Title

Sasha Berkiwitz (818737) & Lara Timm (704157)

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

Abstract:

Key words:

1. INTRODUCTION

The report documents the design and implementation of a computer game, *Shape Invaders*, created using C++14 and object-oriented programming. The basic game mechanics approximate those of the popular 1983 arcade game, Gyruss.

Contained within the following sections are...

2. PROBLEM DEFINITION

2.1 Shape Invaders Game Play

Shape Invaders, modelled on the arcade game Gyruss, is a C++14 developed computer game rendered using SFML 2.3.2. The game features a player character who moves around the circumference of a large circle. The player is capable of shooting at enemies, who spawn at the centre of the screen and travel outwards. The game is inhabited by a variety of enemies who are generated at random time intervals and at random angles. When colliding with an enemy or enemy bullet, the player looses a life. The player starts with five lives and the game ends when all of the players lives are depleted. Game speed increases with increasing time in game. As a result the objective of the game is to get the highest possible score rather than to destroy a limited number of enemies.

2.2 Requirements

The game contains the following basic functionality:

- A player ship, player bullets, more than one enemy ship and enemy bullets exist.
- Enemy ships appear at the circle's centre, move radially outwards and eventually move off the screen.
- The enemy ships fire bullets radially outwards towards the player. The player can fire bullets towards the centre of the circle.
- The game ends when the player is killed.

The game also contains the following features which enhance the game functionality:

- The player begins the game with five lives. The player loses a life when colliding with another game object. Remaining lives are displayed in the bottom left hand corner of the screen.
- A scoring system is implemented. The player is awarded points for destroying other game objects. High scores are saved from one game to the next. The player's current score and the high score are displayed at the top of the screen.
- The game has additional game objects including asteroids, laser-generators and satellites.
- Asteroids travel radially from the circle's centre directly at the player. Asteroids cannot be destroyed.
- Laser generators appear in pairs at the circle's centre and laser force field is generated along an arc between them. If the player shoots either generator, the force field collapses.
- Satellites appear in groups of three in front of the player. They gyrate in small circles and shoot at the player. If all three satellites are shot, the player's gun is upgraded.

2.3 Success Criteria

To be deemed a success, the game should effectively implement the aforementioned functionality. This will result in game which runs smoothly, is easy to understand and is challenging. The game should be constructed using object-oriented programming, make use of good programming practises and should follow the separation of concern principle.

2.4 Constraints

The final product must run on the Windows platform and be coded using the SFML 2.3.2 library. The game must display correctly on a screen with a maximum resolution of 1920×1080 pixels.

3. CODE STRUCTURE AND IMPLEMENTATION

On consideration of the design objectives in Section 2. and in an attempt to code using good coding practises, the separation of layers principle was a pivotal design consideration. The remainder of this section describes the structure of the implemented code.

The code is separated into three distinct layers: the data layer, application logic layer and presentation

layer. This was done in an attempt to implement the separation of concerns principle which will allow for decoupling of dependencies between classes. Communication between layers is carried out using object conversations, ensuring that only the necessary information is exposed to the other layers.

The data layer is handled by the `FileReader` class, application logic layer by the `Game` class and interface layer handled by the `Interface` class.

3.1 Domain Model

The game consists of various objects which move around the game area. These objects are identified as *GameObjects* and are listed below:

- Player
- Player Bullet
- Enemy
- Enemy Bullet
- Asteroid
- Laser Generator
- Arc Segment
- Satellite

All *GameObjects* are required to handle collisions with all other objects, must be able to move and must be drawable.

3.2 Data Layer Class Structure

This layer is responsible for the fetching and handling of data contained in files outside the Game executable. This data can then be provided to the game at runtime.

3.2.1 FileReader Class

This class functions as a simple file reader and writer. The class is responsible for reading in a previous high score from a text file, and in turn writing a new high score to the text file if the high score is beaten in the current game. The high score data is thus obtained by the data layer and passed to the presentation layer, via the application logic layer, so that the high score can be rendered on the screen in game. This process ensures that `FileReader` does not have direct access to the interface layer.

