An In-depth Analysis of the Production and Testing of a C++17 Computer Game Based on *Centipede*

Sachin Govender (1036148) and Darrion Singh (1056673)

***Abstract*—This report documents the design, implementation, testing and analysis of a computer game based on the Atari arcade game Centipede. The game is coded with the aid of the Simple Fast Multimedia Library (SFML), however, the code structure is layered such that the game can is decoupled from the graphics library, so that it may be used with graphics libraries other than SFML. The code is separated into four layers: Game Engine, Logic, Data and Presentation. Alongside adapting the Separation of Concerns principle, inheritance, polymorphism and object orientated were thoroughly employed. These traits allowed the performing of unit testing without the aid of the graphics libraries. Unit testing was performed on the Logic, Data and Presentation layers on a class by class basis, in which all passed their respective tests. Collision detection was perform through a mixture of iterative an algorithmic means, resulting in a best case time complexity of n, and a worst case time complexity of n^2. Through comparing the game code, functional and unit testing quality to a success criteria and performing time complexity analysis on important functionality - the game designed was deemed to be a success. Despite success criteria being met, it was found that further design improvement include, but are not limited to, the implementation of hash mapping to handle collisions, or sorting game entities using a discretized coordinate system and using std::binary_search to find whether a list of entities contained a certain coordinate. The above methods of collision checking would result in a time complexity of $log(n)$, out-performing the implemented collision-checking.**

***Index Terms*—C++17 Inheritance, Polymorphism, Dynamic Binding, SFML, Time complexity, Object Orientated Design, Doctest Framework, Unit testing.**

I. Introduction

This report entails the design, implementation and testing of Centipede - a computer game based on the arcade game created by Atari. The game was created using C++17, the Simple and Fast Multimedia Library (SFML) graphics library as well as the Doctest testing framework for unit testing. The required specifications and constraints are provided in order to develop a success criteria as a standard to hold the game to. An in-depth analysis of the code structure is provided through addressing the Separation of Concerns principle. This is done by detailing the classes within each layer, the inter-layer relationships between the layers' respective classes, and detailing the use of inheritance, role modelling and polymorphism in the program design. Following the code structure, a discussion on the dynamic game behaviour is provided to understand the overall program functionality. To show program integrity, the report provides a description of the Unit Testing that the program underwent. Once the game's code, behaviour and testing procedure is documented, a critical analysis of the program design is provided. Finally, future recommendations to improve the game are discussed.

II. Game Specifications

A. Gameplay

1) Basic Overview: The two main entities within the game are the Centipede and the player-controlled Turret. The Centipede has no region restrictions on the game screen and has its movement restricted to either vertical or horizontal motion. The general movement of the Centipede entails moving across the screen horizontally and changes to vertical motion when it reaches with horizontal boundaries of the game screen or in the event of a collision with certain entities that will be discussed later. The Turret's region of movement is limited to 35% of the game screen from the bottom boundary. Its movement is also limited to horizontal and vertical motion via the keyboard arrow controls. The Turret has the ability to shoot Bullets using the spacebar key. The fundamental aim of the game is to shoot and destroy all the segments of the Centipede before any of the segments collide with Turret enough times to kill it.

2) Entity Behaviour: The two most important entities were discussed in Section II-A1 above. The other entities involved in the game are shown alongside their properties in Table I in the Appendix.

B. Game Constraints

- Game needs to be coded in ANSI/ISO C++.
- Needs to make use of the SFML libraries only.
- Must run on the Windows platform.
- The game resolution must not exceed 1920x1080.

1) Success Criteria: In order to achieve a successful game, all of the functionality shown in Section II-A1, while working within the constraints shown in Section II-B. While achieving the above, it is important to implement good coding practices such as: object orientated programming, separation of concerns, inheritance, role modelling and polymorphism. The code needs to be constructed in such a way that portions of the code can be tested in isolation and that the unit testing performed on the code effectively tests the behaviour of the game layers for a variety of situations.

III. Code Structure

A. Game Layering

The game was layered into the following four layers:

- Data
- Logic
- Presentation
- Game Engine

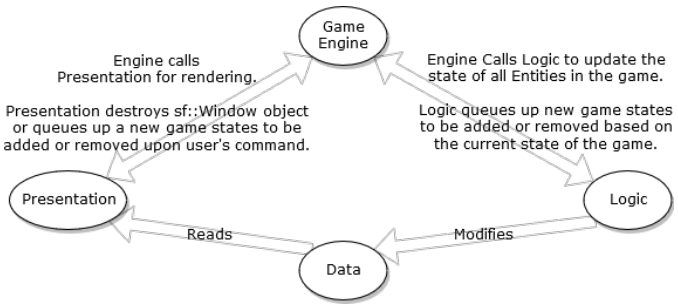

Fig. 1. Separation of Concerns employed in this design.

The code was separated into these layers in order to comply with the Separation of Concerns principle. The separation also enables the program to minimise the its dependency on the graphics library, thus making integration with alternative graphics libraries easier. Figure 1 shows the relationship between the layers. The design of the layering scheme was partially adapted from the open-source software created by Sonar Systems [3], SFML Game Development [1] and SFML Essentials [2]. Sonar Systems implemented a SFML-based version of the popular platform game FlappyBird [3]. Upon consulting the source, it was found that whilst the implementation of the Game Engine successfully initialized the game, managed and initiated changes between game states, the structure did not address the Separation of Concerns principle. This is because in most sources available, including Ref. [1] and [2], the purpose of the code is meant as an introductory guide to using the functionality of the SFML library. Unfortunately, the lack of layering strongly couples the code and SFML. As a result, these sources were considered to be irrelevant for the purposes of creating a Centipede game that is decoupled from any graphics library. That being said, the use of a Game Engine layer and the handling of various game states through the means of a stack proved to be useful.

The layering of the code was implemented through using concepts from the both the Model-View-Controller (MVC) and Model-View-Presenter (MVP) coding models. This results in the Game Engine acting as the controller in which the Logic and Presentation layers having two-way communication with the Game Engine but never with one another, thus effectively removing the game's dependency on the graphics library. This decoupling effect allows for in-depth Unit Testing of the Data, Logic and Game Engine layers in the absence of the Presentation layer.

B. Game Engine

This layer is responsible for managing the entire game, and has the following functionality:

- Initiates game start up.
- Communicates with the Presentation layer (user interface) and Logic layer (internal game mechanics) to determine what game state needs to be loaded.
- Delegates to the Presentation layer what state needs to be rendered.
- Terminates game loop upon sf::Window object destruction.
- Adds or removes states based on user commands (via the Presentation layer) or based on states automatically queued upon winning or losing the game (Logic layer).

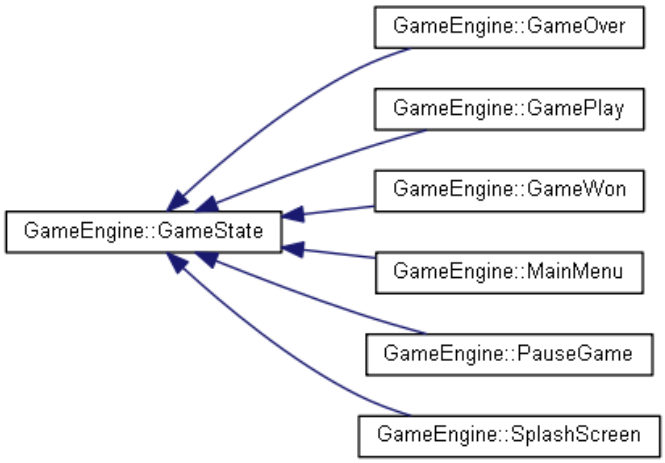

Fig. 2. Inheritance hierarchy for the **GameState** class.

These responsibilities are governed by two classes being: the **Game** Class, **GameState** Classes and the **StateHandler** Class.

1) Game Class: This class is responsible for the running of the entire game. When created, it initializes the game through loading the first state (SplashScreen) to the Presentation layer - thereafter it waits for state changes that will eventually require the program to terminate through the Presentation layer, when the current state destroys the game window object. The Game class also creates objects of the **StateHandler**, **ResourceManager** and **KeyboardControlsHandler** classes such that they can be accessed in multiple classes by means of shared pointers.

2) GameState Class: This polymorphic class is an Abstract/Interface class from which all derived classes inherit their virtual functions. These common functions are:

- *HandleInput* - Receives input via Presentation Layer.
- *Update* - Updates game data through the Logic Layer.
- *Draw* - Calls presentation layer to render changes to the screen.

Each of these virtual functions have their own implementation in each game state, based on that state's requirements. Figure 2 shows the inheritance relationship between the **GameState** class and its Child classes. The inheritance diagram for **GameState** is shown in Figure 11 in the Appendix.

