A Twin-Stick Shooter Game

Rory Byrne

Computer Science NEA

Contents

1. Analysis

1.1 Problem Definition

Playing video games is an amazing way to bring people together and have lots of fun in groups. Marcus, a friend of mine, has always loved doing this. He used to very regularly play small arcade-style video games with his friends and compete against them to get the highest scores. They would spend many evenings together playing things like Snake and Pac-Man where it is designed for competitive play, trying to outdo each other and chatting and bickering, and these are some of his best memories. But, as time has gone on, these games have gone out of fashion, evenings like this have become rarer, and he really wants to bring this back to his friend group. Therefore, he has asked me to create an arcade-style game for him and his friends to play together and compete against each other in.

His favourite arcade-style games were always twin-stick shoot 'em up games, which are video games where a character is controlled typically using one joystick for movement and a second joystick for shooting and they have to shoot at large hordes of enemies while avoiding them or their shots, and it is this type of game that he has asked me to create. His gameplay idea is that you are a character with a gun who has to shoot at hordes of enemies before they can reach you, racking up your score with each kill until you are eventually defeated. He also wants it to be a simple and accessible game that anyone can pick up and play, no matter their experience with video games beforehand. To make it more fun, he would like me to add lots of different power ups that are randomly dropped by enemies, lots of different enemy types, and he'd like for the game to gradually get harder as you play.

1.2 Client Interview

In order to get a better idea of what I need to create, I decided to conduct an interview with my client, Marcus. I will be asking specific questions that will help me know what the most important features are and what parts I won't need to focus on as much in order to eventually gather a collection of important objectives that I will aim to achieve with my solution.

1.2.1 Interview

The questions I asked are in bold and Marcus' responses are displayed underneath.

How often do you play arcade style games?

I used to play similar games all the time with my siblings and friends but as time has gone on, it has become much harder to find good ones that are worth playing or the time to play them.

What are the issues you have with existing games?

The existing games either cost money, have so many ads, or just aren't made for me. They have no multiplayer or scoreboard options, overwhelming graphics, and terrible control systems.

What would you like the gameplay to be like?

It should be a simple game where the player has a gun and has to shoot waves of enemies before they can reach them. I'd like it if the game was endless and would scale in difficulty as time goes on and there should be some different types of enemies that can spawn, with some harder enemies

that only appear later in the game. The enemies should also have random chances to drop power-ups or something like that that give the player boosts and make it more fun. I also like the idea of there being some sort of obstacles that can block your path and your bullets to make it harder and more interesting.

What features would you like the game to have?
The game should definitely have a score system and leaderboard system to compare your scores to your friends'. A two player mode would be great as a lot of fun memories can be made when playing games like this together with someone but it is not a huge priority compared to the single player mode. I'd also love it if people could customise their own characters and have their own designs that people can recognise them for.

How do you think a character customisation system should be implemented? *(this question was not planned but I wanted to know more about his ideas from the last question)*
I think the more control over the character the better. I've always thought it would be really cool if you could actually draw parts of your character yourself. That's definitely something I'd like to see in a game.

What requirements are your highest priority?
For me it is a high priority for the game to be simple to understand and pick up, not overly complex. I think the game should be as accessible as possible and with no limit to who can play it. It's also a high priority to make it easy to compete with friends on the leaderboard.

Do you want a two player option for the game and if yes, how important is online multiplayer over local?
Yes, as said earlier I think a two player option would be great. Local multiplayer with two people playing on one computer could be really fun. Online multiplayer isn't too necessary because I prefer to play games with people in real life but if it can be done then it would be great too.

What platform and controller do you want the game to be made for?
PC and keyboard is the only thing it needs to run on.

How important are the visuals of the game to you?
I really like when games have simple yet visually appealing graphics, so I think it should have graphics that are easy and quick to understand but still look good.

What would make a good user interface for the game?
The user interface should also be kept simple and quick to start playing, I just want to press play and be in the game without having to go through a bunch of menus or cutscenes. It should also look good though.

Would a tutorial be necessary?
I think that if the user interface and graphics are created properly, a tutorial shouldn't be necessary and often tutorials just get in the way of playing.

Do you want the game to have sound?
The game should definitely feature sound effects. Music would be great too but it's not too important.

Does anyone who would be playing the game have any disabilities I can accommodate for in the program?
No but the game could include some accessibility features, like making it so that both sound effects and visuals are used for when a player gets hit, and maybe colour blind options if colour is a big part of the game.

1.2.2 Analysis of Interview

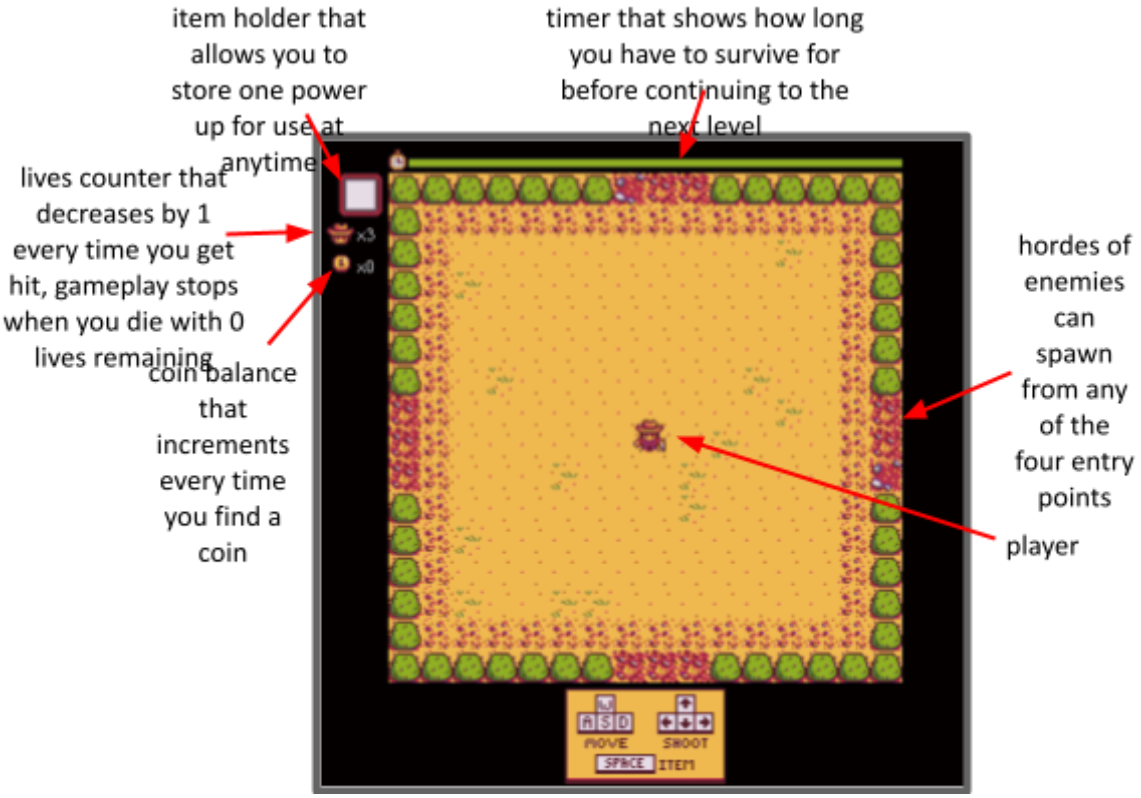
In summary, the game will be made for PC and keyboard. It will feature a simple concise menu that can quickly let you jump straight into the game. A big aspect of the game's design to make it fun will be the ability to compete with friends and see their scores and let them see yours, so I will make sure that leaderboards and high scores are very visible that even if the player doesn't directly look at them, it still subtly goes into their head what place they are in and who has the highest score. The game and user interface will have simple graphics and simple designs that can be quickly understood by anyone and simple sound design that makes understanding the game easier. This is another big aspect of the game's design, that it is easy to understand and concise, for example it should not take much effort for anyone who quickly looks at their friend's screen to know what is going on and how well they are doing in the game. The game will feature options for the players to customise and create their own individual character in order to differentiate between players and give players some creativity that can make playing the game more fun. My client mentioned that he likes the idea of being able to actually draw parts of the character, so I will try to refine this idea and find something that can satisfy him. Having leaderboards and character customisation like this will require some sort of account and login system so that there is a username to put onto the leaderboards and so that the custom character of a player can be saved and used again. The game itself will be endless and have waves of enemies from every side that move towards the player, scaling in difficulty as time progresses. It will feature multiple enemy types that have a range of difficulties, maybe tailored by how fast they move or how they move towards the player. The game will ideally have a two player mode, online or local, that allows two people to play and get a score together, further improving the games ability for playing with friends competitively or cooperatively. However, it was said that two player is not a huge priority so the single player mode will be prioritised.

1.3 Similar solutions

There are many video games and arcade games similar to what my client is looking for but they all lack the important features, are hard to access, or are on sketchy websites filled with intrusive ads. The game I hope to make should solve these problems and be a fun and accessible video game that brings people together and allows Marcus to easily play with and compete against his friends. Looking at the similar games will help me understand what I need to do to achieve this.

1.3.1 Journey of the Prairie King

One game I found that is similar to his idea is Journey of the Prairie King[1], a minigame in the larger game of Stardew Valley[2]. This is a cowboy themed top down shoot 'em up game resembling twin-stick shooters (using WASD and the arrow keys instead of joysticks) where the protagonist must survive large hordes of enemies for a set amount of time by avoiding and shooting at them. The game consists of a variety of enemies that have different movement patterns and speeds, a variety of items that provide different benefits to the player, a few different stages containing curated levels and boss fights at the end of them, and an occasional shop that pops up in some levels to purchase upgrades with the coins you've collected.



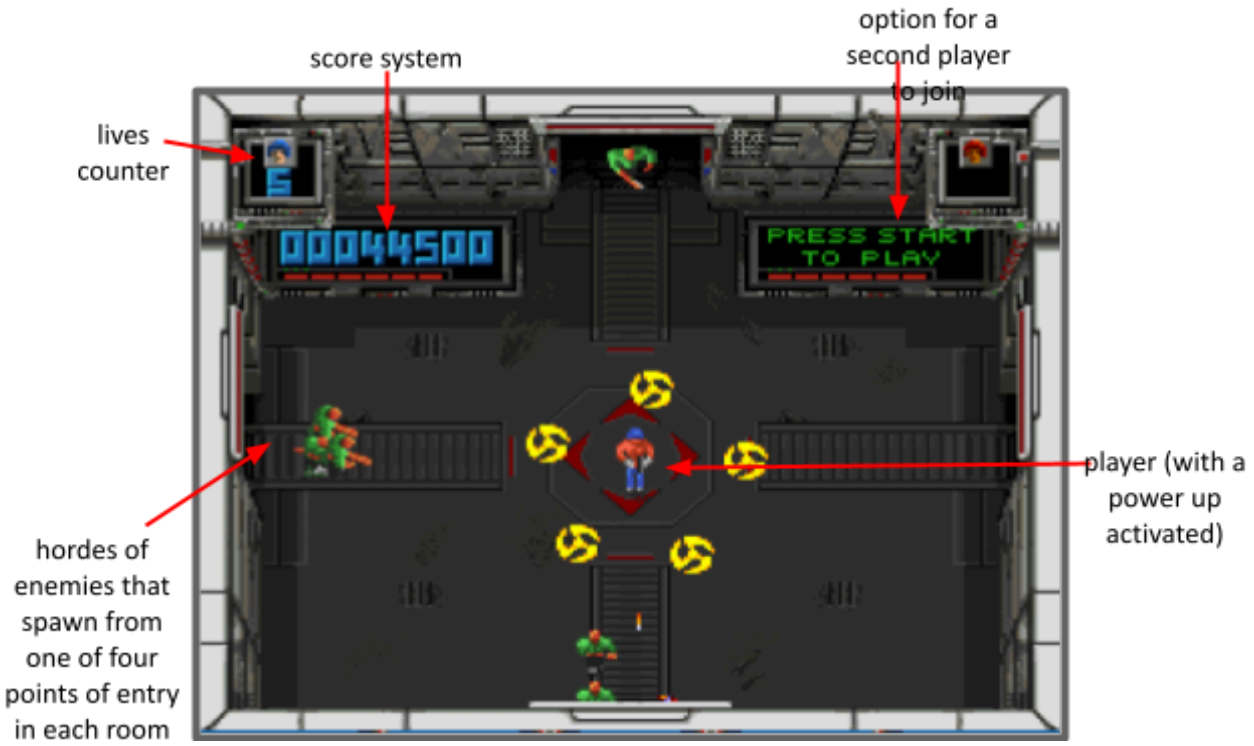
Many features of this game are similar to the desirable features of the game I am aiming to create, for example the character movement and shooting in 8 directions, all the unique power ups that are dropped randomly by enemies to aid players and add excitement, all the different enemy types that have different movement patterns and speeds and health bars, a life system where each player has a limited number of lives that end the game once they run out, appealing and uncomplicated graphics, intuitive controls and simple gameplay.

The huge disadvantages to this game is that it does not have a score system or a leaderboard, it does not support multiplayer, it has a story and is not endless, and it is not very accessible at all as it is a minigame contained within another game. These are all large aspects of the defined problem that I aim to include in my solution.

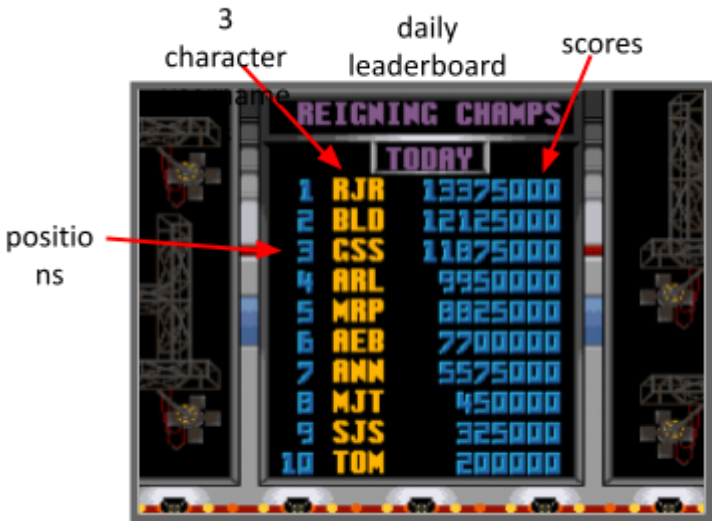
+	-
• similar gameplay and controls system to what I am trying to achieve • a variety of enemies and items • nice, consistent, and accessible art style	• no score system or leaderboard • no customisation options • not easy to access • no multiplayer support • no statistics or saving to the device
What I will need to include: • a score system and leaderboard to entice people to play • options for customising and creating a unique character • it is easy to access and quick to start playing • a multiplayer mode for playing together with friend's • a save system for data about the players that also saves statistics, such as the number of enemies killed, that the user can view	

1.3.2 Smash TV

Another similar game I found is Smash TV[3], a 1990 arcade game that has now been ported to many other systems[4]. Smash TV is a twin-stick shoot 'em up game where the protagonist ploughs through hordes of enemies in order to get the highest score possible and to progress to later levels. This game is made up of a few stages containing simple levels and boss fights at the end of them with a variety of different enemy types of scaling difficulty and a range of unique power ups.



This game does have many of the features I am looking to include, most importantly the score system and leaderboard as well as two player functionality.



The game however is not endless and features a story, it has no customisation options, the graphics can be overwhelming or difficult to understand, and it is very hard to find pc versions of the game (the ones you do find are on sketchy websites with lots of ads).

+	-
• similar gameplay and controls system to what I am trying to achieve • a variety of enemies and items • a score system and leaderboard • ability for a second player to join	• gameplay is not endless • no customisation options • no statistics or saving to the device • not at all easy to find or access • lots of ads when you do find a version of it • confusing and overwhelming graphics
What I will need to include: • endless gameplay • options for customising and creating a unique character • it is easy to access and quick to start playing • no advertisements • a save system for data about the players that also saves statistics, such as the number of enemies killed, that the user can view • simple, neat, consistent, and accessible art style	

From looking at these games and seeing what they do successfully and unsuccessfully, I now know more about what my game needs to include and what would make it fun.

1.4 Acceptable Limitations

One feature I probably will not be able to implement is an online two player game mode. Although I have experimented with and successfully created client server models previously in Python where one computer temporarily acts as a server that both computers are clients for, making an online two player game would be a whole project of its own and I want to and have been asked to create more than just the game. If I do find I have plenty of time near the end of development I may try adding in online multiplayer to the game, however, I don't think I will have enough time for it and there are

plenty of things me and my client think are more important such as local multiplayer which I will focus on implementing.

Another feature I may not be able to implement is an online database and leaderboard system. It wasn't specifically requested by my client but until I researched how they could be implemented I assumed it would be a nice easy feature I'd definitely have. From what I found though, it would require me purchasing expensive monthly subscriptions to database servers or using dodgy free trials that may end before the project is completed. An online database also wouldn't really require any more skill to implement than a locally stored database, maybe even less because lots of the features would already be handled by the server. So, I will probably stick to using a locally stored database and all leaderboards and accounts will be local to the device.

1.5 What makes a game fun?

I also needed to research how to make a game that is actually 'fun' and that people want to play.

To do this I searched online for 'What makes an arcade game fun' and spent time reading the results and picking the most useful ones[5][6]. Making it fun is an unstated but expected objective when creating a game of the type I will be creating. The main reasons I found through my research and experience in why people find arcade-style games fun and addicting are that they are easy to pick up, grant instant gratification, allow for skill improvement, and leave space for competitiveness.

Easy to pick up:

The controls, rules, and objectives are typically straightforward and don't take much to pick up and understand. This makes the games accessible to people of any skill level and means having fun isn't limited to people with skill.

Instant gratification:

You can immediately jump in and start playing with no need for long tutorials or setups. There are no consequences for playing other than getting to play as they require very little commitment. This makes them addicting and great for quick sessions during any sort of break.

Skill improvement:

The player has to feel that they can always improve their skills and get better scores. This could be improvements in their timing, precision, strategy, or just luck. If it feels impossible and they don't see any room for improvement then they will have no motivation to play.

Competitivity:

The games allow for competitiveness between players. There are always high scores to beat or people to show off your skills to. This is one of the things that drive people to do well and what makes doing well so thrilling for the player and anyone around them.

These are all very important features I will have to try to provide a balance of to players if I want the game to be fun.

1.6 Objectives

1. Upon opening the game, the user will be greeted with a login screen
    1.1. The initial screen allows the user to either create a new account or log into a pre-existing account through a username and a pin
    1.2. If, when creating an account, the username is already taken or, when logging in, the username is not recognised or the pin is incorrect, the user will be informed
2. Once logged in, there is a simply designed and understandable* main menu with buttons to access each important feature of the program
    2.1. The game needs to make it easy to get right in and start playing, so the single player button will be accentuated
    2.2. There will be a smaller leaderboard on the main menu that it isn't blocked by a button so the user can quickly check the leaderboard before starting a game
    2.3. A small image of the user's custom character will be visible
3. The user can play single player by clicking the single player button
    3.1. The user can move and shoot from their character in 8 directions
    3.2. Hordes of enemies randomly spawn from one of four sides
    3.3. As the user progresses in the game, the enemies that spawn can be of harder types***
    3.4. After every few hordes of enemies, a small obstacle spawns at a random location that blocks both the player and enemies
    3.5. The user is able to store 1 item and use it at any point in the game and if the user runs into an item when their item spot is already filled it is automatically used
    3.6. Killed enemies have a random chance of dropping items
    3.7. Lots of different item types** that all do very different things to help the player
    3.8. The game can be paused
    3.9. When the player has run out of lives and the game ends, the user's score is displayed as well as an updated leaderboard
    3.10. The user can either return to the menu or play again through buttons
4. There is a two player game that allows two user accounts to play with each other and have combined scores
    4.1. Both players can still move and shoot in 8 directions
    4.2. Enemy spawning is more frequent than in the single player game
5. The user can view the leaderboards by clicking the leaderboard button
    5.1. The user can switch between viewing the single player leaderboard, two player leaderboard, statistics leaderboard, and personal statistics
    5.2. The single player leaderboard tab displays, in descending order, the top 10 single player scores and the username of the player who achieved it
    5.3. The two player leaderboard tab displays, in descending order, the top 10 two player scores and the usernames of the players who achieved it
    5.4. The statistics leaderboard tab displays the top 3 players in one or many statistic categories and a display of their custom character
    5.5. The personal statistics tab lets the user see their personal statistics such as enemies killed, games played, items used, bullets shot etc.
    5.6. The user's personal scores are highlighted if they are on the leaderboard
6. The user can view and customise their character by clicking the customise character button
    6.1. The user can completely customise their character, being able to draw some aspect of the character using a simple drawing application
    6.2. Some colours of this drawing application are initially locked until unlocked through achievements, e.g. killing 500 enemies
    6.3. Once created, the user can save their character, putting the character into a useable format and updating their character on the database and leaderboards
7. The user can view and change the settings by clicking the settings button
    7.1. The user can change the volume of the game
8. The game is consistent and accessible throughout
    8.1. Everything in the game is in a neat, consistent, and accessible art style

8.2. Plenty of sounds to provide audio feedback like when you press a button, kill an enemy, pickup an item, and lose a life
8.3. Plenty of visual cues to provide visual feedback like when you hover over a button, hit an enemy, or use an item

*simple colours, legible text buttons, clear icons, buttons are displayed hierarchically with the most important buttons biggest and at the top

**Table of ideas of items dropped by enemies:

Name	**Description**
Life	increases life count by 1
Boots	temporarily increases movement speed
Machine gun	temporarily increases rate of fire
Shotgun	temporarily shoot bullets in a 3-way spread but with a slower rate of fire
Invulnerability	temporarily make the player invulnerable to any damage
Stun bomb	all enemies stop moving for a few seconds but can still damage and be damaged
Bomb	clear all enemies from the screen, they don't drop any items
Anything	2/3 chance you get a random item from the list 1/3 chance your gun temporarily shoots bubbles

These items can also be operating simultaneously, so for example you could have the shotgun and the machine gun and invulnerability running at the same time giving you fast firing in a 3-way spread and a temporary block to all damage.

***Table of ideas of enemies and ideas for their level of difficulty:

Name	**HP**	**Description**	**Level**
Simple enemy	1	slow predictable movement somewhat towards the player spawn in medium sized hordes from the sides	1
Flying enemy	1	moves directly towards the player spawns alone and from the sides	3
Spike enemy	3, 7	picks a random location on the inner map, runs towards it and then turns into a spike hazard 2 health when running, 7 when it becomes a spike spawns alone and from the sides	1
Fast enemy	2	fast but predictable movement spawns alone and from the sides	2
Grenade enemy	1	runs to the centre and explodes into 8 shrapnel projectiles, one firing in each direction spawns alone and from the sides	4

Unpredictable enemy	2	fast and unpredictable movement, affected by randomness spawns alone and from the sides	5
Tough enemy	3	very slow predictable movement spawns in large hordes and from the sides	6
Bomb enemy	1	normal movement, when shot they explode after a few seconds, damaging you and enemies	7

Not all of these items or enemies will be in my final game as they are only provisional ideas. A lot of them would probably be boring or not fun and very challenging to implement, which I will discover and think further about when it comes to that part of the project.

1.7 Technologies

To create my game I have decided to use the Python programming language[7] and the Integrated Development Environment (IDE) Visual Studio Code[8]. I have chosen to use Python and in this IDE because I have been programming in it for many years now and no time will need to be put into learning a new language or learning to navigate a new IDE.

Visual Studio Code is a good IDE to use because it has a very intuitive design with built in colour coding and indentation. It has its own auto-fill and error highlighting which will save a lot of time and also defines your variables when you hover over them. It is very popular and therefore has plenty of documentation and a large online presence for any type of help.

Python is an object-oriented, event-driven, and very popular programming language. This allows me to create classes for players and enemies and create functions for menus and the popularity means that there is plenty of documentation for it too and videos and forums online to help. It also means that there are countless libraries that I can use, including Pygame which will become very useful.

The Pygame library[9] is an impressive library designed for creating games in Python and is almost necessary for game making so I will be using it a lot. It is very popular too and therefore also has plenty of documentation and videos and help forums online. I will need to use the libraries math and random as well for a lot of calculations and for all the random aspects that come with a game like this.

For saving to files I will use the Python JSON (JavaScript Object Notation) library[10]. This allows me to write Python objects to files and then read them and use them already in their correct syntax.

For my database I will use the Python sqlite3 library[11]. This library will allow me to create a database file stored on the device running the game that I can query and update inside my Python code.

For all the artwork in the game I will be using Aseprite[12], a very intuitive and well designed pixel art creation software that makes making pixel art very simple. It has a lot of systems built in for making sprites and sprite animations and for exporting your art into usable formats which will all be necessary.

For some of the sounds in the game I will be using FamiTracker[13], a tool for making music using sound systems similar to the systems on old game consoles. This will allow me to easily create little tunes or sound effects that are reminiscent of old arcade games.

Finally, for almost all the diagrams in this document I will be using Lucidchart[14], an intelligent diagram application. I find it really easy and satisfying to use and it comes with plenty of extensions and support for creating specific diagrams such as class diagrams.

1.8 Development methodology

I believe that the best methodology for the development of this project and the methodology I have chosen to use as my approach is the agile methodology. This is a methodology that anticipates flexibility and is best when change and deadlines are a big part of the development, which is the case for this project; I may find that some aspects or features of the game may not be feasible within the scope of the A-level course or within the time constraints given and it's also not unlikely that some things are just not that fun, don't add anything beneficial, or create unnecessary complexity when tested during prototyping or when implemented that were overlooked or not thought about enough in other phases. Therefore, change is very likely and a rigid approach to the development would not be appropriate. It is also a methodology where the client is closely involved and I can receive frequent feedback on work I'm doing and decisions I might make and then work from that feedback and towards mutually decided goals instead of just from what was decided at the beginning. This methodology also encourages decomposition of the project, breaking it down into small manageable chunks that can be completed quickly or at least easily comprehended and worked on. This is especially suitable for creating a game where there are so many different aspects that require different solutions that cannot be tackled simultaneously and breaking it down into smaller chunks is almost a requirement.

1.9 Prototyping

1.9.1 Player Movement

I thought the first thing I should prototype would be a character and character movement. At this point I had decided to go for pixel art for all art in the game, this is because it is easy to create so I don't have to spend much time creating the game art, and, I believe the style is neat, accessible, and easy to keep consistent, as per my brief. I furthermore decided on 8x8 pixel art for the game, so most sprites in the game will be 8 pixels wide by 8 pixels tall and everything will fit on a grid of 8 pixel by 8 pixel squares. This makes the art even easier to create, gives the game a nice and cute aesthetic that I believe fits with the brief of needing to be simple and accessible, and, when it comes to making the character customisation, having a small resolution to work with will make it much easier to let the user draw their own aspects of their character.

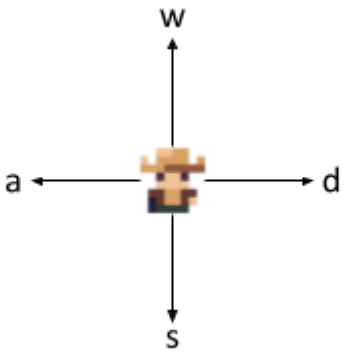
As Pygame works with the computer's pixel sizes and needs all coordinates and lengths to be in terms of screen pixels, I needed to create a constant that would define the pixel size for all my art. This would allow me to convert pixel sizes from my art into computer pixel sizes so that I could create all the objects and sprites with their relative sizes in pixels and, if I make sure every coordinate and distance is used in integer terms of this constant, everything in the game should have a consistent pixel size.

Below is the code for some of my constants defined for the prototype:

```
# dimensions for the screen in computer pixels
SCREEN_SIZE = (SCREEN_WIDTH, SCREEN_HEIGHT) = (720, 720)
# desired dimensions of the pixel art on screen
SCREEN_PIXEL_SIZE = (SCREEN_PIXEL_WIDTH, SCREEN_PIXEL_HEIGHT) = (144, 144)
# size of a pixel necessary to achieve desired pixel dimensions
PIXEL_RATIO = SCREEN_WIDTH / SCREEN_PIXEL_WIDTH
# size of an 8x8 sprite in computer pixels
SPRITE_SIZE = (SPRITE_WIDTH, SPRITE_HEIGHT) = (8 * PIXEL_RATIO, 8 * PIXEL_RATIO)
```

Linking the pixel ratio to the screen size allows me to change the screen size and have the ratio change accordingly. Now I was able to define a sprite size by multiplying the desired size in pixels by the pixel ratio to get a constant that defines how large an 8x8 sprite is in computer pixels.

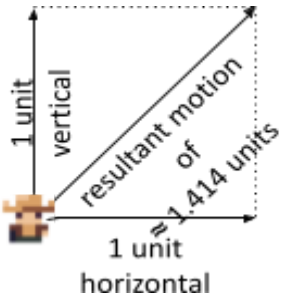
After this, it wasn't that difficult in Pygame to get a simple 8x8 character sprite I designed in Aseprite to display to the screen and start moving using the WASD keys. To get the image into pygame, I exported all the Aseprite designs as pngs and put them into a new folder called images contained in the same folder as the code. Then, you can use Pygame inbuilt functions to load the image from the files into a variable and scale it to the desired size. I also made the player look where they are moving using a changing image depending on which button they are pressing.



On the left shows a diagram of the simple movement system using a player class where if you press the W key, the character moves 1 unit upwards each frame it is held down, if you press the A key, the character moves 1 unit left and so on.

A problem I encountered with this basic movement system is that diagonal movement of the sprite appeared to be faster than side to side or up and down movement.

This is because when pressing two keys that aren't opposite each other and moving diagonally, the character will move 1 unit horizontally and 1 unit vertically.



This movement in two dimensions means that the character ends up covering more distance when moving diagonally, and when calculated using Pythagoras' theorem, equates to a distance of roughly 1.414 units instead of 1 unit:

$$\sqrt{1\ unit^{\,2} + 1\ unit^{\,2}} = \sqrt{2\ units} \approx 1.414\ units$$

For any n amount of movement, the diagonal movement equates to $n\sqrt{2}$:

$$\sqrt{n^2 + n^2} = \sqrt{2n^2} = n\sqrt{2}$$

It is really not desirable in a game to have the diagonal movement be faster, you want movement to be normalised and for it to have the same magnitude as any other direction.

In order to fix this we must first check for when the character is moving diagonally and if they are, you simply divide both the vertical and horizontal movements by $\sqrt{2}$.

So for any amount of movement n, the diagonal movement will also equate to n:

$$\sqrt{\left(\frac{n}{\sqrt{2}}\right)^2 + \left(\frac{n}{\sqrt{2}}\right)^2} = \sqrt{\frac{n^2}{2} + \frac{n^2}{2}} = \sqrt{n^2} = n$$

There was also a problem with the changing frame rate in Pygame. If you don't define a frame rate, the game runs as fast as possible, executing each line of code in turn whenever the program can and reaching frame rates upwards of 1000 fps. This frame rate is also not constant. In a game where time

is based on frames and the player moves one unit per frame, a non-constant frame rate alters the speed of event timings which is unacceptable in a video game and frame rates upwards of 1000 fps is way more than is necessary.
To fix this, you can define a maximum fps for the program to run using the Pygame clock object and its .tick() method.

```
FPS = 60 # maximum FPS
PLAYER_SPEED = 0.44 * PIXEL_RATIO * 60 / FPS # player speed from testing
```

I also linked the player's speed to the FPS constant and the pixel ratio so that it is now in terms of the amount of pixels travelled per frame, meaning that if I decide to change the FPS or the screen size in the future, the player's speed will change accordingly. I first multiplied by 60 because I found the speed when testing at 60FPS .

Character movement prototype code, using the constants defined in the code above:

```
import pygame
from math import sqrt

pygame.init()
screen = pygame.display.set_mode(SCREEN_SIZE)
clock = pygame.time.Clock()

character_front = pygame.transform.scale(pygame.image.load("images/sprite
front.png").convert_alpha(), SPRITE_SIZE) # the images up to the 8x8 pixel size
character_back = pygame.transform.scale(pygame.image.load("images/sprite
back.png").convert_alpha(), SPRITE_SIZE)  # .convert_alpha() preserves transparency
character_left = pygame.transform.scale(pygame.image.load("images/sprite
left.png").convert_alpha(), SPRITE_SIZE)
character_right = pygame.transform.scale(pygame.image.load("assets/sprite
right.png").convert_alpha(), SPRITE_SIZE)

class Player():
    def __init__(self, start_pos):
        self.image = character_front
        self.pos = pygame.math.Vector2(start_pos)
        self.speed = PLAYER_SPEED

    def user_input(self):
        self.velocity_x = 0
        self.velocity_y = 0

        # returns a large list of boolean values of whether each key is pressed
        keys = pygame.key.get_pressed()

        if keys[pygame.K_w]: # pygame constant of the index of the W key in the keys list
           # y is the pixels from the top of the screen, so -speed to move upwards
            self.velocity_y -= self.speed
            self.image = character_back
           # player's image changes depending on the direction they are moving
        if keys[pygame.K_s]:
            self.velocity_y += self.speed
            self.image = character_front
```

```
        if keys[pygame.K_a]:
            self.velocity_x -= self.speed
            self.image = character_left
        if keys[pygame.K_d]:
            self.velocity_x += self.speed
            self.image = character_right

       # if player is moving in both x and y (diagonal),
        if self.velocity_x != 0 and self.velocity_y != 0:
            # divide both x and y movement by root 2
            self.velocity_x /= sqrt(2)
            self.velocity_y /= sqrt(2)

    def move(self):
        self.pos += pygame.math.Vector2(self.velocity_x, self.velocity_y)

    def update(self):
        self.user_input()
        self.move()

    def draw(self, screen):
        screen.blit(self.image, self.pos)

player = Player((200,200)) # creates a player at (200,200)

while True:
    event_list = pygame.event.get()
    # fetches the system events from a queue of events that have occurred since the last fetch
    # system events can be things such as pressing the mouse down or quitting
    for event in event_list:
        if event.type == pygame.QUIT: # allows the exit button to work
            pygame.quit()
            exit()

    player.update()

    screen.fill((255,255,255)) # white screen
    player.draw(screen)

    pygame.display.update()
    clock.tick(FPS) # makes the game run at a maximum of FPS constant (60)
```

Doing this prototyping taught me how to create a moving character in Pygame and how to resolve some key issues that will come up when it comes to implementing the game, such as FPS capping and diagonal movement.

1.9.2 Shooting

I then tried to add a basic shooting mechanic to this basic character as this is another core mechanic of the game.

The idea was that when the arrow keys are pressed, bullets are regularly shot from the character in the direction determined by the pressed arrow keys. To do this I created a simple bullet class and amended the player class to include shooting with the arrow keys. When holding down an arrow key, the player looks in that direction and a bullet object is created every BULLET_RATE (the number of milliseconds between every shot). This bullet object is added to a simple list of all the bullets that is looped through to update and draw each of them individually.

To make the player look in the direction they are shooting and have priority over the direction of movement, I simply use sequencing and run the shooting code after the movement code.

A problem I had to overcome was that when I initially designed the shooting, the bullets would just spawn from the centre of the player and not appear to be coming out of the gun. This is because I was just spawning the bullets from the position of the player, which is the player's centre. To fix this, I had to tailor a slight offset for the position that the bullets spawned in for each of the four directions. I did this in terms of pixels so that it would scale if I scale the screen size and so that it is easier to read and understand.

I also realised that the bullets I was creating never stopped, they just got further and further away every frame and more and more bullets got created. If you kept shooting for a while, the game would start to run very slowly as lots of unnecessary objects would be being updated and drawn in every frame. To fix this, I needed to create a BULLET_LIFETIME constant which defines the time that a bullet lasts for and then in the bullet object, store the time at which the bullet was created and check every frame that the length the bullet has been alive is not longer than the defined lifetime. If it has been alive for too long, the bullet is removed from the list of bullets and will no longer be updated or drawn.

On top of this, the extra diagonal speed found in the movement was also found in the shooting. To fix this, just like with the movement, I first check whether the bullet is going to be shot diagonally and if so, divide both the horizontal and vertical components of its direction by $\sqrt{2}$.

Highlighted lines are the new or amended lines from the previous code

```
BULLET_SIZE = BULLET_HEIGHT, BULLET_WIDTH = 2 * PIXEL_RATIO, 2 * PIXEL_RATIO
BULLET_LIFETIME = 2000
BULLET_SPEED = 0.8 * PIXEL_RATIO * 60 / FPS # bullet speed is also linked to FPS and pixel ratio
BULLET_RATE = 500

bullet_image = pygame.transform.scale(pygame.image.load("images/bullet
image.png").convert_alpha(), BULLET_SIZE)

class Player():
    def __init__(self, start_pos):
        self.image = character_front
        self.pos = pygame.math.Vector2(start_pos)
        self.speed = PLAYER_SPEED
        self.rect = self.image.get_rect(center = self.pos)
        self.last_shot = 0
```

```
    def user_input(self):
        self.velocity_x = 0
        self.velocity_y = 0

        # returns a large list of boolean values of whether each key is pressed
        keys = pygame.key.get_pressed()

        if keys[pygame.K_w]: # pygame constant of the index of the W key in the keys list
           # y is the pixels from the top of the screen, so -speed to move upwards
            self.velocity_y -= self.speed
            self.image = character_back
           # player's image changes depending on the direction they are moving
        if keys[pygame.K_s]:
            self.velocity_y += self.speed
            self.image = character_front
        if keys[pygame.K_a]:
            self.velocity_x -= self.speed
            self.image = character_left
        if keys[pygame.K_d]:
            self.velocity_x += self.speed
            self.image = character_right

       # if player is moving in both x and y (diagonal),
        if self.velocity_x != 0 and self.velocity_y != 0:
            # divide both x and y movement by root 2
            self.velocity_x /= sqrt(2)
            self.velocity_y /= sqrt(2)

        bullet_x = 0
        bullet_y = 0
        x_offset = 0
        y_offset = 0

        if keys[pygame.K_UP]:
            bullet_y -= 1
            x_offset = PIXEL_RATIO * 2.5    # spawn bullet 2.5 pixels to the right
            y_offset = PIXEL_RATIO * 0      # and 0 pixels down
            self.image = character_back
        if keys[pygame.K_DOWN]:
            bullet_y += 1
            x_offset = PIXEL_RATIO * -2.5
            y_offset = PIXEL_RATIO * 4
            self.image = character_front
        if keys[pygame.K_LEFT]:
            bullet_x -= 1
            x_offset = PIXEL_RATIO * -4
            y_offset = PIXEL_RATIO * 1.5
            self.image = character_left
        if keys[pygame.K_RIGHT]:
            bullet_x += 1
```

```
            x_offset = PIXEL_RATIO * 3
            y_offset = PIXEL_RATIO * 1.5
            self.image = character_right

        if bullet_x != 0 and bullet_y != 0:  # diagonal
            bullet_x /= sqrt(2)
            bullet_y /= sqrt(2)

        if bullet_x != 0 or bullet_y != 0:   # if shooting
            time = pygame.time.get_ticks()
            if time - self.last_shot > BULLET_RATE:
                self.last_shot = time
                offset = (x_offset, y_offset)
                dir = pygame.math.Vector2(bullet_x,bullet_y)
                bullet = Bullet(self.pos + offset, dir)
                bullets.append(bullet)

    def move(self):
        self.pos += pygame.math.Vector2(self.velocity_x, self.velocity_y)
        self.rect.center = self.pos

    def update(self):
        self.user_input()
        self.move()

    def draw(self, screen):
        screen.blit(self.image, self.rect)

class Bullet():
    def __init__(self, pos, dir):
        self.image = bullet_image
        self.dir = dir
        self.pos = pygame.math.Vector2(pos)
        self.rect = self.image.get_rect(center = self.pos)
        self.lifetime = BULLET_LIFETIME
        self.speed = BULLET_SPEED
        self.spawn_time = pygame.time.get_ticks()

    def update(self):
        self.pos += self.dir * self.speed
        self.rect.center = self.pos
        if pygame.time.get_ticks() - self.spawn_time > self.lifetime:
            bullets.remove(self)

    def draw(self, screen):
        screen.blit(self.image, self.rect)

bullets = []

player = Player((200,200)) # creates a player at (200,200)
```

```
while True:
    event_list = pygame.event.get()
    # fetches the system events from a queue of events that have occurred since the last fetch
    # system events can be things such as pressing the mouse down or quitting
    for event in event_list:
        if event.type == pygame.QUIT: # allows the exit button to work
            pygame.quit()
            exit()

    player.update()

    screen.fill((255,255,255)) # white screen
    for bullet in bullets:
        bullet.update()
        bullet.draw(screen)
    player.draw(screen)

    pygame.display.update()
    clock.tick(FPS) # makes the game run at a maximum of FPS constant (60)
```

Now the player can shoot bullets from his previously unavailing gun, even when he is moving:

Doing this prototyping taught me how to let my character shoot bullets by dynamically creating bullet objects based on user input, which is a big part of the game.

1.9.3 Saving

I also wanted to test how save files would be implemented as this plays a big part in the game too.
To do this I made a very basic clicker game, with a red, green, and blue button. Each time you click a button, a counter for that button increments by 1 and displays to the screen in text. This wouldn't be too dissimilar to the save system in the actual game, where every time you do a specific action, such as killing an enemy or shooting a bullet, a counter is incremented by 1. I stored the counters in a dictionary so that each counter can be accessed by their unique key, which is likely similar to what I will end up creating for the final game too, having a dictionary of all the data I need to save and store about the player. Creating this program also allowed me to test using text and fonts in Pygame and explore how they work.
To add saving to this clicker game, you need to do some file handling. Saving in a game of this scale is just creating and overwriting a file stored on the device that is read and the relevant data loaded when the game opens. I used JSON files for this because JSON files (originally made for Java Script) can store objects, and usefully dictionaries, and then be read as this object rather than a string. You first have to import the json library, but then you can write to and read from a JSON file in a very similar way to how you would write to and read from a txt file.

This is the code for loading and reading from the save file:

```
try:
    with open("save.json", "r") as file:  # open the save file in read mode
        counters = json.load(file)      # set the counters to the data in the save file
except FileNotFoundError:  # no such file exists
    counters = {'button1' : 0, # create the dictionary for the counters, initially 0
                'button2' : 0,
                'button3' : 0}
```

You open the JSON file with “r” (read) mode to read from the file, and then use the json library function json.load(file) to load the specified file in its object format so you can save it to a variable, the counters dictionary in this example. The try except statement is used because if the game is being opened for the first time, there is no file created yet, trying to open the file produces a FileNotFoundError. This can be caught by an except statement that is only run the first time the game opens, therefore it is used to create the counters dictionary with the values initially at 0.

We then want to write to / overwrite this file every time the game is quit so that it saves the final state to be opened next time.

```
with open("save.json", "w") as file:
    json.dump(counters, file) # puts the counters dictionary into the file
```

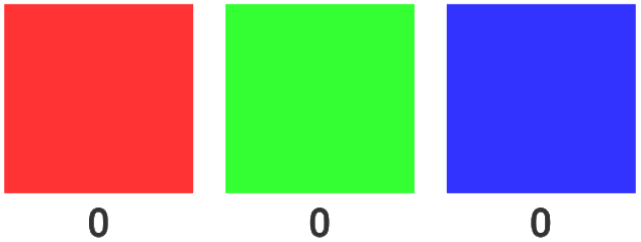
You open the json file with “w” (write) mode to write to the file, and then use the json library function json.dump(data, file) to put the specified data into the specified file in its object format so that it can be read in this format.

The code for the basic clicker game, the lines relevant to saving are highlighted:

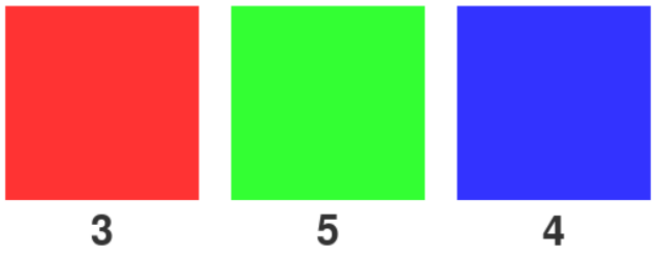
```
import pygame
import json
from sys import exit

pygame.init()

screen = pygame.display.set_mode((700, 700))
font = pygame.font.Font(pygame.font.get_default_font(), 32) # create a font object

button1 = pygame.Rect(100, 150, 150, 150)  # x, y, width, height
button2 = pygame.Rect(275, 150, 150, 150)
button3 = pygame.Rect(450, 150, 150, 150)

try:
    with open("save.json", "r") as file:  # open the save file in read mode
        counters = json.load(file)      # set the counters to the data in the save file
except:  # no such file exists
    counters = {'button1' : 0, # create the dictionary for the counters, initially 0
                'button2' : 0,
                'button3' : 0}

# create text for each of the counters
button1_text = font.render(str(counters['button1']), True, (0,0,0))
button2_text = font.render(str(counters['button2']), True, (0,0,0))
button3_text = font.render(str(counters['button3']), True, (0,0,0))
```

```
while True:
    screen.fill((255,255,255))

    click = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            with open("save.json", "w") as file:
                json.dump(counters, file) # puts the counters dictionary into the file
            pygame.quit()
            exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            click = True

    pygame.draw.rect(screen, (255,0,0), button1) # draw the buttons to the screen
    pygame.draw.rect(screen, (0,255,0), button2)
    pygame.draw.rect(screen, (0,0,255), button3)

    mx, my = pygame.mouse.get_pos()

    if click:
        if button1.collidepoint(mx, my): # if clicking and the mouse is in the rect
            # increment the counter by 1 and update the text
            counters["button1"] += 1
            button1_text = font.render(str(counters['button1']), True, (0,0,0))
        if button2.collidepoint(mx, my):
            counters["button2"] += 1
            button2_text = font.render(str(counters['button2']), True, (0,0,0))
        if button3.collidepoint(mx, my):
            counters["button3"] += 1
            button3_text = font.render(str(counters['button3']), True, (0,0,0))

    screen.blit(button1_text, button1_text.get_rect(center = (175, 325))) # display text
    screen.blit(button2_text, button2_text.get_rect(center = (350, 325)))
    screen.blit(button3_text, button3_text.get_rect(center = (525, 325)))

    pygame.display.update()
```

When the game is opened for the first time, the counters and buttons display as such, with each counter at 0:



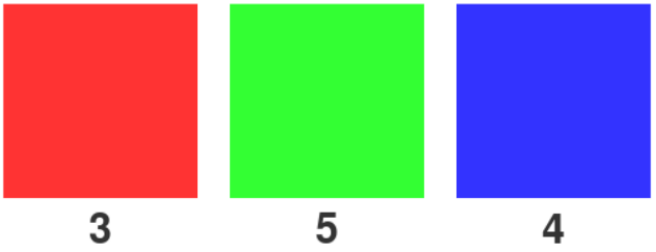
If you click on each a few times, the counters increase:



If you close the game after this, a json save file is created with the counters dictionary inside:

```
{} save.json > ...
  1   {"button1": 3, "button2": 5, "button3": 4}
```

Opening the game the next time opens with the saved counters.

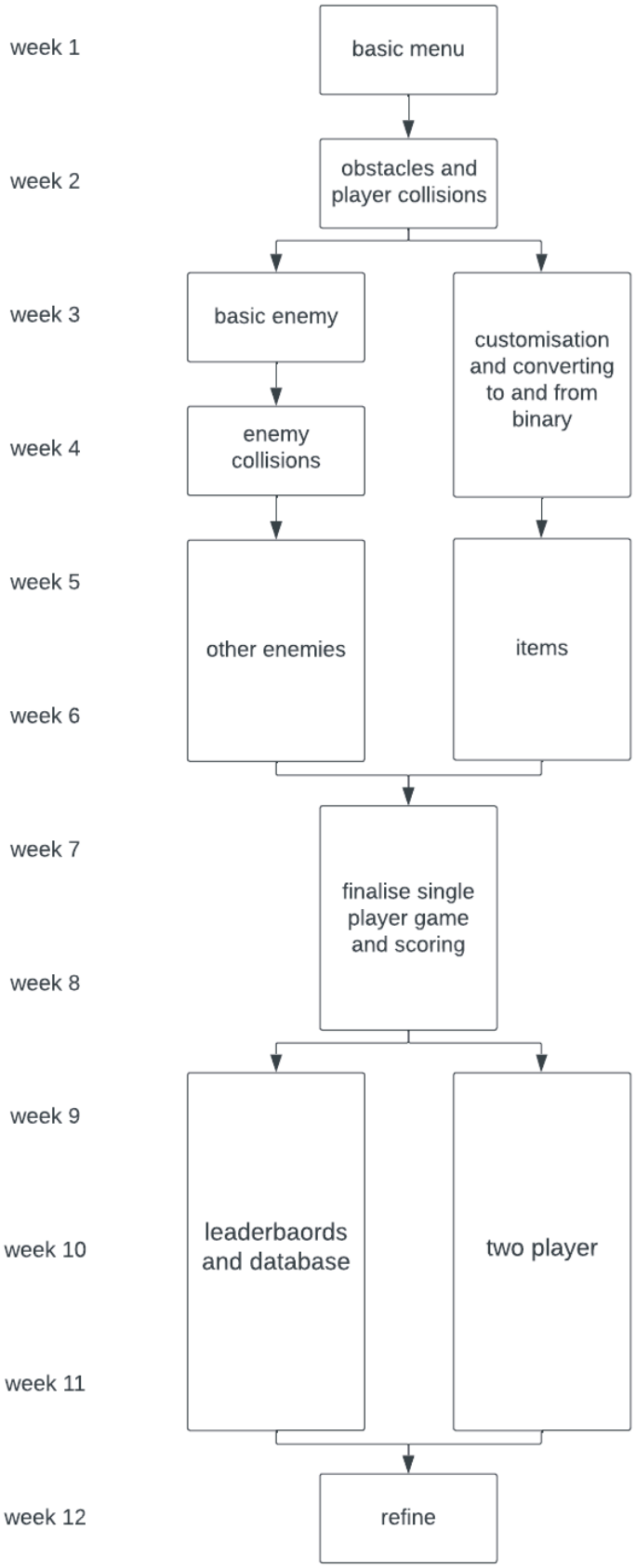


Doing this prototyping allowed me to see how I would save files in my game and also see how to use the Pygame font object. I will need to create my own font for my game as it will need to match the pixel style of the rest of the game but it is still useful to find out how the text is displayed to the screen.

During the design stage I realised that I will not need to use any save files to store the player data as it will all be stored on the local database. It however was still useful prototyping to do as I will be using this system to store the volume setting. I thought it would be better to store the sound volume in a file rather than storing an individual volume setting for each user as that could get irritating when playing with multiple people. The prototyping also just gave me some more pygame practice which is always useful.

2. Design

2.1 Critical Path Diagram



This is a provisional critical path diagram that suggests it will take 12 weeks to complete my project. Some tasks will require more or less time than I have predicted and will result in slightly different paths and timings but it is still very useful to have this diagram to refer back to and check on my progress. Tasks are often simultaneous in my diagram instead of all being separate because if I lose motivation when doing a difficult task I can progress on another one and come back to it.

2.2 System Design

Before creating the game, I need to plan how the game and the menus function and interact with each other based upon my objectives. This will make it much easier when it comes to the implementation stage as I will be able to focus on just the programming, not needing to think much about the design.

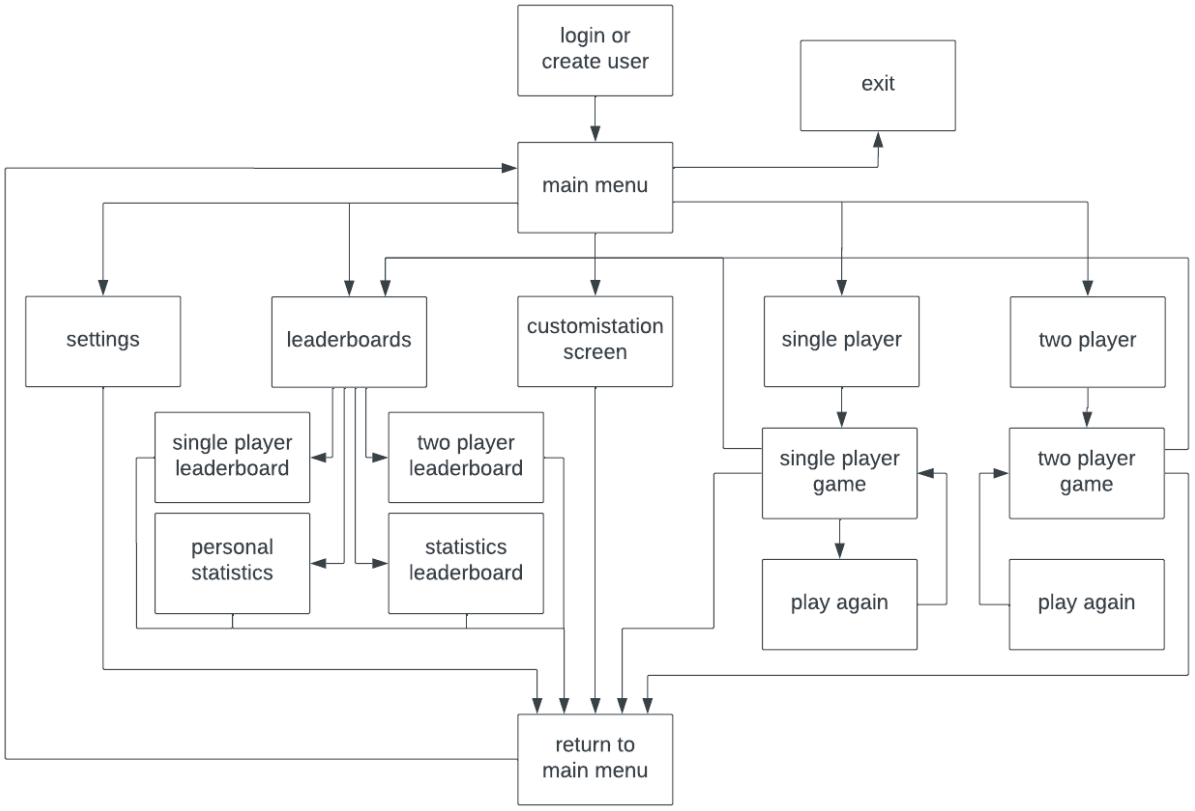
When you open the game, you will find a screen with a button to login, create an account, or quit. If you press the login button, you will be taken to a screen with text prompts for the username and pin. If you type an unrecognised name or an incorrect pin you will be informed of this. For the usernames and pins there will be a 4 character limit with the username only allowing capital letters and the pin only allowing digits. I think that limiting the username length to only 4 characters will make it more fun to make usernames as limitation sparks creativity and you can't always just type in your full name. It also brings about the retro style of arcade games with limited character names and makes it easier for me to make and format leaderboards. Besides, there is no need for more than 4 characters per name, as 4 characters can already make 26^4 or 456 976 possible usernames when only using the upper-case alphabet, which is about 456 970 more than the scope of the game needs. The user database that stores all the usernames and pins is stored locally on each device that plays the game. If you press the create account button, you will be taken to a screen with text prompts for the username and pin. If you type a name already used by another player you will be informed of this.

Once an account is created or logged in to, you are taken to the main menu that has a large title of the game and buttons for the single player game, two player game, the settings, the leaderboard, and the customisation studio. The menu will also feature a picture of your custom character and a small section of the single player leaderboard. To make the menus simple and accessible, as per my objectives, it is good to try and limit the number of button presses the user needs to make to access anything, so including the leaderboard and character display on the main menu means they aren't hidden behind any buttons and you can quickly check them and start playing. The main menu will also have buttons to quit and to log out, which will close the game and return you to the login screen respectively.

If the single player game button on the main menu is pressed, you are taken to the single player game. In the single player game, you can move a character on the screen using WASD and shoot from this character using the arrow keys. Enemies will spawn in waves from the sides of the screen and walk towards your character. You have to shoot them before they reach you otherwise you will lose one of your initial lives. After every few waves of enemies, an obstacle will spawn at a random place in the map that you, your bullets, and enemies cannot move through, a feature that me and my client thought would make the game more interesting. As you progress and kill more enemies, the enemies become of harder types and spawn more regularly. There will be a range of enemies that can appear with different movement patterns and health and some that can fly over other enemies and any obstacles. Occasionally after killing an enemy, an item will appear where they died. This item could boost your abilities or generally help you, such as a speed boost that makes you move faster or a bomb that kills all enemies on the screen. One item can be stored by the player to be used at any point in the current game. For each game, you have a score that starts at 0. This score is incremented slightly for every second or so that you are alive and incremented by a larger amount for each enemy that you kill. Different enemy types will increase your score by different amounts when you kill them. Around the sides of the game, this score will be displayed as well as your personal highest score, your lives counter, and a space for your stored item. The game ends when you run out of lives.

When you do run out of lives, a screen will appear showing you statistics about the game you just played with your time survived, number of enemies killed, bullets shot, items used, and your score. The database will be updated to include this game and to update your statistics. An updated leaderboard is also displayed so you don't have to go through lots of menus to check how things have changed. There will be buttons to play again, check leaderboards, or return to the main menu.
If the two player game button on the main menu is pressed, you are first taken to a login screen for the second player. I thought it would be fun and engaging if the second player wasn't just a guest account with a default character, but instead another user from the database with their character that you can play with. Once a valid account has been logged into, you can begin the two player game. The game is very similar to the single player game with the change that there are two players. Both players can move and shoot using currently undecided controls. Enemy spawning will be slightly more regular as there are two players and two guns, but how much more regular will be decided through testing as there may be unforeseen complexities in the two player mode. The score screen will appear in the same way, however, two player scores are recorded separate to the single player scores and are displayed on different leaderboards.
If the leaderboards button on the main menu is pressed, you are taken to the leaderboards screen. This screen features 4 tabs you can switch between, each with different data displayed. The first tab is the single player leaderboard, which displays the top 10 or so scores for the single player game. Each score is accompanied by the username of the player who achieved it and a little picture of their custom character. The second tab is the two player leaderboard, which displays the top 10 or so scores for the two player game. Each score is accompanied by the usernames of the players who achieved it. The third tab is the statistics leaderboard, which features podiums for a few different statistics categories, e.g. number of enemies killed, number of games played, number of items used, number of bullets shot. Each category consists of a first player and their total, whose username and custom character are displayed in big, as well as the second and third player and their totals, whose usernames and custom characters are displayed but smaller. The final tab is the personal statistics tab, which features a list of statistics about the games you've played.
If the customisation button on the main menu is pressed, you are taken to the customisation screen. This studio allows you to change the look of the character you play as. After talking more with my client, we decided that a good amount of freedom for the player to have is to let them draw their character a hat. The body of the character can be customised by colour selection grids that let you change the colours of each part. This ensures that there are no issues with formatting the player and making the player shoot out of their gun because the position of the gun will be fixed on the body of the character and it also means that players will always look like players and there will be some consistent design. So, there will be a grid you can draw a custom hat in that sits on the head of your character. There is a colour selection grid to select the colour you are drawing with, as well as undo and redo buttons beside the grid that let you undo your recent changes and undo your recent undos. For the body of the character, there will be individual colour grids for each part of the player design that allow you to customise these too. Some of the colours will be locked and are not usable in your custom character until you have attained certain achievements, such as shooting a certain amount of bullets or killing a certain amount of enemies. This is to entice players to play more and interact with the custom character designing because they will get to show off their achievements with their colours.
If the settings button on the main menu is pressed, you are taken to the settings menu. This is a simple menu consisting of a slider to change the sound effect volume.

2.3 System Flowchart



This flowchart represents the ways the user will be able to move between screens and get around the game menus and will help with designing the UI.

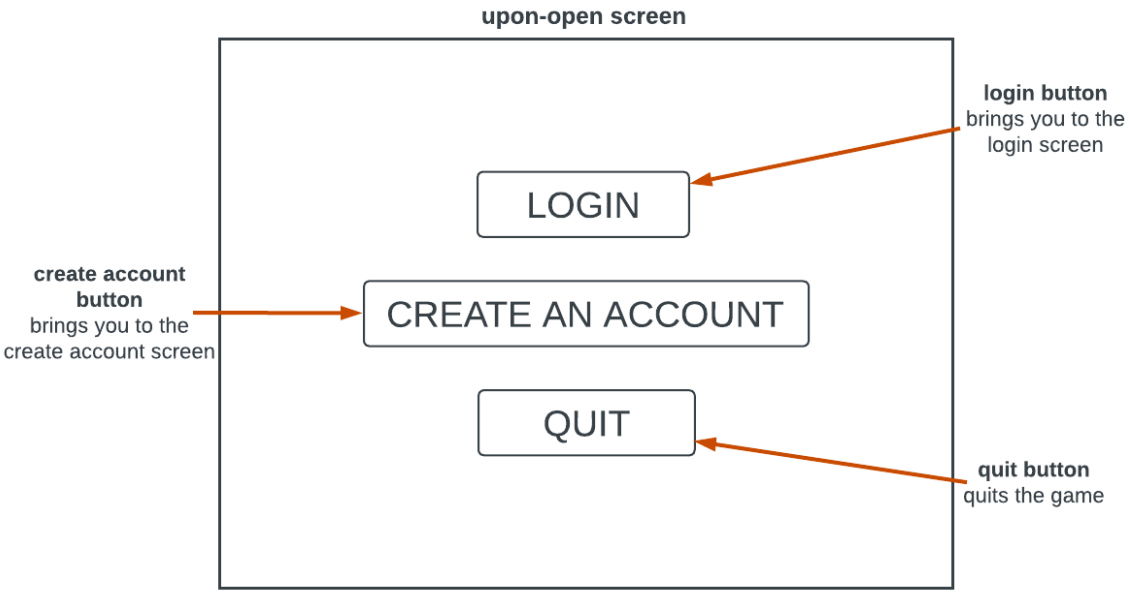
When creating the diagram, I wanted to have a button to go to the leaderboard screen once you complete a game, however, I decided to remove this functionality because I thought it was confusing, hard to fit onto the screen, and more exciting to have to leave the end-game menu to go and check the leaderboards. The menu does still feature a small version of the leaderboard though.

2.4 User Interface Design

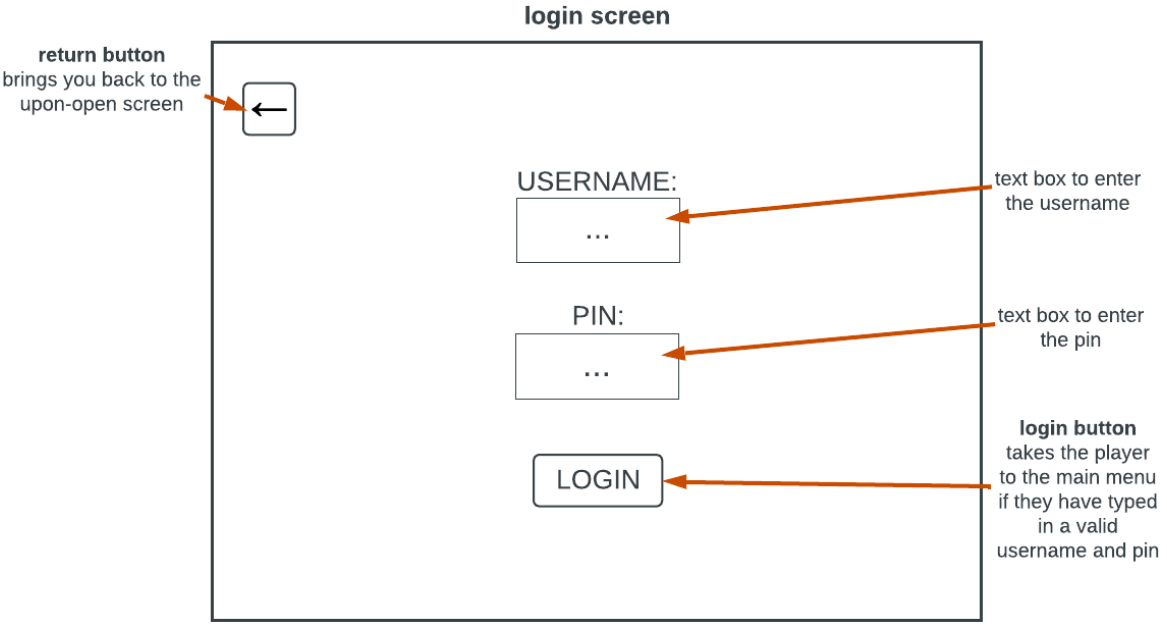
The user interface is a big aspect of the game and must be simple and understandable, as per my objectives, so I will need to test some designs and see what works best. Doing this now will also really help me to understand the way these menus will work and how the user will interact with them because I have to actually visualise and draw them.

To make a good menu, it should be simple, easy to understand, and not have more buttons than necessary. Perceived affordance is important, users should know how to interact with the menu from just looking at it and be able to presume what happens when they click any of the buttons. Buttons also should show when they are being hovered over, so the user knows what they are about to press. Moreover, menus should be efficient and quick to navigate, important features should be easy to get to and not require much clicking.

These screen designs proved very useful and almost all the interfaces in the final version were kept very similar if not the same as the designs I created here.

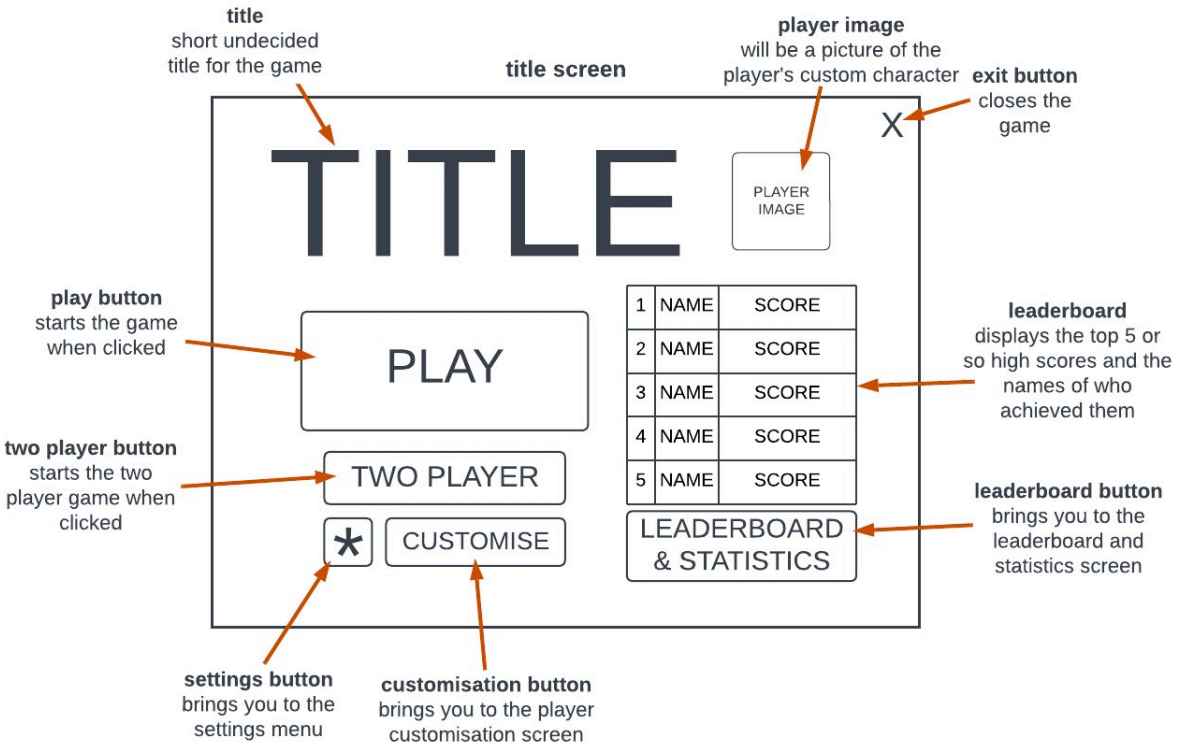


This is a design of the upon-open screen, created in Lucidchart, that is what opens every time you open the game. I made the aspect ratio of the screen and what will be the aspect ratio of the whole game 4:3. This is both because I like the arcade-ish style it gives the game and because I don't really want the game taking up the whole screen. I think the game doesn't need to be a very big deal and that it would work best as just a little game that doesn't overshadow your entire desktop, making it feel like you can open it and play whenever you have a couple minutes spare. This screen needs to be plain and simple so it consists of 3 buttons, one to go to the login screen, one to go to the create account screen, and one to quit the game.



