Analysis

Project Overview

I plan to create an RPG-style game using Pygame with a completable objective, combat system and different environments. The majority of the gameplay will consist of exploring a 2D, top-down world with randomised combat encounters and distinct areas. There will be a playable character who will level up and become stronger as the game progresses. Their level will affect the combat, for which there will be several types of enemies - each with a difficulty that reflects their location.

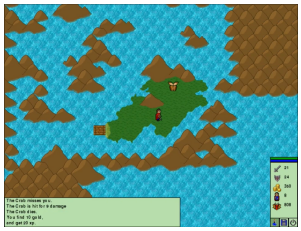
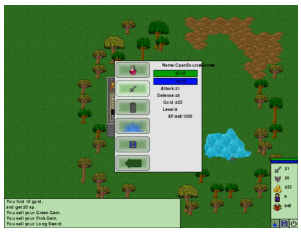
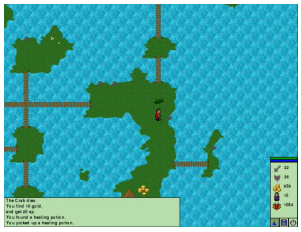
Existing Applications Research

Both of the following examples are open source RPGs made with Pygame.

Dragon Hunt (DH):
[http://emhsoft.com/dh.html]
Dragon Hunt is a single-player RPG where the player battles various enemies, collects & uses items and interacts with NPCs in order to level up their character and battle more difficult enemies. There is a log of everything that happens in the game along with a HUD that displays various pieces of information. Combat occurs randomly and scales in difficulty according to the environment/character level. There are entry and exit points that lead to different areas in the world and each area has its own opponents and features. It is a relatively long game with lots of content.

Features That Will Be Included		
Feature	Justification	Screenshot
Separate areas	I would like to have different areas that can be accessed by entry points, including a cave/underground environment, which is featured in DH.	
Health bars	In battle, health bars are a clear way of displaying the respective health of combatants. These will be easy to implement while making combat easier to understand.	

Player movement	The player is controlled by arrow keys and moves at a steady speed across the screen. There are bounds that the player can collide with, indicated by the graphics on screen.	
Top-down view	The entire game is played from a top-down perspective, with the player sprite maneuvering on the flat screen.	

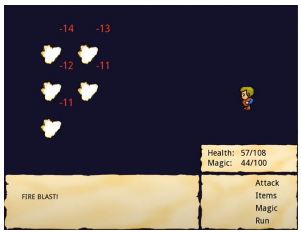
Features That Will Not Be Included		
Feature	Justification	Screenshot
Large world	There is a big, relatively open map in DH with different ways of traversing it. This would be time consuming to code while not adding complexity.	
Menu	The menu has a lot of features that I don't plan on including as this would be unnecessary.	
Shop	A shop system like this would be quite complex; it displays both the player and shopkeeper's items, all of which are interactive for buying/selling.	
Game Log	DH has a text log of everything that occurred in the game. This is unnecessary so I won't be including it.	

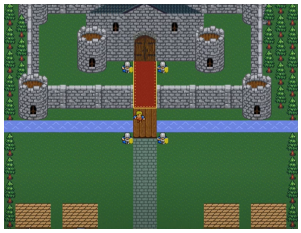
Different terrains	There are a few different types of terrain in DH, such as water and land. Traversing water requires a boat and has different qualities to land. My game will be much smaller and this would be impractical.	

The Stolen Crown (TSC):

[https://github.com/justinmeister/The-Stolen-Crown-RPG]

The Stolen Crown is a simple game revolving around a few different mechanics. Its combat screen is separate from the main screen, which is what I will be implementing in my project. The game is short and succinct, with a linear progression to the objective. The main character has various stats that can be improved along with different combat abilities. NPCs drive the story in this game with short dialogue that gives the player context on their next step.

Features That Will Be Included		
Feature	Justification	Screenshot
Combat	The combat in TSC is relatively simple and comprehensible. I want to incorporate aspects like attack animations and the turn-based style.	
Simple menu	The menu in TSC is very simplistic. I will be focusing on making the menu as simple and functional as possible so I will take inspiration from this.	
Spell	Abilities and spells are staple aspects of most RPGs and will be present in mine. Their use will be the same as in TSC - alternative moves to attacking that cost mana and can have a greater effect.	
Planned combat	When you first launch TSC, after 10 steps a predefined combat encounter will occur. This is helpful for introducing the game and its combat so I will use this idea.	

Features That Will Not Be Included		
Feature	Justification	Screenshot
Second attack	When the character levels up in The Stolen Crown, they gain the ability to attack two times in one turn. I don't plan on including this, levels will instead only increase health/strength/etc.	
Map scrolling	The environment scrolls with the character's movement when they reach a certain distance from the edge of the screen. I plan on instead switching to the next screen when the player reaches the edge of the screen.	
Dialogue	You can talk to NPCs in this game and the dialogue can change depending on how much of the quest you have done. My game won't feature this as it would use more time than it is worth.	

Stakeholders

The game will be reminiscent of early RPGs in gameplay and graphics. This will potentially gain the interest of people aged 30-40, since they would have been the target audience of RPGs released in the 1980-90s[1]. A survey[2] conducted in the US indicates that a fourth of gamers are between ages 34-54 so a game directed to an audience of this age range would be viable. A different 2018 survey[3] regarding video game purchases finds that RPGs are the third most popular genre with 11.3% of purchases. Therefore, a game of the genre RPG targeted towards a common age group would be able to find a market.
As for why people would want to play the game, the responses to an online question[4] suggest that people play RPGs for a multitude of reasons, including escapism, playing with friends, creativity and simple fun. While my game will not support multiplayer, it will offer a storyline and engaging gameplay which would appeal to those who enjoy interacting with a creative environment.
The top-down nature of the game would be particularly reminiscent of older games such as Zelda (1986) or King's Bounty (1990).
The combat system is similar to this type of games' as well, furthering the nostalgic appeal towards middle-aged gamers. It may also service any interest in competition and winning without requiring critical thinking or developed skills.

Simplistic gameplay which is easy to understand would appeal to people who don't have enough time to learn a complex game due to working full-time, alongside clear but simple graphics and an accessible user interface.
The length of the game would accommodate this as well so the audience would be able to play it as a break from work. This would provide an easy way to relax as they will not need to dedicate much time to it.

[1]https://www.pcgamer.com/uk/the-complete-history-of-rpgs/
[2]https://www.statista.com/statistics/189582/age-of-us-video-game-players-since-2010/#:~:text=As%20generations%20have%20grown%20up,are%2065%20years%20and%20older.https://www.statista.com/statistics/189582/age-of-us-video-game-players-since-2010/#:~:text=As%20generations%20have%20grown%20up,are%2065%20years%20and%20older.
[3]https://www.statista.com/statistics/189592/breakdown-of-us-video-game-sales-2009-by-genre/
[4]https://www.quora.com/Why-do-people-play-roleplaying-games#:~:text=from%20their%20own.-,A%20good%20role%20playing%20game%20will%20allow%20the%20player%20to,was%20they%20were%20playing%20as.&text=That's%20why%20people%20enjoy%20role%20playing%20games.

Features

Combat System

While the player explores the map, there will be a chance that they enter combat. If this occurs, the game will change to a combat state and the screen will switch to display the encounter.

The combat menu would consist of several buttons for different actions that can be selected by mouse, including 'Attack', 'Magic' and 'Flee'.
It would also display information such as the player and opponent's respective health along with visuals of each participant.
This is needed to allow the player to make decisions and interact with the battle as well as make judgements based on the information given. For example, if their health is low they may decide to flee the fight.

Combat will be turn-based, alternating between the player and the computer-controlled opponents. Each turn, the player may choose an action to take from the menu. The enemies will have several simple actions they can take (such as attack, block, etc), one of which will be randomly carried out on their turn.
There may be multiple enemies in one encounter, particularly as the game progresses, in which case the player will need to select one to enact on per turn. This can be implemented using a key such as Tab to cycle through the enemies.

There will be visually distinct areas in the game, each with its own potential enemies. These will differ slightly depending on the area, for example an enemy in a later environment may have an additional action.

When either combatant's health has reached 0, the encounter will end. If the player lost, an end game screen will show before they are taken back to the main menu. Otherwise, a short message detailing the results of the encounter, including any 'XP' gained, will show and the game will return to the map exploring state.
The player will also gain an amount of 'XP' proportional to the difficulty of the fight; this allows the player to level up and become stronger when they have enough.
Alternatively, the player can flee the fight and forfeit. While allowing them to escape a losing

situation, this option will lose them ‘XP’, making it harder to level up. When they flee, a message showing how much ‘XP’ they lost will display before the game state returns to the map.

Example:
[https://github.com/Wireframe-Magazine/Wireframe-28]

This example is a simplified version of what I want to achieve without any graphical elements. It is a good example of implementing turn-based combat using object-oriented programming in Python.

```
import random, time

class Action():                                 - A super class for actions the player/enemy can take
    def __init__(self, owner, opponent):       - Constructor defines who is acting on who
        self.owner = owner
        self.opponent = opponent

    def execute(self):
        pass


class Attack(Action):                           - Subclass inheriting from ‘Action’ to represent an attack
    def __init__(self, owner, opponent):
        super().__init__(owner, opponent)

    def execute(self):                          - Overrides the super class’ method
        self.owner.defending = False
        if self.opponent.defending:
            hit = random.randrange(10,20)       - Decreased damage if the attacked entity is defending
        else:
            hit = random.randrange(20,40)
        self.opponent.health -= hit             - Removes from the opponent’s health
        print('{} is hit! (-{})'.format(self.opponent.name, hit))


class Defend(Action):                           - Another subclass for a a defense action
    def __init__(self, owner, opponent):
        super().__init__(owner, opponent)

    def execute(self):
        self.owner.defending = True
        print(self.owner.name, 'is defending!')


class Player():                                 - Class for players, including enemies
    players = []                                - Contains each participant

    def __init__(self, name, inputmethod):     - Constructor, name and input are set
        self.name = name
        self.inputmethod = inputmethod
        self.health = 100
        self.defending = False
        self.players.append(self)               - Adds new player to list

    def __str__(self):                          - Method for displaying information about a player
        description = "Player: {}\n{}\nHealth = {}\nDefending = {}\n".format(
            self.name,
```

```
            '-' * (8 + len(self.name)),
            self.health,
            self.defending)                          - Creates a string to display name, health and defense
        return(description)                          - Returns the string

    @classmethod
    def get_next_player(cls, p):                     - Takes the 'Player' class and object as parameters
        # get the next player still in the game
        current_index = cls.players.index(p)         - Finds the index of the given player in the list
        current_index = (current_index + 1) % len(cls.players)        - Adds 1 to index then finds the
remainder
                                                                when divided by the list length so it will cycle
                                                                through
        while cls.players[current_index].health < 1:             - If the player is dead, keeps looking
            current_index = (current_index + 1) % len(cls.players)
        return cls.players[current_index]            - Returns next player

    def choose_action(self):                         - Method for picking an action
        print(self.name, ': [a]ttack or [d]efend?')
        action_choice = self.inputmethod(['a', 'd'])             - Sets the choices to 'a' or 'd'
        if action_choice == 'a':
            print('Choose an opponent')
            # build up a list of possible opponents
            opponent_list = []                       - List to contain all opponents
            for p in self.players:                   - Iterates through list of all players
                if p != self and p.health > 0:       - Checks that player is not the current one and isn't dead
                    print('[{}] {}'.format(self.players.index(p), p.name))
                    opponent_list.append(str(self.players.index(p)))        - Adds the index to opponent list
            # use input to get the opponent of player's action
            opponent = self.players[int(self.inputmethod(opponent_list))]        - Sets opponent to player's
choice
                                                        from opponent list (using index to find corresponding
                                                    player)
            return Attack(self, opponent)            - Returns construction of 'Attack' class
        else:
            return Defend(self, None)                - Returns construction of 'Defend' class


def human_input(choices):                            - Function for the human player's input, array parameter
    choice = input()                                 - Variable for the player's input
    while choice not in choices:                     - If it isn't in the options provided, they choose again
        print('Try again!')
        choice = input()
    return choice                                    - Returns the choice

def computer_input(choices):                         - Function for the computer player's input
    time.sleep(2)
    choice = random.choice(choices)                  - Computer picks randomly
    print(choice)
    return choice

# add 2 players to the battle, with their own input method
hero = Player('The Hero', human_input)               - Creates an object to represent the player
enemy = Player('The Enemy', computer_input)                  - Creates an object to represent the enemy

# the hero has the first turn
current_player = Player.players[0]                   - Sets the current player to the 'hero'
playing = True
```

```
# game loop
while playing:

    # print all players with health remaining
    for p in Player.players:                          - Iterates through list of players & prints their health
        if p.health > 0:
            print(p, end='\n\n')

    # current player's action executed
    action = current_player.choose_action()           - Creates an object representing the player's choice
    time.sleep(2)
    action.execute()                                  - Calls the 'execute' method to complete the action

    # continue only if more than 1 player with health remaining
    if len([p for p in Player.players if p.health > 0]) > 1:          - Checks if multiple players are alive
        current_player = Player.get_next_player(current_player)       - Moves to next player
        time.sleep(2)
    else:
        playing = False                               - Stops the game loop

for p in Player.players:
    if p.health > 0:                                  - Finds the remaining player
        print('**', p.name, 'wins!')                  - Displays who won
```

Game State

The game will need to switch between menu, map and combat states with different controls and displays for each. This will be implemented using object oriented programming to create classes and objects which will represent parts of the game.

There should be a logical progression between each state - for example, the main menu leads to the map which leads to combat. Any state should offer the option of returning to the main menu as the user may need to stop the game at any time. There will be separate ways of interacting with each state and these should not overlap or interfere with one another.

There will be a super class 'Screen' containing attributes and methods for a general screen with subclasses for each game state/screen. The subclasses will be more specific to their purpose and may overwrite the 'Screen' class methods if necessary.
Each screen may contain buttons, which will be objects of a separate class inheriting from 'Rect', pygame's way of representing rectangles. Each button will have a designated action upon being interacted with. For example, the main menu screen will have a button that exits the game when clicked.

Some screens, such as the map or combat screens, will need to contain sprites so one attribute of the 'Screen' class will be a list of sprites. These will be drawn to the screen using a method.

In order to change between game state or background, each screen will have an attribute that indicates what class the current screen should be. An object will be created to represent the current screen that should be displayed.
Within the game loop, input will be handled and the current screen will be updated if needed as the object will change type.

Example:

I have written a simple psuedocode example, omitting any details that aren't relevant to the structure.

```
class Screen()                              - Super class for all screens
        function constructor()
                […]
                state = self                - Attribute indicating what state the game is in
        function backScreen()               - Method that returns the game to the main menu
                state = mmScreen()
        function draw(area)                 - Method for drawing the screen
                [...]

class mmScreen(Screen)                      - Subclass for main menu, inheriting from Screen
        function constructor()
                [...]
                button1 = Button([...])
                button1.setTask(task1)      - Main menu contains a button and assigns it a purpose
        function task1()
                state = gScreen ()          - Button will call a method to change to the game screen

class gScreen(Screen)
        […]

bg = [...]                                  - Screen to be be drawn on will be made using Pygame
cScreen = mmScreen()                        - Object of the main menu subclass created
running = True
while running == True:                      - Game loop
        if event == K_ESC then
                cScreen.backScreen()        - If Esc key pressed, game will return to main menu
        cScreen = cScreen.state()           - Updates current screen object to the correct state
        cScreen.draw(bg)                    - Draws it onto the Pygame screen
```

Map

Most of the game will be spent exploring the 2D map in order to reach the objective. The map will be implemented using a series of 'rooms' that the player can move between by reaching the edge of the screen.

There will be a progression reflected by the background/environment changing further through the game. This will indicate an increase in difficulty and a sense of nearing the objective. Each area will also affect the potential enemies encountered and have different obstacles in the map itself that may restrict the player's movement. For example, a forest environment may have a fallen log that the player cannot move over.

The rooms can be represented in a 2D array. An object would store the current room which defines what screen is displayed and what the adjacent rooms are. When the player reaches the edge of the screen, the current room will switch to the value in the 2D array which represents the adjacent room.

If there isn't an adjacent room on the side the player moves into, the screen won't change and the player will be prevented from moving further. There will be a graphical indication of which sides can be traversed and which can't.

Whenever the player is moving through the map, there will be a chance they enter a combat encounter. This will be based on time/steps the player has taken in order to be random. However, there will be a predetermined encounter at the start of the game to ensure the player learns the basics of the combat before getting further into the game.

Example:
[https://github.com/Rohan-Bansal/The-Rogue]

I have taken an extract from the code of a roguelike/RPG made using pygame, omitting parts that aren't relevant to the map's functionality.

