

Final Report - Building A Game

Bailey Caplan - Year 3

Supervisor: Vasudha Darbari

Word-Count: 5907

Link to Video of Project: https://youtu.be/STrYQN-3_2E

Gitlab: <https://gitlab.cim.rhul.ac.uk/zkac236/final-year-project>

April 11, 2025

1 Introduction

Game development is more relevant than ever in today's digital landscape, playing a crucial role in entertainment, technology, education, and even social interaction. Since tennis for two (William Higinbotham in 1958)[Kau] the gaming industry has experienced explosive growth with the games market now being worth over an estimated 200 billion Dollars[Cle25]. This is greater than both the film and music markets combined. With this growth came an equally large amount of influence over the digital space and consequently modern popular culture. It is estimated that globally, around 3 billion people[Dua25] play games on a regular basis. Most predictions show massive growth, over the coming years, of both the value of the industry and the market as a whole.

Game development is not just about entertainment; it's a driving force in tech innovation, social interaction, and economic growth. As digital experiences become more immersive, the demand for skilled developers, creative storytellers, and cutting-edge technology will only increase.

I will be taking you through the development process I have engaged in over the past year. My project began as a basic Roguelite; as laid out in both my plan and in interim report. Since then, it has evolved from its original state to incorporate more traditional RPG elements. I feel that this has increased the depth of the gameplay experience by allowing the player to better customize the character they will be spending their time in my game with. It still follows the basic structure of a Roguelite; play through levels collecting items to upgrade your character before dying and starting all over again losing the vast majority of your progress.

The term "Roguelite" is broad, encompassing games that blend procedural generation, risk-reward systems, and meta-progression. Unlike strict "Roguelikes,"[Sz.22] Roguelites often soften the brutality of permadeath with persistent unlocks or meta-progression; making them accessible while retaining challenge.

For example, Hades (Supergiant Games, 2020) epitomizes this balance. Players collect temporary "boons" during runs, losing them upon death but gradually unlocking permanent upgrades. This loop creates a satisfying blend of short-term tactics and long-term strategy.

The genre's flexibility is key to its popularity. Roguelites can integrate action, strategy, or RPG mechanics, appealing to diverse player demographics. My project embraces this adaptability, merging Roguelite structure with RPG customization to offer a fresh take on the formula. My changes will be explained in the next section.

In the following sections, I'll detail my development process from initial design, to the integration of RPG systems and explore how these changes elevate the player experience.

2 Literature Review

The process of developing a game doesn't differ hugely from the development of other software. Like with small software development firms, small game development studios like 'supergiant' (I am aware

of the antiphrasis) tend to use the scrum/agile methodology. This usually means that the project leader will have a greater degree of control over development team and consequently greater design flexibility. The agile methodology [TT20] functions through the use of sprints. Sprints are periods of time in which a team/teams will work on the project before meeting to provide feedback on their progress and take in new ideas from the customer.

The Waterfall Methodology is a traditional, linear approach to software development and project management, where each phase must be completed before the next one begins. It is one of the oldest and most structured project management methodologies. Very often, large (often known as AAA) development studios will employ this methodology. The waterfall methodology requires a well-documented and detailed plan to be laid out before development can begin. This is followed by the design phase in which decision regarding the development of the project are made. The game is then developed based upon the requirements found in the plan and decisions made during the design phase. No major changes to the project can be made at this point. Testing follows development and any bugs found during this period will be fixed before the game is released to the customers.

A difference that can be seen between game development and most other software development fields is that there is often no specific customer in mind. Studios will instead cater to the wants and needs of stakeholders. This often brings game studios into conflict with their own customer base as while the customer base wants an enjoyable gaming experience, the best product for most stakeholders is one that will: ‘make them the most money’. Projects that obviously prioritise the generation of income over producing a high-quality experience are very rarely beloved by the gaming community. For example, the most recent addition to the call of duty franchise: “Call of Duty: Black Ops 6” currently has a mostly negative rating on steam (the leading digital distribution platform and storefront for PC gamers). While still popular, the negative reception to this most recent installation has meant that the franchise no longer sits at the top of the gaming world as it once did.

Most games will be designed in house with little consumer input until the final stages of testing by which time it is unlikely that much can be done if the product is poorly received. This is more often the case with large developers. Famously, ‘Apex Legends’ (Respawn Entertainment, Feb 2019) was released on the same day that it was announced. Fortunately for the developers, this game was well received and was a critical success on launch. However, this is very often not the case. ‘Battlefield 2042’ (released 2021) was criticised for its’ lack of polish by the gaming community during the beta testing phase. The developers (DICE) did very little to address this due to their lack of time and resources available before it was set to release. Because of this, sales of the game were very poor. This directly led to DICE losing development rights to ‘Battlefield’; a property that they created and have had full creative control over since 2003.

However, this is not the case across the board as while the initial idea for a game will still almost always be pitched in house, indie developers will sometimes attempt to engage their communities in the development process. This helps smaller developers for many reasons, for example: it can help to increase peoples’ awareness of your game. It also serves as a great way to gather feedback through releasing concept images and later demo snippets for their community to look over and critique at all stages of development. This can also provide the consumer some level of reassurance as to the completion of the final product as they are able to directly follow it.