This is a design of the login screen. This screen also needs to be plain and simple and so it consists of a textbox to enter a username, a textbox to enter a pin, and a button to log in once a valid username and pin has been entered. This screen also has a return button so that you can return to the upon-open screen.

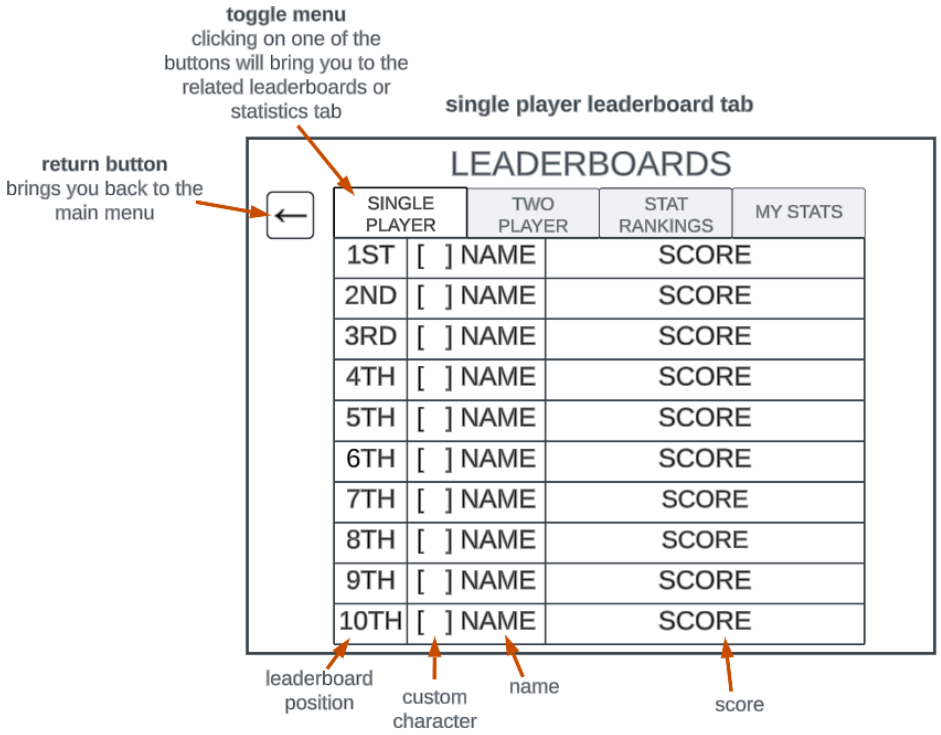
I have not made a design for the create account screen as it will look and function the same as the login screen, only with the login button being changed to a create account button.



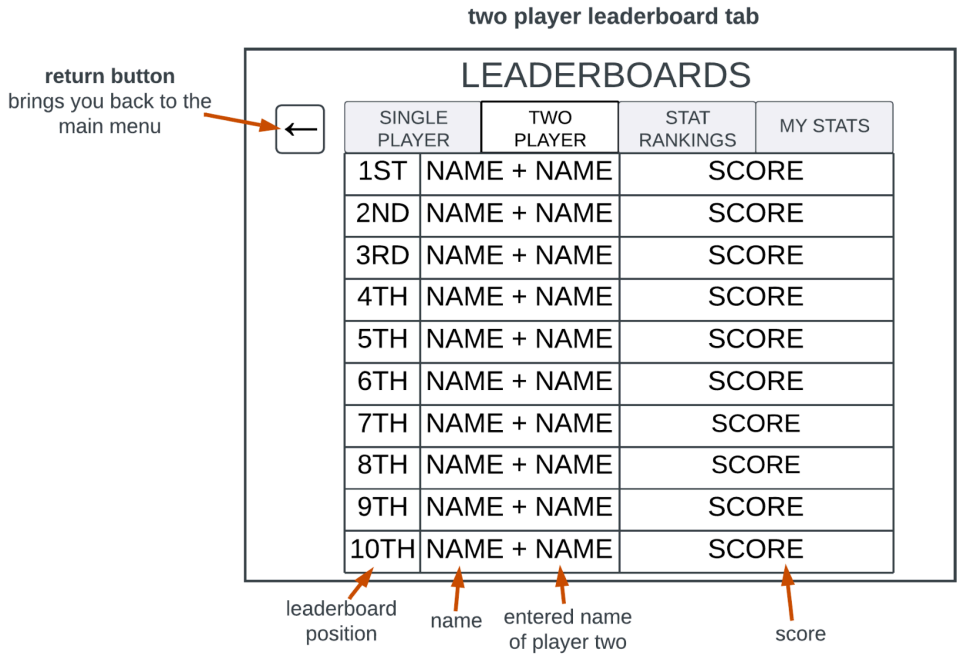
This is an initial design of my main menu. I made the play button the biggest as it is by far the most important button, with smaller buttons for the two player mode, the customisation menu, and the leaderboard and statistics screen. I also made even smaller buttons for the settings and exit. I wanted a leaderboard on the main menu so that you don't have to click to see what place you are in and who is ahead of you, and theoretically it will go into your mind just before you play even if you don't look directly at it. I added a big title that is currently undecided as well as a small picture of your custom character. This picture was ideally going to be next to the customise button but due to size restrictions this was the only place, I do think though that it will fit nicely with the title.

In my design, there is a lot going on for a menu that is meant to be simple and easy to understand so it may be the case that when it comes to implementing the main menu I will find that it is too chaotic and confusing and will alter the layout, most likely removing the leaderboard so that it is hidden behind a button despite it being an important feature. I may also find that I simply cannot write small enough for the buttons or this leaderboard as the main menu will also need to stick to the strict pixel constraint, so that would lead to me needing to alter the layout as well.

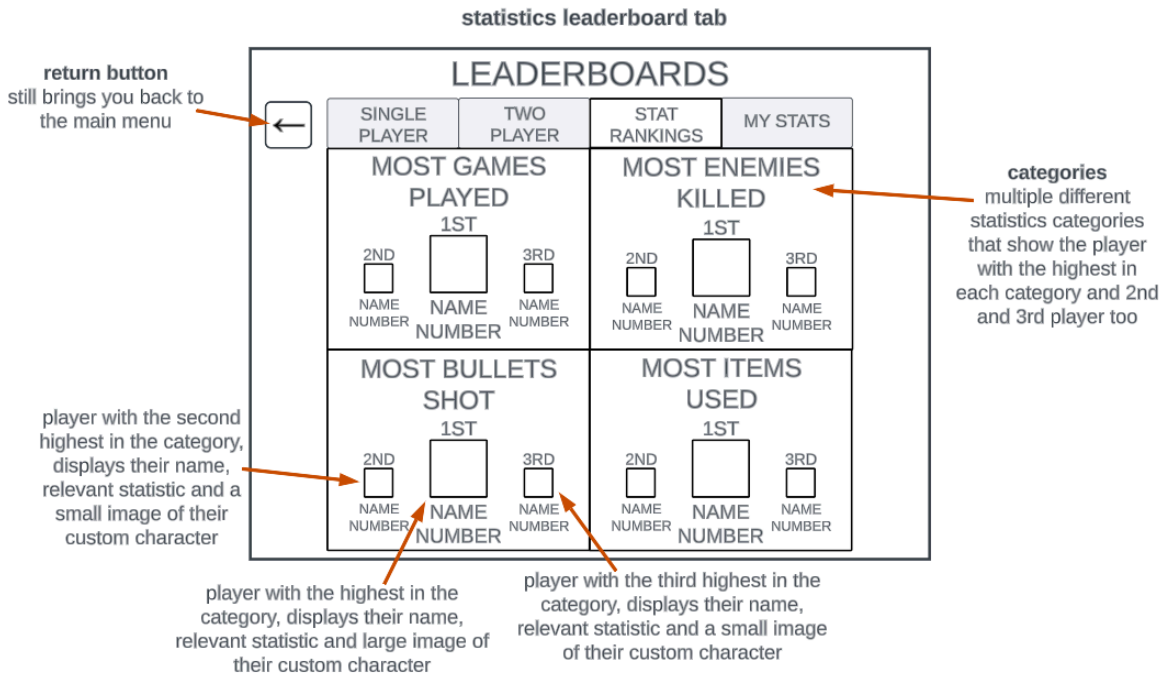
In my final design, after realising the need for a logout button, I changed the customise button into a button with a coat hanger icon next to the player image to grant me the required space. Other than this, the design of the main menu was kept quite similar.



This is a design of the leaderboard screen and specifically the single player leaderboard tab which is opened when you press the leaderboard button on the main menu. I thought the best method to show all the necessary leaderboards was to have them on different tabs on the same screen, so users can simply click on whichever tab they want to see when they reach the leaderboard screen and it should display. The screen also has a return button so that you can return to the main menu.
This single player tab shows the 10 highest scores achieved by players in the game. It shows their username and custom character so that other users can identify them.

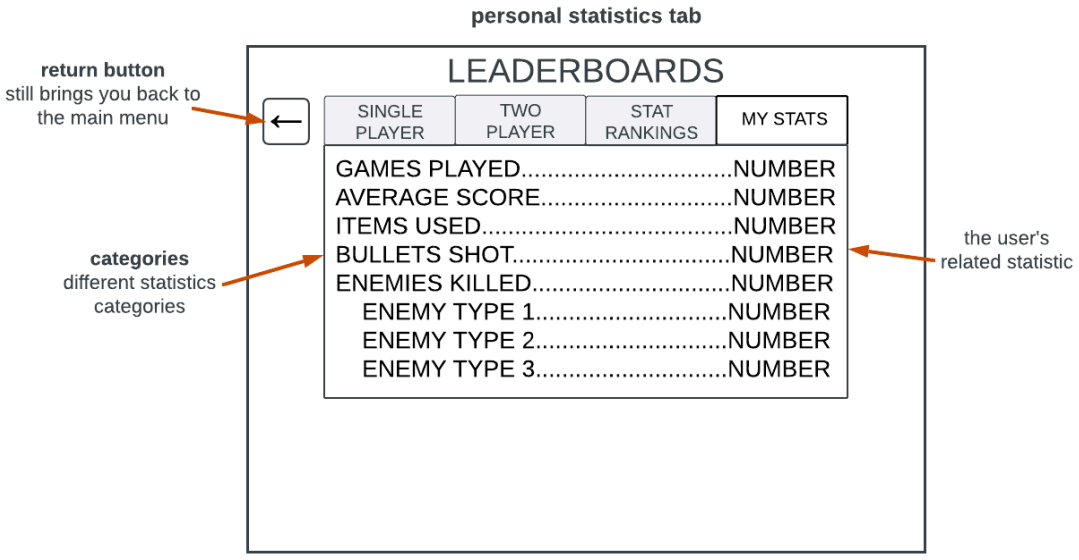


The two player tab shows the 10 highest scores achieved in the two player game mode. It shows the username of the first player and the entered name of the second player.

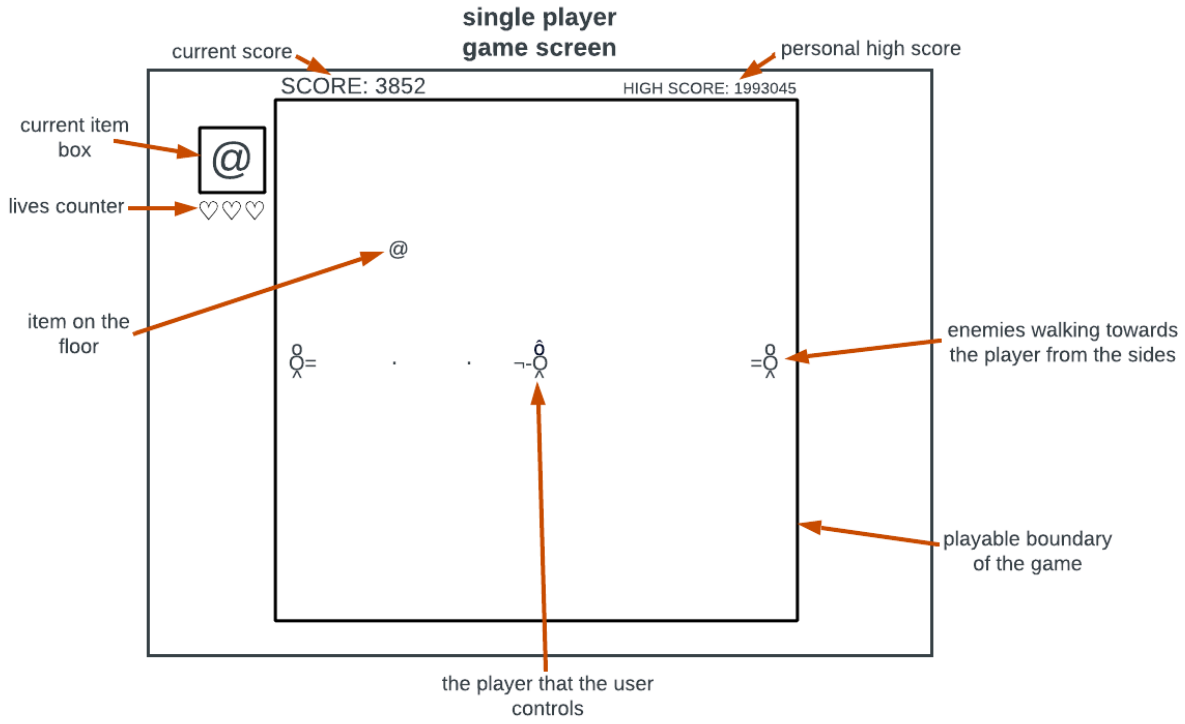


The stat rankings tab shows a few different categories of statistics and who has highest, second highest, and third highest in each. I thought it would be cool to display these in a podium format. It shows each player's username and their custom character. It may be that I have to reduce the number of statistics on the tab if I can't fit them in with the limited size or if I think it is too chaotic, furthermore, it may be somewhat pointless to have all the statistics as the person with the most games played will likely be the person with the most of the others too.

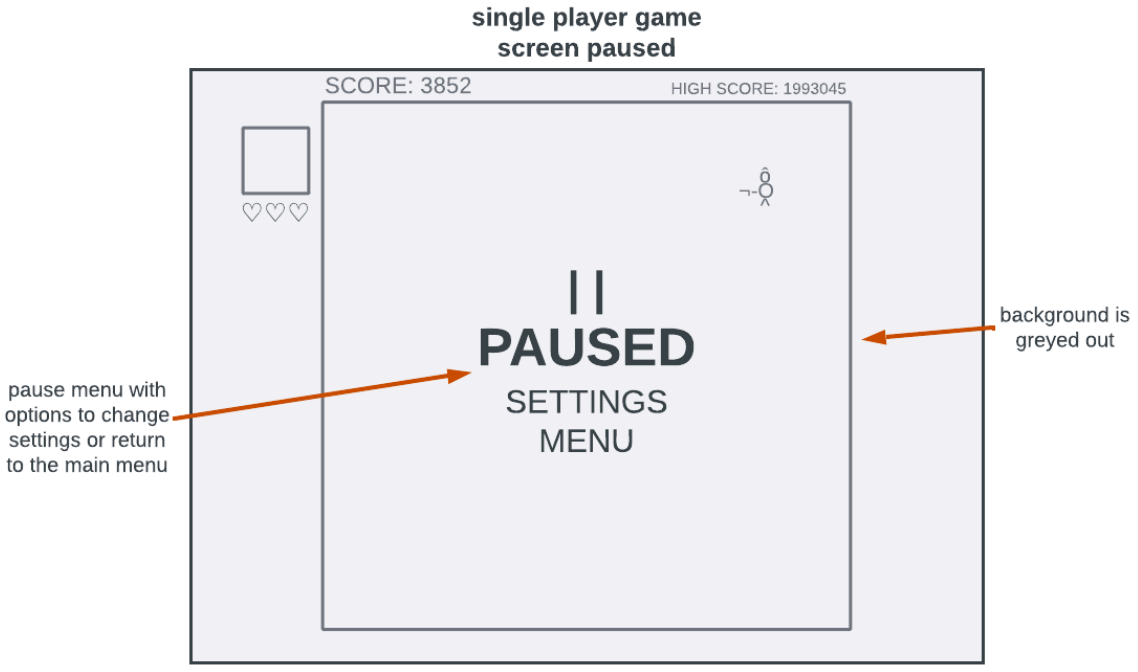
I was indeed not able to fit all 4 statistics onto the rankings page as there was just not enough space and it was getting too complex. Instead, in my implementation, only the most games played and most enemies killed statistics have podiums.



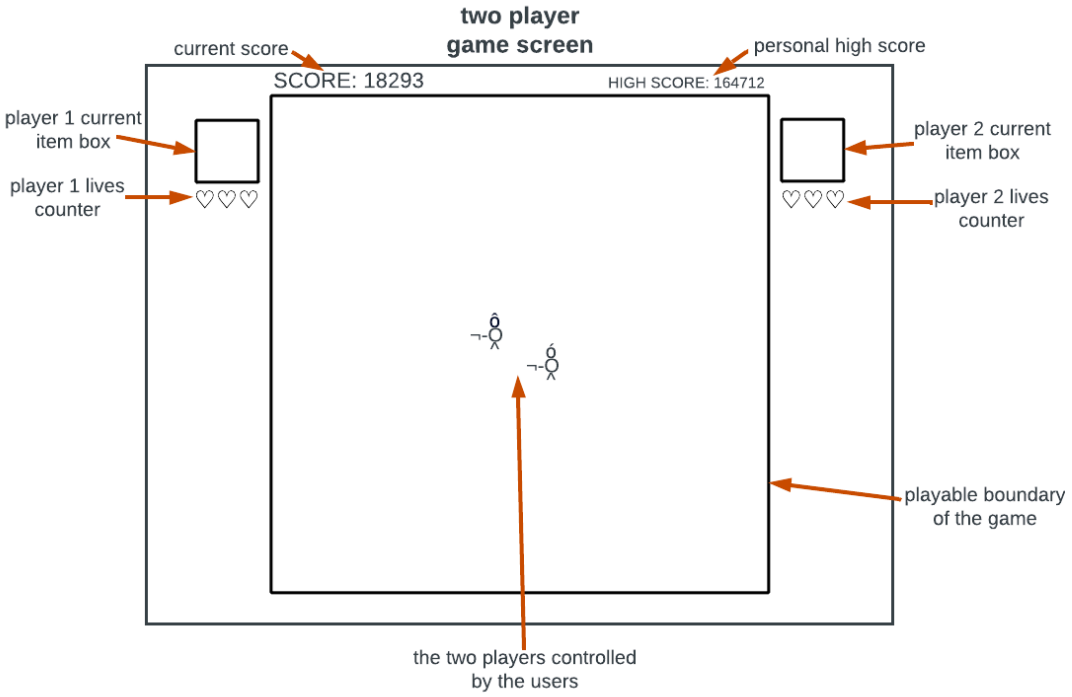
The my stats tab shows the user their personal statistics. It displays lots of different and specific statistics such as how many enemies the player has killed. This data will already be stored on the database so I thought it would just be nice for the user's curiosity.



This is a design of the single player game screen. When the play button on the main menu is pressed, you will be brought to this screen and the game will start. The screen focuses on a square area in the centre which is the playable boundary of the game, your character and bullets and enemies and interactions will all happen inside of it. Above the game screen is a display of your current score and your personal high score. To the left, the current item you have equipped and your lives.

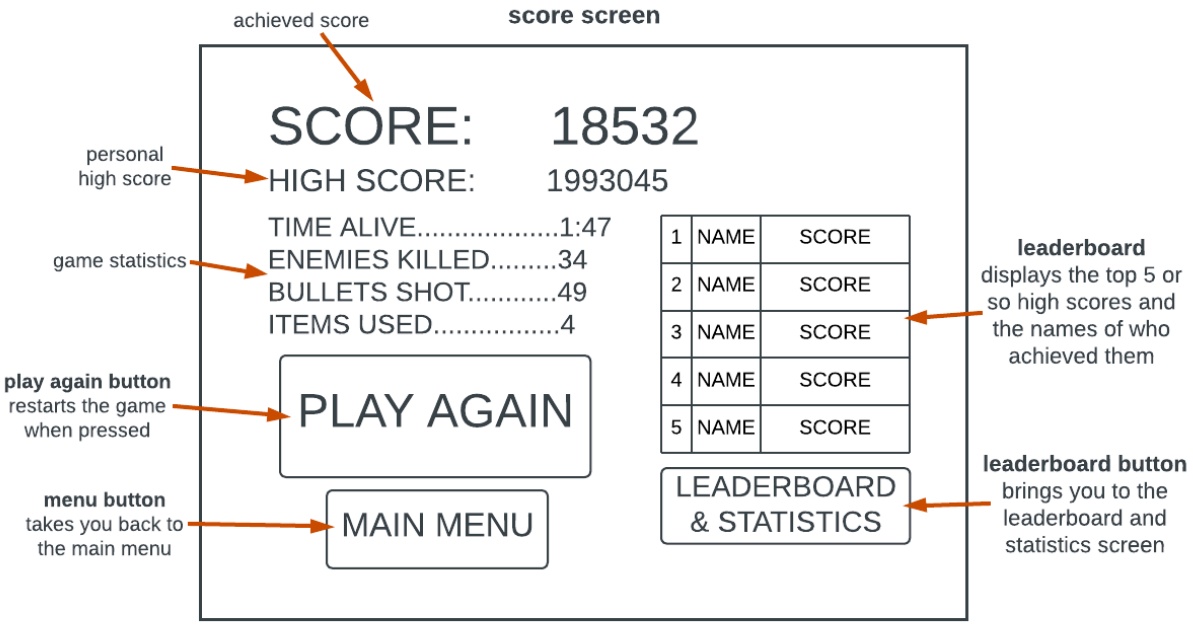


The single player game can be paused using the escape key. The game will be greyed out and paused and a menu will appear with options to change the settings or return to the menu.



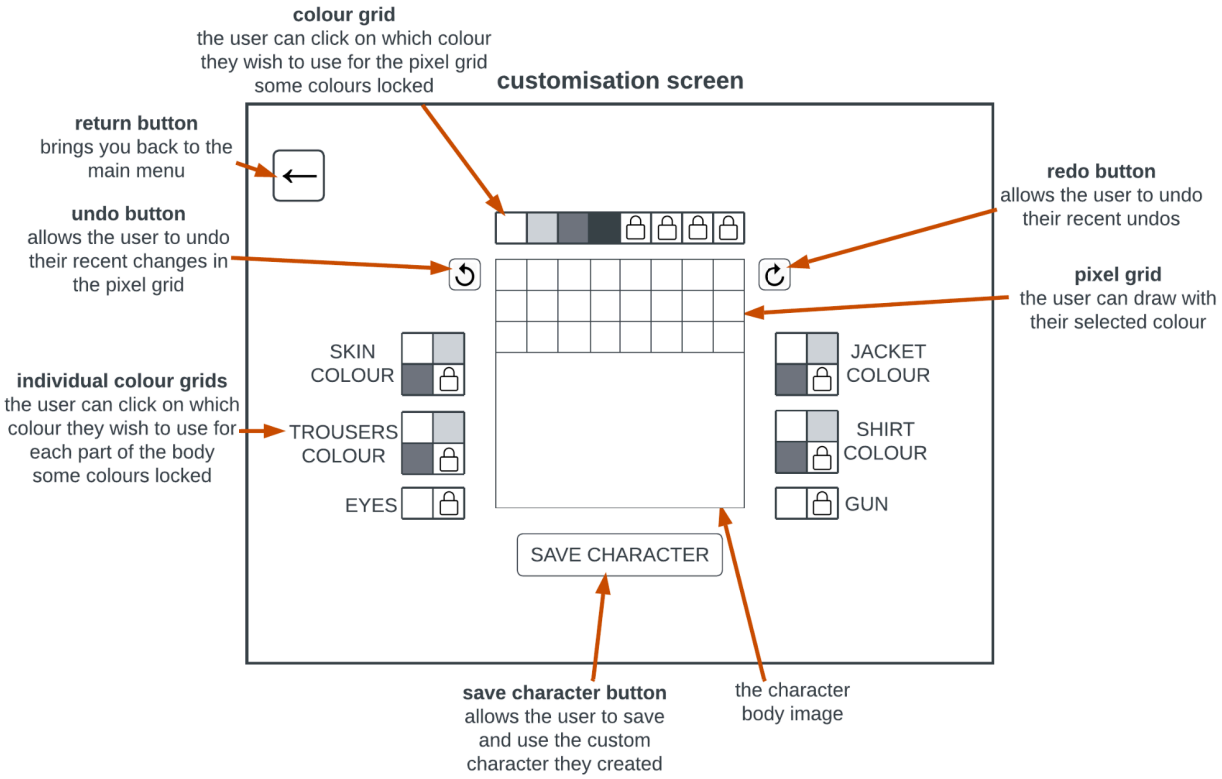
The two player game screen will be similar to the single player game screen, just with two players and a current item box and lives counter for the second player. The two player game will also be able to be paused in the same way with the same pause screen.

In the final design there are no item boxes because items are used as soon as they are collided with in two player in an attempt to reduce the number of keys needing to be pressed.



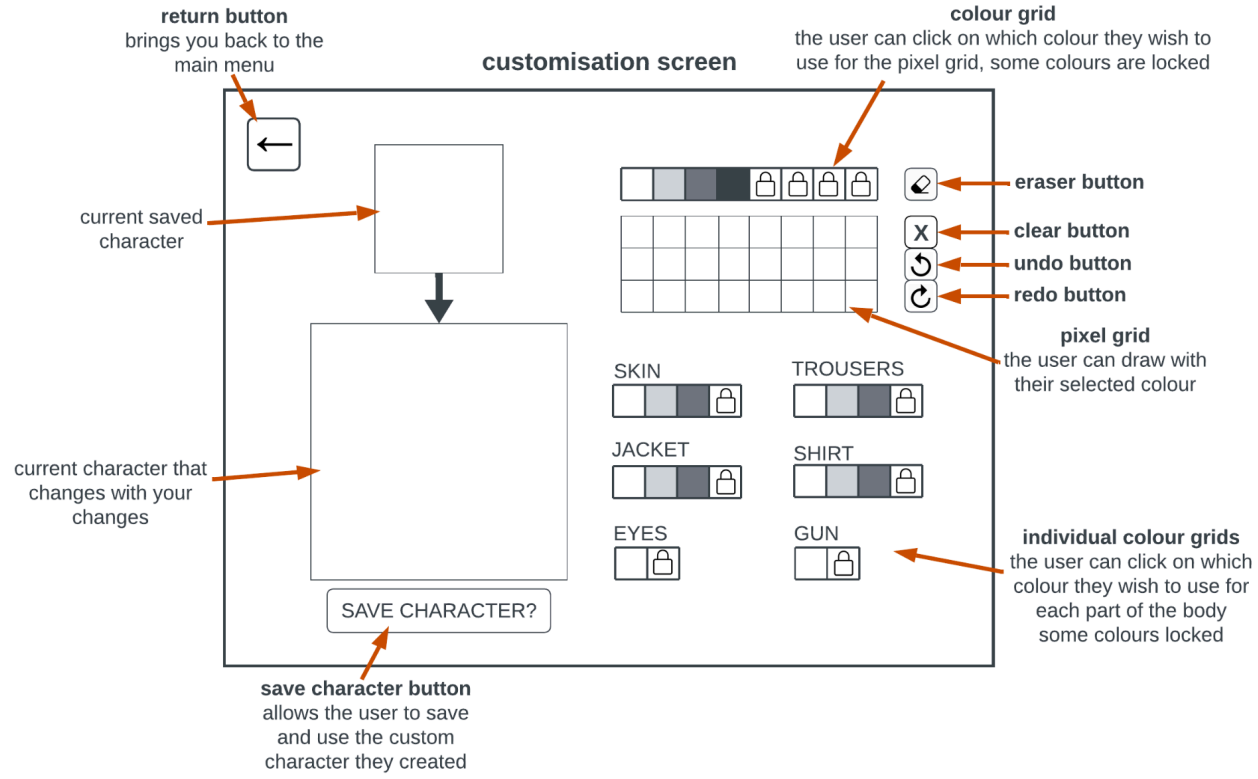
When the game ends and you have no lives left, the score screen appears. This shows you your achieved score, along with some statistics about your game. There is a small leaderboard that displays to show you the highest scores. There are buttons to play again, go straight to the leaderboard and statistics screen, or return to the main menu.

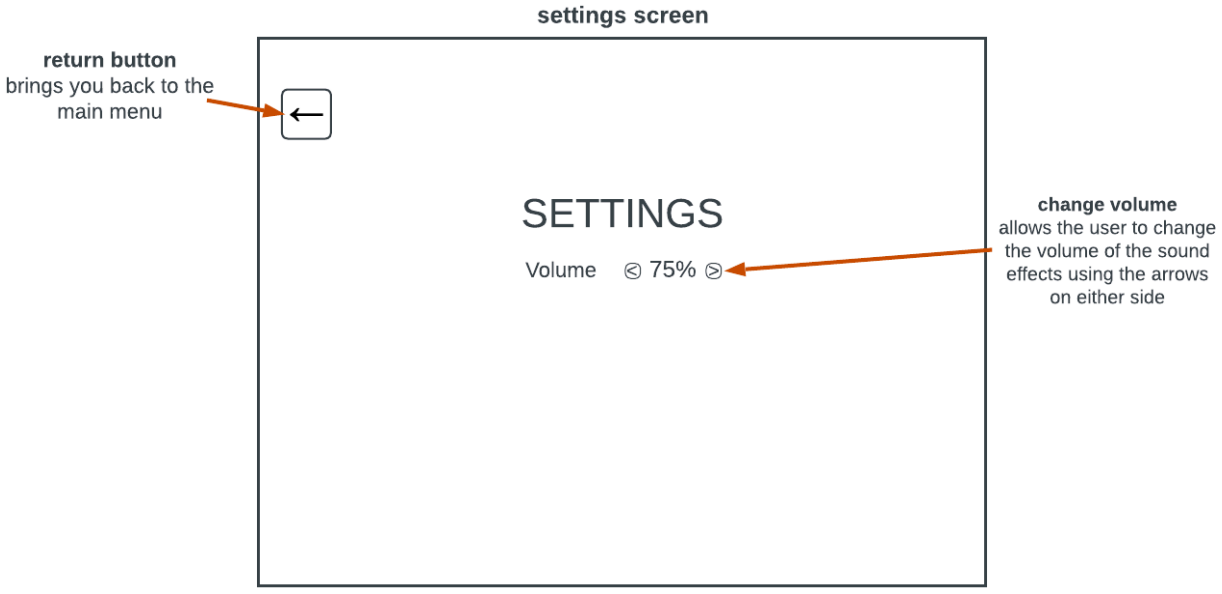
The leaderboard button was removed.



When you press the customise character button on the main menu, you are brought to the customisation screen. This screen allows you to change the appearance of your character. There is a return button to return to the menu, a colour palette, a character grid, and colour grids to pick the colours of individual parts. The colour palette lets you select which colour you are currently drawing the hat with. Some of these colours are locked and can only be unlocked by certain achievements. This is to entice players to play the game more as they'll be able to show off by using their unlocked colours in their custom character. With the selected colour, you can draw anything in the pixel grid to go on top of your character's head. You can also undo and redo using the buttons on either side. Below the pixel grid is a box that shows your character's body. You can change the colours of the individual parts through the grids on either side. Each grid has a locked colour that is also unlocked through certain achievements to let the players show off. I may find that this menu is too overwhelming or unclear for users and I may have to reduce the scope to only being able to change a few parts. Once you have created a character you can save it by pressing the save character button and it will then update on leaderboards and become the character you play as in the game.

I did indeed have to revise the design of the customisation screen when it came to implementing the customisation screen. I found that it really was not clear, especially with the hat drawing grid overlaying the character display, as well as the fact that you cannot see your previous character before you made any changes. This meant I needed to create a new design that makes it clear what you're changing and where and shows you your previous character, so I moved all the customisation options to the right side of the screen and put two character displays onto the left side of the grid, one large display that shows your current character with the changes you are making and a smaller display that shows your character as it was before you made any changes. I also realised when implementing that the drawing grid would need an eraser button that allows you to delete parts of your drawing without having to undo everything and a clear button that allows you to erase everything in the drawing at once. The new design can be seen on the next page.

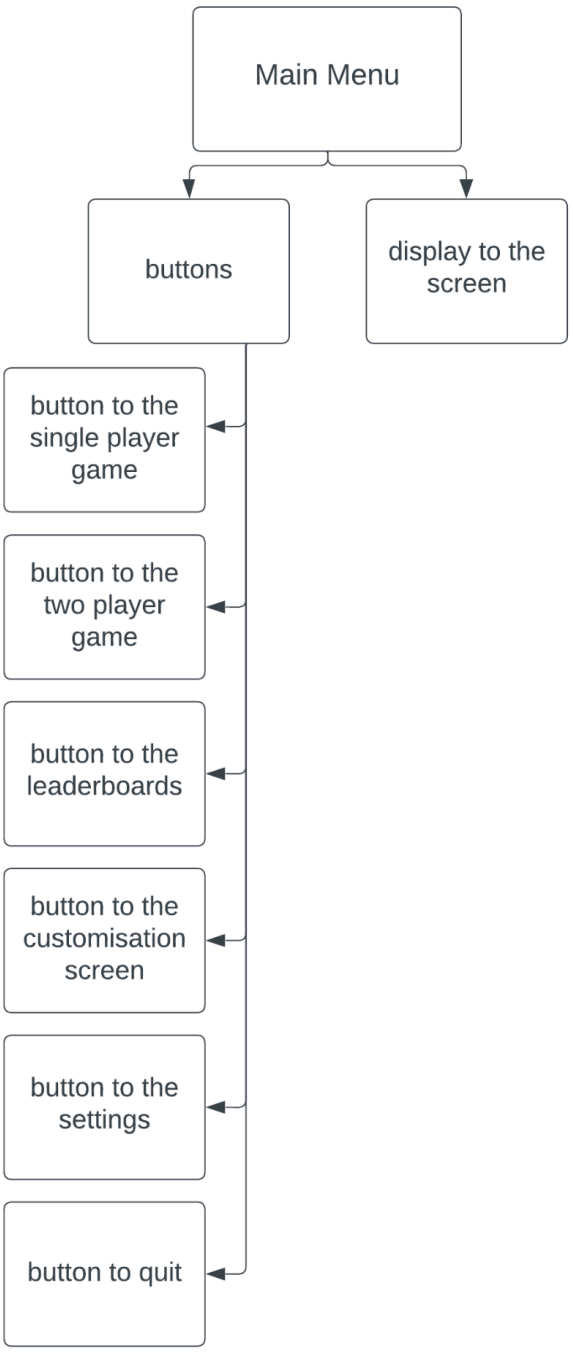


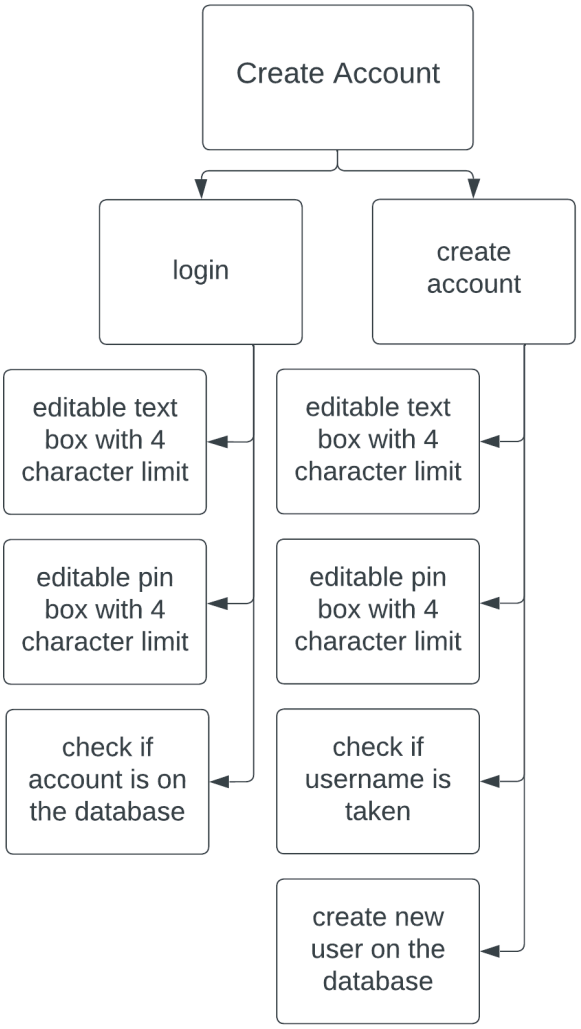


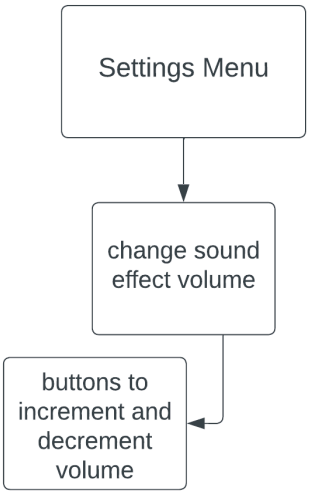
When you press the settings button on the main menu, the settings screen appears. It has options to return to the main menu or change the volume using arrow buttons on either side to go up or down in increments of 5 or so. It may seem a little pointless to have a whole settings screen for one setting but it allows for expansion later on in implementation if I find I want more settings and I think keeping it separate makes the menus easier to understand.

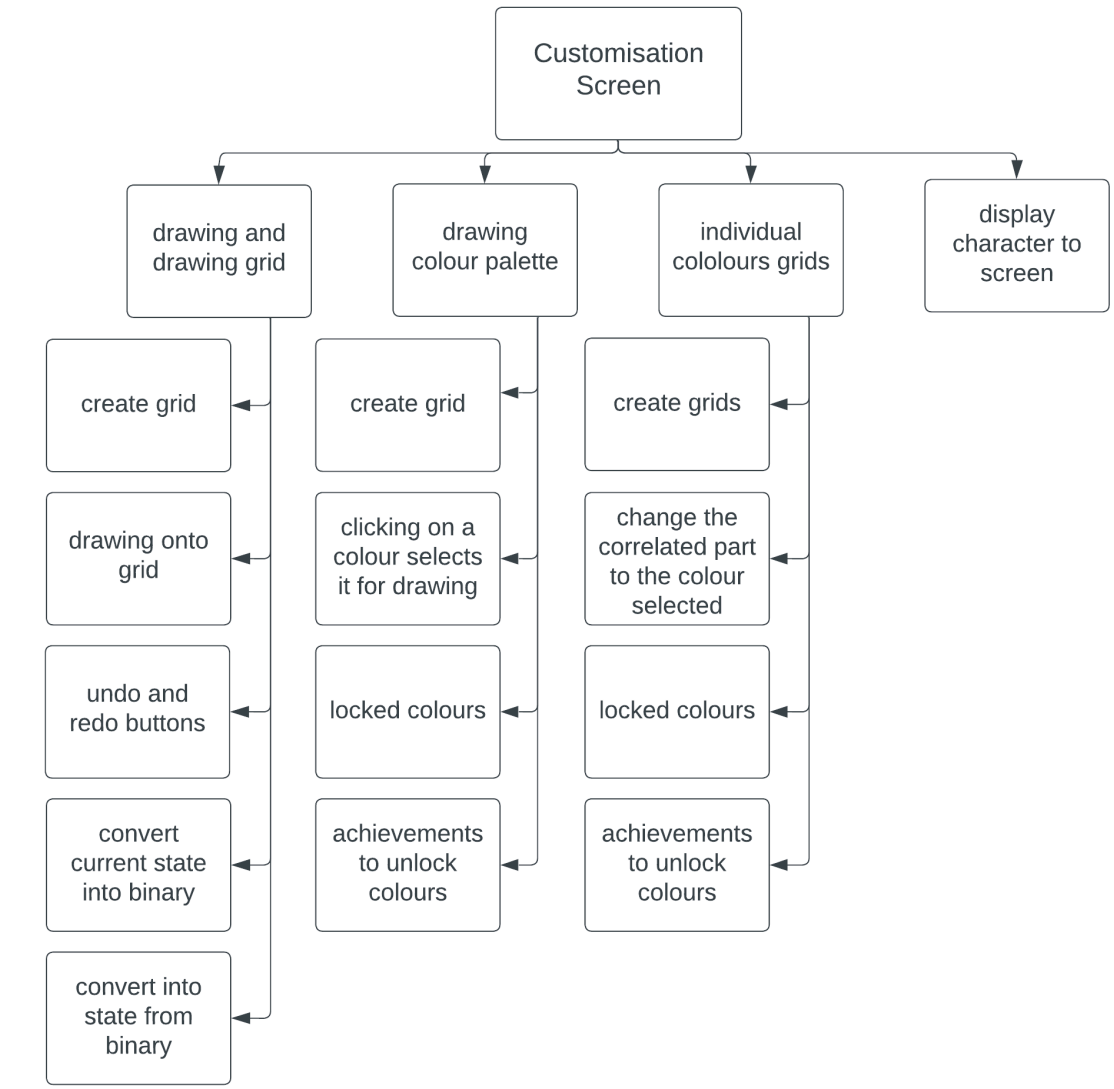
In my implementation, I opted to create a slider for volume control instead of using arrow buttons as I found it way more intuitive and more fun to play with.

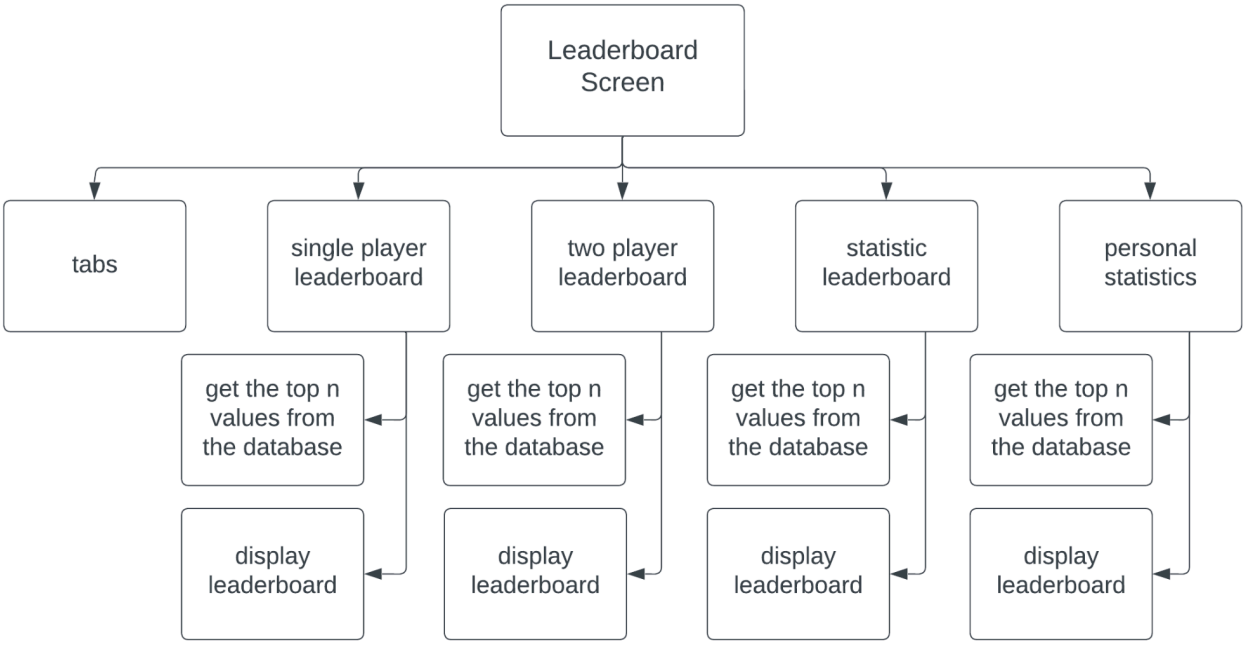
2.5 Structure Diagrams

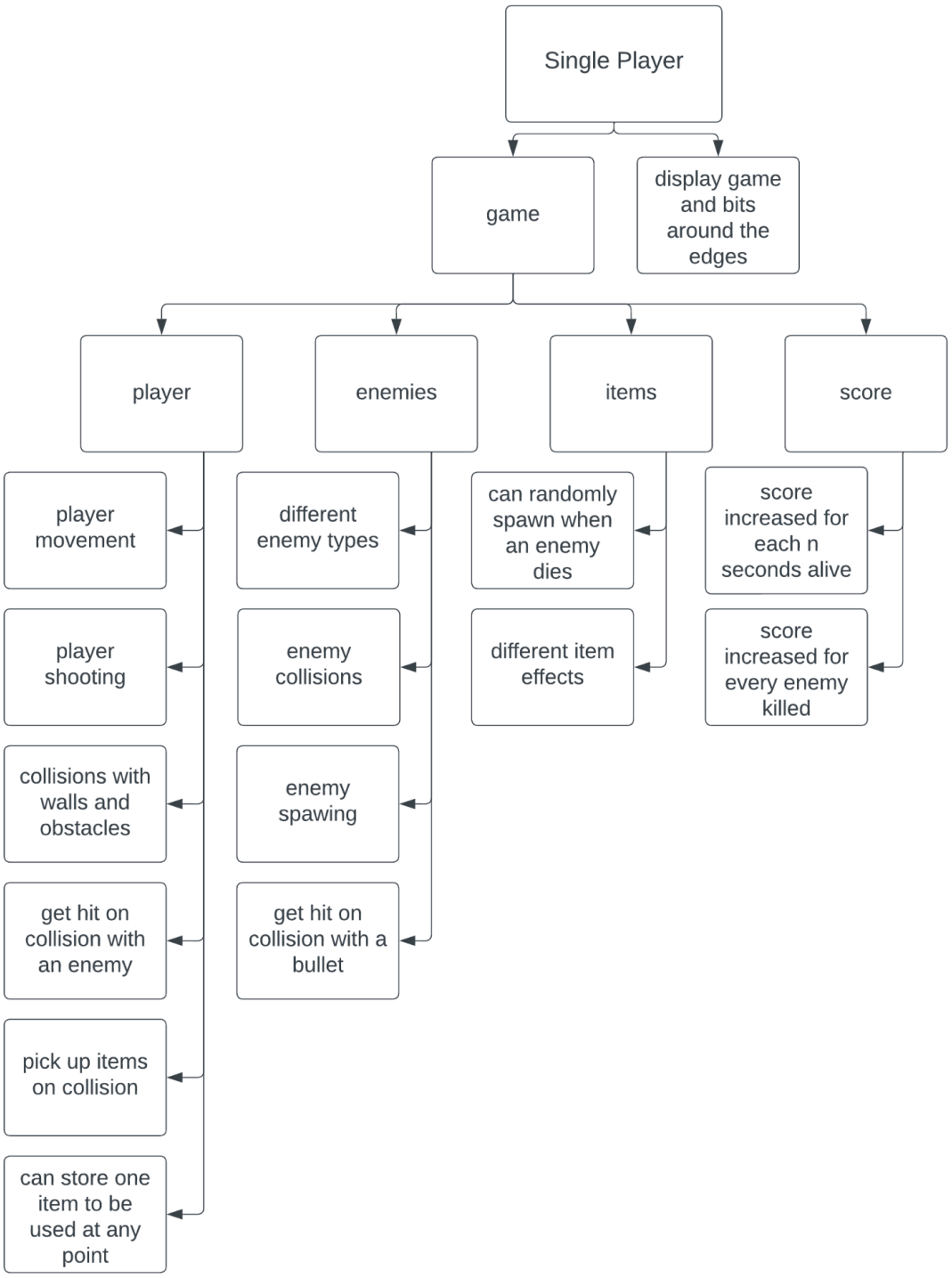







Customisation Screen
drawing and drawing grid
drawing colour palette
individual cololours grids
display character to screen
create grid
drawing onto grid
undo and redo buttons
convert current state into binary
convert into state from binary
create grid
clicking on a colour selects it for drawing
locked colours
achievements to unlock colours
create grids
change the correlated part to the colour selected
locked colours
achievements to unlock colours

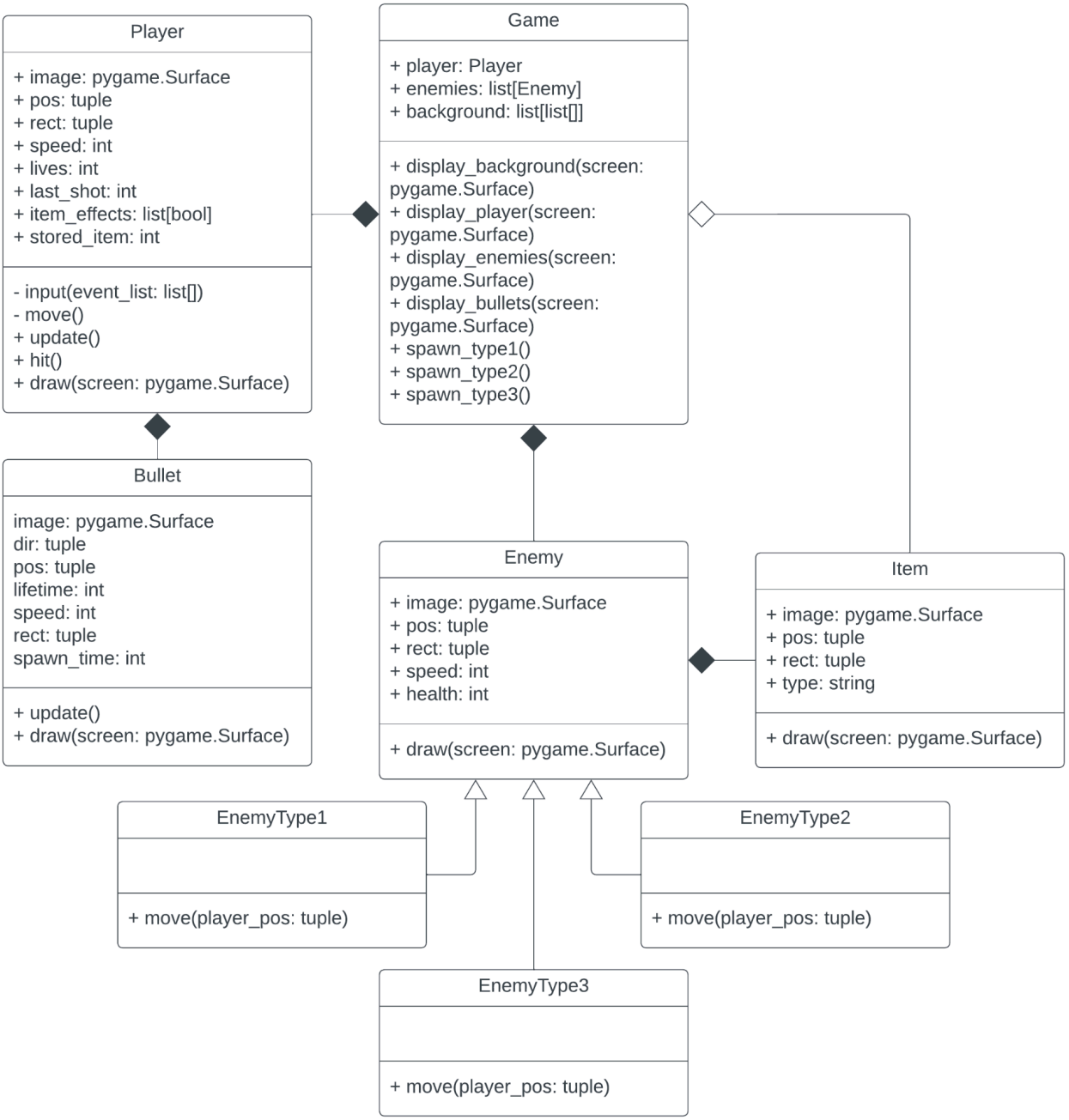

Leaderboard Screen
tabs
single player leaderboard
two player leaderboard
statistic leaderboard
personal statistics
get the top n values from the database
display leaderboard
get the top n values from the database
display leaderboard
get the top n values from the database
display leaderboard
get the top n values from the database
display leaderboard



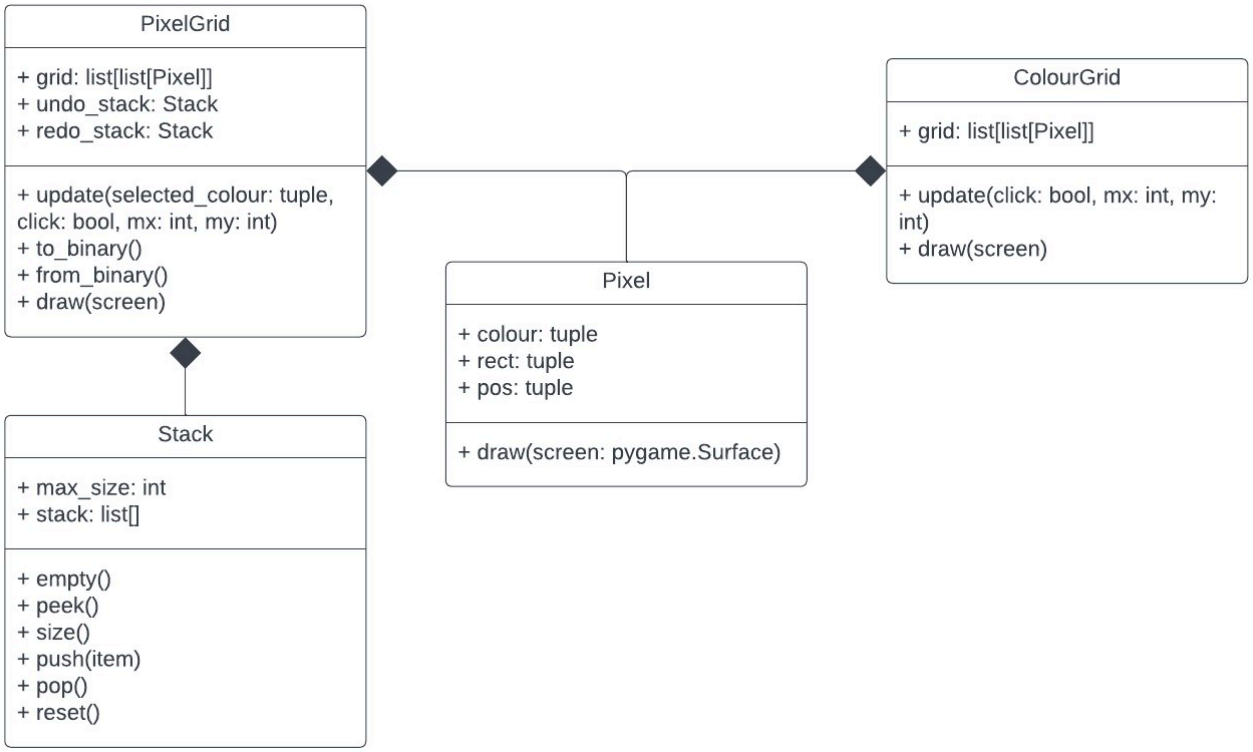
These diagrams are very useful as they decompose the overarching tasks into small and manageable ones that I can understand and complete.

2.6 Class Diagrams

Proposed classes for the game:



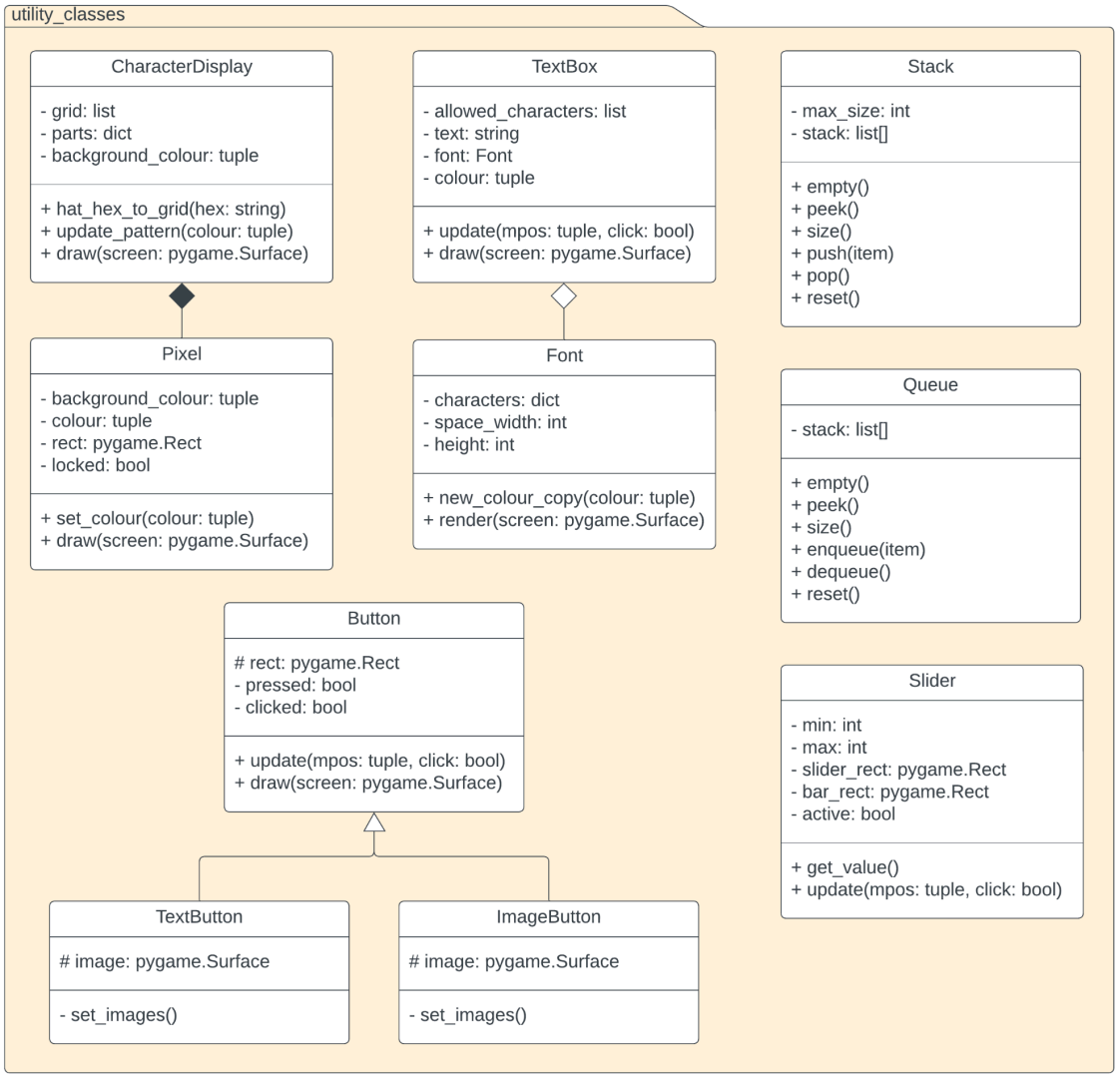
Proposed classes for the customisation menu and drawing application:



These classes are suggestions based on prototyping and experience. There will likely be more classes to implement and they will almost definitely have different attributes and functions but creating these diagrams allowed me to start to actually think about how I will implement each part of the game and will make it much easier when it comes to that stage.

I did indeed underestimate the complexity and needed a lot more classes. Over the next few pages is an updated class diagram that I have been making throughout implementation. I tried to make it as easy to understand as I could, leading me to organise the classes into packages which are similar to different files and imports of the actual implementation. Not every method or attribute was added to the diagram as it would make some classes way too long and I felt that a lot of these (especially getters and setters) were not necessary to include and would only overcomplicate the diagram.

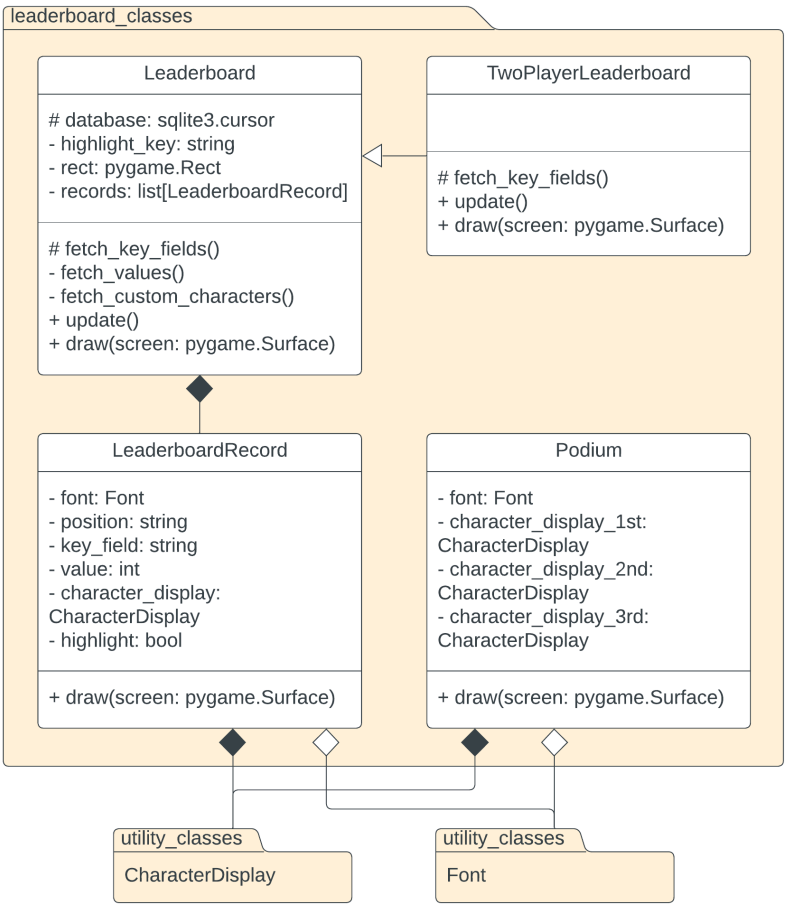
The first is the utility classes package. This contains classes that are used in multiple places across the project or have general functionality that isn't tied to any implementation. This includes classes for a character display that displays a custom character hex string in its uncoded form, a pixel that is mostly just a coloured rectangle, a button that can be hovered over and pressed and is a base class for a text button and an image button that have text and images for icons respectively, a font that takes in an character set image and a character set list to create a renderable font, a text box that the user can type into and checks for the validity of entered strings, a slider that allows you to grab and slide a rectangle to return a value (used for the volume setting), a stack, and a queue.

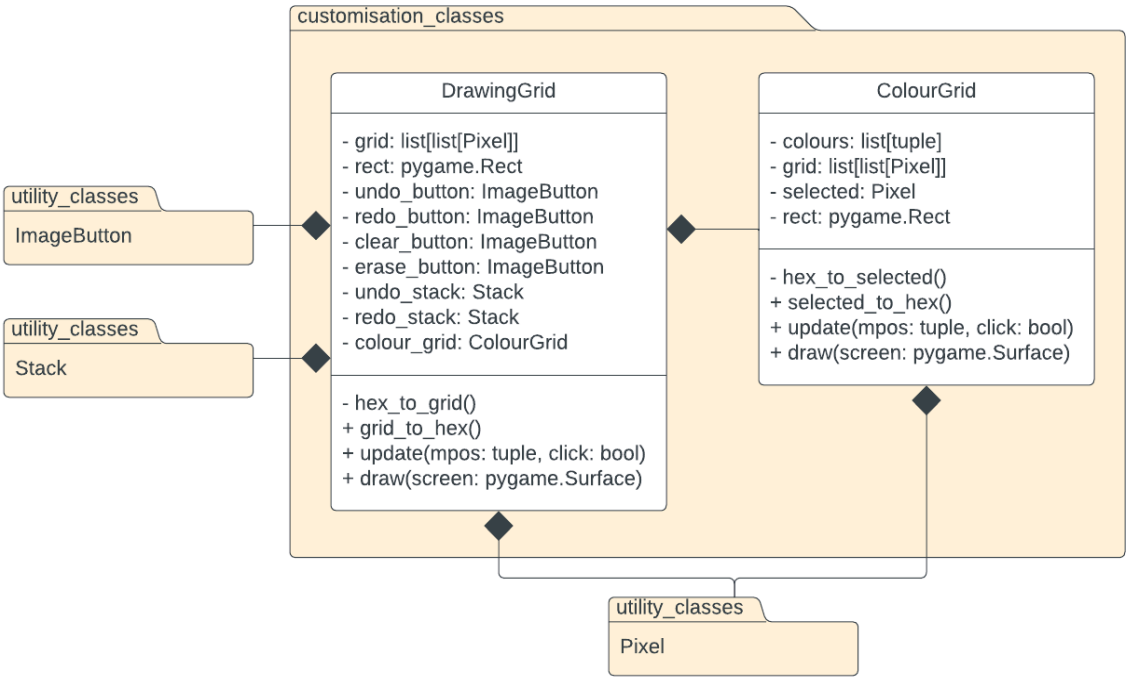


Following this is the leaderboard and customisation classes packages.

