Survival Shooter

Michael Song 7167

A Level H446 Computer Science Coursework Project

Contents

Front cover:

Zombie: Sidequest Saturday (Forsaken Frontier): Undead Emissaries (adventureaweek.com)

Soldier: Man Camouflage Suit Takes Aim Airsoft Stock Footage Video (100% Royalty-free) 10805159 | Shutterstock

Dirt: dust effect tumblr ftestickers 257284747007212 by @lily1424 (picsart.com)

Analysis

Overview of the Problem

My client George's life has recently become extremely busy with an onslaught of A-levels and UCAS, which makes it even more important to have some downtime to relax and calm down. However, George's usual pastime, intense 3D FPS games such as Valorant or Apex Legends, is proving to be not working as well for him currently. This is because a fiery, fast-paced showdown coupled with the imminent threat of death is probably not the best way to soothe your mind after a difficult day of revision. He has also confided in me that recently he has been dealing with a strange case of acute motion sickness, which kicks in strongest when playing these sorts of 3D first-person games. This has proved to be quite a problem: George does not wish to abandon shooting games, yet with his current situation he has no choice but to look for other methods of entertainment. My intention is to help solve his problem with a game that fits snugly into a middle ground, not too boring, not too intense. The game I have in mind is a top-down 2D shooter game with varying levels of difficulty that can be adjusted at the user's discretion.

How this is solvable

This problem can be solved by computational methods due to its features I will be implementing. The largest, most obvious reason is that it is a video game, where you have control over simulating an interesting experience for the player, which can only be done well via a computer (shooting a bunch of people in real life is not preferable).

For example, calculating projectile trajectories is a task that will require a lot of mathematical computation, more specifically trigonometry functions that I can utilize within my code to calculate coordinates of projectiles at any point.

Keeping track of data such as score/health is also another feature that will require a computational approach. By keeping track of object collisions and boundary detection, the program can automatically update variables regarding the player accordingly. A feature I can implement alongside this is a save/load system that allows the player to continue playing with their previous unlocks and scores after reopening the game.

Control over the game environment such as enemy behaviors, obstacles, etc. make this problem necessary to be solved by a computer. Each character, projectile, item will be instantiated as an

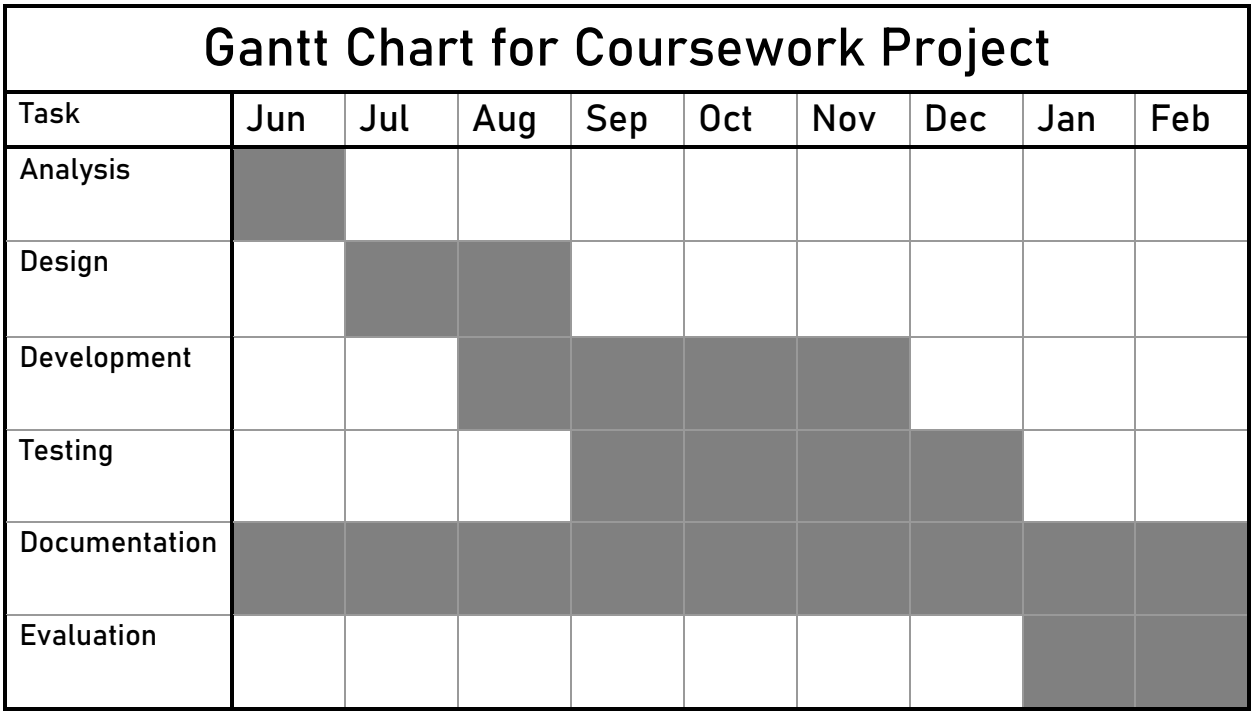
object, each object coming from a class, in an object-oriented programming environment, and the program will have to keep track of the different attributes of each one of them constantly.

Stakeholders

The stakeholders involved in this project will be people with similar situations to my client, therefore in the teenage ≈16 age demographic who need an abstract yet enjoyable experience. The game will be designed to tailor to casual players so the experience is not too stressful and will not be very hardware demanding either. Of course, the simplicity of the game and the ability to select difficulty also means that the game could also be enjoyed by any age demographic. For example, I have a younger brother, age 13, who I know would enjoy this sort of game a lot, since I often see him engrossed in shooting games but isn't quite that good at them yet due to his lack of experience.

Time Scale

In order to visualise the schedule which I will follow during the project, I have created a Gantt Chart which will be easily understandable to follow as a timetable. The deadlines in this schedule won't be exact or compulsory, but will provide me with a framework for how I should organise each stage of the project. Each stage will take place roughly one after the other, with a few overlaps depending on the work in question, and documentation will take place throughout.



Research

Method 1: Questionnaire – This will include broad questions for features I can implement, to make observations about general opinions surrounding the genre of the game. This will be quite easy to carry out, due to the plethora of tools online that allow for easy creation, distribution, and collection of questionnaire data. Furthermore, due to the simplicity involved in filling out a questionnaire, it is likely that I will receive a large amount of feedback since more people would be willing to take a little time to give their opinions. I can also add an optional long answer question at the end, which will help to gauge some specific opinions.

Method 2: Interview – I will also conduct an interview with my client George. During this interview, I will be able to ask much more direct and specific questions regarding George's personal desires for my game's features. This method will likely yield much more useful information because my questions can be tailored in the moment, and I can add follow-up questions based on the direction of the interview.

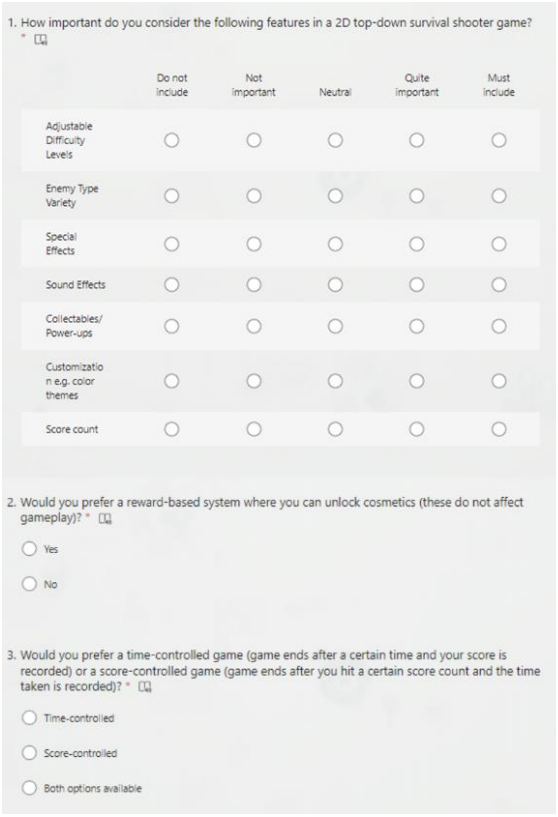
Method 3: Existing Solutions – Finally, I shall be collecting data regarding existing solutions to the problem. By analyzing how others went about implementing their solutions, I can gain a clearer idea for how I should proceed with coding my features and get a few ideas for extra features I could implement.

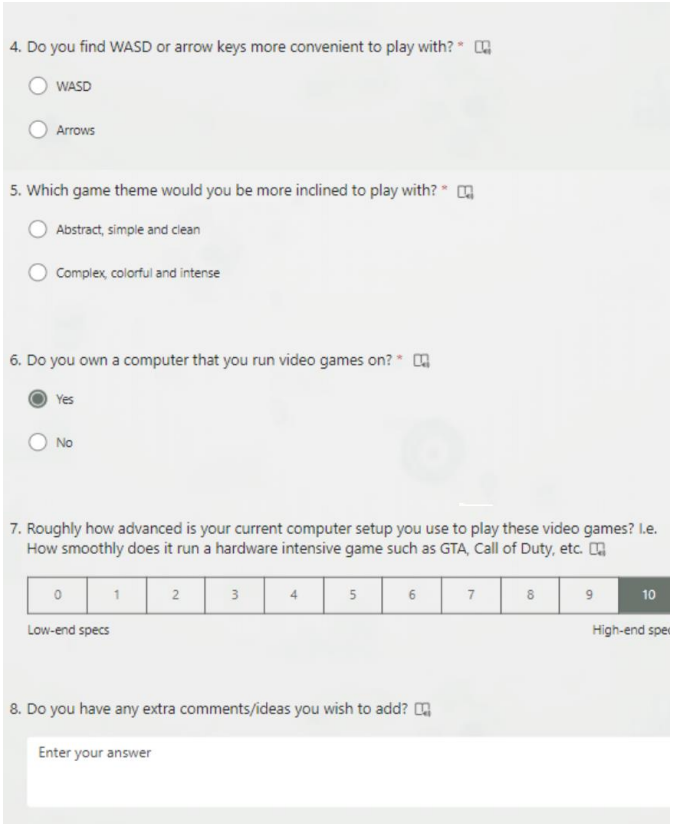
The Questionnaire

As you can see from the screenshots below, each question has been specifically designed to help me make decisions around how I should approach the design of the game:

- Q1-3 will give me an insight into the sort of features my demographic would prefer to be included in my game, so I can focus more on those ones and place less priority on the features more people decided were not as important/didn't think they should be included.
- Q4 allows me to decide on the controls I should code into the game based on majority of user preferences. However if I have time I can give the user the option to select between the two for convenience and allowing everyone's preference.
- Q5 gives me a clearer image of the aesthetic and art style I should use for the game, which will come in handy when designing sprites/environments.

- Q6-7 gives me an indication of how hardware-intensive I should make this game, and also helps me decide on the programming language I should use.
- Finally, Q8 is an optional question I stuck onto the end of the form, to try and gauge some extra engagement and come up with any extra outside opinions for how I should make the game.

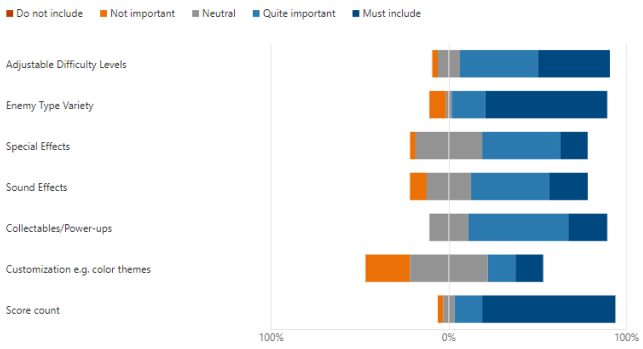




Overall, I received 32 responses to my form. Out of those 32, 23 of them said they own a computer that they can play video games on, which is an amazing number given the total sample size, because a large proportion matched my target demographic. Below are the stats for the responses:

1. How important do you consider the following features in a 2D top-down survival shooter game? (0 point)

More Details



Most of the respondents thought that adjustable difficulty levels, enemy type variety, collectables/power-ups and score count were the most important features to implement in a 2D shooter, while sound effects, special effects and customizations are perhaps less important to prioritize.

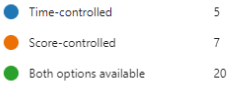
2. Would you prefer a reward-based system where you can unlock cosmetics (these do nc gameplay)?

More Details

The majority voted yes for a battle-pass-like reward system. However, I feel that it is important to first focus on the game itself before moving onto implementing this (a desirable feature).

3. Would you prefer a time-controlled game (game ends after a certain time and your score is recorded) or a score-controlled game (game ends after you hit a certain score count and the time taken is recorded)?

More Details



For the game modes themselves, most people wanted to have the option to choose between the 2 modes. However, time constraints must be considered, so I will first code the score-controlled mode since that had more positive engagement.

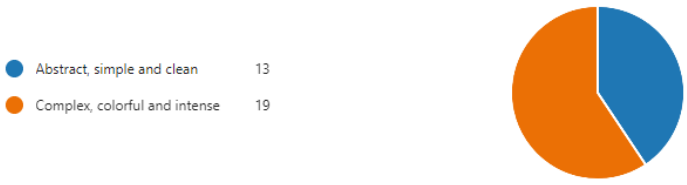
4. Do you find WASD or arrow keys more convenient to play with? (0 point)

More Details Insights

This question was asked to gauge opinions on what controls I should code into the game. This one had a clear-cut winner, with 28/32 voting WASD. Again, however, if I have time, I can code both.

5. Which game theme would you be more inclined to play with? (0 point)

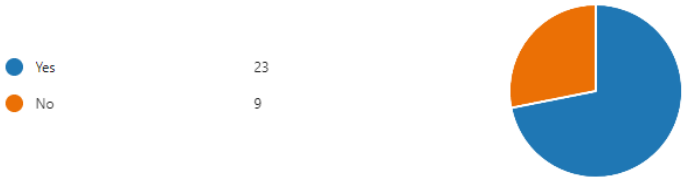
More Details Insights



This question yielded a more mixed set of responses, with similar numbers opting for either abstract or simple. I feel that this feature can be decided later down the line since it doesn't impact too heavily on the coding aspect.

6. Do you own a computer that you run video games on? (0 point)

More Details Insights



This question was a pre-empt for the next question, but also as stated above gave me an indication for the proportion of the sample that were my target demographic.

7. Roughly how advanced is your current computer setup you use to play these video games? I.e. How smoothly does it run a hardware intensive game such as GTA, Call of Duty, etc.

More Details

As you can see, I received quite a large spread results for people's computer performance. However, it tended closer to the lower end, so I will have to take that into account when coding the game.

For Q8, I asked people if they had any extra ideas/opinions for the game. I received quite a few sensible answers that I can consider for my game, highlights included below:

"Regarding question 5, a gritty art style can also be good looking, and it used the low number of pixels/bits to carve more detail and shadow into enemies or monsters I suppose. Don't rule this option out."	I quite liked this suggestion. In terms of deciding a game theme, the questionnaire did not help much in deciding on a clear better option, however the idea given here could potentially be a good consideration when eventually making the decision. This way, I could make the game aesthetic both good-looking and simple to create at the same time.

"I want movement tech"	Movement tech would be a cool feature to have on the surface, however, given that I am making a 2D top-down game, I am not sure how one would implement this, so I may overrule this one. However, if in research I happen to find a way, I will certainly try to see if I can implement something of the sort.
"Have an option for opponents to have shields, i.e., more challenging aspects of the game"	I liked this suggestion, as it goes along with my previous idea of having a variety of enemy types, and an enemy who can equip shields could be one of them. This would probably be a challenging enemy, so would only exist in a harder game mode.
"Make it PvP and have an Elo system"	PvP involves other players acting as the player's enemies, but the problem with this is it gets extremely difficult to control the difficulty of the game since it largely depends on the opponents. The Elo system also applies here to PvP, so same goes for that.
"WASD and arrow keys depends on what hand you like to move with mouse as you can use the other hand for movement"	This suggestion goes along with my earlier thought where I considered coding both controls in and allow the user to choose at their own discretion. Doing this would allow me to cater for a larger demographic and make the game more comfortable and ergonomic for a larger range of users.

"Add boss type enemies"	Again, this suggestion was another good one since it supported my idea of having multiple enemy types, and a boss enemy would be a challenging and fun aspect of the game that could be altered and scaled based on chosen game difficulty.
"Add camera shake when you kill people because that's always cool"	This is a good suggestion, however in my opinion it would distort the user's vision and make the experience more disorienting, which is something I wanted to avoid for my client. If time allows, I could think about implementing this as an optional feature for those who want a more intense experience.
"Good luck!"	I received a lot of these types of comments on this open question, suggesting my proposed solution was received quite positively by my target demographic!

The Interview

Next, I conducted a formal interview with my client George to gain a better grasp of his vision and how I should shape the game to fit his needs and interests. Below is the transcript of the interview, grammar corrected and paraphrased where necessary:

Me	Hello George, thanks for taking the time to do this interview. Following the results of the questionnaire, we found the features that most people found most and least important. Do you have anything to comment on?
George	I agree with most of the results in Q1. I think that a variety of difficulties, enemies and power-ups would make the game far more enjoyable. However, I think sound effects are still quite a major part of the user experience, so personally I would rank it higher up. As for the reward system, I think it would

	be great if it was included, but I don't think an absence of this sort of system would be a detriment to the game.
Me	The questionnaire also stated that most people would prefer the option to choose between both time-controlled and score-controlled game modes. Which game mode would you prefer?
George	I would prefer the option to choose between the 2, as I would have more fun trying out both. Another idea: an endless mode would be nice, where you keep playing with no constraints until you either die or quit manually, with your final score logged.
Me	Ooh, and endless game mode does sound quite exciting. In that case, what do you think the control parameter values should be for time and score-controlled?
George	I think 2 minutes would be good for time-controlled mode. Score controlled would depend on how frequently you score points in your game, but I'd say a game should last roughly 2-3 minutes there as well for an average player. Any longer and it would get quite tedious, and you know how busy I am nowadays.
Me	I'll take those into account. How about the game theme? There seemed to be mixed views on whether I should take the abstract route or the complex route.
George	Complex would certainly be more eye-catching and flashier, and I would definitely have more fun with it, but my acute motion sickness might not enjoy it as much, so abstract may have to be my choice. But you can always make something abstract yet eye-catching at the same time, right?
Me	Of course. I did get some good ideas from the questionnaire which I could think of implementing. The next question is to do with your overall opinions on this genre of games. Are there any views you hold that haven't been mentioned so far?
George	As I've already said, my acute motion sickness doesn't play very well with intense graphical scenes, so recently I haven't been doing very well with this sort of game. The screen gets extremely crowded and way too intense, which isn't what I want thanks to my acute motion sickness. I think what I'm looking

	for is a more calm, relaxed gameplay that is still a fun shooting survival experience.
Me	Interesting, I'll make sure to design the game to fit around that. Finally, what are your computer specs?
George	I have 8GB RAM, 500GB of storage, an AMD Ryzen 5 5600 Processor and AMD Radeon RX 5700 XT GPU. It runs Windows 11 OS.
Me	Thank you. That should be everything. Thank you for doing this interview with me George, you have been such a great help.
George	The pleasure's all mine!

From this interview, I collected the main points and built a summary of information based on the feedback received from George:

- Have a variety of difficulty levels for users to choose from and battle a variety of enemy types.
- Include sound effects.
- A reward system is good to have but wouldn't matter much if I didn't make it.
- Give the user the option to select between time and score controlled modes, as well as an endless mode with no constraints for players who do not wish to be constrained by timed and scored limitations.
- Complex would be well received, but client would prefer abstract due to their situation. Could perhaps find some middle ground that won't stray too far from both ends.
- Enjoys a challenge but doesn't like it when the usual top-down shooters fill his screen up with enemies, practically making it impossible to play and overwhelming him, making it a not-so-enjoyable experience. Would prefer something calmer but still the right amount of intensity.
- Computer specifications: 8GB RAM, 500GB Storage, AMD Ryzen 5 5600 Processor, AMD Radeon RX 5700 XT GPU, Windows 11.

Existing Solutions

As well as the above research, I also delved into the following existing solutions, which also happen to be the ones George mentioned in the interview but didn't really enjoy thanks to his situation.

Figure 1 - Survivor.io logo

Survivor.io

The first solution I came across was a top-down shooter game very reminiscent of the idea I had. It contained all the basic features I intend to implement, such as power-ups, enemy variations, etc. Games are played in a timed fashion, where you play until your death and the length of the game is recorded.

The game works by having the player fixed in the center of the screen, and the environment moving in the opposite direction to the player's "movement" to create the illusion of the player wandering around the environment. For this to work, extensive calculation would have had to happen to calculate dynamic coordinates for everything in the game world since as well as their own coordinate behavior they would also have to be affected by the player's behavior.

Several features that I found were quite interesting were the multiple worlds that the player could level up into based on experience and score earned. There was also a plethora of different power-ups the player could gain, and these were achieved by filling an XP meter up (gained by killing enemies). These included rocket launchers, shields, daggers, speed/health potions, etc. The game also utilized a wave system, where every couple of minutes it would increase the difficulty by bringing in a more challenging wave of enemies, sometimes topped off with a boss fight.

However, one thing I found incredibly frustrating was the speed at which the game would suddenly spawn into the world a huge crowd of enemies that were extremely difficult to fight off, especially for those less experienced and less equipped. This was what my client was complaining about in the interview, so this was something I was not looking to replicate.

Another quirk of the game was that of the movement mechanics. The game works with only 1 control, movement. The aiming and firing were done automatically at regular intervals, and either randomly or in a systematic fashion depending on the weapon. This is not what I have in mind for my game, since I am planning to code both keyboard and mouse control, where the keyboard controls movement and the mouse controls aiming/firing.

Finally, this game also reminded me that neither of my two modes account for if the player dies, so I will need to have a good think about how that will work.

Figure 2 - Zombs Royale logo

Zombs Royale

This game is also a 2D top-down shooter that works in the same way as survivor.io in terms of its mechanics (player static in center), but the game mechanics are slightly different. This one is more like a battle-royale game, where all players land on a map and must collect weapons, power-ups, etc. to kill each other. Each player has their own inventory and health bar that can store a whole host of items.

This game is online, which is different from my version which I intend to be single player against the computer. So, while I can't exactly replicate this aspect of the game, there are still many features to this solution that I can use for my game. For instance, I quite liked the idea of an inventory where you can store items to use, and this game has a proper battle pass like most mainstream video games. The game mechanics are also quite like what I intended, which is keyboard for movement and mouse for aiming. The map environment also consisted of chests, obstacles/barriers and even buildings that the player could enter.

During research, this game seemed to be the best existing solution out of all the games I found. It resembled almost everything I was planning with my game, had almost the exact same game mechanics, and included most of the features I intended to implement as well as a few more that I thought were amazing ideas.

The only downside was the online aspect, where PvP was involved. When I inquired about this game to George, he admitted that he did try this game as an alternative but found that the servers he joined were often filled with "sweaty try-hards", meaning his opponents usually were seasoned players who were extremely skilled at the game, resulting in George getting killed within the first minute or so. On occasions where he somehow managed to survive the initial bloodbath, he complained that due to the size of the map and the pace of the game, it often gets quite boring since a typical game lasts about 10-15 minutes. Of course, George doesn't have that time on his hands. Therefore, this is a problem I aim to solve when coding my version of the game. The single player mechanics will give me greater control over the difficulty levels, since making it PvP would cause the difficulty to largely depend on the opponent's skill level and not be in the user's control, something I sought to help George with from the very beginning. Furthermore, by setting limits to my game-modes, I can ensure the game does not drag on too long or too repetitively and stays exciting throughout.

Overview of the Solution

Following extensive research into my target demographic's preferences and the opinions of my client George, as well as delving into existing solutions to gauge a clearer idea for the path I want to take, I have compiled a list of essential features to the game, as well as desirable features/limitations which could be beneficial to the user's experience but possibly can't implement due to time constraints and complexity.

Essential Features:

- A graphical 2D user interface with the player fixed in the center of the screen, the environment scrolling around it, including moving enemies, obstacles, and projectiles.
- A game menu that allows for selection of game difficulty, game mode, etc.
- Power-ups that the player can equip to aid their in-game progress.
- A variety of enemy types, each with their own sprite design, attributes, powers, etc.
- The player has health attributes and an inventory to display power-ups.
- Time and score-controlled game modes, both should last roughly 2-3 minutes long. If the player dies, the game ends and the final score/time is logged, along with the fact that the player cut the game short with untimely death. And an endless mode, with no constraints and the game continues until the player either dies or quits, and the final score is recorded.
- Easy, normal, hard difficulty levels.
- Sound effects for an immersive experience, e.g., gun shots, footsteps, enemy noises, etc. Music?
- A score count in the game to keep track of game progress.

Desirable Features:

- A reward system with unlockable cosmetics (like a battle pass), including different sprite skins that the user can choose to switch between before a game (these would not affect the gameplay, simply for aesthetic purposes).
- Option to select game theme, e.g., color schemes, game environment, complex/abstract, etc.
- Option for user to select between WASD and arrow controls in settings.
- Optional camera shake?

- Boss battles at regular intervals.

Limitations:

- No more than 5 enemy types, since any more would require a lot of work with sprite designs, behaviors, etc.
- No more than 5 power-ups, since too many would take too long to code each one's functions.
- No other game modes, since 3 game-modes is already a lot of computation and time.
- No PvP/Elo system, since the implementation of this would take too long and may stretch outside my skill level/software requirements and doesn't fit George's needs.

Language Choice:

Based on feedback from both questionnaire and interview, I have decided to go ahead with *Python* and *Pygame* to code my game. I did consider utilizing existing game engines such as Unity or Unreal Engine, but since my experience with the C family of languages is currently limited, using those game engines would end up being quite inefficient in terms of project time constraints, and my familiarity with python means I can spend less time getting to grips with the language and more solving the problem. Furthermore, I seek the personal fulfilment of building a game from scratch by coding it largely by myself, which is unobtainable when using a game engine like Unity where it's possible to make most of the game without even writing any code. To code a graphical interface, however, I will need to import the Pygame module, which introduces a whole host of new syntax that I will need to familiarize myself with.

Hardware Requirements:

	Python 3.11	Client's PC
RAM		8GB
Hard Drive Storage	100MB	500GB
Processor	Intel Core i5	AMD Ryzen 5 5600
Operating System	Windows 7 or later	Windows 11
Clock Speed	2.5GHz	3.5GHz

- As you can see, my client will be able to run the solution on his PC setup.

Software Requirements:

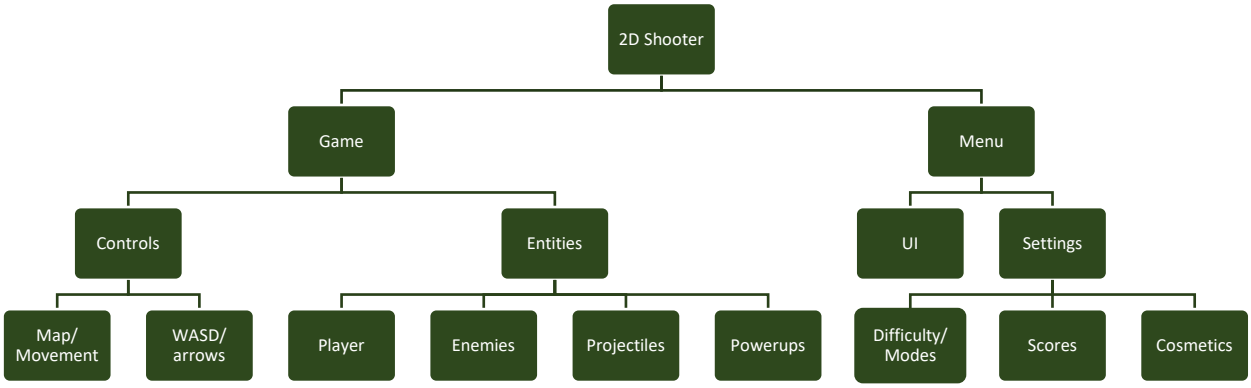
- Of course, to run the game my client will need to have both python and the pygame module installed on his PC. However, if I end up compiling the solution as an .exe file, he will be able to run the game without any further installations.

END OF ANALYSIS

Design

After extensive analysis of the problem and existing solutions, I have a clear idea of the general design of the game. The first step towards the design is to decompose the problem into more manageable sub-problems, which will allow us to easily understand and solve each section of the problem and eventually allow us to be able to create a solution for the whole problem.

Top-Down Design



I have decomposed the game into 2 initial sections: the actual game and the menu. I have then further broken down each of these sections into relevant subproblems, such as the mechanics of each entity involved in the game, and the game mechanics itself such as movement and environment. The menu section has been made separate from the game, so I can focus on the game itself before moving onto outside-game things such as settings and scores. Decomposing the problem this way allows me to focus on each individual aspect of the game, every time ascending a level on the top-down diagram until I solve the full problem.

Controls

Map and Movement

The mechanics of the movement of this game will have the main character static in the middle of the screen, while the map and the rest of the environment pans around it in an inverted direction of motion. This will create the illusion of the character moving around with the camera around the environment, while keeping the coordinates of the sprite itself constant.

```
WHILE game not finished:
      IF input is left:
            move map right
      IF input is right:
            move map left
      IF input is up:
            move map down
      IF input is down:
            move map up
      PRINT map
      PRINT main character @ center of screen
      NEXT frame
ENDWHILE
```

To keep track of the player's location on the map, I can implement a camera scroll variable that stores the current coordinates of the camera (i.e. the world coordinates of the player relative to the map as opposed to the screen coordinates) that will be used not only to determine where to render the map on the screen but also to act as an anchor for all other objects in the game window, as they will all need to remain fixed at their location on the map, albeit sometimes with movement of their own, which we will deal with later. The player will begin at the center of the map, initialized as (0,0). Here is a revamped version of the previous code:

```
set camera_scroll to [0,0]
WHILE game is not finished:
      IF input is left:
            camera_scroll[0] -= player_speed
      IF input is right:
            camera_scroll[0] += player_speed
      IF input is up:
            camera_scroll[1] -= player_speed
      IF input is down:
            camera_scroll[1] += player_speed
      PRINT map @ ( -(map_w - screen_x)/2 - camera_scroll[0], -(map_h -
window_h)/2 - camera_scroll[1] )
```

```
        PRINT player @ center of screen
        NEXT frame
ENDWHILE
```

Control Options

For controls, I can implement a setting which will convert left, right, up, and down inputs between WASD and arrows according to the player's settings.

Entities

The entities in this game will follow an OOP approach, i.e. I will be making each type of entities a class, and have all instances of entities in the game be objects of those classes.

Player

As stated before, my player will be static in the middle of the screen. To make sure it is in the center, I will be implementing the following code to ensure it is in the center, while also accounting for the size of the sprite itself as the pygame coordinate system uses the top left of the image to place it.

Figure 3 – Sample Player Sprite Graphic

```
set center_x, center_y to screen width / 2, screen height / 2
set player_w, player_h to player sprite width and height
PRINT main character @ (center_x – player_w/2, center_y – player_h/2)
```

The Player will have certain attributes for the gameplay, which we can initialize when creating the class for the player as follows:

```
Class Player:
        Private x
        Private y
        Private health
        Private speed

        Public procedure new(newspeed, newhealth):
                x = center_x – player_w/2
                y = center_y – player_h/2
                health = newhealth
                speed = newspeed
```

```
    ENDPROCEDURE
ENDCLASS
```

Enemies

The design for the enemy movement will be slightly more complicated. This is because unlike the player whose coordinates are fixed, the enemies' coordinates not only need to have their own movements but must also move around in a fashion relative to the player's movement and the map. Luckily, with one camera scroll coordinate system we can anchor all objects around this camera so that everything is fixed relative to each other. Currently, we will have the enemies follow the player i.e., the center coordinates of the screen, since the enemy's main goal is to kill the player.

Figure 4 - Sample Enemy Sprite Graphic

```
SET enemy_x, enemy_y to enemy coordinate position
WHILE game is not finished:
    IF enemy_x < center_x + camera_scroll[0]:
        Move enemy right
    ELIF enemy_x > center_x + camera_scroll[0]:
        Move enemy left
    IF enemy_y < center_y + camera_scroll[1]:
        Move enemy down
    ELIF enemy_y < center_y + camera_scroll[1]:
        Move enemy up
    NEXT frame
ENDWHILE
PRINT enemy @ (enemy_x - camera_scroll[0], enemy_y - camera_scroll[1])
```

The enemy of course will provide a threat, which is the ability to deal damage to the player. This could be either through touching the player or shooting the player. Since shooting concerns projectiles, for now we will focus on touching the player. First, we will make a class for the enemy.

```
Class BasicEnemy:
    private damage
    private damage_speed #speed at which health decrements for player
```

```
    private x, y
    private health
    private speed
    public procedure new():
        x = random(x coord on map)
        y = random(y coord on map)
        health = 50
        speed = 1
        damage = 10
        damage_speed = 0.5
    ENDPROCEDURE
ENDCLASS
```

Now we need a way to cause the player and the enemy objects to interact with each other, and for that we will use collision detection. To make sure that the player's health does not go down too fast given the FPS, we will implement a damage timer to ensure that each decrement to the player's health is between given time intervals while the player is touching the enemy.

```
FOR enemy IN enemies:
    Reset this enemy's enemy_damage_timer to 0
    IF enemy collides with player:
        IF enemy_damage_timer == 0:
            player.health -= enemy.damage
        ENDIF
        enemy_damage_timer += 1
        IF enemy_damage_timer <= enemy.damage_speed * FPS:
            reset enemy_damage_timer to 0
        ENDIF
    ELSE:
        reset enemy_damage_timer to 0
    ENDIF
```

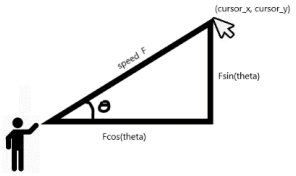
With a basic enemy class in place, this makes the creation of other enemy types incredibly simple, since we can just instantiate more enemies with different starting attributes such as health, speed and sprite images.

Projectiles

There will be 2 types of projectile objects: player projectiles and enemy projectiles. The player projectiles will be simple since they are being instantiated from the center of the screen. The mechanics for the firing of these projectiles will be triggered by user click input, and the program will calculate the trajectory of the projectile using trigonometry, as follows. I will also put the projectile into its own class to make both the declaration and instantiation of the projectiles simpler.

```
Class Projectile:
      Private speed
      Private x
      Private y

      Public procedure new(cursor_x, cursor_y, setspeed, setdamage):
            speed = setspeed
            damage = setdamage
            x, y = center_x, center_y
            dxdy = calc_trajectory(cursor_x, cursor_y)
      ENDPROCEDURE

      private function calc_trajectory(cursor_x, cursor_y):
            angle = arctan((y - cursor_y)/(x - cursor_x))
            dx = speed * cos(angle)
            dy = speed * sin(angle)
            return dx and dy
      ENDFUNCTION

      public procedure main():
            x -= dx
            y -= dy
            print projectile @ x,y
```

```
    ENDPROCEDURE
ENDCLASS
```



The maths for this is using basic vector resolving trigonometry, where the angle θ is found using arctan, then the trajectory (the vector) is resolved into horizontal and vertical components, $F\cos\theta$ and $F\sin\theta$ respectively, where F is the scalar speed of the projectile.

For the enemy projectiles, the basic trajectory calculation by vector resolving will be very similar: the difference is where the projectile instantiates from and where it shoots towards, as well as automatic firing since enemies are not controlled by a player directly. I will make another class for enemy projectiles, but have it be a child of player projectiles since they would share a lot of attributes and methods. In this case, the only significant difference really is the direction the projectile is shooting at, which is the center of the screen where the player is. Therefore, all angles will be calculated with said target in mind.

```
Class EnemyProjectile inherits Projectile:
    Public procedure new(speed, damage, origin_x, origin_y):
        Super.speed = speed
        Super.damage = damage
        x = origin_x
        y = origin_y
        dxdy = calc_trajectory()
    ENDPROCEDURE

    public function calc_trajectory():
        angle = arctan((y - center_y)/(x - center_x))
        dx = speed * cos(angle)
        dy = speed * sin(angle)
        return dx and dy
    ENDFUNCTION

    ENDPROCEDURE
ENDCLASS
```

Of course, projectiles are nothing but simply projectiles without anyone shooting them, so we will have to implement projectile firing for both the player and the enemies.

For the player:

```
projectiles = []
IF user clicked:
      cursor_x, cursor_y = where user clicked
      add Projectile(cursor_x, cursor_y) to projectiles
ENDIF

FOR projectile in projectiles:
      projectile.main()
ENDFOR
```

For the enemy, due to the fact there will be multiple types of enemies ranging from melee-only to shooting, it would probably be wise to create a separate class of shooter enemies, with it being a child class of the basic enemy class we have already created.

```
Class ShooterEnemy inherits BasicEnemy:
      Private projectile_freq
      Private projectile_cooldown
      Public procedure new(setspeed, sethealth, setdamage,
setdamagespeed, setprojectile_freq):
            Super(speed, health, damage, damage_speed)
            projectile_freq = setprojectile_freq
            projectile_cooldown = setprojectile_freq
      ENDPROCEDURE
ENDCLASS
```

We can now iterate through every enemy in the enemy list, containing both basic and shooter enemies, since they both share the same methods. For the shooter enemies, the following code will implement the firing algorithm, which uses a timer (projectile_cooldown) to fire a projectile, and then wait a specific time (the projectile_freq) to cooldown before firing again.

```
FOR enemy in enemies:
      enemy.main()
```

```
    ...
    MELEE DAMAGE CODE (inheritance and polymorphism allows both
    shooter and basic enemies to follow this section of code,
    explained earlier)
    ...
    IF enemy is ShooterEnemy:
        IF enemy.projectile_cooldown reaches 0:
            add EnemyProjectile(5, 10, enemy.x, enemy.y) to
            projectile list
        ENDIF
        increment enemy.projectile_cooldown
        IF enemy.projectile_cooldown >= FPS*enemy.projectile_freq:
            reset enemy.projectile_cooldown to 0
        ENDIF
    ENDIF
ENDFOR
```

The projectile list can now be iterated over, and thanks to the inheritance this makes the code extremely simple and efficient (both enemy projectiles and player projectiles are inside the list, and they both have different versions of the main procedure which ultimately prints):

```
FOR projectile in projectiles:
    projectile.main()
    IF enemy projectile collides with player:
        player.health -= projectile damage
    ENDIF
ENDFOR
```

Powerups

With the basic player, enemy and projectile mechanics designed, we can now go a layer deeper and begin to vary their behaviours and attributes through various powerups.

