SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITSWATERSRAND

ELEN3009 - Software Engineering II

Lecturer: Dr. S.P. Levitt

Project 2017 - Gyruss

17 October 2017

THE DESIGN AND IMPLEMENTATION OF A FIXED-SHOOTER ARCADE GAME

Tyson Cross 1239448 Christopher Maree 1101946

Abstract

This report presents the design, development and refinement of a computer game in the "shoot 'em up" genre written in C++14 and using the SFML v2.4.2 framework in an Object Oriented philosophy of software design. Design decisions are discussed, including the Object Oriented and inheritance models utilized. The overall structure of the code is outlined, along with specific class role modelling, examining the interactions between game objects and different layers of responsibility. The maintainability and orthogonality of the code is appraised, along with the important role of unit testing in developing the software. The advantages and disadvantages of the close coupling with the SFML framework in the solution is discussed. The specific design concerns involved in programming a video game with an emphasis upon visual quality, immersion and computer graphics are compared to more general software projects. The report critically analyses the software implementation and discusses possible design alternatives.

CONTENTS

1 Introduction

Gyruss is a classic arcade game, released by Konami Industry Company Ltd in 1983. This report's project implemented a variant of the original game with the same basic play mechanics. In the built game the player's ship moves around the perimeter of a circle by moving left or right on the keyboard. Enemies spawn from the centre or perimeter of the screen and attempt to kill the player by shooting at them, or colliding with the player ship. The player can shoot a weapon towards the centre of the circle, and destroy enemy ships for points. Enemy creation ("spawning") and translation around the screen is generated with varying patterns and behavioural states. Movement and shooting use randomness and unique events to add challenge and variety to gameplay. Points are awarded for surviving without being hit for long periods, and the entire game slowly increases in speed, the longer the player stays alive. There are a variety of enemy types, including satellites which spawn in sets of three, and if destroyed, grant the player an upgraded weapon, which is lost upon player death. The player loses the game if they die three times, and is victorious if they successfully kill 100 enemies before losing all their lives.

Game design and creation for computer, mobile and video games is a diverse commercial practice with a wide range of creative approaches, and formal methodologies that has significant overlap with the broader discipline of Software Engineering[1]. Video games are particularly concerned with the real-time display of computer graphics and player input, to build the simulation of an imaginative and satisfying experiences providing rewarding and challenging interactions. Video games can have complex technical requirements, incorporating the interaction of logic, demanding visual effects and graphics, and large amounts of data. Designing solutions to solve these problems intrinsically benefits from an engineering approach[2]. This report will begin with the high level conceptualisation and modelling of the domain of the problem, followed by an overview of the code structure, examining the classes and types of classes that make up the solution.

2 Modelling the Game Domain

The gameplay is modelled as a collection of game entities (specifically, visual objects) moving into, around, and out of a constrained volume of space. This volume is conceptualized as a cone, extending into the distance. The apex of the cone points away, into the screen, and the base is aligned parallel to the image plane as in Figure 1. A player ship rotates around a circle of fixed radius on the image plane, shooting projectile entities along the sides of the "cone" into the distant centre, in response to player input from a computer keyboard. Different entities (capable of more complex movement, and independent of player control) spawn in specific areas of the game space, smoothly following varying movement patterns. These enemy entities create projectiles that move outwards along radial lines from the centre towards the player perimeter. When the game objects overlap in space, they have "collided" and none, one or all of them can be affected by this interaction.

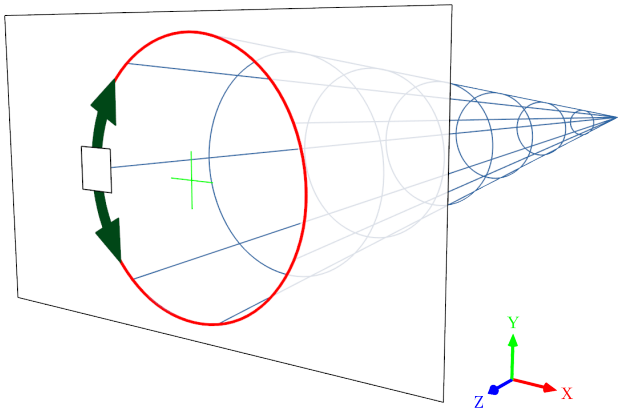

Fig. 1: Conceptualization of game space

The domain of the software is considered the spatial interaction of these game entities, and the interaction of the entities that are generated by the game objects (projectiles, explosions, etc.) The "game" can be in a series of states, one of which is the actual playing, involving the interaction of the game entities. Gameplay is delineated by preparation for play (with instructions,) and the end of play with the outcome of the game (win or lose). These additional states are static, providing information about the gameplay experience. The player, who is the primary actor interacting with the system, must then choose to exit the game or replay it.

Value objects that hold information about the domain events are defined, such as a "score" which holds the numeric values associated with specific events. "Lives" could be considered a value object which counts the number of certain domain events (such as a player ship object colliding with another game object entity). A factory object called `EntityController` is given the responsibility of creating and controlling all non-player ship entities. Helper objects of a template class called `ResourceHolder` are provided to load and transfer asset resources to game entities.

`Entity` states are queried and then each entity is instructed to perform actions of their responsibility in a fixed sequence of domain events, to produce the behaviour and interactions making up gameplay. The sequence of commands are:

1) Set the anticipated future (planned) state of an object, based upon the current knowledge of the entity.
2) The controllers query the current interaction of the entities, and depending on the logic, cause domain events. Based on this, the entity knowledge is modified, and entities can be removed or spawned. New entities are created with knowledge of their next planned action.
3) Any existing entity's planned action is now performed, and all game objects are updated to their planned state.
4) The entity is then visually drawn and then displayed, to inform the player of the new state.

This set/check/update/draw sequence occurs on a fixed timestep, a delta which is measured by the main controlling object Game by comparing a timer to a value threshold[3]. If the timestep is reached, then the timer is reset, and the inner game loop actions are performed. This temporally uncouples the player input from the frame rate and display procedure.

A summation of this sequence can be viewed in the game loop diagram in Figure 2.

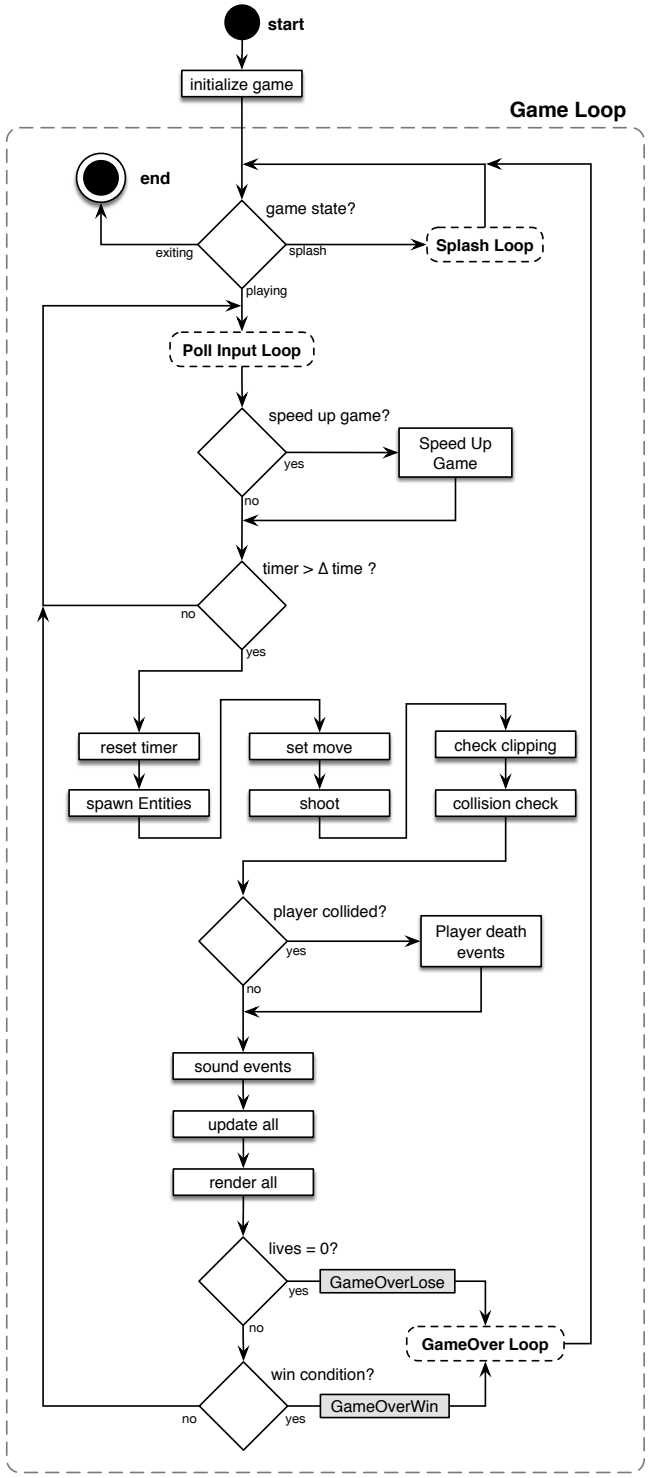

Fig. 2: Main game loop

3 Overview of Code Structure

The game classes and objects can be broken up into three main categories, namely: entity objects, which are conceptual objects with a single identity; helper objects that mediate between with entities and controllers in varying contexts; and object controllers that define how the entities interact with one another.