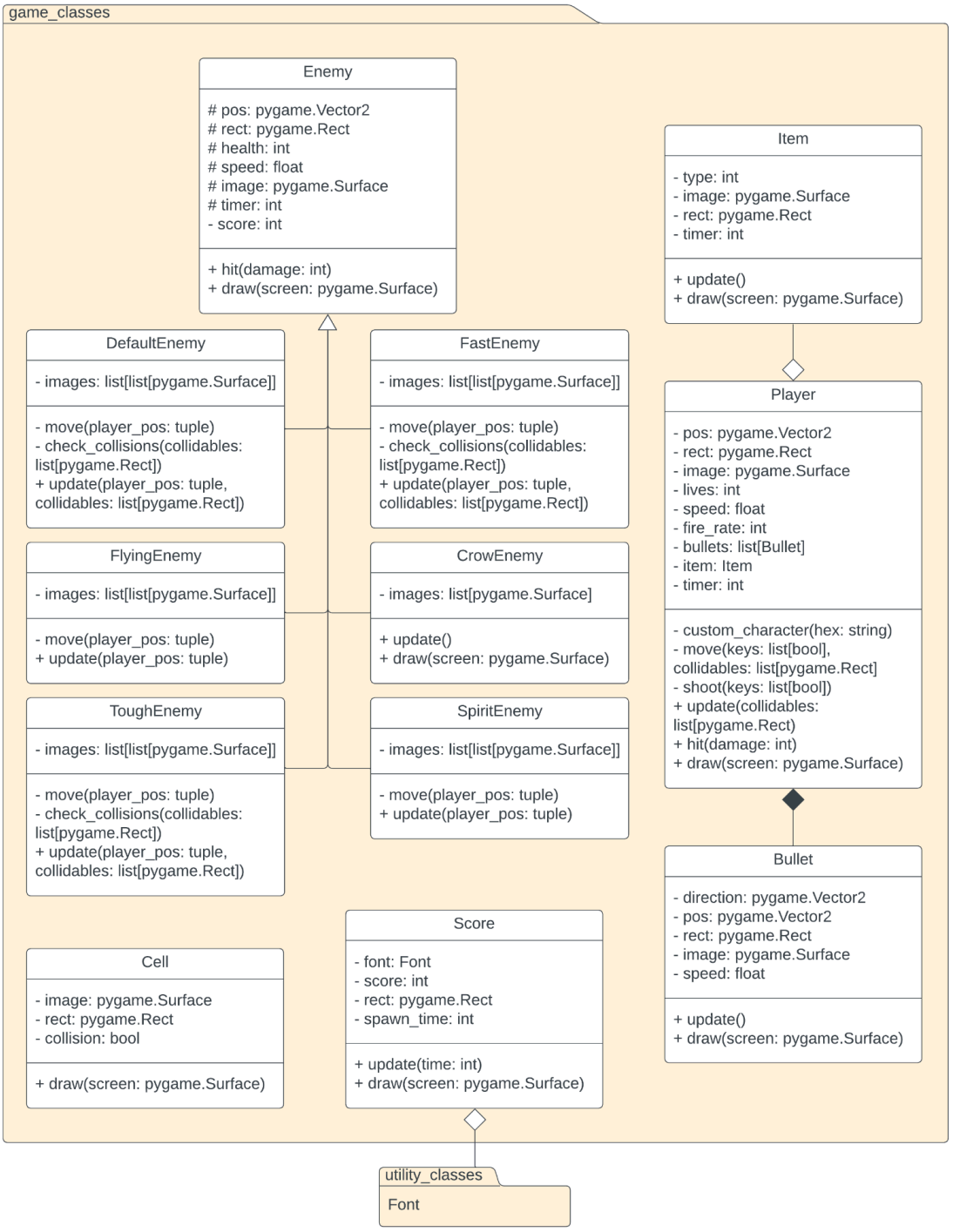
The leaderboard classes package contains classes used for the leaderboards of the project that are displayed in the main menu, the leaderboards screen, and the post-game score screen. This includes classes for a leaderboard that queries the local database based on parameters to create a displayable leaderboard, a two player leaderboard that inherits from the leaderboard class and allows for multiple usernames, a leaderboard record that is initialised for each record in the leaderboard classes with attributes for each of its constituent parts, and a podium which queries the local database for and displays the 3 players with the highest number in a specified statistic.

The customisation classes package contains classes used in the customisation screen for customising a character. This includes a colour grid which is a row of pixels that can be selected and hovered over using the mouse, and a drawing grid which allows users to draw anything onto a grid, using colours provided as well as buttons to clear, undo, redo, and erase, and then encode it into hexadecimal.

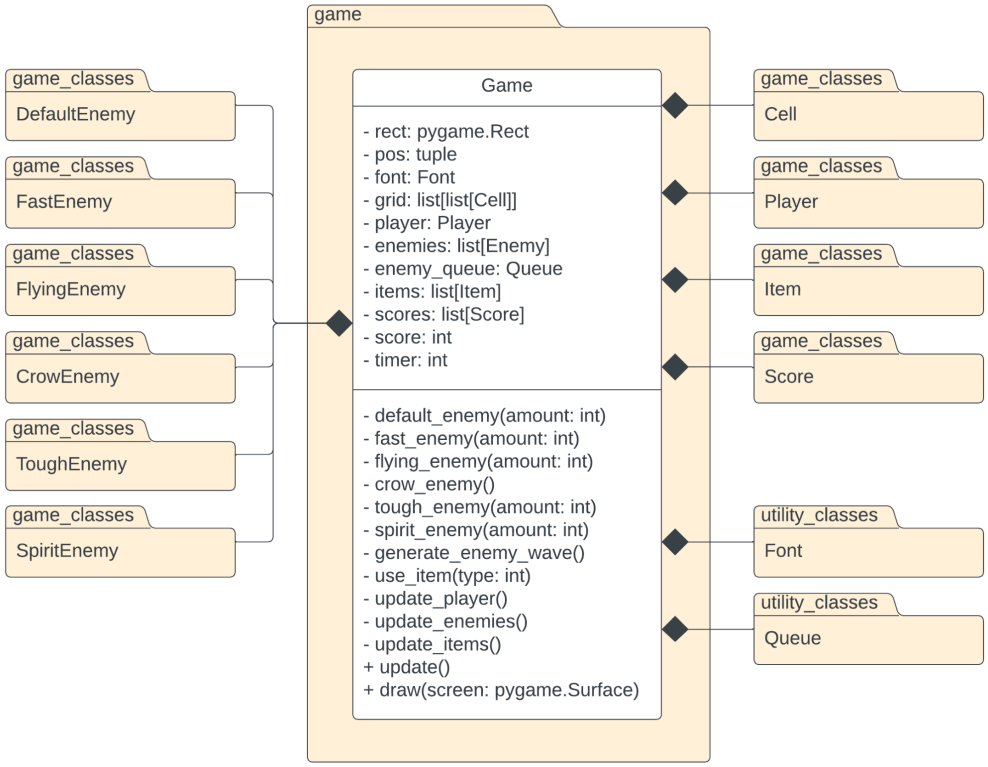

leaderboard_classes
Leaderboard
database: sqlite3.cursor
- highlight_key: string
- rect: pygame.Rect
- records: list[LeaderboardRecord]
fetch_key_fields()
- fetch_values()
- fetch_custom_characters()
+ update()
+ draw(screen: pygame.Surface)
TwoPlayerLeaderboard
fetch_key_fields()
+ update()
+ draw(screen: pygame.Surface)
LeaderboardRecord
- font: Font
- position: string
- key_field: string
- value: int
- character_display: CharacterDisplay
- highlight: bool
+ draw(screen: pygame.Surface)
Podium
- font: Font
- character_display_1st: CharacterDisplay
- character_display_2nd: CharacterDisplay
- character_display_3rd: CharacterDisplay
+ draw(screen: pygame.Surface)
utility_classes
CharacterDisplay
utility_classes
Font


customisation_classes
DrawingGrid
- grid: list[list[Pixel]]
- rect: pygame.Rect
- undo_button: ImageButton
- redo_button: ImageButton
- clear_button: ImageButton
- erase_button: ImageButton
- undo_stack: Stack
- redo_stack: Stack
- colour_grid: ColourGrid
- hex_to_grid()
+ grid_to_hex()
+ update(mpos: tuple, click: bool)
+ draw(screen: pygame.Surface)
ColourGrid
- colours: list[tuple]
- grid: list[list[Pixel]]
- selected: Pixel
- rect: pygame.Rect
- hex_to_selected()
+ selected_to_hex()
+ update(mpos: tuple, click: bool)
+ draw(screen: pygame.Surface)
utility_classes
ImageButton
utility_classes
Stack
utility_classes
Pixel

Next is the game classes package which contains a variety of classes used for the actual game itself. This includes a cell class that makes up the grid of the game being able to have an image and a collision attribute, a score class that is initialised when an enemy dies and shows the score they were worth for a small period of time, an item class that can be initialised based off of random chance when an enemy dies that has a type and an image, a player class that creates a controllable player that can move and shoot, and an enemy class that is a base class for the default enemy class, fast enemy class, flying enemy class, crow enemy class, tough enemy class, and spirit enemy class, which each have their own movement patterns, images, health, and score.



Finally is the game package. This contains the overarching class for the game itself, composed of all the different game classes. The game class creates and spawns waves of enemies, controls the items and spawning of obstacles, lets enemies be hit by the player's bullets, and everything else. This class has its own package rather than being with the other game classes as the file for the game classes was getting too large and I think giving this class its own file also makes it easier to understand.



2.7 Data Structures

2.7.1 Queues

For the game itself, a queue will be used to be filled with and then hold groups of enemies that have yet to be spawned. When the game detects that there are no enemies on the screen and the queue is empty, a function will be run that fills the queue with many groups of enemies. Each group contains one or many of an enemy object that can be dequeued and added to the list of currently spawned enemies. This dequeuing will occur every few seconds, providing a constant threat to the player, with this time between dequeuing decreasing as you play and also influenced by how many enemies the group contains.

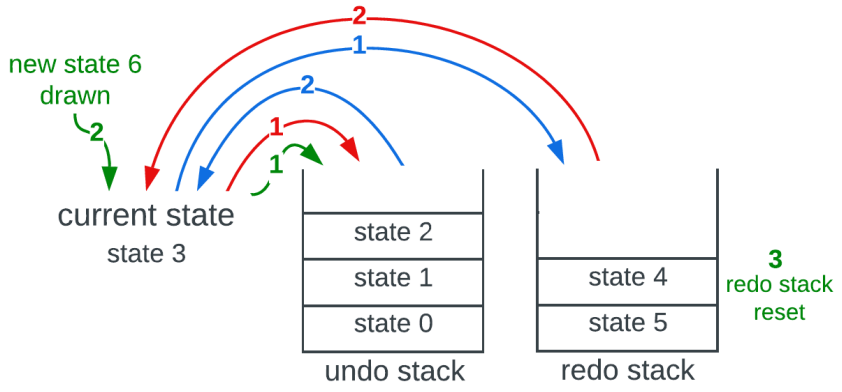
2.7.2 Stacks

For the customisation part of the game, the user will be able to draw their own pixel art hat on a grid and use undo and redo buttons to undo and redo their changes. The undo button will require a stack that holds the previous states of the grid. Every time the user changes this grid, the program will push the prior state onto the stack. Then, when the user wishes to undo, the program will pop states from the top of the undo stack and return the grid to this state, allowing the user to restore all the previous states starting from the most recent state down to the initial state from the bottom of the stack. This logic still works when the user draws onto the grid after undoing, pushing the new states on top of the old states that had yet to be undone, with the states in between being forgotten.

The redo button will also require a stack that holds all the recent undos. Every time the user presses the undo button, the program will push the state prior to the undo onto the redo stack. Then, when the user wishes to redo, the program will pop states off the top of the redo stack and return the grid to this state, allowing the user to restore the states they just undid. When the user draws onto the grid, this redo stack must be emptied, as the redo button is only for undoing the undos just done or else the order of states could turn into chaos and not work as user friendly undo and redo buttons.

When adding these states to the stack, they will first be converted into a string format using a dictionary of colours and codes that represent them, similar to how an uncompressed bitmap image is represented. This simplifies the stacks and the data they store and also will be used when storing each user's custom character in the database, converting them into a string before storing them. I will also need to convert the current state of the grid into the state described by a string when popping states off of the stacks.

This diagram shows the operation of the stacks with user input (either draw, undo, or redo):



undo

1	the current state is pushed onto the redo stack (state 3 would be pushed onto the redo stack)
2	the top state is popped off the undo stack and the current state is set to it (state 2 would be popped off the undo stack and the current state set to it)

redo

1	the current state is pushed onto the undo stack (state 3 would be pushed onto the undo stack)
2	the top state is popped off the redo stack and the current state is set to it (state 4 would be popped off the undo stack and the current state would be set to it)

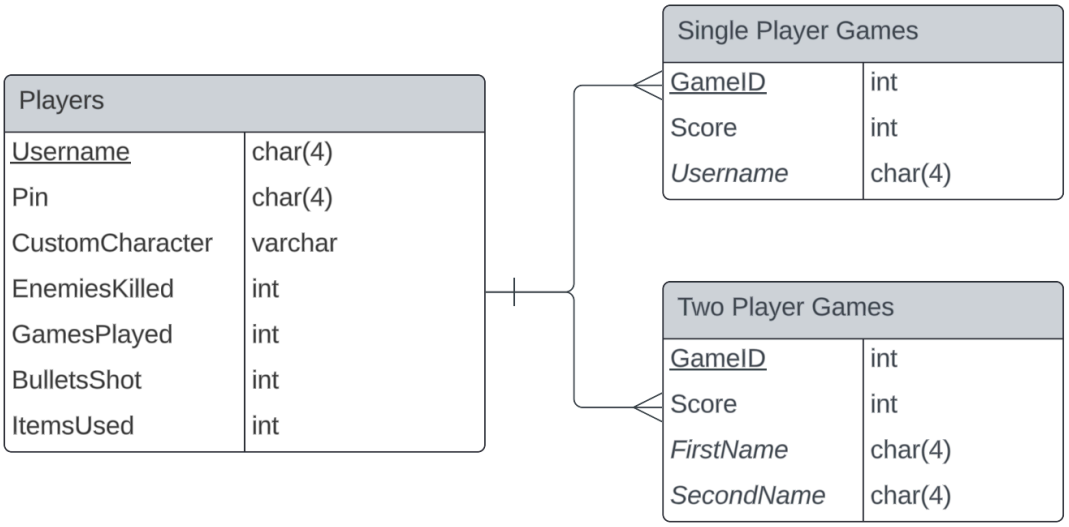
draw

1	the current state is pushed onto the undo stack (state 3 would be pushed onto the undo stack)
2	the current state is set to the new state drawn (the current state would be set to state 6)
3	the redo stack is emptied and its data lost

2.8 The Database

2.8.1 Design and Entity-Relationship Diagram

Here is the ER diagram for my database:



The Players table stores data about every player who plays the game. Each player has their unique 4 character username as a primary key as well as their pin that is used to log into the account. It also stores their custom character, in a format similar to a typical bitmap image format, and their statistics.

The Players table has a one to many relationship with the Single Player Games table, as each player can have many single player games associated with their username. The Players table also has a one to many relationship with the Two Player Games table, as each player can have many two player games associated with their username.

The Single Player Games table stores the player's username and score for every game played. It may become necessary when it comes to implementation to remove some records every so often as the only records needed in this table are those with the highest scores and storing data about every game may become very difficult as more and more games are played.

The Two Player Games table works in the same way as the Single Player Games table, just with an extra field for the second player's name that is also a foreign key to the username field in Players.

2.8.2 Queries

These are prospective queries to be used to create, update, and insert into the database.

I don't need to protect against SQL injection when creating my queries as the only time the user can enter text for a change in the database is when typing in their username, which is only 4 characters long and will not be able to contain special characters.

Creating the player table

This query will create the player table with the fields Username, CustomCharacter, EnemiesKilled, GamesPlayed, BulletsShot, and ItemsUsed. The Username is always 4 characters. CustomCharacter could change in length depending on the number of colours and the bit depth so I put it at 150 ((8*3 + 6)*5) to represent the likely maximum bit depth of 5, allowing for 32 different colours, with an 8 x 3 grid of colours for the hat and 6 colours for the rest of the character.

```
CREATE TABLE tblPlayers(
	Username CHAR(4) NOT NULL PRIMARY KEY,
	Pin CHAR(4) NOT NULL
	CustomCharacter VARCHAR(150) NOT NULL,
	EnemiesKilled INT NOT NULL,
	GamesPlayed INT NOT NULL,
	BulletsShot INT NOT NULL,
	ItemsUsed INT NOT NULL
)
```

The length of the custom character string did change due to the fact I swapped from using a binary string to using a hex string for the custom character, reducing from 150 characters required to 60.

Creating the single player games table

This query will create the single player games table with the fields GameID, Score, and Username. I want GameID to auto increment so that the database handles giving each entry a unique ID instead of me having to create unique IDs in the code.

```
CREATE TABLE tblSinglePlayerGames(
	GameID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
	Score INT NOT NULL,
	Username CHAR(4) NOT NULL FOREIGN KEY REFERENCES tblPlayers(Username)
)
```

Creating the two player games table

This query will create the two player games table with the fields GameID, Score, FirstName, and SecondName.

```
CREATE TABLE tblTwoPlayerGames(
	GameID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
	Score INT NOT NULL,
	FirstName CHAR(4) NOT NULL FOREIGN KEY REFERENCES tblPlayers(Username),
	SecondName CHAR(4) NOT NULL FOREIGN KEY REFERENCES tblPlayers(Username)
)
```

Add a new player

This query will be used to add a new player to the database when they create a new account. I am assuming that I will have strings called username, pin, and character_binary when adding a new player however I may not and would then need to change this.

```
INSERT INTO tblPlayers(Username, Pin, CustomCharacter, EnemiesKilled, GamesPlayed,
BulletsShot, ItemsUsed)
VALUES (username, pin, character_binary, 0, 0, 0, 0)
```

Check if a unique username is entered

This query will be used to check if the user has entered a unique username after they have tried to type one in. EXISTS returns TRUE if any records are returned in the subquery. I am assuming that I will have a string called username when I check for a unique entry however I may not and would then need to change this.

```
EXISTS (SELECT * FROM tblPlayers WHERE Username = username)
```

Inserting a game into the single player games table

This query will be used at the end of a single player game when a new record needs to be added to the single player games table. I am assuming that I will have the variables score and username, as an integer and a string respectively, when the game is ended and the query is executed. GameID uses auto increment so I do not need to insert into this field.

```
INSERT INTO tblSinglePlayerGames (Score, Username)
VALUES (score, username)
```

Inserting a game into the two player games table

This query will be used at the end of a two player game when a new record needs to be added to the two player games table. I am assuming that I will have the variables score and username, as an integer and a string respectively, and that I will have asked the user to enter a second name stored in the variable second_name when the query is executed. GameID uses auto increment so I do not need to insert into this field.

```
INSERT INTO tblSinglePlayerGames (Score, FirstName, SecondName)
VALUES (score, username, second_name)
```

Update the players database at the end of a game

This query will be used at the end of a game when the players table needs to be updated to hold the new updated values for the players EnemiesKilled, GamesPlayed, BulletsShot, and ItemsUsed. I am assuming that I will have variables called enemies_killed, games_played, bullets_shot, and items_used, all as integers, and username as a string at the end of a game when the query is run.

```
UPDATE tblPlayers
SET EnemiesKilled = enemies_killed, GamesPlayed = games_played, BulletsShot =
bullets_shot, ItemsUsed =  items_used
WHERE Username = username
```

Update the database when the player changes their custom character

This query will be used when the user saves their character on the customisation screen. I am assuming that I will have the variables character_binary as the string of binary converted from the character grid and username as a string when the user saves their character and the query is executed.

```
UPDATE tblPlayers
SET CustomCharacter = character_binary
WHERE Username = username
```

Retrieve data for the single player leaderboard

This query will be used to get the data from the database for the single player leaderboard. I want to retrieve the top 10 scores from the database along with their related username and custom character. To only get the top 10, I use descending order of the scores and limit the results to 10. If I need to retrieve a different number of scores, I can change the limit number.

```
SELECT Score, Username, CustomCharacter
FROM tblPlayers, tblSinglePlayerGames
WHERE tblPlayers.Username = tblSinglePlayerGames.Username
ORDER BY Score DESC
LIMIT 10
```

Retrieve data for the two player leaderboard

This query will be used to get the data from the database for the two player leaderboard. I want to retrieve the top 10 scores from the database along with their related names. To only get the top 10, I use descending order of the scores and limit the results to 10. If I need to retrieve a different number of scores, I can change the limit number.

```
SELECT Score, FirstName, SecondName
FROM tblTwoPlayerGames
ORDER BY Score DESC
LIMIT 10
```

Retrieve data for the statistics leaderboard

These queries will be used for the statistics leaderboard to get the top 3 players in enemies killed, games played, bullets shot, and items used, along with their custom characters.

```
SELECT EnemiesKilled, Username, CustomCharacter
FROM tblPlayers
ORDER BY EnemiesKilled DESC
LIMIT 3

SELECT GamesPlayed, Username, CustomCharacter
FROM tblPlayers
ORDER BY GamesPlayed DESC
LIMIT 3

SELECT BulletsShot, Username, CustomCharacter
FROM tblPlayers
ORDER BY BulletsShot DESC
LIMIT 3

SELECT ItemsUsed, Username, CustomCharacter
FROM tblPlayers
ORDER BY ItemsUsedDESC
LIMIT 3
```

I also started to experiment with using the built in sqlite3 Python library. Using the library is actually very simple. First you must connect to the database, which can be done by running the following code (after importing sqlite3):

```
db_connection = sqlite3.connect("database.db")
```

This looks in the current directory for a database file called “database.db” and then connects to it, creating the file if it does not already exist.

From this connection, we can initialise a database cursor that allows us to query and execute commands on the database, which can be done by running the following code:

```
db_cursor = db_connection.cursor()
```

Executing SQL with this cursor is also very simple and is compatible with the same syntax I used in the queries above. The code to create the player table can be as follows:

```
query = """CREATE TABLE tblPlayers(Username CHAR(4) NOT NULL PRIMARY KEY,
                                   Pin CHAR(4) NOT NULL,
                                   CustomCharacter VARCHAR(150) NOT NULL,
                                   GamesPlayed INTEGER NOT NULL,
                                   EnemiesKilled INTEGER NOT NULL,
                                   BulletsShot INTEGER NOT NULL,
                                   ItemsUsed INTEGER NOT NULL)"""
db_cursor.execute(query)
```

After executing the query on the database, the changes must be committed in order to actually change the database that is stored. This can be done using the following code:

```
db_connection.commit()
```

This is only required when the database is actually changed, not every time it is queried.

I can now insert data into the database using the following code:

```
db_cursor.execute("INSERT INTO tblPlayers VALUES (?,?,?,?,?,?,?)",
                  ("TEST","1234","01001010",6,87,142,12))
```

The binary used here is just some random placeholder bits and this does require committing after.

To query this database, I can similarly use the execute method of the cursor:

```
db_cursor.execute("SELECT * FROM tblPlayers")
```

The values from this query now need to be retrieved, which can be done through another cursor method that fetches the result of the query into a variable:

```
result = db_cursor.fetchall()
```

This result variable can now be printed which provides the following output:

```
[('TEST', '1234', '01001010', 6, 87, 142, 12)]
```

The format of the output is a tuple of values in a list. If I were to use the fetchone() method of the cursor instead of fetchall() method, the output would not be in a list as the code would know that I am only trying to retrieve a single value.

2.9 Data Flow Diagram

This diagram shows the flow of data between data stores, such as the database and local file storage, in the game and the menus. Creating this diagram helped me visualise and understand where and when I will need to access and update the different data stores. Colours are featured to help differentiate the different sections.

2.10 Assets

2.10.1 Enemies

In this section I will decide on what enemies to include in the game. This section may change throughout development as deciding on the enemies will require some sort of test that can show how interesting or not interesting the enemies are.

This is the initial table of ideas for enemies that I made during the analysis stage:

Name	**HP**	**Description**	**Level**
Simple enemy	1	slow predictable movement somewhat towards the player spawn in medium sized hordes from the sides	1
Flying enemy	1	moves directly towards the player spawns alone and from the sides	3
Spike enemy	3, 7	picks a random location on the inner map, runs towards it and then turns into a spike hazard 2 health when running, 7 when it becomes a spike spawns alone and from the sides	1
Fast enemy	2	fast but predictable movement spawns alone and from the sides	2
Grenade enemy	1	runs to the centre and explodes into 8 shrapnel projectiles, one firing in each direction spawns alone and from the sides	4
Unpredictable enemy	2	fast and unpredictable movement, affected by randomness spawns alone and from the sides	5
Tough enemy	3	very slow predictable movement spawns in large hordes and from the sides	6
Bomb enemy	1	normal movement, when shot they explode after a few seconds, damaging you and enemies	7

The enemies I have taken from this list to add to the game are the simple enemy (renamed as the default enemy), the flying enemy, the fast enemy, and the tough enemy.

I didn't use the other enemies because I thought they had mechanics that were too complex to be able to easily communicate with the player and they could be very challenging to implement.

I also added two more enemies, the spirit enemy and the crow enemy displayed in the next table as I thought they had interesting features and I wanted more flying enemies rather than just the one.

This is the final table of the enemies I used in the game and their art:

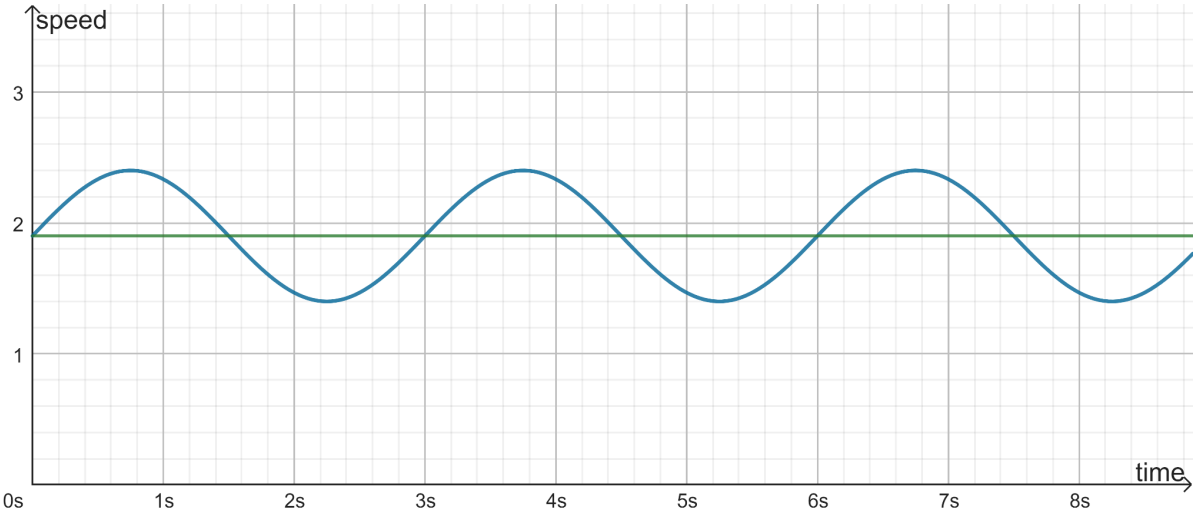
Art	Name	HP	First Wave	Flying/ Ground	Movement
	Default Enemy	1	1	Ground	moves towards the player in straight lines influenced heavily by randomness spawns in big groups
	Fast Enemy	2	1	Ground	runs straight until it meets the player either horizontally or vertically then changes direction towards the player spawns in small groups
	Flying Enemy	1	2	Flying	moves directly towards the player and over obstacles speed varies sinusoidally* spawns in small groups
	Crow Enemy	1	3	Flying	moves very fast in a straight line across the screen warns the player of its position before moving spaws alone
	Tough Enemy	2	4	Ground	moves towards the player in straight lines influenced slightly by randomness spawns in big groups
	Spirit Enemy	3	5	Flying	moves directly towards the player and over obstacles speed does not vary spawns in small groups

The wave column represents the first wave that they can spawn in (waves are separated by the spawning of a new obstacle).

I chose these enemies as they provide a nice range for the game and are also quite easy to understand.

*For the flying enemy, I thought the direct movement towards the player was a little plain and didn't look very interesting. To fix this, I varied the enemy's speed using a sine wave.

Below is a graph of speed against time for the wave used, plotted in GeoGebra[15]:



The abstracted code for this where timer is a variable that is incremented every frame:

```
speed = sin((timer * 2*pi) / (3*FPS)) / 2 + SPEED
```

The reason I multiply by 2 pi is because the sine function uses radians. The reason I divide by 3*FPS is to alter the speed over a cycle of 3 seconds which I just thought worked well. The reason I divide by 2 is to half the effect it has on the speed, meaning it only changes from the mean line by 0.5 instead of 1 as a standard sine wave varies between -1 and 1.

Although I thought it looked plain for the flying enemy, I left the speed constant for the spirit enemy. This is because I wanted the spirit enemy to feel slightly creepy, which is also why I didn't give it any animation, and just different from the flying enemy.

2.10.2 Items

In this section I go over what items I implemented into the game and what they do. This was created alongside the implementation as creating items requires testing to see if they are fun before I can decide whether they should be in the game.

This is the initial table of ideas for items that I made during the analysis stage:

Name	**Description**
Life	increases life count by 1
Boots	temporarily increases movement speed
Machine gun	temporarily increases rate of fire
Shotgun	temporarily shoot bullets in a 3-way spread but with a slower rate of fire
Invulnerability	temporarily make the player invulnerable to any damage
Stun bomb	all enemies stop moving for a few seconds but can still damage and be damaged
Bomb	clear all enemies from the screen, they don't drop any items
Piercing	bullets can travel through enemies and kill any enemy in a line
Anything	2/3 chance you get a random item from the list 1/3 chance your gun temporarily shoots bubbles

I used almost all of the items from the list with some name changes and slight functionality changes. I didn't implement the 'invulnerability' item (although I have an invulnerable attribute for the player that is used in the two player mode) because it was confusing and very powerful. I didn't implement the 'anything' item as it would be very confusing to use and I also didn't like the idea of items possibly hindering your ability. And although I had implemented the 'piercing' item and it had stayed in the game for a long time, I decided to remove it as it was very powerful, confusing, and brought up problems on whether the bullets should instantly kill enemies with health greater than 1 or if it should hit them once and travel through them.

I added a 'backwards shot' item, because I thought it was an interesting and simple item to add. I also altered the 'stun bomb' item to become the 'time freeze' item because after implementing it and creating art for it, it was not obvious what it would do and what it was meant to be, so now the art is a clock to demonstrate the altering of time.

This is the final table of items I used in the game and their art:

Art	Name	Description
	Heart	increases the player's lives by 1
	Shotgun	the player temporarily shoots in a 3 way spread while the item is activated, the player's fire rate is slightly decreased
	Boots	the player temporarily has an increased movement speed
	Rapid Fire	the player temporarily has an increased fire rate
	Bomb	all enemies within a certain radius of the player are immediately killed for no score or items
	Time Freeze	time becomes temporarily frozen enemies stop moving, items don't disappear, but the player can still move
	Backwards Shot	the player temporarily shoots backwards as well as forwards

The life item is very powerful and probably the best item therefore I made it so it can only spawn when the player has or one of the two players have less than 4 lives.

2.10.3 Animations

In early prototypes of the game, the enemies would just float towards the player and it looked a little strange. I realised this was because they had no animation; they were indeed just floating towards the player and there was no walking animation to provide the illusion of walking. So in order to add animations, I would have to figure out how to cycle an enemy's images through frames while also having different animations and sprites for the enemies when moving in each direction.

This is a grid of the animation frames for the default enemy:

	0	1	2	3
up				
left				
down				
right				

For each direction the enemy can face, there are 4 animation frames to create the illusion of walking. The frames can look unnatural by themselves but when they are played one after the other they work well to look like walking.

This grid, as you can see, looks a lot like a 2D array. And so, my method of creating animations is to put these frames into a 2D array, with rows for each direction and columns for each frame index, and to have a 'direction' attribute for each enemy and an index calculated when the enemy is drawn to correctly find the enemy's image and then display it to the screen.

The four constants for indexing the rows of the 2D array:

```
UP = 0
LEFT = 1
DOWN = 2
RIGHT = 3
```

Having directions be represented by numbers also helps when adding random changes to the direction of enemies. For example, I can add 1 or take away 1 from the enemy's direction and then perform the direction modulo 4 (because there are 4 directions) to get them to turn 90° to the left or 90° to the right respectively.

The way I calculate the column index is using a timer that gets incremented by 1 for every frame. I first divide this timer by the FPS to isolate the timer from the FPS and get it into seconds, then multiply by 4 for an animation cycle of 4 frames per second, then perform this value modulo the number of frames in the animation (found by finding the length of the first row in the 2D array), and then finally truncate this value to get an integer.

The abstracted code for indexing the 2D array:

```
image = images[direction][trunc(((timer / FPS) * 4) % len(images[0]))]
```

Now, the default enemy cycles through its animation at 4 frames per second and is also able to change direction with the animation following. This was implemented for almost every other enemy (except for the spirit enemy and crow enemy which I thought had better effects without any animation) with differing frames per second to achieve different effects, for example the fast enemy has a cycle of 8 frames per second and only 2 animation frames to help it look like it is moving fast.

2.10.4 Audio

As per my objectives, audio feedback is quite important. It also can add a lot to a project and games especially. Sound effects help bring a game to life, make it immersive, and also can assist players in understanding their actions or what is happening around them.

Some of the most important sounds in the game are the button interaction sounds. These are the sound that plays when you hover over a button and the sound that plays when you click a button. To find these sounds, I searched on many sites and tested many different sounds and decided on two effects from SoundSnap[16][17], a professional sound effect site that allows non-paying users up to 5 free sounds per month. Actually implementing the sounds was pretty easy using Pygame's in-built mixer class.

The syntax for loading a sound into a variable:

```
button_click = pygame.mixer.Sound("assets/sounds/button click.wav")
```

The syntax for changing the volume of a sound, 0.5 meaning 50% of its original volume:

```
button_click.set_volume(0.5)
```

The syntax for playing a sound:

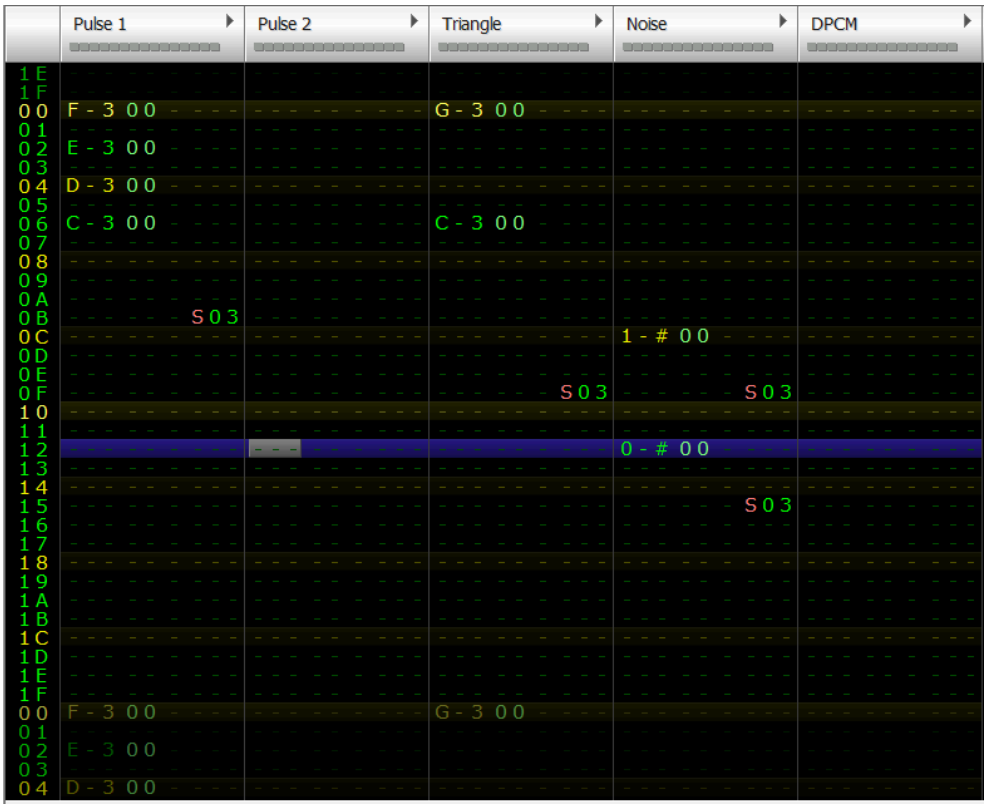
```
button_click.play()
```

The same syntax is used for every sound in the game.

Another important sound is the sound for shooting a bullet as the player will hear this sound very frequently. After searching and testing different options, I found a good retro arcade-style bullet sound effect on Freesound[18][19], a large collaborative database of sound effects. Whenever the player shoots a bullet, this sound is played and provides a nice retro feel

For a lot of sound effects in the project I created them myself in FamiTracker[13], a tool for making MIDI music, based on the 8-bit sound systems from old game consoles like the NES. These include the sounds that are played when you get hit, when you lose the game, when you kill an enemy, when you use an item, and when you use a bomb.

Below is a screenshot of the game over sound file in FamiTracker:



The way FamiTracker works is that there are multiple channels, defaulted to the 5 channels found on the NES' Ricoh 2A03 sound chip, aligned vertically (unlike most other MIDI sound editors that are aligned horizontally) that can each have a different arrangement of MIDI events. For each MIDI event there are separate columns to specify the note, the instrument, the volume, and an effect. When you press play, the MIDI events are played row by row from the top down at a specified BPM. For the game over sound, I used a simple descending scale in the pulse 1 channel with a harmonising track in the triangle channel, followed by 2 descending noise effects in the noise channel.

Once a track has been created in FamiTracker, it can easily be exported as a .wav file and then used in the project in the same way all other sounds are used.

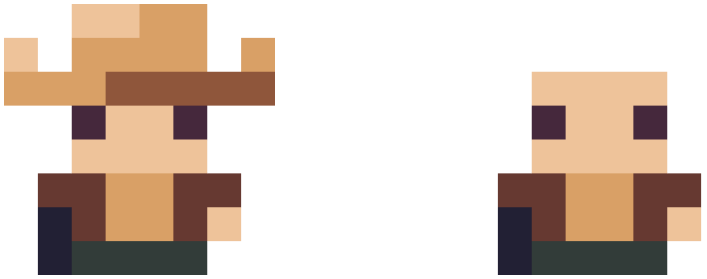
2.11 Algorithms

This section goes over some of the key algorithms I will be using in the project, looking at how they work, how they will be used, and the process of making them.

2.11.1 Converting Custom Characters to and from Strings

For the character customisation, as described in section 2.2 System Design and as seen in section 2.4 User Interface Design, players will be able to draw their own hat using a palette of colours to select from and a range of useful buttons. They will be able to customise the body of the character too by selecting a colour for each aspect of the body, for example the skin colour and the shirt colour.

In order to be able to add character customisation though, you need a character to customise. I had already created a simple 8x8 design for the player's character back in the prototyping of the analysis stage in which he wore a cowboy hat as can be seen on the left, and this is the design I worked from. In order to make the character usable, I just needed to remove the hat, as can be seen on the right.



This will be the default character design and it uses 6 colours. One for the skin colour, one for the eye colour, one for the jacket colour, one for the shirt colour, one for the trousers colour, and one for the gun colour. And so, these will be the 6 customisable options for the body of the player.

The area that the player can draw a hat in was decided to cover the top 3 rows of the character, so 24 pixels overall, just like the cowboy hat of the original design does. This hat will not be expected to spin around with the character as this would require either the program to somehow figure out a rotated version of the hat or the player to draw 4 hats, one for each direction. Neither of these are good solutions so the hat will remain constant as the player turns.

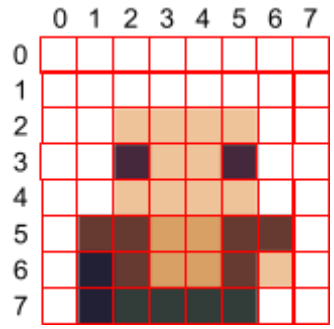
I now needed to figure out a way to write all the information from a custom character into some variable that can be easily inserted into the database and that I can work backwards from to display the character. The way I first thought to do this was to create a sort of bitmap image where the binary for every pixel is stored in a single string using a colour converter where each possible colour is mapped to a binary string. So, to create the binary bitmap string, I would simply loop through every pixel of the character, find the binary code of the current colour, and add it to a string. This successfully converts the custom character into a transferable format that can also be reverted into an image, however, it falls short when you try to turn this character around and let it face another direction like I need the character to do.

The next idea I had, and the idea I have used in the implementation, is to continue to use the bitmap idea for the hat but then store the selected colours for each body part instead of the body itself. Writing this to a string would now involve first writing the binary of the 6 colours selected for the body using the colour converter, followed by looping through the hat pixels and writing the binary code of each colour, again using the colour converter. This not only requires a significantly shorter binary string, but it allows for methods of turning the character around.

Now I needed to figure out how to turn this binary string into 4 character images, one for each direction. The way I can do this is by first creating specific constants that define the location of each pixel of each part of the body for each direction. This means that to create an image of a custom character looking in a certain direction I can loop through a list of coordinates for each part of the body and draw the appropriate colour at each coordinate.
For example, this is what the constants will look like for the character facing forwards:

```
SKIN_FRONT = [(2,2),(2,3),(2,4),(2,5),(3,3),(3,4),(4,2),(4,3),(4,4),(4,5),(6,6)]
JACKET_FRONT = [(5,1),(5,2),(5,5),(5,6),(6,2),(6,5)]
SHIRT_FRONT = [(5,3),(5,4),(6,3),(6,4)]
TROUSER_FRONT = [(7,2),(7,3),(7,4),(7,5)]
EYE_FRONT =  [(3,2),(3,5)]
GUN_FRONT = [(6,1),(7,1)]
```

Each part of the body has a list of coordinates that specify where its colour should be drawn in order to recreate the custom character.



Here is the front facing default character design with a grid and coordinate axes overlayed, separating the design into pixels.
This is how I created the lists of coordinates for each part of the body so as you can see, they line up with the character design.

After drawing the body of the custom character from the first 6 colours of the string, the hat can easily be drawn over the top from the following 24 colours.
I now have a description of an algorithm to turn custom characters into a string, which can easily be stored in the local database, and then a description of an algorithm to turn this string into the 4 character images required, one for each direction. This shouldn't be too hard to implement in Python and I plan to use a dictionary for the colour converter that stores each unique colour as the key and its binary code as the value. The string produced is 30 colours long, so, with an expected bit depth of 5 (due to an expected maximum of 32 colours), the binary string would be 150 characters in length.

Quite a bit later in development, I realised that there is no benefit to using binary for the custom characters rather than hexadecimal. Using binary adds a lot of unnecessary length to the strings and makes it very hard to read or understand for a human which can be fine, because a user will never be able to see their custom character in binary form, but makes it harder for me when creating the project. And so, I swapped from using binary to using hexadecimal. Using hexadecimal codes for each colour reduces the length of each custom character string from 150 characters long to 60 characters long (2 characters of hexadecimal are required to store the same data as 5 bits of binary). There is also equally not much of a reason to use hexadecimal rather than just using base 10 or a higher base such as using all alphanumeric characters, I just like that it keeps some of the authenticity of an actual computer system storing colours. Every method and algorithm I have described still completely works, only with the change of using hex codes not binary codes.

The Python implementation of converting a hex string into four images can be seen and explained in section 3.7 Player Class.

2.11.2 Character Customisation Functions

In the project, I will be using binary strings to hold all the information required to recreate a user's custom character, having each colour in a specified order. To be able to convert data to and from binary, I will need various functions to handle different things.

These functions were created before the change from binary to hexadecimal, however, they can still very much be used, only having to change some variable names and the dictionary creation function to use hexadecimal codes instead of binary codes. Most of these functions can be found in the implementation as either functions in the utility functions file or as small parts of larger methods in the customisation classes.

Function to create a dictionary with unique binary codes for each entry in a list

With a list as an input, return a dictionary where a new key has been created for each unique entry in the list with a unique binary code as the value and the bit depth of the binary codes. Using to create the binary dictionary of colours that is used in converting custom characters to and from binary.

Pseudocode:

```
FUNCTION CreateBinaryDictionary(List)
        ListLength ← Length(List)
        BitDepth ← RoundUp(LogBase2(ListLength))
        KeyValuePairs ← EMPTY LIST
        FOR index ← 0 TO ListLength
                KeyValuePairs ADD (List[index], ConvertToBinary(index))
        Dictionary ← ConvertToDictionary(KeyValuePairs)
        RETURN Dictionary, BitDepth
```

Python Implementation:

```python
def create_binary_dictionary(list):
    bit_depth = ceil(log(len(list), 2))                    #1
    key_value_list = []                                    #2
    for i in range(len(list)):                             #3
        key_value = (list[i], bin(i)[2:].zfill(bit_depth)) #4
        key_value_list.append(key_value)                   #5
    dictionary = dict(key_value_list)                      #6
    return dictionary, bit_depth
```

Description:

1 - calculates the bit depth required to be able to store all the entries in the list

2 - creates an empty list to store the key-value pairs

3 - loops from 0 to the length of the list

4 - creates a key-value pair with the first value being the list at the current index and the second being the current index converted into binary, using [2:] to remove the '0b' prefix that comes with it and .zfill(bit_depth) to ensure all binary codes have the same length

5 - adds this key-value pair to the list

6 - converts the list of key-value pairs into a dictionary

Testing:
- inputting [‘key1’, ‘key2’], the return values are {'key1': '0', 'key2': '1'} and 1
- inputting [1, 2, 3, 4, 5], the return values are {1: '000', 2: '001', 3: '010', 4: '011', 5: '100'} and 3
- inputting [(200,100,0), (120,255,180), (190,0,140), (200,100,0)] the return values are {(200, 100, 0): '11', (120, 255, 180): '01', (190, 0, 140): '10'} and 2

Function to split a string into groups of length n

With a string as an input, return a list containing the string split up into groups of length n. Used in the function to convert a string of binary back into data. This function also works for splitting lists.

Pseudocode:

```
FUNCTION Split (String, SizeOfParts)
	List ← EMPTY LIST
	FOR index ← 0 TO Length(String) STEP SizeOfParts
		List ADD String(index TO index+SizeOfParts)
	ENDFOR
	RETURN List
ENDFUNCTION
```

Python implementation:

```
def split(string, part_size):
    list = []                                   #1
    for i in range(0, len(string), part_size):  #2
        list.append(string[i : i+part_size])    #3
    return list
```

Description:
1 - creates an empty list to hold the split parts of the string
2 - loops from 0 to the length of the string incrementing by part_size each time to get a loop that loops through the indexes for the start of each part
3 - adds the portion of the string between the start of each part and the start of each part plus the part_size to add a part to the list

Testing:
- inputting “110010101011” and 3, the return value is [“110”, “010”, “101”, “011”]
- inputting “hello world” and 5, the return value is [“hello”, “ worl”, “d”]
- inputting “abcdefghij” and 1, the return value is [“a”, “b”, “c”, “d”, “e”, “f”, “g”, “h”, “i”, “j”]

Function to get a dictionary’s key from its value

With a value and a dictionary as an input, return the key associated with the value in the dictionary. Used in the function to convert a string of binary back into data.

Pseudocode:

```
FUNCTION GetKey (SearchValue, Dictionary)
       KeysList ← List(Dictionary.Keys)
       ValuesList ← List(Dictionary.Values)
       Position ← ValuesList.Index(SearchValue)
       RETURN KeysList[Position]
ENDFUNCTION
```

Python implementation:

```
def get_key(value, dictionary):
    key_list = list(dictionary.keys())          #1
    value_list = list(dictionary.values())      #2
    return key_list[value_list.index(value)]    #3
```

Description:

1 - gets the list of keys in the dictionary

2 - gets the list of values in the dictionary

3 - returns the key in the key list at the position where the value is found in the value list

Testing:

```
test_dictionary = {"key1" : 1, "key2" : 2.01, "key3" : "3", "key4" : (4,0,0)}
```

- inputting 1 and test_dictionary returns key1
- inputting 2.01 and test_dictionary returns key2
- inputting “3” and test_dictionary returns key3
- inputting (4,0,0) and test_dictionary returns key4
- inputting 5 and test_dictionary returns None

Function to convert a 2D array of data into binary string using a dictionary

With a 2D array of data and a dictionary of the possible data values with their relative binary codes as inputs, return a single string of binary that represents this 2D array read from left to right. Used in converting a grid of colour values into a storable binary string.

Pseudocode:

```
FUNCTION 2DArrayToBinary(2DArray, Dictionary)
       BinaryString ← EMPTY STRING
       FOR Row IN 2DArray
              FOR Value IN Row
                     BinaryString ← BinaryString + Dictionary[Value]
              ENDFOR
       ENDFOR
       RETURN BinaryString
ENDFUNCTION
```

Python implementation:

```
def array_to_binary(array, dictionary):
    binary = ""                                 #1
    for row in array:                           #2
        for data in row:                        #3
            binary += dictionary[data]          #4
    return binary
```

Description:

1 - creates an empty string for the string of binary

2, 3 - uses two for loops to loop through all the values of a 2D array in order of left to right

4 - adds the binary (values in the dictionary) of each data (keys in the dictionary) to the binary string

Testing:

`test_dictionary = {0:“00”, 1:“01”, 2:“10”, 3:“11”}`

- inputting the array `[[0, 3], [1, 2]]` and test_dictionary, the return value is “00110110”
- inputting the array `[[0], [1], [2], [3]]` and test_dictionary, the return value is “00011011”
- inputting the array `[[3, 3, 3, 3, 3, 3]]` and test_dictionary, the return value is “111111111111”

Function to convert a binary string into a 2D array of data using a dictionary

With a binary string of data, the number of rows and columns of the output array, the bit depth of the data and a dictionary of the possible data values with their relative binary codes as inputs, return a 2D array of the data that the binary string represents. I did play around with using additional bits at the beginning of the binary string as metadata containing the information about the rows, columns, and bit depth, however, in my game, all binary strings created will have the same number of rows and columns and the same bit depth, so it just adds unnecessary length to the string.

Pseudocode:

```
FUNCTION BinaryToArray (Binary, Dictionary, Rows, Columns, BitDepth)
        Array ← EMPTY LIST
        BinaryRows ← Split(Binary, Columns)
        BinaryArray ← Split(BinaryRows, BitDepth)
        FOR index1 ← 0 TO Rows
                FOR index2 ← 0 TO Columns
                        Array[index1][index2] ← GetKey(BinaryArray[index1][index2], Dictionary)
                ENDFOR
        ENDFOR
        RETURN Array
ENDFUNCTION
```

Python implementation:

```
def binary_to_array(binary, rows, columns, bit_depth, dictionary):
    array = [[None for i in range(columns)] for i in range(rows)]   #1
    binary_array = split(split(binary, bit_depth), columns)         #2
    for i in range(rows):                                           #3
        for j in range(columns):                                    #4
            array[i][j] = get_key(binary_array[i][j], dictionary)   #5
    return array
```

Description:

1 - creates an empty 2D array with the number of columns and rows specified in the arguments

2 - splits the binary into a 2D array of the same format as the empty 2D array using the predefined split function, first splitting the string into a list of the individual bit patterns representing each piece of data, and then splitting the list of bit patterns into lists of each row using the number of columns

3, 4 - uses two for loops to loop through all the values of the new binary array in order of left to right

5 - places the data represented by each individual bit pattern into the empty array, using the dictionary and the predefined get_key function, at the same place as it is in the binary array

Testing:

```
test_dictionary1 = {0:“00”, 1:“01”, 2:“10”, 3:“11”}
test_dictionary2 = {0:“000”, 1:“001”, 2:“010”, 3:“011”, 4:”100”, 5:”101”}
```

- inputting “00110110”, 2, 2, 2, and test_dictionary1, the return value is `[[0, 3], [1, 2]]`
- inputting “00011011”, 4, 1, 2, and test_dictionary1, the return value is `[[0], [1], [2], [3]]`
- inputting “001011101”, 1, 6, 3, and test_dictionary2, the return value is `[[1, 3, 5]]`

2.11.3 Enemy Movement

This section goes over the 3 enemy movement algorithms I use in the project, excluding the movement for the crow enemy as it will just move in one direction across the screen.

Flying Enemy Movement

This function was created in an early version of the game with only one enemy. It is a method from the enemy class that takes in the x and y coordinate of the player (or just where the enemy wants to go) and returns horizontal and vertical velocities the enemy should move with (what should be added to the enemy's position). It simply moves the enemy towards the player in the shortest way possible.

Pseudocode:

```
FUNCTION EnemyMovement(EnemyPosition, PlayerPosition, EnemySpeed)
        XVelocity ← 0
        YVelocity ← 0
        XDistance ← PlayerPosition[0] - EnemyPosition[0]
        YDistance ← PlayerPosition[1] - EnemyPosition[1]
        TotalDistance ← (XDistance^2 + YDistance^2)^0.5
        IF TotalDistance !=  0 THEN
                XVelocity = XVelocity + (EnemySpeed * (XDistance / TotalDistance))
                YVelocity = YVelocity + (EnemySpeed * (YDistance / TotalDistance))
        RETURN XVelocity, YVelocity
ENDFUNCTION
```

Python implementation:

```python
def movement(self, player_pos):
    velocity_x = 0
    velocity_y = 0
    x_distance = player_pos[X] - self.pos[X]
    y_distance = player_pos[Y] - self.pos[Y]
    distance = (x_distance**2 + y_distance**2)**0.5
    if distance != 0:
        velocity_x += self.speed * (x_distance / distance)
        velocity_y += self.speed * (y_distance / distance)
    return velocity_x, velocity_y
```

To do this, it first calculates the x and y distances between the enemy and the player. It then uses the Pythagorean theorem to find the total distance between the enemy and the player. If there is distance between them, the velocities of the enemy should be the portion of the respective distance component in the total distance, multiplied by the speed of the enemy. The total distance is always positive, so the sign of the x and y distances are kept.
This function later became the movement for the flying and spirit enemies.

Default Enemy Movement

I wanted the movement for the default enemy to be less floaty (like the previous algorithm is) and more rigid. I wanted it to move in straight lines and also have some randomness involved, as if the enemy wasn't very smart. To do this I made the enemy pick a direction based on the player's location with occasional random flair and then move in this direction for a random period of time before picking direction again. This requires making two functions, one that moves the enemy based on their current direction and one that is only run every few seconds that changes the enemy's direction.

Pseudocode for the function that changes the enemy's direction:

```
FUNCTION ChangeDirection(EnemyPosition, PlayerPosition, EnemySpeed)
        Directions ← ["Up", "Left", "Down", "Right"]
        XDistance ← PlayerPosition[0] - EnemyPosition[0]
        YDistance ← PlayerPosition[1] - EnemyPosition[1]
        IF |XDistance| > |YDistance| THEN
                IF XDistance < 0 THEN
                        DirectionIndex ← 1
                ELSE
                        DirectionIndex ← 3
        ELSE
                IF YDistance < 0 THEN
                        DirectionIndex ← 0
                ELSE
                        DirectionIndex ← 2

        RandomNum ← RANDOM INT BETWEEN -1 AND 3
        IF RandomNum <= 1 THEN
                DirectionIndex ← DirectionIndex + RandomNum
                DirectionIndex ← DirectionIndex MOD 4
        RETURN Directions[DirectionIndex]
ENDFUNCTION
```

This function would be run every few seconds (decided by a random timer) or run when the enemy hits an obstacle and the enemy then moves based on the direction returned from it, keeping moving in this direction in between function calls.

The Python implementation of the change direction function as a method of the enemy class:

```
def __change_direction(self, player_pos):
    x_distance = player_pos[X] - self._pos[X]
    y_distance = player_pos[Y] - self._pos[Y]
    if abs(x_distance) > abs(y_distance):
        if x_distance < 0:
            self.__direction = LEFT
        else:
            self.__direction = RIGHT
```

```
    else:
        if y_distance < 0:
            self.__direction = UP
        else:
            self.__direction = DOWN
    self.__direction_change_time = self._timer
    if 2*EIGHT_PIXELS < (x_distance**2 + y_distance**2)**0.5 < 8*EIGHT_PIXELS:
        self.__random_time = randint(FPS//4, FPS)
        random_int = randint(-1,3)
        if random_int <= 1:
            self.__direction = (self.__direction + random_int) % 4
```

In the Python implementation, I made it so the direction is only affected by randomness when the distance between the player and the enemy is between 16 and 64 pixels so that the enemy can't walk far into a corner and so that it doesn't suddenly change direction right next to the player. I also use constants for each direction, allowing me to directly change the direction of the enemy by adding or subtracting from the direction. Another change is that this function handles the random timer which decides the number of frames before the next direction change.
The constants are defined as follows:

```
UP = 0
LEFT = 1
DOWN = 2
RIGHT = 3
```

The move function that uses the direction generated from the change direction function to move the enemy:

```
def __move(self):
    if self.__direction == LEFT:
        velocity_x = -self._speed
        velocity_y = 0
    elif self.__direction == RIGHT:
        velocity_x = self._speed
        velocity_y = 0
    elif self.__direction == UP:
        velocity_x = 0
        velocity_y = -self._speed
    elif self.__direction == DOWN:
        velocity_x = 0
        velocity_y = self._speed
    return velocity_x, velocity_y
```

These functions are all run in the enemy's update function, where it is first checked whether the enemy should change direction and the change direction function is called if so, followed by the move function being called.

This movement system is also used for the tough enemy whose movement is very similar but with the values tweaked so the tough enemy moves slower than the default enemy and with significantly less randomness.

Fast Enemy Movement

I wanted the fast enemy to have a unique movement pattern that is predictable and isn't affected by randomness. The idea I came up with was a movement system where the enemy picks a direction and runs straight in that direction until it meets the player in the relative axis. For example, the player is far to the right of the enemy so the enemy starts running to the right. It continues running to the right until the player becomes vertically above or beneath the enemy, in which case the enemy changes direction and starts to move towards the player vertically. This is the movement algorithm I implemented for the fast enemy and it worked very well.

Pseudocode for the algorithm:

```
FUNCTION CheckDirection(EnemyPosition, PlayerPosition, CurrentDirection)
        XDistance ← PlayerPosition[0] - EnemyPosition[0]
        YDistance ← PlayerPosition[1] - EnemyPosition[1]
        IF CurrentDirection IS Up AND YDistance > 0 THEN
                NewDirection ← CHANGE DIRECTION TO HORIZONTAL
        ELSE IF CurrentDirection IS Down AND YDistance < 0 THEN
                NewDirection ← CHANGE DIRECTION TO HORIZONTAL
        ELSE IF CurrentDirection IS Left AND XDistance > 0 THEN
                NewDirection ← CHANGE DIRECTION TO VERTICAL
        ELSE IF CurrentDirection IS Right AND XDistance < 0 THEN
                NewDirection ← CHANGE DIRECTION TO VERTICAL
        RETURN NewDirection
ENDFUNCTION
```

What it does is check which direction the player is moving in and if they have reached / gone beyond the player in this direction. If so, the direction is changed to be in the other axis.

Python implementation:

```python
def __check_direction(self, player_pos):
    x_distance = player_pos[X] - self._pos[X]
    y_distance = player_pos[Y] - self._pos[Y]
    if self.__direction == None:                          #1
        if abs(x_distance) > abs(y_distance):
            self.__check_x(x_distance)
        else:
            self.__check_y(y_distance)
    elif self.__direction == UP and y_distance > 0:       #2
        self.__check_x(x_distance)
    elif self.__direction == DOWN and y_distance < 0:
        self.__check_x(x_distance)
```

```
    elif self.__direction == LEFT and x_distance > 0:
        self.__check_y(y_distance)
    elif self.__direction == RIGHT and x_distance < 0:
         self.__check_y(y_distance)

def __check_x(self, x_distance):                          #3
    if x_distance < 0:
        self.__direction = LEFT
    else:
        self.__direction = RIGHT

def __check_y(self, y_distance):                          #4
    if y_distance < 0:
        self.__direction = UP
    else:
        self.__direction = DOWN
```

1 - the first thing I do in this function is check if the enemy has any direction at all because they spawn with direction set to None, so the if statement catches that the enemy has no direction and allocates the enemy's initial direction accordingly
2 - then the function continues like the pseudocode, checking each direction and whether the enemy has reached the player in that direction
3 - this function picks the horizontal direction of the enemy based on its location compared to the player so if the player is to the left of the enemy, the direction of the enemy will now be left, replacing the CHANGE DIRECTION TO HORIZONTAL from the pseudocode
4 - the same as the function in 3 but for the vertical direction, replacing the CHANGE DIRECTION TO VERTICAL from the pseudocode

This movement worked exactly as I wanted it to and paired with the fast speed creates a difficult and threatening enemy.

2.11.4 Spawning Obstacles

A function will be needed to spawn an obstacle at a random point in the map after every few waves of enemies. The grid of the game is stored as a 2D array of Cell classes which have the attribute collision, that tells the game whether the cell should block the player and enemies from moving through, and the attribute image, that decides if and what the cell should display. The way I thought to spawn an obstacle in a random location was to separate the problem into two functions, one that checks if a cell is valid for placing an obstacle and one that randomly selects a valid cell and places the obstacle there, updating its collision and image attributes.

Pseudocode for checking for a valid placement:

```
FUNCTION CheckValidPlace(Grid, Row, Column)
        IF Grid[Row][Column] COLLISION == True THEN
                RETURN False
        IF Grid[Row][Column] COLLIDES WITH PLAYER THEN
                RETURN False
        RETURN True
ENDFUNCTION
```

Pseudocode for placing an object:

```
FUNCTION PlaceObstacle(Grid)
        Available ← EMPTY LIST
        FOR Row IN Grid Height
                FOR Column IN Grid Width
                        IF CheckValidPlace(Grid, Row, Column) THEN
                                Available ADD (Row, Column)

        IF Length(Available) > 0 THEN
                Position ← RANDOM CHOICE FROM Available
                SET Grid[Row][Column] IMAGE TO ObstacleImage
                SET Grid[Row][Column] COLLISION TO True
                RETURN Grid
ENDFUNCTION
```

Python implementations of the functions as methods of the Game class which has the grid attribute (a 2D array of Cells):

```
def __check_valid_placement(self, row, column, collision, center_spawn):
    if self.__grid[row][column].get_collision():
        return False

    if not center_spawn and (row in [trunc((GAME_WIDTH-1)/2), GAME_WIDTH//2]
                             and column in [trunc((GAME_HEIGHT-1)/2), GAME_HEIGHT//2]):
        return False

    if (self.__players == 1 and collision
            and self.__grid[row][column].get_rect().colliderect(self.__player.get_rect())):
        return False
    elif (self.__players == 2 and collision
          and (self.__grid[row][column].get_rect().colliderect(self.__player1.get_rect())
               or self.__grid[row][column].get_rect().colliderect(self.__player2.get_rect()))):
        return False

    return True

def __place_random(self, image, number, min_distance, collision=False, center_spawn=True):
    available = []
    for row in range(min_distance, self.__height - min_distance):
        for column in range(min_distance, self.__width - min_distance):
            if self.__check_valid_placement(row, column, collision, center_spawn):
                available.append((row, column))
    for _ in range(number):
        if len(available) > 0:
            row, column = choice(available)
            self.__grid[row][column].set_image(image)
            if collision:
                self.__grid[row][column].set_collision(True)
                self.__grid[row][column].set_shade(True)
            available.remove((row, column))

    if collision:
        self.__collidable_rects = self.__collidable_rects_list()
```

I did extend the functionality for the Python implementation quite significantly because doing so would allow me to place things other than just obstacles, such as flowers or grass for the background, and have more control over where things can spawn. For this I added the parameters number, min_distance, and center_spawn that specify how many you want to spawn, the minimum distance they need to be from the edges, and whether they can spawn in the central four cells respectively. This required me to change the functionality of the valid placement checking function as well.

2.11.5 Handling Collisions and Using Recursion

Collisions in games is what prevents players walking through walls, over obstacles, and out of the bounds of the game. For the collision systems in my game, I will frequently use the Pygame Rect class, short for rectangle. This is a class from the Pygame library that is initialised using an x position, y position, width and height. They have attributes for the positions of their sides (left, right, top, bottom) and very useful methods that compare points and other rects to the rect, such as collidepoint and colliderect that return True if a point is inside of or if a rect has collided with the rect respectively.

The first thing to do when creating collisions and obstacles is to stop the player moving out of the bounds of the game. My first idea for this was to keep a copy of the players previous position (their position in the previous frame) and then, once the player has moved in the current frame, check if their new position and rect is at all outside the rect of the game and if so, revert the player's position to their old position.

After implementing this I found that it worked fine and did indeed stop the player walking outside the bounds of the game, but it didn't look or feel right. This is because when using this method, the player doesn't actually hit the edge and is just reverted to their position before they touched it; they only get so close to it that their movement in a frame would be greater than the distance between them and the edge. Therefore, it was rare to get the player to actually visually hit the edge or feel like the edge should be completely stopping their movement.

The way to fix this, instead of just reverting the player to their previous position, is to move the player less. So, when the player is trying to move out of the bounds of the game, the player should still move a little bit but only to the point of touching the edge, and then no further.

When implementing this new version I found that it still wasn't perfect due to the fact I was using rects and the sides of rects for all collisions which approximate positions to integer values. This means that when correcting the position of the player I would be moving the position of the player (which is not approximated to integers) by integers when the player was actually, for example, 0.6 pixels from the edge of the game. To fix this I create a dictionary before checking for collisions that stores the sides of the player before moving and their non-integer positions and then use these values when correcting the player's position.

The dictionary of sides:

```
player_sides = {'TOP'    : self.__pos[Y] - EIGHT_PIXELS/2,
                'BOTTOM' : self.__pos[Y] + EIGHT_PIXELS/2,
                'LEFT'   : self.__pos[X] - EIGHT_PIXELS/2,
                'RIGHT'  : self.__pos[X] + EIGHT_PIXELS/2}
```

The method of the player class for correcting collisions with the edges of the game, taking in parameters of the sides dictionary, the current horizontal velocity of the player, and the current vertical velocity of the player:

```
def __edge_collisions(self, player_sides, velocity_x, velocity_y):
        if player_sides['LEFT'] - self.__game_rect.left < self.__speed and velocity_x < 0:
            velocity_x = self.__game_rect.left - player_sides['LEFT']
        elif self.__game_rect.right - player_sides['RIGHT'] < self.__speed and velocity_x > 0:
            velocity_x = self.__game_rect.right - player_sides['RIGHT']
```

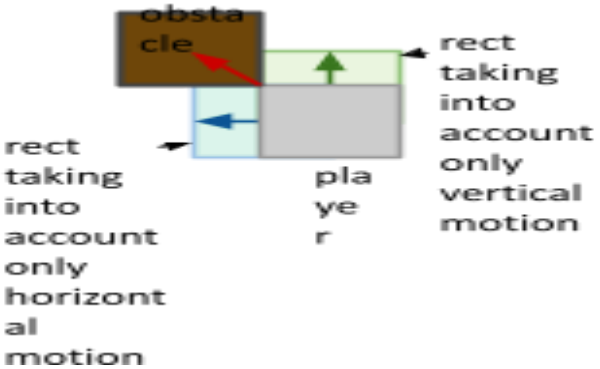
```
        if player_sides['TOP'] - self.__game_rect.top < self.__speed and velocity_y < 0:
            velocity_y = self.__game_rect.top - player_sides['TOP']
        elif self.__game_rect.bottom - player_sides['BOTTOM'] < self.__speed and velocity_y > 0:
            velocity_y = self.__game_rect.bottom - player_sides['BOTTOM']

        return velocity_x, velocity_y
```

So for each edge of the game, if the distance of the player's relative side to the edge is less than the amount they are going to move and the player is moving in the direction towards the edge, their velocity in the relative direction becomes the distance between the player's relative side and the edge, and so they move perfectly to the edge of the game.

