Neo's Enchanting Adventures

Candidate Number: 4763

Centre Number: 12290

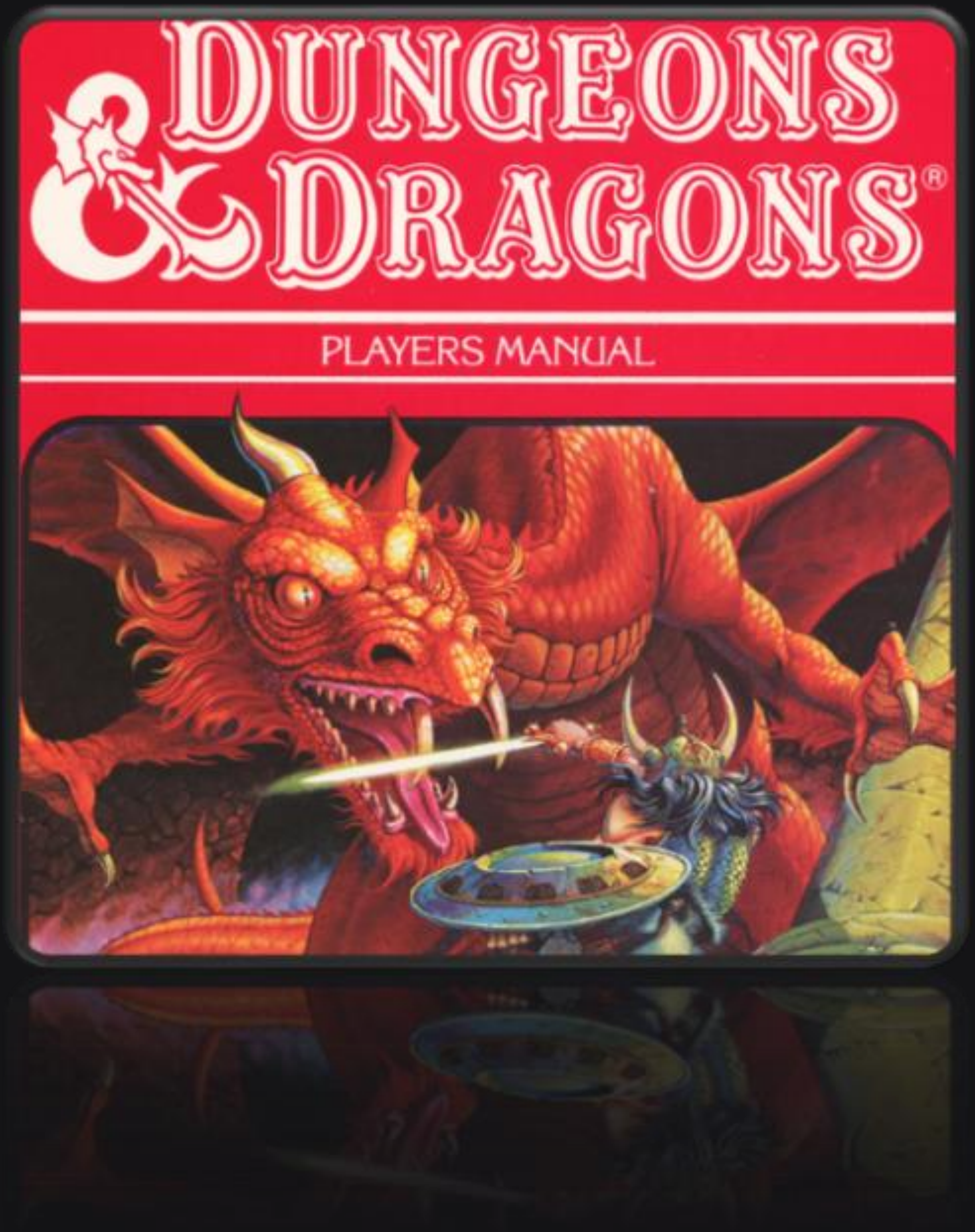
Structure

Contents

1 Analysis

1.1 Background to the role-playing games

The originals



The first instance of what people today would recognise as a modern RPG came into being with the development of the tabletop game, *Dungeons & Dragons* in 1974. Created by Dave Arneson and Gary Gygax and published by their company Tactical Studies Rules, it was a game that popularised many of the RPG conventions that are still being used today, such as character classes and abilities, races, experience and hit points (EXP and HP), levelling up, and turn-based combat.

