Team Paradroid

Snake PVP

Introduction 4

Requirements 5
- UI Requirements 5
- Map Requirements 5
- Game Logic Requirements 6
- Single Player Requirements 7
- MultiPlayer / Networking Requirements 8
- Shop Requirements 8
- Audio Requirements 9
- Non-Functional Requirements 10

Software Design 11
- Game Objects and Logic 12
  - Game objects 12
    - Snake 12
    - Map 12
    - Fruit and PowerUps 13
- Artificial Intelligence 13
- Map Builder 13
- Shop 13
- Audio Controller 13
- FXML 14
- Rendering 14
- MultiPlayer 14

Interface Design 16
- Interface Design Overview 17
- Main Menu 17
- Singleplayer / Multiplayer Selection Screens 18
- The Game 20
- The Shop 22
- Map Builder 24
- Audio 25

Software Engineering Processes and Methodology 26

Risk Analysis 29

Evaluation 31

Strengths 31
Weaknesses 31
Additional features 32
Summary 33
Teamwork 34
Individual Reflections 35
Daniel Batchford 35
Florian-Andrei Blanaru 36
Mohammed Jaber Alqasemi 37
Yuji Fukuta 38
Rahul Gheewala 39
Dilpreet Kang 40
Appendix 41
Software Principles and Coding Standards 41
Test Report 42
Introduction 42
Goals 42
Approach 42
Constraints 42
Testing Timeline 43
Testable Features 43
Pass / Fail criteria 43
Estimates 44
Responsibilities 44
Unit Testing 44
server.ai 44
server.game 46
Black Box Testing 49
Game Logic 49
Networking 51
UI 52
AI 55
Build A Map 55
Audio 56
Shop 57
User Testing 58
User 1 - (Non Gamer) 58
User 2 - (Gamer) 59
Testing Evaluation 60

Subsystem UML Diagram 61
Individual Contributions 67
Assets and References 68
Gitlab Link 68

Introduction

Our proposed game combines the iconic features of the classic Snake game with powerups, allowing for two players to battle it out to be the last snake standing.

Upon starting the game, the player is presented with the option to play against an AI or against another player. This provides both a multiplayer competitive mode and a single player mode against a programmed AI. In both cases the core of the game logic involves eating a piece of fruit to increase your own length and decrement your opponent's length. If either snake bumps into the other or into a wall they will respawn and decrease in length. Once the player's length reaches zero, they lose a life. A player wins once they have caused the other player to run out of lives.

The single player mode features an AI controlled opponent. The AI has three levels of difficulty: Easy, Medium, and Hard. This allows a player to select a difficulty based on their skill level.

The maps within the game are created by the developers, randomised, or created by the player. The pre-made maps include a wide variety of difficulties with maps such as 'The Great Divide', which provides little challenge, whereas maps such as 'Classic Pacman' offer a higher level of difficulty. Each map contains wall objects, randomly spawned food objects and power ups. The power-ups add more depth to the game allowing players to strategize the optimal way to reach the fruit before their opponent. Power-ups include: Freeze, Mine, Controls Inverter, Wall Skip, Boost, and a Coin. Additionally, every map has the ability to wrap around the edges onto the other side adding another mechanic the player can use to their advantage. The view of the map will be from a top-down perspective, similar to the classic snake and pacman games.

The game also includes a shop with various skins on offer for both snakes and maps. Players can earn money by winning games or receiving the Coin powerup. This money is added to the player's balance allowing them to purchase skins. We also included a map builder where players can build their own maps to play, in both single player and multiplayer modes.

The target demographic for the game is the general public, as it has a simple concept which is easy to grasp and become familiar with. Both Snake and Pacman are well known games which have been around for decades - we believe the vast majority of the public are aware of these games.

Requirements

These requirements were produced in week 2. We felt it was important to produce these requirements early, allowing our team to have a clear idea of the scope of the game, the essential features and the game logic. They were later appended as the game developed.

UI Requirements

1. The program **MUST** implement a main screen

1.1 The main screen **MUST** have an option to play in single player mode

1.2 The main screen **MUST** have an option to play in multiplayer mode

1.2.1 There **MUST** be an option for 2 players playing on a local network

1.3 The main screen **MUST** have an option to quit the game

1.4 The main screen **MUST** have ar

1.4.1 The settings **MUST** have a music volum

1.4.2 The settings **MUST** have a sound effects volume slider

1.4.3 The settings **MUST** have an option to mute all audio

1.5 The main screen Fonts **MUST** be at minimum 12pt

1.6 The main screen **SHOULD** have a minimum screen size of 1550px * 800px

1.7 The main screen **COULD** display a balance if the shop is implemented

1.8 The main screen **COULD** have a ‘Build A Map’ Option

1.9 The main screen **COULD** have a ‘Shop’ Screen

1.10 The main screen **COULD** have an option to enter and exit full screen

1.11 The main screen **COULD** have an option to choose a window resolution

Map Requirements

2 The game must include a map

2.1 The map **MUST** have 2 respawn points

2.1.1 The respawn point **MUST** have at least 2 empty spaces above the player

2.1.2 The player **SHOULD** always be facing up when respawning

2.1.3 The respawn point **COULD** be randomized point on the map

2.2 The map **MUST** spawn a fruit at a random coordinate when game starts

2.2.1 The spawn **MUST** not overlap a wall

2.2.2 When a player picks up a fruit, the fruit **MUST** spawn at another random coordinate

2.3 The map **MUST** have a randomly generated map feature

2.3.1 The inner walls **MUST** not occupy more than 12% of map space

2.4 The map **SHOULD** allow a player to enter through one side of the map's outer wall and reappear from the opposite side

2.5 The map **SHOULD** have a size of 30*30 cells

2.6 The game **COULD** have a map building feature

2.6.1 The map builder **SHOULD** have a reset button that removes all walls

2.6.2 The Map Builder **SHOULD** reflect the map using the selected custom map skin

2.6.3 The map builder **MUST** allow the user to set walls

2.6.4 The map builder **MUST** allow the user to remove walls

2.6.5 The custom Map **MUST** be saved locally.

2.6.6 The user **COULD** have an option to remove a map

2.6.7 The user **COULD** be able to rename map/provide a name if does not already exist

2.7 The map **COULD** spawn powerups

2.7.1 The powerups **MUST** randomly spawn at the start of a game

2.7.2 A powerup **MUST** be respawned on the map after a player uses it

2.7.2.1 The powerup **SHOULD** be a new random powerup

Game Logic Requirements

3 The game **MUST** include 2 snakes in each game mode

3.1 The snake size **MUST** be the same number of units as AI when game is initialized

3.1.1 The default size **SHOULD** be 5 units when game begins

3.1.1.1 Initially, when the player respawns, the player and AI **SHOULD** occupy 1 unit and increases by 1 every movement until the expected size is reached

3.1.2 The user **COULD** choose a preferred initial snake size

3.2 The game **MUST** follow a snake like logic

3.2.1 If a snake touches a wall, the snake **MUST** respawn

3.2.2 If a snake touches a wall, the snake **MUST** lose a life

3.2.3 if a player's size reaches 0, the player **MUST** respawn

3.2.4 if a player's size reaches 0, the snake **MUST** lose a life

3.2.5 if a player loses all lives, the game **MUST** end and the opponent **MUST** win

3.2.6 If a snake touches a fruit, the user's snake size **SHOULD** increment by 1

3.2.7 If a player's snake touches a fruit, the opponent's snake size **COULD** decrement by 1

3.2.8 If a snake touches a wall, their initial size **COULD** decrement by 1 unit

3.3 The player **MUST** have controls to dictate movement of snake

3.3.1 Pressing UP arrow key **MUST** make the snake go UP until further interrupt

3.3.2 Pressing DOWN arrow key **MUST** make the snake go DOWN until further interrupt

3.3.3 Pressing LEFT arrow key **MUST** make the snake go LEFT until further interrupt

3.3.4 Pressing RIGHT arrow key **MUST** make the snake go RIGHT until further interrupt

3.4 The game **MUST** grant each player 3 lives

Single Player Requirements

4 The game **MUST** include a single player mode.

4.1 This mode **MUST** include 3 game levels (EASY, MEDIUM, HARD)

4.1.1 The difficulty **SHOULD** determine AI intelligence

4.2 This game mode **MUST** use an AI bot

4.2.1 The AI **MUST** have 3 difficulty levels corresponding to the 3 game difficulties.

4.2.2 The AI **MUST** be sufficiently challenging for a player at each difficulty level.

4.2.3 The AI **MUST** not have access to information unavailable to the opponent, for fairness.

4.2. The AI **SHOULD** appear to behave in a “player-like” fashion

4.2.2.1 The AI **SHOULD** not move in a “jagged” path across the map

4.3 The AI **COULD** consider targeting powerups

4.3.1 The AI **COULD** be able to use these powerups

4.3.2 The AI **SHOULD** use these powerups in useful moments

4.3.3 The AI **SHOULD** decide between targeting a fruit or powerup

4.3 The player **COULD** have the option to play with custom map

4.3.1 All custom maps **SHOULD** be selectable in a drop down list

4.4 When a round finishes, the game **SHOULD** display a message saying if the player won or lost

4.4.1 The game **COULD** award coins depending on the number of fruits picked up

4.4.2 The game **COULD** have a coin multiplier depending on game difficulty

MultiPlayer / Networking Requirements

5 The game **MUST** include a multiplayer mode, over localhost

5.1The player **MUST** have an option to host a game

5.2The player **MUST** have the option to join a game

5.3The host **SHOULD** have the option to choose custom map

5.4The host **SHOULD** have the option to choose the number of lives to start with, for both players

5.5The game **COULD** allow players to use an in game chat messaging system

5.5.1 The chat system **COULD** block potentially offensive words

5.5.2 The chat system **COULD** prevent a user from spamming the chat

Shop Requirements

6 The game **COULD** include a shop feature.

6.1The game **MUST** have an in-game currency

6.2The game **MUST** display this balance to the user

6.3Players **MUST** be able to select a skin

6.4 The game **MUST** assign the default skin if no skin is selected

6.5 The game **MUST** warn the user if there are insufficient funds to buy a skin when attempted

6.6 Each skin **MUST** have a cost to buy

6.7 The shop **MUST** offer a variation of unlockable snake skins and map skins

Audio Requirements

7 The game **MUST** include audio

7.1 The audio engine **MUST** allow audio to be controlled

- 7.1.1 The audio **MUST** allow muting of audio
- 7.1.2 The audio **SHOULD** allow the volume of audio to be changed
- 7.1.3 The audio **COULD** allow playback of music to be paused and resumed
- 7.1.4 The audio **COULD** be able to load different sound packs during runtime

7.2 The game **MUST** have sound effects

- 7.1.1 The game **SHOULD** have a sound for: Collecting a fruit.
- 7.1.2 The game **SHOULD** have a sound for: Collecting a coin
- 7.1.3 The game **SHOULD** have a sound for: Collecting a powerup
- 7.1.4 The game **SHOULD** have a sound for: Using a powerup
- 7.1.5 The game **SHOULD** have a sound for: Crashing into a wall
- 7.1.6 The game **SHOULD** have a sound for: Losing a life
- 7.1.7 The game **SHOULD** have a sound for: Game start
- 7.1.8 The game **SHOULD** have a sound for: Clicking

7.3 The game **SHOULD** have background music

- 7.2.1 The game **MUST** have a sound for: Menu music
- 7.2.2 The game **MUST** have a sound for: Game music

Non-Functional Requirements

8.1 The user interface **MUST** run smoothly

- 8.1.1 The user interface **MUST** load all assets quickly.
- 8.1.2 The user interface **SHOULD** be easy to use and understand
- 8.1.3 The game **SHOULD** follow a consistent UI theme throughout all menus
- 8.1.4 The game **SHOULD** reach the Main Menu within 5 seconds of launching.
- 8.1.5 Each sub-menu **SHOULD** load within 1 second of clicking its button.

8.2 Image files **MUST** load in a reasonable amount of time

- 8.2.1 Image files **MUST** have a sufficiently high resolution as to not appear blurry.
- 8.2.2 Image files **MUST** not contain copyright infringed material
- 8.2.3 Snake Skin images **MUST** be at least 100 by 100 pixels in size.
- 8.2.4 Map Skin images **MUST** be at least 100 by 100 pixels in size.
- 8.2.5 Shop images **SHOULD** load within 100ms selecting a skin.

8.3 The game state **MUST** be consistent over both clients in multiplayer

- 8.3.1 One player **SHOULD** not have a significant advantage due to networking in multiplayer

8.4 The AI **MUST** behave in a human like fashion

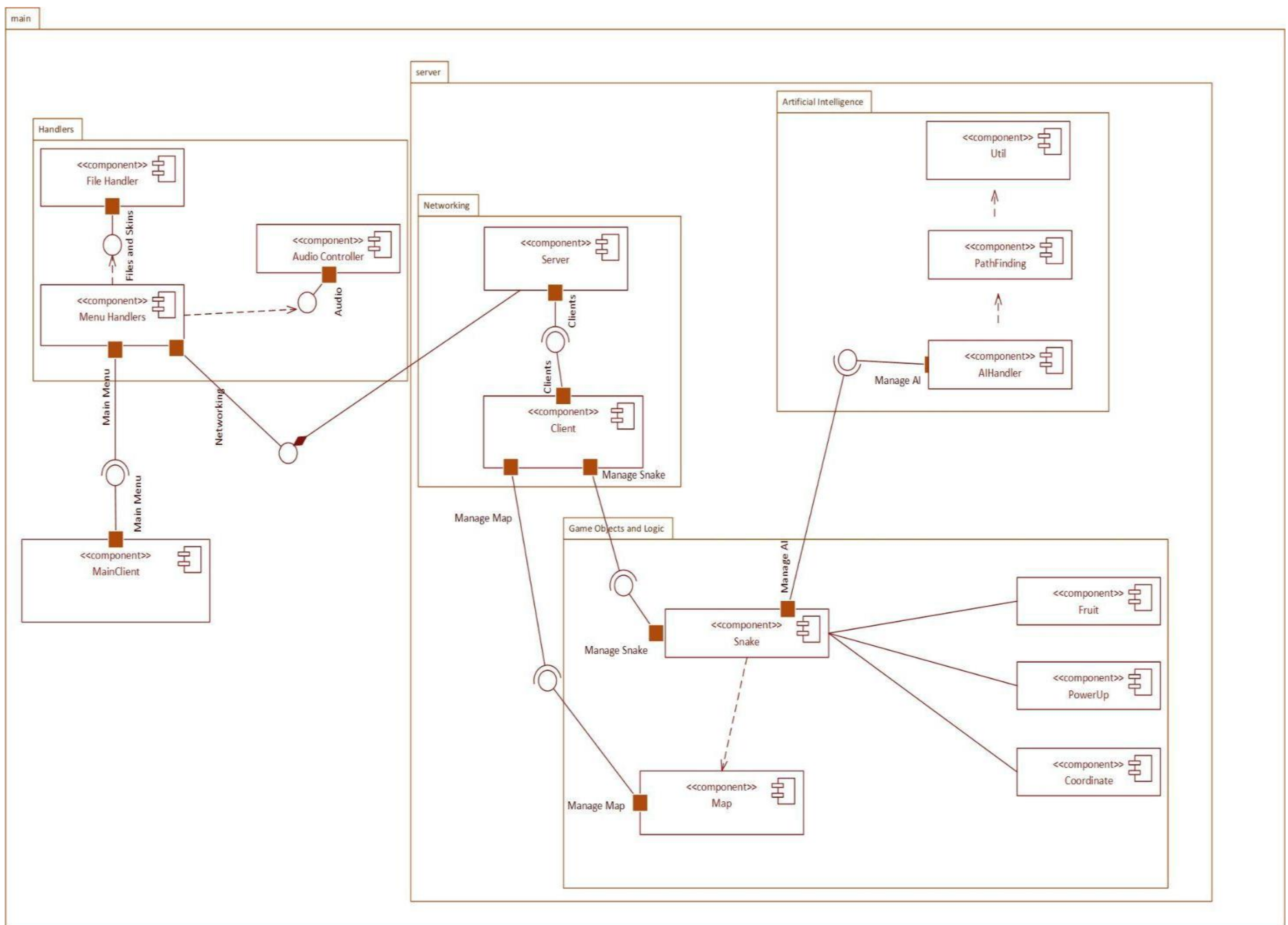
- 8.4.1 The AI **MUST** not have access to information the player does not have

8.5 The game **MUST** be implemented in Java

- 8.5.1 The game **SHOULD** be easy to launch to a user with no programming experience
- 8.5.2 The game **SHOULD** should not be built for a single operating system

Software Design

Figure 1: Component Diagram



The component diagram (Figure 1) shows the main classes of the game and their relationships while the class diagrams (Figure 11-17 in the Appendix) give insight into how these classes communicate with others in their packages.