4 Entity Objects

Entity are individual atomic objects with a specific identity and a set of responsibilities.

4.1 Entity

The `Entity` class defines a parent base class from which the majority of game objects are inherited. This abstract, virtual class defines a common interface that other entities must implement and override. This is an "is-a" relationship, where both class and interface inheritance occurs. For example, a `PlayerShip` "is-an" Entity.

An `Entity` will be transformed spatially and represented visually. This requirement, and the associated knowledge domain are encapsulated in the `Movable` and `Animatable` classes, which Entity inherits, as seen in Figure 3.

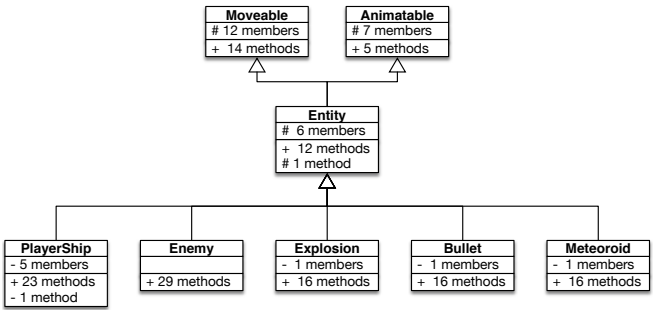

Fig. 3: Entity inheritance model

4.2 Moveable

The virtual `Movable` superclass defines the interface and knowledge to perform translation through polar coordinates. All dimension alterations (including rotation and scaling) are the responsibility of `Movable`. Before a move is executed, a `setMove()` is performed to define the future angle and or position of the entity. Later on an `update()` function is called, whereby the actual move occurs. Several derived classes of `Entity` have unique implementations of the movement functions, based on the type and role of the game object. These implementations will be discussed later on in this report.

4.3 Animatable

This virtual superclass, which `Entity` inherits, defines the members and methods for visually representing the game objects. A simple video game such as this project can be viewed as an exercise in real-time graphics, moving animated pictures (representing the entities) around the screen, based on the gamestate and player input. SFML offers a lightweight and efficient object, Sprite, which contains dimensional and visual knowledge encapsulation. The fast and elegant functionalities offered by this sprite object's methods, facilitates a highly convenient and performant mechanism of combined representation and presentation. The decision to move the sprite object directly into Entity couples the presentation of the object to its representation (and puts the display knowledge in the same context of logical responsibilities.) The consequences and limitations of this design decision are discussed later in the report in the critical evaluation section.

4.4 PlayerShip

A derived class from `Entity`, a `PlayerShip` object is used to represent the player ship within the game. `PlayerShip` objects are capable of moving around the

game screen, rotating at a fixed speed around the player movement circle or remaining still, based on player input. The `PlayerShip` object also stores its current location, scale, angle and distance from centre as private member variables (inherited from its component Base classes). The object knows if the player's gun is currently upgraded, or shooting, and the number of lives the player has left.

4.5 Enemy

The `Enemy` class is derived from `Entity`. Similar in implementation to the `PlayerShip`, the class provides mechanisms to move the enemy ship around the screen. Rather than only providing rotational angular movement, the `Enemy` class is also capable of setting an angle and radius (polar coordinates) to define its movement. The scale of the enemy ship is calculated based on the distance the enemy ship is from the centre of the screen.

The `Enemy` interface also extends the `setMove()` method to specify an origin of rotation for itself. This enables the enemy ship to perform rotations around a defined origin separate from the geometric centre of the screen (the default centre for all other entities). This is useful with enemy ships that do not spawn at the centre or edge of the screen, such as satellite enemies. This extended method allows an enemy ship to perform complex movement patterns by altering the angle and radius while also shifting the centre of rotation. This can be beneficial when more complex patterns are wanted that do not follow a simple spiral or trivial rotation. The specific movement patterns implemented will be discussed in the `EntityController` section of this report.

`Enemy` also has a set of member variable timers to store properties used to define its behaviour. Using clocks to control behavioural changes allows for easy balancing of game difficulty, distributing shooting or particular behavioural states between enemies more evenly throughout time. This helps avoid difficulty spikes inherent in randomly chosen behaviour, which could result in "unfair" gameplay for a player in an unlucky session. For example, each enemy stores how long it has been alive, and the time elapsed between bullets being fired. These are used in `EntityController` to define procedural enemy events and behaviour patterns.

The main `EnemyShip` class encompases four different varieties of enemy ships, `Basic`, `BasicAlternate`, `Satellite` and `Wanderer`. All four variants are controlled with the aforementioned methods. Each type also has its own sprite texture to visually differentiate it from the others to the player.

Enemy ships move and rotate around the play area, translating smoothly around the screen space, moving into and out from the distance. In order to orient the sprite texture that represents the game object to the direction of travel, two additional attributes are stored, that record the previous and current position of the object. These are each stored as 2D vector in Cartesian coordinates. On each move, these two positions vectors are translated to a common origin so vector operations can be performed on them, The new position vector is subtracted from the old position vector, resulting in a third vector. Using the `atan2()` function on this third position vector (the direction that the enemy is pointing) converts the pointing position vector into polar coordinates in radians. Subtracting the previous position vector (also converted to a polar angle in radians) and then subtracting the underlying angle property of the object, results in an angle of orientation that correctly rotates the sprite object to the direction that the sprite is travelling in. As a result, the enemy ships point in the direction they are flying while moving around the play screen.

4.6 Bullet

Inherited from `Entity`, bullets are a simple objects that can be both enemy or player generated. The object constructor defines the spawn angle and radius with respect the to geometric screen centre as well as a flight direction. When the bullet's `setMove()` is called, a distance is provided that defines how far the bullet should move when `update()` is called. The bullet then flies in the flight direction that was specified upon construction. This speed value is passed to the bullet object on each frame. To enable the game to scale the speed of bullets flight as time progresses. As with other non-`PlayerShip` entities the bullets scale non-linearly as they move towards (or away from) the screen centre. Additionally, bullets cycle through a set of animation frames while alive. Different textures tilesets are used for player and enemy bullets.

4.7 Meteoroid

Similar to `Bullet` in implementation and inherited from entity, Meteorites are invulnerable objects that spawn from the centre of the screen and fly outwards, slowing spinning. As with bullets, they move at a provided speed, enabling scaling as the game progresses.

4.8 Explosion

Explosions are graphical animations derived from `Entity` consisting of 15 frames. On construction, a location is specified by angle and radius, defining where the explosion occurred (with respect to the game screen's geometric centre). The explosion object iterates through its animation frames and the object is removed after the animation is complete. A single life/die mechanism is used to provide this functionality.

4.9 Shield

The Shield class does not inherit from `Entity` as its implementation is too simple to justify the number of implemented classes that the base class requires, which would result in many refused bequests. A shield object is visually displayed as a semi-transparent, animating texture on top of the `PlayerShip`, and indicate 1.2 seconds of invulnerability after the player ship respawns. This behaviour is to ensure fairness, so that the player cannot be accidentally killed by a bullet, enemy or meteoroid immediately after spawning.

4.10 Starfield

The `StarField` class is a singleton object used to generate visuals giving the impression of speed, and flying through outer space. It has no direct impact on gameplay, but contributes to the overall experience, and helps with visual immersion in the game, and adds visual richness and complexity to the game-playing experience. The effect is achieved through modelling a rectangular volume of

space, roughly the area of the screen multiplied by a fixed depth, which is populated with pseudo-randomly generated "positions" (a struct made up of three float coordinates). These positions are then pushed forward each frame, with a simplified perspective projective technique. The points are "moved" along their positive Z axis (defined as toward the image plane, away from the maximum depth plane). If any points would be moved past the image plane (into a positive Z value), they are moved back to the maximum depth plane. If the points are behind the maximum depth plane, then they are made invisible (scaled to 0).

