Logbook

Development Diary

Year 11 Software Engineering AT2

Andrew Tran
KING'S QUEST

Contents

Features:

Documentation:

4/07/24

Progress

Achievements	Stumbling Blocks
• Created health bar class and implemented drawRect() method into map.py code • Health bar is visible over player's character and enemies • Health bar width updates in accordance with the player's and enemy's current health • **Merge** health bar branch with main	• Recognising when to use 'self' and when to use class name in code

AT2 > healthBar.py > ...

```python
import pygame

class HealthBar():
    def __init__(self, x, y, w, h, max_hp):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.max_hp = max_hp


    def drawRect(window, x, y, w):
        rect = pygame.rect.Rect(x, y, 100, 10)
        pygame.draw.rect(window, (255, 0, 0), rect)
        rect2 = pygame.rect.Rect(x, y, w, 10)
        pygame.draw.rect(window, (0, 255, 0), rect2)

```

Figure 1.0

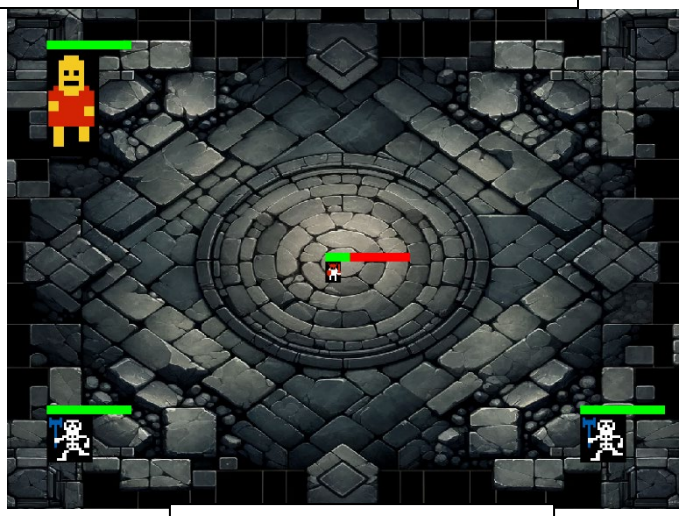


Figure 1.1

Evaluation

Initially, determining how to implement the health bar was a tricky task as I had no previous experience with using pygame tools and had no knowledge of where the health bar would fit in from the base code. After attending the software engineering day at the King's school on 3/07/24, I had returned home with more clarity on how the base code runs and where all the methods are located. The tutors at the day had also help me organise my code properly, moving the movement controls in the correct file (character.py) and instantiating a character object into the map.py file.

I started on the health bar by creating a separate file titled 'healthBar.py' to follow object-oriented programming principles. In the HealthBar class, I created a draw method to create two rectangles that layer on one another. This drawRect() method was called in the map.py file to put the health bars onto the screen.

I added three arguments to the draw method, the position on the x axis, position on the y-axis, and the width. The two positional parameters were given by calling the method with the player's position. The width argument was only used on the second rectangle. This way when the width of the green rectangle was decreased the red rectangle remained with the same dimensions. I

made the rectangles start at a width of 100 pixels which was convenient as the player's health also started at 100. This way when I entered 'current_health' as a parameter for the width it would start at 100 pixels and decrease with the health. In theory, this design should remain to work however, if the dimensions of the health bar change I will have to explore alternative options of decreasing the health (maybe one not decreased with width).

5/07/24

Progress

Achievements	Stumbling Blocks
• Created new combat system with buttons • Combat waits for user input before beginning battle • Player receives small amounts of health as a reward for defeating an enemy	• Slowing down the turns • Tracking the number of times the button is clicked instead of how long the button is held

battle.py > Battle

```
import pygame

class Battle():

    def attacks(self, window):
        icon = pygame.rect.Rect(250, 250, 200, 100)
        pygame.draw.rect(window, (0, 0, 255), icon)
        player_damage = 0
        position = pygame.mouse.get_pos()
        if icon.collidepoint(position):
            if pygame.mouse.get_pressed()[0] == 1:
                player_damage = 10
        return player_damage
```

Figure 1.2

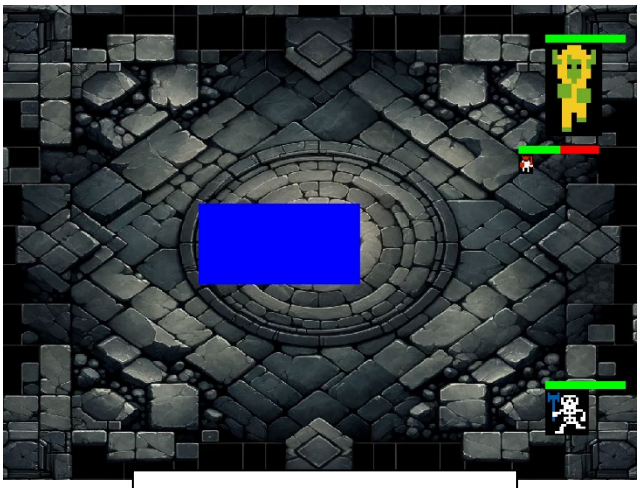


Figure 1.3

Evaluation

My original approach to the combat system was to handle it through the command line. This would involve the player typing in the attack they wanted to use and using this input to update the game and enemies. As I tried running a command line system, I quickly learned that it would not be a viable approach as my separate pygame window, that displays the game, could not update regularly without crashing. Switching back-and-forth between the command line and the game display was not efficient and ruined the flow of the game.

Kowing this, I scrapped that idea and instead tried to implement a button system. This would work by displaying a series of buttons to the user, each holding a different attack, and depending on which button was selected that attack would go through. Although challenging at first because of my lack of knowledge around pygame modules, google searching the mouse position and click of the user helped me learn how to implement this in my code. The main problem right now is that all the turns are used in one click as the program believes the user is holding the button instead of clicking it. This means that when the user uses an attack, the enemies disappear and are killed instantly.

6/07/24

Progress

Achievements	Stumbling Blocks
• Combat responds to user input after clicking a button • Added multiple buttons with different colours, text, and damage values • Shaders on button when user hovers cursor over it	• Finding the right method/library package to fit the requirements • Using logic to create a more accurate turn-based system

```
icon2 = pygame.rect.Rect(500, 250, 200, 100)
pygame.draw.rect(window, (255, 0, 0), icon2)
if icon2.collidepoint(position):
    pygame.draw.rect(window, (139, 0, 0), icon2)
    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                player_damage = 50
            else:
                pass
#Text for the rectangles
TEXTCOLOUR = (255, 255, 255)
fontObj = pygame.font.SysFont("microsoftphagspa", 32)
textSufaceObj = fontObj.render('Blue Attack', True, TEXTCOLOUR, None)
window.blit(textSufaceObj, (170, 275))

TEXTCOLOUR = (255, 255, 255)
fontObj = pygame.font.SysFont("microsoftphagspa", 32)
textSufaceObj = fontObj.render('Red Attack', True, TEXTCOLOUR, None)
window.blit(textSufaceObj, (520, 275))

#Final output result with player's delt damage
return player_damage
```

Figure 1.4

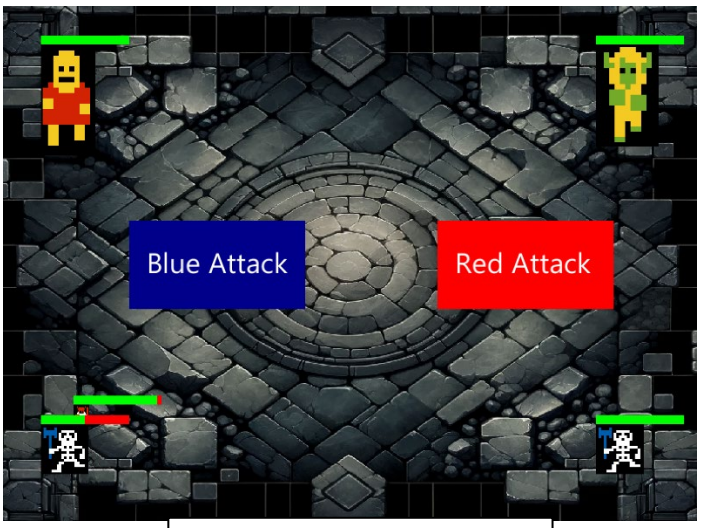


Figure 1.5

Evaluation

To get the attack buttons to register that the player is clicking the button once instead of holding it down, I had to change my pygame.mouse method from 'get_pressed()[0]' to 'MOUSEBUTTONDOWN'. This used the event handling queue to find one mouse click, act on it, and then move it back down to the bottom of the queue. This works better for turn-based combat as it slows down the pace of battles and lets the user have a decision making aspect to their combat. I then duplicated my code from the first button to create the second button, only changing the colour, text, and position of it.

I plan to coordinate these buttons with different attacks from each of the character classes. This will then factor in stamina costs and unlocked player skills, making a more engaging and immersive combat experience. The turn-based combat could improve with animations or some type of visual indicator to show that a turn has passed. This could look like a punch mark against the enemy getting hit or a highlight indicating which turn it is. However, my limited skills with animation and image handling in python make this a challenge for me to implement.

Also, 'one turn' in my game currently is the equivalent of one attack from the player and one attack from the enemy. This means once the player picks their attack, the enemy's attack has already happened and it's back to the player's decision. This is not ideal but projects a system closer to 'turn-based combat' than what the game previously had.

