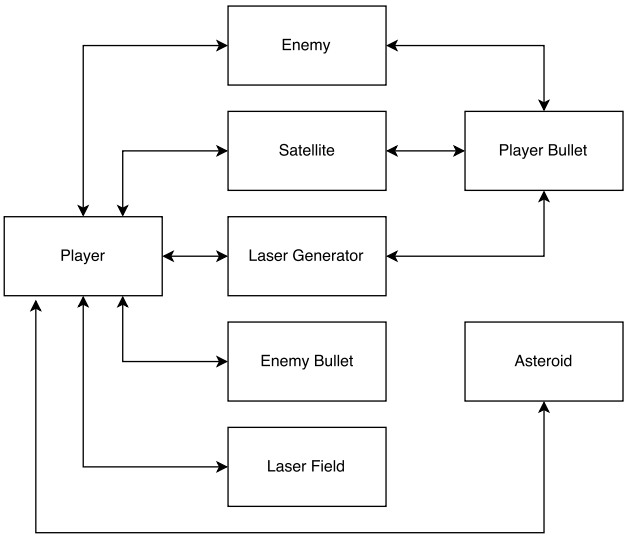

Figure 3 : Entity Collision relationships

All the game sprites are represented by square images whereby placing a circle within the square bounds represents roughly 79% of the area with only a small portion of the 4 corners of the square not being accounted for. The combination of square sprites and the low resolution of the game sprites allows for a simple circle collision algorithm to be implemented. The algorithm takes the centre points of the two circles of two objects and calculates the distance between the two points. If this distance is less than the sum of the two radii there is a collision. This algorithm proved to be sufficient for this application as the player does not perceive any missed collision or latter collisions (in the case of an object being within the cut corners of the square). The collision detection algorithm runs on a vector of `IMovingEntity` which means each entity is checked against every other entity with the algorithm first checking whether a collision is possible according to Figure 3. This implementation of collision detection has a time complexity of $\mathcal{O}(n^2)$ which is inefficient.

5.4 User Input

The user input is polled and captured using the built-in functionality that is present in SFML and is then sent to the logic layer to act on. The timer in the game loop allows for control over the sensitivity of the user input. This helps ensure that the player does not move too quickly when the arrow keys are held down with a further implementation to de-bounce the space key so that the user cannot fire a line of `PlayerBullet` objects by holding down the key.

5.5 Enemy spawning

A total number of 50 enemies can be spawned in one game. `LaserGenerator` and `Asteroid` do not count to this total and although this total number is fixed, the number of `Satellite` and `Enemy` that spawn every game iteration varies This allows for some diversity and opportunities for high-score chasing. These enemies have different spawn chances and these are shown in Table 2.

Table 2 : Non-Player Objects spawn rates per game loop iteration

Object	Spawn Chance (%)
Enemy	0.05
Satellite	0.013
LaserGenerator	0.016
Asteroid	0.0125
Shooting (Enemy)	0.05
Shooting (Satellite)	0.01

6. GAME UNIT TESTING

Unit testing is the process of testing a piece of functionality that a program has, usually an entire class interface and implementation. Test Driven Development (TDD) involves writing the tests first and then the minimum code required to make them pass, followed by re-factoring [3]. Although this does have its benefits, as the overall functionality is built up from small pieces of code, it was not used in the development of Cosmic Journey.

There is still a benefit to testing the objects created by classes of the program, as this allows more control to indicate whether they are functioning as intended, isolated from other objects of the program. The following sections are breakdowns of what functionality of each class was tested. In the following, the life-cycle of an object is the number of iterations of the game loop which take the object from it being spawned,

to when it is destroyed by surpassing the circumference of the circle, without interference from `Player` or `PlayerBullet`. Due to the similarity of how the entities move, the testing will be broken down into the objects that inherit from the `IMovingEntity` class, the `IShootingMovingEntity` class and other class tests. The reason for this grouping is that the objects that utilize inheritance can all be tested very similarly with only a shoot function being different between them.

6.1 Testing Framework

The testing was carried out through the use of doctest 1.2.1 and the mingw64 5.1.0. compiler.