To play it, you needed a character sheet on which to record your stats, books containing the rules, monsters and scenarios, seven multi-sided dice, and ideally, a good imagination. Tackling a quest described in the scenarios, players would collectively decide how to deal with situations as they arose, with dice being rolled to determine things like combat, trap evasion and lock picking. It’s a template that has continued to serve the tabletop RPG ever since.

The original PC RPGs—such as MUDs (a multiplayer real-time virtual world, usually text-based or storyboarded), or multi-user dungeons—appeared in the mid-70s. These weren’t for home computers, but mainframes, typically found in universities. They tended to be based on either Dungeons & Dragons, which itself launched in 1974, or be variously disguised takes on Tolkien. These included Dungeon, DND, Orthanc, and Oubliette. A few, such as *Oubliette*, had simple graphics, though most started out as just text or used ASCII’s standard set of text-mode graphics.

Despite the primitive technology, these games often offered surprising depth. Don *Daglow’s Dungeon* for instance, a 1975 D&D pastiche, offered control of an entire multiplayer party, mapping, NPCs with AI, line-of-sight-based combat, and both melee and ranged attacks. Moria, from the same year, served up wireframe graphics for its characters, and even featured rudimentary 3D views of its corridors.

1980s

For those outside universities, the genre really began around 1980. There had been games for home systems before that, including *Temple of Apshai* for the TRS-80 and Beneath Apple Manor for the Apple II, but few of them made real waves. 1980 saw the launch of *Rogue*, the first true dungeon crawl game, whose combination of randomly generated content and permadeath set the tone for today’s ‘roguelikes’. It would be a few more years before it and its clones would be on home computers—the PC version landed in 1984—but the basics were here.

As home computers became more popular over the ’80s, the RPG style-based games began to take over. *Wizardry*, for instance, launched in 1981, and the series ran until 2001. It used simple graphics and played out mostly using menus, in a way that most Western RPGs would soon try to move away from. However, its popularity in Japan led to it largely defining what that market thought an RPG was. Later games like *Final Fantasy* and *Dragon Quest* still follow its lead today.

Almost all RPGs of this era were fantasy based, though there were a few exceptions like Origin's car based *Autoduel* (1985, based on Steve Jackson Games' Car Wars) and *Starflight* (1986), which swapped the traditional party for the crew of a spaceship on a quest to explore the universe and find out why the stars of the galaxy are flaring and destroying everything around. (It would later inspire the wonderful *Star Control 2*, as well as be one of the lynchpins for BioWare's Mass Effect series.)

Most games of this era didn't have the disk space for text. A 51⁄4 inch floppy disk held around 720KB of data. Its more compact successor, the 3 1⁄2 inch floppy, held about 1.44MB. The floppier disks a game needed, the more expensive it was to produce. This is why, for example, the first Eye of the Beholder doesn't have an ending sequence. One was planned, but it would have required an extra disk. The publisher said no. Instead, your reward for getting to the end was a quick burst of text going, more or less, 'well done you won'.

Some games found ways around this problem. *Wasteland*, for instance, released in 1987, came with a printed book that resembled a Choose Your Own Adventure. The idea was that when you reached a critical part, the game told you which paragraph to read. This saved space on the disks for more maps, graphics and other good stuff that RPGs really needed.

By the end of the '80s, we were firmly in an age of innovation. *Drakkhen*, for instance, released in 1989, offered one of the first fully explorable, real-time 3D worlds. It was a simple one, full of death-traps, random encounters and poorly translated dialogue that made it tough to tell what was actually going on. But it still did it.

1.2 Describe the problem

After going through a brief history of a role-playing game as a genre in video gaming industry, the question is what makes a video game an RPG? What features a video game need to have to be considered an RPG?

Here are some of the key features of an RPG video game:

- Character and character's abilities
- Progression and levelling
- Interactive environment
- Story and narrative
- Items and Inventory
- Game combat
- User Interactions and Graphics

Character and character's abilities. As with any other part of RPG development, character attributes and actions are highly defined by the storyline of the game. These actions are performed indirectly within the game when the player commands the character to perform a specific task.

For instance, in a given RPG there will be at least a couple of character classes. The following are some sample class types:

- Barbarians
- Orcs
- Magicians / Wizards
- Zombies
- Humans

Each character class might even have subclasses of its own, with its own uniquely defined attribute. This will be tightly coupled to the storyline of an RPG. Choosing a certain class also enables different combat options such as the use of magic spells or the use of sword.

