The Design and Implementation of a Sewer Swimmer Computer Game
ELEN3009 Project Report

Matthew Van Rooyen (706692) & Jared Ping (704447)

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

Abstract: The purpose of this project is to design, implement and test a computer based game, Sewer Swimmer, derived from the arcade game Dig Dug. The design presented makes use of Object-Orientated design in order to adhere to the seperation of concerns principle, whereby the data, application, and presentation layers are represented by role-specific classes. Each game entity is contained within it's very own individual class. This collection of classes then inherits basic functionality from a general entity class. Several classes are also implemented in order to manage the collection of entities as a whole. The game screen is rendered through an interface class containing the SFML multimedia library to provide functionality allowing input from the user to be polled through key presses. Furthermore, the game is designed to be compatible with any C++ multimedia library without requiring any data and application layer modification. General gameplay as well as coding and testing structure are analyzed with weaknesses within the game structure identified and possible improvements mentioned for future development. Overall, the development of the computer game is successful, featuring a variety of features to ensure a satisfactory gaming experience.

Key words: Software Development, Object-Oriented Design, C++14, SFML, Google Test

1 INTRODUCTION

The report documents the implementation and design process of a computer game, Sewer swimmer, created using C++14 and Object-Orientated Coding. The game's functionality represents that of the popular 1980's arcade game Dig Dug. The report contains an overview of the game's basic functionality, code structure, testing methods and framework as well as class structures. Important concepts such as inheritance, dependency avoidance, object interactions and polymorphism are also documented with their role in the construction of the game code highlighted. Finally, the report presents a critical evaluation of the game after which future improvements are discussed.

2 PROBLEM DEFINITION

2.1 Sewer Swimmer Game Play

Sewer Swimmer, modelled on the arcade game Dig Dug, is a C++14 developed computer game rendered on a 1000x800 screen. The game contains a player character which swims through the water attempting to eradicate all enemy entities within it. The player is capable of movement as well as launching harpoons using key input from the user. The game is inhabited by multiple AI(artificial intelligence) entities known as enemies. These enemies have computer generated movement and chase the player once the player moves within a defined radius of the enemy. When continuously colliding with a harpoon, the enemy inflates to a point where it pops and is destroyed. The player begins the game with 5 lives and loses a life whenever the entity collides with an enemy or rock and thereafter respawns. The game is lost when the player is destroyed and subsequently, is won when all enemies are destroyed.

2.2 Requirements

The game contains the following basic functionality:

- Player movement is controlled using user input. The player is also bounded within the bounds of the screen.
- Random computer generated movement which is capable of using the players position in order to implement chase functionality.
- The player is required to create paths through the water tiles which the enemies are able to use in order to follow the player.
- Player has the ability to shoot an infinite number of harpoons which can only traverse a finite distance before destroying.

The game is further extended to include the following enhanced features to improve general gameplay:

- The player begins the game with five lives. A life is lost after colliding with an enemy entity and the user loses the game once the number of lives reaches zero.
- The enemy inflates when continuously colliding with a harpoon until a given time elapses and the enemy pops and is destroyed. The enemy is also able to deflate after a given time of where no collisions have occured.
- The player is able to interact with rock icons which then descend through the water and destroy enemies or the player when a collision occurs.
- A scoring system was implemented with both the current and high scores being displayed at the top of the screen.

2.3 *Success Criteria*

The game should implement all functionality mentioned above effectively to deliver game play that runs smoothly and is both simple to understand but challenging for the user. The game should be constructed using object-orientated coding implementing good coding practices and separation of the data, presentation and logic layers. Coding principles such as inheritance, decoupling and polymorphism are also implemented correctly.

2.4 *Assumptions and Constraints*

The game is maximised to a screen size of 1000x800. The game is also limited to keyboard input and cannot accept mouse or joystick operations as a valid input form.

3 CODE STRUCTURE AND IMPLEMENTATION

After considering the proposed problem definition in *Section 2* in addition to the domain model which is discussed in *Section 3.1*, the designed code structure will now be explained. This explanation will include both the classes which form the code structure and the respective design layers which each class falls under.

Correct layer separation has been a pivotal design objective and thus the data, application logic, and presentation layers have been appropriately designed in order to ensure the separation of concerns principle is adhered to, allowing for dependency decoupling. Layer communication is carried out through object conversations which ensures only necessary information is exposed to the other layers whilst keeping majority of layer information hidden from other layers. A class has been designed to function as each layer, whereby the `AssetManager` class represents the data layer, the `Logic` class represents the application layer, and the `Interface` class represents the presentation layer.

3.1 *Domain Model*

The game itself consists of various objects which are contained within the generated game area. These objects are thus identified as Entities and are labelled as follows:

- Rock
- Player
- Enemy
- Ground
- Harpoon
- Tunnel Digger

All the mentioned entities are required to handle collisions between other entities in order to satisfy desired gameplay mechanics. The domain model's hierarchy can be see in *Figure 1*. In order to render the entities to the user it is required that they are drawable, with the exception of the tunnel digger entity which is not meant to be visually rendered to the user. The entities should feature subsets which allow enhanced functionality such as moving and shooting which further enhance the gameplay mechanics to meet the game specifications.

3.2 *Data Layer Class Structure*

This layer contains the smallest component of the three layers as majority of the gameplay mechanics are majorly dependant on the application logic layer. The responsibility of the data layer is simply to manage assets which are required to render the logic of the game as well as provide any necessary data to the game at runtime. These assets could include sounds, text styles and textures, of which the textures form the basis of the data layer since they are required to render sprites in the interface. This responsibility is assigned to the `AssetManager` class which carries out necessary operations. Further class implementation is required to import high scores from past games through a file reader system, with this responsibility being assigned to the `FileReader` class.

3.2.1 AssetManager Class: This class is a unique class due to it's required functionality and is thus implemented as a template class. It serves to improve game performance through optimising the way in which the required assets such as textures, sounds, and text styles are loaded for the presentation layer. This is done by loading all required assets into memory before the game initialises, preventing the game from having to load in assets at a later point or having to do so repeatedly, operations which would tax the game performance on the target system. This process of asset loading also adheres to the DRY principle of coding which was a key factor in designing the template class. The class is then used in the presentation layer to ensure decoupling between the logic and interface is adhered to.