An List of ideas for powerups is as follows:

- Health boost – Increases player's health.
- Speed boost – Increases player's speed for a set amount of time.
- Shield – Prevents any damage done to the player for a set amount of time.
- Poison – Player deals damage to enemies by touching them instead of the other way round for a set amount of time.
- Freeze – Causes all enemies to stop moving for a set amount of time.

A basic algorithm for the powerups would consist of a dictionary of all powerup types and whether they are active or not, a list of currently existing powerups on the map, and a collision detection algorithm to check whether the player has picked it up or not. If it has been picked up, remove the powerup from the list and apply its corresponding upgrade to the other attributes, and update the dictionary.

```
powerup_status = {"Speed":0, "Shield":0, "Poison":0, "Freeze":0}
current_powerups = []
FOR powerup in current_powerups:
      display powerup at its x, y position on map
      IF player collides with powerup:
            remove powerup from current_powerups
            powerup_status[powerup] = 1
      ENDIF
ENDFOR
```

For the spawning of the powerups, I will use a random function to make each spawn have a random chance of spawning. Here, FPS*15 means there is a chance of a powerup spawning every 15 seconds, but we could replace this with any number to alter the frequency of spawns.

```
WHILE game is not finished:
      IF random(0, FPS*15) == 0:
            Add a random powerup to current_powerups
```

```
      ENDIF
ENDWHILE
```

It would probably be wise to implement a class for powerups, since each powerup will have individual attributes and features such as their icon, their x, y position, etc. Then, when we add a random powerup to the current_powerups list like above, we would instantiate a new object using this class and append it to that list.

```
Class Powerups:
      private id
      private icon
      private x, y
      Public procedure new(powerup_id):
            id = powerup_id
            icon = sprite image name found using powerup_id
            x, y = random coordinates on map
      ENDPROCEDURE
      Public procedure main():
            print icon @ (x-camera_scroll[0], y-camera_scroll[1])
      ENDPROCEDURE
```

Now we can place these powerups on the map, When the player picks them up, we will update the powerup status dictionary, which will allow us to know which powerup functions to run.

```
FOR powerup in current_powerups:
      powerup.main()
      IF player collides with powerup:
            remove powerup from current_powerups
            powerup_status[powerup] = 1
      ENDIF
ENDFOR
FOR powerup in powerup_status:
      IF powerup_status[powerup] == 1:
            run powerup
      ENDIF
```

```
ENDFOR
```

As you can see here, the run powerup line of code is quite ambiguous. This is because each powerup will have their own algorithm based on their functionality, which I will explain below.

Health

This powerup boosts the player's health. The med-kit icon is designed to reflect the idea that it aims to heal any of the player's injuries by increasing their health bar. I stole it from fortnite.

The algorithm for this powerup is very simple: we simply need to access the player's health attribute and increase it. However, to maintain fairness, we must make sure that there is a cap to the health attribute, ensuring that the player is not able to stockpile health boosts and have an unreasonable amount of health.

```
IF player picks up health:
      player.health increase by health_boost
      IF player.health is greater than max_health:
            Set player.health to max_health
      ENDIF
ENDIF
```

Speed

This powerup grants the player a speed boost. The icon for this powerup aims to reflect the Flash-like aspect of the powerup, hence it is the character's recognizable lightning bolt.

The algorithm for this powerup will also be quite simple to implement. However, the speed boost will only be active over a set amount of time. Hence at the end of this period, the speed boost must be removed. As well as this, the speed boost along with all other boosts except health will be unstackable (picking up multiple speed boosts while speed boost is still active will not further increase the player's speed, it will only refill the timer): this is to ensure gameplay is balanced throughout, regardless of how many powerups the player has active.

```
SET speed_duration to amount of time allowed for speed boost
IF powerup_status["speed"] == 1:
```

```
      speed_timer += 1
      IF speed_timer <= speed_duration:
            player.speed increase by speed_boost
      ELSE:
            powerup_status["speed"] = 0
      ENDIF
ENDIF
```

Shield

This powerup does exactly what it says on the tin: it provides a shield for the player, allowing them to be immune to all enemy damage for a fixed amount of time. This is shown by the protective shield design of the icon.

The algorithm for the shield will be slightly more complicated than the previous ones. This is because instead of just changing one attribute/variable, we will have to go through the code and add to all sections that deal with the player taking damage as well as add an extra attribute for the player that decides whether the shield is on or off. The pseudocodes for all following powerups will skip the powerup duration for abstraction purposes.

```
IF powerup_status["shield"] == 1:
      Turn on player.shield
ELSE:
      Turn off player.shield

######## Main Code ########
FOR enemy in enemies:
      IF enemy collides with player:
            IF player.shield is off:
                  ... Enemy damages player ...
            ENDIF
      ENDIF
ENDFOR
FOR projectile in projectiles:
```

```
      IF enemy projectile collides with player:
            IF player.shield is off:
                  ... Projectile damages player ...
            ENDIF
      ENDIF
ENDFOR
```

Poison

This powerup flips the script for the player and allows them to deal damage to the enemies by touching them instead of the other way around for a fixed amount of time. The design of this icon signifies the power the player has over the enemies, as well as the "poison" aspect of the powerup.

The algorithm will require the creation of another player attribute, which can determine whether the player has the poison effect or not. If poison is on, the player can then deal damage to the enemies by detecting collisions between the player and the enemies. We can simply modify the enemy damage code for this.

```
IF powerup_status["poison"] == 1:
      Turn on player.poison
ELSE:
      Turn off player.poison
ENDIF
######## Main Code ########
FOR enemy in enemies:
      IF enemy collides with player:
            IF player.poison is on:
                  ... Player damages enemy ...
            ELSE:
                  IF player.shield is off:
                        ... Enemy damages player ...
                  ENDIF
            ENDIF
```

```
        ENDIF
ENDFOR
```

Some logic is required to string the poison and the shield powerup together. These 2 powerups could be activated either at the same time or on their own, each state achieving a different outcome:

Poison ON, shield ON: enemy takes damage from player.

Poison ON, shield OFF: enemy takes damage from player.

Poison OFF, shield ON: neither take any damage.

Poison OFF, shield OFF: player takes damage from enemy.

This logic is reflected in the pseudocode above, with some simplification.

Freeze

This powerup causes all enemies to become frozen in place, unable to move. The icon here reflects that idea, since being frozen is associated with ice and cold.

The algorithm for this powerup will be quite simple. Thanks to our inheritance and polymorphism of our enemies, we can simply go to the enemy class where the movement of the enemies is run, and add an if statement to it, as well as an extra attribute to determine whether the status of the enemies is frozen or not, which will affect all instances of enemies, regardless of enemy type.

```
IF powerup_status[“freeze”] == 1:
        Turn on enemy.freeze for enemy in enemies
ELSE:
        Turn off enemy.freeze for enemy in enemies
ENDIF
######### Enemy class ########
Public procedure main():
        IF enemy.freeze is off:
                ... Enemy moves ...
        ENDIF
ENDPROCEDURE
```

Menu

UI Design

My initial design for the menu will be an initial screen with just the title, play and config settings options. Then when the player clicks play, they will be taken to a different menu with new options including difficulty, modes, high scores, etc. as well as returning to the home screen. Here is a bare bones implementation of the main menu.

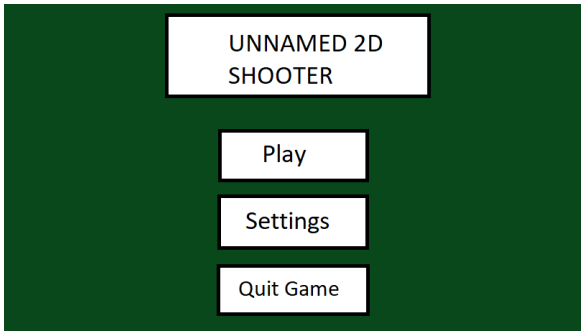


Figure 5 - Main Menu

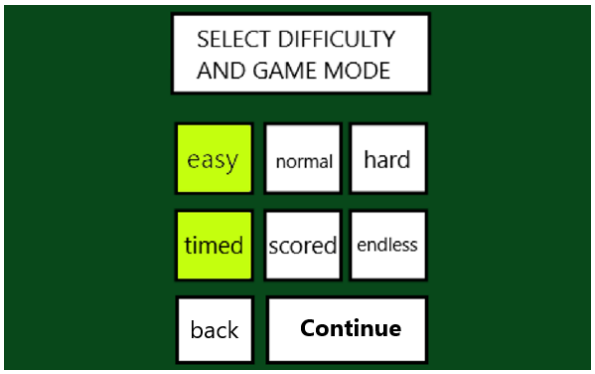


Figure 6 - Game Options Menu

After pressing play, they would be transferred to this next page, where they can select a difficulty and a game mode. The green highlight indicates which option the player selected, i.e., a toggle menu. Once the player has selected their options, they can press continue to move onto other options, or press back to go back to the main menu.

Given that game themes are an optional implementation, this menu screen would be an optional one to implement. But if I did, it would consist of several options for world themes, such as a grassland world, a desert world, a snow world, a cave world, etc. However, given the time constraints for this project there may not be time to implement these, which is why it is optional.

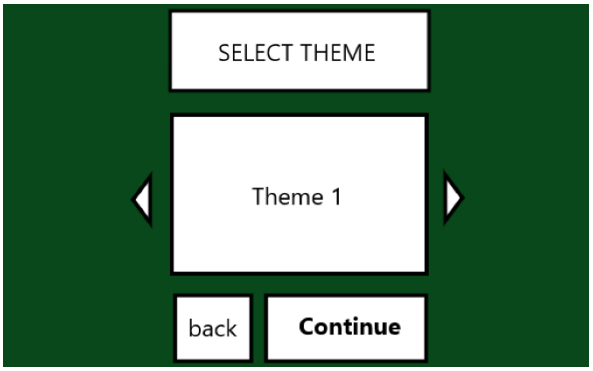


Figure 7 - Game Theme Menu (Optional)

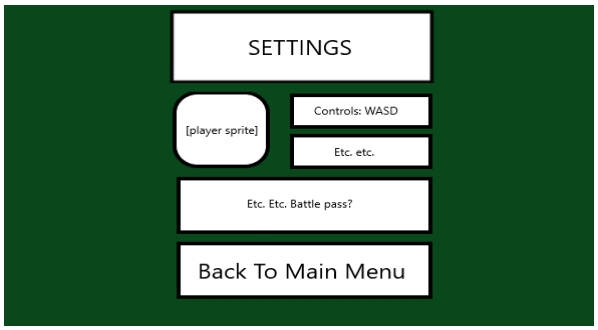


Figure 8 - Settings Menu

Back to the main menu, when the player clicks on settings this menu should appear, with various settings available to alter such as controls and player skin. You may also notice a "battle pass?" in there, which is a desirable feature that may not end up being in there due to time constraints.

We can express these menus in pseudocode to make the relationships between each menu state clearer.

```
menu_stage = 0
WHILE menu_stage < 4:
      IF menu_stage == 0:
            IF play button clicked:
                  menu_stage = 2
            ELSE IF settings clicked:
                  Menu_stage = 1
            ELSE IF quit clicked:
                  quit()
            ENDIF
      ENDIF
      IF menu_stage == 2:
            IF easy clicked:
                  diff = diff(easy)
            ELSE IF normal clicked:
                  diff = diff(normal)
            ELSE IF hard clicked:
                  diff = diff(hard)
            ENDIF
            IF timed clicked:
                  mode = mode(timed)
            ELSE IF scored clicked:
                  mode = mode(scored)
            ELSE IF endless clicked:
```

```
            mode = mode(endless)
      ENDIF
      IF back clicked:
            menu_stage = 0
      ELSE IF continue clicked:
            menu_stage = 3
      ENDIF
ENDIF
IF menu_stage == 3:
      themes = [theme0, theme1, theme2, ...]
      theme = 0
      IF right arrow clicked:
            theme += 1
            IF theme == len(themes):
                  theme = 0
            ENDIF
      ELSE IF left arrow clicked:
            theme -= 1
            IF theme == -1:
                  theme = len(themes) - 1
            ENDIF
      ENDIF
      IF back clicked:
            menu_stage = 2
      ELSE IF continue clicked:
            set_theme = themes[theme]
            menu_stage = 4
      ENDIF
ENDIF
IF menu_stage == 1:
      FOR control in controls:
            IF control clicked:
                  toggle control
            ENDIF
      ENDFOR
```

```
            IF back clicked:
                  menu_stage = 0
            ENDIF
      ENDIF
ENDWHILE
```

Here, menu_stage decides which screen is displayed, where 0 is main menu, 1 is settings menu, 2 is game options menu, 3 is game theme menu and 4 is the game, hence the loop is "while menu_stage < 4" because after that, it is no longer the menu – it is the actual game.

We can also implement various other menus, such as a pause menu in the game, or a game over menu to get an overview of stats at the end of a game.

Settings

Difficulty/Modes

Difficulty

The game will have 3 difficulties: easy, normal, and hard. The general gist of each mode is quite simple: easy will have a powerful player, weak and slow enemies and more frequent powerup spawns; normal will have a less powerful player, stronger enemies and less frequent powerup spawns; hard will have a weak player, stronger and faster enemies, and little to no powerup help.

To solve this sub-problem, we can set some configurations for global variables, which will affect the code everywhere else in the program. For example, we can set a global variable player_speed, which we can apply elsewhere in the code such as the instantiation of the Player object etc. so that to change the player's speed, all we need to do is alter the player_speed global variable. The values and variables used in the following pseudocode are just a general guide for the pattern of each difficulty, and will likely not be the actual number of variables and values used in the game for each difficulty.

```
IF difficulty easy:
      player_speed = 20
      player_health = 200
      player_projectile_damage = 20
      enemy_health = 50
      enemy_speed = 2
```

```
      enemy_damage = 5
      powerup_frequency = 5
      ...
ENDIF
IF difficulty normal:
      player_speed = 10
      player_health = 150
      player_projectile_damage = 10
      enemy_health = 100
      enemy_speed = 5
      enemy_damage = 10
      powerup_frequency = 10
      ...
ENDIF
IF difficulty hard:
      player_speed = 10
      player_health = 100
      player_projectile_damage = 5
      enemy_health = 100
      enemy_speed = 10
      enemy_damage = 20
      powerup_frequency = 20
      ...
ENDIF
```

The selection of the difficulty will be done during the menu phase of the program, in the options menu. The difficulty selected will end up affecting these global variables, which will domino effect over the whole program, effectively transforming the player's experience. This makes the difficulty aspect of the code very efficient.

Modes

The different modes offered in the game will be timed, scored and endless. Timed will have a time limit, approximately 2-3 minutes (to be decided), and at the end the score racked up by the player will be recorded; Scored will have a score goal, and when the player reaches that score the time taken to reach that goal will be recorded; and finally endless will have no limits in place, and the final time and score will be recorded when the player dies.

For this, we will modify the entire code to accommodate each limit. The following pseudocode is a general indicator for how this will work:

```
timer = 0
score = 0
IF mode timed:
      WHILE timer < FPS * time_limit:
            ... MAIN GAME ...
            (Score can be accumulated through killing enemies etc.)
            timer += 1
            next frame
      ENDWHILE
      Record score
ELIF mode scored:
      WHILE score < score_goal:
            ... MAIN GAME ...
            Timer += 1
            Next frame
      ENDWHILE
      Record time/FPS
ELIF mode endless:
      WHILE player not dead:
            ... MAIN GAME ...
            timer += 1
            next frame
      ENDWHILE
      Record time/FPS, score
ENDIF
```

The main game will likely be in its own function, to help with reusability for each mode.

Scores

As shown in the game mode pseudocode earlier, the player's score will be initialized to 0 at the beginning of each game and will be recorded at the end of each game. Therefore, scorekeeping will be necessary in between these moments, during the game. The main method of accumulating score will be killing enemies, which we can detect in the moments before an enemy is removed from the enemies list after their health drops to below 0. However, the

inclusion of multiple difficulties would make the high score system quite broken since the player could just get very high scores on easy mode. Therefore, I will make the accumulation of score on easy and normal mode more difficult than on hard mode, to balance it out.

```
score = 0
IF difficulty easy:
      score_earned = 10
ELIF difficulty normal:
      Score_earned = 50
ELIF difficulty hard:
      Score_earned = 150
ENDIF
...MAIN GAME...
FOR enemy IN enemies:
      IF enemy.health <= 0:
            Remove enemy from enemies
            score += score_earned
      ENDIF
ENDFOR
```

To the player, a massive list of past scores will be largely useless: what the player really wants to see is high scores. I.e., the top score for timed mode, the top time for scored mode, and the top time/score for endless mode. To find this, I can simply add a check function at the end of each game, checking if the new time/score is better than the current top time/score, and replacing it if it is. Each high score/time will be displayed in the menu, separated by mode. Notice that for timed mode, the highest score is recorded, for scored mode the shortest time is recorded, and for endless mode the highest score and longest time is recorded.

```
After timed mode:
      Record score
      IF high_score < score:
            Set high_score = score
      ENDIF
After scored mode:
```

```
      Record time
      IF best_time > time:
            Set best_time = time
            Display best_time as time/FPS
      ENDIF
After endless mode:
      Record time
      Record score
      IF longest_time < time:
            Set longest_time = time
            Display longest_time as time/FPS
      ENDIF
      IF high_score < score:
            Set high_score = score
      ENDIF
```

Cosmetics

In my analysis section, I decided to put purchases, themes, and cosmetics as desirable features, i.e., it is likely I will not be implementing those features, but if time permits, I will add them into the game. In that case, it would be necessary to add an extra item to the game: coins.

In one of the existing solutions, survivor.io, their method of currency was coin collectables that the player could pick up during a game, which would be either scattered around the map or dropped by killed enemies. This seems like a good idea, so I will be implementing this in my solution.

The coins in the game will follow a similar algorithm to the powerups, using a list to store all coins currently on the map, printing each of them, and when the player picks them up it is removed, and the player's coin count is incremented. At the start of the game, I will spawn around 20-40 coins around the map for the player to pick up as well. As for the enemies dropping coins, I have decided to allow them to drop randomly with a 1 in 3 chance of them dropping between 1 and 5 coins. Of course, these values are likely to not be the actual values used in implementation but should provide a general idea for how the game mechanics will work.

```
coins = []
FOR i in range(0, randint(20, 40)):
      Add coin(x,y randomized on map) to coins
ENDFOR
FOR enemy IN enemies:
      IF enemy.health <= 0:
            Remove enemy from enemies
            score += score_earned
            IF randint(1,3) == 1:
                  Add randint (1,5) number of coins(enemy x,y) to coins
            ENDIF
      ENDIF
ENDFOR
FOR coin in coins:
      Print coin @ (x,y)
ENDFOR
```

Of course, coins are useless unless the player can pick them up and use them, so I will implement the following pseudocode, allowing picked coins to be accumulated until the end of the game, when it will be added to the player's total coin bank which can be saved using external file handling.

```
coin_bank = open file containing player's coin count
...MAIN GAME...
coins_this_game = 0
FOR coin in coins:
      IF player collides with coin:
            Remove coin from coins
            Coins_this_game += 1
      ENDIF
ENDFOR
...AFTER GAME...
Display coins_this_game
```

```
coin_bank += coins_this_game
```

Now that the player has a way of accumulating coins in their coin bank, I can allow them to spend the coins on various themes and cosmetics in the menu, such as map images and enemy/player sprite images.

Accessibility features

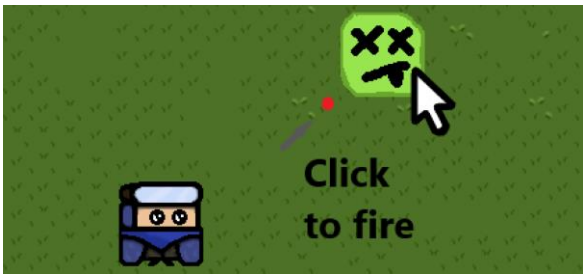
I will also need to add some features into the game that will allow the game to be more accessible by a wide range of audiences, such as disabled people, etc. who would otherwise find the game difficult to navigate or play due to their own setbacks.

One of the accessibility features I will implement is for dyslexia. Dyslexic people sometimes have trouble reading text fluently, so I have found several candidates for the font that I will use in the game that are dyslexia-friendly, i.e. is easier to read.

According to Dyslexia friendly style guide - British Dyslexia Association (bdadyslexia.org.uk), the best fonts for dyslexic people are sans-serif fonts, because they are less crowded and busy-looking. The font size should also be roughly 12-14pt or even bigger if needed. I should also increase the inter-character and inter-word spacing to reduce crowding, but not too much because that can end up reducing readability.

On the topic of readability, we need to also make it easier for players to understand the game mechanics and UI. A messy UI can detract from a pleasurable playing experience, while a clean and clear one can end up being the difference between a fun game and a horrible one. It's also of utmost importance that the player knows how to play.

To accommodate these accessibility features, I will implement a tutorial within the options menu, which will just be several images showcasing the controls and features of the game. I will also ensure that the UI e.g., health bars, powerup icons, etc. stand out from the map, which will require good colour schemes and large enough renders of icons.



Summary of Key Classes/Variables

Here is a summary of the main classes and variables associated with each sub-section that I will be using.

Controls	Map image, controlled by a `camera_scroll` variable which will be global, used by every entity in the environment. Pressing controls updates `camera_scroll`, which allows environment to pan around the player in an inverted fashion.
Player	`Player` class with `health`, `speed`, and `coordinates` attributes. Player is static at the center of the screen.
Enemies	`BasicEnemy` class with `health`, `damage` /`damage_speed`, `speed` and `coordinates` attributes. `ShooterEnemy` class as a child of `BasicEnemy`, with extra firing `projectile_frequency` attributes. Other types of enemies can also be created as more children of `BasicEnemy` . Currently existing enemies will be instantiated and stored in an `enemies` list regardless of enemy class type, so we can iterate over the list and run each of them in turn. All enemies will have their own individual `damage_timer`, so whenever a collision is detected between an enemy and the player, the timer controls the interval for when the player gets dealt `damage`.
Projectiles	`PlayerProjectile` class with `coordinates` and `speed` attributes. Constructor method passes in `cursor_x` and `cursor_y` coordinates as parameters, which is detected when the player clicks on the screen. It then uses a private method `calc_trajectory` to calculate the `velocity` and `angle` at which the projectile moves in the direction of. The `main()` method then uses these vectors to update the coordinates of the projectile. `EnemyProjectile` class as a child of `PlayerProjectile` class, with altered `calc_trajectory` method so is only calculated using the center of the screen as target rather than cursor coordinates passed in. When user clicks, projectiles are instantiated and appended to a `projectiles` list. They are then removed from the list if either collided with an enemy object

	or gone out of bounds. Polymorphism allows both enemy and player projectiles `main()` methods to be run using 1 iterative function.
Powerups	Initially a `powerup_status` dictionary that keeps track of all active powerups, updated when player collides with existing powerup on map. `Powerup` class with powerup `type` and `coordinate` attributes. List of `current_powerups` to store all existing powerups on the map. Every frame there is a small chance for a powerup is spawned on the map, i.e., a powerup object is instantiated, initialized with random co-ords. and appended to `current_powerups`. HEALTH: `health_boost` variable SPEED: `speed_multiplier` variable, `timer` SHIELD: `timer` POISON: `damage_power` variable, `timer` FREEZE: `timer`
Settings	Difficulty variables: `easy`, `normal`, `hard`; `score_earned` from killing enemies varies with difficulty. Mode variables: `timed`, `scored`, `endless`; `score_count`, `timer` used accordingly for each selected mode. `High_score`, `best_time` stored externally with file handling. Cosmetics: `coin_count` variable, `coin_bank` stored externally with file handling.

Testing Approach

To ensure a successful final solution, we must test the program to find out whether the client's specifications have been met. Unlike a command-line interface, where tests would consist of statements to try out, this testing approach will have keyboard and mouse inputs instead since the game is running in a GUI. Therefore, my test plan will consist of not just actions, but also the context surrounding that action, e.g., main menu, game type, etc. My input types will consist of not only normal inputs, but also erroneous and boundary inputs to find errors and bugs, ensuring my program is robust.

Context	Action	Input	Expected Result
Main Menu	Open program	Normal	Game opens onto main menu with correct screen resolution.
	Click blank space	Erroneous	Nothing.
	Click play	Normal	Options menu to select difficulty and game modes.
	Click settings	Normal	Settings menu to change game settings.
	Click quit game	Normal	Program exits.
Options Menu	Click blank space	Erroneous	Nothing.
	Click easy/normal/ hard difficulty	Normal	Toggles between easy, normal and hard buttons, not affecting mode selection.
	Click timed/scored/ endless mode	Normal	Toggles between timed, scored and endless buttons, not affecting difficulty selection.
	Click back	Normal	Main Menu.
	Click continue	Normal	Theme menu(?) to select theme, game will have currently selected options in gameplay.
Theme Menu	Click blank space	Erroneous	Nothing.
	Click right arrow	Normal	Current theme increments to next in list.
	Click left arrow	Normal	Current theme decrements to previous in list.
	Click left arrow when selecting theme 1	Boundary	Current theme loops to last theme.
	Click right arrow when selecting last theme	Boundary	Current theme loops to theme 1.
	Click back	Normal	Options Menu.
	Click continue	Normal	Enter Game, game will have selected difficulty, mode, and theme in gameplay.
Settings Menu	Click blank space	Erroneous	Nothing.
	Click controls	Normal	Toggles WASD to arrows or arrows to WASD.
	Click shop?	Normal	Open shop?
	Click how to play	Normal	How to Play Menu.
	Click back	Normal	Main Menu, settings updated
General Game	Start	Normal	Map, player at center of screen, score count at 0. Correct sprite designs, difficulty (attributes accurate for

			difficulty settings), and mode. Music is playing.
	Press WASD/arrows with the correct control applied in settings.	Normal	Player moves up (W), down (S), left (A) and right (D) on map (or technically map moves in an inverted fashion to controls pressed) as footsteps are played.
	Press WASD/arrows with the wrong control applied in settings.	Erroneous	Nothing.
	Hold keys until player reaches end of map	Boundary	Player stops at border of map, even if controls are still held down.
	Click mouse	Normal	Program calculates correct angle to spawn projectile, originating from player and moving in the direction of the point clicked.
Enemy/Enemy Projectiles	Enemy spawns	Normal	Enemy is placed on map at a random position, moving towards the player.
	Player projectile collides with enemy	Normal	Enemy health is depleted.
	Player projectile collides with enemy with tiny health	Boundary	Enemy health goes below 0, is removed from map and score is added to player's score count. Perhaps drops coin.
	Player collides with enemy	Normal	Player health is depleted at regular intervals while the enemy is touching the player.
	Player collides with enemy projectile	Normal	Player health is depleted.
Powerups	Powerup spawns	Normal	Random powerup icon is displayed on the map at a random position with probability.
	Player collides with powerup (activating it)	Normal	Powerup is removed from map, powerup effect is applied on player for designated amount of time if applicable. Icon is displayed in inventory and timer is initialized.
	Health picked up	Normal	Player health increases.
	Health picked up little under/at max health	Boundary	Player health increases to the max health cap, no further.

	Player collides with enemy/projectile with shield/poison power	Normal	Shield: Neither objects' health depletes. Poison: enemy health depletes at regular intervals, but player health depletes if collides with enemy projectile.
	Player picks up powerup while said powerup still active	Normal	Timer for that powerup refills to full time instead of powerup stacking in power.
	Powerup timer ends	Normal	Icon is removed from inventory, powerup is deactivated in game.
Coins	Coin spawns	Normal	Coin appears at random location on map either naturally or from enemy drop.
	Player collides with coin	Normal	Coin removed from map; player coin count incremented.
End of game	ESC button pressed	Normal	Game paused, open pause menu.
	Click blank space in pause menu	Erroneous	Nothing.
	Quit button pressed in pause menu	Normal	Game ends, score/time is not logged, back to main menu.
	Continue button pressed in pause menu	Normal	Game continues as normal.
	Player health <= 0	Normal	Player dies, game ends, score/time logged, coins added to bank, sent to game over menu.
Timed mode	Start	Normal	Timer initialized to 0.
	Timer reaches chosen max time	Normal	Game ends, final score is recorded.
	Player dies	Normal	Score is not considered for high score.
	Score < high score	Normal	High score is not updated.
	Score > high score	Normal	High score is updated with new score.
	Score = high score	Boundary	High score is not updated.
Scored mode	Start	Normal	Timer initialized to 0.
	Score reaches chosen max score	Normal	Game ends, final time is recorded.
	Player dies	Normal	Time is not considered for best time.
	Time > best time	Normal	Best time is not updated.
	Time < best time	Normal	Best time is updated with new time.
	Time = best time	Boundary	Best time is not updated.
Endless	Start	Normal	Timer initialized to 0.

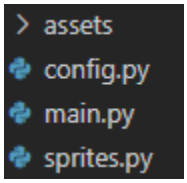
	Player dies	Normal	Game ends, score and time are recorded.
	Score < high score	Normal	High score is not updated.
	Score > high score	Normal	High score is updated with new score.
	Score = high score	Boundary	High score is not updated.
	Time < longest time	Normal	Longest time is not updated.
	Time > longest time	Normal	Longest time is updated with new time.
	Time = longest time	Boundary	Longest time is not updated.
Game Over Menu	Player dies	Normal	Menu opens, score/time and coins collected is displayed, told if surpassed high score/best time.
	Click blank space	Erroneous	Nothing.
	Click Continue	Normal	Back to main menu, high score/time is saved to an external file if applicable.
	Click Restart	Normal	Game begins again, score and time reset to 0.

END OF DESIGN

Implementation

Now, with the analysis and design of my solution sorted, it was time to begin implementing it. The implementation section of my solution will follow an iterative testing approach, where I write some code for a subproblem, test to inform development, fix it if required, and continue writing the next section of code, building up until the entire problem is completed.

Setup



When creating python projects, it is commonly good practice to split your program up into multiple scripts, to make the code more organized and classified with other related code. In this case, I can decompose the program into at least 3 scripts: the main file which we will run the game on, a sprites file which can hold all the classes for the entities, and perhaps a config file that can hold global variables which will be needed many times and can be even used to alter difficulty and mode settings. I will also have an assets folder to store images and other files used.

First things first, I will need to import the pygame module, as well as import sprites.py and config.py, which I will use to put other related code. Then, I will initialize pygame, create the clock for the frames, and create the window and its associated caption.

```
#imports
import pygame, sys
from pygame.locals import *
from sprites import *
from config import *

# Initialise pygame and basic pygame variables
pygame.init()
clock = pygame.time.Clock()

# Create window
WINDOW = pygame.display.set_mode((window_width, window_height))
pygame.display.set_caption("NEA")
```

Of course, I will need to create the window_width and window_height variables, which will be global since these values will need to be used as constants by a lot of processes. They will go in config.py.

```
import pygame, sys
from pygame.locals import *

pygame.init()
FPS = 60
window_width, window_height = 1280, 720
```

You will notice I have put an FPS variable in there as well, which we will use to control the frame rate of the game. I have set this to 60 for now. My width and height follow a 1280:720 aspect ratio for now.

Back in main.py, I will need to implement the main game loop, which is below.

```
while True:
    pygame.display.update()

    WINDOW.fill((255,0,0))

    # Checking for events
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
```

Running main.py now creates a screen of the desired resolution and fills the screen with red, which is what I intended. Now that initial setup has finished, I can move onto the actual game.

Initial Mechanics/Player Sprite

As per the design section, the map's movement will be controlled by a camera_scroll variable, which will be constant throughout the entire game for every entity. I also want the map and the player to spawn in the center of the screen, so I will need a center coordinates variable to keep track of this location. These will also go into config.py, and I will initialize camera_scroll in main.py.

```
 7   # width and height of map
 8   map_width, map_height = 500, 500
 9
10   # Center of screen
11   centre_x = window_width / 2
12   centre_y = window_height / 2
```

For the camera_scroll to control everything, we must factor the values into the coordinates of every entity. We want the coordinates of each entity to be the desired coordinate, minus the camera scroll value, so that everything is relative to the camera_scroll.

```
camera_scroll = [0,0]
```

```
# Background image controlled by camera scroll, placement determined by background size and window size to ensure player spawns in the middle of the map
WINDOW.fill((255,0,0))
pygame.draw.rect(WINDOW, (0,255,0), (-(map_width - window_width)/2 - camera_scroll[0], -(map_height - window_height)/2 - camera_scroll[1], map_width, map_height))
```

Now we must implement a way to control the camera_scroll variable. This is of course going to be by the player, since if the player moves we want the camera scroll to update accordingly. The player will control it through the movement keybinds (in an inverted fashion).

```
    # Checking for any keys pressed
    keys = pygame.key.get_pressed()

    # If any main controls are pressed, alter the camera scroll so the environment can move accordingly (in an inverted fashion)
    if keys[pygame.K_a]:
        camera_scroll[0] -= player_speed

    if keys[pygame.K_d]:
        camera_scroll[0] += player_speed

    if keys[pygame.K_w]:
        camera_scroll[1] -= player_speed

    if keys[pygame.K_s]:
        camera_scroll[1] += player_speed
```

Moving the map will mean nothing without the player, so before we move onto the actual player section for implementation, we will create a Player class to instantiate a player. Of course, this will go into the sprites.py script.

```
import pygame, sys
from pygame.locals import *
import math
from random import randint
from config import *

class Player():

    def __init__(self):
        # Coordinates for centre of screen for position of player
        self.x = centre_x - 15
        self.y = centre_y  - 15
        # Pixel distances set for how far the player can walk (up to the edge of the map)
        self.border_x = (map_width / 2) - 15
        self.border_y = (map_height / 2) - 15

        self.speed = 1

    def main(self, WINDOW):
        pygame.draw.rect(WINDOW, (255,255,255), (self.x, self.y, 30,30))

    def get_speed(self):
        return self.speed

    def get_borders(self, axis):
        if axis == "x":
            return self.border_x
        elif axis == "y":
            return self.border_y
```

Here, I have put in the constructor method the x and y co-ordinates the player will spawn in, which is the center of the screen. The borders for the player can also be created, which are the edges of the map. These border variables show how far from the center of map the player can venture.

Now that the class has been created, we can go back to main.py and instantiate the player.

```
player = Player()

while True:
    pygame.display.update()
```

Notice how I've put it before the main game loop, so we don't end up instantiating the same player repeatedly every frame. What we will put in the main game loop however is the running of the main method for the player, so we can draw the player onto the screen.

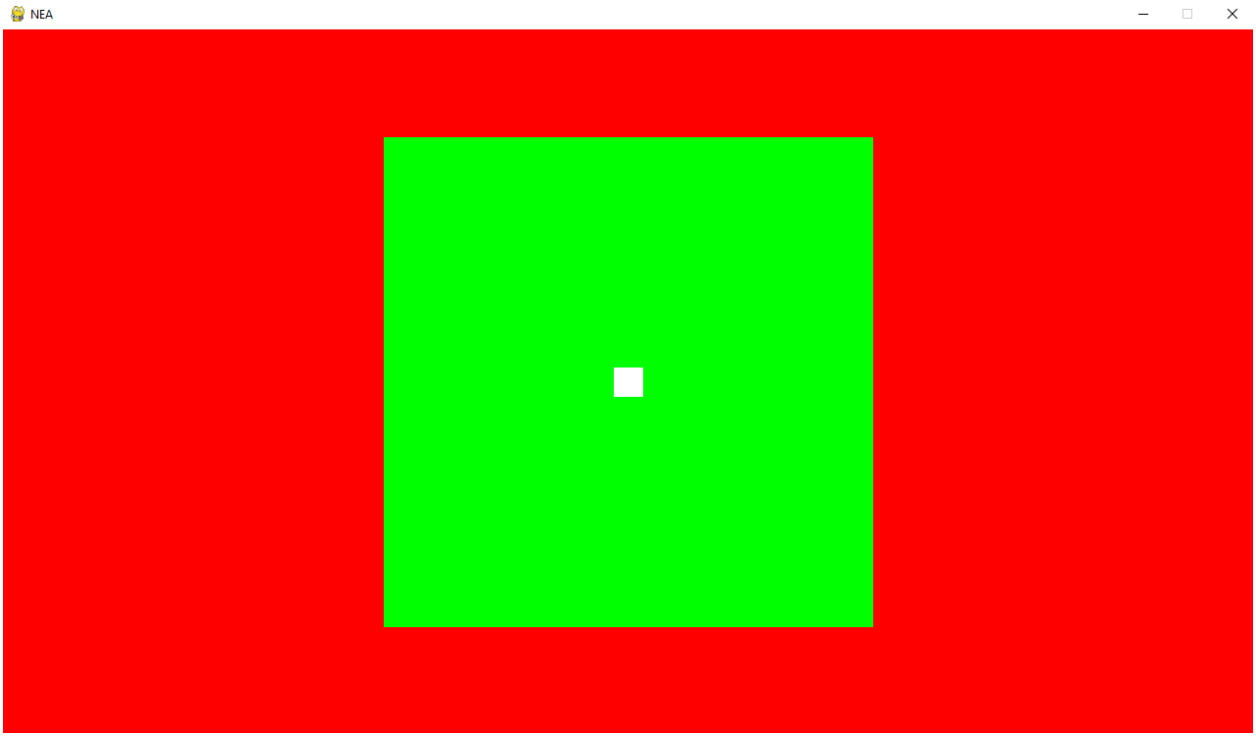
```
53      # Run the main methods for all the objects in the game
54      player.main(WINDOW)
55
56      clock.tick(FPS)
```

Another thing we need to update is our code for updating the camera_scroll: we now need to replace player_speed with player.get_speed(), and a check to make sure the player is not at the border before moving it, keeping the player within the border of the map.