The points are moved outwards in X and Y as the Z value moves closer to the image plane. A single `sf::CircleShape` without texture is drawn at each point on every frame. The shape is scaled (and its colour multiplied down to black) by a scaling factor determined by how far a star is from the centre/depth. This is the same method used to scale and dim each individual Entity sprites in the main game, detailed later under the discussion about graphics and visual effects.

In addition, certain stars rapidly change colour randomly on each frame, producing a subtle rainbow effect as the stars fly past, stylistically emulating a redshift/blueshift effect at "warp speed". The advancement of the star movement is deterministic as with all other entity movement and updates, stepping forward step by step as each frame is drawn and incremented. The speed is also tied to the global game speed modifier, so the stars fly past faster and faster as the game speeds up, and resets back to the starting speed after a player death.

4.11 HUD

The `HUD` class is responsible for holding the text objects which are drawn as an overlay on top of the display, showing useful information during gameplay. Important data such as player score and number of enemies needed to be killed to achieve victory is fetched from the main `Score` object and shown to inform the player. The `HUD` also displays the current `PlayerShip` state to the user, which can be `Alive`, `Invulnerable`, `Upgraded` or `Dead`.

5 Helper Objects

Helper objects are an informal clustering of objects that play an assisting role to entities or controllers.

5.1 Score

The `Score` object (which `Game` has as a private member) is responsible for tracking the player score and point-generating behaviour in the game. When the player kills an enemy, the score object is told, and it records the event. Based on the kind of enemy killed, the player is awarded a different number of points. The score object also tracks the players longest life and calculates the accuracy of fired bullets over an entire game. Relevant game statistics are passed to the game `HUD` for instant feedback, and also passed to `GameOverScreen`.

5.2 PerlinNoise

`PerlinNoise` is used to generate smoothly interpolated deterministic noise patterns. This is used as one of the main methods of moving enemy ships around the screen in `EntityController` as well as some subtle animations on the game's splash screen.

5.3 SplashScreen

When the game initially starts up, a splash screen is presented to the user. This shows the previous game high scores, game controls and instructs players on the point allocations and rewards associated with killing different enemy types. This object manages the generation, frame loop and draw events of itself. Based on the user input, the screen either proceeds to starting up the main game or exits the game. `ScreenSplash` inherits from the `Screen` base class.

5.4 GameOverScreen

At the end of a game (either from victory or loss of all lives) an end game screen is shown. Based on the outcome of the game, a different graphic is presented along with a win or lose sound. Statistics regarding the player's performance are also shown, such as number of enemies killed, player shoot accuracy, longest life, and total score generated from the `Score` object. The player can at this point choose to close the game or play another round. `ScreenGameOver` inherits from the Screen class.

5.5 ResourceHolder

Binary game resources can be "heavy"[4] and take up a large amount of memory, involving file I/O operations which can affect performance. All game textures are loaded into memory using an object of `ResourceHolder` class. The class is a template, and can be instantiated as various types as needed, for example, of `type sf::Texture`, `sf::SoundBuffer` or `sf::Font`. At game startup all required game resources are pre-loaded (and allocated unique pointers) to improve performance and avoid possible stuttering during actual gameplay, which could happen if the resources were loaded upon demand at time of need. This also makes the loading of resources fully deterministic, occurring and completing at a fixed time in the program flow.

The resource holder uses a map between an enumerator of resource names and resource locations on disk. When an object requests a resource, they call `get()` on a `ResourceHolder` object. The resource holder then returns a unique pointer to the object, transferring ownership of the resource to the caller.This means a resource-requesting object does not need to know the name or file path of the resource. The enumerated list of resources exists in a type-specific namespace, and is defined in the `common.hpp` file (a general source file for small, general functions and namespace enums used across the project.) The `ResourceHolder` class also decouples the responsibility of loading game assets from the objects that use them. This means that if the resource are missing or corrupted, the ResourceHandler deals with the error (and throws an exception), rather than the requesting object.

5.6 SoundController

The `SoundController` is responsible for playing all game sounds. A `SoundController` object has a `ResourceHandler` object of type `sf::SoundBuffer`,

responsible for preloading sounds from disk into memory, which can be transferred to a requesting object, and then played when needed. In order to identify specific sounds, a `SoundController` object calls `playSound()` with a parameter of type `sounds::ID`, from an enum in the `sounds` namespace. The `soundController` member object of Game also plays a looping track of game music during gameplay. The sound and music assets are stored on disk in the OGG format.

5.7 InputHandler

All user inputs are processed by the `InputHandler` object. This separates concerns between game logic and user input. The primary need for player input is to control the movement and shooting of the player ship. An `sf::Event` object is passed from `Game` into an `InputHandler` object during a polling loop. Relevant move (or shoot) attributes are set on the `PlayerShip` according to the correct keypress events being detected during the event window of time. In order to make the game more challenging, each shoot event for the player requires an individual pressing of the shoot button (the space bar). Holding down the space bar between `Event` polling (or between individual frame updates) should only result in a single bullet being fired, not a continual stream of bullets. To achieve this, a separate boolean member is used when polling for space-bar pressed, comparing the state of the button on the previous frame, and recording the current state of the button. This ensures that a bullet is only fired if the button is released and then pressed again.

An `InputHandler` object also captures quit events in the form of a hotkey combination `Ctrl+Q`, which sets the game state to Exiting (this state is also set by closing the game window), which will result in the game ending.

The `Inputhandler` and event polling loop occur even if the frame has not advanced, and the amount of time that has passed is known by the `InputHandler` (`Game` provides a pointer to the time elapsed). Once a frame is drawn, the `PlayerShip` is moved by an amount multiplied by this time amount, so even if there is a slowdown in frame rate, the player keypresses faithfully translate into a correct amount of movement, or shooting event, and temporally decoupling input events from frame rate.

6 Entity Controller

`EntityController` is a factory class responsible for managing the creation, movement, interaction and destruction of all non-`PlayerShip` game objects derived from the `Entity` class: enemies, bullets (player and enemy), meteoroids and explosions.

6.1 Entity Object Containers

Every entity is stored in a list of unique pointers: when new entities are spawned, a unique pointer of type Entity is added to the relevant list data member on entityController. Each entity type (`Bullet`, `Enemy`, `Meteoroid` etc.) is added to a specific list named accordingly (i.e. `_enemies`, `_explosions` etc.). Note that the polymorphic inheritance hierarchy allows the type to be of the base class (`Entity`).

For example, when an enemy bullet is fired, it is appended to the `_bulletsEnemy` list. When an entity "dies", it is removed from its respective list, releasing the unique pointer. This facilitates simple collision detection comparison and entity drawing. When collisions need to be checked, every entity in one particular list can be compared against the entities in another list. If a collision is detected, and both objects are destroyed, and then the entities are removed from their respective lists.

The obvious choice of container is vectors or lists. The data structure needs to perform three tasks:

1) Insertion of spawned entities
2) Deletion of destroyed entities
3) Iteration over the structure (for collision detection and rendering)

Both containers have dynamically allocated size for storage. Lists provide constant time insertion, deletion and iteration in $\mathcal{O}(1)$. Vectors perform insertion and deletion in linear time $\mathcal{O}(n)$, shifting each element with insertion and removal, and they requiring resizing when their capacity is reached. Vectors also provide constant time iteration, but lists are a better choice in this context due to the high rate of insertions and deletions that will occur during gameplay. The constant time deletion of lists is achieved by moving the adjacent pointers to the next element after the removed object and releasing the pointer to the deleted object (similarly with insertion).

6.2 Entity Spawning

The `EntityController` defines when, where and how many entities spawn. There are four types of enemy ships, each with with associated initial movement states and spawning position. Each movement pattern has a different chance of spawning, using associated `sf::Clock` timer objects, which distribute the behaviour among spawned enemies using pseudorandom determination.