3.3 Logic Layer Class Structure

The logic layer forms the largest component of the game. This layer is responsible for the handling of all objects in the game, managing all in-game operations and launching and managing the game loop which controls all dynamic activity and game functionality. The layer can be divided into two main subsections. The first, logic, which manages the game operations and the flow of the game, and second, the domain classes, which manage all game objects and their interactions.

3.3.1 Game Class

This class is the most important class in the game as it controls all game logic. Responsibilities of `Game` include setting up all pre-game variables and requirements, instantiating all required game objects for the game and running the game loop. This loop interacts with the `Interface` by providing it with all of the information it needs to render the correct game state as well as the progression and updating of the game during play. The inverse relationship also exists. `Interface` provides `Game` with information regarding user inputs, allowing the game to be updated accordingly.

Other responsibilities of `Game` include updating object positions, keeping track of cooldowns, triggering new object instantiation when cooldowns expire, resetting cooldowns using randomly generated numbers and keeping track of the score and high score status. `Game` sets a new high score via the data layer's `FileReader` class.

3.3.2 GameObject Class

This class is the base class from which all game objects are inherited. `GameObject` contains the base parameters and functionality for all objects which have a position, can move, can collide with other objects and are drawable. Each derived `GameObject` thus has a position and physical attributes which enable the presentation layer to render the object as required by the game.

In addition to an object's position, each `GameObject` has a hit radius which enables the `Game` class to detect collisions between objects and flag them for deletion.

3.3.3 Player Class

This class represents the user controlled object in the game, inherited from the `GameObject` class. A `Player` object moves directly as a result of a keyboard input. This input is detected by the `Interface`, using SFML event polling, and is subsequently converted into logic based events which are passed back to the logic layer for interpretation. The `Player` can move clockwise or anti-clockwise around a large circle. This object is also able to fire a `PlayerBullet` through a keyboard input. A `Player` object's life is reduced when it collides with any other `GameObject`. When the `Player's` life is reduced to zero, the game ends.

3.3.4 PlayerBullet Class

This class represents an object inherited from the `GameObject` class which is spawned at the current position of the `Player` and travels linearly towards the centre of the screen, where it is deleted. A collision of a `PlayerBullet` with an `Enemy`, `LaserGenerator` or `Satellite` object results in both objects being destroyed. A collision of a `PlayerBullet` with a non-destroyable object (`Asteriod` and `ArcSegment`) results in only the bullet being destroyed.

Depending on the gun level of the `Player`, one to three `PlayerBullets` are fired by a single user input. The gun is upgraded when a `Satellite` array is destroyed.

3.3.5 Enemy Class

A class inherited from the `GameObject` class. This class represents an object which spawns at the centre of the screen and travels radially at a random trajectory angle towards the circle's circumference. A collision of an `Enemy` with the `Player` results in the `Enemy` being destroyed. There are an unlimited number of `Enemy` objects that can spawn throughout a game. These objects fire `Enemy Bullets` along their path vector towards the edge of the screen.

3.3.6 EnemyBullet Class

This class represents an automatically generated bullet, fired by an `Enemy` object, which travels along the same trajectory as its `Enemy`. `Enemy Bullets` inherit from the `GameObject` class. A collision of an `Enemy Bullet` with the `Player` results in the bullets destruction.

3.3.7 Asteroid Class

This class represents non-destroyable objects which travel from the centre of the screen towards the `Player's` current position. This class is inherited from the `GameObject` class.

3.3.8 LaserGenerator Class

This class is inherited from the `GameObject` class and represents an object which is instantiated in a pair and travels radially outwards. A pair of `LaserGenerators` is connected by a number of `ArcSegments` which travel in an arc between them. A collision of a `LaserGenerator` with the `Player` results in the destruction of the entire configuration. A collision of a `LaserGenerator` with a `PlayerBullet` results its destruction, as well as the destruction of all associated `ArcSegments`; namely the forcefield collapses.

3.3.9 ArcSegment Class

This class represents an object which forms an arc between two `LaserGenerators`, and is inherited from the `GameObject` class. A collision of an `ArcSegment` with the `Player` results in the destruction of the entire configuration.

3.3.10 Satellite Class

This class represents an object which spawns in an array of three directly in front of the `Player's` current position. `Satellites`, inherited from the `GameObject` class, gyrate in small circles and periodically fire bullets outwards. A collision of a `Satellite` with a `PlayerBullet` results in its destruction. The destruction of three associated `Satellites` results in an upgrade for the `Player's` gun.