7/07/24

Progress

Achievements	Stumbling Blocks
• Made attack buttons display the name of a class's unique attacks • Added more buttons to fit the amount of attacks the player can use	• Retrieving the attack name from a dictionary inside warrior.py and so on

```
y > Battle > attacks
Battle():
f attacks(self, window, character_type):
            if event.button == 1:
                player_damage = 50
            else:
                pass

#Retrieve attack names from attack dictionary
attack_name = []
new_attacks = dict(character_type.attacks.items())
for name in new_attacks:
    attack_name.append(name)

#Text for the rectangles
TEXTCOLOUR = (255, 255, 255)
TEXTCOLOUR2 = (0, 0, 0)
fontObj = pygame.font.SysFont("microsoftphagspa", 25)
textSufaceObj = fontObj.render(attack_name[0], True, TEXTCOLOUR,
window.blit(textSufaceObj, (215, 175))
```

Figure 1.6

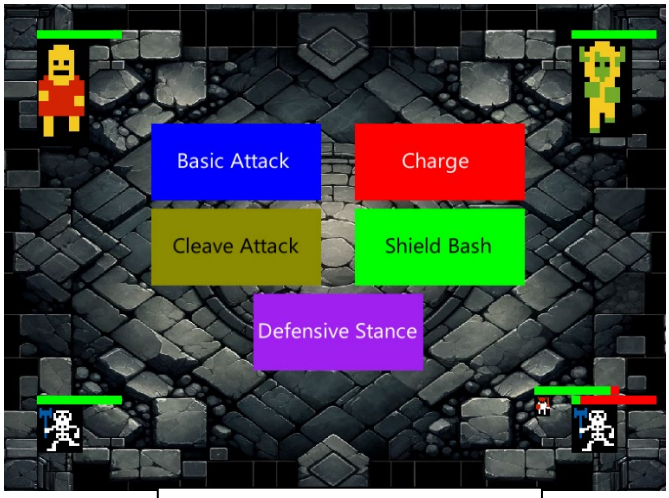


Figure 1.7

Evaluation

At first it was difficult to access the names of a specific attacks as they were stored inside a specific character type. The only way to do this would be to call three different files and access the names of attacks three separate times. However, this strategy wasn't ideal because the player would be using one character class per match meaning I didn't have to display all existing character attacks.

Instead, I came up with a more efficient method where the map.py file that sets the character type would pass in the selected class as a parameter. The battle method would then open a dictionary named 'attacks' that stored all the unique attacks for that class. As I was doing this, it became evident to me that dictionaries are not the most useful for calling on specific indexes and that I would need an index to retrieve a specific attack name. To solve this, I created a 'for loop' that would scan every item in the dictionary for an attack name and then save the attack name into a separate list called 'attack_name'. This way whenever I need to access the names of attacks I have a list with set index positions.

I then duplicated my button code from my previous two buttons and created three more for a total of five buttons, each one set to a different attack. To continue working on the combat, I need to access the damage and stamina values associated with each attack and pass them into each button. This way the turn-based combat is more strategic and diverse. As of right now, all five of my buttons hold the same attack but display different names.

10/07/24

Progress

Achievements	Stumbling Blocks
• Synchronised stamina values with corresponding attack • Implemented stamina factor into combat • Attack button goes 'dead' (turns grey) when player does not have enough stamina to perform an attack • Split Battle class into multiple methods to follow OOP principles • Working with nested dictionaries	• Calling a method where the method name is saved in a variable (the program tries to call the variable name instead of the method inside)

```
attle():
 attacks(self, window, character_type, current_stamina):
    position = pygame.mouse.get_pos()
    if icon.collidepoint(position):
        pygame.draw.rect(window, (0, 75, 139), icon)
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    player_damage = character_type.attack_1()
                    character_type.subtract_stamina(new_attacks
                else:
                    pass
else:
    icon = pygame.rect.Rect(185, 150, 200, 90)
    pygame.draw.rect(window, (128, 128, 128), icon)
    position = pygame.mouse.get_pos()
    if icon.collidepoint(position):
        pygame.draw.rect(window, (90, 90, 90), icon)
```

Figure 1.8

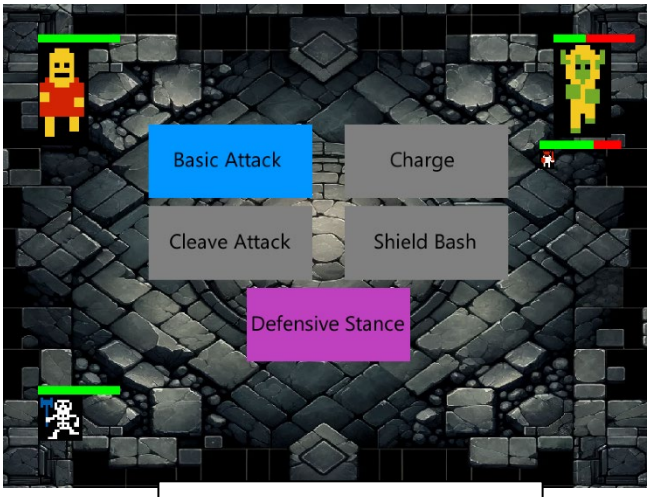


Figure 1.9

Evaluation

To further enhance the turn-based combat I aimed to make use of the stamina values that were carried over from the base files. Because these were already instantiated in the 'attacks' dictionary, implementing them wasn't too hard after a quick google search on calling nested dictionaries.

Getting the pre-existing attack methods to run when the corresponding button is clicked was difficult. My original method was to retrieve the method name from the nested dictionary, give it to 'battle.py' and then run it using the 'character_type' argument given from 'map.py'. Unfortunately, VsCode wouldn't allow me to run a method using a variable that stores the methods name which in turn disrupted my original method. This was a problem because I wanted the attack buttons to work for all of the classes and be class specific. To solve this I had to rename the attack methods to 'attack_1', 'attack_2' and so on. I didn't like this name convention but it allowed me to pass in the correct damage values to the buttons and this naming convention could be used for all classes to make their implementation later on much easier.

Resources

- https://www.programiz.com/python-programming/nested-dictionary

11/07/24

Progress

Achievements	Stumbling Blocks
• Created unique attacks for different character classes • Instantiated all three character types • Set base health, stamina, and armour values for incrementation	• Centring text of different sizes • Removing blue orb after collision • Balancing the character classes with different strengths and weaknesses

```
Rogue(Character):
ef __init__(self, name, max_hp, armor, window):
   super().__init__(name, "Rogue", armor, window, max_hp)
   # Additional attributes and methods specific to the Rogue class
   self.max_stamina = 115
   self.current_stamina = self.max_stamina
   self.stamina_regeneration = 65
   self.base_armor = 2
   self.armor = self.base_armor
   self.base_strength = 18
   self.strength = self.base_strength
   self.attacks = {
       "Quick Jab": {"method": self.attack_1, "stamina_cost": 10},
       "Shadow Blade": {"method": self.attack_2, "stamina_cost": 60},
       "Garrote": {"method": self.attack_3, "stamina_cost": 30},
       "Phantom Strike": {"method": self.attack_4, "stamina_cost": 25},
       "Blade Flurry": {"method": self.attack_5, "stamina_cost": 50}
```

Figure 2.0

```
Mage(Character):
f __init__(self, name, max_hp, armor, window):
   super().__init__(name, "Mage", armor, window, max_hp)
   self.max_stamina = 100
   self.current_stamina = self.max_stamina
   self.stamina_regeneration = 50
   self.base_armor = 1
   self.armor = self.base_armor
   self.base_strength = 30
   self.strength = self.base_strength
   self.attacks = {
       "Basic Spark": {"method": self.attack_1, "stamina_cost": 10},
       "Arcane Bolt": {"method": self.attack_2, "stamina_cost": 20},
       "Void Beam": {"method": self.attack_3, "stamina_cost": 30},
       "Curse Spell": {"method": self.attack_4, "stamina_cost": 15},
       "Empowerment Chant": {"method": self.attack_5, "stamina_cost": 60}
   }
```

Figure 2.1

Figure 2.2

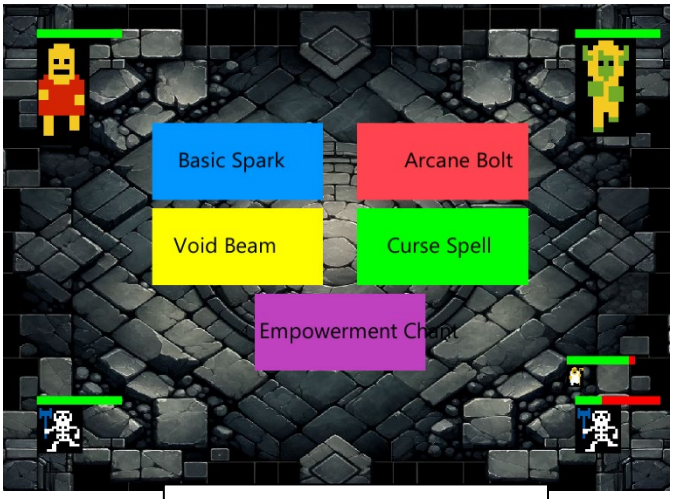


Figure 2.3

Evaluation

Creating the other two character classes and their unique attacks was fairly easy as they were extremely similar to the warrior class. It involved changing the names of the attacks and their corresponding damage values. However, I wanted to make each class feel unique and offer a different approach to the game. I did this by assigning each class a strength and a weakness in the three categories of armour, strength, and stamina. Although this does offer new experiences for each class, I still need to work on balancing the classes as currently the rogue (who specialises in stamina) is the strongest character, and the warrior (who specialises in armour) is the weakest.