The four kinds of `Enemy` starting movement states are as follows:

- Spiraling outwards from the screen centre:
  - Increment both angle and radius
  - Spawn every $\sim$ 2 seconds
- Spiraling inwards from the edge of the screen:
  - Increment angle and decrement radius
  - Spawn every $\sim$ 3 seconds
- `Satellite` movement, hovering opposite the player:
  - Spawns in groups of three and circle around a new non-geometric centre
  - Spawn location is opposite side of game screen play relative to player
  - Radius of circle increase causing satellites to move apart in rotation
  - If the player kills all three satellites, the player's gun is upgraded
  - If the playership dies, so do the satellites
  - Spawn every $\sim$ 20 seconds

- A `Wandering` ship that moves around the screen with no discernable pattern:
  - Spawn on opposite side of screen to playership and move with random coordinates, driven by psuedo-random, smoothly interpolated Perlin noise.
  - The noise seed is predefined so pattern is repeatable
  - Current enemy alive time defines values for 2D Perlin noise
  - Spawn every $\sim$ 6 seconds
- Unique spawning state for meteoroids:
  - `Meteoroid` spawn flying outwards from the screen centre from a random angle
  - No control logic or other properties defined
  - Spawn every $\sim$ 18 seconds

In addition to spawn timers, the `EntityController` object also defines a maximum (8) and minimum (1) number of enemies that can (or must) be alive at any point in time and spawns enemies accordingly (or is prevented from spawning, if the maximum number of enemies are alive, so as not to overwhelm the player completely). Based on the ship spawn state, specific textures are assigned to the `Animatable` inherited sprite object of the enemy ship, so the player can differentiate between movement patterns.

6.3 Enemy Ship Movement

The `EntityController` is also responsible for defining how enemies in different movement states should move. The different movement states are executed through the use of the general `setMove()` function that in turn calls the `setEnemyMoveState()` function. This sets up the future move on each object. Once the enemy move state is set, the enemy ship is moved at the correct time, using the `setEnemyMove()` function, which calls `move()` on each object, performing the actual translation.

Enemies have a chance of changing their movement state, calculated as a random chance on every frame. The defined movement states, (and the chance enemies have of changing to this state) are:

- `CircleOffsetLeft` (5% chance):
  - If ship circle radius is below threshold size: increase circle radius and offset circle to the left
  - If ship circle radius is above threshold size: circle radius remains unchanged and offset centre to left
  - The angle of rotation is incremented to turn the enemy ship in the direction of travel
- `CircleOffsetRight` (5% chance):
  - If ship circle radius is below threshold: increase circle radius and offset circle centre to right
  - If ship circle radius is above threshold: leave circle radius as is and offset centre to right
  - Angle is continuously incremented to rotate the enemy ship in the direction of travel
- `SpiralIn` (2% chance):
  - Decrement the current ship radius to spiral inwards
  - Angle is continuously incremented to rotate the enemy ship in the direction of travel
- `SpiralOut` (2% chance):
  - Increment the current ship radius to spiral outwards
  - Angle is continuously incremented to rotate the enemy ship in the direction of travel

There are two defined regions in which `Enemy` movement state can not change, seen in Figure 4. The first is a small region near the centre of the screen that ensures small ships spiralling outwards can continue to advance until they are large enough to interact with. The second is a cylindrical region at the perimeter of the player movement circle. In this outer area, enemies exiting the screen can continue along their path and are then clipped accordingly. If the enemy ship is not within these regions then there is a possibility that it can change its movement state. These restrictions only apply to enemies that spawned at the edge or centre of the screen (and are also inapplicable for satellites or wanders).

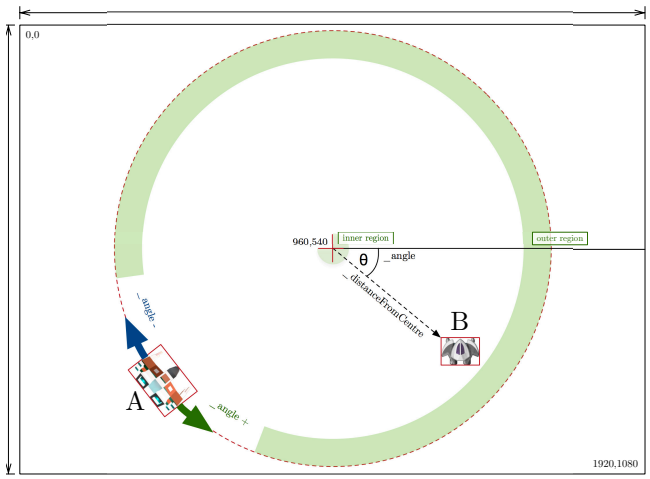

Fig. 4: Screen polar coordinates and the restricted regions for enemy ship behaviour

If the enemy ship is spawned as a satellite or in the wandering state, then it will remain in a single movement state until they are removed from play. A satellite's movement state is defined as "`small circling`" and a ship following the `PerlinNoise` movement pattern is defined as "`Wandering`":

- `SmallCircling`:
  - When spawned, these ships have predefined centres of rotation at coordinates on to the opposite side of the play area from the `PlayerShip` location
  - Each satellite rotates around this spawn point and the radius gradually increases in time, resulting in the satellites flying away from each other
- `Wandering`:
  - On each frame, an X and Y `PerlinNoise` value is calculated. These are then scaled and used to assign to radius and angle of the `PlayerShip` for the next ship movement
  - This results in a purely random yet smooth wandering motion of the ship that differs visually from the other enemy ships

6.4 Game Object Clipping

A desirable constraint is that enemies should not be able to leave the active area of gameplay, and should not be able to reach a position where they cannot be shot by the player (although they should be able to collide with the player.)

When enemies or bullets reach predefined edges of the play area, they are "clipped" and then their position is reset, or they are removed, if bullets or meteoroids. There are two regions where clipping occurs, when an enemy moves outwards from the play circle or inwards towards the distance centre:

1) `EnemyShip` reaches the edge of the play screen. Then the enemy ship is reset, spiralling outwards from the centre of the screen. Bullets that fly off the edge of the play region are removed, and later the unique pointer to the object is released.
2) Alternatively, if an `EnemyShip` is flying inwards, it become tiny at the centre of the screen. For gameplay balance, if a Enemy ship recedes sufficiently far in the distance, becoming too small to see or shoot, then it is reset into the `spiralOutwards` movement state. The same logic is applied to player bullets that fly inwards to the centre of the screen.

6.5 Enemy Bullet Firing

Enemies could shoot a bullet every frame, if they have not shot any bullets after a particular period of time. The `EntityController` object defines a range within which the enemies do not shoot. When shooting occurs, the enemy bullets are spawned from the location of the enemy ship. The bullets move directly outwards towards the edge of the play screen. This is achieved by assigning a positive distance value to `_futureMoveValue`. This float value is added the current bullet's distance from centre, when moved by the update/move step.

6.6 Player Bullet Firing

The `EntityController` is also responsible for spawning player bullets when the player shoots. To achieve this, the `EntityController` watches the state of the `PlayerShip` object and executes a shoot event when needed. The `_isShooting` state that determines if a player bullet is created or not, is set on the `PlayerShip` object by the `InputHandler` object, based on keyboard input.

6.7 Update Events

The last function of `EntityController` is to call update on all game entities to perform execution of movement and advancement of animation frame. This is achieved by iterating through each respective list of entities and calling `update()` on each entity in turn. Given the small variety of entity types (there are 5: `_enemies`, `_bulletsPlayer`, `_bulletsEnemy`, `_satellites`, `_meteoroids`) and considering the specifications of the project, this is a reasonable code burden to individually call without designing a cascading, automatic or procedural approach, such a scene graph structure for game entities, or an observer/listener relationship with an `EventHandler` class.

All functionality defined above for the `EntityController` is called from the main game loop on every frame. The sequence of commands for each layer is outlined in the game loop diagram in Figure 2 and a detailed breakdown of the individual flow of control can be followed in the sequence diagrams in Figures A1 and A2.

7 Game Class

7.1 Initialization

The Game object instantiation procedure, called from `Main.cpp`, creates all the private member objects required for knowledge, timing and control of the gameplay, sets the initial Game state appropriately, and then begins the primary game loop. In this loop, which advances the frame by a fixed frame rate, once a time threshold of $1/60^{\text{th}}$ of a second is reached, the Game object is responsible for calling all individual steps to advance the game logic in a specific sequence, as shown in Figure 2. The main game object is responsible for 4 primary functions:

1) Instantiating singleton objects of type `PlayerShip`, `EntityController`, and individual `ResourceHolder`s (for fonts, sounds and textures), an `InputHandler` and a `Score`.
2) Initializing update procedures on all game objects on every frame in specific order
3) drawing each valid and existing objects to the display window on every frame
4) transitioning between game states based on game events, such as the player winning or losing.

7.2 Game Updating

`Game` updating occurs on every frame. `Update()` is called on the `EntityController` object, which calls `update()` on each individual `Entity` in its appropriate list.