3) SplashScreen Class: This class is responsible for calling the Presentation layer to draw the Splash screen. It also requests the **StateHandler** class replaces the **SplashScreen** state with a new **MainMenu** state on the stack of state pointers after three seconds has passed. The class also polls for the Escape key to be pressed in order to close the game window, terminating the program before the game begins.

4) MainMenu Class: This class does not make use of the Update function, but does handle the input of the Escape key

to exit the game window. It also handles the Space key being pressed in order to request that the **StateHandler** class adds the a new **GamePlay** state to the stack.

5) GamePlay Class: This class works closely with both the Logic and Presentation layers. The HandleInput function checks if the Escape key is pressed in order to close the game window, and checks if the Enter key is pressed to request the **StateHandler** class to add the **PauseGame** state to the stack, which pauses gameplay. If neither are pressed, the user inputs are processed by the **InputHandler** class and saved to Data layer, to be later read by the Logic layer to affect gameplay.

6) PauseGame Class: This class asks the presentation layer to draw the pause menu. The HandleInput function checks for the Escape key being pressed to close the game window, as well as the Enter key in order to request the **StateHandler** to pop the pause state off the stack, such that the previous game state (**GamePlay**) is active (at the top of the stack) again.

7) GameWon Class: This class is responsible for calling the Presentation layer to draw the screen shown when the user has won the game. The class also polls for the Escape key to be pressed in order to close the game window and terminate the program, or for the F12 key to be pressed in order to restart the game.

8) GameOver Class: This class is responsible for calling the Presentation layer to draw the screen shown when the user has lost the game. The class also polls for the Escape key to be pressed in order to close the game window and terminate the program, or for the F12 key to be pressed in order to restart the game.

9) StateHandler Class: This class contains a std::stack container of type std::unique_ptr to GameState. The current (active) game state is on the top of the stack. The **StateHandler** class performs the following functions:

- Adding (pushing) a new state on the top of the stack.
- Removing (popping) a state off the stack.
- Returning the game state at the top of the stack.

C. Data Layer

This layer is responsible for storing the data of all entities, as well as the last known input from the user.

1) Entity Class: This class serves as a parent class in which all entities in the game inherit from. Figure 3 shows the relationship between the Entity class and its respective child classes. The inheritance diagram for **Entity** is shown in Figure 12 in the Appendix. The Entity class contains the common attributes of all entities that the Logic and Presentation layers require to function. The following are the general attributes of all entities:

- Top left x and y position of the entity on the game screen.
- Center x and y position of the entity on the game screen.
- Region.
- Sub-region.
- Direction.
- Flag that tracks if the Entity object is dead. When an entity is dead, the flag is set to *true*.

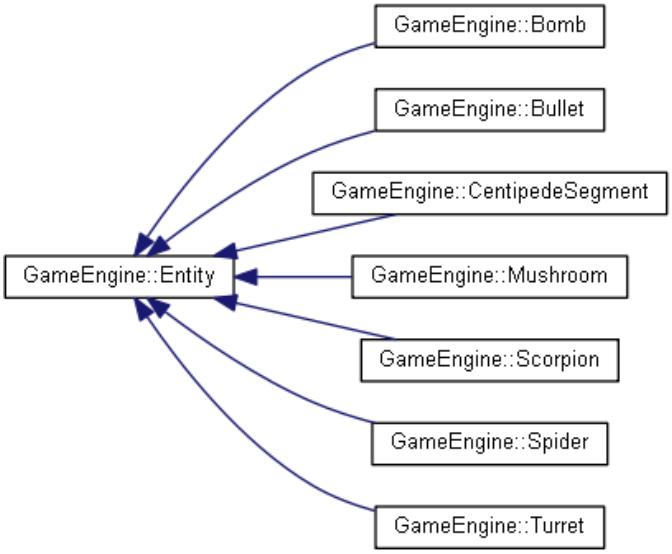

Fig. 3. Inheritance hierarchy for the **Entity** class.

2) CentipedeSegment Class: This class has extra data members upon those inherited from the Entity class:

- Segment vertical trajectory.
- Flag that tracks if the segment is the tail of a centipede section
- Flag that tracks if the segment is turning left at the next impassable object.
- Flag that tracks if the segment is poisoned.

3) Turret Class: The only additional data member that the Turret object has, upon those inherited from the Entity Class, is a vector of Bullet objects.

4) Bullet Class: Has no extra functionality over the Entity class.

5) Mushroom Class: This class has extra data members upon those inherited from the Entity class:

- The number of lives remaining.
- Flag that tracks if the mushroom has been poisoned by a Scorpion entity.
- Flag that tracks if the mushroom has been bitten by a Spider entity.

6) Spider Class: The only additional data member that the Spider object has, upon those inherited from the Entity Class, is the current angle of motion of the spider. The spider has the ability to wrap around the horizontal bounds of the game screen.

7) Scorpion Class: This class has no extra functionality over the Entity class.

8) Bomb Class: This class has no extra functionality over the Entity class.

9) Centipede Class: Contains a vector of CentipedeSegment objects.

10) GameField Class: Container class which holds a vector of Mushroom objects, a vector of coordinates for new Mushroom objects to be spawned, a vector of Spider objects, and a vector Scorpion objects.

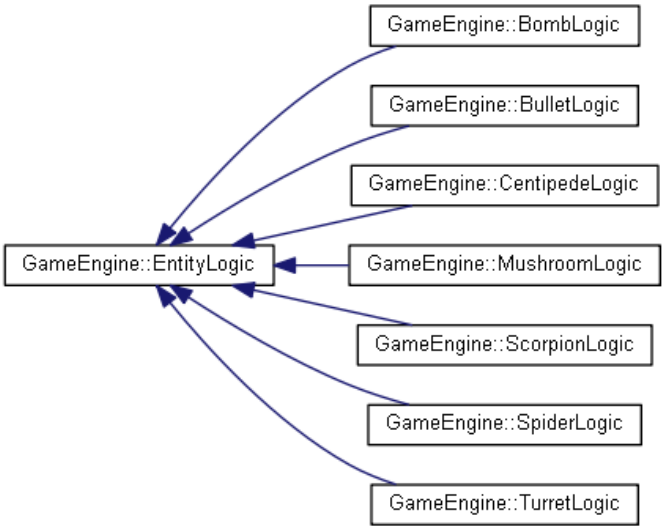

Fig. 4. Inheritance hierarchy for the **EntityLogic** class.

11) ResourceManager Class: Loads textures and stores them in an associative container. Returns the address of textures when requested.

12) HighScoreManager: Loads the high score from a persistent text file, and allows the game to access this high score, as well as change it should the user surpass this score during gameplay.

D. Logic Layer

This layer is responsible for the initiating and updating of all entity functionality such as movement and shooting, collision handling, and accessing the Data layer in order to update **Entity** and **GameField** data. Due to the layer being responsible for collision handling, it also generates the grid with game screen through mapping out all entities to a specific regions and subregions of the game screen. The two way relationship between the layer and Game Engine allows it to initiate the game state change to from the GamePlay state to the GameWon or GameOver state.

1) EntityLogic Class: This class is the Interface class from which all entities inherit their basic logic functions. Figure 4 shows the inheritance relationship between the EntityLogic class and its child classes. The inheritance diagram for **EntityLogic** is shown in Figure 13 in the Appendix. The three functions present in the class are:

- Spawning an entity (Spawn)
- Moving all existing entity (Move)
- Collision handling (CollisionHandle)

2) CentipedeLogic Class: In addition to the inherited functions from the EntityLogic class, private functions relating to the movement of the centipede help facilitate the instance of the move function. Alongside the private functions, there are additonal data memners being the speed, moved distance and number of centipede segments. The Spawn function of this class is responsible for adding centipede segments as well as determining the head and tail segments of the centipede. The Move function ensures that the direction and trajectory of the segments are correct with regards to reaching any boundaries of the game screen, as well as reaching both poisoned and normal mushrooms. CollisionHandle is responsible for removing the centipede segments that were killed during the last game loop iteration, managing the new centipede sections being formed, as well as communicating with the Game Engine to initiate the **GameWon** when the vector of CentipedeSegment objects held in the Centipede class is empty.

3) TurretLogic Class: There are no additional functionality or data members present in this class. The Spawn function of this class creates a Bullet object at the current position of the turret and stores it in a vector of Bullet objects in the data layer. The Move function makes sure that the turret never moves out of the player region of the screen. CollisionHandle either reduces the number of lives that the turret has or communicates with the Game Engine to initiate the Game Lost state when the turret has no lives remaining.

4) BulletLogic Class: This class does not make use of the Spawn function but does make use of the Move and CollisionHandle. The Move function moves the bullet in an upward trajectory and sets the Bullet object as dead when it reaches the upper limit of the game screen. The CollisionHandle function deletes the Bullet object if the Bullet is dead.

