Contents

[Analysis 2](#_Toc81851927)

[Problem Identification: 2](#_Toc81851928)

[Stakeholders 2](#_Toc81851929)

[Research on the problem 2](#_Toc81851930)

[Limitations 2](#_Toc81851931)

[First set of questions to ask stakeholders (I will add results in next draft / final) 2](#_Toc81851932)

[Second set of question to ask stakeholders (I will add results in next draft / final) 2](#_Toc81851933)

[Requirement List 2](#_Toc81851934)

[Design 2](#_Toc81851935)

[Aesthetics (to be done later, will either make them myself or use free assets) 2](#_Toc81851936)

[Menu 2](#_Toc81851937)

[Pause Screen 2](#_Toc81851938)

[On Screen 2](#_Toc81851939)

[UI 2](#_Toc81851941)

[Player Sprite 2](#_Toc81851942)

[Enemy Sprites 2](#_Toc81851943)

[Core Algorithms 2](#_Toc81851944)

[‘Input and Controls’ – Player [Movement] 2](#_Toc81851945)

[‘Input and Controls’ – Player [Combat] 2](#_Toc81851946)

[‘Input and Controls’ – Menu and Settings 2](#_Toc81851947)

[‘RNG’ – Map Generation 2](#_Toc81851948)

[Room Doors (Opening) 2](#_Toc81851949)

[2](#_Toc81851950)

[Room Doors (Closing) 2](#_Toc81851951)

[‘Camera’ Movement 2](#_Toc81851952)

[‘RNG’ – Enemies 2](#_Toc81851953)

[‘Visuals’ – Character animations 2](#_Toc81851954)

[‘Audio and Sound’ 2](#_Toc81851955)

[Classes 2](#_Toc81851956)

[Player 2](#_Toc81851957)

[Bullet 2](#_Toc81851958)

[Sword 2](#_Toc81851959)

[Enemy Type 1 (Random movement and bouncing) 2](#_Toc81851960)

[Enemy Type 2 (Following player and melee attack) 2](#_Toc81851961)

[Enemy Type 3 (Firing from range) 2](#_Toc81851962)

[Boss Type 1 (An ‘extension’ of Enemy Type 1) 2](#_Toc81851963)

[Boss Type 2 (‘Charges’ at player) 2](#_Toc81851964)

[Boss Type 3 (Teleporting and shooting) 2](#_Toc81851965)

[Key Variables 2](#_Toc81851966)

[Player 2](#_Toc81851967)

[Enemy 2](#_Toc81851968)

[Player and Enemy projectile / bullet 2](#_Toc81851969)

[Map 2](#_Toc81851970)

[Other 2](#_Toc81851971)

[Test Data 2](#_Toc81851972)

[General Test Data 2](#_Toc81851973)

[More Specific Test Data 2](#_Toc81851974)

[Implementation 2](#_Toc81851975)

[Stage 1 2](#_Toc81851976)

[Coding 2](#_Toc81851977)

[Stage 1 – Review 2](#_Toc81851978)

[What was achieved: 2](#_Toc81851979)

[Requirements met: 2](#_Toc81851980)

[Testing and fixing: 2](#_Toc81851981)

[Changes to requirements and/or design: 2](#_Toc81851982)

[Stage 2 2](#_Toc81851983)

[Coding 2](#_Toc81851984)

# Analysis

## Problem Identification:

With the advances in technology, video games not only have been increasingly getting more and more impressive throughout the years, but they are also becoming much more popular. Most people that I know all play video games. These games range from just simple mobile games to story-driven single player games and competitive online multiplayer games. Especially with the current pandemic situation, online multiplayer games have been a very popular solution to socialising with others while stuck at home.

However not everyone has access to these online games. This is usually because of hardware limitations, as the most popular multiplayer games require a console or a pc capable of running these games, which can be expensive for many people. Even multiplayer mobile games can require the newest phone. For these people, they would probably only have a mobile and a laptop of some sort. This means single player games seem to be the reasonable solution to this.

But now to the main problems, the first being the difficulty in these single player games. Games now, especially ones developed by larger companies, do not implement difficult as well as they should. They are either too simple, to the point where there is no challenge and enjoyment, or they are too difficult, where the player is severely punished for small mistakes. Many games have tried giving the player the ability to choose the difficulty, but this attempt still suffers from the same problems. For example, to increase difficulty, the enemies damage and overall health is increased. This is what a lot of developers do, instead of also making unique changes that challenges the player’s skill. The second problem is the gameplay and its replay value, which is a result of badly implemented difficulty. Gameplay can become very repetitive and tedious when unique changes are not made when progressing through the game. If the game is too easy, it will become boring, as there is not sense of satisfaction. If the game is too difficult, it will become annoying and tiresome. With both situations, it usually ends with the player giving up.

My proposal is to make a single player game (of the rogue-like genre) where a high score is acquired. This is so that you can compete with friends and make it more rewarding. It will also be simple in the beginning, but slowly increase in difficulty as you progress, by making the mechanics of the player and enemies more complex. So, it should be fun but also challenging and rewarding.

## Stakeholders

My stakeholders are a group of other students at the age of 16 and 17, who have an interest in games and have access to a basic laptop or computer. They would want this game to be challenging but rewarding, and not overly repetitive. They would also want a way of checking a leader board, to see the score that others achieved, so that they can compete. Most of them also don’t care about having a story, so I will keep that to a minimum.

(I will also ask these people questions, to gather some basic requirements for the game, and maybe some other additions that improve on existing requirements.)

## Research on the problem

There are many games that suffer from the problems of difficulty and gameplay repetitiveness:

*‘The Division’*



This is an RPG game, with the main process being on levelling up your character with skills and with better weapons and equipment. To level up your character, the game provides missions to be completed. These missions have different map designs and different enemy types. They can be completed with not just weapons, but also different equipment, such as grenades.

However, even though this may seem like any other decent game, this particular game isn’t very popular. One reason for its negative reviews is because of how repetitive the gameplay is and the lack of variety. According to the majority of the players, the level designs are all very similar and the enemy designs aren’t unique. For example, when the difficulty of the game increases, the enemies only have stat changes. This means only the health and damage of the enemies are increased, instead of introducing more unique abilities. In addition, because of this repetitiveness and lack of variety, there is no incentive to level up your character, which is why many people became bored of it very quickly.

The loot system also suffers from problems, as item drops are weekly based, meaning good items can only be acquired once a week. In addition, the game allows you to upgrade your current weapons, but takes a very long time due to the number of materials you need. This makes the game seem like a chore to play.

What to avoid after researching this game:

* Repetitiveness
* Basic character designs
* Lack of variety

*‘Assassin’s Creed Odyssey’*



This is another RPG game, which suffers from very similar problems as the last. Playing the main story missions in this game is the best way to level up your character and gain better gear. However, the developers have designed it so that you have to be a certain level to play these missions. In addition, they have made it very difficult to level up. A very little amount is gained after each mission; these missions don’t provide a high enough reward for the player.

Because of this, to level up, players are forced to play the side missions. This wouldn’t be a problem if the side missions had good design, however they are flawed. These side missions are extremely repetitive, with very similar map layouts and objectives. This would make playing the game boring, as no changes are being made, even when the players are forced to play many of these missions just to level up.

In addition, there is almost no enemy variety. The main way the enemy types differ is by the amount of health and damage they do. Most of the enemies have no unique abilities, that would bring variety to gameplay.

What to avoid after researching this game:

* Unrewarding gameplay
* Simple enemy designs
* Lazy difficulty increasing

*‘Marvel’s Avengers’*



This game, like the other two, rely on levelling up your characters. However, it also suffers from the exact same problems.

The game has very simple enemy designs, where enemies all have very similar mechanics. In addition, bosses in most games should be unique and challenging. However, in this game, they are basic and repetitive. Again, difficulty is increased only by increasing their total health and damage. The map designs are bland and repetitive. The layout of each level is very similar and also quite linear, with no sense of exploration. The gameplay itself is also dull, with not much variety in its mechanics. This is also the newest game with the better graphics, but it is the worst reviewed out of these three.

Like the previous game, it is not very rewarding. Very little is given to the player for completing levels. This would just discourage people from playing their game.

What to avoid after researching this game:

* Repetitiveness
* Unrewarding gameplay
* Bland level design and character design
* Lack of variety in mechanics and abilities

 But there are also many games that implement difficulty and gameplay very well:

*‘Risk of Rain 2’*



This game is a rogue-like, which is the same genre as what I’m planning to make. This game is extremely popular, likely because of its unique design choices. Firstly, it has an interesting take on difficulty, as its uses ‘scaling difficulty’. As you progress through the game and visit the different levels, the difficulty will steadily rise overtime, making enemies stronger and more numerous. This difficulty change is completely dependant on in game time, and not on which level the player is currently in. And so, the difficulty increases in two ways, with enemy stat increase and also more difficult enemy types are introduced in each level. Though if it were just this difficulty increase, the game would inevitably become impossible. The developer’s solution to this is that after each level, the player’s stats are also increased. In addition, the way to acquire items and abilities in this game is through collecting money by killing enemies, and through drops by killing bosses. Every time the difficulty increases, the amount of money and item drops you get also increases.

With this unique design, the player is always just under the level of the enemies. This means that the game doesn’t become overly difficult (if player right) and not overly easy, which is what the previous heavily suffered from. In addition, there are many different enemies and bosses, and also playable characters. This provides lots of replay ability.

As it is a rogue-like, everything is randomised. This means each time you play the game, it will always be different, due its variety in playable characters and abilities.

Features I should keep in mind:

* Correctly scaling difficulty
* Rewarding gameplay
* Other playable characters

*‘The Binding of Isaac’*



This is another rogue-like, with a huge collection of items, enemies, and playable characters. Like the previous game, the player progresses by defeating enemies and completing levels. There is also a money system in this game, however, unlike the other game, it is given out very rarely. This makes currency in this game quite important, as it is how you can acquire new abilities and extra health. The game compensates for this lack of currency by giving out items to the player for free. Each level is guaranteed a free item. This system makes the game challenging, but also not too punishing. The point system is also very unique. The more time you spend on a level, the less score you get at the end of it. This creates pace, making it challenging to achieve a high score.

The core gameplay is very basic. You are able to move and shoot. Even though this may seem very simple, there are also many other extra mechanics, which makes the gameplay more interesting. For example, you can place bombs down to break down hidden walls, which can then give you access to free items. You have different types of ‘minions’, which can assist you during the game. These are extremely helpful once the difficulty starts to increase.

In addition, there are many challenges in this game. Completing these challenges gives you access to other playable characters. Together with the random map generation and enemy placement, it makes the game very replay able; every ‘run’ is different.