One of the biggest issues within indie development is the overpromising of features. The most infamous example of this was ‘no mans sky’ (Hello Games, 2016). Many features were promised on launch such as there being 18 quintillion unique planets to explore, multiplayer, every planet having unique biomes and creatures and more. A full list of promised features can be seen bellow. The game was delayed from June to August 2016 for which they received death threats. Due to a desire to avoid another delay, the game was rushed to completion and ended up releasing in an unfinished state. This could have been avoided had they communicated better with their community.

An important thing to note is that the developers continued to develop the game after release and have improved the game to the point where it has now won numerous awards, including ‘Best Ongoing Game’ at The Game Awards 2020.

For my project, I chose to follow an agile approach to my programming despite not working as part of a team. The agile methodology has allowed for my project to evolve organically and helped massively when encountering issues. It has meant that problems encountered during the development period can be given the time they deserve, and essential features can be prioritized in line with the speed at which my project has developed. For example, I found that implementing player actions

such as movement and interaction was taking longer than expected, but instead of moving on when reaching my allotted time limit, I continued to work on it until it was functional. I felt that prioritising a fluid and comprehensive gameplay experience was more important than implementing features I had planned to include at later stages of development and had to cut.

In addition to the early adoption of a game, it is also important to ensure that the player is able to engage and stay engaged with the gameplay experience. To this end, effective UX [Wes] (User eXperience) design is critical for player retention, especially in difficult genres like Roguelites. The most prevalent and important of these features is a tutorial. Tutorials are a section found in many games where the controls and key game mechanics are explained. Not including a tutorial can leave players feeling lost and can mean that they stop playing entirely. From personal experience, I recently attempted to play *Fallout* (interplay entertainment, 1997) but found myself very confused about how to do almost anything in the game. Because of this, I stopped playing the game after a short period of time and have not gone back to it since.

However, many gamers find tutorials to be very tedious, boring and in many cases unnecessary. Developers employ a number of different strategies in order to sidestep this issue. Some will attempt to disguise tutorials within their game by working them into the narrative of the game. The first mission of the game *Call of Duty 2: 'Red Army training'* [IG15] involves the player engaging in bootcamp of sorts where the player is made to shoot at teddy bears and plates as well as throw potatoes (practice grenades) before being sent off to the battlefield. This serves to teach the player the mechanics of the game, without breaking the players immersion as the tutorial does not break with the tone established by the games' world. Other examples of this include *Halo* and later *Call of Duty* titles.

Developers may instead design their game to teach the player how to play by putting the player into situations that force them to carry out certain actions (*Super Mario Bros.*, *Dark Souls*). In the case of *Super Mario Bros.*, in the first level, the developers have included simple obstacles like different sized pipes and pits. These obstacles can prevent the player from progressing if they don't employ the ability to hold the jump button as opposed to just pressing it in order to increase Mario's jump height and airtime.

An important introduction in recent years, coinciding with the explosive growth of the games industry, is accessibility measure in videogames [MBC24]. Most video games are inherently accessible by virtue of only needing a console and a screen in order to run. Because of this, implementing further accessibility measures within games wasn't a priority for most developers until fairly recently. Of course, there are a number of examples of older games that did include features like the ability to re-map keys to make it easier for disabled players to enjoy the game (*Mortal Combat 2*, *Street Fighter 2 Turbo* and *DOOM* – All of these games surprisingly released in 1993). Other features that existed before the modern era of gaming (roughly 2010 – current day) include difficulty levels to enable less able gamers to tailor the game to their level ability and closed captions to help deaf and hearing-impaired gamers to better enjoy games (*The Sims*, 2000 and *Halo: Combat Evolved*, 2001). Modern games most often include these accessibility measure as well as new innovations in the field such as colour-blind modes, content filters, audio cues and adaptive controllers.

Game development relies on a diverse ecosystem of tools, engines, and programming practices that vary depending on project scope, team size, and target platform. Game engines provide the foundational framework for rendering, physics, audio, and scripting. These include Unity, the Unreal engine, Frostbite, Godot and Source. The most popular of these are Unity, Unreal and Godot. With all three being used different reasons and by different groups of developers. Unity, for example, is favoured for its versatility and ease of use, especially by smaller teams and for mobile games. The Unreal engine on the other hand is known for its high-fidelity graphics which makes it suited for AAA game development. I have mentioned Godot, as it is growing in popularity rapidly (due to it being free and open source) and may one day be as or more popular than the previously mentioned engines.

For my project I made the mistake of choosing to program in python with pyGame [Shi00] as I am most comfortable programming in python. I call this a mistake as pyGame is not a game engine like Unity or the Unreal Engine. It is a library containing a number of modules that are able to carry out simple tasks. This means that unlike with the previous mentioned game engines, complex features like physics, AI, and networking (etc...) need to be implemented by the programmer. This is not only difficult to do, but also (more importantly) consumes vast quantities of time. I acknowledged this in my interim report but had done too much work at that point to change course. In future projects, it is unlikely that I will continue to use pyGame. However, I do feel that it has provided me with valuable

experience and I am now able to take what I have learned, about the building of features (and the difficulties of pyGame), forward. In the next section I will explain how I went about using pyGame to implement my desired features.

3 The Game

This section hopes to explain the mechanics found within my game and will highlight key pieces of code that helped bring about these mechanics.