I also tried to work on implementing a wave system where the blue orb would reset the map and respawn all enemies instead of ending the game. I figured out the enemies were removed from the map by using the 'remove()' feature in the list that holds them so to respawn them all I had to do was put them back in the list. However, I came across difficulty when finding out how to remove the blue orb after the player makes contact with it. I aim to finish the basics of the wave system tomorrow so the progress from the first round impacts the play of the second round.

12/07/24

Progress

Achievements	Stumbling Blocks
• Successfully implemented wave system • Display wave counter on screen • Restrict player movement to within the dimensions of the window • **Merge** combat branch with main	• Changing enemy positions

```
check_orb_collision(self):
"""
Check if the player has collided with the blue orb.

Returns:
    bool: True if the player has collided with the blue orb, False othe
"""
if self.blue_orb and pygame.math.Vector2(self.orb_position).distance_to
    self.blue_orb = None
    self.wave_counter += 1
    self.enemies = [
        Enemy(GAME_ASSETS["goblin"], [50, 50], self.window),
        Enemy(GAME_ASSETS["orc"], [self.window.get_width() - 120, 50],
        Enemy(GAME_ASSETS["skeleton"], [50, self.window.get_height() -
        Enemy(GAME_ASSETS["skeleton"], [self.window.get_width() - 120,
    ]
```

Figure 2.4

Figure 2.5

Evaluation

I experimented with different ideas of respawning the enemies for a new wave. The original concept I came up with was to implement a countdown timer that would count down from ten seconds once all enemies of that wave were defeated. However, this didn't come to fruition as displaying a live pygame timer on screen was challenging and could take a long time to create. The alternative idea I chose was to remain with the blue orb concept but after collision instead of ending the game all enemies would respawn and the blue orb would disappear. This was much easier to code as opposed to the countdown timer and was just as effective as it serves the same job of giving the player a 'grace period' where they can rest before deciding to enter combat again.

While making this feature, I came across another problem which was that the player could move outside of the window's display and get 'lost'. To fix this I went into the movement code and made it so the player's coordinates can never exceed the dimensions of the window.

I also had ideas of mixing up the enemy spawn positions each round to make the rounds feel different from each other. I thought of doing this by putting the four spawn locations in a list and having the draw() method pick a random spawn from the list however this method meant enemies could all spawn in the same location. I was not able to find a solution to this problem as of now. Consequently, the culmination of my week's progress has resulted in a general turn-based combat system that takes player input and creates a somewhat enjoyable playthrough.

Resources

- https://www.youtube.com/watch?v=UIq3VUzICPU&t=1s

13/07/24

Progress

Achievements	Stumbling Blocks
• Stamina bar implementation • Player levelling and stats improvement • Enemy difficulty • **Merge** stamina bar and main	• Balancing class types

Figure 2.6

```
import pygame

class StaminaBar():
    def __init__(self, x, y, w, h, max_hp):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.max_hp = max_hp

    def drawBar(window, x, y, w, max):
        rect = pygame.rect.Rect(x, y, max, 10)
        pygame.draw.rect(window, (255, 140, 0), rect)
        rect2 = pygame.rect.Rect(x, y, w, 10)
        pygame.draw.rect(window, (0, 0, 255), rect2)
```

Figure 2.7

Evaluation

Implementing the stamina bar wasn't a challenge after I had created the health bar. The code is basically the same as the health bar with minor changes in position and colour. To get the live stamina of the player, I passed in the player's current stamina as a parameter. I chose blue and orange as the colours to separate the look of the stamina bar from the health bar. This makes it easier for the player to read on screen.

To increment the player levels, I used the player.level attribute already created from the base files. I then created a separate method, 'update_stats()' to update the current stats of the player when they levelled up. I did something similar for the enemies as I wanted the game to get slightly harder as the player plays through. I created a new variable to store the enemies' levels in and made it equal to the value of the current wave. This means in wave two the enemy levels would be at two and so on. I then added another 'update_stats()' method for the enemy that would update the stats accordingly.

I also tried to improve the warrior class button on the character selection screen as it was the only button in black and white. I used the listed colouriser website down below. The result isn't perfect but made it blend in with the rest.

Resources

- https://www.img2go.com/colorize-image

20/07/24

Progress

Achievements	Stumbling Blocks
• Class Skills menu created • Switching states in game.py	• Logic problems that come with using a Boolean as a toggle • Getting the state of the game to switch properly • Switching the state of the game back to its original

Figure 2.8

```
ss ClassSkills():

def __init__(self, window):
    self.scroll_image = None
    self.scroll_position = None
    self.window = window
    self.back_image = pygame.image.load(GAME_ASSETS["skills_menu_bac
    self.back_image = pygame.transform.scale(self.back_image, (self.

    self.scroll_image = pygame.image.load(GAME_ASSETS["old_scroll"])
    self.scroll_image = pygame.transform.scale(self.scroll_image, (s

    self.go_back = False
    self.back_button = pygame.Rect(self.window.get_width() / 2, self
    self.font = pygame.font.Font(None, 36)  # Use a default font

def drawButton(self):
    returnButton = pygame.rect.Rect(368, self.window.get_height() -
```

Figure 2.9

Evaluation

To create character skills, I had the idea of creating a small menu that goes over the top of the game as shown in *Figure 2.8*. I was originally skeptical about this idea as I had not switched the 'state' of the game previously and it seemed difficult to pull off. After downloading an old map image, I scanned the code of 'game.py' to follow the logic they use to switch game states. This was tricky initially because I wanted toggle buttons that you could press once to turn on and again to turn off. The only way to make these was by using a Boolean value.

This Boolean value is where most of the problems arose as I didn't understand the logic behind it. My main problem was that I had forgotten to reset the state of the Boolean meaning once the button was toggled on it was stuck in 'on' state forever and the player could never leave. Although it seems simple, this logic took a while for me to get my head around.

At the end of the day my toggle menu was working well enough that I could stage the changes and commit them safely to the new branch. As I develop this branch further I will come closer to merging however as of now, I am still in the early stages of this branch.

Resources

- https://www.youtube.com/watch?v=of9uG6N5cAg&t=609s
- https://stackoverflow.com/questions/52462212/how-to-make-a-toggle-type-button-in-pygame

21/07/24

Progress

Achievements	Stumbling Blocks
• Class Skills buttons created • Skills for warrior class stored inside nested dictionary • Skills can be used anytime on map screen	• Passing variables from one class to another class in python

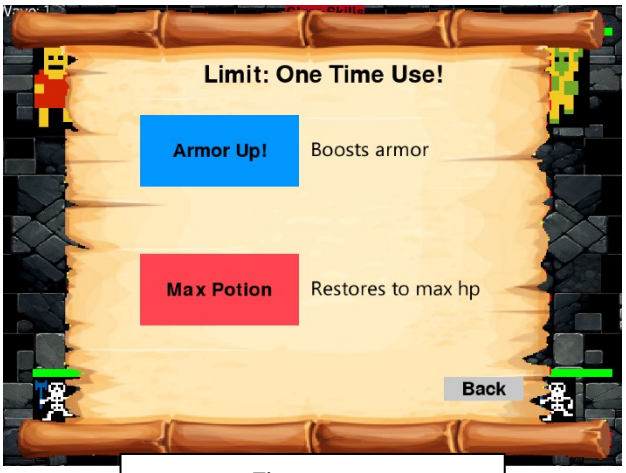


Figure 3.0

```
lassSkills():
 draw_skill_buttons(self):

if self.active_skill_2:
    icon2 = pygame.rect.Rect(185, 325, 200, 90)
    pygame.draw.rect(self.window, (255, 68, 80), icon2)
    back_text = self.font.render(self.skill_names[1], True, (0,
    text_rect = back_text.get_rect(center=icon2.center)
    self.window.blit(back_text, text_rect)
    if icon2.collidepoint(position):
        pygame.draw.rect(self.window, (139, 34, 40), icon2)
        back_text = self.font.render(self.skill_names[1], True,
        text_rect = back_text.get_rect(center=icon2.center)
        self.window.blit(back_text, text_rect)
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    print(self.skill_names[1])
                    self.active_skill_2 = False
```

Figure 3.1

Evaluation

Most of the code from the attack buttons was carried over to the skills menu in order to create the look in *Figure 3.0*. The buttons also shared similar functionality with the exception of the skill buttons being a one time use. I made the skills two per class so I could achieve the bulk of it today and create a more realistic goal. The idea of the skills is to make the gameplay more dynamic and entertaining for the player.

The description beside the button had to be short as pygame protocols don't support multi-lined text meaning if I wrote detailed and long descriptions, the text would go off the page. To continue on, I need to finish creating unique skills for each class and implement their method functionalities into the buttons.