5) MushroomLogic Class: This class only makes use of the Spawn and CollisionHandle functions. The Spawn function creates the initial mushrooms in the field when the GamePlay state is loaded as well as adding mushrooms objects to the mushroom vector at the position of a dead CentipedeSegement. The CollisionHandle function deletes all dead mushrooms in the **GameField** mushroom vector.

6) ScorpionLogic Class: This class only makes use of the inherited functions from **EntityLogic**. The Spawn function ensures that there is always only one scorpion present in the **GameField** scorpion vector during the GamePlay state. The Move function makes sure that the movement of the scorpion is random and does not enter the player movement area (Turret box). The CollisionHandle function deletes the Scorpion object when it is dead.

7) SpiderLogic Class: The class has additional functionality over its inherited functionality. This functionality is found in a private function ChangeDirection. The Spawn function creates a new spider after a set time interval. The Move function makes use of the ChangeDirection function in order to get a random angle of direction for the spider to move in. The Move function also ensures the Spider objects stay within the player movement area by limiting it's vertical movements. The CollisionHandle function deletes all Spider objects that are dead.

8) BombLogic Class: This class make use of all inherited functionality, but has additional data members. These are a StopWatch object, a GameField shared pointer and floating point variable tracking time elapsed between loop iterations. The Spawn function generates four bombs on the screen at random positions. The Move function changes the position

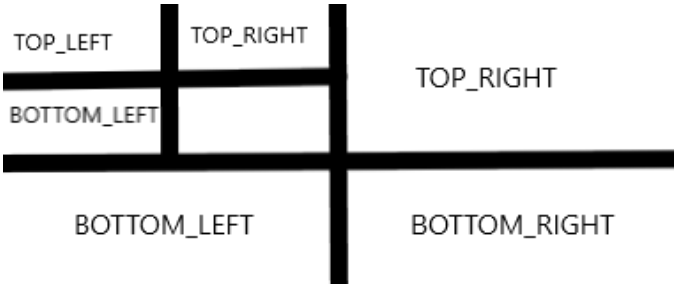

Fig. 5. Diagram depicting Region and Subregion setup. The large quarters are examples of the Region of the screen, whilst the eighths are examples of the subregions of the screen.

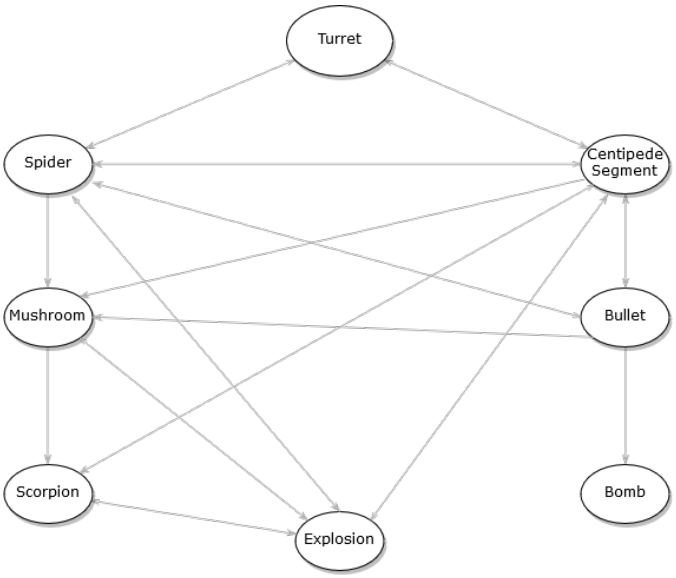

Fig. 6. Diagram showing collision interactions between all **Entity** objects.

of existing bombs after a set time interval. CollisionHandle deletes dead bombs every second.

9) RegionHandler Class: This class receives the Cartesian coordinates of the center position of an Entity object and returns the region and sub-region that the entity resides in on the game screen. The region and sub-region layout can be seen in Figure 5.

10) CollisionHandler Class: The **CollisionHandler** class determines whether or not two entities have collided. The main purpose of this class is to set the $is\_dead\_$, $is\_poisoned\_$ and $is\_bitten\_$ data members of collided objects to true, depending on the type of objects colliding. The above parameters can be found in Figure 12 in the Appendix. The actual deletion of objects are done in their respective **EntityLogic**-derived classes. The entities that are allowed to collide with one another are shown in Figure 6. Note that in Figure 6, a triggered Bomb object (represented by Explosion) has different collision rules to an un-triggered bomb (represented by Bomb). The **CollisionHandler** checks for specific entity-entity collisions whereby it first checks if the two objects are in the same region and sub-region - thereafter checking if the distance between

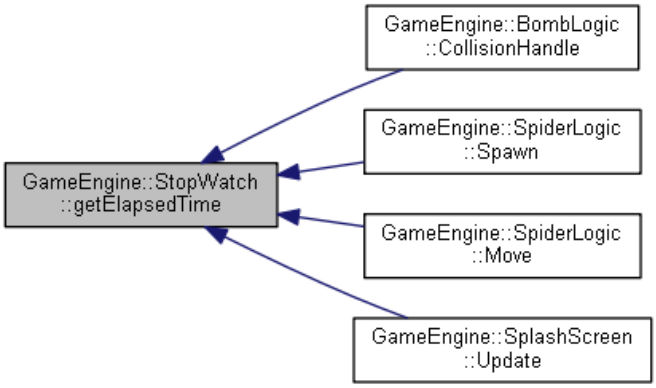

Fig. 7. Call graph for StopWatch::getElapsedTime.

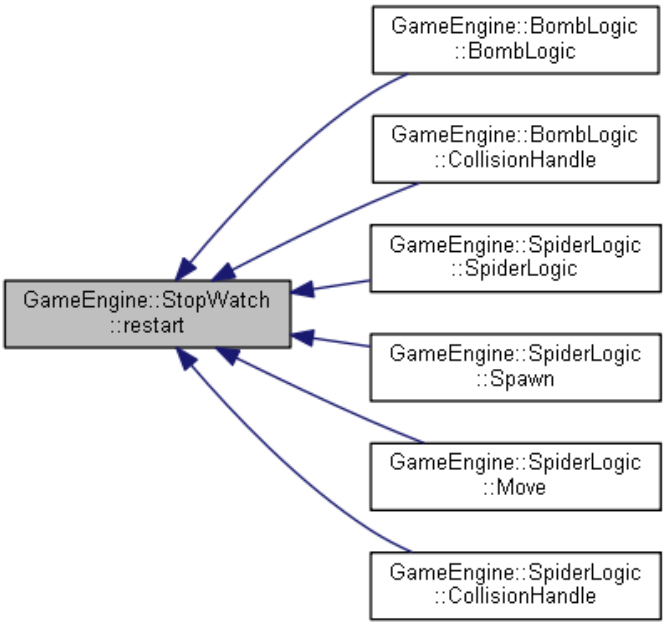

Fig. 8. Call graph for StopWatch::restart.

the two object centers coordinates is less than the sum of the two Entity object's radii. If all these conditions are met, the necessary associated boolean data members are set to true.

11) Stopwatch Class: The class is used as the timer in the game, as a replacement for the sf::Clock and sf::Time objects. It consists of a *getElapsedTime* and *restart* function. The call graphs of the class' functions are depicted in Figure 7 and 8. These illustrate which class/functions require timing to perform their operation.

E. Presentation Layer

This layer is used solely for retrieving inputs from the user and displaying the game on the screen.

1) StateRenderer Class: This renderer either updates the screen on changes made to the player's score in the current state, or renders the screen of a new game state when requested by the Game Engine.

2) EntityRenderer Class: Responsible for drawing all Entity object related, and life indicator sprites onto the current **GamePlay** state.

3) InputHandler Class: This class receives all key presses and releases as an sf::Event object. Makes use of the Direction enumeration class in order to update the **KeyboardControlsHandler** class in the Data layer. The class determines what direction key is pressed, and passes the related direction to the **KeyboardControlsHandler**. The class also debounces the Spacebar key input, and updates the isShooting data member of the KeyboardControlsHandler, representing when the user has set the shoot command by pressing the Spacebar.

F. Enumeration classes

These classes were used to improve the readability and understanding of the code.

1) Direction Class: This class is used to facilitate input handling of the arrow keys being pressed, as well as the modelling of entity movement. The two Entity objects this affects in particular are the CentipedeSegment and Turret objects.

2) Trajectory Class: Used specifically for the CentipedeSegment object in order to handle game screen boundary logic.

3) Region Class: This class is used when the Cartesian coordinates of an Entity object needs to be mapped to a region and subregion of the screen. The members of this enumeration describe an Entity object's region and sub-region, as depicted in Figure 5 above.

IV. Dynamic Game Behaviour

A. Game Loop

As discussed in Section III-B, the Game class is the component of the Game Engine that initiates the game and is where the game loop resides. The game loop runs as long as the game window is open. Depending on the state, the window can be closed in a state due to user input or the events discussed in Section III-D. Within the game loop, the game state handles input, updates data, and requests the state handler to draw the updated state. Figure 9 shows the process flow of the game loop. During the input handling and updating stages, it is possible for the game state to change. Figure 10 shows the transition between the game states as well as the means to terminate the program from these states.