3.2.2 FileReader Class: This class functions as a simple file reader and writer whereby the purpose of the design is to read in the previous high score from a text file before the game begins, and write the new high score to the text file when the game ends, provided it's a higher score. The data that is read from the file is given to the presentation layer via the application logic layer to be displayed to the user. This process ensures that the `FileReader` maintains it's role without accessing more than one desired layer.

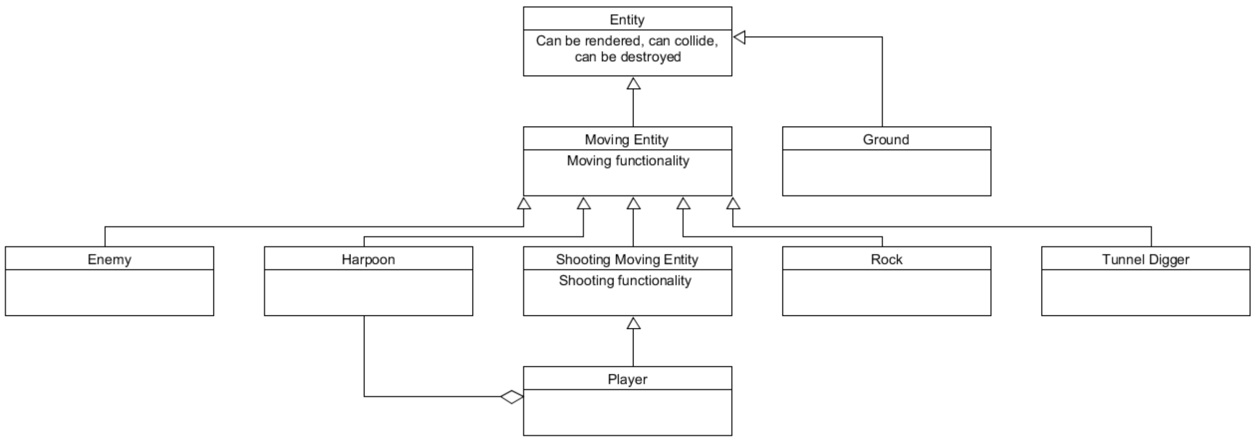

Figure 1: Domain Model Hierarchy

3.3 Application Layer Class Structure

This layer contains the largest component of the three layers as it is responsible for managing all entities of the game, all in-game operations, and the looping of the game to allow dynamic gameplay as desired. This layer can be generalised into two sections namely; the domain classes which manage objects and their interactions, and the logic classes which manage game operations and allow the game to run a specified order of operations.

3.3.1 Vector2f Class: This class is adapted from the SFML version of sf::Vector2f in order to ensure that the logic layer has no SFML dependencies. Doing so simply requires copying the required functionality from the documentation of the sf::Vector2f as all functionality is sufficiently detailed to encapsulate the vector into it's own class to maintain logic and interface separation. The `Vector2f` class is simply a two dimensional vector containing two float values containing the horizontal and vertical axis coordinates. This class serves as a basis of the game logic since the entities are position based on a two dimensional plane and also allows for vector-based mathemical operations to be performed.

3.3.2 Timer Class: This class serves a pivotal role in the functionality of the game as time is being used as the factor by which to quantify game activities. Major implementations of this are seen within the movement of entities, movement of projectiles, and the locking of the screen's frame rate. This is done by quantifying an amount of time which has passed per frame and using that time to scale entity movement and the speed at which in-game activities take place.

3.3.3 Entity Class: This class represents the base functionality of all objects in the game characterised by having a position, allowing collisions to take place, and creating the entity's character data which allows for the interface to then render the object. This allows the user to visually see a collision which has been rendered but is done so after appropriate logic notifies the interface to do so through object conversations. This class also offers additional logic functionality such as inflation and deflation but is not a requirement to be utilised by all derived classes.

3.3.4 CharacterEntity Class: This class allows relevant information regarding the type of entity and it's position to become available for rendering by the interface. This facilitates visual rendering by serving as a means for an object to transfer relevant data to the presentation layer.

3.3.5 EntityContainer Class: This class manages all the active entities in the game. It is used to process collisions between all active entities in order for the game logic to process the collision effects thereafter. This class then provides updates to the interface used to render the entities.

3.3.6 CollisionManager Class: This class is responsible for managing all collision detections between entities, informing the respective entities if a collision does indeed occur, which then allows for desired actions to take place. The entities are able to check for collisions through the use of hitbox generation which will be discussed in detail in *Section* 4.1.

3.3.7 MovingEntity Class: This class inherits from the `Entity` class, providing the `Entity` class with movement capabilities.

3.3.8 ShootingMovingEntity Class: This class inherits from the `MovingEntity` class, providing the `MovingEntity` class with shooting capabilities.

3.3.9 Ground Class: This class inherits from the `Entity` class and forms the basis of the game map as the ground objects are simply static entities which create boundaries for other entities. The ground objects are generated until a certain area of the map is covered and collides with all other entities. The player is able to destroy ground objects upon collision, enemies treat the collision as a barrier and have to change their course, rocks treat the collision as a barrier and movement is prevented, and harpoons are destroyed upon collision with a ground object.

3.3.10 TunnelDigger Class: This class inherits from the `MovingEntity` class and creates objects which are responsible for creating tunnels for the game. A finite amount of `TunnelDigger` objects are created in set positions while the splashscreen loads and will randomly dig a tunnel in either a vertical or horizontal direction in order to add to a randomised game experience. Collisions are only dealt with when contact occurs with a `Ground` object whereby the ground object at that position is destroyed. After each `TunnelDigger` object has destroyed a certain amount of ground objects, they are then destroyed, noting that the user will never see the `TunnelDigger` objects.

3.3.11 Rock Class: This class inherits from the `MovingEntity` class and creates rocks which appear to experience gravity when there is no ground below them. When the `Player` destorys the ground objects below the rock, it will make contact with the player and hold it's state for a short while. Thereafter the rock will begin to fall, regardless of whether the player is still below it, until it hits a ground object at which point it is destroyed. The rock object is capable of destroying both enemies and the player if the collision occurs while it is falling.

3.3.12 Harpoon Class: This class inherits from the `MovingEntity` class and represents a simple projectile which is launched by the `Player`. It will continue to move in the direction it is fired for a finite distance before it is automatically destroyed. It is able to collide with `Ground` and `Rocks` whereby it is destroyed, and `Enemies` which it will destroy if collisions continue to occur for a certain duration. Additionally, the class ensures no more than one harpoon can exist on the map at any given time.

