Shaanuka Gunaratne

Analysis

**Introduction:**

I very much enjoyed playing video games as a child. I’ve played many franchises and spent long hours on campaigns, however I’m now an A-level student and like many of my peers, cannot find the time to play long hours due to studying commitments. This has made me turn to smaller game titles such as arcade games.

From Pacman to Dig Dug, there are a vast number of arcade titles available for me to enjoy in my free time. I enjoy playing these titles a lot and play with my friends from school. However, one thing that most of us agree on is that the arcade games we have come to enjoy were released decades ago, hence they contain many missing and outdated features when compared to games of the current generation.

As this is the case, I’ve decided to implement my own modern 2D arcade-style video game that my friends and I can enjoy playing. This will mean incorporating new features of current games such as online communication and high-resolution images. I believe this should make the gameplay more enjoyable when playing these types of smaller games.



Pacman is a classic example of an arcade game. Released in 1980, it is the highest grossing arcade game to date. [[1]](#footnote-2)

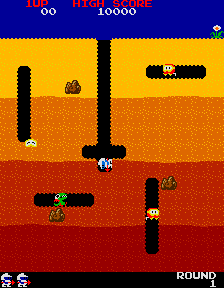
**Analysis of arcade games:**

A classic arcade game I enjoyed playing since my childhood is Dig Dug. This is a 2D single-player maze game where a player is placed in an underground map with a weapon and numerous AI opponents. The aim of the game is to dig your way around the map and eliminate all opponents. You can use the weapon given to you or the objects around you (e.g. falling rocks) as weapons. Completing the goal will make you progress into harder levels with faster pace and more enemies.

Dig Dug was a very popular game when it was released due to its simplistic gameplay but also being very tactical at the same time as it tested the player’s ability to make decisions quickly.



High-score system being implemented for more competitiveness



Objects around the map can be strategically used to benefit the player.

High contrast colours make the game ‘funkier’ and appealing to younger players and retro game lovers even now.

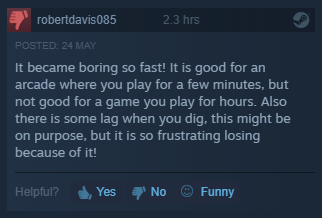
First level of the game[[2]](#footnote-3)

AI enemies are allowed to move in any direction including diagonally, however their pattern of movement is very similar throughout each round and can become repetitive for the player.

**Problems:**

This game was released 1982 so there are some features that the game lacks compared to newer games that have been released in the current decade.

This includes the graphical quality of the game. Although very colourful, there are only a small number of bits encoded in each pixel, which reduces the colour depth.

Another issue is to do with the limited objectives and functionality within the game. Although the game has many levels that get harder as the player progresses, they are still very similar. This recurring aspect may make the game less interesting especially when thinking about a long-term player base.

Number of hours played before the review.

A review taken from the Steam website where the game can be purchased for the PC platform.

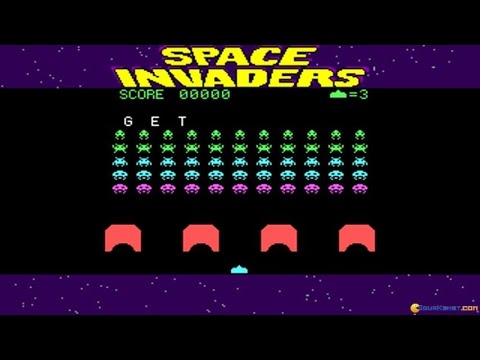
This is proven by several reviews of the game which state that although it’s a good arcade game, it becomes ‘boring’ after continuous play.

As seen, the reviewer on the screenshot[[3]](#footnote-4) had only played the game for 2.3 hours before realising that the gameplay was becoming tedious. This may be due to the fact that this game was originally created to be played at an arcade for a short duration. This is the reason why the PC port has had a lower amount of success compared to the original that was released for arcade machines.

**Analysis of current system:**

As mentioned previously when it comes to arcade games, I particularly like shoot ‘em ups. There’s a vast array of unique games in this genre, but my favourite is the game ‘Space Invaders’.

Space Invaders:

Space invaders is a classic shooter game released in 1978.The game has an easy to grasp concept where a player controls a laser canon in the horizontal plane and their goal is to shoot aliens that are loaded into the arena. Defeating all the enemy aliens results in a larger wave of aliens spawning as the player progressed to the next round. This will repeat until the player is defeated and they will be given a score depending on how well they performed. This game is iconic as it paved the way for the shoot-em up subgenre of games.

Limited life system because the game is infinite.

Gameplay of a version of Space invaders[[4]](#footnote-5)

Simple movement controls in the x-axis making the game easy to pick up for all age groups.

Simple geometric shapes make rendering easier on almost any supported system.

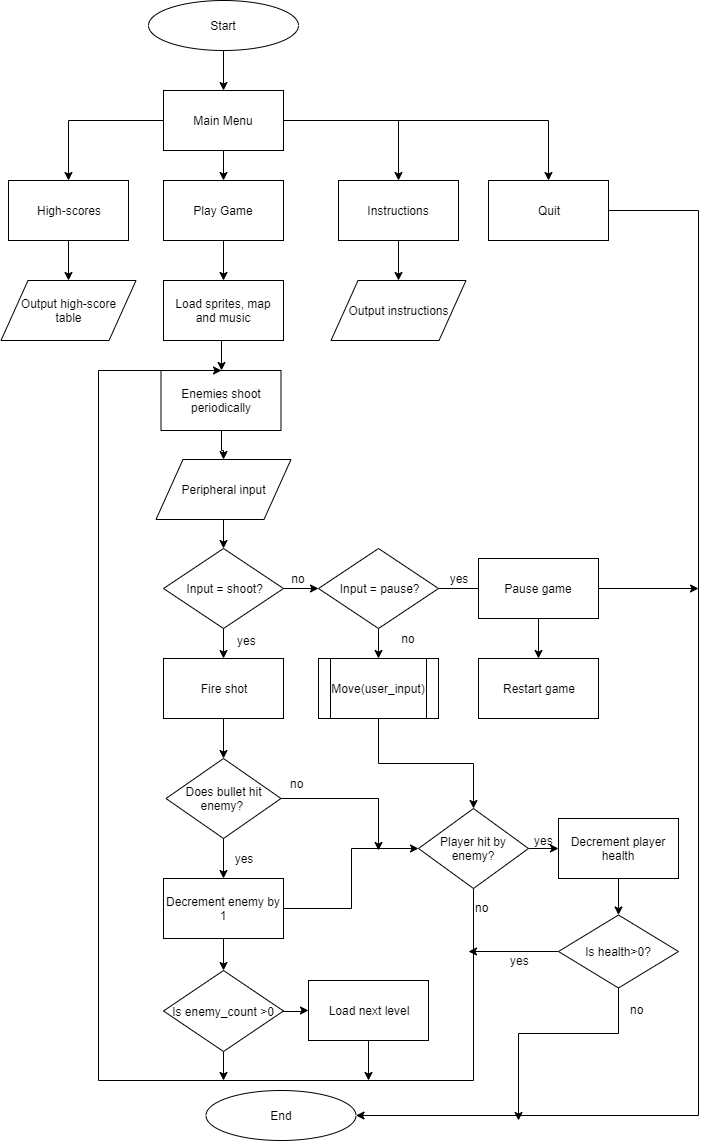
**Positives:**

* Simple controls and objective made it appropriate for a range of age groups as it is easy to learn and good for small gameplay sessions.
* Very colourful visuals made it more appealing, especially for younger audiences.
* The sound effects were revolutionary. It provided players with direct feedback to their actions and interacted emotionally with the player using music.
* Not very graphically demanding, so it can run on many systems at a high frame rate which is beneficial in a fast-paced game.
* The game generates a score depending on the player’s performance. It also saves high-scores, therefore adding a competitive aspect to the single-player game.

**Negatives:**

****Similar to Dig Dug, this game lacks features that modern systems have implemented (e.g. online multiplayer/co-op modes or advanced AI (Artificial Intelligence) enemies that make tactical decisions) to keep players interested in the game. The game’s repetitive structure has a negative impact on its replay value.

These screenshots[[5]](#footnote-6) above are taken from the same gameplay session. The one on the left is from the start of the game with a score of 0, and the other is at a score of 9250 which is a relatively high score. It’s apparent that there aren’t many visual differences between the two. As discussed above, this lack of variety during gameplay reduces the replay value and may result in players losing interest in the game.

**Analysis of similar systems:**

**Key**

Flowline

Decision

Process

Start/End

Subroutine

Input/output

yes

no

Flow Diagram of Current Solution:

I’ve constructed a flowchart of Space Invaders so that I could better understand the mechanics and logic behind the game. This will be helpful during the implementation of my solution due to it having similar aspects to this program.

I analysed several modern 2D arcade-like games. Using thorough research, online reviews and experience playing some of these games, I found 2 games released in this decade that have used an arcade-style approach but still managed to have modern features that made them successful amongst many other games of the current generation.

Duck Game:

Duck game is a fully functional real-time multiplayer arena shooter. You are given a character in the form of a duck which you can customise to your liking. Then you connect to either a LAN or WAN server and compete against other players. The basic controls are simple and consist of directional movement and the trigger key for shooting a weapon. However, there are special functions that can be executed (special character moves) with specific key combinations.



Screenshot of a multiplayer round in the game[[6]](#footnote-7)

Multiple random collectable items scattered around the arena. This reduces repetitiveness in each game, as there could be a number of various weapons or other consumables in different rounds.

Bright, high-contrast graphics. Increases the appeal especially to a younger audience.

The aim of the game is to eliminate all of the other players on the map. This follows a classic 2D arcade-style approach, where there is no storyline or complex progression system, but instead there’s a vibrant and funky game style, with larger arenas through the implementation of a scrolling background. After analysing the game, I realised there were also some drawbacks that come with this arcade shooter.

|  |  |
| --- | --- |
| Positives | Negatives |
| A fully functional LAN and WAN real-time multiplayer battle system, so players can compete with one another which adds a competitive element to the game. | Only a maximum of 4 players in an arena. However, this can be a measure to reduce latency since multiple players can put a toll on the server. |
| Includes multiple arenas, a variety of weapons and special moves. Since the game is not reliant on a narrative, these features keep the game fresh for longer as it adds more depth and excitement to the game. | Instructions manual is limited to only the basic controls. Using different weapons requires different key combinations. This may make the game more exciting as you have to figure out the combinations yourself, but it may also disadvantage inexperienced players. |
| Makes use of a scrolling background to increase the size of the arena without scaling the images to be smaller. | Some potential users may not find the 2D style of graphics appealing as many modern games have more realistic and demanding graphics. |
| Most modern systems should be able to run the game at a high frame rate since the 2D visuals are not very demanding on hardware. | The number of multiplayer sessions available may vary depending on the time. This is due to the lack of private matches and the game not having a large online community. |

I realised I would need to find a method of avoiding any similar drawbacks when developing my proposed solution. There are also some positive features about these products that I would like to implement in my own solution.

**Features that can be implemented in proposed solution:**

* Continuously changing animations depending on player actions. A data structure such as a queue can be used to implement sequences of animations in a specific order.
* Collectable items around the map (e.g. health packs, power-ups). This should add more variety to the gameplay.
* Have an Instructions page detailing the functions of each key combination.
* Bright arcade-style 2D graphics as it would be more appealing for the target audience (arcade gamers).

Bug Butcher:

Bug Butcher is a 2D single-player/co-op shoot em up game where the player’s main goal is to destroy waves of mutant bugs that are above them. This game incorporates a unique style for a shoot em where the enemies always attack the player from above. This means that the player can only shoot them through the vertical axis.

Continuous player feedback on progress and achievements increases their confidence to continue playing.

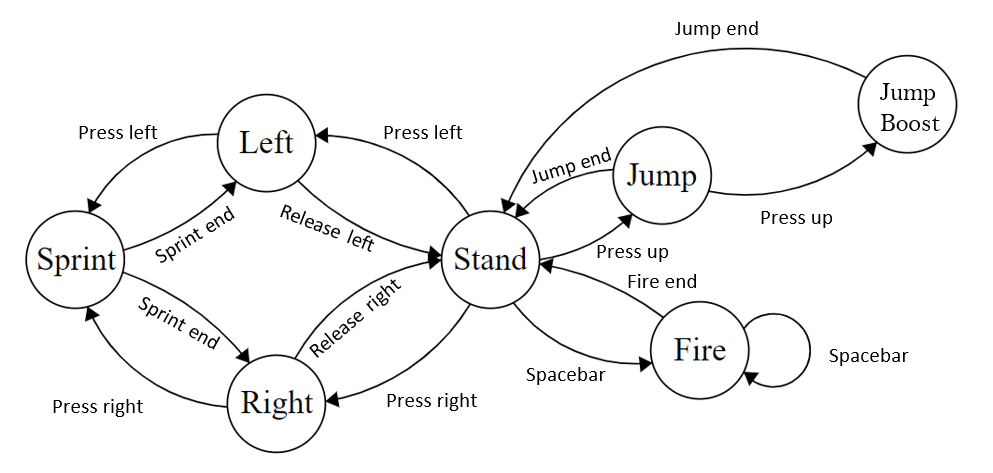
Bright flashy visuals to draw the attention of the player.

As seen in this image, the player is firing multiple projectiles through the vertical plane. This unique feature alongside the bright visual effects creates an exciting environment for the player.

This game also features a co-op mode which allows a maximum of 2 players to play in the same arena. However, it is an offline local co-op therefore the other player will require another peripheral device or they will have to share a keyboard when playing this mode.

|  |  |
| --- | --- |
| Positives | Negatives |
| Very bright and colourful visuals make the game feel more vibrant and exciting. | No online co-op mode, so the player will have to share peripherals or connect multiple controllers to play. |
| Dynamic sound effects add pace to the game and can make kills and combo moves feel more satisfying. | Some players may not find the unorthodox orientation of having to focus on enemies in a vertical plane appealing. |
| Continuous player and game feedback and interaction will keep the player more focused on the game and will give them confidence to continue playing. | Cannot add friends or see their profiles in the game as it does not have an online component. This makes the game less competitive compared to games with this feature. |
| Simple, fun gameplay mechanics and animation enhances the arcade style of the game and will appeal to arcade enthusiasts. | Users may be limited to playing only the single player mode if they do not have access to another peripheral device (e.g. an extra controller). |

Finite State Machine for Bug Butcher control scheme:



The controls in the two similar systems are very simple and easy to learn. I constructed a finite state diagram by analysing the control scheme for Bug Butcher, in which the same set of controls are used throughout the whole game. This expresses the simplicity of the game and hence why many arcade gamers may find it appealing as it’s very good for short gameplay sessions, where the player doesn’t need to worry about any complicated elements such as a large set of moves and controls.

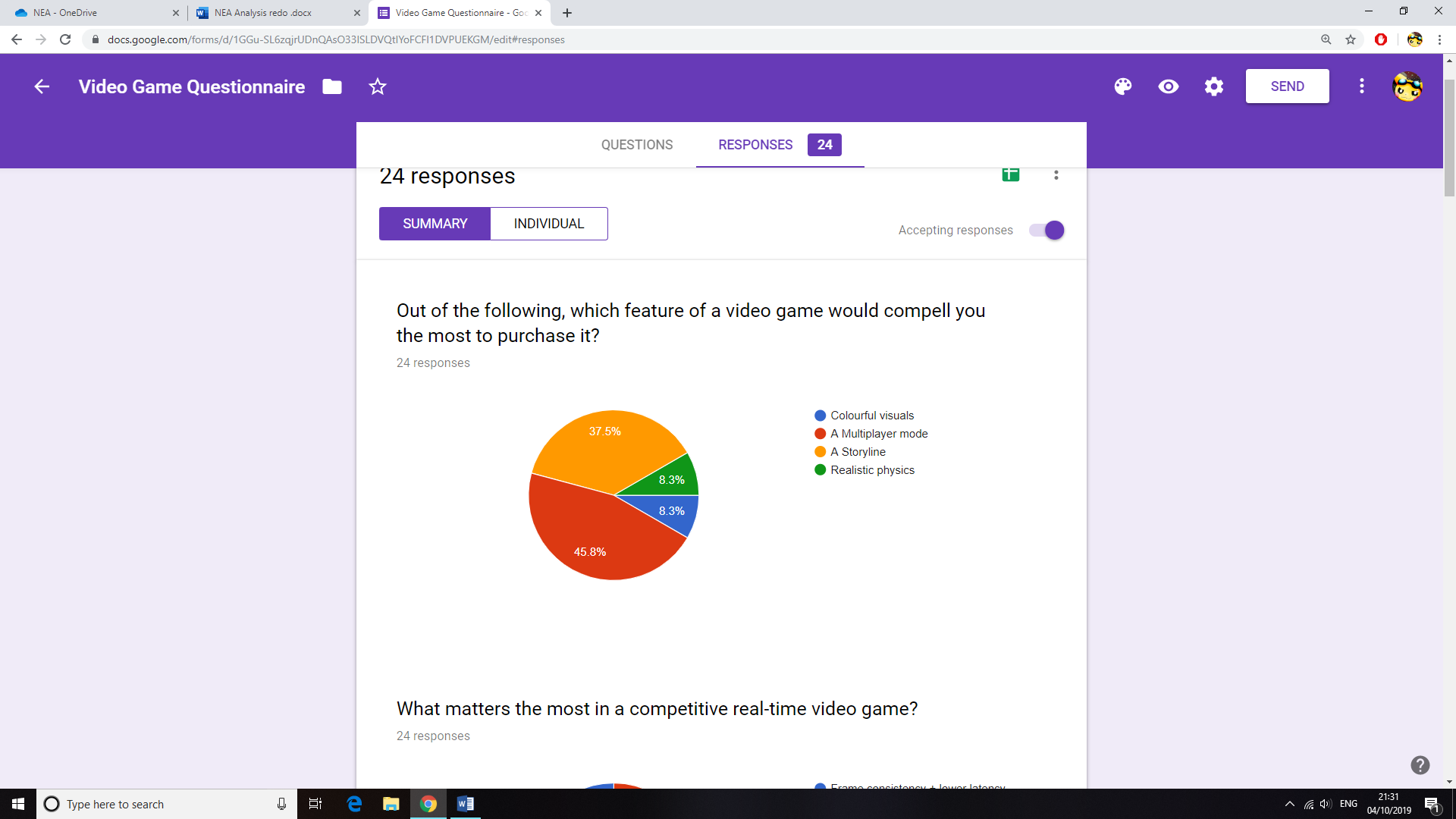
A similar layout to the above can be used within my implementation to make the game more user-friendly and easier to learn for all age groups.

From the analysis of this game, these are some features I found that I could include in my solution to make it better.

**Features that can be implemented in proposed solution:**

* Make a simple control scheme so it’s easier for players of all age groups to play without confusion.
* Online multiplayer such as a LAN or WAN connection so users can play against and challenge their friends.
* A locally stored high-scores table with a point system for extra competitiveness that can increase the replay value of the game.
* Dynamic sound effects that change depending on the events occurring in the game for added tension and excitement especially near the end of a round.
* Fun arcade-style animations and gameplay mechanics.