Features I should keep in mind:

* Challenges
* Unique score system
* Variety of items and abilities

*‘Dark Souls’, ‘Hollow Knight’*





Like the other games, these have their own unique take on difficulty. These ‘souls-like’ games are notorious for their difficulty. These games were made to punish players for mistakes, which would result in multiple attempts in trying to defeat the boss. Each boss is very different from the other, and the smaller enemies that come before it hint at the mechanics and abilities of the boss. Even though this game can be extremely difficult, especially for new players, it rewards the player through satisfaction and progressing into the next area of the open-world map.

From this research altogether, I have gathered features that I should try implement into my own game:

* Modern graphics are not a necessity for a game to be good.
* There does not have to be a story for the game to be interesting.
  + Gameplay mechanics are very important, they need a level of complexity so that the player can learn.
* Bosses should be unique.
* The game should require a certain amount of skill.
* Mistakes should be punished, but not too heavily.
* Players need to be rewarded for completing a difficult task.
* Do not increase difficulty by just increasing enemy health and damage.

## Limitations

For this project, I have assumed that most people will have a basic laptop or computer of some sort, or at least have access to one. Also, that most people will know how to use these devices or will be able to use these devices (e.g. if they have a disability that prevents them from doing so).

Therefore, I will attempt to make the basic controls straightforward, and the images on screen clear and simple.

## First set of questions to ask stakeholders

With these questions, I wanted to find out what they liked in games in general, and what they disliked. I want to make sure that these disliked features are removed from my game / not included in my game.

* How many different game genres have you played?
* What are the genres you enjoy the most?
* What are the most important features in a game?
* Have you played a rogue-like before?
* If previous answer is yes, what features of it did you like?
* What features of it did you dislike?
* What features would you like to have added?

What I have gathered from these questions:

Most people have played around 4 different game genres, which includes a combination of Platformers, Tower-Defence games, Shooters, Dungeon Crawlers, Racing games and others. This is helpful, as it shows most people like to try different games. Most importantly, this means the feedback I will receive from the stakeholders will not be biased towards one genre, and instead will be more generalised.

The genres that these stakeholders enjoyed the most were, with the highest number of votes, Shooter, Platformer and Dungeon Crawler games. This is good as my game is a rogue-like dungeon crawler game.

The important features they mentioned were replay ability, rewarding gameplay and creative level design. These are things I should attempt to include in my game.

Most of the stakeholders have not played a rogue-like before, but from those that have, they mentioned that there needs to be variety in gameplay to prevent it from being repetitive, which is something that can happen very easily to rogue-like games. This means there should be a range of different items, and possibly different characters to choose from too. The features that they wanted to add to these rogue-like games was a better difficulty scaling system; they found the player would become too ‘underpowered’ as they progress in some games. This is something I would need to try avoiding.

## Second set of question to ask stakeholders

With these questions, I wanted to ask multiple choice questions, to find preferences on more specific factors of this game. Most of my main requirements will be decided from this questionnaire.

* How would you like to control your character?
* Keyboard
* Mouse
* Mouse and Keyboard

* What perspective do you prefer?
* Top down
* Side on + Top down (oblique)

* Would you want to be able to play different characters?
* Yes
* No

* How many different characters would you want?
* 3
* 4
* 5
* > 5

* How do you want to unlock these new characters?
* In game currency
* Final game score
* Challenges

* Would you like a character with a melee weapon, instead of a ranged attack?
* Yes
* No

* How long would you want it to take to complete one level?
* 2-3 minutes
* 5 minutes
* 10 minutes
* > 10 minutes

* How many rooms should there be in each map?
* 5
* 6
* 7

* How would you like currency to be earned?
* At the end of each level
* After defeating an enemy

* How long should it take to defeat a boss?
* 1 minute
* 2 minutes
* 3 minutes
* > 3minutes

* How steep do you want the learning curve to be?
* Low
* Medium
* High

* Would you like music in the game?
* Yes
* No

* What type of colours would you prefer?
* Dull
* Vibrant

What I have gathered from these questions:

1. People preferred either keyboard or keyboard with mouse, no one preferred mouse only.
2. Most people preferred an oblique perspective, but it was close to half-half.
3. Nearly everyone wanted to have the option to play different characters.
4. Most wanted number of characters was 3-4.
5. The systems of unlocking characters with high scores and challenges were the most popular.
6. Choice of melee character was close to half-half.
7. 2-3 minutes was the most popular, with 5 minutes coming second.
8. 5-6 rooms was the most popular.
9. Earning currency after defeating an enemy was more preferred.
10. 2 minutes to defeat a boss was more preferred.
11. They preferred a medium learning curve.
12. Music was half-half.
13. People much preferred vibrant colours to dull ones.

## Requirement List

Three most important features for this rogue-like: procedurally generated environments, permanent death, and randomisation of item identities.

Must:

[1.0] Create a way to randomise the location of rooms.

[1.1] Create different starting characters (different starting weapons).

[1.2] Movement, with diagonal movement.

[1.3] A stamina bar (sprinting and stamina regeneration).

[1.4] Aiming with the mouse (implementing attacks with mouse buttons).

[1.5] Create a variety of enemies with different mechanics.

[1.6] Create abilities that have unique mechanics / additions to existing mechanics.

[1.7] A money system (how much is earned depending on enemy and level).

[1.8] A simple levelling system.

[1.9] A pause menu.

[2.0] A HUD with current score, money, health, stamina etc.

[2.1] A camera that locks to the room in which the player is in.

[2.3] Create a menu screen.

[2.4] Implement images for the levels.

[2.5] Randomise bosses.

[2.6] A leader board of different people’s score.

[2.7] Add a timer to record game time.

[2.8] Animations for characters.

[3.1] Ability to go through doors to rooms off-screen.

[3.2] Run the game with minimal stuttering.

[3.3] Randomise location of items and type of items.

Should:

[2.2] Include sound effects.

[2.9] Mini map.

[3.4] A cutscene at the start.

[3.5] Include a soundtrack.

[3.6] A way to check controls / how to play.

Could:

[3.7] Make death animations (mainly for bosses).

[3.8] Make a logo.

[3.0] In game challenges.

[3.9] Make a demo, for viewing purposes.

Hardware and Software Requirements

Will be run in an IDE with Python 3.8

* RAM : 4GB
* Disk space : ---
* OS : OS X 10.11 or Windows 7 (or above)
* CPU : Intel Core i3 (1.5Hz or faster)
* Mouse, keyboard

# Design

To simplify the overall structure of the game and the list of requirements mentioned in the analysis stage, I have made a flowchart. This will help break down each section that will need to be worked on and show the main requirements of each aspect. I will be using Python and Pygame to develop the game, making use of Object-Oriented Programming.



This is what will be the most important aspects to get done for this game:

1. Input and Controls

This will be the priority, as this acts as the base for the game.

* Algorithms for movement and collisions with walls.
* Algorithms for combat (this will include a gun, sword, shield etc.) and collisions with enemy sprites.
* A basic menu screen and pause screen.

1. RNG

This will also be a priority, as this is the core feature of a ‘roguelike’. But it will also be the most difficult aspect to code.

* Algorithms for map generation.
* Making sure the levels are navigable.

1. Visuals and Aesthetics

This is something I plan to work on later, but it is still very important. Map and character designs is what brings the game to life.

* Animations for the player character and enemies (at least 4 enemy types and at least 4 different boss types).
* Make the existing menu more aesthetically pleasing.
* Adding environmental animations (e.g. cloth and grass physics), though this is a low priority requirement.
* Cutscenes (to start the game and to introduce bosses), this is also low priority.

1. Audio and Music

Sound effects and background music will have a lower priority than everything else on the flowchart, and syncing sound effects may be quite tedious.

* Sound effects for weapons and abilities.
* Footsteps etc.
* Music for the different environments.
* Boss music.
* Balancing all these sounds.

To also make the implementation stage easier,  I plan to split the code into different files. For example, I will have a file for the main game functions, a file for maps and a file for classes e.g.



I will link these files together using *import*:



## Aesthetics (to be done later, will either make them myself or use free assets)

### Menu



### Pause Screen



### On Screen

### 

### UI



### Sprites

I plan to experiment with different sprites that can be found online on websites such as Itch.io, which provides a lot of free assets. Here are some tilesets that I could use for my rogue-like:



A tile set with floor tiles, wall tiles and also decoration tiles.

Link: https://analogstudios.itch.io/dungeonold



A tile set with different chest types.

Link: https://greatdocbrown.itch.io/coins-gems-etc



A tile set with player and enemy sprites, which also includes the different sprites needed for animating.

Link:

https://game-endeavor.itch.io/mystic-woods



## Core Algorithms

### ‘Input and Controls’ – Player [Movement]

**Movement with WASD ( requirement [1.2] ):**



This will go in the game loop, and the function from the player class that will work with this is:



Explanation:

The line *pygame.key.get\_pressed()* is used because it gets the state of all keyboard buttons. It will return Booleans values that represent each key. If a key is pressed down, it will return *True*. To make the code clearer, I have called this instruction *keys*.

The line *if keys[pygame.K\_a]*is an if statement checking whether the letter ‘a’ has the value *True* returned by the instruction *keys*. If the value is *True*, this means the ‘a’ key has been pressed on the keyboard, and it will now execute the next line, *player.move(-1, 0)*.

In the function *move*, it has values *x\_val*and *y\_val* and these are substituted by the values in *player.move*, which in this example is (-1,0). By pressing ‘a’, the player sprite will move by a value of -1 in the x-axis, which is to the left of the screen.

**Collisions with Walls**



Explanation:

When the player collides with a wall, I want the sprite to stop in place. So with this code, when a collision happens, the player’s speed will be (0,0), meaning there will be no movement in the x and y axis. In addition, to prevent the player from merging into the wall sprite, I added that the sprite will remain at the position it collided at, which are the last two lines. This was a problem I encountered in a previous project, which I now know how to fix.

**Stamina ( requirement [1.3] ):**





Explanation:

I have a variable *stamina* which has the value 150; this represents the total amount of stamina the player has. The *if* statements in the second picture will be in the game loop. This works the exact same way as the previous code, except now there is *stamina*. The statement checks that a movement key is pressed, the stamina key is pressed (which is temporarily set as ‘j’). I also plan to some lines to check that the stamina count is above 1.

e.g.



When all these requirements for the *if*statement are met, the player sprite will move just like before, but now four times the speed. I will change the values of the ‘Move’ function.

e.g.



In addition, the total stamina is subtracted by 2 and these keys are being held down, which represents the depletion of stamina. But if the stamina key is being pressed, then the stamina will start to add 1 to itself. This is done by the code at the very bottom of the screenshot.

### ‘Input and Controls’ – Player [Combat]

**Gun / Projectile Weapon:**





Explanation:

This code, which will be in the game loop, makes use of existing pygame instructions, most importantly *event.type == MOUSEBUTTONDOWN*and *pygame.mouse.get\_pos()*.