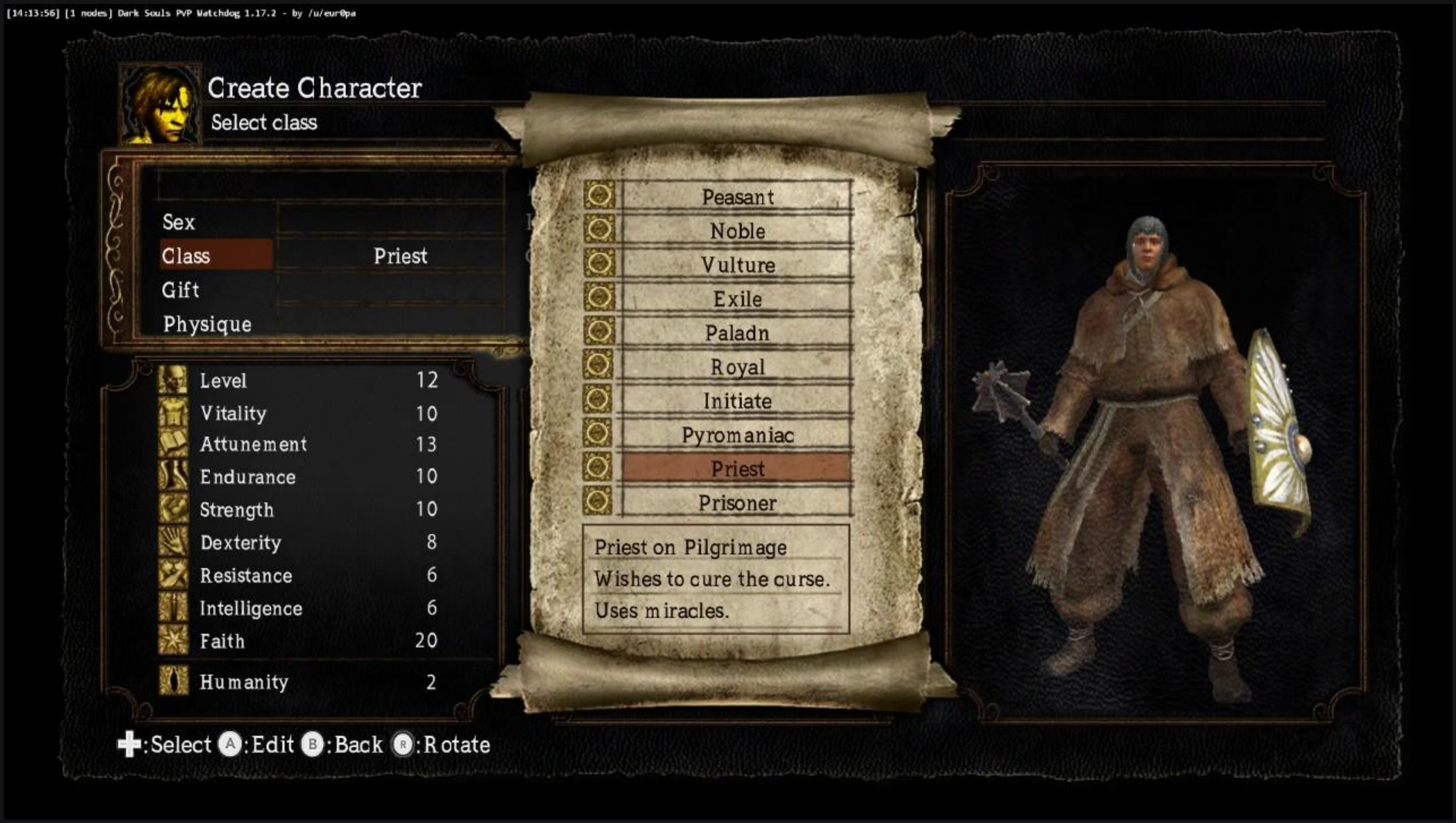
For instance, there is going to be player-character, who is technically the hero of the story and of the game. The hero is usually of a certain character class, e.g., Human class.

The Human class or race, then, will have some specific characteristics that will be inherited by the player character, or any other non-player character of the same type or class.

The strength of a character within the game is defined by the character class it belongs to and the type of actions it can perform. The performance of a character is defined by the value of the attributes defined within the character's class and race.

For instance, if we take two distinctive character classes and compare them side-by-side, such as a human and an orc, the orc will have far superior strength and brute force than the human. However, the human may have higher intelligence and problem-solving skills, which will out-rate the strength of the Orc if applied properly.

Most RPGs allow the player to modify their characters before the game starts, or even during game play. By default, every character class will have some default attributes, and the player is allowed to adjust the values based on some modifier. The basic fundamental features allowed for modification are the sex, class, or race of a character.


