THE DESIGN AND IMPLEMENTATION OF A GYRUSS BASED GAME: PIXEL POTTER

Kayla-Jade Butkow (714227) & Kelvin da Silva (835842)

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

Abstract: This paper presents the design, implementation and testing of a computer based game, *Pixel Potter*, derived from the arcade game *Gyruss*. The code was divided into an application logic layer and a presentation layer, which were decoupled in order to enhance the reusibility of the code. An inheritance hierarchy was implemented consisting of a `movingEntity` base class, which provided the entities with movement functionality, and a `MovingShootingEntity` base class which provided shooting capabilities. A total of 110 unit test cases were implemented, covering all basic movements and general game logic. Relevant tests were grouped into 13 test suites to allow for easy filtering. The game is deemed a success since all required features were implemented, and modern code consisting of good coding principles was created. For future improvements, the inheritance hierarchy should be improved to further reduce the repetition of code, and a mocking test framework should be implemented.

Key words: C++14, inheritance, Object Orientated Programming, SFML, unit test

1. INTRODUCTION

The report documents the design process and implementation of a computer game, *Pixel Potter*, created using C++ and Object Orientated Programming. The report contains a detailed overview of the game's basic functionality, code and class structures and testing methods. Important concepts such as inheritance and information hiding are discussed and their roles within the code are highlighted. The dynamic in-game behaviour and the interactions between objects is analysed. A critical analysis of all aspects of the designed game is also presented and future improvements are given.

2. PROBLEM DEFINITION

2.1 Pixel Potter Game Play

Pixel Potter, modeled on the classic arcade game *Gyrus*, is a game developed using C++ that can be played on a 1920×1080 screen.

The game consists of a `Player` that moves along the perimeter of a circle, faces inwards and is required to destroy `Enemies` advancing from the centre of the circle. The `Player` is able to move clockwise and anticlockwise and to destroy advancing `Enemies` by shooting bullets.

There are several types of enemies, each with unique types of movement and life characteristics. The `Enemy` entities fire bullets towards the perimeter of the circle randomly and are destroyed upon colliding with the `Player` or `Player Bullets`. `Asteroids` cannot be destroyed by `Player Bullets`, cannot fire bullets and cause the `Player` to lose a life upon collision. `Satellites` occur in groups of three and gyrate along a small circle within the circle that the `Player` navigates around. The `Satellites` are able to fire bullets toward the perimeter of the `Player` circle. Additionally, after all three `satellites` in a group are destroyed, the `Player's` gun undergoes an upgrade whereby it fires two bullets at once. `Laser Generators` and a `Laser Arc` are spawned randomly throughout game play. The `Laser Arc` can only be destroyed if one of the `Laser Generators` are shot and destroyed.

The user is able to win the game by killing 20 `Enemy` entities. The user loses the game if the `Player` entity loses five lives before all of the enemies have been killed.

2.2 Success Criteria

The project was deemed successful if all of the above mentioned functionality was implemented. Furthermore, high quality code was required which used good coding principles, effective information hiding and significant separation of logic and presentation layers. Additionally, the correct implementation of coding principles, such as inheritance and role modeling, were required.

High quality unit tests which verified the behaviour of game objects were necessary to ensure effective code development and operation.

2.3 Assumptions and constraints

Game play was constrained to a non-resizeable 1920×1080 screen on a Windows machine. Only keyboard inputs were accepted as inputs into the game.

The *SFML* 2.3.2 library was assumed to be thoroughly tested throughout its development and the need for further testing of the framework was therefore unnecessary.

3. CODE STRUCTURE AND IMPLEMENTATION

The code is divided into two distinct layers:

- Application Logic layer
- Presentation layer

The two layers were separated so as to comply with the separation of concerns principle. This separation allows for a decoupling from the graphics library, which means that the code can be reused with a different graphics library if required by the user. The class that controls the application layer is the `Game` class, and the class that controls the presentation layer is the `Window` class. These classes interact with the other classes in order to create the game.

3.1 Domain Model

The game consists of various objects that appear on the game window and interact with each other. These objects will henceforth be called entities.

The entities involved in the game are as follows:

- Player
- Player bullets
- Enemy
- Enemy bullets
- Satellite
- Laser generator
- Asteroid

These entities are all required to move, and some of them are required to shoot. It is also required for the necessary entities to be able to collide. Furthermore, all of the entities must be able to be drawn onto the screen.

Since the `Player` is required to move along the circumference of a circle and the `Enemies` are required to move radially from the centre of the circle to the circumference, it was decided that a polar coordinate system would be used for the movement of all of the game entities. The model used for the coordinate system is given in Equation 1.

$$(x, y) = (r\cos\theta + x_0, r\sin\theta + y_0) \tag{1}$$

Where:

r = Radius
θ = Angle (radians)
x_0 = x coordinate of the centre of the circle
y_0 = y coordinate of the centre of the circle

3.2 Application Logic Layer Class Structure

The majority of the classes within the game belong in this layer. The application layer is responsible for maintaining all features required in the functioning of the game. Within this layer, the entities are created and updated, interactions between entities are managed, and the game loop is updated.

3.2.1 `movingEntity` class: This class forms the base class for all of the game entities mentioned above. This class provides its child classes with the interface for all of the functions necessary for movement. The class contains virtual functions that are to be overwritten by the child classes. The class also contains the implementation of certain functions that are required by all of the entities, and whose implementation is common for all of the entities, such as setting and getting the number of lives the entity has left. The decision to not make the class a pure interface was taken since it prevents the violation of the DRY principle, by reusing common functions.

The `movingEntity` class also stores the size of the entity, and scales the size in order to scale the graphics. Therefore, the class also manages the size of the hitbox of the entity. This is essential for detecting collisions between entities. This will be discussed further in *Section 4.2*. The class contains a static vector of shared pointers of type `movingEntity` (`entityList`), which contains all of the moving entities that are alive in the game at present. This allows for easy managing of all of the game entities.

The class manages the amount of lives each entity possesses. Upon colliding, the number of lives are decremented by one. This ensures that each entity is responsible for managing the number of lives it possesses.

3.2.2 `MovingShootingEntity` class: This class inherits from the `movingEntity` class, and provides the moving entities with shooting capabilities.

The inheritance hierarchy used in the game can be seen in *Figure 1*.