In the first picture, the *if*statement checks whether the left click of the mouse (*event.button == 1*)and been pressed or not. If it has been pressed, then the variable *click* will become true. When *click*is true, the bullet will be shot. The instruction *pygame.mouse.get\_pos()* takes the x-coordinate and y-coordinate of where the mouse has been clicked. These two values are then used to draw the *Bullet* sprite (the class will be shown later). The bullet sprite will move towards these coordinates, starting from the centre of the player sprite.

Unfortunately, after testing this algorithm, a problem came up with this code. It will need to be worked on during the Implementation stage. If another key is pressed while shooting, *click* will no longer be *True*.

Also, I will need to able to alter the fire rate, as at the current moment it is infinite. I could use a timer in pygame to do this, though I have not tested it:



**Sword - pseudocode:**

Explanation:

With this pseudocode, I intend to make the process slightly simpler by treating the sword as a short-ranged bullet with no velocity when drawn.

However, a problem that I can see occurring will be with the collisions. There could be many collisions detected in one sword swing. This is something that I will need to test and it is something that will likely require some fixing.

### ‘Input and Controls’ – Menu and Settings

**Menu Screen – pseudocode:**



Explanation:

All of this will be under a function called ‘menu()’.

There will be three buttons: one to start the game, one to quit the game and one to go to the settings page. There will also likely be a credits page, for artwork and music. The main feature of this main menu will be the buttons. When the buttons are drawn onto the screen (this is not done in the pseudocode), it should be able to detect whether the cursor has clicked it or not.

After some research, I found ‘collidepoint()’, which returns true if the given point inside the rectangle, where the given point is the coordinates of the cursor.

Example python code would be:

if start\_button.collidepoint( (x, y) ):

if click:

game()

**Pause Screen**



Explanation:

This will be a simple if statement, where the pause screen will activate when the ‘escape’ key is pressed. The game and game timer will also be paused, until the user exits the pause screen.

The actual pause screen will be very similar to the menu screen, with buttons for settings and to quit the game.

### ‘RNG’ – Map Generation

 Explanation:

This is very rough pseudocode, with logic that I have gathered from research during the analysis stage. The idea is that each level will be on a 9x7 grid, which will be represented by a 2D array, with rows and columns. The spawn room will be blank, at it will always start in the middle of the grid, which in this example is at [4][3] in the array. From there, there will be a random cardinal movement to another cell in the grid. So, it will either choose the cell above, below, to the right or to the left of the middle cell. As this second cell will always be empty, a random room will be drawn. I plan to create a function for this drawing, roomcreation(), which will use the screens sizing to draw a border and other shapes inside the room. This will also make use of a 2D array. This is an example from a previous project, which I will likely alter and then implement in this game too:



From this second cell, it will move to another cell. But if the cell it moves to already has a room in it, it will move again from that occupied cell. This feature is the while loop in the repeat until loop. I have done this so that dead-ends are more likely to be created. Levels will be created until the requirements are met for one of them, which are 3 dead-ends and the number of rooms. After this, the special rooms are randomly assigned to the dead-ends.

### Room Doors (Opening)

### 

Explanation:

With my map design, I plan to have these door sprites always on screen, in the same positions. It will not move with the rest of the map (explained later in ‘Camera’). When the spawn room is drawn, these doors are drawn straight after, in the position of 2, 3, 4 and 5 in the map array: 

When the player sprite collides with any of these doors, the game will progress to the next room, depending on the direction. For example, if the player collides with the door on the right, the player will move into the room to the right.

### Room Doors (Closing)







Explanation:

As I planned to have the doors always open on screen, I needed a way to ‘close’ these doors when there are no adjacent rooms. For example, if there is no room above, then the top door needs to be ‘closed’. I created a function that checks the room the player currently is in, and then also checking if any rooms are adjacent to it in all directions (top, right, bottom and left). Where there is an adjacent room, wall sprites are drawn to cover the open doors, so that the player cannot collide with the doors. This makes use of another array, which acts as a grid to show where each room is:



‘mapx’ and ‘mapy’ represent the coordinates of the middle of the map (marked 1)

### ‘Camera’ Movement

or



Explanation:

The screen will have a size of 1280x720. For the first pseudocode, I thought that I could create a Camera class, with a procedure built in that will constantly run; like an update function. This procedure will be able to use the given coordinates, and track the player, keeping the player sprite in the middle.

However, this seems like it would be very difficult to do, and so I also wrote an alternative. This code will be in the game loop, and work with the door sprites. Depending on the direction of the door the player sprite collides with, all sprites in the ‘wall\_group’ will move in that direction. For example, if the player enters the right door, the wall sprites will all move to the left, with a distance of the width of the screen (1280 pixels).

The player sprite will also be teleported to the left, to make it seem like it has moved through the door to the next room. This can be done very easily. For example, in pygame the following code can be used:



### ‘RNG’ – Enemies

**Spawns**



This code allows sprites to be spawned randomly around the map, depending on which level / room the player sprite is in. This would work; however, I would also like to randomise the number of enemies spawned and have the ability to change the range of numbers depending on the level. For example, in the first level, 1-3 enemies would be spawned in a room. In the next level, 2-4 enemies would be spawned in a room etc.

I will also create separate functions like this for each enemy type.

**Movement**

Depending on the enemy type, they will either move randomly (shown in the Classes section under ‘Enemy Type 1’), move towards the player, or stay a specific distance from the player.

For enemies that move towards the player, I will attempt to implement the A\* Algorithm or Dijkstra Algorithm. But if this becomes too difficult, I will use breadth first searching, as the actual rooms / maps will be quite small.

This is pseudocode for basic pathfinding:



Explanation:

The second line of code calculates the vector between the enemy sprite and player sprite. These two values are then used to calculate the shortest distance in a straight line, which is done in the third line. In python, I will be able to use the math module and use math.hypot to calculate the hypotenuse of this ‘triangle’. The fourth line converts the vectors into unit vectors. These values are used in the last two lines for the enemy’s movement.

However, this pseudocode will not consider any objects that will be blocking the enemies’ path. I plan to use this algorithm first to test the game (it will likely be used for an alpha version), and then implement a better solution for path finding, such as Dijkstra’s Algorithm, which will be able to calculate the shortest path even with obstacles in the way.

### 

### ‘Visuals’ – Character animations



Explanation:

Depending on the player sprites movement state, different pictures will be loaded in to animate these movements. For example, when the player sprite is not moving / in idle, the pictures for this idle state will be loaded in. This is shown by the variable ‘idle’ in the game loop, where if none of the movement keys are being pressed, idle will stay True. The if statements outside of the game loop will determine which images to load in, and there will be multiple images for each movement state.

However, there can be improvements to be made to this code, as multiple images being loaded in at once may slow down the game. This will require testing, which will show whether the code will need to be changed or not. The change could be to pre-load the images outside of the game loop and call these animations when necessary.

### ‘Audio and Sound’



Explanation:

Depending on which level the player is on, a different music track will be playing in the background. But when they enter a boss room, new music will start playing, and the previous track will be paused until the boss fight is done.

To avoid the problem of having the song end before finishing the level, pygame has a useful feature ‘pygame.mixer.music.play()’. You can use this to determine how many times a file will repeat, by changing the number in the bracket. By putting 0, it will play once and by putting -1 it will repeat infinitely. I plan to have each song play on repeat and stop it when there is a change in level or a transition into a boss room. I could also make use of ‘pygame.mixert.fadeout’, which gradually fades out the volume of a sound, so that music does not abruptly end.



Explanation:

The pseudocode above will be used for sound effects, such as weapons and walking. This is very similar to the code for playing music.

Though I will also need to change the volumes for sound effects and music during the implementation stage, so that everything is balanced out and not too overwhelming.

## Classes

### Player



This is a simple player class, with values that I can easily change if needed. It also includes the movement function, which works by adding or subtracting from the sprites x and y position. This is also where the player’s attributes such as health, speed, damage, and stamina will be. This is not in the current class above.

### Bullet



This class makes use of the python *math* module to calculate the angle in radians between two different points. In this example, it calculates the angle between the centre of the player sprite and the coordinates of the mouse click. Using the parameters *targety*, *targetx*, *posy* and *posx*, it finds the angle between the player sprite and mouse cursor in radians. With this value, it then calculates the gradient of the x and y line, which are defined as *dx* and *dy*.

In the update method, it uses these the values *x, y, dx* and *dy* to direct the movement and direction of the bullet.

### Sword



As my design of the sword is just a short-ranged bullet sprite, the class is almost identical, except for the method used.

The method ‘strike’ is the same as the bullets update method, but it also includes the use of a timer. After half a second after the sprite is drawn, it is deleted. In pygame, this can be done with ‘pygame.time.get\_ticks()’, which returns the number of milliseconds since pygame.init() was called. This is then used with the sword timer, which starts whenever the sprite is drawn.

By subtracting the sprite timer from the game timer, you can find how long the sprite has been active for and then delete it when it is at 500 milliseconds.

An example would be:

def strike()

if pygame.time.get\_ticks() – self.time >= 500:

self.kill()

### Enemy Type 1 (Random movement and bouncing)



This enemy type will shoot in all directions, with a specific fire rate. For example, one shot every 2 seconds. This enemy type will be constantly moving around the map, in random directions. It will also be able to bounce off walls.

However, with the code above, the bounces off the walls is completely random. This means that the rebounds aren’t predictable, which I think could be a problem when there are multiple of this enemy type in one room.

This ‘update’ function will require improvements during the implementation stage.

### Enemy Type 2 (Following player and melee attack)

Pathfinding has already been covered in the Algorithms section, and the main class is also the same as Enemy1 (the attributes). This enemy type will rely on collisions, with an algorithm in the game loop.



### Enemy Type 3 (Firing from range)



This class acts in a similar way to the Player class and its shooting method. But instead of having the cursor as the target and the player sprite as coordinates where its drawn, here the target is the player sprite, and the origin is the enemy sprite. I will also try to implement movement to increase difficulty as the player progresses.

When the arc tangent is calculated in radians (of the slope / hypotenuse), the class will cosine the angle to find the change in x and sine the angle to find the change in y. These two values are then multiplied by speed, to calculate the movement of the bullet towards its target. The calculations are done outside of the update class, so that the bullets will not keep following the player sprite after they are fired, it will only go towards the coordinates the player was previously at when the enemy shot.

I will also use a timer, to determine the enemy’s fire rate, which can be done by:



This command above sets an event type to appear on the event queue every given number of milliseconds. And it can be used with:



This code states that for every given number of milliseconds, it will call the enemy shoot function. This is how I plan to control enemy fire rate.