3.3.13 Enemy Class: This class inherits from the `MovingEntity` class and represents an entity which spawns within dug tunnels, moves autonomously through open tunnels through the use of time-based updates, and chases the `Player` within a defined radius. It collides with the `Ground` and `Rocks` having to move around them, the `Player` which is will then destroy, and the `Harpoon` which begins to inflate it until destroyed. Additionally, the class ensures that only one enemy is spawned per every open tunnel at the beginning of the game.

3.3.14 Player Class: This class inherits from the `ShootingMovingEntity` class and represents the user controlled entity in the game. It's movement is directly controlled through keyboard interaction using SFML event polling in the interface which is then converted to logic based events that are processed in the application layer to allow the `Player` to move in the direction intended whilst tracking the distance traversed for correct rendering by the interface. The `Player` is able to launch a harpoon which is launched in the direction of the last movement. The `Player` collides with `Enemies` and falling `Rocks` which destroy it, and `Ground` which it destroys. Additionally, the class continuously tracks the number of remaining lives which the `Player` has.

3.3.15 Logic Class: This class carries the responsibility of setting up all pre-game requirements, instantiating all required entities for the game, and finally running the game. The main role of this class is to interact with the `Interface` so that the presentation layer is continuously rendering the updated game, which will be discussed at a later point. Once the game meets the required end conditions, the `Logic` class then ensures that the game ends when the `Player's` lives have diminished thus constituting as a loss, or when all `Enemies` have been destroyed thus constituting as a win.

3.4 Presentation Layer Class Structure

This layer is responsible for all user input polling through SFML events and rendering all game entities. This layer can be thought of an interchangeable layer whereby any multimedia library would suffice, thus SFML is used solely in the presentation layer, allowing the game logic to function independently.

3.4.1 Interface Class: This class communicates with the logic layer through object conversations in order to render all game entities and entity events. Furthermore, user input is polled through SFML events which is pushed back to a vector of logic layer based events and then sent back to the logic layer so that the appropriate entities can then update based on the functionality which they've been assigned. Lastly the game statistics, including the `Player's` number of lives, current game score and high score are rendered at the top of the screen for enhanced game functionality.

4 IN-GAME BEHAVIOUR

The game operations are managed through the active game loop in the `Logic` class which conforms to a standard flow of operations as depicted in *Figure 2*. Thus it can be seen that the repeated loop adheres to a process of polling for user input, creating required entities, managing `Entity` collisions, and finally allowing the `Player` to launch a harpoon if all `Enemy` objects have not yet been destroyed.

The key design in allowing this functionality to work as intended is through the use of object conversations which serve to allow the `Logic` class to access specific classes required to run the game. Such a demonstration is that of a `Enemy` and `Harpoon` collision, whereby the amount of time that the `Enemy` has been hit is monitored to ensure it is inflated and eventually destroyed. A visual illustration of these commincations is further depicted in *Figure 4*.

The communication between the `TunnelDigger` and `Logic` classes takes place through the use of `getDiggerQuanity()`. This is used to monitor how many `TunnelDiggers` have been created to ensure the correct amount are spawned. Communication between the `TunnelDigger` and `Logic` classes takes place through the use of `getEnemyDestroyed()` and `getEnemiesCreated()` which are used to spawn the correct amount of `Enemies` as well as to end the game if all the enemies are destroyed.

An example of more complex class communication becomes evident when considering user input and collisions. However, these communications will be more comprehensively covered below.

4.1 Collision Polling

Collisions form a key component in the functionality of the game as almost every game feature is dependant on the registering of collisions between objects. Since the game consists of many objects that need to be checked for collisions, comprehensive research is necessary in choosing an accurate and computationally efficient algorithm.

The implemented collision detection algorithm is thus chosen to be the separating axis theorem (SAT) which checks for collisions between two convex polygons. The general condition of the algorithm is that if it's possible to find an axis along which the projection of the two polygons do not overlap then the two polygons do not overlap. This algorithm is designed for a large space whereby polygons are more likely not to overlap than they are to overlap which speeds up the collision detection due to fewer collision occurrences. If two polygons are not separated by a single axis, the algorithm then has to iterate through all viable axes to check for where the collision occurs [1].