First, however, I will run through the playing of the game.

Upon launching the game, the player is presented with an intro screen with 4 buttons shown on screen: 'New Game', 'Options', 'Load Game', 'Quit'. While, 'New Game' and 'Load game' will have their current versions explained in the next few sections, the 'Options' menu is still undergoing development and so in addition to explaining its existing functionality, I also hope to lay out my plans for it going forward. The 'Quit' button closes the game window.

3.1 Starting a new game:

Pressing the 'New Game' button will set the players stats to the default. For demonstration purposes, only the money stat is set here will the rest are initialised within the player class (in a later version of the game I will be setting further stats here to allow for better character customisation and to bring the game more in line with an RPG). The player is then allowed into the playable area.

Once in the playable area, the player should be able to see their character model, an enemy character model and the players lives in the top left corner. Currently, I am developing a new animation for the attack and will later be employing the same algorithm to give a new enemy a more interesting AI that is able to detect and follow the player. The red and blue circles are being used in testing for these features. Both additions are still in development but I thought would be interesting to include.

The player is able to move the character model by pressing the arrow keys. Pressing 'n' will save the players current money stat to a database file. I will be going into further detail in the section explaining 'Load Game'. Pressing 'f' will start the attack animation. Currently, the direction of the attack is decided by the momentum of the player character. However, as discussed earlier, I am in the process of changing this as I feel that having the player attack in the direct of the players mouse is more intuitive for most gamers and will hopefully reduce the programming complexity currently present because of the existing method of attack.

If the player presses the 'escape' key it will take them to a pause menu. Will in the pause menu, the game-state is frozen. In the pause menu, the player is presented with three buttons: 'Return', 'Options' and 'Save + Quit'. 'Return' will unfreeze the game-state and allow the player to continue playing the game. 'Options' takes the player to options menu (explained bellow). 'Save + Quit' will save the players progress before close the game window. In previous versions of the game, this option was simply 'Quit'. I made this change as I have often forgotten to save my progress when playing games. This can be very frustrating for players and so I have added the 'save' feature to 'Quit' in order to mitigate the chances of this happening.

Pressing the 'Tab' key will open the players inventory. Initially the players inventory will be empty but will display the amount of money the player currently has. The player is able to use the items in their inventory by selecting the item and pressing 'use'. A shopkeeper NPC has not yet been added to the game. Because of this, the best way to add items to the players inventory is to do so through the in-game testing mode feature. This is accessed and exited by pressing the 'p' key.

3.2 Combat:

The player is able to engage in combat if either their character or their attack collide with the enemy at the moment combat is turn based. The player is able to make choices about how they want to interact with the enemy on their turn before the enemy is then able to take their own turn and do the same. The enemy decides what to do based on a random number generator I have attached to their potential actions. Unlike the player, the enemy can miss attacks. This is also decided by a random number generator. On the players turn they are able to take one of four actions, they can pass their turn (do nothing), they can attack, they can heal (if there are available health potions) or they can

Promises and mentioned features	At Launch
Large-scale joinable space battles between factions	No
Destroy space stations	No
Roaming freighters	No
Factions with differentiating attributes	No
Trade ships dock with freighters	No
NPC ships launching from ground	No
Crashed freighters	No
Ringed planets	No
No skyboxes	No
Butterflies	No
Animals interact with each other and the environment	Not really
Smart AI that knows what planets you've been to	No
Rivers	No
Radio chatter	No
Packs of walking sentinels	No
Name your ship	No
Fly between star systems manually	No
Resources depend on distance from star	No
Play the game exclusively as a trader	Not really
Play as a space pirate	No
Scout for elements from galactic map	No
Sand planets	No
Water worlds	No
Hack reinforced doors	No
Unique ships	No
Portals	No
Land on asteroids	No
Crafting with complexity	No
Planets that have vastly different resources	No
Its own periodic table	Somewhat
Ships with cloaking devices	No
Call wingmen to aide you in battle	No
Glass windshield to see inside ship	No
Manoeuvre like in the trailer	No
Day/night cycles determined by the orbit of the sun	No

Figure 1: The list put together by the no mans sky subreddit of all the promised features that were missing on launch.