22/07/24

Progress

Achievements	Stumbling Blocks
• Finished unique class skills • Implemented methods into appropriate buttons • Attack unlocks through game progression • Going back and commenting on python files • Adding settings menu • **Merge** of ClassSkills branch with Main	• Converting previous coding practices into OOP format and encapsulation

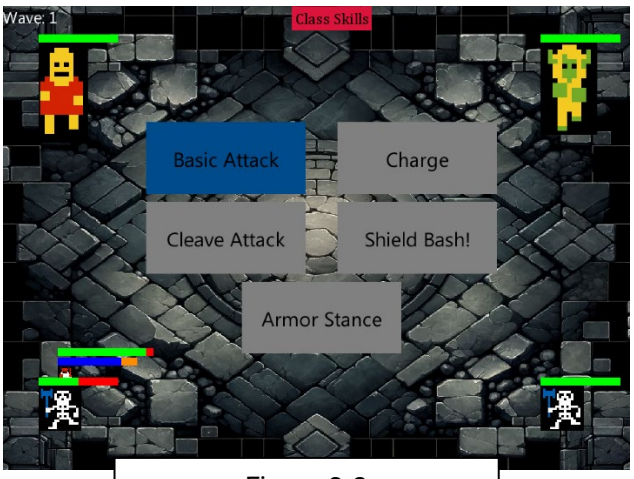


Figure 3.2

```
haracter):
it__(self, name, max_hp, armor, window):
.strength = self.base_strength
.attacks = { #Dictionary that holds Mage's attacks
"Basic Spark": {"method": self.attack_1, "stamina_cost"
"Arcane Bolt": {"method": self.attack_2, "stamina_cost"
"Void Beam": {"method": self.attack_3, "stamina_cost":
"Curse Spell": {"method": self.attack_4, "stamina_cost"
"Empowerment": {"method": self.attack_5, "stamina_cost"

.skills = { #Dictionary that holds Mage's skills
"Regen Trade": {"method": self.skill_1, "description":
"Cursed max": {"method": self.skill_2, "description": "
```

Figure 3.3

Evaluation

To create unique class skills, I copied the dictionary from the Warrior into the Rogue and Mage. I then gave them unique skill names and methods to differentiate their skills from the other classes. To attach the appropriate method to the skill buttons, I followed a similar procedure from when I created the buttons for battle.py. This meant I had to follow a similar naming convention of skill_1, skill_2, shown in *Figure 3.3*.

To add an extra sense of progression in my game I barred the characters other four attacks, as shown in *Figure 3.2*. These attacks unlock at levels 2, 5, 15, and 20 respectively. This means the player has achievable goals to progress towards and is rewarded for progressing and levelling up. This also retains engagement as new content is unlocked by playing the game. To do this, I added an extra condition in battle.py which checked the player's level. If the player's level was not high enough, the button would turn off (go grey). Once the player's level is high enough the button turns on and is now usable.

In light of the due date approaching, I went back through some of my files and added individual and class comments to explain the code more and fill the space. I also implemented the settings.py file from the base files so the settings menu is now functional from the menu screen. At the end of the day, all my goals for creating the skills menu were achieved and everything was working so I merged it to the main branch.

23/07/24

Progress

Achievements	Stumbling Blocks
• New map backgrounds as the player progresses through waves • Player movement is locked when in combat • Encapsulation of two major python classes	• Converting previous coding practices into OOP format • Finding images that fit into a similar theme while being high res

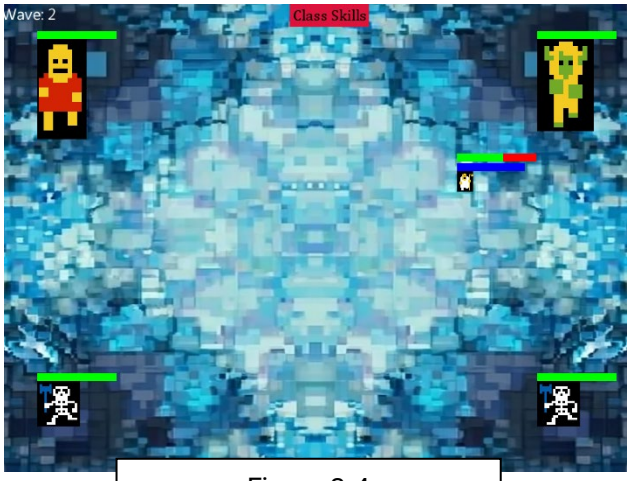


Figure 3.4

```
Draw the game objects on the window.
"""
self.window.fill((0, 0, 0))
if self.wave_counter < 5: #up to wave 5
    self.window.blit(self.map_image, (0, 0))
elif self.wave_counter >= 5 and self.wave_count
    self.window.blit(self.ice_map_image, (0, 0
elif self.wave_counter >= 10 and self.wave_cour
    self.window.blit(self.crept_map_image, (0,
elif self.wave_counter >= 15 and self.wave_cour
    self.window.blit(self.savanna_map_image, (
elif self.wave_counter >= 20: #Wave 20 and on
    self.window.blit(self.map_image, (0, 0))
```

Figure 3.5

Figure 3.6

Figure 3.7

Evaluation

To create new wave backgrounds, I uploaded more images to the map class and set the draw method to draw them when the wave counter was at certain values. Because there are only limited backgrounds, I have set it to the default dungeon map background on waves twenty and above.

Locking the player movement was something I had always wanted to do but never thought out how it would work. When attempting it, it was tricky at first as I couldn't figure out the logic behind it. But as I continued to work on it, I was able to come up with a solution where when the

'in_combat' Boolean is true in the map screen then any presses on the W, A, S, D keys would not advance the player's position until combat was ended.

23/07/24

Progress

Achievements	Stumbling Blocks
• Finished the DFD's for all five features + an overall DFD • Created a context diagram taking in the pygame entity from my feedback • Merged all my UML diagrams into one and modified them to be up to date	• Identifying entities in the data flow diagram • Replacing unnecessary entities with processes

Health Bar DFD (1/6)

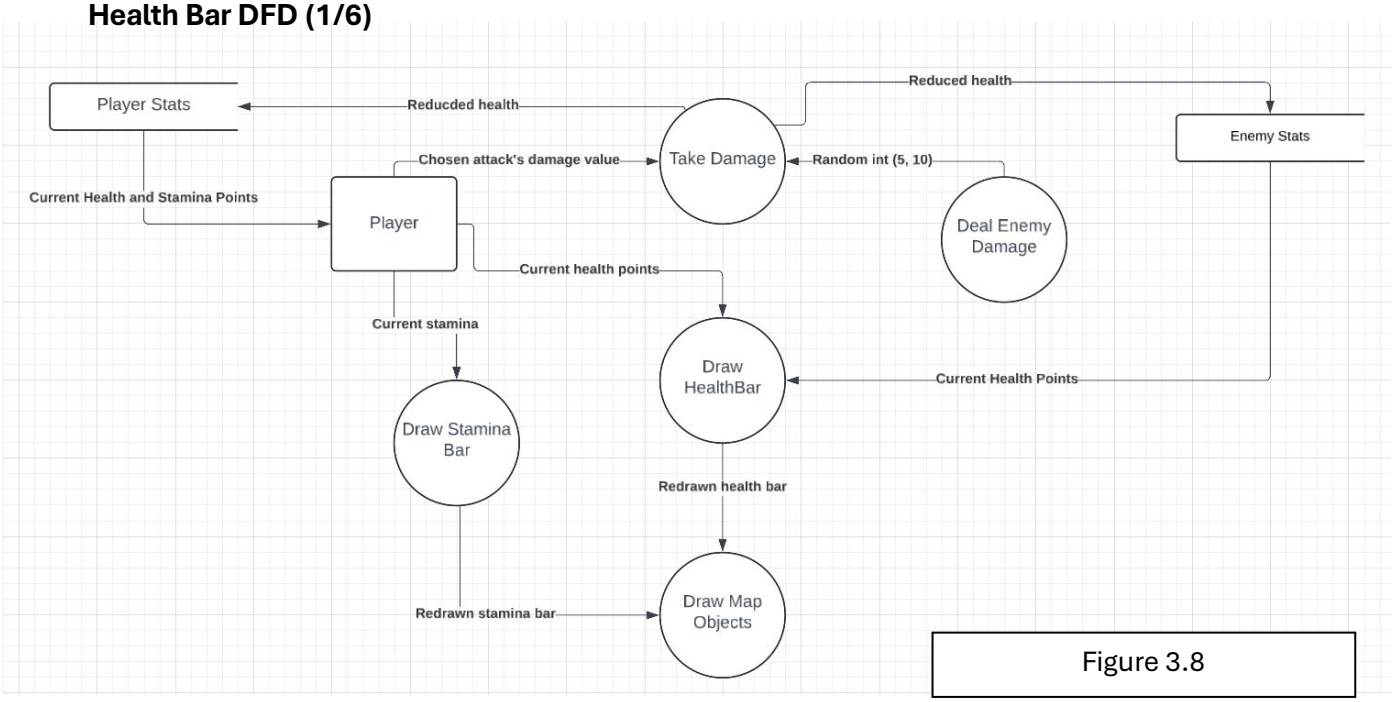


Figure 3.8

Explanation

This DFD shows how the damage value given by the player is used to change the health value of the enemy, updating the data store that houses all the enemy's stats and consequently update how the health bar is drawn. We also see the same flow of data applying to the player as well and the data store holding all the player's stats. Additionally, the player also has a 'draw stamina bar' process that gets the player's current stamina points from the same data store and uses those to draw and redraw the stamina bar.