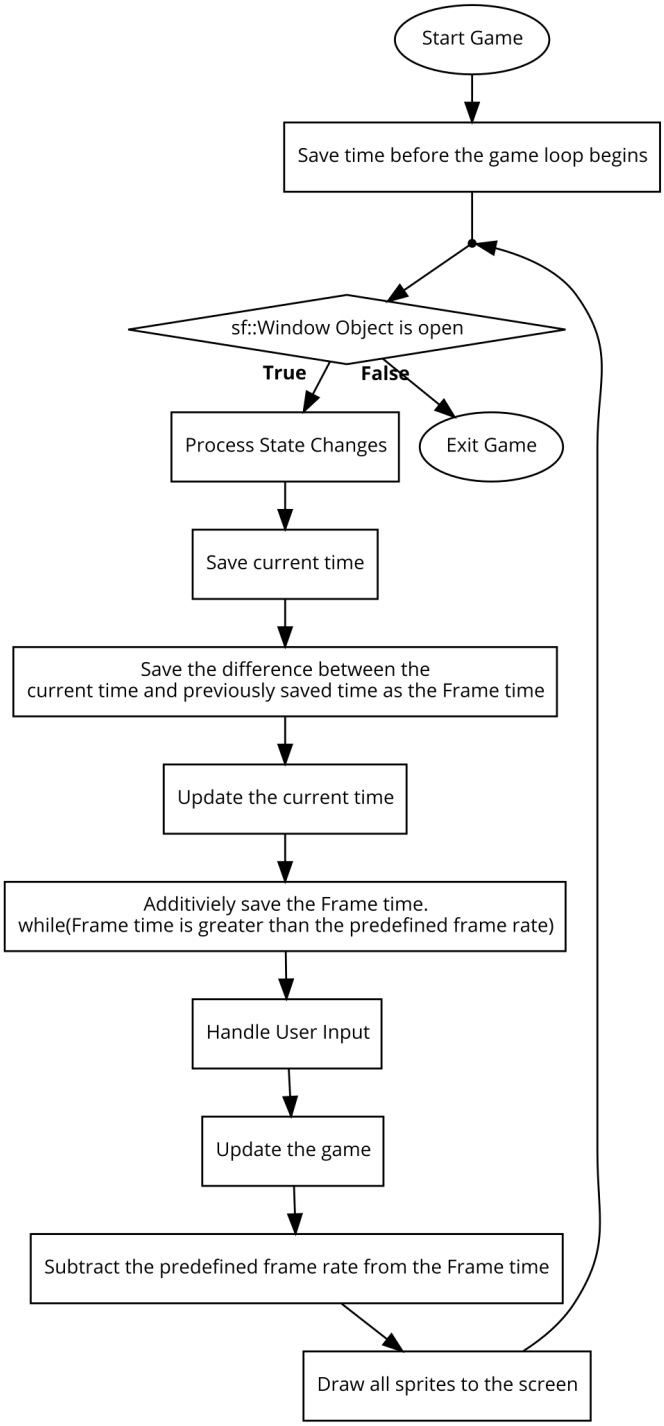

Fig. 9. Flow chart of game loop.

B. Collision Handling and Collision Checking

Collison checking/handling is managed through the *CollisionHandler* class and the CollisionHandle functions found in the **EntityLogic**-derived classes. Figure 14 in the Appendix shows the general process of the collision checking. The CollisionHandle function in the**EntityLogic** layers edit their data layers differently as per the behaviour of the various game entities described in I in the Appendix. However, only the two main entities in the game, namely the Turret and the Centipede can initiate the Game Won and Game Lost states, respectively, through the Game Engine.

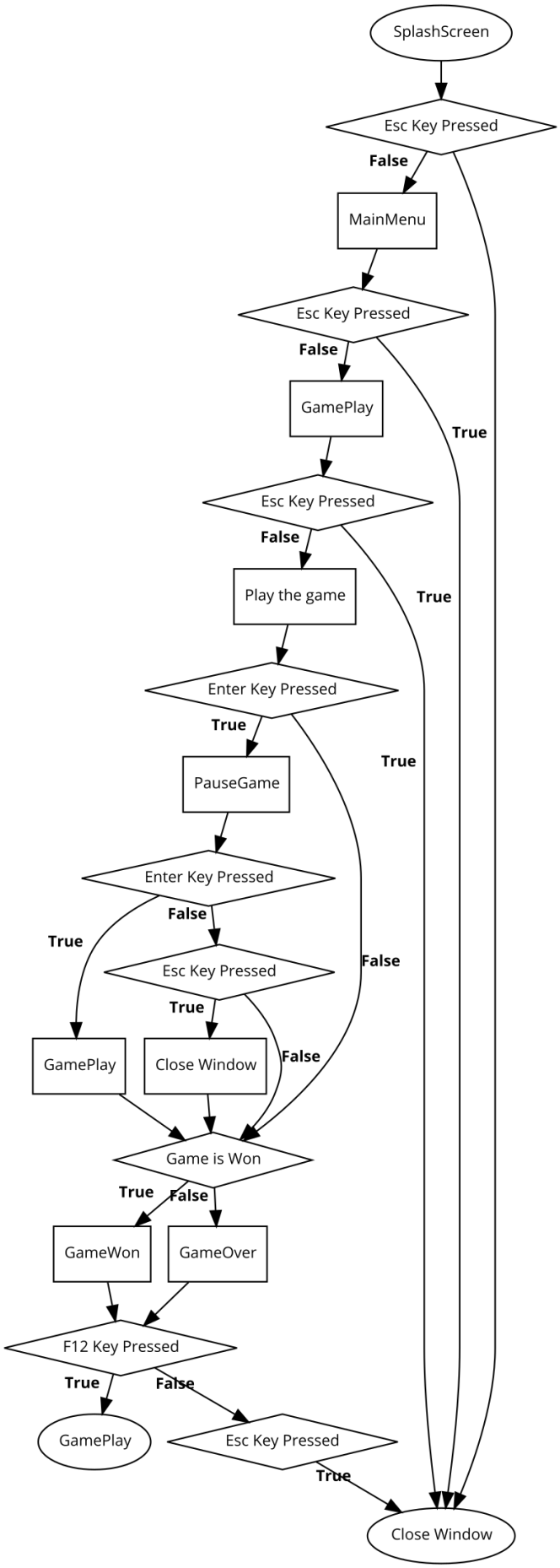

Fig. 10. Flow chart showing GameState transitions.

C. User Inputs

Non **GamePlay** states have direct access to the user inputs due to the inputs involving the exiting of the window and transitioning to another state. The **GamePlay** state requires a wider range of user inputs, and thus it needs a dedicated input handling system. In this case, the user inputs works with the **InputHandler**, **KeyboardControlsHandler** and **Direction** classes. As discussed in Section III-E, the input handler receives the user input and, with the aid of the **Direction** class, stores it in the **KeyboardControlsHandler**. Figure 15 in the Appendix shows the process of receiving the user input and storing it while the **GamePlay** state is active. The data stored in the **KeyboardControlsHandler** is used by the **TurretLogic** class to initiate movements as well as the shooting of Bullet objects.

V. Unit Testing

The unit testing was performed using the Doctest 1.2.9 framework. The testing was done in a class by class basis in which each class was tested through indpendant test files. All of these test files were linked to one executable file that runs all of the tests.

A. Game Engine

The Game Engine was not tested due to the Game class only running the game loop and has no other functionality. The non-**GamePlay** states have Presentation layer functionality that only uses the SFML library functions. The **GamePlay** state makes use of the majority of the game's classes. Therefore, it would be redundant to test the **GamePlay** state if the other classes which make up the state are being tested intensively. The **StateHandler** does not need to be tested due to all of its functionality coming from the std::stack functions, which are assumed to have no errors with respect to stack management.

B. Data Layer

1) Entity Classes: All Entity classes only have one test, being that when the objects are created that they have the correct initial values for their data members.

2) KeyboardControlsHandler: This class was tested to show that it takes in a Direction enumeration type variable and the isShooting boolean, and stores them successfully.

3) HighScoreManager Class: This class is not tested due to it's only major function being to load the high score from a text file. Normally, the appropriate test would be to check for an exceptio being thrown. However, the class has been implemented such that in the absence of the test file, the exception that is thrown is caught immediately and a new file is created with a default score of 100.

C. Logic Layer

All EntityLogic class tests perform boundary condition tests by placing the entity outside of its allowed boundary. Checks are done to see that the entity is placed back at the boundary when the entity is asked to move further across the boundary.

They also share the same collision handling tests by manually calling the *IsDead* function, and checking that the container has not changed in size, followed by calling the *CollisionHandle* function. Since this function performs the collision handling operations on the entire container, the container is checked to have decreased in size, as it is expected that objects set as dead will be deleted.