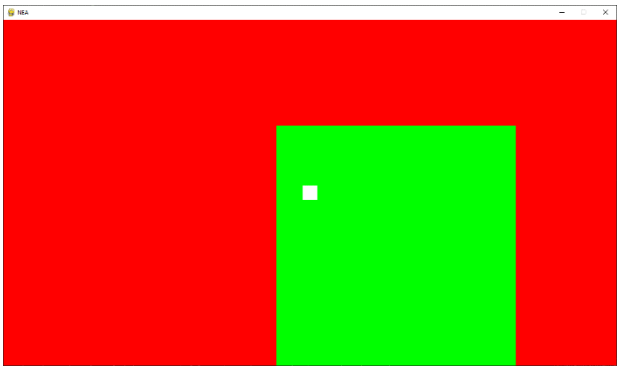
```
40      # Checking for any keys pressed
41      keys = pygame.key.get_pressed()
42
43      # If any main controls are pressed, alter the camera scroll so the environment can move accordingly (in an inverted fashion)
44      if keys[pygame.K_a]:
45          if camera_scroll[0] > -player.get_borders("x"):
46              camera_scroll[0] -= player.get_speed()
47
48      if keys[pygame.K_d]:
49          if camera_scroll[0] < player.get_borders("x"):
50              camera_scroll[0] += player.get_speed()
51
52      if keys[pygame.K_w]:
53          if camera_scroll[1] > -player.get_borders("y"):
54              camera_scroll[1] -= player.get_speed()
55
56      if keys[pygame.K_s]:
57          if camera_scroll[1] < player.get_borders("y"):
58              camera_scroll[1] += player.get_speed()
59
60      # Run the main methods for all the objects in the game
61      player.main(WINDOW)
```

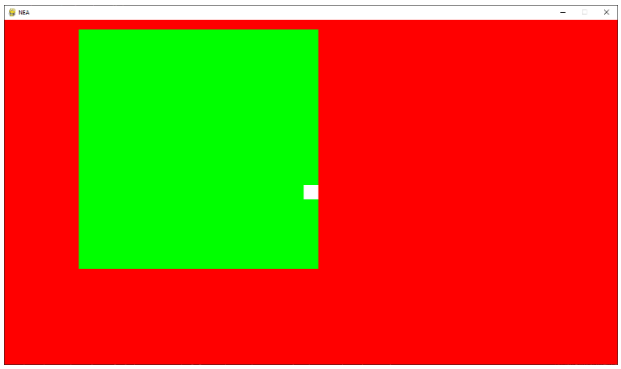
Now that the general map mechanics have been set up, we can run the code and see whether it runs properly.

Typing “python main.py” into the console, the following screen pops up:



As expected, both the map and the player spawn at the center of the screen. Pressing WASD also causes the player to “move” or causing the map to move in the opposite direction, until the player reaches the edge of the map, preventing them from moving off the map.





The game mechanics are now in place; however, it still looks very simple and not like a game at all. We can fix this by replacing the rects with actual images. For now, I have temporarily created

some player sprite images, which I want to be able to have animated to make it look like it is walking. I have also gotten a temporary background, just so I have a placeholder image render for any future background updates. These images are all in the assets folder.

I first imported the images in config.py, then put the animation frame images into a list that I can iterate over every frame.

```
8   bg = pygame.transform.scale(pygame.image.load("assets/bg.png"), (map_width, map_height))
9
10  player_walk_images = [pygame.transform.scale(pygame.image.load("assets/player_sprite.png"), (int(window_width/18), int(window_width/18))),
11                        pygame.transform.scale(pygame.image.load("assets/player_sprite2.png"), (int(window_width/18), int(window_width/18)))]
```

Finally, I replaced the background blitting code and the player main method with these images (the default player sprite is just the first frame of player_walk_images).

Running the code, this is what shows up:

Already much better looking. However, when I move the player, it doesn't look very realistic because he looks the same every frame and looks to his right constantly even when moving left.

To fix this, I can add an orientation attribute to the player class, as well as an iterator to update the animation frame.

To control the animation frame rate, I will also need some extra attributes, such as an animation frame counter and an FPS tracker to keep track of when to update the animation.

Here is the code implemented within the Player class:

```
class Player():

    def __init__(self):
        # Coordinates for centre of screen for position of player
        self.x = centre_x - (player_walk_images[0].get_width() / 2)
        self.y = centre_y  - (player_walk_images[0].get_height() / 2)

        # Pixel distances set for how far the player can walk (up to the edge of the map)
        self.border_x = (map_width / 2) - (player_walk_images[0].get_width() / 2)
        self.border_y = (map_height / 2) - (player_walk_images[0].get_height() / 2)

        # Animation variables: animation_frames decides how fast the animation is
        # i.e. how many frames of game should pass for one frame of animation
        self.animation_count = 0
        self.animation_frames = 8

        # Get state of the player for display purposes
        self.orientation = "right"
        self.moving = False

        self.speed = 10

    def main(self, WINDOW):

        # Loops back round after animation count has reached the end to prevent index error
        if self.animation_count + 1 >= 2*self.animation_frames:
            self.animation_count = 0
        self.animation_count += 1

        # Checks orientation of player and flips the sprite horizontally accordingly
        if self.orientation == "right":
            # If the player is moving the sprite, play the animation at the speed decided
            if self.moving == True:
                WINDOW.blit(player_walk_images[self.animation_count//self.animation_frames], (self.x, self.y))
            # Otherwise, just play the first frame to make it look like they are still
            else:
                WINDOW.blit(player_walk_images[0], (self.x, self.y))
        # Same for left
        elif self.orientation == "left":
            if self.moving == True:
                WINDOW.blit(pygame.transform.flip(player_walk_images[self.animation_count//self.animation_frames], True, False), (self.x, self.y))
            else:
                WINDOW.blit(pygame.transform.flip(player_walk_images[0], True, False), (self.x, self.y))
```

What this code does is firstly check the orientation of the player, and if they are facing left flip the image so it looks left. Then, it checks if the player is moving. If they are, use the animation_count and animation_frames attributes to keep track of when to update the sprite frame image, and every frame increment the animation count and loop back to 0 if the animation count has gone over the number of frames. This means that every animation_frames number of frames, update the frame. If the player isn't moving, just blit the first frame and nothing else.

This code means nothing if we don't utilize these attributes, so as well as add code to main.py I need to add the relevant get and set methods to the Player class.

```
def set_moving(self, new_moving):
    self.moving = new_moving

def set_orientation(self, new_orientation):
    self.orientation = new_orientation
```

And in main.py:

```
# Checking for any keys pressed
keys = pygame.key.get_pressed()

# If any main controls are pressed, alter the camera scroll so the environment can move accordingly (in an inverted fashion)
if keys[pygame.K_a]:
    if camera_scroll[0] > -player.border_x:
        camera_scroll[0] -= player.get_speed()
        # Also decide the direction the sprite surface is facing
        player.set_orientation("left")
    player.set_moving(True)

if keys[pygame.K_d]:
    if camera_scroll[0] < player.border_x:
        camera_scroll[0] += player.get_speed()
        player.set_orientation("right")
    player.set_moving(True)

if keys[pygame.K_w]:
    if camera_scroll[1] > -player.border_y:
        camera_scroll[1] -= player.get_speed()
    player.set_moving(True)

if keys[pygame.K_s]:
    if camera_scroll[1] < player.border_y:
        camera_scroll[1] += player.get_speed()
    player.set_moving(True)

if not (keys[pygame.K_a] or keys[pygame.K_d] or keys[pygame.K_w] or keys[pygame.K_s]):
    player.set_moving(False)

# Run the main methods for all the objects in the game
player.main(WINDOW)
```

Now, the player orientation is controlled by the keys being pressed, and if no keys are being pressed then the moving attribute is set to false.

As you can see, moving left causes the player to flip horizontally and is now facing left. Also, you can't see through screenshots, but the player is now successfully "walking" since each frame is just the player's legs in different positions and stringing them together creates the illusion of footsteps.

Projectiles

Now that we have a good base for the player's mechanics, we can start to implement the actual shooting aspect of the game. To create the projectiles that will shoot from the player, I will create a projectile class as per the design section, so that each projectile will have its own object.

```
class PlayerProjectile():

    def __init__(self, cursor_x, cursor_y, speed):

        self.speed = speed
        self.despawn_range = 500

        self.x = centre_x
        self.y = centre_y
        self.cursor_x = cursor_x
        self.cursor_y = cursor_y
        self.dxdy = self.calc_trajectory()

    # Private method to calcuate the trajectory of the projectile based on cursor coordinates
    def calc_trajectory(self):
        self.angle = math.atan2(self.y - self.cursor_y, self.x - self.cursor_x)
        self.dx = math.cos(self.angle) * self.speed
        self.dy = math.sin(self.angle) * self.speed
        return [self.dx, self.dy]

    def main(self, WINDOW):

        # Update the coordinates according to the x and y velocities
        self.x -= int(self.dxdy[0])
        self.y -= int(self.dxdy[1])

        pygame.draw.circle(WINDOW, (255,0,0), (self.x, self.y), 5)

    def get_despawn_range(self):
        return self.despawn_range
```

The maths for this code is explained in pseudocode in the design section.

In main.py outside the main game loop, I created a list to store all currently existing projectiles in the game.

```
17    # Instantiate basic objects needed for game and create lists for arrays of objects
18    player = Player()
19    player_projectiles = []
```

Inside the game loop, meanwhile, we need to add a check to find when the player clicks the screen and record the coordinates that click registered at. We can then append a newly instantiated projectile object to the list and pass in these cursor coordinates.

```
    # Checking for events
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        # Checking for cursor coordinates
        cursor_x, cursor_y = pygame.mouse.get_pos()
        # Checking if the player clicked mouse
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                # If so, append a projectile object to the list
                player_projectiles.append(PlayerProjectile(cursor_x, cursor_y, 30))
```

Now that we have objects, we can run each of their main methods in turn, and despawn them once they leave the range declared as an attribute in the class. This will reduce lag by ensuring not too many projectiles have to be run at the same time.

```
for projectile in player_projectiles:
    projectile.main(WINDOW)

for projectile in player_projectiles:
    # Checks if the projectiles have gone out of range and delete them if so, to save memory
    if (projectile.x < -projectile.get_despawn_range()) or (projectile.y < -projectile.get_despawn_range()) or (projectile.x > window_width + projectile.get_despawn_range()) or
    (projectile.y > window_height + projectile.get_despawn_range()):
        player_projectiles.remove(projectile)
```

Running this code, we find that clicking anywhere on the screen causes a projectile to spawn from the player's position and move in the direction of the click!

However, there was a problem with these projectiles. With a speed of 30, it wasn't much of a deal, but pulling the speed down, it was very noticeable there was an error with the trajectory of the projectile. The error was that after firing a projectile, moving the player causes the projectile to stay in a position relative to the player instead of relative to the map. For example, moving the player up causes the map to move down, but the projectile does not move down with the map. This was a problem, especially for later developments where I may want a projectile that moves at a slower speed, in which case this bug would become very noticeable.

To fix this, I simply added an extra criterion to the projectile's coordinates, making it move relative to the player's movement. This is shown below. First, I passed into each projectile's main method the player's movement state, i.e., which way they were moving and the speed they were moving at.

```
    for projectile in player_projectiles:
        projectile.main(WINDOW, keys, player.get_speed())
```

Then I used this information to move the projectile by that amount in the opposite direction, which creates the illusion that as the player moves away from the projectile, the projectile moves away in the opposite direction, if that makes sense.

```
def main(self, WINDOW, projectilekey, player_speed, is_moving_x, is_moving_y):

    # Update the coordinates according to the x and y velocities
    self.x -= int(self.dxdy[0])
    self.y -= int(self.dxdy[1])

    # If the player is moving, then update the coordinates of the projectile accordingly
    # to move it relative to the map in the corresponding axis
    if is_moving_x:
        if projectilekey[pygame.K_a]:
            self.x += player_speed
        if projectilekey[pygame.K_d]:
            self.x -= player_speed
    if is_moving_y:
        if projectilekey[pygame.K_w]:
            self.y += player_speed
        if projectilekey[pygame.K_s]:
            self.y -= player_speed

    pygame.draw.circle(WINDOW, (255,0,0), (self.x, self.y), 5)
```

Now, moving the player while a projectile is on the screen causes the projectile to move with the map instead of the player. We fixed it! Or did we?

In fact, we didn't fully fix it. Because while the pressing of the keys did dictate where the map moves when the player is within the map, it does not necessarily mean the player is moving when they are at the border. This meant that if the player was at the border and was trying to move away from the map, the projectiles would swiftly move in the undesired direction at the speed of the player. While this looked quite goofy, it wasn't what I wanted so I had to fix this.

To fix it, I simply added another way of checking if the player was moving: giving the player class moving_x and moving_y Boolean attributes, as well as corresponding get and set methods.

```
23          # Get state of the player for display purpose
24          self.orientation = "right"
25          self.moving_x = False
26          self.moving_y = False
```

With this, I could then go into main.py and add updates to the player moving code: if the map isn't moving, then the player by definition is not moving either. This also applies if the player is at the border.

```
# If any main controls are pressed, alter the camera scroll so the environment can move accordingly (in an inverted fashion)
if keys[pygame.K_a]:
    if camera_scroll[0] > -player.get_borders()["x"]:
        camera_scroll[0] -= player.get_speed()
        # Also decide the direction the sprite surface is facing
        player.set_orientation("left")
        player.set_moving_x(True)
    else:
        player.set_moving_x(False)

if keys[pygame.K_d]:
    if camera_scroll[0] < player.get_borders()["x"]:
        camera_scroll[0] += player.get_speed()
        player.set_orientation("right")
        player.set_moving_x(True)
    else:
        player.set_moving_x(False)

if keys[pygame.K_w]:
    if camera_scroll[1] > -player.get_borders()["y"]:
        camera_scroll[1] -= player.get_speed()
        player.set_moving_y(True)
    else:
        player.set_moving_y(False)

if keys[pygame.K_s]:
    if camera_scroll[1] < player.get_borders()["y"]:
        camera_scroll[1] += player.get_speed()
        player.set_moving_y(True)
    else:
        player.set_moving_y(False)

if not (keys[pygame.K_a] or keys[pygame.K_d] or keys[pygame.K_w] or keys[pygame.K_s]):
    player.set_moving_x(False)
    player.set_moving_y(False)
```

Now, we are sure that if the player is moving, i.e., the map is moving, then the projectiles will move along with it. If not, then the projectiles should just continue their normal path.

Enemies

Now that the player's general mechanics are pretty much complete, we can move onto the other entities in the game, namely the enemies.

To start with, we will need to have a sprite image for basic enemies, which we can name enemy_sprite and pop into config.py.

```
enemy_sprite = pygame.image.load("assets/enemy_sprite.png")
```

We can then go into sprites.py and create an enemy class, using the code designed in pseudocode in the design section earlier.

```
50  class Enemy():
51
52      def __init__(self):
53          # Basic attributes, such as speed, borders, x and y which will initially be random coordinates
54          self.speed = 1
55
56          self.border_x = (map_width / 2) - (enemy_sprite.get_width() / 2)
57          self.border_y = (map_height / 2) - (enemy_sprite.get_height() / 2)
58
59          self.x = randint(centre_x - self.border_x, centre_x + self.border_x)
60          self.y = randint(centre_y - self.border_y, centre_y + self.border_y)
61
62      def main(self, WINDOW):
63
64          # The enemy will always be moving towards the center of the screen, where the player is
65          if self.x <= centre_x + camera_scroll[0] and self.x < centre_x + self.border_x:
66              self.x += self.speed
67          elif self.x > centre_x + camera_scroll[0] and self.x > centre_x - self.border_x:
68              self.x -= self.speed
69
70          if self.y <= centre_y + camera_scroll[1] and self.y <= centre_y + self.border_y:
71              self.y += self.speed
72          elif self.y > centre_y + camera_scroll[1] and self.y > centre_y - self.border_y:
73              self.y -= self.speed
74
75          WINDOW.blit(enemy_sprite, (self.x - camera_scroll[0] - (enemy_sprite.get_width() / 2), self.y - camera_scroll[1] - (enemy_sprite.get_height() / 2)))
```

With the enemy class created, we can create a list of enemies to roam around the map. This can be initialized outside the game loop with the instantiation of the player and projectile list. For now, I will put 8 enemies in there.

```
17  # Instantiate basic objects needed for game and create lists for arrays of objects
18  player = Player()
19  enemies = [Enemy(), Enemy(), Enemy(), Enemy(), Enemy(), Enemy(), Enemy(), Enemy()]
20  player_projectiles = []
```

Then in the actual game loop, I will run the main method for each enemy in turn.

```
for enemy in enemies:
    enemy.main(WINDOW)
```

Running the game now shows 8 enemies in the map, moving towards the player.

Playing this a couple of times brought to light quite a significant problem. After a while of the player moving around the map, the enemies tend to pile on top of each other as they move. This is obviously a problem since it detracts from the game experience when all the player must do is fire at the same position over and over again.

To fix this problem, we must think about how we want the enemies to move. Of course, we want them to follow the player, but what we could also implement is a wandering feature where the enemies sometimes strafe around the player instead of moving directly towards them. This would reduce the tendency for them to move along the same path which causes them to overlap.

To do this, I will need to implement a way for the enemy to move towards a random point in an area surrounding the player, rather than directly on the player. This will be achieved by creating offset attributes, randomly assigning new offset coordinates between regular intervals. This will create the random strafing effect we are going for, where enemies wander around the player but still tending towards it to keep the challenge.

First, I will go into the enemy class's constructor method to create these attributes.

```
class Enemy():

    def __init__(self):
        self.speed = 1

        self.border_x = (map_width / 2) - (enemy_sprite.get_width() / 2)
        self.border_y = (map_height / 2) - (enemy_sprite.get_height() / 2)

        self.x = randint(centre_x - self.border_x, centre_x + self.border_x)
        self.y = randint(centre_y - self.border_y, centre_y + self.border_y)

        self.reset_offset = 0
        self.offset_x = randint(-150, 150)
        self.offset_y = randint(-150, 150)
```

The reset_offset attribute is to keep track of the timing of each offset coordinate reset. The coordinate attributes are of course set to a random number within a range, here being up to 150 pixels away from the player.

```
def main(self, WINDOW):

    if self.reset_offset == 0:
        self.offset_x = randint(-150, 150)
        self.offset_y = randint(-150, 150)
        self.reset_offset = randint(int(FPS*reset_offset_timer), int(FPS*reset_offset_timer + FPS))
    else:
        self.reset_offset -= 1

    if self.x < centre_x + self.offset_x + camera_scroll[0]:
        self.x += self.speed
    elif self.x > centre_x + self.offset_x + camera_scroll[0]:
        self.x -= self.speed

    if self.y < centre_y + self.offset_y + camera_scroll[1]:
        self.y += self.speed
    elif self.y > centre_y + self.offset_y + camera_scroll[1]:
        self.y -= self.speed
```

Then in the main function, I will have the interval between each offset reset be some random time period, and each time when the timer reaches 0 I will reroll the offset coordinates. Finally, all I need to do is add these offset coordinates to the directional if statements so that the enemy moves towards these assigned offset coordinates.

Running the game now is a lot better, since the enemies do not dogpile on top of each other anymore, and the player can enjoy picking off each enemy dispersed across the map around them.

Now we will move on to the next crucial aspect of the game: killing enemies. We can kill enemies by colliding player projectiles with the enemies, deleting them from the enemy list.

To detect these collisions, I am going to make a hitbox method for every entity class, which simply returns a rect holding information on the position and dimensions of the sprite.

```
def hitbox(self):
    return pygame.Rect(self.x, self.y, player_walk_images[0].get_width() - 15, player_walk_images[1].get_height() - 15)
```

```
def hitbox(self):

    return pygame.Rect(self.x - camera_scroll[0] - (enemy_sprite.get_width() / 2), self.y - camera_scroll[1] -
    (enemy_sprite.get_height() / 2), enemy_sprite.get_width(), enemy_sprite.get_height())
```

With these hitboxes created, we can call them to detect collisions that happen between entities, such as projectiles on enemies:

```
for projectile in player_projectiles:
    projectile.main(WINDOW, keys)

    for enemy in enemies:
        if enemy.hitbox().collidepoint(projectile.x, projectile.y):
            enemies.remove(enemy)
            player_projectiles.remove(projectile)
            break
```

Another piece of code we need to add is what happens if we touch the enemy, which is that we die. Right now, we don't have a game over screen, so for now we will simply quit the game, as shown below.

Testing this code, the game successfully quits when the player collides with an enemy, and enemies disappear when hit with a projectile.

```
for enemy in enemies:
    enemy.main(WINDOW)

    if enemy.hitbox().colliderect(player.hitbox()):
        pygame.quit()
        sys.exit()
```

There is a 2nd main enemy we need to create, the shooter enemy. These 2 enemies will form the foundation for any further enemies we make, since I want 2 basic variations: melee and long-range enemies. Creating this will be very simple: All I need to do is create a shooter enemy

class, which is a child of basic enemy, because they share a lot of attributes and methods such as their movement behaviours, speed, health, damage, and damage speed. Inheriting from a parent class will also allow me to utilize polymorphism and put shooter enemies within the enemies list as well and iterate over all of them.

```
class ShooterEnemy(BasicEnemy):

    def __init__(self, speed, health, damage, damage_speed, projectile_freq, sprite_img):
        super().__init__(speed, health, damage, damage_speed, sprite_img)

        self.projectile_freq = projectile_freq
        self.projectile_cooldown = projectile_freq

    def get_projectile_freq(self):
        return self.projectile_freq

    def get_projectile_cooldown(self):
        return self.projectile_cooldown
    def set_projectile_cooldown(self, new_projectile_cooldown):
        self.projectile_cooldown = new_projectile_cooldown

    def identify(self):
        return "ShooterEnemy"
```

You will notice that I have added an additional method, called identify. This will allow me to differentiate between normal enemies and shooter enemies when iterating over a mix of all types of enemies in the list. I have put this method into the parent class BasicEnemy as well, but outputs “BasicEnemy” instead. This is an example of polymorphism in action.

Another class I need to create is an enemy projectile class. The reason for this is because player projectiles calculate trajectories with the cursor coordinates in mind, while enemy projectiles will be firing towards the center of the screen, i.e. the player. However, player projectiles and enemy projectiles are otherwise largely the same in terms of their methods and attributes, so I can create a child class of the player projectile class called EnemyProjectile.

```
class EnemyProjectile(PlayerProjectile):

    def __init__(self, speed, damage, origin_x, origin_y, sprite_img):
        super().__init__(0, 0, speed, damage, sprite_img)
        self.despawn_range = 500

        self.x = origin_x
        self.y = origin_y

        self.dxdy = self.calc_trajectory()

    def calc_trajectory(self):
        self.angle = math.atan2(self.y - centre_y, self.x - centre_x)
        self.dx = math.cos(self.angle) * self.speed
        self.dy = math.sin(self.angle) * self.speed
        return [self.dx, self.dy]

    def identify(self):
        return "EnemyProjectile"
```

Now that the classes for the shooter enemy have been made, we can go into main.py and create the code for how the enemy will behave and shoot projectiles.

First I will make it so enemies spawn manually for now, for debugging purposes. You can see that I have created several global variables in config.py such as basic_enemy_speed, health, damage, damage_speed, etc. to pass into each object. This will be useful later when setting difficulties. I also added another sprite called shooter_sprite, which you will see in the testing screenshots later.

```
if keys[pygame.K_m]:
    enemies.append(BasicEnemy(speed=basic_enemy_speed, health=basic_enemy_health, damage=basic_enemy_damage, damage_speed=enemy_damage_speed, sprite_img=enemy_sprite))
if keys[pygame.K_n]:
    enemies.append(ShooterEnemy(speed=basic_enemy_speed, health=basic_enemy_health, damage=basic_enemy_damage, damage_speed=enemy_damage_speed, projectile_freq=2,
    sprite_img=shooter_sprite))
```

I have the luxury of appending both enemy types onto the same list, because one is a child of another, and I am utilizing polymorphism to run all their methods. You can see this in action in the code below, where I am running the enemy main, hitbox and identify methods on every enemy.

```
for enemy in enemies:
    enemy.main(WINDOW)

    # If any enemy collides with the player's hitbox, the player dies
    if enemy.hitbox(WINDOW).colliderect(player.hitbox(WINDOW)):
        pygame.quit()
        sys.exit()
    # If a shooter enemy's shooting cooldown runs out, fire another shot at the player
    if enemy.identify() == "ShooterEnemy":
        if enemy.get_projectile_cooldown() == 0 and enemy.get_frozen_status() == False:
            projectiles.append(EnemyProjectile(speed=shooter_enemy_proj_speed, damage=shooter_enemy_proj_damage, origin_x=enemy.get_xy()[0], origin_y=enemy.get_xy()[1], sprite_img=(0,0,0)))
        enemy.set_projectile_cooldown(enemy.get_projectile_cooldown() + 1)
        if enemy.get_projectile_cooldown() >= FPS * enemy.get_projectile_freq():
            enemy.set_projectile_cooldown(0)
```

Running the code to test these enemies, they looked and behaved exactly as desired:

However, there was a strange issue going on. Instead of the shooter enemies firing projectiles, they were seemingly vanishing seconds after spawning on the screen! After some further testing, I realized a significant issue: thanks to the enemy projectiles being treated the same way as player projectiles, the projectiles spawning on the enemies were immediately killing them because they were colliding with them.

To fix this bug, I added an if statement check into the projectile killing section of code, using the identify methods of the projectiles to check whether a projectile colliding with an enemy was a player projectile or an enemy projectile. I also added the enemy projectile killing the enemy function as well.

```
for projectile in projectiles:
    projectile.main(WINDOW, keys, player_speed)

    # If the projectile is a player projectile, then deal damage to enemies
    if projectile.identify() != "EnemyProjectile":
        for enemy in enemies:
            if enemy.hitbox().collidepoint(projectile.get_xy("x"), projectile.get_xy("y")):
                enemies.remove(enemy)
                projectiles.remove(projectile)
                break
    # Otherwise, it is an enemy projectile so deal damage to the player
    else:
        if player.hitbox(WINDOW).collidepoint(projectile.get_xy()["x"], projectile.get_xy()["y"]):
            projectiles.remove(projectile)
            projectile_removed = True
            pygame.quit()
            sys.exit()
```

Running the code now, the enemy projectiles no longer harm the enemies, and they can fire projectiles at the player now! The projectiles also killed the player upon contact, which was exactly what I wanted.

For now, the game quitting instantly when the player is first hit is fine for debugging purposes, but when George plays the game, we want there to still be a slightly lengthy aspect to this game. For this reason, I will give enemies and the player a health bar, which will decrease with hits.

First, I need to give all alive entities health attributes, which will be initialized in the constructor methods of said classes. I will then have the main method blit the entity's current health above their sprite as well as blitting the sprite itself.

```
self.health = health
```

```
def get_health(self):
    return self.health
def set_health(self, new_health):
    self.health = new_health
```

```
WINDOW.blit(smallfont.render(str(self.health), True, (0,0,0)), (self.x - camera_scroll[0] - (self.sprite_img.get_width() / 2) + 20, self.y - camera_scroll[1] - (self.sprite_img.
get_height() / 2) - 30))
```

Running the code shows all the entities with their health above their heads, which I initialized to 200 for the player, and 50 for the enemies.

I then altered the killing functions in main.py to decrement health rather than quit the game (until it reaches 0):

```
for projectile in projectiles:

    projectile.main(WINDOW, keys, player.get_speed(), player.get_moving()["x"], player.get_moving()["y"])

    projectile_removed = False
    # If any projectile hits an enemy's hitbox, remove both that enemy and that projectile from their corresponding lists
    if projectile.identify() != "EnemyProjectile":
        for enemy in enemies:
            if enemy.hitbox(WINDOW).collidepoint(projectile.get_xy()["x"], projectile.get_xy()["y"]):
                enemy.set_health(enemy.get_health() - projectile.get_damage())
                projectiles.remove(projectile)
                projectile_removed = True
                break
    # But if it hits a player hitbox, deal damage to the player
    else:
        if player.hitbox(WINDOW).collidepoint(projectile.get_xy()["x"], projectile.get_xy()["y"]):
            player.set_health(player.get_health() - projectile.get_damage())
            projectiles.remove(projectile)
            projectile_removed = True

    # Checks if the projectiles have gone out of range and delete them if so, to save memory
    if ((projectile.get_xy()["x"] < -projectile.get_despawn_range()) or
        (projectile.get_xy()["y"] < -projectile.get_despawn_range()) or
        (projectile.get_xy()["x"] > window_width + projectile.get_despawn_range()) or
        (projectile.get_xy()["y"] > window_height + projectile.get_despawn_range())) and projectile_removed == False:
        projectiles.remove(projectile)
```

```
if enemy.get_health() <= 0:
    enemies.remove(enemy)
```

```
# If the health goes below 0, game over
if player.get_health() <= 0:
    print("YOU DIED")
    pygame.quit()
    sys.exit()
```

Firing projectiles at the enemy now successfully decreased their health, and if their health fell to 0 then the enemies were removed.

However, there was yet another bug. When the player collided with the enemies, the speed at which the enemies dealt damage to the player was at the same speed as the FPS of the game. This meant the player still died almost instantly after touching enemies.

To fix this, I went back into the enemy class and added several delay attributes. This would work by using a timer to detect when it is time to deal damage and have intervals in between, so that the effect generated would be the player's health decrementing every second or so.

```
self.damage_speed = damage_speed
self.damage_delay_timer = 0
```

```
for enemy in enemies:
    enemy.main(WINDOW)

    # If any enemy collides with the player's hitbox, the player takes damage at a delayed interval
    if enemy.hitbox(WINDOW).colliderect(player.hitbox(WINDOW)):

        if enemy.get_damage_delay_timer() == 0:
            player.set_health(player.get_health() - enemy.get_damage())

        enemy.set_damage_delay_timer(enemy.get_damage_delay_timer() + 1)

        if enemy.get_damage_delay_timer() >= FPS * enemy.get_damage_speed():
            enemy.set_damage_delay_timer(0)
    else:
        enemy.set_damage_delay_timer(0)

    # If a shooter enemy's shooting cooldown runs out, fire another shot at the player
    if enemy.identify() == "ShooterEnemy":
        if enemy.get_projectile_cooldown() == 0:
            projectiles.append(EnemyProjectile(speed=shooter_enemy_proj_speed, damage=shooter_enemy_proj_damage, origin_x=enemy.get_xy()[0], origin_y=enemy.get_xy()[1], sprite_img=
            (0,0,0)))
        enemy.set_projectile_cooldown(enemy.get_projectile_cooldown() + 1)
        if enemy.get_projectile_cooldown() >= FPS * enemy.get_projectile_freq():
            enemy.set_projectile_cooldown(0)

    if enemy.get_health() <= 0:
        enemies.remove(enemy)
```

As you can see, I have implemented a timer as well as given enemies a damage speed as an attribute to control how fast enemies can deal damage. Again, this will be useful when deciding difficulties.

Running the code now, the enemies successfully deal damage the way I want to!

The foundation for the enemies has been successfully created. From these 2 base classes, I can easily create variants of these enemies that would create a more complex and immersive game experience, where the player must fight a whole range of different enemies. For now, however, I will focus on getting the rest of the game up to speed before adding these variants.

Powerups

The first thing to do for powerups is to create its classes. I am going to create 2 different classes: a Powerups class for powerup collectables on the map, and an ActivePowerups class that will provide methods for activating powerups for the player.

```
class Powerup():

    def __init__(self, powerup_id):
        # Powerup ID determines which powerup this powerup object is and picks out the relevant icon for it
        self.powerup_id = powerup_id
        self.sprite_icon = powerup_icons[powerup_id]
        self.sprite_icon = pygame.transform.scale(self.sprite_icon, (powerup_size, powerup_size))

        self.border_x = int((map_width / 2)) - int((powerup_size / 2))
        self.border_y = int((map_height / 2)) - int((powerup_size / 2))
        self.x = random.randint(centre_x - self.border_x, centre_x + self.border_x)
        self.y = random.randint(centre_y - self.border_y, centre_y + self.border_y)

        # Timer to control when to despawn
        self.timer = 0

    def main(self, WINDOW):
        # Allow it to flash right before it disappears, like a fading effect almost
        if (self.timer > FPS*(powerup_lifetime - 1) and self.timer < int(FPS*(powerup_lifetime - 0.5)) or
            self.timer > FPS*(powerup_lifetime - 2) and self.timer < int(FPS*(powerup_lifetime - 1.5)) or
            self.timer > FPS*(powerup_lifetime - 3) and self.timer < int(FPS*(powerup_lifetime - 2.5))):
            return

        WINDOW.blit(self.sprite_icon, (self.x - camera_scroll[0] - (powerup_size/2), self.y - camera_scroll[1] - (powerup_size/2)))

    def hitbox(self, WINDOW):
        if HITBOXES:
            pygame.draw.rect(WINDOW, (255,0,0), (self.x - camera_scroll[0] - (powerup_size/2),
                                                 self.y - camera_scroll[1] - (powerup_size/2),
                                                 powerup_size, powerup_size), 1)

        return pygame.Rect(self.x - camera_scroll[0] - (powerup_size/2), self.y - camera_scroll[1] - (powerup_size/2), powerup_size, powerup_size)

    def identify(self):
        return self.powerup_id

    def get_timer(self):
        return self.timer
    def set_timer(self, new_time):
        self.timer = new_time

    # Health is a special powerup since it does not get applied over a period of time, it is just straight away used up when collected, so this method is in this class instead
    def health(self, player):
        player.health += powerup_health_boost
        if player.health > player_max_health:
            player.health = player_max_health
```

Here, you can see that I have created the Powerups class very similarly to other classes such as enemies. That is because I want powerups to spawn randomly across the map and have a hitbox to detect when the player collides with it, as well as an identify method so I know which powerup each object is. Each powerup will also have a timer, to keep track of how long the powerup has existed in the game and to get rid of it accordingly once it has reached the end of its lifetime, signified by a disappearing effect, where it starts flashing in and out of existence as it gets closer to deletion.

Meanwhile in config.py, I imported the icons for each powerup (health, speed, shield, poison, freeze) using a dictionary, as well as icon sizes.

```
# Icons for powerups
powerup_icons = {"health": pygame.image.load("assets/powerup_health.png"),
                 "speed": pygame.image.load("assets/powerup_speed.png"),
                 "shield": pygame.image.load("assets/powerup_shield.png"),
                 "poison": pygame.image.load("assets/powerup_poison.png"),
                 "freeze": pygame.image.load("assets/powerup_freeze.png")}
powerup_size = int(window_width/25)
active_powerup_size = int(window_width/15)
```

With all of this initialized, I can then go into main.py and start on the actual functionality of the powerup collectables.

First, I initialized a powerups list to store all currently existing powerup collectables on the map, and a powerup status dictionary initialized with keys corresponding to all powerups in the game (this is where the powerup_icons dictionary in config.py comes in handy!) with an active switch and a timer for each.

```
powerups = []
powerups_status = {id: {"active": False, "timer": 0} for id in powerup_icons.keys() if id != "health"}
```

I then went into the game loop and had it so every frame there is a small chance (decided by a global variable powerup_chance_freq which I put in config.py) that a powerup will spawn on the map. If this becomes true, then a random powerup is picked from the list of powerup keys and instantiated using the Powerups class, before being appended to the powerups list. This let me iterate over every currently existing powerup and run their relevant methods, as well as control their timers to ensure they despawn after a certain amount of time, and detecting when a player collides with a powerup and updating the powerup_status dictionary accordingly.

```
# At a random chance every frame, spawn a powerup
if random.randint(1, FPS*powerup_chance_freq) == 1:
    powerup_choice = Powerup(random.choice(list(powerup_icons.keys())))
    powerups.append(powerup_choice)

# Run the main for the existing powerups on the map
for powerup in powerups:
    powerup.main(WINDOW)
    powerup.set_timer(powerup.get_timer() + 1)
    # If the lifetime of the powerup runs out, remove it
    if powerup.get_timer() >= FPS * powerup_lifetime:
        powerups.remove(powerup)

    # If the player collides with the powerup, remove it and make it active
    if powerup.hitbox(WINDOW).colliderect(player.hitbox(WINDOW)):
        powerups.remove(powerup)
        if powerup.identify() != "health":
            powerups_status[powerup.identify()]["active"] = True
            powerups_status[powerup.identify()]["timer"] = 0
        else:
            powerup.health(player)
```

Running this code, the powerups successfully spawn on the map at random locations and random times:

However, right now all the player can do with these powerups is collect them. To make them influence the player, I will need a set of methods that run when a powerup is activated. For this, I will create a 2nd class called ActivePowerups, that will contain the methods needed to run each powerup.

```
class ActivePowerups():
    def __init__(self, active_powerups):
        self.active_powerups = active_powerups

    def main(self, WINDOW):
        # Number of active powerups for active icon formatting
        self.num_active_powerups = 0
        for powerup in self.active_powerups:
            if self.active_powerups[powerup]["active"]:
                self.display_icon(WINDOW, powerup, self.num_active_powerups, self.active_powerups[powerup]["timer"])
                self.num_active_powerups += 1
                # Timer for powerups incrementing
                self.active_powerups[powerup]["timer"] += 1
                if self.active_powerups[powerup]["timer"] >= FPS*powerup_duration:
                    self.active_powerups[powerup]["active"] = 0
                    self.active_powerups[powerup]["timer"] = 0

    def activate_powerups(self, player, enemies):
        # Speed powerup
        if self.active_powerups["speed"]["active"]:
            player.speed_on()
        else:
            player.speed_off()
        # Shield powerup
        if self.active_powerups["shield"]["active"]:
            player.shield_on()
        else:
            player.shield_off()
        # Poison powerup
        if self.active_powerups["poison"]["active"]:
            player.poison_on()
        else:
            player.poison_off()
        # Freeze powerup
        if self.active_powerups["freeze"]["active"]:
            for enemy in enemies:
                enemy.freeze_on()
        else:
            for enemy in enemies:
                enemy.freeze_off()

    def display_icon(self, WINDOW, powerup_id, num_active_powerups, timer):
        # Display all active powerups as large icons in the corner, along with how long it has left
        active_powerup_icon = pygame.transform.scale(powerup_icons[powerup_id], (active_powerup_size, active_powerup_size))
        WINDOW.blit(active_powerup_icon, (int(window_width/40) + (num_active_powerups*(active_powerup_size+20)), int(window_width/40)))
        timer_display = smallfont.render(str(int(((FPS*powerup_duration - timer)/FPS) + 1)), True, (0,0,0))
        WINDOW.blit(timer_display, (int(window_width/40) + (num_active_powerups*(active_powerup_size+20)) + int(active_powerup_size/2) - int(timer_display.get_width()/2), int(window_width/40) +
        active_powerup_size))
```

Here, I have not only created a method to activate any active powerups, but also created a private method to display said active powerups in the corner, letting the player know which powerups are active and how long each of them has left.

