Centipede Game Development Project Documentation

Sanele Ndlovu-716411
Simbarashe Fadzi - 745118
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg
ELEN3009:Software Development 2

***Abstract*—This report documents the design and implementation of a PC based remake of the the game Centipede. A layered object-oriented (OO) approach has been used to design the game implemented in ANSI/ISO C++ in conjunction with the SFML 2.5.0 graphics library. The application layers are separated using the Model-View-Presenter design architecture pattern. The 'model' layer contains the game entities and logic modelled partly by inheritance and mostly by composition. The 'view' renders all of the game objects using 'presenter' as a communication interface with the 'model' encapsulates all of the core game logic. The 'presenter'(GameProcess) creates a link between the 'model' and the 'view' layers sending and receiving instructions for state updates. Basic game functionality, three minor features and two major features are achieved. All the game objects required for this specification are present (Centipede , playership spider, bullets and mushrooms). Response to the user-input is implemented accordingly,centipede movement and collision with mushrooms. Maintainability of the solution is increased by the use of separation of concerns where classes are abstract and model the simplest object responsibilities. Design flaws such as role modelling movements, lack of separation of concerns to its granularity in the model exist. inheritance is not used to its full capabilities to avoid re-use. Possible functionality improvements could include the addition of scorpions and game levels. Technical improvements include advanced unit test for the GUI and GameProcess logic implementation to allow more extensive unit tests to object behaviours.**

1 INTRODUCTION

Atari developed and published an arcade game named Centipede in 1981. Centipede became popular and was released on major personal computer platforms like Apple and IBM.The game is a fixed shooter arcade with a 2D perspective gameplay with the shooter moving within a confined perimeter at the bottom of the screen.

The Centipede game involves the main character a humanoid shooter destroying enemies. The shooter is able to kill the centipede segments by shooting them and dies when any collision with the multiple segments present occurs. There are other types of enemies which appear at irregular periods during the game these are scorpions, spiders and fleas. The scorpions appear moving horizontally poisoning mushrooms, and if the centipede comes into contact with these hurtles towards the player area. Fleas appear when there a less than five mushrooms in the player area, they drop vertically and leave additional mushrooms in their path. Spiders move across the screen in a zig-zag manner and occasionally eat mushrooms in their active lifetime.

This report documents the recreation of a Centipede type game, meaning the PC recreation mimics the major game play features but the character types are not exact and other features are not in the project scope. The Centipede recreation is done via the C++ programming language and the SFML library.

The overview of what is documented is as follows: a brief background on the problem and the requirements for a successful implementation are discussed. The design techniques used to solve the problem and the game architecture are discussed in section 3. Section 4 discusses the abstractions and object interactions, and finally unit testing with a critical analysis of the the whole design is given.

2 BACKGROUND

The aim of the project is to design a PC based Centipede like game built on the C++ language and the SFML library. This section is based on [1], and explains in part the approach taken to the problem resolution.

2.1 Object Oriented Programming (OOP)

Object Oriented Programming (OOP) is a design philosophy where a model is made of groupings of self sustainable objects. The objects then constitute a class which models the responsibilities pertaining to the object. This design method allows for the use of inheritance and composition necessary in the game development to unpack the problem complexity. The Centipede game contains objects with different responsibilities therefore modelled by different classes but need to interact by sending messages to each other. The concepts of the OOP used here include encapsulation where the logic is hidden within the class but can request input data of what to do, the game has movable objects the logic behind their motion is encapsulated but can request which object should move. Table 1 summarises the movable entities behaviour identified for the OOP.

TABLE 1: Movable Entities Behaviours

Entities	Behaviour
Centipede	Moves left and right along the horizontal axis Moves up and down along the vertical axis Collides with mushrooms and player-ship
Spiders	Moves vertically from the center Collides with player and eats mushrooms
Bullets	Move vertically Collides with enemies and mushrooms
Player	Moves left and right along the horizontal axis Moves up and down along the vertical axis Collides with spiders and centipedes

2.2 Inheritance

This is the ability of a subclass to have attributes of the superclass and models an "is-a" relationship. The subclass inherits the methods of the superclass and can extend their functionality by having subtypes or overriding functions. The use of inheritance in OOP is beneficial as it provides re-usability of the superclass and increases program reliability and speeds up program development. This is a key coding practice (DRY) avoiding repetition and allows role modelling distinct objects. This principle was attempted in modelling movable entities with similar behaviours by creation of abstract base classes to allow the overriding of the movement function signature,and each object has its respective movement implementation. The design idea being modelled here is that a player "is-a" movable entity in the game in sharp contrast to that a mushroom "is-a" stationary entity.