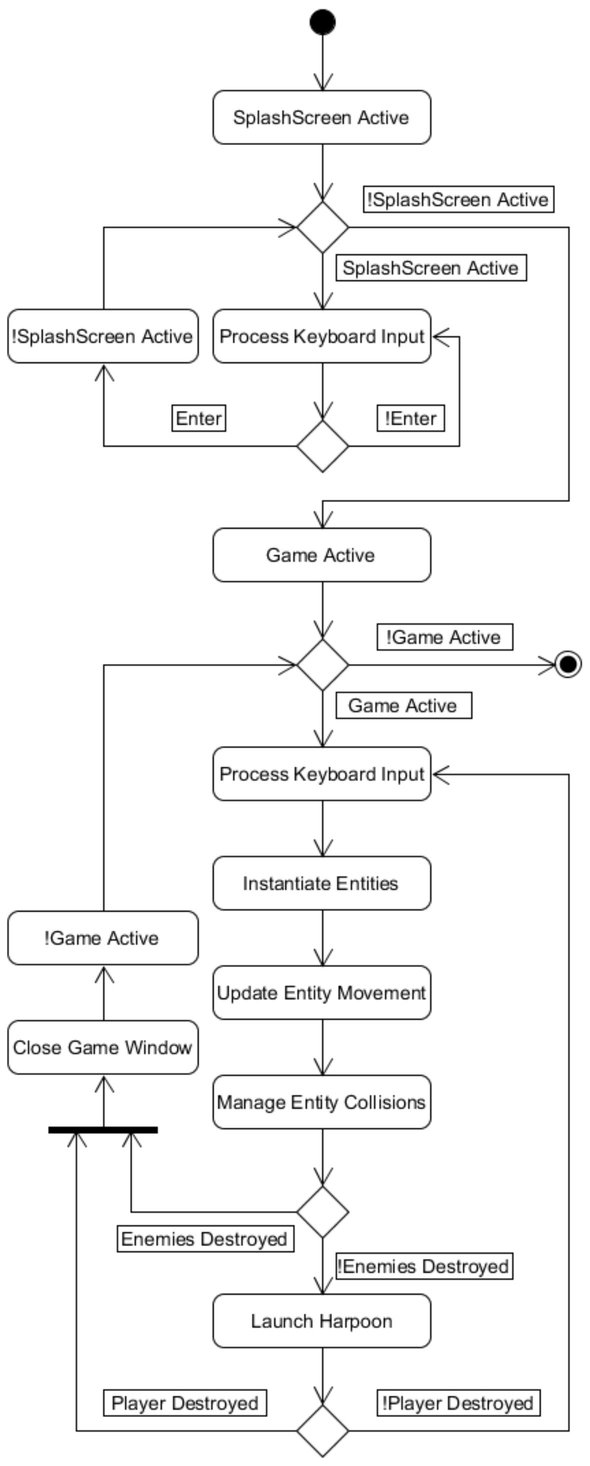

Figure 2: Flow diagram depicting logic flow when game is active

Upon assessing the run-time complexity of the algorithm, it will always vary as not every game object is checked against every other game object since this would be computationally intensive with a complexity of $\mathcal{O}(n^2)$. Instead, the algorithm only iterates through each movable object with every other game object which then reduces the algorithm complexity to $\mathcal{O}(cn)$. It should be noted that as the game progresses and game objects are destroyed, the algorithm complexity will continue to improve until the end of the game whereby it would have a complexity of $\mathcal{O}(n)$. *Figure 3* demonstrates how game objects interact with each other by illustrating the appropriate collision relationships which exist between them.

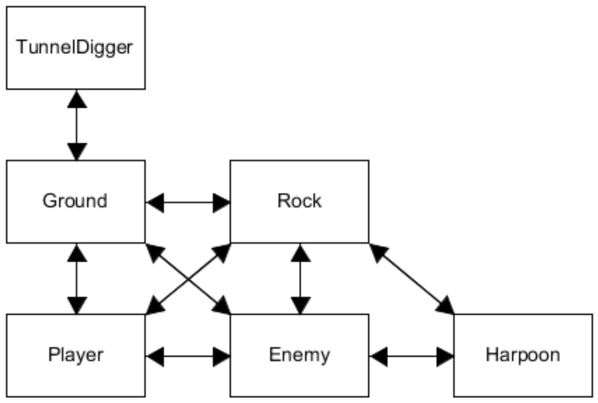

Figure 3: Illustration of collision relationships between objects

4.2 Game Input Polling

The keyboard input from the user is captured through the `Interface` class which polls for keyboard events using `sf::Event` which is done through the `processEvents()` function of the class. These polled events are then converted to `GameEvents` which the `Logic` class then handles. This process adheres to the separation of concerns principle as stated before in order to split the logic and rendering frameworks of the game.

User polled events include `Player` movement, the ability to launch `Harpoons`, pausing the game, as well as quitting the game. Each key is mapped to a specified event, which the logic of the game carries out and the interface renders thereafter. Further consideration of design allows for the use of input polling whereby a key is polled when it both pushed and released. This implementation allows for prolonged `Player` movement and shooting in order to correctly inflate `Enemies` as well as to prevent undesirable events such as the game unpausing when the user tries to pause it but takes too long to release the pause button.

5 CODE TESTING

This section contains an analysis of all testing performed on the source code for Sewer Swimmer. Test driven coding is a programming method by which a test is written for a basic piece of functionality after which code is drawn up in order to pass the given test. Different levels of testing are conducted on each class with some classes being tested more rigorously than others depending on the complexity of the functionality. Table 1 summarizes the total testing performed on all the test cases. The lack of any presentation layer testing is discussed in Section 6.

5.1 Testing Environment

Google Test C++ 1.7.2 is used in order to compile tests on the project source code. The code is compiled and linked using mingw32 5.1.0.

5.2 Data Layer Testing Structure

5.2.1 assetManagerTest: The assetManager class contains 2 public member functions responsible for loading and assigning images to a given texture. Testing is performed to ensure class functionality loaded an image from the valid file path and an error was thrown should the image or file path not exist. These tests require the use of SFML as the images were loaded from SFML storage.

5.2.2 FileReaderTest: FileReaderTest tested a basic reading or writing to a given file name, in this case score.txt. Each member function is tested to ensure that the given file was able to be located and loaded as well as an error thrown should the file path not exist. The classes ability to update the high score once the previous high score has been exceeded is also tested.

5.3 Application Layer Testing Structure

5.3.1 EntityTest: The classes ability to both create a default character object with a valid entity key, manipulate it's position and boundaries as well as destroy the object was tested. the created object's collision and movement functionality is also tested.

5.3.2 EntityContainerTest: Each of the member functions contained within the EntityContainer class are tested. Testing is conducted to ensure the class successfully manages the insertion and deletion of an entity at various positions in the character list.

5.3.3 CharacterEntityTest: The CharacterEntity class' ability to create an object specific to a certain entity with an entity key and position on the game screen is tested. Tests were further conducted returning the character objects entity key and position to ensure the correct values had been assigned.

5.3.4 PlayerTest: `PlayerTest` contains a much larger amount of testing in comparison to majority of the other classes as the entity's functionality was pivotal to the success of the game. The creation of the Player object is initially tested to ensure the objects default position at time of creation is centred to the middle of the game screen and the position could be manipulated further depending on game instances.