In my implementation the player has a game_rect attribute that stores the rect of the game the player is playing in. This is because I use the game's rect in multiple places across the Player class and thought it was easier to make an attribute for it rather than passing it into all the functions it is required for as an argument.

I then had to extend this collision system into working with obstacles. To do this I separated the movement into its horizontal and vertical components for collision checking. My implementation involved creating two rects that represent the new rect of the player when only taking into account their horizontal motion and the new rect of the player when only taking into account their vertical motion. This allowed me to individually check for collisions with every obstacle in the horizontal and vertical directions and correct for the directions separately.

After implementing this, I found a constant problem of the player being able to walk straight through any obstacle if they walked diagonally into its corner. After a very long time of not understanding what was going on, I realised that my method of checking by separating the movement into its horizontal and vertical components does not work for every situation. This is because, when moving perfectly into the corner of an obstacle, if you look for a collision taking into account only the horizontal motion, you will not find a collision, and if you look for a collision taking into account only the vertical motion, you will not find a collision. It's only when you look at the overall combined motion (a red arrow in the diagram below) that you find the corner collision, and therefore the player could move straight through obstacles if they approach on a perfect diagonal.



Fixing this wasn't too difficult, all it required was to add another rect that takes into account both the horizontal and vertical motion (so the new rect of the player) and then check if there is a collision in this rect and not in either of the horizontal or vertical rects.

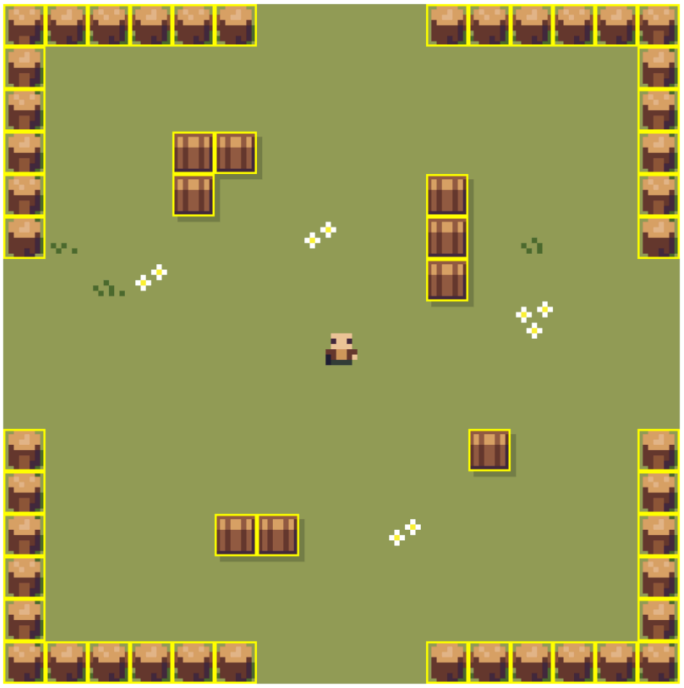
The method of the player class for checking collisions with obstacles is displayed below:

```
def __collidables_collisions(self, player_sides, velocity_x, velocity_y, collidables):
        self.__new_rect_x.center = self.__pos + pygame.math.Vector2(velocity_x, 0)
        self.__new_rect_y.center = self.__pos + pygame.math.Vector2(0, velocity_y)
        self.__new_rect.center = self.__pos + pygame.math.Vector2(velocity_x, velocity_y)
        x = velocity_x
        y = velocity_y

        for collidable in collidables:
            collision_x = collidable.colliderect(self.__new_rect_x)
            collision_y = collidable.colliderect(self.__new_rect_y)
            diagonal = (collidable.colliderect(self.__new_rect)
                        and not (collision_x or collision_y))

            if collision_x or diagonal:
                if x < 0:
                    velocity_x = -(player_sides['LEFT'] - collidable.right)
                elif x > 0:
                    velocity_x = collidable.left - player_sides['RIGHT']

            if collision_y or diagonal:
                if y < 0:
                    velocity_y = -(player_sides['TOP'] - collidable.bottom)
                elif y > 0:
                    velocity_y = collidable.top - player_sides['BOTTOM']

        return velocity_x, velocity_y
```

In my implementation, the player class has the new rects as attributes and just updates their position each time collisions are checked for. I make copies of the x and y velocities so that they don't change between obstacles.

I then had a new problem. This was that when the player was moving diagonally along a line of obstacles, they would stop at every corner of a new obstacle. Ideally, you want the player to keep moving upwards if there is a wall of obstacles to their left and they are moving upwards and left. The reason for the stopping is that the program looks at obstacles separately and so is registering diagonal collisions for the corner of each obstacle, causing motion both horizontally and vertically to be stopped.

Below is a screenshot of what the rects looked like (I added a few lines of code to the game's drawing function to draw each obstacle's rect):



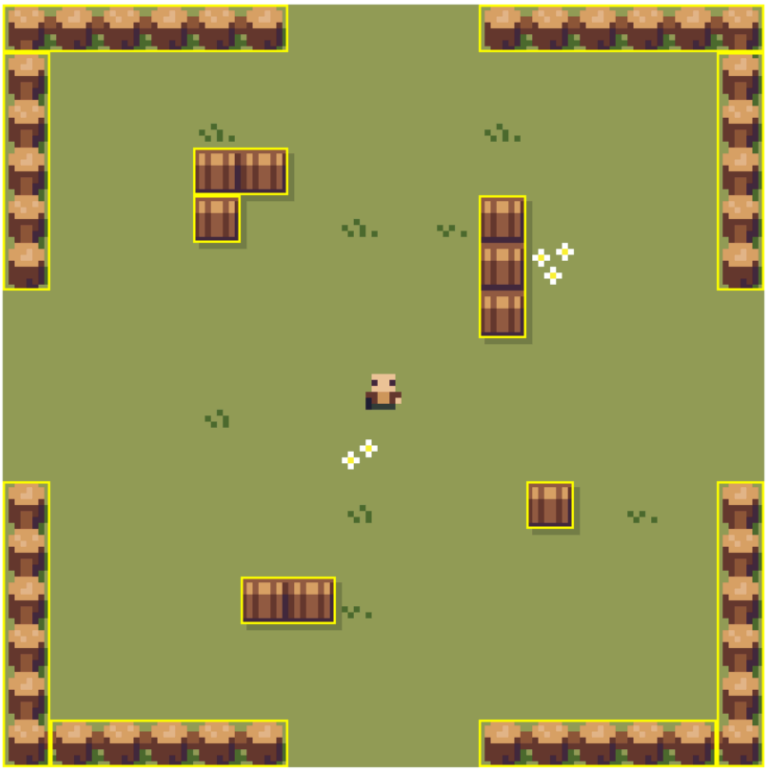
The method I wanted to solve this with was to create an algorithm that combines the rects of adjacent obstacles into larger rects that don't have corners.
I did this using recursion. So for each obstacle, I would look to the right to see if another obstacle is present, recursively checking the right over and over again until the end of a row or a non-collidable cell is found. The algorithm would then unwind and increase the width of the rect of the original obstacle for every collidable obstacle found. If searching horizontally was unsuccessful, then a very similar search vertically would be performed, increasing the height of the rect of the original obstacle for every collidable obstacle found.

The code and result of this algorithm is below:

```
def __collidable_rects_list(self):
        collidable_rects = []
        row = 0
        column = 0
        checked = []
        for row in range(self.__height):
            for column in range(self.__width):
                if (row, column) not in checked and self.__grid[row][column].get_collision():
                    rect = self.__grid[row][column].get_rect().copy()

                    checked_columns = []
                    rect.width, checked_columns = self.__check_next_hor(row, column,
                                                                        checked_columns,
                                                                        checked,
                                                                        EIGHT_PIXELS)
```

```
                for checked_column in checked_columns:
                    checked.append((row, checked_column))

                if len(checked_columns) == 1:
                    checked_rows = []
                    rect.height, checked_rows = self.__check_next_vert(row, column,
                                                                       checked_rows,
                                                                       checked,
                                                                       EIGHT_PIXELS)
                    for checked_row in checked_rows:
                        checked.append((checked_row, column))

                collidable_rects.append(rect)
    return collidable_rects

def __check_next_hor(self, row, column, checked_columns, checked, width):
    try:
        checked_columns.append(column)
        if (row, column+1) not in checked:
            if self.__grid[row][column + 1].get_collision():
                width, checked_columns = self.__check_next_hor(row, column + 1,
                                                               checked_columns, checked,
                                                               width)
                return width + EIGHT_PIXELS, checked_columns
    except IndexError:
        pass
    return width, checked_columns

def __check_next_vert(self, row, column, checked_rows, checked, height):
    try:
        checked_rows.append(row)
        if (row + 1, column) not in checked:
            if self.__grid[row + 1][column].get_collision():
                height, checked_rows = self.__check_next_vert(row + 1, column, checked_rows,
                                                              checked, height)
                return height + EIGHT_PIXELS, checked_rows
    except IndexError:
        pass
    return height, checked_rows
```

Below is a screenshot of what the rects now look like after implementing the recursive function (using the same changes from the previous screenshot to draw each rect):



As you can see, obstacles are now combined to have larger rects and the corner issue has been mostly fixed. The only issue is that rects can only be rectangles so some obstacle groups will still have corner collisions between different rows, but the largest part of the problem was definitely fixed and it is quite rare to have more than 2 adjacent crates spawn anyway.

Enemy collisions are a lot less important than player collisions as the player does not control the enemies so they can't feel any imperfections. This allowed me to just use the previous position method where if a collision is detected, the enemy is reverted to their previous position. Collisions between the player and enemies are not necessary because everytime a collision is detected between the player and an enemy, the game pauses and resets and the player loses a life.

3. Technical Solution

In this section I will explain various sections of important code that are key to the project. The rest of the code can be found in the source code in section 6.

3.1 Utility Functions

This is a file that contains lots of different useful functions that are used in many places around the project.

The imports from libraries:

```
from math import ceil, log
from pygame import Rect, mask
```

The create_hex_dictionary function that takes a list as a parameter to create a dictionary of the items in the list as keys and an incrementing hex string as values:

```
def create_hex_dictionary(list):
    depth = ceil(log(len(list), 16))
    key_value_list = []
    for i in range(len(list)):
        key_value = (list[i], hex(i)[2:].zfill(depth))
        key_value_list.append(key_value)
    dictionary = dict(key_value_list)
    return dictionary, depth
```

To do this, it first works out the depth of hex needed (how many hex digits), and then loops through each item in the list, converting their index into hex and adding the item and hex to a list of pairs which get turned into a dictionary at the end. The [2:] after creating the hex value is to remove the '0x' prefix that comes with values when you turn them into hexadecimal, and the .zfill(depth) is to add as many 0s as required to the left of the hex value to match the required depth so that each hex has the same length.

The check_index function that returns True if an index exists in a list and False if not:

```
def check_index(list, index):
    try:
        list[index]
        return True
    except IndexError:
        return False
```

To do this, I use a simple try except statement. If an error occurs when trying to index the list at that index, the index is not in the list.

The split function that takes a string or a list and splits it into groups of size n:

```
def split(string, part_size):
    list = []
    for i in range(0, len(string), part_size):
        list.append(string[i : i+part_size])
    return list
```

The get_key function that returns the key of a value in a dictionary:

```
def get_key(value, dictionary):
    key_list = list(dictionary.keys())
    value_list = list(dictionary.values())
    return key_list[value_list.index(value)]
```

The clip function that takes an image, a position, and a size, and returns a copy of the image in only the area specified by these:

```
def clip(image, x, y, width, height):
    image_copy = image.copy()
    clip_rect = Rect(x, y, width, height)
    image_copy.set_clip(clip_rect)
    clipped_image = image.subsurface(image_copy.get_clip())
    return clipped_image.copy()
```

It does this by copying the image, creating an appropriate rect from the parameters, and then using pygame surface functions to create a clip and put it into a new image.

The colour_swap function that takes an image and swaps a colour in it:

```
def colour_swap(image, old_colour, new_colour):
    colour_mask = mask.from_threshold(image, old_colour, threshold=(1, 1, 1, 255))
    colour_change_surface = colour_mask.to_surface(setcolor=new_colour,
                                                   unsetcolor=(0, 0, 0, 0))
    image_copy = image.copy()
    image_copy.blit(colour_change_surface, (0, 0))
    return image_copy
```

I do this by creating a colour mask of the image that is basically a blueprint for an image after extracting only the old_colour in it. I then convert this mask into a surface (image), setting the colour as new_colour, so it is essentially a map of only the pixels in the old_colour now in the new_colour. I then copy the original image and blit the new surface over it.

The round_to_nearest function that takes a number and rounds it to the nearest n using some simple maths:

```
def round_to_nearest(x, nearest):
    return nearest * round(x/nearest)
```

The position_list function that takes a range and returns a list of numbers within that range with their positional suffix:

```
def position_list(first, last):
1   if first <= last:
        positions = [str(i) for i in range(first, last+1)]
    else:
1       positions = [str(i) for i in range(first, last-1, -1)]

2   for i in range(len(positions)):
        match positions[i][-1]:
            case '1':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'st'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'st'
            case '2':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'nd'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'nd'
            case '3':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'rd'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'rd'
            case _:
                positions[i] += 'th'

2   return positions
```

1 - The first thing I do is create the range of numbers in a list

2 - I then use many if statements in a match statement to decide which suffix each number uses. I use the match statement because I think it makes it more clear.

3.2 Database Functions

This is a file that contains lots of different functions and procedures for querying and inserting into the database to make coding the rest of the game easier and so that if I need to change anything about the database I can change the functions in this file to accommodate for that.

I first import the default hexadecimal custom character so that it can be inserted into the database when a new player is created:

```
from constants import DEFAULT_HEX
```

The db_create_database procedure that creates all the tables for the database, first checking whether they already exist:

```
def db_create_database(db_cursor, db_connection):
    create_players_table = """CREATE TABLE IF NOT EXISTS Players(
                                        username CHAR(4) NOT NULL PRIMARY KEY,
                                        pin CHAR(4) NOT NULL,
                                        custom_character CHAR(60) NOT NULL,
                                        games_played INTEGER NOT NULL,
                                        enemies_killed INTEGER NOT NULL,
                                        bullets_shot INTEGER NOT NULL,
                                        items_used INTEGER NOT NULL
                                        )"""

    create_single_player_table = """CREATE TABLE IF NOT EXISTS SinglePlayerGames(
                                      game_id INTEGER PRIMARY KEY,
                                      score INTEGER NOT NULL,
                                      username CHAR(4) NOT NULL,
                                      FOREIGN KEY (username) REFERENCES Players(username)
                                      )"""

    create_two_player_table = """CREATE TABLE IF NOT EXISTS TwoPlayerGames(
                                    game_id INTEGER PRIMARY KEY,
                                    score INTEGER NOT NULL,
                                    player1_name CHAR(4) NOT NULL,
                                    player2_name CHAR(4) NOT NULL,
                                    FOREIGN KEY (player1_name) REFERENCES Players(username),
                                    FOREIGN KEY (player2_name) REFERENCES Players(username)
                                    )"""

    db_cursor.execute(create_players_table)
    db_cursor.execute(create_single_player_table)
    db_cursor.execute(create_two_player_table)
    db_connection.commit()
```

I first create all the queries to create the tables, which explains themselves pretty well, as multiline strings to improve readability. I then execute these queries on the database and commit the changes.

The get_1p_highscore function that returns the 1 player highscore of a player:

```
def db_get_1p_highscore(username, db_cursor):
    highscore = db_cursor.execute("""SELECT score
                                     FROM SinglePlayerGames
                                     WHERE username = ?
                                     ORDER BY score DESC""",
                                  (username, )).fetchone()
    if highscore:
        return highscore[0]
    return 0
```

I first query the database to find the highest score of a player by ordering their scores in descending order and only fetching one result. If something was returned from the query, it is the user's highscore and I can return it, indexing it at 0 to take it out of the tuple it is given in. If nothing was returned, the user has not played a game yet and therefore their high score is set to 0.

The get_2p_highscore function that returns the 2 player highscore of a player:

```
def db_get_2p_highscore(username, db_cursor):
    highscore = db_cursor.execute("""SELECT score
                                     FROM TwoPlayerGames
                                     WHERE player1_name = ?
                                     OR player2_name = ?
                                     ORDER BY score DESC""",
                                  (username, username)).fetchone()
    if highscore:
        return highscore[0]
    return 0
```

This is very similar in functionality to the get_1p_highscore function, just querying for the player's username as either the player1_name or the player2_name.

The get_character function that returns the custom character hex string of a player:

```
def db_get_character(username, db_cursor):
    return db_cursor.execute("""SELECT custom_character
                                FROM Players
                                WHERE username = ?""",
                             (username, )).fetchone()[0]
```

The get_stats function that returns a tuple of all the stats of a player:

```
def db_get_stats(username, db_cursor):
    return db_cursor.execute("""SELECT games_played, enemies_killed, bullets_shot, items_used
                                FROM Players
                                WHERE username = ?""",
                             (username,)).fetchone()
```

The get_all_usernames function that returns a list of every username in the database:

```
def db_get_all_usernames(db_cursor):
    usernames = []
    rows = db_cursor.execute("""SELECT username
                                FROM Players""").fetchall()
    for tuple in rows:
        usernames.append(tuple[0])
    return usernames
```

To do this it queries the database to get every username in a list of tuples, then takes each of them out of their tuples and adds them to a new list.

The find_player function that takes a username and pin and returns the username of the linked account and None if no account is found:

```
def db_find_player(username, pin, db_cursor):
    return db_cursor.execute("""SELECT username
                                FROM Players
                                WHERE username = ? and pin = ?""",
                             (username, pin)).fetchone()
```

The add_to_stats procedure takes in statistics to add as parameters and then updates the database to add these a player's statistics:

```
def db_add_to_stats(username, db_cursor, db_connection, games_played=0, enemies_killed=0,
                    bullets_shot=0, items_used=0):
    curr_gp, curr_ek, curr_bs, curr_iu = db_get_stats(username, db_cursor)
    db_cursor.execute("""UPDATE Players
                         SET games_played = ?, enemies_killed = ?,
                                                bullets_shot = ?, items_used = ?
                         WHERE username = ?""",
                      (curr_gp + games_played, curr_ek + enemies_killed,
                       curr_bs + bullets_shot, curr_iu + items_used, username))
    db_connection.commit()
```

It first uses the get_stats function to get all the current statistics of the player and then updates the database with the new statistics adding to the current statistics and finally commits the changes.

The set_character procedure that updates a player's custom character:

```
def db_set_character(username, character_hex, db_cursor, db_connection):
    db_cursor.execute("""UPDATE Players
                         SET custom_character = ?
                         WHERE username = ?""",
                      (character_hex, username))
    db_connection.commit()
```

The insert_player procedure that inserts a new player into the database with initial values for the custom character and statistics:

```
def db_insert_player(username, pin, db_cursor, db_connection):
    db_cursor.execute("""INSERT INTO Players
                         VALUES(?,?,?,?,?,?,?)""",
                      (username, pin, DEFAULT_HEX, 0, 0, 0, 0))
    db_connection.commit()
```

The insert_1p_score procedure that inserts a new 1 player score into the database:

```
def db_insert_1p_score(username, score, db_cursor, db_connection):
    db_cursor.execute("""INSERT INTO SinglePlayerGames(score, username)
                         VALUES(?, ?)""",
                      (score, username))
    db_connection.commit()
```

The insert_2p_score procedure that inserts a new 2 player score into the database:

```
def db_insert_2p_score(username1, username2, score, db_cursor, db_connection):
    db_cursor.execute("""INSERT INTO TwoPlayerGames(score, player1_name, player2_name)
                         VALUES(?, ?, ?)""",
                      (score, username1, username2))
    db_connection.commit()
```

3.3 Stack Class

This is the basic stack class utility_classes.py file that I use for my undo and redo buttons.

The __init__ method:

```
class Stack():
    def __init__(self):
        self.__max_size = 100
        self.__stack = []
```

The empty method that returns True if the stack is empty and false otherwise:

```
    def empty(self):
        if self.__stack:
            return False
        else:
            return True
```

The peek method that returns the last item in the stack (without removing it):

```
    def peek(self):
        return self.__stack[self.size()-1]
```

The size method that returns the length of the stack:

```
    def size(self):
        return len(self.__stack)
```

The push method that pushes an item onto end of stack, removing first item if necessary:

```
    def push(self, item):
        if self.size() == self.__max_size:
            self.__stack = self.__stack[1:]
        self.__stack.append(item)
```

The pop method that removes the last item from the stack and returns it:

```
    def pop(self):
        return self.__stack.pop()
```

The reset method that empties the stack:

```
    def reset(self):
        self.__stack = []
```

3.4 Queue class

The queue class from the utility_classes.py file that is used for holding waves of enemies for the game until they are spawned:
The __init__ method:

```
class Queue():
    def __init__(self):
        self.__queue = []
```

The empty method that returns True if the queue is empty and False otherwise:

```
    def empty(self):
        if self.__queue:
            return False
        else:
            return True
```

The peek method that returns the first item in the queue (without removing it):

```
    def peek(self):
        return self.__queue[0]
```

The size method that returns the length of the queue:

```
    def size(self):
        return len(self.__queue)
```

The enqueue method that pushes an item onto the end of the queue:

```
    def enqueue(self, item):
        self.__queue.append(item)
```

The dequeue method that removes the first item from the queue and returns it:

```
    def dequeue(self):
        return self.__queue.pop(0)
```

The reset method that empties the queue:

```
    def reset(self):
        self.__queue = []
```

3.5 Font Class

This is the font class from the utility_classes.py file that converts an image into a usable font.
An example of a font image, created in Aseprite, is as follows (displayed on a grey background):

with white characters and red lines in between them.
In order to get an image like this into a usable font, you need to separate the characters into individual images, hence the red lines. What the font class does is to turn an image like this into a dictionary of characters as keys and their separated images as values and then when a user wishes to display some text, it will display each character's image to the screen.

The __init__ method that takes in the font image, the list of characters that the image covers, the desired colour of the font, the width of spaces, and the optional parameters for the spacing between characters and the alpha value (transparency) of the font:

```
class Font():
    def __init__(self, font_image, character_list, colour, space_width,
                 character_spacing=PIXEL_RATIO, alpha=255):
        self.__font_image = font_image
        self.__character_list = character_list
        self.__characters = {}
        self.__space_width = space_width
        self.__character_spacing = character_spacing
        self.__height = font_image.get_height()

1       border_colour = (255,0,0)
        text_colour = (255,255,255)

        if colour != border_colour:
            font_image = colour_swap(font_image, text_colour, colour)
        else:
```

```
            different_colour = (127,127,127)
            font_image = colour_swap(font_image, border_colour, different_colour)
            border_colour = different_colour
            font_image = colour_swap(font_image, text_colour, colour)

        if alpha != 255:
1           font_image.set_alpha(alpha)

2       current_width = 0
        character_count = 0
        for x in range(0, font_image.get_width(), int(PIXEL_RATIO)):
            colour = font_image.get_at((x, 0))
            if colour == border_colour:
                character_image = clip(font_image, x - current_width, 0,
                                       current_width, font_image.get_height())
                self.__characters[self.__character_list[character_count]] = character_image
                character_count += 1
                current_width = 0
            else:
2               current_width += PIXEL_RATIO
```

1 - In the first section, the font image is adjusted to fit the parameters passed in. This includes changing the colour of the image if necessary. Because the colours here are properties of the image they are not made as constants in the constants file. If the desired colour of the font is the same as the border colour, the border colour needs to be set as a random other colour (I chose a middle grey), and then the border colour can be swapped followed by the text colour using the colour_swap function I created in the utility functions file. After this, if the alpha parameter is passed in, the image's alpha value is set to this to make it as transparent as desired.

2 - In the second section, the font image is searched through and characters are clipped into individual images and assigned letters in the dictionary. I do this by looping through the width of the font image, incrementing by a pixel each time, and getting the colour at each point. If the colour is equal to the border colour (so a red line has been reached), the image is clipped from this border to the previous border to separate the character using the clip function I created in the utility functions file. An entry is then created in the dictionary of characters for the character and its image.

The new_colour_copy method that takes in a new colour and an optional new alpha value and returns a new initialisation of the font with the different colour and alpha value:

```
    def new_colour_copy(self, new_colour, alpha=255):
        return Font(self.__font_image, self.__character_list, new_colour, self.__space_width,
                    character_spacing=self.__character_spacing, alpha=alpha)
```

The get_text_width method that takes in some text and returns what the width of it would be in this font:

```
    def get_text_width(self, text):
        width = [0]
        line = 0
        width[0] += self.__characters[text[0]].get_width()
        for character in text[1:]:
            if character == NEW_LINE:
                width.append(0)
                line += 1
            elif character != ' ':
                width[line] += self.__character_spacing
                width[line] += self.__characters[character].get_width()
            else:
                width[line] += self.__space_width
        return max(width)
```

It does this by creating an empty list that stores the width of every line in the text (only the first index is used if there is only one line in the text). It then loops through each character in the text and adds the width of it to the appropriate line in the width list, adding a new entry to the list if a new line was created. It then returns the maximum value in the list.

Get methods:

```
    def get_height(self):
        return self.__height

    def get_character_spacing(self):
        return self.__character_spacing
```

The render method that renders some text to the screen at a position, with the optional parameters to change the alignment of the text and whether the text should be hidden (written in dots):

```
    def render(self, screen, text, pos, alignment=LEFT, hide=False):
1       x_offset = [0]
        line = 0
        y_offset = 0
        for character in text:
            if hide:
                character = 'hidden'
            if character == NEW_LINE:
                y_offset += self.__height + 2*PIXEL_RATIO
                x_offset.append(0)
                line += 1
            elif character != ' ':
                if alignment == LEFT:
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y] + y_offset))
```

```
                x_offset[line] += self.__characters[character].get_width()
                x_offset[line] += self.__character_spacing
            else:
                x_offset[line] += self.__space_width

1       widths = x_offset.copy()

2       line = 0

        if alignment == RIGHT:
            x_offset = []

            for width in widths:
                x_offset.append(-width)

            y_offset = 0
            for character in text:
                if hide:
                    character = 'hidden'
                if character == NEW_LINE:
                    y_offset = self.__height + 2*PIXEL_RATIO
                    line += 1
                if character != ' ':
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y] + y_offset))
                    x_offset[line] += self.__characters[character].get_width()
                    x_offset[line] += self.__character_spacing
                else:
2                   x_offset[line] += self.__space_width
3       elif alignment == CENTER:
            x_offset = []

            for width in widths:
                x_offset.append(0 - round_to_nearest(width//2, PIXEL_RATIO))

            y_offset = 0
            for character in text:
                if hide:
                    character = 'hidden'
                if character == NEW_LINE:
                    y_offset = self.__height + 2*PIXEL_RATIO
                    line += 1
                elif character != ' ':
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y] + y_offset))
                    x_offset[line] += self.__characters[character].get_width()
                    x_offset[line] += self.__character_spacing
                else:
3                   x_offset[line] += self.__space_width
```

1 - In the first section, much like the get_text_width method, every character in the list is looped through and their width is added to the x_offset variable, but in this method they are actually blitted to the screen (only if the alignment is to the left at this point though). Once this loop is complete, the x_offset list now represents the width of each line in the text, which can be used to align the text to either the centre or the right, hence only blitting the characters if the alignment is to the left.
2 - Now, to blit text to the screen using an alignment to the right, a new x_offset list is created with each value initially set to the negative width of its corresponding line, so that characters start blitting from the furthest left point up to the passed in position.
3 - Now, to blit text to the screen using a centre alignment, a new x_offset list is created with each value initially set to half the negative width of its corresponding line to the nearest pixel, so that characters start blitting from halfway to the left up to halfway to the right.

3.6 Drawing Grid Class and Using Stacks

This is the drawing grid class from the customisation_classes.py file that allows the user to select colours to draw onto a grid with and use lots of useful buttons such as undo and redo.
The __init__ method that initialises every object the drawing grid uses and sets initial states:

```
class DrawingGrid():
    def __init__(self, pos, columns, rows, colours, initial_hex=None):
        self.__rows = rows
        self.__columns = columns

1       self.__grid = [[Pixel((pos[X]+(column*EIGHT_PIXELS), pos[Y]+(row*EIGHT_PIXELS)),
                              background_colour=WHITE)
                                for column in range(self.__columns)]
                                    for row in range(self.__rows)]
        self.__rect = pygame.Rect(pos, (self.__columns*EIGHT_PIXELS, self.__rows*EIGHT_PIXELS))
        self.__border_rect = pygame.Rect((self.__rect.x - 2*PIXEL_RATIO,
                                          self.__rect.y - 2*PIXEL_RATIO),
                                         (self.__rect.width + 4*PIXEL_RATIO,
                                          self.__rect.height + 4*PIXEL_RATIO))

2       self.__undo_stack = Stack()
        self.__redo_stack = Stack()

        self.__undo_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                          pos[Y] + EIGHT_PIXELS),
                                         (EIGHT_PIXELS, EIGHT_PIXELS),
                                         undo_image)
        self.__redo_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                          pos[Y] + 2*EIGHT_PIXELS),
                                         (EIGHT_PIXELS, EIGHT_PIXELS),
                                         redo_image)
        self.__clear_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                           pos[Y]),
                                          (EIGHT_PIXELS, EIGHT_PIXELS),
                                          clear_image)
       self.__erase_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
```

```
                                        pos[Y] - 2*EIGHT_PIXELS),
                                       (EIGHT_PIXELS, EIGHT_PIXELS),
                                       eraser_image, hold=False,
                                       pressed_image=eraser_image_pressed)

2       self.__colour_grid = ColourGrid((pos[X], pos[Y] - 2*EIGHT_PIXELS),
                                        colours, self.__columns)

        self.__previous_state = None
        self.__changed = False
        self.__drawn = False

3       for i in range(self.__rows):
            row = self.__grid[i]
            for j in range(i % 2, self.__columns, 2):
               row[j].set_background_colour(CUSTOMISATION_GREY)


        if initial_hex:
3           self.hex_to_grid(initial_hex)
```

1 - The first object initialised is the grid. This is a 2D array of pixels, created using for loops.
2 - The rest of the classes are then initialised, including all the buttons, the colour grid for selecting a colour, and the undo and redo stacks.
3 - Then the pixels are looped through and every other pixel's background colour is set to be a light grey to create a transparent effect, common to drawing applications, and finally, if an initial hex string was passed in, the current state of the drawing grid is set to it by calling the hex_to_grid method.

The grid_to_hex method that creates a single line hex string of the current state of the grid:

```
    def grid_to_hex(self):
        hex_string = ""
        for row in self.__grid:
            for pixel in row:
                hex_string += BITMAP_DICTIONARY[pixel.get_colour()]
        return hex_string
```

To do this I simply loop through every pixel and concatenate the bitmap dictionary's hex code for the pixel's colour to the string.

The hex_to_grid method that takes in a hex string and converts the drawing grid into the state specified by it:

```
    def hex_to_grid(self, hex_string):
        hex_array = split(split(hex_string, COLOUR_DEPTH), self.__columns)
        for row in range(self.__rows):
            for column in range(self.__columns):
```

```
                self.__grid[row][column].set_colour(get_key(hex_array[row][column],
                                                            BITMAP_DICTIONARY))
```

To do this it first splits the hex string into individual colours and then into rows in the same format as the grid. This list is then looped through and each pixel in the grid's colour is set to the colour represented by its corresponding hex code.

The update function that updates all the different aspects of the drawing grid, registering and addressing button presses by taking in mpos, clicking, click, and unclick as parameters:

```
    def update(self, mpos, clicking, click, unclick):
1       self.__changed = False

2       self.__colour_grid.update(mpos, click)
        self.__undo_button.update(mpos, click)
        self.__redo_button.update(mpos, click)
        self.__erase_button.update(mpos, click)
2       self.__clear_button.update(mpos, click)

3       if click:
            self.__previous_state = self.grid_to_hex()

4       if self.__undo_button.get_clicked():
            self.__undo()
            self.__changed = True

5       if self.__redo_button.get_clicked():
            self.__redo()
            self.__changed = True

6       if self.__erase_button.get_clicked():
            self.__colour_grid.deselect()

7       if self.__erase_button.get_pressed() and self.__colour_grid.get_selected():
            self.__erase_button.unpress()

8       if self.__clear_button.get_clicked():
            self.__clear()
            self.__undo_stack.push(self.__previous_state)
            self.__changed = True

 9      if clicking:
            for row in self.__grid:
                for pixel in row:
                    if pixel.get_rect().collidepoint(mpos):
                        self.__drawn = True
                        self.__changed = True
                        if not self.__erase_button.get_pressed():
                            pixel.set_colour(self.__colour_grid.get_selected().get_colour())
                        else:
```

```
                        pixel.set_colour(None)

10      if self.__drawn:
            self.__redo_stack.reset()
            if unclick:
                self.__undo_stack.push(self.__previous_state)
                self.__drawn = False
                self.__changed = False
```

1 - The changed boolean attribute is True when the drawing grid has been changed in any way and is used in the customisation function to check whether updates are required on the character display.
2 - All the buttons are updated.
3 - When you click, the current state of the grid is saved to a temporary variable in case you draw anything.
4 - If the undo button was pressed, the undo method is called and changed is set to True.
5 - If the redo button was pressed, the redo method is called and changed is set to True.
6 - If the eraser button was pressed, the selected colour from the colour grid is deselected.
7 - Likewise, if a colour is selected in the colour grid and the eraser button is currently pressed down, the eraser button is released.
8 - If the clear button was pressed, the clear method is called, the previous state of the grid is pushed onto the undo stack, and changed is set to True.
9 - If the user is clicking, every pixel in the grid is looped through to check if the mouse position collides with their rect. If so, the user has tried to draw onto a pixel so drawn is set to true, changed is set to true, and the pixel's colour is updated.
10 - If drawn is set to true, the redo stack is reset and if the user has also unclicked (released the mouse button), they have stopped drawing and therefore the previous state of the grid should be pushed onto the undo stack, drawn set to False, and changed set to False.

The undo method that pushes the current state of the grid onto the redo stack and reverts the grid to the state popped from the top of the undo stack:

```
    def __undo(self):
        if not self.__undo_stack.empty():
            self.__redo_stack.push(self.grid_to_hex())
            self.hex_to_grid(self.__undo_stack.pop())
```

The redo method that pushes the current state of the grid onto the undo stack and reverts the grid to the state popped from the top of the redo stack:

```
    def __redo(self):
        if not self.__redo_stack.empty():
            self.__undo_stack.push(self.grid_to_hex())
            self.hex_to_grid(self.__redo_stack.pop())
```

The clear method that removes the colour from every pixel in the grid:

```
    def __clear(self):
        for row in self.__grid:
            for pixel in row:
                pixel.set_colour(None)
```

The get changed method that returns whether or not the grid has been changed in the current frame:

```
    def get_changed(self):
        return self.__changed
```

Finally, the draw method that draws the drawing grid to the screen:

```
    def draw(self, screen):
        pygame.draw.rect(screen, PALER_BACKGROUND, self.__border_rect)
        for row in self.__grid:
            for pixel in row:
                pixel.draw(screen)
        self.__undo_button.draw(screen)
        self.__redo_button.draw(screen)
        self.__clear_button.draw(screen)
        self.__erase_button.draw(screen)
        self.__colour_grid.draw(screen)
```

It first draws a background rect for the drawing grid to lie on, followed by the drawing grid and then all the buttons.

3.7 Player Class

This is the player class from the game_classes.py file that creates a playable character for the game and is used for both the 1 player and 2 player characters.

```
class Player():
    def __init__(self, hex_string, game_rect, player=0):
        self.__player = player
        match player:
            case 0:
                self.__initial_pos = pygame.math.Vector2((game_rect.center))
            case 1:
                self.__initial_pos = pygame.math.Vector2((game_rect.centerx - EIGHT_PIXELS//2,
                                                          game_rect.centery))
            case 2:
                self.__initial_pos = pygame.math.Vector2((game_rect.centerx + EIGHT_PIXELS//2,
                                                          game_rect.centery))
        self.__pos = self.__initial_pos.copy()
        self.__custom_character(hex_string)
        self.__image = self.__front_image
        self.__immune_image = self.__front_immune
```

```
        self.__rect = self.__image.get_rect(center = self.__pos)
        self.__speed = PLAYER_SPEED
        self.__lives = PLAYER_LIVES
        self.__bullets = []
        self.__bullet_damage = PLAYER_DAMAGE
        self.__last_shot = 0
        self.__fire_rate = FIRE_RATE
        self.__fire_rate_multiplier = 1
        self.__game_rect = game_rect

        self.__item = None
        self.__shoes = False
        self.__shoes_time = -1
        self.__shotgun = False
        self.__shotgun_time = -1
        self.__rapid_fire = False
        self.__rapid_fire_time = -1
        self.__backwards_shot = False
        self.__backwards_shot_time = -1
        self.__immunity = False
        self.__immunity_time = -1

        self.__spawned = True

        self.__new_rect = self.__rect.copy()
        self.__new_rect_x = self.__rect.copy()
        self.__new_rect_y = self.__rect.copy()

        self.__bullets_shot = 0

        self.__timer = 0
```

In this initialising function, the first thing it does is check where the player should spawn using a match statement with the passed in 'player' value, that represents 0 for the single player player, 1 for the first player in 2 player, and 2 for the second player in 2 player. It is then a case of initialising a lot of different attributes and also running the custom_character method.

The custom_character method that creates all the images for the player based on the hex_string passed in:

```
    def __custom_character(self, hex_string):
        self.__front_image, self.__front_immune = self.__hex_to_image(FRONT_PATTERNS,
                                                                       hex_string)
        self.__back_image, self.__back_immune = self.__hex_to_image(BACK_PATTERNS,
                                                                     hex_string)
        self.__left_image, self.__left_immune = self.__hex_to_image(LEFT_PATTERNS,
                                                                     hex_string)
        self.__right_image, self.__right_immune = self.__hex_to_image(RIGHT_PATTERNS,
                                                                       hex_string)
```

For each direction, it runs the hex_to_image method to create each image.

The hex_to_image method that creates a character image and an immune character image (a transparent white version of the image) based on the hex_string passed in and the patterns (a list of coordinates for each body part):

```
    def __hex_to_image(self, patterns, hex_string):
1       image = pygame.Surface((EIGHT_PIXELS, EIGHT_PIXELS), pygame.SRCALPHA)
        select_hex = split(hex_string[:COLOUR_DEPTH*len(patterns)], COLOUR_DEPTH)
1       hat_hex = split(split(hex_string[COLOUR_DEPTH*len(patterns):], COLOUR_DEPTH), 8)

2       for i in range(len(patterns)):
            for row, column in patterns[i]:
                pygame.draw.rect(image, get_key(select_hex[i], BITMAP_DICTIONARY),
                                 pygame.rect.Rect((column*PIXEL_RATIO,row*PIXEL_RATIO),
2                                                 (1*PIXEL_RATIO, 1*PIXEL_RATIO)))

3       for i in range(len(hat_hex)):
            for j in range(len(hat_hex[0])):
                if get_key(hat_hex[i][j], BITMAP_DICTIONARY):
                    pygame.draw.rect(image, get_key(hat_hex[i][j], BITMAP_DICTIONARY),
                                     pygame.rect.Rect((j*PIXEL_RATIO,i*PIXEL_RATIO),
3                                                     (1*PIXEL_RATIO, 1*PIXEL_RATIO)))


4       immune = pygame.mask.from_surface(image).to_surface(setcolor=(255,255,255,100),
                                                            unsetcolor=(0,0,0,0))

4       return image, immune
```

1 - The first thing the method does is create a blank image to draw everything to, and then splits the hex_string into the colours representing each body part and the colours representing the hat. The hat colours are split into a 2D array of rows and hex codes, the same format in which the hat grid is stored.
2 - It then draws the body of the character. So for each pattern and each coordinate in that pattern, a single pixel is drawn of the colour represented by the select_hex colour for that pattern (the colour chosen on the select grids when customising a character)
3 - It then draws the hat of the character. So for each colour in the 2D array of hex codes, it draws a pixel of colour of the hat if the colour exists in the colour dictionary.
4 - It then creates an immune version of the image, so it creates a mask of the surface and then converts it back into a surface with the colour of each pixel now white. Finally, it returns the images.

The move method that allows the character to move from keyboard input and then adjusts for collisions, taking parameter of the keys pressed in the current frame and the list of rects of the possible obstacles the player could collide with:

```
    def __move(self, keys, collidables):
        velocity_x = 0
        velocity_y = 0

        velocity_x, velocity_y = self.__move_input(velocity_x, velocity_y, keys)
```

```
        player_sides = {'TOP'    : self.__pos[Y] - EIGHT_PIXELS/2,
                        'BOTTOM' : self.__pos[Y] + EIGHT_PIXELS/2,
                        'LEFT'   : self.__pos[X] - EIGHT_PIXELS/2,
                        'RIGHT'  : self.__pos[X] + EIGHT_PIXELS/2}

        velocity_x, velocity_y = self.__edge_collisions(player_sides, velocity_x, velocity_y)

        velocity_x, velocity_y = self.__collidables_collisions(player_sides, velocity_x,
                                                               velocity_y, collidables)

        return velocity_x, velocity_y
```

The first thing it does is initialise directional velocities for the player and then sets values for them based on keyboard input using the move_input method. It then creates a dictionary of sides for the character (explained further in the collisions section in design) and adjusts the velocities of the player based on collisions with the edge of the game and with the list of collidable obstacles.

The move_input method that takes in the keys pressed in the current frame as a parameter and returns the velocities of the player that the user is trying to perform:

```
    def __move_input(self, velocity_x, velocity_y, keys):
1       if ((keys[pygame.K_w] and self.__player == 0)
                or (keys[pygame.K_w] and self.__player == 1)
                or (keys[pygame.K_9] and self.__player == 2)):
            velocity_y -= self.__speed
            self.__image = self.__back_image
            self.__immune_image = self.__back_immune
        if ((keys[pygame.K_s] and self.__player == 0)
                or (keys[pygame.K_s] and self.__player == 1)
                or (keys[pygame.K_o] and self.__player == 2)):
            velocity_y += self.__speed
            self.__image = self.__front_image
            self.__immune_image = self.__front_immune
        if ((keys[pygame.K_a] and self.__player == 0)
                or (keys[pygame.K_a] and self.__player == 1)
                or (keys[pygame.K_i] and self.__player == 2)):
            velocity_x -= self.__speed
            self.__image = self.__left_image
            self.__immune_image = self.__left_immune
        if ((keys[pygame.K_d] and self.__player == 0)
                or (keys[pygame.K_d] and self.__player == 1)
                or (keys[pygame.K_p] and self.__player == 2)):
            velocity_x += self.__speed
            self.__image = self.__right_image
1           self.__immune_image = self.__right_immune

2       if velocity_x != 0 and velocity_y != 0:
            velocity_x /= 2**0.5
2           velocity_y /= 2**0.5
```

```
3       if self.__shoes:
            velocity_x *= SHOE_MULTIPLIER
            velocity_y *= SHOE_MULTIPLIER

3       return velocity_x, velocity_y
```

1 - The first thing the method does is check which keys have been pressed, accommodating for different controls depending on the player, and then adding to the velocities depending on these, changing the images of the player at the same time to match the direction the player is moving in.
2 - I then check and adjust for diagonal movement, explained further in the prototyping of the character movement, by dividing each velocity by $\sqrt{2}$ if diagonal movement was detected.
3 - I then multiply these velocities by the speed multiplier of the shoes upgrade if the player has it and finally return the velocities.

The edge_collisions method that checks and adjusts for the player's collisions with the edges:

```
    def __edge_collisions(self, player_sides, velocity_x, velocity_y)
        if player_sides['LEFT'] - self.__game_rect.left < self.__speed and velocity_x < 0:
            velocity_x = self.__game_rect.left - player_sides['LEFT']
        elif self.__game_rect.right - player_sides['RIGHT'] < self.__speed and velocity_x > 0:
            velocity_x = self.__game_rect.right - player_sides['RIGHT']
        if player_sides['TOP'] - self.__game_rect.top < self.__speed and velocity_y < 0:
            velocity_y = self.__game_rect.top - player_sides['TOP']
        elif self.__game_rect.bottom - player_sides['BOTTOM'] < self.__speed and velocity_y >0:
            velocity_y = self.__game_rect.bottom - player_sides['BOTTOM']

        return velocity_x, velocity_y
```

It does this by checking if the distance to an edge is less than the player moves and if the player is moving towards it. If this is detected, it changes the amount moving towards the edge to become the distance between edge and player.

The collidables_collisions method that checks and adjusts for the player's collisions with all rects in the collidables list:

```
    def __collidables_collisions(self, player_sides, velocity_x, velocity_y, collidables):
        self.__new_rect_x.center = self.__pos + pygame.math.Vector2(velocity_x, 0)
        self.__new_rect_y.center = self.__pos + pygame.math.Vector2(0, velocity_y)
        self.__new_rect.center = self.__pos + pygame.math.Vector2(velocity_x, velocity_y)

        x = velocity_x
        y = velocity_y

        for collidable in collidables:
            collision_x = collidable.colliderect(self.__new_rect_x)
            collision_y = collidable.colliderect(self.__new_rect_y)
```

```
            diagonal = collidable.colliderect(self.__new_rect) and not (collision_x
                                                                        or collision_y)

            if collision_x or diagonal:
                if x < 0:
                    velocity_x = -(player_sides['LEFT'] - collidable.right)
                elif x > 0:
                    velocity_x = collidable.left - player_sides['RIGHT']

            if collision_y or diagonal:
                if y < 0:
                    velocity_y = -(player_sides['TOP'] - collidable.bottom)
                elif y > 0:
                    velocity_y = collidable.top - player_sides['BOTTOM']

        return velocity_x, velocity_y
```

This is all the same code as is explained in 2.11.3 Handling Collisions but in summary, it checks what type of collision the player is experiencing with each obstacle using rects, adjusting for whatever collision is found.

The shoot method that converts keyboard input into shooting:

```
    def __shoot(self, keys):
1       bullet_x = 0
        bullet_y = 0
        x_offset = 0
        y_offset = 0

        if ((keys[pygame.K_UP] and (self.__player == 0 or self.__player == 2))
                or (keys[pygame.K_g] and self.__player == 1)):
            bullet_y -= 1
        if ((keys[pygame.K_DOWN] and (self.__player == 0 or self.__player == 2))
                or (keys[pygame.K_b] and self.__player == 1)):
            bullet_y += 1
        if ((keys[pygame.K_LEFT] and (self.__player == 0 or self.__player == 2))
                or (keys[pygame.K_v] and self.__player == 1)):
            bullet_x -= 1
        if ((keys[pygame.K_RIGHT] and (self.__player == 0 or self.__player == 2))
                or (keys[pygame.K_n] and self.__player == 1)):
1           bullet_x += 1

2       if bullet_y == -1:
            x_offset = PIXEL_RATIO * 2.5
            y_offset = PIXEL_RATIO * -1
            self.__image = self.__back_image
            self.__immune_image = self.__back_immune
        elif bullet_y == 1:
            x_offset = PIXEL_RATIO * -2.5
            y_offset = PIXEL_RATIO * 5
            self.__image = self.__front_image
```

```
            self.__immune_image = self.__front_immune
        if bullet_x == -1:
            x_offset = PIXEL_RATIO * -5
            y_offset = PIXEL_RATIO * 1.5
            self.__image = self.__left_image
            self.__immune_image = self.__left_immune
        elif bullet_x == 1:
            x_offset = PIXEL_RATIO * 4
            y_offset = PIXEL_RATIO * 1.5
            self.__image = self.__right_image
2           self.__immune_image = self.__right_immune

3       if bullet_x != 0 and bullet_y != 0:
            bullet_x /= 2**0.5
3           bullet_y /= 2**0.5

4       if ((bullet_x != 0 or bullet_y != 0 )
                and (self.__timer - self.__last_shot >
                     self.__fire_rate * self.__fire_rate_multiplier)):
            offset = (x_offset, y_offset)
            direction = pygame.math.Vector2(bullet_x,bullet_y)
            bullet = Bullet(self.__pos + offset, direction, self.__bullet_damage, self.__timer)
            self.__bullets.append(bullet)
            self.__bullets_shot += 1
            if self.__shotgun:
                self.__shoot_shotgun(offset, direction)
            if self.__backwards_shot:
                self.__shoot_backwards_shot(offset, direction)
4           self.__last_shot = self.__timer
```

1 - The first thing this method does is create variables for the bullet x and y which represent where the player is trying to shoot, accommodating for the different controls of different players.
2 - It then checks through these x and y variables and sets an x and y offset variable which offset the location the bullets to spawn at to match the position of the gun in each direction, also changing the image of the player to match the direction that the player is shooting in. Since this code is run after the movement method, changing the image here gives more significance to the shooting direction
3 - In the same way I adjust the player's movement, I check if the bullets are travelling diagonally and then divide by $\sqrt{2}$ if so.
4 - I then check if the player is trying to shoot (the bullet x and y variables have values) and if enough time has passed since the player's last shot. If both of these are True, I create a bullet, add it to the player's bullet list, check if the player has either the shotgun or backwards shot item and run the procedure for these if so, and then reset the time for when the player last shot.

The shoot_shotgun method that shoots 2 extra bullets from the player at offset angles:

```
def __shoot_shotgun(self, offset, direction):
    bullet1 = Bullet(self.__pos + offset, direction.rotate(-11.25),
                     self.__bullet_damage, self.__timer)
    bullet2 = Bullet(self.__pos + offset, direction.rotate(11.25),
                     self.__bullet_damage, self.__timer)
    self.__bullets.append(bullet1)
    self.__bullets.append(bullet2)
    self.__bullets_shot += 2
```

To do this it creates 2 new bullets with their directions rotated by 11.25 degrees anticlockwise and 11.25 degrees clockwise respectively and then adds these to the player's list of bullets.

The shoot_backwards_shot method that shoots an extra bullet behind the player

```
def __shoot_backwards_shot(self, offset, direction)
    bullet = Bullet(self.__pos + offset, direction.rotate(180),
                    self.__bullet_damage, self.__timer)
    self.__bullets.append(bullet)
    self.__bullets_shot += 1
    if self.__shotgun:
        self.__shoot_shotgun(offset, -direction)
```

To do this it creates a new bullet with its direction flipped (rotated by 180 degrees) and adds it to the player's list of bullets. It also runs the shoot_shotgun method again if the item is active with the negative direction so that the items can work together and combine to make a very powerful combination.

Self-explanatory short functions and procedures that allows the rest of the program to interact with the player:

```
def add_shoes(self):
    self.__shoes = True
    self.__shoes_time = self.__timer

def add_shotgun(self):
    self.__shotgun = True
    self.__shotgun_time = self.__timer

def add_rapid_fire(self):
    self.__rapid_fire = True
    self.__rapid_fire_time = self.__timer

def add_immunity(self, time):
    self.__immunity = True
    self.__immunity_time = self.__timer + time

def add_backwards_shot(self):
    self.__backwards_shot = True
    self.__backwards_shot_time = self.__timer
```

```
    def increase_health(self, amount):
        self.__lives += amount

    def empty_bullets(self):
        self.__bullets = []

    def remove_bullet(self, bullet):
        self.__bullets.remove(bullet)
```

The update procedure that is run every frame and runs the movement, shooting, and collision functions and checks up on all the parts of the player:

```
    def update(self, collidables, other_player_rect=None):
1       self.__timer += 1
        self.__fire_rate_multiplier = ((RAPID_FIRE_MULTIPLIER if self.__rapid_fire else 1)
1                                      * (SHOTGUN_RATE_MULTIPLIER if self.__shotgun else 1))

2       for bullet in self.__bullets:
            for collidable in collidables:
                if collidable.colliderect(bullet.get_rect()):
                    self.__bullets.remove(bullet)
                    break
            bullet.update()
            if self.__timer - bullet.get_spawn_time() > BULLET_LIFETIME:
2               self.__bullets.remove(bullet)

3       if self.__lives > 0 and self.__spawned:
            keys = pygame.key.get_pressed()
            velocity_x, velocity_y = self.__move(keys, collidables +
                                                 ([other_player_rect] if other_player_rect
                                                    else []))
            self.__pos += pygame.math.Vector2(velocity_x, velocity_y)
            self.__rect.center = self.__pos
3           self.__shoot(keys)

4       if self.__shotgun and self.__timer - self.__shotgun_time >= SHOTGUN_LENGTH:
            self.__shotgun = False
        if self.__shoes and self.__timer - self.__shoes_time >= SHOES_LENGTH:
            self.__shoes = False
        if self.__rapid_fire and self.__timer - self.__rapid_fire_time >= RAPID_FIRE_LENGTH:
            self.__rapid_fire = False
        if self.__backwards_shot and (self.__timer - self.__backwards_shot_time >=
                                                                    BACKWARDS_SHOT_LENGTH):
            self.__backwards_shot = False
        if self.__immunity and self.__timer == self.__immunity_time:
4           self.__immunity = False
```

1 - The first thing the method does is updates the timer and the player's fire rate multiplier depending on the items that are active
2 - Then, every bullet is looped through to first check if they have collided with any object and remove them if so, and then to update them and check how long the bullet has been around for and if it is longer than it should be around for, the bullet is removed.
3 - Then, if the player is alive, I run the move method, change the position of the player, update the rect of the player, and then run the shoot method.
4 - Finally, I check how long each item has been active for if they are currently active and turn them off if they have been active longer than they should be.

The hit function that deals damage to the player (if they aren't immune) and resets the player:

```
def hit(self, damage):
    if not self.__immunity:
        self.__lives -= damage
        self.__item = None
        self.__shoes = False
        self.__shotgun = False
        self.__rapid_fire = False
        self.__backwards_shot = False
        self.__pos = self.__initial_pos.copy()
```

The draw function that draws the player to the screen:

```
def draw(self, screen):
    if self.__spawned:
        screen.blit(self.__image, (self.__pos[X] - EIGHT_PIXELS/2,
                                   self.__pos[Y] - EIGHT_PIXELS/2))
        if self.__immunity:
            if FPS//2 < (self.__immunity_time - self.__timer) % FPS < FPS:
                screen.blit(self.__immune_image, (self.__pos[X] - EIGHT_PIXELS/2,
                                                  self.__pos[Y] - EIGHT_PIXELS/2))
```

If the player has immunity, the immunity image will also be drawn to the screen every other 0.5 seconds.

3.8 Enemy Classes

3.8.1 Enemy Base Class

This is the parent enemy class from the game_classes.py file that all the different enemy types inherit from.

```
class Enemy():
    def __init__(self, pos, settings, initial_image, flying):
        self._pos = pygame.math.Vector2(pos)
        self._prev_pos = self._pos.copy()
        self._health = settings["HEALTH"]
        self._speed = settings["SPEED"]
        self.__score = settings["SCORE"]
        self._image = initial_image
        self._timer = 0
        self.__flying = flying
        self._rect = self._image.get_rect(center = self._pos)
        self.__red = False
        self.__hit_timer = 0
```

The initialising function takes in the enemy's position, a settings dictionary that specifies the health, speed, and score of the enemy, initial image, and boolean value for whether or not the enemy is flying as parameters. The attributes created here include the previous position of the enemy which is the position the enemy is reverted to if they collide with anything and the rect of the enemy. Some attributes are private and some are protected depending on whether or not the enemy classes will use them.

Following this are a few get methods that allow different parts of the program to access information from the enemies:

```
    def get_flying(self):
        return self.__flying

    def get_score(self):
        return self.__score

    def get_rect(self):
        return self._rect

    def get_health(self):
        return self._health
```

This is the hit method that takes away a certain amount of health from the enemy, causes the enemy to flash red for a few frames, and plays the enemy death sound if the enemy has lost all their lives:

```
    def hit(self, damage=1):
        self._health -= damage
        self.__red = True
        self.__hit_timer = 0
        if self._health == 0:
            enemy_killed.play()
```

The draw method that draws the enemy to the screen, blitting a red version of the enemy's image over the enemy if the enemy was hit recently to provide visual feedback:

```
    def draw(self, screen):
        self.__hit_timer += 1
        if self.__red and self.__hit_timer == HIT_TIME:
            self.__red = False
        screen.blit(self._image, (self._pos[X] - EIGHT_PIXELS/2, self._pos[Y] - EIGHT_PIXELS/2))
        if self.__red:
            red_image = pygame.mask.from_surface(self._image).to_surface(setcolor=HIT_COLOUR,
                                                                          unsetcolor=(0,0,0,0))
            screen.blit(red_image, (self._pos[X] - EIGHT_PIXELS/2,
                                    self._pos[Y] - EIGHT_PIXELS/2))
```

To create the red version of the enemy's image, a mask is created (which basically holds only the pixels that have colour) from the enemy's current image that is then converted back into an image with the pixels from the mask drawn as a pale red. This red version is then blitted at the same position as the enemy's image.

3.8.2 Default Class and Inheritance

This is the default enemy class from the game_classes.py file that creates a default enemy, inheriting from the enemy base class. The default enemy has 1 health, grants a score of 10 when killed, moves relatively slowly, spawns in large groups, and is easily influenced by randomness.

```
class DefaultEnemy(Enemy):
    def __init__(self, pos, direction):
        super().__init__(pos, DEFAULT_ENEMY, default_enemy_images[direction][1], False)
        self.__direction_change_time = 0
        self.__random_time = 0
        self.__direction = direction
        self.__images = default_enemy_images
```

The settings for the default enemy from the constants.py file:

```
DEFAULT_ENEMY = {"HEALTH" : 1,
                 "SPEED"  : 0.24 * PIXEL_RATIO * (60/FPS),
                 "SCORE"  : 10}
```

The initialising function initialises the super class with the settings for the enemy and initial image, followed by initialising some attributes that describe the last time the direction of the enemy was changed, the time until the direction is next changed, the direction the enemy is currently facing, and the 2D array of animation frames of the enemy respectively.

The next two functions are the move and change direction functions that move the enemy in its current direction and check the direction the enemy is currently facing respectively. These are explained in section 2.11.3 Enemy Movement.

```
    def __move(self):
        if self.__direction == LEFT:
            velocity_x = -self._speed
            velocity_y = 0
        elif self.__direction == RIGHT:
            velocity_x = self._speed
            velocity_y = 0
        elif self.__direction == UP:
            velocity_x = 0
            velocity_y = -self._speed
        elif self.__direction == DOWN:
            velocity_x = 0
            velocity_y = self._speed
        return velocity_x, velocity_y

    def __change_direction(self, player_pos):
        x_distance = player_pos[X] - self._pos[X]
        y_distance = player_pos[Y] - self._pos[Y]
        if abs(x_distance) > abs(y_distance):
            if x_distance < 0:
                self.__direction = LEFT
            else:
                self.__direction = RIGHT
        else:
            if y_distance < 0:
                self.__direction = UP
            else:
                self.__direction = DOWN
        self.__direction_change_time = self._timer

        if 2*EIGHT_PIXELS < (x_distance**2 + y_distance**2)**0.5 < 8*EIGHT_PIXELS:
            self.__random_time = randint(FPS//4, FPS)
            random_int = randint(-1,3)
            if random_int <= 1:
                self.__direction = (self.__direction + random_int) % 4
```

Next is the check collisions method that checks if the enemy has collided with anything after moving and if so, reverts its position back to its previous position. It is acceptable to just revert the enemy back to its previous position because the enemy's collisions don't need to be pixel perfect, unlike the collisions for the player.

```
    def __check_collisions(self, velocity_x, velocity_y, game_rect, collidables, enemy_rects):
        collision = False

1       if self._rect.collidelistall(collidables):
            collision = True

2       if ((self._rect.left <= game_rect.left and velocity_x < 0)
                or (self._rect.top <= game_rect.top and velocity_y < 0)
                or (self._rect.right >= game_rect.right and velocity_x > 0)
                or (self._rect.bottom >= game_rect.bottom and velocity_y > 0)):
            collision = True

        enemy_rects.remove(self._rect)
3       if self._rect.collidelistall(enemy_rects):
            collision = True

4       if collision:
            self._pos = self._prev_pos.copy()
            self._rect.center = self._pos
            self.__random_time = 6
```

1 - The first case checked is if there is any collision between the enemy and a collidable object.
2 - The second case checked is if there is any collision between the enemy and the game sides.
3 - The third case checked is if the enemy collides with another enemy, which first requires removing the enemy's own rect from the list of enemy rects.
4 - Finally, if there is a collision, the enemy is returned to their previous position and the time before the next direction change is reduced.

The final method of the default enemy class is the update method. This is called every frame and is therefore responsible for calling the rest of the methods of the class. The position of a second player can be passed in as a parameter and if it is, the enemy moves towards the closest player.

```
    def update(self, player_pos, game_rect, collidables, enemy_rects, player2_pos=None):
1       if player2_pos:
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
1               player_pos = player2_pos

        self._timer += 1

2       if player_pos:
            if self._timer - self.__direction_change_time > self.__random_time:
                self.__change_direction(player_pos)
            velocity_x, velocity_y = self.__move()
```

```
            self._image = self.__images[self.__direction][trunc(((self._timer * 4)/FPS) %
                                                               len(self.__images[0]))]
            self._prev_pos = self._pos.copy()
            self._pos += (velocity_x, velocity_y)
            self._rect.center = self._pos
2           self.__check_collisions(velocity_x, velocity_y, game_rect, collidables, enemy_rects)
```

1 - The first thing to do is check which player is closest to the enemy if a second position is passed in. To do this, the distance of each player is calculated and compared, the position of the first player being set to the maximum possible value of distance if it is passed in as None (meaning player 1 is dead in a 2 player game). The closest player's position is taken as the position for the enemy to move towards.
2 - The enemy then needs to move towards this position. It first checks if the enemy should change direction, calling the change direction method if so, and then retrieves x and y velocities for the enemy from the move method. The image, previous position, current position, and rect of the enemy are then updated followed by a check for collisions which will return the enemy to their new previous position if any are detected.

The default enemy class does not need any drawing function as this is handled by the base class.
Most of the other enemy classes operate similarly to this, just with different movement algorithms (explained in section 2.11.2 Enemy movement) and collision detection systems.

3.8.3 Crow Class and Polymorphism

This is the default enemy class from the game_classes.py file that creates a default enemy, inheriting from the enemy base class and overriding one of its methods. The crow enemy has 1 health, grants a score of 30 when killed, and moves very quickly in a straight line across the screen. Before it starts flying, a warning sign is displayed and a crow sound effect is played.

```
class CrowEnemy(Enemy):
    def __init__(self, pos, direction, game_rect):
        super().__init__(pos, CROW_ENEMY, crow_enemy_images[direction], True)
        self.__blur_image = self._image.copy()
        self.__blur_image.set_alpha(100)
        self.__blur_rect = self._rect.copy()
        self.__large_rect = pygame.Rect((self._rect.x - EIGHT_PIXELS,
                                         self._rect.y - EIGHT_PIXELS),
                                        (self._rect.width + 2*EIGHT_PIXELS,
                                         self._rect.height + 2*EIGHT_PIXELS))
        self.__game_rect = game_rect
        self.__set_velocities(direction)
        self.__exclamation_pos = (pygame.math.Vector2(self._rect.topleft) +
                                  (pygame.math.Vector2(self.__velocity)*(EIGHT_PIXELS/self._speed)))
```

The settings for the crow enemy from the constants.py file:

```
CROW_ENEMY = {"HEALTH" : 1,
              "SPEED"  : 1.2 * PIXEL_RATIO * (60/FPS),
              "SCORE"  : 30}
```

The initialising method initialises the super class with the settings for the enemy and initial image, followed by initialising some attributes. The blur attributes, blur image and blur rect, are created for displaying a slightly transparent version of the crow enemy a few pixels behind, creating a blur effect for high movement speed. There is also the large rect attribute, which simply creates a larger rect of the enemy used for checking if the enemy has flown off the screen. Finally, the position for the exclamation mark warning sign to be displayed is set. Because the crow is spawned 8 pixels (the width of its image) away from the side of the screen, this position can be calculated by simply copying and altering the position of the crow by 8 pixels in the direction it moves in.

The set velocity method, called in the initialisation, that takes a direction as a parameter and sets velocity attributes relevant to the direction:

```
def __set_velocities(self, direction):
    blur_distance = CROW_BLUR_DISTANCE
    if direction == RIGHT:
        self.__velocity = (self._speed, 0)
        self.__blur_pos = (-blur_distance, 0)
    elif direction == LEFT:
        self.__velocity = (-self._speed, 0)
        self.__blur_pos = (blur_distance, 0)
    elif direction == DOWN:
        self.__velocity = (0, self._speed)
        self.__blur_pos = (0, -blur_distance)
    elif direction == UP:
        self.__velocity = (0, -self._speed)
        self.__blur_pos = (0, blur_distance)
```

This is created and called because the crow enemy only needs one velocity as it moves in a straight line. The position for the blurred image is also set here to be a little bit behind the crow.

The update function that moves and updates the attributes of the enemy, disregarding any arguments passed in (so that the same code to call the update methods can be used on different enemy types):

```
def update(self, *args):
    if self._timer == 0:
        crow_sound.play()
    self._timer += 1
    if self._timer > CROW_PAUSE:
        self._pos += self.__velocity
        self._rect.center = self._pos
        self.__blur_rect.center = self._pos + self.__blur_pos
        self.__large_rect.center = self._pos
        if not self.__large_rect.colliderect(self.__game_rect):
            self._health = CROW_OFFSCREEN
```

It starts by playing the crow sound when the timer of the crow (number of frames since spawning) is 0 because if it was called in the initialisation method, the sound of every crow would be played when

the wave of enemies was generated. It then moves the crow, only moving after the warning sign has been displayed, and if the crow has flown off screen, sets the health of the crow to some constant that is detected by the game class as to not generate any score.

Finally is the draw method that draws the enemy to the screen. Polymorphism was required here to override the base class' draw method as the warning sign and blurred image of the crow need to be displayed too.

```
    def draw(self, screen):
        if self._timer < CROW_PAUSE // 1.5:
            screen.blit(exclamation, self.__exclamation_pos)
        screen.blit(self.__blur_image, self.__blur_rect)
        screen.blit(self._image, self._rect)
```

The warning sign is displayed for the first two-thirds of the pause before the enemy starts moving across the screen, done by comparing the enemy's timer to a constant.

3.9 Dynamically Generating Enemies and Using Queues

There are multiple functions involved in generating enemies. These are all methods extracted from the game class which hasn't been fully demonstrated in the technical solution due to the fact that most of the important parts of it have been explained already in the document and the fact that it is a very large class. It can however be seen in section 6.11 of the source code. Most of the methods for generating enemies are responsible for creating a group of enemies with a different function to handle each enemy type (except for the default enemy and tough enemy whose function is combined as they are similar).