Notice how I have run these random methods on the player and enemies such as freeze_on or speed_on. These mean nothing for now, so we will need to go back into the player and enemy classes and add these methods and attributes accordingly.

```
self.health = health
self.speed = speed

self.shield = False
self.poison = False
```

```
self.frozen = False
```

```
def freeze_on(self):
    self.frozen = True
def freeze_off(self):
    self.frozen = False
```

```
def speed_on(self):
    self.speed = int(player_speed * powerup_speed_multiplier)
def speed_off(self):
    self.speed = player_speed

def get_shield_status(self):
    return self.shield
def shield_on(self):
    self.shield = True
def shield_off(self):
    self.shield = False

def get_poison_status(self):
    return self.poison
def poison_on(self):
    self.poison = True
def poison_off(self):
    self.poison = False
```

Except for speed, these attributes and methods currently don't affect anything, so to have them be able to do something we will need to go back into main.py and code in their functionalities.

In the central enemy iterative loop in the game loop, I have updated the damage dealing code, so it first checks to see if the player has a shield or poison powerup active before dealing the damage.

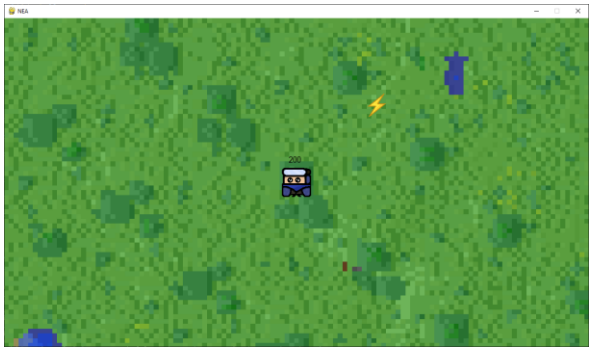
```
for enemy in enemies:
    enemy.main(WINDOW)

    # If any enemy collides with the player's hitbox, the player takes damage at a delayed interval
    if enemy.hitbox(WINDOW).colliderect(player.hitbox(WINDOW)):

        if enemy.get_damage_delay_timer() == 0:
            if player.get_poison_status() == True:
                enemy.set_health(enemy.get_health() - powerup_poison_damage)
            else:
                if player.get_shield_status() == True:
                    continue
                else:
                    player.set_health(player.get_health() - enemy.get_damage())

        enemy.set_damage_delay_timer(enemy.get_damage_delay_timer() + 1)

        if enemy.get_damage_delay_timer() >= FPS * enemy.get_damage_speed():
            enemy.set_damage_delay_timer(0)
    else:
        enemy.set_damage_delay_timer(0)

    # If a shooter enemy's shooting cooldown runs out, fire another shot at the player
    if enemy.identify() == "ShooterEnemy":
        if enemy.get_projectile_cooldown() == 0 and enemy.get_frozen_status() == False:
            projectiles.append(EnemyProjectile(speed=shooter_enemy_proj_speed, damage=shooter_enemy_proj_damage, origin_x=enemy.get_xy("x"), origin_y=enemy.get_xy("y"), sprite_img=
            (0,0,0)))
        enemy.set_projectile_cooldown(enemy.get_projectile_cooldown() + 1)
        if enemy.get_projectile_cooldown() >= FPS * enemy.get_projectile_freq():
            enemy.set_projectile_cooldown(0)
```

I also needed to go back into the enemy class and add the freeze effect to the main method, so they only run the movement code when not frozen:

```
# If frozen effect is false, let the enemies move
if self.frozen == False:
    if self.x < centre_x + self.offset_x + camera_scroll[0] and self.x < centre_x + self.border_x:
        self.x += self.speed
    elif self.x > centre_x + self.offset_x + camera_scroll[0] and self.x > centre_x - self.border_x:
        self.x -= self.speed
    if self.y < centre_y + self.offset_y + camera_scroll[1] and self.y < centre_y + self.border_y:
        self.y += self.speed
    elif self.y > centre_y + self.offset_y + camera_scroll[1] and self.y > centre_y - self.border_y:
        self.y -= self.speed
```

With all powerup functionalities coded, I can test them. Running the game, decreasing my health down and then collecting a health kit successfully increased my health up by 50, my desired amount:

Collecting a speed powerup not only successfully increases speed, but also displays the lightning icon in the corner of the screen, along with the timer telling you there are less than 10 seconds left before the speed goes away. After the timer ends, the speed goes back down to normal:



Collecting a shield powerup displays the icon and its timer, and when colliding with enemies, health does not go down at all until the timer runs out:

Collecting a poison powerup also displays its icon and timer, and when colliding with enemies, the enemy's health goes down instead while keeping the player's health the same:

Collecting a freeze powerup displays its icon and timer, and causes enemies to stop moving for that duration of time:

All powerups have been successfully created!

This marks the end of the main game implementation. However, this is not the end of making changes to this section of the game, since we will still have other things to implement here such as sound effects, better sprite images, coins, enemy variants, etc. But for now, since these are of less priority, we will put a pause on the development of the game itself and begin implementing the other parts of the program.

Menus and Settings

This program is going to include many different menu screens, and for this I will need multiple different functions to be run when needed. The obvious solution here is to create a class, that will provide a host of methods that can be run in turn depending on what menu the program should be on right now.

Creating the class is easy: I simply took my entire game code in main.py and indented it, put it in a class called GameState, and have the main game code be in a method called main_game.

```
class GameState():

    def __init__(self):
        self.state = "main_game"

    def main_game(self):

        # Instantiate basic objects needed for game and create lists for arrays of objects
        player = Player(health=player_health, speed=player_speed)
        enemies = []
        projectiles = []
        powerups = []
```

Notice how in the constructor method I have created an attribute called state. This will ultimately control which menu screen should be on, and for now I have initialized it to be the main game. This will change in due course.

Next, I added another method called run, which will use the state attribute to select the method to run.

```
def run(self):
    if self.state == "main_game":
        self.main_game()
```

Finally, I put this class to use by creating an object, game, at the very end of main.py and run it indefinitely using a while loop:

```
game = GameState()
while True:
    game.run()
```

Testing this code, the program performed as normal, opening the program on the game screen.

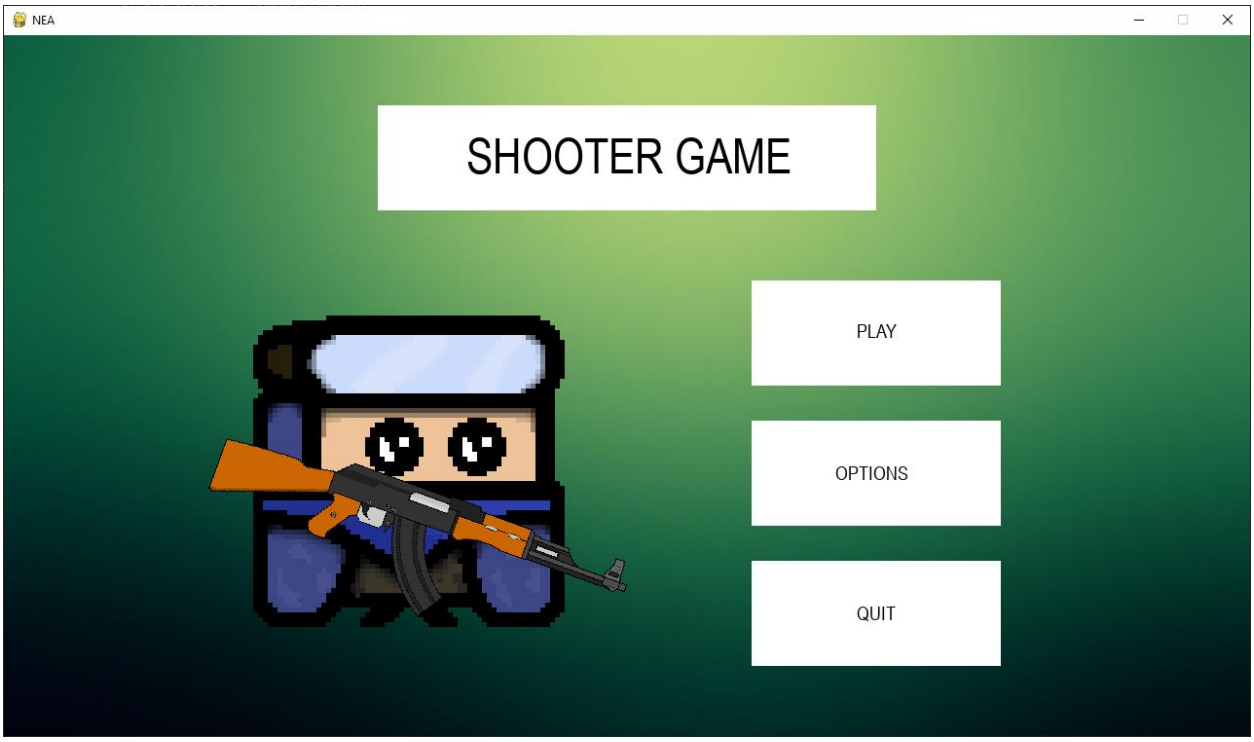
Menu Screens

Now, I needed to create a bunch of other methods to make more menus. The main one of course is the main menu, which I made like so:

```
def main_menu(self):

    while True:
        pygame.display.update()
        # bg colour
        WINDOW.blit(menubg, (0,0))

        # Player and gun
        WINDOW.blit(pygame.transform.scale(player_walk_images[0], (320, 320)), (256, 288))
        WINDOW.blit(pygame.transform.rotate(pygame.transform.scale(gun_img, (458,312)), -20), (150,300))

        # title
        pygame.draw.rect(WINDOW, WHITE, (384, 72, 512, 108))
        WINDOW.blit(mediumfont.render("SHOOTER GAME", True, BLACK), (475, 95))

        # play button
        play_button = pygame.draw.rect(WINDOW, WHITE, (768, 252, 256, 108))
        WINDOW.blit(smallfont.render("PLAY", True, BLACK), (876, 292))

        # settings button
        options_button = pygame.draw.rect(WINDOW, WHITE, (768, 396, 256, 108))
        WINDOW.blit(smallfont.render("OPTIONS", True, BLACK), (854, 438))

        # quit button
        quit_button = pygame.draw.rect(WINDOW, WHITE, (768, 540, 256, 108))
        WINDOW.blit(smallfont.render("QUIT", True, BLACK), (876, 582))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if play_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "dif_mode_menu"
                    return
            if options_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "options"
                    return
            if quit_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "quit"
                    return

        clock.tick(FPS)
```

```
# Controls which method to run depending on current state
def run(self):
    if self.state == "main_menu":
        self.main_menu()
    elif self.state == "settings":
        self.settings()
    elif self.state == "main_game":
        self.main_game()
    elif self.state == "quit":
        pygame.quit()
        sys.exit()
```

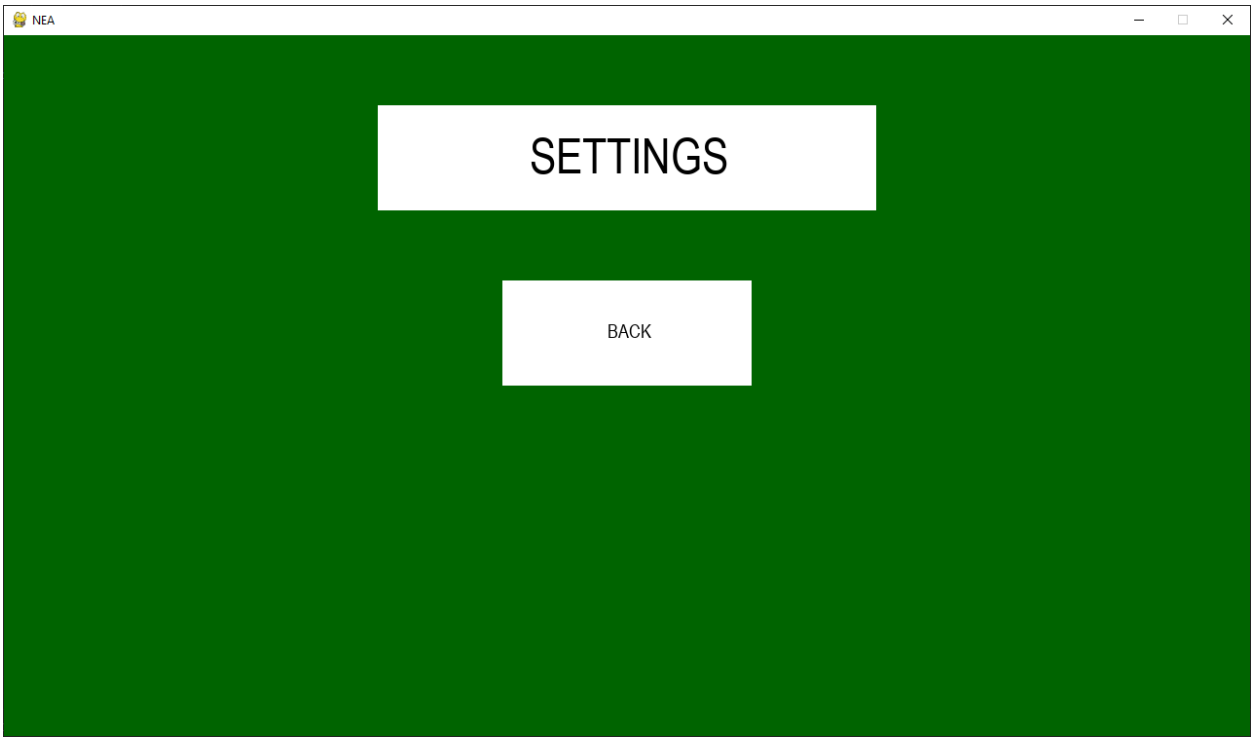
Creating the UI for the menu took a lot of trial and erroring to make sure the positioning of everything was perfect, but in the end, it turned out great and now when I open the program it starts on the main menu:



This also calls for a settings menu to be created, which I will not focus on for now, but will still need to create a method for it to form the foundation for the menu links.

```
def settings(self):
    while True:
        pygame.display.update()

        # bg colour
        WINDOW.fill((0, 100, 0))

        # title
        pygame.draw.rect(WINDOW, WHITE, (384, 72, 512, 108))
        WINDOW.blit(mediumfont.render("SETTINGS", True, BLACK), (540, 95))

        # back
        back_button = pygame.draw.rect(WINDOW, WHITE, (512, 252, 256, 108))
        WINDOW.blit(smallfont.render("BACK", True, BLACK), ((620, 292)))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if back_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "main_menu"
                    return

        clock.tick(FPS)
```

Now when I run the program and click on the settings button:



The main menu works perfectly now, with the settings button redirecting me to settings page, the back button taking me back to main menu, the quit button terminating the program, and the play button taking me to the game page!

However, according to my design section, I need other menus to go before the main game state: the select mode/difficulty menu, and possibly a theme menu. Here is the code for the UI:

```
def dif_mode_menu(self):

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.fill((0, 100, 0))

        # title
        pygame.draw.rect(WINDOW, WHITE, (384, 72, 512, 144))
        WINDOW.blit(mediumfont.render("SELECT DIFFICULTY", True, BLACK), (438, 90))
        WINDOW.blit(mediumfont.render("AND GAME MODE", True, BLACK), (462, 140))

        # easy difficulty
        if self.difficulty == "easy":
            easy_button = pygame.draw.rect(WINDOW, GREEN, (416, 252, 128, 108))
        else:
            easy_button = pygame.draw.rect(WINDOW, WHITE, (416, 252, 128, 108))
        WINDOW.blit(smallfont.render("EASY", True, BLACK), (458, 292))
        # normal difficulty
        if self.difficulty == "normal":
            normal_button = pygame.draw.rect(WINDOW, GREEN, (576, 252, 128, 108))
        else:
            normal_button = pygame.draw.rect(WINDOW, WHITE, (576, 252, 128, 108))
        WINDOW.blit(smallfont.render("NORMAL", True, BLACK), (606, 292))
        # hard difficulty
        if self.difficulty == "hard":
            hard_button = pygame.draw.rect(WINDOW, GREEN, (736, 252, 128, 108))
        else:
            hard_button = pygame.draw.rect(WINDOW, WHITE, (736, 252, 128, 108))
        WINDOW.blit(smallfont.render("HARD", True, BLACK), (778, 292))
```

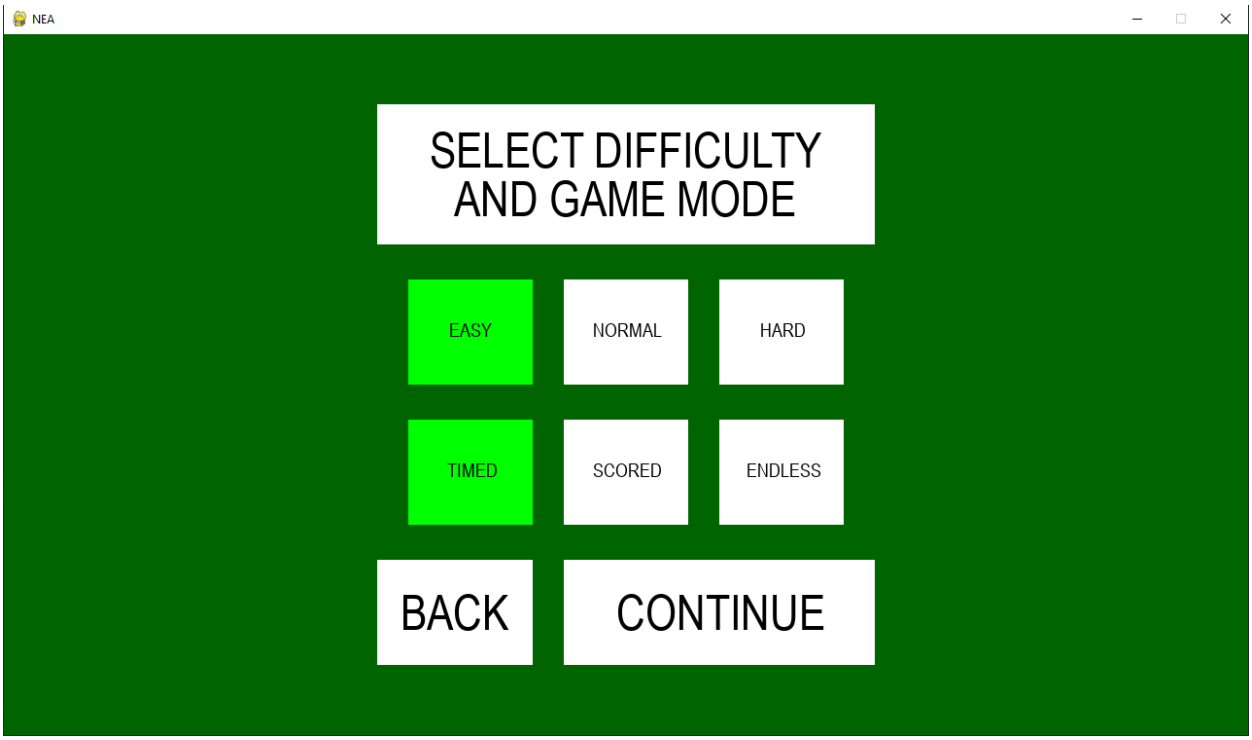
```
        # timed mode
        if self.mode == "timed":
            timed_button = pygame.draw.rect(WINDOW, GREEN, (416, 396, 128, 108))
        else:
            timed_button = pygame.draw.rect(WINDOW, WHITE, (416, 396, 128, 108))
        WINDOW.blit(smallfont.render("TIMED", True, BLACK), (456, 436))
        # scored mode
        if self.mode == "scored":
            scored_button = pygame.draw.rect(WINDOW, GREEN, (576, 396, 128, 108))
        else:
            scored_button = pygame.draw.rect(WINDOW, WHITE, (576, 396, 128, 108))
        WINDOW.blit(smallfont.render("SCORED", True, BLACK), (606, 436))
        # endless mode
        if self.mode == "endless":
            endless_button = pygame.draw.rect(WINDOW, GREEN, (736, 396, 128, 108))
        else:
            endless_button = pygame.draw.rect(WINDOW, WHITE, (736, 396, 128, 108))
        WINDOW.blit(smallfont.render("ENDLESS", True, BLACK), (764, 436))
```

```
# back
back_button = pygame.draw.rect(WINDOW, WHITE, (384, 540, 160, 108))
WINDOW.blit(mediumfont.render("BACK", True, BLACK), (408, 565))
# continue
continue_button = pygame.draw.rect(WINDOW, WHITE, (576, 540, 320, 108))
WINDOW.blit(mediumfont.render("CONTINUE", True, BLACK), (630, 565))

# Checking for events
for event in pygame.event.get():
    if event.type == QUIT:
        pygame.quit()
        sys.exit()

    # Checking for cursor coordinates
    cursor = pygame.mouse.get_pos()
```

```
    # Checking for clicks on each button and changing the settings accordingly
    if easy_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.difficulty = "easy"
    if normal_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.difficulty = "normal"
    if hard_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.difficulty = "hard"
    if timed_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.mode = "timed"
    if scored_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.mode = "scored"
    if endless_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.mode = "endless"
    if back_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.state = "main_menu"
            return
    if continue_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            self.state = "main_game"
            return

clock.tick(FPS)
```

As you can see from the code, I have made it so on the modes and difficulties menu, there are 8 different buttons: 3 for difficulties, 3 for modes, a back button and a continue button. However, unlike the normal buttons, the selection buttons do not redirect to the next menu. Instead they toggle and set a mode and difficulty, which I declared in the constructor method (self.mode and self.difficulty). Once set, I can use the values in these attributes to influence the main game method.

Running the code now and clicking play, the program successfully opens the difficulty and mode menu, like so:



The green highlights are which mode and difficulty are currently selected. I can then select the other modes, shifting the green highlight onto the corresponding button. Then when I press continue, I am dropped into the game with those settings set in the GameState's attributes.

After some thought, I have decided to implement the themes menu, since it would add another fun aspect to the game as well as allow the player to decide what aesthetic they would like.

First, I created a themes dictionary, that would hold the different images for each theme:

```
theme_options = {
    0: {
        "name": "grassland",
        "banner": pygame.transform.scale(pygame.image.load("assets/grassland_banner.jpg"), (448, 216)),
        "map": pygame.transform.scale(pygame.image.load("assets/grassland_map.png"), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png"), (enemy_size, enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png"), (enemy_size, enemy_size))
        }
    },
    1: {
        "name": "desert",
        "banner": pygame.transform.scale(pygame.image.load("assets/desert_banner.jpg"), (448, 216)),
        "map": pygame.transform.scale(pygame.image.load("assets/desert_map.png"), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite_desert.png"), (enemy_size, enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite_desert.png"), (enemy_size, enemy_size))
        }
    },
    2: {
        "name": "city",
        "banner": pygame.transform.scale(pygame.image.load("assets/city_banner.jpg"), (448, 216)),
        "map": pygame.transform.scale(pygame.image.load("assets/city_map.png"), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png"), (enemy_size, enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png"), (enemy_size, enemy_size))
        }
    },
    3: {
        "name": "lava",
        "banner": pygame.transform.scale(pygame.image.load("assets/volcano_banner.png"), (448, 216)),
        "map": pygame.transform.scale(pygame.image.load("assets/volcano_map.png"), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png"), (enemy_size, enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png"), (enemy_size, enemy_size))
        }
    }
}
```

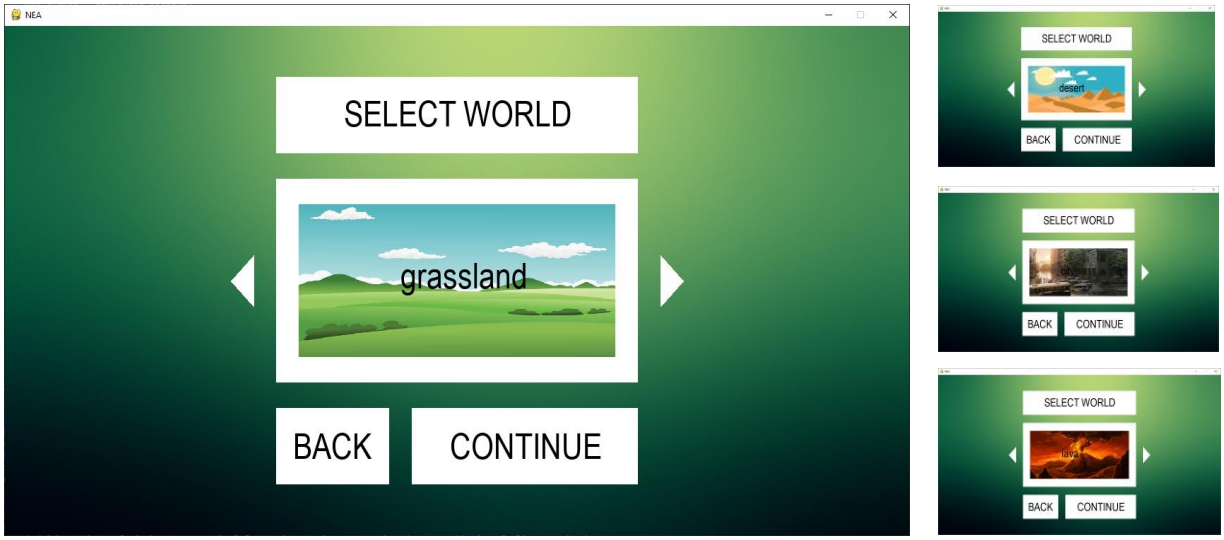
As you can see, I have decided on 4 themes for now: grassland, desert, city and lava (or volcano; still haven't decided on a good name yet).

Next, I added another method into the GameState class, theme_menu, and an attribute theme, as well as updating the run method to accommodate this new game state.

```
def __init__(self):
    self.state = "main_menu"

    self.difficulty = "easy"
    self.mode = "timed"

    self.theme = theme_options[0]
```

```
def theme_menu(self):

    num_themes = len(theme_options)
    theme_index = 0

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.blit(menubg, (0,0))

        # title
        pygame.draw.rect(WINDOW, WHITE, (384, 72, 512, 108))
        WINDOW.blit(mediumfont.render("SELECT WORLD", True, BLACK), (480, 95))

        # themes
        pygame.draw.rect(WINDOW, WHITE, (384, 216, 512, 288))
        WINDOW.blit(theme_options[theme_index]["banner"], (416, 252))
        WINDOW.blit(mediumfont.render(theme_options[theme_index]["name"], True, BLACK), (560, 324))
        # arrows
        left_button = pygame.draw.polygon(WINDOW, WHITE, [[352, 324], [320, 360], [352, 396]])
        right_button = pygame.draw.polygon(WINDOW, WHITE, [[928, 324], [928, 396], [960, 360]])

        # back
        back_button = pygame.draw.rect(WINDOW, WHITE, (384, 540, 160, 108))
        WINDOW.blit(mediumfont.render("BACK", True, BLACK), (408, 565))
        # continue
        continue_button = pygame.draw.rect(WINDOW, WHITE, (576, 540, 320, 108))
        WINDOW.blit(mediumfont.render("CONTINUE", True, BLACK), (630, 565))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if left_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    if theme_index != 0:
                        theme_index -= 1
            if right_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    if theme_index != num_themes - 1:
                        theme_index += 1
            if back_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "dif_mode_menu"
                    return
            if continue_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "main_game"
                    self.theme = theme_options[theme_index]
                    return

        clock.tick(FPS)
```

Running this code, the following screens are displayed when I press the relevant buttons and click on the arrows, allowing me to select different themes:



Finally, I needed a way for this option to affect the images that rendered in the actual game based on the theme_options dictionary. To do this, I simply removed the import image declarations in config.py and took them into main.py's main_game method, and initializing them there like so:

```
def main_game(self):

    mapimg = self.theme["map"]
    enemy_sprite = self.theme["enemies"]["basic"]
    shooter_sprite = self.theme["enemies"]["shooter"]
```

Running the code and going onto the main game, testing each theme in turn, the relevant map and enemy images are rendered onto the screen!

However, a glaring problem arose after testing a few more times. After dying in the main game, I am taken back to the main menu of course. But if during the same run I start a new game, I am spawned at the location of my last death rather than back in the middle. This was a problem since I needed the player to spawn in the middle every time.

Some investigation led me to realize that this was because after the player's death, I wasn't resetting the camera_scroll variable back to [0,0]. It was then when I realized that my mistake was in putting camera_scroll in config.py. This meant that after the variable was initialized, there was no way of manually setting camera_scroll to a new list of values. So, to fix the problem, I removed the variable from config.py and put it at the top of the main_game method. Testing the program now showed that the bug was fixed, and the player spawns at the center of the screen every game!

Difficulties

With the basic menu system created, I now need to add the functionality for each of these menus. The glaringly important one is the difficulties and modes menu, which will directly influence settings in the main game, such as entity attributes and timers.

The lucky part for the difficulty section is that I have given myself an easy job by putting all the global variables in one place, config.py. This will make it much simpler to create difficulty settings, configuring all the different attribute values depending on each difficulty.

First, I made a new dictionary in config.py that will hold the different values for each difficulty. This will also make it easier to alter the values to make the game more balanced after testing.

Next, I moved all the global variables in config.py into main.py's main_game method, and changed them to call the values in the settings dictionary depending on the selected difficulty (set in the dif_mode_menu method).

```
def main_game(self):

    # Difficulty settings
    powerup_chance_freq = settings[self.difficulty]["powerup"]["chance-freq"]
    powerup_lifetime = settings[self.difficulty]["powerup"]["lifetime"]
    powerup_duration = settings[self.difficulty]["powerup"]["duration"]
    powerup_health_boost = settings[self.difficulty]["powerup"]["health-boost"]
    powerup_speed_multiplier = settings[self.difficulty]["powerup"]["speed-mult"]
    powerup_poison_damage = settings[self.difficulty]["powerup"]["poison-damage"]
    # Player attributes
    player_max_health = settings[self.difficulty]["player"]["max-health"]
    player_health = player_max_health
    player_speed = settings[self.difficulty]["player"]["speed"]
    player_proj_speed = settings[self.difficulty]["player"]["proj-speed"]
    player_proj_damage = settings[self.difficulty]["player"]["proj-damage"]
    # Enemy attributes
    basic_enemy_health = settings[self.difficulty]["enemy"]["basic"]["max-health"]
    basic_enemy_speed = settings[self.difficulty]["enemy"]["basic"]["speed"]
    basic_enemy_damage = settings[self.difficulty]["enemy"]["basic"]["damage"]
    shooter_enemy_proj_freq = settings[self.difficulty]["enemy"]["shooter"]["proj-freq"]
    shooter_enemy_proj_speed = settings[self.difficulty]["enemy"]["shooter"]["proj-speed"]
    shooter_enemy_proj_damage = settings[self.difficulty]["enemy"]["shooter"]["proj-damage"]
```

Running the game brought up several errors due to these variables not being in config.py anymore, so can't be imported by several functions in sprites.py, but these were quickly fixed by passing in the relevant variables into the sprite class methods as parameters.

Now, all that was left for difficulties was to test out each difficulty and balance the values in the dictionary to accurately reflect that difficulty level.

Modes

Next, I needed to implement the various game modes for the game. To do this, I needed to add a timer and score counter into the game, since these would be very relevant depending on game mode.

First, I added timer and score count attributes to the GameState class, which were both initialized to 0. Next, I initialized a frame counter right before the game loop starts, and every

frame I had it increment by 1. This would then affect the timer variable, which updates every FPS number of frames, so it goes up by 1 every second.

```
self.timer = 0
self.score_count = 0
```

```
# Initialize frame count for timer
frames = 0

# Main loop
while True:
    pygame.display.update()
```

```
# Timer
frames += 1
if frames % FPS == 0:
    self.timer += 1
    frames = 0

# Displaying the timer, adding a number of 0s to the start to make it 3 digits
time_display = str(self.timer)
zeros = (3 - len(time_display)) % 3
for i in range(zeros):
    time_display = "0" + time_display
WINDOW.blit(mediumfont.render(time_display, True, BLACK), (1190, 20))

clock.tick(FPS)
```

Running this code, the timer does indeed display on the screen with the correct number of 0s!

Next was the score count. This was as simple as just blitting the value of the score attribute:

```
# Display Score
WINDOW.blit(largefont.render(str(self.score_count), True, BLACK), (1170,580))
```

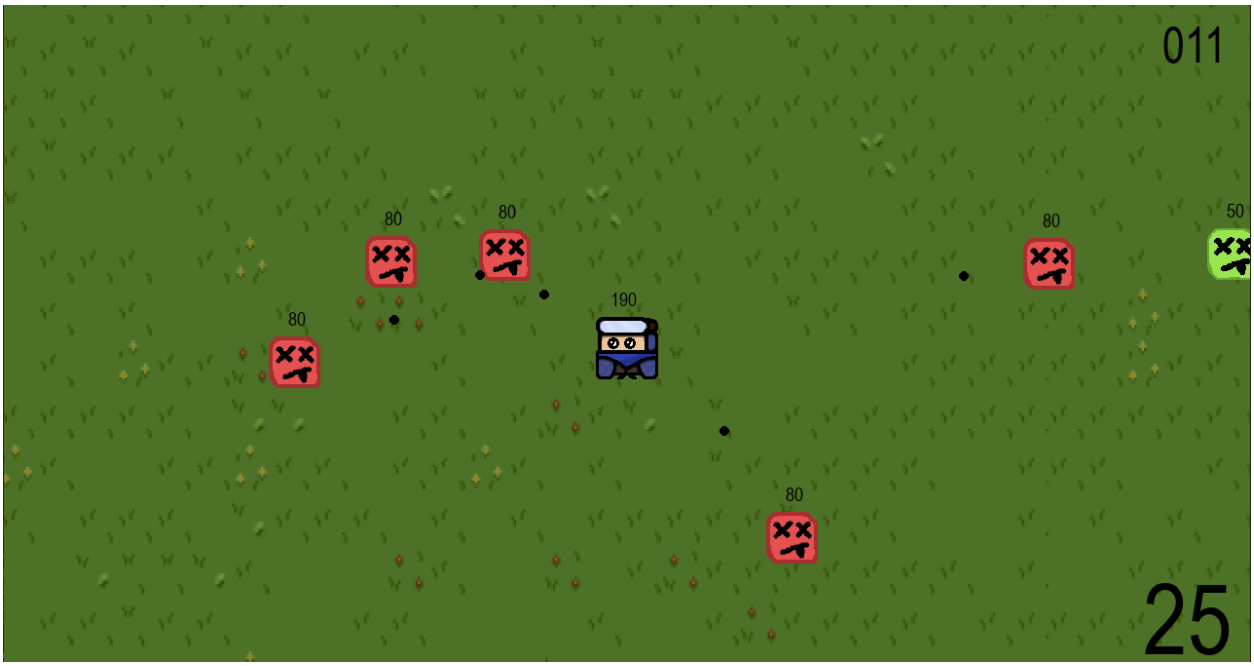
I also needed a way to manipulate this score count, since right now it stays on 0. So I first changed the settings dictionary in config.py by adding a score key to each enemy for each difficulty, which is the score you earn for killing a single enemy of that type.

I set it so on easy mode you get 5 or 10 points for killing basic and shooter enemies respectively, on normal mode you get 10 or 20, and on hard mode you get 20 or 30. Notice the differences in score, because I am aiming to maintain the balance described in this part of the design section. I will modify these values in the future if needed to balance the game.

Finally, I added the incrementing score functionality to the main game code so that the score goes up by the required amount when an enemy is killed.

```
for enemy in enemies:
    if enemy.get_health() <= 0:
        self.score_count += settings[self.difficulty]["enemy"][enemy.identify()]["score"]
        enemies.remove(enemy)
```

Running the code now and killing a few enemies on each difficulty confirmed that this code worked and that the score was indeed going up by the desired amount!



Now it was time to implement the modes themselves.

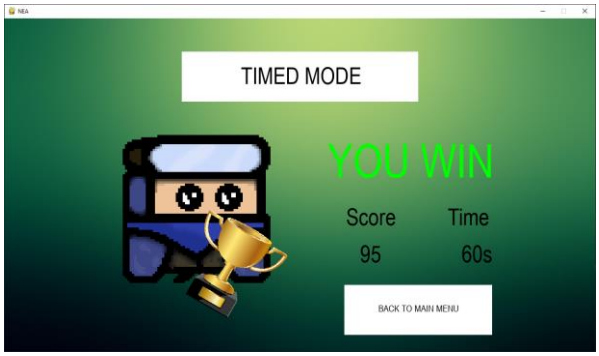
First, I added 2 if-statements that would check if the mode is timed or scored, and end the game if the score or time exceeded the declared amount in config.py, taking the player into an unimplemented game over menu.