6.1.1 Testing of IMovingEntity This is a pure interface class and thus only the derived classes can be tested. The classes and tests that fall under this category are:

- testAsteroid for `Asteroid` class
- testPlayerBullet for `PlayerBullet` class
- testEnemyBullet for `EnemyBullet` class
- testLaserField for `LaserField` class

Each of these objects have all of their functions tested. The constructor is tested to check that it initializes objects to the correct values of having an alive status, being able to get the entity type, whether it can respawn and getting the hit radius. These fields are vital for the controlling of the object during its life-cycle. Further testing is done on these objects to check if they spawn in the correct location, whether they can move outwards or inwards radially, whether they can be set dead and if they are set dead once they exceed the radius of the circle. The checking of the condition for the last test is done internally by the class which is where the need for a loop arises, to take the object to the end of its life-cycle. This is done by calling the move function multiple times until the object leaves the circle circumference. By testing all these functions, each objects' entire life-cycle has been tested, which means that the objects can function independently of other entities, as expected.

6.1.2 Testing of IShootingMovingEntity This is a pure interface class and thus only the derived classes can be tested. The classes and tests that fall under this category are:

- testEnemy for `Enemy` class
- testPlayer for `Player` class
- testSatellite for `Satellite` class
- testLaserGenerator for `LaserGenerator` class

All functions for these objects are tested similarly to the objects from section 6.1.1 as this is another layer of the inheritance hierarchy, so derived classes will have the same functionality. The only new function that is tested is the shoot function. This is tested by calling shoot for an object of `Enemy`, `Player`, `Satellite` and `LaserGenerator`, and since the return type is a vector, the vector size is tested for greater than one. Greater than one means that there exists a (`PlayerBullet`, `EnemyBullet` or `LaserField`) object and thus the shoot function works as intended.

6.1.3 testCollisionHandler: This class had its initialization tested along with all of its public functions to check that two objects that can collide, do, and to make sure that objects that cannot collide, didn't. Multiple objects of `IMovingEntity` and `IShootingMovingEntity` are created, moved to the same position and stored within a vector. This vector was passed into the public function which calls private functions to check the collisions between valid entities and return a vector with entities that have collided set as dead. By checking if the correct entities are dead, the correctness of this class can be tested.

6.1.4 testEnemySpawner: To test this class, the spawn rate was removed as this would hinder any meaningful testing as the testing is to make sure that the entities can be spawned correctly, not the rate at which they are spawned. For this reason, the class is edited to have 100% spawn rate when the relevant function is called. All functions were tested: the spawning of `Enemy`, the spawning of a group of three `Satellite`, the spawning of one laser generator group (two `LaserGenerator` that shoot seven `LaserField` objects), spawning `Asteroid`, the respawning of `Enemy` and the shooting of `Enemy` and `Satellite`. The functions to limit additional spawns of laser generator groups or satellite groups is also tested. Each passes as is expected, indicating the class works correctly when finally spawned by the spawn chance.

6.1.5 testGrid: The initialization of the grid with correct parameters was tested along with all its getter functions for the height, width, radius, and center coordinates.

6.1.6 testHighScoreMaanager: This class was tested to see if it could write and read to and from a text-file to establish whether a link was available between reading and storing data for the use of the high-scores feature.

6.1.7 testLogic: This class has a private interface for most of its functions. The only public functions are the constructor and the run function which starts the game. This encapsulation allows the functions to not be called and manipulated by external classes. The function that deletes dead entities can not be called

externally, however this is an important function, so it is tested through the creation of a vector of dead entities and the deletion algorithm is run on the vector. It passed and this simulated the way the private delete function would work.

6.1.8 testPosition: This class was tested by checking if it could set its internal private members through the setter functions and if the correct values were returned from the getter functions. This is done through the insertion of set parameters that were then tested against the values returned by the functions.

6.1.9 testResourceManager: This class was tested by creating an object of the class and seeing if it can correctly match the string for the path of the resource to the entity type.