2.3 Composition

Composition models a "has-a" relationship between objects. This is the prevalent design approach with the solution presented. Classes are created at a fundamental level to model the simplest behaviours. For example the the centipede is a collection of segments which follow the same movement rules implemented. The centipede "has-a" segment. Using this approach decreases chances of creating monolithic classes handling multiple responsibilities, instead an object of centipede can create a segment and segment knows its own responsibilities.

2.4 SFML

Simple and Fast Multimedia Library (SFML) is a multimedia library which can be used for game development and providing a simple application programming interface (API). SFML is split into five multimedia modules, but the core modules used are system, window, graphics. The system module provides a vector, unicode string and timer facilities. The vector class useful in creating a 2D coordinate system. The graphics module used in hardware acceleration of the sprite objects created and texts. The window module used to render the objects and input device support when playing. The SFML library also includes built in logic functions used to simplify the implementation of the game, the rotate and scaling of the game objects to meet the basic functionality required.

2.5 Project Specifications and Constraints

- The game must be coded in ANSI/ISO C++ language only
- The SFML 2.5.0 graphics is the only one to be used, additionally no other libraries built on SFML to be used
- The game should run on a windows platform
- The game display screen pixel size should be 1920 x 1080
- No OpenGL is to be used

2.6 Success Criteria and Assumptions

With the use of agile methodologies an iterative design of the solution with continual analysis and re-implementation is used. The project submission was split into three interim submissions. The project brief document [1] specifies the required deliverable for each submission. Therefore for the success criteria to be met the project deliverable for each submission were required to be met. The game was also required to make good use of OOP and unit testing. It is assumed that other developers might adopt and improve on the program, therefore the design was designed to be simple reliable and easily maintainable.

- Analysis of a software problem containing numerous interacting objects using Object-Oriented (OO) programming
- Design and implementation of the OO solution in C++
- Use unit test suites to verify the software legitimacy

- Use existing software libraries to support the code
- Provide in-depth and adequate documentation of the solution, and generation of a technical manual
- Achieve at least basic functionality to ascertain a pass

3 Design Architecture

A combination of layered and OO architectures was used in the project. Functionality within each layer is related by a common role or responsibility. Using a layered architecture allows for objects to be individually tested and manipulated within the code. This approach supports flexibility, maintainability and re-usability of code.

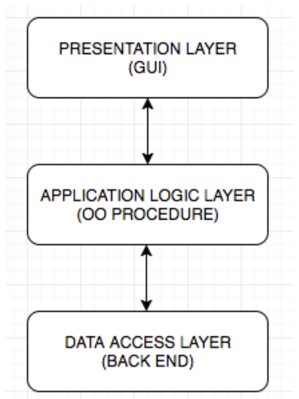

Figure. 1: Layered Design View

The OO architecture tackles the fundamentals of good design and presents minimal classes within the code structure. Figure 1 shows the layer relation of the architecture. The layering targets the design principle noted as separation of concerns (SoC). In SoC the application layers presentation, domain and data are modelled differently. These concepts are explored in detail in the next subsection.

(a) Model-View-Presenter (MVP)

In order to isolate the Presentation layer and the logic layer, a connection must be setup to communicate between the different layers. The two most common design patterns used to separate the presentation layer from the logic layer are: Model-View-Controller (MVC) Model-View-Presenter (MVP). An MVP design patterns have been used in this project which is a derivation of the former, providing for low coupling between model and view [2]. Figure 2 shows the layer interactions and the responsibilities it holds.

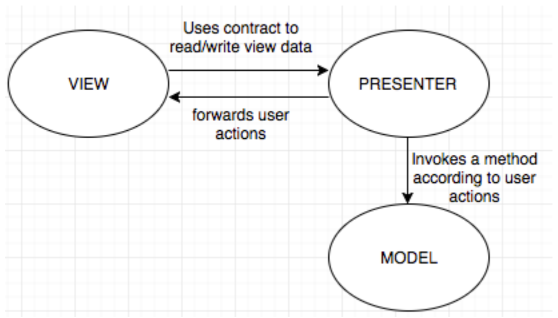
Uses contract to read/write view data
VIEW
PRESENTER
forwards user actions
Invokes a method according to user actions
MODEL

Figure. 2: Overview of MVP architecture

Model

The 'model' expresses the application behaviour relating to the problem domain. This corresponds to the data layer of the software. This exposes the services offered by 'model' to presenter but encapsulating internal detail. This data in the game design is modelled in GameEntity, which contains the position of the game entities via their sprites and the drawable type of sprite extended to view.