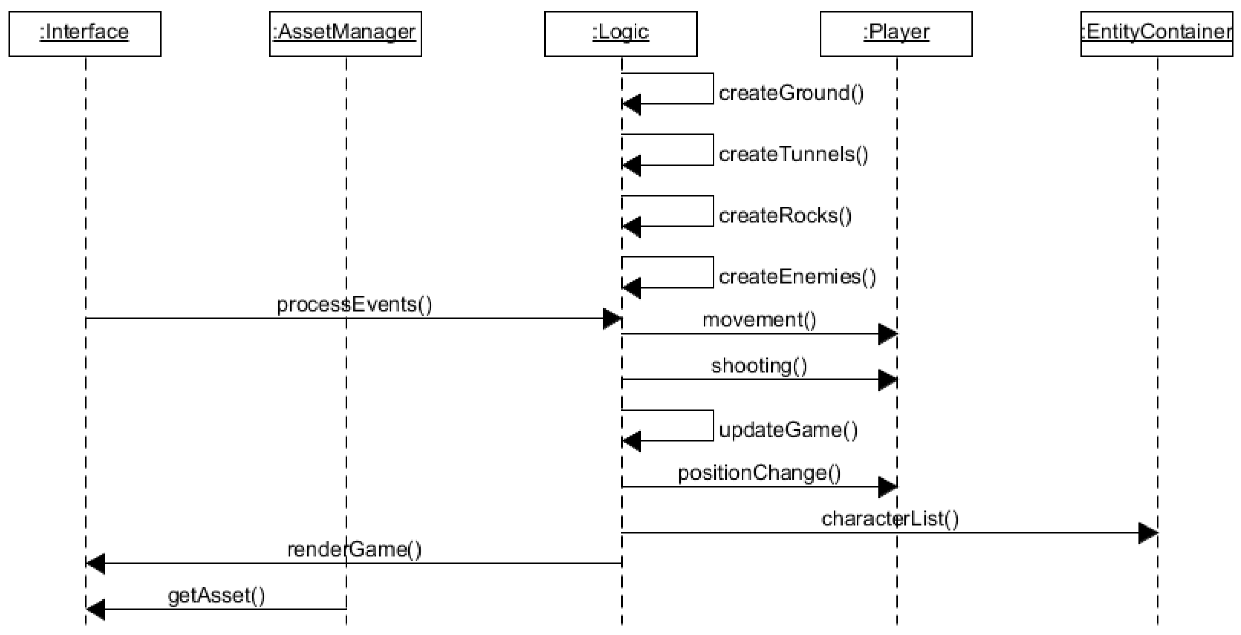

Figure 4: Sequence diagram illustrating in-game class interaction

Table 1: Test results

	Tested Class	Number of Tests
Application layer	Player	17
	Enemy	8
	CollisionManager	1
	Entity	7
	Harpoon	8
	Ground	5
	Rock	8
	TunnelDigger	7
	EntityContainer	7
	Vector2f	24
	CharacterEntity	2
Data Layer	FileReader	4
	AssetManager	2
	Total	100

The players movement is also tested thoroughly as well as the objects shooting capabilities in response to user input. Tests are further conducted to ensure the player did not exceed the predefined map bounds of the game screen. Finally, tests were conducted to ensure the number of lives specific to the player decremented when colliding with an enemy object and the player is destroyed when the lives reach 0.

5.3.5 EnemyTest: The Enemy Class, as per project requirements, contains a large variation of functionality which has to be tested and implemented within the game play. The enemy entities contain an inflate and deflate feature which is tested using instantiated collisions with a harpoon object. Various collisions between the enemy and other game entities are also tested.

The enemies ability to chase the player is tested by ensuring that the distance traversed by the enemy always decreased. This functionality also made use of the `velocity_unit_direction` which calculated the shortest route between the player and enemy. This function is tested implicitly within the test.

5.3.6 GroundTest: The ground test needs limited testing due to the minimal functionality of the class. Importance is initially placed on ensuring the ground tile was created in the correct position. Further tests are then conducted dealing with the ground's interaction with all the other existing entities. Finally, a test is conducted to check the addition of points after each tile is destroyed.

5.3.7 HarpoonTest: `Harpoon` conatains two main member functions which are tested, namely `move` and `collide`. Collisions between the harpoon and various entities are tested. The collision of the bomb with the map bounds is also tested thoroughly, similarly to that of the `PlayerTest` and `EnemyTest` cases.

Important to the specifications of the game is that the harpoon only be able to traverse a certain distance before destroying. This is tested by instantiating an elapsed time to ensure that a distance exceeding the range limit is reached and the destroyed status of the harpoon then checked. Finally, a test is also conducted to ensure that no more than one bomb can exist at a time.

It should be noted that the generation of a harpoon at the player position is tested within the player class test case.

5.3.8 CollisionMangerTest: `CollisonMangerTest` only contains a single test focused on detecting a collision between two non-specific entities. The lack of testing is due to a large number of the entity classes having different implications for each collision and ,therefore, were tested more thoroughly according to each specific functionality. The resultant test need only flag a collision when two objects interact with each other.

5.3.9 RockTest: Due to the `Rock` class containg minimal functionality, a small number of tests are produced for the class. More complex testing methods are needed however to ensure the rock object is implemented correctly. Firstly, the rock should pause briefly when falling when coming into contact with the player. This required the use of a `_timer` variable to avoid the complicated use of the system clock. The destruction of an enemy or loss of one of the players lives should only occur when the rock is falling but the rock should also become stationary when colliding with a ground tile. Each of these traits are assigned a designated test to ensure successful implemtation.

5.3.10 TunnelDiggerTest: Similar to both the `Ground` and `Rock` classes, the tunnelDigger needs only a small number of tests to check the class' functionality. Tests are given to the entity's position when initialized as well as the interaction with the ground tiles. A more compliacted testing structure is needed to ensure the tunnel digger only moves through the ground for a certain distance before being deleted.

5.3.11 Vector2fTest: The Vector2f class consists of a number of mathematical operators which have been overloaded in order to perform vector mathematics. Each overloaded operator is tested as well as mathematical functionality between vector and vector as well as vector and scalar quantities. The Vector2f test case consists of the largest number of tests as majority of the classes use the mathematical operations.

6 CRITICAL EVALUATION

6.1 Game Functionality

6.1.1 Successes: All required functionality for the game to meet the excellent tier rating has been implemented, consisting of all basic functionality as well as two minor features and two major features. Multiple game objects exist in the game: `Player`, `Enemies`, and the `Ground`. The `Player` moves correctly through the `Ground` based on user input, and digs tunnels wherever it moves. `Enemies` move autonomously and chase the `Player`. At the start of the game there is always at least one `Enemy` present in the same tunnel as the `Player`. The `Player` can shoot `Enemies` with a `Harpoon`. The game ends when the `Players` lives run out or when all `Enemies` are destroyed. Furthermore, the game includes a scoring system and display where high scores are saved from one game to the next and can be viewed. As mentioned before, the `Player` has more than one life and the remaining lives are depicted on the screen. `Rocks` exist within the game as well as the ability to inflate `Enemies` with the `Harpoon`, outlined in *Section 2.2.*

All aforementioned functionality is implemented through careful use of modern C++14 coding standards whenever possible, whereby C++11 is used if deemed a better design alternative. Smart pointers are chosen as favourable over generic pointers, which are implemented to ensure that the game is free of memory and resource leaks and are exception-safe, with coupled use of `std::make_unique()` and `std::make_shared()` functions to maintain a C++14 based design. Such an example of the C++11 implementation is the use of the `constexpr` keyword which is coupled with `static const` to improve overall performance. Further examples of performance based choices in using C++11 are those of the C++ `auto` specifier, `const` and `override` keywords are utilised as compiler checks wherever viable. Due to random access of some container indices being required, `std::vector` is exclusively used allowing for random element retrieval at a complexity of $\mathcal{O}(1)$.