[14:13:56] [1 nodes] Dark Souls PVP Watchdog 1.17.2 - by /u/eur0pa

Create Character
Select class

Sex	
Class	Priest
Gift	
Physique	

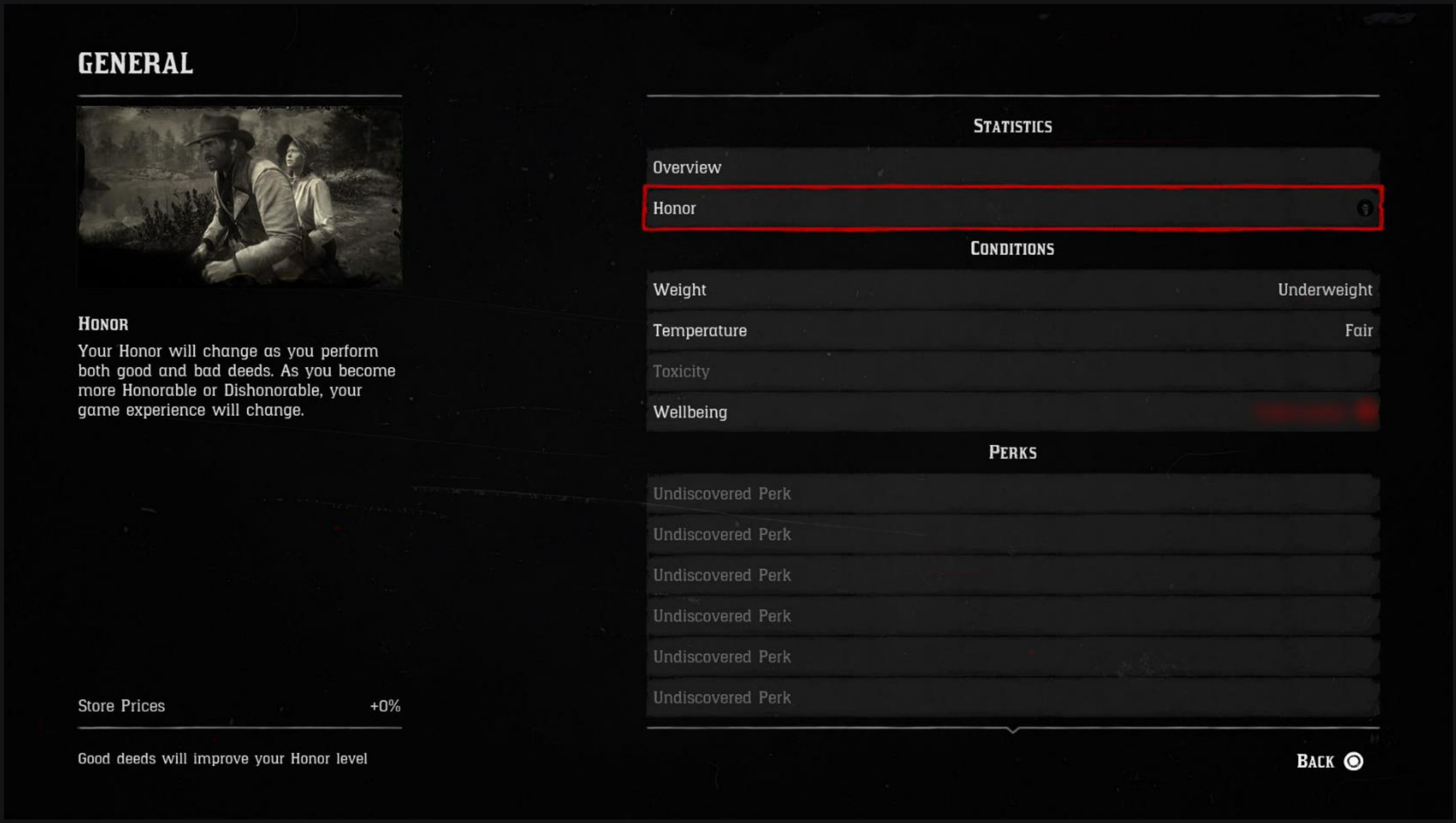
Level	12
Vitality	10
Attunement	13
Endurance	10
Strength	10
Dexterity	8
Resistance	6
Intelligence	6
Faith	20
Humanity	2

Peasant
Noble
Vulture
Exile
Paladn
Royal
Initiate
Pyromaniac
Priest
Prisoner

Priest on Pilgrimage
Wishes to cure the curse.
Uses miracles.