Turn-Based Combat DFD (2/6)

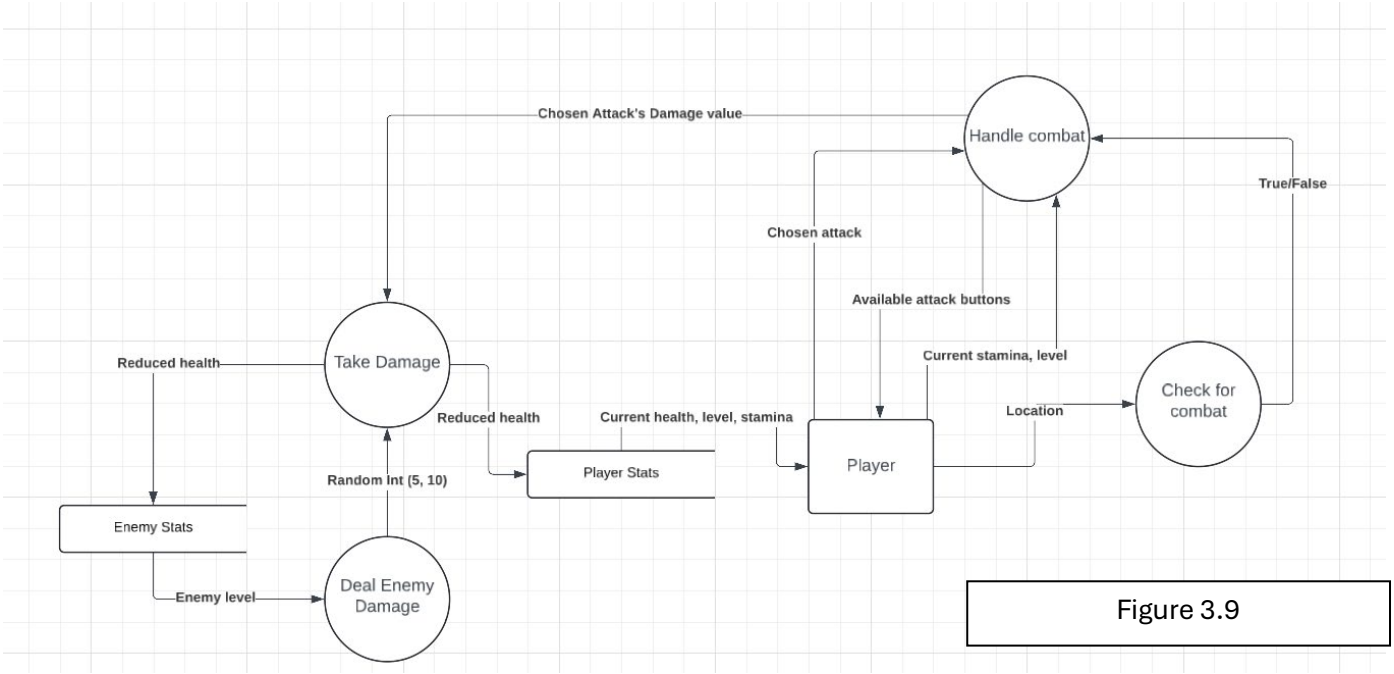


Figure 3.9

Explanation

The player's location is how the game is able to check whether the player is in combat or not. When the player is within a set vicinity of an alive enemy, a Boolean will be switched to True and the 'handle combat' process is run. This process receives the player's current stats to output which attack buttons the player is allowed to use. The player will then click a button which will perform a method holding the corresponding damage value. The damage value is then passed into the enemy's 'take damage' process to subtract the appropriate amount of health. After this the enemy will use their 'deal enemy damage' process to pass in a damage value into the player's 'take damage' process and deduct health points from his stats. The turn is then back with the player until they defeat the enemy or get defeated themselves.

–Continues On Next Page–

Class Skills DFD (3/6)

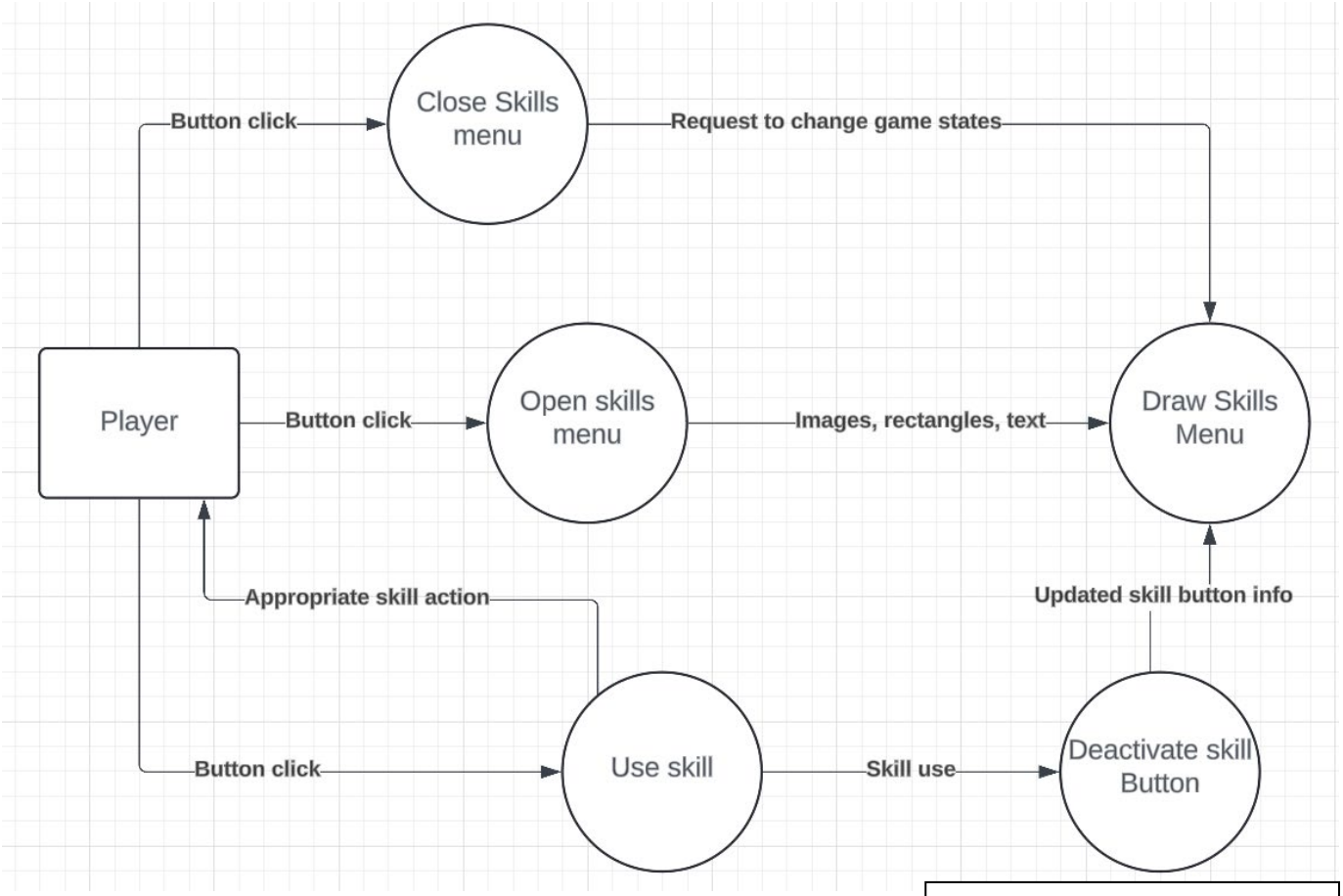


Figure 4.0

Explanation

The player can open the skills menu with one click on the skills menu button located at the top of the map screen. Once this button is clicked on, the game's state is changed to 'Skills Menu' and the 'open skills menu' process will provide the necessary shapes and images to draw the skills menu. When the skills menu is drawn for the player, they can select either of the skills buttons to use a skill or can click the 'back' button which closes the skills menu. When a skill button is clicked on the 'use skill' process will run which will then run the appropriate method and give the player the skill's effects. Simultaneously, as the skills are one time use the skill button's info is updated to make sure the button cannot be used again.