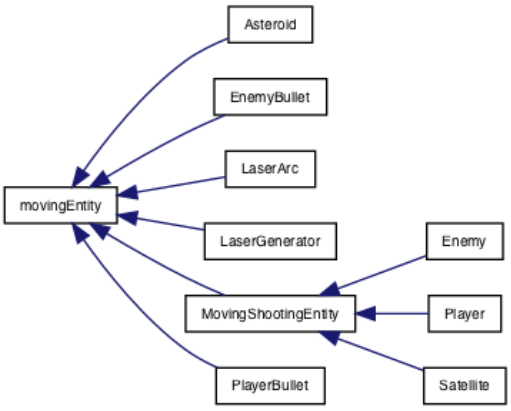

Figure 1 : Diagram showing the inheritance hierarchy used within the game

3.2.3 `Player` class: The `Player` class represents the `Player` entity and inherits from the `MovingShootingEntity` class. This is because the `Player` entity is required to move and shoot bullets. The

Player's movement and shooting are controlled by the user. The Player is instantiated along the perimeter of the circle at Cartesian 270°, and is able to move along the circumference of the circle. The Player shoots along a radial line. The Player initially has five lives, and these lives are managed by the class. The lives are decreased upon colliding with any Enemy entity or Enemy Bullet.

Upon destroying three Satellite entities, the Player's gun is upgraded. This upgrade is controlled by the Player, and it is the Player's responsibility to ensure that the gun remains upgraded for the duration of the game.

3.2.4 Enemy class: The Enemy class inherits from the MovingShootingEntity class. The class represents an Enemy entity. The Enemy is instantiated at a randomly generated angle and moves radially outwards along a straight line. The Enemy shoots bullets randomly. The bullets are created by the Enemy, and are instantiated at the current angle of the Enemy. Each Enemy has one life that is lost upon colliding with the Player or a Player Bullet. If the Enemy reaches the circumference of the circle without losing its life, it is moved back to the centre of the circle, and moves radially outwards at a different angle. This repeats until the Enemy is killed.

Aside from controlling the movement of the Enemies and creating bullets, the Enemy class is also responsible for counting the total number of Enemies generated, the number of Enemies alive and the number of Enemies that have been killed. These values are essential in allowing the user to win the game.

3.2.5 Satellite class: This class represents a Satellite entity and inherits from the MovingShootingEntity class. The Satellites appear in groups of three and move around the circumference of a small circle. The circle of Satellites appear directly in front of the Player's location and the Satellites remain there until they collide with a Player Bullet. The Satellites are responsible for controlling their movement and for randomly shooting bullets towards the perimeter of the circle. These bullets are of type EnemyBullets. The angle of the bullets is controlled by the Satellite and is dependent on the current Cartesian quadrant of the Satellite. This is done to ensure that the bullets always fire towards the perimeter of the circle.

The Satellite class is also responsible for counting the total number of Satellites, the number killed and the number alive. These numbers are used to determine when to generate new Satellites, and also to determine when the Player's gun should be upgraded.

It must be noted that the entityCreator class ensures that three Satellites are created at a time, rather than the Satellite class.

3.2.6 PlayerBullet class: This class inherits from the movingEntity class, and represents bullets that are shot by the Player. The bullets originate at the current position of the Player, and move radially inwards towards the centre of the circle at the angle of the Player at the instant that the bullet was created. Each bullet has one life that is lost upon colliding with an Enemy entity or an Enemy Bullet. If the bullet has not collided, the bullet loses its life when it reaches the centre of the circle.

As mentioned in *Section 3.2.3*, when the Player destroys three Satellite entities, the Player's gun is upgraded. This upgrade is implemented by the PlayerBullet class, through the use of a copy constructor that sets the angle and increased size of the upgraded bullets.

3.2.7 EnemyBullet class: This class inherits from the movingEntity class. The class represents bullets shot by an Enemy entity. The Enemy Bullets are instantiated at an angle that is dependent on the type of enemy that created the bullets. The bullets then travel radially outwards at a faster speed than the enemy that created them. The Enemy Bullets have one life, which is lost upon colliding, or upon reaching the circumference of the circle.

3.2.8 Asteroid class: The Asteroid class inherits from the movingEntity class, and represents Asteroids that cannot be destroyed through collisions with the Player or the Player Bullets. The Asteroids are created at the centre of the circle at the Player's current angle. They then move radially outwards. The Asteroids have one life and the life is only lost upon the Asteroid reaching the circumference of the circle.

3.2.9 LaserGenerator class: This class inherits from the movingEntity class, and represents the generators that create a laser force field. The Laser Generators appear in pairs at the centre of the circle at a random angle, and move radially outwards towards the circumference. A copy constructor is used to create two identical Laser Generators that differ only in the angle of their movement. As with the other enemy entities, the Laser Generators have one life that is lost upon collision or upon reaching the circumference of the circle.

It must be noted that the entityCreator class ensures that two Laser Generators are created at a time.

3.2.10 `LaserArc` *class:* This class represents the laser force field that is created by two `Laser Generators`. It inherits from the `movingEntity` class. The `Laser Arc` travels in the same manner as the `Laser Generators`, and is initiated at an angle that is half way between those of the two `Laser Generators`. The `Laser Arc's` lifespan is controlled by the lifespan of the `Laser Generators`. As soon as one `Laser Generator` has been destroyed, the `Laser Arc` sets its lives to zero, thus destroying itself.

3.2.11 `CollisionManager` *class:* This class manages all of the collisions between entities. Is it responsible for looping through the `entityList` vector and checking if any collisions have occurred. It is important to note that collisions are not checked between all entities. A graphical depiction of the entities that can collide is given in *Figure 2*. In the figure, a double sided arrow implies that both entities will lose a live upon collision, while a single sided arrow implies that only the entity with the arrow pointing to it will lose a life.

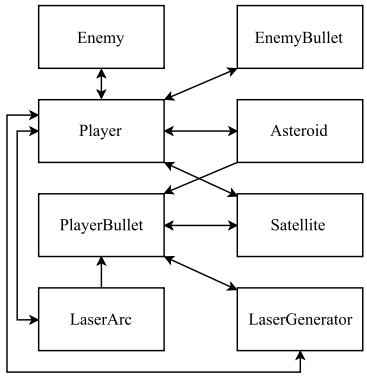

Figure 2 : Diagram of the collisions that can occur between entities

The class is also responsible for deleting any entities that do not have any lives remaining.