1) Collision Handler: The collision handler class has a generic function that checks the distance between the centres of all objects of super-type Entity. Roughly the same algorithm is performed on all object containers. The two Entity objects in question is placed in the vicinity of one other such that a collision should be registered between the two objects. It is expected that a collision should be registered, and the dead flags should be set.

2) RegionHandler: Various Cartesian coordinates were passed into an instance of the **RegionHandler** class, and checks were made for the expected region and subregion to be returned.

D. Presentation Layer

The only component of the Presentation layer tested is the **InputHandler** class, as the renderer classes rely only on SFML functionality. Thus, there is no need to test SFML functions.

1) InputHandler: Various SFML Key Press events were sent to the **InputHandler**. The arrow key presses were checked to return the correct Direction enumeration to the **KeyboardControlsHandler**. The spacebar presses were checked if they had set the is_shooting flag to be true in the **KeyboardControlsHandler**.

VI. Critical Analysis

A. Program Design

As mentioned in the sections above, there is an extensive use of inheritance and polymorphism in the program design. This is seen in the *Entity* class, *EntityLogic* class, and the *GameState* class.

1) Inheritance: The Parent *Entity* class makes extensive use of Inheritance, the effects of which can be seen in the Child classes of Entity, whose declaration only contains attributes that are specific to that particular object. For example, only *CentipedeSegment* objects are concerned with whether they considered to be a head, body, or tail segment. These characteristics are captured in the class interface, where it is possible to read these attributes. However, both *CentipedeSegment* objects and the *Turret* object both contain Cartesian coordinates, and it is possible to read this attribute from either object by use of the same function. Other characteristics such as whether or not the object is dead is also common to both *CentipedeSegment* objects, the *Turret* object, and all other entities in the game, since all entities must have the ability to die upon fatal collision with another entity.

The Liskov Substitution Principle is demonstrated in the *CollisionHandler* class, where the various *Entity*-derived objects have the distance between their centre coordinates calculated. The function that performs this check, namely *CheckDistanceBetweenEntities* is seen to take in references to two objects of type *Entity*. This demonstrates that the function does not know the subtype that it is operating on. This therefore demonstrates the ability of the Child classes of *Entity* to be operated on as if they were the *Entity* class, or in other words, the ability to be *substituted* in the place of an object of type *Entity*. As a result, this opens up the opportunity for further improvements to be made to the client that performs general operations on all *Entity*-derived objects, as opposed to subtype-specific functionality.

2) Polymorphism: The Parent *EntityLogic* class makes extensive use of polymorphism, as it is an Interface class. The affects of this can be seen in the *GamePlay* state, which is the version of the game loop that runs whilst the game is in play. Upon examination of the member functions of *GamePlay*, it is seen that code structure has been greatly improved because of the ability for common functions for each instance of the *EntityLogic* class to be grouped together.

Another advantage of using this approach to designing the*EntityLogic* class in this manner is the presence of dynamic binding. This dynamic binding is implemented through the extensive use of smart pointers for all instances of *EntityLogic*, namely through the use of *std::unique_ptr*. All *EntityLogic* instances are declared through unique pointers, however they are made to point to a *EntityLogic*-derived object, such as *CentipedeLogic*, *BulletLogic* etc.

There are both advantages and drawbacks to the use of polymorphism and thus dynamic binding to implement Centipede. The main advantage of using polymorphism is the ability to write generic code such that it reads as if the code is written in an interpretive language. The base type is preserved, whilst using the functionality of the derived type. For the purposes of game development, which may rely on the ability to easily add features and improvements without changing the class interface, the use of polymorphism is justified.

However, this comes at the cost of computational expense. Because an object of base-type is pointing to an object of derived-type, it will only undergo type deduction at run-time. This is computationally expensive. The process of dynamically binding generic functions as opposed to statically binding less generic functions is also computationally expensive. This is because a virtual table is needed to select the correct function implementation to run. This is also true of all other virtual functions used in this program.

Despite this, a polymorphic design was chosen because of it's flexibility. An example of the power of polymorphic design is seen in the game loop in the *Game* class. The game loop calls the same three functions in every iteration, despite not knowing what subtype of *GameState* is at the top of the stack. This allows for other states of the game, such as a level system to be easily implemented, since each level would just need to be implemented as another subtype of *GameState*. This complete decoupling of the game loop from the current state of the game allows for the Game Engine layer to not only be reused, but to allow for multiple instances of the game loop to

run independently of each other, which creates the opportunity for multi-player capabilities.

B. Collision Checking

A crucial element of the game that makes a significant contribution to the the performance, and is by far the most computationally expensive process of the game. Multiple functions perform collision checking, each with a slightly different computational toll than the next. The std::find() algorithm has a time complexity of n, and thus seemed the most appropriate choice to use for collision checking.By using std::find(), this means that collision checking was performed using a mixture of algorithmic and iterative techniques. Since there are a limited number of Turret objects (one), CentipedeSegment objects (twelve), Bomb objects (four), and Scorpion objects (one) during any point in the game, all collision checking involving the Turret, CentipedeSegment, Bomb, and Scorpion objects can be approximated to a time complexity of n. For objects that have no limit to the number of possible instances, such as Bullet objects, Spider objects, and Mushroom objects, the time complexity of collision checking is n^2. The above time complexities represent the worst case scenarios of the collision checker.

However, it is noteworthy to mention that despite the worst case complexity calculations, there are multiple ways in which the total number of computations per iteration of the game loop is brought to a minimum. This is due to the Region and Subregion checking that is performed using std::find(). Whilst all objects are checked to see if they fall within a specific Region or Subregion, only those that fall into that particular $\frac{1}{16}^{th}$ of the window have mathematical computations performed on them. This means that during normal game play, assuming an absence of the worst case scenario (i.e. objects exist at all points of the window), most objects do not have mathematical calculations performed on them. In addition to that, as seen in the *CheckCollisions* function, checks are only performed when necessary. Game-losing computations take precedence over game-winning computations. If the game has not been won or lost, only then do computations that affect the player's score take place, followed by the rest of the computations that only affect dynamic game behaviour. As seen, whilst the number of checks may still be high, the number of mathematical computations is brought to a minimum whenever possible by only performing checks when they are necessary to winning or losing the game.

VII. Improvements

A. Collision Checking Improvements

As stated in the previous section, the collision checking was performing using an algorithm with time complexity n. An alternate way of performing collision checking would be using hash mapping. This would be possible by restricting the positions of all Entity objects in the game to a particular set of coordinates on the screen, and hashing the coordinates of the Entity object - storing it in a hash-table. Each iteration of the game loop would then check for collisions in the hash table, returning a pointer or a reference to the Entity objects that have collided. The collision in the hash table would only occur if the Entity objects have the same coordinates, thus inferring a collision between multiple Entity objects. Object collision handling would then be performed, based on whether the objects in question are allowed to collide, or whether they are allowed to pass through each other. This can be done using *std::unordered_multimap*, or the *boost::rtree()* function. Both these associative containers have a find() function which has a time complexity of $log(n)$, which would out perform the current collision handling technique.

Another alternative would be to discretize the x and y coordinates of all Entity objects, and store them in a *std::list* container. The next step would be to sort the list, such that *std::binary_search* can be performed on it, also having a time complexity of $log(n)$. The search would return true if a particular coordinate was found in the list, namely, the coordinate of an Entity object on whom collision checking is being performed. If the search returned true, then a collision would be registered between an object being checked and the object in the list, and would thereafter be appropriately dealt with.

VIII. Conclusion

The design, implementation and testing of the Centipede game was given. The expected program specification was established through analyzing the necessary gameplay behaviour, game constraints and established a success criteria. Once the specifications was thoroughly discussed in detail through showing the implementation of the Separation of Concerns principle, the functionality of each class in the layer was discussed. Explanation of the implementation of good coding practices such as inheritance and polymorphism was also discussed, including advantages and drawbacks.

Once a firm foundation on the code structure was provided - the dynamic behaviour of the game was discussed through analyzing the events that occur in the game loop, collision handling and user input handling. Thereafter, a brief discussion on Unit Testing was provided, which resulted in the **Entity**, **EntityLogic**, **CollisionHandler**, **RegionHandler** and **InputHandler** classes being found to be the most important classes to test.

A critical analysis of the collision checking algorithm was performed. The findings thereof was that despite the multiple checks implemented to decrease the number of mathematical computations (namely the calculation of distance between Entity centres), the best case time complexity of the collision checking algorithm was n and the worst case was n^2. A major computational improvement that would eradicate this performance drawback would be the implementation of a hash map, which performed hash collision handling for objects on the basis of a predefined coordinate system to which entities adhered, such that all possible positions could be stored in a hash table.

REFERENCES

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

[2] Milchev, M.G., 2015. SFML essentials. Packt Publishing Ltd.