Table 3 below summarises the number of tests per class:

Table 3 : Summary of amount of tests per class

Tested Class	Number of Tests
Asteroid	5
CollisionHandler	11
Enemy	6
EnemyBullet	6
EnemySpawner	8
Grid	2
HighScoreManager	1
LaserField	5
LaserGenerator	5
Logic	1
Player	6
PlayerBullet	7
Position	3
ResourceManager	1
Satellite	7
Total	74

All tests performed on the basic building blocks of the game passed the expected conditions. This indicates that these classes worked as intended.

7. CRITICAL ANALYSIS AND RECCOMENDATIONS

The following section details what has been achieved by the program and what flaws are present. The project is classified into three main sections: the game functionality which are the features the game has, the implementation which is the code behind those features and the testing which is the validation of the implementation.

7.1 Game Functionality

The following section is an analysis of what functionality has been achieved by the program.

7.1.1 Successes The game met all the requirements and enhanced functionality as detailed in section 2.2. This means it had all basic functionality, three minor and two major features as set out by the project brief [4]. Although there were other features that could be implemented as minor or major features, only the ones that were going to be implemented are presented in this document. Multiple game objects exist: asteroids, a player ship, enemy ships, satellites, and laser generators, with the latter four objects capable of shooting bullets. If the bullets collide with any of these objects they are destroyed or if the objects themselves collide with the player ship. A high-score system is implemented that stores the all time high score achieved. The player also has three lives available to them in their attempt to kill 50 enemies. The features implemented make up the game-play of Cosmic Journey.

The controls have been de-bounced so that it stops users from holding down the space key and firing a line of bullets or erratic and quick movement from left or right key strokes. The bullets are now fired on each space key press and the circular movement of the player ship is smoother and controlled.

7.1.2 Flaws A good aesthetic graphical feature that was present in *Gyruss* was not implemented in Cosmic Journey: the ability for game sprites to change their size in relation to their proximity to the circumference of the circle. The lack of this feature results in the user being unable to perceive the visual effect of moving into the screen, meaning the game does not give the fixed shooter visual effect. The game-play is negatively impacted because all the enemies spawn from the center of the screen and the player bullet had a large chance of hitting the recently spawned enemy with greater certainty due to a lack of scaling, making the game quite easy.

The player bullets travel to the center of the screen which is where all enemies spawn, allowing the user to continuously fire and destroy all enemies as they appear, leading to a very boring and unchallenging experience. This could be prevented by deleting all player bullets a set distance away from the center of the circle. This will allow the enemies to spawn without being hit by a bullet immediately.

The player only re-spawns on the bottom on the playing circle which can be problematic as there exists a case where the player can be killed, re-spawn and that position is occupied by an asteroid or laser field. This would cause the player to die rapidly, decreasing

their lives in quick succession. This problem can be eliminated by implementing a few invincibility frames whenever the player re-spawns to allow entities to pass by without a collision occurring.

The player movement is favoured clockwise when both the left and right key are held down, instead of the key pressed and held last. Although this does not impact significantly on the player movement and the user experience, if the player object had more complex movements, this could give rise to a flaw.

The spawn rates of the non-player entities and their shooting is flawed in the respect that there can be instances of the game where a large period of time goes by whereby no enemies spawn as it is all dependent on the random number generated. This could be fixed by implementing a function that checks if an enemy is present on the screen every few frames and automatically spawns one if there isn't, allowing the user to play the game without any pauses.

7.2 Design and Implementation

The following section is an analysis of the implemented code behind the game.

7.2.1 Successes The code successfully implemented a solution that separated concerns using a design pattern inspired from MVP. The presentation layer can be changed without having to alter the logic or data layers.

The separation of objects that inherit from `IMoving Entity` and `IShootingMovingEntity` was considered a success. The use of pure interface inheritance allowed for an implementation that prevents tight coupling. This implementation of inheritance also allows for polymorphism to be used and was leveraged extensively when updating the movement of all game objects in a single vector.

Modern idiomatic C++11 and C++14 coding standards such as smart pointers to automatically manage memory, lambda functions, ranged-based for loops, auto and override keywords were used where applicable. The complexity of creating copies of variables and objects is reduced as much as possible by passing them by constant reference, which was found to reduce the processing time significantly, viewable on the presentation layer by fixing lagging of sprites to a smooth transition for movement.