### Boss Type 1 (An ‘extension’ of Enemy Type 1)



This is pretty much the same as Enemy Type 1, just with different values and a health bar. However, I do plan to add to this, such as adding a new attack to create more variety. It also suffers from the same random rebounds, which I need to fix. Again like Enemy Type 1, I want to correct its movement with wall during implementation.

### Boss Type 2 (‘Charges’ at player)

This will actually be very similar to Enemy Type 3’s shooting mechanic, except it is the boss itself that is moving in the direction of the player’s previous coordinates. The only changes would be the speed, and the boss will also occasionally spawn random enemies in the room, which I have also covered in the Algorithms section.

When the boss sprite collides with the player sprite, the player will receive damage and also be pushed back a little bit, to prevent multiple collisions from happening.

### Boss Type 3 (Teleporting and shooting)

I will again focus on the methods, as the main attributes will pretty much be the same. I plan to have to of this type of boss for one fight.



This boss, like Enemy Type 3, makes use of the pygame timer. This is used to determine how frequently the sprite will teleport around the room. As the player progresses through the levels, I can increase the frequency to make the game more difficult. This boss also shares the same shooting mechanic as Enemy Type 3 and also a similar class to it, as it will stand still.

## Key Variables

### Player

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Health | Integer | Total health of player |
| Stamina | Integer | Total stamina of player |
| Damage | Float | Total damage of player |
| Speed | Float | Movement speed |
| Attack Rate | Float | Attack delay / fire rate |
| Score | Integer | Total score |
| (Lives) | Integer | Total lives |

### Enemy

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Health | Integer | Total health of enemies |
| Speed | Float | Movement speed |
| Damage | Integer | Total damage |
| Attack Rate | Float | Attack delay / fire rate |

### Player and Enemy projectile / bullet

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Width | Float | Width of projectile |
| Height | Float | Height of projectile |
| Speed | Integer | Speed of projectile |

### Map

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Mapx | Integer | Position of room (x axis) on grid |
| Mapy | Integer | Position of room (y axis) on grid |
| Number | Integer | Number of rooms per level |

### Other

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Size | Integer | Screen size |
| Pause | Boolean | If game has been paused or not |
| Done | Boolean | If game has finished or not |

## Test Data

### General Test Data

#### Player

* Player sprite can move right
* Player sprite can move left
* Player sprite can move up
* Player sprite can move down
* Player sprite can move diagonally in all directions
* Player sprite will collide with map sprites
* Player sprite will collide with enemy sprites
* Player sprite will take damage when colliding with enemy sprites
* Player can shoot in all directions using mouse
* [Additional] Player can use sword in all directions using mouse

#### Enemy

* Enemy sprite will collide with map sprites
* Enemy sprite will (if correct type) bounce of walls
* The rebounding from walls will be predictable
* Enemy sprite can collide with player sprite
* Enemy sprite can collide with the player’s projectile sprites (bullets)
* Enemies will take damage when colliding with projectile sprites
* Enemy sprite (if correct type) will move in the direction of the player sprite
* [Additional] Enemy sprite (if correct type) will find the shortest path to player sprite

#### Menu

* A start button that will begin the game
* A quit button that will end the game
* A setting button, where controls can be seen, and sound settings can be adjusted
* A demo button, which will run the game for display purposes
* Pressing a specific button will bring up the pause menu
* The pause menu will pause the game and the game timer

### More Specific Test Data

#### Player

* Player sprite will be able to collide with the top door
* Player sprite will be able to collide with the bottom door
* Player sprite will be able to collide with the right door
* Player sprite will be able to collide with the left door
* If the player sprite collides with any door, they should be able to move the next room, depending on direction
* The camera will lock onto the room the player is currently in
* Upgrades will affect the player sprite’s attributes and methods
* Upgrades will affect the weapon sprite’s attributes and methods
* Bullets will travel towards the mouse cursor’s current position on screen
* The sword will be drawn in the direction of the mouse cursor
* The player will be able to shoot and move at the same time
* The player will be able to reload
* The player will be able to use the sword and move straight after (the player will stop for a brief moment)

#### Enemy

* Enemy spawn types will be random
* Enemy spawn location in a room will be random (within a specific area)
* Enemy spawn number will be random (between a specified range)
* Enemy sprites can collide with all doors
* Enemies will not be able to move through rooms
* Enemies will spawn only when the player enters that room
* When enemies spawn, all doors in that current room will be closed
* When all enemies in the current room are defeated, all available doors will be opened
* Enemies will not respawn when the player re-enters a room from before
* Once the boss of a level is defeated, then the player can progress to the next level

#### Map and Level

* Maps will be randomly generated
* A room will be assigned as a boss room
* A room will be assigned as a key room
* A room will be assigned as an item room
* A new map will be generated if the criteria is not met
* Doors will be created when rooms are adjacent to each other
* Doors will be closed when there are no adjacent rooms
* The player will be able to reach all rooms that are generated
* The boss room will be the last room generated
* A new map will be generated when the player progresses to the next level
* The new map will have more rooms than the previous one
* If necessary, there will be two boss rooms assigned

# Implementation

I plan to split the development of this game into 4 stages.

1st Stage – User input (movement) and Map Generation

2nd Stage – Adding player shooting, enemies, bosses and keys, basic menus, and additional map changes

3rd Stage – Scoring, UI, money system, level progression, animations, and maybe sound effects

4th Stage – Adding items, abilities, and extra enemies/bosses

There will be a bug testing and fixing during each stage.

## Stage 1

### Coding

**Movement and basic sprites**

In this stage, I will focus on developing the core mechanics of my game. Before starting the programming process, I made separate files for my code, so that it would look cleaner and therefore be easier to make fixes later on.



I also have a file named ‘testcode’, which is where most of my attempts at coding core functions will be, so that I can copy and paste them for testing if necessary.

In ‘main.py’ (where my game loop will be), I have imported pygame and all the other files, so that they are all linked to each other.



--> In this first two lines, I imported pygame and its ‘math’ and ‘random’ modules

--> In these lines, the \* means everything. So, I have imported all the code from each file to my main file

I then coded the base / structure, which I can then expand on:



I pre-made a sprite group, which I can add all my sprites to.



This is the structure of my game loop, which is a while loop that only runs when the variable ‘done’ is false. In the second screenshot, I make use of the sprite group which I made previously outisde of the game loop. With *all\_sprites\_list.update()* and *all\_sprites\_list.draw(screen)*, I can draw all sprites onto the screen at the same time and also have all of them run their ‘update’ method. For example, when I code the enemy classes, it will have an update method containing its movement. With the line *.update()*, I can make all the different enemy types move with just one line.

In the *settings.py* file, I stated variables for colours and the screen.



The colour variables were created so that when I need to instantiate a class, I can type ‘BLACK’ or ‘WHITE’ instead of the numbers. I also defined the screen settings, inlcuding its size and its caption.

In the *variables.py* file, I defined some more variables, which are more general.



The variable *done* is what decides whether the game loop runs or not. The variable *click* is what I plan to use when I implement mouse clicks with shooting. The variable *stamina* is what I will use when I implement sprinting for the player sprite.

I then went into my Class file, and coded the class for my player.



The player class will start off as a 40x40 rectangle with the parameter ‘colour’, which will be placed in the middle of the screen. There is also a ‘health’ attribute, but this won’t be used in Stage 1. In *main.py*, I added these lines outside of the game loop:



Here, I created an instance of the Player class called ‘player’, and I also coloured it white. I then added this instance into the *all\_sprites\_list*, so that it would be drawn during the game loop. The result for this was:



There were no problems with this, so I moved on to coding the movement for the player. I started by creating a new *move* method to the Player class.



This method has the parameters *x\_val* and *y\_val*, which I will give values to in the game loop. With these values, I will then add them to the sprites x and y position, so that I would move in that certain direction. In the game loop in *main.py*, I then coded:



The line *pygame.key.get\_pressed()* will get the state of all keyboard buttons, which means it can detect which ones are being pressed or not. I defined it as *keys*, and I used this with a few *if* statements. I wanted *a* to be left, *d* to be right, *w* to be up and *s* to be down. For example, for moving left, the code will detect if the key *a* is pressed. If it is pressed, then the Player class’s *move* function is called, with the values (-1, 0). This means *x\_val* will be -1, and ­*y­\_val* will be 0. When applied to the function, the sprite will move -1 in the x axis, meaning it would move left by one pixel continuously as the *a* is being held down. With this code, diagonal movement also works. This is requirement [1.2] completed.

The player’s movement also had no problems, so I decided to implement the sprinting mechanic, following the pseudocode in my design:



This is also in the game loop and is very similar to the normal movement code. The difference is that the *if* statement only triggers if a movement key and the sprint key (temporarily bounded to ‘j’) are pressed. If they are pressed, then the player will move, but at twice the speed. This is why the value is either 2 or -2. As both keys are being held down, the stamina counter decreases by -2. This is so that there is an idea of ‘stamina’. In addition, while the player is standing still or moving without sprinting, then the stamina count will increase by 1 until it reaches its 300 cap.



I also added the text, to show the current stamina count. I did this with:



However, there was a small bug with the sprinting, which I noticed immediately after adding the text. The problem was that while the stamina was regenerating, if the sprint key (‘j’) was held down, even when the player sprite was still, then the stamina would stop increasing. It would completely stop at the number it was currently at. This was what was causing the problem:



To fix this, I added an *or* to the *if* statement, with extra requirements for the statement to be triggered.



This states that when the sprint key is held down and the player sprite is unmoving, then the stamina count should regenerate. After testing this new line of code, it seems to work well. The problem doesn’t occur anymore. Though, the stamina count decreasing quite quickly. So, I might need to increase this value after testing the game when enemies are added.

**Map**

After solving the previous problem, I decided to create a basic map, so that I could test collisions. To do this, I created a 2D array in the *levels.py* file:



This is a 32x18 ‘grid’, where each cell in the array represents a wall sprite, which will have a size of 40x40. The 0s in the cells represent areas of blank space; no sprites will be drawn here. The 1s represent places on screen where wall sprites will be. For this test, I plan for them to be red. Lastly, the other numbers (including 2, 3, 4 and 5) represent areas on the map where I will draw door sprites. But for this test, doors won’t be included yet. After creating this map structure, I created the wall sprite class in *class.py*.



This class has the parameters *colour, width, height, posx* and *posy*. The *width* and *height* parameters will determine the size of the sprite block. The *colour* will determine its colour, and *posx* and *posy* will determine the position it will be drawn on the map. I also made a *delete* method, so that when new levels need to be created, the old one can be removed before doing so. This will be useful when I start implementing Stage 2 and 3.

In *main.py*, I created a sprite group called *wall­­\_group*, so that I would be able to make change to all the wall sprites if I needed to. This will be used for when I implement the camera movement.