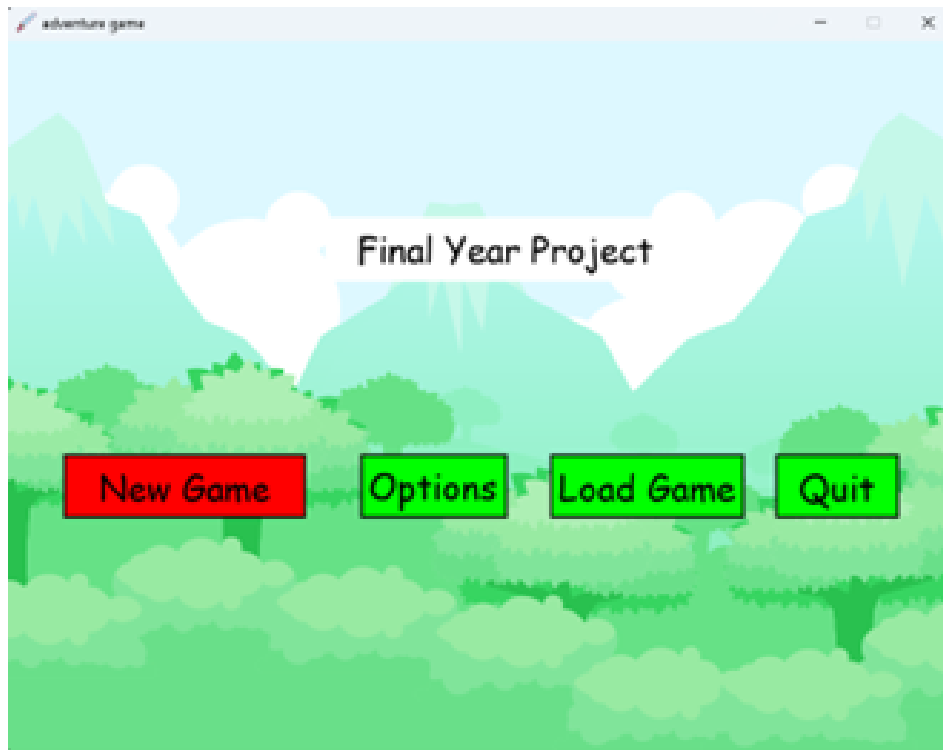


Figure 2: The start screen

run away. The chance that the player is able to get away from the enemy is once again decided by a random number generator. This is a lot of random generation and I am aware that this can cause frustration for the player. As a result, I have decided that this is not how I want combat to look in the final product.

In fact, I have been considering introducing a finite state machine as opposed to using random generation, as it would mitigate the frustration issue while also improving the depth of the gameplay. I have also considered and will be changing combat to be carried out in real time instead. the enemy missing could be replaced with a skill-based mechanic instead, such as a dodge or dash like the aforementioned 'Hades'. The new attack mechanic seen bellow is my first step towards realising this change. I will decide on which method of combat is most suited to my game after having had time to thoroughly play-test both.



Figure 3: notice the blue and red circles

```
def attack(): 2 usages new *
    global atx, aty
    mx, my = pygame.mouse.get_pos()
    circle1 = pygame.draw.circle(screen, color: "blue", center: (atx, aty), radius: 30, width: 0)
    circle2 = pygame.draw.circle(screen, color: "red", center: (mx, my), radius: 50, width: 0)
    screen.blit(swordImg, dest: (atx, aty))

    dx = mx - atx
    dy = my - aty

    angle = math.atan2(dx, dy)

    mvx = math.sin(angle) * 2 # increase number to increase speed
    mvy = math.cos(angle) * 2 #

    atx += mvx
    aty += mvy
```

Figure 4: After starting the game it is very easy to spot two large circles [Pro22] on the screen. The circles themselves are for testing only and will not be present in the final version of the game. The red circle is drawn on the position of the cursor. The blue circle is drawn on the position of the player and will attempt to move towards the position of the mouse at all times. Currently, pressing the 'f' key will reset the blue circle's position to the player character. This mechanic will be used in conjunction with existing functions to produce two new features; an attack that starts at the player and moves towards the position of the mouse and a potential new pathing AI for the enemies which give them the goal of reaching the player. It works by getting the mouse position (mx, my) then drawing two circles: A blue circle at (atx, aty) with radius 30 and A red circle at mouse position with radius 50. Then it calculates the angle between the attacker and mouse position using math.atan2 , before then calculating movement vectors (mvx, mvy) based on that angle. The attacker's position (atx, aty) is then updated.


```

if (event.key == pygame.K_p):
    debug = True
    time.sleep(1)

while (debug == True):
    print("still running")
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            sys.exit()

    # if any key is pressed check whether its left or right
    if event.type == pygame.KEYDOWN:
        print("a keystroke has been pressed")
        if (event.key == pygame.K_t):
            shop()

        if event.key == pygame.K_w:
            player1.increasemaxhealth()
            player1.printhealth()

        if event.key == pygame.K_s:
            player1.get_damage()
            player1.printhealth()

        if event.key == pygame.K_e:

```

Figure 5: debug mode

3.3 Debug Mode

During the development of my game I found that the best way to test newly added functions or alterations to existing functions was to bind said functions to a key and access them within the games' environment. I feel that this helps with visualizing how different part of the game interact with each other. However, I did not want these features and actions to be accessible when playing the game in the way its intended. To this end, I created an in-game mode only accessible with the 'p' key. 'Debug Mode' gives existing keys new functionality when active that allow the player/developer to force changes within the game. This is based on the debug console. A debug console is a tool used during debugging, allowing developers to interact with a running application, view output, and evaluate expressions, ultimately helping to identify and fix issues in the code. They are usually removed from the game on release with some exceptions (most Bethesda Titles, Subnautica, the Binding of Isaac, etc...).

My game in its current state does not yet need a full-fledged debug console as it does not have enough content to warrant one. So, I have compromised and added my debug mode. Whilst in debug mode the user can: access the shop without visiting the shopkeeper NPC by pressing the 't' key (as mentioned earlier this is also the easiest way to add items to the players inventory), manipulate the players health stats with the 'w' and 's' keys, reset the enemies' position and health with the 'e' key, and exit the game with 'y' key. More functions are likely to be added as development continues.