An object oriented solution that encapsulates most of an object's private data was achieved. The modular design of the object oriented solution allows the code to be maintainable, extensible and scalable, preventing other classes from manipulating the game-play functions.

Even though attempts were made to make the game run at the same speed, irrelevant of processor, there are some cases where the game still runs too fast or too slowly. The exact cause of this needs to be determined.

7.2.2 Flaws The implementation of the move function of every derived class from `IMovingEntity` is very similar and thus violates the DRY principle. Apart from a few variable changes, the code structure for moving is almost identical. This problem could be solved through the implementation and use of an abstract base class that implements the move function for these derived classes.

The game has a few data classes, and while these are needed to allow a better flow of control and data encapsulation, a few can be implemented together. An example of two classes that can be integrated together are the `Grid` and `Position` as these two classes share the same radius and origin. A data class like `ResourceManager` which is comprised of two private variables and two functions to return these variables can be eliminated and its functionality implemented in the presentation layer. This however would combine data and presentation layers as the if the presentation layer is removed, so is the data pointing the program to the texture of the sprites.

Multiple vectors are used in the implementation and are computationally expensive to use if erasures are done anywhere in the container except the end. A fix to this would be to implement other data structures like linked lists, doubly linked lists or even trees and using search algorithms to find the needed objects. These are less computational expensive when erasing or searching which could speed up performance of the game.

The implemented collision detection algorithm has a time complexity of $\mathcal{O}(n^2)$, as it has to compare every object in a vector to everything else in the same vector. This is computationally expensive and could hinder the game's performance as the vector of `IMovingEntity` objects gets larger. The implementation of a spatial data structure such as a Quadtree or Spatial Hashing would allow the collision detection time complexity to be reduced to $\mathcal{O}(n)$ [5].

The circle collision algorithm may be accurate for the current implementation of the game. However if the game resolution was to be increased or more complex shapes used as sprites the accuracy of the algorithm would drop and thus would work incorrectly. A more accurate collision detection algorithm for complex shapes such as the Separating Axis Theorem would be more accurate as it can detect the collision between any two convex polygons [6].

The `Logic` class can be seen as a monolithic class as it has many functions. Although all the functions are

required by the game to run accurately, some of these functions can be combined such as implementing one spawn function for all enemies instead of four, for each of the normal enemy, satellites, laser generators and asteroids. The same can be done for the update player and update entities function.

The inheritance hierarchy is flawed, indicated by the respawn function which only serves a purpose on the `Enemy` derived class. The `Player` derived class does not use it either, instead relying on the use of a private variable to indicate a re-spawn. This base class function should be eliminated entirely and the re-spawning should be done through the objects themselves (like in the case of `Player`) and not the `EnemySpawner`.

In terms of complexity and violating DRY, the function that finds the entity type of objects in the main vector of the game that maps textures onto sprites every time the vector is sent to `Presentation` can be replaced with sprites with already mapped textures corresponding to the entity types. This will reduce the time complexity and violation of DRY that happens by repeatedly mapping the textures to sprites.

7.3 Testing

The following section is an analysis of the tests that were written to test the objects of the game.

7.3.1 Successes The unit tests manage to successfully isolate game objects from one another to be tested in most cases. The test cases together make up entities' entire life-cycles which indicates objects can manage and control themselves with their private functions and data members which means the objects correctly adhere to the good programming principle of encapsulation.

All tests are automated which is why user input is not tested but the responses to user input are tested. Since these work as expected, it is assumed that the input would work as it comes from SFML, which is thought to have been tested extensively before being released.

7.3.2 Flaws The testing of some classes was not possible without the use of other classes to assist in the testing like in the case of `CollisionHandler`. To fix this, a mocking framework should have been implemented that would have mock objects to represent the classes that would have needed to be tested on. Due to a lack of time, mocking was not implemented.

Although looping is implemented to advance the life-cycle of objects, it is not a good practice as it can take long processing times to loop through the life-cycle. This was implemented in the testing as no alternative was available without changing the structure of the code.