7.3 Drawing the Game

On each frame, after the update step, the memory address of each entity lists is passed from the `EntityController` to Game, where each respective entity sprite is drawn to the screen. This separates the drawing logic from the gameplay logic layers, and the underlying data and knowledge representations of the individual objects, which are the individual responsibility of each entity.

7.4 Game Speed

The game slowly speeds up over time. This occurs by the game object comparing a timer against a threshold, and then instructing the `EntityController` object to increase a speed modifier value. This increase the movement speed of all game entities except the `PlayerShip` object. It also increases the rate at which enemies spawn. This slowly makes survival more difficult for the player, encouraging them to shoot enemies as fast as possible, to win the game before the difficulty increases beyond the player's reflexes.

7.5 Game State

The main game object has an object which holds the state of the game, and continually checks conditions that change the game from one state into another. Various events and conditions alter the state. For example, when the player's number of lives is zero, the game transitions from Playing to `GameOverLose`, or if the 100^{th} enemy is shot, then the game state becomes `GameOverWin`. If the window is closed at any point, or if `Ctl-Q` is pressed, then the `gameState` becomes `Exiting`, and the program ends. The main game object is in one of five states at any point in time:

- `Splash`:
  - Shown when the game starts up initially
  - Based on player input, start the game or exits
- `Playing`:
  - Main game loop, all high level steps advancing the game, and drawing to the screen 60 times a second.
- `GameOverLose`:
  - If the player loses the game, go into this state
- `GameOverWin`:
  - If the player wins the game, go into this state
- `Exiting`:
  - When the user quits the game, go into this state

8 Graphics and Game Assets

Most of the game art, sound and music assets were sourced from creative commons, GPL or free online websites, as credited in the code and in-game credits. Additional game assets such as the splash screen imagery were created for the project. Audacity v2.1.3 was used to edit and modify sound effects. The player ship texture and the spinning earth planet texture on the splash screen were rendered in an educational version of Maya v2018, and composited in Foundry's Nuke v11.02, and further manipulated in Photoshop CS6 v13.06. Effects such as the player ship's jet engines were rendered in SideEffect's Houdini v16.0.504.20. Image tilesets for the animated assets were processed in Gimp v2.8.10.

8.1 Visual Effects

Non-player ship entities scale in size as they move inwards or outwards from the screen's geometric centre. This gives the game the impression of depth and distance. To achieve the feeling of enemy ships flying out from the distant centre vanishing point, towards the player ship (located on the image plane), scaling was implemented which simulates perspective. This is achieved by subtracting the game object's radius r (distance from from screen centre) from the maximum radius R (the player circle), and then dividing this value by the maximum distance from centre to get scaling factor d, as seen in Equation 1

$$d = \frac{R - r}{R} \tag{1}$$

This scaling value is also multiplied by the underlying screen object scale (the parameter provided to the `Entity` constructor and stored as the private parameter `_scale`.)

The scaling factor that is applied to entities as they move inwards and outwards from the centre of the screen is also applied to the `Animatable` objects' sprite's `sf::Texture` property. To accentuate the falloff in scale with distance, the object's overall sprite colour is also multiplied by the scaling value, darkening the texture of the game object as it recedes from the image plane. In real life, perception of saturation diminishes with distance but SFML does not easily provide a framework for manipulating colour as Hue/Saturation/Luminance [HSL or HSV], but instead uses an RGB colour encoding scheme. However, a correlation has been observed[5] between perception of object depth and luminance. Humans perceive decreased depth with diminished contrast of objects against their background[6]. The visual backdrop of the game is black, so darkening the sprite textures towards RGB [0,0,0] as they move towards the centre also results in a lower luminance contrast, leading to the perception of these dimmer sprite textures being further away.

8.2 Texture Animation

`Animated` tile-sets are achieved by offsetting and moving the sub-rectangle of a texture for a sprite, on each frame. With small visual changes of the pixels in the sub-rectangle between frames and a sufficiently fast frame rate, the illusion of smooth motion is achieved on game objects such as flaming bullets, a pulsing jet engine for the player ship and the cycled animation for the explosions. The images use PNGs with transparency to breakup the underlying rectangular sprite shape with more complex edges.

A special effect is implemented when upon player death where for a short burst of five frames (approx $1/20^{\text{th}}$ of a second) the screen shakes slightly, to emphasis the player death. When the player respawns, they are invulnerable for 1.2 seconds. To visually indicate this brief period of invulnerability, an animated shield object is drawn on top of the player ship sprite. In addition, the HUD also displays the player status (Alive/Dead/Invulnerable) at all times.

9 Program Flow and Object Collaborations

A principle employed throughout the project structure is "tell, don't ask". The game objects discussed are autonomous entities with responsibilities for their internal methods. The game controllers, such as the `EntityController` provide the logic to "tell" these objects when and how they should behave. The entities in turn understand how to interpret these instructions and to act accordingly based on their own knowledge.

The interaction between game objects on different levels of communication is represented in the sequence diagram shown in Figure A1. This diagram shows how different game objects communicate, with the controller objects initiating and interacting with the other game objects. This diagram also shows the passing of information and data between different layers. A more detailed version of this sequence diagram can be seen in Appendix I. This diagram shows the program flow of control moving between the game and `EntityController` objects, this is a technique referred to as inversion of control, where the control moves between the main controlling object (`game`) and a dependent object (`EntityController`).

9.1 Dependency Injection Paradigm

`EntityController` needs to know about `PlayerShip`, but the close and central relationship between of `InputHandler` and `PlayerShip` means it is more desirable and more flexible to have these objects created or owned by Game directly. In the presented solution, `EntityController` has a private member variable reference of type `PlayerShip` that is passed to `EntityController` on construction and assigned to a private reference data member. This relationship, shown in Figure 5 is known as dependency injection via constructor [7], [8]. `PlayerShip` and `EntityController` are created in `Game.cpp`, in a specific order. `EntityController` requires `PlayerShip` to exist in order to function. Injecting

the dependency via constructor enforces a clearly defined order of instantiation, and makes the code more readable, as one can quickly view class dependencies from the signature without needed to look at the class implementation. Using a reference data member means the pointer cannot be reassigned, and can not be a `nullptr`.

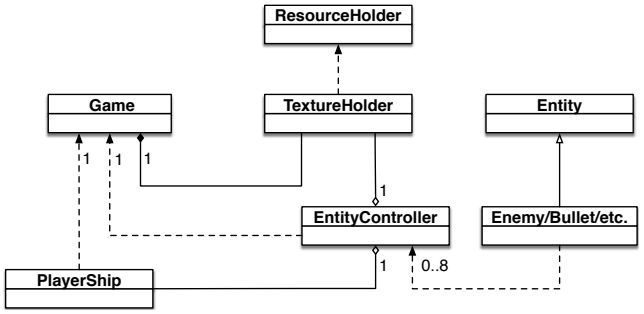

Fig. 5: Object dependency relationships

9.2 Collision Permutations

The `EntityController` is directly responsible for checking collisions between game objects. On each frame, collisions between relevant lists of objects are checked. These interactions are are outlined below.

- Player bullets collide with an enemy ship:
  - both objects are destroyed in an explosion, the player bullet is removed from its container (the pointer to the object is released) and the enemy ship is marked for removal
- The player ship collides with an enemy ship:
  - both the ships are destroyed in an explosion
- An enemy bullets collide with the player ship:
  - player ship explodes, enemy bullet is removed
- A player bullet collides with a meteoroid:
  - player bullet is removed, meteoroid is unaffected
- The player ship collides with a meteoroid:
  - `PlayerShip` explodes

The possible collision combinations are depicted in the Figure 6. If a collision results in the ship being destroyed, then an Explosion object is spawned at the location, visually emphasizing the event to the player. If `PlayerShip` is involved in the collision then the player loses a life, and the game speed resets back to the starting speed.