I then coded a function, which can be called when I want to draw this structure:



The placing of wall sprites will start at (0, 0), which is at the top left of the screen. It will then iterate through each row and each column. It does this by firstly going to the first row, then checking each cell of that row, which is each column. Once the whole row has been checked, it will then move to the next row. This repeats until the whole array has been iterated through.

If the cell it is currently checking has the value ‘1’, the wall sprite is instantiated. The *x = x +* 40 line is there, because each wall sprite is 40x40. When the row is checked, it goes to the row below that. This is why *x* is reset to 0, at the start of the row. And *y* is added by 40, so that it moves down the screen, to the next row. Each wall sprite instantiated defined as ‘w’ for short, and the sprite is added to both the wall group and all sprites group, so that it will be drawn with all the other sprites.

The result of this first test when called:



There didn’t seem to be any problems with this first test, so I moved on to the collisions. The code for the collisions would be in the game loop:



This states that when there is any collision between the player sprites and any sprites within the *wall\_group*, the player method *move* will called with the values (0, 0), which means the player should be unmoving when this happens. However, this was not the case when I tested this algorithm:



As it can be seen in both game screenshots, the player sprite doesn’t actually collide with any of the wall sprites. Instead, the player moves through them, which is why the walls are overlapping the player sprite. So, my previous code, specifically the line *player.move(0, 0)*, is faulty. The solution I came up with was to change the position of player sprite when colliding, instead changing its movement and speed. If this also didn’t work, then I would try implementing velocity, and change the value of velocity when colliding occurs.

This was my first solution, which was changing the sprite’s position:



(The code below is outside and below the shown for loop)



This code states that whenever there is a collision, the player sprite will revert back to its original position. This original position is defined outside the *for* loop, so that it can be substituted when needed. The test for this was:



This was a successful test, and so I moved on to the next algorithm, which would be the doors and the camera movement that is needed.

To do this, I used the *blankMap* that I previously made, and assigned the different numbers to a different door. For example, if the number 3 appears in the 2D array, then the sprites for the right door are drawn.



I then added all these sprites to *all\_sprites\_list* and also to their individual sprite groups, depending on direction. So, outside the game loop I had to add:



This came out as:



This result was what I intended, and so I moved on to the camera movement as there didn’t seem to be any bugs. But before this, I wanted to make four extra rooms: one above, one to the right, one to the left and one below. This is so that when I test the camera movement, I would be able to see if it moved in the correct direction or not. To do this, I needed a new function. I first tested the rooms on the right and the bottom; I also temporarily changed the screen size so that I could see these rooms:



The code above will be under the function:



*x* represents the x-coordinate

*y* represents the y-coordinate

*z* represents the number of rooms

*randomNum* represents the room direction; 1 is right, 2 is left, 3 is up and 4 is down. It is defined as 1 for this example

The values for positioning the blocks for each room was hard-coded, as I found it easier to calculate the change of placement and direction, rather than using a 2D grid of some sort. For example, for the right room. After the blank middle room is drawn, I add 1280 to the x value, so that it is correctly positioned to draw a room to the right of it. This also applies for the bottom room, as the y value is added by 720. This correctly positions the point of drawing below the middle room. As noted, before, all room creations will start in the top left. For this test, I will draw the right room and bottom room separately:



This was a successful test, with no apparent bugs. I coded the algorithm for the top and left room too using the same structure, but with different values. I also altered some values and reverted the screen size changes, so that all rooms could be drawn at once:

To make it easier when I test the camera, I gave each direction a colour, so that I can check whether the camera has moved in the right direction or not.

**Camera Movement**

I planned to use the alternative algorithm that I had, as it would be much easier to code, and it should also work with not problems. The idea was to change the positions of the wall sprites and character sprites whenever the player ‘goes through’ any of the doors. By ‘going through’, the player will actually collide with the door sprites, and depending on which door, sprites will move. They will be moved in a way to make it seem like that the player has moved through the door, where in reality it has actually teleported across the screen.

This code was quite simple to do, just some short collision and for statements:



Each collision has a different set of directions. For example, when the player sprite collides with the door on the left, firstly the player sprite will move 610 pixels to the right of the screen. This is so that the player will end up on the right of the screen, just in front of the right door (exit of left door). Then, all sprites in *wall­\_group* will also shift to the left, by 640 pixels. For some reason, using 1280 did not work.

In general, if the player goes through the door in one direction, all sprites will be shifted in the opposite direction, including the player sprite.

The test for the camera movement for each direction is on the next page (yellow represents the left room, green represents the bottom room, blue represents the right room and white represents the top room):



This seemed to work fine at first, as seen in the screenshots, after further testing, a bug was found. On some occasions, when the player sprite collided with a door, the wall sprites would only move halfway across the screen:



I tried testing the collisions, by approaching the doors from different angles and directions to try and find out how this occurs. After some further testing, I realised it was because the one door contained multiple sprites. This meant that the player sprite would sometimes collide with two sprites, and sometimes only collide with one. This would also affect the value in which the wall sprites would move. The reason why moving the walls by 1280 pixels didn’t work was because for the majority of the time, the player collided with two door sprites, and so the value would be doubled, shifting the map double the distance.

To fix this, I removed the previous code, which made use of the 2D map array. I then replaced it with shorter code with hard-coded values (in the *spawnRoom* function), which was much easier to do:



I also changed the values in my collisions, to match the values of the width and height of the screen (1280x720). After some thorough testing, the previous bug was not occurring and there didn’t seem to be any new bugs.

**Random Map Generation**

My plan for this section was to first create a 2D array, which will act as a grid for the rooms. It will be used to show where each room, and with this, I will be able to check which rooms are next to each other. This will be used with a function to close off and keep open any doors where necessary. This grid will also be useful if I want to create a level which less linear. I can call the *mapCreate* function twice, without worrying about the map sprites over lapping as the position of each room will be permanently stored in the grid until I manually remove it. Removing / changing the values in this array will be quite simple; it should require two for loops only.

The 2D array looks like this:



Cells will have the value 0 in them when there is no room present there, and the value 1 when there is a room present. This is a 27x27 grid, and it needs to be this size because there is a chance, even though small, that all rooms will be drawn in the same direction.

In the middle of the grid, the cell is already assigned as 1, as the spawn room and the player’s starting point will always be in the middle.

I also created a few pre-made rooms, where I can put them in a group and randomly iterate through them to draw when needed.









The 1s in the 2D arrays represent where wall sprites will be drawn, and 0s represent free space in which the player sprite can move around in.

The *myRooms* array is what I will iterate through randomly later on.

To start coding this map generation, I had to alter the code in *mapCreate*. The code below was my first test/attempt at this, which I knew wouldn’t work as I ignored the map over lapping:



The *x* and *y* variables are both defined as 0, as this is where the drawing of sprites will start, which is at the top left of the screen. *z* is just a constant, which was used to determine the number of times the game will loop through the function. Inside the *while* loop, two random variables are defined. *randomNum* is the direction the room will be drawn (left, right, up, down) and *randomRoom* is the type of room. *randomNum* is the value that is used with the *myRooms* array, so that when a room position is decided, the type can then be randomly chosen. One type is chosen out of three different designs, which is what I have so far.

For example, if *randomNum* is equal to 1, then a blue right room will be drawn. The position from where it is drawn is calculated by adding 1280 to the x-axis from 0. Then the algorithm iterates through the 2D room array, placing sprites where the cell has the value 1. At the end of a *row*, depending on what z is, the x value is changed again. Then the y value also changes, so that the point is at the start of the next row. This repeats until the whole room is drawn. Here is a visualisation of this process:



This process is the same for the other directions (left, up, down), just with different values. The code is shown below:





The *if* statements at the very bottom of the algorithm are used to reset the y position of the drawing point. This can be written simply as *y = y – 720* by itself, but I wrote it as that temporarily, in case the values needed to be different depending on the direction.

After some testing, while ignoring the room sprites over lapping, it seemed to be working fine. To fix this over lapping, I knew I had to use the map grid. Each room had to be assigned to a cell, and I had to code an algorithm which would check if there were any other rooms adjacent to the room the player was currently in. This was definitely the most challenging part of the map generation.

This was my first attempt at this:

(The below is an extension of the existing if statements in the *mapCreate* function)



When it is the bottom room, it moves down a row, which is why *originalx* (explained on next page) is added by one, so that it moves down a row.

When it is the top room, it moves up a row by subtracting 1 from *originalx*.

When it is the left room, it moves back a column, which is why one is subtracted from *originaly* (explained on next page).

When it is the right room, one is added to *originaly* so that it moves to the next column in the current row.



I’ve state some new variables *originalx* and *originaly*; both are defined as 13. This is because the point (13,13) is middle cell of the map grid, which will be the starting point of every map generation. The blank spawn room will always be the first room created.

I also added the new if statements, which was my attempt at preventing the over lapping from happening. In summary, it states that:

If the direction randomly chosen already has a room present in it, then the random number will be ‘re-rolled’. It will keep creating a random number until the value isn’t equal to the previously selected direction.

After testing this, I saw that over lapping would still occur, but it did reduce the chances of it happening. The problem with the code above is that I only checked for the one direction selected. For example, if the right direction was randomly selected, the algorithm would only check if the right direction had a room or not. It wouldn’t check the other direction, as my intention with this was to prevent the rooms from going back on themselves. I didn’t consider that, for example, when the left direction is selected, there could be a room below it, not just to the right of it.

So, to fix this, I had to change how the algorithm would check for any adjacent rooms, and then choosing a new room direction using this information.

My next attempt at this was:









Like the previous attempt, this reduced the chances of over lapping from happening. However, it would still occur. But I knew I was getting close to fixing this problem, there were probably a few lines missing and some lines that needed fixing.

However, the code is getting very lengthy, and will definitely need to be shortened later or at the end of the programming stage.



The final and successful attempt was:

(Code has to be in the following order, for it to work)



If the direction selected is right, and the cell isn’t assigned to 1 (no room present), then the direction will stay to the right. But if there is a room present at that location, then it will go through the rest of the options, checking whether they’re empty or not.

After lots of testing, I can say that this algorithm definitely works. There is no overlapping, and there is always the right number of rooms.

If the direction selected is originally above, or if it was selected in the other if statements above, it will check if its available or not. If not, it will again check the other directions. If the new chosen direction is not available, it will select a new random direction.

If the directions selected is left, or if left was selected when the right was full, it will go through a similar process. It checks if left is free, if not, it checks the other directions and sees if they’re available or not. But there is an extra statement for checking the right side, as there is a chance that the right has already been checked previously.

If the direction selected is originally below, or if it was selected from the other if statements.