3.4 Presentation Layer Class Structure

This layer is responsible for polling of all user inputs and managing and rendering to the game window. The majority of presentation layer functionality is dependant of the *SFML* library. This layer can be thought of as an interchangeable layer whereby any multimedia library could be used to replace it. SFML is used solely in the presentation layer allowing the application logic layer to function independently of it.

3.4.1 Interface Class

This class is responsible for polling user inputs and rendering of all objects to the SFML *RenderWindow*. User inputs are communicated to the logic layer via a vector of logic layer events to which all polled input events are added. Additionally, object conversations allow for game objects, passed from the logic layer to the presentation layer, to be rendered correctly to the window.

Other responsibilities of the `Interface` include rendering the `Player's` remaining number of lives, the current score and the high score to the window. The `Interface` also renders the game splashscreen and game-over screen depending on the current game state.

4. IN-GAME BEHAVIOUR

The game is managed by the `Game` class, more specifically by the game loop. This loop is described by the flow of operations seen in Figure 1.

The game begins by launching game-state one, the splashscreen. The `Interface` then polls for user input; the enter key progresses the game to game-state two and the escape key closes the window, ending the game. Game-state two (the playing game state) loops continuously while the window is open and the `Player's` number of lives is greater than zero.

The playing game-state proceeds in the following way. The `Interface` detects user input and the game is updated accordingly. New `GameObjects` are then created, collisions are detected, the score is updated and objects due for clean-up are deleted. If the `Player` object is deleted in clean-up, the end-game state is triggered and the game ends; otherwise the playing-state repeats.

4.1 Game Input Polling

User inputs are detected by the `Interface` using SFML event polling. The `ProcessGameEvents()` function populates a vector with `keyboardInputs` corresponding to specific keys pressed or released. In this way, only the vector of enumerations need be passed from the presentation layer to the application logic layer, allowing the logic layer to not have any dependencies of the SFML library.

User inputs include `Player` movement and the firing of `PlayerBullets`. By polling for both the press and the release of the fire key, a single bullet per keyboard input can be instantiated using boolean logic. When in the splashscreen and end-screen states, the `Interface` polls for start game and exit user input.

4.2 Collision Detection

Object collisions are handled by the `Game` class through the `CheckCollisions()` function. In this function, each element in the vector containing all existing `GameObjects` is checked against every other existing `GameObject` to see if a collision has occurred. A collision is identified as the distance between the objects' origins being less than the sum of their hit radii. If a collision occurs between the player and a destroyable object, the players life is decremented and the objects health is set to zero. If a collision occurs between the player and a non-destroyable object, only the players life is decremented.

This is a computationally intensive algorithm with complexity of $\mathcal{O}(n^2)$. Ideally a less taxing algorithm would have been used to improve the computation time of the game (see Section 7.2). Figure 2 demonstrates the collision relationships that exist between all game objects.

5. CODE TESTING

Unit testing plays an important role in the development of software as it assists in detecting issues in code early on in development. This section highlights the tests written and compiled during the development of the project.

A total of 53 test cases are compiled with 132 unit tests being run.

5.1 Testing Environment

Doctest 1.2.1, a C++ unit testing framework, was used to compile and write tests. Its function 'CHECK' was used to compare expected values and 'CHECK_NOTHROW' and 'CHECK_THROWS_AS' to handle error exceptions.

5.2 Base Functionality Testing Structure

Each class has an extensive set of simple tests written to confirm expected outcomes from each of their respective functions.

5.2.1 GameObject Class The `GameObject` class has no direct tests, however each of its member functions are tested as the classes mentioned below are inherited `GameObject` objects.

5.2.2 Player Class On start of the game, a `Player` object should be present on-screen. A `Game` object is therefore initialised and the above is tested. Its starting position is checked along with its movement function `MovePlayerObject()`, assuring that a `Player` object has the ability to rotate both clockwise and anti-clockwise around the screen. Its starting health of five lives is also tested.