8. CONCLUSION

This report has documented the design, implementation and analysis of a C++ game, Cosmic Journey. The design was shown to take inspiration from the MVP concept, deriving an object oriented design, making use of inheritance and polymorphism. The game attempted to adhere to good programming practices where possible. The rigorous testing of all objects resulted in a game that behaves as expected. The functionality of the game was shown to be worthy of the excellent criteria as the game had the basic, three minor and two major functionality features implemented. The analysis of the code showed that while it did achieve the requirements in section 2.2 while adhering the constraints of section 2.4, it still had a number of flaws with the improvements that can be made to fix these presented. Although these flaws were present, Cosmic Journey still provides an enjoyable and full game-play experience.

REFERENCES

[1] M. Fowler, Separating User Interface, March/April 2001 IEEE SOFTWARE, https://martinfowler.com/ieeeSoftware/separation.pdf [Accessed 17 October 2017]

[2] J. T. Emmatty, Differences between MVC and MVP for Beginners, 23 November 2011, https://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners [Accessed 17 October 2017]

[3] M. Fowler, Test Driven Development, March 2005, https://martinfowler.com/bliki/TestDrivenDevelopment.html [Accessed 17 October 2017]

[4] S. Levitt, Project 2017 - Gyruss, http://dept.ee.wits.ac.za/ levitt/elen3009/assessment/ elen3009-project-2017.pdf , [Accessed 17 October 2017]

[5] S. Lambert, Quick Tip: Use Quadtrees to Detect Likely Collisions in 2D Space, https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space–gamedev-374 [Accessed 17 October 2017]

[6] K. S. Chong, Collision Detection Using the Separating Axis Theorem, https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem–gamedev-169 [Accessed 17 October 2017]

Appendices

A Inheritance Hierarchy

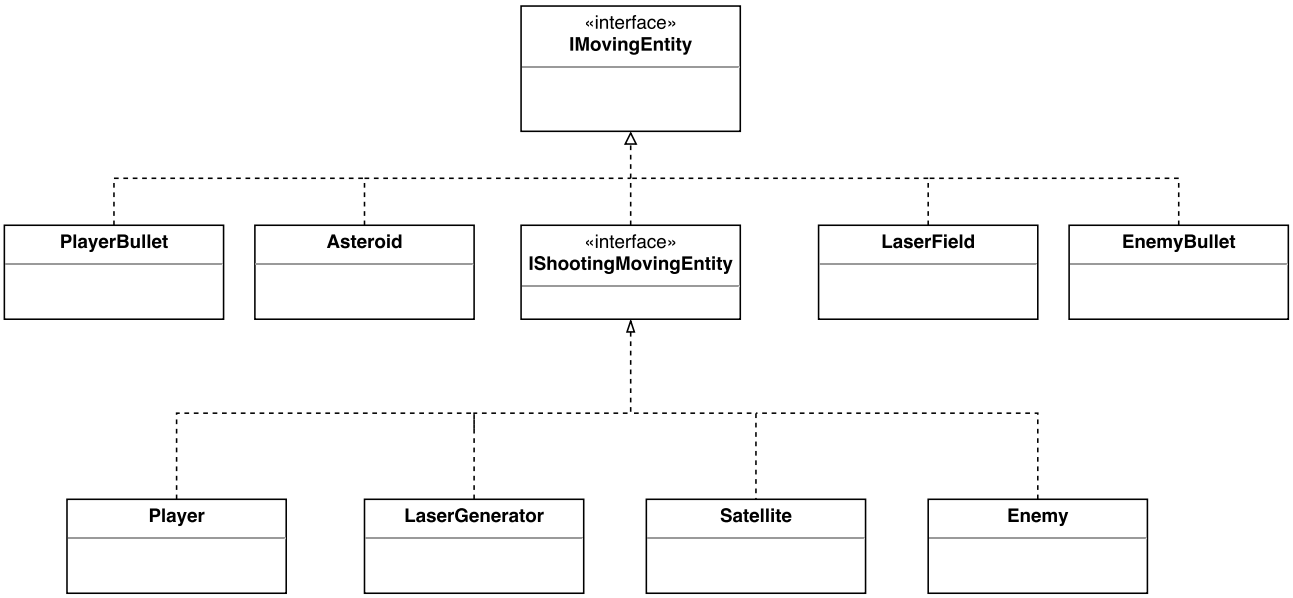

Figure 4 : Inheritance hierarchy diagram

B Sequence Diagram

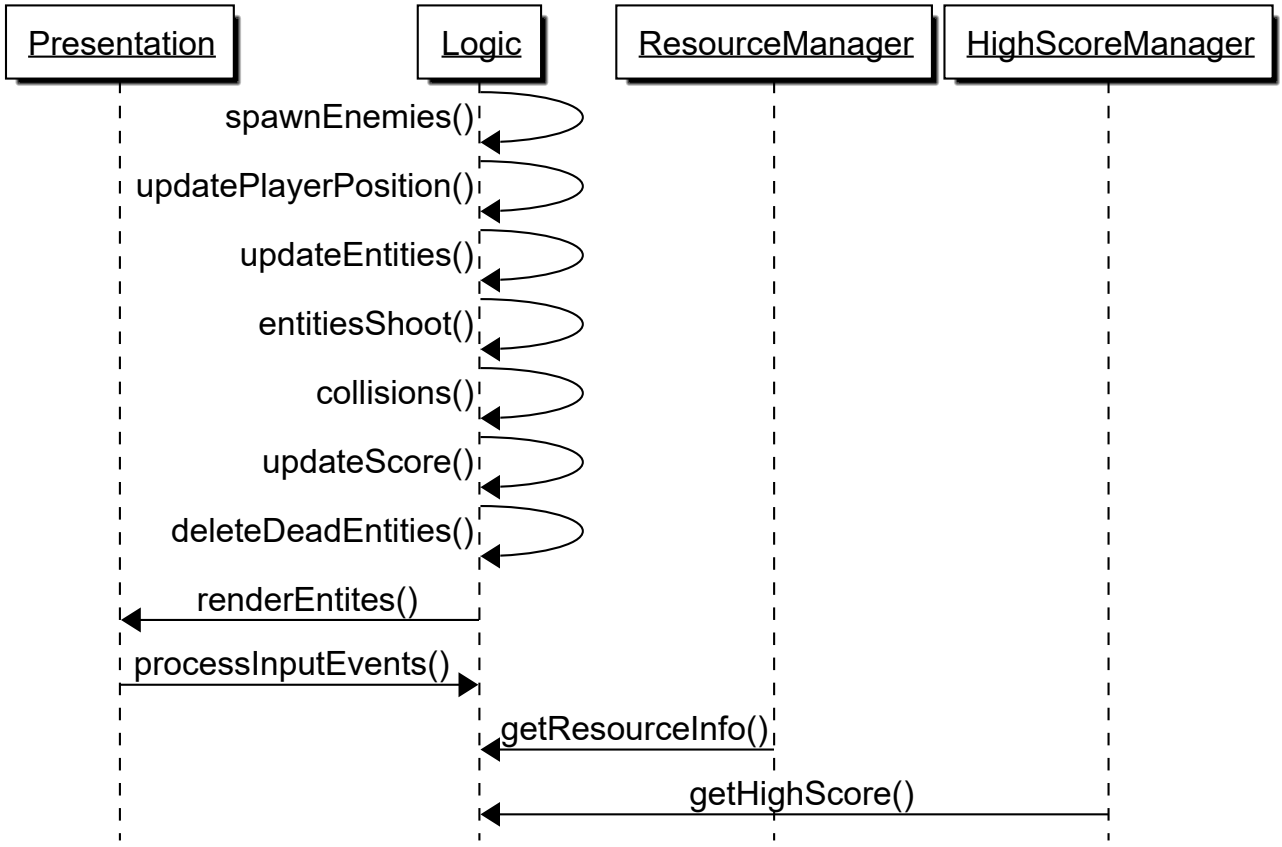

Figure 5 : Sequence diagram