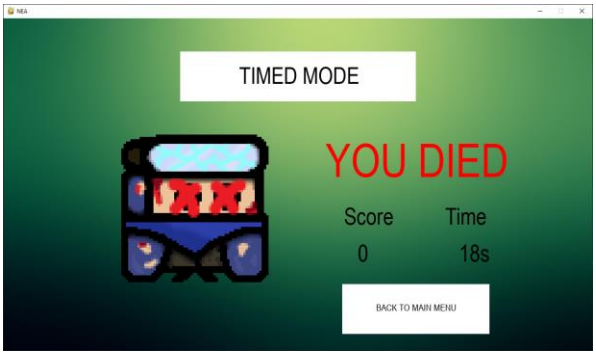
```
# Winning the game
if self.mode == "timed":
    if self.timer >= time_limit:
        self.state = "game_over"
        return
elif self.mode == "scored":
    if self.score_count >= score_goal:
        self.state = "game_over"
        return
```

Of course, we can't run this code without having a game over menu, so I added the method accordingly and updated the inter-method links.

```
def game_over(self):

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.blit(menubg, (0,0))

        # title
        pygame.draw.rect(WINDOW, WHITE, (384, 72, 512, 108))
        WINDOW.blit(mediumfont.render(self.mode.upper() + " MODE", True, BLACK), (512, 95))

        # Image of player either dead or holding a trophy
        if self.died:
            WINDOW.blit(player_dead_img, (256, 252))
            WINDOW.blit(largefont.render("YOU DIED", True, RED), (700, 250))
        else:
            WINDOW.blit(player_menu_img, (256, 252))
            WINDOW.blit(pygame.transform.rotate(pygame.transform.scale(trophy_img,(200,200)), (-30)), (350, 400))
            WINDOW.blit(largefont.render("YOU WIN", True, GREEN), (700, 250))

        # Score/time from this game
        WINDOW.blit(mediumfont.render("Score", True, BLACK), (740, 400))
        WINDOW.blit(mediumfont.render(str(self.score_count), True, BLACK), (770, 480))
        WINDOW.blit(mediumfont.render("Time", True, BLACK), (960, 400))
        WINDOW.blit(mediumfont.render(str(self.timer) + "s", True, BLACK), (990, 480))

        # Back to menu button
        menu_button = pygame.draw.rect(WINDOW, WHITE, (736, 576, 320, 108))
        WINDOW.blit(smallfont.render("BACK TO MAIN MENU", True, BLACK), (810, 615))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if menu_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    self.state = "main_menu"
                    return

        clock.tick(FPS)
```

I also added another attribute called died, which is a Boolean that determines whether the player died or not. This allowed me to generate 2 different game-over screens. Running this code and winning and losing respectively displayed these screens (the left screen was because I set the time limit to 60s):





I also tested a few more times to check that the other modes were working correctly, and they did indeed end the games when I wanted them to (timed: ends at 60s, scored: ends when I reach 100 score, endless: only ends when I die).

I also realized that endless mode could never get the screen on the left, so to make it make more sense I changed the text from "YOU WIN" to "YOU LIVED".

High Scores

For each mode respectively, I will need to log the player's best score, time, and both. For this, I will need to utilize external file handling. I have decided on JSON for this.

Firstly, I created the JSON file and named it save.json because it will be containing the data I want saved from the game. This contains the player's coin balance, and their high scores for each mode. I initialized best time to null instead of 0, because I will be adding shorter times on so it wouldn't make sense to start at 0. This does not apply to endless, where you are trying to survive for as long as possible.

```
{} save.json > ...
{
    "coins": 0,
    "high-score": 0,
    "best-time": null,
    "endless": {
        "high-score": 0,
        "best-time": 0
    }
}
```

Then, I went into main.py and loaded the file, putting the data into a variable called save which I would use to display high scores in the difficulties/modes menu. I then went into the game over method and added the updating save code into it, so it would update the contents of save.json whenever a new high score or best time came up.

```
# Load save data
with open("save.json", "r") as save_file:
    save = json.load(save_file)
```

```
def game_over(self):

    new_high_score = False
    new_best_time = False

    # Updating high scores and best times if applicable
    if not self.died or self.mode == "endless":
        with open("save.json", "r") as update_save:
            update_data = json.load(update_save)
            # If mode is timed, update high score if high enough
            if self.mode == "timed":
                if self.score_count > update_data["high-score"]:
                    update_data["high-score"] = self.score_count
                    new_high_score = True
            # If mode is scored, update best time if lower or not set yet
            elif self.mode == "scored":
                if update_data["best-time"] is not None:
                    if self.timer < update_data["best-time"]:
                        update_data["best-time"] = self.timer
                        new_best_time = True
                else:
                    update_data["best-time"] = self.timer
                    new_best_time = True
            # If mode is endless, update score or time if higher (extra section in json file)
            elif self.mode == "endless":
                update_endless = [False, False]
                if self.score_count > update_data["endless"]["high-score"]:
                    update_data["endless"]["high-score"] = self.score_count
                    new_high_score = True
                if self.timer > update_data["endless"]["best-time"]:
                    update_data["endless"]["best-time"] = self.timer
                    new_best_time = True
        # dump this updated data into json
        with open("save.json", "w") as update_save:
            update_save.write(json.dumps(update_data))
```

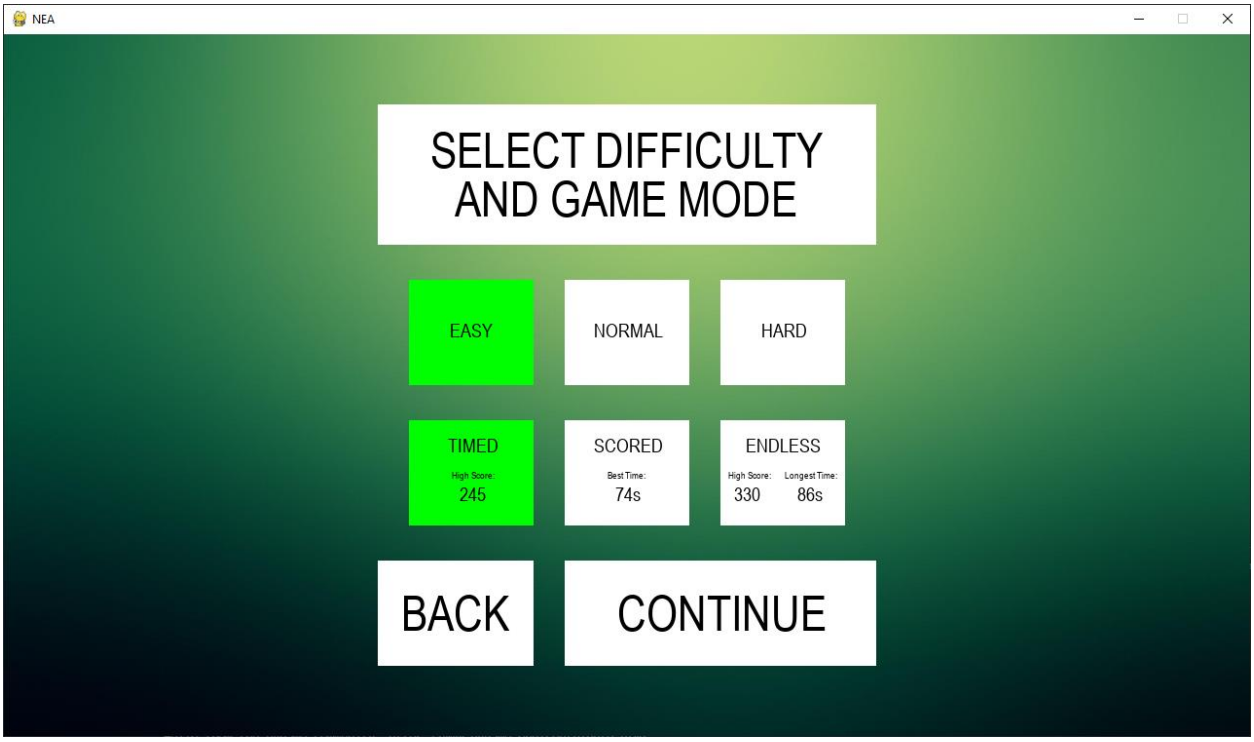
Running the code multiple times and testing each game mode successfully updates the json file where there is a better score or time achieved!

```
{} save.json > ...
1   {"coins": 0, "high-score": 245, "best-time": 74, "endless": {"high-score": 330, "best-time": 86}}
```

Now we need a way to display this data to the user. For this, we need to use the save variable that we loaded earlier. I decided to display this in the difficulties/modes menu, so I used the save variable to display the relevant dictionary contents on the menu.

```
# timed mode
if self.mode == "timed":
    timed_button = pygame.draw.rect(WINDOW, GREEN, (416, 396, 128, 108))
else:
    timed_button = pygame.draw.rect(WINDOW, WHITE, (416, 396, 128, 108))
WINDOW.blit(smallfont.render("TIMED", True, BLACK), (456, 410))
WINDOW.blit(tinyfont.render("High Score:", True, BLACK), (460, 446))
WINDOW.blit(smallfont.render(str(save["high-score"]), True, BLACK), (468, 460))
# scored mode
if self.mode == "scored":
    scored_button = pygame.draw.rect(WINDOW, GREEN, (576, 396, 128, 108))
else:
    scored_button = pygame.draw.rect(WINDOW, WHITE, (576, 396, 128, 108))
WINDOW.blit(smallfont.render("SCORED", True, BLACK), (606, 410))
WINDOW.blit(tinyfont.render("Best Time:", True, BLACK), (620, 446))
WINDOW.blit(smallfont.render(str(save["best-time"]) + "s", True, BLACK), (628, 460))
# endless mode
if self.mode == "endless":
    endless_button = pygame.draw.rect(WINDOW, GREEN, (736, 396, 128, 108))
else:
    endless_button = pygame.draw.rect(WINDOW, WHITE, (736, 396, 128, 108))
WINDOW.blit(smallfont.render("ENDLESS", True, BLACK), (762, 410))
WINDOW.blit(tinyfont.render("High Score:", True, BLACK), (744, 446))
WINDOW.blit(tinyfont.render("Longest Time:", True, BLACK), (800, 446))
WINDOW.blit(smallfont.render(str(save["endless"]["high-score"]), True, BLACK), (750, 460))
WINDOW.blit(smallfont.render(str(save["endless"]["best-time"]) + "s", True, BLACK), (815, 460))
```

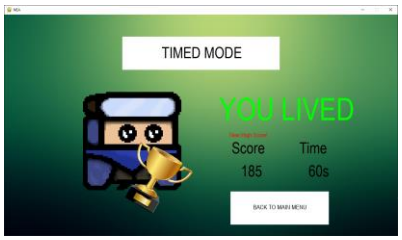
Running the code now displays the player's high score on the screen!

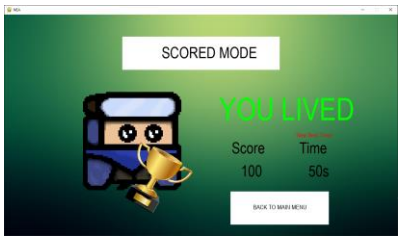


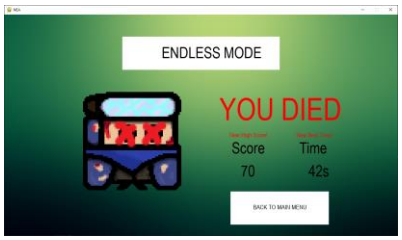
Finally, I added some code to the game over screen to show the player when a new high score or best time had been achieved:

```
# Score/time from this game
if new_high_score:
    WINDOW.blit(smallfont.render("New High Score!", True, RED), (733, 380))
WINDOW.blit(mediumfont.render("Score", True, BLACK), (740, 405))
WINDOW.blit(mediumfont.render(str(self.score_count), True, BLACK), (770, 480))
if new_best_time:
    WINDOW.blit(smallfont.render("New Best Time!", True, RED), (950, 380))
WINDOW.blit(mediumfont.render("Time", True, BLACK), (960, 405))
WINDOW.blit(mediumfont.render(str(self.timer) + "s", True, BLACK), (990, 480))
```

Now, whenever I get a new high score or best time in any game mode, it will let me know:







Coins and Cosmetics

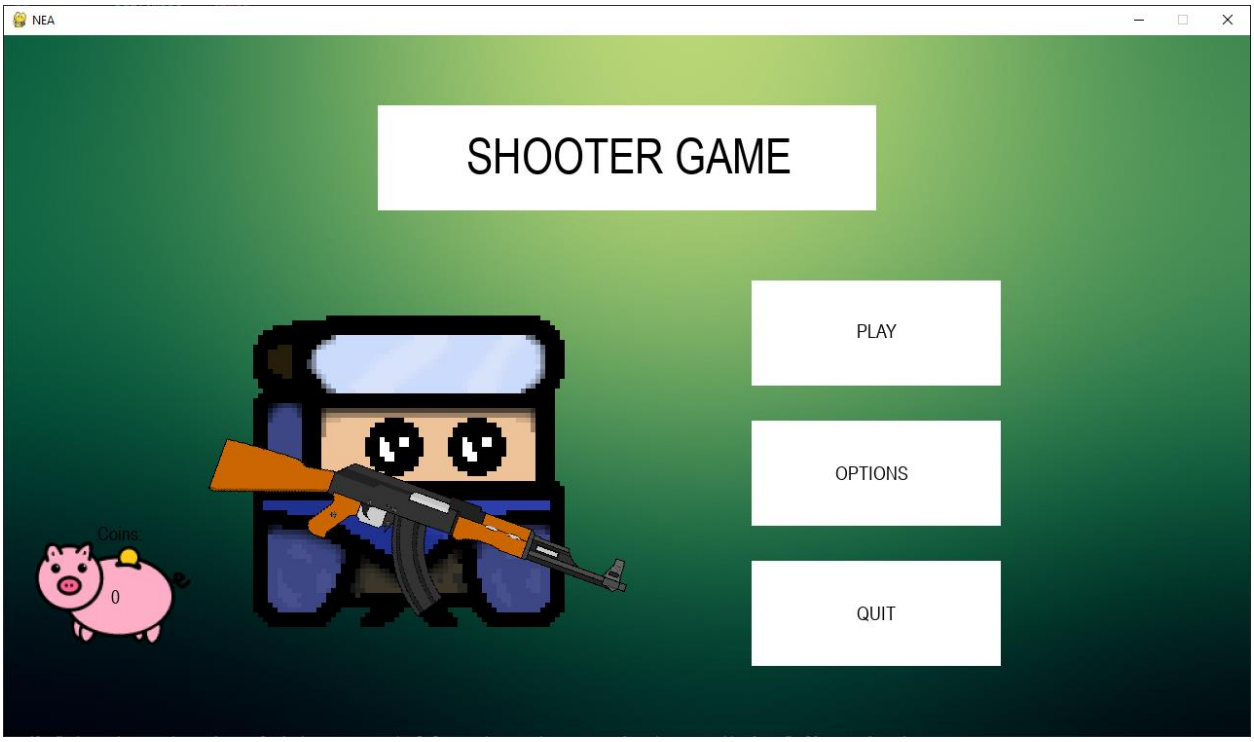
Coins are technically not part of the menu, but since they are so closely linked to cosmetics which are, I will be implementing them here.

I already put the player's coin count inside save.json, so I can use it straight from there. First, I opened it within the main_menu method, and then let it be displayed on the screen there:

```
def main_menu(self):

    # Get save.json for coins
    with open("save.json", "r") as read_save:
        save = json.load(read_save)
```

```
        # Piggy bank
        WINDOW.blit(piggy_img, (64, 520))
        WINDOW.blit(smallfont.render("Coins:", True, BLACK), (64, 432))
        WINDOW.blit(mediumfont.render(str(save["coins"]), True, BLACK), (64, 450))
```

This displayed this little image on the screen, along with my coin count:



Now I needed a way for these coins to be earned in the game. First I initialized self.coins to be 0 at the start of the main_game method, and created an empty list called coins to hold all coin objects currently existing on the map. Next it was time to create the class for the coins, so I did just that:

```
class Coin():
    def __init__(self, x, y, set_coin_img):
        self.x = x
        self.y = y
        self.coin_img = set_coin_img

    def main(self, WINDOW, camera_scroll):
        WINDOW.blit(self.coin_img, (self.x - camera_scroll[0], self.y - camera_scroll[1]))

    def hitbox(self, WINDOW, camera_scroll):
        if HITBOXES:
            pygame.draw.rect(WINDOW, (255,0,0), (self.x - camera_scroll[0] - (self.coin_img.get_width() / 2),
                                                 self.y - camera_scroll[1] - (self.coin_img.get_height() / 2),
                                                 self.coin_img.get_width(), self.coin_img.get_height()), 1)

        return pygame.Rect(self.x - camera_scroll[0] - (self.coin_img.get_width() / 2), self.y - camera_scroll[1] - (self.coin_img.get_height() / 2), self.coin_img.get_width(), self.coin_img.get_height())
```

I also created a coin image which I imported through config.py, which you can see being used here.

Next, I went into main.py and appended coins to the coins list every time an enemy died, passing in the coordinates of the enemy so we know where to spawn the coins. Here, I wrote for 4 to spawn.

```
for enemy in enemies:
    # If enemy health goes down to 0, add score to player's score count, delete the enemy, and drop coins
    if enemy.get_health() <= 0:
        self.score_count += settings[self.difficulty]["enemy"][enemy.identify()]["score"]
        for i in range(4):
            coins.append(Coin(enemy.get_xy("x", camera_scroll), enemy.get_xy("y", camera_scroll)))
        enemies.remove(enemy)

    enemy.main(WINDOW, camera_scroll)
```

Finally, I ran all the main methods of the coins currently existing on the map, so they will be blitted to the screen.

```
# Run the main for all existing coins on the map
for coin in coins:
    coin.main(WINDOW, camera_scroll)
```

Running the code, the coins were successfully getting spawned where they needed to be, but there was a problem:

The coins were all spawning at the exact same spot on the screen! To fix this, I needed to add some random variation to the coin's spawning, which I did below using the random function:

```
for enemy in enemies:
    # If enemy health goes down to 0, add score to player's score count, delete the enemy, and drop coins
    if enemy.get_health() <= 0:
        self.score_count += settings[self.difficulty]["enemy"][enemy.identify()]["score"]
        # Let each coin have some random variation to their spawnings so they don't all overlap each other
        for i in range(4):
            coin_x = int(enemy.get_xy("x", camera_scroll) + camera_scroll[0])
            coin_x = random.randint(coin_x - 50, coin_x + 50)
            coin_y = int(enemy.get_xy("y", camera_scroll) + camera_scroll[1])
            coin_y = random.randint(coin_y - 50, coin_y + 50)
            coins.append(Coin(coin_x, coin_y))
        enemies.remove(enemy)
```

This allowed the coins to spawn a little dispersed apart, which also looked quite cool:

Next, I added the ability for the player to collide with the coins, picking them up and incrementing the coin count. I also added a way for the coin count to be displayed on the screen.

```
# Run the main for all existing coins on the map
for coin in coins:
    coin.main(WINDOW, camera_scroll)
    # If player collides with coin, increment coin count
    if coin.hitbox(WINDOW, camera_scroll).collidepoint(player.hitbox(WINDOW)):
        coins.remove(coin)
        self.coins += 1
```

```
# Displaying coin count
WINDOW.blit(largefont.render(str(self.coins), True, BLACK), (40, 580))
```

This code loops over every coin existing on the map (in the list), running them, and if it collides with the player, then it removes it from the list so the next time it loops over it won't run it again, effectively deleting it from existence.

Running the code, the coins successfully blitted onto the screen every time I killed an enemy, and when the player goes to pick them up, they disappear and the coin count increments!

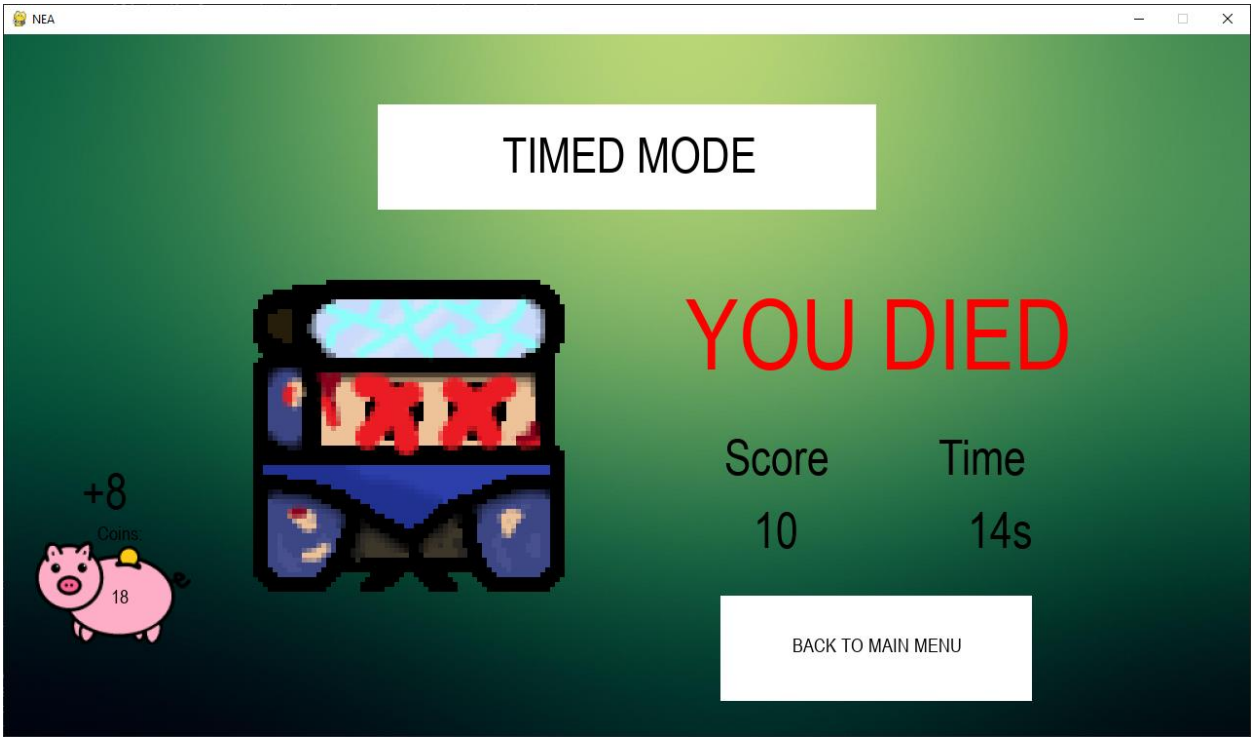
Now I need to display the coins collected in the game over screen. For this, I simply added this code to the game_over method:

```
with open("save.json", "r") as update_save:
    save = json.load(update_save)
    save["coins"] += self.coins
with open("save.json", "w") as add_coins:
    add_coins.write(json.dumps(save))
```

```
# Piggy bank
WINDOW.blit(piggy_img, (32, 520))
WINDOW.blit(smallfont.render("Coins:", True, BLACK), (96, 500))
WINDOW.blit(smallfont.render(str(save["coins"]), True, BLACK), (110, 565))
WINDOW.blit(mediumfont.render("+" + str(self.coins), True, BLACK), (80, 440))
```

This code simply updated the save.json file with the extra coins, and then showed the player their current coin count along with how many they got this game.

Running the code, collecting some coins, and dying showed this screen, proving that the above code works!



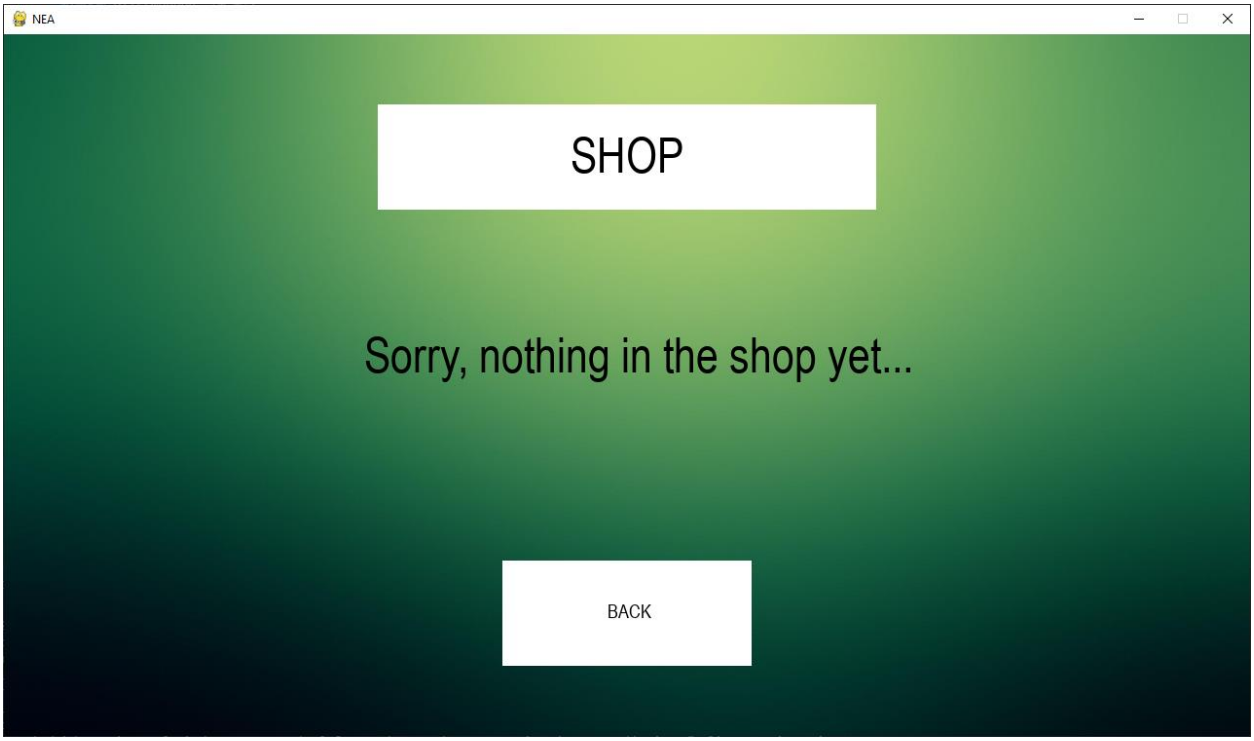
With the basic coin functions created, now I can begin to create the cosmetics shop! First, I went into the options menu and added this code, to create the shop button, as well as adding a new shop method:

```
# shop
shop_button = pygame.draw.rect(WINDOW, WHITE, (512, 252, 256, 108))
WINDOW.blit(smallfont.render("SHOP", True, BLACK), ((620, 292)))
```

```
# Checking for clicks on each button and changing the state accordingly
if shop_button.collidepoint(cursor):
    if event.type == MOUSEBUTTONDOWN:
        self.state = "shop_menu"
        return
```

For now, I will not put anything in the shop because cosmetics were decided as a desirable feature in my analysis section, but if time permits by the end I will certainly try and fill up the shop with different skins and other cool items!

So, for now, when you click on the options menu and go on the shop, this comes up:



Enemy Spawning

One important aspect of the game we have neglected up till now for testing purposes is natural enemy spawning. This entire time, every time I needed enemies to spawn, I used a bit of code to allow me to manually spawn them in using key presses, but this will not do for the actual game, since I don't want the player to have to spawn in the enemies.

A consideration I made for balancing of the game difficulties was how many enemies spawned for the player. However, since I have done a lot of balancing already for the enemy/player health

among other characteristics, I have decided to make the number of enemies that spawn for each game difficulty the same.

I want the enemy spawning to have wave-like characteristics, i.e. they spawn in batches between predetermined intervals. To do this, I simply utilized the timer that I already had in my code:

```
# Every {enemy_wave_period} seconds, spawn some random number of enemies, with a random number of basic and shooter enemies
if self.timer % enemy_wave_period == 1:
    if random.randint(0, enemy_num_chance) == 0:
        if random.randint(0,1) == 0:
            enemies.append(BasicEnemy(speed=basic_enemy_speed, health=basic_enemy_health, damage=basic_enemy_damage, damage_speed=enemy_damage_speed, sprite_img=enemy_sprite))
        else:
            enemies.append(ShooterEnemy(speed=shooter_enemy_speed, health=shooter_enemy_health, damage=shooter_enemy_damage, damage_speed=enemy_damage_speed, projectile_freq=shooter_enemy_proj_freq, sprite_img=shooter_sprite))

# Display how many enemies are currently on the map
WINDOW.blit(mediumfont.render("Enemies: " + str(len(enemies)), True, BLACK), (540,20))
```

You can see that I have decided to let the enemies spawn every “enemy_wave_period” seconds, and the number that is spawned is random. I set these variable values in config.py to be 10 seconds and 10 chance respectively. You can also see that I am also displaying the number of enemies currently on the map as well.

Running the code, the enemies successfully spawn in every 10 seconds, with a nice even spread between basic and shooter enemies, and a nice number of enemies spawning, providing a nice challenge for the player!

Pause Menu

The final thing I wanted to sort out for the settings was the pause menu. This would be accessed during the main game, when the player pressed a certain key such as Esc, which would pause the game and bring up a new menu, allowing the player to quit or go back to the game.

First, I went into the main_game method and created a pause variable, which declared the paused/unpaused state the game would be in, and initialized it to unpaused. Then I indented the whole main game loop code and put it under an if statement "if not paused". This would mean that if the game would pause, the entire game would effectively stop. To enter the menu, the player would need to press the escape key, which would turn paused to True.

```
paused = False

# Main loop
while True:
    pygame.display.update()

    if not paused:
```

```
# If player presses escape, pause the game
if event.type == pygame.KEYUP:
    if event.key == K_ESCAPE:
        paused = True
```

Then in the else part of the if statement, I created my menu:

```
# If player paused the game by pressing ESC
else:
    translucent = pygame.Surface((window_width, window_height), pygame.SRCALPHA)
    translucent.fill((0,0,0,1))
    WINDOW.blit(translucent, (0,0))

    # Pause menu card
    pause_menu = pygame.draw.rect(WINDOW, BLACK, (int(window_width*0.35), int(window_height*0.3), int(window_width*0.3), int(window_height*0.4)))
    pause_title_card = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.1), int(pause_menu.top + pause_menu.height*0.1), int(pause_menu.width*0.8), int(pause_menu.height*0.3)))
    pause_text = mediumfont.render("PAUSED", True, BLACK)
    WINDOW.blit(pause_text, (pause_title_card.centerx - pause_text.get_width()/2, pause_title_card.centery - pause_text.get_height()/2))

    # Quit button
    quit_button = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.2), int(pause_menu.top + pause_menu.height*0.45), int(pause_menu.width*0.6), int(pause_menu.height*0.2)))
    quit_text = smallfont.render("QUIT GAME", True, BLACK)
    quit_text_disclaimer = tinyfont.render("Current game data will be lost!", True, BLACK)
    WINDOW.blit(quit_text, (quit_button.centerx - quit_text.get_width()/2, quit_button.centery - quit_text.get_height()/1.5))
    WINDOW.blit(quit_text_disclaimer, (quit_button.centerx - quit_text_disclaimer.get_width()/2, quit_button.centery + quit_text_disclaimer.get_height()/3))

    # Continue button
    cont_button = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.2), int(pause_menu.top + pause_menu.height*0.7), int(pause_menu.width*0.6), int(pause_menu.height*0.2)))
    cont_text = smallfont.render("CONTINUE GAME", True, BLACK)
    WINDOW.blit(cont_text, (cont_button.centerx - cont_text.get_width()/2, cont_button.centery - cont_text.get_height()/2))

    # Checking for events
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        # Checking for cursor coordinates
        cursor = pygame.mouse.get_pos()

        # If player presses escape, unpause the game
        if event.type == pygame.KEYUP:
            if event.key == K_ESCAPE:
                paused = False

        # Checking for clicks on each button and changing the state accordingly
        if quit_button.collidepoint(cursor):
            if event.type == MOUSEBUTTONDOWN:
                self.state = "main_menu"
                return
        if cont_button.collidepoint(cursor):
            if event.type == MOUSEBUTTONDOWN:
                paused = False
```

Running the code and testing the pause menu, the following screen popped up, and clicking the buttons successfully either brought me back into the game or went to the main menu!

UI Cleanup

This marks the end of the implementation of the general functionality of the game; we have successfully implemented every necessary feature for the program to be fully playable. However, as you will have noticed from the screenshots, the UI is not very nice to look at right now. In the game, none of the numbers are clear and are frankly ugly, and the menus look extremely bland. On top of improving these, I also will make some improvements to the sprite images, since some colour schemes do not work on top of each other, particularly the different map backgrounds for the themes.

A significant problem that I realized at this stage was the fact that all my layouts and sizing was done statically, i.e. I had given them fixed coordinates based on my current screen resolution. This would mean that if the user needed a different screen resolution, the game layout would completely break. So, I first needed to fix this issue.

To fix this, I simply went into all my coordinate declaration lines of code and replaced them to be dependent on my screen width/height variables.

Before I did this, I decided to set an aspect ratio variable in config.py, to make sure that the width and length of the screen were fixed proportion to avoid any extreme layout problems caused by disproportionate screen dimensions:

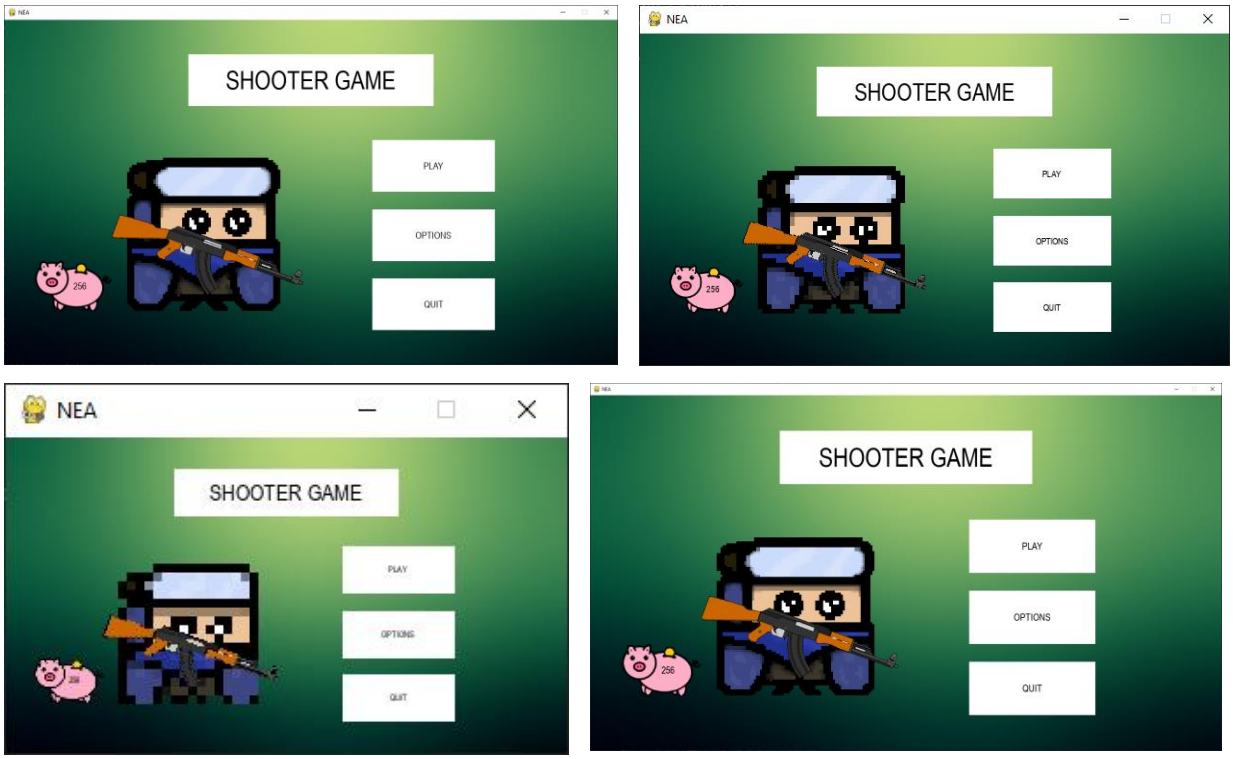
```
aspect_ratio = [16,9]
pixel_factor = 80
window_width, window_height = aspect_ratio[0] * pixel_factor, aspect_ratio[1] * pixel_factor
```

Pixel factor is simply a variable to decide how large we want the screen to be compared to the base ratio, which I set to 80 for now.

Now I simply went into each method and altered the object blitting code coordinates to use the window_width and window_height variables, so that if we changed pixel_factor the UI would resize with it (the following code is just a small snippet as an example of what I did to the rest of the code):

```
# Player and gun
player_menu_icon = WINDOW.blit(player_menu_img, (int(window_width*0.2), int(window_height*0.4)))
WINDOW.blit(pygame.transform.rotate(gun_img, -20), (player_menu_icon.centerx - player_menu_icon.width/1.3, player_menu_icon.centery - player_menu_icon.height/2.2))

# Piggy bank
piggy_icon = WINDOW.blit(piggy_img, (int(window_width*0.05), int(window_height*0.7)))
coins_text = smallfont.render(str(save["coins"]), True, BLACK)
WINDOW.blit(coins_text, (piggy_icon.centerx, piggy_icon.centery - coins_text.get_height()/2))

# title
title_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int(window_height*0.15)))
title_text = mediumfont.render("SHOOTER GAME", True, BLACK)
WINDOW.blit(title_text, (title_card.centerx - title_text.get_width()/2, title_card.centery - title_text.get_height()/2))
```

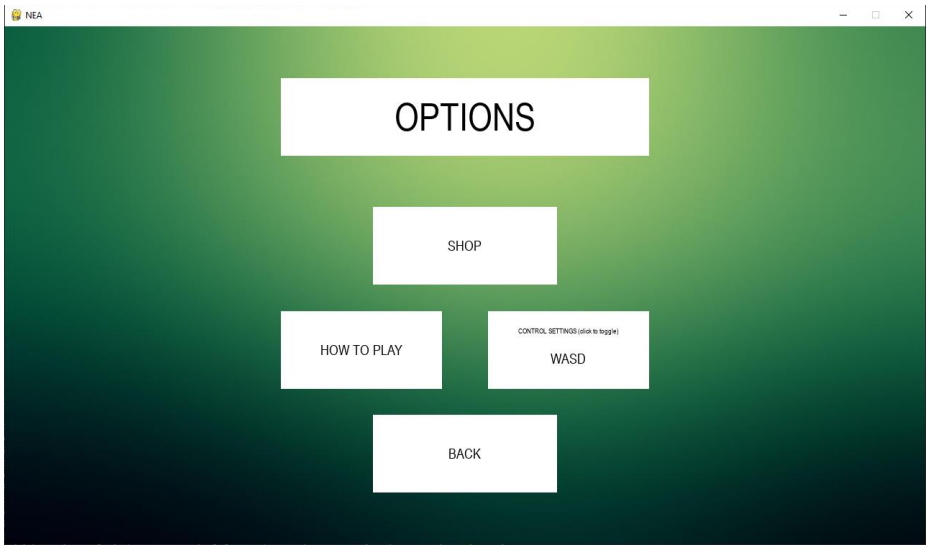
Now, when I vary pixel_factor, the objects on the screen resize along with it! The following screenshots are of pixel factor 80, 40, 20 and 100 respectively (left to right, top to bottom) which you can tell by using the size of the bar at the top.