[3] Sonar Systems, 2014. Flappy Bird C++ Tutorial 1 - Overview. https://github.com/SonarSystems/Cocos2d-x-Flappy-Bird-C—Tutorial-1—Overview

APPENDIX

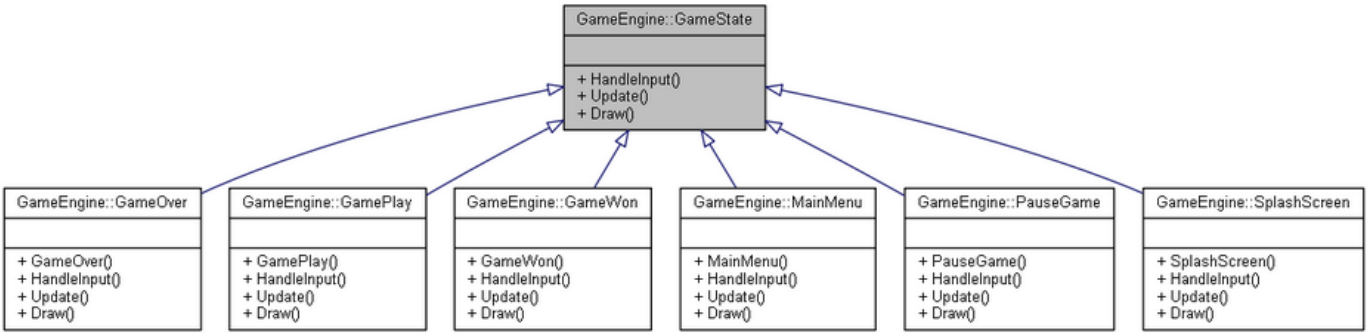

Fig. 11. UML Diagram of GameState and its children classes.

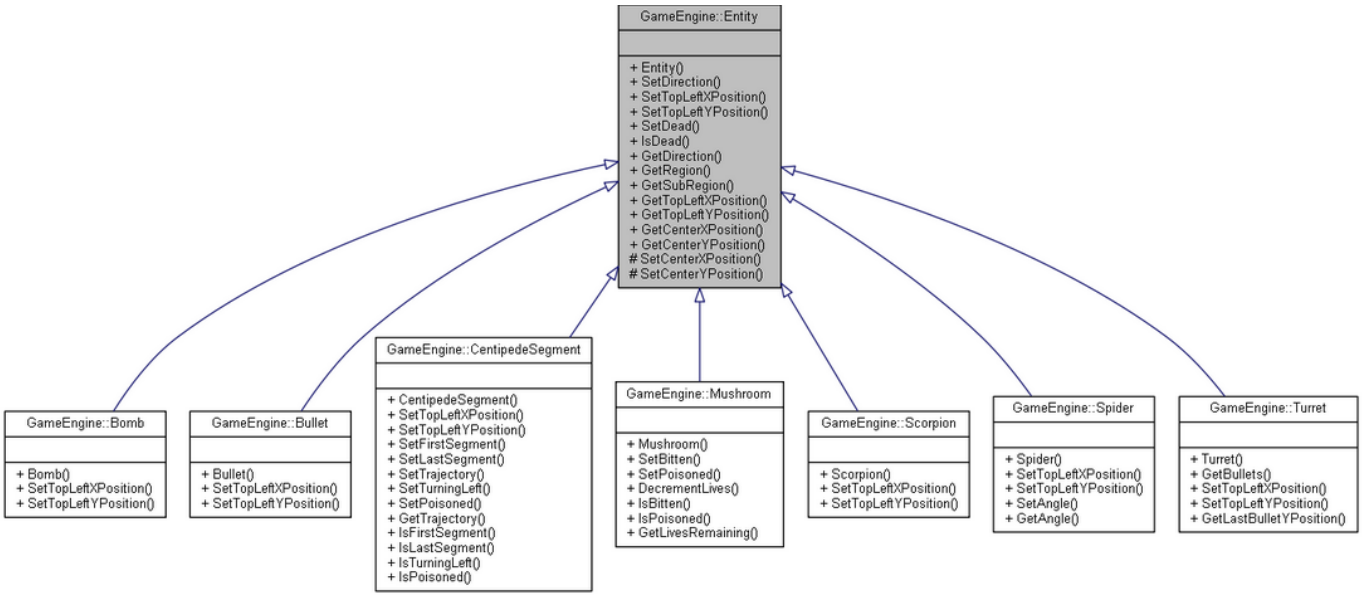

Fig. 12. UML Diagram of Entity and its children classes.

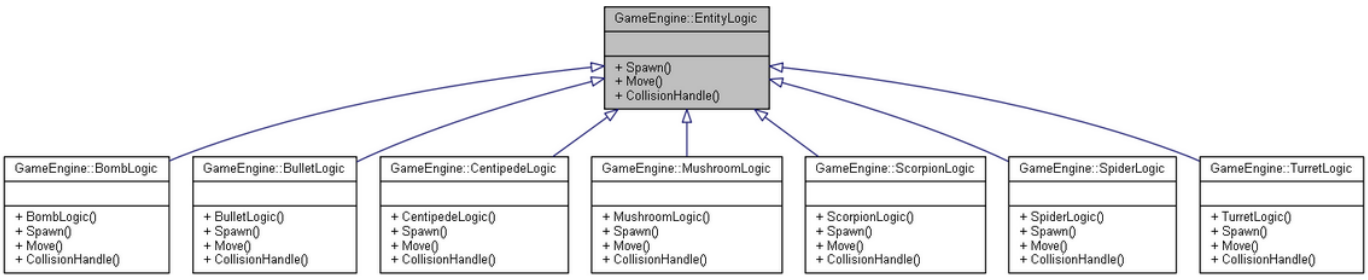

Fig. 13. UML Diagram of EntityLogic and its children classes.

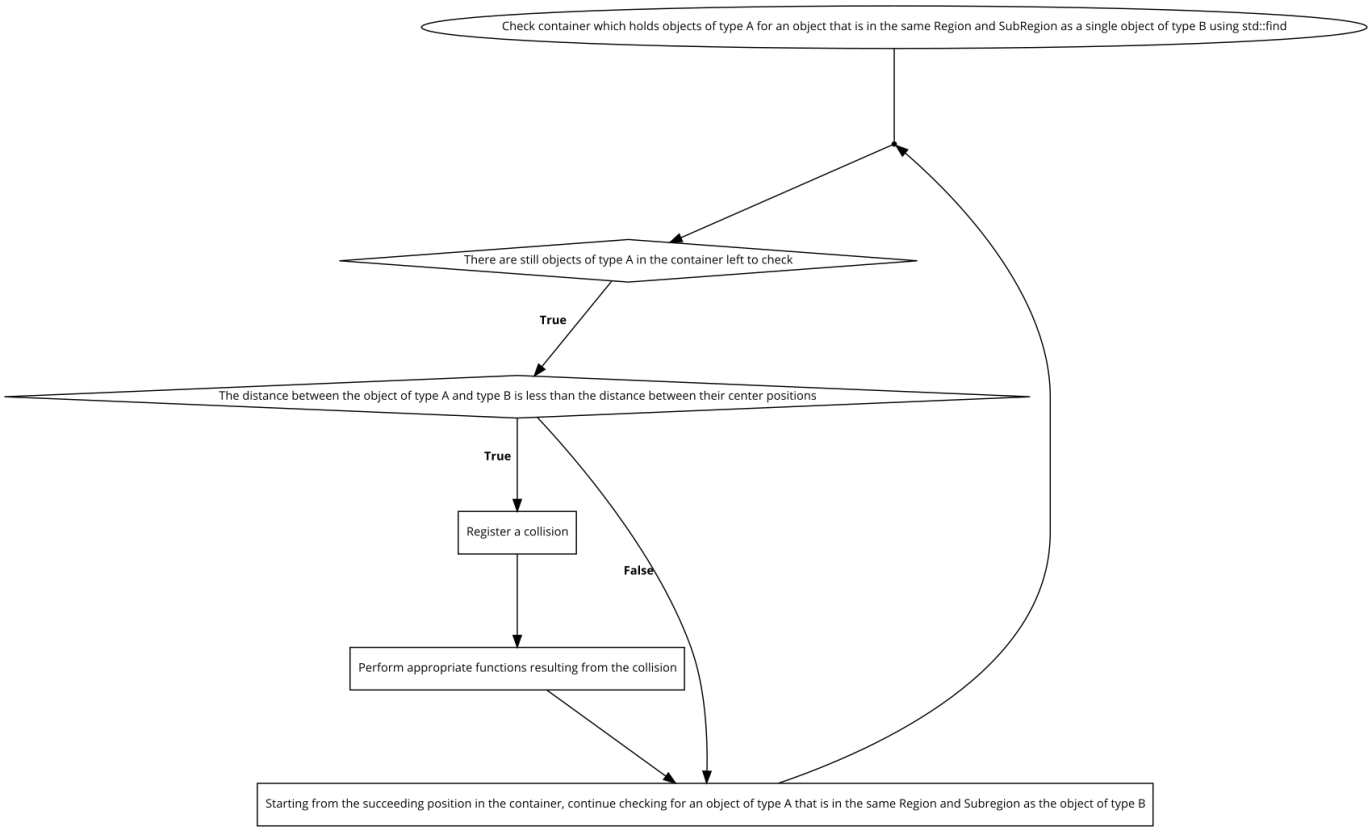

Fig. 14. Flow chart showing the general process of collision checking.