9.3 Collision Detection Algorithm

Because all the visual game objects entities that collide are roughly square, with the texture predominantly in the centre of the image, the collision detection technique used is a simple radius comparison. For each of the two objects being compared, the current scaled width and height (using the `getGlobalBounds()` method from the inherited `Animatable sprite` object) are separately added. This value is then then divided by a constant factor to obtain the average radius from the centre of the object, on screen. This basically models a circle around the centre of each of the scaled objects at their screen size, slightly smaller than the outer rectangular sprite shape. The radii of the two objects are added, and this value is compared to the distance between the objects' centres (using Pythagoras, from the object centre's

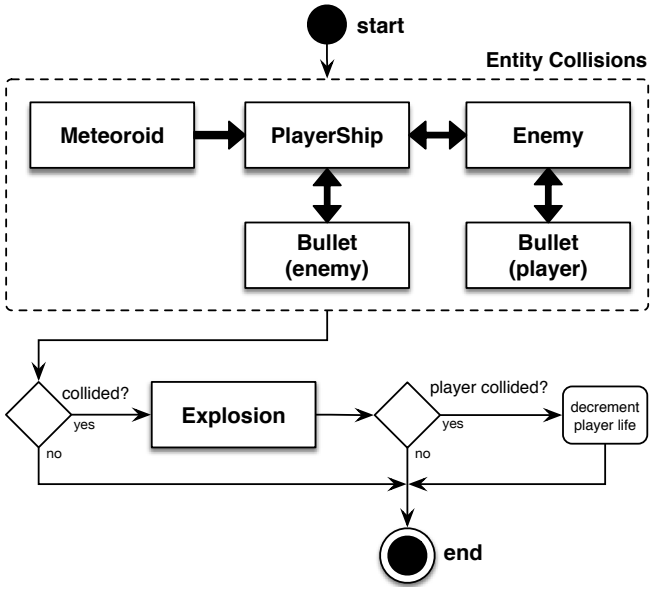

Fig. 6: Entity collisions

X and Y positions on screen). If the sum of the radii is equal or larger to the distance between the two game objects on screen, then the sprites must be sufficiently overlapping to have collided. This logic is implemented in a general `collides()` function in the `EntityController` class.

10 Program Unit Testing

Unit testing was done to verify the correctness of the majority of object operations under a variety of conditions. Key application functionality was tested with automated doctest unit tests which are summarised in Table I. Doctest is a modular testing framework originally developed for the Python language. The C++ variant provides a powerful and efficient testing framework for the C++11 standard, conveniently distributed as a single header file[9].

TABLE I: Unit tests summary

Object	Tests	Assertions
Object Resource Tests		
TextureHolder	6	10
FontHolder	3	4
Total	**9**	**14**
Object Movement Tests		
PlayerShip	9	20
EnemyShip	9	18
Bullet	4	6
Meteorit	4	5
Shield	2	4
Explosion	1	2
Total	**29**	**55**
Object Management Tests		
Collision detection	2	2
Score	6	7
Common functions	5	5
EntityController	6	12
Total	**19**	**26**
Grand Total	**57**	**95**

The public interface of each respective class was tested to ensure that the expected outputs were generated in different

conditions. All moveable game objects were thoroughly tested to verify that the underlying objects correctly executed the intended translation.

Not all game functionality was tested: some mechanics involving temporal randomness was omitted from tests. To test a random event, such as spawning enemies after a random period of time, a wait function is needed, until the random event occurs. This could result in unit tests that do not all run in a reasonable period of time. It is possible that the tests would never complete (although biasing the random value and using a seeded pseudo-random engine could help reduce this possibility.) Due to the randomness of the outcome of these functions, testing the correctness of the outcome becomes difficult as it is hard to predict the expected result.

To deal with this issue and to make the code as testable as possible, separation was enforced between the random functions and the underlying functions that are called. This means that a particular function deals with the generation of random events, such as when enemies are spawned, and another function is responsible for performing the underlying logic, such as spawning the enemies. In this way, the relationship between the random process and underlying executed logic has been separated, enabling tests to be written for the underlying logic without needing to test the random process. At the same time, this introduces a better separation of concerns and encourages a more functional design paradigm.

Some logic embedded within the main game object was not tested. In order to be able to successfully test complex interactions of mid-game logic, full execution of multiple variables of the game up until a specifically desired state is required. This is not an appropriate or feasible use of a unit testing environment, without the use of a mocking framework. A mocking framework could have been implemented to simulate object states at particular points in time to test the underlying logic. Project time constraints and complexity of implementation prevented this from being done.

Graphical interactions at specific points in game play were tested, with a small number of interactive GUI tests. They were compiled as separate executables to the main test executables, and were primarily created to test the correctness of the `ScreenSplash` object under different user input, such as pressing space bar at game start or closing the main window and checking that the object correctly responded.

10.1 Development Paradigm Clashes

Achieving well encapsulated code involves appropriately concealment of object information. This means that many of the primary object functions are kept private, data members are hidden if possible, and accessors and mutators are limited if they are not needed. By contrast, Test Driven Development [TTD] aims to test the public interfaces of objects to ensure each and every function generates the correct output under predefined conditions. Moreover, TTD aims to only test one particular aspect of a function at a time to make the debugging process easier going forward to identify specific problems, isolated from other functions. In some contexts, the tests should be written before the implementation. These two paradigms can sometimes clash: TTD aims to test all functionality of an object but sometimes these functions are private within the encapsulation paradigm making them impossible to test[10].

As a result, functions that should ideally be kept private would need to made public, for complete testing, or else not all functionality can be tested.

A solution to this problem is to take a behavioural approach to unit tests[11]. In this paradigm, public methods are used to modify the underlying object. If the public interfaces interact with private members, and generates the correct result, then the underlying private methods can be inferred to be correct. This is acceptable if the goal is to test the correctness of the public interfaces, with indirect verification of private functions.

It could be argued that the inability to test core object functionality due to encapsulation is a flaw in how the object was designed. However, private member functions should not be seen as a “code smell”: they work hand in hand with the public interfaces to produce well encapsulated and concealed objects[10].

An example of where this problem occurs can be seen in the main game object. One might want the ability to test some of the underlying game object functions, like checking for win conditions with the `endGameCheck()` function. However, as these are private functions of game, they can’t be tested easily. The only option is to make these public facing but this function should only ever be called directly from the main game object and never from any other context. As a result, this function should remain private and so can not be tested directly. Therefore high-level logic within the main game object was not tested.

10.2 Additional Testing

Additional in-game testing, outside of a unit or interactive GUI testing, was implemented during development by a series of developer or “cheat” keys, restricted to debug mode (not contained in the final release of the game, but being contained inside preprocessor definition guards that were only defined in Debug mode, and ignored in the Release compilation.) These keys bound game spawning events and player states of invulnerability and adjusting the number of lives, altered game speed and forced enemy creation. This allowed the developers to test complex interactions in late-stage game states that required complex and specific events to occur that were difficult or impossible to achieve with public interface testing and unit tests. In Debug mode, a small additional class not used in the Release-compiled code was used, which overlaid a FPS counter in the title of the game window, allowing stress testing of the game, with hundreds of enemies flying about, at very fast global game speeds. No slowdown or excessive CPU or GPU usage was observed in any of these informal, but regular tests, throughout game development.

Additional performance testing was done in Xcode confirming no memory leaks and showed a stable CPU usage while

the game was being played. Additionally, the tests showed a constant disk and memory I/O during gameplay without any spikes in usage. The specifications for the system that ran these tests can be seen in Appendix II.

11 Functionality

The project solution meets the project specification for basic functionality and usability of game mechanics, and further enhances the gameplay with: additional enemy types; a scoring system and HUD; persistent highscores; multiple player lives; and weapon upgrades from defeating unique satellite enemies. The enemy entities are capable of complex and varying movement, and the game has unique, animated graphics and immersive visual elements that improve gameplay. This meets the criteria for all four minor features and two of the major feature enhancements. Unit testing is implemented for all basic movement on all game entities, with logic testing for all public interfaces of logic-related game classes. Basic and specific game functionality is tested, including resource loading and exception throwing for missing files. The test and game code compiles without errors, and the game executable plays without errors or crashing, for repeated plays. Pre and post game splashscreens provide information and analysis of gameplay. The game plays smoothly at the project resolution of 1920x1080, without exhibiting any stuttering or slowdown, at 60 FPS. Player input is respected, decoupling keyboard input events from frame rate updates.

12 Maintainability

Software has a life span and requires ongoing maintenance. The need to fix bugs of existing software, as well as the addition of of new features is to be expected. Maintainable code should be easy to be changed and modified (in modular components) without negatively affecting other parts of the system, a property referred to as orthogonality. As such, classes were designed to be as atomic as possible. The code presented is modular and class-based, with small functions and while maintaining the DRY principles. Functions were split up to produce short sections of code, with specific responsibilities, and were carefully named. The classes outlined in the Overview of Code Structure section are all self standing and can be modified independently of one and other. The interfaces of each class and the associated underlying implementations could be modified, and the core interaction would would continue to function correctly. The separation between entity objects and controller objects simplifies program maintainability. The dependency upon non-STL libraries by using SFML and its classes throughout the code, introduces an additional burden of maintainability. This coupling, and other implications of exploiting the functionality that SFML offers, will be discussed in the next section.