An example of character customisation. The game featured: Dark Souls 1

It all comes down to the budget and resources that are available during the production of the game. Some games can also introduce ethical attributes into the characteristics of the character. For instance, if there is the ability to kill or rob innocent bystanders within the game, and the player uses it, then the player will become less liked by the friendly non-player characters (NPCs), and they may not be as friendly or helpful as needed to complete your quest. In other words, a player will live by the consequences of your actions.



HONOR
Your Honor will change as you perform both good and bad deeds. As you become more Honorable or Dishonorable, your game experience will change.

Weight	Underweight
Temperature	Fair
Toxicity	
Wellbeing	
Store Prices	+0%

Good deeds will improve your Honor level

An example of player's actions affecting the environment. Game featured: Red Dead Redemption 2

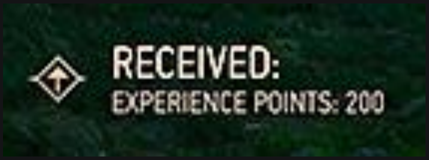
Lastly, character classes define your character attributes and hence define your character's strengths and weaknesses. These physical attributes can be simplified into the following: dexterity and strength, which determine the performance of a character during battle.

Progression and levelling. To engage the player, game designer use mechanics to enhance the performance of the player-character. The progress is what is termed levelling or experience in RPGs.

Levelling and experience are a key element of any role-playing game. A good levelling or experience tree will be defined for any RPG. This allows the player to develop their avatar through game play and become functionally more powerful by gaining more skills, points, and other resources necessary to complete their quest.

The ability to acquire new weapons, armour, clothing, and/or any other gameplay item defined in the world, the player will need to meet some specific thresholds within the game. These thresholds can be a combination of the player's acquired experience points, financial gains, and/or combat experience.

In RPGs, the progress of the character player is measured by counting some defined attributes specified by the game designer. Usually, the advancements are defined by the player completing a certain task to get experience points, and slowly, the tasks and the points rewards are increased throughout the game. The player then can use the experience points to enhance his or her avatar within the game.