View

The 'view' handles the visual representation of the data, the user inputs, and is updated by presenter information. This corresponds to the presentation layer of the software. The presentation layer receives user actions, relays these to the 'presenter' and uses this result to interface the information. The presentation layer receives updated states from the 'presenter', and renders these on the screen. The Game class makes up the 'view' aspect of MVP. The Game class uses the 'data' from model to know the location to draw the sprite object of GameEntity as it relates to the object passed to it as an argument. This type of separation allows for view mock test not dependant on the 'model' layer.

Presenter

The presenter is the immediate interface between 'model and 'view'. The presenter receives the instructions from the user actions via 'view' and forwards these to the 'model'. The services from the 'model' update the states of the game forwarding this information to the 'presenter' enabling view to render the update. The 'presenter' has a contractual relation between the the 'model and 'view'. The class of PlayerShip uses the keyboard inputs to update the sprite position by either adding or subtracting the entity speed to the position as instructed by logic. Presenter manipulates data as directed by logic, Score binds an int to a text type object such that 'view' can access the manipulated data to display.

4 Class Responsibilities

The following section provides a detailed low-level description of the class responsibilities. These classes constitute to the implemented MVP design architecture, therefore are presented accordingly. The class names without their extensions are used as references to the classes.

4.1 View Classes

The following class forms part of the View layer of the design architecture.

Game

As part of the View layer, providing interaction with the user, this class has two main purposes. Firstly, this class is responsible for receiving user inputs and communicating these inputs to the presentation layer for further processing. Secondly, this class receives all game entity updates from the presentation layer and displays these updates to the user.
The private member functions in figure 4 illustrate the responsibilities of this class. To receive user inputs the function *processEvents_()* was implemented from which an object representing the presentation layer, *process_*, is used to communicate these user inputs to the presentation layer. The function *render()_* is responsible for processing updated game states which are communicated by the presentation layer. Thereafter with the use of these game states, the relevant drawer functions are called to draw the updated states and display them to the user.

4.2 Presentation Classes

The presentation layer of the design architecture is represented by the following classes.

GameProcess

This is the main interface between model and view. It is the presentation layer and being at the centre of the design architecture, the main purpose of this class is to have a bi-directional communication with the view and model layers. In doing so this class has three main responsibilities. Firstly to create all game entity objects these include the player, spider and centipede. Secondly this class is responsible for carrying out the game logic loops and algorithms which manipulate the data layer, an example is the logic of how the centipede progresses down the screen and behaviour when each segment collides with the edge of the screen or the mushrooms. Thirdly this class is responsible for updating game entity states and communicating these updated states with the view layer for displaying purposes, similarly the logic that ascertains whether any entity is alive and should still be rendered by the view.

4.3 Model Classes

The following classes constitute the model layer of the design architecture.

ImageHandler

Efficient image processing played a significant role in ensuring that the overall design of the code conforms to good coding principles and effective separating of concerns. Instead of explicitly having each game entity load and process its own corresponding image file, the purpose of this class was to isolate the loading and processing of game entity images. Therefore having an *ImageHandler* object within the implementation of the class representing a game entity, allowed the game entity to load and process its image file by simply passing a string constant representing the image file name to the constructor of the *ImageHandler* when it is declared.

Constants

The presence of magic numbers within code implementation violates good coding practices. Therefore the purpose of this class was to eliminate the use of magic numbers by providing all classes with standard game constants. Variables such as the window dimensions are declared here to allow the movable entities to use them to check the screen bounds and be bounded within this area while still a visible game object.

Class Hierarchy

The following model layer classes effectively implemented inheritance within their implementation. With the use of inheritance, commonalities amongst game entities are highlighted and these commonalities are used to create the entity hierarchical structure.

GameEntity

One commonality that all game entities have is that they all have image files which need to be loaded and processed. The purpose of this class was to load and process game entity image files with the use of the ImageHandler object defined within the class' interface. This class also provides access to the sprite of the game entity, game constants, game entity states and scaling characteristics. GameEntity serves as the main base class for all derived classes representing different game entities. 3 shows the structure of the hierachy.