The Main Menu is called from the MainClient class once it is started. The MainClient class contains the main() method called by the JVM on program execution. As the player chooses one of the Menu options (Audio Settings, SinglePlayer, MultiPlayer, Shop, Build A Map or Quit) they are directed to the selected scene and are provided with their respective features. Each feature has classes grouped into packages which handle the appropriate logic for the scene (e.g: SinglePlayer uses the game and ai packages, Audio uses the AudioSettings class). We

have chosen this approach as we believe it makes it easier for us as developers and anyone that wishes to read the code, to understand. It also provides us an effective way of connecting the various components of the game.

In addition to this, we have also created custom Exception classes to handle cases where the code is unable to run properly (NoPathFoundException, MapImageLoadException).

Interfaces were also used to help increase the modularity of our software design, as well as for the purpose of declaring constant variables, such as resource path constants, game objects constants or AI constants and methods.

Game Objects and Logic

The Game logic and objects of the game reside in the ‘game’ package, which is then divided in ‘managers’, where the Snake, Map and PowerUp classes are, and ‘usables’, which holds the Coordinate class and the Direction enum, which are necessary for the creation and usage of the classes from the ‘managers’ package.

The controls of the playerSnake are handled in the start() method of the SinglePlayer class, where the player can choose the direction they want to go, or use a collected PowerUp.

Game objects

Snake

The main focus of our game, the Snake class has two subclasses: the PlayerSnake class, which is used for both the singleplayer and multiplayer modes, and the AISnake class, which overrides the updateSnake() method in order to use the AI component of the game.

The SnakeSkin and SnakeSkinManager classes are used to load and assign skins to the snake object (from the Shop feature). Here is also where the collisions are handled (fruit, powerups, walls or snakes).

Map

Similar to the Snake class, the map class uses MapSkin and MapSkinManager to load and assign skins, as well as a MapManager class where all the custom maps are loaded. This class provides the drawMap() method which handles the rendering of the map and power ups..

A map object can contain randomly generated walls using the generateRandomWalls() method. The maps are saved and loaded in a .txt format, by using various characters for every object, which are then translated into a BoxStatus enum (e.g. BoxStatus.WALL, BoxStatus.EMPTY).

Fruit and PowerUps

The Fruit and the PowerUps class serve as the means of winning the game. Two PowerUp objects spawn randomly and as they are collected, the player is presented with the option to use them whenever they choose to do so. The other classes in the PowerUpsManager package are used for the execution of said PowerUps, and therefore do not extend the PowerUp class.

Artificial Intelligence

The AI is divided in two packages: 'pathfinding' and 'util' which are used together for the functionality of the AISnake class. The pathfinding package provides the means for the AI to find paths on a grid with blocked squares. This class allows a snake to navigate to a target square, which is selected based on the difficulty. In easy and medium modes, this target square is the Fruit on the map. In hard mode, this target square is either a fruit or a square in front of the enemy (allowing the ai's snake to "cut off" the player's snake). A Node class is used to represent a node, with f, g and h values, a parent node and the node's walkable status stored to allow A* pathfinding. A NodeGrid class is used to store a graph of nodes. The pathfinding code is called exclusively from the AIHandler class.

This feature is only available in the SinglePlayer mode and, by using the Difficulty enum, the player can choose how difficult they want the AI to be (Easy, Medium or Hard).

Map Builder

Another feature of the game is the Map Builder, where the player can build maps for later use by clicking on the tiles they wish the walls to be on. These maps are then saved in the same .txt format as those created by the developers. The whole process is handled in the BuildAMap class, part of the 'handlers' package. This class contains methods to update UI based on the status of the map building process. It also handles writing the map to a text file based on the current map shown to the user by the UI.

Shop

Each Snake and Map object has a SkinManager and Skin class. The Shop class provides the option to buy and equip new sets of skins using the in-game currency that is provided. Both the data that saves information on whether the player has acquired a skin and the amount of money they have is stored in a .txt file using the FileHandler class. Players can gain money by picking up Coin or Fruit objects.

Audio Controller

AudioController is the class that handles the background music (through the startMusic() method) and the sound effects of the game (deathSound(), collectSound() etc.). These can all

be muted or have their volume changed from the AudioSettings class. This has been done by using JavaFX MediaPlayer. This class contains only static fields and methods, allowing the controller to be called in all appropriate places in the program without passing around object references. Having a static approach here is sufficient as only one instance of an effects controller and music controller is needed when handling game audio. Using static fields and methods here allows easier integration of sound effects into the correct classes.

FXML

FXML has been used for the User Interface. Using FXML allows easy alignment and design of elements within the UI. FXML contains containers akin to HTML, allowing vertical and horizontal alignment within groups of UI elements. Furthermore, by using the onAction parameter, we were able to connect menu elements between each other and create a smooth and good-looking interface.

Rendering

The JavaFX library has also been used for rendering, mainly the GraphicsContext class. The drawMap() and renderSnake() methods are called on a separate thread to the game logic, to aid with performance. The render thread calls these methods every 200 milliseconds, which is a good balance between rendering the same game state too many times and not rendering a new game state quickly enough.

MultiPlayer

For the multiplayer aspect of our game, we chose to use a client-server model. The server is responsible for maintaining the game state which includes matters such as the locations of each snake, the map being used and any collisions that may occur. The clients are responsible for receiving data from the server, updating their own game states, and sending any changes back to the server. A huge advantage of this model is that there will be only one game state stored on the server, so the state for the clients will always be consistent with the state of the server. This will therefore provide a reliable experience for both players, which we believe is important. If each client had different game states, it would result in an unfair advantage for one player.

Additionally, this architecture means that the resources required are centralised on the server, and therefore when a bug occurs within the networking it is a lot easier to find where it occurred. In the future if we were to decide to add mechanisms to detect cheating, we believed this model would provide us with the best foundation to do so. This is because all user requests to change the game state are sent through the server and therefore code can be added to the server to only allow legal changes. With architectures such as Peer-to-Peer this would be much more difficult to implement as there is no server which can act as an overall authority.

However, we recognized a major disadvantage of this architecture, which is that there is a single point of failure. The entire functionality of the multiplayer depends upon the server working

stably - therefore if it were to go down or act unreliably it would result in the whole multiplayer mode being dysfunctional. An alternative would be to use the Peer-to-Peer model to avoid this. However, with this model if one machine is slow it slows down the other client which we believed would be a greater disadvantage.

Interface Design

Initially, before starting the UI design process, we decided to research the layout of several games and assess pros and cons of each. We found a common theme in games with too many menu options confusing and unattractive. To avoid this, we decided to minimise the number of options, leaving only the essentials, to create a clean UI and create a clean UI for the user. We decided to include the menu options SinglePlayer, Multiplayer, Shop, Build a Map, Audio and Quit on the main screen. Each of these buttons leads to another UI screen, where it is easy to return to the main screen. This avoided the inconvenience of the user having to restart the game to access a particular page, an error which we had foreseen in other games.

In order to maximise productivity, we decided to take full advantage of the program 'SceneBuilder' which uses fxml files to create scenes. This served as a massive advantage as it readily contained containers such as VBoxes and StackPanes to more easily align JavaFX elements. We were also able to preview the UI without running the game. This was convenient as we didn't have to run the game between UI adjustments. It also allowed us to explore different containers to see if one gave a better UI for the user than another. For example, although buttons were useful and used throughout the UI, we found the use of sliders and choice boxes convenient for some actions, such as for adjusting the volume and choosing a map. Labels, text fields and checkboxes were also used throughout the UI where appropriate.

Though SceneBuilder provided us with a lot of benefits, it had a learning curve. Firstly, we had not been taught about this in our course, so we had to follow several youtube videos to understand the basics. Through videos, we learned how to link FXML documents to a JavaFX project, the use of controller classes which contain the methods used by buttons and their respective annotations. We ran into a major disadvantage during integration of the UI with other team members. After our best efforts and plenty of research, we were disappointed to discover that overlaying the game canvas over the fxml we had created wasn't feasible. Therefore, we solved this issue by manually laying out the actual game canvas and supporting UI inside of Java source code. To give a seamless feel to the game such that the java files and fxml files are interconnected, we emulated the previous pages by using the same buttons, styling and background. However, we were not able to reap the rewards of SceneBuilder with this methodology. However, it did let us appreciate how to work with JavaFX elements manually and without the help of FXML and SceneBuilder.

Although we were aware cosmetics were not too important in collecting marks for the assessment, we felt it was necessary to create a clean looking design. We were able to experiment with several different techniques, but for us it was experimenting with css which we feel brought our game to life. Through the use of css, we were able to create buttons which provided an indication as to which option was highlighted. This meant the user could have full confidence that they had chosen the correct option and was unlikely to choose an incorrect option. We also decided to add several of our own designs to again add a personalised feel to the game.

Interface Design Overview

As we agreed to make the game menu simple and convenient, we created six main screens. We did not create a user guide or tutorial for this game when it is launched, instead opting for a more minimalistic design.

Main Menu

Figure 2: Main Menu Screen

The main menu screen initially starts with an introduction video. This video was carefully produced such that the final video frame matches the main menu layout, allowing a seamless blend into the ui once the intro video ends. The user is then displayed the main screen. This screen contains buttons to navigate to different screens within the game, as well as the current balance of the user. A graphic is used for the background, which is continued throughout the other game screens.

Singleplayer / Multiplayer Selection Screens

Figure 3: Singleplayer Selection Screen

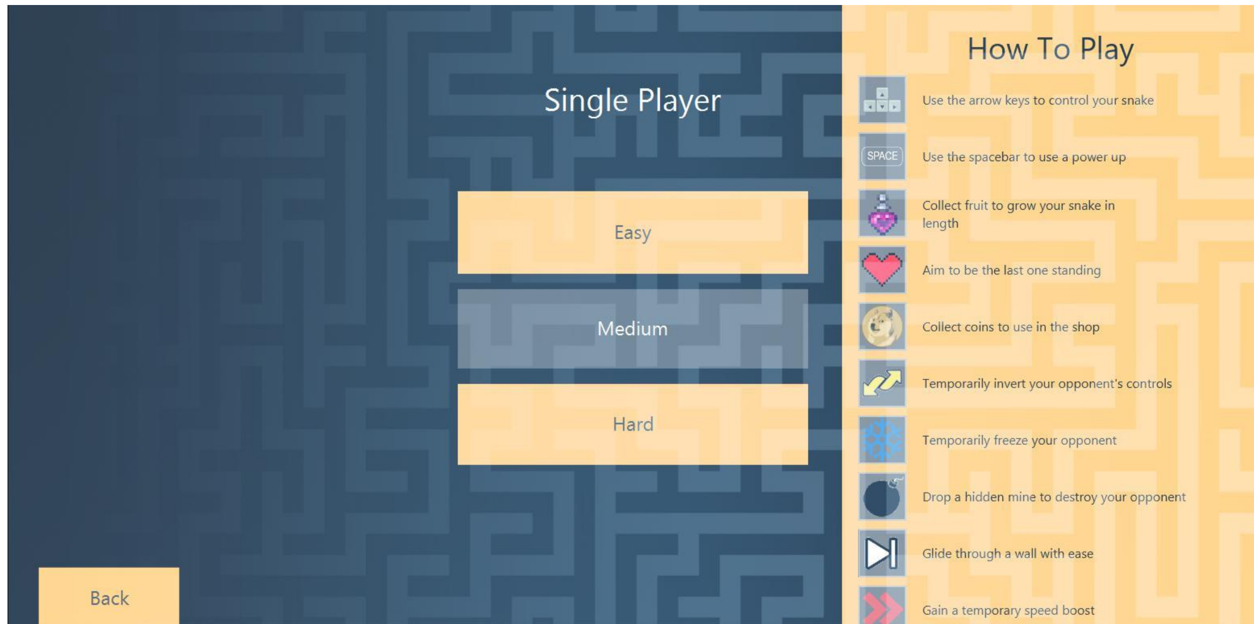
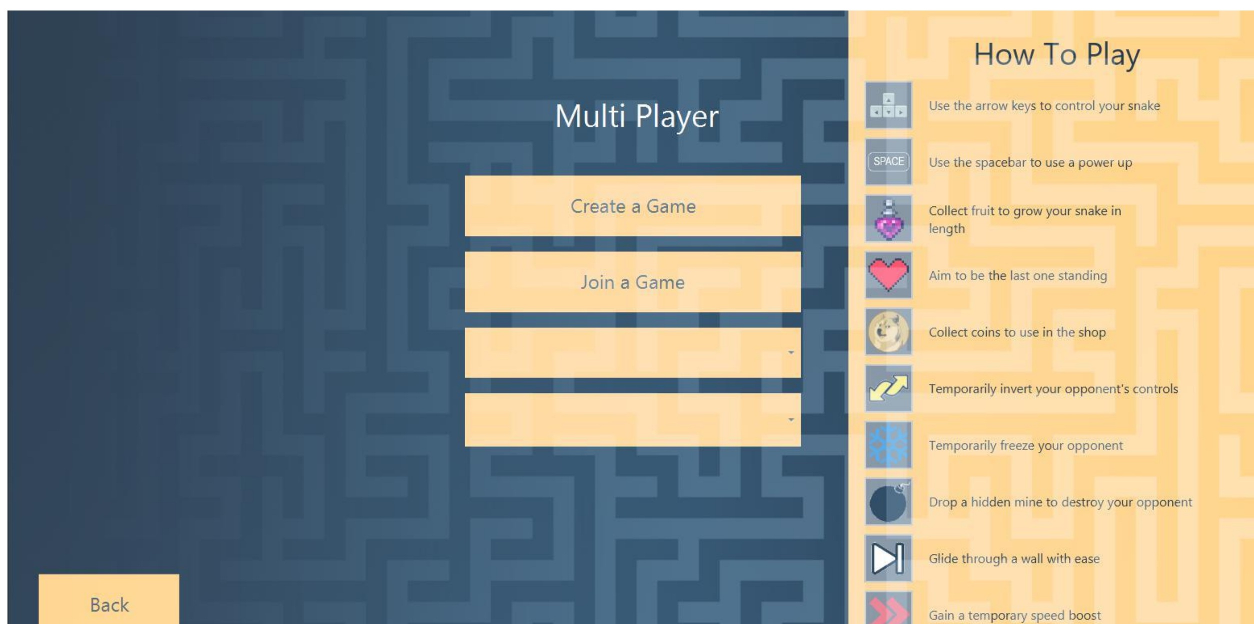


Figure 4: Multiplayer Selection Screen



For this feature, we created three different modes for single player. The mode difficulty dictates the intelligence of the AI and players can choose their own difficulty before starting a game. Originally, we did not have instructions for the player but we later added this, based on user

feedback. We then implement the three difficulty buttons as well as a choice box which lets you load a custom map (either developer made or made in the map builder.) If a map is not selected then the map will be randomly generated. Random maps initially had a lot of bugs such as the wall being next to or over a respawn point.