Levelling and Progression DFD (4/6)

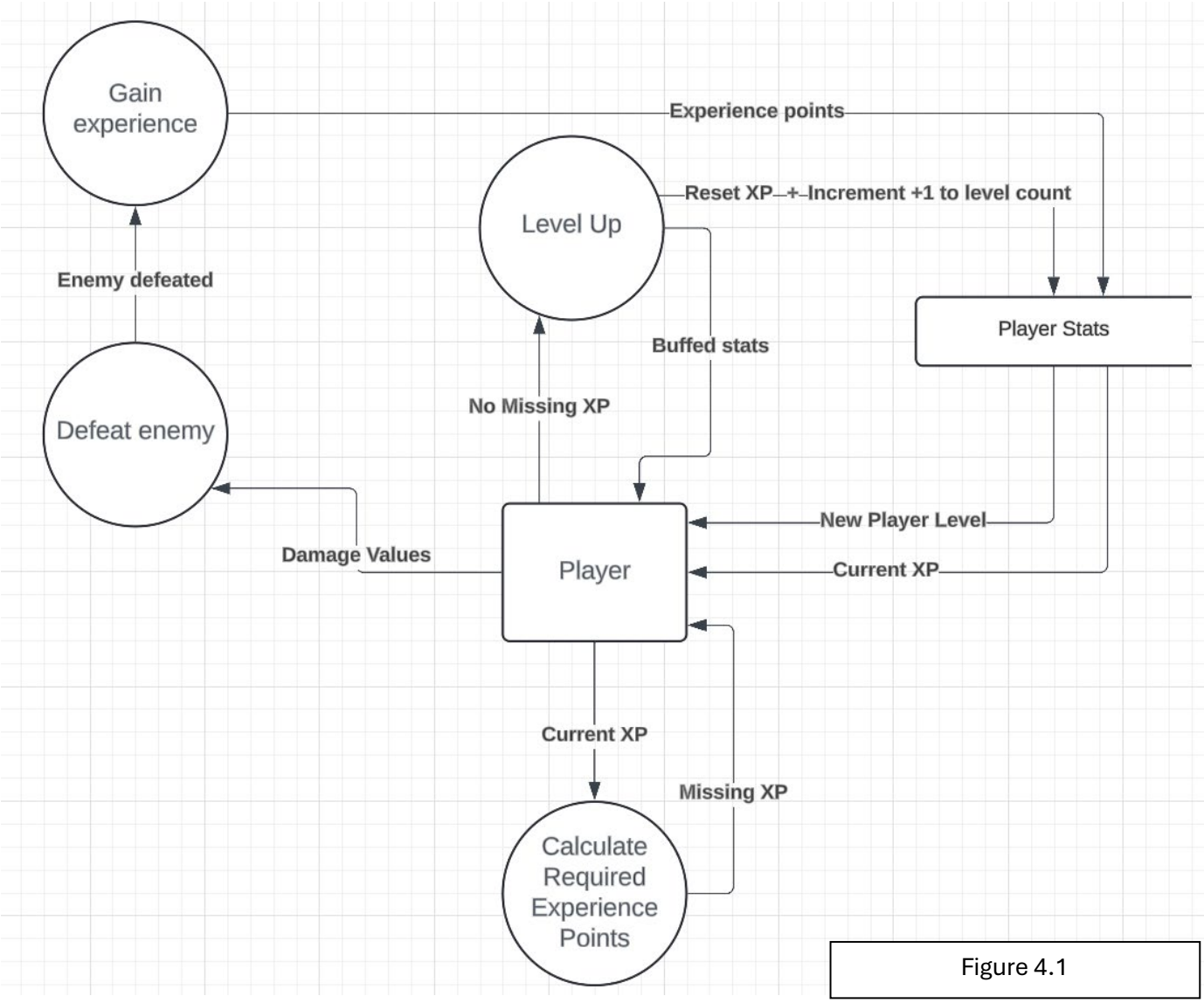


Figure 4.1

Explanation

The player's progression system revolves around receiving experience points (XP) in order to reach the next level and increase their stats for an easier time playing the game. In this game, the player receives XP when they defeat an enemy. When the game is notified that an enemy has been defeated it designates XP to the player, contributing towards their next level up. The system will calculate how much XP is required to move onto the next level to get an indication of how close to levelling up they are. If it's calculated that no amount of XP is needed to level up, then the player is ready to level up through the 'level up' process. This will buff their current stats, and increment their level counter by one, potentially unlocking new abilities (as shown in the turn-based combat DFD). The player's XP is reset to zero so they can start working on gaining XP for their next level up.

World and Levels DFD (5/6)

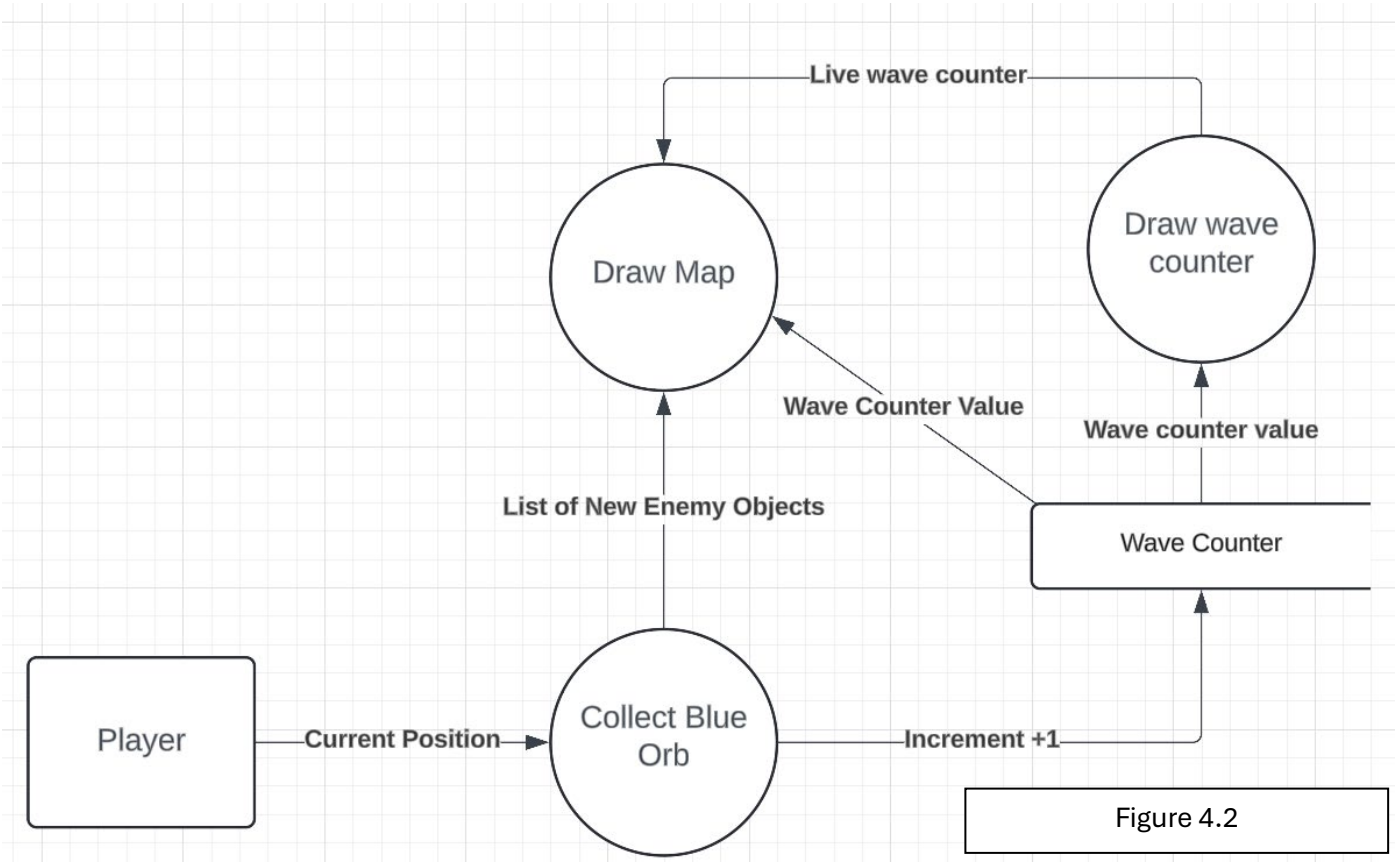


Figure 4.2

Explanation

The game uses a wave system to progress the game. When all the enemies in the map have been defeated, a blue orb spawns in the map's centre. The player's position data is used to determine how close they are to the blue orb and when they have collected it. When the player collects the blue orb, the list that holds the alive enemies is instantiated with four enemies whose levels are equal to the current wave number (on wave 4 enemies are level 4 etc). This is used to redraw the map and have the enemies 'respawn'. At the same time, when the blue orb is collected the wave counter is incremented by one to update the enemies' level. When the map is being redrawn, it is given the current wave to determine which map it will need to draw by loading in a different file.

General/Overall DFD (6/6)