The same process for the directions is applied to this as well. It checks for other directions if the bottom area is not free, and then check these other rooms to see if they’re available or not.

However, there is a chance that there will be no other free spaces. If this happens, the while loop in which these statements are in, is stopped. The function can then be called again, but with a different *z* value, to fill in the missing room or two.

However, the problem with this code is that it is extremely long and quite repetitive. As it is working code, I will leave it like this for now, and make it cleaner and more efficient at a later stage. Most likely at the end of Stage 2, just before adding abilities to the game.

**Doors (basics)**

The last thing to do in Stage 1 was the code for closing off doors where there were no rooms present on the other side. To do this, I create a new function, which I would call whenever the player collided with the doors.









This function would check the adjacent rooms of the room the player is currently in. It will check using the *mapGrid* array and seeing whether the cells adjacent to the current cell has a 0 in it. If it is 0, it means there is no room there, and so will draw sprites over the doors where necessary, so that the player can no longer collide with them.

These sprites are drawn by iterating through a new 2D array I made called *door*, where the 1s represent the right doors, 2s represent the left doors, 3s represent the top door and 4s represent the bottom door:



This new function *mapDoors* will be called with every door collision, so that the sprites can be drawn for the next room. Any previous sprites drawn while in other rooms will also be deleted, so that new ones can be correctly placed. For example:



The delete function for the sprites is a method in its class:

The test for this new function was successful, shown here:



## Stage 1 – Review

### What was achieved:

After the structure of the code (screen settings, importing libraries and files, the game loop and sprite groups) was implemented and some variables were declared, a basic square sprite was implemented using a class to represent the user’s character; this will be replaced by an animated sprite in the later stages, but it will stay as a square for now to make testing easier. Basic input controls have also been implemented, such as movement with the *wasd* keys and shooting with the mouse cursor and mouse keys; both are methods inside of the *Player* class. Before adding collisions to the player, basic walls were created for the player to collide with. This was done in the *main* file and outside of the game loop, and this base room would become the spawn/starting area for the player.

Then the random map generation was implemented. Out of all sections in Stage 1, this took the longest to code, due to the amount of testing and fixing that needed to be done to make sure it worked without any errors. Fortunately, it does seem to work now, but it needs to be shortened. This is so that it will be easier to make additions or changes if needed. With the map generation, the basic structure for doors was also implemented. I will add the algorithms when I start to add enemies and items in the following Stages.

### Requirements met:

[1.0], [1.1]. [1.2], [1.3], and [3.1]

### Testing and fixing:

After the first attempt at implementing an algorithm, I would test it a few times. If bugs didn’t occur, I would then move on. But if they did occur, I made changes to the code to try stop the bug from happening and test it again. For the majority of the time, the problem was fixed in the first edit.

With the map generation algorithm, a lot of testing and fixing was required. This was because there were quite a few small problems that would arise with every change to the code. For example, the walls would overlap over each other, the camera would move by twice the distance and loops would infinitely run. This is why this section of Stage 1 required multiple days of debugging, to make sure it worked without faults. However, after eventually fixing all problems, the algorithm became very long and there were many repetitions. As the code currently works fine, I will shorten it at a later stage if needed.

### Changes to requirements and/or design:

Currently no additions or removals from requirements, and requirements that I planned to implement were implemented correctly in this stage.

The design of the camera was altered, to make it less complex to understand and to code. The way the map generation worked was also slightly different to what I had in my design initially. Instead of making the algorithm repeatedly loop until the criteria was met, now it will always make a level that will fulfil the criteria (for example, the number of rooms, location of boss room). This was done so that the game would not crash from the multiple loops.

# Stage 2

## Coding

**Main Menu**

For the menu, I needed to make a new function, that will be called before the game loop and display the menu buttons. There would be a start button and a quit button; the start button would call the game loop, and the quit button will exit the program. Firstly, the game loop had to be put into a function, so that it can be called when needed:



The only extra code that was added was the line that declared some variables were global. This is so that the game loop would recognise the variables, even while in a function. I also added new variables *player\_x* and *player\_y*, which I planned to use to keep track of the player sprite’s location on screen. These variables were added into the *variables.py* file.

After checking that the function can be called and all the variables worked without problems, I then worked on the menu function, which would be placed after the game function in *main.py*. This is because the menu’s start button would be calling the game function; it would not work if it were placed before. It will also be slightly different to how I planned it out in the Design section; there will now also be a loop running to check for button presses, instead of just using if statements. I started by declaring the values for the function:



The variable *menu* will be used to determine if the loop is running or not, and so is initially set to True when the function is called. The screen is set to black, and text and buttons are declared; they will be drawn later in the code. I then wrote the loop for the menu to check for button presses and draw the buttons, which had a similar structure to the main game loop:



The loop will continuously run until either the start or quit button is pressed, setting the value of Menu as False and therefore ending the loop. It will also end when the window quit button is pressed. The way it checks for button presses is first by checking whether the mouse has been clicked or not. If it has been clicked, it will take the position of the cursor when the mouse was clicked. It will then check if this value is within the boundaries of the buttons; this is done by the code *collidepoint*, which is built into pygame. If the start button is pressed, the game function is called. If the quit button is pressed, menu is set to *False* and the loop ends. The buttons and text for the buttons are also drawn within the loop. The screen is also updated, and the clock tick is set here. The main menu looks like:



I also added text highlighting, where the text of a button would be coloured when the cursor is hovering over the button. This was done by checking the position of the cursor, using the built-in *pygame.MOUSEMOTION*, which detects for cursor movement, and *pygame.collidepoint()*, which detects if a value is within the area of a sprite. If the position of the cursor is within the button sprite, the text colour will be to red. It will change back to black when it moves off the button. The code for this is:



**Pause Screen**

As planned, I then added a basic pause screen. This turned out to be much easier than I thought it would be during the Design phase, where I thought that I had to make a separate function for it. All I had to do was add an *if* function and bind certain keys that would activate the statement’s requirements. These were the changes:



Certain keys being pressed will determine the value of the variable *running* (which is set to True at the start), which is then used in this *if* statement:



If *running* is set as True, the original code in the game loop will be executed. If it is set to False, then the main game code will be temporarily paused, and a new screen will be drawn. The game will be resumed when the key to set *running* as True is pressed. The pause screen at the moment is a black screen with ‘paused’ in white text.

**Player Shooting**

For the player to shoot, I first needed to create a bullet class, using the plan from my design:



The Bullet is given an initial x and y position, which when called, would be the centre co-ordinates of the player sprite. It is also given an x and y position of its target, which would be the position of the cursor on screen, as I intend to use the mouse to aim. It is also given a size and speed.

Using the target and starting position co-ordinates, it finds the angle from the player to the cursor in radians with the *math.atan2* function. Using this angle, it finds the change in x and change in y, using sine and cosine. These are then multiplied by the speed, which will determine how fast the bullet sprite moves. It then uses these values in the *move* method to direct the bullets in the direction of the cursor. The attributes of *rect.x* and *rect.y* are subtracted by 5 so that the bullets appear from the centre of the player sprite. Without this slight adjustment, the positioning would seem off.

For the actual shooting, this would need to be done inside of the game loop. It would need to constantly check if the mouse if being clicked or not:



If a click is detected, specifically mouse1 (indicated by *event.button ==1*), it will draw a bullet sprite, giving it the appropriate values. The bullet sprite is then added to the two sprite groups, and I also added the collisions for it.

This looks like:



**Enemy Type 1**

This enemy will be able to shoot the player, while moving around randomly, whilst also being able to bounce off walls in random directions. The enemy class (*arraySpeed* is defined in the Variables files, with the value [-1, 1]):



The position of the Enemy sprite at the moment will start at (50, 50). This is so that I can test that the movement and collisions are working. But once the movement and shooting is finalised, I will alter it so that it can be drawn in either of the four corners of a room, and the number of enemies will also be randomised.

The *update* method is the movement, where *speed\_x* and *speed­­\_y* determines the direction it will first travel in. Every time the enemy sprite collides with a wall, theses variables will be set a new value, so that it will move in a different direction, imitating a wall bounce.

The enemy movement and collisions seemed to work fine after some test runs (the smaller square being the Enemy sprite):



I also changed its colour to purple, so that it is easier to distinguish the different sprites for now. I then worked on the enemy’s shooting mechanics. The class for the enemy bullet looks like:



The bullet is given a starting position *posx* and *posy*, which are the centre coordinates of an enemy sprite. It is also given a target, which would be the position of the player sprite. This works the exact same way as the Player bullet class works. Using the parameters *targety*, *targetx*, *posy* and *posx*, it finds the angle between the player sprite and mouse cursor in radians. With this value, it then calculates the gradient of the x and y line, which are defined as *dx* and *dy*. In the update method, it uses the values *x, y, dx* and *dy* to direct the movement and direction of the bullet.

To test whether the enemy bullets worked, I needed to create a separate function, which I named *enemyShoot*:



I wanted to give the enemy a rate of fire, which is why I used the *pygame.time.set\_timer* function. This sets a timer for a number of milliseconds; in this case it is 150 milliseconds. Every 150ms, a bullet will be drawn for every enemy sprite present in the sprite group. The enemy’s current position will be the starting position of the bullet, and the player sprite’s current position will be the bullet’s destination. This function is called in the game function, inside of the while loop. the After some testing, with the enemy standing still to make it easier to see, this seemed to work fine:



However, there was a slight problem that I noticed. The bullets weren’t actually firing every 150ms, instead they were firing in bursts, which can be seen in the screenshot above. It was also inconsistent, sometimes firing two times other times 4. Although this is not what I intended, I decided to keep it for now, as it is much less predictable which could be better for the game.

The next thing I wanted to work on was the advanced doors.

**Doors (advanced)**

These doors will be closed when the player enters a new room and will only open when all enemies inside that room are defeated. If the player enters a room that has already been previously visited, then the doors won’t close, and enemies won’t spawn. To do this, I created a new function:



When called, this function will draw blocks that will cover up every entrance, whether there is a room on the other side of it or not. The variable *enemyCount* will be used later, so that doors can be reopened when this variable is equal to 0. I then needed a way to call this function, whilst drawing the enemy sprites with it. So, I wrote a function that would just create one enemy sprite for now:



Later, I can add a for loop to this, so that a random number of enemies can be spawned.

This would be called in the same place the function *mapDoors()* is called to tests for now, which is whenever the player sprite enters one of the doors.



I also added the line *mapGrid[mapx][mapy] = 3*, where the value 3 represents a room that has been visited. I will use this later to determine whether to call *doorClose()* or not. The new doors appear as:



These doors seemed to work fine, as they would activate whenever the player enters the next room, and an enemy would also be created at the same time. I also added the collisions for the sprites, using the new *doorclose\_group*.