For the multiplayer game screen, we used the base game selection screen created for the single player mode, with the option to create a game, host a game and choose an initial number of lives, between 1 to 5.

The Game

Figure 5: Singleplayer Game Screen (Default snake skin, 3rd map skin)

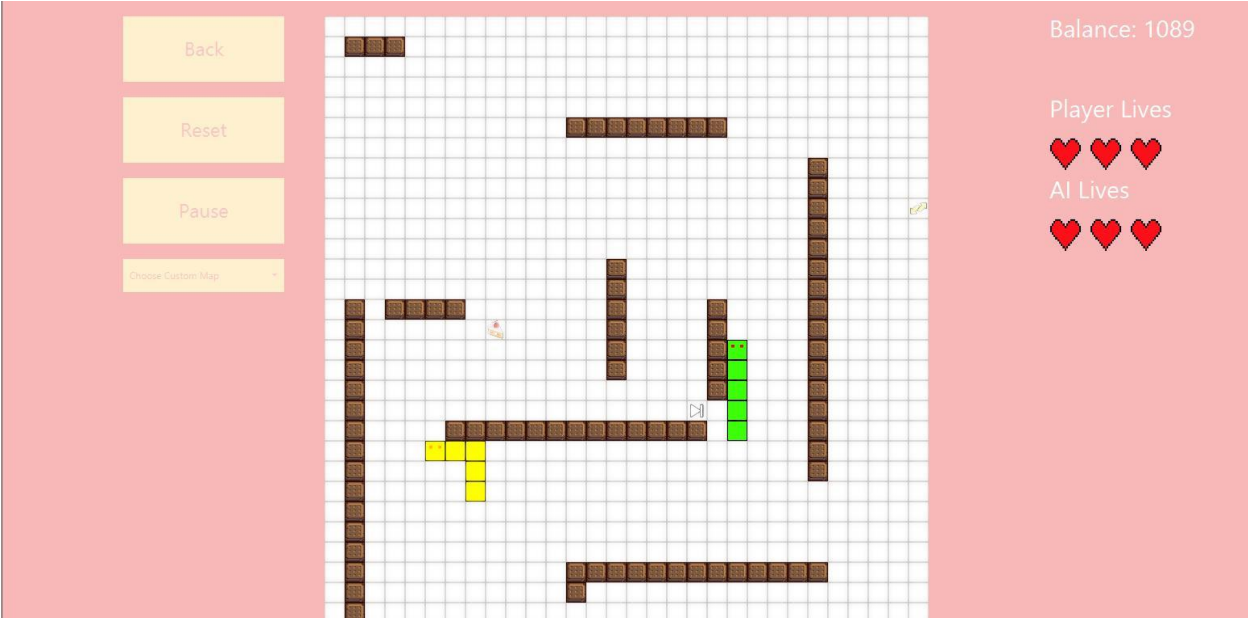


Figure 6: Singleplayer Game Screen (Snake skin 2, map skin 2)

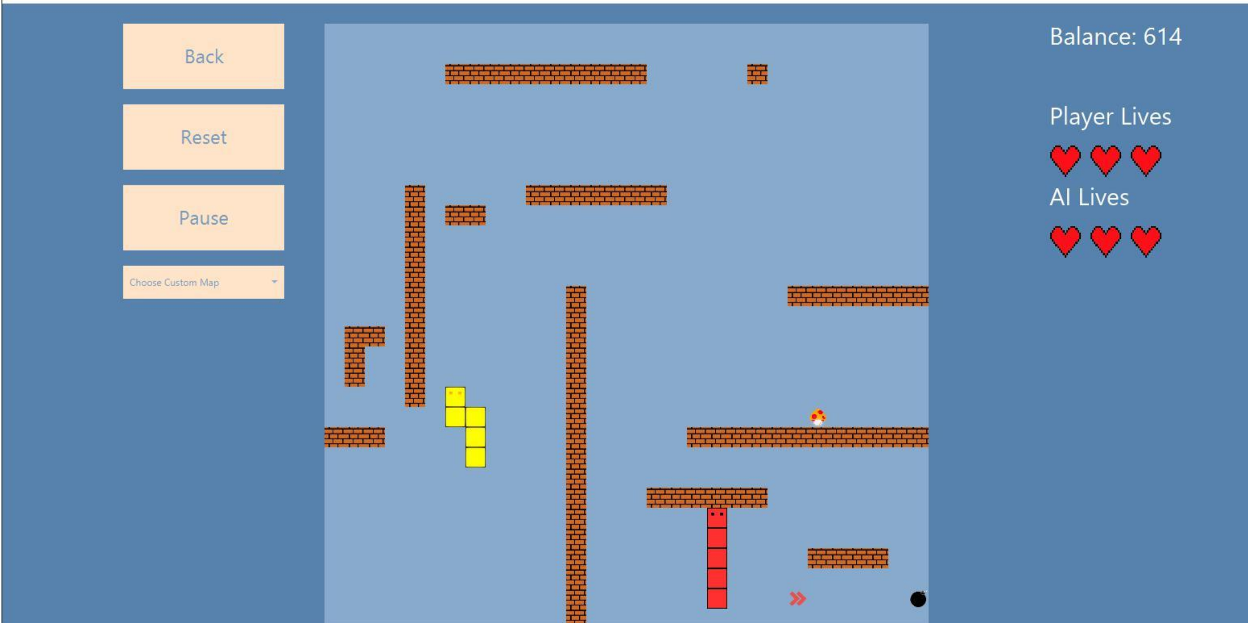
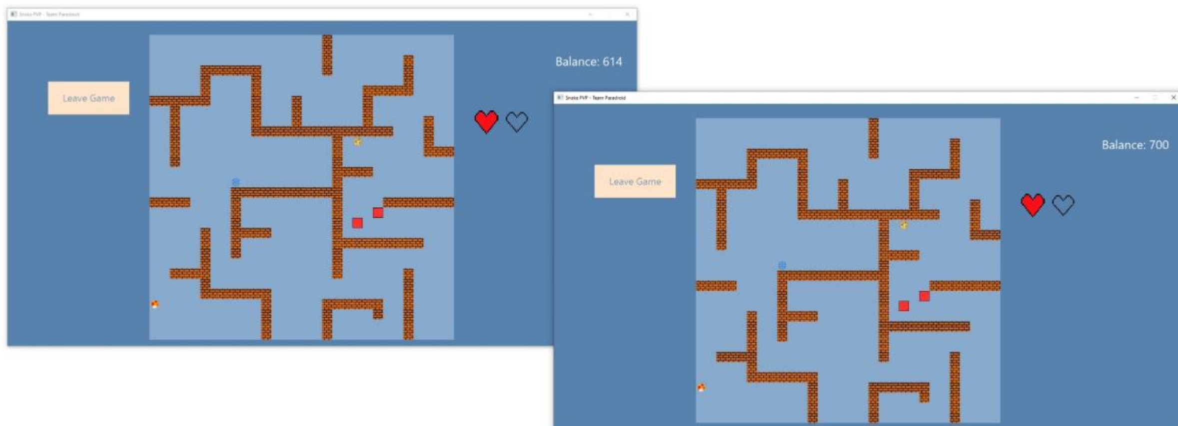


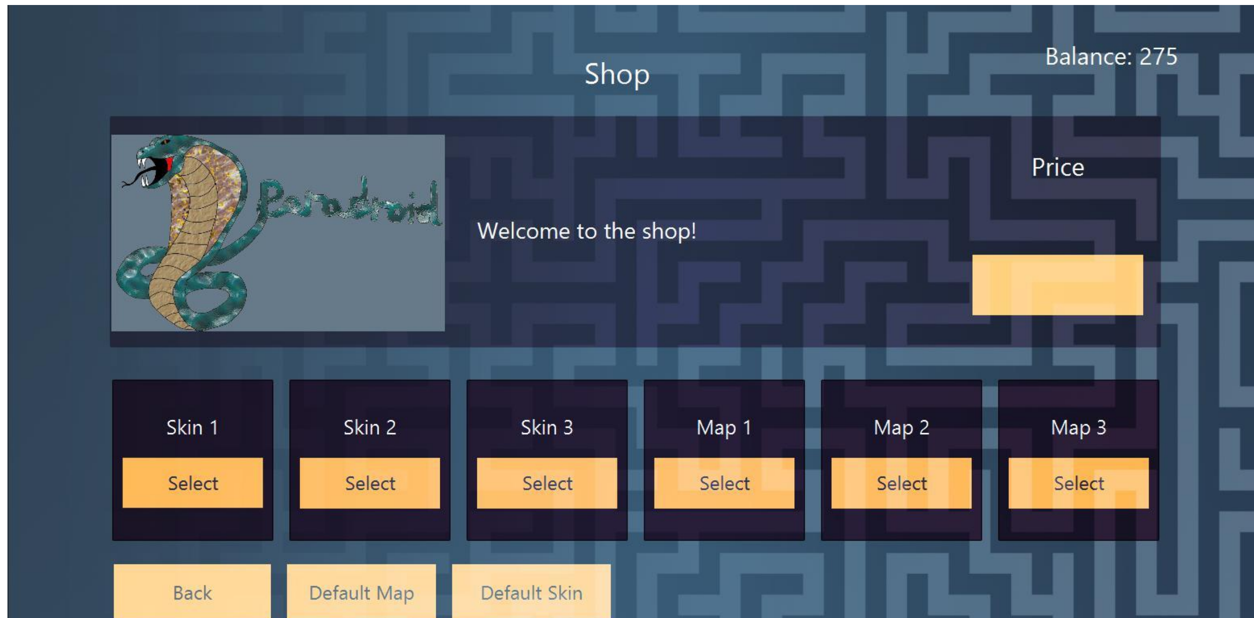
Figure 7: Multiplayer Game Screens



The game screen features a canvas centered in the window, which displays the game. We display the balance in the upper right corner at all times, to allow a player to see how their balance is increasing during the game. We also show the player's lives in both singleplayer and multiplayer, and the AI's lives in singleplayer. This indicates the game's current progression and allows the player to see if they are winning or losing. Alongside these elements, in singleplayer, we included a few controls - namely "back", "pause", "reset" and "choose custom map". These options are self descriptive. "Choose custom map" allows a player to load a custom map, either developer made or player made. During testing during development, we found that the game started too soon once a difficulty was selected. We therefore decided to add a 3 second countdown timer before a game starts. Additionally, the UI's background colour is chosen based on the currently selected map skin, allowing the game canvas to blend into the rest of the UI in an aesthetically pleasing fashion.

The Shop

Figure 8: Shop Screen



From a very early stage of development, we decided that we had a couple of options of measuring a player's progress. We considered a scoreboard system or an in-game currency. It was only after careful consideration that we felt that a currency would be more appropriate to our game model. We decided to take this route because we saw an excellent opportunity to add the option for customisable skins and maps, which wouldn't add too much extra work for us but would add significant value to our game.

A player can acquire this currency by playing in either the Easy, Medium or Hard mode where eating the item provides 10, 25, or 50 respectively to the player's balance. We felt this was optimum, because a player who has just begun playing can slowly raise their balance by taking advantage of the Easy game mode. However, this has a time cost as each fruit only rewards 10. Alternatively, a player can play in the hard mode where they have the chance of collecting fruits valued at a much higher value of 50, albeit in a much more difficult setting, as in this game mode the AI snake travels the optimum path making it significantly harder for the user. On the whole, players can expect to play approximately the same amount of time but challenge themselves at their leisure. By accumulating the in-game currency, a player can enter the shop where they have the option to purchase a selection of items, including both snake skins and map skins. These skins are each accompanied with their own description.
If a player has sufficient funds, they are able to purchase an item and this is reflected through a reduction on the current balance and a unique message telling the user that an item has been purchased. If a player does not have sufficient funds but attempts to buy an item, a message is delivered saying there are insufficient funds. At any given moment, a player is allowed to choose 1 skin and 1 map, and these can be equipped in any combination. If the player wants to revert

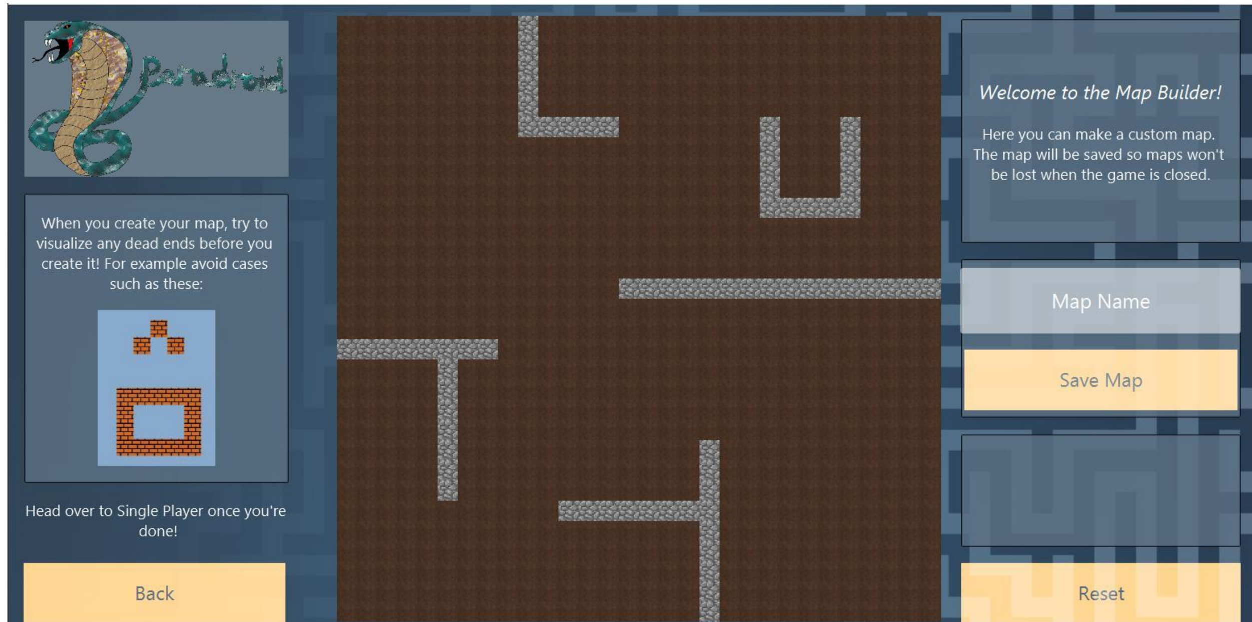
to the default skin/map, this can easily be done by clicking on the default options, free of cost. A player is always given the option to reequip a skin bought in the past if they do not wish to use the most recently purchased. Due to the limited variety of skins, we did not feel it was appropriate to add a refund button when the player could have reclaimed some funds spent for a skin.

When we began programming the Shop, we met a few hurdles. The main one being, a player could play for hours on end and upon closing the game, their in game balance is reset. We understood this would be a major inconvenience and demotivate players to want to retain their progress. After all, the game model relies heavily on incentivizing players to eventually collect all the items available. To solve this issue, we decided to save the user's progress using a text file saved in the game's directory. This file contains encodings such as FFTFT where each character represents whether a player has bought an item or not. If a player decides to buy or equip an item, it is compared with the current balance and whether the item is already owned - the appropriate action is then taken.

Looking back at the Shop after completion, it appears to be a major success. The shop is fully functional and easy to use. Many games use an in game currency to attract players and create a form of loyalty to the game, in order to hold onto players. We feel we have achieved this through making the items hard to earn, but definitely achievable.

Map Builder

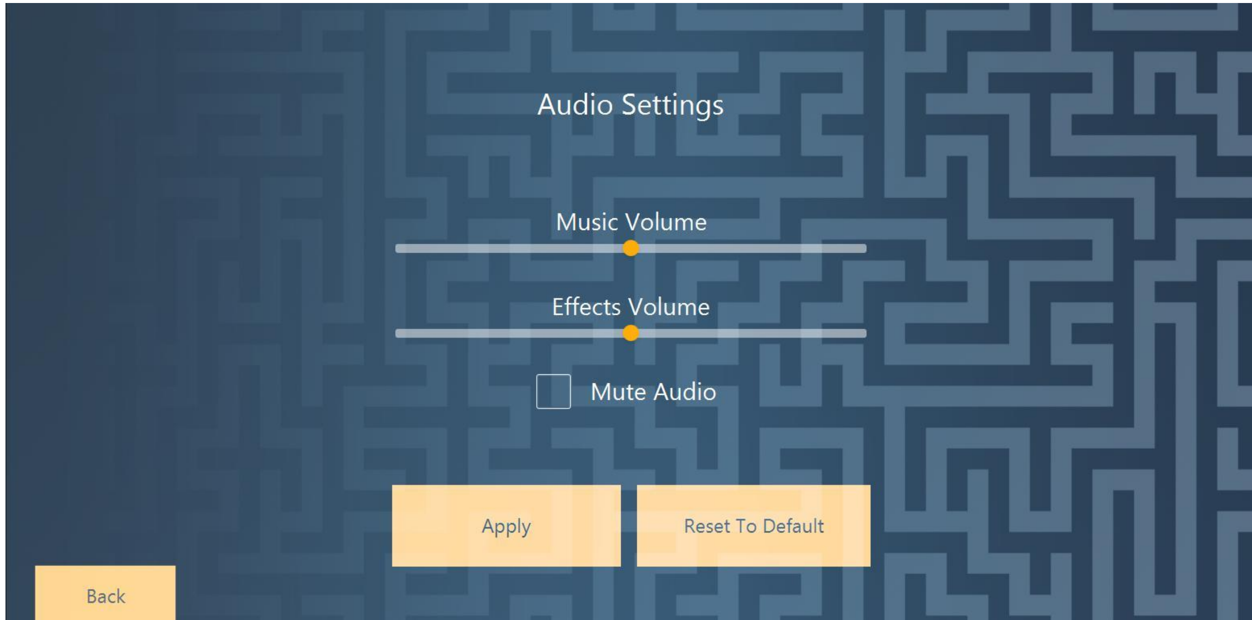
Figure 9: Map Builder Screen



Having completed the Shop, to continue challenging ourselves and to again improve user experience through customisation, we decided to implement a map building feature. Fortunately, the way the maps were written made the process a bit easier. Maps are written using .txt files, 30*30 where a '0' represents empty space and a '/' corresponds to a block. Therefore, we devised a GUI where clicking on the square of a gridpane represented a block or space. Originally, we had encoded each block to a 'G' or 'W' where it represented ground and wall, but eventually changed it to images of the skin currently equipped in the shop. This meant the user could have a visual idea of what the map could look like in advance and tweak any changes to their preference, before saving. We then implemented a choice box which contains the list of all the custom made maps, as well as developer made maps. Unfortunately, due to time constraints, we were unable to add a few extra features such as a check to see if a map has been made where a fruit would be surrounded by walls, as well as maps containing dead ends. To partially solve this issue, we provided a warning to aim not to produce maps which contain dead ends. Additionally, due to time constraints, we were unable to add an option to allow users to edit previously produced maps, or to delete a custom map.

Audio

Figure 10: Audio Settings Screen



Our audio interface is fairly easy to understand. We set up two types of audio: music and effects. We set the volume size for each of these sounds independently. In addition, knowing that the shop feature uses text files to save currency values allowed us to easily save the music values in the same data file. This allows game volumes to be saved after the game is quit.

We also linked a slide bar with the volume size. When the slide bar is moved, the volume will change immediately to the current value on the slide bar. Effect volume can also be checked when the apply button is pressed. We also added a mute feature which mutes both the music and effect volumes.

Software Engineering Processes and Methodology

Our team followed a waterfall development method. Initially, requirements were drafted in week 3. Then, our design was laid out. Our team produced the folders and packages with empty classes. This allowed us to see which classes should extend others and where interfaces and resources should be placed. Our layout allowed us to understand how the client and it's rendering should interact with the game logic and ai. Once this design was finalised, we implemented the required code, using the initial requirements as a guide. Verification was then performed through test classes, with both user testing for logic based classes and testing with end users for ui and audio choices.

Initially, we set rough internal deadlines for the development process, using an Excel spreadsheet. Deadlines could be marked off as they were achieved. We found this document very useful as it allowed us to visualise our development process and assign team members to specific tasks. These deadlines included deadlines for code integration (Week 3), testing (Week 8) as well as deadlines for compiling the final report and final presentation.

Our team met weekly on Mondays and with the TA on Thursdays or Fridays. Our Monday meetings allowed everyone to share their progress and thoughts ahead of the rest of the week. We then divided up items that needed to be focused on between team members. Having meetings in this fashion gave each team member specific tasks that should be completed by the end of the week. We found these meetings very valuable and continued to meet weekly throughout the entire project.

Our team used Git for version control and collaboration. This allowed synchronization of the code base. Without this, we would have struggled with version control and ensuring every team member had the correct / latest version of the project. Furthermore, we used git branches to implement features independently. These branches allowed us to work on a specific version of the code while it was unfinished or buggy. Each team member either had their own branch or shared it with another team member. When a feature was completed on a branch, this was merged with the *dev* branch. Once *dev* compiled into a working version, this was merged with the *master* branch.

The specific branches used are detailed below:

- master
- dev
- networking
- ui
- ai
- mechanics
- demo
  (Note that the *demo* branch was used temporarily for the week 7 game demonstration)

Some team members worked in pairs. Rahul and Yuji worked together on the UI, while Andrei and Mohammed worked together on the game logic. Programming in pairs allowed us to develop these features faster than otherwise, since errors in code from one member could be detected from the other member more easily. We also assigned members to focus on different areas of the game, as these could be implemented without code collision from other members.

These assignments were as follows:

- Daniel - AI
- Dilpreet - Networking
- Mohammed & Andrei - Game Logic
- Rahul & Yuji - UI & Audio

Once team members had finished with their areas of the game, they worked on other parts of the code base. Towards the end of the development process, we all worked collectively on the entire code base, using the weekly meetings to decide which features / bugs each member should work on. For example, Daniel and Yuji completed the AI and Audio sections and then went on to work on the singleplayer mode, file handling and ui features.

Our team used JUnit for testing. Test classes were used to test individual classes in the codebase. This allowed changes to be tested before committing code, allowing bugs to be detected before committing code to git. We achieved a high level of test coverage, giving us confidence that testing was covering a sufficient amount of each class.

Logging was used throughout the project. Initially, we attempted to use log4j but failed to get it working. Therefore, Daniel made a custom logging class, which formats errors / debug messages in a similar way to the console. Logging helped us see where “rogue” print lines were in the code, by displaying file, class and method names alongside the logged message. This helped us debug our code much quicker than otherwise.

While we considered other game engines such as libgdx, we decided on JavaFX as our game engine for a number of reasons. Our team was confident in using JavaFX given previous experience of the engine in 1st year. Moreover, JavaFX makes implementing a UI fairly straight forward while also allowing flexibility in our UI design. JavaFX FXML allowed us to design scenes to our liking with the use of containers such as VBoxes and StackPanes. Using FXML allowed us to keep UI layouts contained in their own layout files, preventing most styling options from being controlled by the Java code itself. JavaFX also allows css styling, which was useful when applying slider and button styles across the whole UI without repetition. For example, the UI button style used css to modify the button blending mode to screen over the background - this would otherwise have to be repeated for each button in the UI.

Weekly meetings with our TA allowed us to gain valuable feedback about our progress, changes we need to make and deadlines to focus on. We also used these meetings to ask questions we

had about the project. We then made it a priority to focus on the feedback given to ensure our project was as closely aligned to the brief as possible.

We could have improved our methodology by being more careful of merge conflicts within git. Often, these conflicts led to lost time during our development. This could have been avoided by discussing more actively on specific files that were being worked on. Furthermore, mass renaming files often led to conflicts - this could have been agreed on in advance to prevent overwriting of another member's work. We also could have utilised feedback from end users such as friends or family at an earlier stage in the process, which would have allowed us to develop based on this feedback. Game behaviour and the UI could have benefited from end user feedback and this could have been acted on accordingly.

Risk Analysis

Throughout the implementation of this project there were several occasions where we encountered risks. However, we were able to overcome these potential problems through careful planning and revising our existing strategies.

The first risk we encountered was that a majority of the team were unfamiliar with how to use Git when working in a team. This posed a significant risk as it would be very difficult to produce the project without the correct use of a version control system. Luckily, our team member Daniel was confident using Git and had no problem helping others learn how to commit their work to the repository. Furthermore, everyone was also willing to learn and use online resources to overcome any difficulties they came across.

Another potential risk was the possibility of not meeting the deadline. We had several deadlines to meet which were in week 3, week 7 and week 12. Meeting these deadlines whilst also keeping on top of work from other modules could have been a major impediment. However, we were able to effectively stay on top of this by carefully planning the project at regular intervals. Firstly, every week we would have a team meeting discussing everyone's tasks for the week, any difficulties encountered, and other topics related to the progress of the project. These meetings proved to be highly effective in keeping everyone motivated and on track. Having only one meeting a week could have also been a risk as some might view it as not being enough communication for such a large project. However, our team was highly adaptable meaning in weeks where there was a deadline everyone was willing to attend an extra meeting. Also, all team members were active on Discord so no one felt as if they had to wait a week until they could discuss their issues. Lastly, we found the weekly TA meetings to be very effective in keeping us on track as we wanted to ensure we had some progress to show each week.

An unforeseen risk we encountered was the integration of the multiplayer game with the main UI. Originally the networking part of the project ran as three separate programs, one for the server and two for the clients. This was so progress could be made on the networking whilst the main UI was still being produced. One of the issues was that as each client program was calling the *launch(args)* method of JavaFX, once it was integrated with the UI which also called *launch(args)* an error occurred as the *launch* method cannot be called twice within the same program. By removing the call to *launch* more errors were encountered caused by concurrency. These were due to not being able to update the UI from a different thread. As the clients and server depended on the use of threads this was a major cause for concern. However, we were able to overcome this risk by researching different methods to integrate the network. In the end we used the *Platform.runLater()* method to arrive at a fully integrated multiplayer game.

Deciding on a set of requirements is very important in projects as it provides a way to measure whether the outcome fulfils the predetermined expectations. Therefore, a potential risk could have been that our project deviated from what we originally planned. To mitigate this risk, we ensured that our requirements were agreed by all members of the team, to ensure everyone was happy with the expected outcome. We also kept requirements precise to ensure there was no room for misinterpretation between team members, which could have led to problems arising

in the future. We also considered the risk of becoming too focused on the initial requirements, preventing us from coming up with new ideas or alternative game behaviour. To avoid this, we kept the requirements flexible, meaning that if all team members agreed we could change the requirements as we needed to.

One of our requirements was the addition of a shop for players to buy skins with money earnt from playing the game. A risk associated with this was not being able to save the users purchased skins so that they were able to retain their progress from previous sessions. One idea was to create a database, but due to the lack of database experience in our team and the time constraints we decided there were too many risks associated with it. So, in order to solve this problem, we decided on using text files that were capable of saving which items had been purchased and then reading this text file whenever needed.

Overall, there were many risks we encountered; however, we were able to either reduce the risk posed by the problem or in some cases we were able to overcome the issue entirely. In future projects we should refine and be specific about all functional requirements including the “could” requirements. Additionally we should start the integration process much sooner to avoid problems occurring later on in the development process.

Evaluation

The team project has turned out to be quite successful, in our opinion. The game logic is solid, the networking is very good, the AI is exceptional, and the user interface that connects all parts of the game is remarkable and aesthetically pleasing. The game is up to the team's standards and has achieved all set out goals, all thanks to the high level of commitment of the team and their previous programming experience.

Strengths

One of the main reasons that our team stayed on top of things is the realistic plan set early on. Goals have been set for each week, splitting the work into sections, and assigning members to each part. Deadlines were set for each part, and the work was always done on time. The combination of a fair plan and sensible deadlines meant that there was no sudden shake up to the workflow, resulting in a good level of productivity each week, with stress being kept at a minimum.

A clear channel of communication was a key benefit to our team. With Discord being our platform of choice, which many of us already used daily, getting in touch with others and receiving a reply was much quicker than other means. This facilitated collaboration between members and helped us save time. In addition, Discord was used to have an hour-long meeting at the beginning of every week - this time was used to solidify the week's plan and discuss code implementations. Discord also served to quickly bring game bugs to light and assist members with fixing them swiftly.

Another strength that the team possesses is the past experiences of its members. Everyone proved to be quite competent at programming; moreover, all members have worked on at least one of the parts required in the team project like artificial intelligence and networking. This proved to be very useful and made the project proceed more smoothly. Additionally, the team was able to make use of our good research skills to look for ways to code a feature, this cut down on time used on experimenting instead of implementing.

Weaknesses

Even though our planning was concise, some issues were hard to anticipate. Due to the scale of the project and the time frame, parts of the code were not as modular as hoped and took extra time to reconstruct. On the other hand, other parts were too large in scale and would have taken more time to reconstruct than we liked, therefore, they were left as is. Given more time, the team could have implemented them in a better way. An example of which is the Singleplayer and Multiplayer code for the Snake and Map, which needed to be coded differently in both cases to account for networking in Multiplayer.

Another weakness that affected collaboration between members at the beginning of the project is that most of the team had little to no experience with Git. This led to some wasted time early on due to having to manually move code around - however, this only affected the first few weeks

of development. Nevertheless, team members gained valuable experience from using Git, that could be applied in future team and personal projects.

Additional features

Although the team is happy about the project's current state, there are a few additions that could be made to enhance the game even further.

An obvious upgrade would be adding more players to the game. In both singleplayer and multiplayer, a new game mode could be added that utilizes the increase in player count. A "Last Man Standing" mode that pits all players against each other would increase the competitive aspect of the game.

A levelling system would fit nicely to the game's reward system. As of now the only way to show the time invested by a player in the game is the number of coins collected and skins bought. Implementing a system that rewards the player for playing the game by giving them exclusive unlockable skins based on a player's level would be a nice addition to the game.

Currently, the enemy AI's difficulty is scalable and it's able to target apples. However, at its current state it's not able to use any of the powerups on the map. Adding this functionality would make the gameplay more interesting.

The team project was a great experience for everyone. Despite us starting with little to no experience in game development, we were able to explore a new path of programming. The team feels more confident in tackling future game projects and we realise the importance of teamwork while working on a project of this scale. This project proved to all of us that it is not hard to learn a part of programming we are not familiar with, and it can be achieved by spending time on research and planning, and commitment to the project.

Summary

To summarise, we have worked effectively and produced a final product that we are all satisfied with. At the beginning of this project, we decided on a concept which we could achieve within the 12-week timeframe. Everyone came up with great ideas and we chose to make a game based on Snake, adding our own original features. Despite the project being quite challenging due to the COVID-19 pandemic, we have all tried our hardest to finish developing the game. Setting up weekly team meetings helped to motivate us and the use of GitLab allowed us to track our progress. We carefully split up the work equally each week which allowed us to stay organised. Furthermore, as we all have different backgrounds everyone was able to share their knowledge meaning we have all learnt many skills that are useful for software development.

In addition, we believe there were several aspects that we could have improved, given more time. We could have extended the complexity of the game by adding more game modes, including a levelling system, allowing for more than two players, and getting the AI to use power-ups. Besides not getting to implement the aforementioned features, overall, we are happy with our final project.

Teamwork

Our team recognised the importance of collaboration on a project of this scale, so we laid out the groundwork of our project at the start. That began with setting up a clear channel of communication, the team's preferred choice was voice calls via Discord. We chose to have an hour-long online meeting at the start of every week, where each team member's ideas and thoughts were heard. All members attended every meeting and played a crucial part in the project, offering valuable ideas and programming experience. Discord was also used to report bugs, give feedback on work done by other members, and plan individual meetings between team members.

Tasks were split fairly and according to each member's preference; thus, the work was done efficiently, and we were able to assist each other on the tougher parts. The team split the project into four major sections: artificial intelligence, networking, game mechanics, and the user interface. Every team member ultimately worked on at least two parts towards the end of the project. Therefore, we believe everyone is familiar with the entire code base and it's behaviour.

At the end of the first few weeks, the team worked on first choosing a game idea, then planning folder layouts in the design phase. We then worked on integrating our code. After working on independent branches for a short period, we merged the code into the dev branch and had an initial working version. This gave us a massive improvement in our workflow and consequently our productivity. Integration became much easier after the first working version and we managed to maintain a working version for the rest of the development time.

Careful planning and team commitment resulted in exceptional team productivity - all members contributed immensely and everyone got along very well. Work was always done on time and team members supported each other with various issues and bugs they encountered.

Individual Reflections

Daniel Batchford

I believe we worked very effectively as a team and I believe this is evident in our final game. The team was very easy to work with and everyone got on very well. Our team worked professionally and with motivation - tasks were completed on time and to a high standard, and members were always willing to take on a new task. When code issues arose, team members helped each other to resolve them, through discord. Team members were always willing to help (even in the early morning hours!). I believe we communicated well as a team, in a professional manner. Weekly meetings were effective and allowed us to plan for the week ahead. Everyone contributed with ideas and questions about the development - these included current bugs, potential future bugs and architecture decisions. Furthermore, the team came up with new ideas throughout the project, which helped with the evolution of our game as well as our overall motivation.

By tailoring tasks based on each individual's strength and knowledge of the task, we were able to produce high quality work from each member. For example, I have had previous experience writing a snake AI for a single player version of a game, so I tackled the AI task. This included a pathfinding library which could handle wrapping around the edge of a grid, akin to a torus geometry from a topological perspective. I wrote the behaviour of this AI and once this was completed, I worked on implementing the single player mode alongside Rahul and Yuji. This included multithreading the single player mode to help with performance, implementing UI button behaviour in the game and adding new features such as a countdown screen. I also helped Rahul with the Map Builder code and UI design. Next, I worked on the game logic by optimising code written by Mohammad and Andrei. This included implementing a more OOP focused approach to the Snake class, as well as writing code to allow communication between the game logic and AI code.

I also worked on FXML and CSS styling with Rahul. I created the CSS styling for UI elements in the game, such as sliders and buttons, as well as reorganising previous FXML documents by organising elements in containers, allowing a resizable UI to format elements in a more predictable way. Throughout the project, I also maintained the Git repo - this involved merging branches on a regular basis as well as fixing merge conflicts which arose. Towards the end of the project, I also wrote JavaDoc for the entire code base, test classes for the AI and an intro video and main menu background.

At the end of the project, we all contributed to write sections of this report. I wrote the software engineering processes section, the introduction section and the unit tests inside of the test report.

I believe our individual contributions came together to form a great game. This was possible because of our team's willingness and hard work throughout the term.

Florian-Andrei Blanaru

The team project was a quite intriguing experience. Working alongside all of Team Paradroid's members, I had the pleasure of developing my teamwork skills, as everyone was open-minded and hardworking towards creating a product that, in the end we can all say that we are proud of. I believe that using Discord helped us communicate effectively, as we had weekly meetings on Mondays to sort our tasks for the week and a few spontaneous meetings to discuss problems which needed our immediate attention. Everyone was open and happy to help whenever needed, outside those meetings. All the team members solved their tasks efficiently and on time, making the whole process run smoothly.

At the start of the project, we managed to agree quickly and efficiently on our project idea and plan. Initially, our idea was not complex, but as everyone worked and got their parts done, we were able to progressively add new features to the game, which proved to be an efficient way of addressing the workload. Thus, we did not end up planning for features that we would have to scrap because of time constraints, and we were able to add all our ideas to the game.

Mohammed and I agreed to work on the logic and mechanics of the game, which proved to be a fruitful collaboration, as we were both able to build the basic structure of the game, so that the rest of the team could start adapting their code on the one provided by us. Mohammed was always open to discussion and provided useful ideas and solutions on how we could implement and, later, improve the game.

I was directly involved in writing the code for collisions (with the walls, fruit, other snake, or powerups), the random map generator and the powerups. I communicated constantly with the team on what ideas we should implement and how, such that, in the end, everyone would be happy with the result. I have also been actively open to help other teammates, which helped me understand how other parts of the project worked and how they were brought together. This was also enforced by the JUnit tests I conducted on the game features that Mohammed and I worked on, which involved going through most of the code to write tests that could help us find other bugs or coding aspects that we did not consider before.

In conclusion, I believe that the whole project went smoothly, as everyone was involved and helpful towards building the best product we could, given the time and resources. All the team members were motivated and dedicated towards our goal, which was one of the main reasons we succeeded.

Mohammed Jaber Alqasemi

The team project was quite the experience for me, there is much that I have learned from working in a team. All team members spent a fair amount of effort on the game and improved it to their best of abilities given the limited time. We all worked together very effectively; we were able to collaborate more often given that Discord was used for communication. Issues that arose were fixed quickly and efficiently, while team members provided support for anyone that needed it. All major parts of the game were polished and up to a high standard due to the team's great programming experience.

I worked on a few different parts of the project, starting with game mechanics. Since this was one of the very first parts to be worked on, this required a careful thought process to avoid making foundational mistakes that could prove detrimental later. Thankfully, I was paired with Andrei who I could share ideas with and hear his suggestions. We were swiftly able to construct a good groundwork for displaying elements on screen using JavaFX, that is used throughout the main game, the Map Builder, and the Shop.

Another major part which I worked on is Skins, which I believe is a major part that was reused in all parts of the game. I created the code that loads Map and Snake skins from images; these were later constructed into objects that are displayed in game. Skins are also part of the reward system of the game, they can be bought and equipped in the shop for players to customize their experience. I have also implemented the code that manages all new skins that are added to the Map and Snake skin folders, this facilitates the process of adding new skins and allows the game to be expanded easily if need be.

I also helped Rahul with integrating Map skins with the Map Builder, which made the Map Builder retroactively update the map skin whenever the player equips a new map skin from the shop. In addition, I updated the Map Builder code which on its initial implementation did not use map background and wall skins, instead it used letters to differentiate between the two.

An additional part that I worked on that involves no coding was making the image files for the skins and some audio sounds for the snake. The image files required combining, editing, and resizing images. As for the audio sounds, besides looking for some that are suitable to the game, I had to do minor editing such as trimming and equalizing.

I believe the team project was a success, everyone always finished their work and always turned up on time, without the team's efforts and contributions the game would not have been possible.

Yuji Fukuta

Through this group work, there were so many things that I have learned. This project was my first huge project creating something from scratch, except what we did for the summer bridgework. Everyone in this group was very motivative and in the coding part they gave me so much advice. I was really surprised that all the members had creative brains which made the game more enjoyable. I also enjoyed this project, especially achieving the part I have entrusted.

During this project I have worked on the User Interface and Audio mainly. In the week we started, we have created a separate UI with Rahul and sum up to create a base for our game. After that I have started linking music to the button and the screen. Adding new features was very challenging for me, but as I got used to it, it took less time to research, however it was still difficult to add something to the game during the whole project.
On the other hand, I worked on the read/write files. We are using the text files to save quite a lot of stuff, for example mute selected, balance, music volume etc... so it needs to be loaded whenever the value is needed. At the very end, I also make the Lives counter and result screen. They were some bugs and took me a while to figure out all the features worked fine by adding it. It was a great experience to touch the other fields which I did not write about. This gave me a great idea of how the game works in the background and utilized the other people's codes.

The thing that made me most motivated was the meeting every week on Discord and checking our progress. Discord was a great service to communicate with group members efficiently. We also got a small group to complete those tasks and I worked with Rahul most of the time. He was the best partner for me and gave me good advice. In addition these small groups make me effective at the work and if I did not have someone to talk to then I don't think I would manage to finish it.

Working with Team Paradroid was enjoyable and a good experience. Everyone worked so hard and ended up with the great game. However, I always feel my skills level is a bit behind compared to everyone so this project was a good experience to improve my skills and would like to continue improving in the future. Finally, I believe that everyone in this group did fair amounts of work and I don't think I could have worked comfortably without having this member so I would like to say great thanks to everyone.

Rahul Gheewala

I thoroughly enjoyed working with the other team members. I consider myself to be extremely fortunate to be put in Team Paradroid, due to the fact everyone was highly motivated, self driven and shared a common vision to produce a high quality and fully fledged game. The team worked very smoothly all throughout, and I believe this was a result of not only the expertise of the team, but the fact that we took full advantage of online services such as discord, google services and git services. It was also noticeable how many of us sacrificed a lot of our free time and worked non-office hours to help improve and develop the game.

In this project, I dedicated most of my time to the User Interface, Shop feature, Map Builder, Audio, optimization of the game and reformatting of the whole project (such as rewriting code to improve efficiency, reduce overall file size and optimize speed). From the very first week, I knew designing and functionality is a strong suit of mine. For this reason, I decided to take upon the task of constructing the base of the game where I added all the buttons and containers to at least a sub working standard. From here, I continued to develop and improve the UI, taking on the advice from my teammates and finally created a complete, minimalist looking UI which serves all the functionality a user would require. As aforementioned, collaborating the code from other team members was a challenge, but after trying multiple options I was able to do so successfully and learn many skills in the process to become very proficient.

Being an avid gamer who plays many MMOs, I couldn't imagine our game any other way than having a Shop feature. I committed from a very early stage that I would go through the extra effort of using File Handling techniques in order to save user progress. After completing the base of the shop, I developed a currency which I linked to the game. With the help of Mohammed, skins and maps were linked to the game code. Throughout the Shop's implementation, I further developed my logic skills and eventually created what I feel is a well-developed shop which I hope will create an enjoyable experience from the user. The map feature was another feature I was keen to work on, here I developed a working Map Builder but with the help of Daniel and Mohammed I was able to improve the efficiency and linkage to items in the shop. I also worked alongside Yuji where I helped to debug issues with the Audio and link to the UI.

Aside from coding, I helped out in other areas of the report such as the requirements specification and reporting of the features, designing as well as trying to help others in the team whenever it was needed.

Dilpreet Kang

I had a very enjoyable experience during this team project mainly due to how easily and effectively the team worked together. Having weekly team meetings helped us to stay focused and organised, allowing us to make progress smoothly. All team members were happy to help even outside of normal working hours.

I took the opportunity to work on the networking aspect of the game, as it has always been an area I wanted to learn more about. As this field was completely new to me, I had to spend a bit of time researching the different ways of implementing a multiplayer game. During my research I became familiar with object streams, sockets, and the importance of concurrency, which proved to be helpful throughout the project. In the end I decided on a client-server model using a TCP connection. As Andrei and Mohammed added game mechanics, I continuously incorporated the new features into the multiplayer game. This was difficult at times, as aspects that would work well in single player did not work as expected over the network. However, I was able to overcome these setbacks by modifying the code or taking a different approach to the problem. By the end of the project, I had designed and implemented a server class that could communicate the game state between two players. I had also created a client class that was able to send and receive game updates from the server.

As part of getting the multiplayer to work I had to work partially on the UI. This included adding the relevant code to the buttons Rahul and Yuji had previously created. I also created new dropdown menus to allow the host to choose a map and number of lives. This allowed me to understand more about how the UI was designed and gave me an insight into FXML files. Initially I was working on the networking through three separate programs, one for the server and two for the clients. So, I also worked on integrating the multiplayer game itself with the UI.

Incorporating the shop with multiplayer was another part I worked on. As our game had functionality for two players, we believed it was important to also allow the second player to be able to unlock skins and save their progress. In order to do this, I created a file that tracked how many clients were open in order to assign an ID to the player that was used to access their shop.

Overall, everyone worked hard to produce the game and I believe this is reflected in the final product.

Appendix

Software Principles and Coding Standards

- We modified IntelliJ's default code style to minimise spaces between methods and rearrange fields based on their visibility.
- Naming Styles
    - Package names were lowercase (e.g. server.ai.pathfinding.finders, server.game.managers.mapmanager)
    - Constants are typed in constant case (e.g X_SCREEN_WIDTH, UPDATE_INTERVAL)
    - Class names and annotations are typed in upper camel case (e.g AudioController, @Override
    - Methods and fields are typed in lower camel case (e.g getSinglePlayer(), playerSnake)
- Field hierarchy
    - Public static fields
    - Package - private static fields
    - Protected static fields
    - Private static fields
    - Public fields
    - Package - private static fields
    - Private fields
- Method hierarchy:
    - Public static methods
    - Package - private static methods
    - Protected static methods
    - Private static methods
    - Public methods
    - Package - private static methods
    - Private methods
    - Getters, setters, toString(), hashCode(), equals()
- JavaDoc was used to annotate all public methods
- Where possible, inheritance was used to reduce the size of the codebase and enable easier future expansion
- Interfaces were used in some packages to increase modularity
- Constants were stored in interfaces where possible for easy access throughout the codebase

Test Report

Introduction

The purpose of this test report is to demonstrate how the needs represented by the functional and non-functional requirements have been met. Additionally, we aim to show that the possibility of bugs arising once the game is distributed is low, and the game functions as users would expect it to.

Goals

We aim to achieve a relatively bug free game. However, extreme edge cases that would arise if a player purposely tried to break a game were deemed as being acceptable, given the time constraints of the project. We instead focused on avoiding the game crashing and more significant bugs which would affect gameplay and consequently the user's experience.

Approach

Due to the nature of our game, we decided to take a largely black-box testing approach with a lesser emphasis on white-box testing. Black-box testing allowed every member of the team to test each component without needing to know about the underlying functionality. Additionally, it allowed us to test the game as a whole and ensure individual features were working correctly together. In order to do this, we created a pass/fail criteria with each test case representing a functional requirement. As the project progressed, we were able to fill in the outcomes of the tests and catch bugs early. For parts of the code that heavily relied on calculations or had many boundary conditions we also used white box testing methods, to ensure the logic was correct. For this we created J-Unit tests that were able to test a wide variety of cases and ensure important methods work as they should. In these cases, we used J-Unit tests and we aimed for at least 75% coverage.

Additionally, we used user testing to test the game. We curated a set of questions which gave a broader look at the game. After working on this project for a significant amount of time, there is a risk of becoming too used to how features should behave. By having user testing, we were able to gain feedback we would otherwise have missed. User feedback was gained from a user not familiar with games and one who was. This gave us a wider range of feedback which more closely represents our target demographic. Specific user's were kept anonymous, for privacy reasons.

Constraints

Our main constraint with our testing approach was time. While more white-box testing would give an improvement to our testing approach, it is more time consuming than black-box testing. We wanted to focus this time on core functionality as opposed to unit tests. Furthermore, white-box testing proved difficult with some classes, as they included JavaFX components. These components proved hard to test and the introduction of javafx.swing was needed to test some packages such as *server.game.managers*.

Testing Timeline

Week 3 - Initial Integration - Initial black-box testing of core functionality.

In week 3, we aimed to have a fully integrated codebase. However, we failed to meet this deadline so testing could not occur during this week. However, progress screenshots were made which gave us an insight into the future direction of the project.

Week 8 - White-box & user testing

In week 8, we aimed to have white-box test classes for areas of the project which were more predictable, such as the *server.game.managers* & *server.game.ai* packages. While our testing did not yet achieve good coverage, it gave us an indication that these packages worked as they were intended. All tests in these packages passed.

We also conducted user testing in this week, using a relative of a team member and a flatmate of another member as test subjects.

Following this week, we continued aiming for a higher level of coverage for the packages tested with white-box testing.

Week 12 - Final version - Black-box testing

During the final week, we conducted black box testing to ensure all core functionality met the specified requirements. We also ensured all previous white-box tests passed and ensured user feedback had been acted on.

Testable Features

Testable features included:

- Game mechanics
- AI behaviour
- Networking behaviour
- UI design and functionality
- Audio behaviour
- Shop functionality
- Map building functionality

Pass / Fail criteria

For unit tests, the pass and fail criteria is fairly straightforward, using the results of a JUnit test to dictate whether a test passed or failed. However, for black-box testing, the pass-fail criteria is far more subjective. If one team member believed a certain behaviour was a bug whereas another believed it was expected behaviour, we simply cross referenced the actual behaviour with the expected behaviour noted by the requirements.

Estimates

White-box testing was fairly quick, as it can be run within a few seconds. For black-box testing, we estimate that a full test of all the cases provided below will take roughly 1 hour. More time was allocated towards testing in week 12, in line with our week 12 testing deadline.

Responsibilities

To ensure an equal workload, we dedicated testing of functionality to the user who wrote that portion of code. For unit tests, we expected that each team member should write a corresponding test class for each class that member created.

For black-box testing, we used a secondary opinion from another team member to help reduce any subjectivity introduced with this type of testing.

Unit Testing

server.ai

Package	Test Class	Test	Result	Coverage
server.ai.pathfinding.finders	TestAStarFinder	testSimplePath	Pass	100%
		testBlockedPath	Pass	
		testSemiBlockedPath	Pass	
		testComplexPath	Pass	
		testNotDisjointPath	Pass	
		testNullOriginNode1	Pass	
		testNullOriginNode2	Pass	
		testTwoLengthPath	Pass	
		testTenLengthPath	Pass	
	TestLongestPathFinder	testSimplePath	Pass	100%
		testBlockedPath	Pass	
		testSemiBlockedPath	Pass	
		testComplexPath	Pass	
		testNotDisjointPath	Pass	
		testNullOriginNode1	Pass	
		testNullOriginNode2	Pass	

	TestTorusAStarFinder	testSimplePath	Pass	100%
		testBlockedPath1	Pass	
		testBlockedPath2	Pass	
		testSemiBlockedPath	Pass	
		testComplexPath	Pass	
		testNotDisjointPath1	Pass	
		testNotDisjointPath2	Pass	
		testNotDisjointPath3	Pass	
		testNullOriginNode1	Pass	
		testNullOriginNode2	Pass	
		testTwoLengthPath	Pass	
		testTenLengthPath	Pass	
server.ai.pathfinding	TestNode	testGetNode	Pass	81%
		testSetParent	Pass	
	TestNodeGrid	testInvalidDimensions1	Pass	95%
		testInvalidDimensions2	Pass	
		testNullDimensions	Pass	
		testNegDimensions1	Pass	
		testNegDimensions2	Pass	
server.ai.util	TestUtil	testGetDistanceA1	Pass	100%
		testGetDistanceA2	Pass	
		testGetDistanceB	Pass	
		testGetDistanceC	Pass	
		testGetDistanceD	Pass	
		testGetDistanceE	Pass	
		testSubtract2DIntArray1	Pass	

		testSubtract2DIntArray2	Pass	
		testSubtract2DIntArray3	Pass	
		testDotProduct2DIntArray1	Pass	
		testDotProduct2DIntArray2	Pass	
		testDotProduct2DIntArray3	Pass	
		testDotProduct2DIntArray4	Pass	

server.game

Package	Test Class	Test	Result	Coverage
server.game.managers.snakemanager	SnakeTests	testSpawnSnake	Pass	72%
		testChangeSpawn	Pass	
		testUpdatePlayerSnakeA	Pass	
		testUpdatePlayerSnakeB	Pass	
		testHandleFruitA	Pass	
		testHandleFruitB	Pass	
		testHandlePowerUp	Pass	
		testFreeze	Pass	
		testWallSkip	Pass	
		testControlsInverter	Pass	
		testSetMine	Pass	
	TestSnakeSkinManager	testInit	Pass	85%
		testGetSnakeSkin	Pass	
		testGetSelectedSkin	Pass	

		testGetSelectedSkinID	Pass	
	TestSnakeSkin	testLoadingSkinsA	Pass	100%
		testLoadingSkinsB	Pass	
		testLoadingSkinsC	Pass	
server.game.managers.mapmanager	MapTests	testMapFileRead	Pass	80%
		testGenerateMapAndGenerateRandomWalls	Pass	
		testGetX	Pass	
		testGetY	Pass	
		testSetXA	Pass	
		testSetXB	Pass	
		testSetYA	Pass	
		testSetYB	Pass	
		testGetMapValue	Pass	
		testSetMapValue	Pass	
		testGetMap	Pass	
		testSetMatrix	Pass	
	TestMapManager	testInitAndCreateMaps	Pass	100%
		testGetAllFileNames	Pass	
		testGetMap	Pass	
		testGetAllMaps	Pass	
	TestMapSkin	testLoadingSkinsA	Pass	100%
		testLoadingSkinsB	Pass	
		testLoadingSkinsC	Pass	
	TestMapSkinManager	testInit	Pass	100%
		testGetMapSkin	Pass	
		testGetAllSkins	Pass	
		testGetSelectedMap	Pass	

		testGetSelectedMapID	Pass	
		testClosestBgColor	Pass	
		testClosestBgColorID	Pass	
server.game.managers.powerupsmanager	PowerUpsTest	testGetPowerType	Pass	90%
		testSpawnPowerUp	Pass	
		testDespawnPowerUp	Pass	
		testSetControlsInvertTimer	Pass	
		testGetControlsInvertTimer	Pass	
		testSetFreezeTimer	Pass	
		testGetFreezeTimer	Pass	
		testSetWallSkipTimer	Pass	
		testGetWallSkipTimer	Pass	
		testPlantMine	Pass	
server.game.usables	CoordinateTests	testGetY	Pass	76%
		testSetY	Pass	
		testGetX	Pass	
		testSetX	Pass	
		testAddX	Pass	
		testAddY	Pass	
		testIsWalkable	Pass	
		testToIntArray	Pass	
		testEquals	Pass	
		testHashCode	Pass	
		testToString	Pass	
server.game	FruitTests	testSpawnFruit	Pass	80%
		testDespawnFruit	Pass	

Black Box Testing

Game Logic

Testing of game mechanics

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	Player should move in the direction of an arrow key if the current direction is not the opposite	When the player is not moving right press the left arrow key	Player moves left	PASS
2	Player should not move in the direction of an arrow key if the current direction is the opposite	When the player is moving right press the left arrow key	Player continues moving right	PASS
3	Player should be able to use an equipped powerup by pressing spacebar	After the player has received a powerup press space	Player can use powerup	PASS
4	Player should not be able to use a powerup by pressing space if they do not have a powerup equipped	Before the player collides with a powerup press the space button	Nothing related to powerups should happen	PASS
5	The freeze powerup should freeze the other player for a short period	After the player has received the freeze powerup press space	Other player freezes	PASS
6	The invert powerup should invert the other player for a short period	After the player has received the invert powerup press space	Other player controls are inverted	PASS
7	The mine powerup should allow the user to plant a mine at the position of the players last body coordinate, if either player collides with this planted mine, they should shrink, and the planted mine should be removed	After the player has received the mine powerup press space to plant it and collide with the mine position twice	On first collision player should shrink and respawn, on second collision nothing should happen	PASS
8	The boost powerup should speed up the current player for a short period	After the player has received the boost powerup press space	Player speed should be doubled	PASS
9	The coin powerup should add a random amount to the players balance	Check the player balance before and after they receive the coin powerup	Player balance should increase	PASS
10	The wall skip powerup should let the current player skip through walls for a short period	After the player has received the wall skip powerup press space and move into a wall	Player should not collide with the wall	PASS
11	Player should be able to collide with fruit to increase their length by 1 and decrease other players length by 1	Move the player over a fruit and observe the length of this player, and length of the other player	This players length should increase by 1 and other players length should decrease by 1	PASS
12	Player should be able to collide with fruit to	Move the player over a fruit and observe the players balance	This players balance should increase by 50	PASS

	increase their balance by 50			
13	The outer boundaries of the map should teleport the player to opposite side	Move the snake left towards the outer bounds	Player should be teleported to the other side with the same y coordinate	PASS
14	The map should always start by spawning 1 fruit and 2 powerups at random coordinates	Observe map at start of the game	There should be 1 fruit and 2 powerups	PASS
15	When a player collides with fruit the fruit should be despawned and respawned at a random coordinate	Move player over a piece of fruit	Fruit should despawn from original location and respawn at a new location	PASS
16	When a player collides with a powerup the powerup should be despawned and a new random powerup is spawned	Move a player that does not have any powerups equipped over a powerup	Powerup should despawn and new powerup should spawn at a new location	PASS
17	When a player that already has a powerup equipped collides with a powerup nothing should happen	Move a player that already has a powerup equipped over a powerup	The powerup should not despawn and remain as it is	PASS
18	When a player's head collides with a wall, they should be respawned to their start location and lose 1 length	Collide a player with a wall	The players length should decrease by 1 and they should be respawned at their start point	PASS
19	When a player's head collides with another player the player should be respawned to their start location and lose 1 length	Collide a player with the other player	The players length should decrease by 1 and they should be respawned at their start point	PASS
20	When the players length reaches 0, they should lose a life	Keep colliding with a wall until the players length is 0 and observe this players hearts	Players hearts should decrease by 1	PASS
21	When the players lives reach 0 the game should end	Keep reducing the players lives until they have no lives left	The game should end	PASS
22	When the game ends the UI should tell the player whether they won or lost	Purposely lose the game and then win the game	First the game should display that you lost and then that you won	PASS

Networking

Testing of the multiplayer aspect of the game

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	The game should start once two players are connected to the server	Create a game in one MainClient and join a game in another	The game should begin for both players once the second player joins the game	PASS
2	Each players movement should be replicated on the other players client	Move one player's snake randomly and observe this snake on the other players client	The snake's movement, direction and speed should be identical on both clients	PASS
3	Each client should receive an identical map from the server	Once the game begins observe the map	Both maps' walls, fruit and powerups locations should be identical	PASS
4	Each client should receive the same number of lives to begin with, from the server	When creating the game select 4 lives	Each client should start with 4 red hearts	PASS
5	The server should communicate information about collisions	Collide a snake with a wall, a piece of fruit, a powerup and the other player	In each case the clients should show this specific snake colliding, respawning, and losing 1 length	PASS
6	Each client should receive each players' equipped snakeskin from the server and display this identically on each client	Equip a different skin for each player and start the game	Both clients should display the snake with the correct skin	PASS
7	When a player leaves the game, the server should be turned off	Start a game and then leave the game before a player has won/lost, then try to join a game	You should not be able to join the game as the server is off	PASS
8	When the game is over due to a win/lose the server should be turned off	Wait till the game is over and try to join a game	You should not be able to join the game as the server is off	PASS
9	A client should not be able to join a game before a game has been created	Try to join a game without creating a game	You should not be able to join a game	PASS
10	More than one server should not be allowed to be run at once	Try creating a game on both clients	Only one client should be able to create the game, nothing will happen for the other client	PASS

UI

Testing of the user's interaction with the game and menus

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	The main menu should display the following buttons: Singleplayer, Multiplayer, Build a Map, Shop, Audio, Quit	Start the game and observe the main menu	All these buttons should be present and clickable	PASS
2	When the Singleplayer button is clicked the following buttons should be shown: Easy, Medium, Hard	Start the game and select Singleplayer from the main menu	All these buttons should be present and clickable	PASS
3	When any single player difficulty is selected there should be a countdown before the game begins	Select easy difficulty in single player	There should be a 3 second countdown displayed on the screen before the game begins	PASS
4	When any single player difficulty is selected there should be a button to reset the game	Select easy difficulty in single player and press reset	The map should be randomised, and the players lives, and positions should be reset	PASS
5	When any single player difficulty is chosen there should be a button to pause the game	Select easy difficulty in single player and press pause	The game should stop in its current state	PASSs
6	When any single player difficulty is chosen if the game is paused there should be a button to un pause it	Select easy difficulty in single player and press pause followed by un pause	The game should stop in its current state and the restart in its previous state	PASS
7	When any single player difficulty is chosen there should be a back button	Select easy difficulty in single player and press back	The client should return to the single player difficulty selection screen	PASS
8	When any single player difficulty is chosen there should be a dropdown list to use a custom map	Select easy difficulty in single player and choose a custom map	The game should restart the countdown and start the game on your chosen map	PASS
9	In single player the lives of both the player and AI snake should be displayed and updated	Start the game and cause the player to lose a life	The players hearts should start with 3 and then decrease to 2	PASS
10	The balance of the current player should always be displayed in the top right-hand corner in single player and multiplayer modes	Observe that the balance remains in the top right corner when in single player and multiplayer modes	The balance of the relevant player should be visible	PASS

11	When the Multiplayer button is pressed the following buttons should be displayed: Create a Game, Join a Game	Press the Multiplayer option and observe the visible buttons	Create a Game and Join a Game should be clickable and two drop down buttons should be unclickable	PASS
12	In multiplayer when Create a Game is pressed one dropdown should be clickable	Press the Multiplayer button and then press Create a Game	The dropdown to select a number of lives should be clickable and show numbers 1-5	PASS
13	In multiplayer when the number of lives is chosen the second dropdown should become clickable	Create a game and finish selecting the number of lives	The dropdown to choose a map should become clickable and show custom maps as well as random map	PASS
14	In multiplayer when join a game is clicked the client should be able to join a game	First create a game in one client and then join a game in another client	The game should start	PASS
15	In multiplayer when join a game is clicked but a game has not been created a player should not be able to join a game	Select join a game without creating a game first	Nothing should happen	PASS
16	In multiplayer two clients should not be able to create a game	Select create a game in one client and then create a game in another client	In the first client the player should be able to create a game and "Finding an Opponent" should be displayed, nothing should happen for the second client	PASS
17	When the Multiplayer button is pressed there should be a button to go back	Select the multiplayer button and then click back button	The client should return to the main menu	PASS
18	When the Build a Map button is pressed there should be a clickable map	Press Build a Map and click the displayed map	The squares that are clicked should be filled with a wall	PASS
19	When the Build a Map button is pressed there should be a text field to enter a map name	Press Build a Map and enter a name into the text field	The text field should be responsive and display the character being typed	PASS
20	When the Build a Map button is pressed there should be a button to save the map	Create a random map, name it testMap and click save map button	The testMap should be visible in the games map files	PASS

21	When the Build a Map button is pressed there should be an option to reset the current map	Add random walls to the map and press reset	The map should revert to a blank map	PASS
22	When the Build a Map button is pressed there should be a back button	Press the back button	The client should return to the main menu	PASS
23	When the Build a Map button is pressed there should be instructions explaining the feature	Press the Build a Map button and observe the screen	There should be instructions on the left and top right of the screen	PASS
24	When the Shop button is pressed there should be buttons to select a snakeskin and map skin	Press the Shop button and select the first snakeskin and last map skin	A notification on the bottom should be displayed: "Skin 1 and Map 3 has been selected"	PASS
25	When the Shop button is pressed there should be a back button	Press the Shop button and then the back button	The client should return to the main menu	PASS
26	When the Audio button is selected there should be a slider to select music volume	Press the Audio button and move the music slider all the way to the right	The music should be very loud	PASS
27	When the Audio button is selected there should be a slider to select effects volume	Press the Audio button and move the effects slider all the way to the right	The sound of button clicking should be very loud	PASS
28	When the Audio button is selected there should be a tick box to mute all audio	Press the Audio button and check the box Mute Audio	There should be no music playing and no sound when buttons are pressed	PASS
29	When the Audio button is selected there should be a button to Reset to Default	Press the Audio button and press the Reset to Default button	The sliders should revert to the middle and the music and effect volume should be moderate	PASS
30	When the Audio button is selected there should be a button to go back	Press the Audio Button and then press back	The client should return to the main menu	PASS
31	When the Quit button is pressed the clients' UIs should close	Open the game and press Quit	The game should close	PASS

AI

Testing the functionality of the AI

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	In each single player difficulty mode, an AI snake should be present	Start a game in easy, medium, and hard	An AI snake should be visibly moving and playing the game	PASS
2	In single player hard mode, the AI snake should be using the feature to wrap around the map edges	Start a game in hard mode and observe the AI snake	The AI snake should be using the wrapping feature to get to a fruit if it is faster to do so	PASS
3	In single player easy and medium mode, the AI snake should not be using the feature to wrap around the map edges	Start a game in easy mode and observe the AI snake	The AI snake should not be using the wrapping feature to get a piece of fruit	PASS

Build A Map

Testing the functionality of the map builder

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	The build a map feature should use the players equipped map skin	Equip map skin 3 and view the build a map feature	The displayed map should be white and walls should appear as brown biscuits	PASS
2	When a tile is pressed a wall should be rendered, when the same tile is pressed again the wall should be removed	Press 5 random tiles and then repress these 5 tiles	The map should end up blank	PASS
3	Build a map should save maps when complete	Create a random map and save it as testMap2, then go into single player and choose this map	The map you drew previously should be displayed as the current map	PASS
4	Build a map should revert the map to a blank map when reset is pressed	Place 10 random tiles on the map and press reset	The map should be blank with no tiles	PASS

Audio

Testing the sound within the game

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	The music should start playing once the MainClient is opened	Start the game	The music should be playing at the client's music volume setting	PASS
2	When a button is pressed a clicking sound effect should be played	Press a variety of buttons	Each click should make a clicking sound	PASS
3	When a snake collides with a wall a collision sound effect should be played	Collide with a wall in both single player and multiplayer mode	A collision sound effect should be heard immediately	PASS
4	When a snake collides with another player a collision sound effect should be played	Collide with the other snake in both single player and multiplayer mode	A collision sound effect should be heard immediately	PASS
5	When a snake collects a powerup that is not a coin a collection sound effect should be played	Collect a powerup in both single player and multiplayer mode	A collection sound effect should be immediately heard	PASS
6	When a snake collects a coin powerup a coin collection sound effect should be played	Collect a coin powerup in both single and multiplayer mode	A coin collection sound should be heard	PASS
7	When a snake uses a powerup a powerup sound effect should be played	Collect a powerup in both single player and multiplayer mode	A powerup sound effect should be heard immediately	PASS
8	When a snake eats a fruit a consuming sound effect should be played	Eat a piece of fruit in both single player and multiplayer mode	A consumption sound effect should be heard immediately	PASS
9	When a snake loses a life a death sound effect should be played	Collide with a wall until a life is lost in both single player and multiplayer mode	A death sound effect should be heard immediately	PASS
10	The audio settings should be saved even when the game has been closed	Turn the music volume to full and sound effect volume to zero click apply and close the game	When the game is loaded the music should be loud, but no sound effects should be heard	PASS

Shop

Testing of the functionality and display of the shop feature

Test ID	TEST DESCRIPTION	TEST STEPS	EXPECTED RESULT	PASS/FAIL
1	When a snake or map skin has not yet been purchased it should be available to purchase	With a balance of 1000 purchase the first snakeskin	The first test should cause the purchase button to change to equip	PASS
2	If a player does not have enough money, they should not be able to purchase a skin	With a balance of 0 purchase the first map skin	You should not be able to purchase the skin and you should be alerted you do not have enough money	PASS
3	When a player equips a skin, it should be rendered as their skin in a single player game	Equip snakeskin 1 and map skin 3 and start a single player game	The map should be white with deserts as food and the snake should be purple	PASS
4	When a player equips a skin, it should be rendered as their skin in a multiplayer game	Equip snakeskin 1 and map skin 3 and host a multiplayer game Equip snakeskin 2 and map skin 2 and join a multiplayer game	This clients map should be white with deserts as food and this snake should be purple This clients map should be blue with mushrooms as food and this snake should be red	PASS
5	A user should be given a free default snake and map skin	Create a fresh version of the game and start a single player game	This snake should be green, and the map should be brown with apples as food	PASS
6	Equipped skins should be saved even when the game is closed	Equip snakeskin 1 and map skin 3 and close the game, then start a single player game	The map should be white, and the snake should be pink	PASS
7	Equipped skins should be saved even when the game is closed	Open two MainClients: In the first select snakeskin 1 and map skin 3 In the second select snakeskin 2 and map skin 2 Then close the game and start a multiplayer game	The first client should have a white map and a pink snake The second client should have a blue map and red snake	PASS

User Testing

User 1 - (Non Gamer)

Question	Response	Action Taken
Do you think the game is easy to understand?	Yes, but no explanation on controls and what the different items mean etc.	An information screen was added to both singleplayer and multiplayer modes
Do you think the game is easy to play?	Yes	None
Do you find the menu items easy to use?	Yes - self explanatory and lots of options	None
Do you think there is a good level of difficulty to the game?	Yes with the 3 levels	None
Do you feel like there is enough incentive to keep playing the game?	No - it's a bit boring.	
Did you encounter any bugs whilst playing the game?	No - was all good	None
What do you think could be changed about the game?	Maybe a leaderboard to make it more exciting.	We will consider adding this feature in a future version of the game, if there is enough time to do so

User 2 - (Gamer)

Question	Response	Action Taken
Do you think the game is easy to understand?	Yes, the main menu clearly explains everything needed to play the game	None
Do you think the game is easy to play?	Yes, the mechanics are simple to understand	None
Do you find the menu items easy to use?	Yes	None
Do you think there is a good level of difficulty to the game?	Yes, the difficulties of the AI are very good	None
Do you feel like there is enough incentive to keep playing the game?	No after a short time playing, I unlocked most of the skins so there wasn't much else to work towards	None
Did you encounter any bugs whilst playing the game?	Yes, when I play on a custom map in single player there are no powerups on the map	Fixed this issue
What do you think could be changed about the game?	More items in the shop and multiplayer working on more than one PC	We will add more shop versions in a future game version, time permitting. Making the game work on multiple PC's would require networking outside of localhost, which we believe is too ambitious given the time frame.

Testing Evaluation

In summary, we believe a combination of white-box, black-box and user testing gave us the flexibility to test our work in the most efficient way. However, we believe that more white-box testing would have been useful in order to anticipate future bugs earlier and before they were pushed to version control. An improvement to the modularity of our game would have made it easier to develop unit tests for certain components in our game. In the future, we would aim to produce game components that are more flexible and therefore more testable. Looser coupling of components would have also helped with detecting bugs that occurred due to testing.

However, due to the nature of the project, black-box testing is absolutely necessary in order to produce a high quality game. A high quality game needs to not only have minimal bugs but also a nice user experience. Black-box testing allowed us to test this experience from a broader and more natural perspective than unit testing allows.

Overall, while our testing strategy was effective, it could be improved by increasing code modularity through focusing more on testing during the earlier stages of development.

Subsystem UML Diagram

Figure 11: Objects and Usables Class Diagram

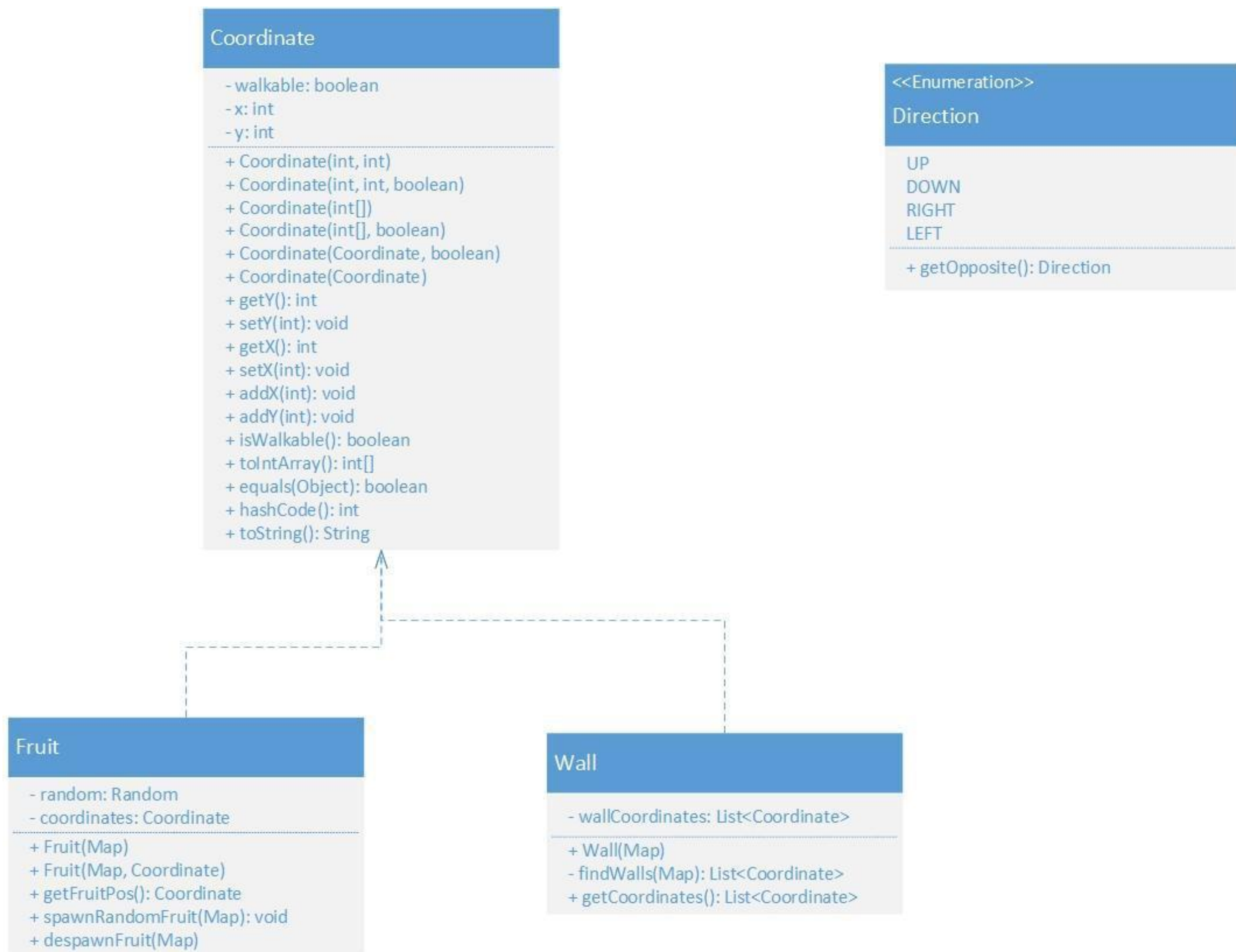


Figure 12: Snake Class Diagram

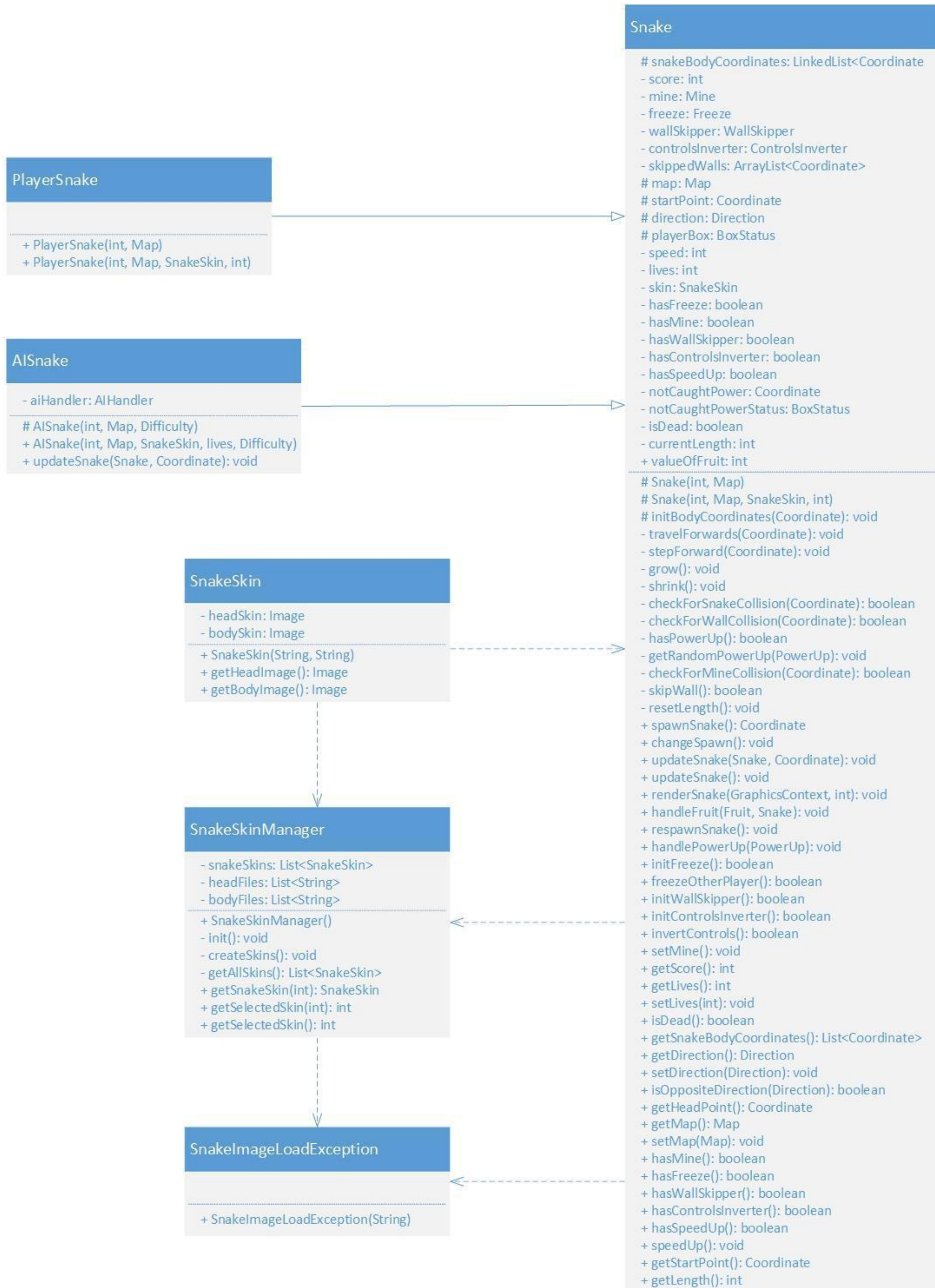


Figure 13: Map Class Diagram

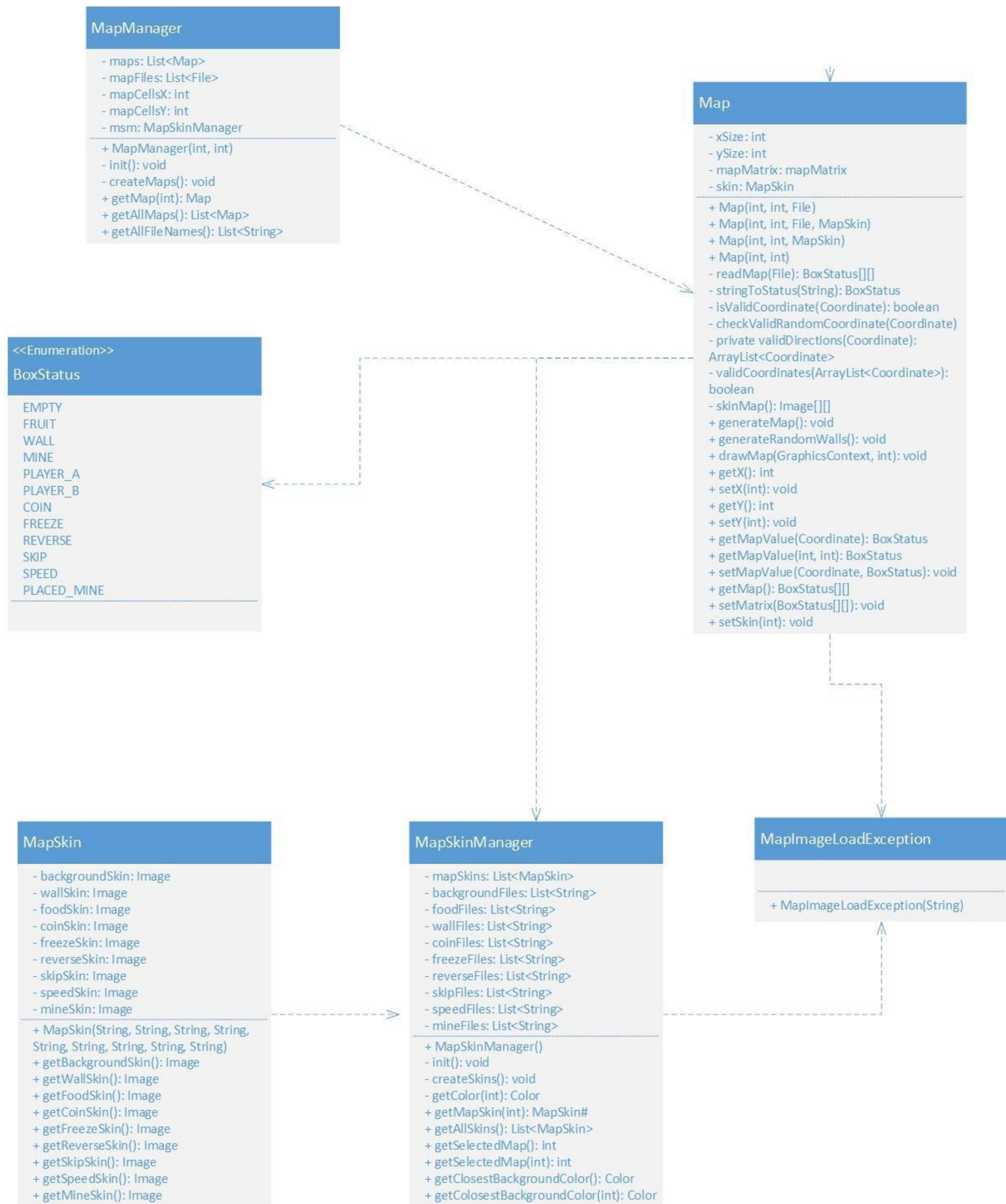


Figure 14: Networking Diagram

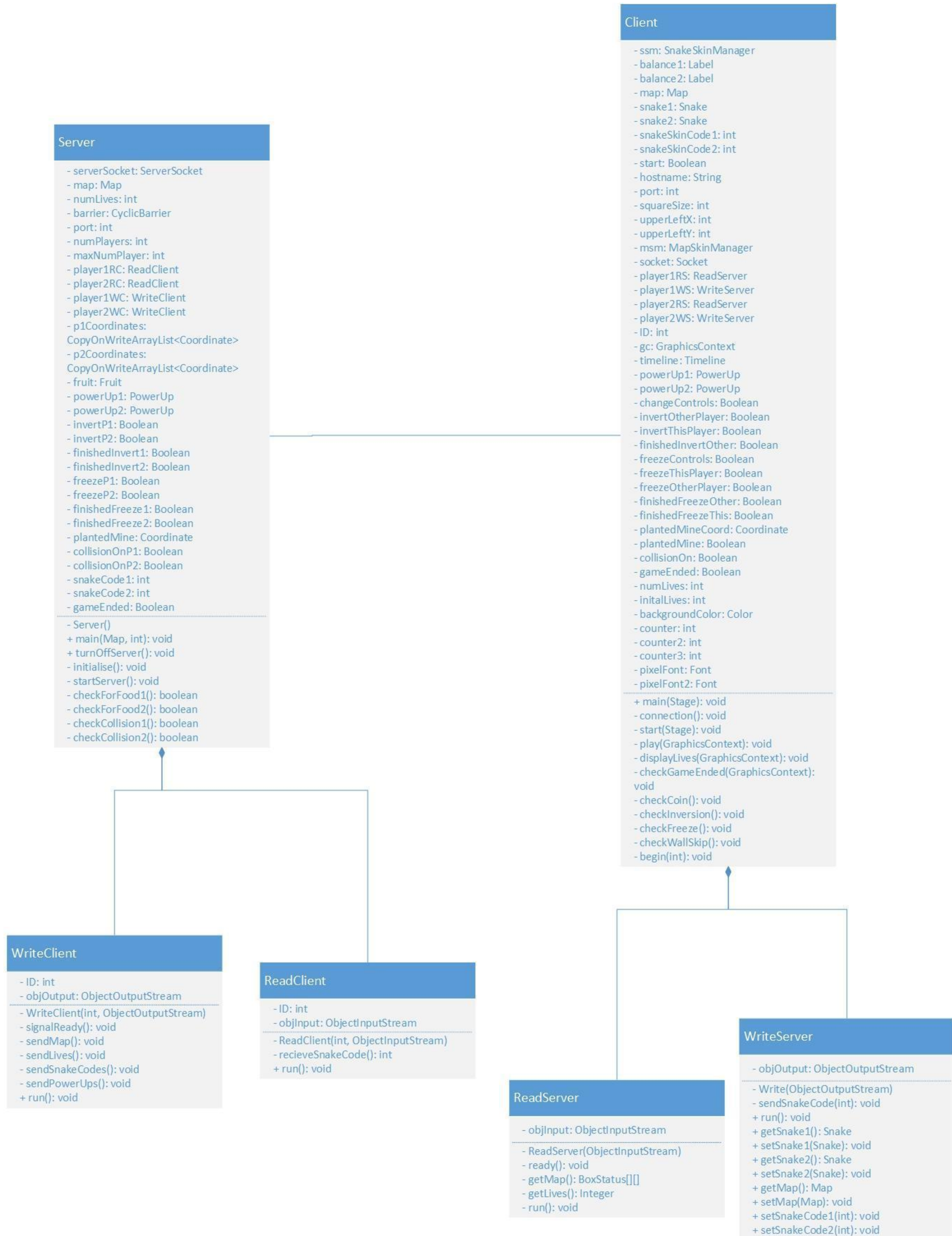


Figure 15: Handlers Class Diagram

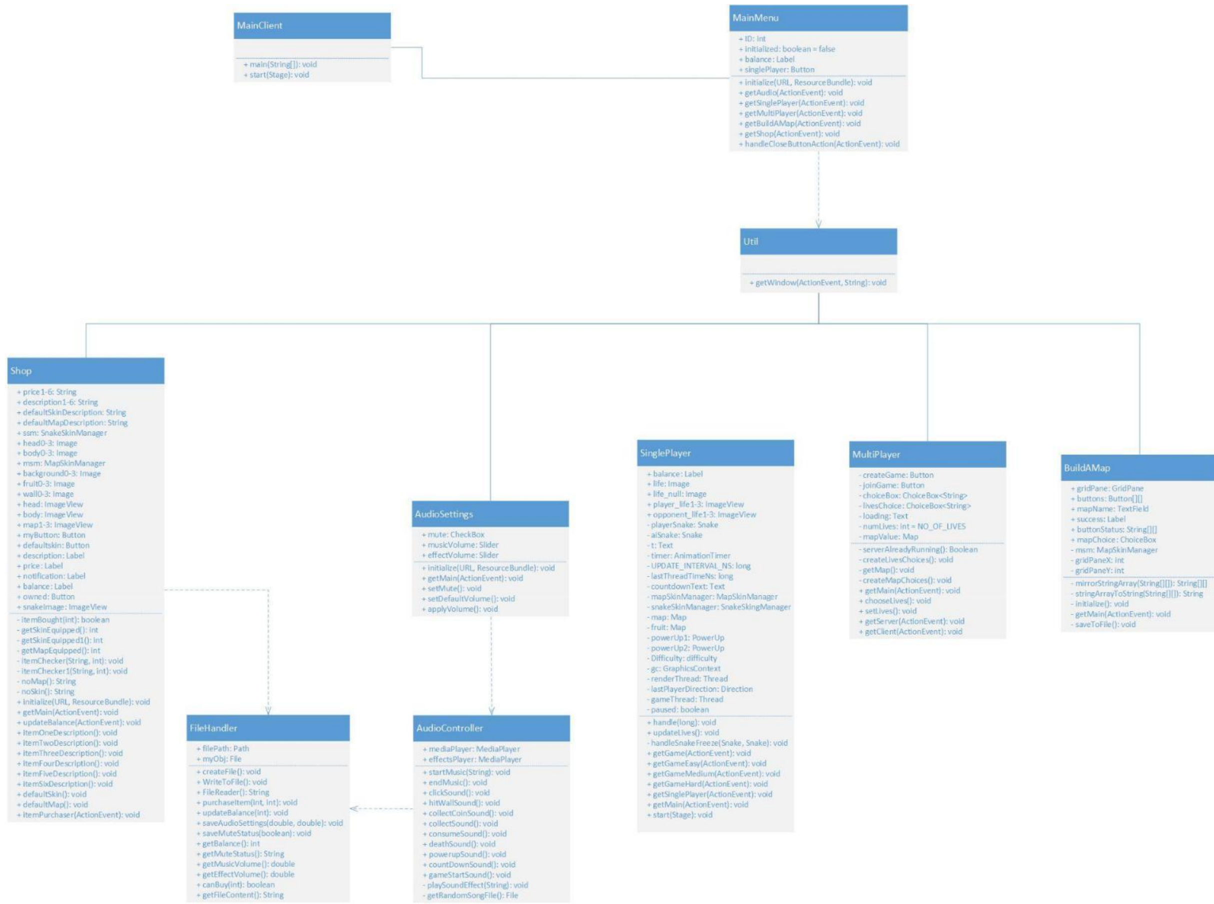
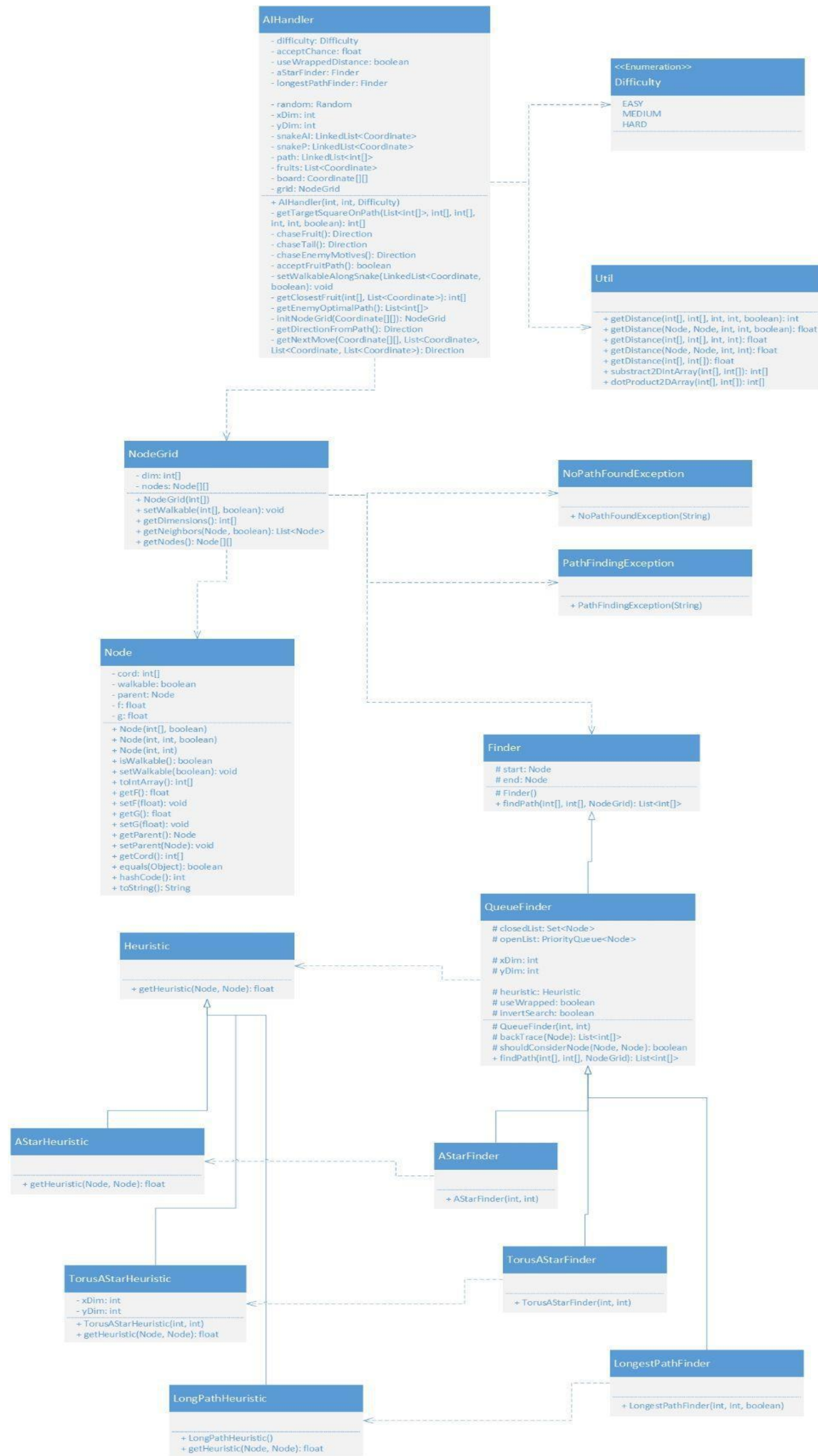


Figure 16: PowerUp Class Diagram

Figure 17: AI Class Diagram



Individual Contributions

Name	Contribution
Daniel Batchford	16.66%
Florian-Andrei Blanaru	16.66%
Mohammed Jaber Alqasemi	16.66%
Dilpreet Kang	16.66%
Yuji Fukuta	16.66%
Rahul Gheewala	16.66%

We believe that these contributions are fair, as we believe that each member submitted an equal amount of work. All team members worked hard on the project throughout the entire term and we believe this is reflected in the final game and report.

Assets and References

- JavaFX 15.0.1 - https://openjfx.io/
- Junit 5.4.2 - https://junit.org/junit5/
- JavaDoc - https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
- Pathfinding Library - https://github.com/danielbatchford/PathFinding
    - Note that this library was produced previously by a team member (Daniel Batchford), although it was heavily modified to accommodate this project
- Music & Sound Effects - Elvira Burlacu - https://www.linkedin.com/in/elvira-burlacu/
    - Permission was verbally granted (Daniel Batchford knows Elvira Burlacu)
- Map Icons (Default Icons) - https://minecraft.fandom.com/wiki/Category:Icons
- Background image - designed by Daniel Batchford
- Intro video - designed by Daniel Batchford
    - Apple texture - https://www.reddit.com/r/Art/comments/huktuk/apple_me_pixel_art_2020/
- Snake and map custom icons - designed by Mohammed Jaber Alqasemi

Gitlab Link

https://git-teaching.cs.bham.ac.uk/mod-team-project-2020/paradroid/-/tree/master