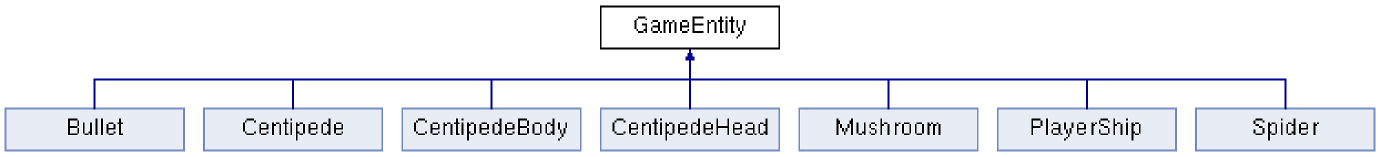

Figure. 3: Hierarchy of all game entities

Movable

This is an abstract base class modelling the movement functions of all movable game entities. The base class models the movement of all four compass cardinal points.There are three entities that qualify to complete the contract offered by this class, these are spider, player and centipede. The game models only the spider and player as the two movable entities. The shortfall of centipede is due to a design flaw implemented in earlier iteration and will be discussed in the critical analysis. Spider and Player override the function signature and implement their movement functions, and allows for the screen bound checks to occur.4 shows the structure.

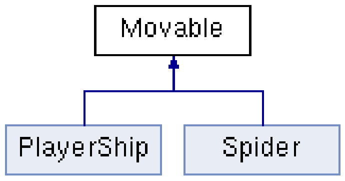

Figure. 4: Movable Hierachy.

Composition

Having modelled the the game entities as movable and stationary objects, the rest of the structure uses composition to communicate across classes.

CollissionHandler and CollisionDetector

The function of CollisionHandler is to resolve the conditions of collisions detected by the CollisionDetector. In this case CollisionHandler has an object of CollisionDetector which sends the message whether a collision has occurred or not. The summary of the conversation is such that if a collision has occurred the handler function uses that to resolve the case of the collision and knows which entities have collided.

PlayerShip and Bullet

A player has a bullet relationship is modelled here, a user input requests for a player to release the bullet. This request allows a player to call the bullet constructor and create a bullet object initialized at the current player position. The conversation in this relationship is for the player to invoke the bullet constructor when shoot is pressed, the object supplies the position to be initialized at as well. The bullet updates its position and the object to player is aware of the position because it has to send a delete message to the bullets vector with the bullet index.

Mushroom and MushroomField

A MushroomField is a collection of multiple mushrooms. The field is randomly generated every game by a random number generator assigning these values to the mushroom position. A check is done to avoid placing mushrooms in the same position, if this case is true the integers are regenerated. MushroomField has an object of type mushroom to access the responsibilities of mushroom. MushroomField object requests information on the mushrooms, for example if a mushroom is shoot its life is decreased. These are the object conversations occuring these messages allow for MushroomField to delete the mushroom from the field but the behaviours of a mushroom are modelled in its class, a case of separation of concerns having a class model a specific object responsibility.

Centipede, CentipedeBody and CentipedeHead

A centipede is a collection of both body and head segments. The segments follow the head in a train like fashion. The only difference between the head and body is the sprite, and collision behavior. The centipede has objects of head and body to allow for the creation of different GameEntities. The objects are of different types the centipede has two vectors to store the two object types. The object conversations are similar such that centipede requires to know the updated positions to move the vector object collections. The centipede also needs to be aware of the segment state to allow for deletion of objects.

5 Dynamic Behaviour and Game Logic

An object-orientated and dynamically well modelled design illustrates how objects behave and their collaborations. The sequence diagram depicted in figure 5 illustrates the dynamic behaviour and the game logic of the design. The object collaborations illustrate the implemented MVP architecture. The view classes have a bi-directional collaboration with the presenter classes and are isolated from the model classes. The presenter classes have dual collaboration such that they can bi-directionally collaborate with the view and model classes.