**Opinions from target group: (put screenshot of form)**

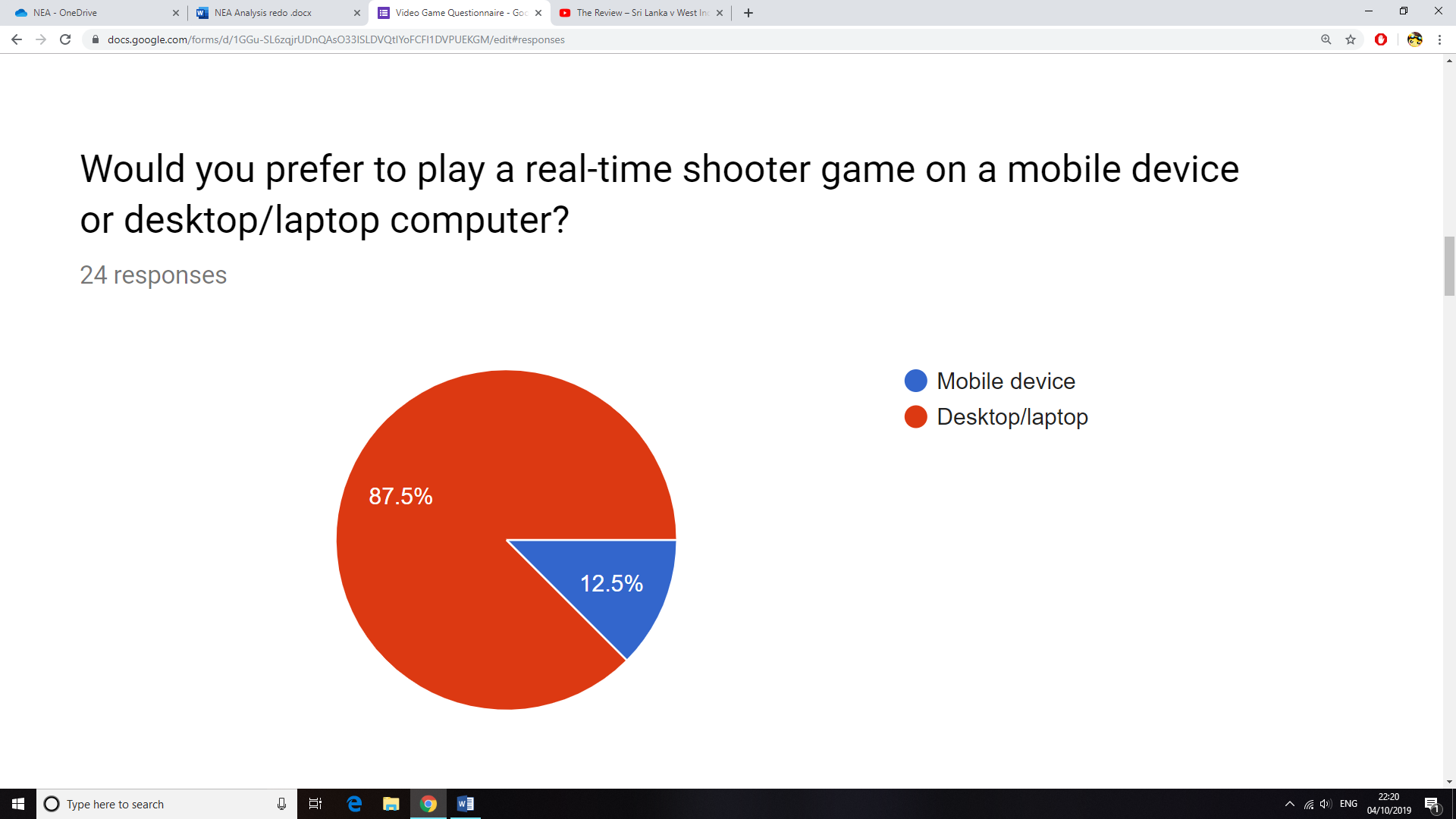
I conducted an online survey for my peers in my computer science class who mostly have had experience playing different types of video games. These responses allowed me to gain more ideas and showed me what aspects of the game I should focus on during development.

Like the similar systems I analysed, I thought of creating a multiplayer mode for my game as it is a predominant feature of many games released in recent years. This is clearly shown by the pie chart (above) generated from my survey. The main point of interest seems to be in the multiplayer mode, followed by the storyline of the game. Although it is very clear that a storyline is important when developing a lot of games, it does not apply to mine since my project is for more casual players with less time on their hands to invest in a continuous storyline.

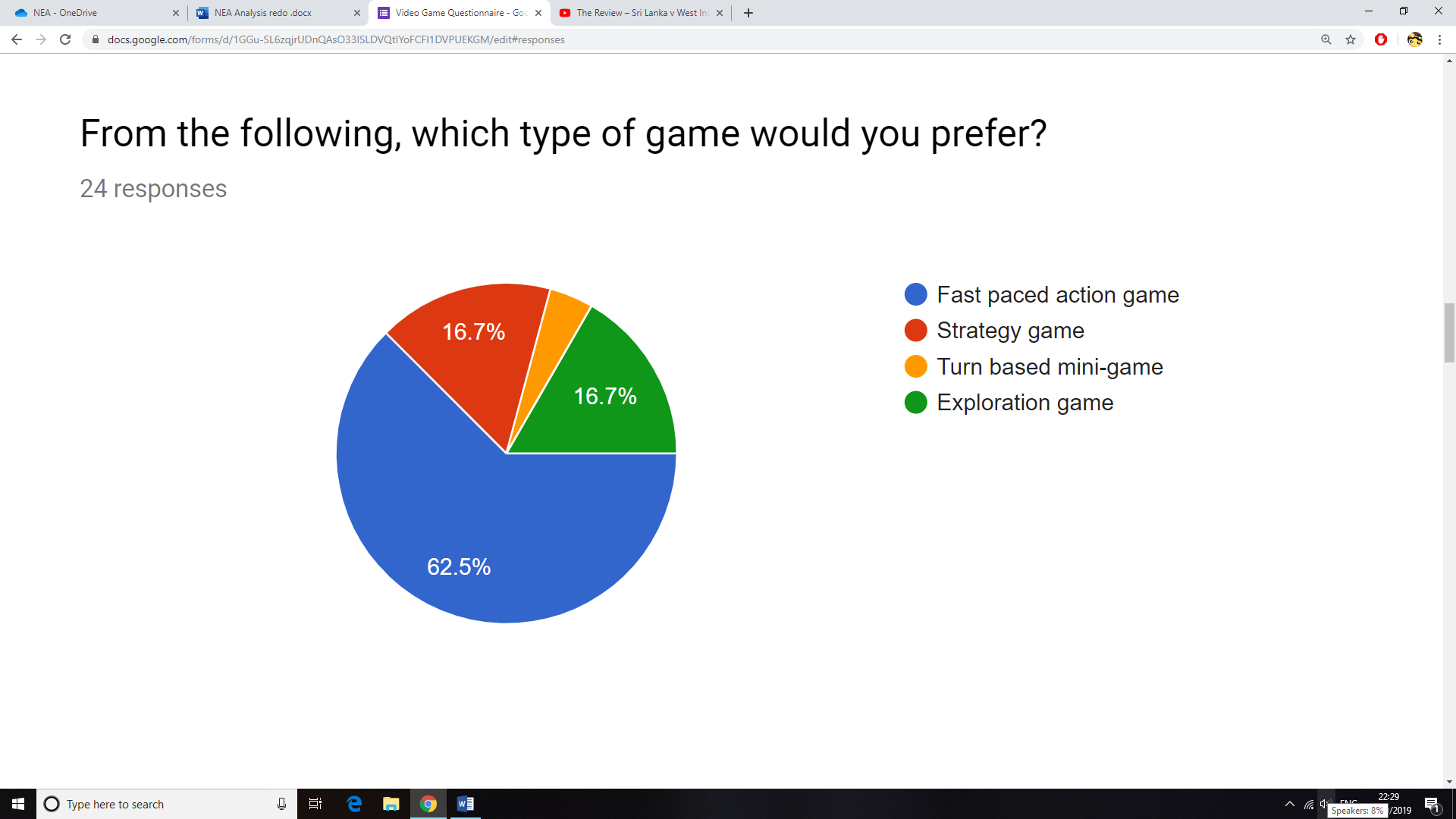


One problem I will need to test and prevent is latency within the game. Since I’ve decided to program a multiplayer game, I will need to make sure the network and server side of the game are stable. This will mean implementing measures (e.g. slowing down the game to reduce the effect of lag) to make sure the game will run with minimal latency.

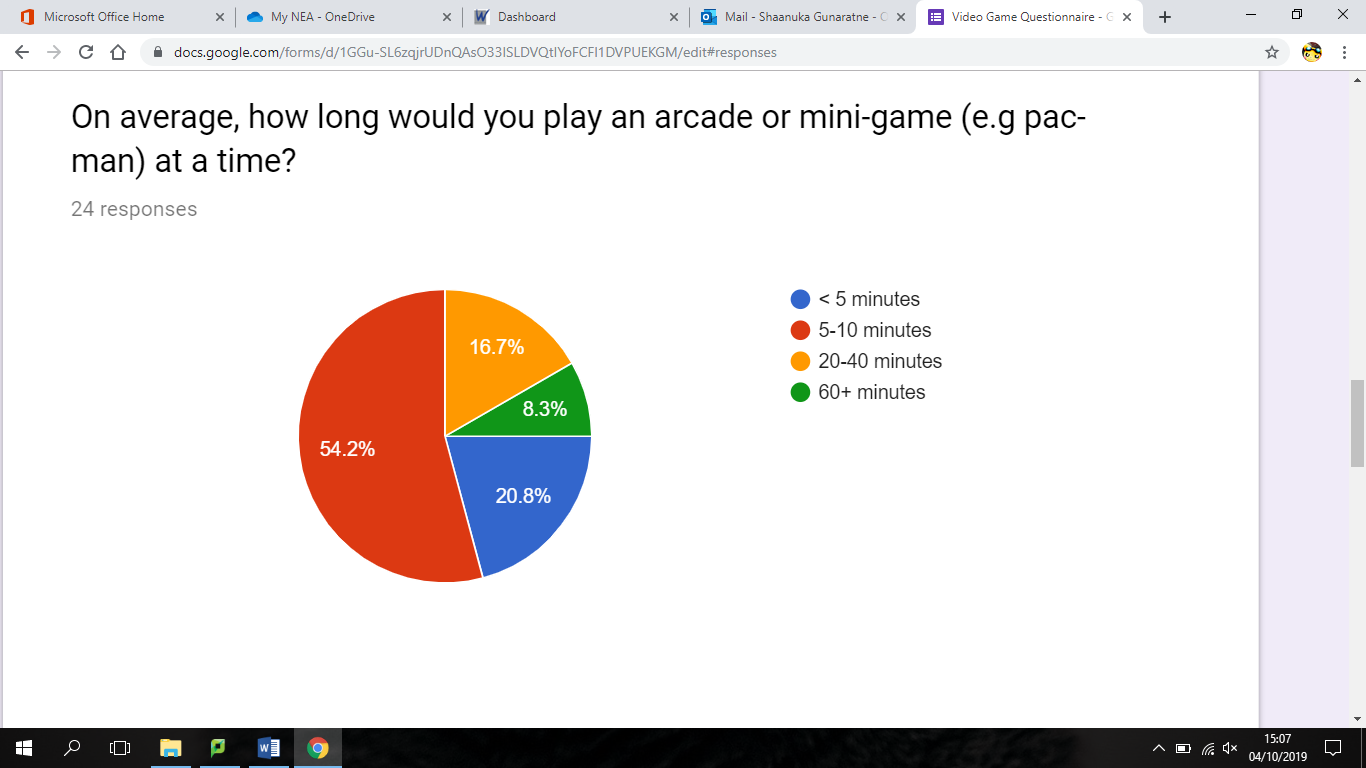
If latency is not an issue, then another problem is framerate inconsistency during gameplay. This can be particularly annoying especially in a competitive scenario since the pace of the game will change due to frame drops. I will have to test the game on my system and several other systems to find a stable framerate to cap the game at. Furthermore, I will have to optimise my algorithms to a low time complexity, hence reducing the use of hardware resources.



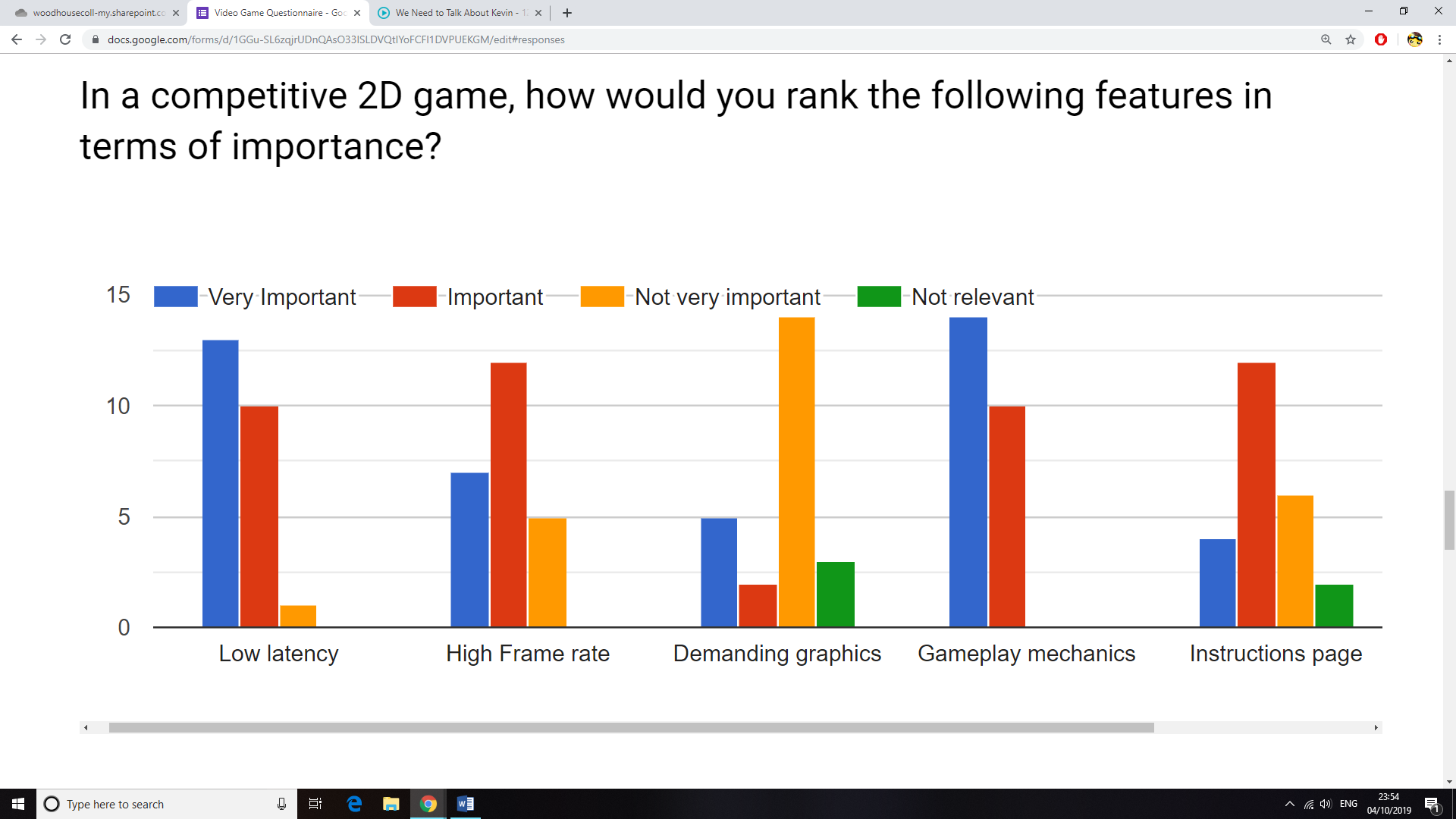
The majority of votes were towards desktop or laptop computers, because they allow for a larger display with a more flexible control scheme due to external peripheral controls. Although there are many mobile arcade games, a lot of gamers (as seen in the chart) will prefer to play a real-time competitive game on their desktop computers, hence I will create my game for a desktop/laptop computer.



The majority has selected action games as their preferred genre from the options above. I will need to implement features to make my shooter game fast paced, possibly with the use of SFX and other methods such as randomly appearing collectable items which should increase variety and uncertainty of winning the game.



Arcade games are mostly played in small intervals. From the survey, it’s clear that a player would usually play such a game for 5 to 10 minutes at a time. For my implementation, I could add a countdown timer for each match that can be altered by the player, therefore the user would know how long they will be playing for. This feature would be very useful in a multiplayer scenario where it would not be ideal to allow the player to pause the game, therefore adding a controllable time limitation would make it more convenient for the user.



It was crucial that I knew which parts of the project to focus on during development. From the rankings of features I gathered from my survey, I’ve established a hierarchical order of features that are most important in my game.

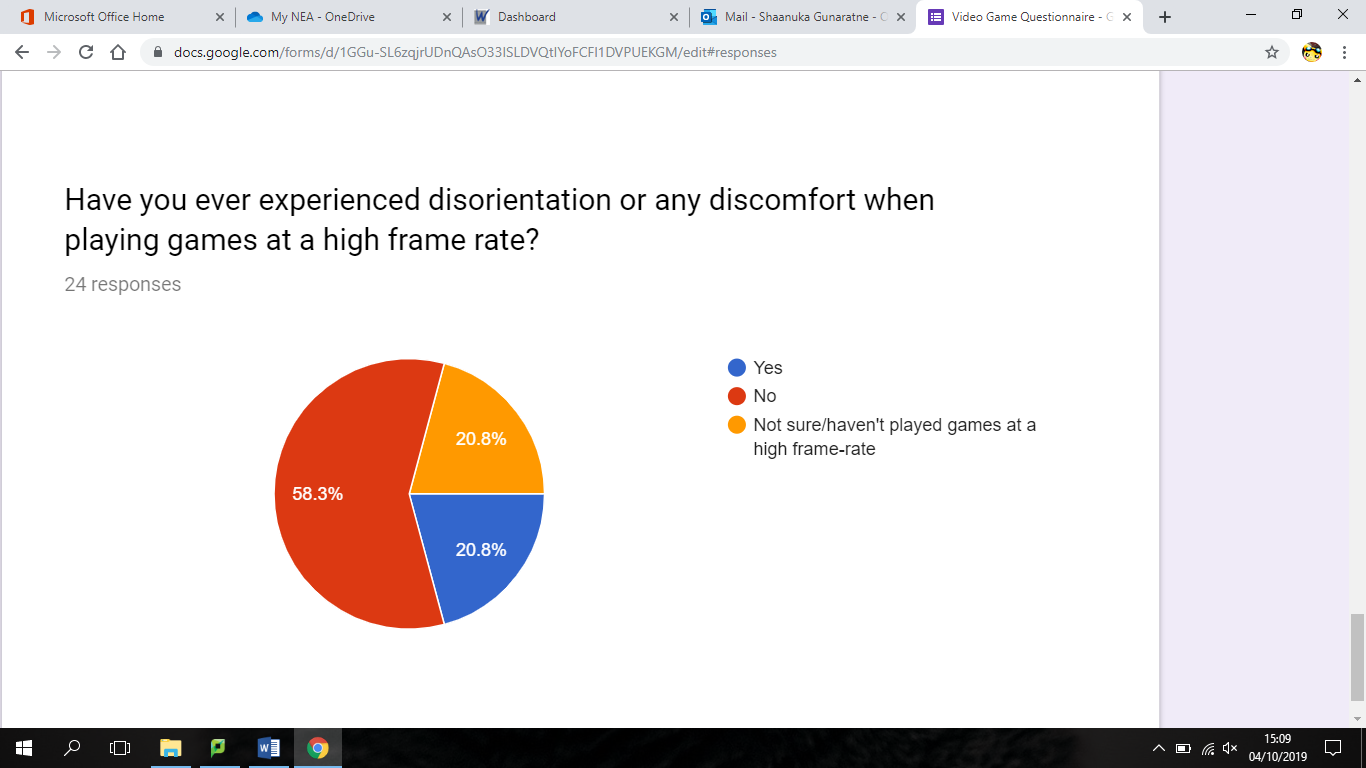
**Gameplay mechanics** understandably having the most votes suggests that my audience would like me to focus on the core elements of gameplay. This means I will need to refine certain aspects of the game such as the ability to jump and fall due to gravity, the effect of collisions between sprites or projectiles and sprites, and the other similar functions.