5.2.3 PlayerBullet Class The instantiation of a `PlayerBullet` of the correct type at the player's position is tested. As a `PlayerBullet` moves inwards from the position at which it is instantiated, its movement in the correct x- and y-directions are tested in each quadrant, as well as along the four axes. The deletion of the bullet object upon reaching the centre of the screen is tested by looping its `LineMove()` function, inherited from the `GameObject` class, until the object no longer exists.

5.2.4 Enemy Class An `Enemy` object is instantiated at a random angle at the centre of the screen. The ability for such, along with the correct object type, is tested. Similar to the `PlayerBullet` object as mentioned above, the enemy moves outwards and its movement in the correct x- and y-directions in each quadrant and along the four axes is tested. Its `LineMove()` is looped through in order to test deletion once it is off-screen and out of scope.

5.2.5 EnemyBullet Class The instantiation of an `EnemyBullet` object of the correct type is tested. Its movement and deletion is tested as the `Enemy` class above.

5.2.6 Asteroid Class As the `Enemy` and `EnemyBullet` classes above, the `Asteroid` class is tested to be of the

correct type, as well as its movement ability and deletion when out of scope.

In addition, as the asteroid 'follows' the player, its angle at instantiation is compared to that of the player. Both the `Player` and the `Asteroid` make use of the `getAngle()` function inherited from the `GameObject` class.

5.2.7 LaserGenerator Class `LaserGenerator` objects are created in pairs, along with seven `ArcSegment` objects. Each are given a matching, unique `ID`.

The creation of the nine objects with their correct types are tested. The `ID` of each object is tested to be the same as the others in its set, and a second set is instantiated in order to check that their `ID` is different to that of the first.

The deletion of the objects moving off-screen is tested using the `LineMove()` function as mentioned above.

5.2.8 Satellite Class Similar to the `LaserGenerator` object, satellites are created in groups of threes, all having the same, unique `ID`. The creation of such, their correct type, `ID` and separate `ID` to that of another set is tested.

As the satellites 'follow' the player, their angles are compared to that of `Player` using the inherited `getAngle()` function, as with the `Asteroid` object explained above.

Lastly, on destruction of all three satellites in the set, the player experiences a gun upgrade with an extra bullet. These upgrades are tested by creating a bullet and counting the number of new `GameObject` objects added to the game.

5.3 FileReader Class

The `FileReader` class is tested through checking no exception is thrown when accessing the file storing high scores, and expecting an exception when trying to access non-existent files. These make up the data-layer tests.

5.4 Advanced Game-play Testing Structure

In order to check that collisions occur and that their outcomes are those which are expected, more complicated tests are compiled. These attempt to simulate game-play through the testing framework. This assures that the most important factors of the expected playing experience is achieved.

The following sub-sections describe the code written for each collision in order to achieve the above described.

5.4.1 Asteroid Collisions On a collision between an `Asteroid` and the `Player`, the `Asteroid` is deleted and the `Player` loses a life.

In order to force such a collision, an `Asteroid` object is instantiated and its `lineMove()` function repeatedly called until the number of objects in existence is decreased. This is possible due to an `Asteroid` object's instantiation at the angle of the player. The decrease in health of the `Player` is tested.

On collision between an `Asteroid` and `PlayerBullet`, the bullet is destroyed and the Asteroid remains unaffected.

The same as above can therefore be simulated, but with a `PlayerBullet` created in addition to the `Asteroid` and its movement function called. The Asteroid's health and bullet's deletion are tested.

5.4.2 Enemy Collisions A collision between `Enemy` and `Player` objects is more complicated as `Enemy` objects are created at a random angle. The position of the `Player` is therefore adjusted using its `circularMove()` function and the enemy's `getAngle()` function. Once in-line, the `Enemy` will collide with the `Player` upon use of a `lineMove()` function.

Using the same methodology as with the `Asteroid` object, an `Enemy-Player` collision is simulated and the Player's decrease in health tested, as is done with an `EnemyBullet`.

Additionally, a collision between the `PlayerBullet` and `Enemy` is forced. The objects' deletions are tested, along with the increase in `Score`.

5.4.3 LaserGenerator Collisions Similar to testing collisions with an `Enemy`, the player is moved in order to be in-line with a `LaserGenerator` or an `ArcSegment`.

A `Player` collision with the `LaserGenerator` is tested with relevant deletions and damage checked, as well as a `PlayerBullet` one with deletions and score updates tested.

Similarly, collisions with an `ArcSegment` are tested where, on collision with the `Player`, damage and deletions are tested and, on collision with a `PlayerBullet`, deletion of the bullet is tested.

5.4.4 Satellite Collisions Tests mentioned above are applied to collisions with a `Satellite`. In addition, the player's `GunLevel` is tested on deletion of

all three `Satellite` objects.

5.4.5 GameEnd Simulation The game ends and the `Player` object is deleted once it loses all its lives and its `Health` is reduced to zero.

In order to test this, a collision between an `Asteroid` and the `Player` is caused five times. The player's deletion and initialisation of a new `Game` is then tested.

5.4.6 HighScore Functionality Using the above methodology to destroy enemies and increase score as well as start new games, the game's high score functionality is tested through forcing separate scores and analysing expected results and changes.

6. DISCUSSION

6.1 Functionality

The game meets the functionality requirements of the brief set out in Section 2.2. Also the level of functionality meets the requirement to be rated within the excellent tier; namely, basic functionality, three minor features and two major features have been implemented (see Section 2.2 for detailes).

The group members' intention was to implement better graphics, rather than simple SFML shapes. Due to time constraints, simple shapes, identifiable by their specific shape and colour, are used . It must be noted that enemies do scale depending on their proximity to the circle's circumference.

In addition to the game functionality described, the game also has additional functionality which improves the overall playability of the game. The first feature is that the game speeds up as play progresses. This is done to increase the difficulty of the game once the `Player's` gun has been upgraded to the maximum level. Making the frame rate faster also makes the `Player` move faster, requiring the user to adjust their playing style as the game progresses. Another feature which improves the playability of the game is that every time the `Player` loses a life the screen flashes red. This makes the user more aware of their lives decreasing as well as instils a sense of stress in the player as if they themselves had been injured.

6.2 Code Structure and Implemented Design

An attempt was made to make use of C++14 functionality and keywords where appropriate. Thorough use of smart pointers, or more specifically `std::shared_ptr` was used throughout the code. Smart pointers provide a much simplified

6.3 Testing structure and Implementation

Sufficient basic tests were written to cover all relevant member functions of the classes in the game. In addition, more advanced tests were written to tests the occurrence of events such as collisions in the game.

It should be noted that the collision tests written are complicated and as a result may need testing of their own. This is a cost incurred for keeping functions used to set positions and angles of objects private, making one unable to code simple collisions. Additionally, as a result of these private functions, while loops were used in order to test objects moving out of scope. Both of the above were decided to be worthwhile tradeoffs for the program itself having cleaner and more secure code.

There were no tests written for the game's presentation layer due to time constraints. Rather, it was decided to rely on visual cues whilst running the game.

7. FUTURE RECOMMENDATIONS

7.1 Game Functionality

Although the game met the functionality requirements of the brief, some functionality was lacking substance. One key area which could be improved was the movement patterns of the `Enemy` objects. Making these objects capable of more complex movement patterns would make the game more dynamic and unpredictable, increasing the difficulty of the game and the challenge posed to the user.

As mentioned, better graphics would be implemented if time has allowed it. The intention was to make satire of the vegan stereotype where butchers would be the enemies throwing their meat products at the vegan player. "VleisInvaders" would be an instant hit in South African culture with the introduction of 'wors-fields' and much more. This would definitely pose improvement for future.

Additional improvements could be made to the `Interface` class. At present the frame rate is determined by this class as the use of `sf::Clock` was a convenient choice to flag how often the display should be updated. As this is a key aspect of the logic layer, this rate should be controlled by `Game` through the system clock and timers in C++. In the `RenderGameObject()` method, difficulty was experienced when drawing shapes that weren't of type `sf::RectangleShape`. This made rendering `Satellites` and `Asteriods` tricky. This method should be re-factored to improve rendering simplicity.

7.2 *Code Structure and Design*

In addition to the suggestions for improvement discussed in Section 6.2 additional improvements to code structure are mentioned below.

Separation of layers was a key factor to the designing of the code, yet it was only implemented after the second release of the game. This was a bad decision as it became difficult to move around large portions of the code and many functions that should have been private had to be public to achieve the desired functionality. It is recommended that in future coding projects key aspects should be considered from the beginning to prevent the need for code refactoring once much of the functionality has already been introduced.

In the current code there is minimal error handling, with the only error management being in the `FileReader` and `Interface` classes where files (scores, textures, fonts) are being loaded into the game from external sources. Minimal error handling could result in an unexpected error which could result in the game being unplayable. More error handling should be implemented to reduce these risks.

The chosen collision detection algorithm is computationally taxing as every existing object is compared to every other existing object. An alternative collision detection algorithm, such as the 'Separation Axis Theorem for Collision Detection (as discussed in [1]) could reduce the computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(cn)$, a far more favourable result for large numbers of objects.

7.3 *Testing structure*

At the time of final submission, there are no tests implemented for the game's presentation layer. Future versions of the game could implement tests checking the successful loading of graphics, etc.

Automated tests could also be added. By doing so, it would irradiate the need for the over-complicated tests currently used to detect collisions, and the results of user inputs could be tested.

Finally, with more features added to the game, other relevant, additional tests would be required.

8. CONCLUSION

REFERENCES

[1] Metanet Software Inc. "Collision Detection and Response.", 2011. URL `http://www.metanetsoftware.com/technique/tutorialA.html`. Last Accessed: 16/10/2017.

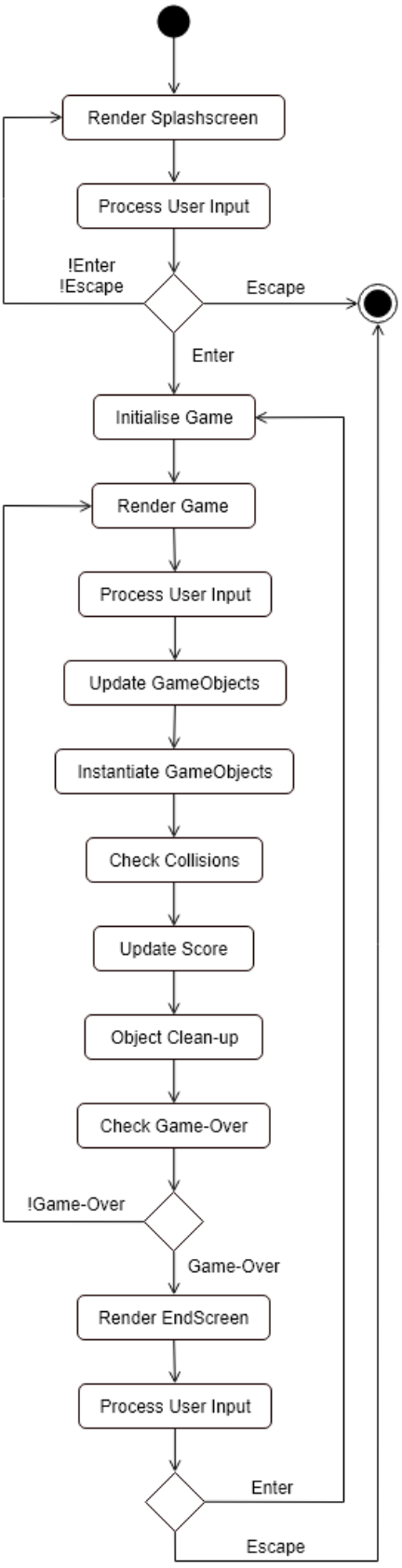

Figure 1: Flow Diagram depicting the process of events in the game loop.

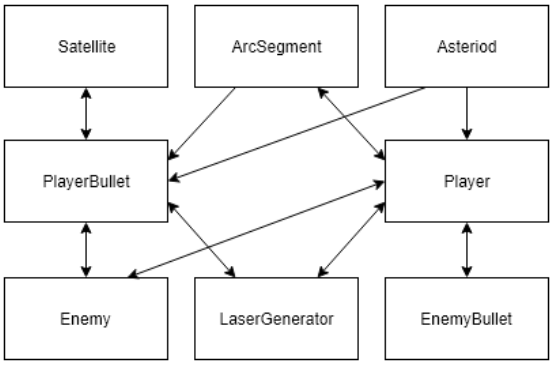

Figure 2: Illustration of the collision relationships that exist between game objects. Arrowheads indicate damage taken.