3.2.12 `entityCreator` *class:* This class is responsible for creating all of the enemy entities during the game play. This includes the `Enemy`, `Satellite`, `Asteroid` and `Laser Generator` entities. The `Enemy` and `Asteroid` entities are created randomly based on time, and the `Satellite` and `Laser Generator` entities are generated periodically in time. The clock is responsible for ensuring that three `Satellites` are created at a time, and also that two `Laser Generator` entities are created at a time.

This class has a composition relationship with the presentation layer `Clock` class.

3.2.13 `Game` *class:* The `Game` class is the most important class in the application layer. The class manages all processes required in the game.

The responsibilities of the class are as follows:

- Controlling the state of the game
- Controlling the creation of the enemy entities
- Controlling the updating of the game entities
- Managing the collision detection
- Interacting with the `Window` class to ensure that the entities are rendered onto the screen
- Ensuring that the game loop continues as long as the game window is open
- Restarting the game clock as necessary

3.3 Presentation Layer Class Structure

The presentation layer consists of two classes: the `Clock` class and the `Window` class. These two classes interface the game logic with the SFML graphic library.

3.3.1 `Clock` *class:* The `Clock` class is crucial to the functioning of the game. The game uses the elapsed time in each game loop to control the speed of all of the entities. This is important as it makes the movement framerate independent, implying that the entity will move a certain distance per second [1, 2]. The class is responsible for restarting clocks as well as for returning the elapsed time.

3.3.2 `Window` *class:* This class is the most important class in the presentation layer. The `Window` class renders all of the entities onto the screen so that the game can be visualized. The `Window` class is also responsible for drawing the screen corresponding to each of the four game states: within the splashscreen, playing the game, in a lose state or in a win state. This is essential as it tells the user how to play the game. The class also displays how many lives the `Player` has remaining and how many `Enemies` must still be killed in order to win the game.

The `Window` class also polls for user input from the keyboard. The class then converts the keyboard inputs into an enumerator type so that the `Game` class can process the events independently of SFML.

A UML diagram of the implemented inheritance hierarchy is given in *Figure 3*. *Figure 4* depicts the interclass relationships, without the inheritance hierarchy.

4. DYNAMIC GAME BEHAVIOR

4.1 Main Game Loop

The main game loop is controlled by the `Game` class. Within the loop, polling for user input is done by the

`Window` class. These user inputs are used to control the `Player` (as described in *Section 4.3*). The enemy entities are then created. The creation of these entities is randomized according time (except for the `Satellite` entities, which appear periodically). Thereafter, the position of the entities are updated, and collision detection is performed. Based on the collision detection, the state of the game is set to a win or lose state, or is left in a playing state. The required entities are then drawn onto the screen. This process repeats until the state of the game is changed to a win or lose state. In a win state, a screen is displayed which informs the user that they have won, and then allows them to restart. A different screen is displayed for a lose state. At any state in the game, the user may exit the game by pressing the exit button.

This process is shown diagrammatically in *Figure 5*.

Figure 6 shows the important interactions that occur between objects during the main game loop.

For a losing state, the state is set by the `entityCleanUp()` function in the `CollisionManager` class. This function returns a losing state when the player's lives have been depleted. The winning state is controlled by the `Game` class through communications with the `getTotalNumberofEnemies()` and `getNumberofEnemiesAlive()` functions. These functions are essential in ensuring that the game is only won once the required number of enemies have been killed. These functions are also used to spawn the correct amount of `Enemy` entities.

4.2 Collision Detection

The method used calculates the distance between the centres of each entity, and then subtracts half of the length of each entity. If the resulting length is less than zero, the two entities are considered to have collided.

During collision detection, each element of the `EntityList` vector is compared to each other element in the vector. If the entities are of types that can collide with one another (as depicted by *Figure 2*), then the above method is applied to check whether they have collided or not. If the entities have collided, their remaining lives are decremented by one.

After collision detection, all entities that have zero lives remaining are deleted from the vector. If the player has zero lives remaining, the state of the game is set to a lose state.

4.3 User Input Polling

The user inputs are polled by the `Window` class, which lies in the presentation layer. In order to maintain a complete separation of the presentation and logic layers, the user input is converted into a strongly typed enumerator variable. Once the user input has been converted, it is passed to the `Player` in order to control the `Player's` movement and shooting. In the polling of user inputs, in order to ensure that the user cannot continuously shoot, a boolean variable is toggled. This means that the user is required to release the space bar before a second bullet can be created. However, the polling is done in such a way that it is still possible for the user to continuously move the `Player` by holding down the left and right arrow buttons. This improves the user's ease and enjoyment of playing the game.

5. CODE TESTING

This section contains the details of all testing performed on the source code for *Pixel Potter* using the *Doctest 1.2.1* unit testing framework. Test driven coding was used to allow for efficient development of code. However, testing of certain features was only done after the feature was implemented. This will be discussed in *Section 5.3*. All aspects of the game logic were tested, however certain functionality was analysed more rigorously than others. The intensity of testing of specific features varied according to the complexity of the functionality of said feature. As is discussed in *Section 5.3*, certain features were unable to be tested.

5.1 Presentation Layer Testing

The presentation layer was not tested. The reasoning behind this decision is discussed in *Section 5.3*.

5.2 Logic Layer Testing

The testing structure for this layer is such that all moving objects of the game undergo movement and creation tests (which are unique for each object), followed by general game logic tests. The general logic tests include tests of collision detection, number of lives, entity deletion, `Player Bullet` upgrades and game state tests.

The designed testing framework made extensive use of test suites [3]. Use of this feature ensured that relevant tests were grouped together. This also meant that groups of tests could be run separately by filtering the tests. While the *Doctest* framework was able to operate effectively without the use of the command line, control over test filtering was accessible with use of the command line [3].

In each of the sections that follow, the name of the suite in which all of the relevant tests are located, is provided. According to [3], test suites can be filtered by using the command line to run the testing executable file *PixelPotterTests.exe* followed by `--test-suite=*<Test-Suite-Name>*` where `<Test-Suite-Name>` is the name of the desired

test suite to run. In total, 13 test suites were created.

In the following description of the tests, the radius is defined as the distance between the centre of the circle of the `Player's` movement and the entity in question.