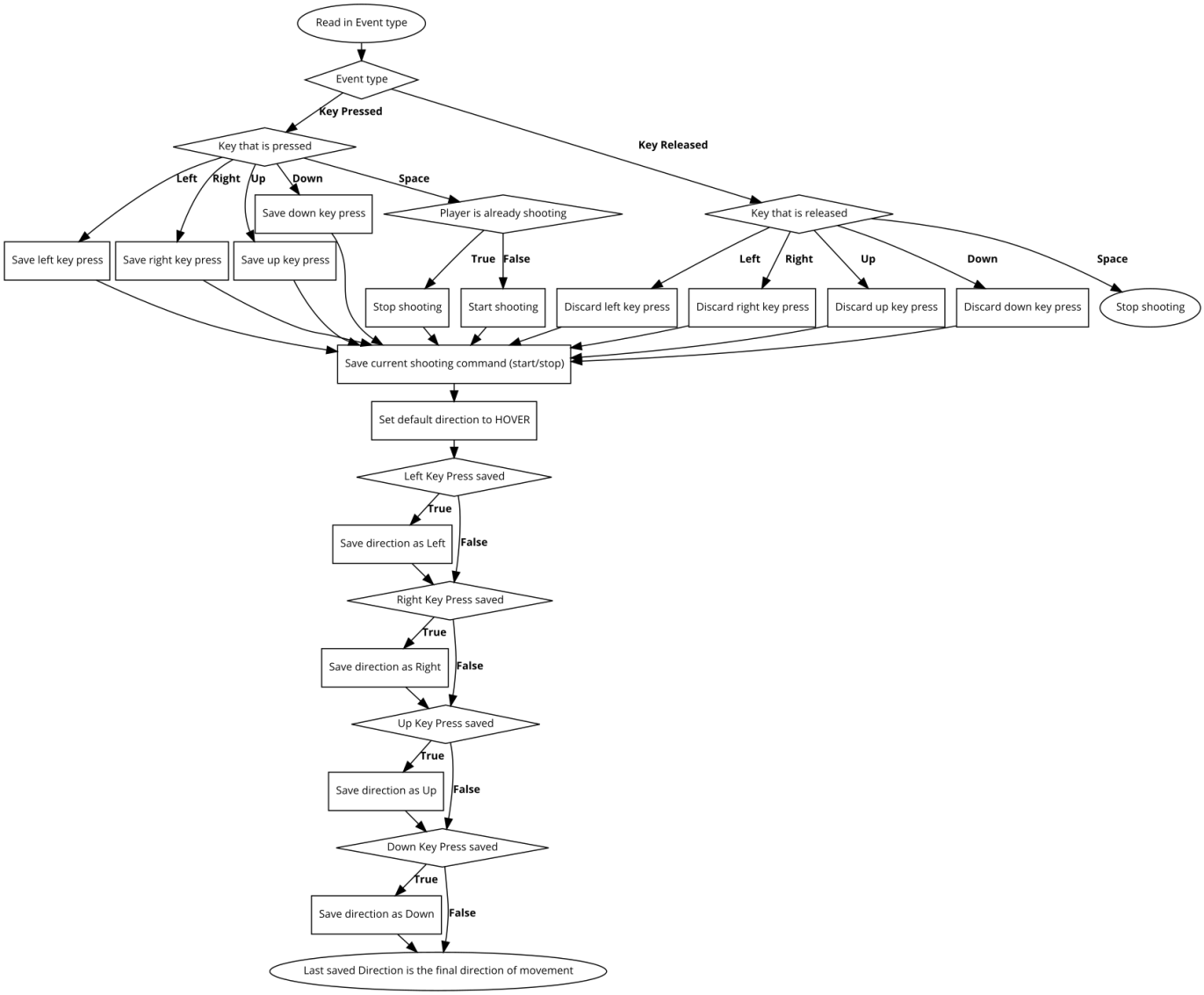

Fig. 15. Flow chart showing the process of saving user input.

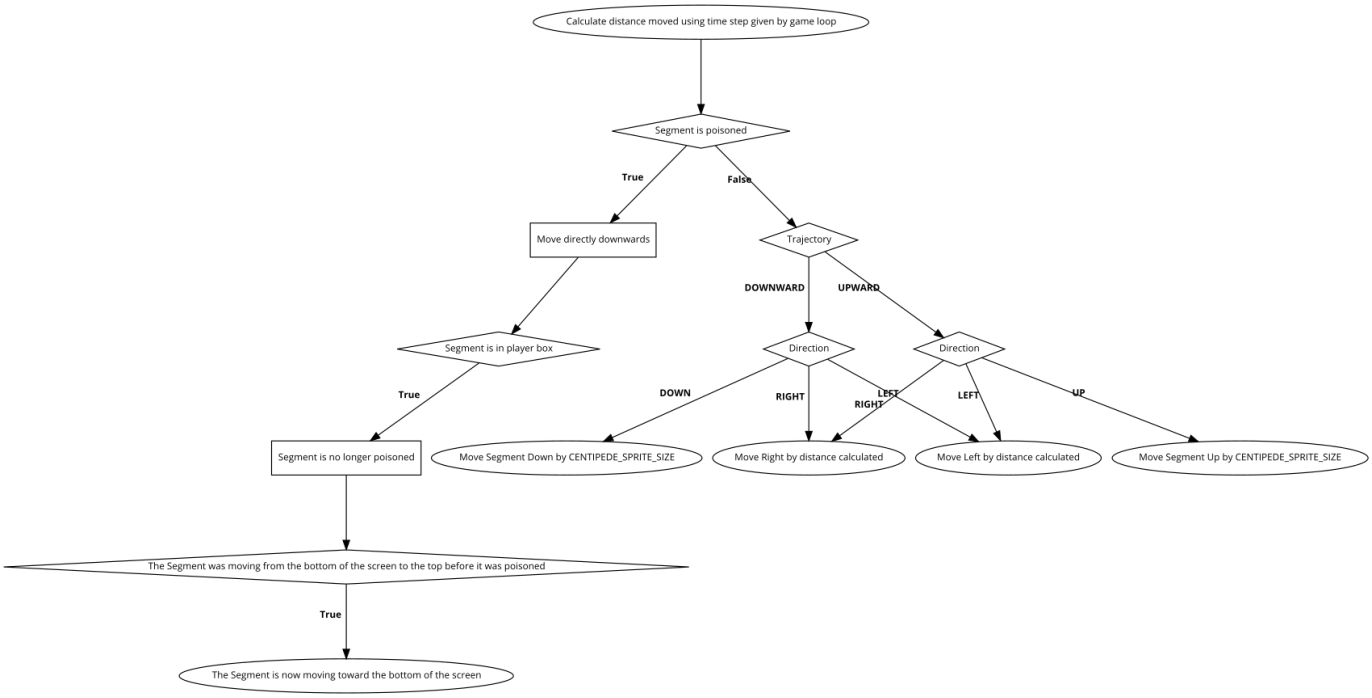

Fig. 16. Flow chart showing the process of the CentipedeSegment object Move function.

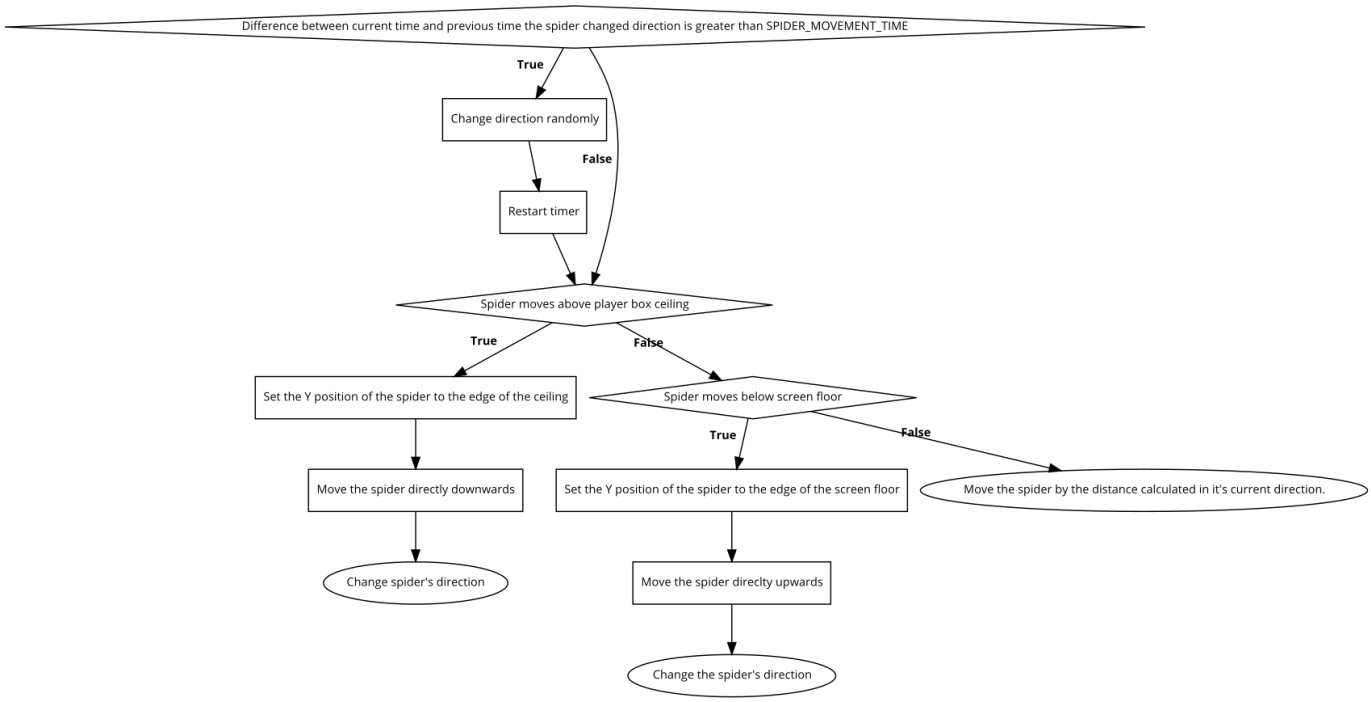

Fig. 17. Flow chart showing the process of the Spider/Scorpion object Move function.

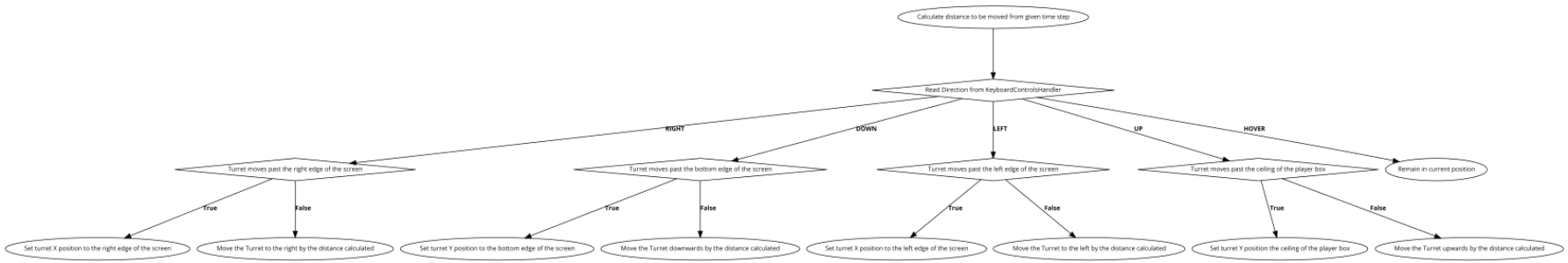

Fig. 18. Flow chart showing the process of the Turret object Move function.

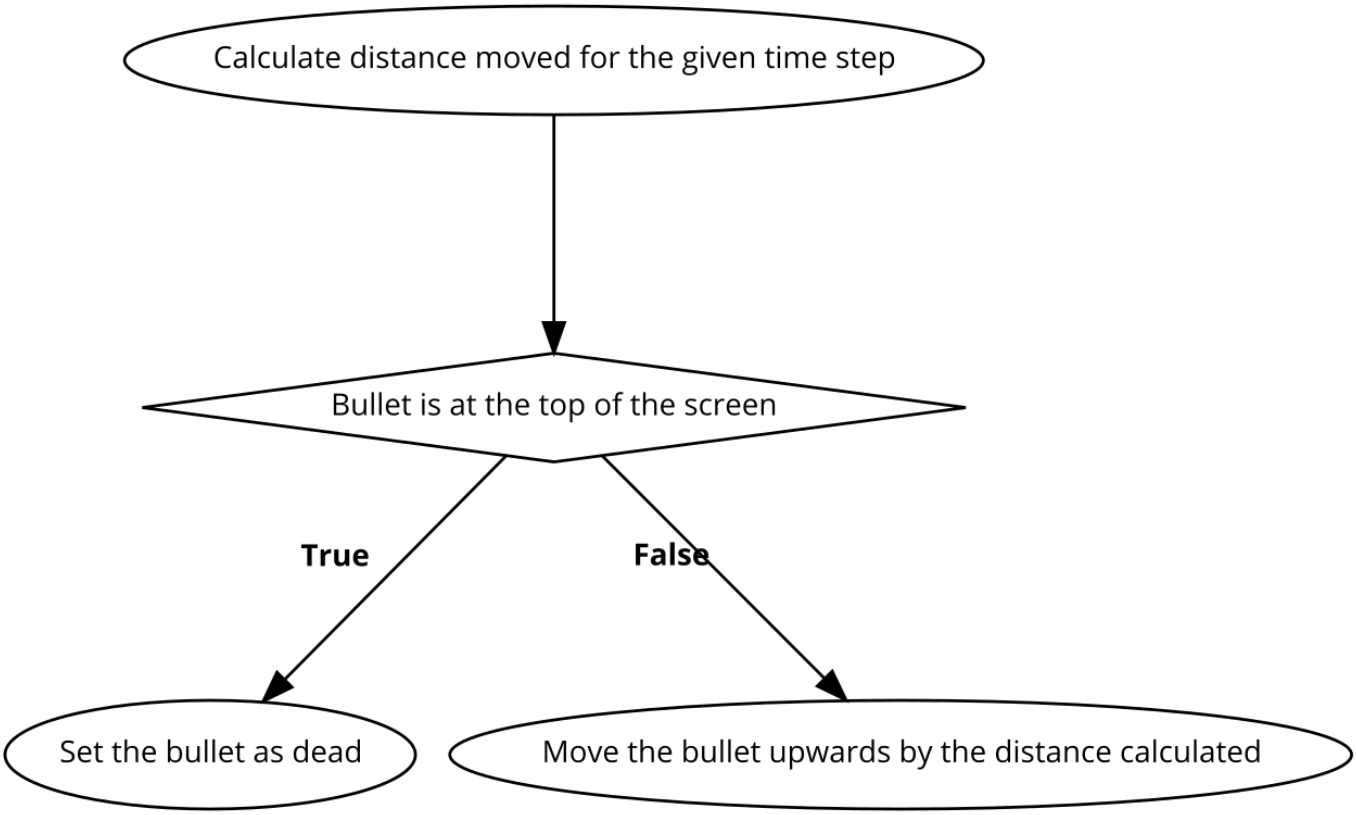

Fig. 19. Flow chart showing the process of the Bullet object Move function.

TABLE I
ENTITY PROPERTIES

Entity	Spawn Response	Region of Existence	Movement	Collisions	Number of Lives	Special Features
Mushroom	Populates the screen when the game begins. Spawns at the position of a dead Centipede segment.	Top three quarters of the screen.	None.	Collides with all Entities except the Turret, and an un-triggered DDT bomb.	Four	Upon collision with a Scorpion, becomes poisoned. Sections of the Centipede which collide with a poisoned Mushroom travel vertically until they reach the ceiling of the Turret box.
Scorpion	Spawns whenever there is no Scorpion in the game.	Above the Turret box.	Random movement, 360 degree range of motion.	Collides with Mushrooms and triggered DDT bombs.	One	Upon collision with a Mushroom, turns the Mushroom into a poisoned Mushroom.
Spider	Spawns two seconds after another Spider in the game is killed. If none has been killed, spawns two seconds after the last Spider was spawned.	Inside the Turret box.	Random movement, 360 degree range of motion.	Collides with Bullets, Mushrooms, and the Turret.	One	Upon collision with a Mushroom, has a chance to instantly kill the Mushroom.
DDT Bombs	The game checks whether there are four Bombs on the screen every second. If there are less than four, the gamespawns one Bomb.	Anywhere on the screen.	Does not move like other Entities; changes it's position to a random position on the screen every two seconds.	Un-triggered bomb: Only collides with Bullets. Triggered bomb (Explosion): Collides with all Entities except Bullets and the Turret.	One	Upon collision with a Bullet, triggers an Explosion. Explosions are instantly fatal to all Entities except Bullets and the Turret.