3.4 Options

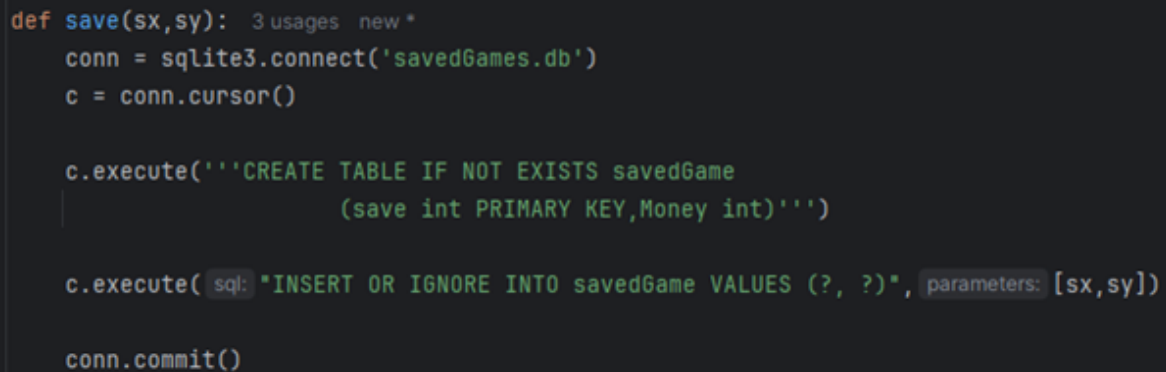
Pressing the ‘Options’ button will take the player to the options menu. Here the player can turn on/off the games’ music.

As mentioned previously, the options menu is still under development will not become a priority until very close to release. This is because only I am testing the game currently and so the game is being developed with that in mind. As the game approaches completion, it will become necessary to implement a number of accessibility features to ensure that the final product can be enjoyed by as many people as possible. These features will include the ability to rebind actions to different keys, closed captions, optional audio description (this feature is not currently in use by most games and will be exciting to go about implementing), and potentially colour-blind modes.

3.5 Load game and the Save mechanic:

Pressing the ‘Load Game’ button allows the player to input into the python console an ID number (representative of an existing value from the database) in return for the amount of money the players character had the last time they played to use in place of the default starting value for money. The player can see the ID number of there most recent save displayed in the python console upon closing the game. This will be changed an in game display in the near future, and the database will be expanded to include other player related information. I have use sqLite3 to create, access and alter this database. In my interim report I mentioned that I would be using mySQL rather than SQLite. However, I wasn’t particularly attached to mySQL and after finding it easier to find documentation for sqLite3, I switched over. This change has made no real difference that I can see.

A number of functions were created to handle saving and accessing saved games as detailed bellow:

A screenshot of a code editor showing a Python function named 'save(sx,sy)'. The code is written in a dark-themed editor with syntax highlighting. The function connects to a SQLite database named 'savedGames.db', creates a table named 'savedGame' if it doesn't exist, and then inserts the values of 'sx' and 'sy' into the table. The code is as follows:

```
def save(sx,sy): 3 usages new *
    conn = sqlite3.connect('savedGames.db')
    c = conn.cursor()

    c.execute('''CREATE TABLE IF NOT EXISTS savedGame
                (save int PRIMARY KEY,Money int)''')

    c.execute(sql: "INSERT OR IGNORE INTO savedGame VALUES (?, ?)", parameters: [sx,sy])

    conn.commit()
```

Figure 6: The ‘save(sx,sy)’ function does most of the work. First it connects to the existing database or creates one if it doesn’t exist. It then does same for the table inside the database before then populating it with the values inserted as arguments.

```

def checkSave(): 4 usages new *
    conn = sqlite3.connect('savedGames.db')
    c = conn.cursor()
    num = 0
    for row in c.execute("SELECT save FROM savedGame"):
        num += 1
        print (num)
    return num + 1

```

Figure 7: The 'checkSave()' function is used to check for and return the next available save ID, using a for loop to iterate through the table. This function also has to connect to the database as it is its own function and while I have used global variables throughout my code, I prefer not to use them when I can.

```

def useSave(num): 1 usage new *
    conn = sqlite3.connect('savedGames.db')
    c = conn.cursor()
    for row in c.execute('SELECT save FROM savedGame'):
        testnum = "(" + str(num) + ","
        textnum = "".join(testnum)
        if str(row) == str(textnum.strip()):
            c.execute(sql: "SELECT money FROM savedGame WHERE save = (?)", parameters: [num])
            hope = c.fetchone()

            return hope[0]

    else:
        print("something went wrong")

```

Figure 8: The 'useSave(num)' function also connects to the database for the same reasons as 'checkSave()'. useSave locates the value of money at the index indicated by the argument 'num' then returns that value. This function appears convoluted on the surface as result of the awkward string/tuple handling. During testing, I noticed that the values output by the SELECT SQL query had excess characters around them. As I wasn't sure how to remove this excess, I instead had to add the excess to the argument before carrying out the if comparison. And although the if statement is only being used for error handling and could be removed, the function works as intended, so I have no reason to change it other than for aesthetics. I may do so at a later date, but for now, changing it feels unnecessary.