The object collaborations and game logic is initiated when the *Game* object receives user-input. The user-input is then communicated to the *GameProcess* object for further processing. Before any of the user-input is processed, the *Game* object checks with the *GameProcess* object if the current game state is GamePlay which indicates that the game has started. If so, the *Game* object initiates the game loop of the *GameProcess* object. The game loop processes the user-input and moves all the movable game entities accordingly by initiating conversations with these objects. A collaboration between model classes is illustrated by the conversation that the *Playership* object has with the *Bullet* object. If the user-input corresponds to the firing of bullets, the *GameProcess* object communicates this with the *Centipede* object. The *Playership* object then initiates a conversation with the the *Bullet* object telling it to move accordingly. This kind of model class collaboration is also observed with the *Enemy* and *Bullet* objects, where the collaboration is independent from user-input. Once all game entities have been moved the *GameProcess* object initiates a conversation with the *CollisionHandler* object, asking if any sprites have collided. If so the consequences of these collisions are internally handled by the *GameProcess* object. Thereafter the *GameProcess* object initiates a conversation with the *Centipede* object requesting an downward movement. The *GameProcess* object internally processes the state changes and communicates these updates with the *Game* object. These updated states are then processed by the *Game* object and the updated game entities are displayed to the user.

6 Unit Tests

Google unit test framework is a software used in this design project to test unit functionality of the software program code. The design procedures involve decisions on what can and cannot be tested. After careful planning, test cases are obtained and test names according to the functionality tested. Measure of success and progress is found from the number of tests meeting requirements. Measuring is the use of the test data obtained or simulated test data to verify the unit and the comparison of the units actual behavior with its required behavior as specified in the units requirements documentation [3].

Test driven development of software helps reduce defects in code in the process of development and it makes it less challenging to see when a code breaks and makes it easy to detect the violation introduced by added code to previous code that ran.

All moving objects were tested to an attainable degree. The use of inheritance allowed the move functions to be tested rather by each movable object that derives from the movables base class. It was made that logic rules being tested are met despite the object used or tested.

Resource Loading

Although necessary to test if all resources are being loaded correctly, time constraints could not allow. The resources testes for correct loading are the splash screen and the game background. A throw is thrown by the loading classes if there was an issue loading. The test checks if there is any such throw that has been thrown and passes if these were not thrown.

Mushroom

At the start of every game, random positions should be allocated to mushrooms. System time is used to seed the random numbers. To test for randomness a constructor that takes in arguments is implemented and a function that takes in the passed time argument initializes the field. With the above flexibility introduced, fake system times were used. Making two objects of mushroom means calling them at the same time and the seed cannot differ. The fake seeds are passed when calling a constructor in tests and randomness of the field initialization evaluated .This is a valid way since the system clock is forever running and can never have similar values. Randomness testing is essential to adhere to game mechanics.

Collision

Bounded box algorithm is used to detect collision of on screen objects. The separation of collision detector as a class and collision handling class narrowed the unit test for collision. All possible collisions between objects are tested and within the collision class function and the returned Boolean indicated the collision result to be true of false. The verification of successful functionality of the collision is done by setting two objects to the same position . These objects are then passed as arguments to