13 Critical Evaluation of Solution

The implementation presented meets the specified success criteria for the project. However, there are a few major design decisions that warrant discussion and evaluation. Possible improvements are also suggested as well as design alterations.

13.1 Coupling with SFML

A design decision that was carried through the implementation of all entities was the coupling between the `Entity` objects and SFML sprites, through the inheritance of the `Animatable` class. This somewhat entangles the separation of concerns between the presentation and logic layers. However the added spatial, structural and computational complexity required to modify the game design and logic to split up these concerns did not justify a late shift in the design paradigm. With the current implementation, any transition to a different rendering framework other than SFML, would require substantial changes. A change in rendering engine is outside the project specifications.

The added complexity in breaking the coupling is due to the original real-time, graphically-focused, visual-object domain model. SFML's (`Sprite`) is a convenient, well-designed and multi-purpose class that enables simple drawing, animating and moving of game entities with a low overhead. The use of a `Sprite` data member and associated methods in the `Animatable` base class, couples `Entity` to SFML's `Graphic.hpp`. In order to break this dependency, it would be necessary to replicate several structures, properties, methods and classes provided by `Sprite`, in order to store the same dimensional and coordinate information, prior to being passed to a `Drawable` object (such as `Sprite`).

For example, in order to draw a sprite at a location, you need the screen position, scale and rotation of the object. This information would need to be stored within a custom game structure on the object and then passed to another, controlling object, such as a "`DisplayManager`" class. This theoretical class would be responsible for the presentation layer (i.e. drawing all the game objects). The `DisplayManager` object would need to generate sprite objects with the correct dimensions, positions and rotations, duplicating knowledge that was already contained within the system. `DisplayManager` would then need to load and assign the correct textures before rendering the objects to the screen. If all game objects have static textures, then a single sprite object with a specific image could be used multiple times, once for each instance of a specific entity type (similar to the way that the `StarField` class works). But in the more likely and visually interesting scenario that the textures are animated (such as an `Explosion`, `Bullet` or the `PlayerShip`), then the custom game object would also need to store and pass the animation data. In order to show the correct frame on screen, `DisplayManager` would need to use a unique sprite for each unique entity object. This would include: the `entity::ID` ; the `textures::ID`; a unique total frame count per texture; individual tile size and offsets; and the current animation frame per object. These parameters would need to be individually passed to `DisplayManager`, to set the sprite attributes per object. With the current solution implemented in the project, these responsibilities are kept inside the object itself and there is no duplication of knowledge. If the object's tileset was designed to work with a different frame rate compared to the global frame-rate, this would also need to be stored and passed, further complicating the mechanism.

To summarise, the disadvantage of clean separation is the requirement to maintain multiple, separate but related data structures, each of which would need to be iterated through separately, increasing computational complexity from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.

Despite the presented solution's linked encapsulation of presentation within game entities, separation of layers was still partially implemented. Entity inherits the `Animatable` properties, which locates the `Sprite` object as a member of a parent class. This means that additional work in modifying the framework to an alternative renderer is isolated to changing how `Animatable` works.

Additional separation of concerns was achieved by defining domains of responsibility between different layers of game objects. The render event only occurs within the main `Game` object loop. The `EntityController` has no knowledge of the mechanism used to draw entities, how they are animated, or how the textures are managed. By coupling each entity object to an associated sprite and texture directly, respective objects are in charge of dealing with their own animation cycles and visual appearance. Each object does not need to load its own textures but rather requests a shared pointer from the `ResourceHolder`.

13.2 SFML Inclusion and Unit Testing

Another problem with the design decision to inherit SFML's `Sprite` class inside `Entity`, is the intimate link with `EntityController`'s collision detection method, whereby `Sprite`'s `getGlobalBounds()` function is used to query the geometric space used by the game object when checking for collisions.Additionally, the coupling with SFML makes the process of unit testing game objects more complex. Each tested object needs to have an associated sprite loaded to create the object, before the underlying logic can be tested. This is a less direct testing methodology, considering that the unit tests are only testing the game objects.

13.3 Alternative Data Structures

Another possible design limitation is the lack of a scene graph, and the use of individual lists of entities in `EntityController`. A linked graph of game entities in, for example, a tree or graph structure could use a graph traversal algorithm to propagate events and messages from the root node through its children, without having to call methods on each list of entities (or individual entities) separately. This structure is not appropriate for collision detection, however, and the entities would have to be pushed into a different structure for that stage. QuadTrees, which segment entities based on their spatial relationships, would provide an fast, efficient and convenient data structure for collision detection algorithms, but this data structure would have to rebuilt with all entities on every frame.

13.4 Enemy Bullet Propagation

When an enemy bullet is fired from an enemy ship, it propagates directly outwards from that location. This method was implemented to simplify the process of controlling enemy bullets as they only need to increment their distance from the screen's geometric centre while flying and adjust their non-linear scale accordingly. This however makes the game play more simplistic as there is only a narrow sector directly in front of the player ship where they can be killed. On either side of this region, the player ship can't be hit with enemy bullets under any conditions. As a result, the player can merely sit at one location of the screen and dodge projectiles flying directly towards them.

An improvement to this method would be implementing a mechanism whereby bullets can fly in non-radial lines outwards from the screen centre, enabling more complex bullet patterns to be generated. For example, a wandering enemy ship could fire bullets aimed directly towards the player ship irrespective of where the either ships' position on screen.

13.5 Alternative Classes

Some of the primary game classes implemented have a domain of responsibility that could be split into multiple smaller objects via a compositional relationship. Entity inherits from the Moveable class, providing an interface to be overridden by its child classes. This premise could be further extended to other generic functions, such as bullet generation.

`EntityController`'s domain of responsibility is too large to add many more methods. Some of this functionality could be split into further smaller classes, through the use of a compositional "has-a" relationship. For example, the production of bullets could be moved into a "BulletFactory" factory object, responsible for spawning all bullet entities. The issue with this kind of refactoring is that by keeping the domain of responsibility for controlling entities within the `EntityController`, the ability to do other functions becomes simpler, such as collision detection. If one wanted to pull these functions out into other compositional inheritance structures, one would now need to pass around many more lists and pointers to entities to enable this kind of interaction. For this reason, given the scale of this project, the domain's of responsibility presented seem reasonable.

The main game object also encompasses a large domain of responsibility. Many of these functions could also be pulled out into other context, such as into the previously discussed `DisplayManager` to manage the process of drawing sprites to the screen.

13.6 Application Performance

The game was developed and run on a system with specifications outlined in Appendix I. A system load benchmark was conducted to verify the efficiency of the game code. This benchmark process checked for memory leaks and plotted application memory usage against time. This can be seen in Appendix I. Additionally, the game was tested for memory leaks and passed all associated tests.

14 Conclusion

The game meets the basic specifications for the project functionality and usability, and adds two major features along with all minor features. The software solution presented is performant and successfully models the movement and interaction of varying game entities with each other, correctly detecting collisions and interpreting player input into

movement and shooting. The game has achievable victory conditions with no major logical or graphical bugs. The game draws and displays using SFML 2 at 60 FPS. There is a clearly defined class hierarchy leveraging hierarchal inheritance to reduce client code length, using clear role modelling in the object oriented code interactions. SFML's classes are used widely throughout the project, resulting in coupled code, but advantageously making use of the efficiently designed multimedia framework layer that SFML provides.

REFERENCES

[1] A. Marchand and T. Hennig-Thurau, "Value creation in the video game industry: Industry economics, consumer benefits, and research opportunities," *Journal of Interactive Marketing*, vol. 27, no. 3, pp. 141 – 157, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1094996813000170

[2] D. Callele, E. Neufeld, and K. Schneider, "Requirements engineering and the creative process in the video game industry," in *13th IEEE International Conference on Requirements Engineering (RE'05)*, Aug 2005, pp. 240–250.

[3] G. Fiedler, "Fix your timestep! gaffer on games," https://gafferongames.com/post/fix_your_timestep/, June 2004, (Accessed on 10/16/2017).

[4] J. Haller and H. V. Hansson, *SFML Game Development*. Packt Publishing Ltd, 2013.

[5] M. Ashley, "Concerning the significance of intensity of light in visual estimates of depth." *Psychological Review*, vol. 5, no. 6, p. 595, 1898.

[6] R. P. O'Shea, S. G. Blackburn, and H. Ono, "Contrast as a depth cue," *Vision Research*, vol. 34, no. 12, pp. 1595 – 1604, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0042698994901163

[7] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[8] M. Mattsson, "Object-oriented frameworks," *Licentiate thesis*, 1996.