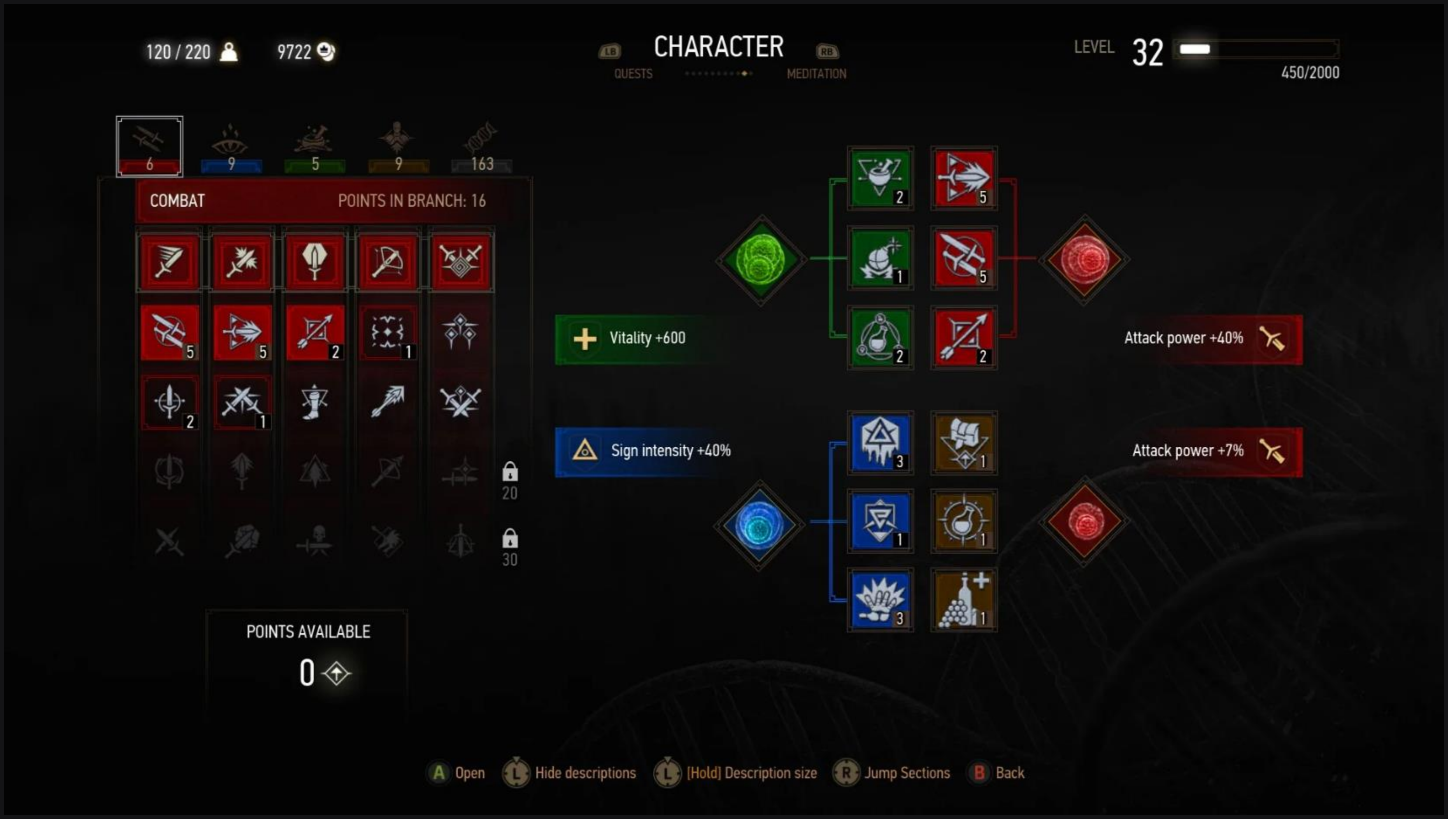
Game featured: the Witcher 3

Again, this is highly integrated with the storyline, character classes, and/or race the player has selected. Some common ways to acquire points are by killing enemies, combating non-player characters of no importance, and performing quests that have been defined within the game.

Just like in real-life, the more you play and apply your skills, the more experienced you become. The accumulation of your experience will then allow you to acquire better weapons and/or armour to strengthen your attack or defence for the next quest. Some games might give the player 100+ points and allow the player to