3.6 Additional Interesting Pieces of Code

```
def save(sx,sy): 3 usages new *
    conn = sqlite3.connect('savedGames.db')
    c = conn.cursor()

    c.execute('''CREATE TABLE IF NOT EXISTS savedGame
                (save int PRIMARY KEY,Money int)''')

    c.execute(sql: "INSERT OR IGNORE INTO savedGame VALUES (?, ?)", parameters: [sx,sy])

    conn.commit()
```

Figure 9: The 'save(sx,sy)' function does most of the work. First it connects to the existing database or creates one if it doesn't exist. It then does same for the table inside the database before then populating it with the values inserted as arguments.

```
def isCollision(enemyX, enemyY, swordX, swordY): 1 usage ⬆ bailey caplan
    distance = math.sqrt((math.pow(enemyX - swordX, 2) + (math.pow(enemyY - swordY, 2)))
    if distance < 27:
        return True
    else:
        return False
```

Figure 10: This function uses the Euclidean distance formula to determine whether or not the enemy has come within range of the players sword.

```

class button(): 16 usages 1 bailey caplan
    def __init__(self, color, x, y, width, height, text=''): 16 usages 1 bailey caplan
        self.color = color
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.text = text

    def draw(self, screen, outline=None): 16 usages (1 dynamic) 1 bailey caplan
        if outline:
            pygame.draw.rect(screen, outline, rect(self.x - 2, self.y - 2, self.width + 4, self.height + 4), width: 0)
            pygame.draw.rect(screen, self.color, rect(self.x, self.y, self.width, self.height), width: 0)

        if self.text != '':
            font = pygame.font.SysFont(name: 'comicsans', size: 30)
            text = font.render(self.text, antialias: 1, color: (0, 0, 0))
            screen.blit(text, (
                self.x + (self.width / 2 - text.get_width() / 2), self.y + (self.height / 2 - text.get_height() / 2)))

    def isOver(self, pos): 21 usages (2 dynamic) 1 bailey caplan
        if pos[0] > self.x and pos[0] < self.x + self.width:
            if pos[1] > self.y and pos[1] < self.y + self.height:
                return True
        return False

```

Figure 11: Above is the class I have used throughout this project to generate buttons. It is used for button objects found in: the main menu, the options menu, the pause menu and the shop. It first initialises a class with attributes: color, x, y, width, height and text. It contains two functions; draw and isOver, to draw the shape of the button, and identify if a position falls within the range of that button.

```

def viewinventory(): 1 usage 1 bailey caplan +1*
def which_invselected(): 1 bailey caplan
    print("At {0}".format(invselect.curselection()))
    return int(invselect.curselection()[0])

def use_entry(): 1 bailey caplan +1*
    selectedItem = inventory[which_invselected()]
    if selectedItem.name == "health potion":
        player1.get_health()
        del inventory[which_invselected()]
        set_invselect()
    elif selectedItem.name == "heart":
        player1.increasemaxhealth()
        del inventory[which_invselected()]
        set_invselect()

def make_window(): 1 bailey caplan +
    global invselect
    win = Tk()
    win.title("Inventory")
    frame2 = Frame(win) # Row of buttons
    b2 = Button(frame2, text="USE ITEM", command=use_entry)
    b2.pack(side=LEFT)
    L = Label(win, text=('money: $' + str(player1.money)))
    L.pack()
    frame4 = Frame(win) # list of items in inventory
    scroll = Scrollbar(frame4, orient=VERTICAL)
    invselect = Listbox(frame4, yscrollcommand=scroll.set, height=6)
    scroll.config(command=invselect.yview)
    scroll.pack(side=RIGHT, fill=Y)
    invselect.pack(side=LEFT, fill=BOTH, expand=1)
    frame2.pack()
    frame4.pack()
    return win

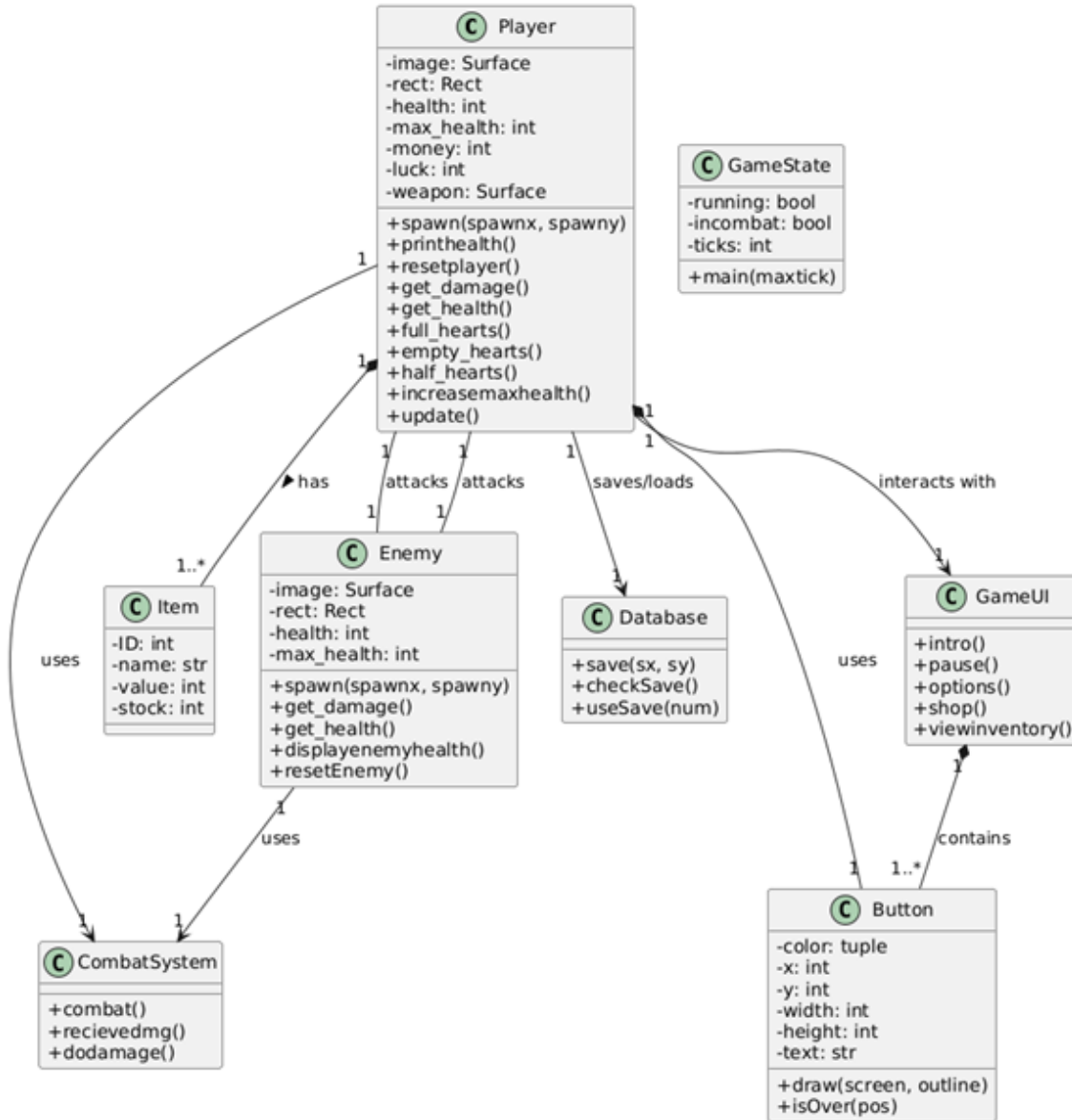
def set_invselect(): 1 bailey caplan +
    #shoplist.sort(key=lambda record: record[1])
    invselect.delete(first=0, END)
    for item in inventory:
        invselect.insert(END, *elements: "{0}".format(item.name))
win = make_window()
set_invselect()
win.mainloop()