With the UI positioning sorted, I could now move on to improving the sprite images and the general look of the game.

I also needed to design the UI to fit my accessibility features laid out in my design section. For this, I needed to use a clear, large enough font (preferably sans-serif for dyslexic users), and clearly understandable controls and GUI features. For this, I also created a tutorial menu within the options menu which would use the same arrow control layout as the theme's menu (the player can switch back and forth between pages that explain how to play the game), as well as control editing options so that the player can toggle between WASD and arrow keys (a desirable

feature) depending on which side they prefer to play on. I also made sure since the start that the font I used was sans-serif.



The health texts above the sprites are also quite difficult to read. I fixed this issue by designing a health bar that would be displayed above all the sprites, and using some proportionality mathematics, display the amount of health the player has using a bar that gets shorter the less health the sprite has:

```
# health bar coordinates
self.health_bar_x = centre_x - int(health_bar_length/2)
self.health_bar_y = self.y - health_bar_height - 2*health_bar_border
# drawing on health bar, calculating length depending on proportion between max health and current health
health_bar_circle = pygame.draw.circle(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y + int(health_bar_height/2)), int(health_bar_height*0.8))
self.health_px = int((self.health/self.max_health) * (health_bar_length - int(health_bar_circle.width/2)))
pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y, health_bar_length, health_bar_height))
pygame.draw.rect(WINDOW, GREEN, (self.health_bar_x + int(health_bar_circle.width/2), self.health_bar_y, self.health_px, health_bar_height))
pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y, health_bar_length, health_bar_height), health_bar_border)
# health bar text to display numerical health value
health_text = tinyfont.render(str(self.health), True, WHITE)
WINDOW.blit(health_text, (health_bar_circle.centerx - health_text.get_width()/2, health_bar_circle.centery - health_text.get_height()/2))
```

Applying this code to all the sprites that have a health attribute and running the game showed that the health bar was successfully implemented, and the bar's length varied depending on the sprite's current health:

Finally, I added solid backgrounds to the text of the UI (i.e. the timer, the score count, etc.) so that they stood out in front of the maps, especially the darker ones such as city or volcano:

Music and Sound Effects

In my analysis section, I put down music and sound effects as an essential feature, because I decided that it was extremely important to allow me to create an immersive playing experience. However, the reason I left it for last is because once my entire game is complete, it will be extremely easy to import all the sound files I need and insert them into the code at the desired points in the program.

Here is a general list of the sound effects I will be importing:

- Lobby (menu) background music
- Menu button clicking sounds
- Gunshot sounds
- Footsteps for each game theme
- Player sounds (e.g. getting hurt, dying, etc.)
- Victory/Loss sounds
- Ambient background sounds during game
- Enemy sounds (e.g. growling, getting hurt, etc.)
- Powerup sound effects (e.g. picking up, theme sound, etc.)

The general sounds that were either looping as background music or triggered by actions such as clicking were very simple: All I had to do was import the sound in config.py, and in main.py, whenever I have a checking for action statement, I play the sound.

```
# Music and Sounds
pygame.mixer.init()
lobby_music = pygame.mixer.music.load("assets/lobby_music.wav")
pygame.mixer.music.set_volume(0.6)

gun_sound = pygame.mixer.Sound("assets/gunshot.wav")
```

```
# Checking if the player clicked mouse
if event.type == pygame.MOUSEBUTTONDOWN:
    if event.button == 1:
        # If so, append a projectile object to the list
        projectiles.append(PlayerProjectile(cursor_x, cursor_y, speed=player_proj_speed,
        gun_sound.play()
```

```
if back_button.collidepoint(cursor):
    if event.type == MOUSEBUTTONDOWN:
        guncock_sound.play()
        self.state = "dif_mode_menu"
        return
```

However, sounds that required other triggers such as ongoing powerup background sounds were a bit trickier. Since everything was within a game loop, I couldn't simply write "if powerup active: play sound, else: stop sound"; this would have caused a new instance of the sound to play every frame, instantly bombarding the user with earrape.

To avoid this, I decided to check for a powerup being activated and deactivated by utilizing the powerups_status dictionary's timer key:

```
# If powerup is active, blit the shield to the screen
if powerups_status["shield"]["active"]:
    WINDOW.blit(shield_bubble, (int(window_width*0.5) - shield_bubble.get_width()/2, int(window_height*0.5) - shield_bubble.get_height()/2))
# Start to play the unshield sound 1 second before it runs out, and stop the shield background sound right before it runs out
if powerups_status["shield"]["timer"] == powerup_duration*FPS - FPS:
    unshield_sound.play()
if powerups_status["shield"]["timer"] == powerup_duration*FPS - 1:
    shield_sound.stop()
```

This successfully allowed me to play ongoing background sounds while a powerup was activated. Then to start the sounds I looped it over after the player picks it up:

```
# If the player collides with the powerup, remove it and make it active
if powerup.hitbox(WINDOW, camera_scroll).colliderect(player.hitbox(WINDOW)):
    powerups.remove(powerup)
    if powerup.identify() != "health":
        powerups_status[powerup.identify()]["active"] = True
        powerups_status[powerup.identify()]["timer"] = 0

        if powerup.identify() == "shield":
            shield_sound.stop()
            shield_sound.play(-1, fade_ms=1000)
    else:
        powerup.health(player, powerup_health_boost)
        health_sound.play()
```

Now when I equip the shield powerup, it begins looping the shield background sound, and right before it finishes, the unshield sound effect plays before the shield background sound stops! I then went ahead and implemented similar procedures for the other powerups.

In the end, I managed to apply sound effects and music to every point on the list above!

Wrapping Up

With pretty much every essential feature detailed in my analysis section complete, I can confidently sit back and confirm that the implementation stage of my solution is complete. Of course, evaluative testing will probably end up flagging up several errors that will need to be fixed, and actual player testing will also have to take place so that I can balance the game settings to better suit my client's needs, so this is not the last I am seeing of my program code. In future, I may also want to come back and improve on it further, such as adding to the shop or creating new game modes, enemy types, etc. However, the bulk of the programming is complete, and I am extremely satisfied with the result. It came out exactly how I intended it to, achieving every essential feature and almost every desirable feature, and hopefully my client will also be pleased with the result.

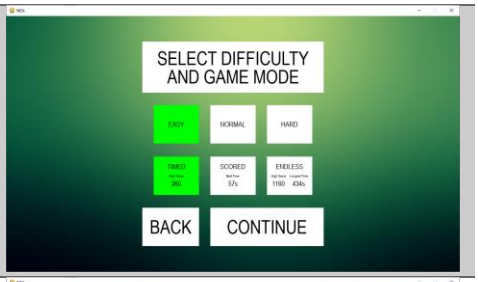
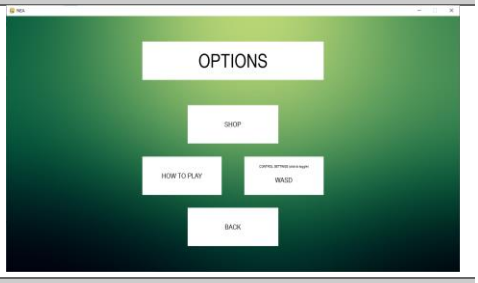
END OF IMPLEMENTATION

Evaluation

Evaluative Testing

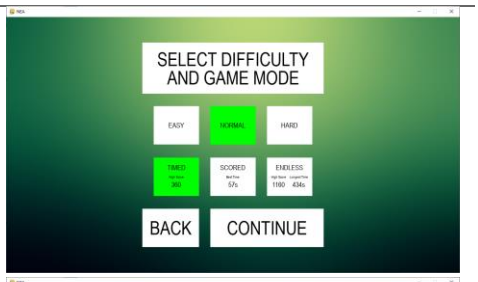
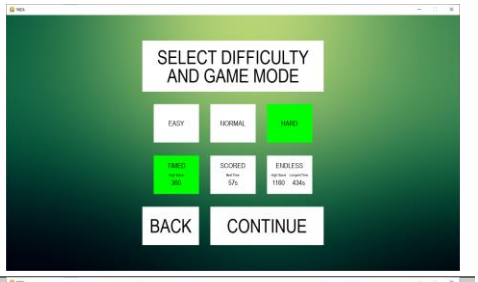
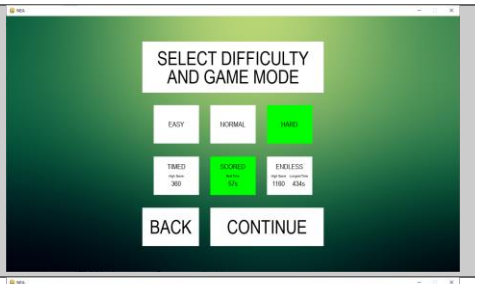
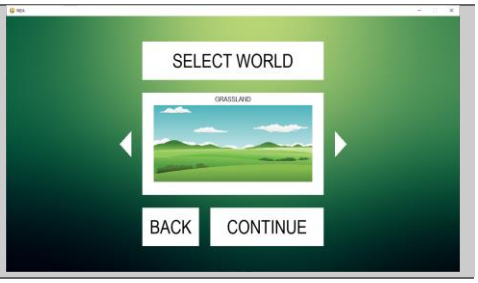
Now that I have fully implemented my game, I can begin evaluative testing (as opposed to iterative testing to inform development, which is testing while implementing), which is where I take the test plan I created in the design section and test each part on my program, documenting results and changing things if needed.

Main Menu

Action	Input	Expected Result	Actual Result
Open program	Normal	Game opens onto main menu with correct screen resolution.	Program opens in desired resolution: 
Click blank space	Erroneous	Nothing.	Nothing.
Click play	Normal	Options menu to select difficulty and game modes.	
Click settings	Normal	Settings menu to change game settings.	
Click quit game	Normal	Program exits.	Program quits to desktop.

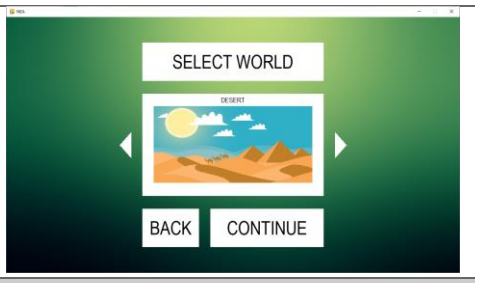
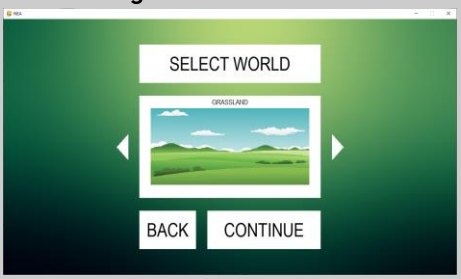
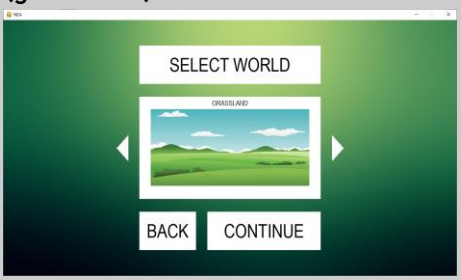
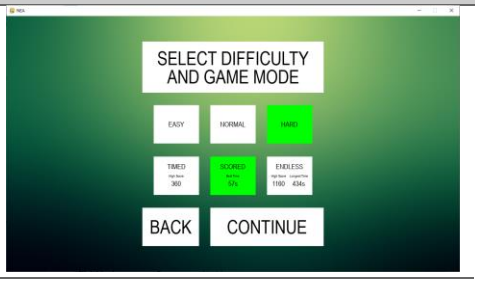
Difficulty/Mode Menu

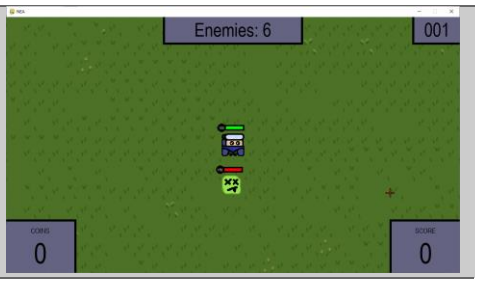
Action	Input	Expected Result	Actual Result
Click blank space	Erroneous	Nothing.	Nothing (clicking on all menu buttons plays a sound)

Click easy/normal/ hard difficulty	Normal	Toggles between easy, normal and hard buttons, not affecting mode selection.	 
Click timed/scored/ endless mode	Normal	Toggles between timed, scored and endless buttons, not affecting difficulty selection.	
Click back	Normal	Main Menu.	
Click continue	Normal	Theme menu(?) to select theme, game will have currently selected options in gameplay.	

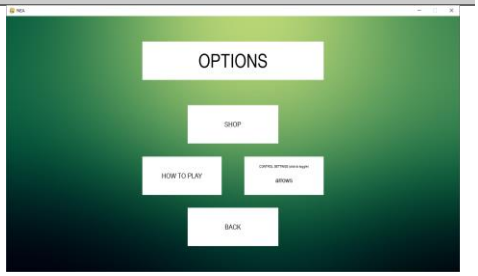
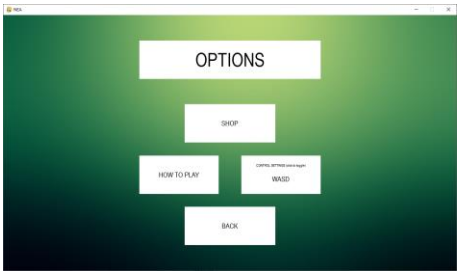
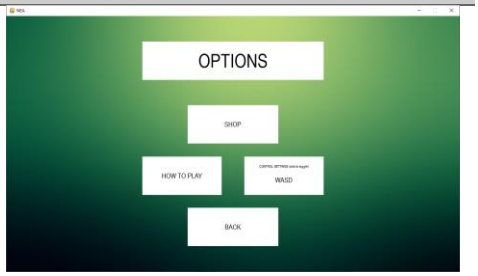
Theme Menu

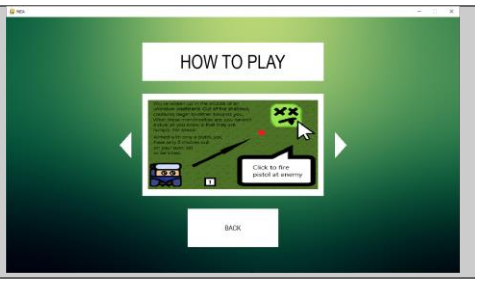
Action	Input	Expected Result	Actual Result
Click blank space	Erroneous	Nothing.	Nothing.
Click right arrow	Normal	Current theme increments to next in list.	List is grassland – desert – city – volcano, so clicking here takes you from grassland to desert.

			
Click left arrow	Normal	Current theme decrements to previous in list.	Clicking from here takes you from desert to grassland. 
Click left arrow when selecting theme 1	Boundary	Current theme loops to last theme.	Clicking from first one (grassland) takes you to the last one (volcano). 
Click right arrow when selecting last theme	Boundary	Current theme loops to theme 1.	Clicking from last one (volcano) takes you to the first one (grassland). 
Click back	Normal	Options Menu.	

Click continue	Normal	Enter Game, game will have selected difficulty, mode, and theme in gameplay.	

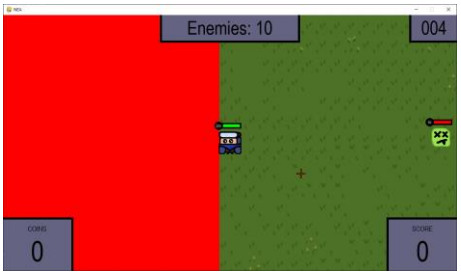
Options Menu

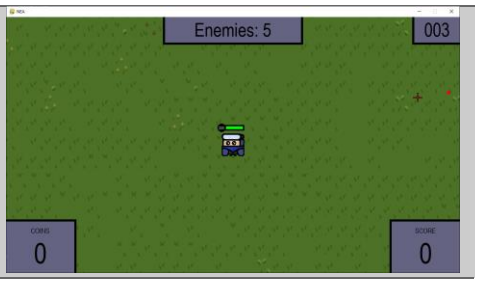
Action	Input	Expected Result	Actual Result
Click blank space	Erroneous	Nothing.	Nothing
Click controls	Normal	Toggles WASD to arrows or arrows to WASD.	 
Click shop	Normal	Open shop.	Nothing in the shop yet (desirable feature) but can design and add skins into the game in the future. 
Click back (shop)	Normal	Options Menu.	

Click how to play	Normal	How to Play Menu.	
How to Play functionality	Normal	Arrows and slides behave exactly like Theme Menu.	Clicking the arrows causes the slides to behave exactly like the theme menu: left arrow switches slide to previous (or last if currently on first), right arrow switches slide to next (or first if currently on last).  
Click back	Normal	Main Menu, settings updated.	

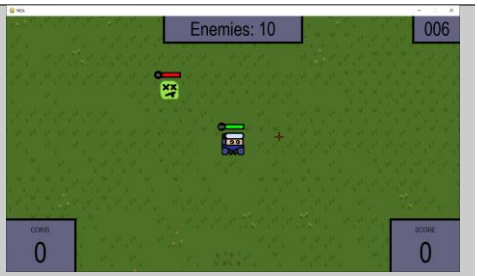
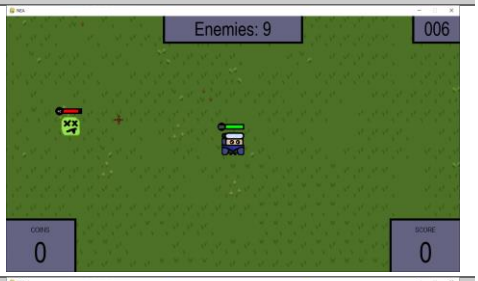
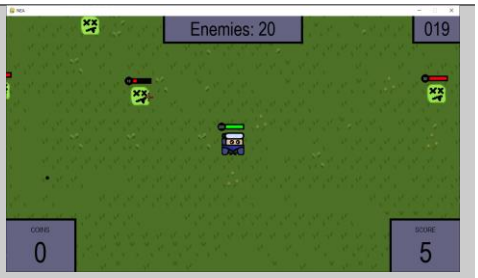
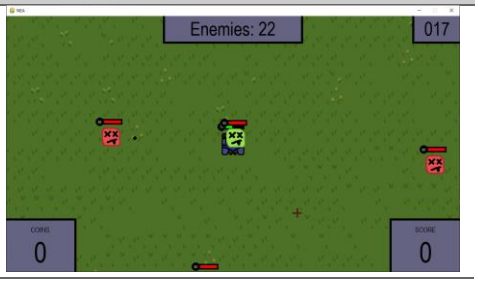
General Game

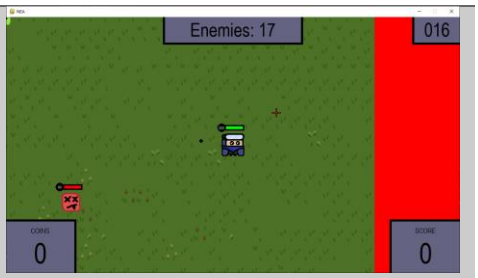
Action	Input	Expected Result	Actual Result
Start	Normal	Map, player at center of screen, score count at 0. Correct sprite designs, difficulty (attributes accurate for	Correct theme, mode and difficulty applied (checked via console), correct sounds playing, variables displaying on the screen correctly with the timer increasing every

		difficulty settings), and mode. Music is playing.	second, enemy count displaying the number of existing enemies on the map accurately. 
Press WASD/arrows with the correct control applied in settings.	Normal	Player moves up (W), down (S), left (A) and right (D) on map (or technically map moves in an inverted fashion to controls pressed) as footsteps are played.	Player correctly moves up, down, left, and right according to the keys the user presses. If the settings in options menu were on WASD, those are the keys that move the player, and likewise for arrows.
Press WASD/arrows with the wrong control applied in settings.	Erroneous	Nothing.	Nothing; if the settings applied are WASD, pressing arrows does not move the player, and if the settings applied are arrows, WASD does not move the player.
Hold keys until player reaches end of map	Boundary	Player stops at border of map, even if controls are still held down.	Player does not move further even with controls held down. 
Click mouse	Normal	Program calculates correct angle to spawn projectile, originating from player and moving in the direction of the point clicked.	Projectile correctly spawned and moving in the correct direction at the right speed.

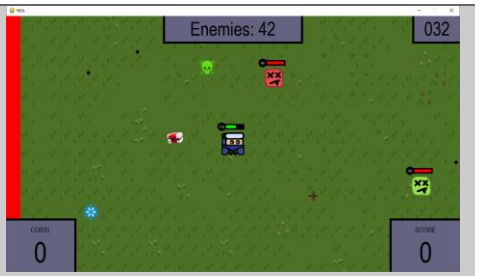
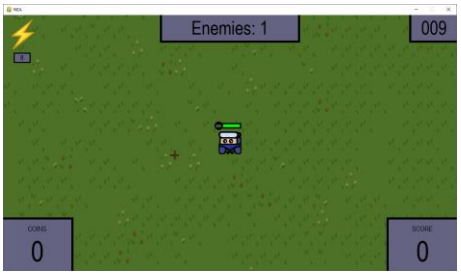
			

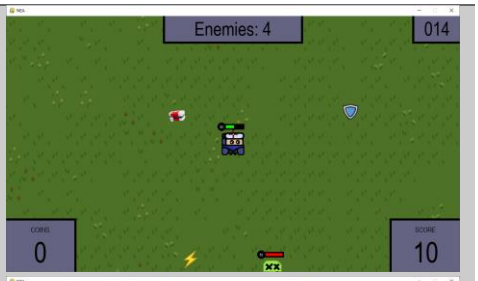
Enemy/Enemy Projectiles

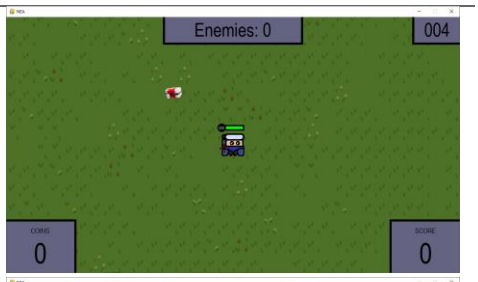
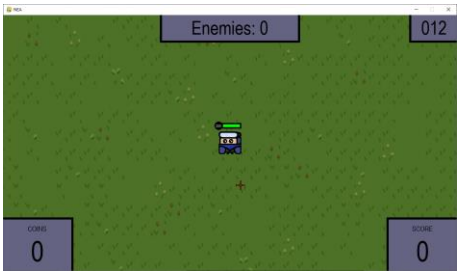
Action	Input	Expected Result	Actual Result
Enemy spawns	Normal	Enemy is placed on map at a random position, moving towards the player.	
Player projectile collides with enemy	Normal	Enemy health is depleted by relevant amount as per the selected difficulty.	
Player projectile collides with enemy with tiny health	Boundary	Enemy health goes below 0, is removed from map and score is added to player's score count. Perhaps drops coin.	 *Enemy dies in next shot
Player collides with enemy	Normal	Player health is depleted at regular intervals while the enemy is touching the player.	

Player collides with enemy projectile	Normal	Player health is depleted.	 *Health depletes in the next shot

Powerups

Action	Input	Expected Result	Actual Result
Powerup spawns	Normal	Random powerup icon is displayed on the map at a random position with probability.	
Player collides with powerup (activating it)	Normal	Powerup is removed from map, powerup effect is applied on player for designated amount of time if applicable. Icon is displayed in inventory and timer is initialized.	All powerups play their relevant sound effects when applied. Speed powerup: player moves faster.  Shield: player does not get damaged, shield overlay rendered.  Poison: player damages enemies, poison overlay rendered.

			 Freeze: all enemies freeze in place, freeze overlay rendered.  Below: All powerup effects applied! 
Health picked up	Normal	Player health increases.	 
Health picked up little under/at max health	Boundary	Player health increases to the max health cap, no further.	Player health goes from 200 to 200 when health effect is applied.

			 
Player collides with enemy/projectile with shield/poison power	Normal	Shield: Neither objects' health depletes. Poison: enemy health depletes at regular intervals, but player health depletes if collides with enemy projectile.	Shield: player does not get damaged, shield overlay rendered. Neither the player nor enemy depletes health.  Poison: player damages enemies, poison overlay rendered. Enemy depletes health between intervals, player does not deplete health. Player health still depletes upon contact with enemy projectile. 
Player picks up powerup while said powerup still active	Normal	Timer for that powerup refills to full time instead of powerup stacking in power.	Here, the player picks up a shield with a shield equipped. The timer refills to 10 seconds (screenshotted @ 8 seconds because I'm too slow).

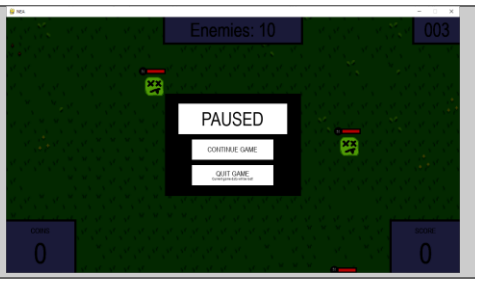
			 
Powerup timer ends	Normal	Icon is removed from inventory, powerup is deactivated in game.	Relevant sounds play for deactivation, icon removed from scree, powerup stops operating.

Coins

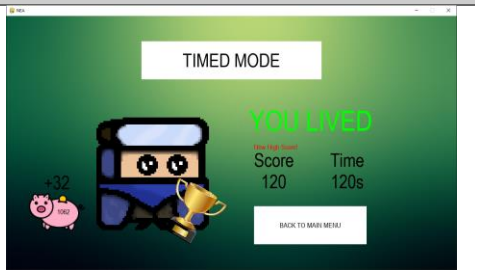
Action	Input	Expected Result	Actual Result
Coin spawns	Normal	Coin appears at random location on map either naturally or from enemy drop.	Coins spawn upon enemy death. 
Player collides with coin	Normal	Coin removed from map; player coin count incremented.	

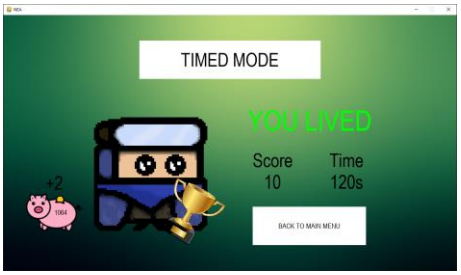
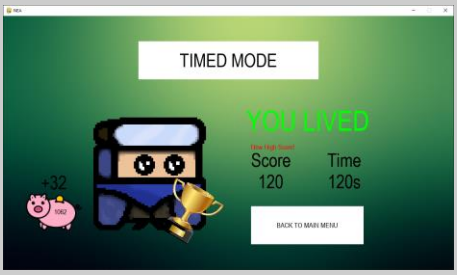
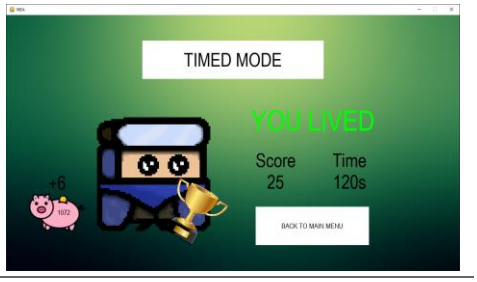
End of Game

Action	Input	Expected Result	Actual Result

ESC button pressed	Normal	Game paused, open pause menu.	
Click blank space in pause menu	Erroneous	Nothing.	Nothing.
Quit button pressed in pause menu	Normal	Game ends, score/time is not logged, back to main menu.	
Continue button pressed in pause menu	Normal		Exit pause menu, game continues.
Player health <= 0	Normal	Player dies, game ends, score/time logged, coins added to bank, sent to game over menu.	

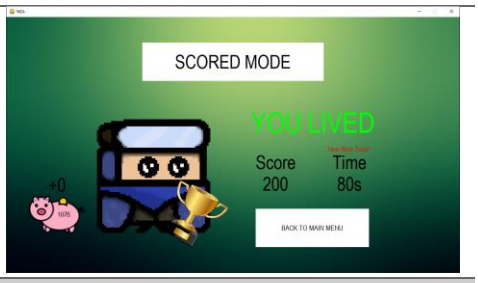
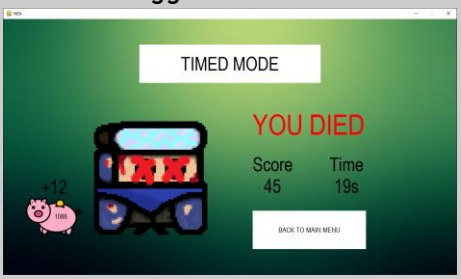
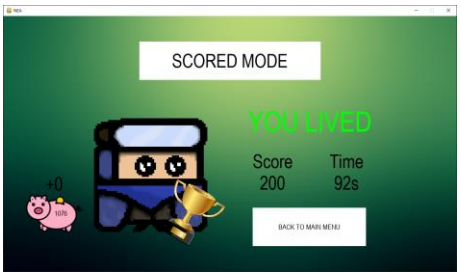
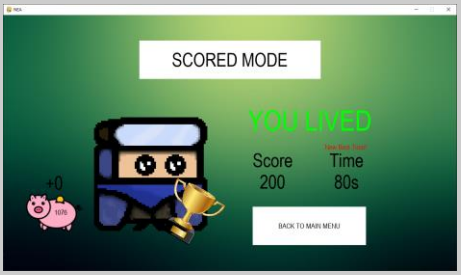
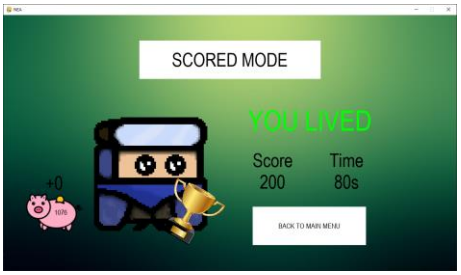
Timed mode

Action	Input	Expected Result	Actual Result
Start	Normal	Timer initialized to 0.	Timer initialized to 0.
Timer reaches chosen max time	Normal	Game ends, score is recorded if high score.	
Player dies	Normal	Score is not considered for high score.	Old high score is 25, new score of 35 is not flagged as a high score.

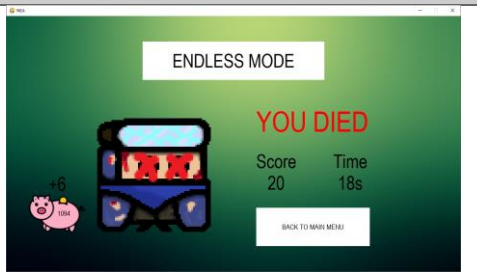
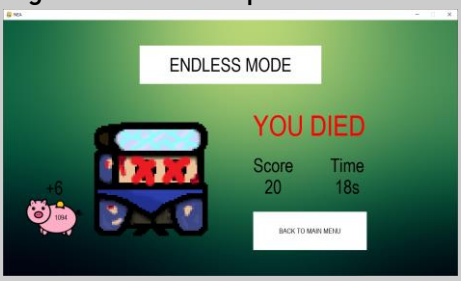
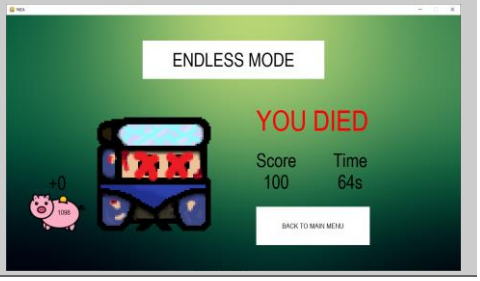
			
Score < high score	Normal	High score is not updated.	Old high score was 25, 10<25 so not recorded. 
Score > high score	Normal	High score is updated with new score.	Old high score was 25, 120>25 so recorded. 
Score = high score	Boundary	High score is not updated.	Old high score was 25, 25=25 so not recorded. 

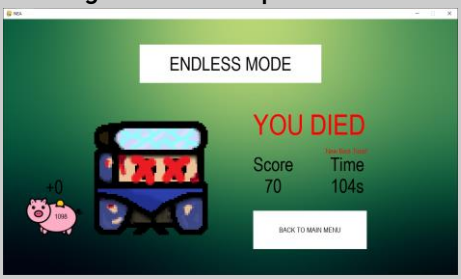
Scored mode

Action	Input	Expected Result	Actual Result
Start	Normal	Timer initialized to 0.	Timer initialized to 0.

Score reaches chosen max score	Normal	Game ends, final time is recorded.	
Player dies	Normal	Time is not considered for best time.	Old best time was 80, new time of 19 is not flagged as a best time. 
Time > best time	Normal	Best time is not updated.	Old best time was 80, 90>80 so not recorded. 
Time < best time	Normal	Best time is updated with new time.	Old best time was null, 80>null so recorded. 
Time = best time	Boundary	Best time is not updated.	Old best time was 80, 80=80 so not recorded. 

Endless mode

Action	Input	Expected Result	Actual Result
Start	Normal	Timer initialized to 0.	Timer initialized to 0
Player dies	Normal	Game ends, score and time are recorded.	
Score < high score	Normal	High score is not updated.	Old high score was 100, 20<100 so high score is not updated. 
Score > high score	Normal	High score is updated with new score.	Old high score was 100, 120>100 so high score is updated. 
Score = high score	Boundary	High score is not updated.	Old high score was 100, 100=100 so high score is not updated. 
Time < longest time	Normal	Longest time is not updated.	Old longest time was 88, 18<88 so longest time is not updated.

			
Time > longest time	Normal	Longest time is updated with new time.	Old longest time was 88, 104>88 so longest time is updated. 
Time = longest time	Boundary	Longest time is not updated.	Old longest time was 88, 88=88 so longest time is not updated. 

Game Over Menu

Action	Input	Expected Result	Actual Result
Player dies	Normal	Menu opens, score/time and coins collected is displayed, told if surpassed high score/best time.	See tests above for timed, scored and endless modes. Checking save.json shows the new data successfully written in.
Click blank space	Erroneous	Nothing.	Nothing.
Click Continue	Normal	Back to main menu, high score/time is saved to an external file if applicable.	

Click Restart	Normal	Game begins again, score and time reset to 0.	Game restarts.

As you can see, every aspect of the test plan has succeeded. Now we can move on to the assessment of the success criteria outlined in the analysis section.

Testing Against Success Criteria

Essential Features

Criteria	Achieved?	Comments
A graphical 2D user interface with the player fixed in the center of the screen, the environment scrolling around it, including moving enemies, obstacles, and projectiles.	Yes	See General Game.
A game menu that allows for selection of game difficulty, game mode, etc.	Yes	See Difficulty/Mode Menu.
Power-ups that the player can equip to aid their in-game progress.	Yes	See Powerups. I ended up creating 5 powerups, which I am very satisfied with and is thoroughly complex.
A variety of enemy types, each with their own sprite design, attributes, powers, etc.	Yes	See Enemy/Enemy Projectiles. I ended up making 2 enemy types, even though I originally gave myself a limit of 5, but in the end time constraints forced me down to 2: a basic enemy and a shooter enemy. In future I can implement new varieties by adding new classes.
The player has health attributes and an inventory to display power-ups.	Yes	See Powerups. The health attribute is displayed in a well-designed health bar.
Time and score-controlled game modes, both should last roughly 2-3 minutes long. If the player dies, the game ends and the final score/time is logged, along with the fact that the player cut the game short with untimely death. And an endless mode, with no constraints and the game continues until the player	Yes	See Timed/Scored/Endless Modes. All points in this criterion were successfully implemented exactly.

either dies or quits, and the final score is recorded.		
Easy, normal, hard difficulty levels.	Yes	See Difficulty/Mode Menu and General Game.
Sound effects for an immersive experience, e.g., gun shots, footsteps, enemy noises, etc. Music?	Yes	See General Game. In the end, I implemented menu click sounds, in-game sound effects such as player/enemy sounds and music/ambient background noise.
A score count in the game to keep track of game progress.	Yes	See General Game.

Desirable Features

Criteria	Achieved?	Notes
A reward system with unlockable cosmetics (like a battle pass), including different sprite skins that the user can choose to switch between before a game (these would not affect the gameplay, simply for aesthetic purposes).	Sort of	See Coins and Options Menu. I ended up implementing a cosmetics system through collecting coins, but I have not put anything in the shop. This could be done in future, it's just that since it's not crucial to the solution, I have skipped designing and creating extra skins for time constraint reasons.
Option to select game theme, e.g., color schemes, game environment, complex/abstract, etc.	Yes	See Theme Menu. In the end, I managed to create 4 different themes (grassland, desert, city and volcano) each with their own map, ambient background noise and enemy appearances.
Option for user to select between WASD and arrow controls in settings.	Yes	See Options Menu and General Game.
Optional camera shake?	No	I was unable to implement this feature due to time constraints and its high complexity.
Boss battles at regular intervals.	No	Likewise^

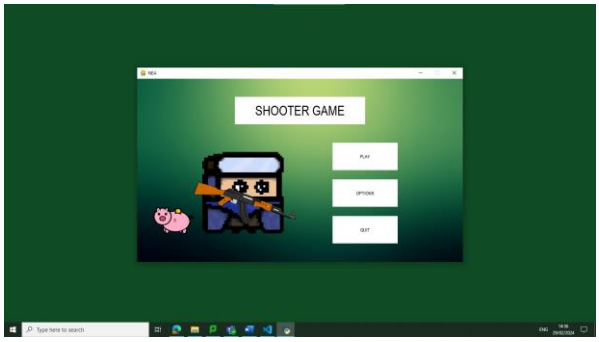
Overall, all essential features were implemented successfully, and even most of my desirable features, meaning that this solution was a success! In future, I can work on creating skins and cosmetic items to put in my shop which can be traded for using the coins earned in-game, which can make the game even more immersive and enjoyable.