The method for creating a group of default or tough enemies can be seen below, taking in an enemy class (default or tough) as well as the amount of enemies to spawn as parameters. It returns a dictionary that stores the list of enemies from the group under the 'enemies' key and the sum of the enemies scores under the 'difficulty' key. The reason it is called the difficulty is because it can be seen as a measure of an enemy groups' difficulty and is used for determining how long the game waits before spawning more enemies.

```
    def __enemy_group(self, enemy_class, amount):
1       enemies = []
        spawn_side = randint(0,3)
1       side_positions = [2, 1, 3, 0]

2       for i in range(amount):
            if spawn_side == UP:
                enemies.append(enemy_class((self.__rect.centerx - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS,
                                            self.__rect.top + EIGHT_PIXELS/2),
                                           direction=(spawn_side+2)%4))
            elif spawn_side == LEFT:
                enemies.append(enemy_class((self.__rect.left + EIGHT_PIXELS/2,
                                           self.__rect.centery - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS),
```

```
                                        direction=(spawn_side+2)%4))
            elif spawn_side == DOWN:
                enemies.append(enemy_class((self.__rect.centerx - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS,
                                        self.__rect.bottom - EIGHT_PIXELS/2),
                                        direction=(spawn_side+2)%4))
            elif spawn_side == RIGHT:
                enemies.append(enemy_class((self.__rect.right - EIGHT_PIXELS/2,
                                        self.__rect.centery - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS),
2                                       direction=(spawn_side+2)%4))

3       difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score()

3       return {'enemies':enemies, 'difficulty':difficulty}
```

1 - The first thing the method does is create an empty to store the enemies in and then randomise which side the enemies should spawn from. After this, a list is initialised that specifies the order of preference for where enemies should spawn from their side (each side has 4 possible spawning locations). This is done so that if for example you are trying to spawn 6 enemies from the top of the game, the first 4 will spawn in a row and once they have moved away, the 5th and 6th enemies will spawn in the middle behind them, not spawning at either of the edge spawning locations.
2 - A for loop is then used to initialise each enemy and add them to the list, spawning them at specific positions with their initial directions being opposite to the side they spawn on.
3 - The difficulty of the group is then found by summing the score of each enemy in the list which is returned in a dictionary along with the list of enemies.

This function is similar to the functions for spawning the other enemy types, all returning a dictionary of the same format but using different initial parameters or different spawning patterns for the enemies.

The next function used in generating enemies is the generate enemy waves function that adds all the dictionaries of enemy groups for a wave to the enemy queue after a crate has spawned. Below is the 1 player version of the function:

```
    def __generate_enemy_waves_1p(self):
1       if self.__wave_index < 8:
            total_difficulty = 500 + 400*self.__wave_index - (self.__wave_index**3)
        else:
            total_difficulty = 3000
1       difficulty = 0

2       while difficulty < total_difficulty:
            match randint(0,(self.__wave_index + 1) if self.__wave_index < 5 else 5):
                case 0:
                    group = self.__enemy_group(DefaultEnemy,
                                        randint(2, 6) if self.__wave_index < 3
```

```
                                        else (randint(4, 10) if self.__wave_index < 6
                                                else randint(2,6)))
                case 1:
                    group = self.__fast_enemy(1 if self.__wave_index < 3
                                            else (randint(1, 2) if self.__wave_index < 6
                                                  else randint(2,3)))
                case 2:
                    group = self.__flying_enemy(1 if self.__wave_index < 2
                                                else (randint(1, 2) if self.__wave_index < 7
                                                      else randint(2,4)))
                case 3:
                    group = self.__crow_enemy()
                case 4:
                    group = self.__enemy_group(ToughEnemy,
                                            randint(4,6) if self.__wave_index < 4
                                                else (randint(4, 10) if self.__wave_index < 8
                                                      else randint(6,12)))
                case 5:
                    group = self.__spirit_enemy(1 if self.__wave_index < 5
                                                else (randint(1, 3) if self.__wave_index < 8
                                                      else randint(2,4)))
            self.__enemy_queue.enqueue(group)
2           difficulty += group['difficulty']
```

1 - First, the total difficulty of the whole wave is decided. This is done by a simple equation created from trial and error of $500 + 400x - x^3$ where x is the index of the wave. This equation is limited to inputs beneath 8 so that it firstly limits to 3000 but also because the total difficulty would start to decrease after wave index 12.

2 - A while loop then continuously runs until enough enemies have been enqueued to the enemy queue for their difficulties to sum to a value greater than the total difficulty. In each iteration, a random number is generated for which enemy type to spawn and the function to spawn a group of that enemy type is called. This random number for which enemy to spawn is limited by the wave index so that there is a progression of enemies spawning with only default and fast enemies being able to spawn in the first wave, followed by flying enemies being able to spawn from the second wave, crow enemies being able to spawn from third, tough enemies being able to spawn from the fourth, and finally spirit enemies being able to spawn from the fifth. The enemy group functions are called with a randomly generated number specifying how many enemies to spawn with the bounds of this number being different for each enemy type and dynamically changing with the wave index. Finally, the group of enemies generated from the function is enqueued and the difficulty is added to the sum.

The two player version of the function differs by having a different equation for the total difficulty and a different max value, as well as allowing more enemies of each type to spawn at once.

The next function is a short function for generating the opening wave. Creating this function allows me to have complete control over what is first spawned in the game.

```
    def __first_waves(self):
        self.__enemy_queue.enqueue(self.__enemy_group(DefaultEnemy, randint(3,6)))
        self.__enemy_queue.enqueue(self.__enemy_group(DefaultEnemy, randint(4,8)))
```

The last function involved with spawning enemies is the spawn enemy function that handles dequeuing the enemies and spawning them as well as spawning crates.

```
    def __spawn_enemies(self):
1       if self.__wave_index == -1:
            self.__first_waves()
1           self.__wave_index += 1

2       if self.__enemy_queue.empty():
            if not self.__enemies and not self.__enemies_to_spawn:
                if not self.__crate_countdown:
                    self.__crate_countdown = 1*FPS # first countdown, until a crate
                elif self.__crate_countdown == 1:  # 0 would be caught by first if
                    crate_thud.play()
                    self.__place_random(crate_image, 1, 2, collision=True, center_spawn=False)
                    if self.__players == 2: # place a second crate in 2 player
                        self.__place_random(crate_image,1,2,collision=True,center_spawn=False)

                    if self.__players == 1:
                        self.__generate_enemy_waves_1p()
                    elif self.__players == 2:
                        self.__generate_enemy_waves_2p()

                    self.__wave_index += 1
2                   self.__enemy_countdown = 1*FPS # time before the enemies are spawned
3       elif self.__enemy_countdown == 0: # spawn enemies
            group = self.__enemy_queue.dequeue()
            self.__enemies_to_spawn.extend(group['enemies'])
            self.__enemy_countdown =  DELAY_MULTIPLIER * (group['difficulty'] -
                                                          self.__wave_index * INDEX_MULTIPLIER)
            if self.__enemy_countdown < MIN_FRAMES:
3               self.__enemy_countdown = MIN_FRAMES
```

1 - The first part of this function is run when the wave index is -1 so that the first wave of enemies can spawn.
2 - The second part of this function is run when the enemy queue is empty, the enemies list is empty, and the enemies waiting to spawn list is empty. This means a wave has been completed and a crate and more enemies need to be spawned. The first thing checked in this part is if the crate countdown has yet to be started. If this is the case, the crate countdown is started that represents the delay before a crate is spawned. This is decremented every frame. When this countdown has reached 1, a crate is spawned and the next wave of enemies is generated. This involves playing the crate thud sound, placing a crate, placing a second crate if it is two player, calling the appropriate enemy wave

generator method, incrementing the wave index, and initialising an enemy countdown. The enemy countdown, similarly to the crate countdown, is decremented each frame and represents the delay before enemies are spawned.

3 - The last part of this function is to actually spawn enemies and is run when the enemy queue is not empty and the enemy countdown has reached 0. This involves dequeuing a group from the enemy queue, adding the enemies from it to the waiting list of enemies to spawn, and finally deciding the length of the next enemy countdown. This is calculated from a simple equation that is influenced by the sum of scores of the group that just spawned as well as the current wave index, each having effects of different proportions using multipliers. This is limited to a minimum number of frames so that there is always at least a small delay between groups of enemies spawning.

3.10 Updating Enemies and Generating Items

This is the update enemies method from the game class that runs the update function for each enemy, checks if they've been shot, and checks if they're dead, creating a Score object and having a chance to spawn an item if they are.

```
    def __update_enemies(self):
        for enemy in self.__enemies:
1           if self.__players == 1:
                for bullet in self.__player.get_bullets():
                    if enemy.get_rect().colliderect(bullet.get_rect()):
                        enemy.hit(bullet.get_damage())
                        self.__player.remove_bullet(bullet)
            elif self.__players == 2:
                for bullet in self.__player1.get_bullets():
                    if enemy.get_rect().colliderect(bullet.get_rect()):
                        enemy.hit(bullet.get_damage())
                        self.__player1.remove_bullet(bullet)
                for bullet in self.__player2.get_bullets():
                    if enemy.get_rect().colliderect(bullet.get_rect()):
                        enemy.hit(bullet.get_damage())
1                       self.__player2.remove_bullet(bullet)

2           if not self.__time_freeze:
                if self.__players == 1:
                    if not enemy.get_flying():
                        enemy.update(self.__player.get_pos(), self.__rect,
                                     self.__collidable_rects, self.__enemy_rects.copy())
                    else:
                        enemy.update(self.__player.get_pos())
                elif self.__players == 2:
                    player1_pos, player2_pos = None, None
                    if self.__player1.get_lives() > 0 and not self.__player1.get_immunity():
                        player1_pos = self.__player1.get_pos()
                    if self.__player2.get_lives() > 0 and not self.__player2.get_immunity():
                        player2_pos = self.__player2.get_pos()

                    if not enemy.get_flying():
```

```
                    enemy.update(player1_pos, self.__rect, self.__collidable_rects,
                                 self.__enemy_rects.copy(), player2_pos=player2_pos)
                else:
2                   enemy.update(player1_pos, player2_pos=player2_pos)

3           if enemy.get_health() == CROW_OFFSCREEN:
                self.__enemies.remove(enemy)
            elif enemy.get_health() <= 0:
                chance = ITEM_CHANCE_1P if self.__players == 1 else ITEM_CHANCE_2P
                if self.__item_countdown == 0 and randint(1, chance) == 1:
                    if ((self.__players == 1 and self.__player.get_lives() < 4)
                            or (self.__players == 2 and (self.__player1.get_lives() < 4
                                                         or self.__player2.get_lives() < 4))):
                        spawn_lives = True
                    else:
                        spawn_lives = False
                    self.__items.append(Item(enemy.get_rect().center,
                                             randint(0, len(item_images)-1 -
                                                     (0 if spawn_lives else 1))))
                    self.__item_countdown = ITEM_COUNTDOWN
                self.__enemies.remove(enemy)
                self.__scores.append(Score(self.__small_font, WHITE,
                                           enemy.get_score(), enemy.get_rect(),
                                           alpha=SCORE_ALPHA))
                self.__enemy_score += enemy.get_score()
3               self.__enemies_killed += 1
```

1 - In the for loop that loops through every enemy, the first thing done is a check for if the enemy has been shot by any bullet. This consists of first checking if the two player mode is being played and then just looping through each bullet to find any collisions. If a collision is found, the bullet is destroyed and the enemy's health is reduced.

2 - Then, if time is not frozen, the update function for each enemy is run. This consists of finding the positions of the players and then calling the update functions with different parameters for enemies that can fly and for enemies that can't.

3 - Finally, the enemy's health is checked. This includes a first check to see if the enemy was a crow that flew off of the screen and then a check to see if the enemy has 0 or less health. If they do, they must have been killed by the player and a random number can be generated to see if an item should spawn. If this was successful, a check is done for if the life item is able to spawn and then a random Item object is initialised and added to the list of items. The enemy is removed from the list of enemies and a Score object is also initialised and added to the list of scores. The game's score is increased by the score of the enemy and the number of enemies killed is incremented.

3.11 Leaderboard Classes

3.11.1 Leaderboard Class and Parameterised Database Queries

This class creates a customisable leaderboard, taking in lots of parameters to allow lots of control. The way it does this is by querying the database and creating a certain number of leaderboard records that each display a single user and their score.

```
class Leaderboard():
1   def __init__(self, db_cursor, pos, width, font, value_field, value_field_table,
                 key_field_table, key_field="username", rows=10, record_height=10*PIXEL_RATIO,
                 highlight_key=None, position_colour=SLIGHT_GREY, display_characters=False,
                 characters_field="custom_character"):
        self._database = db_cursor
        self._value_field_table = value_field_table
        self._value_field = value_field
        self._key_field = key_field
        self._key_field_table = key_field_table
        self.__display_characters = display_characters
        self.__characters_field = characters_field
        self.__highlight_key = highlight_key
        self._rows = rows
1       self.__rect = pygame.Rect(pos, (width, record_height*rows + (rows+1)*PIXEL_RATIO))

2       key_fields = self._fetch_key_fields()

        values = self.__fetch_values()

        if display_characters:
            characters = self.__fetch_custom_characters()
        else:
            characters = None

2       positions = position_list(1, rows)

3       self.__records = []
        for i in range(len(positions)):
            highlight = False
            if check_index(key_fields, i):
                key_field = key_fields[i][0] # the key field for the record
                if highlight_key in key_field:
                    highlight = True
            else:
                key_field = ""

            if check_index(values, i):
                value = values[i][0] # the value for the record
            else:
                value = None

            if display_characters and check_index(values, i):
```

```
                character_hex = characters[i][0] # the character hex string for the record
            else:
                character_hex = None

            self.__records.append(LeaderboardRecord((pos[X]+PIXEL_RATIO,
                                                     pos[Y] + i*record_height +
                                                                      (i+1)*PIXEL_RATIO),
                                                    (width-2*PIXEL_RATIO, record_height),
                                                    font, positions[i], key_field,
                                                    value, character_hex=character_hex,
                                                    position_colour=position_colour,
3                                                   highlight=highlight))
```

1 - The first thing done is initialising all the attributes of the class. Some of these attributes are protected and some are private depending on whether or not the two player leaderboard subclass needs to use them. A lot of the attributes refer to what fields are used in the database queries. For all purposes of this project, these attributes remain the same but I decided to make it customisable to allow for any changes in the design of the database as well as leaving the possibility of making a leaderboard that displays something other than scores open.
2 - After this, all the database queries are run and put into lists. This includes querying for the key fields (usernames), values (scores), and custom characters. As well as this, a list of positions is created using the position list function from the utility_functions.py file.
3 - Finally, the leaderboard records are created. This involves looping through all the positions and setting variables for each thing displayed in the leaderboard record (position, key field, value, and custom character), as well as whether the record should be highlighted or not depending on what account the user is logged into. A record is then initialised with these variables as parameters and added to the list of records.

The fetch key fields method that fetches and returns all the key fields for the leaderboard:

```
    def _fetch_key_fields(self):
        key_fields_query = f"""SELECT {self._key_field_table}.{self._key_field}
                               FROM {self._key_field_table}, {self._value_field_table}
                               WHERE {self._key_field_table}.{self._key_field} =
                                                    {self._value_field_table}.{self._key_field}
                               ORDER BY {self._value_field_table}.{self._value_field} DESC
                               LIMIT {self._rows}"""
        return self._database.execute(key_fields_query).fetchall()
```

This is a protected method instead of private so that it can be overwritten by its two player leaderboard subclass.

The fetch values method that fetches and returns all the values for the leaderboard:

```
    def __fetch_values(self):
        values_query = f"""SELECT {self._value_field}
                           FROM {self._value_field_table}
                           ORDER BY {self._value_field} DESC
                           LIMIT {self._rows}"""
        return self._database.execute(values_query).fetchall()
```

The fetch custom characters method that fetches and returns all the custom characters for the leaderboard

```
    def __fetch_custom_characters(self):
        characters_query = f"""SELECT {self._key_field_table}.{self.__characters_field}
                               FROM {self._key_field_table}, {self._value_field_table}
                               WHERE {self._key_field_table}.{self._key_field} =
                                                    {self._value_field_table}.{self._key_field}
                               ORDER BY {self._value_field_table}.{self._value_field} DESC
                               LIMIT {self._rows}"""
        return self._database.execute(characters_query).fetchall()
```

The update method that re-queries all the data in the leaderboard and updates the records to match:

```
    def update(self):
1       key_fields = self._fetch_key_fields()
        for i in range(len(self.__records)):
            if check_index(key_fields, i):
                self.__records[i].set_key_field(key_fields[i][0])

2       values = self.__fetch_values()
        for i in range(len(self.__records)):
            if check_index(values, i):
                self.__records[i].set_value(values[i][0])

3       if self.__display_characters:
            characters = self.__fetch_custom_characters()
            for i in range(len(self.__records)):
                if check_index(characters, i):
                    self.__records[i].set_character_display(characters[i][0])

        # checks for changes in record highlighting
4       for record in self.__records:
            if self.__highlight_key in record.get_key_field():
                record.set_highlight(True)
            else:
                record.set_highlight(False)
```

1 - It first queries for all the key fields, then loops through the records to update them
2 - It then queries for all the values, then loops through the records to update them
3 - It then queries for all the characters, then loops through the records to update them
4 - It then looks through the key fields of each record to check if they should be highlighted or not, updating the highlight attribute of the records.

Finally is the draw method that draws all the records of the leaderboard to the screen:

```
    def draw(self, screen):
        pygame.draw.rect(screen, TEXT_BUTTON_HOVER_COLOUR, self.__rect)
        for record in self.__records:
            record.draw(screen)
```

As well as the records, it draws a background rect for the leaderboard.

The leaderboard record class is pretty self-explanatory and is mostly just formatting so I won't go over it here but it can be seen in section 6.8 of the source code.

3.11.2 Two Player Leaderboard Class and Polymorphism

This class creates a customisable two player leaderboard, also taking in lots of parameters to allow lots of control. It inherits from the leaderboard class and works by overriding the fetch key fields method to return a list that includes both player 1 and player 2.

```
class TwoPlayerLeaderboard(Leaderboard):
    def __init__(self, db_cursor, pos, width, font, value_field, value_field_table,
                 key_field_table, key_field="username", key_field1="player1_name",
                 key_field2="player2_name", rows=10, record_height=10*PIXEL_RATIO,
                 highlight_key=None, position_colour=SLIGHT_GREY, display_characters=False,
                 characters_field="custom_character"):
        self._key_field1 = key_field1
        self._key_field2 = key_field2
        super().__init__(db_cursor, pos, width, font, value_field, value_field_table,
                         key_field_table, key_field=key_field, rows=rows,
                         record_height=record_height, highlight_key=highlight_key,
                         position_colour=position_colour,
                         display_characters=display_characters,
                         characters_field=characters_field)
```

The initialising method first sets some new protected attributes and then initialises the super class. Nothing else is required in this method as the polymorphism of the next method handles all the changes between the normal leaderboard and the two player leaderboard.

The new fetch key fields method which now returns a list (in the same format as the original method of the leaderboard class did) of concatenated strings containing the names of both players 1 and 2:

```
    def _fetch_key_fields(self):
        key_fields1_query = f"""SELECT {self._key_field_table}.{self._key_field}
                                FROM {self._key_field_table}, {self._value_field_table}
                                WHERE {self._key_field_table}.{self._key_field} =
                                                {self._value_field_table}.{self._key_field1}
                                ORDER BY {self._value_field_table}.{self._value_field} DESC
                                LIMIT {self._rows}"""
        key_fields2_query = f"""SELECT {self._key_field_table}.{self._key_field}
                                FROM {self._key_field_table}, {self._value_field_table}
                                WHERE {self._key_field_table}.{self._key_field} =
                                                {self._value_field_table}.{self._key_field2}
                                ORDER BY {self._value_field_table}.{self._value_field} DESC
                                LIMIT {self._rows}"""
        fields1 = self._database.execute(key_fields1_query).fetchall()
        fields2 = self._database.execute(key_fields2_query).fetchall()
        key_fields = []
        for i in range(len(fields1)):
            # the player 1 and player 2 fields are combined into single key fields
            key_fields.append((f"{fields1[i][0]} + {fields2[i][0]}", ))
        return key_fields
```

The way it does this is simply by querying for both the first player names and the second player names. These are then looped through, concatenated together, and added to a list of the key fields which is returned at the end.

The line that initialises the main menu's single player leaderboard:

```
leaderboard = Leaderboard(db_cursor, (13*EIGHT_PIXELS, 7.5*EIGHT_PIXELS), 10*EIGHT_PIXELS,
                          small_font, "score", "SinglePlayerGames", "Players",
                          rows=5, record_height=9*PIXEL_RATIO, highlight_key=username)
```

The line that initialises the leaderboard screen's single player leaderboard:

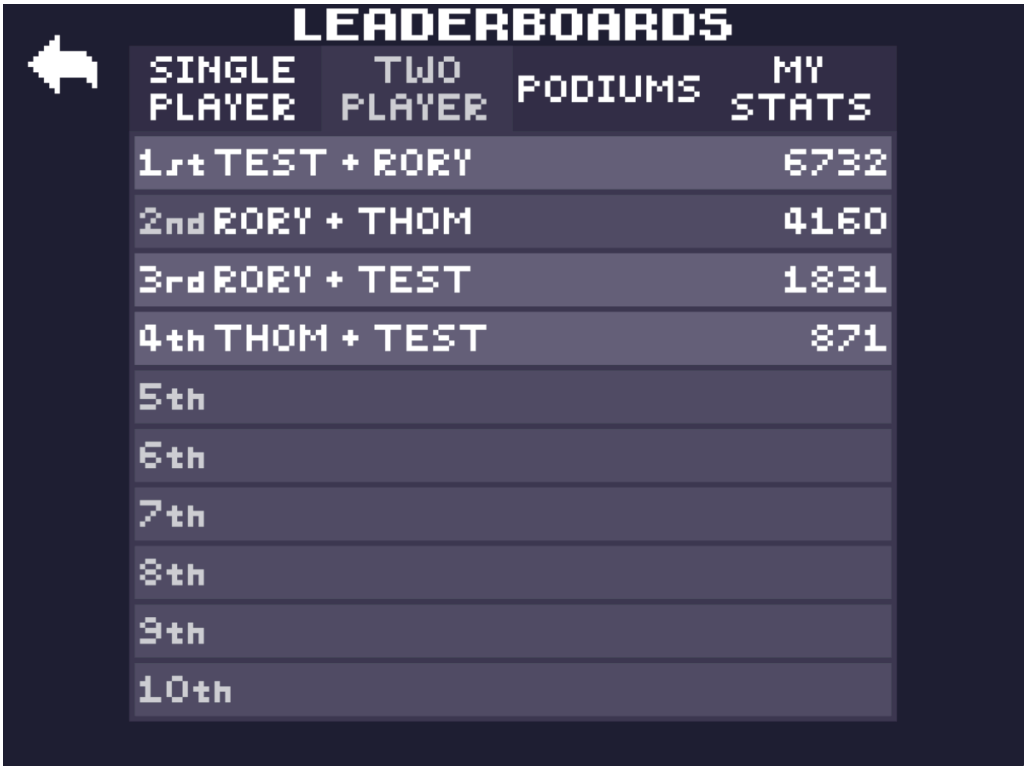
```
one_player_leaderboard = Leaderboard(db_cursor, (3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                                     18*EIGHT_PIXELS, small_font,
                                     "score", "SinglePlayerGames", "Players",
                                     rows=10, highlight_key=username, display_characters=True)
```

The line that initialises the leaderboard screen's two player leaderboard:

```
two_player_leaderboard = TwoPlayerLeaderboard(db_cursor, (3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                                              18*EIGHT_PIXELS, small_font,
                                              "score", "TwoPlayerGames", "Players",
                                              rows=10, highlight_key=username)
```

The single player leaderboard on the leaderboards screen from TEST's account:

The two player leaderboard on the leaderboards screen from TEST's account:



3.12 Main Menu Function

This is the function from the main.py file of the main menu that controls and allows access to each screen.

```
def main_menu(username):
1   one_player_button = TextButton((2*EIGHT_PIXELS, 7.5*EIGHT_PIXELS),
                                   (10*EIGHT_PIXELS, 4*EIGHT_PIXELS),
                                   "SINGLE"+NEW_LINE+"PLAYER", big_font)
    two_player_button = TextButton((2*EIGHT_PIXELS, 12*EIGHT_PIXELS),
                                   (10*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                   "TWO PLAYER", medium_font)
    settings_button = ImageButton((2*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                  (2*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                  settings_image)
    customise_button = ImageButton((20.5*EIGHT_PIXELS, 3.5*EIGHT_PIXELS),
                                   (2*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                   customise_image)
    leaderboard_button = TextButton((14*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                    (8*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                    "LEADERBOARDS"+NEW_LINE+"+ STATISTICS", small_font)
    log_out_button = TextButton((4.5*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                (3.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                "LOG"+NEW_LINE+"OUT", small_font)
    quit_button = TextButton((8.5*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                             (3.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                             "QUIT", small_font)

    player_character = db_get_character(username, db_cursor)
    character_display = CharacterDisplay((16*EIGHT_PIXELS, 2.5*EIGHT_PIXELS),
                                         4, GRASS_GREEN, player_character,
                                         extra_border_colour=CHARACTER_DISPLAY_BORDER)

    leaderboard = Leaderboard(db_cursor, (13*EIGHT_PIXELS, 7.5*EIGHT_PIXELS), 10*EIGHT_PIXELS,
                              small_font, "score", "SinglePlayerGames", "Players",
                              rows=5, record_height=9*PIXEL_RATIO, highlight_key=username)

    title_font_silver = huge_font.new_colour_copy(SILVER)
    title_font_white = huge_font.new_colour_copy(WHITE)
    title_top = 7*PIXEL_RATIO
1   title_left = 10*PIXEL_RATIO

    mpos = pygame.mouse.get_pos()
    display_mouse = True
    while True:
2       click = False
        event_list = pygame.event.get()
        for event in event_list:
            if event.type == pygame.QUIT:
                quit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
```

```
            if event.button == 1:
                click = True
        elif event.type == pygame.KEYDOWN:
2           display_mouse = False

    screen.fill(BACKGROUND_COLOUR)

    previous_mpos = mpos
    mpos = pygame.mouse.get_pos()
    if mpos != previous_mpos:
        display_mouse = True

3   one_player_button.update(mpos, click)
    two_player_button.update(mpos, click)
    settings_button.update(mpos, click)
    customise_button.update(mpos, click)
    leaderboard_button.update(mpos, click)
    quit_button.update(mpos, click)
3   log_out_button.update(mpos, click)

4   if one_player_button.get_clicked():
        player_character = db_get_character(username, db_cursor)
        highscore = db_get_1p_highscore(username, db_cursor)
        play = True
        while play:
            game_data = game(player_character, highscore)
            if game_data:
                play = score_screen(username, *game_data, highscore)
            else:
                play = False
        leaderboard.update()
4           mpos = pygame.mouse.get_pos()

5   elif two_player_button.get_clicked():
        username_two = login(title="PLAYER TWO LOGIN:", button_text="START",
                             blocked_names=[username])
        if username_two:
            player1_character = db_get_character(username, db_cursor)
            player2_character = db_get_character(username_two, db_cursor)
            highscore = db_get_2p_highscore(username, db_cursor)
            play = True
            while play:
                game_data = game(player1_character, highscore,
                                 players=2, player2_hex=player2_character)
                if game_data:
                    play = score_screen(username, *game_data, highscore,
                                        username_two=username_two)
                else:
                    play = False
5           mpos = pygame.mouse.get_pos()
```

```
6       elif settings_button.get_clicked():
            settings_screen()
6           mpos = pygame.mouse.get_pos()

7       elif customise_button.get_clicked():
            player_character = db_get_character(username, db_cursor)
            customise(player_character, username)
            player_character = db_get_character(username, db_cursor)
            character_display.hex_to_grid(player_character)
7           mpos = pygame.mouse.get_pos()

8       elif leaderboard_button.get_clicked():
            leaderboards(username)
8           mpos = pygame.mouse.get_pos()

9       elif log_out_button.get_clicked():
            return

10      elif quit_button.get_clicked():
            quit()

11      one_player_button.draw(screen)
        two_player_button.draw(screen)
        settings_button.draw(screen)
        customise_button.draw(screen)
        leaderboard_button.draw(screen)
        quit_button.draw(screen)
        log_out_button.draw(screen)

        title_font_silver.render(screen, "UNTITLED"+NEW_LINE+"GUN GAME",
                                 (title_left, title_top), alignment=LEFT)
        title_font_white.render(screen, "N", (title_left + 14*PIXEL_RATIO, title_top))
        title_font_white.render(screen, "E", (title_left + 78*PIXEL_RATIO, title_top))
        title_font_white.render(screen, "A", (title_left + 62*PIXEL_RATIO,
                                              title_top + 26*PIXEL_RATIO))

        medium_font.render(screen, username, (18*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                           alignment=CENTER)
        character_display.draw(screen)

        leaderboard.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
11      clock.tick(FPS)
```

1 - The first thing I do is initialise all the different buttons (of which there are a lot) as well as the leaderboard and character display.
2 - In the while loop, I first get and remove all the events from the queue and put them into an event_list variable. I then check each event to see if I need to quit the game, register a button click, or to disable the mouse if the user presses any key.
3 - I then update all the buttons with the mouse position and the click boolean to update their images and 'clicked' attribute.
4 - The first button I check is the single player button. If this button has been pressed, I get the character and high score of the player to pass in as arguments to the game function. I then start a while loop that allows the player to keep playing the game without returning to the main menu. To do this I call the game function, which returns data about the game if the game was completed and None if the game was quit. Then, if data was returned, I call the score screen function, which returns True if the player wishes to keep playing and False otherwise. Once they have finished playing, I reset the mouse position and update the leaderboard.
5 - The second button I check is the two player button. If this button has been pressed, I get the user to enter the username and pin of the second player they are playing with using the login function which returns the username of the player if they successfully logged into and None if they pressed return. I then get the character of both players and the high score to pass in as arguments to the game function and start a similar while loop to the while loop of the single player game.
6 - The third button I check is the settings button. If this button has been pressed, I run the settings function which already handles actually updating the settings.
7 - The fourth button I check is the customise button. If this button has been pressed, I get the character of the player to pass in as an argument to the customisation function. This function handles updating the database, so once it has been exited I get the character of the player again from the database and update the character display.
8 - The fifth button I check is the leaderboards button. If this button has been pressed, I run the leaderboards function. Nothing can be changed in the leaderboard screen so nothing needs to be updated.
9 - The sixth button I check is the logout button. If this button has been pressed, I return from the function to go back to the open screen.
10 - The final button I check is the quit button. If this button has been pressed, I run the quit function.
11 - After checking all the buttons I draw everything to the screen including the title with its hidden message.

The main menu function operates similarly to all other functions in the main.py file. It starts with object initialisation and then a while loop. In the loop, the objects are updated, then they are checked for what has happened to them and if any buttons have been pressed, and finally everything is drawn to the screen. I could've done the menus as classes with everything outside the while loops in the initialising functions and everything inside while loops as update functions that are called every frame and but there was just no point as they would be extremely specific to the implementation and only created once. Using functions also allows me to return the previous menu screen by simply returning.

The in-game main menu screen:

3.13 Creating Accounts Function

This is the function from the main.py file that handles creating accounts.

```
def create_account():
1   taken_names = db_get_all_usernames(db_cursor)

2   name_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 5*EIGHT_PIXELS),
                       (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                       medium_font, small_font, 4, name="NAME",
                       not_allowed_strings=taken_names, allowed_characters=UPPER_ALPHABET)
    pin_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 68*PIXEL_RATIO),
                      (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                      medium_font, small_font, 4, name="PIN",
                      allowed_characters=NUMBERS, hide=True)
    create_button = TextButton((SCREEN_WIDTH//2 - 6*EIGHT_PIXELS//2, 12*EIGHT_PIXELS),
                               (6*EIGHT_PIXELS, 16*PIXEL_RATIO),
                               "CREATE", medium_font, disabled=True)
    return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
2                                (2*EIGHT_PIXELS, 2*EIGHT_PIXELS), return_image)

    mpos = pygame.mouse.get_pos()
    display_mouse = True
    while True:
3       click = False
        event_list = pygame.event.get()
        for event in event_list:
            if event.type == pygame.QUIT:
                quit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    click = True
            elif event.type == pygame.KEYDOWN:
3               display_mouse = False

        screen.fill(BACKGROUND_COLOUR)

        previous_mx_my = (mpos)
        mpos = pygame.mouse.get_pos()
        if (mpos) != previous_mx_my:
            display_mouse = True

4       name_box.update(mpos, click, event_list)
        pin_box.update(mpos, click, event_list)
        create_button.update(mpos, click)
4       return_button.update(mpos, click)

5       if name_box.get_valid() and pin_box.get_valid():
            create_button.set_disabled(False)
        else:
5           create_button.set_disabled(True)
```

```
6       if create_button.get_clicked():
            db_insert_player(name_box.get_text(), pin_box.get_text(), db_cursor, db_connection)
            return name_box.get_text()
        elif return_button.get_clicked():
6           return None

7       medium_font.render(screen, "CREATE ACCOUNT:",
                           (SCREEN_WIDTH//2, 3*EIGHT_PIXELS), alignment=CENTER)
        name_box.draw(screen)
        pin_box.draw(screen)
        create_button.draw(screen)
        return_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
           screen.blit(cursor_image, mpos)

        pygame.display.update()
7       clock.tick(FPS)
```

1 - The first thing I do is find out which accounts are already on the device by using a function I created from the database functions file to retrieve the name of every account. This list is then used as the not_allowed_strings list for the name text box, which means it will tell the user when they have typed in a taken name.

2 - I then initialise all the textboxes and the return button, making it so that the name box only accepts characters and the pin box only accepts numbers.

3 - In the while loop, I first get and remove all the events from the queue and put them into an event_list variable. I then check each event to see if I need to quit the game, register a button click, or to disable the mouse if the user presses any key.

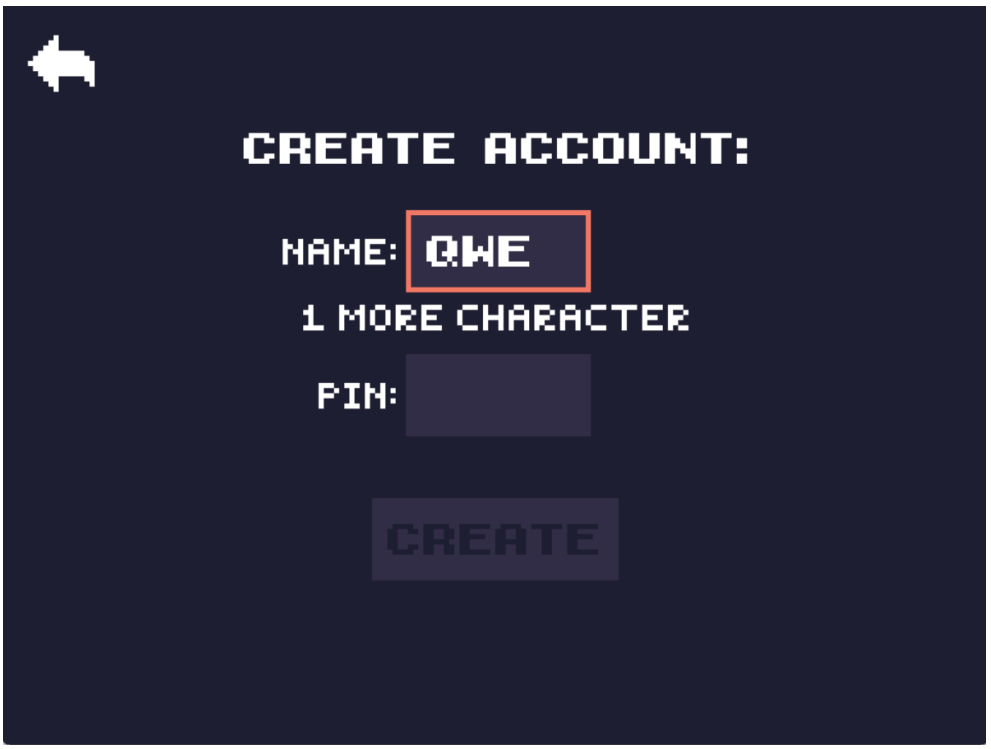
4 - I first update the text boxes and buttons with click as a parameter, which, in the case of the text boxes, also take in event_list as a parameter to check if the user typed anything and, in the case of the buttons, update their image and their 'clicked' attribute which is True the button has been clicked in the current frame.

5 - I then test if both text boxes have valid inputs, in which case the create account button can be enabled.

6 - I then check all the buttons to see if their 'clicked' attribute has been set to True, meaning they've been clicked on, and then perform the action they should perform if this is the case. For the create account button, this means creating a new user in the database, and so I run the db_insert_player function from the database functions to insert a new player with the typed in name and pin. I then return the username of the account so that the open screen function can then load the main menu with the username. For the return button, this means returning None from the function to show that no account was created.

7 - I finally draw everything, including a text heading for the screen, and then update the screen and tick the clock.

The in-game create account screen:



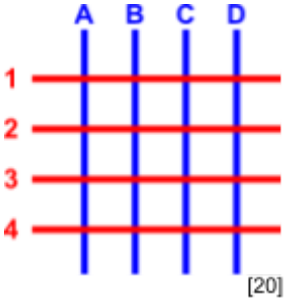
3.14 Two Player and Two Player Issues

The way I decided to implement the two player gamemode was not to create any new classes but just add parameters to both the player class and the game class to specify if the game is two player or not. I did this because most of the two player mode functions the same, just with changes in specific cases, so no redundant code is required. Some functions required a new two player version and some just required an if statement to change their functionality based on the number of players. I also reused the login function for letting the second player log into their account, just changing the title it displays. I haven't displayed code here as the code for two player is spread out across classes which can be seen in section 3.7 Player Class and the source code.

The main focus of this section though is the issues that I have come into when creating the two player game. My first idea for the controls of the players was to still use a similar movement system, using 4 buttons to move in 8 directions, but with two buttons to shoot, allowing you to shoot backwards or forwards. The controls for player 1 were WASD to move around, C to shoot backwards (opposite to the direction you are facing), and V to shoot forwards (in the direction you are facing). For player 2 the controls were the arrow keys to move around, P to shoot backwards, and O to shoot forwards. I implemented this and did a lot of testing with friends and found that it was way too hard. It did not feel natural at all, was really hard to get your head around and get used to, and didn't feel like you were in much control of your character. It is very hard to kill enemies if you have to be moving either directly towards or directly away from them, often leading to annoying deaths. There were also occasional issues with registering inputs but, at this point, I was focused on making a good control system and ignored it. This control system needed to change so I decided to return to a less complicated control system with both players being able to move and shoot in 8 directions.

The controls for player 1 are WASD to move around and GVBN to shoot. The controls for player 2 are 9IOP to move around and the arrow keys to shoot. Implementing this certainly made it a lot easier to play two player when controlling only one player but it did not come without issues. The first is that the keyboard does get a little crammed when playing with another person, but the second much more major issue is there were consistent issues where pressing buttons wouldn't be registered at all or wouldn't stop being registered until all other buttons were released. At first I didn't know what the problem was and assumed it was an issue with my code, so I spent a very long time looking at every possible thing it could be and couldn't find anything. I was also trying to research the issue at the same time but it is hard to research an issue when you can't really define it. This was very frustrating and I then tried a test to see if it was an issue with the keyboard inputs and not my code. I made a simple program that just displays what keys are being registered in each frame and I experimented on it with pressing lots of buttons simultaneously, finding that the inputs were indeed resulting in different keys being pressed, not being registered at all, or not stopping being registered until all other keys were released. I then knew it was not an issue with my code and tried updating Pygame which unfortunately did not fix anything. I finally realised it may be a keyboard issue, so I researched how keyboards worked and found the exact problems I was experiencing[20][21]. This misregistration of inputs is purely a hardware issue and I believe it to be due to the way the keyboard matrix is created.

Instead of having different connections for every key on a keyboard and having a microprocessor with 100 inputs, most keyboards work by using a matrix of connections with a node for each key.



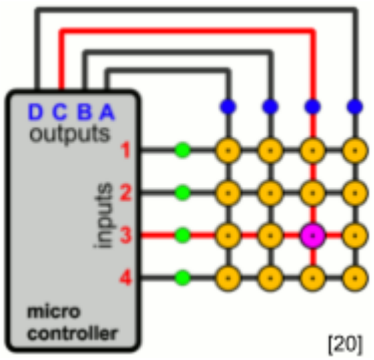
[20]

Here is a diagram of a 4 x 4 matrix with 4 columns and 4 rows, the columns being blue and the rows being red. There are 16 'knots' where the rows and columns intersect, which can each represent a key on a keyboard. When pressed down, these keys will connect the row to the column the key corresponds to and power is transferred between them.

These buttons can be labelled from A1 to D4.

The matrix of a keyboard will be much larger and more complex than these 4 x 4 diagrams but it is helpful to model the keyboard in this way as the systems and issues are the same when scaled up.

The keyboard works using the columns as outputs and the rows as inputs, powering each column in sequential order and registering if any rows return power. For example, if the microcontroller powers column C and the button at C3 is being pressed, row 3 will be powered. The microcontroller can now see that when column C is being powered, row 3 becomes powered, and therefore there must be a button pressed at C3.

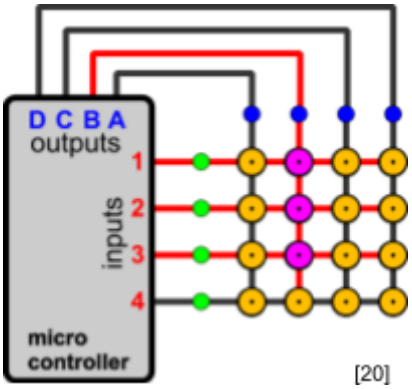


[20]

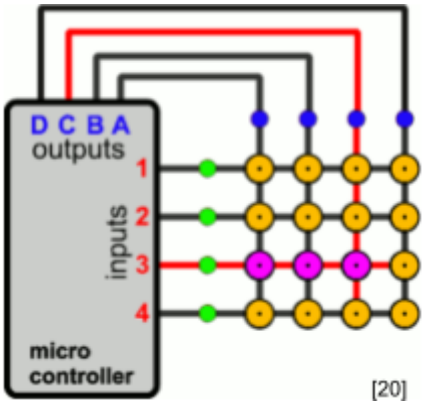
Here is a diagram of what was just explained. Column C is currently being powered and the key at C3 is currently being pressed, so row 3 is powered.

The columns do not get powered at the same time and alternate in sequential order at extremely high speeds.

This system comes into issues when multiple buttons are pressed. It works in a lot of situations, for example if the keys B1, B2, and B3 or C3, B3, and A3 are pressed simultaneously, as shown below:



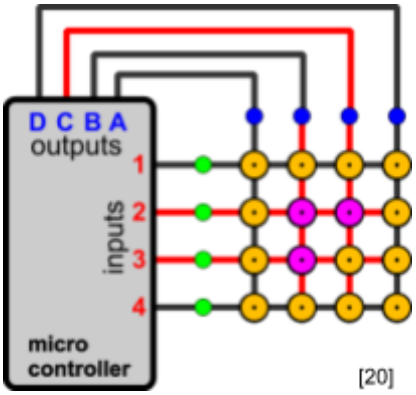
[20]



[20]

The microcontroller can handle these situations and as long as the software can too, everything works fine. However, there are two common problems that can occur, called the ghosting problem and the masking problem.

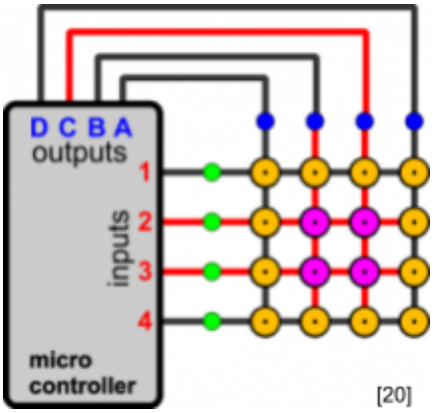
In this example, column C is currently powered and keys C2, B2, and B3 are being pressed.



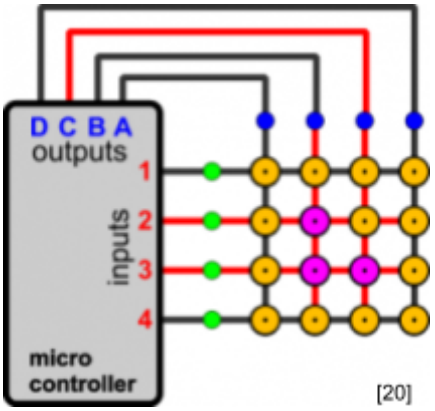
[20]

Because column C is currently powered and C2 is pressed down, row 2 becomes powered. Because row 2 is powered and B2 is being pressed down, column B becomes powered. Because column B is powered and B3 is being pressed down, row 3 becomes powered.
From the microcontrollers point of view, column C is being powered and row 2 and 3 have become powered. Therefore, it thinks that C2 and C3 are being pressed, however, C3 is not actually being pressed. This is called ghosting and is a big problem when lots of keys are being pressed.

The masking problem is a continuation of the ghosting problem. Once the previous situation has occurred, where C2, B2, and B3 are pressed simultaneously, and C3 is pressed down as well, no changes occur. The microprocessor cannot detect that C3 has been pressed. And then, if C2 is released, no changes occur. The microprocessor cannot detect that C2 has been released.



[20]



[20]

One of the ways modern computers avoid ghosting and masking is by ignoring keys that might result in it. So when they detect that ghosting may occur, instead of registering a fourth key, the computer will ignore the third key, called jamming. This does stop incorrect key presses being registered but at the cost of ignoring lots of button presses.
All of these problems make the game feel really unresponsive and are very annoying. The only ways to fix these problems is to either rearrange the controls and require less button presses or buy a very expensive keyboard where this problem is prevented. I won't be doing the latter and I have already tried having reduced controls which did not produce a good control system, so, I have left the two player implementation as is. I still got to create a two player version though, experiencing the challenges and improving my skills, so I'm not too disappointed. Also, it would theoretically work perfectly on better hardware, which I can still be proud of. It did however make it very hard to test the two player version and its systems as it is very annoying and very difficult to play, so I was never able to do much tweaking or perfecting.

3.15 Setting the Difficulty

One great difficulty with creating a game is, ironically, setting the difficulty of the game. This is due to many factors including the sheer number of things that can be altered.

Here is a list of some of the things that can be altered to change the difficulty of the game:

- the initial number of lives of the player
- the chance for items to spawn
- the length of effect of items
- the extent of effect of items
- the time before items despawn
- the speeds of enemies
- the health of enemies
- the movement patterns of enemies
- the time between enemies spawning
- the way enemies spawn
- the number of enemies that can spawn simultaneously
- the equation to calculate the total score of each wave of enemies
- the score of each enemy
- the speed of the player
- the rate of fire of the player
- the speed of the player's bullets
- the time between crates spawning
- the leeway in detecting collisions between the player and enemies

As you can see, there are a lot of things that can be changed, some having small effects and some having big effects. Due to this, a lot of these have remained the same since their implementation when I gave them values that looked good

Another big factor in why it is so hard to get the difficulty right is due to the limitations of play testers. With actual game releases, the game will be tested by paid testers who understand the project and can give detailed feedback on what could change. I do not have the facilities or budget for that so I have to rely on my own play testing and the limited feedback I can receive from my friends. One of the big problems with using yourself as a play tester is that you will be very good at your own game as you have been playing almost daily for a long period of time. This means a change in difficulty that you find fun may make it a very unfriendly and difficult game for other people who have not been playing daily for a long period of time.

A final factor is that different people enjoy different levels of difficulty. Some people love when a game makes them feel weak and presents them with challenges they have to work hard to overcome and some people hate that, so, even if someone tests your game and they think it is perfect and the best game ever, another person could have a very different opinion.

All of this makes it so hard to find the right balance so ultimately, you have to accept it will never be perfect. It is still very possible to test and get feedback and I have made many changes over the course of development, tweaking enemies if I see that they become too difficult, tweaking time delays if I think they would benefit from a change, tweaking items if I think it would make the game more fun, however, you could spend a very very long time tweaking and making minor improvements and it gets to a point where it is no longer worth it. You just have to stop when you're happy with how the game feels, which is what I did.

3.16 Extra Features

Throughout the creation of the project, there were small things I realised I could add to improve the project that I had missed in the design stage or didn't know would be possible.
One of these is the Score class. For a long time during the development of the game the enemies would just disappear when they died and it felt a little boring and unsatisfying. I thought of two ways to solve this, either by creating animations for the deaths of each enemy or by adding a little pop-up after you kill an enemy that shows the score you just got from killing them. The second option not only requires less art creation but also has the benefit of giving the player a sense of progress and achievement as they can dynamically see how much their score is increasing by, so I went for this option. It wasn't too hard to implement, requiring a new Score class and a new list in the Game class that Score objects get added to once an enemy has been killed and then get removed from shortly after. The Score class is displayed below:

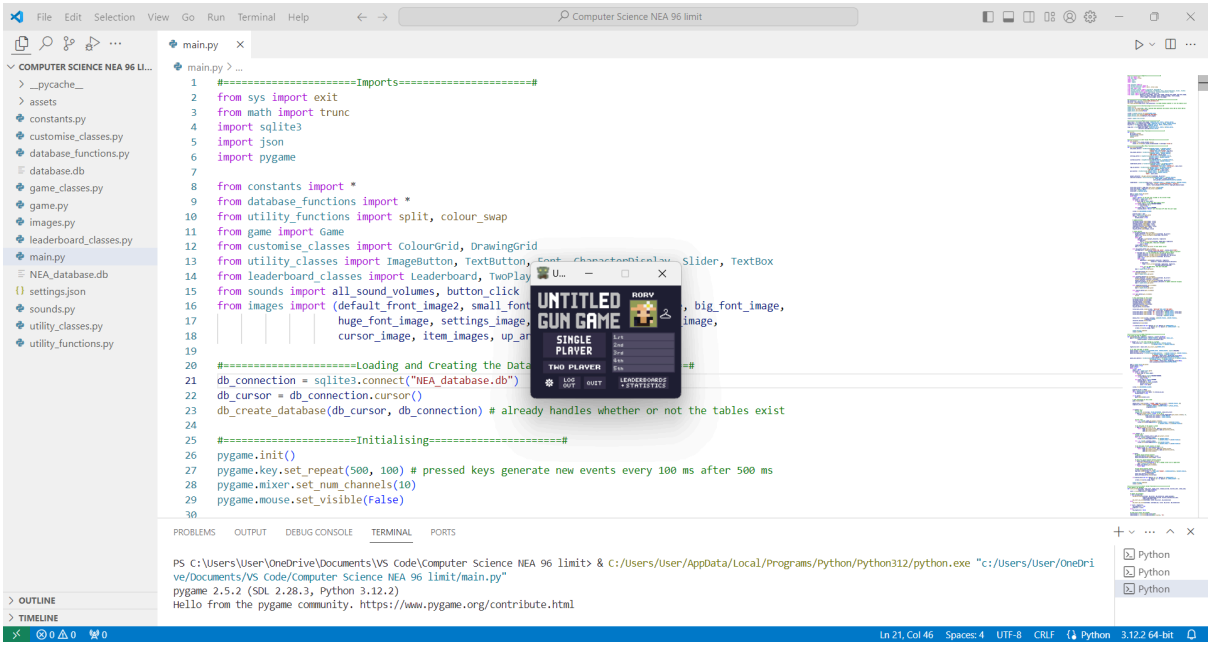
```
class Score():
    def __init__(self, font, colour, score, rect, alpha=255):
        self.__font = font.new_colour_copy(colour, alpha=alpha)
        self.__score = str(score)
        self.__rect = rect
        self.__timer = 0

    def update(self): # check how long the score has been around for
        self.__timer += 1
        if self.__timer > SCORE_LENGTH:
            return True # true returned if the score should be removed from the list
        return False

    def draw(self, screen): # draw the score
        self.__font.render(screen, self.__score, (self.__rect.centerx,
                                                   self.__rect.top + PIXEL_RATIO),
                                                   alignment=CENTER)
```

This class features an update method and a draw method. It is initialised with a font, score, rect, and timer. The update function updates the score's timer and returns True if the score has lasted the length it is meant to last for and False otherwise. The draw function simply renders the score's score in its font at its position.

Another extra feature I didn't realise that I could add until late in development was a new settings option to change the size of the game window. From the start of development, I have based all distances and lengths and sizes and speeds off of one constant, the pixel ratio, which defines how many pixels the screen uses for one pixel of the game. Despite having the scalability of the game window in mind when I created this, it took me a long time to realise it might be a fun setting to add to the settings menu. Adding this feature wasn't too difficult and just required me to move the loading of settings from the main file to the constants file so that the pixel ratio could be loaded with the value from the settings. Changing the size of the window does require restarting the game but I think it is a very fun feature to play around with as everything still works exactly the same with each screen size and the game can still be played. It also allows for consistency changes across different

displays as Pygame works in terms of your display's pixels, so playing on a lower resolution screen results in a greater size of the game window

A screenshot with the size set to 1 (1 to 1 ratio between screen pixels and game pixels):



4. Testing

In this section I will be testing the project to demonstrate that it works as intended. I created testing videos that can be found from the URL below and a table to show the completeness of each objective. This table also includes which testing video the objective is featured in and the related sections of this document where the objective is explored or the solution is explained.

The testing video site URL: **redacted**

Main Objective	Sub-Objective	Related sections	Level of Completeness	Related Testing Video
Upon opening the game, the user will be greeted with a login screen	The initial screen allows the user to either create a new account or log into a pre-existing account through a username and a pin	2.4 2.8	Fully complete	Creating Accounts
	If, when creating an account, the username is already taken or, when logging in, the username is not recognised or the pin is incorrect, the user will be informed	3.13	Fully complete	
Once logged in, there is a simply designed and understandable main menu with buttons to access each important feature of the program	The game needs to make it easy to get right in and start playing, so the single player button will be accentuated	2.4	Fully complete	All videos
	There will be a smaller leaderboard on the main menu that it isn't blocked by a button so the user can quickly check the leaderboard before starting a game	2.4	Fully complete	
	A small image of the user's custom character will be visible	2.4	Fully complete	
The user can play single player by clicking the single player button	The user can move and shoot from their character in 8 directions	1.9.1 3.7	Fully complete	Game Walkthrough / Game Playthrough
	Hordes of enemies randomly spawn from one of four sides	3.9	Fully complete	
	As the user progresses in the game, the enemies that spawn can be of harder types	2.10.1 2.11.3 3.9	Fully complete	
	After every few hordes of enemies, a small obstacle spawns at a random location that blocks both the player and enemies	2.2 2.11.4 2.11.5 3.9	Fully complete	

	The user is able to store 1 item and use it at any point in the game and if the user runs into an item when their item spot is already filled it is automatically used	2.2	Fully complete	
	Killed enemies have a random chance of dropping items	3.10	Fully complete	
	Lots of different item types that all do very different things to help the player	2.10.2	Fully complete	
	The game can be paused	2.4	Fully complete	
	When the player has run out of lives and the game ends, the user's score is displayed as well as an updated leaderboard	2.4	Fully complete	
	The user can either return to the menu or play again through buttons	2.3 2.4	Fully complete	
There is a two player game that allows two user accounts to play with each other and have combined scores	Both players can still move and shoot in 8 directions	3.14	As complete as is possible	Two Player
	Enemy spawning is more frequent than in the single player game	3.14	Fully complete	
The user can view the leaderboards by clicking the leaderboard button	The user can switch between viewing the single player leaderboard, two player leaderboard, statistics leaderboard, and personal statistics	2.3 2.4	Fully complete	Leaderboards and Statistics
	The single player leaderboard tab displays, in descending order, the top 10 single player scores and the username of the player who achieved it	2.4 2.8.2 3.11.1	Fully complete	
	The two player leaderboard tab displays, in descending order, the top 10 two player scores and the usernames of the players who achieved it	2.4 2.8.2 3.11.2	Fully complete	
	The statistics leaderboard tab displays the top 3 players in one or many statistic categories and a display of their custom character	2.4 2.8.2	Fully complete	

	The personal statistics tab lets the user see their personal statistics such as enemies killed, games played, items used, bullets shot etc.	2.4 2.8.2	Fully complete	
	The user's personal scores are highlighted if they are on the leaderboard	3.11	Fully complete	
The user can view and customise their character by clicking the customise character button	The user can completely customise their character, being able to draw some aspect of the character using a simple drawing application	2.2 2.4 2.11.1 3.6	Fully complete	Customising a Character
	Some colours of this drawing application are initially locked until unlocked through achievements, e.g. killing 500 enemies	2.2 2.4	Fully complete	Leaderboards and Statistics
	Once created, the user can save their character, putting the character into a useable format and updating their character on the database and leaderboards	2.4 2.8.2 2.11.1	Fully complete	Customising a Character
The user can view and change the settings by clicking the settings button	The user can change the volume of the game	2.4 3.16	Fully complete	Changing Settings
The game is consistent and accessible throughout	Everything in the game is in a neat, consistent, and accessible art style	1.9.1 2.4	Fully complete	All videos
	Plenty of sounds to provide audio feedback like when you press a button, kill an enemy, pickup an item, and lose a life	2.10.4	Fully complete	
	Plenty of visual cues to provide visual feedback like when you hover over a button, hit an enemy, or use an item	3.8.1	Fully complete	

5. Evaluation

5.1 Objective Fulfilment

Objective 1 - The login screen:

- I have successfully created a functioning menu to allow players to create accounts and log into existing accounts
- When a player creates an account, it correctly inserts a new record into the database
- When the player tries to login, it correctly searches for the username and pin in the database and if there is no match found it correctly informs the user of their error

Objective 2 - The main menu:

- I have successfully created an intuitive main menu that allows easy access to all necessary parts of the application with the most important buttons being accentuated
- The main menu has a correctly displaying leaderboard that shows the top 5 single player scores and a small display of your custom character
- All of the buttons of the main menu clearly show what they lead to and do indeed lead to those things when pressed

Objective 3 - The single player game:

- I have successfully created a controllable player that can move and shoot in 8 directions
- As the user plays, hordes of enemies successfully spawn from the sides of the screen and move towards the player with lots of types of enemies that can spawn
- After every few hordes of enemies, a crate successfully spawns that blocks enemies, players, and bullets from moving through it
- When an enemy dies, there is a chance an item can drop and there are lots of item types that could drop, all functioning and helping the player in unique and fun ways
- The player can successfully store one item to use at any time
- The player can press the escape key to successfully pause the game and bring up a pause menu, allowing access to the settings screen and a button to quit the game
- When an enemy touches the player, the player loses a life and all enemies on screen die
- When the player runs out of lives, the game ends and correctly displays the players score along with the statistics of game just played and allows the option to play again or return to the main menu
- The database is successfully updated to include the score achieved by the player and the player's statistics

Objective 4 - The two player game:

- I have somewhat successfully created a functioning two player version of the single player game, the only errors that come across in the two player version are due to hardware limitations and would be very difficult to find a solution for
- Two player specific features function as they should e.g. when one player has lost all their lives and the other has lives to spare they can steal one, items are used instantly on the correct player when a player touches them
- Enemy spawning is indeed more frequent and threatening
- The database is successfully updated to include the score achieved by the players

Objective 5 - The leaderboard screen:

- I have successfully created a functioning leaderboard screen and menu that allows players to look at how their scores compare to other players' scores

- The single player tab correctly shows the top 10 scores achieved by players, along with showing the username and custom character of the player who achieved it
- The two player tab correctly shows the top 10 scores achieved by players in two player games, displaying the name of both players
- Scores that the player's account achieved or were part of for two player are highlighted on the leaderboards
- The podiums tab correctly shows the top 3 players in both games played and enemies killed in a comprehensive and engaging format
- The statistics tab correctly tells the player their account's statistics including games played, enemies killed, bullets shot, and items used

Objective 6 - Character customisation:

- I have successfully created a screen that allows players to make their character completely unique and make them stand out on leaderboards
- There are technically $(8 * 3)^9 * 4 * 4 * 4 * 4 * 2 * 2$ or $2.7052109x10^{15}$ possible characters, most of these however would be completely nonsensical characters with dots for hats but it is still an impressive number of unique combinations
- The player can draw their own hat using a functioning drawing menu with useful buttons as well as customise the body of their character by changing the colours of all aspects
- The option to change the eye colour and the option to change the gun colour are locked behind working achievements, requiring the player to kill 500 enemies to unlock the eye colour and shoot 1000 bullets to unlock the gun colour
- The database is successfully updated to include the player's custom character if it is saved
- The game successfully allows the player to play in the game as their custom character by creating art based upon the custom character for each direction they can face

Objective 7 - The settings screen:

- I have successfully created an intuitive settings screen with sliders for each setting
- The player can change the volume of every sound in the game using a slider as well as change the size of the game window
- These settings are successfully stored locally on a file and used for every account on a device

Objective 8 - Everything is accessible and in a consistent style:

- I have successfully created and used a consistent art style that I believe to be intuitive and simple to comprehend
- I have successfully added visual feedback to lots of different aspects of the program
- I have successfully added sound effects to lots of different aspects of the program

5.2 Client Feedback

Although I have been receiving relatively regular feedback from my client, Marcus, throughout the project, he wasn't familiar with the entire system in one collective application so for the first part of the feedback I simply gave him the project and watched as he explored it.
Whilst he was exploring the project I took some notes on what he did and said:

- He didn't struggle at all with creating an account and found it pretty intuitive
- He found the menus understandable and easy to navigate and he liked the art style
- He was a big fan of the character customisation and loved the freedom you are given with being able to draw your own hat, having a lot of fun with coming up with his own personal character
- He thought the leaderboards looked cool and was excited to see himself on the podium
- He enjoyed playing the game too and played for quite a while, finding it challenging but fun
- He found all the enemies interesting, would get excited when he came across a new one, and asked a lot of questions about how they worked
- He enjoyed the variety of items and loved that you can combine them to become very powerful
- He was disappointed by the two player mode

After talking to him about it, he seems really pleased with how the game turned out. The biggest issue raised was the two player issue but he understood that it was due to hardware constraints of mapping too many keys to a keyboard that can't handle it. He still said it was very cool that you could log in with a second account and have both players playing with their custom characters but he was understandably disappointed.
He mentioned the benefit that online multiplayer would have on the game, which I do agree with.
He also pointed out that the controls didn't display for long and could only be seen during the 3 second countdown, suggesting the idea of a controls screen.
Overall though, Marcus was really happy and impressed with what I managed to do and had a lot of fun playing the game.

5.3 Improvements

Reflecting on my clients feedback, the biggest improvement that could be made to the project would be a better two player mode. The current implementation is rather disappointing and an online system for multiplayer could've been really cool and really fun and I do love when games offer good multiplayer support. I know that I wasn't able to make anything better within the time constraints but if I were to return to the project with improvements in mind, the two player mode would be a big focus.

Another thing my client mentioned that could improve the project is a controls screen that allows users to see all the controls and get used to them before they start playing because it can be quite hard to read all the controls for the first time within the 3 seconds they are visible.

Another thing that I believe would improve the application would be security features, such as encryption of pins. Although the data stored on each account or progress in the game probably isn't significantly important to anyone, it would still be good to have security features implemented.

In the current implementation, if I were to create a new Python file, connect to the database, and run the following code:

```
print(db_cursor.execute("SELECT username, pin FROM Players").fetchall())
```

I would get this result:

```
[('TEST', '1234'), ('RORY', '0000')]
```

It is clear to see that this is not particularly secure and would be improved by some form of encryption.

Another thing that the game feels like it is missing, although it can be hard to notice, is background music. Background music can tie everything together and really amplify the feelings when playing a game, but, due to time constraints and not enough experience in music composition, this was not added. This is an area I'd like to improve on though and maybe someday I will come back to the project and compose some background music for it.

Finally, something small I completely overlooked was a way to tell the player how to unlock the two locked colours. It's not too important at this scale but if I were to make the project at a larger scale it would be necessary to have it implemented.

5.4 Overall Project Reflection

Overall, I am very happy with how my project turned out and my client is really pleased with it too. I found enough time and motivation to implement a lot of the things I could want to implement and to create something that feels mostly finished. I had a lot of fun, gained so many skills, learnt a lot, and discovered so much. My coding and coding style improved a lot over the course of this project. My method of approaching large scale projects improved a lot after realising the benefits that fully analysing a problem and designing or at least attempting to design lots of the systems before creating them have. My skills in game design and problem solving improved a lot and I found that I really enjoyed the process of creating a game and overcoming all the problems that come with it. My art skills also improved as I was required to create art for each enemy, item, and font. So, I am very proud of what I was able to achieve and all the things I learnt in the process.

6. Source Code

A lot of the source code has had to be formatted strangely to accord with the line character limit.
In my code I do use a lot of comments despite it being self-documenting because I value being able to read the comments as code summaries that can explain why things work without requiring much reading of the code.