[9] "Github - onqtam/doctest: The fastest feature-rich c++98/c++11 single-header testing framework for unit tests and tdd," https://github.com/onqtam/doctest, (Accessed on 10/16/2017).

[10] "Enemies of test driven development part i: encapsulation Âń jason in a nutshell," https://jasonmbaker.wordpress.com/2009/01/08/enemies-of-test-driven-development-part-i-encapsulation/, (Accessed on 10/16/2017).

[11] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 564–569.

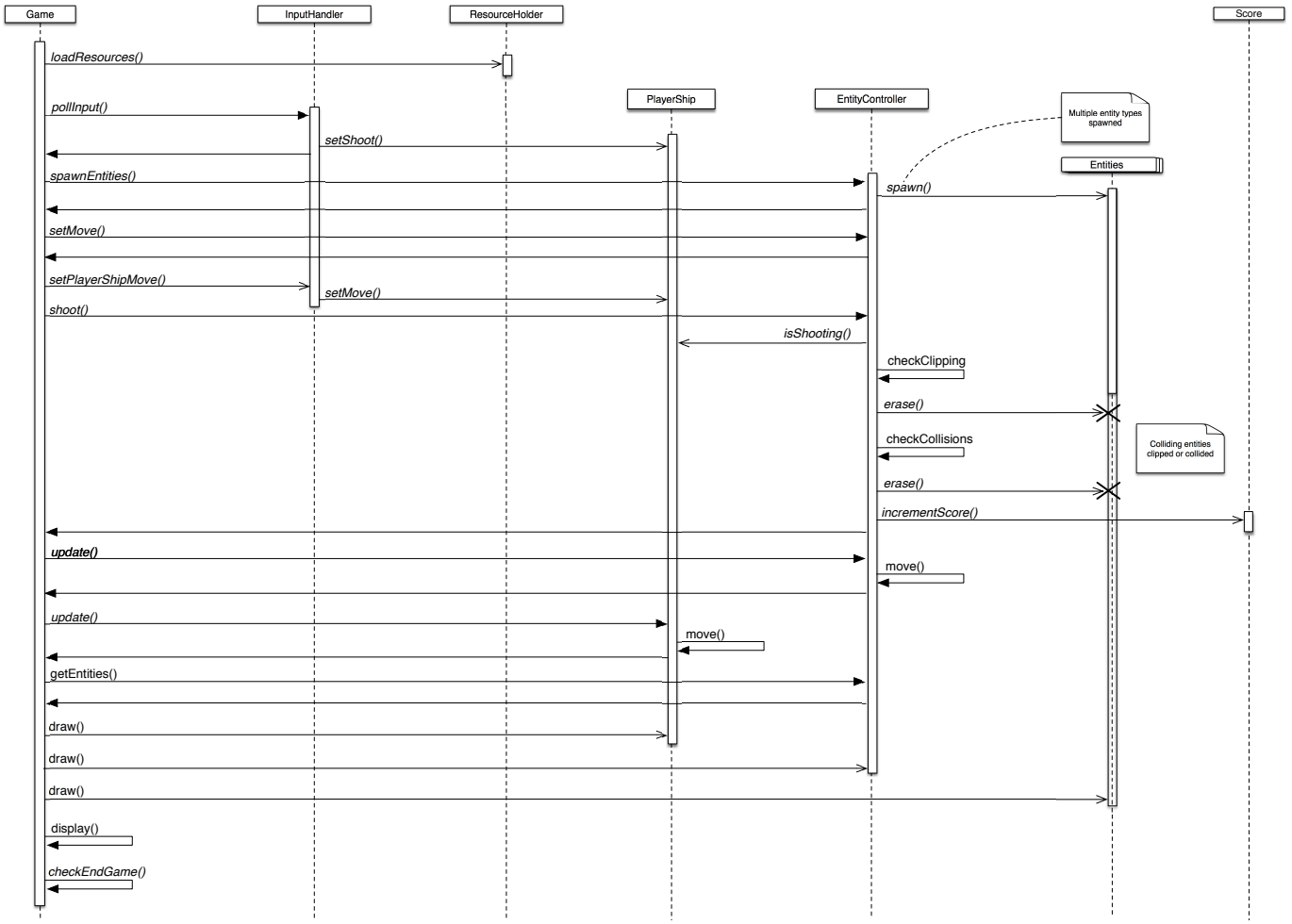

Fig. A1: Simplified sequence diagram

TABLE II: Development environment

CPU	2.2 GHz Intel Core i7 (4-core)
RAM	16 GB 1600 MHz DDR3 RAM (1600MHz)
Compiler	GCC 5.1.0 (tdm-64.1)
System	x86 architecture
RTE	Win32 executables

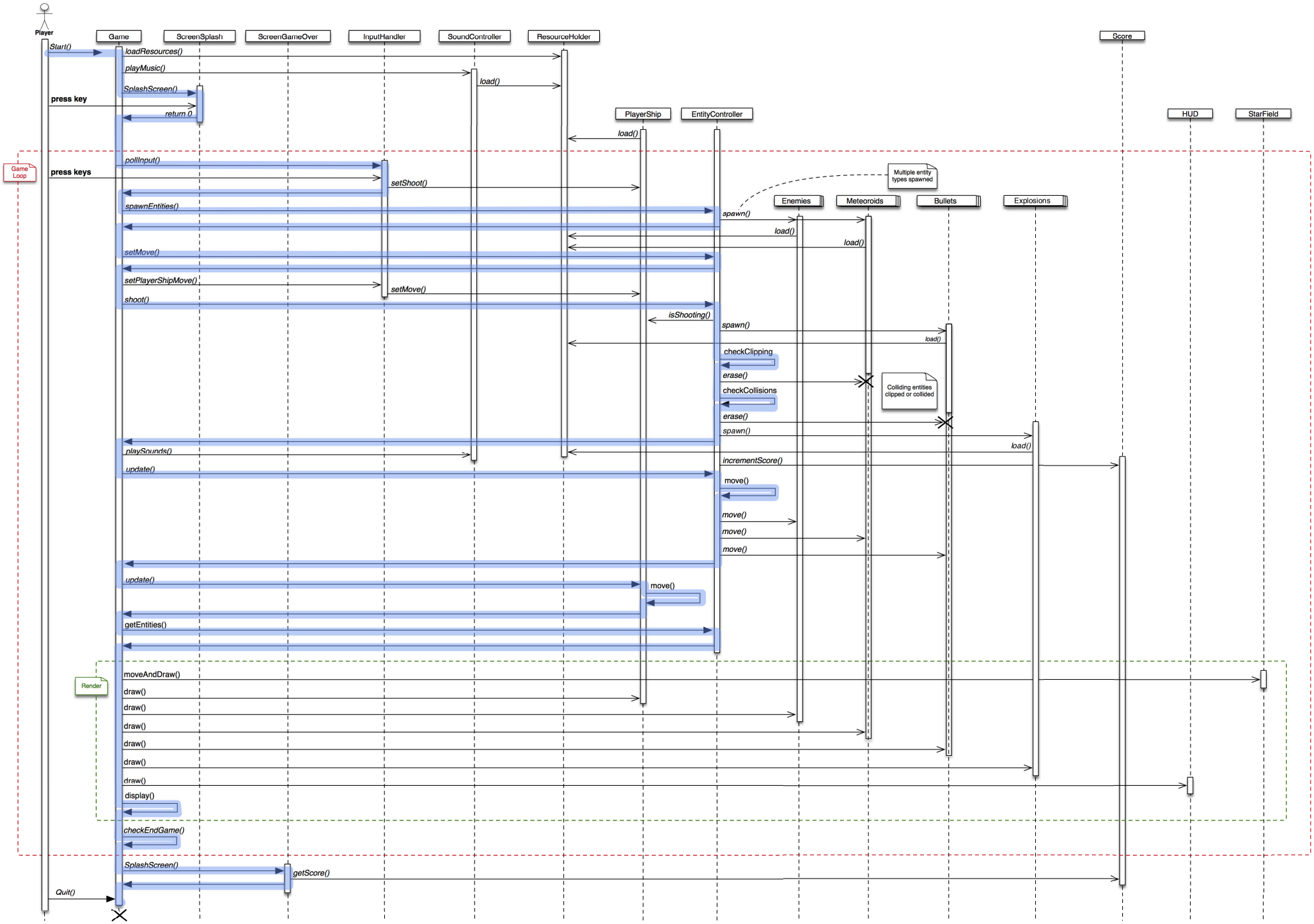

Fig. A2: Detailed sequence diagram