Client Usability Testing

Now that I am confident that the program is successfully completed, I can send it to my client George to test on his machine and report back with his overall thoughts on the game. Since George also has Python and Pygame installed on his computer, there was no problem with booting up the program on it. However, it was at this moment I realized I had forgotten to account the screen resolution.

The resolution of my game had been controlled by a 2-element list called aspect_ratio and a variable called pixel_factor that I put in config.py, and at the time that I had sent the program to George, I had set these to [16,9] and 80 respectively. The program would take the aspect ratio, multiply each element by the pixel factor, and use the 2 values returned as the resolution of the program (in this case, 16*80=1280 and 9*80=720 which fits my 720p monitor).

The obvious problem here was that I did not create a way for the program to detect the current device's screen resolution and adapt accordingly. George has a 1080p desktop monitor, meaning the dimensions was 1920x1080 pixels. This was clearly a big difference from my 720p monitor that I had been using to test my code. So, George had to go into the program code himself and edit the pixel_factor to 120, so that it showed fullscreen for a 1920x1080 monitor:



Before: pixel_factor = 80

After: pixel_factor = 120

This was very inconvenient in my opinion, since even though George was able to spot this pretty quickly and fix the problem, I am worried that other clients who may not be versed in their monitor resolutions will have trouble navigating and modifying the screen resolution variables. Due to time constraints, I will not be able to implement an automatic screen adaptation function for now, but in future I can make sure that I take this into account.

Now that the game was fully configured for George's monitor, it was time for him to begin testing.

Final Thoughts

After George concluded his testing, to my greatest delight he reported back to me the same successful results that I found in my evaluative testing. He also left the following comments:

"This game has been the most fun experience I've had in a long time. Almost everything you have implemented is perfect, from the aesthetics of the menu to the complexity of the game structures. I loved playing every minute of it!

I was less pleased with several of the sprite designs and felt like the general aesthetics of the images could have been improved. The shop was also empty, which was quite disappointing. But the elephant in the room was the annoying fact that during the game, I had to constantly spam my mouse to shoot fast. This may have given me carpal tunnel. However other than that, the game was a huge success for me, and I look forward to playing it again!" - George

---

I am very satisfied with these comments. It was frustrating to hear about his carpal tunnel problem, but I can make sure future clients do not suffer the same consequences as him by implementing something along the lines of a continuous firing function which only requires the user to hold the mouse down and another trigger key to let the program know it's active. This would include a firing delay, e.g. firing a bullet every half a second, to limit the number of bullets going out and to prevent unfair advantage.

For future developments, I asked my client George as well as other clients in similar situations to George about what they would like to see in future releases. They told me that the main focus should be on developing new enemy types, which would be simple to implement thanks to my object-oriented approach that allows for inheritance and polymorphism of enemy classes to be done efficiently. This would also involve designing new sprite icons for these enemies, as well as adding more worlds which will also be very simple thanks to the way I designed the world selection system. George also confessed that he was looking forward to using his earned coins to buy some items in the shop but was disappointed to see the shop empty. Therefore, I would also create a variety of new player skins and cosmetics to fill up the shop, which players can use their coins to buy.

Maintainability is another concern I have with the program. The program depends on pygame, and right now whenever a user wants to play the game, they must ensure they are using the correct version of pygame: If in future pygame release new versions that cannot run old pygame programs, I will have to rewrite some of my program to allow it to be compatible with said newer versions. There is also the question of how convenient it is for a new user to play the game, since it requires the installation of both python and pygame on their machines, which may be inconvenient. However, as far as I or my prospective clients are concerned, this does not have much impact for them, so for them, this project is a success, and I can release it to my family and close friends! I hope that they enjoy playing this game as much as I have enjoyed creating it.

END OF EVALUATION

Appendix – Final Program Code

config.py

```
import pygame, sys
from pygame.locals import *
import os

pygame.init()

FPS = 60

# Screen Resolution
aspect_ratio = [16,9]
pixel_factor = 100
window_width, window_height = aspect_ratio[0] * pixel_factor, aspect_ratio[1] * pixel_factor

WINDOW = pygame.display.set_mode((window_width, window_height))

map_width, map_height = 4000,4000

# Fonts
pygame.font.init()
tinyfont = pygame.font.SysFont("Arial", int(pixel_factor/8))
smallfont = pygame.font.SysFont("Arial", int(pixel_factor/4))
mediumfont = pygame.font.SysFont("Arial", int(pixel_factor/1.5))
largefont = pygame.font.SysFont("Arial", int(pixel_factor))

# Colours
BLACK, WHITE = (0,0,0), (255,255,255)
GRAY = (100,100,130)
RED, GREEN, BLUE = (255,0,0), (0,255,0), (0,0,255)

# Music and Sounds
pygame.mixer.init()
lobby_music = pygame.mixer.music.load("assets/lobby_music.wav")
pygame.mixer.music.set_volume(0.3)

gun_sound = pygame.mixer.Sound("assets/gunshot.wav")
guncock_sound = pygame.mixer.Sound("assets/guncock.wav")
lets_roll_sound = pygame.mixer.Sound("assets/lets_roll.wav")
player_damaged_sound = pygame.mixer.Sound("assets/player_damaged_sound.wav")
enemy_damaged_sound = pygame.mixer.Sound("assets/enemy_damaged_sound.wav")
enemy_death_sound = pygame.mixer.Sound("assets/enemy_dead.wav")
win_sound = pygame.mixer.Sound("assets/game_lived.wav")
lose_sound = pygame.mixer.Sound("assets/game_died.wav")
health_sound = pygame.mixer.Sound("assets/health.wav")
speed_sound = pygame.mixer.Sound("assets/speed.wav")
shield_sound = pygame.mixer.Sound("assets/shield.wav")
unshield_sound = pygame.mixer.Sound("assets/unshield.wav")
freeze_sound = pygame.mixer.Sound("assets/freeze.wav")
unfreeze_sound = pygame.mixer.Sound("assets/unfreeze.wav")
poison_sound = pygame.mixer.Sound("assets/poison.wav")
coin_sound = pygame.mixer.Sound("assets/coin.wav")

menubg = pygame.transform.scale(pygame.image.load("assets/menubg.jpg"), (window_width, window_height))

UI_border = int(window_width/200)

enemy_size = int(window_width/24)

health_bar_length = int(window_width/20)
health_bar_height = int(window_width/80)
health_bar_border = int(health_bar_length/15)
```

```
time_limit = 120
score_goal = 200

htp_page_num = 14
htp_pages = []
for i in range(htp_page_num):
    htp_pages.append(pygame.transform.scale(pygame.image.load("assets/htp" + str(i + 1) + ".png").convert(), (int(
window_width*0.375), int((window_height*0.35)))))

# 0: Grassland
# 1: Desert
# 2: City
# 3: Volcano
theme_options = {
    0: {
        "name": "GRASSLAND",
        "banner": pygame.transform.scale(pygame.image.load("assets/grassland_banner.jpg").convert(), (int(window_width*0.35
), int((window_height*0.3)))),
        "map": pygame.transform.scale(pygame.image.load("assets/grassland_map.png").convert(), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png").convert_alpha(), (enemy_size,
enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png").convert_alpha(), (
enemy_size, enemy_size))
        },
        "music": pygame.mixer.Sound("assets/grassland_bg.wav"),
        "footsteps": pygame.mixer.Sound("assets/footsteps_grass.wav")
    },
    1: {
        "name": "DESERT",
        "banner": pygame.transform.scale(pygame.image.load("assets/desert_banner.jpg").convert(), (int(window_width*0.35),
int((window_height*0.3)))),
        "map": pygame.transform.scale(pygame.image.load("assets/desert_map.png").convert(), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite_desert.png").convert_alpha(), (
enemy_size, enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite_desert.png").convert_alpha
(), (enemy_size, enemy_size))
        },
        "music": pygame.mixer.Sound("assets/desert_bg.wav"),
        "footsteps": pygame.mixer.Sound("assets/footsteps_grass.wav")
    },
    2: {
        "name": "CITY",
        "banner": pygame.transform.scale(pygame.image.load("assets/city_banner.jpg").convert(), (int(window_width*0.35),
int((window_height*0.3)))),
        "map": pygame.transform.scale(pygame.image.load("assets/city_map.png").convert(), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png").convert_alpha(), (enemy_size,
enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png").convert_alpha(), (
enemy_size, enemy_size))
        },
        "music": pygame.mixer.Sound("assets/grassland_bg.wav"),
        "footsteps": pygame.mixer.Sound("assets/footsteps_rock.wav")
    },
    3: {
        "name": "LAVA",
        "banner": pygame.transform.scale(pygame.image.load("assets/volcano_banner.png").convert(), (int(window_width*0.35
), int((window_height*0.3)))),
        "map": pygame.transform.scale(pygame.image.load("assets/volcano_map.png").convert(), (map_width, map_height)),
        "enemies": {
            "basic": pygame.transform.scale(pygame.image.load("assets/enemy_sprite.png").convert_alpha(), (enemy_size,
enemy_size)),
            "shooter": pygame.transform.scale(pygame.image.load("assets/shooter_enemy_sprite.png").convert_alpha(), (
enemy_size, enemy_size))
        },
        "music": pygame.mixer.Sound("assets/volcano_bg.wav"),
        "footsteps": pygame.mixer.Sound("assets/footsteps_volcano.wav")
    }
}
```

```
# Player images
player_walk_images = [pygame.transform.scale(pygame.image.load("assets/player_sprite.png").convert_alpha(), (int(window_width/20),
int(window_width/20))),
                      pygame.transform.scale(pygame.image.load("assets/player_sprite2.png").convert_alpha(), (int(window_width/20
), int(window_width/20)))]
player_img_damaged = pygame.transform.scale(pygame.image.load("assets/player_sprite_damaged.png").convert_alpha(), (
player_walk_images[0].get_width(), player_walk_images[0].get_height()))
player_img_poison = pygame.transform.scale(pygame.image.load("assets/player_sprite_poison.png").convert_alpha(), (
player_walk_images[0].get_width(), player_walk_images[0].get_height()))
player_menu_img = pygame.transform.scale(player_walk_images[0], (int(window_width/4), int(window_width/4)))
player_dead_img = pygame.transform.scale(pygame.image.load("assets/player_dead.png").convert_alpha(), (int(window_width/4), int(
window_width/4)))

# Menu images
gun_img = pygame.transform.scale(pygame.image.load("assets/gun.png").convert_alpha(), (int(window_width/3),int((window_width/3)/1.5
)))
trophy_img = pygame.transform.scale(pygame.image.load("assets/trophy.png").convert_alpha(),(int(window_width/8),int(window_width/8
)))
piggy_img = pygame.transform.scale(pygame.image.load("assets/piggybank.png").convert_alpha(), (int(window_width/8), int(
window_width/12)))

# Coin images and settings
coin_img = pygame.transform.scale(pygame.image.load("assets/coin.png").convert_alpha(), (int(window_width/30), int(window_width/30
)))
coin_num = 2
coin_spawn_range = 50

cursor_img = pygame.transform.scale(pygame.image.load("assets/cursor.png").convert_alpha(), (int(window_width/50), int(window_width
/50)))

# Enemy AI coordinate offset timer
reset_offset_timer = 0.5
enemy_damage_speed = 0.5
enemy_wave_period = 6
enemy_num_chance = 8 # the higher the number, the smaller the number of enemies spawned

# Center of screen for player
centre_x = int(window_width / 2)
centre_y = int(window_height / 2)

# Display hitboxes for debugging
HITBOXES = False

# Icons for powerups
powerup_icons = {"health": pygame.image.load("assets/powerup_health.png").convert_alpha(),
                 "speed": pygame.image.load("assets/powerup_speed.png").convert_alpha(),
                 "shield": pygame.image.load("assets/powerup_shield.png").convert_alpha(),
                 "poison": pygame.image.load("assets/powerup_poison.png").convert_alpha(),
                 "freeze": pygame.image.load("assets/powerup_freeze.png").convert_alpha()}
powerup_size = int(window_width/25)
active_powerup_size = int(window_width/15)

shield_bubble = pygame.transform.scale(pygame.image.load("assets/shield_bubble.png").convert_alpha(), (int(window_width/8), int(
window_width/8)))
freeze_overlay = pygame.transform.scale(pygame.image.load("assets/freeze_overlay.png").convert_alpha(), (window_width,
window_height))

projectile_size = int(window_width/250)

settings = {
    "easy": {
        "player": {
            "max-health": 200,
            "speed": 10,
            "proj-speed": 30,
            "proj-damage": 10
        },
        "powerup": {
            "chance-freq": 2,
            "lifetime": 50,
            "duration": 10,
            "health-boost": 50,
            "speed-mult": 1.5,
            "poison-damage": 10
        },
```

```
        "enemy": {
            "basic": {
                "max-health": 50,
                "speed": 1,
                "damage": 10,
                "score": 5
            },
            "shooter": {
                "max-health": 80,
                "speed": 1,
                "damage": 10,
                "proj-freq": 5,
                "proj-speed": 5,
                "proj-damage": 10,
                "score": 10
            }
        }
    },
    "normal": {
        "player": {
            "max-health": 150,
            "speed": 10,
            "proj-speed": 30,
            "proj-damage": 10
        },
        "powerup": {
            "chance-freq": 4,
            "lifetime": 30,
            "duration": 10,
            "health-boost": 50,
            "speed-mult": 1.5,
            "poison-damage": 10
        },
        "enemy": {
            "basic": {
                "max-health": 100,
                "speed": 1,
                "damage": 10,
                "score": 10
            },
            "shooter": {
                "max-health": 150,
                "speed": 1,
                "damage": 10,
                "proj-freq": 5,
                "proj-speed": 5,
                "proj-damage": 20,
                "score": 20
            }
        }
    },
    "hard": {
        "player": {
            "max-health": 100,
            "speed": 10,
            "proj-speed": 30,
            "proj-damage": 10
        },
        "powerup": {
            "chance-freq": 6,
            "lifetime": 20,
            "duration": 10,
            "health-boost": 50,
            "speed-mult": 1.5,
            "poison-damage": 10
        },
        "enemy": {
            "basic": {
                "max-health": 150,
                "speed": 1,
                "damage": 20,
                "score": 20
            },
            "shooter": {
                "max-health": 200,
                "speed": 1,
                "damage": 20,
                "proj-freq": 5,
                "proj-speed": 5,
                "proj-damage": 20,
                "score": 30
            }
        }
    }
}
```

sprites.py

```
import pygame, sys
from pygame.locals import *
import math
import random
from config import *


class Player():

    def __init__(self, health, speed, powerup_speed_multiplier):
        # Coordinates for centre of screen for position of player
        self.x = centre_x - (player_walk_images[0].get_width() / 2)
        self.y = centre_y  - (player_walk_images[0].get_height() / 2)

        # Pixel distances set for how far the player can walk (up to the edge of the map)
        self.border_x = (map_width / 2) - (player_walk_images[0].get_width() / 2)
        self.border_y = (map_height / 2) - (player_walk_images[0].get_height() / 2)


# Animation variables: animation_frames decides how fast the animation is i.e. how many frames of game should pass for one frame o
f animation
        self.animation_count = 0
        self.animation_frames = 8

        # Get state of the player for display purposes
        self.orientation = "right"
        self.moving_x = False
        self.moving_y = False

        self.health = health
        self.max_health = health
        self.speed = speed
        self.original_speed = speed
        self.powerup_speed_multiplier = powerup_speed_multiplier

        self.shield = False
        self.poison = False

    def get_borders(self):
        return {"x":self.border_x, "y":self.border_y}

    def get_speed(self):
        return self.speed

    def get_health(self):
        return self.health
    def set_health(self, new_health):
        self.health = new_health

    def get_orientation(self):
        return self.orientation
    def set_orientation(self, new_orientation):
        self.orientation = new_orientation

    def get_moving(self, axis):
        if axis == "x":
            return self.moving_x
        elif axis == "y":
            return self.moving_y

    def set_moving_x(self, new_moving_x):
        self.moving_x = new_moving_x
    def set_moving_y(self, new_moving_y):
        self.moving_y = new_moving_y

    def get_xy(self, axis):
        if axis == "x":
            return self.x
        elif axis == "y":
            return self.y
```

```
def main(self, WINDOW):

    # Loops back round after animation count has reached the end to prevent index error
    if self.animation_count + 1 >= 2*self.animation_frames:
        self.animation_count = 0
    self.animation_count += 1

    # Checks orientation of player and flips the sprite horizontally accordingly
    if self.orientation == "right":
        # If the player is moving the sprite, play the animation at the speed decided
        if self.moving_x or self.moving_y:
            WINDOW.blit(player_walk_images[self.animation_count//self.animation_frames], (self.x, self.y))
        # Otherwise, just play the first frame to make it look like they are still
        else:
            WINDOW.blit(player_walk_images[0], (self.x, self.y))
    # Same for left
    elif self.orientation == "left":
        if self.moving_x or self.moving_y:
            WINDOW.blit(pygame.transform.flip(player_walk_images[self.animation_count//self.animation_frames],
True, False), (self.x, self.y))
        else:
            WINDOW.blit(pygame.transform.flip(player_walk_images[0], True, False), (self.x, self.y))

    # If poison effect on, run poison overlay
    if self.poison:
        self.show_poison(WINDOW)

    # health bar coordinates
    self.health_bar_x = centre_x - int(health_bar_length/2)
    self.health_bar_y = self.y - health_bar_height - 2*health_bar_border
    # drawing on health bar, calculating length depending on proportion between max health and current health
    health_bar_circle = pygame.draw.circle(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y + int(
health_bar_height/2)), int(health_bar_height*0.8))
    self.health_px = int((self.health/self.max_health) * (health_bar_length - int(health_bar_circle.width/2)))
    pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y, health_bar_length, health_bar_height))
    pygame.draw.rect(WINDOW, GREEN, (self.health_bar_x + int(health_bar_circle.width/2), self.health_bar_y, self.
health_px, health_bar_height))
    pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y, health_bar_length, health_bar_height),
health_bar_border)
    # health bar text to display numerical health value
    health_text = tinyfont.render(str(self.health), True, WHITE)
    WINDOW.blit(health_text, (health_bar_circle.centerx - health_text.get_width()/2, health_bar_circle.centery -
health_text.get_height()/2))
def hitbox(self, WINDOW):
    if HITBOXES:
        pygame.draw.rect(WINDOW, (255,0,0), (self.x, self.y, player_walk_images[0].get_width(), player_walk_images
[1].get_height()), 1)

    return pygame.Rect(self.x, self.y, player_walk_images[0].get_width(), player_walk_images[1].get_height())

def show_poison(self, WINDOW):
    if self.orientation == "left":
        WINDOW.blit(pygame.transform.flip(player_img_poison, True, False), (self.x, self.y))
    elif self.orientation == "right":
        WINDOW.blit(player_img_poison, (self.x, self.y))

def speed_on(self):
    self.speed = int(self.original_speed * self.powerup_speed_multiplier)
def speed_off(self):
    self.speed = self.original_speed
```

```
def get_shield_status(self):
        return self.shield
    def shield_on(self):
        self.shield = True
    def shield_off(self):
        self.shield = False

    def get_poison_status(self):
        return self.poison
    def poison_on(self):
        self.poison = True
    def poison_off(self):
        self.poison = False


class BasicEnemy():

    def __init__(self, speed, health, damage, damage_speed, sprite_img):


# Pass in the image of the sprite, since there will be multiple enemy types so each will need its own sprite image
        self.sprite_img = sprite_img

        # Border attributes to ensure enemies don't leave the map
        self.border_x = (map_width / 2) - (int(self.sprite_img.get_width() / 2))
        self.border_y = (map_height / 2) - (int(self.sprite_img.get_height() / 2))

        # Initialise a random position on the map to spawn the enemy
        self.x = random.randint(centre_x - self.border_x, centre_x + self.border_x)
        self.y = random.randint(centre_y - self.border_y, centre_y + self.border_y)

        # Basic attributes for enemy
        self.speed = speed
        self.health = health
        self.max_health = health
        self.damage = damage
        self.damage_speed = damage_speed
        self.damage_delay_timer = 0

        # Offset attributes to give the enemy a strafing ability (preventing enemies all piling on top of each other)
        self.reset_offset = 0
        self.offset_x = random.randint(-300, 300)
        self.offset_y = random.randint(-300, 300)

        self.frozen = False


    def main(self, WINDOW, camera_scroll):

        if self.reset_offset == 0:
            self.offset_x = random.randint(-300, 300)
            self.offset_y = random.randint(-300, 300)
            self.reset_offset = random.randint(int(FPS*reset_offset_timer), int(FPS*reset_offset_timer + FPS))
        else:
            self.reset_offset -= 1

        self.direction = None

        # If frozen effect is false, let the enemies move
        if self.frozen == False:
            if self.x < centre_x + self.offset_x + camera_scroll[0] and self.x < centre_x + self.border_x:
                self.x += self.speed
            elif self.x > centre_x + self.offset_x + camera_scroll[0] and self.x > centre_x - self.border_x:
                self.x -= self.speed
            if self.y < centre_y + self.offset_y + camera_scroll[1] and self.y < centre_y + self.border_y:
                self.y += self.speed
            elif self.y > centre_y + self.offset_y + camera_scroll[1] and self.y > centre_y - self.border_y:
                self.y -= self.speed
```

```
    WINDOW.blit(self.sprite_img, (self.x - camera_scroll[0] - (self.sprite_img.get_width() / 2), self
.y - camera_scroll[1] - (self.sprite_img.get_height() / 2)))

    # health bar coordinates
    self.health_bar_x = self.x - camera_scroll[0] - int(health_bar_length/2)
    self.health_bar_y = self.y - camera_scroll[1] - (self.sprite_img.get_height()/2) - health_bar_height - 2
*health_bar_border
    # drawing on health bar, calculating length depending on proportion between max health and current health
    health_bar_circle = pygame.draw.circle(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y + int
(health_bar_height/2)), int(health_bar_height*0.8))
    self.health_px = int((self.health/self.max_health) * (health_bar_length - int(health_bar_circle.width/2)))
    pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self.health_bar_y, health_bar_length, health_bar_height))
    pygame.draw.rect(WINDOW, RED, (self.health_bar_x + int(health_bar_circle.width/2), self.health_bar_y, self
.health_px, health_bar_height))
    pygame.draw.rect(WINDOW, BLACK, (self.health_bar_x, self
.health_bar_y, health_bar_length, health_bar_height), health_bar_border)
    # health bar text to display numerical health value
    health_text = tinyfont.render(str(self.health), True, WHITE)
    WINDOW.blit(health_text, (health_bar_circle.centerx - health_text.get_width()/2
, health_bar_circle.centery - health_text.get_height()/2))
def hitbox(self, WINDOW, camera_scroll):

    if HITBOXES:
        pygame.draw.rect(WINDOW, (255,0,0), (self.x - camera_scroll[0] - (self.sprite_img.get_width() / 2),
                                             self.y - camera_scroll[1] - (self.sprite_img.get_height() / 2),
                                             self.sprite_img.get_width(), self.sprite_img.get_height()), 1)

    return pygame.Rect(self.x - camera_scroll[0] - (self.sprite_img.get_width() / 2), self.y - camera_scroll[1] - (
self.sprite_img.get_height() / 2), self.sprite_img.get_width(), self.sprite_img.get_height())

def identify(self):
    return "basic"
def get_health(self):
    return self.health
def set_health(self, new_health):
    self.health = new_health

def set_damage_delay_timer(self, new_damage_delay_timer):
    self.damage_delay_timer = new_damage_delay_timer
def get_damage_delay_timer(self):
    return self.damage_delay_timer
def get_damage(self):
    return self.damage
def get_damage_speed(self):
    return self.damage_speed

def get_xy(self, axis, camera_scroll):
    if axis == "x":
        return self.x - camera_scroll[0] - (self.sprite_img.get_width() / 2)
    elif axis == "y":
        return self.y - camera_scroll[1] - (self.sprite_img.get_height() / 2)

def get_sprite_img(self):
    return self.sprite_img
def get_frozen_status(self):
    if self.frozen == True:
        return True
    else:
        return False

def freeze_on(self):
    self.frozen = True
def freeze_off(self):
    self.frozen = False
```

```
class ShooterEnemy(BasicEnemy):

    def __init__(self, speed, health, damage, damage_speed, projectile_freq, sprite_img):
        super().__init__(speed, health, damage, damage_speed, sprite_img)

        self.projectile_freq = projectile_freq
        self.projectile_cooldown = projectile_freq

    def get_projectile_freq(self):
        return self.projectile_freq

    def get_projectile_cooldown(self):
        return self.projectile_cooldown
    def set_projectile_cooldown(self, new_projectile_cooldown):
        self.projectile_cooldown = new_projectile_cooldown

    def identify(self):
        return "shooter"


class PlayerProjectile():

    def __init__(self, cursor_x, cursor_y, speed, damage, sprite_img):

        self.speed = speed
        self.damage = damage
        self.despawn_range = 500

        self.sprite_img = sprite_img

        self.x = centre_x
        self.y = centre_y
        self.cursor_x = cursor_x
        self.cursor_y = cursor_y
        self.dxdy = self.calc_trajectory()


    def calc_trajectory(self):
        self.angle = math.atan2(self.y - self.cursor_y, self.x - self.cursor_x)
        self.dx = math.cos(self.angle) * self.speed
        self.dy = math.sin(self.angle) * self.speed
        return [self.dx, self.dy]


    def main(self, WINDOW, projectilekey, controls, player_speed, is_moving_x, is_moving_y):

        self.x -= int(self.dxdy[0])
        self.y -= int(self.dxdy[1])

        if is_moving_x:
            if (projectilekey[pygame.K_a] and controls == "WASD") or (projectilekey[pygame.K_LEFT] and controls == "arrows"):
                self.x += player_speed
            if (projectilekey[pygame.K_d] and controls == "WASD") or (projectilekey[pygame.K_RIGHT] and controls == "arrows"):
                self.x -= player_speed
        if is_moving_y:
            if (projectilekey[pygame.K_w] and controls == "WASD") or (projectilekey[pygame.K_UP] and controls == "arrows"):
                self.y += player_speed
            if (projectilekey[pygame.K_s] and controls == "WASD") or (projectilekey[pygame.K_DOWN] and controls == "arrows"):
                self.y -= player_speed

        pygame.draw.circle(WINDOW, self.sprite_img, (self.x, self.y), projectile_size)

    def get_damage(self):
        return self.damage

    def get_xy(self, axis):
        if axis == "x":
            return self.x
        elif axis == "y":
            return self.y

    def get_despawn_range(self):
        return self.despawn_range

    def identify(self):
        return "player"
```

```
class EnemyProjectile(PlayerProjectile):

    def __init__(self, speed, damage, origin_x, origin_y, sprite_img):
        super().__init__(0, 0, speed, damage, sprite_img)
        self.despawn_range = 500

        self.x = origin_x
        self.y = origin_y

        self.dxdy = self.calc_trajectory()

    def calc_trajectory(self):
        self.angle = math.atan2(self.y - centre_y, self.x - centre_x)
        self.dx = math.cos(self.angle) * self.speed
        self.dy = math.sin(self.angle) * self.speed
        return [self.dx, self.dy]

    def identify(self):
        return "enemy"


class Powerup():

    def __init__(self, powerup_id):
        # Powerup ID determines which powerup this powerup object is and picks out the relevant icon for it
        self.powerup_id = powerup_id
        self.sprite_icon = powerup_icons[powerup_id]
        self.sprite_icon = pygame.transform.scale(self.sprite_icon, (powerup_size, powerup_size))

        self.border_x = int((map_width / 2)) - int((powerup_size / 2))
        self.border_y = int((map_height / 2)) - int((powerup_size / 2))
        self.x = random.randint(centre_x - self.border_x, centre_x + self.border_x)
        self.y = random.randint(centre_y - self.border_y, centre_y + self.border_y)

        # Timer to control when to despawn
        self.timer = 0

    def main(self, WINDOW, camera_scroll, powerup_lifetime):
        # Allow it to flash right before it disappears, like a fading effect almost
        if (self.timer > FPS*(powerup_lifetime - 1) and self.timer < int(FPS*(powerup_lifetime - 0.5)) or
            self.timer > FPS*(powerup_lifetime - 2) and self.timer < int(FPS*(powerup_lifetime - 1.5)) or
            self.timer > FPS*(powerup_lifetime - 3) and self.timer < int(FPS*(powerup_lifetime - 2.5))):
            return

        WINDOW.blit(self.sprite_icon, (self.x - camera_scroll[0] - (powerup_size/2), self.y - camera_scroll[1] - (powerup_size/
2)))

    def hitbox(self, WINDOW, camera_scroll):
        if HITBOXES:
            pygame.draw.rect(WINDOW, (255,0,0), (self.x - camera_scroll[0] - (powerup_size/2),
                                                 self.y - camera_scroll[1] - (powerup_size/2),
                                                 powerup_size, powerup_size), 1)

        return pygame.Rect(self.x - camera_scroll[0] - (powerup_size/2), self.y - camera_scroll[1] - (powerup_size/2
), powerup_size, powerup_size)

    def identify(self):
        return self.powerup_id

    def get_timer(self):
        return self.timer
    def set_timer(self, new_time):
        self.timer = new_time


# Health is a special powerup since it does not get applied over a period of time, it is just straight away used up when collec
ted, so this method is in this class instead
    def health(self, player, powerup_health_boost):
        player.health += powerup_health_boost
        if player.health > player.max_health:
            player.health = player.max_health
```

```
class ActivePowerups():
    def __init__(self, active_powerups):
        self.active_powerups = active_powerups

    def main(self, WINDOW, powerup_duration):
        # Number of active powerups for active icon formatting
        self.num_active_powerups = 0
        for powerup in self.active_powerups:
            if self.active_powerups[powerup]["active"]:
                self.display_icon(WINDOW, powerup, self.num_active_powerups, self.active_powerups[powerup]["timer"
], powerup_duration)
                self.num_active_powerups += 1
                # Timer for powerups incrementing
                self.active_powerups[powerup]["timer"] += 1
                if self.active_powerups[powerup]["timer"] >= FPS*powerup_duration:
                    self.active_powerups[powerup]["active"] = 0
                    self.active_powerups[powerup]["timer"] = 0

    def activate_powerups(self, player, enemies):
        # Speed powerup
        if self.active_powerups["speed"]["active"]:
            player.speed_on()
        else:
            player.speed_off()
        # Shield powerup
        if self.active_powerups["shield"]["active"]:
            player.shield_on()
        else:
            player.shield_off()
        # Poison powerup
        if self.active_powerups["poison"]["active"]:
            player.poison_on()
        else:
            player.poison_off()
        # Freeze powerup
        if self.active_powerups["freeze"]["active"]:
            for enemy in enemies:
                enemy.freeze_on()
        else:
            for enemy in enemies:
                enemy.freeze_off()

    def display_icon(self, WINDOW, powerup_id, num_active_powerups, timer, powerup_duration):
        # Display all active powerups as large icons in the corner, along with how long it has left
        active_powerup_icon = pygame.transform.scale(powerup_icons[powerup_id], (active_powerup_size, active_powerup_size))
        active_powerup_blit = WINDOW.blit(active_powerup_icon, (int(window_width*0.01) + (num_active_powerups*(active_powerup_size+
20)), int(window_width*0.01)))
        timer_display = smallfont.render(str(int(((FPS*powerup_duration - timer)/FPS) + 1)), True, (0,0,0))
        timer_card = pygame.draw.rect(WINDOW, GRAY, (active_powerup_blit.centerx - active_powerup_blit.width/4
, active_powerup_blit.bottom + int(window_width*0.01), active_powerup_blit.width/2, timer_display.get_height()))
        pygame.draw.rect(WINDOW, BLACK, (timer_card.left - int(UI_border/2), timer_card.top - int(UI_border/2), timer_card.width +
int(UI_border/2)*2, timer_card.height + int(UI_border/2)*2), int(UI_border/2))
        WINDOW.blit(timer_display, (timer_card.centerx - timer_display.get_width()/2
, timer_card.centery - timer_display.get_height()/2))


class Coin():
    def __init__(self, x, y, set_coin_img):
        self.x = x
        self.y = y
        self.coin_img = set_coin_img

    def main(self, WINDOW, camera_scroll):
        WINDOW.blit(self.coin_img, (self.x - camera_scroll[0], self.y - camera_scroll[1]))

    def hitbox(self, WINDOW, camera_scroll):
        if HITBOXES:
            pygame.draw.rect(WINDOW, (255,0,0), (self.x - camera_scroll[0] - (self.coin_img.get_width() / 2),
                                                 self.y - camera_scroll[1] - (self.coin_img.get_height() / 2),
                                                 self.coin_img.get_width(), self.coin_img.get_height()), 1)

        return pygame.Rect(self.x - camera_scroll[0] - (self.coin_img.get_width() / 2), self.y - camera_scroll[1] - (self
.coin_img.get_height() / 2), self.coin_img.get_width(), self.coin_img.get_height())
```