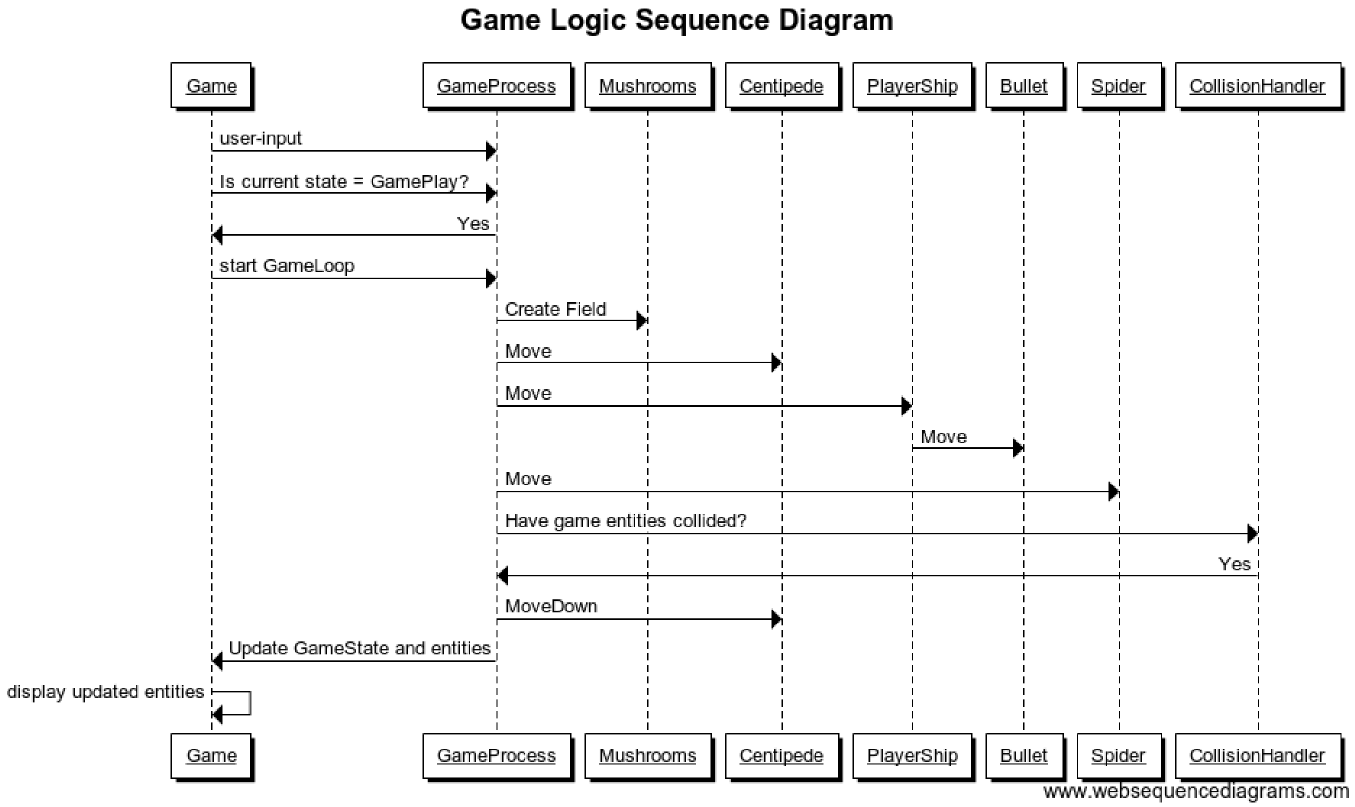

Figure. 5: Game logic sequence diagram

the collision detection function. The collision function checks eight conditions of possible collision and if one is true it reports a collision using its returned Boolean. Two objects with different positions are also used as a test case to ensure that the function does not flag objects that are not in the same area as collided. The need for any further tests that include collision is rather object specific depending on nature of objects colliding and functions invoked by a successful collision of two such objects. In essence the response to the nature of collision or the kind of objects that collide should be unit tested from collision handler class, but this was not done. Collision that involve the bullet could not be tested due to bullets being part of a player and implementation flaw on the responsible functions in bullet.

Spider movements

The spider moves vertically on the screen and it is crucial to test that it doesn't go beyond the window bounds hence done in this project. The spider is supposed to reach the play area, a test is performed for this argument. The procedure used was to set a the spider object position to the edge of the screen (up, down) and move it up or down and see if it passes the defined boundaries.

Player movements

The player movements use base class movables. The tests are separated into units consisting of move left, move right, move up, move down. The boundaries are also tested for left right bottom and the upper boundary that defines the play area.

It is not sufficient to test only movements across the screen, but to also see if the object bounds itself to the defined window size. Boundary tests are performed for the player and the main drive being to ensure that it meets the required bounding box for the play area. A player object is created and position set to the left bounder Player movements use base class movables. The tests are be separated into units that move left, move right, move up. A position known to be the bound for each angle and direction movable is assigned to player and object move function called in the direction being tested. The position is later checked if it has passed the bound or it has kept at it. Keeping at the bound or moving away dictates a success in the condition tests. General up, down, left and right of the player are tested by setting a position and calling the move direction function under test. If the move to specified direction works correctly the player position increases or decreases the coordinate and thus showing displacement along a direction. The player tests as described above are important in making sure that the right incremental movements are done and the player keeps within the boundary. This forces programmers to abide to the required game mechanics.

Score and life

The tests for score and life could not be implemented because bullet could not be easily accessed from the tests code. The test is important and has implications on the game process since the lives affect the game state of the objects, hence stated.

7 Critical Analysis

7.1 Design Flaws

Duplicated Code

The CentipedeBody and CentipedeHead classes duplicate the movement functions within their code. This is an unnecessary increase in code base and maintenance burden. This is the case because of a poor initial abstraction and trying to change this at a final stage proved to be time consuming and a technical debt had been incurred. The movement functions take in different shared pointer types of Head and Body respectively. In order to solve this a template type can be employed to cater for the different types and achieve a DRY principle by allowing each entity to re-use this code. Since the behaviour is the same a non-virtual function with a mandatory implementation can be used passing different arguments only.