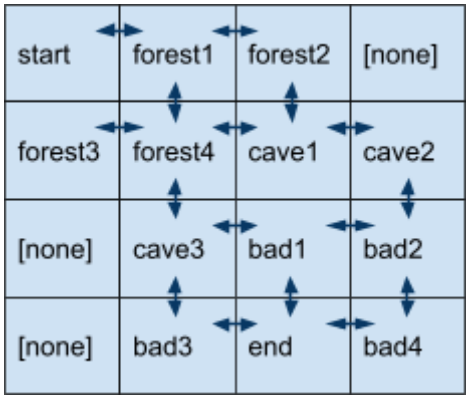
```
[...]

levelIndex = {                                  - Setting up the levels (predefined)
    1 : LevelOne.StageOne(),
    2 : LevelTwo.StageTwo(),
    3 : LevelThree.StageThree(),
    4 : LevelFour.StageFour()
}

rooms = {                                       - Setting up the rooms
    1 : [2, "east"], # from room 1, go to room 2 via direction east.
    2 : [3, "north", 1, "west"], # from room 2, go to room 3 via direction north. Go back to room 1 via
direction west.
    3 : [4, "west", 2, "south"],                - Each room contains the directions it can be exited from
    4 : [5, "north", 3, "east"]
}

roomBorders = {                                 - Setting up the distance the player has to be to exit a
room
    "east" : "hero.x + hero.width >= 960",
    "north" : "hero.y <= 1",
    "west" : "hero.x <= 1",
    "south" : "hero.y + hero.height >= 895"
}

oppDir = {                                      - Swaps a direction to be the opposite
    "east" : "west",
    "west" : "east",
    "south" : "north",
    "north" : "south"
}

startPositions = {                              - Setting up coordinates of the player when entering a
room
    "east" : (20, 440),
    "north" : (440, 800),
    "west" : (930, 440),
    "south" : (440, 80)
}

[...]

def start(heroCoords = None):                   - Function for the start of the game
    global menuActive, itemList, obstCoords
    menuActive = False
    if 'heroChar' in globals() or 'heroChar' in locals():
```

```
        pass
    else:
        inventorySlots.clear()
        for x in range(4, 11):
            inventorySlots.append(sprite("Sprites/Inventory/inventory_slot.png", x * 64, 903, "slot" + str(x)))

    levelIndex[currentLevel].generateGround()                - Generating the level's objects
    levelIndex[currentLevel].generateObstacles()
    levelIndex[currentLevel].generateHazards()
    itemList = itemList + levelIndex[currentLevel].spawnTreasure()
    if musicActive == True:
        levelIndex[currentLevel].startMusic()
    if heroCoords != None:
        spawnHero(heroCoords)                -
    else:
        spawnHero()

[...]

while(True):

            [...]

        if eval(roomBorders[rooms[currentLevel][1]]):
            previousLevel = currentLevel
            toExit = oppDir[rooms[currentLevel][1]]
            levelIndex[currentLevel].destroy()
            currentLevel = rooms[currentLevel][0]
            start(startPositions[rooms[previousLevel][1]])
        elif currentLevel != 1:
            if toExit != None:
                if eval(roomBorders[toExit]):
                    previousLevel = currentLevel
                    toExit = oppDir[toExit]
                    levelIndex[currentLevel].destroy()
                    currentLevel = rooms[currentLevel][2]
                    start(startPositions[rooms[previousLevel][3]])
                elif eval(roomBorders[rooms[currentLevel][3]]):
                    previousLevel = currentLevel
                    toExit = oppDir[rooms[currentLevel][3]]
                    levelIndex[currentLevel].destroy()
                    currentLevel = rooms[currentLevel][2]
                    start(startPositions[rooms[previousLevel][3]])

            if heroSpawned == True:
                        temp = levelIndex[currentLevel].checkCollision(hero)
                        cannotWalkHere = temp

            [...]
```

This is an example map in pseudocode, implemented as a 2D array. The diagram below is a visual representation of the array and shows how the game progression may be implemented while giving the player freedom of choice (start -> forest -> cave -> bad -> end).

map = [[start, forest1, forest2, None], [forest3, forest4, cave1, cave2], [None, cave3, bad1, bad2], [None, bad3, end, bad4]]

currentRoom = map[0,0]



Character

The player will be in control of a character which I will refer to as ‘Bob’ from now on for clarity’s sake. Bob is the player’s way of interacting with the game and they will control Bob’s actions throughout.

On the map, Bob will be represented with a sprite that can be moved around the screen. Bob will react to obstacles if their hitboxes meet one another, for example colliding with a log. In order to progress through the game, the player will have to navigate Bob through the map.

When in a combat situation, Bob will be displayed on one side of the screen but cannot be moved. Instead, Bob’s actions are controlled via the combat options. In these encounters, Bob will be represented as an object of a class used for the combat participants. Information such as Bob’s health, strength, mana, etc will be attributes of this class and called to be displayed/updated.

The player can level up Bob by winning fights and gaining ‘XP’, thus increasing Bob’s maximum health & strength. This means that, though the difficulty of the enemies will increase, Bob’s strength will scale as well. All of these traits will be consistent between encounters.

Bob’s sprite will have an appropriate graphic so the player can see where they are/what they are doing at any given moment. While on the map, the sprite will change according to which direction Bob is facing and enter a walking animation when Bob is moving. During combat, Bob’s stationary sprite will have animations for actions such as attacking or defending.

Requirements

User Stories

User Story	**Priority**	**Story Score**
Combat system	High	80
Map	High	70
Game states	High	60

User controls	High	60
Menu	Low	50

Hardware and Software

I will be using Pygame on Python 3 to code the project. Python can run on a Raspberry Pi, therefore my minimum requirements for hardware will be at least the storage/memory of a Raspberry Pi. A Raspberry Pi has 512MB of RAM, therefore any computer with at least 512MB of RAM should be able to run Python.
Python can be downloaded from www.python.org. I am using Python as it is useful when coding large projects; it is a high-level language that reports bugs when found, making debugging for a long piece of code easier.
Pygame is a set of Python modules designed for creating video games. It uses the SDL[1] library. SDL provides access to keyboard, mouse and graphics hardware - which I will need for the project.

[1] http://www.libsdl.org/

Limitations

Limitation	**Explanation**	**Justification**
NPC interactions	Being able to talk to/interact with NPCs in the environment.	NPCs are often used in games to drive the story/objective. For this project, the focus is on creating a mechanically functional and complex game. Therefore, coding NPCs would be unnecessary as it would take up time and not enable the game to function better.
Character attributes	Having different strengths/weaknesses for the player.	The game will be short and concise. This means that specific levelling will be limited since it would normally take time for a player to gain or implement particular skills. This would also be detrimental to the balance of the game since there could be over/underpowered builds. Testing each attribute thoroughly would take too much time to justify the little complexity.
Difficulties	Being able to alter how hard it is to complete the objective of the game/win fights.	Since the game will have one completable objective, it won't have much replayability. Therefore, different difficulties would most likely go unused.

Success Criteria

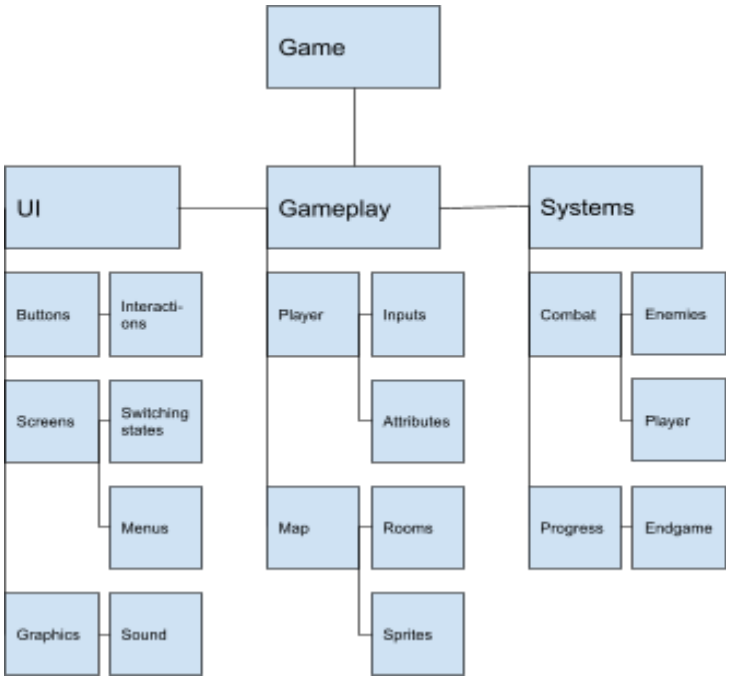
	Success Criteria	**How it is Measured**	**Justification**

Usability	Accessible start game	Should be available within 1-2 clicks from the menu.	Users will expect to be able to easily start the game when it is launched since they will be launching it to play it.
	Consistent combat screen	Compare how quickly users can make decisions in combat before & after the first encounter.	Combat is intended to be an action based part of the game. Therefore, the interface should be consistent so users can make quick decisions.
	Comprehensible map	Measure how many in-game steps the user takes to traverse the map and reach the objective.	Points of interest such as the objective or changes in environment should be easily evident from the map's visuals. This way users will be able to understand their objective and focus on gameplay.
	Natural controls	All major controls (movement, action keys) should be within 1 button press and be intuitive.	Certain keys are often used for certain purposes in games, such as arrow keys/WASD for movement or Z, X & C for actions along with the mouse being the main interaction device. Users will expect these controls based on other games so including them will make the game more intuitive.
	Visual feedback	Significant inputs from the user such as walking/interacting/attacking should have a visual cue.	In order to make the user's inputs and effect on the game obvious to the user, there should be cues like walking animations to indicate that their inputs are working.
Features	Functional map	Test the controls for the player sprite to ensure they can move and attempt to enter other areas.	For the map feature to be considered successful, the player must be able to control their sprite. The different areas should also function as intended with the player being able to enter them when they reach the edge of the previous area.
	Combat system	Enter an encounter and take all available actions. Ensure the encounter ends when it should.	The combat system should start and end correctly. There should be actions the player can take and each should function as intended. Enemies should be able to take turns with the player.
	Working menus	Interact with each button in the menus and check they function correctly.	Every button should be interactive and have the intended effect. The menus should open/close at the correct moments.
	Player movement	Move around the map, attempt different key combinations.	Player will need to navigate the map and control their character. Speed/collisions of the character need to be consistent. Any areas the player shouldn't be able to reach should be

			inaccessible.
Robustness	Confines	Attempt to reach out of bounds by going to the edge of the screen and attempting various key combinations.	The player should only be able to access areas that are available to them. There shouldn't be any way of leaving the game area as that would break the game.
	Progression	Play the game through, attempt to skip areas.	The game should only end when the final enemy is defeated and the player shouldn't be able to reach the enemy before they're supposed to/end the game prematurely.
	No crashes	Spam inputs throughout the game, test bounds	If there are unexpected inputs, the game shouldn't crash and should remain stable. There shouldn't be any point the player can reach that would crash the game.

Design

Decomposition



I have decomposed my game into the contents of each prototype.

The UI model is important to construct first as it would be much more difficult to program the rest of the game without it. I'm including buttons, screens and graphics in this model, though I may not include graphics specifically in the 1st prototype. This model will provide a way for the user to interact with the game, which is imperative.

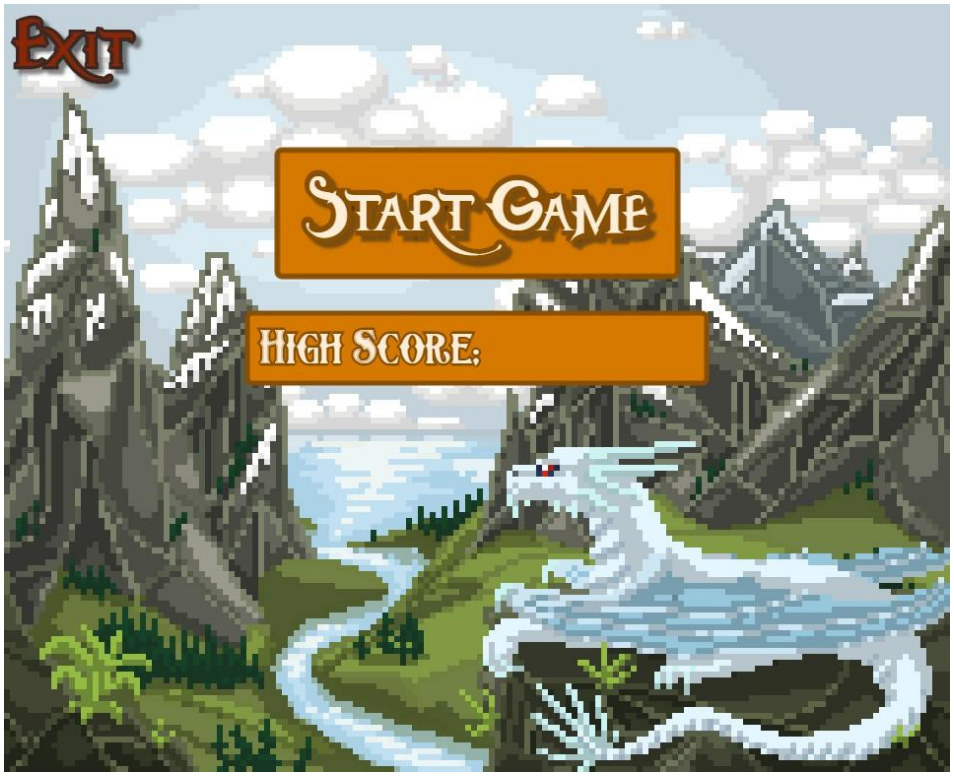
The screens and the structure to switch them will be ideal to code before the game systems. It will be easiest to set up the screens for the game then code the individual systems so I can ensure that they will work together. I may only build the menus to be functional at first and finish them in the final prototype since the game is a higher priority in that regard.
This will also allow my code to be modular as I can set up a class for all screens which all future classes inherit from, including any in future prototypes. Any graphics can be attributes of the screen included in a draw method.

Gameplay includes the player's controls and sprite alongside the map they can traverse and any relevant entities. I could code the player and map screen classes separately to ensure that the controls work, then incorporate it into the rest of the game via my screen structure. Therefore, I will be developing the gameplay modularly as the classes and methods required for it would function independently of the rest of the code.

After this will be systems like combat and progression. The combat system will be the priority since it involves the most complexity. There will be a different screen with new inputs, visuals and sprites. As such, there needs to be a separate player class specifically for combat along with an enemy class. Finally, the game will need a start and end, though this doesn't add a huge amount of technical complexity which is why it is last. I will need to build upon the gameplay by adding a new screen which can be reached from the map to lead to combat. However, I could code the combat system outside of the rest of the code then incorporate it into the main game to retain modularity in my program.

Screen Design

Menu Screen (P1):



For the menu screen, the background will be a fantasy pixel art image/animation to demonstrate the theme of the game. The colours will be natural - green, blue, etc - so the buttons for starting/exiting the game will contrast with orange/red.
The button to start the game will be central to the screen and largest to make it easy to recognise and use.
Below, the player's high score will be displayed so they are aware of how far they have progressed previously. This will be helpful if they haven't played for a while or are competitive about playing.
The exit button is red to show that the game will be closed if clicked on. Red is a colour often used to warn or instil caution, therefore making it less likely the game is closed by accident.
The menu screen will be navigable by mouse as the start game and exit can be clicked on to perform their respective operations.
If the player attempts to click elsewhere on the screen, nothing should happen. Each click should be validated to ensure it only has an effect if it is over a button. No other keys should have an effect in this screen.

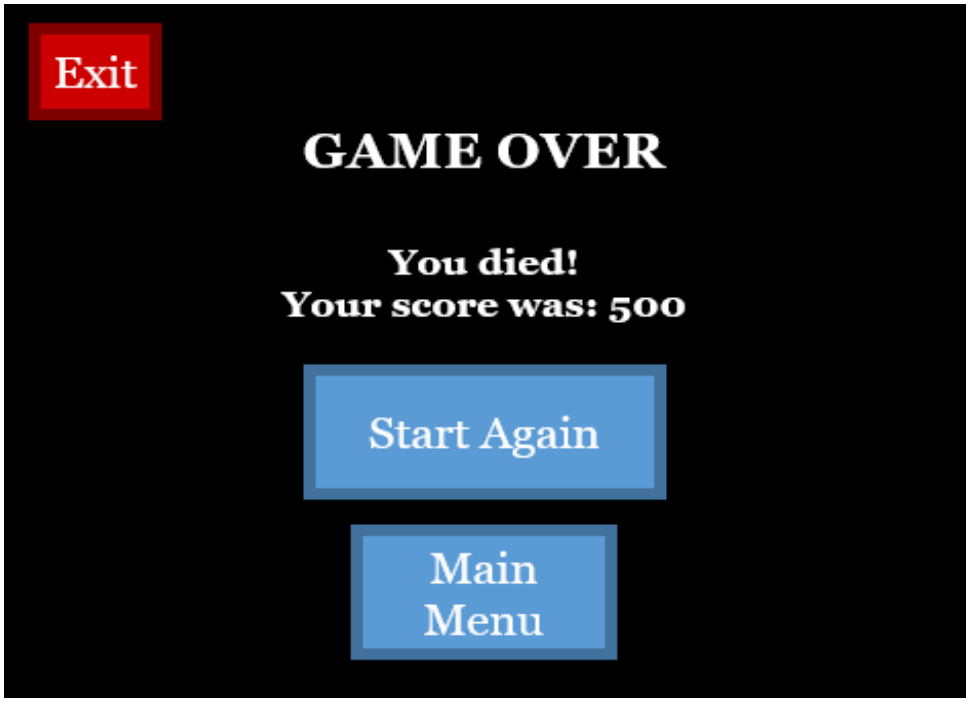
Map Screens (P2):

The map screens will consist of several distinct backgrounds with the player sprite imposed over them.
Environmental features that the player cannot walk through - such as boulders or the edge of the map - will be clearly shown by the graphics. Areas that can be moved over will appear flat and bland. If the edge of a screen doesn't connect to another room, this will be indicated by a visual blockade to prevent the player attempting to go that way.
The theme of the background will indicate how far through the game the player is, with darker and more unnatural colours/features as the game progresses.
The map screen will also relay information to the player via text appearing when a new section is reached. Visuals like the player sprite changing directions will also provide feedback on the user's movement.
The player will be able to use arrow or 'WASD' keys to move but other keys should not have an effect and the player's movement in the induced direction should stop when the key stops being pressed. Any buttons should have the same validation as the menu screen's.