**Low latency** and **high frame rate** are again voted as important due to their effect on consistent playability of the game. There will need to be measures taken to prevent frame inconsistencies and latency, which I have discussed above.

**Special effects** have also been noted as important. This may be due to the fact that both the visual and sound effects work together in most games to control the pace of the game. There are some mixed views in the rating for this feature, therefore I will need to talk to one of my peers to get further details.

**An instructions page** has also been deemed of importance. In oppose to an in-game tutorial during the first minutes of gameplay, many gamers do like to have a detailed instructions page so that they can re-visit it if any information if forgotten. I will make sure to add this to my final version of the game.

Finally, **demanding graphics** has not been a popular choice, therefore I will not focus on this aspect of the game as it’s clear that most players care more about the gameplay mechanics when it comes to playing a fast-paced action game.

From the responses I got for the additional features that could be implemented into my solution, many suggested the addition of a point system and the ability to save profiles. I have already considered creating a scoring system, but I can increase the competitiveness with a system that stores the points such as a highscores table for each player. This can be done via a login system so that a certain player can be identified and their scores can be tallied and stored.

Finally, It has come to my attention that some users do feel motion sickness and are prone to headaches when playing certain types of games[[7]](#footnote-8). As seen in the chart above, several peers of mine admitted to this. However, from research I have conducted I’ve learnt that the main issue is in first person shooter games. This will not be as much of a problem with my game as it will be in a 2D side-view perspective. However, I will need to cap the game at a suitable framerate so that the user would feel comfortable when playing for longer durations.

**Detailed requirements for my target audience:**

After conducting the survey, I have decided to create a competitive multiplayer shooter game. My final goal is to make an arcade shooter game with features from modern titles of similar genres. To gather more information for my project I contacted one of my peers, David, who is an avid action gamer. I told him the specifics of my project and asked several questions that would help me with my project.

1. **Are there any features in a competitive multiplayer action game that specifically stand compared to other video games?**

*“To me the main thing that differentiates a multiplayer action game from any other game is its competitive nature. I enjoy playing a lot of multiplayer games such as ‘Street Fighter’ and ‘CSGO’. The main reason being the fact that I can play with my friends and compete in fast paced matches. I also like the ranking system in CSGO where prestige is gained from each successful match. But the main focus when playing is the point system and trying to get to the top of the leader board in each game.”*

It’s clear from David’s response that I should focus on the competitive aspect of the game. One of his main points being a leader board and point system increasing the competitiveness of the match. This was also requested on my initial survey, so I think it will be a good idea to implement either a high-score table or a player leader-board in my solution so that the players can gauge their performance and compare it whilst playing against each other.

1. **Do you think there will be a benefit of adding a login system and providing a player with their own account?**

*“Most modern games I’ve played allowed me to make my own account. It adds to the competitive aspect of the game since you can save your high scores and achievements. However, I wouldn’t mind the lack of a login system for smaller titles because I won’t invest as much time playing them.”*

A popular request from my survey was a high-score system. For this to work, I thought of creating a login system for each player so that their scores can be saved individually. From David’s response, I realise that making a login system would add to my game in terms of extra functionality. If I’m to make the scoring system, I’ll need to find a format to store player high scores and login details either using a CSV file format or database. However, CSV files can be easily be read and modified, leaving the login details and high-scores prone to tampering. Such issues can be avoided with the use of a database. This method allows the access rights to the data to be modified so that they cannot be accessed by an unauthorised user.

1. **For my survey, the majority of the class voted against high-resolution graphics. I am assuming it may be because of hardware requirements and issues with frame consistency. What do you think of this?**

*“I love playing graphic intensive games like ‘The Witcher 3’ and ‘Forza Horizon 4’, but neither of these are fast-paced action shooters. For me even with the recommended hardware requirements to run a graphically intensive game, when playing it at a faster pace the graphical quality is less noticeable as my main focus will be on the current events in the game whilst trying to complete the objective. Another reason that it will be less important is the fact that your game is more casual and for smaller gameplay sessions, therefore I don’t think the visuals should be very demanding.”*

Since I am creating this game for a casual player, I understand David’s response saying that there is no need for high-end graphics. I think I should shift my focus to the mechanics of the game and making sure that the multiplayer functionality is stable instead of opting for high resolution textures and enhancing…

1. **If the graphics should not be a concern, what are your thoughts on visual and sound effects during specific moments in the game?**

*“The SFX are definitely a bonus and adds to the pace of the game, especially for increasing the tension such as in a 1v1 situation. For me, these effects add an extra touch to make a game seem complete. If you take Pacman for example, the game has a fairly basic goal and the visuals are bright and colourful but not nearly at the standard that graphics are at right now. The only things that still stands out to me from that game is the sound effects; the change in music from when your Pac-man is chasing the ghosts, compared to when he is being chased by them adds to the tension and makes the game a lot more exciting in my opinion.”*

From his reply, I can understand that sound effects and certain visual effects can also be used to change the pace of the game. The basic sound effects of my shooter game will consist of sounds such as shots being fired, getting hit by a projectile, and the sounds for player movement. I will need to add special sound effects that change depending on the events of a game. For example, if one player is low on health, I could increase the pace of the music output for that specific client, therefore the sound effects will be different for each player. For this to be possible I will need to create an automated sound effect function that can identify each player and their attributes and provide the relevant output accordingly.

1. **Finally, what components would you personally like the final program to consist of?**

***“****For me, I would like it if the final product has 3 main components. A player should be introduced to a main menu screen at start-up. If you decide to add a login screen, then the menu should be loaded after the player logs in. Then he or she should have the option to play the game or quit. Another feature I see in many games in an instructions page. I would expect this in the main menu alongside the play and quit options. If you are deciding to create an online multiplayer either with a WAN or LAN network, you should allow the user to set up or join a server. Finally, the main focus of the program should be the game loop, where all the processes and events of the game during gameplay will be controlled and outputted to the player.”*

As an experienced arcade gamer, David’s listed out the main components of the game that he would like me to focus on when programming my solution, those being the main menu, the server and the main game itself. From this decomposition, I think I should separate these components into independent subroutines and work on them individually on separate files during my implementation stage, as they are separate components of the final game and it will allow me to focus on each of them one at a time.

**Problems to tackle with proposed solution:**

Before starting the development process, there are several potential problems that will need to be considered so that measures can be implemented to tackle them.

**Synchronised frames for both clients:**

My multiplayer system will run on a LAN based system so there should be a lower amount of latency compared to when connecting to a WAN server. However, there can still be network slowdowns present, therefore I will need to control the pace of the game and make sure the frame rate of the output is compatible with the networking speeds so video that video output is consistent and synchronised between the client computers.

**Securing user information:**

As a result of adding a high-scores table and saving the player’s best scores, a login system is to be implemented. The user’s authentication details and high-scores will need to stored securely to prevent account theft or tampering with the scores. This is not a huge issue as the game is not designed for a player that will invest long hours on it and is mostly for quick and fun sessions. However, if measures are to be implemented, the data should be stored in a database format or similar. This will allow the administrator to control access rights to the information that is stored within the database, hence it will be fairly secure.

**Time constraint:**

There will be a time limit for developing this project, alongside the fact that I have other academic commitments and extracurricular activities that are ongoing throughout the year. This will force me to focus on the main aspect of the game (the real-time multiplayer battle system) rather than the other aspects such as the database for the user details. This has made me understand that I will have to prioritise certain parts of the project more, such as the main multiplayer battle system over the database for the player data. Although I’m going to focus on the main aspects of the game, I will still try my best to implement all of the features in my final game.

**Hardware and software constraints:**

I have chosen not to make the graphics very demanding for my game and will also try to make my algorithms efficient so that they do not take up a significant proportion of hardware resources. Most modern computer systems should be able to run the game but I will benchmark it on my own system to test its performance. The basic specifications for my current computer system:

**Intel core i5 … processor**

**Integrated Intel HD 4400 graphics processor**

**8GB DDR4 RAM**

**64-bit Windows 10 Home operating system**

Additionally, I’ve decided to program the game on Python using the Pygame library. Python is an interpreted language, so a Python interpreter with a reference to Pygame will need to be installed on the computer system before running the game.

**Knowledge constraint:**

This is the first time I am developing a video game using an external game library, therefore it will take up a moderate amount of time to learn to use and incorporate the Pygame library in my program. Furthermore, I will be using sockets and networking to add multiplayer functionality. I will also have to learn how the software will interact with the server and client and learn the commands that come with the Sockets library.

I will have to create a few prototypes to familiarise myself with the game engine before I start to implement my final project.

Proposed Solution

From the analysis on video games that I conducted, I have planned to create a real-time multiplayer arena shooter running on a local area network.

The player will be welcomed by a main menu screen which will include an option to view the instructions, quit or play the game. Then they can either chose to create or join a server, where they will connect with another player on the same network playing the game. The game will start and a countdown will be displayed at the top of the screen. The players will each be given a blaster and the aim of the game will be to eliminate the other player. There will be features such as collectable items and various objects on the map. This is a simple objective and it’s similar to the retro gaming style of the current solution (Space Invaders). Every hit taken will reduce the player’s health. There will also be collectible items on the arena which will provide the players with random resources. The last player to survive, or the player with the most health after the time runs out will win. The user’s high-scores will be stored in a database and they will be required to login at the start of the game.

**Planning my implementation:**

I’ve decided to use the Python 3 programming language for the implementation for several reasons, one of them is the fact that it’s the programming language I am most familiar with. This choice should allow me to spend more time solving logical problems within my program rather than learning the syntax. The other factors affecting my decision are listed below.

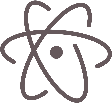
**Advantages of Python 3**

* Python is a very popular programming language that comes with a lot of online resources. By 2019, it was used by more than 32% of the programming community.[[8]](#footnote-9) Owing to this fact, there is a large community of Python programmers providing a lot of helpful resources that I could use to solve problems within my program and to make it better.
* Python’s syntax is known for its closer resemblance to modern English compared to other programming languages like C. It is the same for the libraries such as Pygame and Socket which I am not very familiar with as of now. This should make it easier for me to learn and memorise new commands that I will need when programming my solution.
* Python also has a very compact syntax compared to many other languages[[9]](#footnote-10). Due to this fact, the development time for my project should be lower when using Python in comparison to languages with a less compact syntax as it will take longer to write the source code.

**Disadvantages of Python 3**

Python is an interpreted programming language. This means that it uses an interpreter as its translator to translate the high-level source code into machine code. Unlike compilers, interpreters translate source code line by line every time the program is run, whereas compilers translate the whole program at once into a single executable file that is independent of the source code. This causes several limitations for the program.

* One limitation is that a version of Python 3 will need to be installed on the client’s computer and the source code will need to be present for them to run the program. This is because the interpreter needs to translate from the source every time the program is run.
* The previous point raises a vulnerability as the source code will need to be distributed for the program to work. This will allow the original code to be modified by the user. This is may be a problem as the users can modify the source code for unfair advantages.
* Finally, the fact that Python is interpreted may cause performance issues as interpreters are known to be limited in terms of performance as a result of the line by line translation process during the running of the program. It will be vital for me to optimise the game and not make it too demanding for the system to reduce any performance issues as a result of this.



**Atom Source Code Editor**

I decided to use Atom, a free source-code editor to develop my program as it has several advantages over IDLE (the default development environment that comes with Python).

Atom allows multiple files to be loaded into one window with multiple tabs. This will be helpful when comparing code from multiple files and will allow me to have every file in my project to be opened simultaneously in one window, making it faster to edit code.

Furthermore, Atom provides the user access to thousands of packages made by its community, providing various functions such as multiple debuggers to choose from. This is extremely useful due to features such as debugging allowing a programmer to troubleshoot their program and to detect any errors within a given region. I think this would be useful for me since I am creating an abstract environment for my game, which will include multiple variables and data structures that I’ll need to keep track of. Debugging will allow me to assess each component of my code to make sure everything is consistent and allow me to spot any logic errors in my program.

Atom also allows the integration of a Github account. Github is a platform for software developers that allows them to manage their code. I can link my Github repository onto atom so that my code will automatically be updated and saved.

It also has small features that will save time and make the development process easier, such as auto-correcting syntax with suggested inputs and an auto-save feature which saves changes periodically and when terminated. This makes it more reliable and more developer friendly compared to IDLE

**Libraries:**

**Pygame**

I’ve decided to use the Pygame library as my main graphical user interface and game engine for developing my program. There were other choices that I researched such as Blender, Pyglet and Panda 3D, but Pygame has several advantages over other game engines available on Python 3.

Pygame has a very large community with a lot of support such as active forums and tutorials compared to other engines with similar performance. This will make it easier to learn the syntax I will need for my program to work and will allow me to use the library to its full potential.

Unlike some of the engines mentioned such as Blender and Panda 3D, Pygame provides the programmer with a simple 2D environment where 2D images can easily be imported from many file types without the need to worry about a 3rd dimension. My game will run on 2D graphics; this will advantageous since I plan to focus more on the logic of the game and less on demanding visuals.

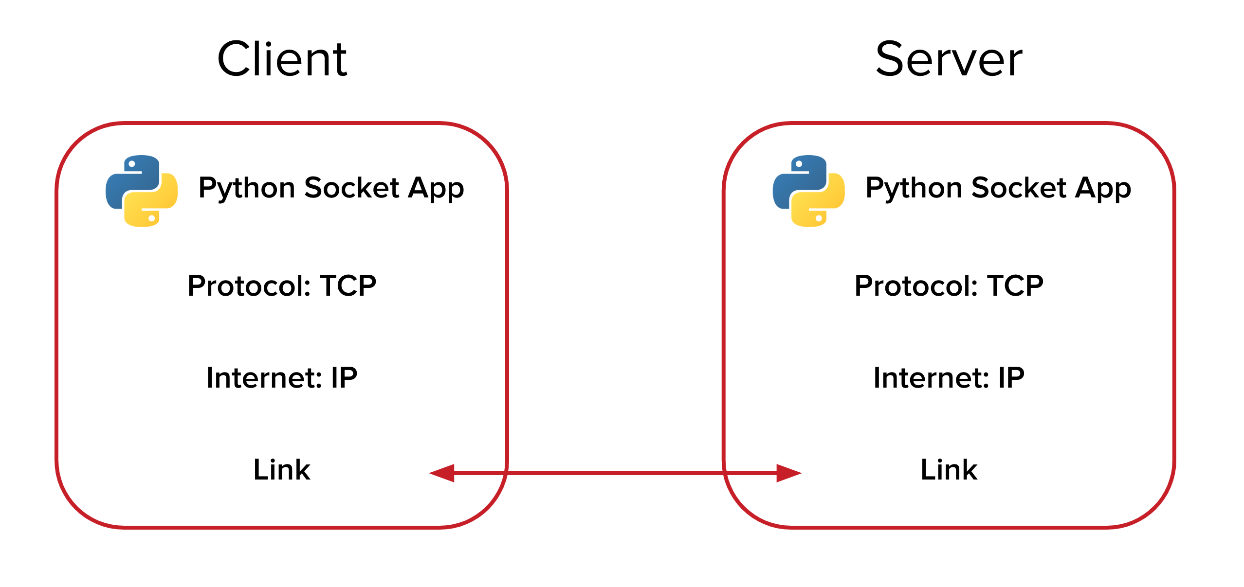
Pygame is known for its support of arcade style graphics as seen above[[10]](#footnote-11)

Pygame is also supported on multiple platforms such as Windows, Linux, Mac OS and many other operating systems. This is a significant advantage not only because I can program the solution on multiple operating systems, but mainly the fact that my software can be distributed to my peers and it will most likely work on their systems.

On the other hand, there are some drawbacks to using Pygame that I will have to be aware of. Pygame is not considered a high-performance game engine. This is partially due to the fact that it’s not hardware accelerated; hence it lacks optimisation compared to some other game engines. I’ll need to make sure there aren’t a lot of sprites and that a large proportion of video memory is not used in the final version of the game.

**Sockets**

The socket library allows information to be exchanged between devices across a network. With streaming sockets, data can be transferred to a device continuously and at any time whilst the program is running.

Using the sockets library, I can create a central server for the clients to connect to. This is called a client to server network. The server will dictate the transmission of data to the two clients when the game has begun[[11]](#footnote-12)

I’m using sockets because it allows me to implement a communication line between two devices on a network. Most importantly, it causes low network traffic. Since I’m running a real-time multiplayer game, lower network traffic should reduce the latency between the client and server, therefore increasing the performance and smoothness of the game.

However, Socket based communications will only send raw data to and from the server, so I will need to implement a mechanism for re-collecting the data and processing it back into information. Although this will work for small amounts of raw data, there will be a lot of different data such as player coordinates, projectile coordinates, character move-sets and items on the arena and implementing such a mechanism for each one of these data types will be inefficient.

**Prototyping:**

I wanted to test sending and receiving data using the socket library, therefore I set up a client-server network and sent data between 2 clients and a server.

These are the libraries that I used for my server. The socket library for data transfer across a network and the thread library which allows multiple functions to run simultaneously. This will be useful when more than one client is connected to the server.

import socket

from \_thread import \*

server = "192.168.1.01"

I used port 555 as it’s normally open and will be available to use for my program.

port = 5555

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

try:

s.bind((server, port))

except socket.error as e:

str(e)

s.listen(2)

I sent location data of sprite objects from 2 clients to a server that I set up. The integers within the tuple data structure must be converted into string format before being sent across the network, which is what the create\_position () subroutine does.

print("Waiting for a connection, Server Started")

def create\_position(tup):

return str(tup[0]) + "," + str(tup[1])

position = [(0,0),(50,50)]

def receive\_position(str):

This subroutine takes the string data that has been received from the client and converts it into a tuple data type so that the integer values can be used by the server.

str = str.split(",")

return int(str[0]), int(str[1])

def threaded\_connection(conn, player):

conn.send(str.encode(create\_position(position[player])))

I’m using a separate thread so that the server can continue to wait for incoming connections from other clients (while loop) whilst sending and receiving data from a client that is already connected.

reply = ""

while True:

try:

location\_data = receive\_position(conn.recv(2048).decode())

position[player] = location\_data

Number of bits received.

if not location\_data:

print("Disconnected")

break

else:

if player == 1:

reply = pos[0]

else:

reply = pos[1]

print("Received: ", location\_data)

print("Sending : ", reply)

conn.sendall(str.encode(create\_position(reply)))

except:

Data must be encoded and converted into bits before transferring them using sockets.

break

print("Connection lost")

conn.close()

**Problems with the prototype:**

As mentioned before, the information that is sent between the client and server network will need to be converted into raw data. This can be seen in my prototype where the data was converted from string to tuple and encoded before sending. The data will then have to be decoded at the clients end and be reconverted into useful information. Several subroutines had to be created to implement this way of communication, and more will be needed when more data types (e.g. projectile data and other player variables) have to be transferred. This will make the source code very repetitive and will be time consuming.

**The Pickle library and Object-Oriented Programming**

One efficient method of implementing multiple data transmissions is by using an object-oriented approach. Instead of transferring individual data elements, I could transfer object data. Due to the encapsulation of attributes that is available with OOP, a class for players can be created, and instantiating player objects would allow the client to send all of the player’s attributes (not just coordinates) at once. This will save a lot of time and reduce complications within the source code. It will also make it easier to debug the program and test the transfer of data since whole objects can be tested at once instead of testing variables individually.

For this to be possible, the object data will still need to be converted into byte data as it’s a requirement for transmission with sockets. The pickle library in Python will allow me to do this using the ‘pickle.dumps()’ function, which can be used to convert data types including objects into a byte-stream for transmission

**Objectives:**

* Create a login system that works with a database to create or verify an account.
* A main menu with the following options: Start, Instructions, High-Scores and Quit.
* An instructions page with an introduction detailing the objective of the game and a list of all the controls and their purposes in-game.
* High-scores page that displays all the high-scores from all stored accounts in the player database.
* If the Start button is selected, it will take the player to the server menu.
* A server that will maintain and synchronise the game with both clients and allow the clients to exchange data with each other.
* A main game subroutine where the players will be loaded into the arena.

**Main game:**

* Basic control scheme for movement and corresponding sprite animations.
* Gravity that keeps the players from hovering mid-air.
* The arena will be designed with walls and floors to both assist and hinder player movement.
* Background animation to make the game more visually appealing.
* Each player should have weapons that shoot projectiles that travel at a constant velocity until they collide or move beyond the map.
* A countdown that will start at the beginning of the match.
* Music and sound effects depending on the scenario in the game (e.g. the sound of a fast heartbeat when the player’s health is low).
* A health and damage system.
* Random collectable items should randomly appear and should provide the player with an advantage if picked up (e.g. addition of health).
* A system that determines the winner of a round. Either when a player dies or time runs out.
* An endgame display showing the winner of a round.
* A high-score comparison function that should determine whether a player’s score is their highest, if so, it should be stored in the player database.

**Server**

* Allow a maximum of 2 players to connect to the server but optimise the game so that it will be easy to add improvements such as allowing for more than 2 clients to connect and play the game. This will also make the game easier to update and add new features.
* Should have the host’s IP address stored in a variable.
* A network port should be available for the server to use.
* A command-line interface that provides the user with feedback on the server status while running (e.g. “Waiting for connection”).
* Should continuously send packets of data to the client and detect if a client has disconnected.
* Should convert the raw data received into information to process a response.
* Should have a memory efficient method of sending multiple data types simultaneously between the client and server.

**Sprite design**

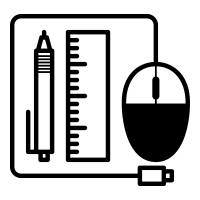
* Design 2 character models for player 1 and player 2.
* Design a sprite for the projectile.
* Multiple sprites should be created for animations.

**Possible additions and deductions:**

There’s a time limit for completing this project, so at the end I may either have extra time or lack time to complete some features of the program.

**Extra features:** I will optimise my code so that if I have time remaining on the project, I can make improvements such as support for more than 2 clients in a game round, and multiple animations for player actions.

**Limiting features:** I will prioritise making the main game work, then the performance optimisations, then the player database and finally user interface.

Documented Design

Description of how each part of the program will run:

**Login system:**

* Player will be invited with a login screen that requests account details.
* There will be an option to create a new account.
* All details will be checked or added to the players table in the database.

**Game Menu:**

* After login, the player will be directed to the main menu with the following options: Start, Instructions, High-Scores and Quit.
* Clicking ‘instructions’ will open a page with an introduction detailing the objective of the game and a list of the controls and their purposes.
* If the ‘High-scores’ option is chosen, the player will be taken to a page which displays all the high-scores from all stored accounts in the player database.
* If the user choses the Quit option, the program will terminate.
* If the Start button is selected, it will take the player to the server menu.
* Within the server menu the player will be prompted to either run the server file which hosts a server or enter the IP address of the server that they want to connect to.
* If the player choses to run their own server, there will be a button that connects them to their own server without needing to type their IP address.
* If an IP address is entered, the program will search for local server associated with it. If not found, a message will be displayed to the user notifying that there is no server present. They will have the option to go back to the game menu and create a server.
* If a server has been found or created, the main game subroutine will run and the player will be loaded into the arena with their default player sprite.

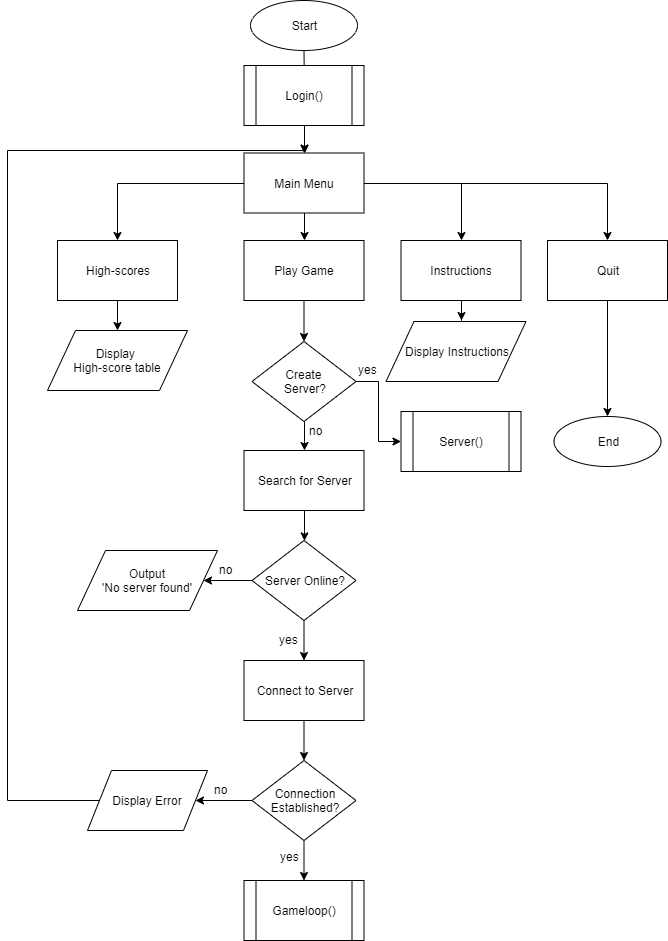
**Main game:**

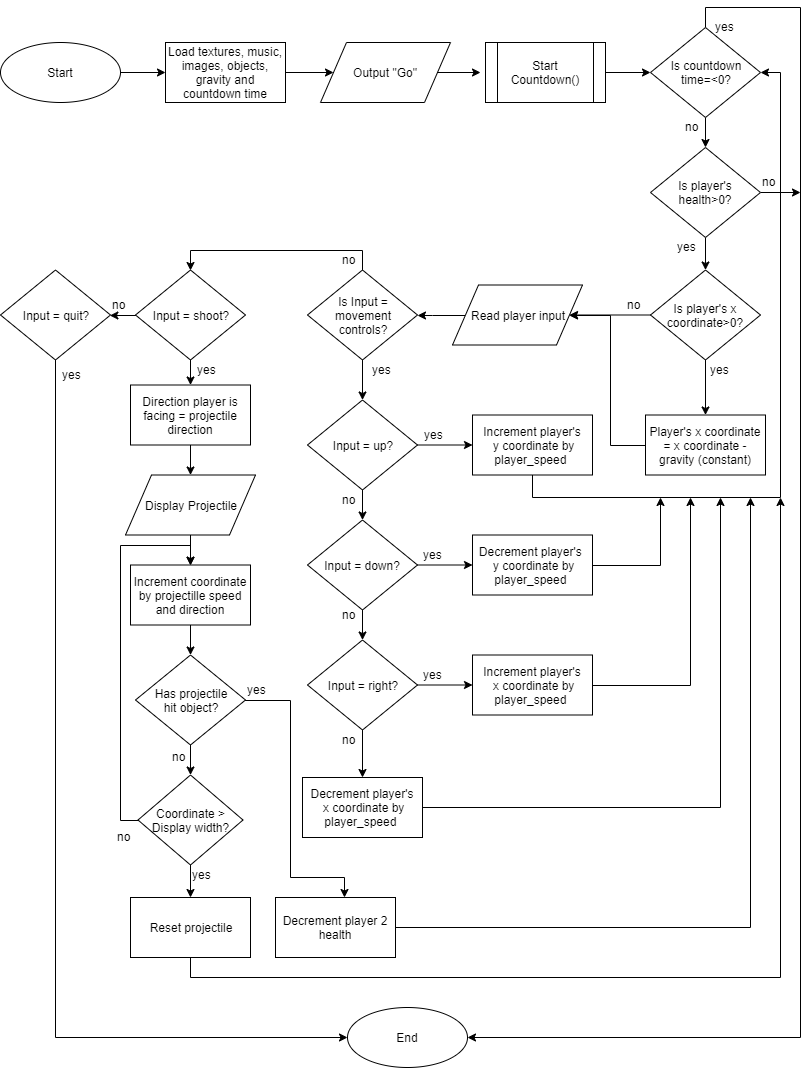
* The players will have basic movement controls such as left, right and jump, with the corresponding character animations.
* An integer constant will be created that simulates gravity within the game.
* If another player connects, the default sprite for player 2 will be loaded into the game. A maximum of 2 players can connect to the server.
* The arena will be designed with walls and floors to both assist and hinder player movement.
* A message in large font saying ‘Begin’ will indicate the start of the battle, where the players will be able to shoot projectiles by pressing the spacebar key.
* The projectiles should move in the horizontal axis at a constant speed. If they collide with an object, they will be removed from the player’s projectile list and will not stop rendering.
* A countdown timer will start at the beginning of the match.
* Music and sound effects will be present and will change depending on the events in the game.
* The players will each have a health attribute which will decrease whenever they get shot.
* A player is eliminated when their health value reaches 0. If so, the other player will win the match and a victory message will be displayed.
* If the time runs out, the player with the most health will win. If they both have the same health, the match will be a tie.
* Both players will then be redirected to the main menu, and the sever will terminate.

**Server**

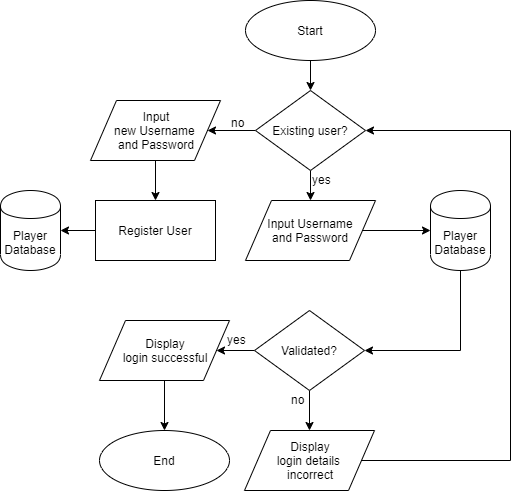
* Will act as an interface between the 2 clients and control certain aspects of the game.
* Will instantiate the player objects and update the data for both clients simultaneously.
* Will provide feedback to the user on its status (e.g. ‘Connection Lost’).

**High level overview of my game**

The following flowcharts represent the logic behind my program. These will help me to decompose the project into smaller individual problems, making it easier to solve.

**Main game loop:**

**Login system:**

****

**Setting up a client-server network:**

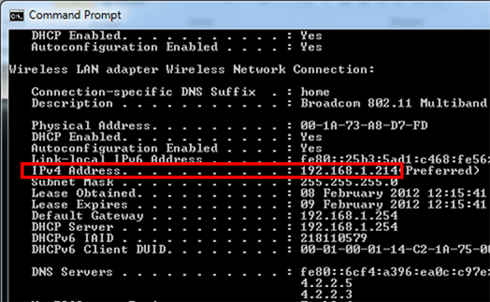
As mentioned in my analysis, I have decided to use the sockets library to set up a client-server network. A client-server network allows a user’s computer (client) to connect to a server and exchange data with it.

**Server**

The central server controls the flow of data from client to client. This will be a LAN server that’s in charge of connecting 2 local clients together and processing their inputs. It will then send a valid output to both the clients through packet switching using the sockets library.

**IP Address and ports**

To setup a server, first I will need to identify my computer’s private IP address, specifically my IPV4 address that I’ll use to host the server. This is a unique set of values that will identify a device on a network. For other clients to join my server, they will need the IP address of the computer that the server is running on. I’ll be using IP version 4 (IPV4) as it is the most common version right now.



The IP address and other network information can be accessed using the ‘ipconfig’ command in the Windows command prompt as displayed in the image above[[12]](#footnote-13)

For the communication line to work between the client and the server, I will need to use an unused port so that the client can connect to the host IP. A port is a number that allows a client from another computer to connect to a service hosted by a specific IP address. Ports are used in several protocols such as HTTP and FTP. I will need to use a free port on my router that is not used by other services.

A port that is usually free to use is port **555**, therefore I will use this port to test my server.

**Establishing Connection**

The **listen ()** function will be used so the server listens on the selected port for an incoming connection. When a known connection is detected, the function will create a new socket called a child socket and will establish a connection.

Transferring data:

When transferring raw data between client and server, the socket library usually requires the data to be in a certain format so that it can be converted into bits for packet switching. As seen in my prototype in the analysis section, I had to convert position data into a string an encode it.

def create\_position(tup):

return str(tup[0]) + "," + str(tup[1])

position = [(0,0),(50,50)]

def receive\_position(str):

str = str.split(",")

return int(str[0]), int(str[1])

This process will become very tedious and will make the program inefficient in terms of storage when numerous other variables are also processed using this method.

**Object data instead of string data**

One solution to this problem is to use object data. The Object-Oriented Programming paradigm will allow me to create objects such as players and projectiles that will each have attributes such as player’s position instead of independent variables. These attributes have a relationship with the object so when I send an object though the network, it will contain all its attributes and I won’t have to send them separately. This will also make it easier to debug the program as I won’t have to test a multitude of variables.

**Using Pickle to send object data**

I’ll need to use the pickle library in Python to send object data through the network. Pickle will let me convert the object data directly into a byte stream, where they can be sent through the network using sockets.

Pickle.dumps() – this function will convert the data into a byte stream and it will be ready for transfer.

Pickle.loads() – this function converts the byte stream back into the original data so that it can be read from the other side (server or client).

**Sending multiple objects using a dictionary**

I’m going to be transferring multiple objects. Sending each object separately could introduce increased lag time as the server will be listening in for multiple objects to be received. There could also be problems if the data is not received in synch. Due to this, I thought it would be more reliable to send all the objects at once by storing the data for each player in a dictionary, then sending each dictionary in a list.

**Server side:**

all\_data = [{

'player1':Player(x,y,[sprite\_1,sprite\_2,sprite\_3,1,direction),

'projectiles':[],

This method will also make it easier to further develop the game and add extra features. For example, adding another player by storing a ‘player3’ dictionary in the list.

'collectables':Collectable\_items(x,y,type)

}, {

'player2': Player(x2,y2,[sprite2\_1,sprite2\_2,sprite2\_3],2,direction),

'projectiles':[],

'collectables':Collectable\_items(x,y,type)

}]

The server will send the list of dictionaries to the client.

Send\_to\_client(pickle.dumps(all\_data))

**Client side:**

The client will receive the list and can chose the object that it needs. This line on the client side stores the first dictionary in the containing the player1 and projectile instances, into the player1 variable.

Player1 = pickle.loads(data\_from\_server[0])

**Objects in the main game:**

There are multiple classes within my game. The classes in my main game contain objects that have dynamic attributes some depend on each other. The game will run on a main game class, which will contain all the necessary attributes. This class will be connected to all the objects within the game. These are the objects that link to the main game:

**Player object**

The player object will be one of the objects that it sent from the client to the server in order to replay information of the actions taken by the player. They will contain the following attributes:

* Y\_position- player’s position coordinate in the y-axis.
* X\_position- player’s position coordinate in the x-axis.
* Player – player identification (e.g. Player 2).
* Speed- speed that player moves when moving in the x and y directions.
* Sprites – a list of the names of sprite images that are loaded and displayed depending on the actions performed by the player.
* Direction – Direction that the player is facing.
* Projectiles – List with bullet objects that are added whenever the player shoots
* Projectile\_limit – Maximum amount of shots allowed to be fired at a time.
* Health – Player’s health value.
* Score – The player’s score.
* Collectibles – list of collectible items that the character has picked up.
* Death Animation – list of sprites for the animation when a player is destroyed.
* Collision up/down/left/right – Boolean value that indicates whether the player has collided with an object on the map.

**Projectile object**

The projectiles in the game are fired using the blasters given to each player. If a player comes into contact with the other player’s projectile, they will lose health. The projectile object will have the following attributes:

* Y\_position - projectile's position within the y-axis.
* X\_position – projectile's position within the x-axis.
* Player – Identifies which player the projectile belongs to (used for score system).
* Speed – The projectile’s speed after being fired (constant).
* Image – The imported image for the projectile.
* Direction – direction projectile’s being fired at.
* Hitbox – A square-shaped model of the projectile that’s used to determine collisions.
* Collided – Indicated whether projectile has collided.
* Remove – Indicates whether projectile should be removed or not.

**Collectable Item objects:**

These are multiple objects randomly placed around the map that will provide a player with a special ability (e.g. More than 100% health, increased fire-rate and the ability to heal). They are collected by the player automatically if the player moves to their position. These objects will have their own class and will have the following attributes:

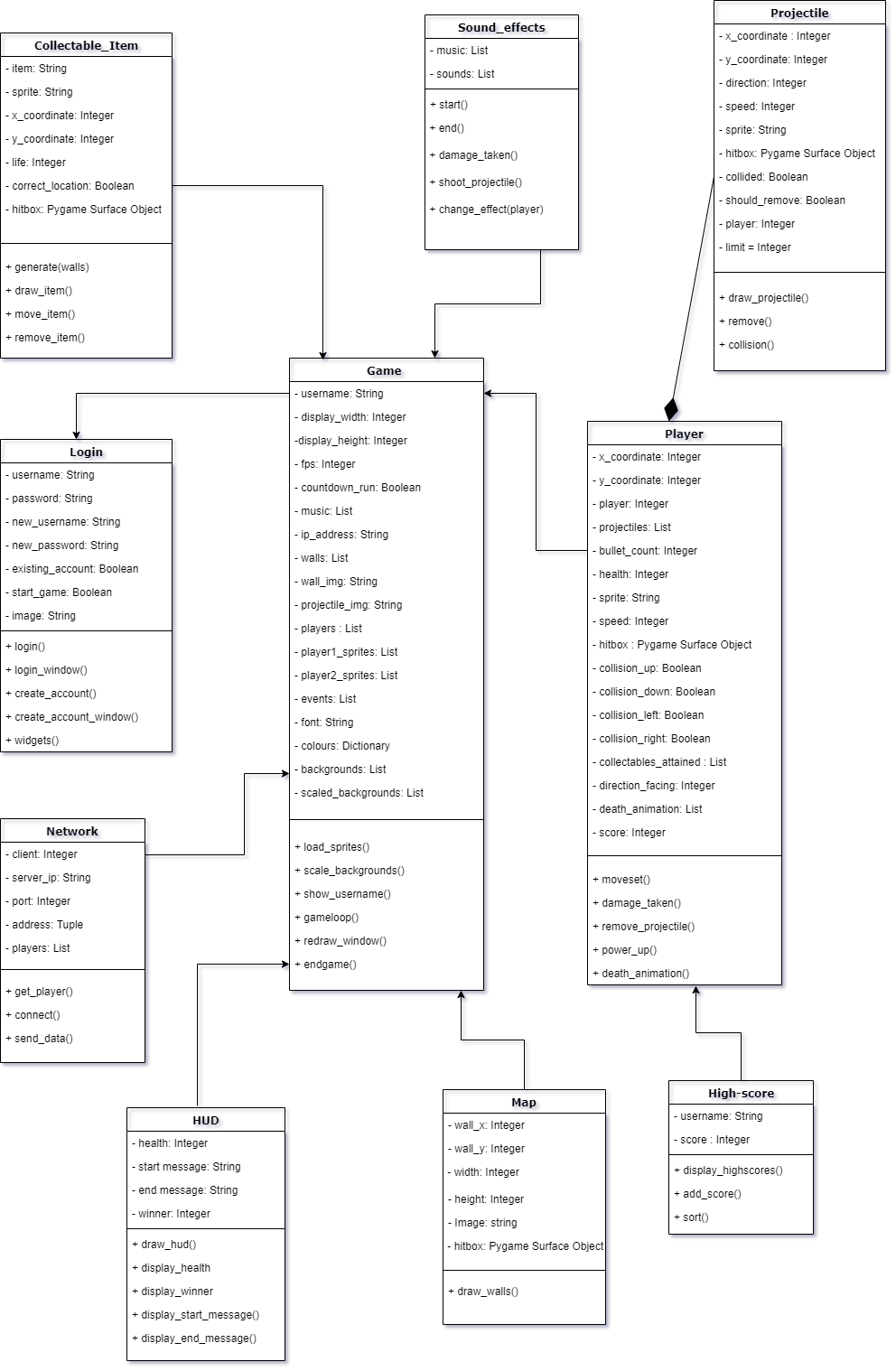
* Item – String that indicates the type of item.
* X-position – x coordinate on the display.
* Y-position – y coordinate on the display.
* Sprite – the name of the sprite image that will be displayed.
* Life – Item will de-spawn when this reaches 0.
* Correct\_location – Boolean value that’s controlled by a method inside the function that makes sure the item is spawned in an appropriate location (e.g. not inside a wall).

**Map objects:**

These include walls, floors, platforms and other geometric objects the map. The player will not be able to pass through these objects and they will have the following attributes:

* X – coordinate of object in the x-axis.
* Y – coordinate of object in the y-axis.
* Height – height of the shape.
* Width – width of the shape.
* Hitbox – Pygame rectangle object created to check for collisions.
* Image – Name of the image file displayed to represent object.

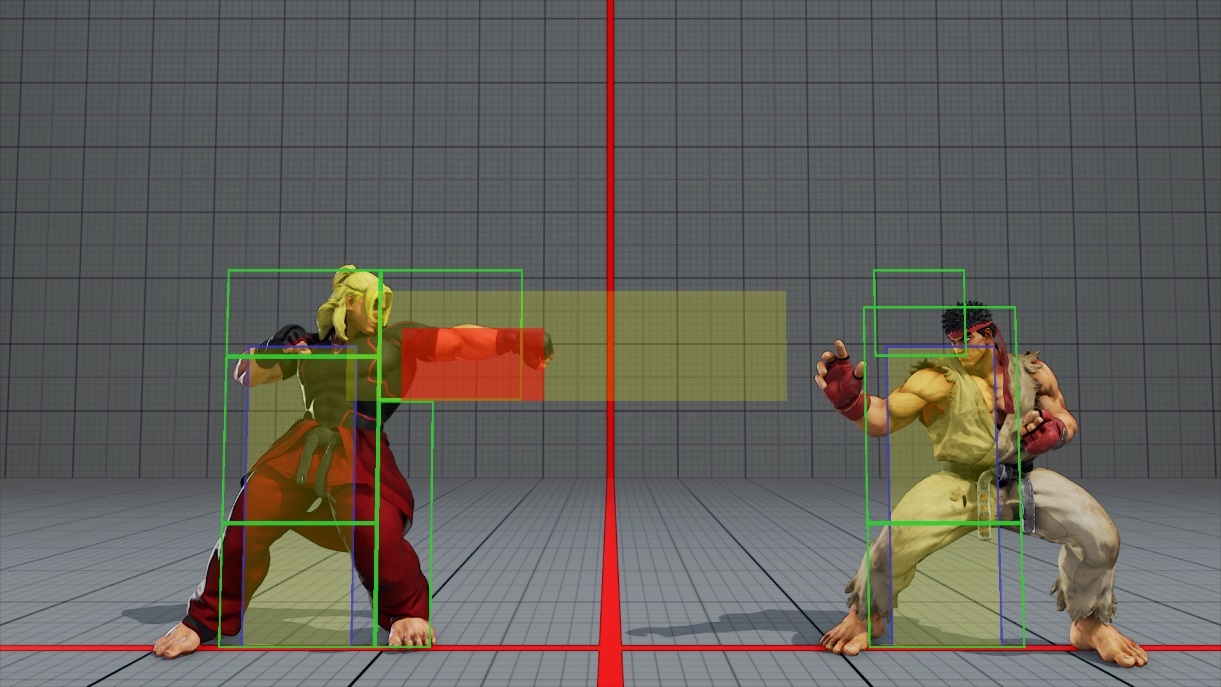
**Class diagram:**

This class diagram represents how all of the classes from the program link together:

**Collision detection:**

Collision detection is another aspect that is essential for other parts of the game to work, such as health reduction, number of projectiles displayed and the scoring system. Collision detection will be used to identify when and where a projectile has collided in order to remove it and deduct player health if necessary. It will also be used to detect the player’s collisions with objects on the map, such as walls and the other player.

All player and projectile objects will have a sprite associated with them. However, abstraction can be used to make collision detection faster and easier to implement. The players and projectiles can be modelled into simple geometric shapes such as squares and rectangular hitboxes.



Hitboxes are a popular method of collision detection due to their simplicity. This is used in many games such as the Street Fighter games as seen in the image above.

By doing this, it will be easier to program the detection by assigning ranges to the hitboxes depending on their width and height. Then an algorithm such as the one below can be used to detect the collisions:

Player.hitbox 🡨 (x,ysprite\_width,sprite\_height)

Projectile.hitbox🡨(projx,projy,proj\_sprite\_width,proj\_width\_height)

Def collide(rect\_one,rect\_two)

If rect\_two.x-rect\_two.width/2 < rect\_one.x < rect\_two.x + rect\_two.width/2

or rect\_two.y - rect\_two.heigth/2 < rect\_one.y < rect\_two.y + rect\_two.height/2 Then

return True

else

return False

if collide(projectile.hitbox, player.hitbox) Then

projectile.collided🡨True

For collisions with walls, the algorithm is more complex because rather than the game detecting the collision and simply removing the and object as with the projectiles, the function should also stop the player from advancing once they have collided with the object. This means that there will need to be collision variable for each direction that the player is facing and two more to verify if the player has collided with a wall on top or below them.

collision\_left🡨False

collision\_right🡨False

collision\_up🡨False

collision\_down🡨False

if collision(wall.hitbox,player.hitbox):

if player.direction == left and collision\_left == False Then

collision\_left🡨True

if player.direction == right and collision\_right == False Then

collision\_right 🡨True

if (wall.y+wall.height/2)<(player.y-player.height/2) and collision\_up==False Then

collision\_up 🡨 True

if (wall.y-wall.height/2)>(player.y+player.height/2) and collision\_up==False Then

collision\_down 🡨 True

else

collision\_left🡨False

collision\_right🡨False

collision\_up🡨False

collision\_down🡨False

**Countdown Timer**

The timer for my game will determine when a round will end. At first, I tried implementing a local timer that runs on the client side and is independent of the server:

class Timer():

def \_\_init\_\_ (self,time):

self.time=time

self.start=False

self.end=False

self.font=pygame.font.Font("Images/arcade.TTF", 35)

def show\_time(self,player2,gameDisplay):

colour=(255,255,255)

The timer starts when player 2 is loaded into the game. When player 2 is loaded into the game, they will drop vertically down from their original position due to gravity in the game. This comparison statement will detect the moment when player 2 moves from their original position and initiated the countdown.

if self.start==False:

if player2.x!=100:

self.start=True

if self.start==True:

if self.time<1:

print("Time's up")

self.start=False

else:

timer = self.font.render(str(int(self.time)), 0, colour)

gameDisplay.blit(timer, (width/2 - (200), 20))

self.time-=(1/60)

Timer is decremented by 1/60 each time it’s called in the while loop because the game runs at 60fps, so it will decrement by 1 each second, therefore emulating real time.

if self.time<1:

self.end=True

**Problems with the timer:**

The timer function shown above is local to the client, hence there’s no method of synching the timer on both clients to make sure the output is the same for both clients. Factors such as random frame drops can affect the rate of change of time since it’s proportional to the frame rate of the game. This can cause the time output to vary as a round progresses…

**Using multi-threading for the timer:**

One method of fixing this issue was to use a separate CPU thread to execute the timer in parallel to the running game so that it won’t be affected by any frame drops from the main game itself. However, after testing this I realised that multithreading causes performance issues due to Python’s GIL (Global Interpreter Lock), which prevents multiple threads from executing Python bytecodes at once. By conducting further research[[13]](#footnote-14) I found that running a function on one thread is much faster when it comes to more demanding programs such as my game due to this restriction.

**Synching the timer between each client:**

A better method would be to run the timer on the server side and synchronise the time value with both clients by passing the value using sockets. This will make sure that the time value is the same for both clients. Calling the timer on the server side will also allow me to start it when the second player has connected, instead of the previous method where the timer started when the player moves from their initial position.

This is the class definition for the timer:

class Timer(object):

def \_\_init\_\_(self):

self.start\_time = None

self.stop\_time = None

def start(self):

self.start\_time = time.time()

The calculations for finding the time elapsed have been decomposed into several functions. The functions start () and stop() both retrieve the current time from the OS and assign the values to their corresponding attribute.

def stop(self):

self.stop\_time = time.time()

def time\_elapsed(self):

if not self.has\_started():

return 0

return int(time.time() - self.start\_time)

Time elapsed calculates the number of seconds elapsed by subtracting the starting time from the current time and returns an integer from the result.

def has\_started(self):

return self.start\_time() != None

**Server-side:**

timer = Timer ()

all\_data = [{ 'player':Player(0,0,["sprite1.png","right.png","left.png"],1,1,0),

"timer": timer,

}, {

'player': Player (100,100, ["sprite2.png","player2right.png","player2left.png"],2,2,0),

"timer": timer,

}]

As seen above, I have instantiated a new ‘timer’ object from the class defined previously. Then I added it to the ‘all\_data’ dictionary for both players. This means that both clients will receive the ‘timer’ object and it will be updated centrally by the server.

if not data\_received\_from\_client:

print("Client Disconnected")

total\_connections -= 1

break

else:

all\_data[0]['timer'] = timer

all\_data[1]['timer'] = timer

if total\_connections == 2:

if not timerHasStarted:

timer.start()

timerHasStarted = True

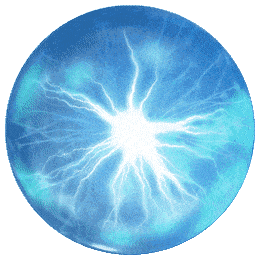
Finally, this function will continuously check for data received from each client. Whenever a client connects, the value of ‘total\_connections’ is incremented, and then decremented when disconnected. I’ve set the timer to start when there are 2 clients connected as that is when the round should begin.

**Generating player score:**

For there to be a high-scores table, the player should have different scores that depend on multiple events occurring within the game. This will include, the amount of damage dealt by the player, the collection and usage of collectable items and the difference between each player’s health at the end of the game. A real time score-count will be displayed at the top of the game screen…

…

**Collectable Items**

There will be multiple collectable items that randomly spawn on the map. They will each have a lifetime in which they will spawn for and then disappear. If the player moves to its position, it’s automatically picked up and provides the player with power-ups such as additional health, faster movement and invisibility.

Heart: Health

Potion: Invisibility

Orb: Super speed

**Method 1**

I wanted the collectable items to appear in the map spontaneously in a random location, but the spawn times and locations had to be the same for both clients. This meant that the items must be instantiated and given a random location by the server before being sent to both clients.

This method is shown by the pseudocode below.

Spawn\_chance is the chance that an item will spawn during the game.

Itemlist = []

Spawn\_chance = 2

While True

Using a random integer generator, a value is generated. If the value is smaller than the spawn\_chance, it will generate a collectable item.

Using probability allows me to change the average number of collectables spawning in a game whilst still keeping it random.

Spawn\_value = RandomInteger(0,100)

If Spawn\_value < Spawn\_chance Then

Item = Collectable()

Item.generate()

Itemlist.append(Item)

Item.life = Item.life-1

If Item.life <0 or item.collided = True Then

An item will have a random value for life, which will continually decrement. If life is 0 or the player collides with the item, it’s removed.

Remove item from Itemlist

all\_data = {player:playerdata,timer:time,collectables:Itemlist}

Send all\_data to Clients

**Generate item method:**

The generate method makes sure the location is random and that the item does not spawn inside a wall. A list containing all the wall objects is passed as an argument and the method will repeat recursively if the item’s location is within a wall.

Public Procedure generate(self,walls)

self.type = RandomInteger(1,3)

self.x = RandomInteger(0,width)

self.y = RandomInteger (0,height)

self.hitbox = PygameRectangleObject(self.x,self.y,25,25)

self.wrong\_position\_count = 0

for wall in walls

if self.hitbox collides(wall) Then

self.wrong\_position\_count = +=1

if self.wrong\_position\_count>0 Then

generate(walls)

**Problems encountered with method:**

The main issue with this method is that the server continuously sends and updates the list of collectable items on the client side. This means that any changes to the items made on the client side will be overwritten by the server. When the player picks up an item, the item is removed from the list on the client side, however the sever keeps updating the list in order spawn more items, and as a side-effect it adds the already collected item back to the list. This caused the item to stay spawned until its lifetime reached 0.

To fix this problem, the processing of the item had to be moved to the client side. This meant that the items will be generated on the client side. However, this caused synchronisation issues as the item locations and lifetimes were different for each client.

**Method 2**

To fix this problem, I instantiated the items on the client side, then used a 2D array to send the items’ attributes to the client, where the client will then create their own collectable item objects and take the values from the array to be their attributes. This way the client will replicate the objects received from the server and the client will have full control of these objects from their side, preventing the server from overriding them.

The array will look be similar to the one below. Note: Python 3.0 does not natively support the use of arrays, therefore a list will be used during development.

[ [ item.x, item.y, item.spawn\_time, item.life, item.type], [item.x, item.y…], [item.x, item.y… ] ]

**Synching spawn times:**

As detailed previously, the countdown timer object is continuously in synch with the clients and is running on the server. Each item will be given a ‘spawn time’. If the spawn time is equal to the time on the timer, the item will spawn. Since the timer for both clients is already synchronised, the items will spawn at the same time for both clients.

This is method is represented by the pseudocode below:

The server generates the collectables items and adds them to an array. A new attribute ‘time’ is added and generated randomly. This is to reference the time elapsed from the ‘timer’ object. Each item has its own array with its attributes stored. Each array is added to the main 2D array which is sent to both clients.

**Server Side:**

collectable\_data = []

for number in range 0 to RandomInteger(10,20)

item = Collectable()

item.generate(walls)

collectable\_data.append([item.x,item.y,item.time,item.life,item.type])

Send {player:playerdata,timer:time,collectables:Itemlist} to Clients

**Client Side:**

Collectable\_list = []

Collectable\_data = received(data.collectables)

For data in collectable\_data

The clients receive the data from the list (x, y, spawn time, life and type). It uses this data to re-create the collectable item object on the client side. As seen on the left, ‘item’ is instantiated and its attributes are provided by the received array of data.

item = Collectable()

item.x = data[0]

item.y = data[1]

item.time = data[2]

item.life = data[3]

item.type = data[4]

Collectable\_list.append(Item)

If item.hitbox collides(player.hitbox) Then

Remove item from Collectable\_list

**Power-ups:**

The player will store the items in a list attribute and will have a usage timer. When used, the usage timer will start and be incremented in the game loop. The item is removed from the list and when the usage time reaches a certain value, the item will be removed from the list and the power-up will stop.

**Speed** – When the player picks up a speed orb, their speed attribute is changed to equal another attribute called ‘super\_speed’, which is a higher value. This value will now be the player’s speed until the usage time of the orb runs out. The speed wasn’t incremented as this will cause the speed to increase every time an orb Is picked up to the point where the player’s speed increase may cause a disadvantage, hence ‘super\_speed’ is a constant.

**Invisibility** – When the player uses a potion, their sprite will stop displaying until the item’s use time has run out. The player will still be able to see their username which is situated above their player sprite, but the other player will not, hence the invisible player’s location will be completely hidden. The Boolean attribute *‘visible’* will be given to them. When the item is picked up it will become false, and true when the usage time expires.

**Health –** Picking up the heart will increment the player’s heath by 50 health points. I may change this value when testing my game to make sure the health increment doesn’t disadvantage the other player.

**Optimisation**

**Sorting High-scores using Merge Sort:**

The high-scores list will need to be in order of highest to lowest scores with the players’ usernames beside their score. I’ve decided to use a recursive merge sort algorithm to sort the high scores. The merge sort algorithm has an O(nlogn) time complexity during execution. This makes it efficient especially if the dataset increases with more player accounts. Furthermore, it reduces dependence on the database since the sorting occurs before displaying the data and therefore will not result in problems during storage or retrieval.

I’m going to use a list to sort the scores and a dictionary to pair the scores with the correct usernames. The algorithm is represented by the pseudocode below:

Function merge\_sort(score\_list)

if length(score\_list) <=1

return score\_list

middle = length(score\_list)/2

left = merge\_sort(score\_list[0 to mid])

right = self.merge\_sort(score\_list[mid+1 to EndofList])

return self.merge(left,right)

Function merge(left,right)

final\_list = []

left\_pointer = 0

right\_pointer = 0

While left\_pointer<length(left)and right\_pointer < length(right)

if left[left\_pointer] > right[right\_pointer] Then

final\_list.append(left[left\_pointer])

A standard merge sort algorithm will sort the scores in ascending order. The comparison statement is reversed here (left > right) as the scores will be in descending order with the highest scoring player displayed first.

left\_pointer = left\_pointer + 1

else

final\_list.append(right[right\_pointer])

right\_pointer = right\_pointer + 1

for index 0 to left pointer

final\_list.append(left[index])

for index 0 to right pointer

final\_list.append(right[index])

return final\_list

**Input/Output/Storage table:**

I’ve constructed a table showing how the game will handle different inputs by the user.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Process** | **Storage** | **Output** |
| Create account-  New account details | -Verify format  -Check if they already exist  -Encrypt password | Username and encrypted password | Account created/unsuccessful message with the reason |
| Login –  Existing account details | Encrypt password and  check database for matching data | None | Login successful/details incorrect |
| Instructions | Load Instructions subroutine | None | Display instructions |
| High-scores | Retrieve username and high-scores from all accounts on the database file. | None | Display high-score table for all local accounts |
| Play game | Load game menu | None | Display server options |
| Create server | -Run server.py file  -Run main game subroutine | Update high-score | -Server status  -Display map, sprites and text |
| Join existing server | -Listens for server and joins if one is available  -Run main game subroutine | Update high-score | -Connection status  -Display map, sprites and text |
| Movement controls  (in-game input) | Change player coordinates and update sprite attribute | None | Display new sprite |
| Fire key  (in-game input) | -Load projectile image  -increment x coordinate | None | Display projectile |
| Quit | End processes and close game and server window | None | None |

**Using a Database**

The database for my program is not extensive, therefore it doesn’t require normalisation. The main purpose is to store an individual player’s account details and high score locally so that they can be viewed by the user.

**Data Dictionary:**

I will need to store the following data for each player in my local database.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data item | Data type | Data size | Description | Example | Validation |
| PlayerID | INT | 4 | Unique identifier for the user (primary key) | 1031 | Is numeric |
| Username | VARCHAR | 20 | The name the user wishes to identify as within the game. | Player21 | Not null and 3< Length <20 |
| Encrypted  Password | CHAR | 20 | Password of the user that’s encrypted for security | $53hbr35 | Not null and 5<length <20 |
| HighScore | INT | 5 | The player’s high-score which will be compared and updated between matches. | 02984 | none |

Useful SQL Commands:

As a developer, I will need to know the following SQL commands to control and maintain my database:

**CREATE –** creates a new table in the database. I will use this to create a table that will store the user’s details:

*CREATE TABLE Player (*

*PlayerID INT Primary Key,*

*Username VARCHAR (20),*

*EncryptedPassword VARCHAR (20),*

*HighScore INT*

*);*

**SELECT-** This will allow the user to view specific details such as high-scores without displaying other records that are not necessary:

*SELECT Username, Highscore*

*FROM Player*

**INSERT-** This will be used to store details for new accounts. For example:

*INSERT INTO Player*

*VALUES (3042, ‘legend27’, ‘3efje43fs’, 2899);*

**UPDATE –** Used to modify existing values such as the previous high-score after it’s been exceeded:

*UPDATE Player*

*SET HighScore = new\_highscore*

*WHERE UserID = current\_userID;*

I will need to integrate these SQL statements into the main program so that the database can be queried automatically. E.g. checking and updating the high-scores after every game.

**Encrypting Passwords**

My program will encrypt the account passwords before they’re stored on the database so that it’s hidden from anyone that reads the data directly from the database file.

There are several types of encryption that I could use, but since this is a local database it will not affect any outside users’ data, so there is less need for the most secure algorithms available. Because of this I decided to base my encryption on the Caesar Cipher algorithm.

This process is shown in the pseudocode below.

Function encrypt (password,shifts)

encrypted\_ascii 🡨 [ ]

encrypted\_string 🡨 ""

For character in password

encrypted\_ascii.append (char\_to\_code(character)-shifts)

EndFor

For num in encrypted\_ascii

encrypted\_string🡨 encrypted\_string + code\_to\_char(num)

EndFor

Return encrypted\_string

Function decrypt (encrypted\_string,shifts)

decrypted\_acii 🡨 [ ]

decrypted\_string 🡨 " "

For character in enctryped\_string

decrypted\_ascii.append(char\_to\_code(character))

EndFor

For num in decrypted\_ascii

decrypted\_string 🡨 decrypted\_string + code\_to\_char(num)

EndFor

Return decrypted\_string

The encrypt function will take the user’s input when creating an account and will process it, returning an encrypted string. This will be stored as a database record.

When the user tries to login, the program will retrieve the encrypted string and the decrypt function will return the original password. This can be compared to the user’s input to verify their login details.

The shift for the cipher cannot be larger than 32 because that’s the lowest possible ASCII value. If the shift value is larger than the input character, the generated ASCII value would be negative and its conversion to a string would not work since ASCII values cannot be negative.

**Using Multiple Threads to Connect Multiple Clients:**

Multi-threading allows multiple problems to be processed in parallel to each other. This will be helpful for the server side of my network as it will be dealing with more than one client.

The Threading library can be used to implement this on Python 3. This pseudocode represents how threading will be used to connect multiple clients (players) to the server.

While True

If client connected Then

Start\_new\_thread(threaded\_client (connection,player))

The server will listen for incoming connections and start a new thread for each one. This way the game can support many clients simultaneously without wait time in-between.

Function threaded\_client(connection,player)

While True

Alldata🡨data received from client

If data received = none Then

Display(‘disconnected’)

Break

Else

If player=1 Then

send (pickle.dumps(alldata[0]))

Else

send (pickle.dumps(alldata[1]))

Data Structures

I’ll be using multiple data structures within my program for different purposes.

**Dictionaries** – A dictionary is used to store all the data that is used in the game for each player. This is initially stored in the server, then it’s also sent to the client. Using a dictionary is memory efficient because all the player data can be sent at once, and the client can choose which data to use once it’s received. This will also make it easier to add and remove features in the game, because values in a dictionary aren’t stored in a specific order, meaning that there’s no indexing of data. So adding more data to this structure will not change how the previous data is accessed (no indexing so references will not change), hence it won’t affect the rest of the game.

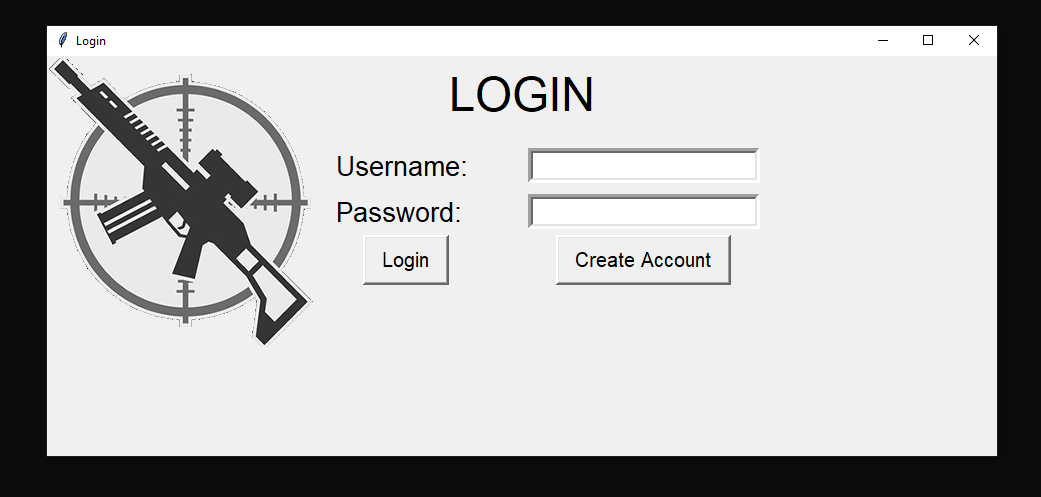
**Linked lists –**This is a dynamic data structure, hence it’s perfect for storing data such as projectiles that are generated by keyboard input. Every time the player presses the fire key, a projectile object will be added to the list. The list is dynamic, so it will take up less memory when the player hasn’t generated any projectiles. The projectiles will be removed from the list when they collide or move beyond the map, hence saving more main memory.

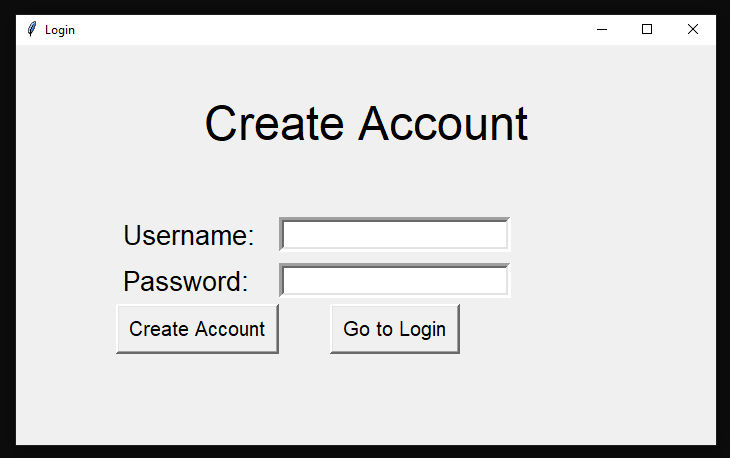
**Queue –** A queue data structure can be used for pre-made animations such as the death animation of a player. Each image of the animation can be stored in an array and displayed and removed in a ‘First in First Out’ order. This is more memory efficient than just looping through a list because the animations such as the player’s death cannot be changed with user input, therefore the image references can be added and removed from the list as they won’t be required later.

**Files –** Files are very important in my game as there will be several that are imported and used in several parts of the program. These include the database file for user details, image files for player sprites and game animations, and TTF files for different fonts. My program will also be separated into several files such as the 4 main menu options, a file containing all the main game classes, the server and a networking file used by the client to communicate with the server. These will all be imported into the main game file which will work as an interface to control which parts of the game are to be accessed by the user.

Designing the UI

Login Screen Design:

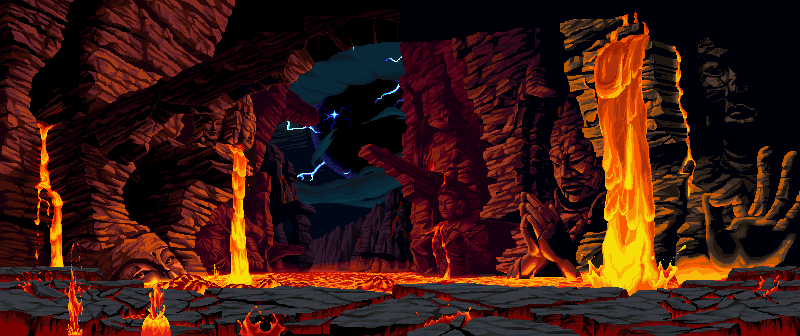
As seen below, I’ve designed a simple login screen with 2 authentication fields, an option to create a new account and on the left, an image representative of the shooter genre. This will be the first window that will open when the game is run. If the user details are verified, the user will be taken to the main menu screen.

This window will open if the user choses to create a new account. The user will enter their authentication details and if successful, they will be returned to the login screen again.

Option to return to login screen (in case of accidental clicks).

If information is successfully verified, the data will be stored in the user database.

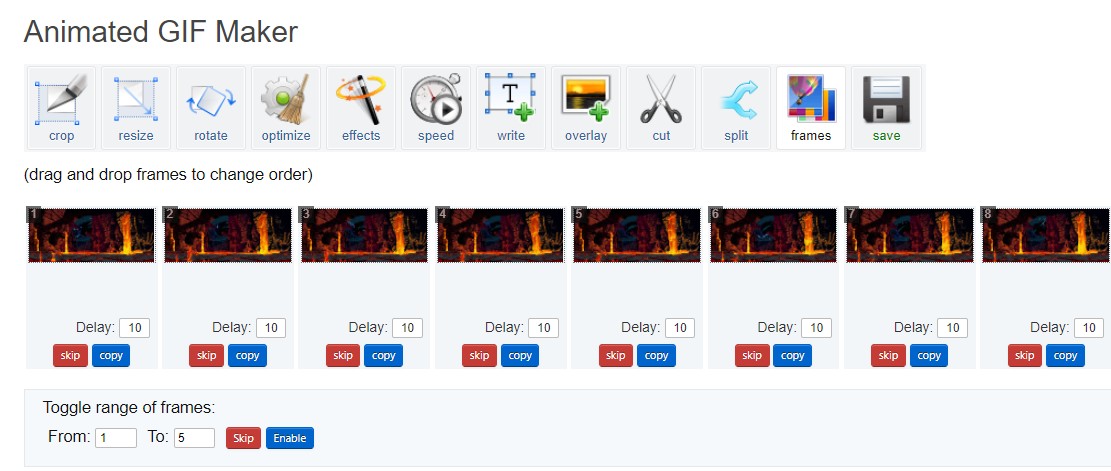
In-Game Visuals:

Alongside the sound effects, I decided to add in visual effects to make the game more immersive. I did this through background animations to make each round look more realistic. 

I decided on using this gif file for my animated main menu screen. However, I found out that Pygame doesn’t support animated gifs and displays a static image instead.

This means I must implement animation process manually.

Splitting frames:

One method is to split the frames from the animated image and import them individually to a list in my main menu subroutine. Then I can loop through the list and display each image consecutively as an animation. I can also control the framerate of the display to change the speed of the animation.

Individual frames from gif

I used an online tool[[14]](#footnote-15) to split the gif into separate frames. There were 8 frames that made up the animation.

**Looping through the frames:**

backgrounds 🡨[background1,background2,background3,background4, background5,background6]

The list ‘backgrounds’ will store the background images that are scaled to the appropriate resolution.

Within the loop, the counter will be incremented and used as an index to loop through the list. It’s also used to track which image in the list is being displayed to prevent an ‘index out of range’ error.

counter 🡨 0

While True

If counter>6 THEN

counter🡨0

Else

counter🡨counter+1

EndIf

Counter = Counter + 5/framerate

Display backgrounds[int(counter)](x=0,y=0)

Counter value is incremented by a fraction that depends on the frame rate of the game. This allows the framerate output of the background to be separate to the framerate of the game. This means the animation speed can be changed without affecting the framerate of the main game.

EndWhile

Designing the Main Menu:

The main menu screen with have 4 options: Start, Instructions, High Scores and Quit. The main game will only use keyboard peripheral inputs, so I thought would be appropriate to make the main menu operate using keyboard inputs.

Background animation:

Like the main game background, I’ve seen many modern games that have an animated main menu background. This feature would increase the aesthetics of my game and will make it look more professional the moment it’s opened.

Creating the menu interface:

Before creating the animated background, I will need to design the main menu interface. It’s important to make the main menu screen look good as it’s the first part of the program that the user will see after they log in.

The ‘arcade classic’ font for the text as it’s aimed toward arcade gamers.

Selected option will be in a different colour so that the user can see which option they are choosing.



Four vertically stacked options that will take the user to different parts of the program.

Animated background that is relevant to the shooter genre of the game.

**Using a list and pointer to track user input:**

There will be 4 options vertically listed: start, instructions, high-scores and quit. The user will use the arrow keys to navigate through the options. I devised a method to efficiently track the user’s choice of either options through their peripheral inputs.

Tracker🡨['start','instructions','highscores','quit']

Pointer🡨 0

I used a list containing the options and a pointer that tracks the output shown to the user.

For selected in window

If selected = quit Then

quit()

If selected = Keypress Then

If selected = Up\_key Then

selected🡨'start'

If pointer>0 Then

pointer🡨pointer-1

selected🡨tracker[pointer]

If selected = Down\_key Then

if pointer<2 Then

pointer🡨 pointer+1

selected🡨tracker[pointer]

If selected = Enter\_key Then

if selected = 'quit':

quit()

Else If selected = 'start' Then

game\_loop()

Else If selected = 'instructions' Then

instructions()

Else If selected = 'highscores' Then

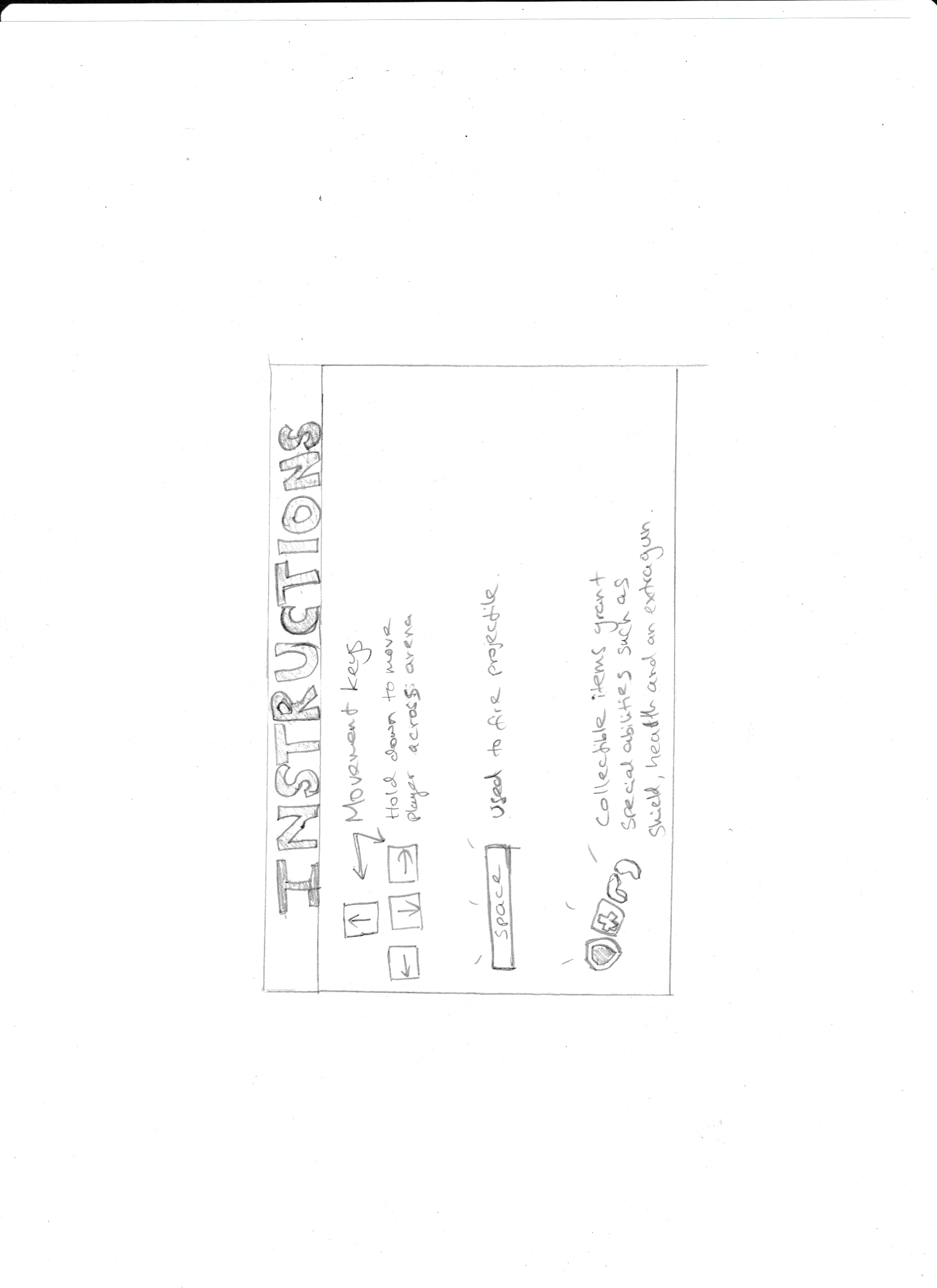
High\_scores()

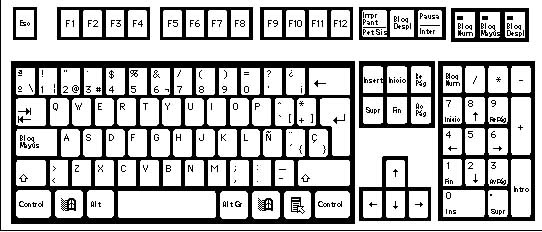
EndIf

EndFor

When the user clicks either the up or down arrow key, the pointer will be incremented or decremented up to a limit and will reference an index on the list. The strings in the list will be passed into a text processing function, which will output the corresponding text in the correct font.

**Instructions page:**

****The instructions page will have a funky arcade style design with arcade fonts and bright colourful images.



Large colourful text (in arcade font) with a bright background to showcase the casual nature of the game.

Instead of a back arrow, there will be an image of a backspace key, indicating that the user has to press that to go back.

Images detailing the controls for the player and the features of other objects such as collectable items that are found around the map.

The instructions page is guide for the player that will display the objective of the game, different controls in-game and their effects.

**High-scores page:**

The high scores page will display a table with all the usernames and scores from all the accounts that have been created. This will be created using the Tkinter library on Python 3.0 as it’s a good option for small graphical tasks. When selected in-game, a separate Tkinter window will run with the use of multi-threading and it will display the high scores. The table will be in descending order of score and will look like the following:

|  |  |
| --- | --- |
| **Username** | **High-score** |
| TheLegend27 | 8989 |
| Terminator249 | 3241 |
| JusticeWarrior569 | 242 |
| DarthTyrant | 241 |
| Noobmaster68 | 62 |

Although this table looks manageable, it will grow larger with more accounts, therefore I will need to limit the number of scores displayed so that it doesn’t become too large to display.

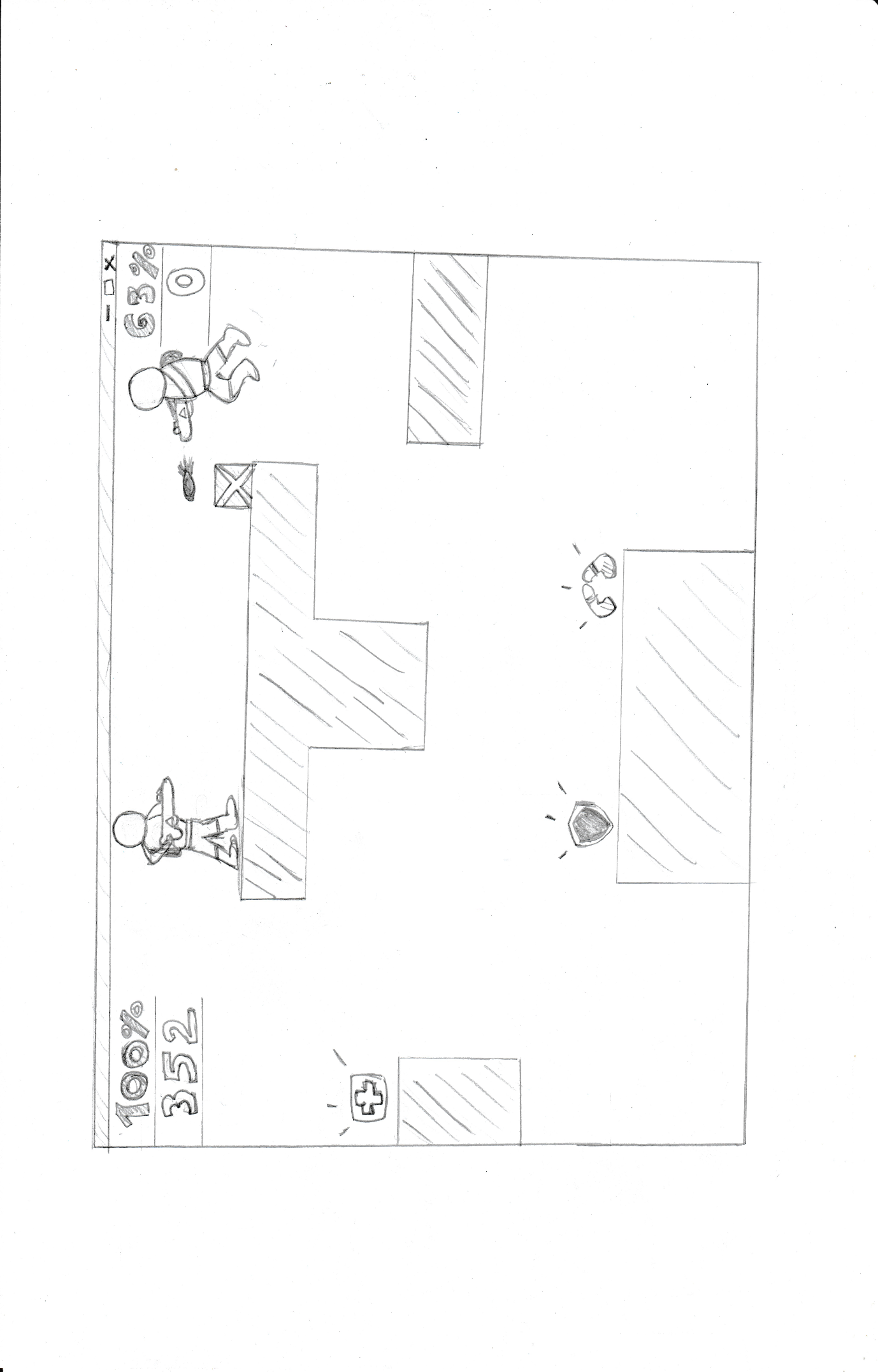
**Using a threshold score:**

One way of fixing the problem mentioned above is to introduce a threshold score. First there will need to be a maximum value that will limit the number of scores that are viewed by the user. The records will be arranged in descending order of score and the maximum number of highest scores will be displayed. The lowest score from here will be stored in the main game as the threshold score. If the player beats the threshold score and is not in the high scores list, the threshold score will be replaced by the new high-score and the table will be ordered again in descending order of score.

This method will be of use if the user has multiple accounts or allows others to log in and play the game on their system. However, due to time restrictions and the fact that this feature is not as important compared to optimising and developing the main game, this will not be prioritised.

**Main Game:**

This is how to main game will look like when the players have successfully been loaded in. I’ve drawn the main features in the foreground of the display, but I will also have a background image of the map.



Health percentage and score of player 1 1

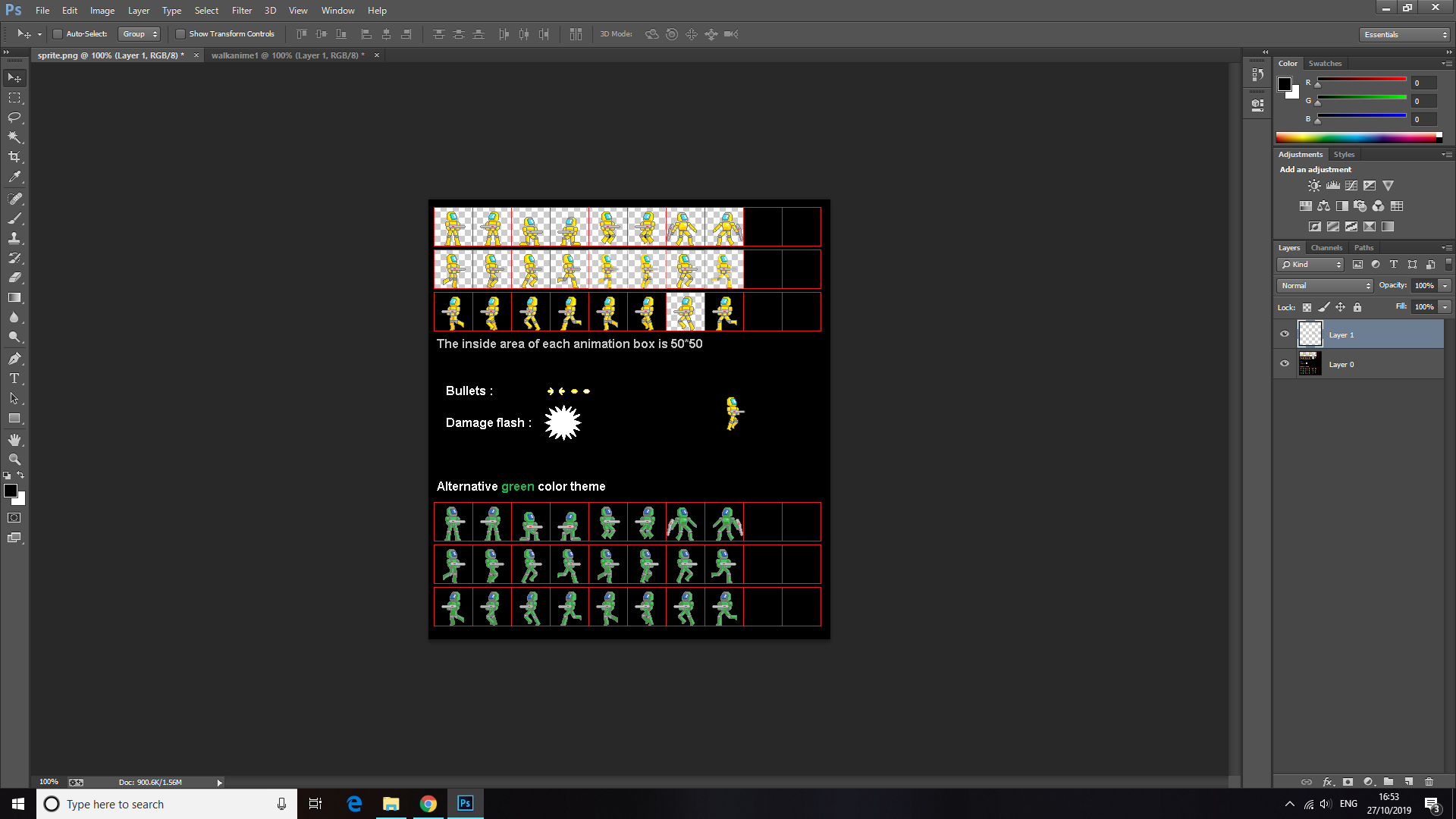
Projectile being fired.

Rectangular platforms around the map allow for more strategic gameplay such as using it for cover and elevation.

Collectable items such as shield, health packs and dual weapons for extra player abilities.

**Creating and Editing Visuals:**

I decided to use Adobe Photoshop CC 2018 to edit and create the visuals of my game.

This software will be used to crop and resize downloaded sprites. I can also alter their visual apsects such as shading and hue.

Editing sprite packs and relevant images allows me to create animations and visual effects to enhance the player’s immersion in-game.

**Feedback from target audience.**

After discussing my current development plan with several of my peers, I received feedback on parts of the game such as security and exploitability the game’s mechanics. I’ve made

**Technical Solution**

**Client Side**

**Game class attributes:**

class Game():

def \_\_init\_\_(self, ip, username):

self.username = username

self.width = 1300

self.height = 700

self.gameDisplay = pygame.display.set\_mode((self.width, self.height))

self.backgrounds = [

pygame.image.load("sprites/Background anime/frame0.gif"),

pygame.image.load("sprites/Background anime/frame1.gif"),

pygame.image.load("sprites/Background anime/frame2.gif"),

pygame.image.load("sprites/Background anime/frame3.gif"),

pygame.image.load("sprites/Background anime/frame4.gif"),

pygame.image.load("sprites/Background anime/frame5.gif"),

pygame.image.load("sprites/Background anime/frame6.gif"),

pygame.image.load("sprites/Background anime/frame7.gif")

]

self.scaled\_backgrounds = []

self.clock = pygame.time.Clock()

self.fps = 30

self.count = 100

self.music = pygame.mixer.music.load("power\_music.wav")

self.ip = ip

self.p2 = None

self.p = None

self.n = None

self.walls = [

Map(400, 400, 44, 228, "wall1.png"),

Map(600, 600, 44, 228, "wall1.png")

]

self.wall\_rect = [self.walls[0].rect, self.walls[1].rect]

self.wall\_img = pygame.image.load("wall1.png")

#self.items=[Collectables(1,"sprite.png")]

self.proj\_img = pygame.image.load("projectile1.png")

self.run = False

self.playerObj = None

self.secondPlayerObj = None

self.players = None

self.movement = False

self.events = None

self.background\_index = 0

self.player1\_sprites = ["sprite1.png", "right.png", "left.png"]

self.player2\_sprites = [

"sprite2.png", "player2right.png", "player2left.png"

]

self.loaded\_player1 = []

self.loaded\_player2 = []

self.arcade\_font = pygame.font.Font("Images/arcade.TTF", 12)

self.text\_font = pygame.font.Font("Images/arcade.TTF", 28)

self.black = (0, 0, 0)

self.white = (255, 255, 255)

self.yellow = (255, 255, 0)

self.green = (0, 255, 0)

self.blue = (0, 0, 255)

self.light\_blue = (173, 216, 230)

**Game methods:**

Loading sprites into the game:

def load\_sprites(self):

for i in self.player1\_sprites:

image=pygame.image.load(i)

self.loaded\_player1.append(image)

for j in self.player2\_sprites:

image=pygame.image.load(j)

self.loaded\_player2.append(image)

Scaling the backgroud images to fit the resolution of the game window:

def show\_username(self):

self.playerObj.username=self.username

username = self.arcade\_font.render(str(self.username), 0, self.light\_blue)

player2\_username=self.arcade\_font.render(str(self.secondPlayerObj.username), 0, self.light\_blue)

self.gameDisplay.blit(username, (self.playerObj.x,self.playerObj.y-20))

self.gameDisplay.blit(player2\_username, (self.secondPlayerObj.x,self.secondPlayerObj.y-20))

def background\_scale(self):

for background in self.backgrounds:

self.scaled\_backgrounds.append(pygame.transform.scale(background,

(self.width,self.height)))

Showing each player’s username above their player sprite:

Redraw function that redraws the window and updates the display output:

def redraw\_window(self):

self.gameDisplay.fill((255,255,255))

score1 = self.text\_font.render(str(self.playerObj.score), 0, self.white)

score2 = self.text\_font.render(str(self.secondPlayerObj.score), 0, self.white)

index=int(self.background\_index)

self.gameDisplay.blit(self.scaled\_backgrounds[index],(0,0))

self.gameDisplay.blit(score1, (20, 50))

self.gameDisplay.blit(score2, (self.width-40, 50))

self.playerObj.draw(self)

self.secondPlayerObj.draw(self)

time = self.text\_font.render(str(self.timer.time\_elapsed), 0, self.white)

self.gameDisplay.blit(time, (self.width/2, 20))

for bullet in self.playerObj.projectiles:

if bullet.collides(self)=="hit":

self.playerObj.score+=10

if bullet.should\_remove():

self.playerObj.remove\_projectile(bullet)

continue

bullet.draw\_bullet(self.gameDisplay,self.players)

for bullet in self.secondPlayerObj.projectiles:

if bullet.collides(self)=="hit":

if bullet.should\_remove():

self.secondPlayerObj.remove\_projectile(bullet)

continue

bullet.draw\_bullet(self.gameDisplay,self.players)

The main gameloop loads instantiates all the attributes and runs the all the methods in a while loop at a certain clock speed (depends on self.fps attribute).

def gameloop(self):

print(self.username)

self.load\_sprites()

self.background\_scale()

music=self.music

pygame.mixer.music.play(-1)

self.run = True

try:

self.n = Network(self.ip)

self.p= self.n.getP()

print(self.p)

print(self.p['player'])

self.playerObj = self.p['player']

self.timer = self.p['timer']

except:

print("Cannot connect to server")

self.movement=True

while self.run:

if self.background\_index>6:

self.background\_index=0

else:

self.background\_index+=0.30

self.p2 = self.n.send(self.p)

self.secondPlayerObj=self.p2['player']

self.timer = self.p2['timer']

self.players=[self.playerObj,self.secondPlayerObj]

self.events=pygame.event.get()

for event in self.events:

if event.type == pygame.QUIT:

self.run = False

pygame.quit()

self.playerObj.move(self.events)

self.redraw\_window()

self.show\_username()

if self.playerObj.dead==True or self.secondPlayerObj.dead==True:

self.run=False

self.endgame(self.playerObj.dead,self.playerObj.health,str(self.playerObj.username),self.secondPlayerObj.dead,self.secondPlayerObj.health,str(self.secondPlayerObj.username))

self.clock.tick(self.fps)

pygame.display.update()

After a round has ended, the user will be directed to the engame screen that displays the winner:

def endgame (self,player1\_dead,player1\_health,name1,player2\_dead,player2\_health,name2):

count=0

new\_font = pygame.font.Font("Images/arcade.TTF", 80)

player\_font = pygame.font.Font("Images/arcade.TTF", 60)

game\_over = new\_font.render(str("GAME OVER"), 0, self.black)

player1 = player\_font.render(str(name1 +" wins!"), 0, self.yellow)

player2 = player\_font.render(str(name2 +" wins!"), 0, self.green)

tie = player\_font.render(str("Match Tied"), 0, self.blue)

player1\_wins=False

player2\_wins=False

game\_tie=False

if player1\_dead==True:

player2\_wins=True

print("player 2 wins!")

elif player2\_dead==True:

player1\_wins=True

print("player 1 wins!")

if player1\_dead==False and player2\_dead==False:

if player1\_health>player2\_health:

player1\_wins=True

player2\_wins=False

if player1\_health<player2\_health:

player2\_wins=True

player1\_wins=False

else:

player1\_wins=False

player2\_wins=False

game\_tie=True

while True:

if count==12:

start\_check()

count+=1

self.gameDisplay.fill((255,255,255))

self.gameDisplay.blit(game\_over, ((width/2) - (200), (height/2)))

if player1\_wins==True:

self.gameDisplay.blit(player1, ((self.width/2) - (200), (self.height/2+100)))

if player2\_wins==True:

self.gameDisplay.blit(player2, ((self.width/2) - (200), (self.height/2+100)))

if game\_tie==True:

self.gameDisplay.blit(tie, ((self.width/2) - (200), (self.height/2+100)))

pygame.mixer.music.stop()

print("Game Over")

self.clock.tick(2)

pygame.display.update()

**Player class:**

class Player():

def \_\_init\_\_(self, x, y, sprite, player, direction, score):

self.x = x

self.y = y

self.player = player

self.projectiles = []

self.speed = 8

self.gravity = 8

self.sprite = sprite

self.direction = direction #1=right, 2=left

self.health = 100

self.display = sprite[direction]

self.bullet\_count = 2 #limits number of bullets per shot

self.dead = False

self.collision\_left = False

self.collision\_right = False

self.collision\_up = False

self.collision\_down = False

self.hitbox = pygame.Rect(self.x, self.y, 45, 68)

self.username = None

self.score = score

Redraw window:

def draw(self,game):

self.hitbox=rect = pygame.Rect(self.x, self.y, 45, 68)

image=pygame.image.load(self.display)

if self.player==1:

game.gameDisplay.blit(game.loaded\_player1[self.direction],(self.x,self.y))

elif self.player==2:

game.gameDisplay.blit(game.loaded\_player2[self.direction],(self.x,self.y))

rect = pygame.Rect(self.x, self.y, 45, 68)

rect2=pygame.Rect(game.walls[1].x,game.walls[1].y,25,25)

game.gameDisplay.blit(game.wall\_img,(game.walls[0].x,game.walls[0].y))

game.gameDisplay.blit(game.wall\_img,(game.walls[1].x,game.walls[1].y))

if rect.colliderect(game.walls[0].rect) or rect.colliderect(game.walls[1].rect):

if self.direction==1 and self.collision\_left==False:

self.collision\_right=True

self.collision\_left=False

self.collision\_up=False

self.collsion\_down=False

elif self.direction==2 and self.collision\_right==False:

self.collision\_left=True

self.collision\_right=False

self.collision\_up=False

self.collsion\_down=False

if (game.walls[0].y+20)>(self.y+40)<(game.walls[0].y) or (self.y+40)<(game.walls[1].y) and self.collision\_up==False:

self.collision\_down=True

self.collision\_left=False

self.collision\_right=False

self.collision\_up=False

elif (game.walls[0].y+30)<(self.y)<(game.walls[0].y+game.walls[0].height) or (self.y)<(game.walls[1].y+game.walls[1].height):

print("collided up")

self.collision\_up=True

self.collsion\_down=False

self.collision\_right=False

self.collision\_left=False

else:

self.collision\_right=False

self.collision\_left=False

self.collision\_up=False

self.collision\_down=False

new\_font = pygame.font.Font("Images/arcade.TTF", 28)

colour = (255,255,255)

health = new\_font.render(str(self.health), 0, colour)

if self.player==1:

game.gameDisplay.blit(health, (30, 20))

elif self.player==2:

game.gameDisplay.blit(health, (width-110, 20))

Move method:

def move(self,events):

projectile\_sound=pygame.mixer.Sound("laser.wav")

keys = pygame.key.get\_pressed()

'''Dont use pygame.key.pressed for projectiles because fps makes more than one bullet shoot at a time'''

if keys[pygame.K\_LEFT]:

if 0<=self.x and self.collision\_left!=True:

self.x -= self.speed

self.direction=2

self.display=self.sprite[2]

if keys[pygame.K\_RIGHT]:

#print(self.x)

if self.x<=width-50:

if self.collision\_right==False:

self.x += self.speed

self.direction=1

self.display=self.sprite[1]

if keys[pygame.K\_UP]:

if self.collision\_up==False:

if self.y>0:

self.y -= self.speed\*3

elif keys [pygame.K\_DOWN]:

if self.collision\_down==False:

if self.y <(height-50):

self.y += self.speed

for event in events:

if event.type == pygame.KEYDOWN:

if event.key==pygame.K\_SPACE:

if len(self.projectiles)<4:

projectile\_sound.play()

self.projectiles.append(Projectile(self.x, self.y, self.direction, 'projectile1.png', self.player))

if self.y <(height-80):

if self.collision\_down==False:

self.y+=self.gravity

Other Player methods:

def remove\_projectile(self,proj):

self.projectiles.remove(proj)

def damage\_taken(self):

if self.health<2:

self.dead=True

self.health-=10

def update\_username(self,username):

self.username=username

**Projectile class:**

class Projectile (Player):

def \_\_init\_\_(self,x,y,direction,sprite,player):

self.x=x

self.y=y

self.direction=direction

self.speed=35

self.sprite=sprite

self.player=player

self.shouldRemove = False

self.hit\_radius=50

self.collided=False

self.hitbox= pygame.Rect(self.x, self.y, 10, 10)

**Drawing projectile:**

def draw\_bullet (self, gameDisplay,players):

self.hitbox=pygame.Rect(self.x, self.y, 10, 10)

proj=pygame.image.load("projectile1.png")

gameDisplay.blit(proj, (self.x+10, self.y+20))

if self.x<width and self.x>0:

if self.direction==1:

self.x+=self.speed

elif self.direction==2:

self.x-=self.speed

else:

self.shouldRemove = True

for p in players:

p.bullet\_count=2

**Collision check:**

def collides (self,game):

for p in game.players:

if self.x<=p.x<(self.x+25) and (self.y-self.hit\_radius)<p.y<self.y+40 and self.player!=p.player:

self.shouldRemove=True

self.collided =True

print("collided player {}".format(p.player))

print(p.player)

p.damage\_taken()

if p.player==1:

return ("hit")

elif p.player==2:

return("hit")

**Removing bullet:**

def should\_remove(self):

return self.shouldRemove

**Map class:**

class Map():

def \_\_init\_\_(self,x,y,width,height,image):

self.x = x

self.y = y

self.height = height

self.width = width

self.image=image

self.rect = pygame.Rect(self.x,self.y,228,44)

def collision(self,player):

if (self.x-(self.width/2)<player.x<self.x+(self.width/2)) or(self.y-(self.height/2)<player.y<self.y+(self.height/2)):

return True

**Collectables class:**

class Collectable():

def \_\_init\_\_(self, item, sprite):

self.item = item

self.sprite = sprite

self.x = random.randint(0, width)

self.y = random.randint(0, height)

self.life = 300

self.hitbox = pygame.Rect(self.x, self.y, 20, 20)

self.correct\_position = False

def generate(self, wall):

self.x = random.randint(0, width)

self.y = random.randint(0, height)

self.hitbox = pygame.Rect(self.x, self.y, 40, 40)

if self.hitbox.colliderect(wall):

self.correct\_position = False

else:

self.correct\_position = True

if self.correct\_position == False:

self.generate(wall)

def display\_items(self, gameDisplay):

object = pygame.Rect(self.x, self.y, 40, 40)

self.life -= 1

pygame.draw.rect(gameDisplay, (99, 99, 99), object)

**Network Class:**

class Network:

def \_\_init\_\_(self, ip):

self.client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.server = ip

self.port = 5555

self.addr = (self.server, self.port)

self.p = self.connect()

def getP(self):

return self.p

def connect(self):

try:

self.client.connect(self.addr)

return pickle.loads(self.client.recv(2048 \* 2))

except:

pass

def send(self, data):

try:

self.client.send(pickle.dumps(data))

return pickle.loads(self.client.recv(2048 \* 2))

except socket.error as e:

print(e)

**Function that instantiates a game object when game is run:**

def start\_check(player):

host=socket.gethostname()

ip=socket.gethostbyname(host)#'192.168.1.225'

message=menu(False)

if message==True:

game=Game(ip,player)

game.gameloop()

**Login system using Tkinter and SQLite 3:**

with sqlite3.connect("playerdata.db") as db:

cursor =db.cursor()

cursor.execute ("CREATE TABLE IF NOT EXISTS player(username TEXT NOT NULL, password TEXT NOT NULL, highscores INTEGER)")

db.commit()

db.close()

class Login\_system():

def \_\_init\_\_(self,root):

self.root=root

self.username=StringVar()

self.password = StringVar()

self.new\_username=StringVar()

self.new\_pass=StringVar()

self.widgets()

self.start\_game=False

def login(self):

with sqlite3.connect("playerdata.db") as db:

cursor =db.cursor()

player\_search=("SELECT \* FROM player WHERE username=? and password=?")

cursor.execute(player\_search,[self.username.get(),self.password.get ()])

returned = cursor.fetchall()

if returned:

self.logframe.pack\_forget()

self.header['text'] = self.username.get() + '\n Logged In'

self.header["pady"] = 150

self.start\_game=True

self.root.quit()

start\_check(self.username.get())

else:

print("not found")

ms.showerror("Error","Account details not found. Please make sure the details are entered properly.")

def create\_account(self):

with sqlite3.connect('playerdata.db') as db:

cursor = db.cursor()

player\_search = ('SELECT \* FROM player WHERE username = ?')

cursor.execute(player\_search,[(self.username.get())])

if cursor.fetchall():

ms.showerror("Error","This username has already been taken. Please chose another one")

else:

ms.showinfo("Success","Account created successfully")

self.log\_frame()

store = 'INSERT INTO player(username,password) VALUES(?,?)'

cursor.execute(store,[(self.new\_username.get()),(self.new\_pass.get())])

db.commit()

def log\_frame(self):

self.username.set('')

self.password.set('')

self.header['text'] = 'Login'

self.createframe.pack\_forget()

self.logframe.pack()

def create\_frame(self):

self.new\_username.set('')

self.new\_pass.set('')

self.header['text'] = 'Create Account'

self.logframe.pack\_forget()

self.createframe.pack(fill="both", expand=True)

def widgets(self):

self.header = Label(self.root,text = 'Login',font = ('',35),pady = 10)

self.header.pack()

self.logframe = Frame(self.root,padx =10,pady = 10)

Label(self.logframe,text = 'username: ',font = ('',20),pady=20,padx=55).grid(sticky = W)

Entry(self.logframe,textvariable = self.username,bd = 5,font = ('',15)).grid(row=0,column=1)

Label(self.logframe,text = 'Password: ',font = ('',20),pady=20,padx=55).grid(sticky = W)

Entry(self.logframe,textvariable = self.password,bd = 5,font = ('',15),show = '\*').grid(row=1,column=1)

Button(self.logframe,text = ' Login ',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.login).grid()

Button(self.logframe,text = ' Create Account ',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.create\_frame).grid(row=2,column=1)

self.logframe.pack(fill="both", expand=True)

self.createframe = Frame(self.root,padx =10,pady = 10)

Label(self.createframe,text = 'username: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

Entry(self.createframe,textvariable = self.new\_username,bd = 5,font = ('',15)).grid(row=0,column=1)

Label(self.createframe,text = 'Password: ',font = ('',20),pady=5,padx=5).grid(sticky = W)

Entry(self.createframe,textvariable = self.new\_pass,bd = 5,font = ('',15),show = '\*').grid(row=1,column=1)

Button(self.createframe,text = 'Create Account',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.create\_account).grid()

Button(self.createframe,text = 'Go to Login',bd = 3 ,font = ('',15),padx=5,pady=5,command=self.log\_frame).grid(row=2,column=1)

root=Tk()

root.geometry("700x400")

login=Login\_system(root)

root.mainloop()

**Main Menu:**

def menu(start):

pygame.init()

width = 900

height = 550

display = pygame.display.set\_mode((width, height))

clock = pygame.time.Clock()

white=(255, 255, 255)

black=(0, 0, 0)

brown=(150,75,0)

red=(255, 0, 0)

yellow=(255, 255, 0)

blue=(0, 0, 255)

green=(0, 255, 0)

grey=(128, 128, 128)

music=pygame.mixer.music.load("fire.mp3")

pygame.mixer.music.play(-1)

backgrounds=[]

anime1=pygame.image.load("Images/tmp-0.gif")

anime2=pygame.image.load('Images/tmp-1.gif')

anime3=pygame.image.load('Images/tmp-2.gif')

anime4=pygame.image.load('Images/tmp-3.gif')

anime5=pygame.image.load('Images/tmp-4.gif')

anime6=pygame.image.load('Images/tmp-5.gif')

anime7=pygame.image.load('Images/tmp-6.gif')

anime8=pygame.image.load('Images/tmp-7.gif')

backgrounds.append(pygame.transform.scale(anime1,(width,height)))

backgrounds.append(pygame.transform.scale(anime2,(width,height)))

backgrounds.append(pygame.transform.scale(anime3,(width,height)))

backgrounds.append(pygame.transform.scale(anime4,(width,height)))

backgrounds.append(pygame.transform.scale(anime5,(width,height)))

backgrounds.append(pygame.transform.scale(anime6,(width,height)))

backgrounds.append(pygame.transform.scale(anime7,(width,height)))

backgrounds.append(pygame.transform.scale(anime8,(width,height)))

def process\_text(message, font, size, color):

new\_font = pygame.font.Font(font, size)

edited = new\_font.render(message, 0, color)

return edited

font = "Images/arcade.TTF"

n=0

def main\_menu(n,start):

tracker=["start","instructions","highscores","quit"]

pointer=0

menu=True

selected="start"

while menu:

if n>6:

n=0

else:

n+=1

for event in pygame.event.get():

if event.type==pygame.QUIT:

pygame.quit()

quit()

if event.type==pygame.KEYDOWN:

if event.key==pygame.K\_UP:

selected="start"

if pointer>0:

pointer-=1

selected=tracker[pointer]

if event.key==pygame.K\_DOWN:

if pointer<3:

pointer+=1

selected=tracker[pointer]

if event.key==pygame.K\_RETURN:

if selected=="quit":

pygame.quit()

quit()

if selected=="start":

print("start")

start=True

return start

display.fill(grey)

display.blit(backgrounds[n],(0,0))

font\_size=32

if selected=="start":

text\_start=process\_text("START", font, font\_size, white)

else:

text\_start = process\_text("START", font, font\_size, yellow)

if selected=="quit":

text\_quit=process\_text("QUIT", font, font\_size, white)

else:

text\_quit = process\_text("QUIT", font, font\_size, yellow)

if selected == "highscores":

text\_highscores=process\_text("HIGHSCORES", font, font\_size, white)

else:

text\_highscores =process\_text("HIGHSCORES", font, font\_size, yellow)

if selected =="instructions":

text\_instructions = process\_text("INSTRUCTIONS",font,font\_size,white)

else:

text\_instructions = process\_text("INSTRUCTIONS",font,font\_size,yellow)

start\_rect=text\_start.get\_rect()

quit\_rect=text\_quit.get\_rect()

highscores\_rect=text\_highscores.get\_rect()

instruct\_rect=text\_instructions.get\_rect()

pos=width/2

display.blit(text\_start, (pos - (start\_rect[2]/2), 100))

display.blit(text\_instructions, ( pos- (instruct\_rect[2]/2), 140))

display.blit(text\_highscores, (pos - (highscores\_rect[2]/2),180))

display.blit(text\_quit, (pos - (quit\_rect[2]/2), 220))

pygame.display.update()

clock.tick(5)

pygame.display.set\_caption("Main Menu")

return main\_menu(n,start)

pygame.quit()

**Server:**

totalConnections = 0

port = 5555

host = socket.gethostname()

IP = socket.gethostbyname(host)

server = IP

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

try:

s.bind((server, port))

except socket.error as e:

str(e)

s.listen(2)

print("Waiting for a connection, Server Started")

timer = Timer()

all\_data = [

{

'player': Player(0, 0, ["sprite1.png", "right.png", "left.png"], 1, 1,

0),

"timer": timer,

#"collectables": items

},

{

'player':

Player(100, 100,

["sprite2.png", "player2right.png", "player2left.png"], 2, 2,

0),

"timer":

timer,

#"collectables": items

}

]

'''talk about why parallel data sets not sent due to synch issues and buffer needed'''

timerHasStarted = False

def threaded\_client(conn, player):

global totalConnections, timerHasStarted, timer

conn.send(pickle.dumps(all\_data[player]))

reply = ""

print("TIMER STARTED:" + str(timer.has\_started))

while True:

try:

data = pickle.loads(conn.recv(2048 \* 2))

all\_data[player] = data

if not data:

print("Client Disconnected")

totalConnections -= 1

break

else:

all\_data[0]['timer'] = timer

all\_data[1]['timer'] = timer

print("TIMERS =============")

print(timer.time\_elapsed)

print(all\_data[0]['timer'].time\_elapsed)

print("=========================")

if totalConnections == 2:

if not timerHasStarted:

print("STARTING")

timer.start()

timerHasStarted = True

if player == 1:

reply = all\_data[0]

else:

reply = all\_data[1]

conn.sendall(pickle.dumps(reply))

except:

break

print("Lost connection")

conn.close()

currentPlayer = 0

while True:

conn, addr = s.accept()

print("Connected to:", addr)

totalConnections += 1

start\_new\_thread(threaded\_client, (conn, currentPlayer))

currentPlayer += 1

def error():

print("You can't do that")

**Improvements:**

Make the markscheme titles the titles

Hierarchy chart

Explanation for flow diagram – for each part (add title)

Update – changes made in flow diagram + why has It changed

Explain – Can’t normalise database due to simplicity. Explain (1,2,3NF)

Why normalise? Why it won’t work here?

Data dictionary (each field)

Identification of main algorithms – Title

Purpose, Pseudocode, Explanation

Hashing – instead of caesar cipher (explain)

File structure and organisation – how files are saved (indexed storage – access to data from primary key), sprites and images stored.

UML diagram (no links)

Justify analysis from design methods – why you used that method.

Technical solution:

Use recursive merge sort for highscores (faster) – Just when displaying (Select all).

1. <https://www.usgamer.net/articles/top-10-biggest-grossing-arcade-games-of-all-time> [↑](#footnote-ref-2)
2. https://upload.wikimedia.org/wikipedia/en/a/a0/Digdug.png [↑](#footnote-ref-3)
3. <https://store.steampowered.com/app/403400/ARCADE_GAME_SERIES_DIG_DUG/> [↑](#footnote-ref-4)
4. https://i.ytimg.com/vi/5em8leAa8Po/hqdefault.jpg [↑](#footnote-ref-5)
5. <https://www.youtube.com/watch?v=QTPy4iO-9Do> [↑](#footnote-ref-6)
6. https://cdn3.dualshockers.com/wp-content/uploads/2019/05/Multiplayer-Abraham-Lincoln.jpg [↑](#footnote-ref-7)
7. <https://www.howtogeek.com/241285/why-video-games-make-you-feel-sick-and-what-you-can-do-about-it/> [↑](#footnote-ref-8)
8. <https://techsore.com/best-programming-language/> [↑](#footnote-ref-9)
9. <https://excelwithbusiness.com/blog/say-hello-world-in-28-different-programming-languages/> [↑](#footnote-ref-10)
10. <https://opensource.com/sites/default/files/styles/image-full-size/public/lead-images/python3-game.png?itok=jG9UdwC3> [↑](#footnote-ref-11)
11. <https://www.pubnub.com/blog/socket-programming-in-python-client-server-p2p/> [↑](#footnote-ref-12)
12. <https://btbusiness.custhelp.com/app/answers/detail/a_id/7629/~/how-do-i-find-my-ip-address%3F/c/5099/> [↑](#footnote-ref-13)
13. <https://wiki.python.org/moin/GlobalInterpreterLock> [↑](#footnote-ref-14)
14. <https://ezgif.com/> [↑](#footnote-ref-15)