Player collision (in game loop): 

Enemy collision (in its update method):



I also grouped all the collisions with bullets/projectiles into one function, to make it easier to add other collisions: 

I knew all these collisions worked, as they were essentially the exact same as the previous ones. So, I then worked on writing code for opening the doors when the number of enemies in a room becomes zero.

First, I used a for loop in the game loop, to change the number of enemies. The previously mentioned variable *enemyCount* will be subtracted by one when ever the player’s bullet collides with an enemy sprite. The enemy sprite will also disappear after any collision with the player’s bullet:



Just after the code above, I then added code that would open the doors when all enemies are killed in a room, which is when *enemyCount* has a value of 0. This is done by:



This uses the *delete* method in the Wall class.

I did some tests for this, and it worked fine:



Next, I worked on checking if a room had been previously visited or not. To do this, I used the previously mentioned *mapGrid[mapx][mapy] = 3* to identify the rooms. I also added the value 2 to identify rooms which the boss will spawn in; the code for this was added in the *mapCreate* function:



I then edited the existing code in the *game* function to check for these different values (*bossSpawn* is currently an empty function):



When the player enters a new room, it will check whether the value in the array. If the value is anything besides 2 or 3, it will create the doors and also spawn in the enemies. If the value is 2, then no functions will be called, which is when a room has already been visited before. If the value is 2, then the boss creation function will be called.

I initially thought that this would be hard to do, but after testing the code above, I found no errors, it all worked fine.

**Boss 1 (boss 2 from the design)**

The idea of this boss is that it would charge towards the player, and then when it collides with a wall, it will bounce and charge towards the player again.

I started by creating the class, which would need multiple methods:



This class is quite similar to both of the bullet classes from before, as the boss needs to be given target before it starts to move. Its initial position will always be the middle of the screen, and at the moment it has health value of 25, where it will take 25 bullet collisions for it to die. I can do this by adding this to the main game loop:



This states that for every collision that a player bullet has with the boss sprite, the boss’s health will decrease by one. With this, I also added a very basic health counter, so I can check that collisions are working properly:



This is the update method of the boss class, which is where the game will check when the boss sprite needs to be deleted and when to draw the health counter.

To draw the boss, I created the function *bossSpawn*:



This draws the boss sprite and also activates the doors. The new variable *bossCount* will be used later, when I create a way to progress to the next level.

When run, this looks like:



After 25 bullet collisions, the boss sprite would disappear. The health counter would also disappear. Next, I needed to add the movement methods for the boss. I created a method called *attack*:



This method is the exact same as the update methods for the bullets, as they all move in the same way. After testing, I saw that it would always move to the player’s initial position when entering the room. I needed a way to stop the boss when it hits a wall, and then also redirect it to the player’s new position.

I created a new class method called *stop*:



I planned to call this method whenever the boss sprite collides with a wall. The method would calculate a new value for the angle, using the player’s new position. It will then stop the boss sprite’s movement, by defining *self.dx* and *self.dy* as 0. Then it will start moving again, using the new angle to move towards the player sprite.

I now required a way to call these different methods in the right order. I created a function called *bossAttack*, that would be called from inside of the game loop, so that it is always checking if something needs to be activated. The function itself is outside of the loop:



Two different collisions are defined, one with the boss and the walls and another with the boss and the doors. For every boss sprite on screen, the *attack* method for it is called. This is when the boss will ‘charge’ towards the player in one direction. When a collision with the walls or doors is detected, the *stop* method is called. This will initially stop the sprite, and then using the player’s new position, it will start moving again. This process will continue to loop, until the boss dies, which is when the sprite is deleted.

Testing showed that there were no errors with this:



## Stage 2 – Review

### What was achieved:

I started this stage by creating a main menu, with a start and quit button. To do this, I needed to transfer the game loop into a separate function, so that it could be called from the menu function when needed. I am also planning to add a ‘controls’ button when I start making the UI, so that the user is able to check how the game is played beforehand. I also made a very simple pause screen, which at the moment just displays the text ‘paused’ in the top left over a black screen.

I created the shooting ability of the player, where the mouse is used to aim the bullets.

I created the first enemy type. This enemy shoots towards the player, where its rhythm is determined by a timer. It also moves randomly and is able to bounce off walls, where its direction after bouncing is also random. These enemy types take one bullet to kill, and in Stage 4, I will add different enemies and make the spawn number and location random. I also created the first boss type, which works by ‘charging’ towards the player sprite. When it collides with a wall, it will redirect itself towards the player again.

I improved the door system, by creating doors that would close in unvisited rooms and spawn in enemies. These same doors will then open when all enemies are killed. If a player enters a room that has been previously visited, then these doors won’t be activated, which I did so that the player would be able to traverse through the level quicker.

### Requirements met:

[2.3] [1.9] [1.4]

Partly – [1.5] [2.5]

### Testing and fixing:

There weren’t many problems in this stage, only very small syntax errors, but nothing I needed to spend lots of time to fix. Everything went according to plan in this stage.

I tested the game again after finishing this stage, and there seemed to no bugs present.

### Changes to requirements and/or design:

I decided on removing the key room, as I felt it wouldn’t add much to the game. It would instead just make the length of the game unnecessarily longer. Removing the need to get a key to access the boss room also gives the player more freedom. They have the choice to explore and collect more money, or to just move on to the next level.

I also decided to remove the player sword variation. At this stage of development, I found that a melee weapon would not fit well with this game, and it wouldn’t be enjoyable chasing down enemies to hit them.

But other than that, I did what I planned to do for this stage.

# Stage 3

## Coding

**Level Progression**

To progress to the next level, you will need to first defeat the boss in your current level. After the boss is defeated, I plan to create a teleporter in the boss room, which the player can collide with to move on to the next level. To do this, I needed to make a teleporter class, which would be a very simple square sprite. I can replace this with an image later.



To draw this sprite, I will only draw it when the boss is defeated, which is when the number of bosses present is 0. In the game loop, I added:



I then added a very basic collision checker for the teleporter, which is also in the game loop:



I tested this out, and it worked fine:



I also added a way to move the teleporter, if the player decides to explore any previous unvisited rooms. This works the exact same way as how the walls are moved:



Next, I worked on the teleporter’s function, which is to create a new level, reset the player’s position and reset some variables. To do this, I needed a new function outside of the game loop, that can be called when necessary. The function looks like:



In this function, the number of rooms a level can have is increased, and the player’s position is reset to the middle of the screen. The health value of the player is also reset to its initial value. After the variables are set, I called a new function called *deleteWall()*, which will remove all of the extra sprites from the screen. This is done so that I can draw in a new map, with new doors.

The *deleteWall* function:



In this deletion function, all wall and door sprites are completed removed, so that the only sprite left would be the player’s sprite. Then, the grid for the level is also completely reset. This is so that the door system works, and rooms are correctly detected. If it wasn’t reset, then there would be the values 1s, 2s and a 3 in the array where rooms are not actually present.

However after some testing with this, it didn’t seem that the grid was reset, as there were bosses spawning in the incorrect rooms, and missing rooms.

e.g. 

After more tests, I found out that it was because there were errors with the new mapGrid, which is why this happened:



To fix this, I checked through the functions I created. I realised that I was missing variables in *teleport()*, which were the values of where the first room would be. So I re-defined the variables *mapx* and *mapy*:



This appeared to work at first, but again after more testing I found an error. For some reason, the rooms would be correctly placed for the up, right, and bottom doors. However, when entering a left door, the room would be missing. This also happens when backtracking rooms, for example, going to the right room and then back to the left room. This is shown below:



After some more testing, I found that only in the newly created map, the left door would detect two collisions, which meant that the values used to move the walls (but not player) were doubled. However, this double detection does not occur with the other doors. It sometimes also detects the left door four times.

I found that there would always be an extra left door in newly generated levels. As it can be seen in the screenshot, the number of left doors is 2, whilst the other doors have a count of 1. After further testing, I saw that the old left door wasn’t actually being deleted in the *deleteWall* function.

Turns out the reason why this wasn’t working was due to missing brackets. It worked as intended after adding them in.

**UI**

To start the UI, I first created some extra space on the side of the screen to display the UI. This is different to what I initially wanted in the design, as I thought that it would be too overwhelming with the UI in front of the actual game.

I created this extra space by extending the width of the screen and drawing a black rectangle in that same area, so that it can be written or drawn over.



After making space, I created a template for all the different counters that needed to be seen: 



I have included a health counter, a stamina counter, an ammo counter (I plan to add the mechanism of reloading in the last stage), a coin counter and a score counter.

**Player Health**

The player starts off with 6 health, which is set in the *Player* class. I also added an extra attribute called *self.healthMax*, which would also be set to 6. I created this attribute as I think that it would be useful for implementing health-up items later on.

In the game loop, I entered this code to decrease the player’s health by 1 every time it is hit by an enemy bullet:



I then did the same thing, but also with the enemy and boss sprites.



I also made it so that enemies would die when colliding with the player.

This seemed to work after some testing, but I now needed a way for the player to die when its health is at 0. I also plan to create an algorithm that will give the player some immunity after getting it, just so that there won’t be repeated collisions when in contact with enemy sprites.

First, I created a method for the player called *delete*, which I can call whenever I want to kill the player.



With this method, I can call it in the main loop when the value of the player’s health is 0.



Setting running to False will pause the game, and I will also display text when this happens.



Next, I worked on the temporary immunity after getting hit. I first created new variables called *collision\_immune* and *collision\_det*. These both would be used to check if the player has been hit or not. Outside the game loop, I have set the value of *collision\_immune* to False, and within the game loop, there is this if statement:



This states that if the player’s immunity is off, then they can get hit. However once they get hit, first, immunity is ‘turned on’ and the other variable shows that the player has made a collision. I repeated this with enemy and boss collisions too. I then made two other if statements, which would make use of these variables, with a new variable named *collision\_time* which is initially set to 999999… : 

Here, when *collision\_det* is set to True, which is when the player sprite colides with an enemy, the current time is taken and the player turns red to indicate immunity. This time, defined as *collision\_time*, is used to determine when to turn the player’s immunity off by comparing it to the times after it. If the difference in time is 1500ms, then the player’s immunity will be removed, meaning the player’s immunity will end 1.5 seconds after getting hit, and now can be hit again.

Below are screenshots of how the player’s health works:





**Player Ammo**

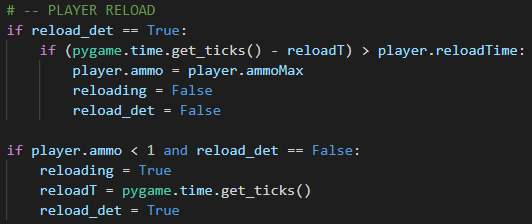
In the player class, I added two attributes *self.ammo* and *self.ammoMax*. The first will be used to keep track of the player’s current ammo count, and the other will be used for reloading and also abilities in the last stage. These both will be initially to 5.