6.1.2 Issues: Collision handling between an `Enemy` and `Ground` occassionally glitches if the `Enemy` is actively chasing the `Player` which can thus affect gameplay is a negative manner.

If the `Player` respawns and there happens to be an `Enemy` at the respawn position, the respawn process will continue to happen whilst the `Player` lives are drained until the game finally ends, causing quite a dull game experience when this happens.

Game animations could definitely be improved as they currently occur through rendering a new texture over the original texture which is sometimes noticeable and thus not aesthetically pleasing to the user.

Error catching in the code structure is minimal, consisting of mostly error handling for logic and runtime errors in the process of loading both assets or high score files. Thus an unexpected error could have detrimental effects on the game, possibly preventing the user from playing at all.

Within the `Logic` class whereby the `Entity`, `MovingEntity`, and `ShootingMovingEntity` lists are iterated through to carry out relevant operations, it has become apparent that the use of the `ShootingMovingEntity` list has become redundant and simply serves as a placeholder for `ShootingMovingEntities` rather than improving performance.

6.2 Code Structure and Implemented Design

6.2.1 Successes: The basis of the code structure design is to consist of many classes, composing a single objective, comprising of smaller efficient classes which together obtain the desired functionality of the game. These classes incorporate inheritance for efficient role modelling which are characterised by the following functionalities:

- Ability to move
- Ability to shoot
- Ability to collide

Thus the principle of don't repeat yourself (DRY) is mostly well adhered to due to code reuse throughout all `Entity` based classes.

The implementation of both the `EntityContainer` and `AssetManager` classes provide additional functionality to compliment the retrieval of data within the game. The `EntityContainer` class interacts with the `CollisionManager` class to allow collisions to take place within the game. The `EntityContainer` class also interacts with the `CharacterEntity` class to allow game `Entities` to be rendered by the `Interface` class.

In order to limit what the user has access to in the interface of the game's code, private functions are implemented wherever possible in order to reduce the presence of unnecessarily large public interfaces as well as protect certain inner-game processes. Such a demonstration of this design implementation can be seen in the `Timer` class whereby public functions run with a collective call to private functionss and in the `CollisionManager` whereby the collision handling algorithm is run with a collective call to private functions.

Most classes are majorly independent of each other and encapsulated in order to allow them to collaborate without sharing too much of their information. The use of getters and setters were used to provide necessary information to other classes, allowing private information access to be restricted to the permitted getter functions.

Separation of concerns has been successfully implemented as each class has been specifically assigned to either the data, application or presentation layer. As mentioned above, class interaction is explicitly controlled which is used to enhance layer separation by preventing each layer from knowing how the other layers function.

The decoupling of the `Logic` and `Interface` is successfully implemented as it would be possible to use a different rendering library without the data and application logic layers requiring any modifications.

Finally, it's notable that the overall design and implementation of the code structure allows for an easily expandable code base allowing for additional functionality to be added with minimal redesign required. This allowed for minor and major features to be easily implemented once the base game functionality had been implemented.

6.2.2 Issues: The DRY principle has been mostly well adhered to however, there are some notable violations present. A demonstration of this is the `move()` function from the `Enemy` and `TunnelDigger` classes which are almost identical in structure aside from some minor changes. It is thus deduced that should a change be required for the function, it would have to be done in both classes rather than a central class. Instead, the base functionality of the function could have been inherited from a base class, with the derived class simply adding any additional modifications required.

The `Entity` class, which serves as a base class for entities in the game, has become less of a conceptual model within the designed domain model and features more functionality than intended. Thus any functional modification to enhance additional features becomes tedious and requires additional code design to implement, with the possibility of preventing certain functiontionality enhancements depending on their requirements. An example of this is the inflation logic used in the `Entity` class rather than a more suitable class since not all entities should be able to inflate.

A level of technical debt has become apparent within the code functionality of the game. An example of this is the ability for the `Enemy` object to glitch through `Ground` objects when chasing the `Player`, which is due to poorly handled collision management within the `Enemy` class due to time constraints in meeting submission deadlines. It should be noted that any changes to that code would not affect any other classes and is thus a well designed functionality.

The implementation of the strongly typed `EntityList` enumeration class is used for correct texture assignment of each `Entity` object and forms the backbone for collision handling as each item in the enumeration class is checked through switches in the collision handling function of each `Entity` that is capable of collisions. Although this simplifies the collision checking process for each `Entity`, the addition or removal of items in the `EntityList` class would require every collision capable class to have the relevant switches updated in order to handle the changes. This becomes time consuming and more difficult to succesfully track as the amount of derived `Entity` classes increase.

6.3 Testing Structure and Implementation

A test case is created for majority of the classes within the game code with various levels of testing being performed on the class depending on the scale of it's functionality. The three main aspects of the each entities functionality, namely moving, shooting and colliding, were tested thoroughly in each instance.

Commonly, each member function found within the class needs a single test. Certain functions however need multiple test cases where the functionality varied for several instances. When testing that an entity can not exceed the screen boundaries, for instance, a test is to be performed on all four boundaries to ensure the functionality was successful.It should also be noted that the Google Testing Framework does not allow the testing of any private member functions defined within the class.

Static and non-virtual functions are tested under the base class' test case as the functionality does not vary from class to class. Virtual member functions may require some testing within the base class ensuring basic functionality but majority of the functions are tested thoroughly within the inherited classes to ensure the test catered to that class' specific functionality.

`Entity`, `MovingEntity` and `ShootingEntity` are conceptual classes as they contain limited to no functionality. These classes are not given their own test case but are instead tested in various other classes where their functionality has been instantiatiated.