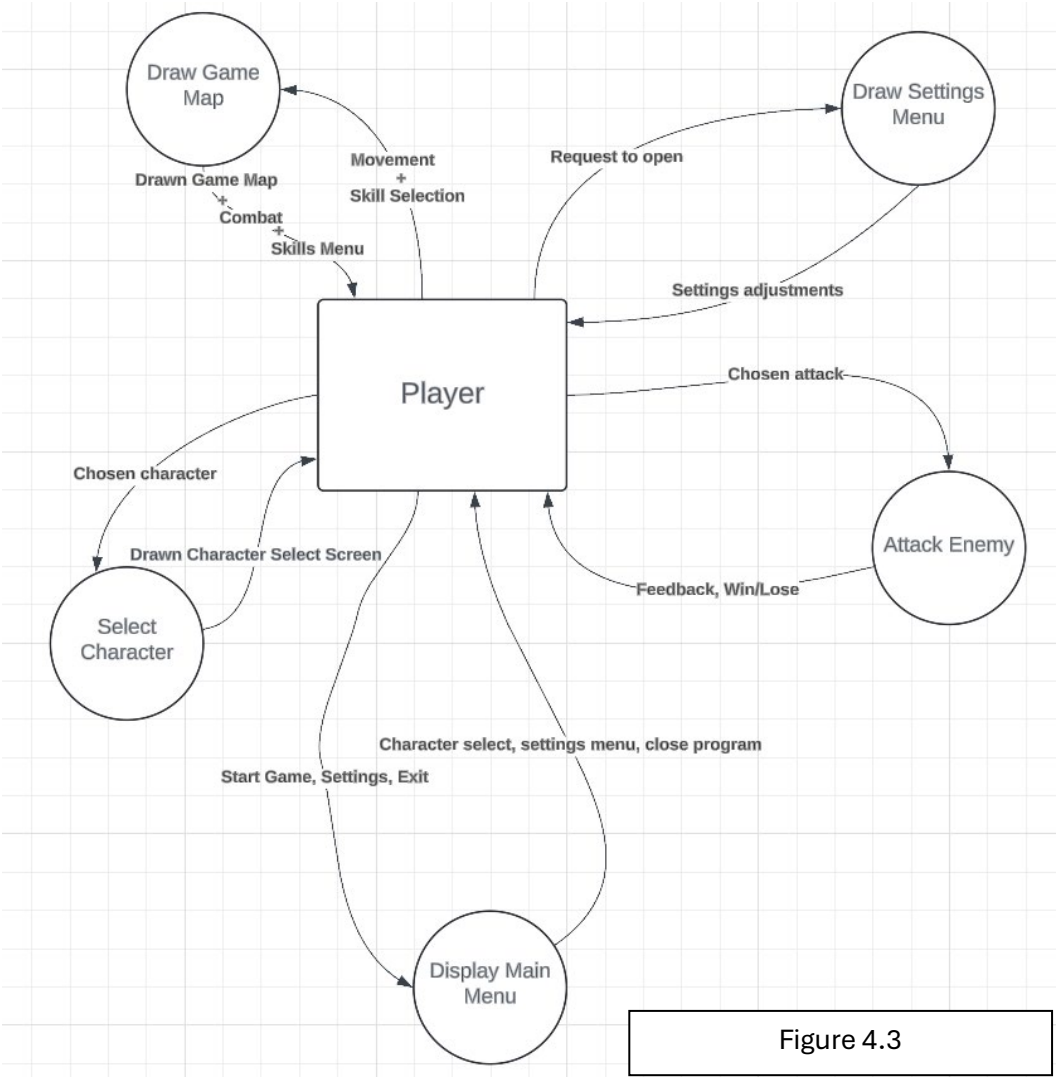


Figure 4.3

Explanation

The general navigation of the game is split up into game states and core actions, all as processes. When first loaded, the game will 'display main menu' and present the player with menu options. The player selects which option they would like, and the process gives them the appropriate data from their selection. This is the same process with all other menu screens including: Select Character and Draw Settings Menu. After finishing their menu selections, the player will use the W, A, S, D keys to move around, to which the game map will constantly redraw based on the player's position. When the player is in combat, they will select their chosen attack and the game will update the player with live combat feedback and information.

King's Quest Context Diagram

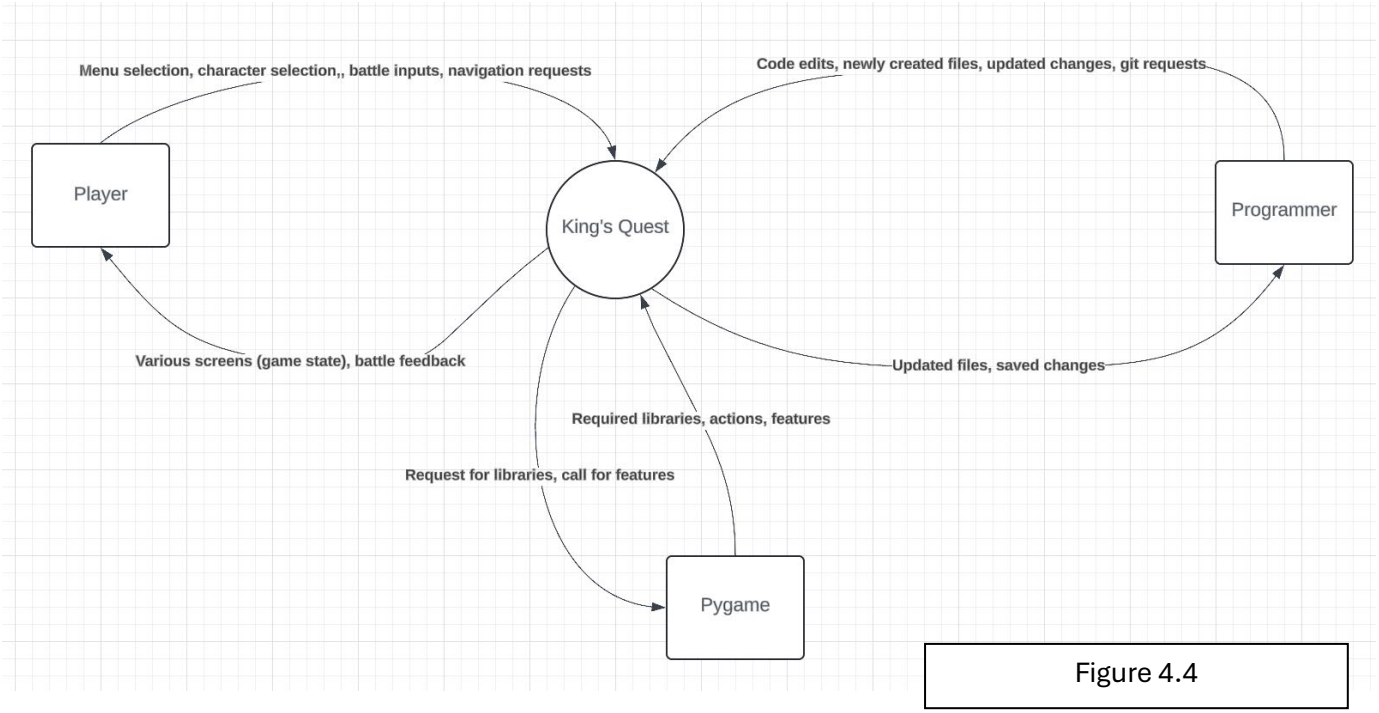


Figure 4.4

Explanation

The context diagram showcases the macro approach of data movement in the King's Quest videogame. The three identified entities are the player who plays the game, the programmer who codes the game in VS Code, and Pygame, a python library contributing huge amounts of resources to the game. Github and VS Code were not included as entities as they were thought to be encompassed by the 'King's Quest' process.

The programmer will create and code features for the game, being saved in the game's repository where they also update features and create git requests such as pushes, pulls, and merges. The main process will let the programmer know that their update are being saved to the files by live updating the repository for them.

The pygame module provides the pre-existing libraries needed to code the game (also used by the programmer). The main process will make requests for these libraries in the form of import statements and references/uses of them in the code.

The player will input their play-through data such as navigation, character movement, button selections, etc. The main process will then update the player's screen accordingly, keeping the game updated live and incrementing the player's data stores due to the feedback they get in-game.

King's Quest UML Diagram

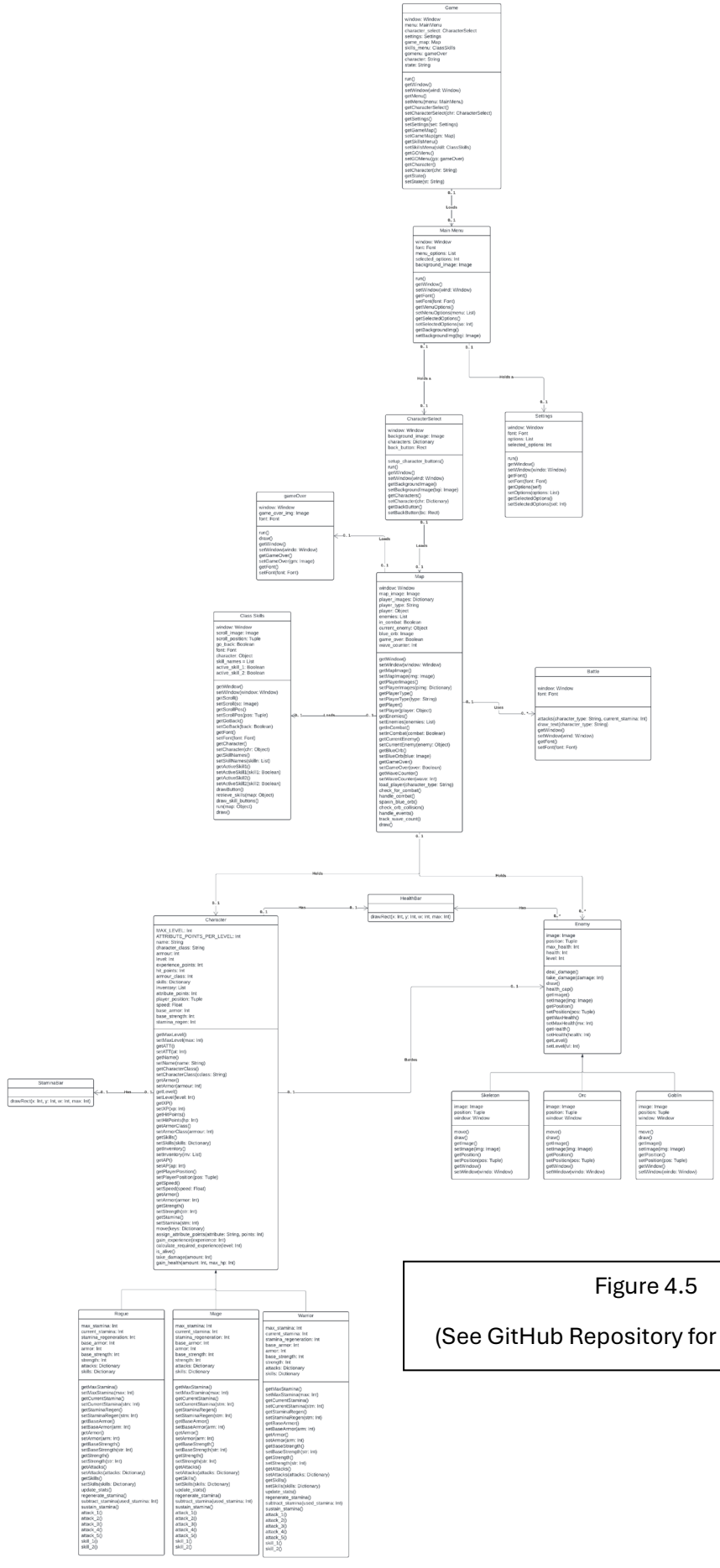


Figure 4.5

(See GitHub Repository for PDF Version)