6.1 Contents Page

6.2 Constants File

constants.py

```
from utility_functions import create_hex_dictionary, round_to_nearest
import json

#======================Loading Settings======================#
try:
    with open("settings.json", "r") as file:  # open the settings file in read mode
        settings = json.load(file)            # set the settings to the data in the save file
except FileNotFoundError:
    settings = {'volume' : 0.50, 'size' : 5}  # if no file, create new settings dictionary
    with open("settings.json", "w") as file:  # create new file and put the settings into it
        json.dump(settings, file)

#======================General Constants======================#
BACKGROUND_SIZE = (BACKGROUND_WIDTH, BACKGROUND_HEIGHT) = (192, 144) # 4:3
PIXEL_RATIO = settings['size']
SCREEN_SIZE = (SCREEN_WIDTH, SCREEN_HEIGHT) = (PIXEL_RATIO*BACKGROUND_WIDTH,
                                               PIXEL_RATIO*BACKGROUND_HEIGHT)
EIGHT_PIXELS = 8*PIXEL_RATIO # makes positioning easier to read, comprehend, and change
GAME_WIDTH = 16
GAME_HEIGHT = 16
FPS = 60

UP = 0
LEFT = 1
CENTER = DOWN = 2
RIGHT = 3

X = 0
Y = 1

#======================Player======================#
# all speeds are in terms of FPS and PIXEL_RATIO so will adjust with any changes
PLAYER_SPEED = 0.55 * PIXEL_RATIO * (60/FPS) # * 60 as the speed was found at 60FPS
PLAYER_LIVES = 3
PLAYER_DAMAGE = 1
FIRE_RATE = 0.25 * FPS
BULLET_SPEED = 1.25 * PIXEL_RATIO * (60/FPS)
BULLET_LIFETIME = 5 * FPS

#======================Enemies======================#
DEFAULT_ENEMY = {"HEALTH" : 1,
                 "SPEED"  : 0.24 * PIXEL_RATIO * (60/FPS),
                 "SCORE"  : 10}
FAST_ENEMY = {"HEALTH" : 2,
              "SPEED"  : 0.45 * PIXEL_RATIO * (60/FPS),
              "SCORE"  : 30}
FLYING_ENEMY = {"HEALTH" : 1,
```

```
                "SPEED"  : 0.38 * PIXEL_RATIO * (60/FPS),
                "SCORE"  : 20}
CROW_ENEMY = {"HEALTH" : 1,
              "SPEED"  : 1.2 * PIXEL_RATIO * (60/FPS),
              "SCORE"  : 30}
TOUGH_ENEMY = {"HEALTH" : 2,
               "SPEED"  : 0.22 * PIXEL_RATIO * (60/FPS),
               "SCORE"  : 20}
SPIRIT_ENEMY = {"HEALTH" : 3,
                "SPEED"  : 0.28 * PIXEL_RATIO * (60/FPS),
                "SCORE"  : 40}

CROW_PAUSE = 2*FPS
CROW_BLUR_DISTANCE = 2*PIXEL_RATIO
CROW_OFFSCREEN = -5

HIT_TIME = int(0.1 * FPS)
HIT_COLOUR = (255,0,0,100)
SCORE_LENGTH = 0.25*FPS

#======================Enemy Spawning======================#
INDEX_MULTIPLIER = 2.5
DELAY_MULTIPLIER = 1.6
MIN_FRAMES = 12

#======================Items======================#
BOMB = 0
SHOES = 1
RAPID_FIRE = 2
SHOTGUN = 3
TIME_FREEZE = 4
BACKWARDS_SHOT = 5
HEART = 6

SHOES_LENGTH = 8*FPS
RAPID_FIRE_LENGTH = 8*FPS
SHOTGUN_LENGTH = 8*FPS
TIME_FREEZE_LENGTH = 4*FPS
BACKWARDS_SHOT_LENGTH = 8*FPS

ITEM_CHANCE_1P = 10  # one in
ITEM_CHANCE_2P = 8
ITEM_TIME = 6 * FPS
ITEM_COUNTDOWN = FPS//2
ITEM_FLASH_TIME = round_to_nearest(2 * FPS, 9)
SHOE_MULTIPLIER = 1.4
RAPID_FIRE_MULTIPLIER = 0.7**2
SHOTGUN_RATE_MULTIPLIER = 1/0.7
SCREEN_SHAKE_LENGTH = 0.4*FPS
BOMB_RANGE = 7*EIGHT_PIXELS
```

```
#======================Achievements======================#
ENEMY_ACHIEVEMENT = 500    # to unlock special eye colour
BULLET_ACHIEVEMENT = 1000  # to unlock special gun colour

#======================Other======================#
BUTTON_PRESSED_DELAY = 2 * FPS

FENCE_LIST = [(0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(0,10),(0,11),(0,12),(0,13),(0,14),(0,15),
              (1,0),(1,15),(2,0),(2,15),(3,0),(3,15),(4,0),(4,15),(5,0),(5,15),
              (10,0),(10,15),(11,0),(11,15),(12,0),(12,15),(13,0),(13,15),(14,0),(14,15),
              (15,0),(15,1),(15,2),(15,3),(15,4),(15,5),(15,10),(15,11),(15,12),(15,13),
              (15,14),(15,15)]

#======================Colours======================#
WHITE = (255,255,255)
CUSTOMISATION_GREY = (220,220,220)
SLIGHT_GREY = (206,205,210)
GREY = (128,128,128)
PALER_GREY = (140,140,140)

BACKGROUND_COLOUR = (34,32,52)
PALISH_BACKGROUND = (50,47,70)
PALISHER_BACKGROUND = (60,57,80)
PALER_BACKGROUND = (70,67,90)
PALER_BACKGROUND2 = (80,77,100)
PALER_BACKGROUND3 = (100,97,120)
EVEN_PALER_BACKGROUND = (110,107,130)
PALEST_BACKGROUND = (160,157,163)

GRASS_GREEN = (146,156,89)
DARK_GREEN = (75,105,47)

SPIRIT_ENEMY_COLOUR = (25,0,25,180)

GOLD = (213,176,56)
SILVER = (192,192,192)
BRONZE = (140,103,33)#(149,112,42)

RED = (240,120,100)#(200,120,100)
AMBER = (220,180,100)
GREEN = (140,200,100)

PAUSE_COLOUR = (128, 128, 128, 110)
PAUSE_BUTTON_HOVER = (128, 128, 128, 55)

MINOR_TEXT = SLIGHT_GREY
SCORE_ALPHA = 235

LOCKED_COLOUR = GREY

BUTTON_ICON_COLOUR = WHITE
```

```
BUTTON_ICON_PRESSED_COLOUR = SLIGHT_GREY
BUTTON_BACKGROUND_COLOUR = BACKGROUND_COLOUR
BUTTON_HOVER_COLOUR = PALISH_BACKGROUND
BUTTON_DISABLED_COLOUR = PALISH_BACKGROUND

BUTTON_TEXT_COLOUR = WHITE
BUTTON_TEXT_PRESSED_COLOUR = SLIGHT_GREY
BUTTON_TEXT_DISABLED_COLOUR = BACKGROUND_COLOUR
TEXT_BUTTON_BACKGROUND_COLOUR = PALISH_BACKGROUND
TEXT_BUTTON_HOVER_COLOUR = PALISHER_BACKGROUND
TEXT_BOX_BACKGROUND = PALISH_BACKGROUND

CHARACTER_DISPLAY_BORDER = PALER_BACKGROUND

LEADERBOARD_POSITION_COLOUR = SLIGHT_GREY
LEADERBOARD_DEFAULT_COLOUR = PALER_BACKGROUND2
LEADERBOARD_HIGHLIGHT_COLOUR = PALER_BACKGROUND3

#======================Customisation======================#
HAT_COLOURS = [(237,179,120), (87,55,43), (172,50,50), (50,112,139),
               (232,223,50), (240,240,240), (50,45,50), (20,20,20)]
SKIN_COLOURS = [(238,195,154), (192,140,91), (141,85,36), (87,54,27)]
JACKET_COLOURS = [(102,57,49), (19,16,28), (13,84,104), (37,73,10)]
SHIRT_COLOURS = [(207,150,92), (220,220,220), (223,196,13), (40,40,40)]
TROUSER_COLOURS = [(50,60,57), (23,23,27), (16,52,75), (41,27,21)]
EYE_COLOURS = [(55,32,50), (250,245,127)]
GUN_COLOURS = [(34,32,52),(190,150,0)]
ALL_COLOURS = ([None] + HAT_COLOURS + SKIN_COLOURS + JACKET_COLOURS + SHIRT_COLOURS +
                                                TROUSER_COLOURS + EYE_COLOURS + GUN_COLOURS)

BITMAP_DICTIONARY, COLOUR_DEPTH = create_hex_dictionary(ALL_COLOURS)

# the pixel coordinates for each changeable aspect of the character in each direction
SKIN_FRONT = [(2,2),(2,3),(2,4),(2,5),(3,3),(3,4),(4,2),(4,3),(4,4),(4,5),(6,6)]
JACKET_FRONT = [(5,1),(5,2),(5,5),(5,6),(6,2),(6,5)]
SHIRT_FRONT = [(5,3),(5,4),(6,3),(6,4)]
TROUSER_FRONT = [(7,2),(7,3),(7,4),(7,5)]
EYE_FRONT =  [(3,2),(3,5)]
GUN_FRONT = [(6,1),(7,1)]
FRONT_PATTERNS = [SKIN_FRONT, JACKET_FRONT, SHIRT_FRONT, TROUSER_FRONT, EYE_FRONT, GUN_FRONT]
SKIN_BACK = [(2,2),(2,3),(2,4),(2,5),(3,2),(3,3),(3,4),
             (3,5),(4,2),(4,3),(4,4),(4,5),(6,1),(6,6)]
JACKET_BACK = [(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(6,2),(6,3),(6,4),(6,5)]
SHIRT_BACK = []
TROUSER_BACK = [(7,2),(7,3),(7,4),(7,5)]
EYE_BACK = []
GUN_BACK = [(4,6)]
BACK_PATTERNS = [SKIN_BACK, JACKET_BACK, SHIRT_BACK, TROUSER_BACK, EYE_BACK, GUN_BACK]
SKIN_LEFT = [(2,2),(2,3),(2,4),(2,5),(3,2),(3,4),(3,5),(4,2),(4,3),(4,4),(4,5),(6,4)]
JACKET_LEFT = [(5,2),(5,4),(5,5),(6,2),(6,5)]
SHIRT_LEFT = [(5,3),(6,3)]
```

```
TROUSER_LEFT = [(7,2),(7,3),(7,4),(7,5)]
EYE_LEFT = [(3,3)]
GUN_LEFT = [(5,0),(5,1)]
LEFT_PATTERNS = [SKIN_LEFT, JACKET_LEFT, SHIRT_LEFT, TROUSER_LEFT, EYE_LEFT, GUN_LEFT]
SKIN_RIGHT = [(2,2),(2,3),(2,4),(2,5),(3,2),(3,3),(3,5),(4,2),(4,3),(4,4),(4,5),(6,3)]
JACKET_RIGHT = [(5,2),(5,3),(6,2),(6,5)]
SHIRT_RIGHT = []
TROUSER_RIGHT = [(7,2),(7,3),(7,4),(7,5)]
EYE_RIGHT = [(3,4)]
GUN_RIGHT = [(5,4),(5,5),(5,6),(6,4)]
RIGHT_PATTERNS = [SKIN_RIGHT, JACKET_RIGHT, SHIRT_RIGHT, TROUSER_RIGHT, EYE_RIGHT, GUN_RIGHT]

DEFAULT_HEX = ("09"+"0d"+"11"+"15"+"19"+"1b"             # 6 colours for the body parts
               +"00"+"00"+"00"+"00"+"00"+"00"+"00"+"00"  # 3 rows of 8 blank pixels for the hat
               +"00"+"00"+"00"+"00"+"00"+"00"+"00"+"00"
               +"00"+"00"+"00"+"00"+"00"+"00"+"00"+"00")

#======================Character Sets======================#
UPPER_ALPHABET = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O',
                  'P','Q','R','S','T','U','V','W','X','Y','Z']
LOWER_ALPHABET = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
                  'p','q','r','s','t','u','v','w','x','y','z']
NUMBERS = ['0','1','2','3','4','5','6','7','8','9']
SPECIAL_CHARACTERS = ['.',':','+','?','!','&','hidden','|',"'","%"]
CONTROLS = ["^","_","<",">",";"]
CHARACTER_LIST_U = UPPER_ALPHABET + NUMBERS + SPECIAL_CHARACTERS
CHARACTER_LIST = UPPER_ALPHABET + LOWER_ALPHABET + NUMBERS + SPECIAL_CHARACTERS + CONTROLS
NEW_LINE = '/'
```

6.3 Images File

images.py

```
from pygame import display, image, transform, mask

# PIXEL_RATIO shortened to fit onto page
from constants import (PIXEL_RATIO as PR, SCREEN_SIZE, BACKGROUND_COLOUR,
                       SPIRIT_ENEMY_COLOUR, WHITE)
from utility_functions import colour_swap

# display must be initialised and a video mode set for image loading and manipulation
display.init()
display.set_mode(SCREEN_SIZE)

# all images are scaled up by PR to ensure a consistent pixel size

#======================Enemies======================#
default_back_image1 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_back1.png").convert_alpha(), PR)
default_back_image2 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_back2.png").convert_alpha(), PR)
default_back_image3 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_back3.png").convert_alpha(), PR)
default_back_image4 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_back4.png").convert_alpha(), PR)
default_left_image1 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_left1.png").convert_alpha(), PR)
default_left_image2 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_left2.png").convert_alpha(), PR)
default_left_image3 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_left3.png").convert_alpha(), PR)
default_left_image4 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_left4.png").convert_alpha(), PR)
default_front_image1 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_front1.png").convert_alpha(), PR)
default_front_image2 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_front2.png").convert_alpha(), PR)
default_front_image3 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_front3.png").convert_alpha(), PR)
default_front_image4 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_front4.png").convert_alpha(), PR)
default_right_image1 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_right1.png").convert_alpha(), PR)
default_right_image2 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_right2.png").convert_alpha(), PR)
default_right_image3 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_right3.png").convert_alpha(), PR)
default_right_image4 = transform.scale_by(
    image.load("assets/images/enemies/default enemy/goblin_right4.png").convert_alpha(), PR)
default_enemy_images = [
```

```
    [default_back_image1,default_back_image2,default_back_image3,default_back_image4],
    [default_left_image1,default_left_image2,default_left_image3,default_left_image4],
    [default_front_image1,default_front_image2,default_front_image3,default_front_image4],
    [default_right_image1,default_right_image2,default_right_image3,default_right_image4]]

fast_up_image1 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_up1.png").convert_alpha(), PR)
fast_up_image2 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_up2.png").convert_alpha(), PR)
fast_left_image1 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_left1.png").convert_alpha(), PR)
fast_left_image2 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_left2.png").convert_alpha(), PR)
fast_down_image1 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_down1.png").convert_alpha(), PR)
fast_down_image2 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_down2.png").convert_alpha(), PR)
fast_right_image1 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_right1.png").convert_alpha(), PR)
fast_right_image2 = transform.scale_by(
    image.load("assets/images/enemies/fast enemy/mushroom_right2.png").convert_alpha(), PR)
fast_enemy_images = [[fast_up_image1,fast_up_image2],
                     [fast_left_image1,fast_left_image2],
                     [fast_down_image1,fast_down_image2],
                     [fast_right_image1,fast_right_image2]]

flying_up_image1 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_up1.png").convert_alpha(), PR)
flying_up_image2 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_up2.png").convert_alpha(), PR)
flying_left_image1 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_left1.png").convert_alpha(), PR)
flying_left_image2 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_left2.png").convert_alpha(), PR)
flying_down_image1 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_down1.png").convert_alpha(), PR)
flying_down_image2 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_down2.png").convert_alpha(), PR)
flying_right_image1 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_right1.png").convert_alpha(), PR)
flying_right_image2 = transform.scale_by(
    image.load("assets/images/enemies/flying enemy/flying_right2.png").convert_alpha(), PR)
flying_enemy_images = [[flying_up_image1,flying_up_image2],
                       [flying_left_image1,flying_left_image2],
                       [flying_down_image1,flying_down_image2],
                       [flying_right_image1,flying_right_image2]]

crow_up_image = transform.scale_by(
    image.load("assets/images/enemies/crow enemy/crow_enemy_up.png").convert_alpha(), PR)
crow_left_image = transform.scale_by(
    image.load("assets/images/enemies/crow enemy/crow_enemy_left.png").convert_alpha(), PR)
```

```
crow_down_image = transform.scale_by(
    image.load("assets/images/enemies/crow enemy/crow_enemy_down.png").convert_alpha(), PR)
crow_right_image = transform.scale_by(
    image.load("assets/images/enemies/crow enemy/crow_enemy_right.png").convert_alpha(), PR)
crow_enemy_images = [crow_up_image, crow_left_image, crow_down_image, crow_right_image]

tough_back_image1 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_back1.png").convert_alpha(), PR)
tough_back_image2 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_back2.png").convert_alpha(), PR)
tough_back_image3 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_back3.png").convert_alpha(), PR)
tough_back_image4 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_back4.png").convert_alpha(), PR)
tough_left_image1 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_left1.png").convert_alpha(), PR)
tough_left_image2 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_left2.png").convert_alpha(), PR)
tough_left_image3 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_left3.png").convert_alpha(), PR)
tough_left_image4 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_left4.png").convert_alpha(), PR)
tough_front_image1 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_front1.png").convert_alpha(), PR)
tough_front_image2 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_front2.png").convert_alpha(), PR)
tough_front_image3 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_front3.png").convert_alpha(), PR)
tough_front_image4 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_front4.png").convert_alpha(), PR)
tough_right_image1 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_right1.png").convert_alpha(), PR)
tough_right_image2 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_right2.png").convert_alpha(), PR)
tough_right_image3 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_right3.png").convert_alpha(), PR)
tough_right_image4 = transform.scale_by(
    image.load("assets/images/enemies/tough enemy/tough_right4.png").convert_alpha(), PR)
tough_enemy_images = [
    [tough_back_image1,tough_back_image2,tough_back_image3,tough_back_image4],
    [tough_left_image1,tough_left_image2,tough_left_image3,tough_left_image4],
    [tough_front_image1,tough_front_image2,tough_front_image3,tough_front_image4],
    [tough_right_image1,tough_right_image2,tough_right_image3,tough_right_image4]]

spirit_up_image = transform.scale_by(
    image.load("assets/images/enemies/spirit enemy/spirit_back.png").convert_alpha(), PR)
spirit_left_image = transform.scale_by(
    image.load("assets/images/enemies/spirit enemy/spirit_left.png").convert_alpha(), PR)
spirit_down_image = transform.scale_by(
    image.load("assets/images/enemies/spirit enemy/spirit_front.png").convert_alpha(), PR)
spirit_right_image = transform.scale_by(
```

```
    image.load("assets/images/enemies/spirit enemy/spirit_right.png").convert_alpha(), PR)
spirit_enemy_images = [spirit_up_image, spirit_left_image,
                       spirit_down_image, spirit_right_image]
for i in range(len(spirit_enemy_images)):
    face_mask = mask.from_threshold(spirit_enemy_images[i], (255,255,255),
                                    threshold=(1, 1, 1, 255))
    face = face_mask.to_surface(unsetcolor=(0,0,0,0))
    # pixels to be turned transparent are red
    body_mask = mask.from_threshold(spirit_enemy_images[i], (255,0,0),
                                    threshold=(1, 1, 1, 255))
    face.blit(body_mask.to_surface(setcolor=SPIRIT_ENEMY_COLOUR, unsetcolor=(0,0,0,0)), (0,0))
    spirit_enemy_images[i] = face


#=====================Game=====================#
white_flowers1_image = transform.scale_by(
    image.load("assets/images/game/white flowers.png").convert_alpha(), PR)
white_flowers2_image = transform.scale_by(
    image.load("assets/images/game/white flowers2.png").convert_alpha(), PR)

grass1_image = transform.scale_by(
    image.load("assets/images/game/grass.png").convert_alpha(), PR)
grass2_image = transform.scale_by(
    image.load("assets/images/game/grass2.png").convert_alpha(), PR)
grass3_image = transform.scale_by(
    image.load("assets/images/game/grass3.png").convert_alpha(), PR)

fences_image = transform.scale_by(
    image.load("assets/images/game/fences.png").convert_alpha(), PR)
crate_image = transform.scale_by(
    image.load("assets/images/game/crate.png").convert_alpha(), PR)
bullet_image = transform.scale_by(
    image.load("assets/images/game/bullet.png").convert_alpha(), PR)
exclamation = transform.scale_by(
    image.load("assets/images/game/exclamation.png").convert_alpha(), PR)

#=====================Buttons=====================#
eraser_image = transform.scale_by(
    image.load("assets/images/buttons/erase_button.png").convert_alpha(), PR)
eraser_image_pressed = transform.scale_by(
    image.load("assets/images/buttons/erase_button_pressed.png").convert_alpha(), PR)
undo_image = transform.scale_by(
    image.load("assets/images/buttons/undo_button.png").convert_alpha(), PR)
redo_image = transform.scale_by(
    image.load("assets/images/buttons/redo_button.png").convert_alpha(), PR)
clear_image = transform.scale_by(
    image.load("assets/images/buttons/clear_button.png").convert_alpha(), PR)
return_image = transform.scale_by(
    image.load("assets/images/buttons/return_button.png").convert_alpha(), PR)
settings_image = transform.scale_by(
    image.load("assets/images/buttons/settings_button.png").convert_alpha(), PR)
```

```
customise_image = transform.scale_by(
    image.load("assets/images/buttons/customise_button.png").convert_alpha(), PR)

#======================Other======================#
cursor_image = transform.scale_by(
    image.load("assets/images/other/mouse_cursor.png").convert_alpha(), PR)
padlock_image = transform.scale_by(
    image.load("assets/images/other/padlock.png").convert_alpha(), PR)
padlock_image = colour_swap(padlock_image, WHITE, BACKGROUND_COLOUR)
down_arrow_image = transform.scale_by(
    image.load("assets/images/other/down_arrow.png").convert_alpha(), PR)
up_arrow_image = transform.flip(down_arrow_image, False, True) # flip in y

#======================Fonts======================#
small_font_image = transform.scale_by(
    image.load("assets/images/fonts/small_font.png").convert_alpha(), PR)
medium_font_image = transform.scale_by(
    image.load("assets/images/fonts/medium_font.png").convert_alpha(), PR)
big_font_image = transform.scale_by(
    image.load("assets/images/fonts/big_big_font.png").convert_alpha(), PR)
huge_font_image = transform.scale_by(
    image.load("assets/images/fonts/massive_font.png").convert_alpha(), PR)

#======================Items======================#
bomb_image = transform.scale_by(
    image.load("assets/images/items/bomb.png").convert_alpha(), PR)
shotgun_image = transform.scale_by(
    image.load("assets/images/items/shotgun.png").convert_alpha(), PR)
shoes_image = transform.scale_by(
    image.load("assets/images/items/shoes.png").convert_alpha(), PR)
rapid_fire_image = transform.scale_by(
    image.load("assets/images/items/rapid_fire.png").convert_alpha(), PR)
time_freeze_image = transform.scale_by(
    image.load("assets/images/items/clock.png").convert_alpha(), PR)
backwards_shot_image = transform.scale_by(
    image.load("assets/images/items/backwards_shot.png").convert_alpha(), PR)
heart_image = transform.scale_by(
    image.load("assets/images/items/heart.png").convert_alpha(), PR)

item_images = [bomb_image, shoes_image, rapid_fire_image, shotgun_image,
               time_freeze_image, backwards_shot_image, heart_image]
```

6.4 Sounds File

sounds.py

```
from pygame import mixer

mixer.init()

button_click = mixer.Sound("assets/sounds/button click.wav")
hover_effect = mixer.Sound("assets/sounds/hover effect.wav")

default_shoot = mixer.Sound("assets/sounds/gun sounds/pew sound.wav")
enemy_killed = mixer.Sound("assets/sounds/famitracker/score.wav")
crow_sound = mixer.Sound("assets/sounds/enemy sounds/crow.mp3")
player_hit_sound = mixer.Sound("assets/sounds/famitracker/death3.wav")
player_death_sound = mixer.Sound("assets/sounds/famitracker/death2.wav")
power_up = mixer.Sound("assets/sounds/famitracker/life_up.wav")
crate_thud = mixer.Sound("assets/sounds/thud.wav")
bomb_sound = mixer.Sound("assets/sounds/famitracker/bomb.wav")

item_sounds = [bomb_sound, power_up, power_up, power_up, None, power_up, power_up]

all_sound_volumes = {default_shoot:      0.2,
                     crow_sound:         0.4,
                     button_click:       0.4,
                     hover_effect:       0.4,
                     player_hit_sound:   0.5,
                     player_death_sound: 0.5,
                     power_up:           0.5,
                     enemy_killed:       0.5,
                     crate_thud:         0.5,
                     bomb_sound:         0.8}
```

6.5 Utility Functions File

utility_functions.py

```
from math import ceil, log

from pygame import Rect, mask

def create_hex_dictionary(list):
    # creates a dictionary where each item in the provided list has an associated hex number
    depth = ceil(log(len(list), 16))
    key_value_list = []
    for i in range(len(list)):
        key_value = (list[i], hex(i)[2:].zfill(depth)) # [2:] to remove the '0x' prefix
        key_value_list.append(key_value)
    dictionary = dict(key_value_list)
    return dictionary, depth

def check_index(list, index):
    # checks if an index in a list exists
    try:
        list[index]
        return True
    except IndexError:
        return False

def split(string, part_size): # splits a string into groups of <part_size>
    list = []
    for i in range(0, len(string), part_size):
        list.append(string[i : i+part_size])
    return list

def get_key(value, dictionary): # returns the key of a value in a dictionary
    key_list = list(dictionary.keys())
    value_list = list(dictionary.values())
    return key_list[value_list.index(value)]

def clip(image, x, y, width, height): # clips an image to the rect dimensions provided
    image_copy = image.copy()
    clip_rect = Rect(x, y, width, height)
    image_copy.set_clip(clip_rect)
    clipped_image = image.subsurface(image_copy.get_clip())
    return clipped_image.copy()

def colour_swap(image, old_colour, new_colour): # swaps a colour in an image to another colour
    colour_mask = mask.from_threshold(image, old_colour, threshold=(1, 1, 1, 255))
    colour_change_surface = colour_mask.to_surface(setcolor=new_colour,
                                                   unsetcolor=(0, 0, 0, 0))
    image_copy = image.copy()
    image_copy.blit(colour_change_surface, (0, 0))
    return image_copy
```

```
def round_to_nearest(x, nearest): # rounds a number to the nearest n
    return nearest * round(x/nearest)

def position_list(first, last): # returns a list of positional strings within range provided
    if first <= last:
        positions = [str(i) for i in range(first, last+1)]
    else:
        positions = [str(i) for i in range(first, last-1, -1)]

    # lots of if statements to find out which number should have which suffix
    for i in range(len(positions)):
        match positions[i][-1]: # compare the last digit of the number
            case '1':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'st'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'st'
            case '2':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'nd'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'nd'
            case '3':
                if len(positions[i]) > 1:
                    if positions[i][-2] != '1':
                        positions[i] += 'rd'
                    else:
                        positions[i] += 'th'
                else:
                    positions[i] += 'rd'
            case _:
                positions[i] += 'th'

    return positions
```

6.6 Database Functions File

database_functions.py

```
from constants import DEFAULT_HEX

def db_create_database(db_cursor, db_connection): # create the database tables
    create_players_table = """CREATE TABLE IF NOT EXISTS Players(
                                        username CHAR(4) NOT NULL PRIMARY KEY,
                                        pin CHAR(4) NOT NULL,
                                        custom_character CHAR(60) NOT NULL,
                                        games_played INTEGER NOT NULL,
                                        enemies_killed INTEGER NOT NULL,
                                        bullets_shot INTEGER NOT NULL,
                                        items_used INTEGER NOT NULL
                                        )"""

    create_single_player_table = """CREATE TABLE IF NOT EXISTS SinglePlayerGames(
                                      game_id INTEGER PRIMARY KEY,
                                      score INTEGER NOT NULL,
                                      username CHAR(4) NOT NULL,
                                      FOREIGN KEY (username) REFERENCES Players(username)
                                      )"""

    create_two_player_table = """CREATE TABLE IF NOT EXISTS TwoPlayerGames(
                                    game_id INTEGER PRIMARY KEY,
                                    score INTEGER NOT NULL,
                                    player1_name CHAR(4) NOT NULL,
                                    player2_name CHAR(4) NOT NULL,
                                    FOREIGN KEY (player1_name) REFERENCES Players(username),
                                    FOREIGN KEY (player2_name) REFERENCES Players(username)
                                    )"""

    db_cursor.execute(create_players_table)
    db_cursor.execute(create_single_player_table)
    db_cursor.execute(create_two_player_table)
    db_connection.commit()

def db_get_1p_highscore(username, db_cursor):
    # returns the highscore for 1 player games of a user
    highscore = db_cursor.execute("""SELECT score
                                     FROM SinglePlayerGames
                                     WHERE username = ?
                                     ORDER BY score DESC""",
                                  (username, )).fetchone()
    if highscore:
        return highscore[0] # take it out of the tuple
    return 0

def db_get_2p_highscore(username, db_cursor):
    # returns the highscore for 2 player games of a user
```

```
    highscore = db_cursor.execute("""SELECT score
                                     FROM TwoPlayerGames
                                     WHERE player1_name = ?
                                     OR player2_name = ?
                                     ORDER BY score DESC""",
                                  (username, username)).fetchone()
    if highscore:
        return highscore[0] # take it out of the tuple
    return 0

def db_get_character(username, db_cursor): # returns the custom character of a user
    return db_cursor.execute("""SELECT custom_character
                                FROM Players
                                WHERE username = ?""",
                             (username, )).fetchone()[0]

def db_get_stats(username, db_cursor): # returns the statistics of a character
    return db_cursor.execute("""SELECT games_played, enemies_killed, bullets_shot, items_used
                                FROM Players
                                WHERE username = ?""",
                             (username,)).fetchone()

def db_get_all_usernames(db_cursor): # returns all the usernames in the database in a list
    usernames = []
    rows = db_cursor.execute("""SELECT username
                                FROM Players""").fetchall()
    for tuple in rows:
        usernames.append(tuple[0])
    return usernames

def db_find_player(username, pin, db_cursor): # checks if the pin for a user is correct
    return db_cursor.execute("""SELECT username
                                FROM Players
                                WHERE username = ? and pin = ?""",
                             (username, pin)).fetchone()

def db_add_to_stats(username, db_cursor, db_connection, games_played=0, enemies_killed=0,
                    bullets_shot=0, items_used=0): # increases the players stats
    curr_gp, curr_ek, curr_bs, curr_iu = db_get_stats(username, db_cursor)
    db_cursor.execute("""UPDATE Players
                         SET games_played = ?, enemies_killed = ?,
                                                bullets_shot = ?, items_used = ?
                         WHERE username = ?""",
                      (curr_gp + games_played, curr_ek + enemies_killed,
                       curr_bs + bullets_shot, curr_iu + items_used, username))
    db_connection.commit()

def db_set_character(username, character_hex, db_cursor, db_connection):
    # updates a player's custom character
    db_cursor.execute("""UPDATE Players
                         SET custom_character = ?
```

```
                        WHERE username = ?""",
                    (character_hex, username))
    db_connection.commit()

def db_insert_player(username, pin, db_cursor, db_connection):
    # adds a new player to the database
    db_cursor.execute("""INSERT INTO Players
                        VALUES(?,?,?,?,?,?,?)""",
                    (username, pin, DEFAULT_HEX, 0, 0, 0, 0))
    db_connection.commit()

def db_insert_1p_score(username, score, db_cursor, db_connection): # adds a new 1 player score
    db_cursor.execute("""INSERT INTO SinglePlayerGames(score, username)
                        VALUES(?, ?)""",
                    (score, username))
    db_connection.commit()

def db_insert_2p_score(username1, username2, score, db_cursor, db_connection):
    # adds a new 2 player score
    db_cursor.execute("""INSERT INTO TwoPlayerGames(score, player1_name, player2_name)
                        VALUES(?, ?, ?)""",
                    (score, username1, username2))
    db_connection.commit()
```

6.7 Utility Classes File

utility_classes.py

```
from math import ceil

import pygame

from constants import *
from utility_functions import colour_swap, clip, round_to_nearest, split, get_key
from images import padlock_image
from sounds import button_click, hover_effect

#======================Stack Class======================#
# used for the undo and redo buttons in the DrawingGrid class
class Stack():
    def __init__(self):
        self.__max_size = 100 # the maximum size of the stack
        self.__stack = []     # the stack itself implemented as a list

    def empty(self): # returns true if the stack is empty and false otherwise
        if self.__stack:
            return False
        else:
            return True

    def peek(self): # returns the last item in the stack
        return self.__stack[self.size()-1]

    def size(self): # returns the size of the stack
        return len(self.__stack)

    def push(self, item): # pushes item onto end of stack, removing first item if necessary
        if self.size() == self.__max_size:
            self.__stack = self.__stack[1:]
        self.__stack.append(item)

    def pop(self): # removes the last item from the stack and returns it
        return self.__stack.pop()

    def reset(self): # empties the stack
        self.__stack = []

#======================Queue Class======================#
class Queue():
    def __init__(self):
        self.__queue = []     # the queue itself implemented as a list

    def empty(self): # returns true if the queue is empty and false otherwise
        if self.__queue:
            return False
```

```
            else:
                return True

        def peek(self): # returns the last item in the queue
            return self.__queue[0]

        def size(self): # returns the size of the queue
            return len(self.__queue)

        def enqueue(self, item): # pushes an item to the end of the queue
            self.__queue.append(item)

        def dequeue(self): # removes the first item from the queue and returns it
            return self.__queue.pop(0)

        def reset(self): # empties the queue
            self.__queue = []

    #======================Button Class======================#
    # creates a pressable button that can be interacted with
    # the parent class of TextButton and ImageButton, that are used all over the project
    class Button():
        def __init__(self, pos, size, hold=True, pressed=False, disabled=False,
                     pressed_delay=BUTTON_PRESSED_DELAY, border_colour=None):
            self._rect = pygame.Rect(pos, size)
            self.__hold = hold # whether or not the button has to be held down to stay pressed
            self.__pressed = pressed
            self.__clicked = False
            self.__hover = False
            self.__pressed_time = 0
            self.__pressed_delay = pressed_delay
            self.__border_colour = border_colour
            self.__disabled = disabled

        def set_disabled(self, disabled): # change the disabled status of the button
            self.__disabled = disabled

        def unpress(self): # unpress the button
            self.__pressed = False
            self._image = self._default_image

        def get_disabled(self):
            return self.__disabled

        def get_pressed(self): # return if the button is currently pressed
            return self.__pressed

        def get_clicked(self): # return if the button is currently clicked
            return self.__clicked

        def update(self, mpos, click):
```

```
        # update the image and state of the button based on the mouse position and click
        if self.__disabled:
            self._image = self._disabled_image
        else:
            if (pygame.time.get_ticks() - self.__pressed_time > self.__pressed_delay
                                                                    and self.__hold):
                self.unpress() # pressed set to false if enough time has elapsed

            if self._hover_image and self._image == self._hover_image:
                # temporarily set to default if you were hovering over the button
                self._image = self._default_image

            self.__clicked = False # clicked is true only in the frame the button is clicked on

            if self._rect.collidepoint(mpos): # if the mouse is in the button rect
                if self.__hover == False:
                    # if you were previously not hovering, play the hover sound
                    hover_effect.play()

                if self._hover_image:
                    self._image = self._hover_image # if there is a hover image, display it
                self.__hover = True

                if click:
                    if not self.__pressed:
                         # if the button was previously not pressed, play the click sound
                        button_click.play()
                    self.__clicked = True
                    self.__pressed = True
                    self.__pressed_time = pygame.time.get_ticks()
            else:
                self.__hover = False

            if self.__pressed and self._pressed_image:
                # if the button is pressed and there is a pressed image, display it
                self._image = self._pressed_image

    def draw(self, screen): # draws the button
        if self._image: # if the button has an image, display it
            screen.blit(self._image, self._rect)
        else: # otherwise draw a coloured rect of the button
            pygame.draw.rect(screen, PALER_BACKGROUND, self._rect)
        if self.__border_colour:
            pygame.draw.rect(screen, self.__border_colour, self._rect, width=int(PIXEL_RATIO))

#======================Text Button Class======================#
# creates a button with text on it
# used all over the project, for example the login button
class TextButton(Button):
    def __init__(self, pos, size,  text, font, hold=True, pressed=False, disabled=False,
                                    text_colour=BUTTON_TEXT_COLOUR,
```

```
                                    pressed_text_colour=BUTTON_TEXT_PRESSED_COLOUR,
                                    disabled_text_colour=BUTTON_TEXT_DISABLED_COLOUR,
                                    background_colour=TEXT_BUTTON_BACKGROUND_COLOUR,
                                    hover_background_colour=TEXT_BUTTON_HOVER_COLOUR,
                                    pressed_delay=BUTTON_PRESSED_DELAY, border_colour=None):
        super().__init__(pos, size, hold, pressed, disabled, pressed_delay, border_colour)
        self.__set_images(text, font, text_colour, pressed_text_colour, disabled_text_colour,
                          background_colour, hover_background_colour)
        self._image = self._default_image

    # creates the images for the text button
    def __set_images(self, text, font, text_colour, pressed_text_colour, disabled_text_colour,
                     background_colour, hover_background_colour):
        text_x = self._rect.width//2
        if NEW_LINE in text:
            # if there are multiple lines, start the text a little bit down to not touch tops
            text_y = font.get_character_spacing() + PIXEL_RATIO
        else:
            # if there are not, start the text in the middle of the button
            text_y = self._rect.height//2 - font.get_height()//2

        default_font = font.new_colour_copy(text_colour)

        # pygame.SRCALPHA allows for transparency
        self._default_image = pygame.Surface(self._rect.size, pygame.SRCALPHA)
        if background_colour:
            self._default_image.fill(background_colour) # fill with the colour
        # render the text on top
        default_font.render(self._default_image, text, (text_x, text_y), alignment=CENTER)

        self._hover_image = pygame.Surface(self._rect.size, pygame.SRCALPHA)
        if hover_background_colour:
            self._hover_image.fill(hover_background_colour) # fill with the colour
        # render the text on top
        default_font.render(self._hover_image, text, (text_x, text_y), alignment=CENTER)

        self._pressed_image = pygame.Surface(self._rect.size, pygame.SRCALPHA)
        if hover_background_colour:
            self._pressed_image.fill(hover_background_colour)
        # different colour font for when the button is pressed
        pressed_font = font.new_colour_copy(pressed_text_colour)
        pressed_font.render(self._pressed_image, text, (text_x, text_y), alignment=CENTER)

        self._disabled_image = pygame.Surface(self._rect.size, pygame.SRCALPHA)
        if background_colour:
            self._disabled_image.fill(background_colour)
        disabled_font = font.new_colour_copy(disabled_text_colour)
        disabled_font.render(self._disabled_image, text, (text_x, text_y), alignment=CENTER)

#======================Image Button Class======================#
# creates a button with an image on it
```

```
# used all over the program, for example the return buttons
class ImageButton(Button):
    def __init__(self, pos, size, image, hold=True, pressed=False, disabled=False,
                                        pressed_image=None, image_colour=BUTTON_ICON_COLOUR,
                                        pressed_image_colour=BUTTON_ICON_PRESSED_COLOUR,
                                        disabled_image_colour=BUTTON_DISABLED_COLOUR,
                                        background_colour=BUTTON_BACKGROUND_COLOUR,
                                        hover_background_colour=BUTTON_HOVER_COLOUR,
                                        pressed_delay=BUTTON_PRESSED_DELAY,
                                        border_colour=None):
        super().__init__(pos, size, hold, pressed, disabled, pressed_delay, border_colour)
        self.__set_images(image, pressed_image, image_colour, pressed_image_colour,
                          disabled_image_colour, background_colour, hover_background_colour)
        self._image = self._default_image

    # creates the images for the image button
    def __set_images(self, image, pressed_image, image_colour, pressed_image_colour,
                     disabled_image_colour, background_colour, hover_background_colour):
        # create an empty surface
        self._default_image = pygame.Surface(self._rect.size)
        # fill it with the background colour
        self._default_image.fill(background_colour)
        # blit the image to it
        self._default_image.blit(colour_swap(image, WHITE, image_colour),(0,0))

        self._hover_image = pygame.Surface(self._rect.size)
        self._hover_image.fill(hover_background_colour)
        # images are created in white
        self._hover_image.blit(colour_swap(image, WHITE, image_colour),(0,0))

        if pressed_image:
            # passing in a pressed image overwrites the custom pressed image
            self._pressed_image = pressed_image
        else:
            self._pressed_image = pygame.Surface(self._rect.size)
            self._pressed_image.fill(hover_background_colour)
            self._pressed_image.blit(colour_swap(image, WHITE, pressed_image_colour),(0,0))

        self._disabled_image = pygame.Surface(self._rect.size)
        self._disabled_image.fill(background_colour)
        self._disabled_image.blit(colour_swap(image, WHITE, disabled_image_colour),(0,0))

#======================Font Class======================#
class Font():
    def __init__(self, font_image, character_list, colour, space_width,
                 character_spacing=PIXEL_RATIO, alpha=255):
        self.__font_image = font_image
        self.__character_list = character_list
        self.__characters = {}
        current_width = 0
        character_count = 0
```

```
        self.__space_width = space_width
        self.__character_spacing = character_spacing
        self.__height = font_image.get_height()

        border_colour = (255,0,0) # not in settings file as it is a property of the font images
        text_colour = (255,255,255)

        # if desired colour is not the colour of the border between letters,
        # swap the text colour to the new colour
        # otherwise,
        # set the colour of the borders to a different colour and then swap the text colour
        if colour != border_colour:
            font_image = colour_swap(font_image, text_colour, colour)
        else:
            different_colour = (127,127,127)
            # can be any random colour, just not border_colour or text_colour
            font_image = colour_swap(font_image, border_colour, different_colour)
            border_colour = different_colour
            font_image = colour_swap(font_image, text_colour, colour)

        if alpha != 255:
            font_image.set_alpha(alpha) # make transparent

        # loop through the width of the font image, incrementing by a pixel each time
        for x in range(0, font_image.get_width(), int(PIXEL_RATIO)):
            colour = font_image.get_at((x, 0)) # get the colour of the current pixel
            if colour == border_colour:
                # clip the font image from the last border colour to the new one
                character_image = clip(font_image, x - current_width, 0,
                                       current_width, font_image.get_height())
                # create an entry in the character dictionary for the current letter and image
                self.__characters[self.__character_list[character_count]] = character_image
                character_count += 1
                current_width = 0
            else:
                # stores the current width of the character, width since last border colour
                current_width += PIXEL_RATIO

    def new_colour_copy(self, new_colour, alpha=255): # return a copy with a different colour
        return Font(self.__font_image, self.__character_list, new_colour, self.__space_width,
                    character_spacing=self.__character_spacing, alpha=alpha)

    def get_text_width(self, text): # returns the width of some text in this font
        width = [0] # stores width of each line
        line = 0
        # first character first so no extra spacing
        width[0] += self.__characters[text[0]].get_width()
        for character in text[1:]:
            if character == NEW_LINE:
                width.append(0) # if there is a new line, create a new entry in the widths list
                line += 1
```

```
            elif character != ' ':
                # add the width of the spacing between characters
                width[line] += self.__character_spacing
                # add the width of the character
                width[line] += self.__characters[character].get_width()
            else:
                # if character is a space, add the width of a space
                width[line] += self.__space_width
        return max(width) # return the highest value in the list of line widths

    def get_height(self): # return the height of the font
        return self.__height

    def get_character_spacing(self): # return the width of the spacing between characters
        return self.__character_spacing

    def render(self, screen, text, pos, alignment=LEFT, hide=False): # render a string of text
        x_offset = [0]
        line = 0
        y_offset = 0
        for character in text:
            if hide:
                character = 'hidden' # 'hidden' is a character in character list
            if character == NEW_LINE:
                # if there is a new line, start drawing the characters further down
                y_offset += self.__height + 2*PIXEL_RATIO
                x_offset.append(0) # create a new entry in the offset list
                line += 1
            elif character != ' ':
                if alignment == LEFT: # only draw the characters if the alignment is left
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y] + y_offset))
                # add the necessary widths to the x_offset
                x_offset[line] += self.__characters[character].get_width()
                x_offset[line] += self.__character_spacing
            else:
                x_offset[line] += self.__space_width

        widths = x_offset.copy() # x_offset list now represents the width of each line
        line = 0

        if alignment == RIGHT:
            x_offset = []

            for width in widths:
                # x_offset list starts as the negative width of each line
                x_offset.append(-width)

            y_offset = 0
            for character in text:
                if hide:
```

```
                    character = 'hidden'
                if character == NEW_LINE:
                    y_offset = self.__height + 2*PIXEL_RATIO
                    line += 1
                if character != ' ':
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y]))
                    x_offset[line] += self.__characters[character].get_width()
                    x_offset[line] += self.__character_spacing
                else:
                    x_offset[line] += self.__space_width
        elif alignment == CENTER: # if alignment is to the center
            x_offset = []
            for width in widths:
                # x_offset list starts as the negative width/2 of each line
                x_offset.append(0 - round_to_nearest(width//2, PIXEL_RATIO))
            y_offset = 0
            for character in text:
                if hide:
                    character = 'hidden'
                if character == NEW_LINE:
                    y_offset = self.__height + 2*PIXEL_RATIO
                    line += 1
                elif character != ' ':
                    screen.blit(self.__characters[character], (pos[X] + x_offset[line],
                                                               pos[Y] + y_offset))
                    x_offset[line] += self.__characters[character].get_width()
                    x_offset[line] += self.__character_spacing
                else:
                    x_offset[line] += self.__space_width

#======================Text Box Class======================#
# creates a text box that a message can be written into
# used for entering usernames and pins for logging in or creating an account
class TextBox():
    def __init__(self, pos, size, type_font, message_font, max_length, name="",
                 allowed_strings = None, not_allowed_strings = None,
                 allowed_characters=CHARACTER_LIST_U, background_colour=TEXT_BOX_BACKGROUND,
                 border_width=int(PIXEL_RATIO), initial_text="", hide=False, typing=False):
        self.__rect = pygame.Rect(pos, size)
        self.__type_font = type_font
        self.__message_font = message_font
        self.__max_length = max_length
        self.__name = name
        # if allowed_strings are supplied, they are the only strings the text box accepts
        self.__allowed_strings = allowed_strings
        # if not allowed strings are supplied, they are not accepted by the text box
        self.__not_allowed_strings = not_allowed_strings
        self.__allowed_characters = allowed_characters
        self.__background_colour = background_colour
        self.__border_width = border_width
```

```
        self.__text = initial_text
        # hide is a boolean value that, when True, displays the entered text as dots
        self.__hide = hide
        # error_message used to tell the user when they have entered something invalid
        self.__error_message = ""

        self.__typing = typing # whether or not the text box is selected

    def __check_typing(self, mpos, click):
        if click:
            if self.__rect.collidepoint(mpos):
                if not self.__typing:
                    button_click.play() # play click sound if they weren't clicked on before
                self.__typing = True # if they click on the text box, typing becomes true
                self.__error_message = "" # reset the error message if they begin to type
            else:
                # if they have clicked anywhere else, typing becomes false
                self.__typing = False

    def __check_allowed(self): # returns an error message and a border colour based on the text
        # if there is not enough text,
        # border colour is red and it displays to the user how many more characters needed
        # if the text is in the not-allowed list,
        # border colour is amber and it displays to the user that the text is already taken
        # if the text is not in the allowed list,
        # border colour is amber and it displays to the user that the text is not recognised
        # if none of these have occurred,
        # border colour is green and no error message is displayed
        if len(self.__text) != self.__max_length:
            characters_left = self.__max_length - len(self.__text)
            message = f"{characters_left} MORE CHARACTER"
            message += "S" if characters_left != 1 else ""
            return RED, message
        elif self.__not_allowed_strings != None and self.__text in self.__not_allowed_strings:
            return AMBER, f"{self.__name} ALREADY TAKEN"
        elif self.__allowed_strings != None and self.__text not in self.__allowed_strings:
            return AMBER, f"{self.__name} NOT RECOGNISED"
        return GREEN, ""

    def get_valid(self): # returns whether or not the entered text is valid
        if len(self.__text) != self.__max_length:
            return False
        elif self.__not_allowed_strings != None and self.__text in self.__not_allowed_strings:
            return False
        elif self.__allowed_strings != None and self.__text not in self.__allowed_strings:
            return False
        return True

    def get_text(self): # returns the text
        return self.__text
```

```
def display_message(self, error_message): # allows the box to display custom error messages
    self.__error_message = error_message

def update(self, mpos, click, event_list): # converts keyboard input into text
    self.__check_typing(mpos, click)
    if self.__typing:
        for event in event_list:
            if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_BACKSPACE:
                        # if backspace pressed, remove a letter
                        self.__text = self.__text[:-1]
                    else:
                        if len(self.__text) < self.__max_length:
                            # convert keyboard input into a character and add to the text
                            character = event.unicode.upper()
                            if character in self.__allowed_characters:
                                self.__text += character

def draw(self, screen): # draw the text box
    if self.__background_colour:
        pygame.draw.rect(screen, self.__background_colour, self.__rect)

    if self.__name:
        self.__message_font.render(screen, f"{self.__name}:",
                                   (self.__rect.left - PIXEL_RATIO,
                                    self.__rect.centery -
                                                self.__message_font.get_height()//2),
                                   alignment=RIGHT)

    self.__border_colour, message = self.__check_allowed()

    if self.__typing: # display the border and __check_allowed message if typing
        pygame.draw.rect(screen, self.__border_colour, self.__rect,
                         width=self.__border_width)
        self.__message_font.render(screen, message,
                                       (self.__rect.centerx,
                                        self.__rect.bottom +
                                                  self.__message_font.get_height()//2),
                                       alignment=CENTER)

    self.__message_font.render(screen, self.__error_message,
                                       (self.__rect.centerx,
                                        self.__rect.bottom +
                                               self.__message_font.get_height()//2),
                                       alignment=CENTER)

    self.__type_font.render(screen, self.__text, (self.__rect.x + 4*PIXEL_RATIO,
                                                   self.__rect.centery -
                                                         self.__type_font.get_height()//2),
                                                   hide=self.__hide)
```

```
#=====================Character Display Class=====================#
# creates a display of a custom character specified by hex input
# the size of each pixel of the display can be specified
# the character can be changed and customised with functions
class CharacterDisplay():
    def __init__(self, pos, pixel_width, background_colour, hex_string,
                 extra_border_colour=None):
        self.__height = 8
        self.__width = 8
        self.__hat_height = 3
        self.__background_colour = background_colour
        self.__extra_border_colour = extra_border_colour

        # creates a dictionary of each part pattern as the value and the part as the key
        self.__parts = dict(zip([0,1,2,3,4,5], FRONT_PATTERNS))

        self.__grid = []
        for row in range(self.__height):
            current_row = []
            for column in range(self.__width):
                current_row.append(Pixel((pos[X]+(column*pixel_width*PIXEL_RATIO),
                                          pos[Y]+(row*pixel_width*PIXEL_RATIO)),
                                         background_colour=background_colour,
                                         width=pixel_width, height=pixel_width))
            self.__grid.append(current_row)

        self.__rect = pygame.Rect(pos, (self.__width*pixel_width*PIXEL_RATIO,
                                        self.__height*pixel_width*PIXEL_RATIO))
        self.__border_thickness = ceil(pixel_width / 2)
        self.__border_rect = pygame.Rect((pos[X] - self.__border_thickness*PIXEL_RATIO,
                                          pos[Y] - self.__border_thickness*PIXEL_RATIO),
                                         (self.__rect.width +
                                                   2*self.__border_thickness*PIXEL_RATIO,
                                          self.__rect.height +
                                                   2*self.__border_thickness*PIXEL_RATIO))

        if hex_string:
            # the initial state of the grid is defined by the hex string passed in
            self.hex_to_grid(hex_string)

    def select_hex_to_grid(self, hex_string):
        # converts each part of the custom character to each colour defined by the hex
        hex_list = split(hex_string, COLOUR_DEPTH) # split the hex into individual colours
        for part_index in self.__parts.keys():
            # update each part to its respective colour
            self.update_pattern(part_index, get_key(hex_list[part_index], BITMAP_DICTIONARY))

    def update_pattern(self, part, colour): # changes the colour of a part of the character
        # goes through each coordinate in the part and changes the colour of the pixel
        for row,column in self.__parts[part]:
            # uses background colour so that a hat doesn't overwrite the character's forehead
```

```
            self.__grid[row][column].set_background_colour(colour)

    def hat_hex_to_grid(self, hex_string):
        # takes a hex string and converts the hat into the state the hex describes
        # split hex string into individual colours in rows in the same format as the grid
        hex_array = split(split(hex_string, COLOUR_DEPTH), self.__width)
        for row in range(len(hex_array)):
            for column in range(self.__width):
                # converts each pixel into each colour
                self.__grid[row][column].set_colour(get_key(hex_array[row][column],
                                                            BITMAP_DICTIONARY))

    def hex_to_grid(self, hex_string):
        # takes a hex string and converts the character grid into the state it describes
        # the first part of the hex string is the 6 colours of the select grids
        self.select_hex_to_grid(hex_string[:-COLOUR_DEPTH*self.__hat_height*self.__width])
        # the latter part is the 24 colours of each pixel in the hat
        self.hat_hex_to_grid(hex_string[-COLOUR_DEPTH*self.__hat_height*self.__width:])

    def get_rect(self): # return the rect of the grid
        return self.__rect

    def draw(self, screen): # draws the character to the screen
        if self.__background_colour:
            pygame.draw.rect(screen, self.__background_colour, self.__border_rect)
        if self.__extra_border_colour:
            pygame.draw.rect(screen, self.__extra_border_colour, self.__border_rect,
                             width=int(ceil(self.__border_thickness/2)*PIXEL_RATIO))
        for row in self.__grid:
            for pixel in row:
                pixel.draw(screen)

#======================Pixel Class======================#
# a coloured pixel that has a rect and can be drawn
# used in the ColourGrid, CharacterDisplay, and DrawingGrid classes
class Pixel():
    def __init__(self, pos, background_colour=PALER_BACKGROUND, colour=None,
                 width=8, height=8):
        # background colour so that it can hold what should be behind it
        self.__background_colour = background_colour
        self.__colour = colour
        self.__pos = pos
        # the rect, used to check if the a coordinate collides with the pixel
        self.__rect = pygame.Rect(pos, (width*PIXEL_RATIO, height*PIXEL_RATIO))
        self.__locked = False # pixels can be locked in colour grids

    def get_rect(self): # return the rect of the pixel
        return self.__rect

    def get_pos(self): # return the position of the pixel
        return self.__pos
```

```
    def get_colour(self): # return the colour of the pixel
        return self.__colour

    def get_locked(self): # return the locked status of the pixel
        return self.__locked

    def set_locked(self, bool): # change the locked status of the pixel
        self.__locked = bool

    def set_colour(self, colour): # change the colour of the pixel
        self.__colour = colour

    def set_background_colour(self, colour): # change the background colour of the pixel
        self.__background_colour = colour

    def draw(self, screen): # draws the pixel to the screen
        # if the pixel is locked, draw a padlock
        # otherwise if the pixel has a colour, draw it
        # otherwise draw the background colour
        if self.__locked:
            pygame.draw.rect(screen, LOCKED_COLOUR, self.__rect)
            screen.blit(padlock_image, self.__rect.topleft)
        elif self.__colour:
            pygame.draw.rect(screen, self.__colour, self.__rect)
        elif self.__background_colour:
            pygame.draw.rect(screen, self.__background_colour, self.__rect)

#======================Slider Class======================#
# creates a slider that you can drag up and down to represent a value
# used for the volume setting
class Slider():
    def __init__(self, pos, size, min, max, value, bar_colour=PALER_BACKGROUND,
                 slider_colour=WHITE):
        self.__min = min # the minimum value of the slider
        self.__max = max # the maximum value of the slider
        self.__bar_colour = bar_colour # the colour of the bar
        self.__slider_colour = slider_colour # the colour of the slider
        self.__active = False # whether or not the slider is being used

        # the rect of the bar
        self.__bar_rect = pygame.Rect(pos, size)
        # the rect of the slider
        self.__slider_rect = pygame.Rect((pos[X], pos[Y] - 2*PIXEL_RATIO),
                                         (4*PIXEL_RATIO, size[Y] + 4*PIXEL_RATIO))
        # moves the slider to where the initial value would be
        self.__slider_rect.centerx = self.__x_from_value(value)

    def __x_from_value(self, value): # takes a value and returns the x coordinate for slider
        return (((value - self.__min) / (self.__max - self.__min)) * self.__bar_rect.width
                                                                   + self.__bar_rect.left)
```

```
def __value_from_x(self, x): # takes an x coordinate and returns the value it represents
    return (((x - self.__bar_rect.left) /
                        self.__bar_rect.width) * (self.__max - self.__min) + self.__min)

def get_value(self): # returns the value of the slider
    return round(self.__value_from_x(self.__slider_rect.centerx), 2)

def update(self, click, unclick, mpos): # update the slider from clicking and mouse pos
    # if the slider is clicked on
    if not self.__active and click and self.__slider_rect.collidepoint(mpos):
        button_click.play()
        self.__active = True # become active

    if self.__active:
        if mpos[X] <= self.__bar_rect.left:
            # stop the slider at the left boundary
            self.__slider_rect.centerx = self.__bar_rect.left
        elif mpos[X] >= self.__bar_rect.right:
            # stop the slider at the right boundary
            self.__slider_rect.centerx = self.__bar_rect.right
        else:
            # if within range, the x of the slider becomes the x of the mouse position
            self.__slider_rect.centerx = mpos[X]

        if unclick: # if active and the mouse button is unclicked, no longer active
            self.__active = False
             # true returned if slider has been used so the program can update the volume
            return True
    return False

def draw(self, screen): # draw the slider
    pygame.draw.rect(screen, self.__bar_colour, self.__bar_rect)
    pygame.draw.rect(screen, self.__slider_colour, self.__slider_rect)
```

6.8 Leaderboard Classes File

leaderboard_classes.py

```
import pygame

from constants import *
from utility_functions import position_list, check_index
from utility_classes import CharacterDisplay

#=====================Leaderboard Class=====================#
# a customly formatted single player leaderboard
# uses lots of parameters to allow this customisability and changes to database functionality
class Leaderboard():
    def __init__(self, db_cursor, pos, width, font, value_field, value_field_table,
                 key_field_table, key_field="username", rows=10, record_height=10*PIXEL_RATIO,
                 highlight_key=None, position_colour=SLIGHT_GREY, display_characters=False,
                 characters_field="custom_character"):
        self._database = db_cursor
        self._value_field_table = value_field_table
        self._value_field = value_field
        self._key_field = key_field
        self._key_field_table = key_field_table
        self.__display_characters = display_characters
        self.__characters_field = characters_field
        self.__highlight_key = highlight_key
        self._rows = rows
        # large rect for the leaderboard to be displayed in
        self.__rect = pygame.Rect(pos, (width, record_height*rows + (rows+1)*PIXEL_RATIO))

        key_fields = self._fetch_key_fields()

        values = self.__fetch_values()

        if display_characters:
            characters = self.__fetch_custom_characters()
        else:
            characters = None

        positions = position_list(1, rows)

        # loops through all the positions and creates a LeaderboardRecord for each
        self.__records = []
        for i in range(len(positions)):
            highlight = False # whether the record should be highlighted
            if check_index(key_fields, i):
                key_field = key_fields[i][0] # the key field for the record
                if highlight_key in key_field:
                    highlight = True
            else:
                key_field = ""
```

```
            if check_index(values, i):
                value = values[i][0] # the value for the record
            else:
                value = None

            if display_characters and check_index(values, i):
                character_hex = characters[i][0] # the character hex string for the record
            else:
                character_hex = None

            self.__records.append(LeaderboardRecord((pos[X]+PIXEL_RATIO,
                                                     pos[Y] + i*record_height +
                                                                      (i+1)*PIXEL_RATIO),
                                                    (width-2*PIXEL_RATIO, record_height),
                                                    font, positions[i], key_field,
                                                    value, character_hex=character_hex,
                                                    position_colour=position_colour,
                                                    highlight=highlight))

    def _fetch_key_fields(self): # the query to fetch all the key fields for the leaderboard
        key_fields_query = f"""SELECT {self._key_field_table}.{self._key_field}
                               FROM {self._key_field_table}, {self._value_field_table}
                               WHERE {self._key_field_table}.{self._key_field} =
                                                    {self._value_field_table}.{self._key_field}
                               ORDER BY {self._value_field_table}.{self._value_field} DESC
                               LIMIT {self._rows}"""
        return self._database.execute(key_fields_query).fetchall()

    def __fetch_values(self): # the query to fetch all the values for the leaderboard
        values_query = f"""SELECT {self._value_field}
                           FROM {self._value_field_table}
                           ORDER BY {self._value_field} DESC
                           LIMIT {self._rows}"""
        return self._database.execute(values_query).fetchall()

    def __fetch_custom_characters(self): # the query to fetch all the custom characters
        characters_query = f"""SELECT {self._key_field_table}.{self.__characters_field}
                               FROM {self._key_field_table}, {self._value_field_table}
                               WHERE {self._key_field_table}.{self._key_field} =
                                                    {self._value_field_table}.{self._key_field}
                               ORDER BY {self._value_field_table}.{self._value_field} DESC
                               LIMIT {self._rows}"""
        return self._database.execute(characters_query).fetchall()

    def update(self): # requery all data in the leaderboard
        # fetch all the key fields again and update each record to display this
        key_fields = self._fetch_key_fields()
        for i in range(len(self.__records)):
            if check_index(key_fields, i):
                self.__records[i].set_key_field(key_fields[i][0])
```

```
        # fetch all the values again and update each record to display this
        values = self.__fetch_values()
        for i in range(len(self.__records)):
            if check_index(values, i):
                self.__records[i].set_value(values[i][0])

        # fetch all the custom characters again and update each record to display this
        if self.__display_characters:
            characters = self.__fetch_custom_characters()
            for i in range(len(self.__records)):
                if check_index(characters, i):
                    self.__records[i].set_character_display(characters[i][0])

        # checks for changes in record highlighting
        for record in self.__records:
            if self.__highlight_key in record.get_key_field():
                record.set_highlight(True)
            else:
                record.set_highlight(False)

    def draw(self, screen): # draw the leaderboard to the screen
        pygame.draw.rect(screen, TEXT_BUTTON_HOVER_COLOUR, self.__rect)
        for record in self.__records:
            record.draw(screen)

#=====================Two Player Leaderboard Class=====================#
# a customly formatted two player leaderboard
# inherits from the leaderboard class
class TwoPlayerLeaderboard(Leaderboard):
    def __init__(self, db_cursor, pos, width, font, value_field, value_field_table,
                 key_field_table, key_field="username", key_field1="player1_name",
                 key_field2="player2_name", rows=10, record_height=10*PIXEL_RATIO,
                 highlight_key=None, position_colour=SLIGHT_GREY, display_characters=False,
                 characters_field="custom_character"):
        self._key_field1 = key_field1
        self._key_field2 = key_field2
        super().__init__(db_cursor, pos, width, font, value_field, value_field_table,
                         key_field_table, key_field=key_field, rows=rows,
                         record_height=record_height, highlight_key=highlight_key,
                         position_colour=position_colour,
                         display_characters=display_characters,
                         characters_field=characters_field)

    def _fetch_key_fields(self): # polymorphism, changes how key fields are fetched
        # the key fields for the 1st players and the 2nd players are fetched
        key_fields1_query = f"""SELECT {self._key_field_table}.{self._key_field}
                                FROM {self._key_field_table}, {self._value_field_table}
                                WHERE {self._key_field_table}.{self._key_field} =
                                                {self._value_field_table}.{self._key_field1}
                                ORDER BY {self._value_field_table}.{self._value_field} DESC
```

```
                            LIMIT {self._rows}"""
        key_fields2_query = f"""SELECT {self._key_field_table}.{self._key_field}
                            FROM {self._key_field_table}, {self._value_field_table}
                            WHERE {self._key_field_table}.{self._key_field} =
                                        {self._value_field_table}.{self._key_field2}
                            ORDER BY {self._value_field_table}.{self._value_field} DESC
                            LIMIT {self._rows}"""
        fields1 = self._database.execute(key_fields1_query).fetchall()
        fields2 = self._database.execute(key_fields2_query).fetchall()
        key_fields = []
        for i in range(len(fields1)):
            # the player 1 and player 2 fields are combined into single key fields
            key_fields.append((f"{fields1[i][0]} + {fields2[i][0]}", ))
        return key_fields

#======================Individual Leaderboard Record Class======================#
# a single record in a leaderboard
class LeaderboardRecord():
    def __init__(self, pos, size, font, position, key_field, value, character_hex=None,
                 position_colour=SLIGHT_GREY, highlight=False):
        self.__rect = pygame.Rect(pos, size)
        self.__font = font
        self.__position = position
        self.__position_font = self.__font.new_colour_copy(position_colour)
        self.__position_width = self.__font.get_text_width(position)
        self.__key_field = key_field
        self.__value = value
        self.__highlight = highlight

        match position: # checks for special borders
            case '1st':
                display_border = GOLD
            case '2nd':
                display_border = SILVER
            case '3rd':
                display_border = BRONZE
            case _:
                display_border = PALEST_BACKGROUND

        if character_hex: # initialise the character display if needed
            self.__character_display = CharacterDisplay(
                                (self.__rect.x + self.__position_width + 4*PIXEL_RATIO,
                                 self.__rect.y + (self.__rect.height//2 - 4*PIXEL_RATIO)),
                                1, GRASS_GREEN, character_hex,
                                extra_border_colour=display_border)
        else:
            self.__character_display = None

        self.__font_y = self.__rect.y + (self.__rect.height - self.__font.get_height())//2
        self.__position_x = self.__rect.x + PIXEL_RATIO
        self.__name_x = self.__rect.x + self.__position_width + 3*PIXEL_RATIO
```

```
        self.__name_x += (EIGHT_PIXELS + 4*PIXEL_RATIO) if self.__character_display else 0

    def set_highlight(self, highlight): # set whether the record is highlighted or not
        self.__highlight = highlight

    def get_key_field(self): # return the key field of the record
        return self.__key_field

    def set_value(self, value): #
        self.__value = value

    def set_key_field(self, key_field):
        self.__key_field = key_field

    def set_character_display(self, character_hex):
        self.__character_display.hex_to_grid(character_hex)

    def draw(self, screen):
        if self.__highlight:
            # position displayed in white and rect is paler if it is your score
            pygame.draw.rect(screen, LEADERBOARD_HIGHLIGHT_COLOUR, self.__rect)
            self.__font.render(screen, self.__position, (self.__position_x, self.__font_y))
        else:
            pygame.draw.rect(screen, LEADERBOARD_DEFAULT_COLOUR, self.__rect)
            self.__position_font.render(screen, self.__position, (self.__position_x,
                                                                  self.__font_y))
        self.__font.render(screen, self.__key_field, (self.__name_x, self.__font_y))
        if self.__character_display:
            self.__character_display.draw(screen)
        if self.__value != None: # in case a 0 is achieved
            self.__font.render(screen, str(self.__value), (self.__rect.x + self.__rect.width,
                                                           self.__font_y), alignment=RIGHT)

#====================Podium Class====================#
# a podium style display of the top 3 players in a certain field
class Podium():
    def __init__(self, db_cursor, field, pos, spacing, width_1st,
                 width_others, font_1st, font_others):
        # pos is taken as (centre_x, bottom_y)
        self.__font_1st = font_1st
        self.__font_others = font_others
        self.__width_1st = width_1st
        self.__width_others = width_others

        # query to fetch all the usernames
        self.__usernames = db_cursor.execute(f"""SELECT username
                                                 FROM Players
                                                 WHERE {field} > 0
                                                 ORDER BY {field} DESC
                                                 LIMIT 3""").fetchall()
        # query to fetch all the custom character strings
```

```
        custom_characters = db_cursor.execute(f"""SELECT custom_character
                                                  FROM Players
                                                  WHERE {field} > 0
                                                  ORDER BY {field} DESC
                                                  LIMIT 3""").fetchall()
        # query to fetch all the values
        self.__values = db_cursor.execute(f"""SELECT {field}
                                              FROM Players
                                              WHERE {field} > 0
                                              ORDER BY {field} DESC
                                              LIMIT 3""").fetchall()

        # initialise the character displays with gold, silver, and bronze backgrounds
        character1st = custom_characters[0][0] if check_index(custom_characters, 0) else None
        character2nd = custom_characters[1][0] if check_index(custom_characters, 1) else None
        character3rd = custom_characters[2][0] if check_index(custom_characters, 2) else None
        self.__character_display_1st = CharacterDisplay(
                (pos[X] - (width_1st/2)*EIGHT_PIXELS,
                 pos[Y] - width_1st*EIGHT_PIXELS),
                width_1st, GOLD, character1st)
        self.__character_display_2nd = CharacterDisplay(
                (pos[X] - (width_1st/2)*EIGHT_PIXELS - spacing - width_others*EIGHT_PIXELS,
                 pos[Y] - width_others*EIGHT_PIXELS),
                width_others, SILVER, character2nd)
        self.__character_display_3rd = CharacterDisplay(
                (pos[X] + (width_1st/2)*EIGHT_PIXELS + spacing,
                 pos[Y] - width_others*EIGHT_PIXELS),
                width_others, BRONZE, character3rd)

    def draw(self, screen): # draw the podium to the screen
        # displays the character displays
        self.__character_display_1st.draw(screen)
        self.__character_display_2nd.draw(screen)
        self.__character_display_3rd.draw(screen)
        # check if users should be displayed and displays the relevant information if so
        if check_index(self.__usernames, 0):
            self.__font_1st.render(screen, self.__usernames[0][0],
                    (self.__character_display_1st.get_rect().centerx,
                     self.__character_display_1st.get_rect().top -
                            self.__font_1st.get_height() - (self.__width_1st-1)*PIXEL_RATIO),
                    alignment=CENTER)
            self.__font_others.render(screen, str(self.__values[0][0]),
                    (self.__character_display_1st.get_rect().centerx,
                     self.__character_display_1st.get_rect().bottom +
                                                        self.__width_1st*PIXEL_RATIO),
                    alignment=CENTER)
            if check_index(self.__usernames, 1):
                self.__font_others.render(screen, self.__usernames[1][0],
                        (self.__character_display_2nd.get_rect().centerx,
                         self.__character_display_2nd.get_rect().top -
                                        self.__font_others.get_height() -
```

```
                                (self.__width_others-1)*PIXEL_RATIO - PIXEL_RATIO),
            alignment=CENTER)
    self.__font_others.render(screen, str(self.__values[1][0]),
            (self.__character_display_2nd.get_rect().centerx,
             self.__character_display_2nd.get_rect().bottom +
                                        self.__width_others*PIXEL_RATIO),
            alignment=CENTER)
    if check_index(self.__usernames, 2):
        self.__font_others.render(screen, self.__usernames[2][0],
                (self.__character_display_3rd.get_rect().centerx,
                 self.__character_display_3rd.get_rect().top -
                            self.__font_others.get_height() -
                                (self.__width_others-1)*PIXEL_RATIO - PIXEL_RATIO),
                alignment=CENTER)
        self.__font_others.render(screen, str(self.__values[2][0]),
                (self.__character_display_3rd.get_rect().centerx,
                 self.__character_display_3rd.get_rect().bottom +
                                            self.__width_others*PIXEL_RATIO),
                alignment=CENTER)
```