Combat Screen (P2):

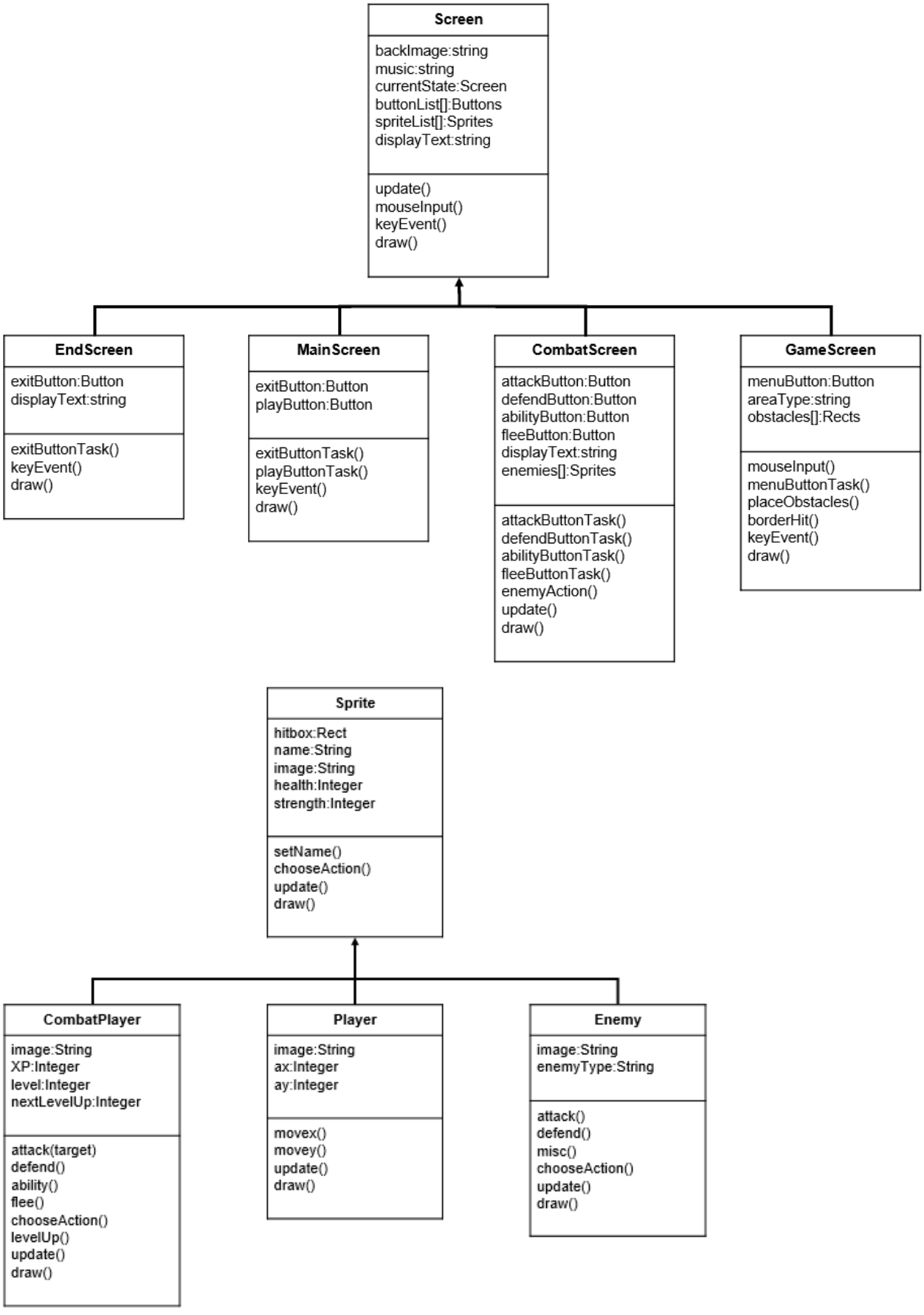
The combat screen will be mostly consistent across each encounter with only the enemies varying. There will be visuals of the player character and any enemies involved, including health bars for each participant to make their status readable. A pointer will indicate the selected enemy so the player knows which they are taking action on.
The interface will be in the lower half of the screen. Each button will have clear, distinct labels in order to make their function obvious to the player. The ‘Ability’ and ‘Flee’ buttons have distinctive colours to show that they are special actions. In order to warn and discourage the player from fleeing fights, red has been used for that button. The buttons are situated apart from one another so the player is less likely to accidentally click on the wrong one.
In the centre of the interface will be information that the player may use, such as text showing the last action. The player’s stats will also be displayed and updated so the player may keep track of them during the encounter. When the event ends, the results will be shown in this section of the screen including any updates to the player’s level.
Neither the player nor enemies should be able to take an action when it is not their turn, so their inputs will be validated to ensure it is the correct turn beforehand. Additionally, clicking should not cause anything to happen unless it is over a button.

End Screen (P3):



The end game screen doesn't need much in terms of visuals or function. It will display relevant text, such as whether the player died or won and what their score was. The button to start again will be central to make playing again easy. There will be an exit button to close the game directly in case the player wants to stop after the game ends. There will also be a central but smaller main menu button in case the player wants to return there.

Structure Design


Screen
backImage:string
music:string
currentState:Screen
buttonList[]:Buttons
spriteList[]:Sprites
displayText:string
update()
mouseInput()
keyEvent()
draw()
EndScreen
exitButton:Button
displayText:string
exitButtonTask()
keyEvent()
draw()
MainScreen
exitButton:Button
playButton:Button
exitButtonTask()
playButtonTask()
keyEvent()
draw()
CombatScreen
attackButton:Button
defendButton:Button
abilityButton:Button
fleeButton:Button
displayText:string
enemies[]:Sprites
attackButtonTask()
defendButtonTask()
abilityButtonTask()
fleeButtonTask()
enemyAction()
update()
draw()
GameScreen
menuButton:Button
areaType:string
obstacles[]:Rects
mouseInput()
menuButtonTask()
placeObstacles()
borderHit()
keyEvent()
draw()
Sprite
hitbox:Rect
name:String
image:String
health:Integer
strength:Integer
setName()
chooseAction()
update()
draw()
CombatPlayer
image:String
XP:Integer
level:Integer
nextLevelUp:Integer
attack(target)
defend()
ability()
flee()
chooseAction()
levelUp()
update()
draw()
Player
image:String
ax:Integer
ay:Integer
movex()
movey()
update()
draw()
Enemy
image:String
enemyType:String
attack()
defend()
misc()
chooseAction()
update()
draw()

Variable	Data Type	Validation	Justification
gMap	2D Array		The map will be a series of connected rooms so a 2D array is suitable for storing each room in a grid.
running	Boolean		This variable will decide whether the game should be running or not and is checked before the game loop each iteration.
mapx	Integer		Stores the row that the current room is in within the 2D array. In order to be used or manipulated it must be an integer.
mapy	Integer		Stores the column that the current room is in within the 2D array. In order to be used or manipulated it must be an integer.
Screen			
Attribute	**Data Type**	**Validation**	**Justification**
playButton	Button	When clicked, it should perform its given task.	All buttons will need a range of attributes so a class inheriting from Rect is needed to represent them.
state	Screen		This attribute is used when switching screens so will always be an object of one of the Screen classes.
spriteList	Array		An array is suitable as any sprites currently on the screen must be kept track of so they can be drawn.
Player			
Attribute	**Data Type**	**Validation**	**Justification**
x	Integer	Shouldn't be able to intersect obstacles/ escape map.	The x and y coordinates of the player are best represented as integers so they can be manipulated.
dy	Integer	Should change when the relevant key is pressed.	The velocity of the character should be able to be added to the player's coordinates.
image	String		Needs to contain the file name of the image representing that itself so Pygame can display it.

name	String		Stores the name of the player which will be a word, used to identify/display/etc.
Button			
Attribute	**Data Type**	**Validation**	**Justification**
text	String		Each button should have text indicating its purpose to the player.
colour	Tuple	If mouse is over the button, its colour should change.	A tuple will be ideal as Pygame recognises it as the data structure for colour.

Test Data

To test my first prototype, I will focus on swapping between screens and the functionality of anything on them. This will include testing each button on any screen to ensure it works as intended. I will also need to switch between any available screens to check that they swap when and as desired. Any other controls or keys will also require testing to see if there are any inputs that can produce unintended effects. This would allow me to fix any errors which might let the player do things they shouldn't or crash the game.

For my second prototype, I will be adding more user controls. This means each new input will be tested alongside any potential combinations of inputs that could break the game. More screens will be added so I will test switching to and pressing inputs on those. If I fixed any bugs found in the previous prototype, testing them will allow me to verify if they are solved. Entering and exiting combat should function as should any buttons in the combat screen. Every action the player and enemies can take will be tested, alongside any potential ends to combat and where they should take the player.

The third prototype will introduce a progression system to the game. In order to test this, I will try to finish the game and ensure it ends as and when expected. Also, any XP/levels/score gained should be accurately stored and displayed within the game. I will likely be adding a new screen to handle the end of the game so any buttons or controls within that should be tested to ensure that they are functional.

Post Development Test Data

When my project is finished, I will need to test the entire game to ensure that it can be completed. All of the systems should be accessible and lead towards the end of the game. This will consist of starting the game from the menu; entering the map and moving around; entering and finishing combat encounters; and finishing the game in all possible ways. This is needed to ensure that the game is playable and will therefore meet the user's expectations.

To verify that my game is robust, I will need to test for any unexpected events as well as the game's confines. In order to make sure the game will not crash, I would spam inputs or use unintended inputs in each screen. This will prove that the game can handle unanticipated events without crashing or otherwise breaking.
When moving around the map, the player should not be able to reach places such as within obstacles or outside the map bounds. Therefore, I will test the confines by attempting to intersect obstacles and escape the map from all angles and by spamming the movement keys. If the player cannot enter those areas then the confines will be successful.
It should not be possible to win the game without defeating the end boss - otherwise the player will be able to cheat. To test for this, I would try each combat resolution with the boss. If any conclusion apart from winning the combat results in winning the game, it is not robust and can be broken.

Prototype 1

Introduction

In prototype 1, I will code the screens that my game will use. This will include the main menu, game screen and the foundation of the combat screen. Buttons and their functions will be added alongside any switches between screens.

Algorithm Design

The user stories this prototype will cover are the game states, menu and some user controls.

```
gMap = [["", "", "", ""],["", "", "", ""],["", "", "", ""],["", "", "", ""]]
mapx = 0
mapy = 0
running = True

class Screen()
    procedure constructor()
        state = self
        [...]

    procedure backScreen()
        state = menuScreen()

    [...]

class menuScreen(Screen)
    [...]

    procedure keyEvent(key)
        if key == Esc then
            exit()
        elif key == Enter then
            play()
```

```
        procedure play()
                state = gMap[mapx, mapy]

class gameScreen(Screen)
        [...]
        procedure keyEvent()
                if key = Esc then
                        backScreen()
                [...]

class combatScreen(Screen)
        [...]

class endScreen(Screen)
        [...]

class Button(Rect)
        [...]

forest1 = new gameScreen([...])
[...]
lava3 = new gameScreen([...])

gMap[0,0] = forest1
[...]
gMap[3,3] = lava3

backdrop = [pygame screen]
currentScreen = new menuScreen([...])

while running == True
        currentScreen.keyEvent(playerCharacter)
        currentScreen.mouseInput()
        currentScreen = currentScreen.state()
        currentScreen.draw(backdrop)
```

Implementation

```
from pygame import *
from random import *

init()

#~---------------------------Initial Variables---------------------------~

gMap = [[None, None, None, None, None],[None, None, None, None, None],[None,
None, None, None, None],[None, None, None, None, None],[None, None, None, None,
None],[None, None, None, None, None]]
mapx = 0
mapy = 1
width = 800
height = 600
running = True
score = 0

#~---------------------------Pygame Setup---------------------------~
```

```
backdrop = display.set_mode((width,height))
display.set_caption("Game")
icon = image.load("drgn.png")
display.set_icon(icon)
clock = time.Clock()

#~---------------------------Screens----------------------------~
```

Introduction: This will contain classes for each screen including a parent class for all screens to inherit from since they share similar qualities. Each screen will have buttons and an attribute used for switching screens. Therefore, this section will cover part of the game states and menu requirements.

```
class Screen():                #Super class for all screens to inherit from
      global mapx, mapy, gMap, PC          #Global variables that are needed
both inside and outside the classes

      def __init__(self):
            self.listOfButtons = []        #Attribute to contain any buttons on
the screen
            self.state = self  #Attribute used to switch screens

      def mouseInput(self,pos):            #Method for handling mouse input
across screens
            for b in self.listOfButtons:
                  b.checkClick(pos)
            return self.state

      def keyEvent(self):      #Empty methods to be overridden
            pass

      def draw(self,pScreen):
            pass

      def collide(self):
            pass

      def backScreen(self):    #Method that always returns to the main menu
            self.state = MenuScreen()

      def playGame(self):      #Method that switches the screen to the current
map screen
            self.state = gMap[mapx][mapy]

      def exitGame(self):
            self.state = None

      def startGame(self):     #Method to set up all of the game screens
            PC.x = 100
            PC.y = 100
            mapx = 0
            mapy = 0
```

```
            start = GameScreen("forest")
            forest1 = GameScreen("forest")
            forest2 = GameScreen("forest")
            forest3 = GameScreen("forest")
            forest4 = GameScreen("forest")
            cave1 = GameScreen("cave")
            cave2 = GameScreen("cave")
            cave3 = GameScreen("cave")
            bad1 = GameScreen("bad")
            bad2 = GameScreen("bad")
            bad3 = GameScreen("bad")
            bad4 = GameScreen("bad")
            end = GameScreen("bad")
            gMap[0][1] = start
            gMap[1][1] = forest1
            gMap[2][1] = forest2
            gMap[0][2] = forest3
            gMap[1][2] = forest4
            gMap[2][2] = cave1
            gMap[3][2] = cave2
            gMap[1][3] = cave3
            gMap[2][3] = bad1
            gMap[3][3] = bad2
            gMap[1][4] = bad3
            gMap[3][4] = bad4
            gMap[2][4] = end

      def startAgain(self):
            self.startGame()
            self.playGame()

class MenuScreen(Screen):                    #Class for menu screen
      def __init__(self):
            super().__init__()
            exitButton = Button(20,20,100,50,"
Exit",[180,40,40],"georgia",40,(255,240,240))
            exitButton.setTask(self.exitGame)
            self.listOfButtons.append(exitButton)
            playButton = Button(280,170,240,70," Play
Game",[180,100,40],"georgia",45,(255,250,240))
            playButton.setTask(self.playGame)
            self.listOfButtons.append(playButton)
            resetButton = Button(300,270,200,55," Start
Again",[180,100,40],"georgia",35,(255,250,240))
            resetButton.setTask(self.startAgain)
            self.listOfButtons.append(resetButton)
            self.bg = image.load("IMG_0270.png").convert()
            self.bg = transform.scale(self.bg, (800,600))

      def keyEvent(self):            #Method for handling key input, calls mouse
input method
            for e in event.get():
                  if e.type == QUIT:
                        self.state = None
```

```
                elif e.type == MOUSEBUTTONDOWN:
                        self.state = self.mouseInput(e.pos)
                elif e.type == KEYDOWN:
                        if e.key == K_ESCAPE:
                                self.state = None

        def draw(self,pScreen):         #Method for drawing main menu
                pScreen.blit(self.bg, (0,0))
                for b in self.listOfButtons:
                        b.draw(pScreen)


class GameScreen(Screen):                       #Class for game screens

        def __init__(self,areaType):
                super().__init__()
                self.areaType = areaType        #Attribute that defines the type of
the room
                menuButton = Button(20,20,120,50,"
Menu",[100,150,200],"georgia",40,(240,250,255))
                menuButton.setTask(self.backScreen)
                self.listOfButtons.append(menuButton)
                self.listOfObstacles = []               #Attribute for any obstacles
on the screen
                self.placeObstacles()
                if self.areaType == "forest":
                        self.bg = image.load("foresbg.png").convert()
                elif self.areaType == "cave":
                        self.bg = image.load("cavebg.png").convert()
                elif self.areaType == "bad":
                        self.bg = image.load("badbg.png").convert()
                self.bg = transform.scale(self.bg, (800,600))

        def placeObstacles(self):
                potentialCoords =
[(505,90),(370,225),(150,350),(100,200),(260,100),(510,350)]
                numbofobs = randint(3,5)
                x = 0
                while x < numbofobs:
                        typeob = randint(0,2)
                        coords = potentialCoords[randint(0,5-x)]
                        potentialCoords.remove(coords)
                        obs = Obstacle(0,0,170,90,"rok.png")
                        if self.areaType == "forest":
                                if typeob == 0:
                                        obs = Obstacle(0,0,170,90,"rok.png")
                                elif typeob == 1:
                                        obs = Obstacle(0,0,100,100,"bsh.png")
                                else:
                                        obs = Obstacle(0,0,120,90,"logg.png")
                        elif self.areaType == "cave":
                                if typeob == 0:
                                        obs = Obstacle(0,0,70,140,"stalagmite.png")
                                elif typeob == 1:
```

```
                              obs = Obstacle(0,0,100,100,"cob.png")
                        else:
                              obs = Obstacle(0,0,70,70,"hol.png")
                  elif self.areaType == "bad":
                        pass
                  obs.giveCoords(coords)
                  self.listOfObstacles.append(obs)
                  x += 1

      def draw(self,pScreen):  #Method to draw game screen, adds relevant
background
            pScreen.blit(self.bg, (0,0))
            PC.draw(pScreen)
            for o in self.listOfObstacles:
                  o.draw(pScreen)
            for b in self.listOfButtons:
                  b.draw(pScreen)

      def keyEvent(self):            #Method for key input, calls the player
character's key input method
            for e in event.get():
                  if e.type == QUIT:
                        self.state = None
                  elif e.type == MOUSEBUTTONDOWN:
                        self.state = self.mouseInput(e.pos)
                  elif e.type == KEYDOWN:
                        if e.key == K_ESCAPE:
                              self.backScreen()
                  PC.keyInput(e)

      def collide(self):             #Validation of x & y variables to prevent
player from intersecting objects
            global mapx, mapy
            i = 0
            while i < len(self.listOfObstacles):
                  if self.listOfObstacles[i].colliderect(PC):
                        PC.stop()
                  i += 1
            if PC.x <= 0:
                  if gMap[mapx-1][mapy] == None:
                        PC.stop()
                  else:
                        mapx -= 1
                        self.state = gMap[mapx][mapy]
                        PC.x = 715
            elif PC.x >= 720:
                  if gMap[mapx+1][mapy] == None:
                        PC.stop()
                  else:
                        mapx += 1
                        self.state = gMap[mapx][mapy]
                        PC.x = 5
            if PC.y <= 0:
                  if gMap[mapx][mapy-1] == None:
```

```
                        PC.stop()
                  else:
                        mapy -= 1
                        self.state = gMap[mapx][mapy]
                        PC.y = 515
            elif PC.y >= 520:
                  if gMap[mapx][mapy+1] == None:
                        PC.stop()
                  else:
                        mapy += 1
                        self.state = gMap[mapx][mapy]
                        PC.y = 5


class EndScreen(Screen):       #Class for the ending screen, doesn't need to be
finished yet
      def __init__(self):
            super().__init__()
            self.endGameText = ""
            menuButton =
Button(20,20,120,50,"Menu",[100,150,200],"georgia",40,(240,250,255))
            menuButton.setTask(self.backScreen)
            self.listOfButtons.append(menuButton)
            playAButton = Button(20,20,100,50,"Play
Again",[100,150,200],"georgia",40,(255,255,255))
            playAButton.setTask(self.startGame)
            self.listOfButtons.append(playAButton)
            self.bg = transform.scale(self.bg, (800,600))

      def draw(self,pScreen):
            pScreen.blit(self.bg, (0,0))
            for b in self.listOfButtons:
                  b.draw(pScreen)


#~---------------------------Screen Things----------------------------~
```

Introduction: The buttons and obstacles are included in this section. The button class is set up to allow a method to be given to a new button. This is called when the user clicks on the button so this section builds on the user controls and menu stories.

```
class Button(Rect):      #Class for buttons, inheriting from pygame rectangle
      def __init__(self,x,y,w,h,text,colour,fontType,size,textColour):
            super().__init__(x,y,w,h)            #Sets up pygame rectangle
            self.colour = colour
            self.font = font.SysFont(fontType, size)
            self.text = text
            self.textColour = textColour

      def draw(self,pScreen):        #Method to draw the button
            mx,my = mouse.get_pos()
            colour2 = (self.colour[0]+20, self.colour[1]+20, self.colour[2]+20)
      #Sets up two colours that are similar to the given one
```

```
            colour3 = (self.colour[0]-35, self.colour[1]-35, self.colour[2]-35)
            if self.collidepoint((mx,my)):
                        #Changes the colour of the button if the mouse is
hovering over it - helpful for the user
                  draw.rect(pScreen,colour2,self)
            else:
                  draw.rect(pScreen,tuple(self.colour),self)
            draw.rect(pScreen,colour3,self,7)
                        #Adds a border to the button
            textImage = self.font.render(self.text, True, self.textColour)
            pScreen.blit(textImage,self)

      def setTask(self,task):  #Allows a function to be given to the button
            self.task = task

      def checkClick(self, pos):
                  #Checks whether the mouse is over the button or not
            if self.collidepoint(pos):
                  self.task()


class Obstacle(Rect):          #Class for obstacles that are added to game
screens inheriting from rectangle
      def __init__(self,x,y,w,h,pimage):
            super().__init__(x,y,w,h)
            self.image = image.load(pimage)
            self.image = transform.scale(self.image, (self.w,self.h))

      def draw(self,pScreen):
                              #Method to draw obstacle
            pScreen.blit(self.image,self)

      def giveCoords(self,cods):
            self.x, self.y = cods


#~--------------------------Players---------------------------~
```

Introduction: This section contains the player class, which I didn't originally intend to include in this prototype. The player's inputs while on the map are set up here so it covers some user controls.

```
class Player(Rect):                  #Super class for any 'player' or similar
sprite inheriting form pygame rectangle
      def __init__(self,x,y,w,h,pimage,name):
            super().__init__(x,y,w,h)
            self.image = pimage
            self.name = name
            self.dx = 0
            self.dy = 0

      def update(self):        #Method to update the sprite's position
            self.move_ip(self.dx,self.dy)
```

```
	def draw(self):			#Method to draw sprite
		pScreen.blit(self.image,(self.x,self.y))

	def keyInput(self):		#Method to be overwritten
		pass

	def stop(self):
		if self.dx > 0:
			self.x -= 6
			self.dx = 0
		elif self.dx < 0:
			self.x += 6
			self.dx = 0
		if self.dy > 0:
			self.y -= 6
			self.dy = 0
		elif self.dy < 0:
			self.y += 6
			self.dy = 0

class PlayerCharacter(Player):		#Class for the player character inheriting
from Player
	def __init__(self):
		super().__init__(100,100,80,80,"playerImage.png","Bob")
		self.image = image.load(self.image)
		self.image = transform.scale(self.image, (80,80))

	def keyInput(self,e):	#Method that takes in 'event' parameter and
changes the character's movement based on key input
		if e.type == KEYDOWN:
			if e.key == K_RIGHT or e.key == ord("d"):
				self.dx = 6
			elif e.key == K_LEFT or e.key == ord("a"):
				self.dx = -6
			elif e.key == K_UP or e.key == ord("w"):
				self.dy = -6
			elif e.key == K_DOWN or e.key == ord("s"):
				self.dy = 6

		if e.type == KEYUP:
			if e.key == K_RIGHT or e.key == K_LEFT or e.key == ord("a")
or e.key == ord("d"):
				self.dx = 0
			elif e.key == K_UP or e.key == K_DOWN or e.key == ord("w") or
e.key == ord("s"):
				self.dy = 0


#~---------------------------Map---------------------------~

def setStates(pMap):			#Function that will reset all of the map screens'
states to themselves
	x = 0
```

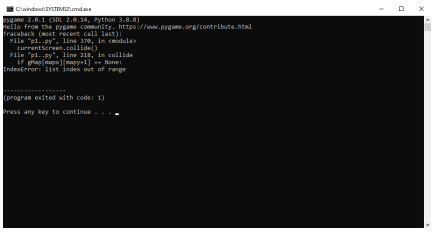
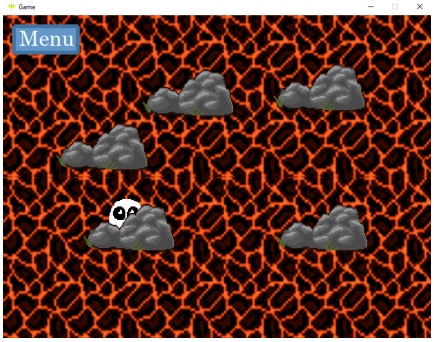
```
        while x < 4:
                y = 0
                while y < 4:
                        if pMap[x][y] != None:
                                pMap[x][y].state = pMap[x][y]
                        y += 1
                x += 1


currentScreen = MenuScreen()
PC = PlayerCharacter()
currentScreen.startGame()

#~----------------------------Game Loop----------------------------~
```

Introduction: The game loop defines what will happen every tick that the game is running. This includes setting the current screen and calling the key event method. This is the final part of the game states and user controls requirements.

```
while running == True:

        setStates(gMap)

        currentScreen.collide()
        currentScreen.keyEvent()
        PC.update()

        if currentScreen.state == None:
                running = False
        else:
                currentScreen.draw(backdrop)

        currentScreen = currentScreen.state
        display.update()
        display.flip()

        clock.tick(60)

quit()
```

Testing

Test	Findings	Screenshot
Move around game and between screens.	The screens will generally switch as intended; however, there is a bug in the screens that are further through the map array. The screen will sometimes	

	flicker between two, with the obstacles appearing and disappearing every frame. The player's movement would also become jerkier as they progress.	
Run player sprite into the edge of the screen where it shouldn't go through.	The player will successfully be stopped and unable to go out of the screen. In the later sections, the game will crash when this is attempted as it tries to access the next value in the map array, which doesn't exist.	
Run the player sprite into obstacles.	This feature also works as the player will be stopped and unable to move through obstacles. However, due to the screen flickering issue the player can edge behind a flickering obstacle by tapping the relevant movement key as they can move while it is immaterial. The screenshot cannot show the flickering as it happens every frame.	
Try 'play game' button on menu.	On the menu screen, I found that clicking anywhere other than the buttons would not yield any results, which is desired. The 'play game' button usually functions as intended as the game will return to the previous map area, although pressing it while in the later areas will result in the same screen flickering issue, just between the menu and the game screen.	
Try 'start game' button on menu.	This button is half working, in that it will reset the map and create new obstacles for each area. However, it doesn't reset the player's position - if they are in a later area, it will return there.	
Try the 'menu' button in game screen.	The menu button works entirely as intended. Even if pressed when the screen is already flickering, it will stop and simply return to the menu.	

While testing, I found that the screen would flicker when moving between later screens (e.g. gMap[3][2] to gMap[3][3]). This is not intended and I believe is caused by the fact that, for the

game screens, instead of changing the variable ‘currentScreen’ to be a new instance of the screen class, I created objects of the class beforehand and switched it between them. My fix for a previous issue caused by this was the method ‘SetStates’ - the reason this isn’t working when further through the map is because it works by iterating through the 2D array.
To fix this and other issues caused by it, in my next prototype I will need to restructure how the game screens are represented. I will need an array to indicate what type of area it is when the screen is created instead of storing the screens themselves. In order to prevent the obstacles of each screen being re-created every time a new screen is entered, I will need to create them initially, store and then recall them.
Other than this, there are a few minor bugs I will need to fix. The start game button should reset the player’s position and progress through the map. This should be simple to fix as I will only need to add some extra lines of code in the method that the button calls. To fix the game crashing when the player attempts to move beyond the game boundary in the later areas, I could change the way the check works or simply add an extra empty array on the end of the 2D array.
I will need to monitor the placement of the obstacles, especially given the varying sizes, so that the player cannot spawn inside one and thus get trapped. I did not encounter this during testing, though.

Review

Overall, my first prototype has fulfilled my initial goals. I have also incorporated features such as the player and their controls, which I didn’t plan on including. My implementation differs from my algorithm design but shares the same general structure. Some unprecedented issues, such as the screen flickering, are caused by errors in my design and therefore will be changed. This prototype is faithful to my screen designs and general gameplay, although I have not included the ‘visual blockade’ I planned to be at the edge of any room that doesn’t link to another. Having coded and tested the prototype, it will be sufficient to simply block the player from walking through. If I have extra time in a future prototype, however, I may introduce this feature.
As I have exceeded my expectations, I will try to have a larger scope for my next prototype so I don’t have to diverge from my design. It is possible that my current system will need to be altered when adding later features; however, I think it is flexible enough that the majority of this code will remain the same.

Prototype 2

Introduction

For prototype 2, I will create the combat system alongside fixing any leftover issues with the code. I will focus on making the system functional - any buttons should be responsive, encounters should begin & end appropriately and the player should have agency in their actions.

Algorithm Design

The user stories covered in this prototype will be the combat system and the user controls contained within that.

```
class Combatants(Player)
        procedure constructor(pHp, pAtk, pDef, [...])
                parent constructor([...])
                health = pHp
                strength = pAtk
                defense = pDef
                currentDef = 0
                alive = True

        function attack(target)
                currentDef = defense
                num = randint(0,4)
                dmg = strength
                if target.isDefending() == True then
                        if num == 0 then
                                health = health - target.strength
                                displayText = “Parried!”, name, “took”, target.strength, “damage!”
                                return displayText
                dmg = dmg - target.currentDef
                displayText = name, “hit”, target.name, “for”, dmg, “damage!”
                target.health = target.health - dmg
                return displayText

        function defend()
                currentDef = defense * 2
                defending = True
                displayText = name, “is defending. Defense is now: ”, currentDef
                return displayText

        function flee()
                displayText = name, “fled the fight!”
                alive = False
                return displayText

        function action()
                pass

        function checkDead()
                if health <= 0 then
                        return True
                else
                        return False

        function isDefending()
                if defending == True:
                        return True
                else
                        return False

class Enemy(Combatants)
        procedure constructor([...])
                parent constructor([...])

        procedure action()
```

```
                p = 1
                if health < maxHealth * 0.1 then
                        p = 2
                num = randint(0,p)
                if num == 0 then
                        attack()
                elif num == 1 then
                        defend()
                elif num == 2 then
                        flee()

class combatPlayer(Combatants)
        procedure constructor([...])
                parent constructor([...])

        procedure action(key)
                [...]


class combatScreen(Screen)
        procedure constructor()
                attackButton = Button([...])
                attackButton.setTask(attack())
                defendButton = Button([...])
                defendButton.setTask(defend())
                abilityButton = Button([...])
                abilityButton.setTask(ability())
                fleeButton = Button([...])
                fleeButton.setTask(flee())
                buttonList.append(attackButton)
                buttonList.append(defendButton)
                buttonList.append(abilityButton)
                buttonList.append(fleeButton)
                displayText = “”
                entityList = []
                turn = 1

        procedure keyEvent(key)
                if turn == 1 then
                        entityList[0].action(key)

        procedure addEnemies(type)
                if type == “forest” then
                        possEnemies = [Rat, Tree, Wolf]
                        a = 1
                        d = 2
                elif type == “cave” then
                        possEnemies = [Spider, Wyrm, Skeleton]
                        a = 1
                        d = 3
                elif type == “bad” then
                        possEnemies = [Dragon, Lava, Goblin]
                        a = 2
                        d = 4
                num = randint(a,d)
                while num > 0
                        x = randint(0,2)
                        entityList.append(possEnemies[x])
```

```
                num = num - 1

        function pickTarget(key)
                x = 1
                displayText = “Pick a target.”
                chosen = False
                while chosen == False
                        if key == K_DOWN then
                                x = x + 1
                                if x == entityList.length - 1 then
                                        x = 1
                        elif key == K_UP then
                                x = x - 1
                                if x == 0 then
                                        x = entityList.length - 2
                        elif key == K_RETURN then
                                chosen = True
                                return entityList[x]


        procedure attack()
                if turn == 1 then
                        pickTarget()
                        displayText = entityList[0].attack(target)
                        turn = 0

        procedure defend()
                if turn == 1 then
                        displayText = entityList[0].defend()
                        turn = 0


        procedure flee()
                if turn == 1 then
                        displayText = entityList[0].flee()
                        turn = 0


        procedure ability()
                if turn == 1 then
                        displayText = entityList[0].ability()
                        turn = 0
```

Implementation

```
from pygame import *
from random import *

init()

#~--------------------------Initial Variables--------------------------~

gMap = [[None, None, None, None, None, None],[None, "start", "forest",
"forest", "forest", None],[None, "forest", "forest", "cave", "cave",
None],[None, None, "cave", "cave", "bad", None],[None, "bad", "bad", "bad",
"bad", None],[None, None, None, None, None, None]]
obstacleList = [[None, None, None, None, None, None],[None, None, None, None,
None, None],[None, None, None, None, None, None],[None, None, None, None, None,
```

```
None],[None, None, None, None, None, None],[None, None, None, None, None,
None]]
mapx = 1
mapy = 1
width = 800
height = 600
running = True
score = 0

#~---------------------------Pygame Setup---------------------------~

backdrop = display.set_mode((width,height))
display.set_caption("COOL and EPUC game Game")
display.set_icon(image.load("drgn.png"))
clock = time.Clock()

#~---------------------------Screens---------------------------~

class Screen():          #Super class for all screens to inherit from
      global mapx, mapy, gMap, PC, CP, obstacleList
      #Global variables that are needed both inside and outside the classes

      def __init__(self, pbg):
            self.listOfButtons = []  #Attribute to contain any buttons on the
screen
            self.state = self
            self.bg = image.load(pbg).convert()  #Attribute used to switch
screens
            self.bg = transform.scale(self.bg, (800,600))

      def mouseInput(self,pos):            #Method for handling mouse input
across screens
            for b in self.listOfButtons:
                  b.checkClick(pos)
            return self.state

      def keyEvent(self):      #Empty methods to be overridden
            pass

      def draw(self,pScreen):
            pass

      def collide(self):
            pass

      def cTurn(self):
            pass

      def backScreen(self):    #Method that always returns to the main menu
            self.state = MenuScreen()

      def playGame(self):      #Method that switches the screen to the current
map screen
            self.state = GameScreen(gMap[mapx][mapy])

      def exitGame(self):
            self.state = None

      def placeObstacles(self):            #Method to set up obstacles and store
in 2D array
            tempList = []
            y = 1
```

```
            while y < 6:
                  x = 1
                  while x < 6:
                        if gMap[y][x] != None:
                              potentialCoords =
[(505,90),(370,225),(150,350),(100,200),(260,100),(510,350)]
                              numbofobs = randint(3,5)
                              n = 0
                              while n < numbofobs:
                                    coords = potentialCoords[randint(0,(5-n))]
                                    potentialCoords.remove(coords)
                                    obs = Obstacle(0,0,170,90,"rok.png")
                                    typeob = randint(0,2)
                                    if gMap[y][x] == "forest" or gMap[y][x] ==
"start":
                                          if typeob == 0:
                                                obs =
Obstacle(0,0,170,90,"rok.png")
                                          elif typeob == 1:
                                                obs =
Obstacle(0,0,100,100,"bsh.png")
                                          else:
                                                obs =
Obstacle(0,0,120,90,"logg.png")
                                    elif gMap[y][x] == "cave":
                                          if typeob == 0:
                                                obs =
Obstacle(0,0,70,140,"stalagmite.png")
                                          elif typeob == 1:
                                                obs =
Obstacle(0,0,100,100,"cob.png")
                                          else:
                                                obs =
Obstacle(0,0,70,70,"hol.png")
                                    elif gMap[y][x] == "bad":
                                          if typeob == 0:
                                                obs =
Obstacle(0,0,70,140,"stalagmite.png")
                                          elif typeob == 1:
                                                obs =
Obstacle(0,0,100,100,"cob.png")
                                          else:
                                                obs =
Obstacle(0,0,70,70,"hol.png")
                                    obs.giveCoords(coords)
                                    tempList.append(obs)
                                    n += 1
                              obstacleList[y][x] = tempList
                              tempList = []
                        x += 1
                  y += 1

      def startGame(self):      #Method to set up all of the game screens
            global mapx, mapy
            mapx = 1
            mapy = 1
            PC.x = 100
            PC.y = 100
            self.placeObstacles()

      def startAgain(self):
            self.startGame()
```

```
            self.playGame()

class MenuScreen(Screen):            #Class for menu screen
      def __init__(self):
            super().__init__("IMG_0270.png")
            exitButton = Button(20,20,100,50,"
Exit",[180,40,40],"georgia",40,(255,240,240))
            exitButton.setTask(self.exitGame)
            self.listOfButtons.append(exitButton)
            playButton = Button(280,170,240,70," Play
Game",[180,100,40],"georgia",45,(255,250,240))
            playButton.setTask(self.playGame)
            self.listOfButtons.append(playButton)
            resetButton = Button(300,270,200,55," Start
Again",[180,100,40],"georgia",35,(255,250,240))
            resetButton.setTask(self.startAgain)
            self.listOfButtons.append(resetButton)

      def keyEvent(self):      #Method for handling key input, calls mouse input
method
            for e in event.get():
                  if e.type == QUIT:
                        self.state = None
                  elif e.type == MOUSEBUTTONDOWN:
                        self.state = self.mouseInput(e.pos)
                  elif e.type == KEYDOWN:
                        if e.key == K_ESCAPE:
                              self.state = None

      def draw(self,pScreen):        #Method for drawing main menu
            pScreen.blit(self.bg, (0,0))
            for b in self.listOfButtons:
                  b.draw(pScreen)


class GameScreen(Screen):            #Class for game screens

      def __init__(self,areaType):
            self.areaType = areaType #Attribute that defines the type of the room
            if self.areaType == "forest" or self.areaType == "start":
                  pbg = "foresbg.png"
            elif self.areaType == "cave":
                  pbg = "cavebg.png"
            elif self.areaType == "bad":
                  pbg = "badbg.png"
            super().__init__(pbg)
            menuButton = Button(20,20,120,50,"
Menu",[100,150,200],"georgia",40,(240,250,255))
            menuButton.setTask(self.backScreen)
            self.listOfButtons.append(menuButton)
            self.listOfObstacles = obstacleList[mapy][mapx]   #Attribute for any
obstacles on the screen


      def draw(self,pScreen):  #Method to draw game screen, adds relevant
background
            pScreen.blit(self.bg, (0,0))
            PC.draw(pScreen)
            for o in self.listOfObstacles:
                  o.draw(pScreen)
            for b in self.listOfButtons:
                  b.draw(pScreen)
```

```
	def combatEncounter(self):
		if self.areaType == "forest":
			p = 600
		elif self.areaType == "start":
			p = 0
		elif self.areaType == "cave":
			p = 450
		elif self.areaType == "bad":
			p = 300
		x = randint(0,p)
		if x == 69:
			self.state = CombatScreen(self.areaType)

	def keyEvent(self):	#Method for key input, calls the player
character's key input method
		self.combatEncounter()
		for e in event.get():
			if e.type == QUIT:
				self.state = None
			elif e.type == MOUSEBUTTONDOWN:
				self.state = self.mouseInput(e.pos)
			elif e.type == KEYDOWN:
				if e.key == K_ESCAPE:
					self.backScreen()
			PC.keyInput(e)
		PC.update()

	def collide(self):
		global mapx, mapy
		i = 0
		while i < len(self.listOfObstacles):
			if self.listOfObstacles[i].colliderect(PC):
				PC.stop()
			i += 1
		if PC.x <= 0:
			if gMap[mapy][mapx-1] == None:
				PC.stop()
			else:
				mapx -= 1
				self.state = GameScreen(gMap[mapy][mapx])
				PC.x = 715
		elif PC.x >= 720:
			if gMap[mapy][mapx+1] == None:
				PC.stop()
			else:
				mapx += 1
				self.state = GameScreen(gMap[mapy][mapx])
				PC.x = 5
		if PC.y <= 0:
			if gMap[mapy-1][mapx] == None:
				PC.stop()
			else:
				mapy -= 1
				self.state = GameScreen(gMap[mapy][mapx])
				PC.y = 515
		elif PC.y >= 520:
			if gMap[mapy+1][mapx] == None:
				PC.stop()
			else:
				mapy += 1
				self.state = GameScreen(gMap[mapy][mapx])
```

```
                        PC.y = 5


class EndScreen(Screen):        #Class for the ending screen, doesn't need to be
finished yet
      def __init__(self):
            super().__init__()
            self.endGameText = ""
            menuButton =
Button(20,20,120,50,"Menu",[100,150,200],"georgia",40,(240,250,255))
            menuButton.setTask(self.backScreen)
            self.listOfButtons.append(menuButton)
            playAButton = Button(20,20,100,50,"Play
Again",[100,150,200],"georgia",40,(255,255,255))
            playAButton.setTask(self.startGame)
            self.listOfButtons.append(playAButton)
            self.bg = transform.scale(self.bg, (800,600))

      def draw(self,pScreen):
            pScreen.blit(self.bg, (0,0))
            for b in self.listOfButtons:
                  b.draw(pScreen)


#~---------------------------Screen Things----------------------------~

class Button(Rect):
                                    #Class for buttons, inheriting from pygame
rectangle
      def __init__(self,x,y,w,h,text,colour,fontType,size,textColour):
            super().__init__(x,y,w,h)
                              #Sets up pygame rectangle
            self.colour = colour
            self.font = font.SysFont(fontType, size)
            self.text = text
            self.textColour = textColour

      def draw(self,pScreen):  #Method to draw the button
            mx,my = mouse.get_pos()
            colour2 = (self.colour[0]+20, self.colour[1]+20, self.colour[2]+20)
      #Sets up two colours that are similar to the given one
            colour3 = (self.colour[0]-35, self.colour[1]-35, self.colour[2]-35)
            if self.collidepoint((mx,my)):       #Changes the colour of the
button if the mouse is hovering over it - helpful for the user
                  draw.rect(pScreen,colour2,self)
            else:
                  draw.rect(pScreen,tuple(self.colour),self)
            draw.rect(pScreen,colour3,self,7)    #Adds a border to the button
            textImage = self.font.render(self.text, True, self.textColour)
            pScreen.blit(textImage,self)

      def setTask(self,task):        #Allows a function to be given to the button
            self.task = task

      def checkClick(self, pos):     #Checks whether the mouse is over the button
or not
            if self.collidepoint(pos):
                  self.task()


class Obstacle(Rect):          #Class for obstacles that are added to game
screens inheriting from rectangle
```

```
	def __init__(self,x,y,w,h,pimage):
		super().__init__(x,y,w,h)
		self.image = image.load(pimage)
		self.image = transform.scale(self.image, (self.w,self.h))

	def draw(self,pScreen):		#Method to draw obstacle
		pScreen.blit(self.image,self)

	def giveCoords(self,cods):
		self.x, self.y = cods


#~---------------------------Players----------------------------~

class Player(Rect):			#Super class for any 'player' or similar sprite
inheriting form pygame rectangle
	def __init__(self,x,y,w,h,pimage,name):
		super().__init__(x,y,w,h)
		self.image = pimage
		if isinstance(pimage, str):	#Checks if there is already an
instance of the player, to avoid loading image multiple times
			self.image = image.load(self.image)
		self.name = name
		self.dx = 0
		self.dy = 0

	def update(self):		#Method to update the sprite's position
		self.move_ip(self.dx,self.dy)

	def draw(self, pScreen):
		pScreen.blit(self.image,(self.x,self.y))

	def keyInput(self):
		pass

	def stop(self):
		if self.dx > 0:
			self.x -= 6
			self.dx = 0
		elif self.dx < 0:
			self.x += 6
			self.dx = 0
		if self.dy > 0:
			self.y -= 6
			self.dy = 0
		elif self.dy < 0:
			self.y += 6
			self.dy = 0

class PlayerCharacter(Player):		#Class for the player character inheriting
from Player
	def __init__(self):
		super().__init__(100,100,80,80,"playerImage.png","Bob")
		self.image = transform.scale(self.image, (80,80))

	def keyInput(self,e):	#Method that takes in 'event' parameter and
changes the character's movement based on key input
		if e.type == KEYDOWN:
			if e.key == K_RIGHT or e.key == ord("d"):
				self.dx = 6
			elif e.key == K_LEFT or e.key == ord("a"):
				self.dx = -6
```

```
                  elif e.key == K_UP or e.key == ord("w"):
                        self.dy = -6
                  elif e.key == K_DOWN or e.key == ord("s"):
                        self.dy = 6

            if e.type == KEYUP:
                  if e.key == K_RIGHT or e.key == K_LEFT or e.key == ord("a")
or e.key == ord("d"):
                        self.dx = 0
                  elif e.key == K_UP or e.key == K_DOWN or e.key == ord("w") or
e.key == ord("s"):
                        self.dy = 0

#Combat - Prototype 2
```

Introduction: I grouped together all new combat additions for prototype 2 below. The class for the combat screen inherits from the parent screen class but has additional/overwritten methods & attributes. There are methods for adding enemies and taking turns alongside buttons which call the player's action methods when pressed. This is the bulk of the combat system and user controls for this prototype.

```
class CombatScreen(Screen):    #Class for any combat screens - to be moved to be
with other screens when combat is finished
      def __init__(self,aType):
            super().__init__("Picture1.png")
            attackButton = Button(20,410,150,60,"
Attacj",[75,130,200],"georgia",40,(255,240,240))
            attackButton.setTask(self.playAttack)
            self.listOfButtons.append(attackButton)
            defendButton = Button(20,500,150,60,"
Defebd",[75,130,200],"georgia",40,(255,250,240))
            defendButton.setTask(self.playDefend)
            self.listOfButtons.append(defendButton)
            fleeButton = Button(630,500,150,60,"
Fkee",[180,40,40],"georgia",40,(255,250,240))
            fleeButton.setTask(self.playFlee)
            self.listOfButtons.append(fleeButton)
            self.arrow = Obstacle(540,20,50,50,"arrow.png")
                  #An arrow to indicate which enemy is being targeted, inheriting
from obstacle as it shares many attributes
            self.arrowy = 20         #A value for the y coordinates of the arrow
            self.displayText = ""    #Attribute for text describing what is
happening
            self.entityList = []     #A list to contain each enemy
            self.turn = 0            #A value which stores which turn it is
            self.addEnemies(aType)
            self.l = 0   #An integer to indicate which enemy is being targeted
            self.enmyact = 0   #An integer for the enemy that is currently doing
an action
            self.timeTrack = 0       #A value to track the time passed since the
text last changed

      def keyEvent(self):
            for e in event.get():
                  if self.turn % 2 == 0:
                        if e.type == KEYDOWN and len(self.entityList) > 1:
                  #When down key is pressed, the arrow and targeted enemy should
move to the next one down
                              if e.key == K_DOWN:
                                    if self.l >= (len(self.entityList) - 1):
```

```
            #If it's already at the lowest enemy, should wrap to the first one
                                        self.l = 0
                                        self.arrowy = 20
                                  else:
                                        self.l += 1
                                        self.arrowy += 120
                            elif e.key == K_UP:
                                  #When up key pressed, arrow and targeted
enemy should go back up one
                                  if self.l <= 0:
                  #If already at highest enemy, should wrap to the last one
                                        self.l = len(self.entityList) - 1
                                        self.arrowy = 20 +
((len(self.entityList)-1) * 120)
                                  else:
                                        self.l -= 1
                                        self.arrowy -= 120
                            self.arrow.giveCoords((540,self.arrowy))
                  #Updates the arrow's coordinates
                        if e.type == MOUSEBUTTONDOWN:
                              self.state = self.mouseInput(e.pos)

      def cTurn(self):          #Method which handles the enemy's turn
            if len(self.entityList) == 0:        #If there are no enemies left,
return to game screen
                  self.state = GameScreen(gMap[mapy][mapx])
            else:
                  if self.enmyact >= len(self.entityList):    #If every enemy
has taken an action then note down time, reset enmyact and increment turn
                        self.timeTrack = time.get_ticks()
                        self.enmyact = 0
                        self.turn += 1
                  if self.entityList[self.enmyact].alive == False:
            #If an enemy has died, remove it from the list and reduce the
targeted enemy by 1
                        #Should implement way of setting arrow to be able to
move between remaining enemies, probably by changing the way keyEvent works
                        self.l -= 1
                        self.entityList.remove(self.entityList[self.enmyact])
                  elif CP.alive == False:        #If the player has died, reset
their alive attribute for next encounter and return to game screen
                        CP.alive = True
                        self.state = GameScreen(gMap[mapy][mapx])
                  else:
                        if time.get_ticks() > self.timeTrack + 2000:
                  #Only do next action if enough time has passed since last text
change
                              if self.turn % 2 == 1:         #If it's the enemy
turn, current enemy does its action and returns string to display, then time is
noted and acting enemy is incremented
                                    self.displayText =
self.entityList[self.enmyact].action(CP)
                                    self.timeTrack = time.get_ticks()
                                    self.enmyact += 1


      def addEnemies(self,foeType):  #Method to add enemies to the encounter
            ycoords = 20
            enmy = None
            Rat = Combatant(100,50,10,5,2,"Rat","rat.png")
            Tree = Combatant(60,100,10,5,2,"Tree","tree.png")
            Wolf = Combatant(100,80,10,5,2,"Wolf","wolf.png")
```

```
            if foeType == "forest":
                  possFoes = [Rat,Tree,Wolf]
                  leastAmount = 1
                  maxAmount = 2
            elif foeType == "cave":
                  possFoes = [Spider,Wyrm,Skeleton]
                  leastAmount = 1
                  maxAmount = 3
            elif foeType == "bad":
                  possFoes = [Dragon,Lava,Goblin]
                  leastAmount = 2
                  maxAmount = 4
            elif foeType == "end":
                  possFoes = [WALRUS]
                  leastAmount = 1
                  maxAmount = 1
            #elif foeType == "start":
            #      possFoes = [Rat]
            #      leastAmount = 1
            #      maxAmount = 1
            amount = randint(leastAmount,maxAmount)
            while amount > 0:
                  x = randint(0,len(possFoes)-1)
                  enmy = possFoes[x].again()           #Creates a copy of the
enemy chosen so multiple instances can be ussed
                  enmy.giveCoords(ycoords)
                  ycoords += 120
                  self.entityList.append(enmy)
                  amount -= 1

      def draw(self,pScreen):
            pScreen.blit(self.bg, (0,0))
            draw.rect(pScreen,(100,150,200),(10,380,780,210))
            self.arrow.draw(pScreen)
            for b in self.listOfButtons:
                  b.draw(pScreen)
            for e in self.entityList:
                  e.draw(pScreen)
            fnot = font.SysFont("georgia",30)
            txt = fnot.render(str(self.displayText), False, (255,255,255))
            pScreen.blit(txt,(200,400))
            if time.get_ticks() > self.timeTrack + 2000:
                  if self.turn % 2 == 0:
                        self.displayText = "Your turn!"
                        self.timeTrack = time.get_ticks()
            CP.draw(pScreen)

      def playAttack(self):    #Methods for the player's actions when a button is
pressed
                  self.displayText = CP.attack(self.entityList[self.l])
                  self.timeTrack = time.get_ticks()
                  self.turn += 1

      def playDefend(self):
                  self.displayText = CP.defend()
                  self.timeTrack = time.get_ticks()
                  self.turn += 1

      def playFlee(self):
                  self.displayText = CP.flee()
                  self.timeTrack = time.get_ticks()
                  self.turn += 1
```

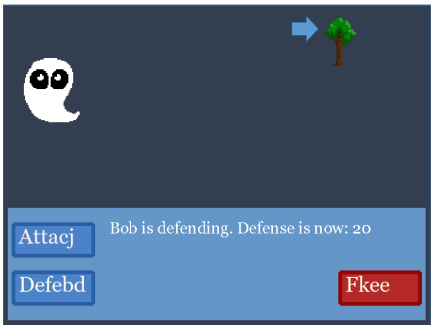
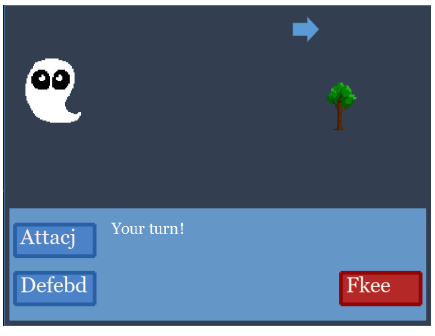
Introduction: The combatant class contains attributes and methods to represent any participant in combat. There is a method for each action that can be made, allowing the player or enemies to take part in the combat. This adds functionality to the combat system requirement.

```
class Combatant(Player):       #Class for any combatants
      def __init__(self,w,h,pHp,pAtk,pDef,pname,pimage):
            super().__init__(600,0,w,h,pimage,pname)
            self.image = transform.scale(self.image, (self.w,self.h))
            self.health = pHp        #Attributes to store maximum health, defense
and strength alongside current counterparts if necessary
            self.strength = pAtk
            self.defense = pDef
            self.currentDef = 0
            self.currentHealth = self.health
            self.alive = True
            self.num = 0       #Stores the amount of a particular combatant

      def giveCoords(self,cods):
            self.y = cods

      def attack(self,target):       #Method for attacking a target given as a
parameter
            self.currentDef = 0
            dmg = self.strength
            if target.currentDef > 0:            #If the target is defending,
there is a chance it will parry the attack. Otherwise, its defense will be taken
away from the damage given
                  num = randint(0,4)
                  if num == 0:
                        self.currentHealth = self.currentHealth -
target.strength
                        displayText = "Parried! " + self.name + " took " +
str(target.strength) + " damage!"
                        return displayText
                  else:
                        dmg -= target.currentDef
            else:
                  dmg -= target.defense
            if dmg <= 0:
                  dmg = 0
                  displayText = self.name + " tried to attack but " +
target.name + "'s defense is too high! 0 damage dealt."
            else:
                  displayText = self.name + " hit " + target.name + " for " +
str(dmg) + " damage!"
            target.currentHealth -= dmg
            if target.currentHealth <= 0:  #If the target's current health is 0
or below, set alive to false
                  target.alive = False
                  displayText = self.name +  " hit " + target.name + " for " +
str(dmg) + " damage and killed " + target.name + "!"
            return displayText

      def defend(self):        #Method for defending
            self.currentDef = self.defense * 2
            displayText = self.name + " is defending. Defense is now: " +
str(self.currentDef)
```

```
            return displayText

      def flee(self):          #Method for fleeing the fight
            displayText = self.name + " fled the fight!"
            self.alive = False
            return displayText

      def action(self,target):       #Method for enemy actions, there will only
be a chance of the enemy fleeing if it is low health
            p = 1
            if self.currentHealth < (self.health*0.2):
                  p = 2
            num = randint(0,p)
            if num == 0:
                  return self.attack(target)
            elif num == 1:
                  return self.defend()
            else:
                  return self.flee()

      def again(self):         #Method to create a copy of itself, increments num
and adds it to the copy's name
            self.num += 1
            return Combatant(self.w, self.h, self.health, self.strength,
self.defense, self.name + " " + str(self.num), self.image)

class CombatPlayer(Combatant):       #Class for the player during combat
inheriting from combatant
      def __init__(self):
            super().__init__(52,52,10,10,10,"Bob","playerImage.png")
            self.image = transform.scale(self.image, (120,120))
            self.x = 30
            self.y = 100
            self.XP = 0
            self.level = 0

#~---------------------------Setup---------------------------~

currentScreen = MenuScreen()
PC = PlayerCharacter()
CP = CombatPlayer()
currentScreen.startGame()

#~---------------------------Game Loop---------------------------~

while running == True:

      currentScreen.collide()
      currentScreen.keyEvent()
      currentScreen.cTurn()

      if currentScreen.state == None:
            running = False
      else:
            currentScreen.draw(backdrop)

      currentScreen = currentScreen.state
      display.update()
      display.flip()

      clock.tick(60)
```

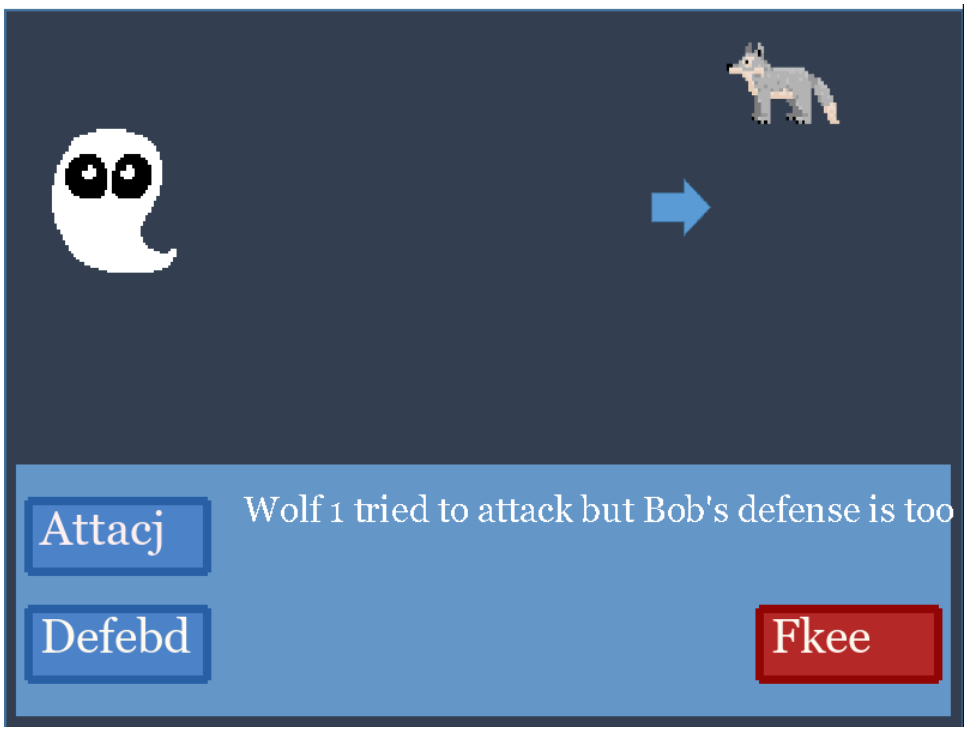
```
quit()
```

Testing

Test	Findings	Screenshot
Move around game and between screens.	There is no longer the screen flickering issue of prototype 1. The drops in frames are also no more and the player's movement is smooth.	
Test menu buttons.	The 'play game' button no longer has any issues since the flickering bug has been solved. The 'start game' button also functions as intended and will return the player to the beginning position.	
Wait for combat encounter.	Combat encounters start when a random number is rolled. Moving or remaining still in most areas should result in a combat encounter. This works, but I should introduce a more sophisticated method.	
Try combat screen buttons.	Each button on the combat screen currently functions as intended. Importantly, the player cannot press any of the buttons while it is the enemy turn so they cannot be spammed. This also prevents the player from doing multiple actions per turn.	
Try selecting an enemy.	While it is the player's turn, they can press the up/down keys to select an enemy to act upon. This feature works until one enemy has died. At this point, the player's selection will still work but the arrow supposed to indicate which enemy is being selected will become stuck in the dead enemy's position.	
Exit combat encounter.	If the player defeats all enemies, dies or flees, the combat encounter will end. This is intended, although there isn't a	

	wait to allow the final display text to show so it ends abruptly. There currently isn't a distinction between the player fleeing or dying so this should be added in prototype 3. I found a bug where, if the player is moving when the combat begins, they will continue moving when it ends.	

One bug I found was an issue with the arrow which is supposed to indicate the targeted enemy. If there are multiple enemies and one dies, the arrow would get stuck at its (now blank) position. I believe this is because the arrow's position is linked to the length of the list containing the enemies and the number of steps through the list. When there is only one enemy remaining, the player isn't able to move the arrow as their selection can't change, hence the arrow gets stuck. A way to fix this would be to tie the arrow's position to the enemies' coordinates rather than their order so it would instead snap to the remaining enemies' coordinates.



There is a visual bug involving the display text. When the display text is too long, it will extend beyond the edge of the screen. This doesn't impact the functionality of the game so I will have to decide in prototype 3 whether I want to spend time implementing text wrapping or find a simpler solution, such as shortening the text that causes this.

One more minor bug is that, while the enemies' actions are properly spaced out thanks to the system of noting down the time each takes place, after the last enemy's action the player doesn't have to wait the same amount of time in order to take their action. This means they can mostly skip the last enemy's action text and the 'Your turn!' text. However, I don't think this bug

needs fixing as - despite not being as intended - it provides the player with a choice on how long they want to read the display text. If the player knows what action they want to take and wants the combat to be as short as possible, they can do so sooner. If they would prefer to read all of the text and think about their next action, they can wait.

Apart from this, the bug involving the player retaining their movement values should be an easy fix as I can call the player's 'stop' method whenever a combat encounter is entered.

Review

In this prototype, I implemented the majority of what I planned in the algorithm design alongside fixing the leftover issues from prototype 1. The combat system is functional, although I need to add enemies for the rest of the areas other than the forest and fix some bugs found in testing. I also want to include health bars and other visual touches in the combat screen. One thing I omitted from my algorithm design was the ability action, although I don't think this would be very difficult to add in prototype 3 since the framework is present. I may add an end of combat screen including a display of anything gained from the combat. I also need to implement a levelling system for the player and create the final boss enemy, which will lead on to the end of the game.

Prototype 3

Introduction

Prototype 3 will be centred around the progression of the game. I will implement levelling and scoring systems. The level will allow the player to strengthen their character as they progress while the score will track and rate the player's progression. The end of the game will also be added including the end game screen, which will display their score. I may also add an end of combat screen to prevent it from ending as abruptly as it currently does, though this isn't priority. There will need to be a system to reset the game if the player wins or loses, since currently they are simply returned to where they were before. The bugs found in prototype 2 will be fixed and missing features such as the ability or health bars will be added.

Algorithm Design

This prototype will build upon the menu and combat system user stories.

```
class healthBar(Button)
        procedure constructor([...])
                [...]

        procedure task()
                pass

class endCombatScreen(Screen)
        procedure constructor(pText)
                displayText = pText
                contButton = Button([...])
                contButton.setTask(playGame())
```

```
                buttonList.append(contButton)


class endGameScreen(Screen)
        procedure constructor(pText)
                displayText = pText
                playAgainButton = Button([...])
                playAgainButton.setTask(playGame())
                exitButton = Button([...])
                exitButton.setTask(exit())
                menuButton = Button([...])
                menuButton.setTask(backScreen())
                buttonList.append(playAgainButton)
                buttonList.append(exitButton)
                buttonList.append(menuButton)

class combatPlayer(Combatant)
        procedure constructor([...])
                [...]
                XP = 0
                level = 1
                score = 0

        function levelUp(XPgain)
                if (XP + XPgain) > (level * 10) then
                        level = level + 1
                        health = health + 5
                        strength = strength + 5
                        defense = defense + 5
                        return True
                else
                        return False

        procedure ability()
                displayText = [...]
                for t in targets
                        t.currentHealth - strength
                return displayText
```

Implementation

(Parts that weren't changed and weren't relevant to prototype 3's algorithm omitted.)

```
from pygame import *
from random import *

init()

class CombatScreen(Screen):           #Class for combat screens
      def __init__(self,aType):
            super().__init__("Picture1.png")
            attackButton = Button(20,410,150,60,"
Attack",[75,130,200],"georgia",40,(255,240,240))
            attackButton.setTask(self.playAttack)
            self.listOfButtons.append(attackButton)
```

```
            defendButton = Button(20,500,150,60,"
Defend",[75,130,200],"georgia",40,(255,250,240))
            defendButton.setTask(self.playDefend)
            self.listOfButtons.append(defendButton)
            fleeButton = Button(630,500,150,60,"
Flee",[180,40,40],"georgia",40,(255,250,240))
            fleeButton.setTask(self.playFlee)
            self.listOfButtons.append(fleeButton)
            abButton = Button(630,410,150,60,"
Ability",[230,165,45],"georgia",40,(255,250,240))
            abButton.setTask(self.playAbility)
            self.listOfButtons.append(abButton)
            self.displayText = ""                #Text to be displayed
            self.enemyList = []                        #List to store enemies
            self.track = "Capturism.mp3"
            self.addEnemies(aType)               #Method to add enemies
            self.arrow = Obstacle(510,self.enemyList[0].y +
10,50,50,"arrow.png")    #Arrow to indicate selected enemy
            self.turn = 0                  #Attribute to track whose turn it is
            self.lastabturn = 0                  #Attribute to track the last
turn the player used their ability
            self.l = 0              #Integer variable of the targeted enemy
            self.timeTrack = 0             #Stores the time that the display
text last changed
            self.enmyact = 0        #Stores which enemy is acting
            self.XPgain = 0                #Stores the XP gained during the
combat
            CP.currentHealth = CP.health               #Resets player's health
            CP.hbar.change(CP.currentHealth,CP.health)
            mixer.music.load(self.track)
            mixer.music.play(-1, 0)

      def keyEvent(self):
            for e in event.get():
                  if self.turn % 2 == 0:
                        if e.type == KEYDOWN:
                              if e.key == K_DOWN:                    #Allows the
arrow keys to control which enemy is selected, resets to the 1st/last value if
the player exceeds the enemyList's bounds
                                    if self.l >= (len(self.enemyList) - 1):
                                          self.l = 0
                                    else:
                                          self.l += 1
                              elif e.key == K_UP:
                                    if self.l <= 0:
                                          self.l = len(self.enemyList) - 1
                                    else:
                                          self.l -= 1

self.arrow.giveCoords((510,self.enemyList[self.l].y + 10))                #Makes
the arrow follow the player's selection
                        if e.type == MOUSEBUTTONDOWN:
                              self.state = self.mouseInput(e.pos)
```

```
      def cTurn(self):                        #Method for the enemy turn
            if self.turn % 2 == 1:            #Checks if it is the enemy turn
                  for e in self.enemyList:       #Loop to remove any enemies
that have died
                        if e.alive == False:
                              if e.name == "WALRUS":                #If the
enemy was the final boss, it will go to the end game screen
                                    self.state =
EndCombatScreen("wong",self.XPgain)
                              else:
                                    self.enemyList.remove(e)
                                    self.XPgain += 3
                                    if len(self.enemyList) >= 1:
#Resets the arrow coordinates
                                          self.l = 0

self.arrow.giveCoords((510,self.enemyList[self.l].y + 10))
            if len(self.enemyList) == 0:               #Checks if the list of
enemies is empty, if so goes to the end combat screen
                  while time.get_ticks() < self.timeTrack + 700:
                        print()
                  self.state = EndCombatScreen("won",self.XPgain)
            elif CP.alive == False:        #Checks if the player has died or
fled from combat - if dead, goes to lost game screen. If fled, goes to fled
combat screen.
                  if CP.currentHealth <= 0:
                        while time.get_ticks() < self.timeTrack + 700:
                              print()
                        self.state = EndCombatScreen("died",self.XPgain)
                  else:
                        while time.get_ticks() < self.timeTrack + 700:
                              print()
                        CP.alive = True
                        self.state = EndCombatScreen("fled",self.XPgain)
            else:
                  if self.enmyact >= len(self.enemyList):                #If
the acting enemy is beyond the list length, the value is reset and the turn
incremented
                        self.timeTrack = time.get_ticks()
                        self.enmyact = 0
                        self.turn += 1
                  else:
                        if time.get_ticks() > self.timeTrack + 1500:
                              if self.turn % 2 == 1:                    #If
it's the enemy turn, the current enemy takes its action and enmyact is
incremented
                                    self.displayText =
self.enemyList[self.enmyact].action(CP)
                                    self.timeTrack = time.get_ticks()
                                    self.enmyact += 1

      def addEnemies(self,foeType):                    #Method to add enemies
            ycoords = 20            #Value to store the y coordinates for each
enemy
```

```
            enmy = None                     #Temporary object
            Rat = Combatant(100,50,10,5,2,"Rat","rat.png")
            Tree = Combatant(60,100,10,5,2,"Tree","tree.png")
            Wolf = Combatant(100,80,10,5,2,"Wolf","wolf.png")
            Spider = Combatant(100,80,8,10,2,"Spider","spoder.png")
            Skeleton = Combatant(60,110,14,8,5,"Skeleton","skele.png")
            Wyrm = Combatant(100,100,10,12,2,"Wyrm","wrrm.png")
            Dragon = Combatant(100,110,20,15,5,"Dragon","daga.png")
            Lava = Combatant(120,60,15,12,5,"Lava","lavsala.png")
            Goblin = Combatant(50,100,10,10,2,"Goblin","gobly.png")
            WALRUS = Walrus()
            leastAmount = 0                 #Variables to store the least and
most number of enemies a combat can have
            maxAmount = 0
            if foeType == "forest":
                  possFoes = [Rat,Tree,Wolf]
                  leastAmount = 1
                  maxAmount = 2
            elif foeType == "cave":
                  possFoes = [Spider,Wyrm,Skeleton]
                  leastAmount = 1
                  maxAmount = 3
                  self.XPgain = 5
            elif foeType == "bad":
                  possFoes = [Dragon,Lava,Goblin]
                  leastAmount = 2
                  maxAmount = 3
                  self.XPgain = 10
            elif foeType == "end":
                  self.track = "RISING.mp3"
                  self.enemyList.append(WALRUS)
                  amount = 0
                  self.XPgain = 30
            elif foeType == "start":
                  possFoes = [Rat]
                  leastAmount = 1
                  maxAmount = 1
            amount = randint(leastAmount,maxAmount)                 #Variable to
store the number of enemies in that combat
            while amount > 0:
                  x = randint(0,len(possFoes)-1)                    #Variable to
pick a random type of enemy from the list of possible enemies
                  enmy = possFoes[x].again()                        #Creates a
copy of the enemy type and gives it the y coordinates, then increments the y
coordinates by 120
                  enmy.giveCoords(ycoords)
                  ycoords += 120
                  self.enemyList.append(enmy)                       #Adds the
enemy to the enemy list
                  amount -= 1

      def draw(self,pScreen):
            pScreen.blit(self.bg, (0,0))
            draw.rect(pScreen,(100,150,200),(10,380,780,210))
```

```
            self.arrow.draw(pScreen)
            for b in self.listOfButtons:
                  b.draw(pScreen)
            for e in self.enemyList:
                  e.draw(pScreen)
            fnot = font.SysFont("georgia",25)
            fond = font.SysFont("georgia",30)
            if len(self.displayText) > 40:                    #If the display
text is longer than 40 characters, it will split into 2 and display one below
the other
                  longtxt =
fnot.render(self.displayText[40:len(self.displayText)], False, (255,255,255))
                  shortxt = fnot.render(self.displayText[0:40], False,
(255,255,255))
                  pScreen.blit(shortxt,(190,400))
                  pScreen.blit(longtxt,(190,430))
            else:
                  txt = fnot.render(self.displayText, False, (255,255,255))
                  pScreen.blit(txt,(190,400))
            turnxt = fond.render("Turn count: " + str(self.turn), False,
(255,255,255))
            if time.get_ticks() > self.timeTrack + 1000:
                  if self.turn % 2 == 0:
                        self.displayText = "Your turn!"
                        self.timeTrack = time.get_ticks()
            pScreen.blit(turnxt,(10,10))
            CP.draw(pScreen)

      #Button methods for each player action
      def playAttack(self):
            self.displayText = CP.attack(self.enemyList[self.l])
            self.timeTrack = time.get_ticks()
            self.turn += 1

      def playDefend(self):
            self.displayText = CP.defend()
            self.timeTrack = time.get_ticks()
            self.turn += 1

      def playFlee(self):
            self.displayText = CP.flee()
            self.timeTrack = time.get_ticks()
            self.turn += 1

      def playAbility(self):
            if ((self.turn - self.lastabturn) / 2) > 2:             #If at least
2 turns have passed since the ability was last used, lastabturn is set to the
current turn and the player uses their ability
                  self.displayText = CP.ability(self.enemyList)
                  self.timeTrack = time.get_ticks()
                  self.lastabturn = self.turn
                  self.turn += 1
            else:                     #Otherwise, text is displayed showing how
many turns are left until the ability may be used
```

```
                  self.displayText = "Ability on cooldown! Wait " + str(3 -
int((self.turn-self.lastabturn)/2)) + " more turns."
                  self.timeTrack = time.get_ticks()
```

Introduction: I added an end of combat screen which will be different depending on the outcome of the previous combat. There will also be different buttons available to the player if they won/lost the game or if they just ended a combat. This is important for the end of the game alongside showing the player their progression.

```
class EndCombatScreen(Screen):                              #Class for end
combat/game screens

      def __init__(self,pWon,pxp):
            super().__init__("Picture1.png")
            mixer.music.stop()
            self.displayText = ""
            menuButton = Button(20,20,120,50,"
Menu",[100,150,200],"georgia",40,(240,250,255))
            menuButton.setTask(self.backScreen)
            contButton = Button(275,270,250,70,"
Continue",[100,150,200],"georgia",40,(255,255,255))
            contButton.setTask(self.contGame)
            resetButton = Button(275,270,250,70," Start
Again",[100,150,200],"georgia",40,(255,250,240))
            resetButton.setTask(self.startAgain)
            exitButton = Button(20,20,100,50,"
Exit",[180,40,40],"georgia",40,(255,240,240))
            exitButton.setTask(self.exitGame)
            if pWon == "won" or pWon == "fled":              #Series of
decisions which affect the displayed text, buttons and score of the player
based on the outcome of their combat
                  if CP.levelUp(pxp) == True:
                        self.displayText = "Level up! you are now level " +
str(CP.level) + "!"
                  else:
                        self.displayText = "You gained " + str(pxp) + " XP. You
need " + str((CP.level * 10) - CP.XP) + " more to level up."
                  if pWon == "fled":
                        self.displayText = "Coward. no xp for u."
                        CP.score -= 3
                        if CP.score <= 0:
                              CP.score = 0
                  else:
                        CP.score += 5
                  self.listOfButtons.append(contButton)
                  self.listOfButtons.append(menuButton)
            else:
                  self.listOfButtons.append(resetButton)
                  self.listOfButtons.append(exitButton)
                  if pWon == "wong":
                        CP.score += 10
```

```
                        self.displayText = "You won da game so cool wow. your
score was " + str(CP.score)
                  else:
                        self.displayText = "You lose dumbass"


      def keyEvent(self):
            for e in event.get():
                  if e.type == QUIT:
                        self.state = None
                  elif e.type == MOUSEBUTTONDOWN:
                        self.state = self.mouseInput(e.pos)

      def draw(self,pScreen):
            pScreen.blit(self.bg, (0,0))
            for b in self.listOfButtons:
                  b.draw(pScreen)
            fnot = font.SysFont("georgia",27)
            txt = fnot.render(self.displayText, False, (255,255,255))
            pScreen.blit(txt,(50,400))

      def contGame(self):
            self.state =
GameScreen(gMap[mapy][mapx],time.get_ticks()-1000,True)

      def startAgain(self):
            self.startGame()
            self.state = GameScreen(gMap[mapy][mapx],0,True)

#~---------------------------Screen Things----------------------------~

class Button(Rect):
                                    #Class for buttons, inheriting from pygame
rectangle
      def __init__(self,x,y,w,h,text,colour,fontType,size,textColour):
            super().__init__(x,y,w,h)
                              #Sets up pygame rectangle
            self.colour = colour
            self.font = font.SysFont(fontType, size)
            self.text = text
            self.textColour = textColour

      def draw(self,pScreen):
                              #Method to draw the button
            mx,my = mouse.get_pos()
            colour2 = (self.colour[0]+20, self.colour[1]+20, self.colour[2]+20)
      #Sets up two colours that are similar to the given one
            colour3 = (self.colour[0]-35, self.colour[1]-35, self.colour[2]-35)
            if self.collidepoint((mx,my)):
                        #Changes the colour of the button if the mouse is
hovering over it - helpful for the user
                  draw.rect(pScreen,colour2,self)
            else:
                  draw.rect(pScreen,tuple(self.colour),self)
```

```
            draw.rect(pScreen,colour3,self,7)
                        #Adds a border to the button
            textImage = self.font.render(self.text, True, self.textColour)
            pScreen.blit(textImage,self)

      def setTask(self,task):
                              #Allows a function to be given to the button
            self.task = task

      def checkClick(self, pos):
                              #Checks whether the mouse is over the button or
not
            if self.collidepoint(pos):
                  self.task()
```

Introduction: There is a new class for health bars during combat, inheriting from the button class since they share similar attributes. The health bar adds to the combat system by providing the player with useful information.

```
class HealthBar(Button):        #Class for the health bars
      def __init__(self,text,maxWidth):

super().__init__(0,0,maxWidth,32,text,[60,200,70],"georgia",30,(255,255,255))
            self.mwidth = maxWidth

      def draw(self,pScreen):
            draw.rect(pScreen,tuple(self.colour),self)
            textImage = self.font.render(self.text, True, self.textColour)
            pScreen.blit(textImage,self)

draw.rect(pScreen,(200,40,30),(self.x+self.width,self.y,self.mwidth-self.width,
32),False)         #Draws a red rectangle indicating damage
            draw.rect(pScreen,(10,20,60),(self.x,self.y,self.mwidth,32),3)
                                          #Draws a green rectangle beneath

      def change(self,h,maxh):
            self.width = self.mwidth * (h/maxh)        #Changes the width of
the red rectangle to be proportional to damage taken
            self.text = str(h)

      def giveCoords(self,cods):
            self.x,self.y = cods

      def task(self):
            pass

#~---------------------------Players----------------------------~

class Player(Rect):
                                    #Super class for any 'player' or similar
sprite inheriting form pygame rectangle
      def __init__(self,x,y,w,h,pimage,name):
```

```
            super().__init__(x,y,w,h)
            self.image = pimage
            if isinstance(pimage, str):
                  self.image = image.load(self.image)
            self.name = name
            self.dx = 0
            self.dy = 0

      def update(self):
                              #Method to update the sprite's position
            self.move_ip(self.dx,self.dy)

      def draw(self, pScreen):
            pScreen.blit(self.image,(self.x,self.y))

      def keyInput(self):
            pass

      def stop(self):
            if self.dx > 0:
                  self.x -= 6
                  self.dx = 0
            elif self.dx < 0:
                  self.x += 6
                  self.dx = 0
            if self.dy > 0:
                  self.y -= 6
                  self.dy = 0
            elif self.dy < 0:
                  self.y += 6
                  self.dy = 0

class Combatant(Player):
      def __init__(self,w,h,pHp,pAtk,pDef,pname,pimage):
            super().__init__(600,0,w,h,pimage,pname)
            self.image = transform.scale(self.image, (self.w,self.h))
            self.health = pHp
            self.strength = pAtk
            self.defense = pDef
            self.currentDef = 0
            self.currentHealth = self.health
            self.alive = True
            self.num = 0
            self.hbar = HealthBar(str(self.currentHealth),self.health * 10)
      #Creates a health bar for the object

      def giveCoords(self,cods):
            self.y = cods
            self.hbar.giveCoords((self.x - 30,self.y + self.height))

      def draw(self,pScreen):
            self.hbar.change(self.currentHealth,self.health)
            self.hbar.draw(pScreen)
            pScreen.blit(self.image,(self.x,self.y))
```

```
	def attack(self,target):
		self.currentDef = 0
		dmg = self.strength
		if target.currentDef > 0:
			num = randint(0,4)
			if num == 0:
				self.currentHealth = self.currentHealth -
target.strength
				displayText = "Parried! " + self.name + " took " +
str(target.strength) + " damage!"
				if self.currentHealth <= 0:
					self.alive = False
					displayText = "Parried! " + self.name + " took "
+ str(target.strength) + " damage and was   killed by " + target.name + "!"
				return displayText
			else:
				dmg -= target.currentDef
		else:
			dmg -= target.defense
		if dmg <= 0:
			dmg = 0
			displayText = self.name + " tried to attack but " +
target.name + "'s defense is too high! 0 damage dealt."
		else:
			displayText = self.name + " hit " + target.name + " for " +
str(dmg) + " damage!"
		target.currentHealth -= dmg
		if target.currentHealth <= 0:
			target.alive = False
			displayText = self.name +  " hit " + target.name + " for " +
str(dmg) + " damage and killed   " + target.name + "!"
		return displayText

	def defend(self):
		self.currentDef = self.defense * 2
		if self.currentHealth < self.health:
			self.currentHealth += 1
		displayText = self.name + " is defending. Defense is now: " +
str(self.currentDef)
		return displayText

	def flee(self):
		displayText = self.name + " fled the fight!"
		self.alive = False
		return displayText

	def action(self,target):
		p = 1
		if self.currentHealth < (self.health*0.2):
			p = 2
		num = randint(0,p)
		if num == 0:
			return self.attack(target)
```

```
            elif num == 1:
                  return self.defend()
            else:
                  return self.flee()

      def again(self):
            self.num += 1
            return Combatant(self.w, self.h, self.health, self.strength,
self.defense, self.name + " " + str(self.num), self.image)
```

Introduction: I added attributes and methods to the combat player class to enable levelling up and progressing through the game. I also added the ability method which can be used during combat so this section adds to both the progression and combat system.

```
class CombatPlayer(Combatant):
      def __init__(self):
            super().__init__(120,120,15,5,3,"Bob","playerImage.png")
            self.x = 30
            self.y = 100
            self.XP = 0
            self.level = 1
            self.score = 0
            self.hbar.giveCoords((self.x - 10,self.y + 130))

      def levelUp(self,XPgain):            #Method to level up the player with
a gained XP parameter, updates their stats and returns true/false depending on
whether they levelled up or not
            self.XP += XPgain
            if self.XP >= self.level * 10:
                  self.level += 1
                  self.health += 5
                  self.strength += 5
                  self.defense += 5
                  self.XP -= self.level * 10
                  self.score += 10
                  return True
            else:
                  return False

      def ability(self,targets):                  #Method for the player's
ability, damages all enemies and reports any killed enemies
            displayText = self.name + " used their ability! All enemies took "
+ str(self.strength // 2) + " damage!"
            aDead = False
            for t in targets:
                  t.currentHealth -= (self.strength // 2)
                  if t.currentHealth <= 0:
                        if aDead == True:
                              displayText = displayText + " and " + t.name
                        else:
                              displayText = self.name + " used their ability
and killed " + t.name
```

```
                        t.alive = False
                        aDead = True
            return displayText
```

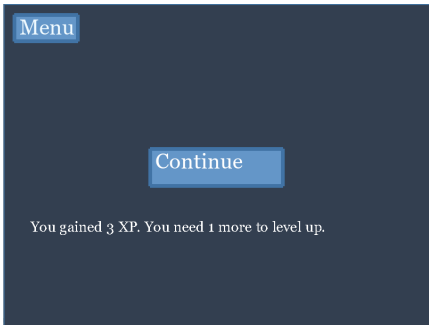
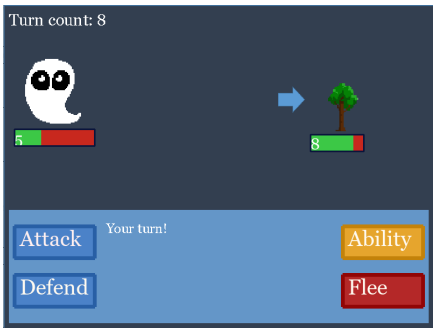
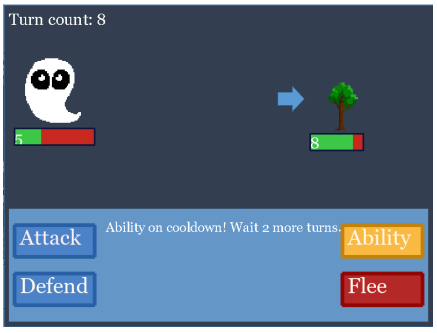
Introduction: This new class represents the final boss of the game, the Walrus. It inherits from the combatant class but has different versions of the actions as well as an additional ability. Defeating this boss is the clause for winning the game so the class is essential for the progression, alongside adding new combat system features.

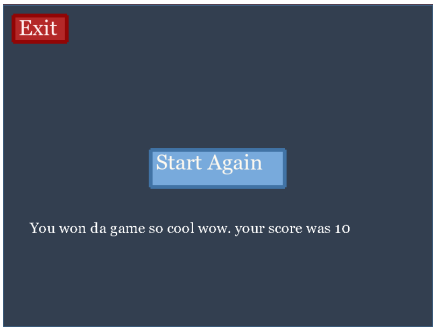
```
class Walrus(Combatant):        #Class for the final boss
      def __init__(self):
            super().__init__(300,180,20,5,2,"WALRUS","WALRUS.png")
            self.x = 300
            self.y = 100
            self.hbar.giveCoords((self.x - 10,self.y + 200))

      #Overrided methods from Combatant class which are stronger versions of
the methods

      def defend(self):
            self.currentHealth += 3
            if self.currentHealth > self.health:
                  self.currentHealth = self.health
            self.currentDef = 2 * self.defense
            displayText = self.name + " is defending. Defense is now: " +
str(self.currentDef)
            return displayText

      def attack(self,target):
            self.currentDef = 0
            dmg = self.strength
            if target.currentDef > 0:
                  num = randint(0,6)
                  if num == 0:
                        self.currentHealth = self.currentHealth -
target.strength
                        displayText = "Parried! " + self.name + " took " +
str(target.strength) + " damage!"
                        if self.currentHealth <= 0:
                              self.alive = False
                              displayText = "Parried! " + self.name + " took "
+ str(target.strength) + " damage and was   killed by " + target.name + "!"
                        return displayText
                  else:
                        dmg -= target.currentDef // 2
            else:
                  dmg -= target.defense // 2
            if dmg <= 0:
                  dmg = 0
                  displayText = self.name + " tried to attack but " +
target.name + "'s defense is too high! 0 damage dealt."
            else:
```

```
                  displayText = self.name + " smacked " + target.name + " for "
+ str(dmg) + " damage!"
            target.currentHealth -= dmg
            if target.currentHealth <= 0:
                  target.alive = False
                  displayText = self.name +  " smashed " + target.name + " for
" + str(dmg) + " damage and killed   " + target.name + "! You die to the
WALRUS."
            return displayText

      def ability(self):       #Method for the walrus' ability, which restores
it to full heatlth
            self.currentHealth = self.health
            displayText = self.name + " used its ability and restored itself to
full health! F%$*!"
            return displayText

      def action(self,target):       #Overriden method for action, has a
relatively low chance of using its ability
            p = 15
            num = randint(0,p)
            if num < 7:
                  return self.attack(target)
            elif num > 7:
                  return self.defend()
            else:
                  return self.ability()

#~---------------------------Setup---------------------------~

currentScreen = MenuScreen()
PC = PlayerCharacter()
CP = CombatPlayer()
currentScreen.startGame()

#~---------------------------Game Loop---------------------------~

while running == True:

      currentScreen.collide()
      currentScreen.keyEvent()
      currentScreen.cTurn()

      if currentScreen.state == None:
            running = False
      else:
            currentScreen.draw(backdrop)
      currentScreen = currentScreen.state

      display.update()
      display.flip()

      clock.tick(60)
```

```
quit()
```

Testing

Test	Findings	Screenshot
End combat in different ways (defeat all enemies, flee, die).	The combat will end when intended and the end combat screen will appear with the correct text. It will also display the amount of XP gained by the combat. The text for the player's final action, however, won't appear. This is because the screen switches to the end of combat screen before drawing the display text.	
Try targeting enemies and observe the arrow's movement.	The arrow will respond to the arrow keys correctly. When an enemy dies, instead of becoming stranded the arrow will snap to the next player's position as intended.	
Test the ability button (cooldown, killing multiple enemies).	The ability button should damage all enemies by half the player's strength but have a cooldown so it cannot be abused. The ability will correctly damage/kill affected enemies. However, there is a quirk where the cooldown will last 3 turns rather than 2 in the beginning of combat, then 2 for the rest. This is inconsistent but I don't think it requires fixing, since the ability should be less available at the start of combat anyway.	

Try different combat encounters and check whether the text wrapping is suitable.	The text wrapping works for most of the display text. However, there are some strings where the text will get cut off partway through rather than at the end of a word. This doesn't impact gameplay and the text is still readable.	
Finish the game in all available ways.	Each 'ending' works as intended - winning and losing results in a screen with the relevant text. The player also only has the option to reset the game, not continue, as they have finished it.	

While testing, I found that the previous bug from prototype 2 where the player would resume moving after combat ended without a key being pressed to initiate it. My fix for this had been calling the method stop() before the game switches to the combat screen which sets all the movement values of the player to 0. In theory, this means that the player would be stationary and their movement values could not be changed until the combat ends, at which point the user could begin to control their movement once more. The bug only occurs sometimes and I have not found a way of reliably reproducing it so I cannot determine where the issue is. However, the bug is immediately fixed as soon as a movement key is pressed or the player collides with anything so it is not game-breaking nor inconvenient for the player.

Other than that, there were minor bugs such as my text wrapping implementation not entirely solving the issue and the inconsistent initial cooldown of the player's ability. Neither of these have a significant impact on the gameplay so would be time inefficient to fix. The extra turn before the ability can first be used may even improve the game's balance since it will force the player to use their more standard actions, giving the enemies a chance to have an advantage.

Review

Overall, I have achieved what I wanted to accomplish from prototype 3. There are functioning scoring, levelling and end game systems and most of the bugs from prototype 2 are fixed. I added a new class to represent the final boss of the game, which has its own actions and ability. This was beyond what I intended to include but is a worthwhile addition since it ties in with the game progression - the boss being strong is motivation to level up and defeating the boss is the win condition of the game.

There are some aspects which aren't as polished as I wanted them to be, such as the end combat screen being barren and simplistic. The combat isn't completely balanced either as my

focus was making it functional and being able to test it. There is also the carried over movement bug from prototype 2 and the smaller issues during combat. Despite this, I think prototype 3 was successful as the game is playable and I included all of my desired features.

Evaluation

Testing

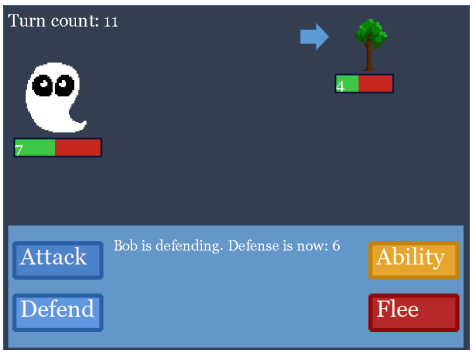
Test	Test Data	Expected Result	Actual Result
Combat System			
1. Try each combat button.	Clicking on each button on the combat screen - Attack, Defend, Ability and Flee.	Each button should call its respective method and have the desired effect. The user should not be able to press the buttons in the enemy turn.	The buttons are responsive and work properly. The flee button doesn't cause the display text to show the correct message but the end of combat screen compensates for this. During the enemy turn, the buttons will still change colour if the mouse is hovered over them but won't be clickable.
2. Attempt to reach all conclusions of combat.	Defeating all enemies, fleeing and dying.	The correct end of combat screen should appear for each possible resolution of combat.	Each combat resolution does result in the correct end of combat screen.
3. Test combat controls.	Arrow key controls.	The up/down arrow keys should allow the player to select an enemy. A visual arrow should track which enemy is selected.	This works as intended. The user cannot select an enemy during the enemy turn, which would be harmless but also unnecessary so is not an issue.
4. Try to enter combat in all regions.	Entering combat, correct enemies.	Each area should have different types and amounts of enemies. There should also be higher likelihoods of combat encounters in some areas.	Although this is difficult to test due to relying on randomness, the enemy types are correctly chosen and I did not encounter any anomalies.
Map			
5. Attempt to enter/exit areas.	Switching to a different area.	When the player reaches the edge of the screen and there is an adjacent area, they should be able to enter it.	This works as intended. There are no fps drops in the later areas which was a bug in prototype 1.
6. Try to go	Moving into the	The player should not be	The player is stopped at the edge of

outside the allowed areas.	edge of a screen without an adjacent area.	able to escape the defined areas or the game would crash.	any area without an adjacent area if they attempt to traverse it.
Game States			
7. Try any controls from a previous screen.	Controls from a previous screen on a new one.	If the game has switched states, any controls from the previous state should no longer have an effect if they are not carried over.	This appears not to be an issue, however the problem of the player moving without a key press after combat finishes could be a result of this.
8. Try to enter a combat encounter.	Combat encounter beginning.	The game should eventually switch to combat while the player remains on the map.	Combat encounters do occur and the game successfully switches states. However, they occur at random so can be difficult to test.
User Controls			
9. Check if the player will correctly collide with obstacles.	Walking into obstacles.	The player should not be able to intersect with any obstacles.	When the player reaches an obstacle, they are stopped and cannot pass through it.
10. Investigate whether clicking can yield incorrect results.	Click on parts of the screen that do not have a button, click on button edges.	Buttons should only have an effect if the player clicks directly on them.	Clicking does not have an effect unless on a button so the game is robust. The button borders will not cause the button's method to be called.
11. Try all movement controls.	Arrow keys and WASD should allow the player to move in a desired direction.	The player should be able to move in any direction at a constant speed.	The player is capable of moving in their desired direction using the appropriate keys. However, their velocity is stopped when they lift the key of that direction so switching directions is not smooth.
Menu			
12. Test each button.	Clicking on each button - Exit, Play Game and Start Again.	Each button should call its respective method when clicked.	The buttons function as intended.
Progression			
13. Try to reach the end of the game.	Defeat the final boss & attempt to win without defeating it.	When the final boss is defeated, the game should end. The player shouldn't be able to win otherwise.	The correct end of combat screen is displayed when the boss is defeated. Any other conclusions from the final combat do not result in a win.
14. Attempt to	Win combat	After each combat	The player does gain XP and score

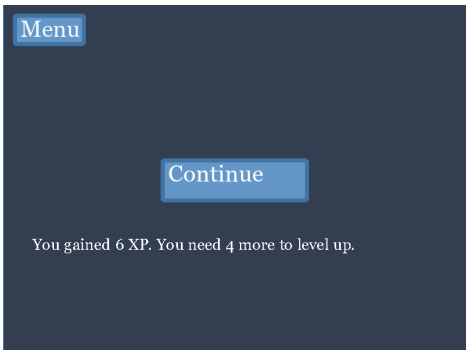
gain XP/score.	encounters.	encounter, the player should gain a given amount of XP and score.	at the end of combat, though only XP is displayed after each combat and score at the end of the game.

Screenshots:

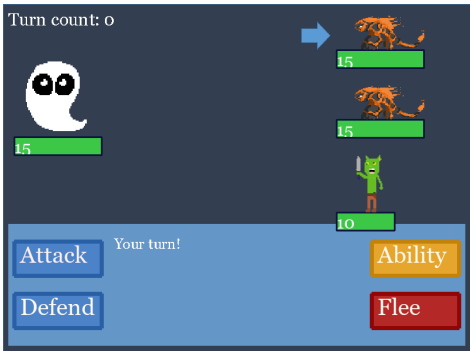
Test 1.



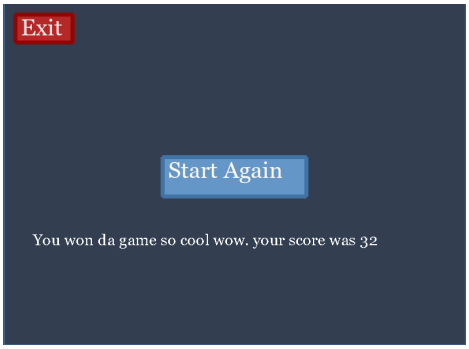
Test 2 & 14.



Test 4.



Test 12.



Evaluation of Success Criteria

Features:

Functional Map:

The map which the player can traverse and explore (tests 5-6) is a central feature of my game. There are several areas, all of which can be entered and moved around in at will. When the edge of the screen is reached, the player will either be stopped or enter the next area if there is one.

Obstacles on the map cannot be moved through and won't obstruct the player's progression through the game. They are placed with enough distance from any edge of the screen that the player can avoid them and also cannot be teleported inside one when they enter a new area.

There are no issues with the map while playing the game, but its implementation is not ideal. In order to circumvent the inability to create a new copy of an object in python, I created an extra global 2D array to store the obstacles for each area. However, later in development I ran into

the same problem while trying to make duplicate enemies in combat. This time, I found a more elegant solution of adding a method which would return all of that object's attributes and creating a new object using those. If I applied this to the map, I would be able to eliminate the need for a global variable and 2D array traversals.

Despite this, the current map fully meets the success criteria, though if it were larger it may cause issues.

Combat System:

While testing the combat system (tests 1-4, 8), I found that all buttons and controls work as desired. The player can use the on-screen buttons to take actions and can select an enemy with the arrow keys to suffer the results. There is a quirk with the ability button - it is supposed to have a cooldown of 2 turns but will have a cooldown of 3 on the 1st turn.

Combat will also correctly start in different areas of the map with the relevant types/amounts of enemies. However, the combat is programmed to begin mostly at random, meaning that it could take too much or too little time to occur. This isn't an ideal system and could negatively impact the player's experience. If the player reaches any resolution of the combat, such as winning, it will finish and the end of combat screen will display the conclusion.

The player and enemies take turns within the combat and neither can act while it is the other's turn. There is a misleading visual effect where the buttons will continue to light up while moused over when it is the enemy turn, implying that they can be clicked. This is just a facet of the button's draw method and clicking will not yield any results when it shouldn't. I used a time tracking system to ensure that the displayed text has time to be understood alongside pacing the turns.

Overall, my combat system success criteria has been met. There are only very minor bugs, most visual, which don't detract from its functionality.

Working Menus:

The menus (tests 10, 12) are one of the simpler but still essential features. From testing I found that every button, including those in other screens, has the correct function when clicked. The main menu can be reached from the map and end combat screens but not during combat. This is ideal since the player should be able to pause the game at most times; however, going to the menu during combat would allow them to start a new game and cheat if they were going to lose the combat.

Each menu can be left as desired by the player via a button leading to another screen. The end of combat screens differ based on the outcome of combat - the player should be able to return to their previous position unless they won or lost the game.

Due to how the combat screen switches to the end of combat screen, the display text for an action taken by the player which ends the combat (such as attacking and killing the last enemy

to win the combat) will not be shown. Instead, the area where it should be displayed remains empty until the screen switches. Once the end of combat screen is reached, it becomes clear what the outcome was. This means that the player should be able to extrapolate what happened if they were confused by the lack of text.

As the menus and buttons work as intended and each screen appears when desired so I have met the success criteria.

Player Movement:

Within the map, the player can use the arrow keys or WASD keys to move in any direction they desire (tests 9, 11). When colliding with an obstacle or the edge of the screen (with no area beyond it), the sprite's movement is stopped so it cannot go beyond it. There is no way for the player to reach areas they shouldn't while playing.

The movement isn't perfect, however, as the player's movement in all directions is stopped upon collision. This means that the player will have to press a key again to move if they collide with an obstacle, even if they were still holding down a key that was not sending them in its direction. For example, the player could be holding down both left and up keys to move diagonally and collide with the right side of an obstacle. Instead of stopping their horizontal motion and allowing them to continue moving up, the player will be stopped altogether and will have to release and press the key again. This can make movement tedious for the player.

There is also a bug I found during design testing which I could not fix - after combat ends and the player returns to the map screen, they will sometimes be moving without any input from the player. I could not diagnose the source of the problem since it only happens sometimes. I called a function to stop the player's movement before a combat encounter so it should not be carried over from pre-combat. This bug is not game-breaking as it is solved as soon as the player presses a movement key and it cannot bring the player out of bounds. It is still an issue with the game and could confuse or detract from a player's experience.

Since there are multiple issues with the movement that are not cosmetic, I have partially met the success criteria. I could solve the collision issue by implementing a check to see how the player collided with the obstacle and setting the relevant velocity to 0 rather than setting both velocities to 0. This would allow the player to slide along the edge of an obstacle without releasing the movement key. The other bug may be more difficult, but I believe it would be fixed by calling the player's stop method before the end combat screen switches to the map screen, rather than before combat.

Robustness:

Confines:

Via testing (test 6) I found that the player cannot escape the intended bounds of the game. Obstacles cannot be moved through and the player cannot reach areas which aren't in the map.

If they could escape the intended areas, the game would crash as it would try to draw a screen which does not exist.

Therefore, the game is robust when it comes to its bounds.

Progression:

Each area type can only be reached from the previous intended type (i.e. lava areas can be reached from cave areas but not directly from forest areas). There is no way of skipping an area to circumvent the progression of the game.
Defeating the end boss is the only way to win the game, as shown by test 13. The end combat screen for when the player wins or loses the game does not allow the player to return to their game without starting a new one so they cannot win the game multiple times in one game.

As such, there is no way to cheat the progression of the game.

No crashes:

Test 7 demonstrates that incorrect inputs will not break the game. Inputs from a previous screen won't have an undesired effect on a different screen. This means that my screen switching system works and cannot be broken or abused.
I also found that testing edge cases with mouse inputs (test 10) would not yield unexpected results. Buttons' borders won't result in their method being called while the rest of the button will. Clicking elsewhere or near a button also will not incorrectly call its method.

This means that my game can't be crashed with unexpected inputs.

Final Feedback Meeting

Meeting Agenda:

1. Is it simple/easy to start playing?
2. Can they learn combat easily?
3. Is the map easy to traverse?
4. Are the controls instinctive/natural?
5. Does the game provide enough visual feedback to determine what is happening?

User Comments:

1. The game is very easy to start as the play game button is immediately accessible. However, the lack of keyboard controls in the main menu makes it slightly awkward to navigate, especially since other controls are keyboard based.

2. The basic controls of combat (such as attack and defend) are intuitive. There is no indication of what the ‘ability’ does nor the fact that you can select an enemy using arrow keys so these aspects of the combat were harder to learn. The buttons continue to light up when moused over during the enemy turn despite not being clickable, which is counter-intuitive. The lack of a final displayed text at the end of combat can also make it confusing and seem broken since the game still pauses for the duration it should be displayed. This is particularly confusing if the player dies to a ‘parry’, since there is no indication of what happens - it seems like you just randomly lose the combat after attacking.

3. It can be tedious to navigate the map due to the obstacles, particularly given that your movement is stopped altogether when you collide with one. There is also no indication of whether you can traverse through the edge of the screen to another area apart from colliding with it. Other than that, there is enough space to go around any obstacles and reach other areas; each obstacle is clear visually also. Each area type is clearly telegraphed thanks to the change in background/obstacles so keeping track of your progress is not difficult.

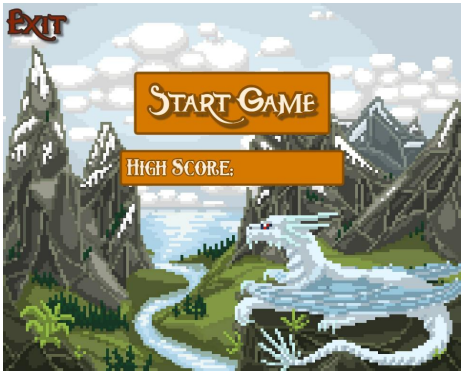
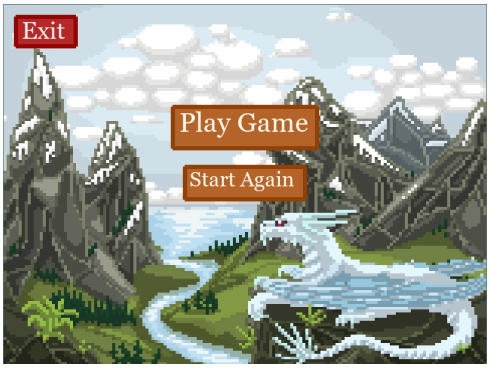
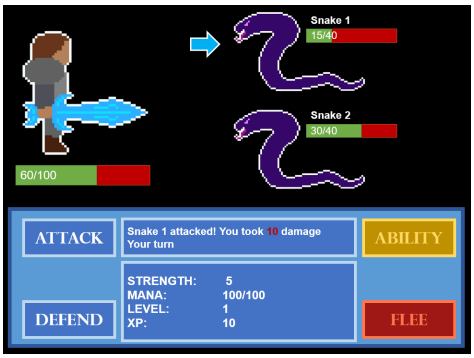
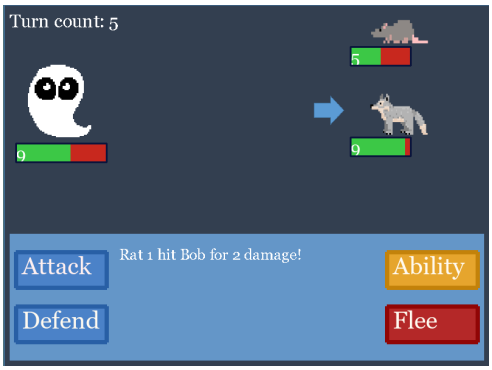
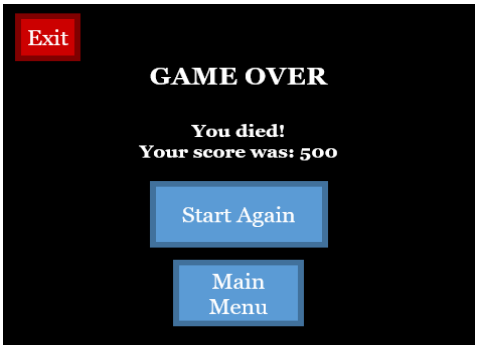
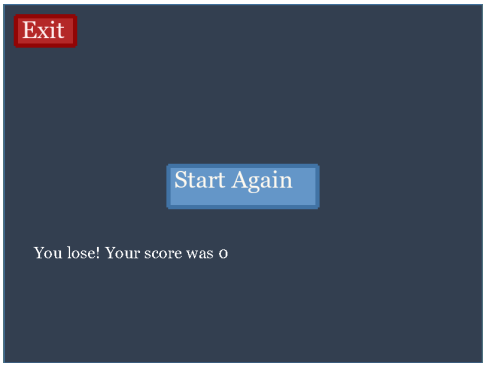
4. The controls while moving around the map are very simple to learn since the arrow keys are natural to use. Clicking buttons is also intuitive despite the lack of keyboard controls on the menu.

5. Despite the lack of animations, it is easy to track the player on the screen due to the sprite’s contrast with the background. Combat encounters are mostly easy to follow as well due to the arrow, health bars and text describing the last action. It would be better if the last few actions were displayed as the text can be missed if you aren’t paying attention.

Usability:

Screen Comparisons:

Mock Up Design	**Actual Design**	**Comments**

		I used the same colour scheme and background image as I thought they were appropriate. However, I omitted the high score section and replaced it with a start again button since that will be more helpful to the player.
		For the final screen, I made the player sprite a ghost to reduce the impact of the lack of animations (ghosts float). I also ended up not including the barrier at the edge of the screen to indicate it could not be passed, mostly due to prioritising other parts of the program.
		Again, I used a similar colour scheme to my mock up. There is not as much information displayed in the actual screen as adding text in pygame can take more time than it is worth. I did add a turn counter to help keep track of the length of combat as well as the ability cooldown.
		This screen was the last one I coded so I had less time to refine it, hence the difference in text placement. The main menu button was omitted as it would allow the player to reach the ‘play game’ button which would return them to where they were before winning/losing.

Accessible Start Game:

Starting the game only requires either the ‘play game’ or ‘start game’ buttons to be clicked on the main menu, which is the first screen to appear when the game is loaded. Therefore it is very easy to begin playing. The user feedback noted that the menu lacked keyboard controls, which could be jarring since there are keyboard controls in other screens.

This criteria has been met since the game can be started within 1-2 clicks from the main menu.

Consistent Combat Screen:

The combat screen is the same for all encounters - only the enemies vary, though they have the same layout. However, user feedback pointed out that there is no explanation of what the ‘ability’ button does, meaning that the player must try it and deduce its effect by themselves. This would make combat more difficult to learn as a player could have misconceptions about the ability’s effect or misuse it. Additionally, there is no guidance to selecting an enemy with the arrow keys. In other menus, there are no keyboard controls only mouse. This sets the precedent that there won’t be keyboard controls in the combat screen and nothing suggests otherwise. This would also hamper the player’s ability to learn and interact in combat, since selecting an enemy allows for more strategy.

The user also mentioned bugs which were explained earlier - the buttons lighting up when the cursor is over them during the enemy turn and the missing final text when the player finishes combat. These detract from the usability in the combat screen since the player is receiving misleading feedback.

The success criteria for this has been partially met since the combat screen is consistent across all encounters. Despite this, the unexplained systems and unhelpful visual errors make the combat harder to interact with than it should be.

Comprehensible Map:

The map is visually well designed as it is easy to recognise obstacles or areas within it, which is supported by the user feedback. Although there are no animations, the player sprite is distinctive enough to keep track of, as are the obstacles.

I created different obstacles for the forest and cave areas but used the cave obstacles for the lava areas. This is due to time restraints and the fact that it would not add any complexity to the code itself. Despite this, the difference in background is enough to clearly indicate which area is which.

Altogether, the map meets my success criteria as it was easy for the user to understand and everything is clearly represented.

Natural Controls:

I chose arrow keys and WASD controls for movement as they are commonly used keys. The user feedback confirmed that these are suitable controls which can be understood quickly without need of explanation. Clicking buttons is equally simple to grasp, especially since they light up when hovered over.

One point raised by the user is that it can be awkward to move the player due to the collision with obstacles. As discussed earlier, the obstacles stop movement in all directions when touched, resulting in movement which is not smooth. This can make the controls feel less responsive.

The success criteria for this has been met as the controls are quick to use and understand. The movement can be clunky but the controls are effective.

Visual Feedback:

The user feedback generally suggests that the visual feedback of the game is sufficient to understand what is happening within it.

I didn't include an indication of which sides of the screen can be passed through, which does reduce the readability of the map. It means that the user must test each side and may have less direction in the game or find it more difficult to understand where they are in the map. However, all other elements in the map are represented well enough.

As mentioned previously, the combat screen has some visual elements which are unhelpful for the user. Also, only one line of text is displayed at once. If multiple were displayed showing the last few actions taken or the player's stats, it would be easier to strategize and keep track of the combat.

This criteria has been partially met. There are a few missing features which would be helpful for the user as well as some counter-productive aspects in the combat screen. These could be improved on by implementing a system to add obstacles to the sides of screens that cannot be passed through, fixing the visual bugs in combat and adding more descriptive text.

Evaluation of Maintainability

When it comes to maintainability, the map is an issue. To solve the problem of not being able to create a new copy of an object in python, I created an extra global variable which is a 2D array to store lists of each obstacle in each area, to be called when that area is entered. This also required a method to set up the obstacles with a 2D array traversal. The use of global variables is not a modular approach and 2D array traversals are inefficient. Therefore, the implementation of the map is not very maintainable.

Besides this, I used other global variables which could be encapsulated to make the code more modular. Both the player character in the map and the player during combat are global objects of their respective class. This means they can be called anywhere within the code despite only

being needed within certain screens. If I made the variables local, it would be easier to manage them in future changes as well as keeping them contained within the screens where they are needed.

Lastly, my code is relatively long at about 800 lines. This can make it difficult to isolate specific parts of it in order to debug or add to a specific feature. It also means that later segments of code can be reliant on earlier ones, making it more difficult to alter the earlier code without breaking the program. Trying to follow the code's process can require scrolling back and forth in the code to find the next step, for example finding where a function is called. This could make it hard to figure out where the code is going wrong.

Future Updates

To improve my game, my first focus would be the visual bugs and obstacle collision as these are issues that affect gameplay. In order to fix the obstacle collision hindering movement, I would implement a check of what side of the obstacle the player collided with and only set one of the player's velocities to 0 depending on whether is was top/bottom or left/right. This would let the player slide against obstacles without being stopped completely or intersecting with them.

Beyond gameplay fixes, I would work on reducing errors in my code's design. I have used multiple global variables in my implementation, which are not ideal for keeping my code modular and maintainable. If I could transfer the global variables to local ones within classes, my code would be improved. Another change I would make is to encapsulate my classes better. Currently, attributes from an object are accessed directly rather than through a public function. I would give the combatant and player classes methods to retrieve and change particular attributes in order to encapsulate them.

Another change I could make to improve maintainability would be to split my project into multiple python files. The length of my code is a hindrance in debugging and improving it. By creating different files for each feature, the code would be broken down into separate parts. This would make my code significantly more modular and eliminate issues to do with systems being too reliant on each other.

Finally, I think a tutorial would be a good addition to the game. User feedback illustrated some controls which were not suitably intuitive or explained. A tutorial would help the user become familiar with those controls. It could also aid the player's direction within the game by telling the player that they must defeat the final boss in order to win.