Long Functions

The CollisionHandler class has long functions resolving complicated cases of bullet and segment collisions. The resemblance of arrow code is present. The functions can further be broken down to call other functions within the loops. As the functionality increases the collisionHandler cases increase in the same vain will the arrow code complexity making maintainability a challenge.

Monolithic Class

GameProcess is the interface between ’view’ and ’data’ but has other extended responsibilities which can be separated into another class such as an EventsHandler for the keyprocess decision making.

Refused Bequest

The movable class does not capture the whole movable objects spectrum. An attempt for bullet to use the class is problematic as bullet does not complete other movements. The centipede satisfies the abstraction but does not reuse the class. A solution would be to model each movement as an abstract base class since all movable entities can inherit those abstractions they can fulfill.

Separation of Concerns

In order to fully achieve logic separation from view, a coordinates class could have been implemented to allow for independence from SFML getPosition. The coordinates 2D struct can be used to keep track of entity positions and only set the positions at the interface of the view and data.

7.2 Functionality

The implemented solution meets the basic functionality with three minor features. The solution is capped at good, from the designer viewpoint this meets the success criteria defined in section 2.5. This is in light of the deadlines and deliverables required, so a time and functionality trade off is made. The game can be further improved to be more robust by adding the minor and major features beyond those added. The addition of a scoring system within the game makes it more goal oriented as the player can aim to get a high score. The addition of more enemy types as major features improves the game level, and becomes harder to navigate the game. The close mirroring of the entity behaviour to the real game with Centipede getting a gun upgrade after killing all segments.

7.3 Testing

Test driven programming is taken as the acceptable and advantageous way of developing code, hence the use of the method at the initial stage of the code. Unfortunately this method was later dropped and replaced with test after procedure as the complexity of the code increased and as the design started breaking down due to implementation of acceptable standards. Detailed discussion on performed tests can be seen in Section 6. The test cases are clearly defined and each case is modelled with enough assertions to verify just a functionality sufficiently. Doctest can identify a failed test case and the exact assertion failed with also the expected result thus this type of testing is used and found to be time saving. All to do with bullet is not tested because of the used implementation and design flaw, which made it hard to call the right functions in tests.

7.4 *Improvements*

The view class can be further separated and introducing a drawing class. The class responsibility would be to draw all objects on the screen as required by the presenter update. The advantage of this is a decoupling of the current Game Process to have a dedicated view layer with separated concerns. The maximum attainable result is an assessment of excellent therefore functionality of all major features can be implemented and mimic the Centipede game reality This is a more competitive and advanced solution that satisfies a required success criteria. On unit testing framework inclusion of the mocking framework helps test interacting object, and inclusion of GUI testing interaction to cover the presentation layer. The code should be refactored to allow easy testing of bullet functionality and those that involve it. [4].

8 Conclusion

The solution presented here meets the basic functionality and three minor features stipulations but can be further improved. The use of (OO) and composition is implemented to separate concerns within the code layout and introduce aspects of code modularity. The addition of a minor and three major features will increase the game quality and allow more realistic gameplay. An in depth overview of the main functions categorised in their layers is presented and further recommendations for improved maintainability and game logic are made. The overall view of the developed game is assessed as good while more enhancements can still be made by more adequately equipped developers provided enough time.

References

[1] S. Levitt, “Project 2018- centipede,” University of Witswatersrand, 08 October 2018.

[2] Y. Zhang and Y. Luo, “An architecture and implement model for model-view-presenter pattern,” in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 8, July 2010, pp. 532–536.

[3] L. Gren and V. Antinyan, “On the relation between unit testing and code quality,” in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2017, pp. 52–56.

[4] IEEE, “Ieee standard for software unit testing,” 1986.

Appendix A
DECLARATION

We hereby declare that this project titled Centipede Game Development submitted by S.Fadzi and S.Ndlovu on the 8th of October 2018, is a bonafide work, which was jointly undertaken with equal duties and responsibilities. Therefore any discretion credit given **must** be equally shared i.e 50 % each member.

Member 1
Simbarashe Fadzi 745118

Member 2
Sanele Ndlovu 716411