6.9 Customisation Classes File

customise_classes.py

```
from math import ceil

import pygame

from constants import *
from utility_functions import get_key, check_index, split
from utility_classes import Pixel, ImageButton, Stack
from images import undo_image, redo_image, clear_image, eraser_image, eraser_image_pressed
from sounds import button_click, hover_effect

#=====================Colour Grid Class=====================#
# a grid of selectable coloured pixels
# used to change colours of the CharacterDisplay grid and to change colour in the DrawingGrid
class ColourGrid():
    def __init__(self, pos, colours, max_width, hex_colour=None, locked_list=None):
        self.__colours = colours
        self.__max_width = max_width

        # creates the grid as a 2D array of pixels
        self.__grid = []
        for row in range(ceil(len(self.__colours)/self.__max_width)): # number of rows
            current_row = []
            for column in range(self.__max_width):
                # only creates a pixel if there is a colour to go with it
                if check_index(self.__colours, column + row*max_width):
                    current_row.append(Pixel((pos[X] + column*EIGHT_PIXELS,
                                              pos[Y] + row*EIGHT_PIXELS),
                                              colour=self.__colours[column +
                                                                    row*self.__max_width]))
            self.__grid.append(current_row)
        self.__selected = self.__grid[0][0] # the selected pixel, defaulted to the first pixel

        self.__hover = None

        if len(self.__colours) / self.__max_width >= 1:
            width = self.__max_width * EIGHT_PIXELS
        else:
            width = (len(self.__colours) % self.__max_width) * EIGHT_PIXELS
        height = ceil(len(self.__colours)/self.__max_width)*EIGHT_PIXELS
        self.__rect = pygame.Rect(pos, (width, height))
        self.__border_rect = pygame.Rect((self.__rect.x - 5*PIXEL_RATIO,
                                          self.__rect.y - 2*PIXEL_RATIO),
                                         (self.__rect.width + 10*PIXEL_RATIO,
                                          self.__rect.height + EIGHT_PIXELS + 4*PIXEL_RATIO))

        # if a hex colour is passed as a parameter, set it as the selected colour
        if hex_colour:
```

```
            self.__hex_to_selected(hex_colour)
        self.__update_select_rect()

        # if there are any locked colours
        if locked_list:
            for locked in locked_list:
                query = f"SELECT {locked[3]} FROM Players WHERE username = ?"
                number = locked[1].execute(query, (locked[2], )).fetchone()[0]
                if number <= locked[4]:
                    self.__grid[locked[0][0]][locked[0][1]].set_locked(True)

    def __hex_to_selected(self, hex_colour): # changes selected pixel to pixel with hex colour
        index = self.__colours.index(get_key(hex_colour, BITMAP_DICTIONARY))
        self.__selected = self.__grid[index//self.__max_width][index%self.__max_width]

    def selected_to_hex(self): # returns the hex of the colour of the current selected pixel
        return BITMAP_DICTIONARY[self.__selected.get_colour()]

    def get_changed(self): # returns the changed status of the grid
        return self.__changed

    def get_rect(self): # returns the rect of the grid
        return self.__rect

    def get_selected(self): # returns the selected pixel
        return self.__selected

    def deselect(self): # deselect any selected pixel
        self.__selected = None

    def update(self, mpos, click): # changes the selected pixel if a pixel is clicked on
        self.__changed = False
        # if the user has clicked, loops through all pixels
        # check if mouse collides with any and updates the selected pixel if necessary
        if not self.__rect.collidepoint(mpos):
            self.__hover = None
        for row in self.__grid:
            for pixel in row:
                if pixel.get_rect().collidepoint(mpos) and not pixel.get_locked():
                    if click:
                        if self.__selected != pixel:
                            button_click.play()
                        self.__selected = pixel
                        self.__update_select_rect()
                        self.__changed = True  # changed used to check if an update is needed
                    else:
                        if not self.__hover or self.__hover != pixel:
                            if self.__selected != pixel:
                                hover_effect.play()
                            self.__hover = pixel
                            self.__update_hover_rect()
```

```
    def __update_select_rect(self): # updates rect used as the border for the selected pixel
        self.__select_rect = pygame.Rect((self.__selected.get_pos()[X] - 1*PIXEL_RATIO,
                                          self.__selected.get_pos()[Y] - 1*PIXEL_RATIO),
                                         (EIGHT_PIXELS + 2*PIXEL_RATIO,
                                          EIGHT_PIXELS + 2*PIXEL_RATIO))

    def __update_hover_rect(self): # update rect used as the border for the hovered over pixel
        self.__hover_rect = pygame.Rect((self.__hover.get_pos()[X] - 1*PIXEL_RATIO,
                                         self.__hover.get_pos()[Y] - 1*PIXEL_RATIO),
                                        (EIGHT_PIXELS + 2*PIXEL_RATIO,
                                         EIGHT_PIXELS + 2*PIXEL_RATIO))

    def draw_border_rect(self, screen): # draws a border for the grid
        pygame.draw.rect(screen, BUTTON_HOVER_COLOUR, self.__border_rect)

    def draw(self, screen): # draws the grid to the screen
        # draws each pixel
        for row in self.__grid:
            for pixel in row:
                pixel.draw(screen)
        # if there is a selected pixel, draw the select_rect with the same colour
        if self.__hover:
            pygame.draw.rect(screen, self.__hover.get_colour(), self.__hover_rect)
        if self.__selected:
            pygame.draw.rect(screen, self.__selected.get_colour(), self.__select_rect)

#======================Drawing Grid Class======================#
# creates a grid that can be drawn onto by the user
# the colour can be picked by a colour grid
# there are also buttons to undo, redo, clear, and use an eraser
class DrawingGrid():
    def __init__(self, pos, columns, rows, colours, initial_hex=None):
        self.__rows = rows
        self.__columns = columns

        self.__grid = [[Pixel((pos[X]+(column*EIGHT_PIXELS), pos[Y]+(row*EIGHT_PIXELS)),
                              background_colour=WHITE)
                        for column in range(self.__columns)]
                            for row in range(self.__rows)]
        self.__rect = pygame.Rect(pos, (self.__columns*EIGHT_PIXELS, self.__rows*EIGHT_PIXELS))
        self.__border_rect = pygame.Rect((self.__rect.x - 2*PIXEL_RATIO,
                                          self.__rect.y - 2*PIXEL_RATIO),
                                         (self.__rect.width + 4*PIXEL_RATIO,
                                          self.__rect.height + 4*PIXEL_RATIO))

        # the drawing grid has an undo, redo, clear, and eraser button
        self.__undo_stack = Stack()
        self.__redo_stack = Stack()

        self.__undo_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
```

```
                                    pos[Y] + EIGHT_PIXELS),
                                   (EIGHT_PIXELS, EIGHT_PIXELS),
                                   undo_image)
        self.__redo_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                    pos[Y] + 2*EIGHT_PIXELS),
                                   (EIGHT_PIXELS, EIGHT_PIXELS),
                                   redo_image)
        self.__clear_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                     pos[Y]),
                                    (EIGHT_PIXELS, EIGHT_PIXELS),
                                    clear_image)
        self.__erase_button = ImageButton((pos[X] + self.__columns*EIGHT_PIXELS + EIGHT_PIXELS,
                                    pos[Y] - 2*EIGHT_PIXELS),
                                   (EIGHT_PIXELS, EIGHT_PIXELS),
                                   eraser_image, hold=False,
                                   pressed_image=eraser_image_pressed)

        # has a colour grid to select the drawing colour from
        self.__colour_grid = ColourGrid((pos[X], pos[Y] - 2*EIGHT_PIXELS),
                                  colours, self.__columns)

        self.__previous_state = None
        self.__changed = False
        self.__drawn = False

        for i in range(self.__rows): # alternating grey and white background to show blank
            row = self.__grid[i]
            for j in range(i % 2, self.__columns, 2):
                row[j].set_background_colour(CUSTOMISATION_GREY)

        # if a hex string is passed in, set the current state of the drawing grid to it
        if initial_hex:
            self.hex_to_grid(initial_hex)

    def grid_to_hex(self): # creates a single line hex string of the current state of the grid
        hex_string = ""
        for row in self.__grid:
            for pixel in row:
                hex_string += BITMAP_DICTIONARY[pixel.get_colour()]
        return hex_string

    def hex_to_grid(self, hex_string):
        # takes hex input and converts the drawing grid into the state specified by it
        # split hex string into individual colours in rows in the same format as the grid
        hex_array = split(split(hex_string, COLOUR_DEPTH), self.__columns)
        for row in range(self.__rows):
            for column in range(self.__columns):
                # converts each pixel into each colour
                self.__grid[row][column].set_colour(get_key(hex_array[row][column],
                                                            BITMAP_DICTIONARY))
```

```
def update(self, mpos, clicking, click, unclick):
    # updates all the different aspects of the drawing grid
    self.__changed = False # changed is used to check if an update is needed elsewhere
    # update each button and importantly their pressed attributes
    self.__colour_grid.update(mpos, click)
    self.__undo_button.update(mpos, click)
    self.__redo_button.update(mpos, click)
    self.__erase_button.update(mpos, click)
    self.__clear_button.update(mpos, click)
    # when you click, save the current state to a temporary variable in case you draw
    if click:
        self.__previous_state = self.grid_to_hex()
    if self.__undo_button.get_clicked():
        self.__undo()
        self.__changed = True
    if self.__redo_button.get_clicked():
        self.__redo()
        self.__changed = True
    # deselect the colour from the select grid if the erase button is pressed
    if self.__erase_button.get_clicked():
        self.__colour_grid.deselect()
    # unpress the eraser button if the colour grid is selected
    if self.__erase_button.get_pressed() and self.__colour_grid.get_selected():
        self.__erase_button.unpress()
    if self.__clear_button.get_clicked():
        self.__clear()
        # push the state of the grid before anything was changed onto the stack
        self.__undo_stack.push(self.__previous_state)
        self.__changed = True
    if self.__drawn: # if you've drawn
        self.__redo_stack.reset()
        if unclick:  # if you've drawn and just released the mouse button
            # push the state of the grid anything was changed onto the stack
            self.__undo_stack.push(self.__previous_state)
            self.__drawn = False
            self.__changed = False
    if clicking:
        for row in self.__grid:
            for pixel in row:
                if pixel.get_rect().collidepoint(mpos):
                    # drawn is only true if you have drawn onto the grid
                    # changed refers to any aspect of the drawing grid
                    self.__drawn = True
                    self.__changed = True
                    if not self.__erase_button.get_pressed():
                        pixel.set_colour(self.__colour_grid.get_selected().get_colour())
                    else:
                        pixel.set_colour(None) # erase

def __undo(self):
    # pushes current state onto redo stack, pops previous state off of undo stack
```

```
        if not self.__undo_stack.empty():
            self.__redo_stack.push(self.grid_to_hex())
            self.hex_to_grid(self.__undo_stack.pop())

    def __redo(self):
        # pushes current state onto undo stack, pops previous state off of redo stack
        if not self.__redo_stack.empty():
            self.__undo_stack.push(self.grid_to_hex())
            self.hex_to_grid(self.__redo_stack.pop())

    def __clear(self): # clears the grid
        for row in self.__grid:
            for pixel in row:
                pixel.set_colour(None)

    def get_changed(self): # return the changed status of the drawing grid
        return self.__changed

    def draw(self, screen): # draws the drawing grid and all its aspects to the screen
        pygame.draw.rect(screen, PALER_BACKGROUND, self.__border_rect)
        for row in self.__grid:
            for pixel in row:
                pixel.draw(screen)
        self.__undo_button.draw(screen)
        self.__redo_button.draw(screen)
        self.__clear_button.draw(screen)
        self.__erase_button.draw(screen)
        self.__colour_grid.draw(screen)
```

6.10 Game Classes File

game_classes.py

```
from random import randint
from math import trunc, sin, pi

import pygame

from constants import *
from utility_functions import split, get_key
from images import (item_images, bullet_image, default_enemy_images, fast_enemy_images,
                    flying_enemy_images, crow_enemy_images, tough_enemy_images,
                    spirit_enemy_images, exclamation)
from sounds import crow_sound, enemy_killed, default_shoot

#======================Cell Class======================#
# an 8 pixel by 8 pixel square that can have collision and an image
# makes up the grid of the game as plain grass, flowers, fences, and crates
class Cell():
    def __init__(self, pos):
        self.__image = None # an image for the cell to display
        self.__pos = pos
        self.__collision = False
        self.__shade = False # if the cell should cast a shadow
        self.__rect = pygame.Rect(pos, (EIGHT_PIXELS, EIGHT_PIXELS))
        self.__shade_surface = pygame.Surface((EIGHT_PIXELS, EIGHT_PIXELS), pygame.SRCALPHA)
        pygame.draw.rect(self.__shade_surface, (0,0,0,50), (0, 0, EIGHT_PIXELS, EIGHT_PIXELS))

    def has_image(self): # returns True if the cell has an image
        if self.__image:
            return True
        return False

    def get_collision(self): # returns the collision attribute of the cell
        return self.__collision

    def get_rect(self): # returns the rect of the cell
        return self.__rect

    def set_image(self, image): # sets the image of the cell
        self.__image = image

    def set_shade(self, shade): # sets the shade attribute of the cell
        self.__shade = shade

    def set_collision(self, collision): # sets the collision attribute of the cell
        self.__collision = collision

    def draw(self, screen): # draws the cell to the screen
        pygame.draw.rect(screen, GRASS_GREEN, self.__rect)
```

```
        if self.__shade:
            # shade surface is displayed one pixel down and across
            screen.blit(self.__shade_surface, (self.__pos[X]+PIXEL_RATIO,
                                               self.__pos[Y]+PIXEL_RATIO))
        if self.__image:
            screen.blit(self.__image, self.__rect)

#=====================Player Class=====================#
# a controllable character that can move and shoot
# one or two initialised for the game
class Player():
    def __init__(self, hex_string, game_rect, player=0):
        self.__player = player # 0 for single player, 1 for 1st of 2 player, 2 for 2nd player
        match player:
            case 0: # spawn single player at the center
                self.__initial_pos = pygame.math.Vector2((game_rect.center))
            case 1: # spawn first player a little to the left
                self.__initial_pos = pygame.math.Vector2((game_rect.centerx - EIGHT_PIXELS//2,
                                                          game_rect.centery))
            case 2: # spawn second player a little to the right
                self.__initial_pos = pygame.math.Vector2((game_rect.centerx + EIGHT_PIXELS//2,
                                                          game_rect.centery))
        self.__pos = self.__initial_pos.copy()
        self.__custom_character(hex_string) # turn the hex string into images
        self.__image = self.__front_image
        self.__immune_image = self.__front_immune
        self.__rect = self.__image.get_rect(center = self.__pos)
        self.__speed = PLAYER_SPEED
        self.__lives = PLAYER_LIVES
        self.__bullets = []
        self.__bullet_damage = PLAYER_DAMAGE # lots of variables, future items easy to implement
        self.__last_shot = 0
        self.__fire_rate = FIRE_RATE # frames between each shot
        self.__fire_rate_multiplier = 1
        self.__game_rect = game_rect

        self.__item = None
        self.__shoes = False
        self.__shoes_time = -1
        self.__shotgun = False
        self.__shotgun_time = -1
        self.__rapid_fire = False
        self.__rapid_fire_time = -1
        self.__backwards_shot = False
        self.__backwards_shot_time = -1
        self.__immunity = False
        self.__immunity_time = -1

        self.__spawned = True # used for two player when player has yet to respawn

        self.__new_rect = self.__rect.copy()
```

```
        self.__new_rect_x = self.__rect.copy()
        self.__new_rect_y = self.__rect.copy()

        self.__bullets_shot = 0

        self.__timer = 0

    def __custom_character(self, hex_string): # converts a custom hex string into images
        self.__front_image, self.__front_immune = self.__hex_to_image(FRONT_PATTERNS,
                                                                       hex_string)
        self.__back_image, self.__back_immune = self.__hex_to_image(BACK_PATTERNS,
                                                                     hex_string)
        self.__left_image, self.__left_immune = self.__hex_to_image(LEFT_PATTERNS,
                                                                     hex_string)
        self.__right_image, self.__right_immune = self.__hex_to_image(RIGHT_PATTERNS,
                                                                       hex_string)

    def __hex_to_image(self, patterns, hex_string): # converts hex string into an image
        # blank surface to draw pixels to
        image = pygame.Surface((EIGHT_PIXELS, EIGHT_PIXELS), pygame.SRCALPHA)
        # splits the hex codes referring to the selected colours into a list
        select_hex = split(hex_string[:COLOUR_DEPTH*len(patterns)], COLOUR_DEPTH)
        # splits the hex codes referring to the hat colours into a list of rows
        hat_hex = split(split(hex_string[COLOUR_DEPTH*len(patterns):], COLOUR_DEPTH), 8)

        # for each pixel pattern, draw its pixels onto the blank surface in the correct colours
        for i in range(len(patterns)):
            for row, column in patterns[i]:
                pygame.draw.rect(image, get_key(select_hex[i], BITMAP_DICTIONARY),
                                 pygame.rect.Rect((column*PIXEL_RATIO,row*PIXEL_RATIO),
                                                  (1*PIXEL_RATIO, 1*PIXEL_RATIO)))

        # for each hex code in the hat list, draw a pixel to the screen in the correct colour
        for i in range(len(hat_hex)):
            for j in range(len(hat_hex[0])):
                if get_key(hat_hex[i][j], BITMAP_DICTIONARY):
                    pygame.draw.rect(image, get_key(hat_hex[i][j], BITMAP_DICTIONARY),
                                     pygame.rect.Rect((j*PIXEL_RATIO,i*PIXEL_RATIO),
                                                      (1*PIXEL_RATIO, 1*PIXEL_RATIO)))

        # transparent white version of the image to draw when the player is immune to damage
        immune = pygame.mask.from_surface(image).to_surface(setcolor=(255,255,255,100),
                                                            unsetcolor=(0,0,0,0))

        return image, immune

    def __move(self, keys, collidables):
        velocity_x = 0 # the movement of the player's x position
        velocity_y = 0 # the movement of the player's y position

        # convert keyboard input into movement
```

```
        velocity_x, velocity_y = self.__move_input(velocity_x, velocity_y, keys)

        # rects approximate to integer values, not pixel perfect for collisions
        # dict used for each side and their positions in non-approximate forms
        player_sides = {'TOP'    : self.__pos[Y] - EIGHT_PIXELS/2,
                        'BOTTOM' : self.__pos[Y] + EIGHT_PIXELS/2,
                        'LEFT'   : self.__pos[X] - EIGHT_PIXELS/2,
                        'RIGHT'  : self.__pos[X] + EIGHT_PIXELS/2}

        # check and correct for collisions with the edges of the game
        velocity_x, velocity_y = self.__edge_collisions(player_sides, velocity_x, velocity_y)

        # check and correct for collisions with each collidable object in the game
        velocity_x, velocity_y = self.__collidables_collisions(player_sides, velocity_x,
                                                               velocity_y, collidables)

        return velocity_x, velocity_y

    def __move_input(self, velocity_x, velocity_y, keys): # converts keyboard input to movement
        if ((keys[pygame.K_w] and self.__player == 0)
                or (keys[pygame.K_w] and self.__player == 1)
                or (keys[pygame.K_9] and self.__player == 2)):
            velocity_y -= self.__speed # position is taken from the top left, so upwards is -y
            self.__image = self.__back_image # change the player image to be facing upwards
            self.__immune_image = self.__back_immune
        if ((keys[pygame.K_s] and self.__player == 0)
                or (keys[pygame.K_s] and self.__player == 1)
                or (keys[pygame.K_o] and self.__player == 2)):
            velocity_y += self.__speed
            self.__image = self.__front_image
            self.__immune_image = self.__front_immune
        if ((keys[pygame.K_a] and self.__player == 0)
                or (keys[pygame.K_a] and self.__player == 1)
                or (keys[pygame.K_i] and self.__player == 2)):
            velocity_x -= self.__speed
            self.__image = self.__left_image
            self.__immune_image = self.__left_immune
        if ((keys[pygame.K_d] and self.__player == 0)
                or (keys[pygame.K_d] and self.__player == 1)
                or (keys[pygame.K_p] and self.__player == 2)):
            velocity_x += self.__speed
            self.__image = self.__right_image
            self.__immune_image = self.__right_immune

        # if the player is moving diagonally, divide each velocity by root 2 to normalise
        if velocity_x != 0 and velocity_y != 0:
            velocity_x /= 2**0.5
            velocity_y /= 2**0.5

        if self.__shoes:
            velocity_x *= SHOE_MULTIPLIER
```

```
            velocity_y *= SHOE_MULTIPLIER

        return velocity_x, velocity_y

    def __edge_collisions(self, player_sides, velocity_x, velocity_y): # check edge collisions
        # if distance to an edge is less than the player moves and player is moving towards it
        # the amount moving towards the edge becomes the distance between edge and player
        if player_sides['LEFT'] - self.__game_rect.left < self.__speed and velocity_x < 0:
            velocity_x = self.__game_rect.left - player_sides['LEFT']
        elif self.__game_rect.right - player_sides['RIGHT'] < self.__speed and velocity_x > 0:
            velocity_x = self.__game_rect.right - player_sides['RIGHT']
        if player_sides['TOP'] - self.__game_rect.top < self.__speed and velocity_y < 0:
            velocity_y = self.__game_rect.top - player_sides['TOP']
        elif self.__game_rect.bottom - player_sides['BOTTOM'] < self.__speed and velocity_y >0:
            velocity_y = self.__game_rect.bottom - player_sides['BOTTOM']

        return velocity_x, velocity_y

    def __collidables_collisions(self, player_sides, velocity_x, velocity_y, collidables):
        # check and correct for collisions with each collidable object in the game
        # rects that show where the player is about to move
        # only only taking into account x, one for y, and one for both
        self.__new_rect_x.center = self.__pos + pygame.math.Vector2(velocity_x, 0)
        self.__new_rect_y.center = self.__pos + pygame.math.Vector2(0, velocity_y)
        self.__new_rect.center = self.__pos + pygame.math.Vector2(velocity_x, velocity_y)

        x = velocity_x # temporary variables so it doesn't change between loops
        y = velocity_y

        for collidable in collidables:
            collision_x = collidable.colliderect(self.__new_rect_x)
            collision_y = collidable.colliderect(self.__new_rect_y)
            # diagonal collision when the player only moves into collidable with both x and y
            # if you just looked at x and y separately, you would not see any collision
            diagonal = collidable.colliderect(self.__new_rect) and not (collision_x
                                                                        or collision_y)

            # adjust the motion of the player to the distance between them and the collidable
            if collision_x or diagonal:
                if x < 0:  # moving left
                    velocity_x = -(player_sides['LEFT'] - collidable.right)
                elif x > 0:  # moving right
                    velocity_x = collidable.left - player_sides['RIGHT']

            if collision_y or diagonal:
                if y < 0:  # moving up
                    velocity_y = -(player_sides['TOP'] - collidable.bottom)
                elif y > 0:  # moving down
                    velocity_y = collidable.top - player_sides['BOTTOM']

        return velocity_x, velocity_y
```

```
def __shoot(self, keys): # converts keyboard input into shooting
    bullet_x = 0 # direction of the bullet being shot
    bullet_y = 0
    x_offset = 0 # offset to match the position of the gun
    y_offset = 0

    # single player and 2nd player of two player both use the arrow keys for shooting
    if ((keys[pygame.K_UP] and (self.__player == 0 or self.__player == 2))
            or (keys[pygame.K_g] and self.__player == 1)):
        bullet_y -= 1
    if ((keys[pygame.K_DOWN] and (self.__player == 0 or self.__player == 2))
            or (keys[pygame.K_b] and self.__player == 1)):
        bullet_y += 1
    if ((keys[pygame.K_LEFT] and (self.__player == 0 or self.__player == 2))
            or (keys[pygame.K_v] and self.__player == 1)):
        bullet_x -= 1
    if ((keys[pygame.K_RIGHT] and (self.__player == 0 or self.__player == 2))
            or (keys[pygame.K_n] and self.__player == 1)):
        bullet_x += 1

    if bullet_y == -1:                  # if shooting up
        x_offset = PIXEL_RATIO * 2.5    # spawn bullet 2.5 pixels to the right
        y_offset = PIXEL_RATIO * -1     # and 1 pixel up (-1 pixel down)
        self.__image = self.__back_image
        self.__immune_image = self.__back_immune
    elif bullet_y == 1:
        x_offset = PIXEL_RATIO * -2.5
        y_offset = PIXEL_RATIO * 5
        self.__image = self.__front_image
        self.__immune_image = self.__front_immune
    if bullet_x == -1:
        x_offset = PIXEL_RATIO * -5
        y_offset = PIXEL_RATIO * 1.5
        self.__image = self.__left_image
        self.__immune_image = self.__left_immune
    elif bullet_x == 1:
        x_offset = PIXEL_RATIO * 4
        y_offset = PIXEL_RATIO * 1.5
        self.__image = self.__right_image
        self.__immune_image = self.__right_immune

    if bullet_x != 0 and bullet_y != 0: # diagonal
        bullet_x /= 2**0.5
        bullet_y /= 2**0.5

    # if player is trying to shoot and enough time has passed since the last shot, shoot
    if ((bullet_x != 0 or bullet_y != 0 )
            and (self.__timer - self.__last_shot >
                 self.__fire_rate * self.__fire_rate_multiplier)):
        offset = (x_offset, y_offset)
```

```
            direction = pygame.math.Vector2(bullet_x,bullet_y)
            bullet = Bullet(self.__pos + offset, direction, self.__bullet_damage, self.__timer)
            self.__bullets.append(bullet)
            self.__bullets_shot += 1
            if self.__shotgun:
                self.__shoot_shotgun(offset, direction)
            if self.__backwards_shot:
                self.__shoot_backwards_shot(offset, direction)
            self.__last_shot = self.__timer

    def __shoot_shotgun(self, offset, direction): # shoot 2 extra bullets at offset angles
        bullet1 = Bullet(self.__pos + offset, direction.rotate(-11.25),
                         self.__bullet_damage, self.__timer)
        bullet2 = Bullet(self.__pos + offset, direction.rotate(11.25),
                         self.__bullet_damage, self.__timer)
        self.__bullets.append(bullet1)
        self.__bullets.append(bullet2)
        self.__bullets_shot += 2

    def __shoot_backwards_shot(self, offset, direction): # extra bullet in opposite direction
        bullet = Bullet(self.__pos + offset, direction.rotate(180),
                        self.__bullet_damage, self.__timer)
        self.__bullets.append(bullet)
        self.__bullets_shot += 1
        if self.__shotgun:
            self.__shoot_shotgun(offset, -direction) # shoot shotgun backwards too

    def add_shoes(self): # gives the player shoes
        self.__shoes = True
        self.__shoes_time = self.__timer

    def add_shotgun(self): # gives the player shotgun
        self.__shotgun = True
        self.__shotgun_time = self.__timer

    def add_rapid_fire(self): # gives the player rapid fire
        self.__rapid_fire = True
        self.__rapid_fire_time = self.__timer

    def add_immunity(self, time): # gives the player immunity
        self.__immunity = True
        self.__immunity_time = self.__timer + time

    def add_backwards_shot(self): # gives the player backwards shot
        self.__backwards_shot = True
        self.__backwards_shot_time = self.__timer

    def increase_health(self, amount): # increase the player's health
        self.__lives += amount

    def empty_bullets(self): # reset the player's bullets
```

```
        self.__bullets = []

    def remove_bullet(self, bullet): # remove a bullet from the list
        self.__bullets.remove(bullet)

    def get_bullets_shot(self): # return the number of bullets shot
        return self.__bullets_shot

    def get_immunity(self): # return the immunity status
        return self.__immunity

    def get_spawned(self): # return the spawned status
        return self.__spawned

    def get_item(self): # return the item of the player
        return self.__item

    def get_lives(self): # return the health of the player
        return self.__lives

    def get_player(self): # return the player number
        return self.__player

    def get_rect(self): # return the rect of the player
        return self.__rect

    def get_bullets(self): # return the bullets list
        return self.__bullets

    def get_pos(self): # return the position of the player
        return self.__pos

    def set_item(self, item): # set the item of the player
        self.__item = item

    def set_immunity(self, immunity): # set the immunity status
        self.__immunity = immunity

    def set_spawned(self, spawned): # set the spawned status
        self.__spawned = spawned

    def update(self, collidables, other_player_rect=None): # update player and check input
        self.__timer += 1

        # fire rate slightly reduced if the player has shotgun
        # fire rate increased if the player has rapid fire
        self.__fire_rate_multiplier = ((RAPID_FIRE_MULTIPLIER if self.__rapid_fire else 1)
                                       * (SHOTGUN_RATE_MULTIPLIER if self.__shotgun else 1))


        for bullet in self.__bullets:
```

```
            # remove every bullet that has collided with a collidable object
            for collidable in collidables:
                if collidable.colliderect(bullet.get_rect()):
                    self.__bullets.remove(bullet)
                    break # could otherwise try to remove the bullet from the list twice
            bullet.update()
            # so that bullets don't last forever if they don't hit anything
            if self.__timer - bullet.get_spawn_time() > BULLET_LIFETIME:
                self.__bullets.remove(bullet)

        if self.__lives > 0 and self.__spawned:
            keys = pygame.key.get_pressed()
            velocity_x, velocity_y = self.__move(keys, collidables +
                                                 ([other_player_rect] if other_player_rect
                                                    else []))
            self.__pos += pygame.math.Vector2(velocity_x, velocity_y)
            self.__rect.center = self.__pos
            self.__shoot(keys)

        # check the lifetime of power ups
        if self.__shotgun and self.__timer - self.__shotgun_time >= SHOTGUN_LENGTH:
            self.__shotgun = False
        if self.__shoes and self.__timer - self.__shoes_time >= SHOES_LENGTH:
            self.__shoes = False
        if self.__rapid_fire and self.__timer - self.__rapid_fire_time >= RAPID_FIRE_LENGTH:
            self.__rapid_fire = False
        if self.__backwards_shot and (self.__timer - self.__backwards_shot_time >=
                                                                    BACKWARDS_SHOT_LENGTH):
            self.__backwards_shot = False
        if self.__immunity and self.__timer == self.__immunity_time:
            self.__immunity = False

    def hit(self, damage): # damage and reset the player
        if not self.__immunity:
            self.__lives -= damage
            self.__item = None
            self.__shoes = False
            self.__shotgun = False
            self.__rapid_fire = False
            self.__backwards_shot = False
            self.__pos = self.__initial_pos.copy()

    def draw(self, screen): # draw the player
        if self.__spawned:
            screen.blit(self.__image, (self.__pos[X] - EIGHT_PIXELS/2,
                                       self.__pos[Y] - EIGHT_PIXELS/2))
            if self.__immunity:
                # flash immunity image on and off every 0.5 seconds
                if FPS//2 < (self.__immunity_time - self.__timer) % FPS < FPS:
                    screen.blit(self.__immune_image, (self.__pos[X] - EIGHT_PIXELS/2,
                                                      self.__pos[Y] - EIGHT_PIXELS/2))
```

```
#======================Bullet Class======================#
# a bullet that travels in a given direction
# shot by the player during the game
class Bullet():
    def __init__(self, pos, direction, damage, spawn_time):
        self.__direction = direction
        self.__pos = pygame.math.Vector2(pos)
        self.__speed = BULLET_SPEED
        self.__image = bullet_image
        self.__damage = damage
        self.__rect = self.__image.get_rect(center = self.__pos)
        self.__spawn_time = spawn_time
        default_shoot.play()

    def get_rect(self): # return the rect of the bullet
        return self.__rect

    def get_damage(self): # return the damage of the bullet
        return self.__damage

    def get_spawn_time(self): # return the spawn time of the bullet
        return self.__spawn_time

    def update(self): # update the position of the bullet
        self.__pos += self.__direction * self.__speed
        self.__rect.center = self.__pos

    def draw(self, screen): # draw the bullet
        screen.blit(self.__image, self.__rect.topleft)

#======================Item Class======================#
# a temporary item that has an image and a type and a rect
class Item():
    def __init__(self, pos, type):
        self.__type = type
        self.__image = item_images[self.__type]
        self.__rect = self.__image.get_rect(center = pos)
        self.__timer = 0 # how many frames the item has been around for

    def get_type(self): # return the type of the item
        return self.__type

    def get_rect(self): # return the rect of the item
        return self.__rect

    def update(self): # check how long the item has been around for
        self.__timer += 1
        if self.__timer > ITEM_TIME:
            return True # true returned if the item should be removed from the list
        elif self.__timer > ITEM_TIME - ITEM_FLASH_TIME:
```

```
            if self.__timer % (ITEM_FLASH_TIME//9) == 0: # flip the image on and off 9 times
                self.__image = None if self.__image else item_images[self.__type]
        return False

    def draw(self, screen): # draw the item
        if self.__image:
            screen.blit(self.__image, self.__rect)

#======================Score Class======================#
# a temporary display of a killed enemie's score
class Score():
    def __init__(self, font, colour, score, rect, alpha=255):
        self.__font = font.new_colour_copy(colour, alpha=alpha)
        self.__score = str(score)
        self.__rect = rect
        self.__timer = 0

    def update(self): # check how long the score has been around for
        self.__timer += 1
        if self.__timer > SCORE_LENGTH:
            return True # true returned if the score should be removed from the list
        return False

    def draw(self, screen): # draw the score
        self.__font.render(screen, self.__score, (self.__rect.centerx,
                                                   self.__rect.top + PIXEL_RATIO),
                                                   alignment=CENTER)

#======================Enemy Class======================#
# a base class for all the different enemy types
class Enemy():
    def __init__(self, pos, settings, initial_image, flying):
        self._pos = pygame.math.Vector2(pos)
        self._prev_pos = self._pos.copy() # pos reverted to previous pos if a collision occurs
        self._health = settings["HEALTH"] # settings is a dict of the health, speed, and score
        self._speed = settings["SPEED"]
        self.__score = settings["SCORE"]
        self._image = initial_image
        self._timer = 0
        self.__flying = flying # whether the enemy is flying or not
        self._rect = self._image.get_rect(center = self._pos)
        self.__red = False
        self.__hit_timer = 0

    def get_flying(self): # return whether the enemy is flying or not
        return self.__flying

    def get_score(self): # return the score of the enemy
        return self.__score

    def get_rect(self): # return the rect of the enemy
```

```
        return self._rect

    def get_health(self): # return the health of the enemy
        return self._health

    def hit(self, damage=1): # damage the enemy
        self._health -= damage
        self.__red = True # temporarily red after hit
        self.__hit_timer = 0
        if self._health == 0:
            enemy_killed.play()

    def draw(self, screen):
        self.__hit_timer += 1
        if self.__red and self.__hit_timer == HIT_TIME:
            self.__red = False
        screen.blit(self._image, (self._pos[X] - EIGHT_PIXELS/2, self._pos[Y] - EIGHT_PIXELS/2))
        if self.__red:
            red_image = pygame.mask.from_surface(self._image).to_surface(setcolor=HIT_COLOUR,
                                                                          unsetcolor=(0,0,0,0))
            screen.blit(red_image, (self._pos[X] - EIGHT_PIXELS/2,
                                    self._pos[Y] - EIGHT_PIXELS/2))


#======================Default Enemy Class======================#
# ground enemy, the first enemy the player sees
# moves towards the player in straight lines with random influence
class DefaultEnemy(Enemy):
    def __init__(self, pos, direction):
        super().__init__(pos, DEFAULT_ENEMY, default_enemy_images[direction][1], False)
        self.__direction_change_time = 0 # last time the direction was changed
        self.__random_time = 0 # time until direction is next changed
        self.__direction = direction # direction the enemy is currently facing
        self.__images = default_enemy_images # 2d array of animation frames

    def __move(self): # return velocities for the enemy based on its direction
        if self.__direction == LEFT:
            velocity_x = -self._speed
            velocity_y = 0
        elif self.__direction == RIGHT:
            velocity_x = self._speed
            velocity_y = 0
        elif self.__direction == UP:
            velocity_x = 0
            velocity_y = -self._speed
        elif self.__direction == DOWN:
            velocity_x = 0
            velocity_y = self._speed
        return velocity_x, velocity_y

    def __change_direction(self, player_pos): # check and change the direction of the enemy
```

```
        x_distance = player_pos[X] - self._pos[X]
        y_distance = player_pos[Y] - self._pos[Y]
        if abs(x_distance) > abs(y_distance): # direction initially from distance to player
            if x_distance < 0:
                self.__direction = LEFT
            else:
                self.__direction = RIGHT
        else:
            if y_distance < 0:
                self.__direction = UP
            else:
                self.__direction = DOWN
        self.__direction_change_time = self._timer

        # if the enemy is between 2 and 8 cells from the player, chance for random direction
        if 2*EIGHT_PIXELS < (x_distance**2 + y_distance**2)**0.5 < 8*EIGHT_PIXELS:
            # random time before next direction check between 15 and 60 frames
            self.__random_time = randint(FPS//4, FPS)
            random_int = randint(-1,3) # 2/5 chance to move randomly
            if random_int <= 1: # accepts -1, 0, or 1, added to direction to turn left / right
                self.__direction = (self.__direction + random_int) % 4 # mod 4 for 4 directions

    def __check_collisions(self, velocity_x, velocity_y, game_rect, collidables, enemy_rects):
        # doesn't need to be pixel perfect like for the player
        # enemy returned to their previous position if a collision is detected
        collision = False

        # collision is true if any collision between the enemy and a collidable object
        if self._rect.collidelistall(collidables):
            collision = True

        # collision is true if any collision between the enemy and the game sides
        if ((self._rect.left <= game_rect.left and velocity_x < 0)
                or (self._rect.top <= game_rect.top and velocity_y < 0)
                or (self._rect.right >= game_rect.right and velocity_x > 0)
                or (self._rect.bottom >= game_rect.bottom and velocity_y > 0)):
            collision = True

        enemy_rects.remove(self._rect) # remove the enemy's rect from the list
        # collision is true if enemy collides with another enemy
        if self._rect.collidelistall(enemy_rects):
            collision = True

        if collision: # if there is a collision, return the enemy to their previous position
            self._pos = self._prev_pos.copy()
            self._rect.center = self._pos
            self.__random_time = 6 # reduce random time if colliding

    def update(self, player_pos, game_rect, collidables, enemy_rects, player2_pos=None):
        # update and move the enemy
        # player_pos always supplied in one player
```

```
        # player_pos only supplied if player 1 is alive in two player
        # player2_pos only supplied if player 2 is alive in two player
        if player2_pos: # check who is closest if there is a second position to look at
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS # arbitrarily large
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
                player_pos = player2_pos

        self._timer += 1

        if player_pos:
            if self._timer - self.__direction_change_time > self.__random_time:
                self.__change_direction(player_pos)
            velocity_x, velocity_y = self.__move()
            self._image = self.__images[self.__direction][trunc(((self._timer * 4)/FPS) %
                                                                len(self.__images[0]))]
            self._prev_pos = self._pos.copy() #.copy() so that they aren't linked
            self._pos += (velocity_x, velocity_y)
            self._rect.center = self._pos # update the position of the rect
            self.__check_collisions(velocity_x, velocity_y, game_rect, collidables, enemy_rects)

#=====================Fast Enemy Class=====================#
# ground enemy, the second enemy the player will see
# runs straight until it meets the player in either x or y, then changes direction
class FastEnemy(Enemy):
    def __init__(self, pos, direction):
        super().__init__(pos, FAST_ENEMY, fast_enemy_images[direction][0], False)
        self.__direction = None
        self.__random_move_delay = 0 # to keep moving in a direction for n frames
        self.__images = fast_enemy_images

    def __move(self): # return velocities for the enemy based on its direction
        if self.__direction == LEFT:
            velocity_x = -self._speed
            velocity_y = 0
        elif self.__direction == RIGHT:
            velocity_x = self._speed
            velocity_y = 0
        elif self.__direction == UP:
            velocity_x = 0
            velocity_y = -self._speed
        elif self.__direction == DOWN:
            velocity_x = 0
            velocity_y = self._speed
        return velocity_x, velocity_y
```

```
def __check_direction(self, player_pos): # check and change the direction of the enemy
    x_distance = player_pos[X] - self._pos[X]
    y_distance = player_pos[Y] - self._pos[Y]
    if self.__direction == None:
        if abs(x_distance) > abs(y_distance):
            self.__check_x(x_distance)
        else:
            self.__check_y(y_distance)
    elif self.__direction == UP and y_distance > 0: # if reached player in y
        self.__check_x(x_distance) # swap to x movement
    elif self.__direction == DOWN and y_distance < 0:
        self.__check_x(x_distance)
    elif self.__direction == LEFT and x_distance > 0: # if reached player in x
        self.__check_y(y_distance) # swap to y movement
    elif self.__direction == RIGHT and x_distance < 0:
         self.__check_y(y_distance)

def __check_x(self, x_distance): # pick x direction based on an x distance
    if x_distance < 0:
        self.__direction = LEFT
    else:
        self.__direction = RIGHT

def __check_y(self, y_distance): # pick y direction based on a y distance
    if y_distance < 0:
        self.__direction = UP
    else:
        self.__direction = DOWN

def __check_collisions(self, velocity_x, velocity_y, game_rect, collidables, enemy_rects):
    collision = False

    if self._rect.collidelistall(collidables):
        collision = True

    if ((self._rect.left <= game_rect.left and velocity_x < 0)
            or (self._rect.top <= game_rect.top and velocity_y < 0)
            or (self._rect.right >= game_rect.right and velocity_x > 0)
            or (self._rect.bottom >= game_rect.bottom and velocity_y > 0)):
        collision = True

    enemy_rects.remove(self._rect)

    if self._rect.collidelistall(enemy_rects):
        collision = True

    if collision:
        self._pos = self._prev_pos.copy()
        self._rect.center = self._pos
        random_int = randint(-1,3)
        if random_int <= 1:
```

```
                self.__direction = (self.__direction + random_int) % 4
            # if collision, possibly moves in random direction for 0.5 seconds
            self.__random_move_delay = FPS//2

    def update(self, player_pos, game_rect, collidables, enemy_rects, player2_pos=None):
        # update and move the enemy
        if player2_pos: # check who is closest
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS # arbitrarily large
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
                player_pos = player2_pos

        self._timer += 1

        if player_pos:
            if self.__random_move_delay == 0:
                self.__check_direction(player_pos)
            self._prev_pos = self._pos.copy()
            velocity_x, velocity_y = self.__move()
            self._image = self.__images[self.__direction][trunc(((self._timer * 8)/FPS) %
                                                                len(self.__images[0]))]
            self._pos += (velocity_x, velocity_y)
            self._rect.center = self._pos
            self.__check_collisions(velocity_x, velocity_y, game_rect, collidables, enemy_rects)
            if self.__random_move_delay > 0:
                self.__random_move_delay -= 1

#======================Flying Enemy Class======================#
# flying enemy, the third enemy the player will see
# moves directly towards the player with speed varying sinusoidally
class FlyingEnemy(Enemy):
    def __init__(self, pos, direction):
        super().__init__(pos, FLYING_ENEMY, flying_enemy_images[direction][0], True)
        self.__cycle = FPS * 3 # time period of the sine wave, varies over 3 seconds
        self.__direction = direction
        self.__images = flying_enemy_images

    def __move(self, player_pos): # return velocities for the enemy based on player position
        velocity_x = 0
        velocity_y = 0
        x_distance = player_pos[X] - self._pos[X]
        y_distance = player_pos[Y] - self._pos[Y]
        distance = (x_distance**2 + y_distance**2)**0.5 # pythagoras
        if distance != 0:
            velocity_x += self._speed * (x_distance / distance) # portion of direction in x
            velocity_y += self._speed * (y_distance / distance) # portion of direction in y
```

```
        return velocity_x, velocity_y

    def __check_direction(self, velocity_x, velocity_y): # alter direction based on velocities
        if abs(velocity_y) + 0.5 > abs(velocity_x): # favour image being up or down
            if velocity_y > 0:
                self.__direction = DOWN
            else:
                self.__direction = UP
        else:
            if velocity_x > 0:
                self.__direction = RIGHT
            else:
                self.__direction = LEFT

    def update(self, player_pos, player2_pos=None): # update and move the enemy
        if player2_pos: # check who is closest
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS # arbitrarily large
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
                player_pos = player2_pos

        self._timer += 1

        if player_pos:
            # sinusoidal speed variation
            self._speed = ((sin((self._timer * 2*pi) / self.__cycle) / 2)
                           * PIXEL_RATIO/5 * FPS/60 + FLYING_ENEMY["SPEED"])
            velocity_x, velocity_y = self.__move(player_pos)
            self.__check_direction(velocity_x, velocity_y) # so direction is correct
            # no checking collisions for flying enemies
            self._image = self.__images[self.__direction][trunc(((self._timer * 4)/FPS) %
                                                                len(self.__images[0]))]
            self._pos += velocity_x, velocity_y
            self._rect.center = self._pos

#=====================Crow Enemy Class=====================#
# flying enemy, the fourth enemy the player will see
# moves very fast in a straight line across the game
# warns the player of its position before moving
class CrowEnemy(Enemy):
    def __init__(self, pos, direction, game_rect):
        super().__init__(pos, CROW_ENEMY, crow_enemy_images[direction], True)
        # crow enemy has a transparent version of the same image that follows behind them
        # adds motion blur effect
        self.__blur_image = self._image.copy()
        self.__blur_image.set_alpha(100)
```

```
        self.__blur_rect = self._rect.copy()
        # larger rect created to detect if the crow has flown far off screen
        self.__large_rect = pygame.Rect((self._rect.x - EIGHT_PIXELS,
                                         self._rect.y - EIGHT_PIXELS),
                                        (self._rect.width + 2*EIGHT_PIXELS,
                                         self._rect.height + 2*EIGHT_PIXELS))
        self.__game_rect = game_rect
        self.__set_velocities(direction)
        # exclamation mark is drawn for a few seconds before the enemy appears
        # position is calculated by adding 8 pixels of movement in the enemy's direction
        self.__exclamation_pos = (pygame.math.Vector2(self._rect.topleft) +
                            (pygame.math.Vector2(self.__velocity)*(EIGHT_PIXELS/self._speed)))

    def __set_velocities(self, direction):
        blur_distance = CROW_BLUR_DISTANCE
        if direction == RIGHT: # velocity set at the beginning as enemy doesn't change velocity
            self.__velocity = (self._speed, 0)
            self.__blur_pos = (-blur_distance, 0)
        elif direction == LEFT:
            self.__velocity = (-self._speed, 0)
            self.__blur_pos = (blur_distance, 0)
        elif direction == DOWN:
            self.__velocity = (0, self._speed)
            self.__blur_pos = (0, -blur_distance)
        elif direction == UP:
            self.__velocity = (0, -self._speed)
            self.__blur_pos = (0, blur_distance)

    def update(self, *args): # *args to get and disregard any other arguments passed in
        if self._timer == 0:
            crow_sound.play() # sound played here otherwise it plays at initialisation
        self._timer += 1
        if self._timer > CROW_PAUSE: # paused for a few seconds before moving
            self._pos += self.__velocity
            self._rect.center = self._pos
            self.__blur_rect.center = self._pos + self.__blur_pos
            self.__large_rect.center = self._pos
            if not self.__large_rect.colliderect(self.__game_rect):
                self._health = CROW_OFFSCREEN # constant used so enemy death sound is not played

    def draw(self, screen): # draw the enemy, polymorphism necessary for blur image
        if self._timer < CROW_PAUSE // 1.5:
            screen.blit(exclamation, self.__exclamation_pos) # visible for 2/3 of pause time
        screen.blit(self.__blur_image, self.__blur_rect)
        screen.blit(self._image, self._rect)

#======================Tough Enemy Class======================#
# ground enemy, the fifth enemy the player will see
# similar movement to the default enemy but with less random movement
class ToughEnemy(Enemy):
    def __init__(self, pos, direction):
```

```
        super().__init__(pos, TOUGH_ENEMY, tough_enemy_images[direction][0], False)
        self.__direction_change_time = 0
        self.__random_time = 0
        self.__direction = 0
        self.__images = tough_enemy_images

    def __move(self): # return velocities for the enemy based on its direction
        if self.__direction == LEFT:
            velocity_x = -self._speed
            velocity_y = 0
        elif self.__direction == RIGHT:
            velocity_x = self._speed
            velocity_y = 0
        elif self.__direction == UP:
            velocity_x = 0
            velocity_y = -self._speed
        elif self.__direction == DOWN:
            velocity_x = 0
            velocity_y = self._speed
        return velocity_x, velocity_y

    def __change_direction(self, player_pos): # check and change the direction of the enemy
        x_distance = player_pos[X] - self._pos[X]
        y_distance = player_pos[Y] - self._pos[Y]
        if abs(x_distance) > abs(y_distance):
            if x_distance < 0:
                self.__direction = LEFT
            else:
                self.__direction = RIGHT
        else:
            if y_distance < 0:
                self.__direction = UP
            else:
                self.__direction = DOWN
        self.__direction_change_time = self._timer

        if 2*EIGHT_PIXELS < (x_distance**2 + y_distance**2)**0.5 < 8*EIGHT_PIXELS:
            # random time before next direction check between 20 and 80 frames
            self.__random_time = randint(20,80)
            random_int = randint(-1,7) # 2/7 chance to move randomly
            if random_int <= 1:
                self.__direction = (self.__direction + random_int) % 4

    def __check_collisions(self, velocity_x, velocity_y, game_rect, collidables, enemy_rects):
        collision = False

        if self._rect.collidelistall(collidables):
            collision = True

        if ((self._rect.left <= game_rect.left and velocity_x < 0)
                or (self._rect.top <= game_rect.top and velocity_y < 0)
```

```
                or (self._rect.right >= game_rect.right and velocity_x > 0)
                or (self._rect.bottom >= game_rect.bottom and velocity_y > 0)):
            collision = True

        enemy_rects.remove(self._rect)
        if self._rect.collidelistall(enemy_rects):
            collision = True

        if collision:
            self._pos = self._prev_pos.copy()
            self._rect.center = self._pos
            self.__random_time = 5 # reduce random time if colliding

    def update(self, player_pos, game_rect, collidables, enemy_rects, player2_pos=None):
        if player2_pos: # check who is closest
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS # arbitrarily large
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
                player_pos = player2_pos

        self._timer += 1

        if player_pos:
            if self._timer - self.__direction_change_time > self.__random_time:
                self.__change_direction(player_pos)
            velocity_x, velocity_y = self.__move()
            self._image = self.__images[self.__direction][trunc(((self._timer * 3)/FPS) %
                                                                len(self.__images[0]))]
            self._prev_pos = self._pos.copy()
            self._pos += (velocity_x, velocity_y)
            self._rect.center = self._pos
            self.__check_collisions(velocity_x, velocity_y, game_rect, collidables, enemy_rects)

#======================Spirit Enemy Class======================#
# flying enemy, the sixth and final enemy the player will see
# similar movement to the flying enemy but without the sinusoidal speed
class SpiritEnemy(Enemy):
    def __init__(self, pos, direction):
        super().__init__(pos, SPIRIT_ENEMY, spirit_enemy_images[direction], True)
        self.__images = spirit_enemy_images
        self.__direction = direction

    def __move(self, player_pos): # return velocities based on the player position
        velocity_x = 0
        velocity_y = 0
        x_distance = player_pos[X] - self._pos[X]
```

```
        y_distance = player_pos[Y] - self._pos[Y]
        distance = (x_distance**2 + y_distance**2)**0.5 # pythagoras
        if distance != 0:
            velocity_x += self._speed * (x_distance / distance) # portion of direction in x
            velocity_y += self._speed * (y_distance / distance) # portion of direction in y
        return velocity_x, velocity_y

    def __check_direction(self, velocity_x, velocity_y): # alter direction based on velocities
        if abs(velocity_x) > abs(velocity_y) + 0.5: # favour image being up or down
            if velocity_x > 0:
                self.__direction = RIGHT
            else:
                self.__direction = LEFT
        else:
            if velocity_y > 0:
                self.__direction = DOWN
            else:
                self.__direction = UP

    def update(self, player_pos, player2_pos=None): # update and move the enemy
        if player2_pos: # check who is closest
            if player_pos:
                player1_distance = ((player_pos[X] - self._pos[X])**2 +
                                    (player_pos[Y] - self._pos[Y])**2)**0.5
            else:
                player1_distance = GAME_WIDTH*EIGHT_PIXELS # arbitrarily large
            player2_distance = ((player2_pos[X] - self._pos[X])**2 +
                                (player2_pos[Y] - self._pos[Y])**2)**0.5
            if player2_distance <= player1_distance:
                player_pos = player2_pos

        self._timer += 1

        if player_pos:
            velocity_x, velocity_y = self.__move(player_pos)
            self.__check_direction(velocity_x, velocity_y)
            self._image = self.__images[self.__direction]
            self._pos += velocity_x, velocity_y
            self._rect.center = self._pos
```