distribute the points across the available character attributes for their avatar. Sometimes, the game automatically applies all of the experience to a specific area, such as strength.

Gaining experience will also allow the user to unlock more features and skills to be acquired by the player during gameplay.

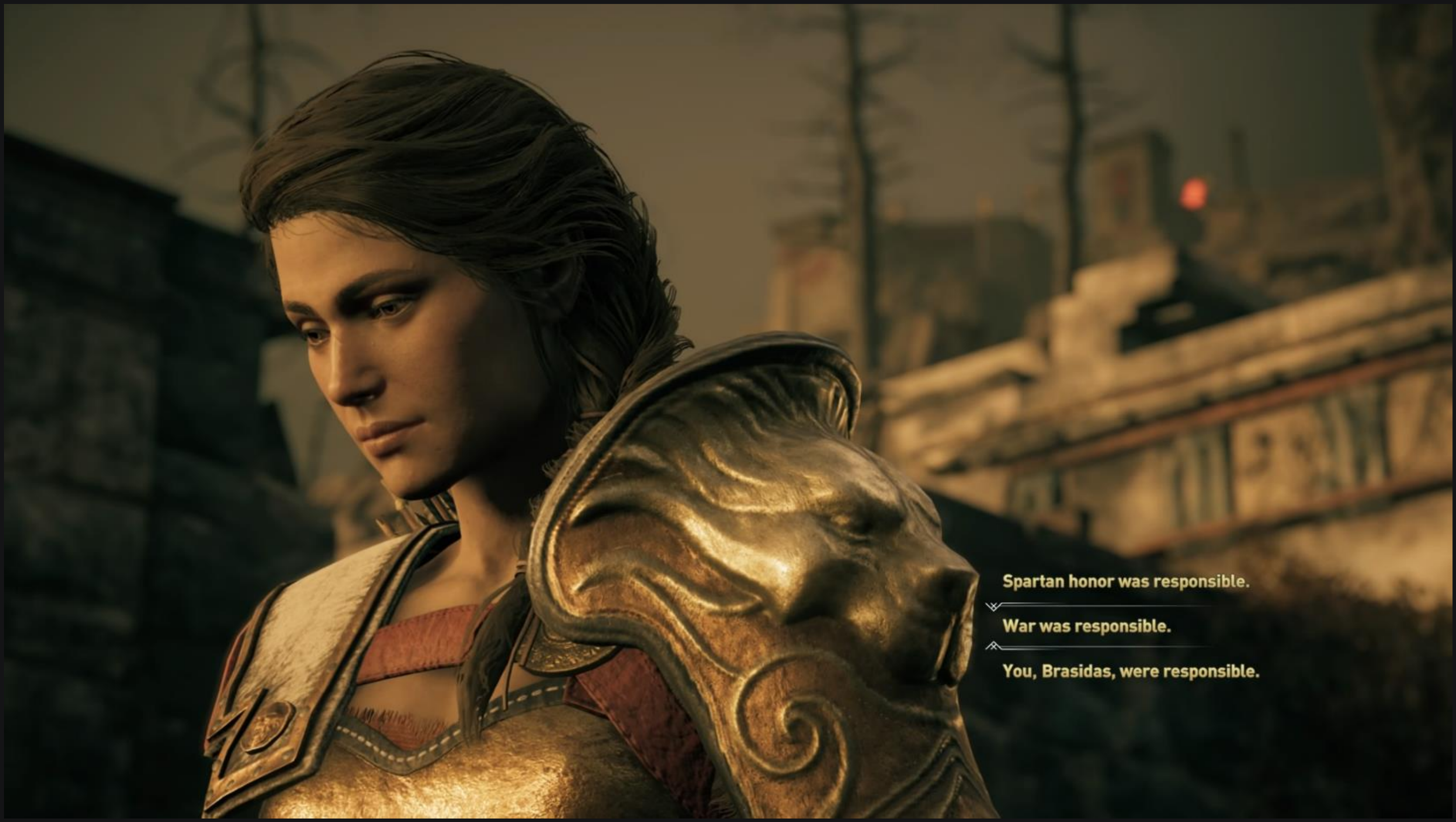
How is this implemented? Just like the inventory system, we need a way to keep track of the progress of the player's skills. This is usually done through a skill tree. Learning or acquiring a particular skill in the tree will unlock more powerful skills and give the player the ability to utilise the skills in the game.