To create the reload function, I took a similar approach to how I did the player immunity. I created three new variables:



I first created an algorithm that would automatically reload whenever the ammo count is 0.



The second if statement checks for when the ammo count is 0 and that it also isn’t currently reloading. If this is true, then *reloading* is set to True, *reload\_det* is set to true and *reloadT* is defined. The first statement checks for this second variable, and if it is true, the same timer method from the player immunity is used here.

The variable *reloading* is used to determine if the player can shoot or not:



I tested this automatic reload and it seemed to work fine with no errors.

Lastly for ammo, I needed a manual reload. This algorithm would activate the reload whenever a certain key is pressed. This would be very similar to the automatic reload check, with a few differences:



It first checks for the condition that the ammo count is within the range 0 < ammo < ammoMax. Then if the ‘r’ key is pressed whenever the ammo count is in this range, then it activates the reload by defining those three reload variables.

I tested this manual reload and there seemed to be no problems with it.

**Coins**

This game’s money system is very basic. Money is earned after killing an enemy or a boss.



These two algorithms are located in the main game loop. All they check is if an enemy is killed, then the player is given a certain number of coins. For a boss its 5 coins, and for a standard enemy its 1 coin.

This system had no problems after testing, and these coins will be used in the last stage of implementation to open chests.

**Scoring**

The scoring system, like the money system, is very basic. Score is determined by progressing through different rooms and levels. But there are also additional features, such as challenges.



In the first screenshot, 2 points are given to the player when the enemyCount is 0, which is when a new room is completed, and enemyCount is then set to 1 just so that it doesn’t continuously add points. There is also a bonus, which is when the player completes a room with full health, the player receives another 3 points. The new variable *chestA* will be used in the last stage, to draw a chest when a room is completed. 3 points are given to the player when a boss is defeated, and a bonus of 4 points is also given if they complete it with full health. *abilitiesBoss* is a function I will use in the last stage to give an item to the player.

The second screenshot shows how the player is given 2 points when completing a whole level.

**Mini Map**

I started this by creating classes, one for the mini map blocks and another for the player pointer.



To draw these sprites, I thought the best way to do this was to create a function. So outside of the game loop I created this function and started by just drawing the first room in the centre. There are two new variables *minimapx* and *minimapy*, which I will use to change the drawing location of the miniMap blocks. I also created an instance of the MiniPlayer class:



This function is called with *miniMap(0)* at the start of the game loop, and it looks like this:



Now I needed to add to this function so that it can keep track of where the player has been. First, I needed to assign different values to specific directions. So 1 would be right, 2 would be left, 3 would be up and 4 would be down.

I then wrote these other if statements in the *miniMap* function:



Here, I use the two mini-map variables mentioned previously to change the position of the new map block to be drawn. I also change the position of the currently drawn *MiniPlayer*, which is used to indicate which room the player is in. The function is called when the player sprite collides with one of the directional doors made in Stage 1, for example:



In game, this appears as:



I did some testing and it seemed to work fine, but when progressing to a new level, the old mini-map would overlap with the new one.

However this was a simple fix. All I had to do was delete the old sprites when entering a new level, which I included in the *deleteWall*:



**Animations and Sound Effects**

I’ve decided to not include any animations or sound effects in the game. Although it isn’t as visually appealing, having simple shapes with distinct colours would make identifying each object much easier. Instead of animations, I added still images to most of the unmoving objects in the game. I did this by first creating a new folder called *assets*, to contain all the sprite images I needed, which were .png files.

I used some of the tilesets that I showed in the Design section to create images for walls, chests, and the teleporter, by cutting of these tilesets and resizing if necessary.

Then all I needed to do was alter the classes for these sprites. This is what the wall class looks like now after editing it:



The difference is that now there is a new line, which loads the .png from the folder names *assets*. This new attribute is used with ‘image.blit’ to draw this image on screen; this replaces the previously used ‘image.fill’. The (0,0) represents from what area of the image would be shown.

I repeated this edit with the wall and teleporter class. This is what the game now looks like with some images:



## Stage 3 – Review

### What was achieved:

I completed the level progression system, by using a teleporter to ‘transfer’ the player to the next level. This works by resetting the whole game loop. This includes replacing the player sprite in the correct starting position, deleting the old walls, drawing a new level with a higher number of rooms, resetting the mini-map, and so on.

This level progression also includes the money system. Money is given to the player after every enemy is defeated, and depending on what type of enemy, a different number of coins is earned. This money system will be used in the last stage when I plan to add chests.

Lastly to level progression, I added a very basic scoring system. Scores are given depending on the number of rooms, but I also added challenges. If the player completes a room with maximum health, meaning no damage taken, then a bonus score is given.

I created the player’s health algorithm. The player has a max health and a current health counter. Also, to prevent one collision from taking away all the player’s health in one go, I created a way to give the player temporary immunity after getting hit. This is so that multiple collisions aren’t taken when the player sprite collides with an enemy sprite.

I created the player’s reload algorithm, using timers and variables to control when to activate the reload and give the player the ability to reload.

### Requirements met:

[2.9] [2.4] [1.7] [2.0] [3.0]

### Testing and fixing:

The main problem from this stage was during the algorithm for the teleporter. When the new map was drawn, there were rooms that had missing wall pieces. In addition, the 2D array used to create the map did not match what was actually drawn. After extensive testing with trial and error, I found out that all I needed to do was to add a pair of brackets, then it started working as intended.

### Changes to requirements and/or design:

I decided to not include an in-game timer, as I felt it wouldn’t add to the gameplay.

I decided to not include animations to moving sprites, to keep it simple and easier for the eyes.

I also did not include sound effects.

# Stage 4

## Coding

**Chests**

To create a chest, I first made a new class for it. This would be a basic sprite with an image:



There is also one method called *delete*, which will be used to reset the map when progressing to a new level.

I want to draw chest every time a room is completed, but I don’t this to happen al the time. Instead it should be a 50-50 chance of getting a chest. So outside of the game loop, I created a function that can be called when needed:



The variable chestA will be used to determine if a chest should be drawn or not. This algorithm appears inside of the game loop:



This says that when a room is completed, which is when all enemies are defeated, the variable chestA is defined as True. I then created another short if statement to call the function:



I could’ve put this in the previous *if enemyCount == 0* statement, but there is a reason to why they are separate. The reason is so that the chest will actually appear on screen and be drawn over everything else. Before I made this change, *createChest* was called as soon as the *enemyCount* was 0. Every time a chest should’ve appeared on screen, it was actually being drawn behind everything else on screen.

This is what the chest looks like:



Now I needed to include the money system with these chests. I plan to let the player open chests with 5 coins. When a chest is opened, there will be different items that the player could get.

**Chest Items**

I created a blank function outside the game loop called *abilities*. This would be called when the player collides with the chest image, but it will only activate if the player has enough coins, which is 5 to open it. This is an if statement that I wrote in the game loop, to activate the chest if appropriate:



This says the detection between the player and a chest is only activated when the player has enough coins to open the chest, and the chest will disappear when this happens.

I needed to write the code for the *abilities* function. This is what I started with:



This makes *coins* as global, so that changes made to it stay changed. A random number is defined, which I will use to determine what item to give to the player. Then the number of coins is decreased. With this, I did some testing to make sure the collisions were being detected when needed, just by printing a string. The collisions worked fine, as the string was only printed when the player had enough coins.

Now I needed to add to this function, by creating the different items the player could get. These are the items that I want to add to the chest:

* Health up
* Increase max health
* Increase max ammo
* Increase max stamina

This is what I came up with for both health items:



The first if statement is very simple, all it says it that the player’s maximum health will increase by one, and the player’s current health will match it. The second if statement says that if the player’s current health is less than their maximum health, then add one to the current health. This health up item also has a higher chance of being picked out than the other. This is because it is quite easy to get hit in the game, so a health up item should be the most common one.

I tested both of these items, and both worked fine. However, with the second item, if the player had maximum health, then the player would get no item. To fix this, I needed a way to call the function again when this happens. This is what I did:



This says that if the player’s health is already maximum, then the function will be called again to ‘re-roll’ the random number to get a different item.

I then repeated this with the other items I mentioned previously:



I tested both of these, and all the items worked completely fine. But there was a bug that seemed to occur once in a while. Whenever the player collided, the collisions would be detected twice, and so the 10 coins would be taken instead of 5. To fix this, I defined a local variable *tempCoins* to equal to the global variable *coins* at the start of the function. I then added this if statement to the end of the function:



This states that if 10 coins had been taken away originally, then add the extra 5 coins back to the player. I tested this and it seemed to fix the problem, as the number of collisions detected would never exceed 2.

**Abilities**

Initially, I wanted the player to be able to acquire different abilities as well as items through chests. Instead, I thought gaining abilities through defeating a boss would be better. This system would be very similar to the chest system, with this being the base of the function:



These are the abilities that I want to implement into the game:

* Teleport
* Double Damage
* Reduce reload time
* Bullet speed increase
* Bullet size increase
* Passive (double damage when health is 1)

This is what I started off with for the teleport ability:



In player class.



Now I needed an algorithm for this teleport ability, which would need to be placed in the game loop. This would be just below the player shooting system. This is my first attempt at the algorithm:



I tested this, and I noticed a problem with this code, and that something was missing. If I clicked on wall or if I clicked on the black area (where the UI is), the player would either end up inside of the wall sprites or outside of the room. I did not take this into account in my code.



To fix this, I will need to find a way to detect for a collision and to also check the range of values in which the player can teleport into.

I created a new variable called *teleportCollide*, and it is initially set to False. My idea is to check the values given by the mouse and to compare it the coordinates of the wall sprites. This is similar to how I created my buttons when I made the menu. This is what I added to the algorithm:



Here, I used to *collidepoint* function to compare the coordinates of the mouse to the coordinates of the wall sprites. If these values match, then *teleportCollide* is set to True. If *teleportCollide* is False, then this means the player hasn’t clicked a wall and so the separate x and y values of the mouse are taken. Then the x value is checked to see if it less than 1280. This is so that the player can’t teleport outside of the level (into the area behind the UI screen).

I tested this a lot, and none of the same problems occurred anymore.

I also added this to the *teleport* function, so that the teleport ability count is reset when entering a new level:



Next, I worked on the double damage ability, which is set like this in the *abilitiesBoss* function:



*doubleDam* is another attribute I added to the player class. I then also added *damage* as another attribute, with it initially set to 1. With this, I also slightly changed how the boss would take damage from the player.

Just above this boss collision statement, is where the value of the player’s damage is changed:



I tested this and it worked without any problems.

I then worked on the passive ability, which is again in the main abilities function: