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; a 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. The code is separated in this way in an attempt to decouple the 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

The following requirements are imposed on all `GameObjects`: they must be able to handle collisions with all other objects, they must be able to move and they must be drawable.

3.2 Data Layer Class Structure

This layer is responsible for fetching and handling 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 the game high score from a text file, and in turn writing a new high score to the text file if a new high score is achieved in the current game. The high score data is 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 derived. `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 physical attributes (position, size, colour, etc.) 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. Details regarding the inplemented collision detection algorithm can be found in Section 4.2.

3.3.3 Player Class

This class represents the user controlled object in the game. 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 which is spawned at the current position of the `Player` and travels linearly towards the centre of the screen. 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, two or three `PlayerBullets` are fired by a single user input. The gun is upgraded when a `Satellite` array is destroyed.

3.3.5 Enemy 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`. When colliding with the `Player`, the bullet is destroyed.

3.3.7 Asteroid Class

This class represents a non-destroyable object which travels from the centre of the screen towards the `Player's` current position.

3.3.8 LaserGenerator Class

This class represents an object which is instantiated as 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 `PlayerBullet` with a `LaserGenerator` results its destruction, as well as the destruction of all associated `ArcSegments`; namely the force field collapses.

3.3.9 ArcSegment Class

This class represents an object which forms an arc between two `LaserGenerators`. 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. These `Satellites` 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 the `Player's` gun being upgraded.

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 on 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 `sf::RenderWindow`. User inputs are passed to the logic layer in a vector of `keyboardInputs`. The logic events in the vector correspond to all relevant `sf::Events` detected in that frame. The shape properties of all game objects are passed from the logic layer to the presentation layer to enable them 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.

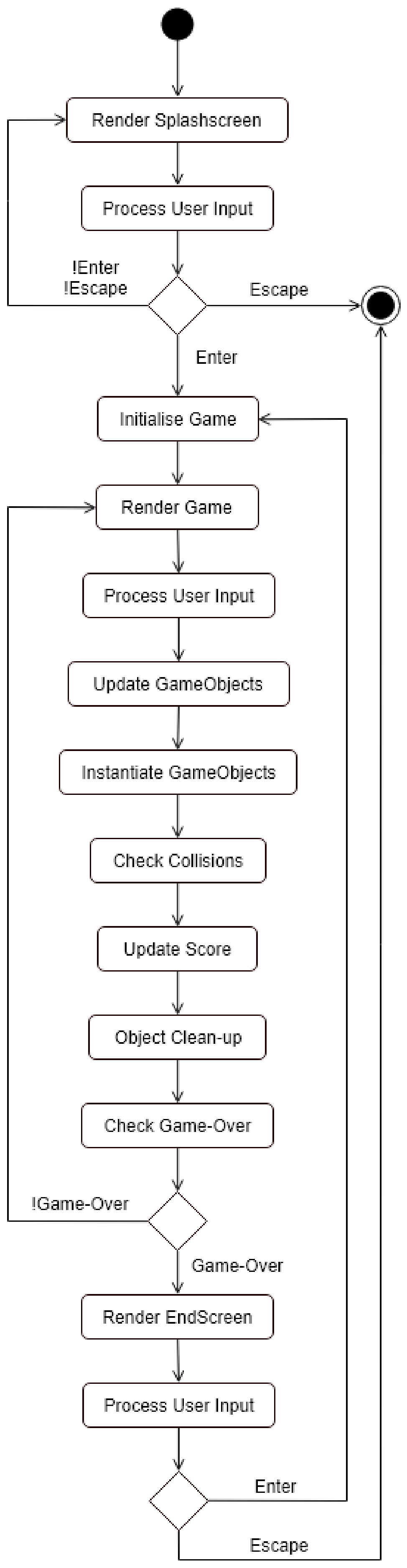

Figure 1: Flow Diagram depicting the process of events in the game loop.

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 control `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 `Player's` 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 `Player's` 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.

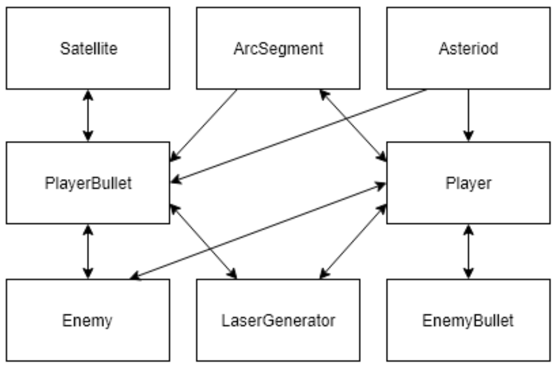

Figure 2: Illustration of the collision relationships that exist between game objects. Arrowheads indicate damage taken.

5. CODE TESTING

5.1 Testing Environment

5.2 Data Layer Testing Structure

5.3 Application Layer Testing Structure

6. DISCUSSION

6.1 Functionality

The game meets the functionality requirements of the brief, set out in Section 2.2. The level of functionality achieved is sufficient 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 details).

The group members' intention was to implement proper graphics, rather than simple SFML shapes. Due to time constraints, simple shapes, identifiable by their specific geometry and colour, are used. Scaling of objects is implemented whereby all moving objects scale depending on their proximity to the circle's circumference.

In addition to the game functionality described, the game has extra features which improve 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, which seems fairly easy 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 modern C++11/C++14 features, where appropriate, to improve the quality of code. Smart pointers, or more specifically the `std::shared_ptr`, is used throughout the code to mitigate memory management problems experienced with traditional C++ pointers. The `auto` keyword was used where appropriate to allow the compiler to define types dynamically. Also, the expression `for(auto element:vector)` is used to iterate through vectors without the use of indices or iterators. The `GameObject` class makes use of `virtual` functions which are defined for specific derived objects using the `override` keyword. This ensures the compiler knows to make use of the object defined implementation.

In the current code implementation, the presentation layer handles some functionality which does not fall within its expected responsibilities. In this way, the separation of concerns principle is not fully adhered to. The data layer should be responsible for loading the game textures into the game at runtime, rather than the `Interface` class. A class which loads all SFML required files, which may include textures, fonts and audio files, should be implemented. `Interface` also makes use of the SFML clock to keep track of time and control the frame rate. This should be the responsibility of the logic layer as this level of game logic should not depend on the interface in any way.

A success of the `Interface` class is its efficiency. All textures are loaded only once, upon interface instantiation. Also, the background animation is rendered using a single sprite-sheet. In this way, only the sprite rectangle needs to updated every frame rather than its texture. This reduces the computational complexity of rendering during play.

Throughout the code, the STL `vector` container is used. This choice was made in an attempt to reduce computational complexity as much as possible while still having the code remain intuitive. Throughout, indices are used rather than iterators. For this container, a complexity of $\mathcal{O}(1)$ exists for access using indices and adding objects to the end of the vector using `push_back()` [1]. Deleting objects randomly from the vector however introduces a complexity of $\mathcal{O}(n)$, as every iterator after the deleted object has to be reassigned [1]. Almost every collision results in an object being deleted from the `_GameObjectsVector`, and thus the trade-off between intuitiveness and computation time was made. For this implementation, deleted objects are most often near the beginning of the vector and thus deletions are very inefficient.

Within the `Game` class, bad code smells have been introduced. This class is monolithic and needs to be broken up into smaller, more abstracted classes with prioritised functionality. These smaller classes could perform functions such as collision detection and timing to reduce the complexity of the logic class, making the code more understandable. It is also noted that the `spawnGameObject()` method contains a large switch statement which depends on the type of `GameObject` being spawned. This is also a bad code smell and should be re-factored for better implementation.

The `GameObject` class is used as the base class from which all objects are derived. `GameObject` provides its child classes with the attributes and functionality they require. Regarding the `linemove()` function, as well as getters and setters, inheritance has been used for code reuse. This introduces tight coupling between the base and derived classes, yet is implemented as such to follow the DRY principle.

In an attempt to further the DRY principle, static `shared_ptr` downcasting is used to access methods which only exist in specific derived class implementations. Only `Satellite`, `LaserGenerator` and `ArcSegment` objects are required to have an ID, and thus these parameters should not be inherited by all `GameObjects`. In addition, only the `Player` object is required to have an `UpgradeGun()` method. By making use of static pointer casting, it is ensured that the compiler determines if any rules are violated before run-time, thus preventing run-time errors.

To prevent code reuse through inheritance and pointer typecasting, a composition relationship may be preferred to an inheritance relationship. Derived object classes would instead contain a `GameObject` object, as well as their own parameters and methods which would be easily accessible. This would however require more than one container for all game objects and all code would need to be re-factored.

All variables in the `GameObject` base class are protected, rather than private. Although these variables are only accessible by their derived classes, the `GameObject` class cannot preserve any invariants. A better alternative would be to have private data members and protected functions that access these variables. Overall, the `Game` and `Interface` classes make good use of private data members and functions, and provide public access to the methods that other classes need to make the game function properly.

6.3 Testing structure and Implementation

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.

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 `Asteroids` 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 re-factoring 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 [2]) could reduce the computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(cn)$, a far more favourable result for large numbers of objects.

In an attempt to further reduce computational complexity, an alternative to vectors, the `STL list` and `iterators` should be used for all containers from which objects are deleted. This will reducing the complexity of all required functionality to $\mathcal{O}(1)$, which would significantly reduce computation time.

7.3 Testing structure

8. CONCLUSION

REFERENCES

[1] J. Ahlgren. "STL Container Performance.", October 2013. URL `http://john-ahlgren.blogspot.co.za/2013/10/stl-container-performance.html`. Last Accessed: 16/10/2017.

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