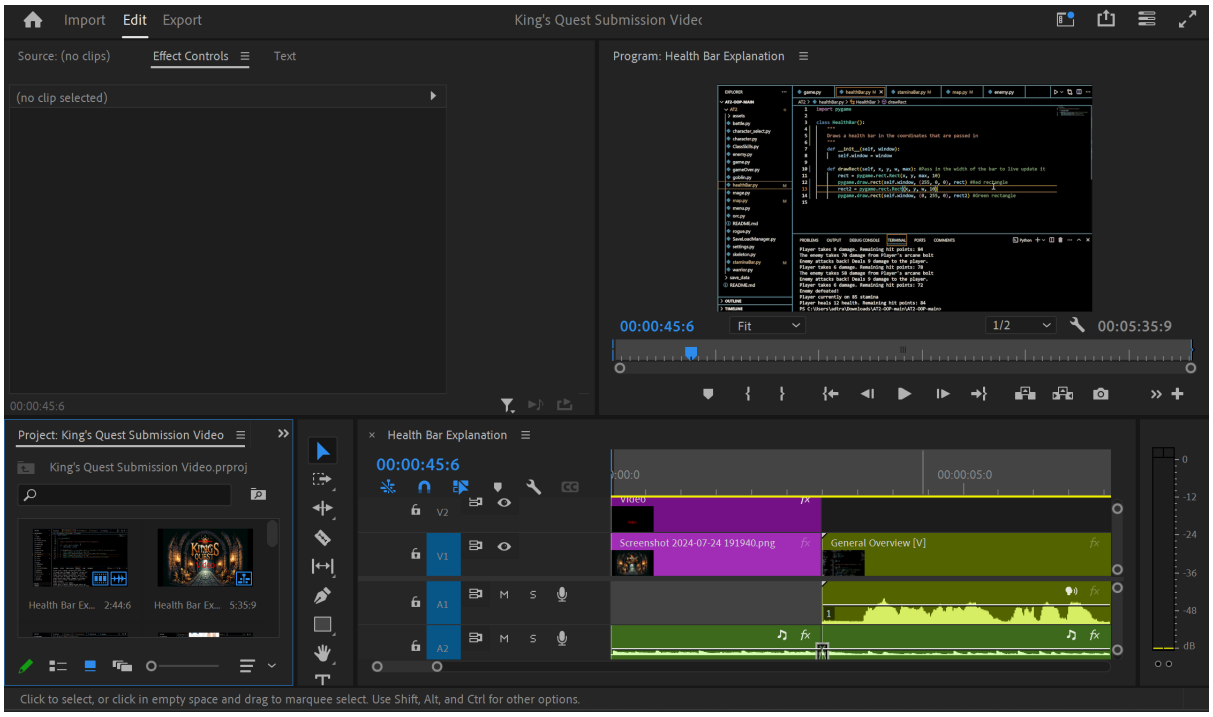
Explanation

All the menu classes (including: Main Menu, Character Select, Settings and Game Over) have a relationship with each other as through navigation you can switch from one screen to another. The map class has relationships with Game Over, Class Skills, and Battle as all of these classes are accessed through the map object. For Game Over and Class Skills this is through a switching of the game state given by the map class. For Battle, the only time it's instantiated is in the map class in the 'handle_combat()' method. The map class holds connections with the Character and Enemy classes as these are instantiated as objects in the map to be drawn. The Character and Enemy both share a relationship with Health Bar as the health bar is instantiated into an object to be used by these two classes. Only Character has a relationship with Stamina Bar as this is only drawn for the player and not enemies. Following the Character and Enemy are the Rogue, Mage and Warrior, and the Skeleton, Orc, and Goblin respectively for their inheritance relationships. These relationships allow the subclasses to be unique while also receiving the attributes and methods from their parent classes.

–Continues On Next Page–

King’s Quest Video

(See SharePoint Folder for Video File)

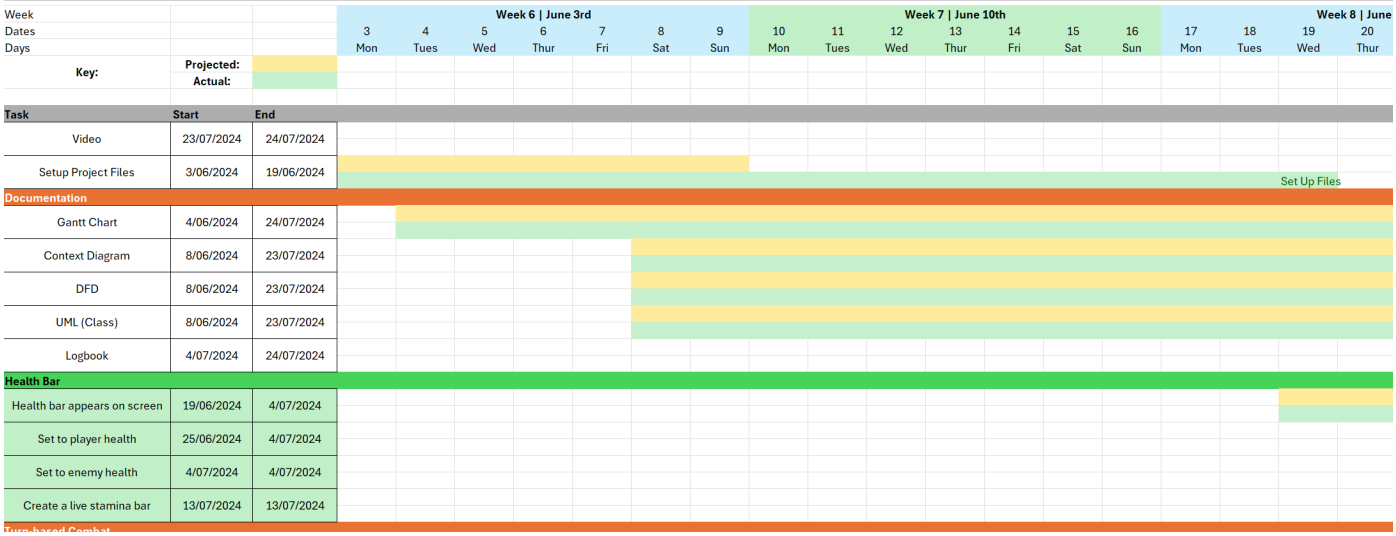


Explanation

In the video’s construction, I focused on six specific categories. These included a general overview where I could talk about how my code functioned and how in-game navigation was possible and then sections on the five core features. When creating the visuals I did my best to blend the code I was talking about with how the feature looked in-game, as there wasn’t enough time to give these their own sections.

King's Quest Gantt Chart

(See GitHub Repository for Excel Version)



Week			Week 6 | June 3rd							Week 7 | June 10th							Week 8 | June	
Dates			3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Days			Mon	Tues	Wed	Thur	Fri	Sat	Sun	Mon	Tues	Wed	Thur	Fri	Sat	Sun	Mon	Tues
Key:	Projected:																	
	Actual:																	
Task	**Start**	**End**																
Video	23/07/2024	24/07/2024																
Setup Project Files	3/06/2024	19/06/2024																
Documentation																		
Gantt Chart	4/06/2024	24/07/2024																
Context Diagram	8/06/2024	23/07/2024																
DFD	8/06/2024	23/07/2024																
UML (Class)	8/06/2024	23/07/2024																
Logbook	4/07/2024	24/07/2024																
Health Bar																		
Health bar appears on screen	19/06/2024	4/07/2024																
Set to player health	25/06/2024	4/07/2024																
Set to enemy health	4/07/2024	4/07/2024																
Create a live stamina bar	13/07/2024	13/07/2024																
Turn-based Combat																		

Explanation

I created the template of my Gantt chart sometime near the assessment's opening week. I created it with the structure I wanted and left the time slots empty so as I worked on the project, I could fill the time slots out with the actual time it took me to create features and reach milestones. Whenever I started a feature, I would record the date I started it in the Gantt chart and set an estimated/ideal date for me to complete that feature by (indicated by the beige coloured bars). As I completed more and more features and reached more milestones, I used the Gantt chart as a checklist of checking which features I needed to complete and what smaller steps had to be done to fill those features. This also helped me check that I had completed all the required work for the assessment.