6.11 Game Class File

game.py

```
from math import trunc, ceil
from random import randint, choice

import pygame

from constants import *
from utility_classes import Queue, Font
from game_classes import (Cell, Player, Item, Score, DefaultEnemy, FastEnemy,
                          FlyingEnemy, CrowEnemy, ToughEnemy, SpiritEnemy)
from images import (white_flowers1_image, white_flowers2_image, grass1_image,
                    grass2_image, grass3_image, crate_image, small_font_image,
                    huge_font_image, item_images, fences_image)
from sounds import player_hit_sound, player_death_sound, crate_thud, item_sounds

#=====================Game Class=====================#
class Game():
    def __init__(self, pos, character_hex, initial_obstacles, players=1, player2_hex=None):
        self.__width = GAME_WIDTH
        self.__height = GAME_HEIGHT
        self.__rect = pygame.Rect((0,0),(self.__width*EIGHT_PIXELS,self.__height*EIGHT_PIXELS))
        self.__grid = [[Cell((i*EIGHT_PIXELS, j*EIGHT_PIXELS)) for i in range(self.__width)]
                                                               for j in range(self.__height)]
        self.__players = players
        if players == 1:
            self.__player = Player(character_hex, self.__rect)
        elif players == 2:
            self.__player1 = Player(character_hex, self.__rect, 1)
            self.__player2 = Player(player2_hex, self.__rect, 2)
            self.__player1_respawn = -1 # the frame on which player 1 will respawn
            self.__player2_respawn = -1
        self.__pos = pos

        self.__small_font = Font(small_font_image, CHARACTER_LIST, WHITE, 2*PIXEL_RATIO)
        self.__countdown_font = Font(huge_font_image, CHARACTER_LIST_U,
                                     WHITE, 4*PIXEL_RATIO, alpha=230)

        # for each initial obstacle coordinate, add collision to the cell it represents
        for row, column in initial_obstacles:
            self.__grid[row][column].set_collision(True)

        self.__collidable_rects = self.__collidable_rects_list()

        # place random background tiles
        self.__place_random(grass1_image, randint(1,4), 2)
        self.__place_random(grass2_image, randint(1,4), 1)
        self.__place_random(grass3_image, randint(1,4), 2)
        self.__place_random(white_flowers1_image, randint(1,3), 3)
```

```
        self.__place_random(white_flowers2_image, randint(1,2), 3)

        self.__items = []
        self.__enemies = []
        self.__enemies_to_spawn = [] # list that gets looped through to try and spawn enemies
        self.__scores = []

        self.__enemy_queue = Queue()
        self.__enemy_rects = self.__enemy_rects_list()

        self.__countdown = 3*FPS

        self.__time_score = 0 # score gained over time
        self.__enemy_score = 0 # score gained by killing enemies

        self.__enemies_killed = 0 # number of enemies killed
        self.__items_used = 0 # number of items used

        self.__item_countdown = 0 # countdown between two items spawning
        self.__crate_countdown = 0 # delay after wave before crate
        self.__enemy_countdown = 0 # delay before the next enemies spawn
        self.__timer = 0 # overall frame counter that increments every frame
        self.__wave_index = -1  # wave number, used to set the difficulty of the wave

        self.__shake = False # if the screen should be shaking
        self.__bomb_time = 0 # the last time (frame number) a bomb was used
        self.__time_freeze = False # if time should be frozen
        self.__time_freeze_time = 0 # the last time a time freeze was used
        self.__freeze_surface = pygame.Surface((self.__rect.width, self.__rect.height),
                                               pygame.SRCALPHA)

    def __place_random(self, image, number, min_distance,
                       collision=False, center_spawn=True): # place a random image on the grid
        available = [] # list of available coordinates

        for row in range(min_distance, self.__height - min_distance):
            for column in range(min_distance, self.__width - min_distance):
                if self.__check_valid_placement(row, column, collision, center_spawn):
                    available.append((row, column)) # append all valid coordinates to the list

        for _ in range(number):
            if len(available) > 0:
                row, column = choice(available) # random coordinate from the list
                self.__grid[row][column].set_image(image)
                if collision:
                    self.__grid[row][column].set_collision(True)
                    self.__grid[row][column].set_shade(True)
                available.remove((row, column))

        if collision:
            self.__collidable_rects = self.__collidable_rects_list() # update collision list
```

```
def __check_valid_placement(self, row, column, collision, center_spawn):
    # check if there is already a collidable object in the position
    if self.__grid[row][column].get_collision():
        return False

    # check if it shouldn't spawn in the center and it is in the center
    if not center_spawn and (row in [trunc((GAME_WIDTH-1)/2), GAME_WIDTH//2]
                             and column in [trunc((GAME_HEIGHT-1)/2), GAME_HEIGHT//2]):
        return False

    # check if the position collides with a player
    if (self.__players == 1 and collision
            and self.__grid[row][column].get_rect().colliderect(self.__player.get_rect())):
        return False
    elif (self.__players == 2 and collision
          and (self.__grid[row][column].get_rect().colliderect(self.__player1.get_rect())
           or self.__grid[row][column].get_rect().colliderect(self.__player2.get_rect()))):
        return False

    return True # otherwise, placement is valid, return True

def __collidable_rects_list(self): # return a list of rects for cells with collisions
    # loops through every coordinate in the grid and checks for their collision
    # recursively combines adjacent rects into singular wider / taller rects
    collidable_rects = [] # list of rects
    row = 0
    column = 0
    checked = []
    for row in range(self.__height):
        for column in range(self.__width):
            if (row, column) not in checked and self.__grid[row][column].get_collision():
                rect = self.__grid[row][column].get_rect().copy() # base rect

                # check and adjust the width for adjacent cells horizontally
                checked_columns = []
                rect.width, checked_columns = self.__check_next_hor(row, column,
                                                                    checked_columns,
                                                                    checked,
                                                                    EIGHT_PIXELS)
                for checked_column in checked_columns:
                    checked.append((row, checked_column))

                if len(checked_columns) == 1: # if nothing was found in the horizontal
                    # check and adjust the height for adjacent cells vertically
                    checked_rows = []
                    rect.height, checked_rows = self.__check_next_vert(row, column,
                                                                       checked_rows,
                                                                       checked,
                                                                       EIGHT_PIXELS)
                    for checked_row in checked_rows:
```

```
                        checked.append((checked_row, column))

                collidable_rects.append(rect)
        return collidable_rects

    def __check_next_hor(self, row, column, checked_columns, checked, width):
        # checks the next coordinate horizontally in the grid
        # if it exists, hasn't been checked, and has a collision, the function calls itself
        # once an invalid coordinate is found, function unwinds and adds to the width per call
        try:
            checked_columns.append(column)
            if (row, column+1) not in checked:
                if self.__grid[row][column + 1].get_collision():
                    width, checked_columns = self.__check_next_hor(row, column + 1,
                                                                   checked_columns,
                                                                   checked, width)
                    return width + EIGHT_PIXELS, checked_columns
        except IndexError:
            pass # to move onto the return
        return width, checked_columns # base case

    def __check_next_vert(self, row, column, checked_rows, checked, height):
        # checks the next coordinate vertically in the grid
        # if it exists, isn't checked, and has a collision, the function calls itself
        # once an invalid coordinate is found, function unwinds and adds to the height per call
        try:
            checked_rows.append(row)
            if (row + 1, column) not in checked:
                if self.__grid[row + 1][column].get_collision():
                    height, checked_rows = self.__check_next_vert(row + 1, column,
                                                                  checked_rows,
                                                                  checked, height)
                    return height + EIGHT_PIXELS, checked_rows
        except IndexError:
            pass # to move onto the return
        return height, checked_rows # base case

    def __enemy_rects_list(self): # returns a list of rects of all ground enemies
        rect_list = []
        for enemy in self.__enemies:
            if not enemy.get_flying():
                rect_list.append(enemy.get_rect())
        return rect_list

    def __enemy_group(self, enemy_class, amount): # returns dict of enemies and difficulty
        enemies = []
        spawn_side = randint(0,3) # random side for spawning
        side_positions = [2, 1, 3, 0] # prioritise spawning enemies in the middle

        for i in range(amount):
            if spawn_side == UP:
```

```
                enemies.append(enemy_class((self.__rect.centerx - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS,
                                            self.__rect.top + EIGHT_PIXELS/2),
                                           direction=(spawn_side+2)%4))
            elif spawn_side == LEFT:
                enemies.append(enemy_class((self.__rect.left + EIGHT_PIXELS/2,
                                            self.__rect.centery - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS),
                                           direction=(spawn_side+2)%4))
            elif spawn_side == DOWN:
                enemies.append(enemy_class((self.__rect.centerx - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS,
                                            self.__rect.bottom - EIGHT_PIXELS/2),
                                           direction=(spawn_side+2)%4))
            elif spawn_side == RIGHT:
                enemies.append(enemy_class((self.__rect.right - EIGHT_PIXELS/2,
                                            self.__rect.centery - (3/2)*(EIGHT_PIXELS)
                                                    + side_positions[i%4]*EIGHT_PIXELS),
                                           direction=(spawn_side+2)%4))

        difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score() # sum of scores is used as measurement of difficulty

        return {'enemies':enemies, 'difficulty':difficulty}

    def __fast_enemy(self, amount): # returns dict of fast enemies and difficulty
        enemies = []
        for _ in range(amount):
            spawn_side = randint(0,3) # random side for spawning
            side_position = randint(0,3) # random position on that side
            if spawn_side == UP:
                enemies.append(FastEnemy(
                    (self.__rect.centerx - (3/2)*(EIGHT_PIXELS) + side_position*EIGHT_PIXELS,
                     self.__rect.top + EIGHT_PIXELS/2),
                    direction=(spawn_side+2)%4))
            elif spawn_side == LEFT:
                enemies.append(FastEnemy(
                    (self.__rect.left + EIGHT_PIXELS/2,
                     self.__rect.centery - (3/2)*(EIGHT_PIXELS) + side_position*EIGHT_PIXELS),
                    direction=(spawn_side+2)%4))
            elif spawn_side == DOWN:
                enemies.append(FastEnemy(
                    (self.__rect.centerx - (3/2)*(EIGHT_PIXELS) + side_position*EIGHT_PIXELS,
                     self.__rect.bottom - EIGHT_PIXELS/2),
                    direction=(spawn_side+2)%4))
            elif spawn_side == RIGHT:
                enemies.append(FastEnemy(
                    (self.__rect.right - EIGHT_PIXELS/2,
                     self.__rect.centery - (3/2)*(EIGHT_PIXELS) + side_position*EIGHT_PIXELS),
                    direction=(spawn_side+2)%4))
```

```
        difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score()

        return {'enemies':enemies, 'difficulty':difficulty}

    def __flying_enemy(self, amount):  # returns dict of flying enemies and difficulty
        enemies = []
        for _ in range(amount):
            side = randint(0,3)
            position = randint(1, GAME_WIDTH-2) # random position along an edge
            if side == RIGHT:
                enemies.append(FlyingEnemy((self.__rect.right + EIGHT_PIXELS//2,
                                            position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == LEFT:
                enemies.append(FlyingEnemy((self.__rect.left - EIGHT_PIXELS//2,
                                            position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == DOWN:
                enemies.append(FlyingEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                            self.__rect.bottom + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == UP:
                enemies.append(FlyingEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                            self.__rect.top - EIGHT_PIXELS//2),
                                           direction=(side+2)%4))

        difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score()

        return {'enemies':enemies, 'difficulty':difficulty}

    def __crow_enemy(self): # returns dict of a crow enemy and difficulty
        enemies = []
        side = randint(0,3)
        position = randint(1, GAME_WIDTH-2) # random position along an edge
        if side == RIGHT:
            enemies.append(CrowEnemy((self.__rect.right + EIGHT_PIXELS//2,
                                      position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                     (side+2)%4, self.__rect))
        elif side == LEFT:
            enemies.append(CrowEnemy((self.__rect.left - EIGHT_PIXELS//2,
                                      position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                     (side+2)%4, self.__rect))
        elif side == DOWN:
            enemies.append(CrowEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                      self.__rect.bottom + EIGHT_PIXELS//2),
                                     (side+2)%4, self.__rect))
```

```
        elif side == UP:
            enemies.append(CrowEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                      self.__rect.top - EIGHT_PIXELS//2),
                                     (side+2)%4, self.__rect))

        difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score()

        return {'enemies':enemies, 'difficulty':difficulty}

    def __spirit_enemy(self, amount):  # returns dict of spirit enemies and difficulty
        enemies = []
        for _ in range(amount):
            side = randint(0,3)
            position = randint(1, GAME_WIDTH-2) # random position along an edge
            if side == RIGHT:
                enemies.append(SpiritEnemy((self.__rect.right + EIGHT_PIXELS//2,
                                            position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == LEFT:
                enemies.append(SpiritEnemy((self.__rect.left - EIGHT_PIXELS//2,
                                            position*EIGHT_PIXELS + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == DOWN:
                enemies.append(SpiritEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                            self.__rect.bottom + EIGHT_PIXELS//2),
                                           direction=(side+2)%4))
            elif side == UP:
                enemies.append(SpiritEnemy((position*EIGHT_PIXELS + EIGHT_PIXELS//2,
                                            self.__rect.top - EIGHT_PIXELS//2),
                                           direction=(side+2)%4))

        difficulty = 0
        for enemy in enemies:
            difficulty += enemy.get_score()

        return {'enemies':enemies, 'difficulty':difficulty}

    def __generate_enemy_waves_1p(self): # enqueues dictionaries of enemies to the enemy queue
        # total difficulty specifies the total sum of enemy scores for the enemy waves group
        if self.__wave_index < 8:
            total_difficulty = 500 + 400*self.__wave_index - (self.__wave_index**3)
        else:
            total_difficulty = 3000
        difficulty = 0

        # randomly select the enemy type to create a wave of
        # create a wave of that enemy, randomly generating the amount of them to spawn
        # enqueue that wave and add the sum of scores to the overall sum
        # range of enemies that can spawn and number of enemies that can spawn at once change
```

```
    while difficulty < total_difficulty:
        match randint(0,(self.__wave_index + 1) if self.__wave_index < 5 else 5):
            case 0: # default enemies can spawn from wave 0
                group = self.__enemy_group(DefaultEnemy,
                                           randint(2, 6) if self.__wave_index < 3
                                             else (randint(4, 10) if self.__wave_index < 6
                                                     else randint(2,6)))
            case 1: # fast enemies can spawn from wave 0
                group = self.__fast_enemy(1 if self.__wave_index < 3
                                           else (randint(1, 2) if self.__wave_index < 6
                                                 else randint(2,3)))
            case 2: # flying enemies can spawn from wave 1
                group = self.__flying_enemy(1 if self.__wave_index < 2
                                             else (randint(1, 2) if self.__wave_index < 7
                                                     else randint(2,4)))
            case 3: # crow enemies can spawn from wave 2
                group = self.__crow_enemy() # always 1 crow enemy
            case 4: # tough enemies can spawn from wave 3
                group = self.__enemy_group(ToughEnemy,
                                           randint(4,6) if self.__wave_index < 4
                                             else (randint(4, 10) if self.__wave_index < 8
                                                     else randint(6,12)))
            case 5: # spirit enemies can spawn from wave 4
                group = self.__spirit_enemy(1 if self.__wave_index < 5
                                             else (randint(1, 3) if self.__wave_index < 8
                                                     else randint(2,4)))
        self.__enemy_queue.enqueue(group)
        difficulty += group['difficulty']

def __generate_enemy_waves_2p(self): # slightly more enemies for two player
    # total_difficulty is greater for two player
    if self.__wave_index < 8:
        total_difficulty = 800 + 600*self.__wave_index - (self.__wave_index**3)
    else:
        total_difficulty = 5000
    difficulty = 0

    # the number of enemies that can spawn in each wave is increased for two player
    while difficulty < total_difficulty:
        match randint(0, (self.__wave_index + 1) if self.__wave_index < 5 else 5):
            case 0: # spawns from wave 0
                group = self.__enemy_group(DefaultEnemy,
                                           randint(2, 6) if self.__wave_index < 3
                                             else (randint(4, 12) if self.__wave_index < 6
                                                     else randint(6,14)))
            case 1: # spawns from wave 0
                group = self.__fast_enemy(1 if self.__wave_index < 2
                                           else (randint(2, 3) if self.__wave_index < 6
                                                 else randint(2,4)))
            case 2: # spawns from wave 1
                group = self.__flying_enemy(1 if self.__wave_index < 2
```

```
                                            else (randint(2, 3) if self.__wave_index < 7
                                                  else randint(3,5)))
                case 3: # spawns from wave 2
                    group = self.__crow_enemy()
                case 4: # spawns from wave 3
                    group = self.__enemy_group(ToughEnemy,
                                               randint(4,8) if self.__wave_index < 4
                                                 else (randint(6, 12) if self.__wave_index < 8
                                                       else randint(8,14)))
                case 5: # spawns from wave 4
                    group = self.__spirit_enemy(1 if self.__wave_index < 5
                                                 else (randint(2, 3) if self.__wave_index < 8
                                                       else randint(3,5)))
            self.__enemy_queue.enqueue(group)
            difficulty += group['difficulty']

    def __first_waves(self): # allows the very first wave to be controlled
        self.__enemy_queue.enqueue(self.__enemy_group(DefaultEnemy, randint(3,6)))
        self.__enemy_queue.enqueue(self.__enemy_group(DefaultEnemy, randint(4,8)))

    def __use_item(self, type, player=0): # use an item
        match player: # which player used the item
            case 0:
                player_class = self.__player
            case 1:
                player_class = self.__player1
            case 2:
                player_class = self.__player2

        # check the item type and run the appropriate method
        if type == BOMB:
            self.__bomb(player_class)
        elif type == SHOES:
            player_class.add_shoes()
        elif type == SHOTGUN:
            player_class.add_shotgun()
        elif type == RAPID_FIRE:
            player_class.add_rapid_fire()
        elif type == TIME_FREEZE:
            self.__freeze_time()
        elif type == BACKWARDS_SHOT:
            player_class.add_backwards_shot()
        elif type == HEART:
            player_class.increase_health(1)

        if item_sounds[type]: # if there is a sound for the item
            item_sounds[type].play()

        self.__items_used += 1

    def __bomb(self, player_class): # use the bomb item
```

```
        to_kill = [] # the enemies to be killed
        # calculate the distance of each enemy and
        # add them to the list if they are within the bomb's range
        for enemy in self.__enemies:
            distance = ((enemy.get_rect().centerx - player_class.get_rect().centerx)**2 +
                        (enemy.get_rect().centery - player_class.get_rect().centery)**2)**0.5
            if distance < BOMB_RANGE:
                to_kill.append(enemy)
        # remove the enemies in the list from the main enemy list
        for enemy in to_kill:
            self.__enemies.remove(enemy)
            self.__enemies_killed += 1
        self.__bomb_time = self.__timer
        self.__shake = True

    def __freeze_time(self): # use the freeze time item
        self.__time_freeze = True
        self.__time_freeze_time = self.__timer

    def get_player_item(self): # return the player's item's type
        if self.__player.get_item():
            return self.__player.get_item().get_type()
        return None

    def get_player_lives(self): # return the player's lives
        if self.__players == 1:
            return self.__player.get_lives()
        elif self.__players == 2:
            return self.__player1.get_lives(), self.__player2.get_lives()

    def get_score(self): # return the total score
        return trunc(self.__time_score + self.__enemy_score)

    def get_time_score(self): # return the score from time
        return self.__time_score

    def get_enemy_score(self): # return the score from killing enemies
        return self.__enemy_score

    def get_enemies_killed(self): # return the number of enemies killed
        return self.__enemies_killed

    def get_bullets_shot(self): # return the number of bullets shot
        if self.__players == 1:
            return self.__player.get_bullets_shot()
        elif self.__players == 2:
            return self.__player1.get_bullets_shot() + self.__player2.get_bullets_shot()

    def get_items_used(self): # return the number of items used
        return self.__items_used
```

```
def __check_enemy_spawn(self): # spawn enemies that won't collide with another enemy
    for enemy in self.__enemies_to_spawn:
        if enemy.get_rect().collidelist(self.__enemy_rects) == -1 or enemy.get_flying():
            self.__enemies.append(enemy)
            self.__enemy_rects.append(enemy.get_rect()) # update enemy_rects list
            self.__enemies_to_spawn.remove(enemy)

def __check_player_hit(self): # check if a player has been hit
    # checking for hit is done with distance not rects to be more forgiving to the player
    # a hit is counted if an enemy's centre is within 7 pixels of the player's centre
    # 7 pixels is 1 less than the width of a player to add some leeway
    if self.__players == 1:
        for enemy in self.__enemies:
            distance = ((enemy.get_rect().centerx-self.__player.get_rect().centerx)**2 +
                        (enemy.get_rect().centery-self.__player.get_rect().centery)**2)**0.5
            if distance < 7*PIXEL_RATIO:
                if not self.__player.get_immunity(): # first check if the player is immune
                    self.__player.hit(1) # take one life and return player to the centre
                    if self.__player.get_lives() == 0:
                        player_death_sound.play()
                        pygame.time.delay(2000) # pause of 2 seconds if player is dead
                    else:
                        player_hit_sound.play()
                        pygame.time.delay(1200) # pause of 1.2 seconds if player isn't dead
                    self.__enemies = [] # reset the enemy list
                    self.__player.empty_bullets() # reset the player's bullets
                    self.__items = [] # reset the items
                break # stop looking at enemies to prevent errors
    elif self.__players == 2:
        for enemy in self.__enemies:
            # calculate distances from both player 1 and two
            dist1 = ((enemy.get_rect().centerx - self.__player1.get_rect().centerx)**2 +
                    (enemy.get_rect().centery - self.__player1.get_rect().centery)**2)**0.5
            dist2 = ((enemy.get_rect().centerx - self.__player2.get_rect().centerx)**2 +
                    (enemy.get_rect().centery - self.__player2.get_rect().centery)**2)**0.5
            if dist1 < 7*PIXEL_RATIO:
                if not self.__player1.get_immunity() and self.__player1.get_lives() > 0:
                    self.__player1.hit(1)
                    self.__player1.add_immunity(7*FPS) # 7s accounts for 3s of respawn
                    self.__player1_respawn = self.__timer + 3*FPS # not spawned for 3s
                    self.__player1.set_spawned(False)
                    self.__enemies.remove(enemy)
                    break
            if dist2 < 7*PIXEL_RATIO:
                if not self.__player2.get_immunity() and self.__player2.get_lives() > 0:
                    self.__player2.hit(1)
                    self.__player2.add_immunity(7*FPS)
                    self.__player2_respawn = self.__timer + 3*FPS
                    self.__player2.set_spawned(False)
                    self.__enemies.remove(enemy)
                    break
```

```
def __update_players(self): # update the player(s)
    if self.__players == 1:
        self.__player.update(self.__collidable_rects) # update player 1
    elif self.__players == 2:
        # pass in the rect of the other player for collisions to each player's update
        rect1, rect2 = None, None
        if self.__player1.get_lives() > 0 and self.__player1.get_spawned():
            rect1 = self.__player1.get_rect()
        if self.__player2.get_lives() > 0 and self.__player2.get_spawned():
            rect2 = self.__player2.get_rect()
        self.__player1.update(self.__collidable_rects, other_player_rect = rect2)
        self.__player2.update(self.__collidable_rects, other_player_rect = rect1)

def __update_enemies(self): # update the enemies
    for enemy in self.__enemies:
        # check if the enemy has been hit by a bullet
        if self.__players == 1:
            for bullet in self.__player.get_bullets():
                if enemy.get_rect().colliderect(bullet.get_rect()):
                    enemy.hit(bullet.get_damage())
                    self.__player.remove_bullet(bullet)
        elif self.__players == 2:
            for bullet in self.__player1.get_bullets():
                if enemy.get_rect().colliderect(bullet.get_rect()):
                    enemy.hit(bullet.get_damage()) # damage enemy with the bullet's damage
                    self.__player1.remove_bullet(bullet)
            for bullet in self.__player2.get_bullets():
                if enemy.get_rect().colliderect(bullet.get_rect()):
                    enemy.hit(bullet.get_damage())
                    self.__player2.remove_bullet(bullet)

        # run the enemy's update function if time isn't frozen
        if not self.__time_freeze:
            if self.__players == 1:
                if not enemy.get_flying():
                    enemy.update(self.__player.get_pos(), self.__rect,
                                 self.__collidable_rects, self.__enemy_rects.copy())
                else:
                    enemy.update(self.__player.get_pos())
            elif self.__players == 2:
                # enemies only travel towards alive and non-immune players
                player1_pos, player2_pos = None, None
                if self.__player1.get_lives() > 0 and not self.__player1.get_immunity():
                    player1_pos = self.__player1.get_pos()
                if self.__player2.get_lives() > 0 and not self.__player2.get_immunity():
                    player2_pos = self.__player2.get_pos()

                if not enemy.get_flying():
                    enemy.update(player1_pos, self.__rect, self.__collidable_rects,
                                 self.__enemy_rects.copy(), player2_pos=player2_pos)
```

```
                else:
                    enemy.update(player1_pos, player2_pos=player2_pos)

            if enemy.get_health() == CROW_OFFSCREEN: # set to constant if crow flown off screen
                self.__enemies.remove(enemy) # no score added
            elif enemy.get_health() <= 0:
                chance = ITEM_CHANCE_1P if self.__players == 1 else ITEM_CHANCE_2P
                if self.__item_countdown == 0 and randint(1, chance) == 1:
                    if ((self.__players == 1 and self.__player.get_lives() < 4)
                            or (self.__players == 2 and (self.__player1.get_lives() < 4
                                                         or self.__player2.get_lives() < 4))):
                        spawn_lives = True # only spawns lives if a player has less than 4
                    else:
                        spawn_lives = False
                    self.__items.append(Item(enemy.get_rect().center,
                                             randint(0, len(item_images)-1 -
                                                     (0 if spawn_lives else 1))))
                    self.__item_countdown = ITEM_COUNTDOWN # can't spawn 2 items within 0.5s
                self.__enemies.remove(enemy)
                self.__scores.append(Score(self.__small_font, WHITE,
                                           enemy.get_score(), enemy.get_rect(),
                                           alpha=SCORE_ALPHA))
                self.__enemy_score += enemy.get_score() # add the enemy's score to the total
                self.__enemies_killed += 1

    def __update_scores(self): # update the score displays
        for score in self.__scores:
            if score.update(): # update returns True if the score should be removed
                self.__scores.remove(score)

    def __update_items(self): # update the items
        for item in self.__items:
            if self.__players == 1:
                # if the player collides with the item and already has an item, use the item
                # if the player collides and does not, set the player's item to the item
                if self.__player.get_rect().colliderect(item.get_rect()):
                    if not self.__player.get_item() and self.__players == 1:
                        self.__player.set_item(item)
                    else:
                        self.__use_item(item.get_type())
                    self.__items.remove(item)
                    break # break to stop the item needlessly updating
            elif self.__players == 2:
                # check both players for a collision
                # if a player collides with an item, use the item
                if self.__player1.get_rect().colliderect(item.get_rect()):
                    self.__use_item(item.get_type(), player=1)
                    self.__items.remove(item)
                    break
                elif self.__player2.get_rect().colliderect(item.get_rect()):
                    self.__use_item(item.get_type(), player=2)
```

```
                self.__items.remove(item)
                break
        if not self.__time_freeze: # only update if time isn't frozen
            if item.update(): # update returns True if the item should be removed
                self.__items.remove(item)

    # check and remove time freeze if necessary
    if self.__time_freeze and self.__timer - self.__time_freeze_time >= TIME_FREEZE_LENGTH:
        self.__time_freeze = False

def __steal_a_life(self): # swaps a life from a player with many lives to a player with 0
    if self.__player1.get_lives() <= 0 and self.__player2.get_lives() > 1:
        self.__player2.increase_health(-1) # decreases health by 1
        self.__player1.hit(-1) # increases health by 1, spawns back at start
        self.__player1.add_immunity(4*FPS) # spawns with with immunity
        self.__player1.set_spawned(True)
    elif self.__player2.get_lives() <= 0 and self.__player1.get_lives() > 1:
        self.__player1.increase_health(-1)
        self.__player2.hit(-1)
        self.__player2.add_immunity(4*FPS)
        self.__player2.set_spawned(True)

def __spawn_enemies(self): # handles spawning enemies and crates
    if self.__wave_index == -1: # immediately spawn first wave
        self.__first_waves()
        self.__wave_index += 1

    if self.__enemy_queue.empty(): # to catch an empty queue
        if not self.__enemies and not self.__enemies_to_spawn: # all enemies have spawned
            if not self.__crate_countdown:
                self.__crate_countdown = 1*FPS # first countdown, until a crate
            elif self.__crate_countdown == 1:  # 0 would be caught by first if
                crate_thud.play()
                self.__place_random(crate_image, 1, 2, collision=True, center_spawn=False)
                if self.__players == 2: # place a second crate in 2 player
                    self.__place_random(crate_image,1,2,collision=True,center_spawn=False)

                if self.__players == 1:
                    self.__generate_enemy_waves_1p()
                elif self.__players == 2:
                    self.__generate_enemy_waves_2p()

                self.__wave_index += 1
                self.__enemy_countdown = 1*FPS # time before the enemies are spawned
    elif self.__enemy_countdown == 0: # spawn enemies
        group = self.__enemy_queue.dequeue()
        self.__enemies_to_spawn.extend(group['enemies'])
        self.__enemy_countdown =  DELAY_MULTIPLIER * (group['difficulty'] -
                                                      self.__wave_index * INDEX_MULTIPLIER)
        if self.__enemy_countdown < MIN_FRAMES:
            self.__enemy_countdown = MIN_FRAMES
```

```
def update(self, event_list):
    for event in event_list:
        if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
            if self.__players == 1: # in 1p, space uses an item
                if self.__player.get_item():
                    self.__use_item(self.__player.get_item().get_type())
                    self.__player.set_item(None)
            elif self.__players == 2: # in 2p, space steals a life
                self.__steal_a_life()

    if self.__players == 2: # check if players should respawn, only necessary in 2p
        if not self.__player1.get_spawned() and self.__timer == self.__player1_respawn:
            self.__player1.set_spawned(True)
        if not self.__player2.get_spawned() and self.__timer == self.__player2_respawn:
            self.__player2.set_spawned(True)

    self.__enemy_rects = self.__enemy_rects_list() # update the enemy_rects list
    self.__check_enemy_spawn() # check if any more enemies can spawn

    if not self.__countdown: # update if not during the 3 second countdown
        self.__update_enemies()
        self.__update_scores()
        self.__update_items()

    self.__update_players() # player can move during the countdown

    if not self.__countdown and not self.__time_freeze:
        self.__spawn_enemies()

    if self.__item_countdown > 0:
        self.__item_countdown -= 1

    if not self.__countdown:
        if not self.__time_freeze:
            self.__time_score += 1/FPS  # 1 score point for each second alive
            if self.__enemy_countdown:
                self.__enemy_countdown -= 1
            if self.__crate_countdown:
                self.__crate_countdown -= 1
        self.__timer += 1
    else:
        self.__countdown -= 1

def __display_controls_1p(self, image): # display the single player controls
    self.__small_font.render(image, "MOVE:",
                             (4*EIGHT_PIXELS, 3*EIGHT_PIXELS), alignment=CENTER)
    self.__small_font.render(image, "W"+NEW_LINE+"A S D",
                             (4*EIGHT_PIXELS, 4*EIGHT_PIXELS), alignment=CENTER)
    self.__small_font.render(image, "SHOOT:",
                             (12*EIGHT_PIXELS, 3*EIGHT_PIXELS), alignment=CENTER)
```

```
        self.__small_font.render(image, "^"+NEW_LINE+"< _ >",
                                 (12*EIGHT_PIXELS, 4*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "USE ITEM:",
                                 (8*EIGHT_PIXELS, 10*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "SPACE",
                                 (8*EIGHT_PIXELS, 11*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "PAUSE:",
                                 (8*EIGHT_PIXELS, 12.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "ESC",
                                 (8*EIGHT_PIXELS, 13.5*EIGHT_PIXELS), alignment=CENTER)

    def __display_controls_2p(self, image): # display the two player controls
        self.__small_font.render(image, "PLAYER 1",
                                 (4*EIGHT_PIXELS, 2*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "MOVE:",
                                 (4*EIGHT_PIXELS, 3*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "W"+NEW_LINE+"A S D",
                                 (4*EIGHT_PIXELS, 4*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "SHOOT:",
                                 (4*EIGHT_PIXELS, 6.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "G"+NEW_LINE+"V B N",
                                 (4*EIGHT_PIXELS, 7.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "PLAYER 2",
                                 (12*EIGHT_PIXELS, 2*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "MOVE:",
                                 (12*EIGHT_PIXELS, 3*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "9"+NEW_LINE+"I O P",
                                 (12*EIGHT_PIXELS, 4*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "SHOOT:",
                                 (12*EIGHT_PIXELS, 6.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "^"+NEW_LINE+"< _ >",
                                 (12*EIGHT_PIXELS, 7.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "STEAL A LIFE:",
                                 (8*EIGHT_PIXELS, 10*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "SPACE",
                                 (8*EIGHT_PIXELS, 11*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "PAUSE:",
                                 (8*EIGHT_PIXELS, 12.5*EIGHT_PIXELS), alignment=CENTER)
        self.__small_font.render(image, "ESC",
                                 (8*EIGHT_PIXELS, 13.5*EIGHT_PIXELS), alignment=CENTER)

    def draw(self, screen):
        image = pygame.surface.Surface(self.__rect.size) # empty image to blit everything to

        above = []
        for row in self.__grid:
            for cell in row:
                if cell.get_collision() and cell.has_image():
                    above.append(cell) # draw cells with collisions and images after
                else:
                    cell.draw(image)
```

```
if self.__countdown:
    if self.__players == 1:
        self.__display_controls_1p(image)
    elif self.__players == 2:
        self.__display_controls_2p(image)

image.blit(fences_image, (0,0))

for item in self.__items:
    item.draw(image)

if self.__players == 1:
    bullets = self.__player.get_bullets()
elif self.__players == 2:
    bullets = self.__player1.get_bullets() + self.__player2.get_bullets()

for bullet in bullets:
    bullet.draw(image)

if self.__players == 1:
    self.__player.draw(image)
elif self.__players == 2:
    if self.__player2.get_lives() > 0:
        self.__player2.draw(image)
    if self.__player1.get_lives() > 0:
        self.__player1.draw(image) # draw player 1 on top

for enemy in self.__enemies:
    if not enemy.get_flying():
        enemy.draw(image)

for cell in above:
    cell.draw(image)

for score in self.__scores:
    score.draw(image)

for enemy in self.__enemies:
    if enemy.get_flying():
        enemy.draw(image)

if self.__countdown:
    self.__countdown_font.render(image, f"{ceil(self.__countdown/FPS)}",
                                 (8*EIGHT_PIXELS, 8*EIGHT_PIXELS -
                                                  self.__countdown_font.get_height()//2),
                                 alignment=CENTER)

self.__check_player_hit()

offset = (0,0)
```

```
if self.__shake:
    if self.__timer - self.__bomb_time == SCREEN_SHAKE_LENGTH and self.__shake:
        self.__shake = False
    else:
        if self.__timer - self.__bomb_time < SCREEN_SHAKE_LENGTH // 2:
            multiplier = 2 # first half of shakes is more extreme
        else:
            multiplier = 1
        if self.__timer % 8 > 4: # one full shake over 8 frames
            offset = (PIXEL_RATIO * multiplier, 0)
        else:
            offset = (-PIXEL_RATIO * multiplier, 0)

if self.__time_freeze:
    time = self.__timer - self.__time_freeze_time
    # hazy freeze overlay that changes colour over its time
    self.__freeze_surface.fill((100 + trunc(time / 6), 100 + trunc(time / 6),
                                160 + trunc(time / 6), 70 - trunc(time / 6)))
    image.blit(self.__freeze_surface, (0,0))

screen.blit(image, (self.__pos[X] + offset[X], self.__pos[Y] + offset[Y]))
```

6.12 Main File

main.py

```
#======================Imports======================#
from sys import exit
from math import trunc
import sqlite3
import json
import pygame

from constants import *
from database_functions import *
from utility_functions import split, colour_swap
from game import Game
from customise_classes import ColourGrid, DrawingGrid
from utility_classes import ImageButton, TextButton, Font, CharacterDisplay, Slider, TextBox
from leaderboard_classes import Leaderboard, TwoPlayerLeaderboard, Podium
from sounds import all_sound_volumes, button_click
from images import (default_front_image2, small_font_image, medium_font_image, big_font_image,
                    huge_font_image, settings_image, customise_image, return_image,
                    cursor_image, item_images, up_arrow_image)

#======================Loading and Creating the Database======================#
db_connection = sqlite3.connect("NEA_database.db")
db_cursor = db_connection.cursor()
db_create_database(db_cursor, db_connection) # already handles whether or not the tables exist

#======================Initialising======================#
pygame.init()
pygame.key.set_repeat(500, 100) # pressed keys generate new events every 100 ms after 500 ms
pygame.mixer.set_num_channels(10)
pygame.mouse.set_visible(False)

screen = pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("UNTITLED GUN GAME")
pygame.display.set_icon(default_front_image2)

clock = pygame.time.Clock()

#======================Fonts======================#
small_font = Font(small_font_image, CHARACTER_LIST, WHITE, 2*PIXEL_RATIO)
medium_font = Font(medium_font_image, CHARACTER_LIST_U, WHITE, 5*PIXEL_RATIO)
big_font = Font(big_font_image, CHARACTER_LIST_U, WHITE, 4*PIXEL_RATIO,
                character_spacing=2*PIXEL_RATIO)
huge_font = Font(huge_font_image, CHARACTER_LIST_U, WHITE, 6*PIXEL_RATIO,
                 character_spacing=2*PIXEL_RATIO)

#======================Quit Function======================#
def quit():
    db_cursor.close()
```

```
    db_connection.close()
    pygame.quit()
    exit()

#======================Set Volume Function======================#
def set_volume():
    for sound in all_sound_volumes.keys():
        sound.set_volume(all_sound_volumes[sound] * settings['volume'])

#======================Main Menu Function======================#
def main_menu(username):
    one_player_button = TextButton((2*EIGHT_PIXELS, 7.5*EIGHT_PIXELS),
                                   (10*EIGHT_PIXELS, 4*EIGHT_PIXELS),
                                   "SINGLE"+NEW_LINE+"PLAYER", big_font)
    two_player_button = TextButton((2*EIGHT_PIXELS, 12*EIGHT_PIXELS),
                                   (10*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                   "TWO PLAYER", medium_font)
    settings_button = ImageButton((2*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                  (2*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                  settings_image)
    customise_button = ImageButton((20.5*EIGHT_PIXELS, 3.5*EIGHT_PIXELS),
                                   (2*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                   customise_image)
    leaderboard_button = TextButton((14*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                    (8*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                    "LEADERBOARDS"+NEW_LINE+"+ STATISTICS", small_font)
    log_out_button = TextButton((4.5*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                                (3.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                "LOG"+NEW_LINE+"OUT", small_font)
    quit_button = TextButton((8.5*EIGHT_PIXELS, 14.5*EIGHT_PIXELS),
                             (3.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                             "QUIT", small_font)

    player_character = db_get_character(username, db_cursor)
    character_display = CharacterDisplay((16*EIGHT_PIXELS, 2.5*EIGHT_PIXELS),
                                         4, GRASS_GREEN, player_character,
                                         extra_border_colour=CHARACTER_DISPLAY_BORDER)

    leaderboard = Leaderboard(db_cursor, (13*EIGHT_PIXELS, 7.5*EIGHT_PIXELS), 10*EIGHT_PIXELS,
                              small_font, "score", "SinglePlayerGames", "Players",
                              rows=5, record_height=9*PIXEL_RATIO, highlight_key=username)

    title_font_silver = huge_font.new_colour_copy(SILVER)
    title_font_white = huge_font.new_colour_copy(WHITE)
    title_top = 7*PIXEL_RATIO
    title_left = 10*PIXEL_RATIO

    mpos = pygame.mouse.get_pos()
    display_mouse = True
    while True:
        click = False # if the user has clicked in the current frame
```

```
event_list = pygame.event.get()
for event in event_list:
    if event.type == pygame.QUIT:
        quit() # quit if the user has tried to quit
    elif event.type == pygame.MOUSEBUTTONDOWN:
        if event.button == 1:
            click = True
    elif event.type == pygame.KEYDOWN:
        display_mouse = False # turn the mouse off when the user types

screen.fill(BACKGROUND_COLOUR)

previous_mpos = mpos
mpos = pygame.mouse.get_pos()
if mpos != previous_mpos:
    display_mouse = True

# update buttons
one_player_button.update(mpos, click)
two_player_button.update(mpos, click)
settings_button.update(mpos, click)
customise_button.update(mpos, click)
leaderboard_button.update(mpos, click)
quit_button.update(mpos, click)
log_out_button.update(mpos, click)

# check buttons
if one_player_button.get_clicked():
    player_character = db_get_character(username, db_cursor)
    highscore = db_get_1p_highscore(username, db_cursor)
    play = True
    while play:
        game_data = game(player_character, highscore)
        if game_data:
            play = score_screen(username, *game_data, highscore)
        else: # if no game_data, they quit mid_game
            play = False
    leaderboard.update()
    mpos = pygame.mouse.get_pos() # reset mouse cursor

elif two_player_button.get_clicked():
    username_two = login(title="PLAYER TWO LOGIN:", button_text="START",
                         blocked_names=[username]) # can't login as yourself
    if username_two: # if no username_two, the user pressed return
        player1_character = db_get_character(username, db_cursor)
        player2_character = db_get_character(username_two, db_cursor)
        highscore = db_get_2p_highscore(username, db_cursor)
        play = True
        while play:
            game_data = game(player1_character, highscore,
                             players=2, player2_hex=player2_character)
```

```
            if game_data:
                play = score_screen(username, *game_data, highscore,
                                    username_two=username_two)
            else: # if no game_data, they quit mid_game
                play = False
        mpos = pygame.mouse.get_pos()

    elif settings_button.get_clicked():
        settings_screen()
        mpos = pygame.mouse.get_pos()

    elif customise_button.get_clicked():
        player_character = db_get_character(username, db_cursor)
        customise(player_character, username)
        player_character = db_get_character(username, db_cursor)
        character_display.hex_to_grid(player_character) # update character display
        mpos = pygame.mouse.get_pos()

    elif leaderboard_button.get_clicked():
        leaderboards(username)
        mpos = pygame.mouse.get_pos()

    elif log_out_button.get_clicked():
        return

    elif quit_button.get_clicked():
        quit()

    # draw everything to the screen
    one_player_button.draw(screen)
    two_player_button.draw(screen)
    settings_button.draw(screen)
    customise_button.draw(screen)
    leaderboard_button.draw(screen)
    quit_button.draw(screen)
    log_out_button.draw(screen)

    title_font_silver.render(screen, "UNTITLED"+NEW_LINE+"GUN GAME",
                             (title_left, title_top), alignment=LEFT)
    title_font_white.render(screen, "N", (title_left + 14*PIXEL_RATIO, title_top))
    title_font_white.render(screen, "E", (title_left + 78*PIXEL_RATIO, title_top))
    title_font_white.render(screen, "A", (title_left + 62*PIXEL_RATIO,
                                          title_top + 26*PIXEL_RATIO))

    medium_font.render(screen, username, (18*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                       alignment=CENTER)
    character_display.draw(screen)

    leaderboard.draw(screen)

    if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
```

```
                                    or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
                screen.blit(cursor_image, mpos)

            pygame.display.update()
            clock.tick(FPS)

#======================Game Function======================#
# for both one and two players
def game(player_hex, highscore, players=1, player2_hex=None):
    game = Game((4*EIGHT_PIXELS, 1*EIGHT_PIXELS), player_hex, FENCE_LIST,
                players=players, player2_hex=player2_hex)

    if players == 1: # no item storage in 2 player
        item_store_rect = pygame.Rect((6*PIXEL_RATIO, 14*PIXEL_RATIO),
                                      (2.5*EIGHT_PIXELS, 2.5*EIGHT_PIXELS))

    highscore_font = small_font.new_colour_copy(MINOR_TEXT)

    # for when the game is paused
    pause_surface = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT), pygame.SRCALPHA)
    pause_surface.fill(PAUSE_COLOUR)
    pause_settings_button = TextButton((SCREEN_WIDTH//2 - 7.5*EIGHT_PIXELS//2, 8*EIGHT_PIXELS),
                                       (7.5*EIGHT_PIXELS, 1.5*EIGHT_PIXELS),
                                       "SETTINGS", medium_font, background_colour=None,
                                       hover_background_colour=PAUSE_BUTTON_HOVER)
    pause_exit_button = TextButton((SCREEN_WIDTH//2 - 10.5*EIGHT_PIXELS//2, 9.5*EIGHT_PIXELS),
                                   (10.5*EIGHT_PIXELS, 1.5*EIGHT_PIXELS),
                                   "EXIT TO MENU", medium_font, background_colour=None,
                                   hover_background_colour=PAUSE_BUTTON_HOVER)

    mpos = pygame.mouse.get_pos()
    display_mouse = True
    pause = False
    while True:
        click = False
        event_list = pygame.event.get()
        for event in event_list:
            if event.type == pygame.QUIT:
                quit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    click = True
            elif event.type == pygame.KEYDOWN:
                display_mouse = False
                if event.key == pygame.K_ESCAPE:
                    button_click.play()
                    pause = not pause

        screen.fill(BACKGROUND_COLOUR)

        previous_mx_my = (mpos)
```

```
mpos = pygame.mouse.get_pos()
if not display_mouse and (mpos) != previous_mx_my:
    display_mouse = True

if not pause:
    game.update(event_list)

# draw everything to the screen
game.draw(screen)

medium_font.render(screen, f"SCORE: {game.get_score()}", (4*EIGHT_PIXELS, 0))
highscore_font.render(screen, f"HIGHSCORE: {highscore}",
                      (20*EIGHT_PIXELS, 17*EIGHT_PIXELS + 1*PIXEL_RATIO),
                      alignment=RIGHT)

if players == 1:
    # draw item
    pygame.draw.rect(screen, PALER_BACKGROUND, item_store_rect)
    if game.get_player_item() != None: # can be 0
        screen.blit(pygame.transform.scale_by(item_images[game.get_player_item()], 2),
                    (item_store_rect.centerx - EIGHT_PIXELS,
                     item_store_rect.centery - EIGHT_PIXELS))

    # draw lives
    for i in range(0, game.get_player_lives()):
        screen.blit(item_images[HEART], (1*EIGHT_PIXELS + i%2*EIGHT_PIXELS,
                                         4.5*EIGHT_PIXELS + i//2*EIGHT_PIXELS))

    # end the game if the player is dead
    if game.get_player_lives() == 0:
        return (game.get_time_score(), game.get_enemy_score(),
                game.get_enemies_killed(), game.get_bullets_shot(),
                game.get_items_used())

elif players == 2:
    # draw lives
    player1_lives, player2_lives = game.get_player_lives()
    for i in range(0, player1_lives):
        screen.blit(item_images[HEART], (2.5*EIGHT_PIXELS,
                                         1.5*EIGHT_PIXELS + i*EIGHT_PIXELS))
    for i in range(0, player2_lives):
        screen.blit(item_images[HEART], (20.5*EIGHT_PIXELS,
                                         1.5*EIGHT_PIXELS + i*EIGHT_PIXELS))

    # end the game if both players are dead
    if player1_lives <= 0 and player2_lives <= 0:
        return (game.get_time_score(), game.get_enemy_score(),
                game.get_enemies_killed(), game.get_bullets_shot(),
                game.get_items_used())

if pause:
```

```
            # update pause-relevant buttons
            pause_exit_button.update(mpos, click)
            pause_settings_button.update(mpos, click)

            # check for pause-relevant button presses
            if pause_settings_button.get_clicked():
                settings_screen(size=False) # can't change screen size in game menu
                mpos = pygame.mouse.get_pos()
            if pause_exit_button.get_clicked():
                return None

            # draw pause-relevant things
            screen.blit(pause_surface, (0,0))
            big_font.render(screen, "||"+NEW_LINE+"PAUSED", (SCREEN_WIDTH//2, 4*EIGHT_PIXELS),
                            alignment=CENTER)
            pause_exit_button.draw(screen)
            pause_settings_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Score Screen Function======================#
# and updates the database
def score_screen(username, time_score, enemy_score, enemies_killed, bullets_shot, items_used,
                 highscore, username_two=None):
    score = trunc(time_score) + enemy_score

    # update the database
    if not username_two:
        db_add_to_stats(username, db_cursor, db_connection, games_played=1,
                        enemies_killed=enemies_killed, bullets_shot=bullets_shot,
                        items_used=items_used)
        db_insert_1p_score(username, score, db_cursor, db_connection)
    else:
        db_insert_2p_score(username, username_two, score, db_cursor, db_connection)

    if score > highscore:
        new_highscore = True
        highscore = score
    else:
        new_highscore = False

    # time_score counts the seconds
    time_minutes = str(trunc(time_score//60))
    time_seconds = str(trunc(time_score%60)).rjust(2, "0")

    highscore_font = small_font.new_colour_copy(MINOR_TEXT)
```

```
right_bound = 12.5*EIGHT_PIXELS # used for positioning

if not username_two:
    leaderboard = Leaderboard(db_cursor, (14*EIGHT_PIXELS, 5.5*EIGHT_PIXELS),
                              9*EIGHT_PIXELS, small_font,
                              "score", "SinglePlayerGames", "Players",
                              rows=10, record_height=7*PIXEL_RATIO, highlight_key=username)
    play_again_pos = (1.5*EIGHT_PIXELS, 11.5*EIGHT_PIXELS)
    play_again_size = (11*EIGHT_PIXELS, 3*EIGHT_PIXELS)
    return_pos = (2.5*EIGHT_PIXELS, 15*EIGHT_PIXELS)
    return_size = (9*EIGHT_PIXELS, 1.5*EIGHT_PIXELS)
else:
    leaderboard = TwoPlayerLeaderboard(db_cursor, (3*EIGHT_PIXELS, 12*EIGHT_PIXELS),
                                       18*EIGHT_PIXELS, small_font,
                                       "score", "TwoPlayerGames", "Players",
                                       rows=5, record_height=7*PIXEL_RATIO,
                                       highlight_key=username)
    play_again_pos = (13*EIGHT_PIXELS, 6*EIGHT_PIXELS)
    play_again_size = (10*EIGHT_PIXELS, 3*EIGHT_PIXELS)
    return_pos = (13.5*EIGHT_PIXELS, 9.5*EIGHT_PIXELS)
    return_size = (9*EIGHT_PIXELS, 1.5*EIGHT_PIXELS)

play_again_button = TextButton(play_again_pos, play_again_size, "PLAY AGAIN", big_font)
return_button = TextButton(return_pos, return_size, "RETURN TO MENU", small_font)

mpos = pygame.mouse.get_pos()
display_mouse = True
while True:
    click = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            quit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                click = True
        elif event.type == pygame.KEYDOWN:
            display_mouse = False

    screen.fill(BACKGROUND_COLOUR)

    previous_mx_my = (mpos)
    mpos = pygame.mouse.get_pos()
    if (mpos) != previous_mx_my:
        display_mouse = True

    # update buttons
    play_again_button.update(mpos, click)
    return_button.update(mpos, click)

    # check buttons
```

```
        if play_again_button.get_clicked():
            return True
        if return_button.get_clicked():
            return False

        # draw everything to the screen
        big_font.render(screen, f"SCORE:", (EIGHT_PIXELS, EIGHT_PIXELS + 6*PIXEL_RATIO))
        huge_font.render(screen, f"{score}", (7.25*EIGHT_PIXELS, EIGHT_PIXELS))
        highscore_font.render(screen, f"HIGHSCORE: {highscore}",
                              (1*EIGHT_PIXELS, 4.25*EIGHT_PIXELS))
        if new_highscore:
            small_font.render(screen, "NEW HIGHSCORE!", (7.25*EIGHT_PIXELS,3*PIXEL_RATIO))

        small_font.render(screen, f"TIME ALIVE:", (1*EIGHT_PIXELS, 6.5*EIGHT_PIXELS))
        small_font.render(screen, f"{time_minutes}:{time_seconds}",
                          (right_bound, 6.5*EIGHT_PIXELS), alignment=RIGHT)
        small_font.render(screen, f"ENEMIES KILLED:", ((1*EIGHT_PIXELS, 7.5*EIGHT_PIXELS)))
        small_font.render(screen, f"{enemies_killed}",
                          (right_bound, 7.5*EIGHT_PIXELS), alignment=RIGHT)
        small_font.render(screen, f"BULLETS SHOT:", ((1*EIGHT_PIXELS, 8.5*EIGHT_PIXELS)))
        small_font.render(screen, f"{bullets_shot}",
                          (right_bound, 8.5*EIGHT_PIXELS), alignment=RIGHT)
        small_font.render(screen, f"ITEMS USED:", ((1*EIGHT_PIXELS, 9.5*EIGHT_PIXELS)))
        small_font.render(screen, f"{items_used}",
                          (right_bound, 9.5*EIGHT_PIXELS), alignment=RIGHT)

        leaderboard.draw(screen)
        play_again_button.draw(screen)
        return_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Customisation Function======================#
def customise(character_hex, username):
    # character display that shows the character that the user has saved
    saved_character = CharacterDisplay((4*EIGHT_PIXELS, 1.5*EIGHT_PIXELS),
                                       4, GRASS_GREEN, character_hex,
                                       extra_border_colour=CHARACTER_DISPLAY_BORDER)
    # character display that shows the character that the user is currently customising
    custom_character = CharacterDisplay((2*EIGHT_PIXELS, 7*EIGHT_PIXELS),
                                        8, GRASS_GREEN, character_hex,
                                        extra_border_colour=CHARACTER_DISPLAY_BORDER)

    width = 8
    hat_height = 3
```

```
# splits the hex between the hat and the select grids, select hex split into colours
hat_hex = character_hex[-COLOUR_DEPTH*hat_height*width:]
select_hex = split(character_hex[:-COLOUR_DEPTH*hat_height*width], COLOUR_DEPTH)

select_x = 12*EIGHT_PIXELS
select_y = 8*EIGHT_PIXELS

# creates each colour grid for each body part of the custom character
skin_select = ColourGrid((select_x, select_y),
                         SKIN_COLOURS, 4, hex_colour=select_hex[0])
jacket_select = ColourGrid((select_x + 6*EIGHT_PIXELS, select_y),
                           JACKET_COLOURS, 4, hex_colour=select_hex[1])
shirt_select = ColourGrid((select_x, select_y + 3*EIGHT_PIXELS),
                          SHIRT_COLOURS, 4, hex_colour=select_hex[2])
trouser_select = ColourGrid((select_x + 6*EIGHT_PIXELS, select_y + 3*EIGHT_PIXELS),
                            TROUSER_COLOURS, 4, hex_colour=select_hex[3])
eye_select = ColourGrid((select_x + EIGHT_PIXELS, select_y + 6*EIGHT_PIXELS),
                        EYE_COLOURS, 4, hex_colour=select_hex[4],
                        locked_list=[((0,1), db_cursor, username,
                                      'enemies_killed', ENEMY_ACHIEVEMENT)])
gun_select = ColourGrid((select_x + 7*EIGHT_PIXELS, select_y + 6*EIGHT_PIXELS),
                        GUN_COLOURS, 4, hex_colour=select_hex[5],
                        locked_list=[((0,1), db_cursor, username,
                                      'bullets_shot', BULLET_ACHIEVEMENT)])
select_grids = [skin_select, jacket_select, shirt_select,
                trouser_select, eye_select, gun_select]
select_texts = ["SKIN", "JACKET", "SHIRT", "TROUSERS", "EYES", "GUN"] # text for the grids


# creates the hat drawing grid with the initial hat hex
draw_hat_grid = DrawingGrid((13*EIGHT_PIXELS, 3.5*EIGHT_PIXELS),
                            width, hat_height, HAT_COLOURS, initial_hex=hat_hex)

up_arrow = colour_swap(up_arrow_image, WHITE, PALER_BACKGROUND)

save_button = TextButton((3*EIGHT_PIXELS, 16*EIGHT_PIXELS),
                         (5.5*EIGHT_PIXELS, 1.5*EIGHT_PIXELS),
                         "SAVE", medium_font, disabled=True)
return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
                            (2*EIGHT_PIXELS, 2*EIGHT_PIXELS), return_image)

custom_hex = character_hex

clicking = False # if the left mouse button is down
mpos = pygame.mouse.get_pos()
display_mouse = True
while True:
    click = False # if the left mouse button has just been pressed in the current frame
    unclick = False # if the left mouse button has just been unpressed in the current frame
    event_list = pygame.event.get()
    for event in event_list:
```

```
    if event.type == pygame.QUIT:
        quit()
    elif event.type == pygame.MOUSEBUTTONDOWN:
        if event.button == 1:
            clicking = True
            click = True
    elif event.type == pygame.MOUSEBUTTONUP:
        if event.button == 1:
            clicking = False
            unclick = True
    elif event.type == pygame.KEYDOWN:
        display_mouse = False

screen.fill(BACKGROUND_COLOUR)

previous_mx_my = (mpos)
mpos = pygame.mouse.get_pos()
if (mpos) != previous_mx_my:
    display_mouse = True

# update the grids and buttons
draw_hat_grid.update(mpos, clicking, click, unclick)
for grid in select_grids:
    grid.update(mpos, click)
save_button.update(mpos, click)
return_button.update(mpos, click)

changed = False

# if the hat drawing grid has changed, update the hat of the custom character display
if draw_hat_grid.get_changed():
    changed = True
    custom_character.hat_hex_to_grid(draw_hat_grid.grid_to_hex())

# if a colour grid is changed, update the relevant part of the custom character
for grid in select_grids:
    if grid.get_changed():
        changed = True
        custom_character.update_pattern(select_grids.index(grid),
                                        grid.get_selected().get_colour())

if changed:
    # create a new empty custom_hex variable
    custom_hex = ""
    # first part of the hex string is the 6 colours representing each part
    for grid in select_grids:
        custom_hex += grid.selected_to_hex()
    # second part is the bitmap of the hat grid
    custom_hex += draw_hat_grid.grid_to_hex()

if character_hex == custom_hex: # if the character is unchanged
```

```
            save_button.set_disabled(True)
        else:
            save_button.set_disabled(False)

        if save_button.get_clicked():
            character_hex = custom_hex
            saved_character.hex_to_grid(character_hex) # update the saved character
            db_set_character(username, character_hex, db_cursor, db_connection)
        elif return_button.get_clicked():
            return # no need to return anything as it already updates the database

        # draw everything to the screen
        saved_character.draw(screen)
        custom_character.draw(screen)
        screen.blit(up_arrow, (5*EIGHT_PIXELS, 5*EIGHT_PIXELS + 6*PIXEL_RATIO))
        draw_hat_grid.draw(screen)
        medium_font.render(screen, "CUSTOMISATION STUDIO", (12*EIGHT_PIXELS, 1*PIXEL_RATIO),
                           alignment=CENTER)
        for i in range(len(select_grids)):
            grid = select_grids[i]
            grid.draw_border_rect(screen) # draws a border for the grid
            grid.draw(screen)
            # write the relevant text above each grid
            small_font.render(screen, select_texts[i],
                              (grid.get_rect().centerx,  grid.get_rect().y + 11*PIXEL_RATIO),
                               alignment=CENTER)
        save_button.draw(screen)
        return_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Leaderboards and Statistics Function======================#
def leaderboards(username):
    one_player_tab_button = TextButton((3*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                                       (4.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                       "SINGLE"+NEW_LINE+"PLAYER", small_font,
                                       hold=False, pressed=True)
    two_player_tab_button = TextButton((7.5*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                                       (4.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                       "TWO"+NEW_LINE+"PLAYER", small_font, hold=False)
    stat_podiums_tab_button = TextButton((12*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                                         (4.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                         "PODIUMS", small_font, hold=False)
    my_stats_tab_button = TextButton((16.5*EIGHT_PIXELS, 1*EIGHT_PIXELS),
                                     (4.5*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                                     "MY"+NEW_LINE+"STATS", small_font, hold=False)
```

```
return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
                            (2*EIGHT_PIXELS, 2*EIGHT_PIXELS), return_image)

# single player tab
one_player_leaderboard = Leaderboard(db_cursor, (3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                                     18*EIGHT_PIXELS, small_font,
                                     "score", "SinglePlayerGames", "Players",
                                     rows=10, highlight_key=username,
                                     display_characters=True)

# two player tab
two_player_leaderboard = TwoPlayerLeaderboard(db_cursor, (3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                                              18*EIGHT_PIXELS, small_font,
                                              "score", "TwoPlayerGames", "Players",
                                              rows=10, highlight_key=username)

# podiums tab
podiums_rect = pygame.Rect((3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                           (18*EIGHT_PIXELS, 14*EIGHT_PIXELS))
games_played_rect = pygame.Rect((3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                                (18*EIGHT_PIXELS, EIGHT_PIXELS + PIXEL_RATIO))
enemies_killed_rect = pygame.Rect((3*EIGHT_PIXELS, 10*EIGHT_PIXELS),
                                  (18*EIGHT_PIXELS, EIGHT_PIXELS + PIXEL_RATIO))
games_played_podium = Podium(db_cursor, 'games_played',
                             (12*EIGHT_PIXELS, 8.5*EIGHT_PIXELS + PIXEL_RATIO),
                             2*EIGHT_PIXELS, 3, 2, medium_font, small_font)
enemies_killed_podium = Podium(db_cursor, 'enemies_killed',
                               (12*EIGHT_PIXELS, 15.5*EIGHT_PIXELS + PIXEL_RATIO),
                               2*EIGHT_PIXELS, 3, 2, medium_font, small_font)

# personal statistics tab
player_stats = db_get_stats(username, db_cursor)
my_stats_rect = pygame.Rect((3*EIGHT_PIXELS, 3*EIGHT_PIXELS),
                            (18*EIGHT_PIXELS, 4*EIGHT_PIXELS + PIXEL_RATIO))
my_stats_x = my_stats_rect.left + 2*PIXEL_RATIO
my_stats_y = my_stats_rect.top + 2*PIXEL_RATIO
my_stats_spacing = EIGHT_PIXELS

current_tab = 1
mpos = pygame.mouse.get_pos()
display_mouse = True
while True:
    click = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            quit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                click = True
        elif event.type == pygame.KEYDOWN:
```

```
        display_mouse = False

screen.fill(BACKGROUND_COLOUR)

previous_mpos = mpos
mpos = pygame.mouse.get_pos()
if mpos != previous_mpos:
    display_mouse = True

# update buttons
one_player_tab_button.update(mpos, click)
two_player_tab_button.update(mpos, click)
stat_podiums_tab_button.update(mpos, click)
my_stats_tab_button.update(mpos, click)
return_button.update(mpos, click)

# check buttons, unpress every other button and swap tabs if a tab button is pressed
if one_player_tab_button.get_clicked():
    two_player_tab_button.unpress()
    stat_podiums_tab_button.unpress()
    my_stats_tab_button.unpress()
    current_tab = 1
elif two_player_tab_button.get_clicked():
    one_player_tab_button.unpress()
    stat_podiums_tab_button.unpress()
    my_stats_tab_button.unpress()
    current_tab = 2
elif stat_podiums_tab_button.get_clicked():
    one_player_tab_button.unpress()
    two_player_tab_button.unpress()
    my_stats_tab_button.unpress()
    current_tab = 3
elif my_stats_tab_button.get_clicked():
    one_player_tab_button.unpress()
    two_player_tab_button.unpress()
    stat_podiums_tab_button.unpress()
    current_tab = 4
elif return_button.get_clicked():
    return

# draw everything to the screen
one_player_tab_button.draw(screen)
two_player_tab_button.draw(screen)
stat_podiums_tab_button.draw(screen)
my_stats_tab_button.draw(screen)
return_button.draw(screen)

medium_font.render(screen, "LEADERBOARDS", (12*EIGHT_PIXELS, 0), alignment=CENTER)

match current_tab:
    case 1: # single player leaderboard tab
```

```
                one_player_leaderboard.draw(screen)
            case 2: # two player leaderboard tab
                two_player_leaderboard.draw(screen)
            case 3: # podiums tab
                pygame.draw.rect(screen, TEXT_BUTTON_BACKGROUND_COLOUR, podiums_rect)
                pygame.draw.rect(screen, TEXT_BUTTON_HOVER_COLOUR, games_played_rect)
                pygame.draw.rect(screen, TEXT_BUTTON_HOVER_COLOUR, enemies_killed_rect)
                games_played_podium.draw(screen)
                enemies_killed_podium.draw(screen)
                small_font.render(screen, "GAMES PLAYED",
                                  (12*EIGHT_PIXELS, 3*EIGHT_PIXELS + 2*PIXEL_RATIO),
                                  alignment=CENTER)
                small_font.render(screen, "ENEMIES KILLED",
                                  (12*EIGHT_PIXELS, 10*EIGHT_PIXELS + 2*PIXEL_RATIO),
                                  alignment=CENTER)
            case 4: # personal stats tab
                pygame.draw.rect(screen, TEXT_BUTTON_HOVER_COLOUR, my_stats_rect)
                small_font.render(screen, "GAMES PLAYED:",
                                  (my_stats_x, my_stats_y))
                small_font.render(screen, "ENEMIES KILLED:",
                                  (my_stats_x, my_stats_y + my_stats_spacing))
                small_font.render(screen, "BULLETS SHOT:",
                                  (my_stats_x, my_stats_y + 2*my_stats_spacing))
                small_font.render(screen, "ITEMS USED:",
                                  (my_stats_x, my_stats_y + 3*my_stats_spacing))
                small_font.render(screen, str(player_stats[0]),
                                  (my_stats_rect.right - 2*PIXEL_RATIO, my_stats_y),
                                  alignment=RIGHT)
                small_font.render(screen, str(player_stats[1]),
                                  (my_stats_rect.right - 2*PIXEL_RATIO,
                                   my_stats_y + 1*my_stats_spacing),
                                  alignment=RIGHT)
                small_font.render(screen, str(player_stats[2]),
                                  (my_stats_rect.right - 2*PIXEL_RATIO,
                                   my_stats_y + 2*my_stats_spacing),
                                  alignment=RIGHT)
                small_font.render(screen, str(player_stats[3]),
                                  (my_stats_rect.right - 2*PIXEL_RATIO,
                                   my_stats_y + 3*my_stats_spacing),
                                  alignment=RIGHT)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Settings Function======================#
def settings_screen(size=True):
    return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
```

```
                            (2*EIGHT_PIXELS, 2*EIGHT_PIXELS), return_image)
save_button = TextButton((SCREEN_WIDTH//2 - 5*EIGHT_PIXELS//2, 12*EIGHT_PIXELS),
                         (5*EIGHT_PIXELS, 1.5*EIGHT_PIXELS), "SAVE", medium_font,
                         disabled=True)
volume_slider = Slider((SCREEN_WIDTH//2 - 9*EIGHT_PIXELS//2, 6.5*EIGHT_PIXELS),
                       (9*EIGHT_PIXELS, 4*PIXEL_RATIO), 0, 100, settings['volume'] * 100)
if size:
    size_slider = Slider((SCREEN_WIDTH//2 - 9*EIGHT_PIXELS//2, 10*EIGHT_PIXELS),
                         (9*EIGHT_PIXELS, 4*PIXEL_RATIO), 1, 6, settings['size'])

mpos = pygame.mouse.get_pos()
display_mouse = True
saved = False
while True:
    click = False
    unclick = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            quit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                click = True
        elif event.type == pygame.MOUSEBUTTONUP:
            if event.button == 1:
                unclick = True
        elif event.type == pygame.KEYDOWN:
            display_mouse = False

    screen.fill(BACKGROUND_COLOUR)

    previous_mx_my = (mpos)
    mpos = pygame.mouse.get_pos()
    if (mpos) != previous_mx_my:
        display_mouse = True

    # update buttons and sliders
    return_button.update(mpos, click)
    save_button.update(mpos, click)
    volume_slider.update(click, unclick, mpos)
    if size:
        size_slider.update(click, unclick, mpos)

    # check if settings have been changed from their initial value
    if (round(volume_slider.get_value()) / 100 == settings['volume']
            and (not size
                    or (size and round(size_slider.get_value()) == settings['size']))):
        save_button.set_disabled(True)
    else:
        save_button.set_disabled(False)
```

```
        # check buttons
        if save_button.get_clicked():
            settings['volume'] = round(volume_slider.get_value()) / 100
            if size:
                settings['size'] = round(size_slider.get_value())

            # overwrite settings on file
            with open("settings.json", "w") as file:
                json.dump(settings, file)

            saved = False
        elif return_button.get_clicked():
            return

        if not saved and not pygame.mixer.get_busy(): # changes when no sound is being played
            set_volume()
            saved = True

        # draw everything to the screen
        medium_font.render(screen, "SETTINGS", (SCREEN_WIDTH//2, 3*EIGHT_PIXELS),
                           alignment=CENTER)
        small_font.render(screen, f"VOLUME: {round(volume_slider.get_value())}%",
                          (SCREEN_WIDTH//2, 5*EIGHT_PIXELS), alignment=CENTER)
        if size:
            small_font.render(screen, f"SIZE: {round(size_slider.get_value())}",
                              (SCREEN_WIDTH//2, 8.5*EIGHT_PIXELS), alignment=CENTER)

        return_button.draw(screen)
        save_button.draw(screen)
        volume_slider.draw(screen)
        if size:
            size_slider.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#=====================Login Function=====================#
# same function used for logging the second player in in 2 player
def login(title="LOGIN:", button_text="LOGIN", blocked_names=[]):
    valid_names = db_get_all_usernames(db_cursor)

    for name in blocked_names:
        try:
            valid_names.remove(name)
        except ValueError: # blocked name not used by a player
            pass
```

```
name_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 5*EIGHT_PIXELS),
                   (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                   medium_font, small_font, 4, name="NAME",
                   allowed_strings=valid_names, allowed_characters=UPPER_ALPHABET)
pin_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 68*PIXEL_RATIO),
                  (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                  medium_font, small_font, 4, name="PIN",
                  allowed_characters=NUMBERS, hide=True)
login_button = TextButton((SCREEN_WIDTH//2 - 6*EIGHT_PIXELS//2, 12*EIGHT_PIXELS),
                          (6*EIGHT_PIXELS, 16*PIXEL_RATIO),
                          button_text, medium_font, disabled=True)
return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
                            (2*EIGHT_PIXELS, 2*EIGHT_PIXELS),
                            return_image)

mpos = pygame.mouse.get_pos()
display_mouse = True
while True:
    click = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            quit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                click = True
        elif event.type == pygame.KEYDOWN:
            display_mouse = False

    screen.fill(BACKGROUND_COLOUR)

    previous_mx_my = (mpos)
    mpos = pygame.mouse.get_pos()
    if (mpos) != previous_mx_my:
        display_mouse = True

    # update buttons and text boxes
    name_box.update(mpos, click, event_list)
    pin_box.update(mpos, click, event_list)
    login_button.update(mpos, click)
    return_button.update(mpos, click)

    # if both text boxes have valid inputs, allow the login button to be pressed
    if name_box.get_valid() and pin_box.get_valid():
        login_button.set_disabled(False)
    else:
        login_button.set_disabled(True)

    # check buttons
    if login_button.get_clicked():
        username_tuple = db_find_player(name_box.get_text(), pin_box.get_text(), db_cursor)
```

```
            if username_tuple:
                return username_tuple[0]
            else:
                pin_box.display_message("INCORRECT PIN")
        elif return_button.get_clicked():
            return None

        # draw everything to the screen
        medium_font.render(screen, title, (SCREEN_WIDTH//2, 3*EIGHT_PIXELS), alignment=CENTER)
        name_box.draw(screen)
        pin_box.draw(screen)
        login_button.draw(screen)
        return_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Create Account Function======================#
def create_account():
    taken_names = db_get_all_usernames(db_cursor)

    name_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 5*EIGHT_PIXELS),
                       (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                       medium_font, small_font, 4, name="NAME",
                       not_allowed_strings=taken_names, allowed_characters=UPPER_ALPHABET)
    pin_box = TextBox((SCREEN_WIDTH//2 - 35*PIXEL_RATIO//2, 68*PIXEL_RATIO),
                      (36*PIXEL_RATIO, 16*PIXEL_RATIO),
                      medium_font, small_font, 4, name="PIN",
                      allowed_characters=NUMBERS, hide=True)
    create_button = TextButton((SCREEN_WIDTH//2 - 6*EIGHT_PIXELS//2, 12*EIGHT_PIXELS),
                               (6*EIGHT_PIXELS, 16*PIXEL_RATIO),
                               "CREATE", medium_font, disabled=True)
    return_button = ImageButton((EIGHT_PIXELS//2, EIGHT_PIXELS//2),
                                (2*EIGHT_PIXELS, 2*EIGHT_PIXELS), return_image)

    mpos = pygame.mouse.get_pos()
    display_mouse = True
    while True:
        click = False
        event_list = pygame.event.get()
        for event in event_list:
            if event.type == pygame.QUIT:
                quit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    click = True
            elif event.type == pygame.KEYDOWN:
```

```
                display_mouse = False

        screen.fill(BACKGROUND_COLOUR)

        previous_mx_my = (mpos)
        mpos = pygame.mouse.get_pos()
        if (mpos) != previous_mx_my:
            display_mouse = True

        # update buttons and textboxes
        name_box.update(mpos, click, event_list)
        pin_box.update(mpos, click, event_list)
        create_button.update(mpos, click)
        return_button.update(mpos, click)

        # if both text boxes have valid inputs, allow the create account button to be pressed
        if name_box.get_valid() and pin_box.get_valid():
            create_button.set_disabled(False)
        else:
            create_button.set_disabled(True)

        # check buttons
        if create_button.get_clicked():
            db_insert_player(name_box.get_text(), pin_box.get_text(), db_cursor, db_connection)
            return name_box.get_text()
        elif return_button.get_clicked():
            return None

        # draw everything to the screen
        medium_font.render(screen, "CREATE ACCOUNT:",
                           (SCREEN_WIDTH//2, 3*EIGHT_PIXELS), alignment=CENTER)
        name_box.draw(screen)
        pin_box.draw(screen)
        create_button.draw(screen)
        return_button.draw(screen)

        if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                                  or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
            screen.blit(cursor_image, mpos)

        pygame.display.update()
        clock.tick(FPS)

#======================Upon-Open Menu function======================#
def open_screen():
    login_button = TextButton((SCREEN_WIDTH//2 - 6*EIGHT_PIXELS//2, 4*EIGHT_PIXELS),
                              (6*EIGHT_PIXELS, 2.25*EIGHT_PIXELS),
                              "LOGIN", medium_font)
    create_account_button = TextButton((SCREEN_WIDTH//2 - 14*EIGHT_PIXELS//2, 8*EIGHT_PIXELS),
                                       (14*EIGHT_PIXELS, 2.25*EIGHT_PIXELS),
                                       "CREATE ACCOUNT", medium_font)
```

```
quit_button = TextButton((SCREEN_WIDTH//2 - 5*EIGHT_PIXELS//2, 12*EIGHT_PIXELS),
                         (5*EIGHT_PIXELS, 2.25*EIGHT_PIXELS),
                         "QUIT", medium_font)

mpos = pygame.mouse.get_pos()
display_mouse = True
while True:
    click = False
    event_list = pygame.event.get()
    for event in event_list:
        if event.type == pygame.QUIT:
            quit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                click = True
        elif event.type == pygame.KEYDOWN:
            display_mouse = False

    screen.fill(BACKGROUND_COLOUR)

    previous_mx_my = (mpos)
    mpos = pygame.mouse.get_pos()
    if (mpos) != previous_mx_my:
        display_mouse = True

    # update buttons
    login_button.update(mpos, click)
    create_account_button.update(mpos, click)
    quit_button.update(mpos, click)

    # check buttons
    if login_button.get_clicked():
        username = login()
        if username:
            main_menu(username)
    elif create_account_button.get_clicked():
        username = create_account()
        if username:
            main_menu(username)
    elif quit_button.get_clicked():
        quit()

    # draw everything to the screen
    login_button.draw(screen)
    create_account_button.draw(screen)
    quit_button.draw(screen)

    if display_mouse and not (mpos[X] == 0 or mpos[X] == SCREEN_WIDTH - 1
                              or mpos[Y] == 0 or mpos[Y] == SCREEN_HEIGHT - 1):
        screen.blit(cursor_image, mpos)
```

```
        pygame.display.update()
        clock.tick(FPS)

if __name__ == "__main__":
    set_volume()
    open_screen()
```

7. References

[1] E. Barone, Feb. 2016, Journey of the Prairie King, https://stardewvalleywiki.com/Journey_of_the_Prairie_King

[2] E. Barone, Feb. 2016, Stardew Valley, https://www.stardewvalley.net/

[3] E. Jarvis and M. Turmell, 1990, Smash TV, https://en.wikipedia.org/wiki/Smash_TV

[4] Arcade Spot, browser version of Smash TV, https://arcadespot.com/game/super-smash-t-v/

[5] Medium, Sep. 2023, 'The Fun World of Arcade Games: Simple Enjoyment for Everyone', https://medium.com/@arcadegamerental9/the-fun-world-of-arcade-games-simple-enjoyment-for-everyone-b8fa81a3ea2e

[6] RPGnet, Aug. 2008, 'What makes a game an "arcade" game?', https://forum.rpg.net/index.php?threads/what-makes-a-game-an-arcade-game.410336/

[7] Python Version 3.12, Oct. 2023, https://docs.python.org/3/

[8] Visual Studio Code IDE, https://code.visualstudio.com/

[9] Pygame Version 2.5, Sep. 2023, https://www.pygame.org/docs/

[10] JSON Python library, https://docs.python.org/3/library/json.html

[11] SQLite3 Python library, https://docs.python.org/3/library/sqlite3.html

[12] Aseprite, https://www.aseprite.org/

[13] FamiTracker, https://famitracker.org/

[14] LucidChart, https://www.lucidchart.com/pages/

[15] GeoGebra, https://www.geogebra.org/

[16] Sound Ex Machina, 'Vintage Button 3 sound effect', https://www.soundsnap.com/vintage_button_03_wav

[17] Soundsnap, https://www.soundsnap.com/

[18] MATRIXXX_, Feb. 2019, 'Retro Pew Shot sound effect', https://freesound.org/people/MATRIXXX_/sounds/459145/

[19] Freesound, https://freesound.org/

[20] Giorgos Lazaridis, June 2010, 'How a Key Matrix Works', https://pcbheaven.com/wikipages/How_Key_Matrices_Works/

[21] Wikipedia, 2024, 'Key rollover', https://en.wikipedia.org/wiki/Key_rollover