5.2.1 `Player` *Movement and Spawning:* The `Player` object was tested by ensuring that it was always created at Cartesian 270°. The `Player's` circular movement was tested by confirming that its radius remained constant and only its angle changed during movement of the entity. The suite containing these tests is named `Player_movement_and_spawning`.

5.2.2 `Player Bullet` *Movement and Spawning:* A test was done to ensure that upon creation (due to a simulated space-bar press), the `Player Bullet` has the same position and angle as the `Player`. This object was also tested by confirming that a `Bullet` was not created if the space-bar was not pressed. The `Player Bullet` movement was also tested by ensuring that its angle remained constant throughout its flight while its radius decreased, thus implying inward radial movement. The ability to track the position of the bullet throughout various periods of time was confirmed through several tests. The suite containing these tests is named `Player_bullet_movement_and_spawning`.

5.2.3 `Enemy` *Movement and Spawning:* The `Enemy` object was tested by checking whether `Enemy` entities were spawned at the centre of the screen. This was done regardless of the preexisting number of `Enemies`. Testing was performed to ensure that the `Enemy's` angle of departure from the centre of the screen was constant while its radius increased with time. This implied outward radial movement. Additionally, the ability to track the positions of all `Enemies` was tested. It should be noted that the random generation of `Enemy` objects was not tested as discussed in *Section 5.3*. The suite containing these tests is named `Enemy_movement_and_spawning`.

5.2.4 `Enemy Bullet` *Movement and Spawning:* The spawning of the `Enemy Bullet` was tested in the same manner that the `Player Bullet` was tested. These spawning tests ensured that the `Enemy Bullet` had the same position and angle as the `Enemy` at the instant of creation. As discussed in *Section 5.3*, the random generation of `Enemy Bullets` was bypassed. Since the `Enemy Bullets` move radially outwards, the same tests as those used in *Section 5.2.3* were implemented for movement testing. The suite containing these tests is named `Enemy_bullet_movement_and_spawning`.

5.2.5 `Asteroid` *Movement and Spawning:* The random generation of these objects was bypassed for testing, as in the previous cases. Testing was constructed to show that the `Asteroid` spawns at the centre of the screen and has an initial angle equal to the current player angle. Additional tests showed that the angle remained constant while the radius increased. A multitude of tests was created to ensure that the `Asteroid's` angle remained constant during flight. These tests ensured that once the entity had been spawned, even with a changing `Player` angle, the `Asteroid's` angle remained constant. The suite containing these tests is named `Asteroid_movement_and_spawning`.

5.2.6 `Satellite` *Movement and Spawning:* For testing purposes, the random generation of these objects was bypassed. Upon spawning a group of three entities, tests were used to verify that each `Satellite` was created around a small circle within the larger circle of the `Player's` movement.

It was tested that the centre of the `Satellite's` circle was located at an angle equal to the `Player's` current angle. It was proven that the group of objects had equal radii and angles that differed by 120° to each other. The movement tests of the satellites involved verifying that while gyrating, the radii to the centre of the `Satellite's` circle remained constant and only the angles changed. It was also tested that the difference in angle between each `Satellite` was 120° at all times. The suite containing these tests is named `Satellite_movement_and_spawning`.

5.2.7 `Satellite` *Bullet Movement and Spawning:* `Satellite` bullets were generated manually for the sake of testing. The bulk of the testing was based on verifying the expected bullet flight path when the circle of `Satellites` was located in different quadrants of the `Player` circle. The suite containing these tests is named `Satellite_bullet_movement_and_spawning`.

5.2.8 `Laser Generator` *Movement and Spawning:* Testing ensured that the spawning of a pair of `Laser Generators` occurred at the centre of the screen. The difference in the angles of the two `Laser Generators` was proven to be 50° at all instances of flight. Tests were also implemented to verify that both laser generators moved outwards in an radial manner. The random generation of `Laser Generators` was bypassed. The suite containing these tests is named `Laser_generator_and_arc_movement_and_spawning`.

5.2.9 `Laser Arc` *Movement and Spawning:* This object type was tested in conjunction with the `Laser Generators`. The spawning tests verified that the starting position of the `Laser Arc` was at the centre of the screen and that its starting angle was the midpoint between the angles of the two `Laser`

`Generators`. The `Laser Arc` moved in the same manner as the `Laser Generators` and therefore underwent the same movement tests to verify outward radial movement. The suite containing these tests is named `Laser_generator_and_arc_movement_and_spawning`.

5.2.10 Lives: This testing structure consisted of tests that proved that the initial number of lives that each object possessed was equal to an expected value. Additionally, testing was implemented to ensure that specified entities lost a life once their radius exceeded a maximum value. Tests were also done to ensure that a `Player Bullet` lost a life once it reached the centre of the `Player's` circle. Finally, these tests show the relationship between the `Laser Generators` and the `Laser Arc`. The suite containing these tests is named `Lives`.

5.2.11 Collision Detection: These tests were performed using the successful results of the *Lives* tests. The number of lives that an object possessed after a forced collision was compared to a value that was one less than the original number of lives. A forced collision involved moving two objects to the same position. Certain collisions could not be tested as discussed in *Section 5.3*. The collisions that were tested were done according to *Figure 2*. This figure indicates the lives that should be lost upon relevant collisions. Any collision not indicated in the figure was tested to ensure that upon collision, there was no effect. The suite containing these tests is named `Collision_detection`.

5.2.12 Entity deletion tests: The testing structure was used to prove that all objects having zero lives, with exception of the player, were deleted from the vector of all moving entities. Lives of objects to be tested were set to zero instead of being deducted by a forced collision. This reduced the complexity of the tests. These tests were based on the success of the *Collision Detection* Tests. The suite containing these tests is named `Entity_deletion`.

5.2.13 Player Bullet Upgrade Test: The killing of three `Satellite` objects was simulated by constructing three `Satellites` and calling the destructor of each. Once these steps were implemented, a test was performed to prove that two `PlayerBullet` objects were created after a single simulated spacebar press. This implied that the `Player's` bullets were upgraded after shooting three `Satellites`. Additionally, a test was constructed to verify that upgraded `Player Bullets` were spawned with an angle of 5° between them. The suite containing these tests is named `Player_bullet_upgrade`.

5.2.14 Game State Tests: Testing was performed to verify that the game remained in a playing state until the player had zero lives left. These tests also proved that the game entered a lose state upon the player having zero lives left. The tests for the two game states mentioned were in the form of automatic unit tests. Testing for a change in game state to a win state proved to be complicated to implement as a unit test. The reasoning for the lack of tests verifying transition to a winning state is discussed in *Section 5.3*. The suite containing these tests is named `Game_state`.

5.3 Functionality not tested

The presentation layer of Pixel Potter was not tested because it was heavily dependent on the use of the SFML library. A major assumption in the testing of source code was that the SFML library had already been tested adequately throughout its development. This implies that the correct performance and results from the library can be expected. For this reason, the logic layer of the game was the only layer that was tested.

The testing structure had limitations in the testing of game logic. The limitations were due to the time dependent random generation of the `Enemy`, `EnemyBullets`, `Asteroids`, `Satellites`, `SatelliteBullets`, `Laser- Generator` and `LaserArc` objects. Testing the above mentioned entities by means of their random generation resulted in testing framework instability. This instability caused tests to run for unexpected periods of time. It was decided that in order for all tests to run as expected, testing of objects that were created randomly would be performed by brute-force creation. This would result in the random generation being bypassed and therefore, not being tested. Possible ways to test this behaviour is discussed in *Section 7.3*.

The brute-force method for creating the above mentioned objects was used extensively in the collision detection tests. Although this method improved the range of tests that could be written, some collision tests could not be performed. The collision cases that could not be tested with unit tests included: an `Enemy` object colliding with a `Player` object; a `LaserArc` object colliding with a `Player` object and a `LaserGenerator` object colliding with a `Player` object. As discussed in *Section 5.2.11*, collision cases were tested by moving the two objects to the same position. However, it was not possible to move a `Player` object to the same position as the above mentioned objects (due to the random generation of their initial angle) without the use of unstable tests. While these features were not tested using unit tests, possible methods for constructing these tests are proposed in *Section 7.3*.

The transition of the game into a winning state would

involve the simulation of killing 20 enemies, calling the relevant function and comparing the current game state to an expected game state. The difficulty in constructing a test for this functionality came about due to the fact that the `Update` function, which sets the winning state, was a private member function of the `Game` class. The trade off between effective information hiding and the ease of performing tests is discussed further in *Section 6.3.3*.

6. CRITICAL ANALYSIS

6.1 Game Functionality

In this section, a critical analysis of the functionality of the game is given.

6.1.1 Successes: All of the required functionality in order to obtain an excellent grade was implemented. This functionality includes all of the basic features, and well as two major and three minor features. The game contains all of the required entities (as presented in *Section 4.2*), and all of the required entities are able to collide with one another.

The `Player` object is able to move around the circumference of a circle based on user input. The `Player` object is also able to shoot bullets upon user input. When shooting, only one bullet is shot per a button press and release and the user is thus unable to hold down a button and create a continuous stream of bullets. This is essential to the game functionality, as it means that the user cannot win the game by simply holding down the key that creates bullet. The `Player` object has five lives, and these lives decrement by one upon each collision. Once all five lives have been lost, the game is lost and it ends. Once 20 `Enemy` objects have been destroyed, the game is won and it ends.

A further success of the system is that upon winning or losing the game, the user is able to restart a new game by pressing a button. This enhances the user's experience of the game as the user does not have to rerun the game each time the game ends.

The graphics implemented in the game are also considered to be a success. The entities get larger in size as they move towards the circumference of the circle, and smaller as they move towards the centre of the circle. Along with the visual scaling, the stored size of the entities are also scaled as the entity moves. This allows for more accurate collision detections as the collision detection is performed on the scaled size of the entity.

6.1.2 Weakness and Issues: An issue with the functionality of the game lies within the upgrade to the `Player's` gun upon killing three `Satellite` objects. The first time this event occurs, the gun is upgraded, and two larger bullets are henceforth fired, rather than one smaller bullet. However, if another set of `Satellite` objects is destroyed, the gun will not be upgraded further.

When designing the `Satellite` class, difficulty was experienced in making the satellites shoot towards the `Player`, as the bullets would, at times, shoot towards the centre of the circle and thus never collide with the `Player`. It was thus decided that a fixed angle would be set for the movement of the bullet to ensure that the bullets would always shoot towards the perimeter of the circle.

Lastly, the lack of animations of the entities is considered to be a weakness of the game. While animations were not essential to the functionality of the game, they would have made for a more enjoyable game experience.

6.1.3 Tradeoffs: A tradeoff was made in the choice of the collision detection mechanisms. While the implemented method works sufficiently, a more accurate method would have been to use the Separating Axis Theorem (SAT). The SAT is a collision detection method that is specifically used for rotated shapes as are used in the game [4]. However, the SAT algorithm is much more complicated to implement and thus a tradeoff was made between complexity and accuracy.

6.2 Code Design and Quality

This section provides a critical analysis of the code design and quality.

6.2.1 Successes: All of the implemented code makes use of modern C++, and C++14 features were used whenever possible. Wherever it is not possible to use C++14 features, C++11 features were used as an alternative. The code places a heavy emphasis on Object Orientated Programming.

In order to manage the entities, a vector of smart pointers was used. The use of smart pointers protects the game against possible resource or memory leaks [5]. The `std::vector` container was used exclusively within the code, as it allows for easy random access to elements within the container. Wherever possible, iterator-based loops were used over traditional index-based loops.

A complete separation of code layers was achieved, and the logic layer was completely decoupled from the SFML graphics library. This is regarded to be a major success since it means that the code can easily be reused with a different graphics library and only the presentation layer would need to be altered.

The implemented code base makes use of many min-

imal classes, that each have focused responsibilities. The use of large monolithic class was avoided in order to create code that was easy to reuse. Wherever possible, private functions were used in order to protect a class's information, and to hide the game's functionality from the user. When necessary, `getter` and `setter` functions were used to allow for conversations between objects, to avoid the use of public variables. Furthermore, to large extent, classes are not dependent on one another, leading to decoupling within the code. This is essential as it means that the code base can easily be adapted to include more entities.

The use of the inheritance hierarchy also constitutes a large success of the code design. This hierarchy helped to limit the amount of repetition that occured in the code, by grouping entities with similar functionality. Thus, the violation of the *Do Not Repeat Yourself* (DRY) principle was minimised within the code base.

6.2.2 Weakness and Issues: A weakness within the code design is the use of type codes within the collision detection.The consequence of this is that if a new entity is created, new code must be written in order to create collision checking cases for that new object. This same weakness is found within the strongly typed enumerator class `EntityList`. Since this enumerator stores the type of the entity, upon adding a new entity, the entity would need to be added to the enumerator.

An area where the DRY principle was violated is within the `Window` class's `fileLoader()` function. In this function, the same two lines of code are repeated for each texture that needs to be loaded. This is redundant and inefficient. The `Window` class is also a very long class, containing many functions. This class should have been divided into smaller classes.

Another noticeable violation of the DRY principle is within the movement of certain of the enemy entities (`Enemy`, `Asteroid`, `Laser Generator` and `Laser Arc`), and the `Enemy` and `Player` bullets. Since all of these entities move linearly and differ only in the incrementing or decrementing of their radius, inheritance should have been used in order to limit the repetition of code. This would also have made the code easier to modify, as changes could have been made only in the base class, rather than in all of the individual classes.

There is minimal error checking present within the game. The result of this is that if an unexpected error occurs at run time, the user's game experience will be greatly impaired.

A final issue with the code lies within the `Satellite` class. Since the `Satellite` entities are required to spawn in groups of three, it should be the `Satellite` class's responsibility to create three entities at a time. However, due to implementation difficulties, the responsibility was given to the `entityCreator` class. This same issue is present in the `LaserGenerator` class where the class does ensure that two laser generators are instantiated at a time.

6.2.3 Tradeoffs: The use of the type code within the collision detection presented a large tradeoff. While the type code made the collision detection easier to implement, it reduces the ease with which the code can be adapted and reused. Thus, a technical debt was incurred in which a short term, easy, but not necessary good, solution was implemented at the expense of increased work at a later stage.

6.3 Unit Tests

The following sections provide a critical analysis of the implemented unit tests.

6.3.1 Successes: The testing provided confirmation of the desired spawning and movement of all game objects. While certain functionality was unable to be tested, vast and comprehensive coverage of all aspects of game logic was achieved with high quality testing. The large number of automatic unit tests designed and created (110 in total) are an indication of this coverage.

A feature that enabled a large variety of automatic tests to be constructed was being able to simulate user key presses. This was performed by mimicking members of the `UserInput` enumerator class, and passing them to the `Player`. This was deemed as success as it eliminated the need for manual input to the tests, thus allowing the tests to be automatic.

The advanced use of the *Doctest* framework through the implementation of test suites is considered a success of the project. The use of test suites to group relevant tests together provided the ability to filter tests according to suites. This allowed for the running of isolated tests when code relevant to a group of tests was altered. Instead of running all of the tests every single time, only suites relevant to the altered code were executed.

6.3.2 Weaknesses and Issues: The bypassing of the random generation of the above specified objects lead to some tests being long and possibly complicated for external developers to examine. In addition to added complexity, the lengthy tests increased chances of possible bugs in the tests themselves. These bugs usually only became noticeable when tests failed after changes to source code infrastructure. This behaviour decreases the maintainability of testing structure.

Ideally, unit testing should be performed on every function of every class. Through implementation of

layer separation and information hiding within the source code, certain private member functions could not be tested because they were inaccessible to the testing framework. The inability to test and use these functions, resulted in further increases in length and complexity of the tests.

The collision cases, mentioned in *Section 5.2.11* could not be tested due to the random spawning direction of the mentioned entities. The difficulty in attempting to test these collisions was further enhanced by information hiding and layer separation of the source code.

6.3.3 Tradeoffs: There were two major tradeoffs that were present throughout the development of tests. The implementation of comprehensive coverage of all game logic came at a cost of having more lengthy and complicated tests. However, having increased coverage of the game logic improved understanding of game behaviour and provided opportunities for improving the source code efficiently.

Another tradeoff involved having enhanced layer separation and information hiding at the cost of decreasing ease of testing. Improving source code infrastructure was considered the priority to ensure that content was hidden and protected.

7. FUTURE IMPROVEMENTS

7.1 Game Functionality

For future improvements, the `Player` object and the `Satellite` objects should scale in size as they move around the circumference of the circle. This would allow for a more visually appealing game. Moving animations should also be implemented to improve the visual appeal of the game.

In order to make the game more challenging, the game should increase in speed as the elapsed time increases.

The game should also include a pause button to allow the user to pause the game as they wish.

7.2 Code Design and Quality

For future improvements, a class dedicated to loading different types of files should be created. These files should include images, sound files and font files.

A `LinearMovingEntity` base class should be implemented from where the linearly moving entities could inherit. This would reduce the repetition involved in the movement of the entities.

Finally, collision detection should be improved both in accuracy (by using the SAT) and in terms of reusability (by removing the type casting).

7.3 Unit Tests

While testing covered all aspects of the game logic comprehensively, certain features could be tested more sustainably in the future. One method by which such an objective could be achieved is through the use of a mocking framework. While Doctest 1.2.1 does not support mocking directly, the mocking framework can be implemented by integrating specific third party libraries [6]. The libraries that may be useful in this implementation include *googlemock* and *FakeIt* [6].

Through use of a mocking framework, aspects of game logic such as random behaviour and GUI interaction can be tested more efficiently than within a standard *Doctest* framework.

8. CONCLUSION

The design, implementation and testing of the game *Pixel Potter* has been presented. A breakdown of each class used in the game was given, with an emphasis placed on highlighting the responsibilities of each class. Important dynamic game behaviour was discussed, and a complete description of the implemented unit tests was presented. The game was then thoroughly analysed with regard to game functionality, code quality and test quality. The game functionality was deemed to be successful as all of the necessary features were implemented. The code was deemed to be of high quality, with successes lying in the use of modern C++, the inheritance hierarchy, and the complete decoupling layers present within the code. The implemented tests provided a comprehensive coverage of all basic movement and collision tests, as well as of important game logic. For future improvements, graphics and animations within the game should be improved. The inheritance hierarchy should be enhanced to reduce repetition and a mocking framework should be used to improve test quality.

REFERENCES

[1] V. Pereira. *Learning Unity 2D Game Development by Example*. Packt Publishing Ltd, August 2014.

[2] M. Barbier. *SFML Blueprints*. Packt Publishing Ltd, May 2015.

[3] "Command Line.", September 2017. URL `https://github.com/onqtam/doctest/blob/master/doc/markdown/commandline.md`. Last Accessed: 9/10/2017.

[4] J. Gregory. *Game Engine Architecture*. CRC Press, June 2009.

[5] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, May 2012.

[6] "Frequently Asked Questions.", September 2017. URL `https://github.com/onqtam/doctest/blob/master/doc/markdown/faq.md`. Last Accessed: 27/9/2017.

Appendix

A Code Structure and Implementation

Figure 3 depicts the UML diagram of the inheritance hierarchy that was implemented in the game. *Figure 4* describes the inter-class relationships, without the inheritance hierarchy.

Figure 3 : UML diagram showing the implemented inheritance hierarchy

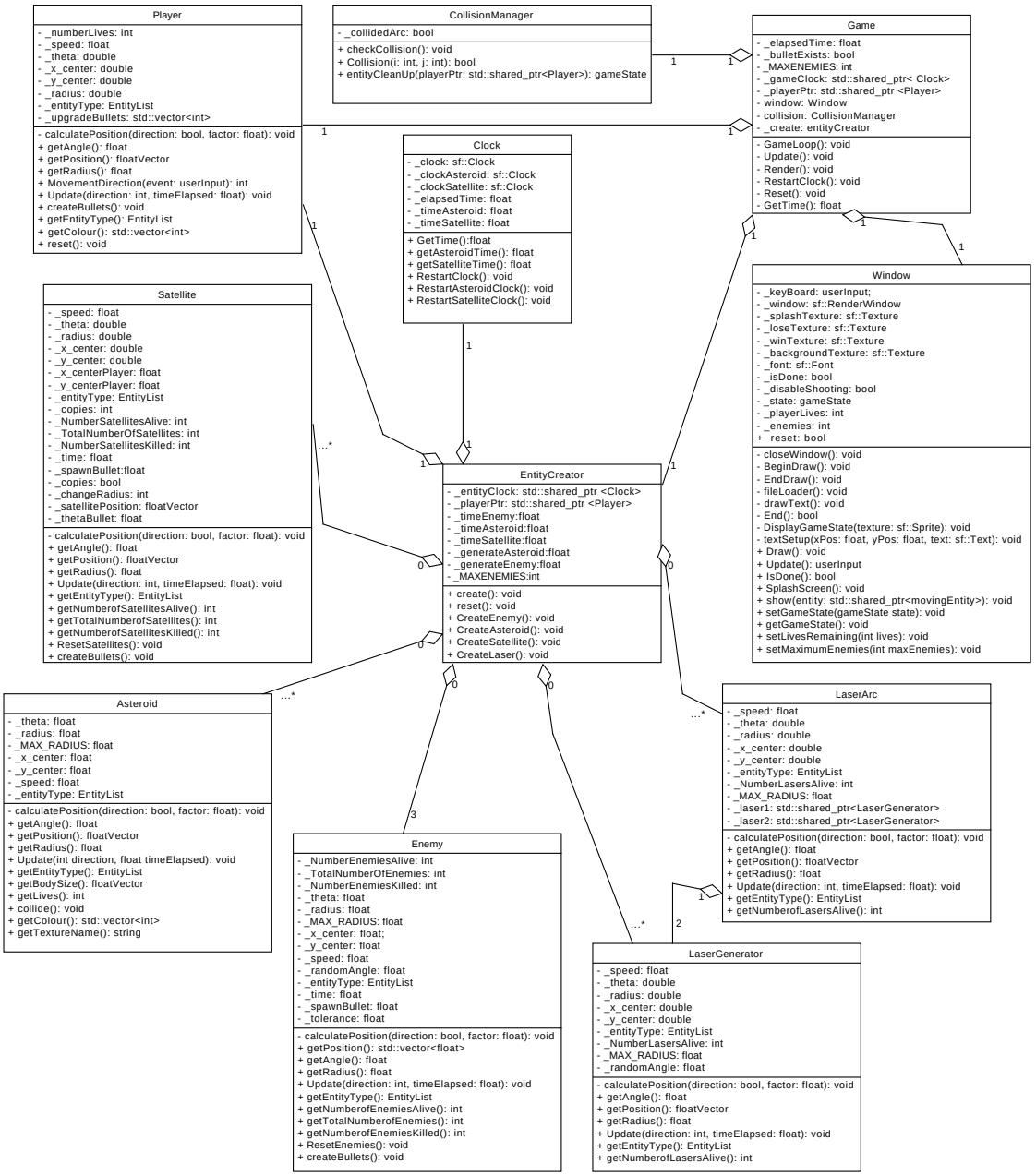

Figure 4 : UML diagram representing inter-class functionality

B Dynamic Game Behavior

Figure 5 provides a flow diagram depicting the main game loop. A sequence diagram of the main in-game object interactions is given in *Figure 6*.