```

Figure 12: I feel I need to explain that here I have used Tkinter[[LvR91](#)] which is a binary module that contains the low-level interface to Tcl/Tk. Tk is a Gui toolkit for python. Due to it producing small grey boxes when run, I have tried to remove as much of it as possible from my project. For example, in the place of the pause menu, this game used to generate a Tkinter window to handle the task. The same was true for the shop. I made the decision to change this part of the game as it was pointed out to me that the Tkinter windows feel like they take you out of the game. The only remaining pieces of Tkinter are found in combat which is currently receiving an overhaul and the inventory where it really doesn't look too out of place. However, at a later date I will also be replacing the tkinter inventory with an inventory created in pygame that will hopefully incorporate some sort of way to manage items more effectively such as the ability to drag and drop items where the player might want them.

4 UML

UML (Unified Modelling Language) is a standardised modelling language used in software engineering to visualise, design, and document software systems. It provides a set of graphical notations to represent different aspects of a system, helping developers, architects, and stakeholders understand and communicate system design. Below is a UML diagram for my project:



5 Results and Gameplay Experience

The final version of the game will hope to successfully combine Roguelite mechanics, such as procedural generation and permadeath with meta-progression, with RPG elements, including stat customization, an inventory system, and persistent upgrades. Key features will include mouse-driven combat, where players aim attacks using cursor positioning for more intuitive gameplay, replacing the original momentum-based system. SQLite integration allows players to save and load their progress across sessions, while a built-in debug mode provides tools for testing and iteration. The user interface will hopefully be refined by replacing Tkinter with native PyGame menus, enhancing immersion and responsiveness. However, the game in its current state is defined by one word, “unfinished”. A number of features promised originally have been implemented though. These features include: a moveable

player sprite, an interactive enemy sprite, a functioning shop, a functioning inventory, character data is store in a database via SQL, basic combat has been implemented. Most importantly, a basic gameplay loop has been established.

The Agile methodology played a crucial role in shaping the game’s evolution, allowing for iterative improvements based on player feedback (insights made by myself, my supervisor and by people a had test it). For example, the combat system’s transition from turn-based to real-time was a direct response to playtester preferences. However, this flexibility also introduced technical debt, such as reliance on global variables, which has complicated certain areas of development such as converting Tkinter sections to pyGame.

UML-driven design helped structure systems like the inventory and save features, reducing spaghetti code, but will very likely prove less useful for real-time combat’s event-driven logic. Modular coding practices, particularly in isolating the SQLite save system, made it easier to extend features, whereas tight coupling between combat and enemy AI made fine-tuning attack patterns more challenging than anticipated.