main.py

```
#imports
import pygame, sys
from pygame.locals import *
import json
from sprites import *
from config import *

# Initialise pygame and basic pygame variables
pygame.init()
clock = pygame.time.Clock()

# Create window
pygame.display.set_caption("NEA")

class GameState():

    def __init__(self):

        self.state = "main_menu"

        self.difficulty = "easy"
        self.mode = "timed"
        self.theme = theme_options[0]

        self.controls = "WASD"

        self.timer = 0
        self.score_count = 0
        self.coins = 0
        self.died = False

        pygame.mixer.music.play(-1)

    def main_menu(self):

        # Get save.json for coins
        with open("save.json", "r") as read_save:
            save = json.load(read_save)

        while True:
            pygame.display.update()
            # bg colour
            WINDOW.blit(menubg, (0,0))

            # Player and gun
            player_menu_icon = WINDOW.blit(player_menu_img, (int(window_width*0.2), int(window_height*0.4)))
            WINDOW.blit(pygame.transform.rotate(gun_img, -20), (player_menu_icon.centerx - player_menu_icon.width/1.3
, player_menu_icon.centery - player_menu_icon.height/2.2))

            # Piggy bank
            piggy_icon = WINDOW.blit(piggy_img, (int(window_width*0.05), int(window_height*0.7)))
            coins_text = smallfont.render(str(save["coins"]), True, BLACK)
            WINDOW.blit(coins_text, (piggy_icon.centerx, piggy_icon.centery - coins_text.get_height()/2))

            # title
            title_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int
(window_height*0.15)))
            title_text = mediumfont.render("SHOOTER GAME", True, BLACK)
            WINDOW.blit(title_text, (title_card.centerx - title_text.get_width()/2, title_card.centery - title_text.get_height()/2))

            # play button
            play_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.6), int(window_height*0.35), int(window_width*0.2), int
(window_height*0.15)))
            play_text = smallfont.render("PLAY", True, BLACK)
            WINDOW.blit(play_text, (play_button.centerx - play_text.get_width()/2, play_button.centery - play_text.get_height()/2))

            # settings button
            options_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.6), int(window_height*0.55), int(window_width*0.2),
int(window_height*0.15)))
            options_text = smallfont.render("OPTIONS", True, BLACK)
            WINDOW.blit(options_text, (options_button.centerx - options_text.get_width()/2
, options_button.centery - options_text.get_height()/2))
```

```
        # quit button
        quit_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.6), int(window_height*0.75), int(window_width*0.2), int
(window_height*0.15)))
        quit_text = smallfont.render("QUIT", True, BLACK)
        WINDOW.blit(quit_text, (quit_button.centerx - quit_text.get_width()/2, quit_button.centery - quit_text.get_height()/2))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if play_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    self.state = "dif_mode_menu"
                    return
            if options_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    self.state = "options"
                    return
            if quit_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    self.state = "quit"
                    return

        clock.tick(FPS)

def options(self):

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.blit(menubg, (0,0))

        # title
        options_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int
(window_height*0.15)))
        options_text = mediumfont.render("OPTIONS", True, BLACK)
        WINDOW.blit(options_text, (options_card.centerx - options_text.get_width()/2, options_card.centery - options_text.get_height()/2))

        # shop
        shop_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.4), int(window_height*0.35), int(window_width*0.2), int
(window_height*0.15)))
        shop_text = smallfont.render("SHOP", True, BLACK)
        WINDOW.blit(shop_text, (shop_button.centerx - shop_text.get_width()/2, shop_button.centery - shop_text.get_height()/2))

        # HTP
        htp_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.55), int(window_width*0.175), int
(window_height*0.15)))
        htp_text = smallfont.render("HOW TO PLAY", True, BLACK)
        WINDOW.blit(htp_text, (htp_button.centerx - htp_text.get_width()/2, htp_button.centery - htp_text.get_height()/2))

        # Controls
        controls_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.525), int(window_height*0.55), int(window_width*0.175), int
(window_height*0.15)))
        controls_instruction = tinyfont.render("CONTROL SETTINGS (click to toggle)", True, BLACK)
        WINDOW.blit(controls_instruction, ((controls_button.centerx - controls_instruction.get_width()/2
, controls_button.top + controls_button.height/5)))
        controls_text = smallfont.render(self.controls, True, BLACK)
        WINDOW.blit(controls_text, (controls_button.centerx - controls_text.get_width()/2, controls_button.centery))

        # back
        back_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.4), int(window_height*0.75), int(window_width*0.2), int
(window_height*0.15)))
        back_text = smallfont.render("BACK", True, BLACK)
        WINDOW.blit(back_text, (back_button.centerx - back_text.get_width()/2, back_button.centery - back_text.get_height()/2))
```

```
# Checking for events
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()

                # Checking for cursor coordinates
                cursor = pygame.mouse.get_pos()

                # Checking for clicks on each button and changing the state accordingly
                if shop_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        self.state = "shop"
                        return
                if htp_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        self.state = "htp"
                        return
                if controls_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        if self.controls == "WASD":
                            self.controls = "arrows"
                        elif self.controls == "arrows":
                            self.controls = "WASD"
                if back_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        self.state = "main_menu"
                        return

            clock.tick(FPS)

    def shop(self):

        while True:
            pygame.display.update()

            # bg colour
            WINDOW.blit(menubg, (0,0))

            # title
            shop_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int
(window_height*0.15)))
            shop_text = mediumfont.render("SHOP", True, BLACK)
            WINDOW.blit(shop_text, (shop_card.centerx - shop_text.get_width()/2, shop_card.centery - shop_text.get_height()/2))

            # sorry, nothing here yet
            sorry_text = mediumfont.render("Sorry, nothing in the shop yet...", True, BLACK)
            WINDOW.blit(sorry_text, (window_width/2 - sorry_text.get_width()/2, window_width/2 - sorry_text.get_width()/2))

            # back
            back_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.4), int(window_height*0.75), int(window_width*0.2), int
(window_height*0.15)))
            back_text = smallfont.render("BACK", True, BLACK)
            WINDOW.blit(back_text, (back_button.centerx - back_text.get_width()/2, back_button.centery - back_text.get_height()/2))

            # Checking for events
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()

                # Checking for cursor coordinates
                cursor = pygame.mouse.get_pos()

                # Checking for clicks on each button and changing the state accordingly
                if back_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        self.state = "options"
                        return

            clock.tick(FPS)
```

```
def htp(self):

    current_page = 0

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.blit(menubg, (0,0))

        # title
        htp_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int(window_height*
0.15)))
        htp_text = mediumfont.render("HOW TO PLAY", True, BLACK)
        WINDOW.blit(htp_text, (htp_card.centerx - htp_text.get_width()/2, htp_card.centery - htp_text.get_height()/2))

        # themes
        pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.3), int(window_width*0.4), int(window_height*0.4)))
        WINDOW.blit(htp_pages[current_page], (int(window_width*0.3125), int(window_height*0.325)))

        # arrows
        left_button = pygame.draw.polygon(WINDOW, WHITE, [[int(window_width*0.275), int(window_height*0.45)],
                                                          [int(window_width*0.25), int(window_height*0.5)],
                                                          [int(window_width*0.275), int(window_height*0.55)]])
        right_button = pygame.draw.polygon(WINDOW, WHITE, [[int(window_width*0.725), int(window_height*0.45)],
                                                           [int(window_width*0.725), int(window_height*0.55)],
                                                           [int(window_width*0.75), int(window_height*0.5)]])

        # back
        back_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.4), int(window_height*0.75), int(window_width*0.2), int
(window_height*0.15)))
        back_text = smallfont.render("BACK", True, BLACK)
        WINDOW.blit(back_text, (back_button.centerx - back_text.get_width()/2, back_button.centery - back_text.get_height()/2))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if left_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    if current_page != 0:
                        current_page -= 1
                    else:
                        current_page = htp_page_num - 1
            if right_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    if current_page != htp_page_num - 1:
                        current_page += 1
                    else:
                        current_page = 0
            if back_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    self.state = "options"
                    return

        clock.tick(FPS)

def dif_mode_menu(self):

    # Load save data
    with open("save.json", "r") as save_file:
        save = json.load(save_file)

    while True:
        pygame.display.update()

        # bg colour
        WINDOW.blit(menubg, (0,0))
```

```
# title
select_dif_mode_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int
(window_height*0.2)))
select_dif_text = mediumfont.render("SELECT DIFFICULTY", True, BLACK)
WINDOW.blit(select_dif_text, (select_dif_mode_card.centerx - select_dif_text.get_width()/2
, select_dif_mode_card.centery - select_dif_text.get_height()/1.1))
and_mode_text = mediumfont.render("AND GAME MODE", True, BLACK)
WINDOW.blit(and_mode_text, (select_dif_mode_card.centerx - and_mode_text.get_width()/2
, select_dif_mode_card.centery - and_mode_text.get_height()/20))

# easy difficulty
if self.difficulty == "easy":
    easy_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.325), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
else:
    easy_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.325), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
easy_text = smallfont.render("EASY", True, BLACK)
WINDOW.blit(easy_text, (easy_button.centerx - easy_text.get_width()/2, easy_button.centery - easy_text.get_height()/2))
# normal difficulty
if self.difficulty == "normal":
    normal_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.45), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
else:
    normal_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.45), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
normal_text = smallfont.render("NORMAL", True, BLACK)
WINDOW.blit(normal_text, (normal_button.centerx - normal_text.get_width()/2, normal_button.centery - normal_text.get_height()/2))
# hard difficulty
if self.difficulty == "hard":
    hard_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.575), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
else:
    hard_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.575), int(window_height*0.35), int(window_width*0.1), int
(window_height*0.15)))
hard_text = smallfont.render("HARD", True, BLACK)
WINDOW.blit(hard_text, (hard_button.centerx - hard_text.get_width()/2, hard_button.centery - hard_text.get_height()/2))

# timed mode
if self.mode == "timed":
    timed_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.325), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
else:
    timed_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.325), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
timed_text = smallfont.render("TIMED", True, BLACK)
WINDOW.blit(timed_text, (timed_button.centerx - timed_text.get_width()/2, timed_button.top + int(timed_button.height*0.15)))
high_score_text = tinyfont.render("High Score", True, BLACK)
WINDOW.blit(high_score_text, (timed_button.centerx - high_score_text.get_width()/2, timed_button.top + int(timed_button.height*0.45)))
high_score_value = smallfont.render(str(save["high-score"]), True, BLACK)
WINDOW.blit(high_score_value, (timed_button.centerx - high_score_value.get_width()/2, timed_button.top + int(timed_button.height*0.6)))
# scored mode
if self.mode == "scored":
    scored_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.45), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
else:
    scored_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.45), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
scored_text = smallfont.render("SCORED", True, BLACK)
WINDOW.blit(scored_text, (scored_button.centerx - scored_text.get_width()/2, scored_button.top + int(scored_button.height*0.15)))
best_time_text = tinyfont.render("Best Time", True, BLACK)
WINDOW.blit(best_time_text, (scored_button.centerx - best_time_text.get_width()/2, scored_button.top + int(scored_button.height*0.45)))
if save["best-time"] is not None:
    best_time_value = smallfont.render(str(save["best-time"]) + "s", True, BLACK)
else:
    best_time_value = smallfont.render("- - s", True, BLACK)
WINDOW.blit(best_time_value, (scored_button.centerx - best_time_value.get_width()/2, scored_button.top + int(scored_button.height*0.6)))
# endless mode
if self.mode == "endless":
    endless_button = pygame.draw.rect(WINDOW, GREEN, (int(window_width*0.575), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
else:
    endless_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.575), int(window_height*0.55), int(window_width*0.1), int
(window_height*0.15)))
endless_text = smallfont.render("ENDLESS", True, BLACK)
WINDOW.blit(endless_text, (endless_button.centerx - endless_text.get_width()/2, endless_button.top + int(endless_button.height*0.15)))
endless_high_score_text = tinyfont.render("High Score", True, BLACK)
```

```
endless_high_score_text_blit = WINDOW.blit(endless_high_score_text, (endless_button.left + int(endless_button.width*0.1
), endless_button.top + int(endless_button.height*0.45)))
endless_longest_time_text = tinyfont.render("Longest Time", True, BLACK)
endless_longest_time_text_blit = WINDOW.blit(endless_longest_time_text, (endless_button.left + int(endless_button.width*0.5
), endless_button.top + int(endless_button.height*0.45)))
endless_high_score_value = smallfont.render(str(save["endless"]["high-score"]), True, BLACK)
WINDOW.blit(endless_high_score_value, (endless_high_score_text_blit.centerx - endless_high_score_value.get_width()/2
, endless_button.top + int(endless_button.height*0.6)))
endless_longest_time_value = smallfont.render(str(save["endless"]["best-time"]) + "s", True, BLACK)
WINDOW.blit(endless_longest_time_value, (endless_longest_time_text_blit.centerx - endless_longest_time_value.get_width()/2
, endless_button.top + int(endless_button.height*0.6)))

# back
back_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.75), int(window_width*0.125), int
(window_height*0.15)))
back_text = mediumfont.render("BACK", True, BLACK)
WINDOW.blit(back_text, (back_button.centerx - back_text.get_width()/2, back_button.centery - back_text.get_height()/2))
# continue
continue_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.45), int(window_height*0.75), int(window_width*0.25), int
(window_height*0.15)))
continue_text = mediumfont.render("CONTINUE", True, BLACK)
WINDOW.blit(continue_text, (continue_button.centerx - continue_text.get_width()/2
, continue_button.centery - continue_text.get_height()/2))

# Checking for events
for event in pygame.event.get():
    if event.type == QUIT:
        pygame.quit()
        sys.exit()

    # Checking for cursor coordinates
    cursor = pygame.mouse.get_pos()

    # Checking for clicks on each button and changing the settings accordingly
    if easy_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.difficulty = "easy"
    if normal_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.difficulty = "normal"
    if hard_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.difficulty = "hard"
    if timed_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.mode = "timed"
    if scored_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.mode = "scored"
    if endless_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.mode = "endless"
    if back_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.state = "main_menu"
            return
    if continue_button.collidepoint(cursor):
        if event.type == MOUSEBUTTONDOWN:
            guncock_sound.play()
            self.state = "theme_menu"
            return
clock.tick(FPS)
```

```
def theme_menu(self):

    num_themes = len(theme_options)
    theme_index = 0

    while True:
        pygame.display.update()
        # bg colour
        WINDOW.blit(menubg, (0,0))

        # title
        select_world_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int
(window_height*0.15)))
        select_world_text = mediumfont.render("SELECT WORLD", True, BLACK)
        WINDOW.blit(select_world_text, (select_world_card.centerx - select_world_text.get_width()/2
, select_world_card.centery - select_world_text.get_height()/2))
        # themes
        world_frame = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.3), int(window_width*0.4), int
(window_height*0.4)))
        WINDOW.blit(theme_options[theme_index]["banner"], (int(window_width*0.325), int(window_height*0.35)))
        world_text = smallfont.render(theme_options[theme_index]["name"], True, BLACK)
        WINDOW.blit(world_text, (world_frame.centerx - world_text.get_width()/2, world_frame.top + int(world_frame.height*0.125/2
) - world_text.get_height()/2))
        # arrows
        left_button = pygame.draw.polygon(WINDOW, WHITE, [[int(window_width*0.275), int(window_height*0.45)],
                                                          [int(window_width*0.25), int(window_height*0.5)],
                                                          [int(window_width*0.275), int(window_height*0.55)]])
        right_button = pygame.draw.polygon(WINDOW, WHITE, [[int(window_width*0.725), int(window_height*0.45)],
                                                           [int(window_width*0.725), int(window_height*0.55)],
                                                           [int(window_width*0.75), int(window_height*0.5)]])

        # back
        back_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.75), int(window_width*0.125), int
(window_height*0.15)))
        back_text = mediumfont.render("BACK", True, BLACK)
        WINDOW.blit(back_text, (back_button.centerx - back_text.get_width()/2, back_button.centery - back_text.get_height()/2))
        # continue
        continue_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.45), int(window_height*0.75), int(window_width*0.25), int
(window_height*0.15)))
        continue_text = mediumfont.render("CONTINUE", True, BLACK)
        WINDOW.blit(continue_text, (continue_button.centerx - continue_text.get_width()/2
, continue_button.centery - continue_text.get_height()/2))

        # Checking for events
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

            # Checking for cursor coordinates
            cursor = pygame.mouse.get_pos()

            # Checking for clicks on each button and changing the state accordingly
            if left_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    if theme_index != 0:
                        theme_index -= 1
                    else:
                        theme_index = num_themes - 1
            if right_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    if theme_index != num_themes - 1:
                        theme_index += 1
                    else:
                        theme_index = 0
            if back_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    guncock_sound.play()
                    self.state = "dif_mode_menu"
                    return
            if continue_button.collidepoint(cursor):
                if event.type == MOUSEBUTTONDOWN:
                    lets_roll_sound.play()
                    self.state = "main_game"
                    self.theme = theme_options[theme_index]
                    return

        clock.tick(FPS)
```

```
def main_game(self):

    # Play bg music
    pygame.mixer.music.stop()
    self.theme["music"].play(-1)

    # Difficulty settings
    powerup_chance_freq = settings[self.difficulty]["powerup"]["chance-freq"]
    powerup_lifetime = settings[self.difficulty]["powerup"]["lifetime"]
    powerup_duration = settings[self.difficulty]["powerup"]["duration"]
    powerup_health_boost = settings[self.difficulty]["powerup"]["health-boost"]
    powerup_speed_multiplier = settings[self.difficulty]["powerup"]["speed-mult"]
    powerup_poison_damage = settings[self.difficulty]["powerup"]["poison-damage"]
    # Player attributes
    player_max_health = settings[self.difficulty]["player"]["max-health"]
    player_health = player_max_health
    player_speed = settings[self.difficulty]["player"]["speed"]
    player_proj_speed = settings[self.difficulty]["player"]["proj-speed"]
    player_proj_damage = settings[self.difficulty]["player"]["proj-damage"]
    # Enemy attributes
    basic_enemy_health = settings[self.difficulty]["enemy"]["basic"]["max-health"]
    basic_enemy_speed = settings[self.difficulty]["enemy"]["basic"]["speed"]
    basic_enemy_damage = settings[self.difficulty]["enemy"]["basic"]["damage"]
    shooter_enemy_health = settings[self.difficulty]["enemy"]["shooter"]["max-health"]
    shooter_enemy_speed = settings[self.difficulty]["enemy"]["shooter"]["speed"]
    shooter_enemy_damage = settings[self.difficulty]["enemy"]["shooter"]["damage"]
    shooter_enemy_proj_freq = settings[self.difficulty]["enemy"]["shooter"]["proj-freq"]
    shooter_enemy_proj_speed = settings[self.difficulty]["enemy"]["shooter"]["proj-speed"]
    shooter_enemy_proj_damage = settings[self.difficulty]["enemy"]["shooter"]["proj-damage"]

    # Images controlled by the selected theme
    mapimg = self.theme["map"]
    enemy_sprite = self.theme["enemies"]["basic"]
    shooter_sprite = self.theme["enemies"]["shooter"]

    player_footsteps = self.theme["footsteps"]
    footsteps = pygame.mixer.Channel(5)

    # Instantiate basic objects needed for game and create lists for arrays of objects
    player = Player(player_health, player_speed, powerup_speed_multiplier)
    enemies = []
    projectiles = []
    powerups = []
    powerups_status = {id: {"active": False, "timer": 0} for id in powerup_icons.keys() if id != "health"}
    coins = []

    # Initialize timer, score count and coin count to 0
    self.timer = 0
    self.score_count = 0
    self.coins = 0

    # Player not dead yet
    self.died = False


# Camera scroll controls where the objects in the scene move depending on the direction the player moves in, since the camera follows the pla
yer
    camera_scroll = [0,0]

    # Initialize frame count for timer
    frames = 0

    # Initialize as unpaused
    paused = False

    # Main loop
    while True:
        pygame.display.update()

        # When the game isn't paused, run the main game code
        if not paused:

            # Make cursor invisible so we can blit the crosshair image instead
            pygame.mouse.set_visible(False)


# Background image controlled by camera scroll, placement determined by background size and window size to ensure player spawns in the middl
e of the map
            WINDOW.fill(RED)
            WINDOW.blit(mapimg, (-(map_width - window_width)/2 - camera_scroll[0], -(map_height - window_height)/2 - camera_scroll[1]))
```

```
# Checking for events
for event in pygame.event.get():
    if event.type == QUIT:
        pygame.quit()
        sys.exit()

    # If player presses escape, pause the game
    if event.type == pygame.KEYUP:
        if event.key == K_ESCAPE:
            paused = True

    # Checking for cursor coordinates
    cursor_x, cursor_y = pygame.mouse.get_pos()
    # Checking if the player clicked mouse
    if event.type == pygame.MOUSEBUTTONDOWN:
        if event.button == 1:
            # If so, append a projectile object to the list
            projectiles.append(PlayerProjectile(cursor_x, cursor_y, speed=player_proj_speed, damage=player_proj_damage, sprite_img=RED))
            gun_sound.play()

# Checking for any keys pressed
keys = pygame.key.get_pressed()

#-----------------------------------------PLAYER/CONTROLS-----------------------------------------#

# If any main controls are pressed, alter the camera scroll so the environment can move accordingly (in an inverted fashion)
if (keys[pygame.K_a] and self.controls == "WASD") or (keys[pygame.K_LEFT] and self.controls == "arrows"):
    if camera_scroll[0] > -player.get_borders()["x"]:
        camera_scroll[0] -= player.get_speed()
        # Also decide the direction the sprite surface is facing
        player.set_orientation("left")
        player.set_moving_x(True)
    else:
        player.set_moving_x(False)
if (keys[pygame.K_d] and self.controls == "WASD") or (keys[pygame.K_RIGHT] and self.controls == "arrows"):
    if camera_scroll[0] < player.get_borders()["x"]:
        camera_scroll[0] += player.get_speed()
        player.set_orientation("right")
        player.set_moving_x(True)
    else:
        player.set_moving_x(False)
if (keys[pygame.K_w] and self.controls == "WASD") or (keys[pygame.K_UP] and self.controls == "arrows"):
    if camera_scroll[1] > -player.get_borders()["y"]:
        camera_scroll[1] -= player.get_speed()
        player.set_moving_y(True)
    else:
        player.set_moving_y(False)
if (keys[pygame.K_s] and self.controls == "WASD") or (keys[pygame.K_DOWN] and self.controls == "arrows"):
    if camera_scroll[1] < player.get_borders()["y"]:
        camera_scroll[1] += player.get_speed()
        player.set_moving_y(True)
    else:
        player.set_moving_y(False)
# If the player isn't pressing any controls, the player is not moving
if self.controls == "WASD":
    if not(keys[pygame.K_a] or keys[pygame.K_d] or keys[pygame.K_w] or keys[pygame.K_s]):
        player.set_moving_x(False)
        player.set_moving_y(False)
    else:
        if not footsteps.get_busy():
            footsteps.play(player_footsteps)
elif self.controls == "arrows":
    if not(keys[pygame.K_LEFT] or keys[pygame.K_RIGHT] or keys[pygame.K_UP] or keys[pygame.K_DOWN]):
        player.set_moving_x(False)
        player.set_moving_y(False)

# Run the main methods for all the main objects in the game
player.main(WINDOW)
player.hitbox(WINDOW)

# If the health goes below 0, game over
if player.get_health() <= 0:
    print("YOU DIED")
    self.state = "game_over"
    self.died = True
    pygame.mouse.set_visible(True)
    return
```

```
#---------------------------------------------ENEMIES---------------------------------------------#
# Every {enemy_wave_period} seconds, spawn some random number of enemies, with a random number of basic and shooter enemies
if self.timer % enemy_wave_period == 1 and len(enemies) < 1000:
    if random.randint(0, enemy_num_chance) == 0:
        if random.randint(0,1) == 0:
            enemies.append(BasicEnemy(speed=basic_enemy_speed, health=basic_enemy_health, damage=basic_enemy_damage, damage_speed
=enemy_damage_speed, sprite_img=enemy_sprite))
        else:
            enemies.append(ShooterEnemy(speed=shooter_enemy_speed, health=shooter_enemy_health, damage=shooter_enemy_damage, damage_speed
=enemy_damage_speed, projectile_freq=shooter_enemy_proj_freq, sprite_img=shooter_sprite))

if keys[pygame.K_m]:
    enemies.append(BasicEnemy(speed=basic_enemy_speed, health=basic_enemy_health, damage=basic_enemy_damage, damage_speed
=enemy_damage_speed, sprite_img=enemy_sprite))
if keys[pygame.K_n]:
    enemies.append(ShooterEnemy(speed=shooter_enemy_speed, health=shooter_enemy_health, damage=shooter_enemy_damage, damage_speed
=enemy_damage_speed, projectile_freq=shooter_enemy_proj_freq, sprite_img=shooter_sprite))


for enemy in enemies:
    # If enemy health goes down to 0, add score to player's score count, delete the enemy, and drop coins
    if enemy.get_health() <= 0:
        enemy_death_sound.play()
        self.score_count += settings[self.difficulty]["enemy"][enemy.identify()]["score"]
        # Let each coin have some random variation to their spawnings so they don't all overlap each other
        for i in range(coin_num):
            coin_x = int(enemy.get_xy("x", camera_scroll) + camera_scroll[0])
            coin_x = random.randint(coin_x - coin_spawn_range, coin_x + coin_spawn_range)
            coin_y = int(enemy.get_xy("y", camera_scroll) + camera_scroll[1])
            coin_y = random.randint(coin_y - coin_spawn_range, coin_y + coin_spawn_range)
            coins.append(Coin(coin_x, coin_y, coin_img))
        enemies.remove(enemy)

    # If any enemy collides with the player's hitbox, the player takes damage at a delayed interval
    if enemy.hitbox(WINDOW, camera_scroll).colliderect(player.hitbox(WINDOW)):

        if enemy.get_damage_delay_timer() == 0:
            if player.get_poison_status() == True:
                enemy.set_health(enemy.get_health() - powerup_poison_damage)
                enemy_damaged_sound.play()
            else:
                if player.get_shield_status() == False:
                    player_damaged_sound.play()
                    player.set_health(player.get_health() - enemy.get_damage())

        enemy.set_damage_delay_timer(enemy.get_damage_delay_timer() + 1)

        if enemy.get_damage_delay_timer() >= FPS * enemy.get_damage_speed():
            enemy.set_damage_delay_timer(0)
    else:
        enemy.set_damage_delay_timer(0)

    enemy.main(WINDOW, camera_scroll)

    # If a shooter enemy's shooting cooldown runs out, fire another shot at the player
    if enemy.identify() == "shooter":
        if enemy.get_projectile_cooldown() == 0 and enemy.get_frozen_status() == False:
            projectiles.append(EnemyProjectile(speed=shooter_enemy_proj_speed, damage=shooter_enemy_proj_damage, origin_x=enemy.get_xy("x"
, camera_scroll), origin_y=enemy.get_xy("y", camera_scroll), sprite_img=BLACK))
        enemy.set_projectile_cooldown(enemy.get_projectile_cooldown() + 1)
        if enemy.get_projectile_cooldown() >= FPS * enemy.get_projectile_freq():
            enemy.set_projectile_cooldown(0)

#---------------------------------------------PROJECTILES---------------------------------------------#

for projectile in projectiles:

    projectile.main(WINDOW, keys, self.controls, player.get_speed(), player.get_moving("x"), player.get_moving("y"))

    projectile_removed = False
    # If any projectile hits an enemy's hitbox, remove both that enemy and that projectile from their corresponding lists
    if projectile.identify() != "enemy":
        for enemy in enemies:
            if enemy.hitbox(WINDOW, camera_scroll).collidepoint(projectile.get_xy("x"), projectile.get_xy("y")):
                enemy.set_health(enemy.get_health() - projectile.get_damage())
                enemy_damaged_sound.play()
                projectiles.remove(projectile)
                projectile_removed = True
                break
```

```
    # But if it hits a player hitbox, deal damage to the player
    else:
        if player.hitbox(WINDOW).collidepoint(projectile.get_xy("x"), projectile.get_xy("y")):
            if player.get_shield_status() == False:
                player_damaged_sound.play()
                player.set_health(player.get_health() - projectile.get_damage())
            projectiles.remove(projectile)
            projectile_removed = True
    # Checks if the projectiles have gone out of range and delete them if so, to save memory
    if ((projectile.get_xy("x") < -projectile.get_despawn_range()) or
        (projectile.get_xy("y") < -projectile.get_despawn_range()) or
        (projectile.get_xy("x") > window_width + projectile.get_despawn_range()) or
        (projectile.get_xy("y") > window_height + projectile.get_despawn_range())) and projectile_removed == False:
        projectiles.remove(projectile)

#-----------------------------------------POWERUPS/COINS-----------------------------------------#

# Blit the freeze powerup overlay on the screen, and if the freeze runs out, play the unfreeze sound effect 1 second before
if powerups_status["freeze"]["active"]:
    WINDOW.blit(freeze_overlay, (0,0))
if powerups_status["freeze"]["timer"] == powerup_duration*FPS - FPS:
    unfreeze_sound.play()

# If powerup is active, blit the shield to the screen
if powerups_status["shield"]["active"]:
    WINDOW.blit(shield_bubble, (int(window_width*0.5) - shield_bubble.get_width()/2, int(window_height*0.5
) - shield_bubble.get_height()/2))
# Start to play the unshield sound 1 second before it runs out, and stop the shield background sound right before it runs out
if powerups_status["shield"]["timer"] == powerup_duration*FPS - FPS:
    unshield_sound.play()
if powerups_status["shield"]["timer"] == powerup_duration*FPS - 1:
    shield_sound.stop()

# At a random chance every frame, spawn a powerup
if random.randint(1, FPS*powerup_chance_freq) == 1:
    powerup_choice = Powerup(random.choice(list(powerup_icons.keys())))
    powerups.append(powerup_choice)

# Run the main for the existing powerups on the map
for powerup in powerups:
    powerup.main(WINDOW, camera_scroll, powerup_lifetime)
    powerup.set_timer(powerup.get_timer() + 1)
    # If the lifetime of the powerup runs out, remove it
    if powerup.get_timer() >= FPS * powerup_lifetime:
        powerups.remove(powerup)

    # If the player collides with the powerup, remove it and make it active
    if powerup.hitbox(WINDOW, camera_scroll).colliderect(player.hitbox(WINDOW)):
        powerups.remove(powerup)
        if powerup.identify() != "health":
            powerups_status[powerup.identify()]["active"] = True
            powerups_status[powerup.identify()]["timer"] = 0

            if powerup.identify() == "speed":
                speed_sound.play()
            if powerup.identify() == "freeze":
                freeze_sound.play()
            if powerup.identify() == "shield":
                shield_sound.stop()
                shield_sound.play(-1, fade_ms=1000)
            if powerup.identify() == "poison":
                poison_sound.play()
        else:
            powerup.health(player, powerup_health_boost)
            health_sound.play()
# Display the active powerups as icons in the corner
active_powerups = ActivePowerups(powerups_status)
active_powerups.main(WINDOW, powerup_duration)
active_powerups.activate_powerups(player, enemies)
```

```
    # Run the main for all existing coins on the map
    for coin in coins:
        coin.main(WINDOW, camera_scroll)
        # If player collides with coin, increment coin count
        if coin.hitbox(WINDOW, camera_scroll).colliderect(player.hitbox(WINDOW)):
            coin_sound.play()
            coins.remove(coin)
            self.coins += 1

    #--------------------------------------------GAME SETTINGS-------------------------------------------#

    # Timer
    frames += 1
    if frames % FPS == 0:
        self.timer += 1
        frames = 0

    # Displaying the timer, adding a number of 0s to the start to make it 3 digits
    time_card = pygame.draw.rect(WINDOW, GRAY, (int(window_width*0.9), 0, int(window_width*0.1), int(window_height*0.1)))
    pygame.draw.rect(WINDOW, BLACK, (int(window_width*0.9) - UI_border, -UI_border, int(window_width*0.1) + 2*UI_border, int(window_height*
0.1) + 2*UI_border), UI_border)
    time_display = str(self.timer)
    if len(time_display) < 4:
        zeros = (3 - len(time_display)) % 3
        for i in range(zeros):
            time_display = "0" + time_display
    time_blit = mediumfont.render(time_display, True, BLACK)
    WINDOW.blit(time_blit, (time_card.centerx - time_blit.get_width()/2, time_card.centery - time_blit.get_height()/2))

    # Display how many enemies are currently on the map
    enemy_count_card = pygame.draw.rect(WINDOW, GRAY, (int(window_width*0.35), 0, int(window_width*0.3), int(window_height*0.1)))
    pygame.draw.rect(WINDOW, BLACK, (int(window_width*0.35) - UI_border, -UI_border, int(window_width*0.3) + 2*UI_border, int(window_height*
0.1) + 2*UI_border), UI_border)
    enemy_count_display = mediumfont.render("Enemies: " + str(len(enemies)), True, BLACK)
    WINDOW.blit(enemy_count_display, (enemy_count_card.centerx - enemy_count_display.get_width()/2
, enemy_count_card.centery - enemy_count_display.get_height()/2))

    # Display Score
    score_card = pygame.draw.rect(WINDOW, GRAY, (int(window_width*0.85), int(window_height*0.8), int(window_width*0.15), int(window_height*
0.2)))
    pygame.draw.rect(WINDOW, BLACK, (int(window_width*0.85) - UI_border, int(window_height*0.8) - UI_border, int(window_width*0.15) + 2
*UI_border, int(window_height*0.2) + 2*UI_border), UI_border)
    score_display = largefont.render(str(self.score_count), True, BLACK)
    WINDOW.blit(score_display, (score_card.centerx - score_display.get_width()/2, score_card.top + int(score_card.height*0.3)))
    score_text = smallfont.render("SCORE", True, BLACK)
    WINDOW.blit(score_text, (score_card.centerx - score_text.get_width()/2, score_card.top + int(score_card.height*0.1)))

    # Displaying coin count
    coin_card = pygame.draw.rect(WINDOW, GRAY, (0, int(window_height*0.8), int(window_width*0.15), int(window_height*0.2)))
    pygame.draw.rect(WINDOW, BLACK, (-UI_border, int(window_height*0.8) - UI_border, int(window_width*0.15) + 2*UI_border, int(window_height*
0.2) + 2*UI_border), UI_border)
    coin_count = largefont.render(str(self.coins), True, BLACK)
    WINDOW.blit(coin_count, (coin_card.centerx - coin_count.get_width()/2, coin_card.top + int(coin_card.height*0.3)))
    coin_text = smallfont.render("COINS", True, BLACK)
    WINDOW.blit(coin_text, (coin_card.centerx - coin_text.get_width()/2, coin_card.top + int(coin_card.height*0.1)))

    # Winning the game
    if self.mode == "timed":
        if self.timer >= time_limit:
            self.state = "game_over"
            pygame.mouse.set_visible(True)
            return
    elif self.mode == "scored":
        if self.score_count >= score_goal:
            self.state = "game_over"
            pygame.mouse.set_visible(True)
            return

    # Show cursor
    cursor_x, cursor_y = pygame.mouse.get_pos()
    WINDOW.blit(cursor_img, (cursor_x - cursor_img.get_width()/2, cursor_y - cursor_img.get_height()/2))

# If player paused the game by pressing ESC
else:

    pygame.mouse.set_visible(True)

    translucent = pygame.Surface((window_width, window_height), pygame.SRCALPHA)
    translucent.fill((0,0,0,1))
    WINDOW.blit(translucent, (0,0))
```

```
            # Pause menu card
            pause_menu = pygame.draw.rect(WINDOW, BLACK, (int(window_width*0.35), int(window_height*0.3), int(window_width*
0.3), int(window_height*0.4)))
            pause_title_card = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.1), int
(pause_menu.top + pause_menu.height*0.1), int(pause_menu.width*0.8), int(pause_menu.height*0.3)))
            pause_text = mediumfont.render("PAUSED", True, BLACK)
            WINDOW.blit(pause_text, (pause_title_card.centerx - pause_text.get_width()/2
, pause_title_card.centery - pause_text.get_height()/2))

            # Continue button
            cont_button = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.2), int
(pause_menu.top + pause_menu.height*0.45), int(pause_menu.width*0.6), int(pause_menu.height*0.2)))
            cont_text = smallfont.render("CONTINUE GAME", True, BLACK)
            WINDOW.blit(cont_text, (cont_button.centerx - cont_text.get_width()/2
, cont_button.centery - cont_text.get_height()/2))

            # Quit button
            quit_button = pygame.draw.rect(WINDOW, WHITE, (int(pause_menu.left + pause_menu.width*0.2), int
(pause_menu.top + pause_menu.height*0.7), int(pause_menu.width*0.6), int(pause_menu.height*0.2)))
            quit_text = smallfont.render("QUIT GAME", True, BLACK)
            quit_text_disclaimer = tinyfont.render("Current game data will be lost!", True, BLACK)
            WINDOW.blit(quit_text, (quit_button.centerx - quit_text.get_width()/2
, quit_button.centery - quit_text.get_height()/1.5))
            WINDOW.blit(quit_text_disclaimer, (quit_button.centerx - quit_text_disclaimer.get_width()/2
, quit_button.centery + quit_text_disclaimer.get_height()/3))

            # Checking for events
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()

                # Checking for cursor coordinates
                cursor = pygame.mouse.get_pos()

                # If player presses escape, unpause the game
                if event.type == pygame.KEYUP:
                    if event.key == K_ESCAPE:
                        paused = False

                # Checking for clicks on each button and changing the state accordingly
                if quit_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        pygame.mixer.stop()
                        guncock_sound.play()
                        pygame.mixer.music.play(-1)
                        self.state = "main_menu"
                        return
                if cont_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        paused = False

        clock.tick(FPS)

def game_over(self):

    new_high_score = False
    new_best_time = False

    pygame.mixer.stop()
    if self.died:
        lose_sound.play()
    else:
        win_sound.play()
    pygame.mixer.music.play(-1)
```

```
# Updating high scores and best times if applicable
if not self.died or self.mode == "endless":
    with open("save.json", "r") as update_save:
        update_data = json.load(update_save)
        # If mode is timed, update high score if high enough
        if self.mode == "timed":
            if self.score_count > update_data["high-score"]:
                update_data["high-score"] = self.score_count
                new_high_score = True
        # If mode is scored, update best time if lower or not set yet
        elif self.mode == "scored":
            if update_data["best-time"] is not None:
                if self.timer < update_data["best-time"]:
                    update_data["best-time"] = self.timer
                    new_best_time = True
            else:
                update_data["best-time"] = self.timer
                new_best_time = True
        # If mode is endless, update score or time if higher (extra section in json file)
        elif self.mode == "endless":
            update_endless = [False, False]
            if self.score_count > update_data["endless"]["high-score"]:
                update_data["endless"]["high-score"] = self.score_count
                new_high_score = True
            if self.timer > update_data["endless"]["best-time"]:
                update_data["endless"]["best-time"] = self.timer
                new_best_time = True
    # dump this updated data into json
    with open("save.json", "w") as update_save:
        update_save.write(json.dumps(update_data))
with open("save.json", "r") as update_save:
    save = json.load(update_save)
    save["coins"] += self.coins
with open("save.json", "w") as add_coins:
    add_coins.write(json.dumps(save))

while True:
    pygame.display.update()

    # bg colour
    WINDOW.blit(menubg, (0,0))

    # title
    title_card = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.3), int(window_height*0.1), int(window_width*0.4), int(window_height*
0.15)))
    title_text = mediumfont.render(self.mode.upper() + " MODE", True, BLACK)
    WINDOW.blit(title_text, (title_card.centerx - title_text.get_width()/2, title_card.centery - title_text.get_height()/2))


    # Image of player either dead or holding a trophy
    if self.died:
        WINDOW.blit(player_dead_img, (int(window_width*0.2), int(window_height*0.4)))
        WINDOW.blit(largefont.render("YOU DIED", True, RED), (int(window_width*0.55), int(window_height*0.35)))
    else:
        player_menu_icon = WINDOW.blit(player_menu_img, (int(window_width*0.2), int(window_height*0.4)))
        WINDOW.blit(pygame.transform.rotate(trophy_img, (-30)), (player_menu_icon.centerx, player_menu_icon.centery))
        WINDOW.blit(largefont.render("YOU LIVED", True, GREEN), (int(window_width*0.54), int(window_height*0.35)))

    # Score/time from this game
    if new_high_score:
        WINDOW.blit(smallfont.render("New High Score!", True, RED), (int(window_width*0.55), int(window_height*0.5)))
    score_text = WINDOW.blit(mediumfont.render("Score", True, BLACK), (int(window_width*0.55), int(window_height*0.525)))
    score_display = mediumfont.render(str(self.score_count), True, BLACK)
    WINDOW.blit(score_display, (score_text.centerx - score_display.get_width()/2, int(window_height*0.61)))
    if new_best_time:
        WINDOW.blit(smallfont.render("New Best Time!", True, RED), (int(window_width*0.71), int(window_height*0.5)))
    time_text = WINDOW.blit(mediumfont.render("Time", True, BLACK), (int(window_width*0.72), int(window_height*0.525)))
    time_display = mediumfont.render(str(self.timer) + "s", True, BLACK)
    WINDOW.blit(time_display, (time_text.centerx - time_display.get_width()/2, int(window_height*0.61)))

    # Piggy bank
    piggy_icon = WINDOW.blit(piggy_img, (int(window_width*0.05), int(window_height*0.7)))
    coins_text = smallfont.render(str(save["coins"]), True, BLACK)
    WINDOW.blit(coins_text, (piggy_icon.centerx, piggy_icon.centery - coins_text.get_height()/2))
    earned_coins_text = mediumfont.render("+" + str(self.coins), True, BLACK)
    WINDOW.blit(earned_coins_text, (piggy_icon.centerx - earned_coins_text.get_width()/2, piggy_icon.top - earned_coins_text.get_height()))
```

```
# Back to menu button
            menu_button = pygame.draw.rect(WINDOW, WHITE, (int(window_width*0.55), int(window_height*0.75), int(window_width*0.25), int
(window_height*0.15)))
            menu_text = smallfont.render("BACK TO MAIN MENU", True, BLACK)
            WINDOW.blit(menu_text, (menu_button.centerx - menu_text.get_width()/2, menu_button.centery - menu_text.get_height()/2))

            # Checking for events
            for event in pygame.event.get():
                if event.type == QUIT:
                    pygame.quit()
                    sys.exit()

                # Checking for cursor coordinates
                cursor = pygame.mouse.get_pos()

                # Checking for clicks on each button and changing the state accordingly
                if menu_button.collidepoint(cursor):
                    if event.type == MOUSEBUTTONDOWN:
                        guncock_sound.play()
                        self.state = "main_menu"
                        return

            clock.tick(FPS)

    def run(self):

        match self.state:
            case "main_menu":
                self.main_menu()
            case "options":
                self.options()
            case "shop":
                self.shop()
            case "htp":
                self.htp()
            case "dif_mode_menu":
                self.dif_mode_menu()
            case "theme_menu":
                self.theme_menu()
            case "main_game":
                self.main_game()
            case "game_over":
                self.game_over()
            case "quit":
                pygame.quit()
                sys.exit()

game = GameState()
while True:
    game.run()
```

END OF COURSEWORK