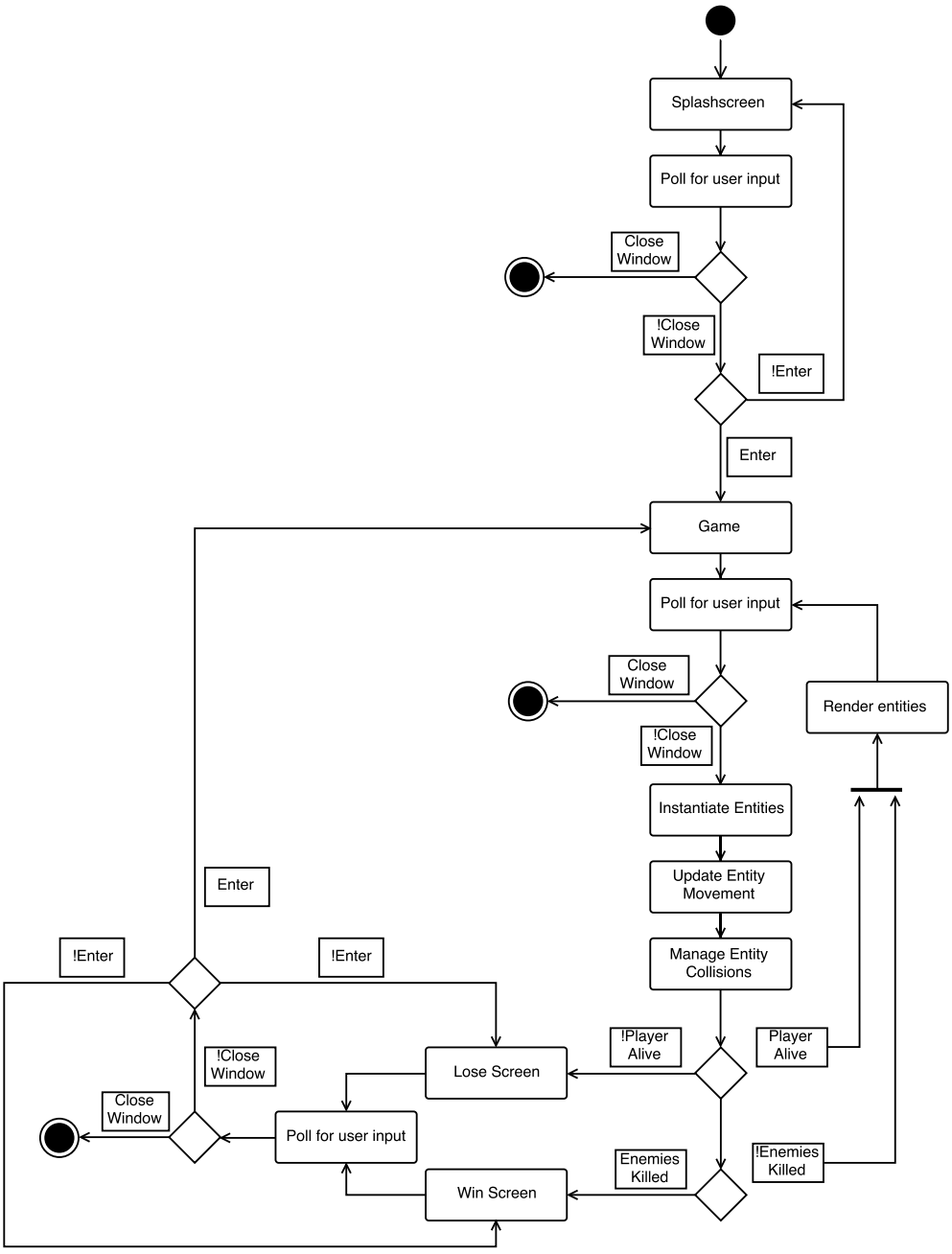

Figure 5 : Flow chart of the game loop

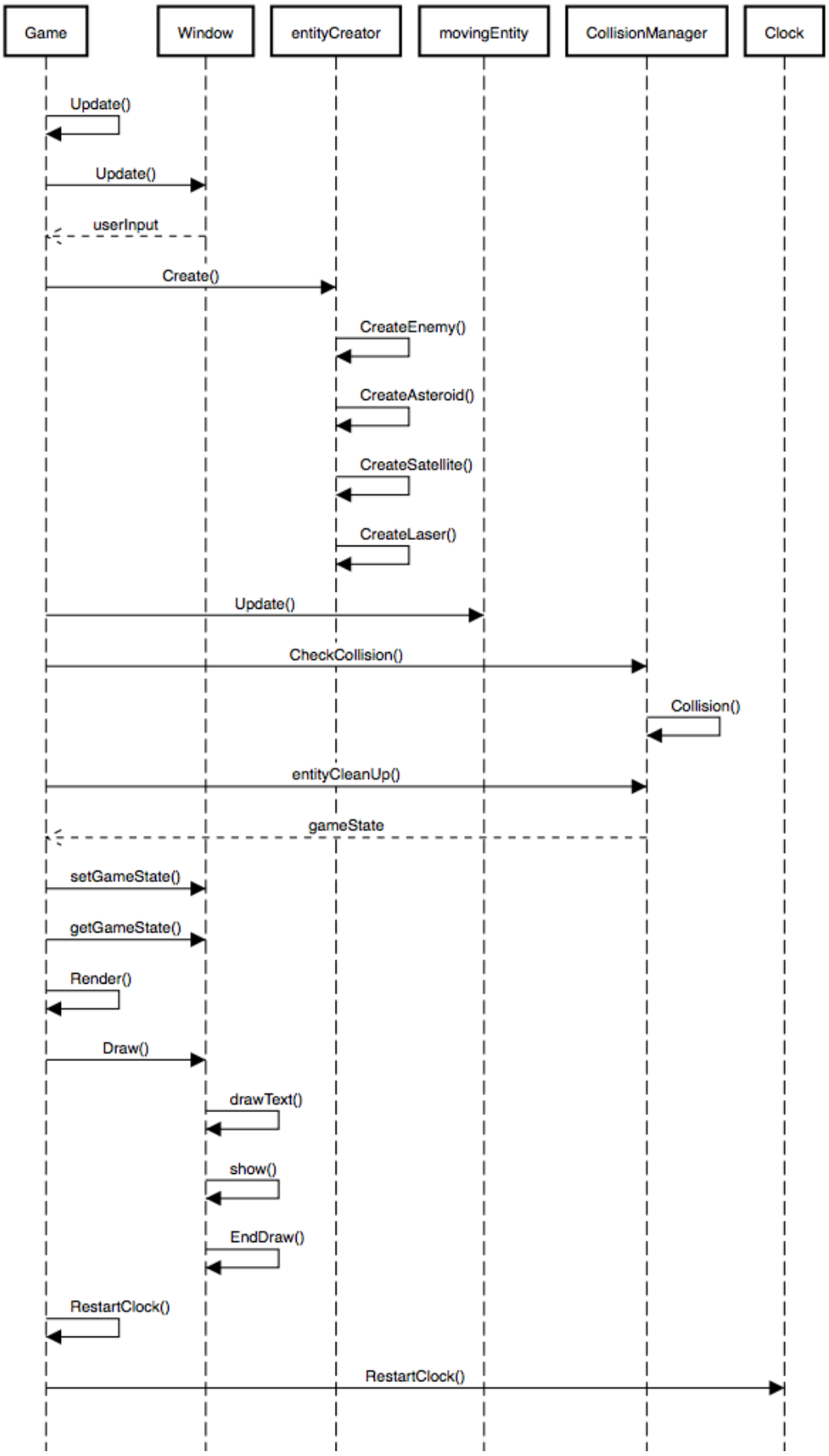

Figure 6 : Sequence diagram showing important objects interactions