The project underscored the importance of methodological flexibility, with Agile proving far more effective than a rigid Waterfall approach for an indie-scale project. Player-centric design choices, such as prioritizing UX polish over feature bloat, will significantly improve retention. However, underestimating PyGame’s limitations, particularly its lack of built-in physics and advanced UI tools, led to unnecessary hurdles. Moving forward, migrating to a more robust engine like Unity could alleviate these constraints while enabling advanced features like procedural content generation and better AI pathing.

From a technical standpoint, the game performed efficiently on mid-tier PCs, maintaining 60 FPS at 1080p resolution. However, PyGame’s single-threaded architecture limited scalability for more complex scenes, highlighting potential bottlenecks for future expansions.

6 Conclusion

This game development project has been an extensive and rewarding exploration of the Roguelike genre, enriched with RPG mechanics to create a more immersive and customizable player experience. Over the course of development, the project evolved significantly from its original design, incorporating new systems, refining existing mechanics, and adapting to unforeseen challenges. All while adhering to Agile development principles. The iterative process allowed for continuous improvement, ensuring that core gameplay elements such as combat, inventory management, and progression remained engaging and functional.

One of the most valuable lessons from this project was the importance of tool selection. While PyGame provided a comfortable entry point due to my familiarity with Python, its limitations: particularly the lack of built-in physics, advanced rendering, and efficient UI tools; required extensive custom implementations for features that would have been streamlined in engines like Unity or Godot. Despite these hurdles, working with PyGame deepened my understanding of low-level game systems, from collision detection to state management, and reinforced the necessity of choosing the right tools for a project’s scope.

The technical implementation of features such as save/load functionality (via SQLite), mouse-driven combat, and a debug mode demonstrated the project’s progression from a basic prototype to a more structured game. The combat system, though still in flux, highlighted the tension between turn-based strategy and real-time action, prompting critical design decisions about pacing and player agency. Similarly, the inventory system’s shift away from Tkinter to a more integrated PyGame UI improved immersion, though further refinements, such as visual item representations and drag-and-drop interactions, would enhance usability.

Beyond code, this project hopes to emphasize the significance of player experience (UX) in game design. Tutorial integration, accessibility considerations, and menu responsiveness are all areas that demand attention, as even small oversights could disrupt engagement. The decision to auto-save progress upon quitting, for example, was a direct response to player frustration observed in other games. Future iterations would benefit from additional UX polish, including controller support, audio cues, and dynamic feedback (e.g., hit effects, screen shake).

Looking ahead, I feel that the game has strong potential for expansion. Procedural level generation, deeper meta-progression (e.g., unlockable abilities or persistent upgrades), and more sophisticated

enemy AI would align it more closely with successful Roguelites like Hades or Dead Cells. Community feedback, gathered through public demos or beta testing, could also guide development, ensuring the final product resonates with players.

Ultimately, this project was as much about learning as it was about creation. It reinforced the iterative nature of game development, where design, technical constraints, and player feedback constantly shape the outcome. Whether refining this prototype or applying these lessons to a new engine, the experience has been invaluable, not just in building a game, but in understanding the delicate balance between vision, execution, and adaptability that defines great game development.

References

- [Cle25] Jessica Clement. Global video game market value from 2022 to 2032. <https://www.statista.com/statistics/292056/video-game-market-value-worldwide/>, 2025.
- [Dua25] Fabio Duarte. How many gamers are there? (new 2025 statistics). <https://explodingtopics.com/blog/number-of-gamers>, 2025.
- [IG15] Matt Triplett IGN-GameGuides, Tyler. Call of duty 2 guide. <https://www.ign.com/wikis/call-of-duty-2/Walkthrough>, 2015.
- [Kau] Anna Kaufman. What was the first video game? detailing the first at-home video game, who invented it and more. <https://eu.usatoday.com/story/tech/2022/09/15/what-was-the-first-oldest-video-game/8021599001/>.
- [LvR91] Steen Lumholt and Guido van Rossum. tkinter — python interface to tcl/tk. <https://docs.python.org/3/library/tkinter.html>, 1991.
- [MBC24] Steve Andrews CITP MBCS. Accessibility in gaming. <https://www.bcs.org/articles-opinion-and-research/accessibility-in-gaming/>, 2024.
- [Pro22] Python Project. How to make a sprite follow your mouse cursor. <https://www.youtube.com/watch?v=dYa69SXoo80>, 2022.
- [Shi00] Pete Shinnars. pygame- about. <https://www.pygame.org/wiki/about>, 2000.
- [Sz.22] SZABADOS GY.N.; BÁCSNÉ B.É.; FENYVES V.; Bács Z.; Molnár A.; Ráthonyi G.; Rizwan H.; Orbán Sz.G. Roguelikegames-the way we play. <https://ojs.lib.unideb.hu/IJEMS/article/view/11610/11127>, 2022.
- [TT20] Martin Burchardt Theo Thesinga, Carsten Feldmann. Agile versus waterfall project management: Decision model for selecting the appropriate approach to a project. 2020.
- [Wes] Kevin Bierre; Jonathan Chetwynd; Barrie Ellis; D. Michelle Hinn; Stephanie Ludi; Thomas Westin. Game not over: Accessibility issues in video games.