A test case could not be performed on the `Timer` as the level of accuracy needed in order for a compaison was too great to predict each time. Getters and Setters found within each of the classes were also not given a test but were instead tested implicitly within the other test cases in order to obtain integer or boolean values for comparison.

Despite the fact that the randomness of the enemy movement could not be tested, movement of the Enemy entity in all directions could be tested by forcing a case of movement in a given direction. The testing of user input is also found to be quite difficult as the Google testing framework does not allow for testing during game play.

The lack of any presentation layer testing being present was due to the interface class only containing SFML functionality. By retesting each of the SFML functions, the DRY principle is violated as an assumption can be made that testing has been performed previously.

Finally, the `Logic` class is also not tested due to all member functions declared as private. This was done by design to ensure the user did not have access to the functions but simply controlled an object of the class to instantiate the game. Having said that, majority of the functionality implemented within the `Logic` class had been tested in the various base classes.

7 IMPROVEMENTS AND ALTERNATIVES

7.1 Game Functionality

Not all game features are included in the game and thus leaves room for improvement. This includes the enhancement of certain graphical features, the ability for enemies to transform into disembodied forms with unique movement capabilities, bonus items which spawn in game allowing for bonus points to be achieved, and finally the ability for dragons to be implemented with their own unique disembodied form and capability of firing projectiles at the player.

It should be noted that several issues which are apparent in the functionality of the game. The collision handling of enemies whilst chasing the player leads to the enemies being able to glitch in the ground. This requires a better collision management of enemies to ensure they move in an allowed direction, whilst still being able to track and move towards the player. Another issue is the player being able to initiate the rock timer to begin falling by touching any other side of the rock rather than the intended bottom side. This was hastily addressed by slightly moving the player away from the rock when a collision occurred from an undesired side of the rock by can sometimes still occur. The aforementioned issues thus prove that overall collision management is lacking improved movement restrictions are required to ensure correct desired effects.

7.2 Code Structure and Design

As the game has been further developed, functionality has taken priority over coding structure and thus it is apparent that a somewhat general domain model has been created. It would thus be beneficial to have more specific conceptual classes being implemented to fulfil a more conceptual design rather than prioritised functionality. The use of inheritance could be better structured in order to prevent unnecessary member function repetition in derived classes and allow for minimal code reuse, whilst also ensuring derived classes do not have access to member functions which are intended for them. Furthermore, the use of pure interface classes could be implemented to help reduce tight coupling and code accessibility from required classes.

Although overall game performance of the game was deemed more than satisfactory, possible performance optimisations are always possible. This is could be done by removing the vector of `ShootingMovingEntity` pointers as iterating through the vector of `MovingEntity` pointers serves the same purpose due to the `ShootingMovingEntity` class inheriting from the `MovingEntity` class. Overall performance could be improved through the implemenatation of multithreading whereby a CPU with multiple threads could take advantage of the program by running multiple process simultaneously on different CPU threads [2].

7.3 Testing Structure

The testing structure has the capabilities of expanding even further despite the large number of tests being conducted at present. `CollisionManagerTest` could be expanded on to check that collisions do not occur between objects that are not in contact with each other. Instead of simply detecting collisions, the tests could be further expanded to detect on which border of the hitbox the collision takes place.

A mocking framework could be used in order to further isolate code and allow tests to be performed on the classes with a large number of dependencies. This might allow for the testing of the `Logic` class which is heavily linked with `Interface`. Automated tests could also be implemented to account for random movement or random button presses from the user. This could further be expanded on to check functionality, such as the generation of a splashscreen and the pause feature, while the game window is open. Should any features be added to the game in the future, the testing framework will also need to expand in order to account for the additional functionality.

8 CONCLUSION

The design, implementation and testing of the game Sewer Swimmer has been thoroughly analysed. The game's functionality and code structure has been discussed in great detail with emphasis placed on the use of Object-orientated programming, inheritance and polymorphism within the code. The game satisfies all criteria listed in order to reach a level of excellent for game play whilst adhering to all constraints listed in Section 2.4. The code has been structured to limit the number of dependencies and illustrates a well designed separation of concerns through correct modelling of the presentation, application and data layers. The report also details the use of a Google Testing framework in order to create unit tests for the program. Functionality, game play and testing have been analysed and evaluated in Section **??** with possible improvements and alternatives stated in Section **??**.

REFERENCES

[1] Metanet Software Inc. "Collision Detection and Response" `http://www.metanetsoftware.com/technique/tutorialA.html`, 2011. Last Accessed: 24 September 2016.

[2] Barbier, M. (2015) SFML blueprints. Page 175. United Kingdom: Packt Publishing.

Appendix A

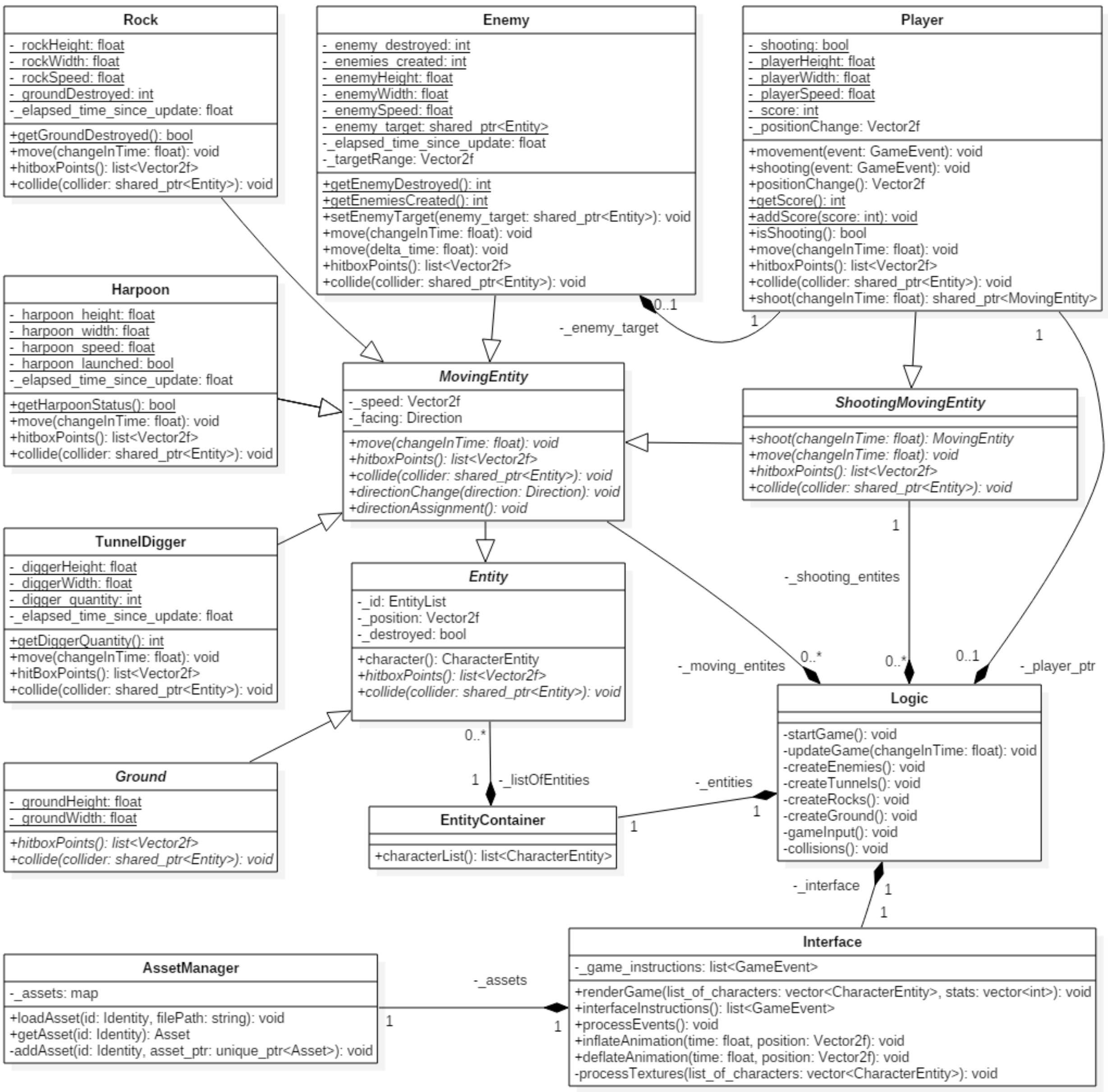
Rock
- rockHeight: float
- rockWidth: float
- rockSpeed: float
- groundDestroyed: int
-_elapsed_time_since_update: float
+getGroundDestroyed(): bool
+move(changeInTime: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
Enemy
- enemy_destroyed: int
- enemies_created: int
- enemyHeight: float
- enemyWidth: float
- enemySpeed: float
- enemy_target: shared_ptr<Entity>
-_elapsed_time_since_update: float
-_targetRange: Vector2f
+getEnemyDestroyed(): int
+getEnemiesCreated(): int
+setEnemyTarget(enemy_target: shared_ptr<Entity>): void
+move(changeInTime: float): void
+move(delta_time: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
Player
- shooting: bool
- playerHeight: float
- playerWidth: float
- playerSpeed: float
- score: int
-_positionChange: Vector2f
+movement(event: GameEvent): void
+shooting(event: GameEvent): void
+positionChange(): Vector2f
+getScore(): int
+addScore(score: int): void
+isShooting(): bool
+move(changeInTime: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
+shoot(changeInTime: float): shared_ptr<MovingEntity>
0..1
-_enemy_target
1
1
Harpoon
- harpoon_height: float
- harpoon_width: float
- harpoon_speed: float
- harpoon_launched: bool
-_elapsed_time_since_update: float
+getHarpoonStatus(): bool
+move(changeInTime: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
MovingEntity
-_speed: Vector2f
-_facing: Direction
+move(changeInTime: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
+directionChange(direction: Direction): void
+directionAssignment(): void
ShootingMovingEntity
+shoot(changeInTime: float): MovingEntity
+move(changeInTime: float): void
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
1
-_shooting_entites
TunnelDigger
- diggerHeight: float
- diggerWidth: float
- digger_quantity: int
-_elapsed_time_since_update: float
+getDiggerQuantity(): int
+move(changeInTime: float): void
+hitBoxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
Entity
-_id: EntityList
-_position: Vector2f
-_destroyed: bool
+character(): CharacterEntity
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
-_moving_entites
0..*
0..*
0..1
-_player_ptr
Logic
-startGame(): void
-updateGame(changeInTime: float): void
-createEnemies(): void
-createTunnels(): void
-createRocks(): void
-createGround(): void
-gameInput(): void
-collisions(): void
Ground
- groundHeight: float
- groundWidth: float
+hitboxPoints(): list<Vector2f>
+collide(collider: shared_ptr<Entity>): void
0..*
1
-_listOfEntities
EntityContainer
+characterList(): list<CharacterEntity>
-_entities
1
1
-_interface
1
1
AssetManager
-_assets: map
+loadAsset(id: Identity, filePath: string): void
+getAsset(id: Identity): Asset
-addAsset(id: Identity, asset_ptr: unique_ptr<Asset>): void
-_assets
1
1
Interface
-_game_instructions: list<GameEvent>
+renderGame(list_of_characters: vector<CharacterEntity>, stats: vector<int>): void
+interfaceInstructions(): list<GameEvent>
+processEvents(): void
+inflateAnimation(time: float, position: Vector2f): void
+deflateAnimation(time: float, position: Vector2f): void
-processTextures(list_of_characters: vector<CharacterEntity>): void

Figure 5: Class diagram representing inter-class functionality and design

Group Peer Assessment

Documented below, is the group structure of how the project was delegated among members as well as member feedback in each field of assessment.

Table 2: Allocated project components for each member in group

Member Name	Assigned Project Components
Jared Ping	Report Write-up, Back-end game design, Code Implementation, Technical Reference Manual
Matthew Van Rooyen	Report Write-up, Front-end game design, Code Testing, Test Report

Table 3: Group member feedback for required fields of assessment

	Jared Ping	Matthew Van Rooyen
Group Leadership	5.0	5.0
Volume and Quality of Work Done	5.0	5.0
Depth of Knowledge and Understanding	5.0	5.0
Participation in and Contribution to Discussions	5.0	5.0
Other (if not mentioned above)	5.0	5.0
Final Average Score (Out of 10)	5.0	5.0

Decrees of Group Member Approval:

JPing

Jared Ping

MVRooyen

Matthew Van Rooyen