An example of a skill tree system. Game featured: The Witcher 3

Story and narrative. The premise of most role-playing games tasks the player with saving the world, or whichever level of society is threatened. There are often twists and turns as the story progresses, such as the surprise appearance of estranged relatives, or enemies who become friends or vice versa. The game world tends to be set in a historical, fantasy, or science fiction universe, which allows players to do things they cannot do in real life, and helps players suspend their disbelief about the rapid character growth.

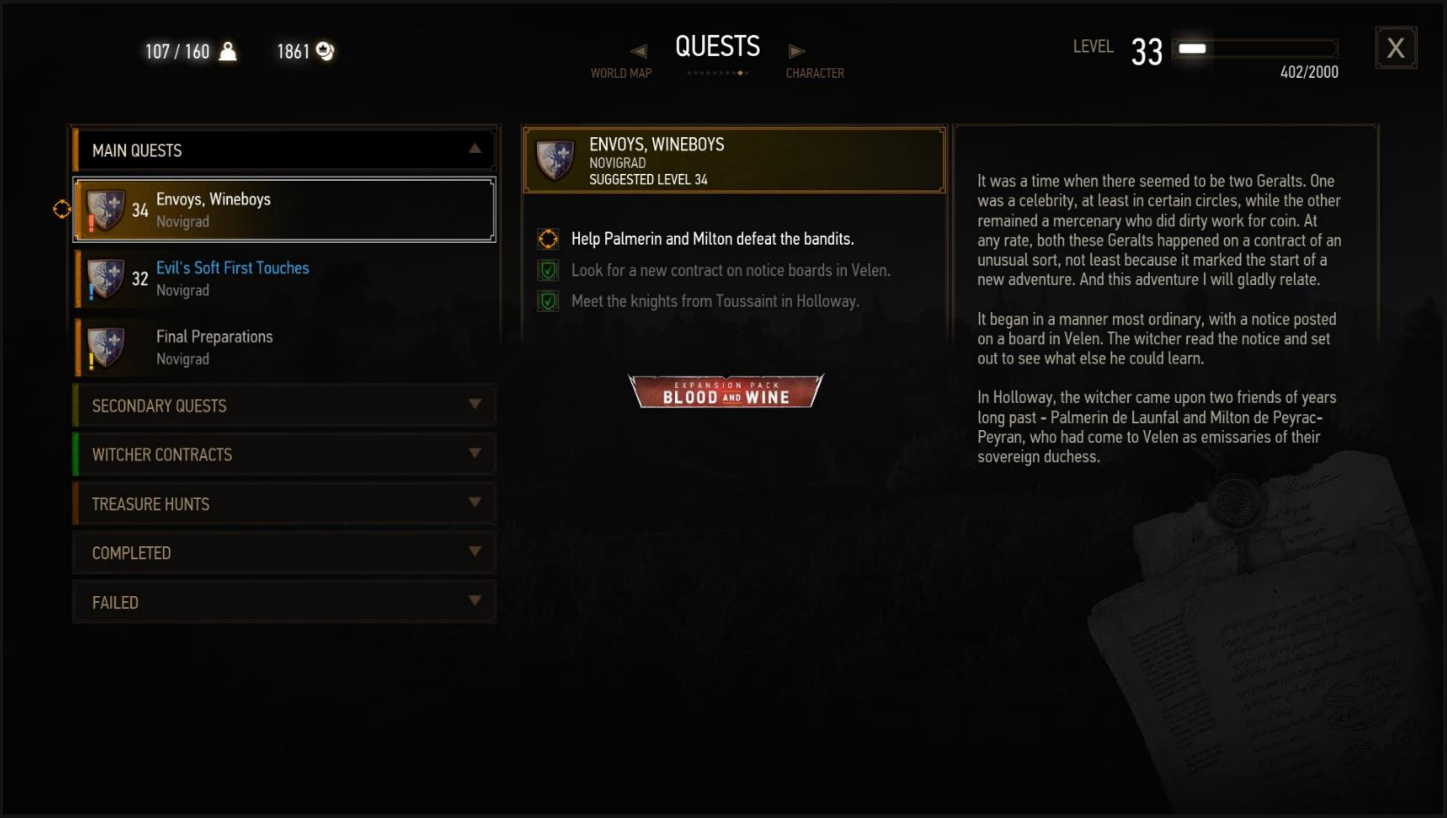
As stated previously, RPGs are heavily invested in storytelling. This is one of the main key entertainment factors of the genre. Due to this fact, when you are developing your RPG, you close attention needs to be paid on development of the story and the characters that are within the story. This, in turn, translates into the kind of environments and settings and characters within the game. Traditionally, RPGs progress the plot based on decisions that the player character makes during gameplay. This puts a great deal of pressure on the game designer, who needs to be able to implement it in the gameplay with the main storyline of the game. This also raises the issue of how to program the game to take into consideration of all the different paths within the story. To make the game more interesting and attractive, the game designer can introduce special triggers within the story to make it more interesting or challenging. This is usually done by introducing new characters and/or areas to discover within an existing level


Spartan honor was responsible.
War was responsible.
You, Brasidas, were responsible.

The yellow text on the right-hand side is the choices the player can make. Game Featured: Assassin's Creed Odyssey

Interactive environment. The whole idea behind an RPG is the ability of the player to have the freedom to explore the world in which they have been immersed. The more well defined the world is, the more interesting it will be for the player to explore and in return retain their curiosity and engagement throughout the gameplay.

This is achieved by the narrative of the story developed for the RPG. Players will be specifically given the opportunity to walk around the world and explore their surroundings in order to meet their objectives. In an open-world RPG, the player is free to roam in the world event after they have met their objective, set by the storyline. In such cases, the player can still explore any area which is no longer needed for the continuation of the quest, but they can spend time exploring the area and maybe meet some other non-player characters that they hadn't previously met while completing their mission. But this is not done by the player; once they meet their goal, they are eager to move on to the next quest. Historically, the player follows a linear sequence of quests in order to realise their goals and objectives within the game. To make the game more engaging, the developer can introduce mini quests within the main plot of the game at that particular location, to give the player the ability to explore and gain more skills and/or abilities. Since these are not part of the main storyline, they can be triggered anytime a player enters a specific area. For instance, when the player has completed the main objective of the level and is ready to move on to the next objective. In an open-world environment where the user can revisit the world anytime they choose, if the player decides to go back and explore a certain area of the world they just completed, they can trigger the event to launch this mini-quest. However, those mini quests should not affect the main storyline, but they can be used to enhance the player experience. Quests may involve defeating one or many enemies, rescuing a non-player character, item-fetch quests, or location puzzles, such as mysteriously locked doors.



Help Palmerin and Milton defeat the bandits.

Look for a new contract on notice boards in Velen.

Meet the knights from Toussaint in Holloway.

It was a time when there seemed to be two Geralts. One was a celebrity, at least in certain circles, while the other remained a mercenary who did dirty work for coin. At any rate, both these Geralts happened on a contract of an unusual sort, not least because it marked the start of a new adventure. And this adventure I will gladly relate.

It began in a manner most ordinary, with a notice posted on a board in Velen. The witcher read the notice and set out to see what else he could learn.

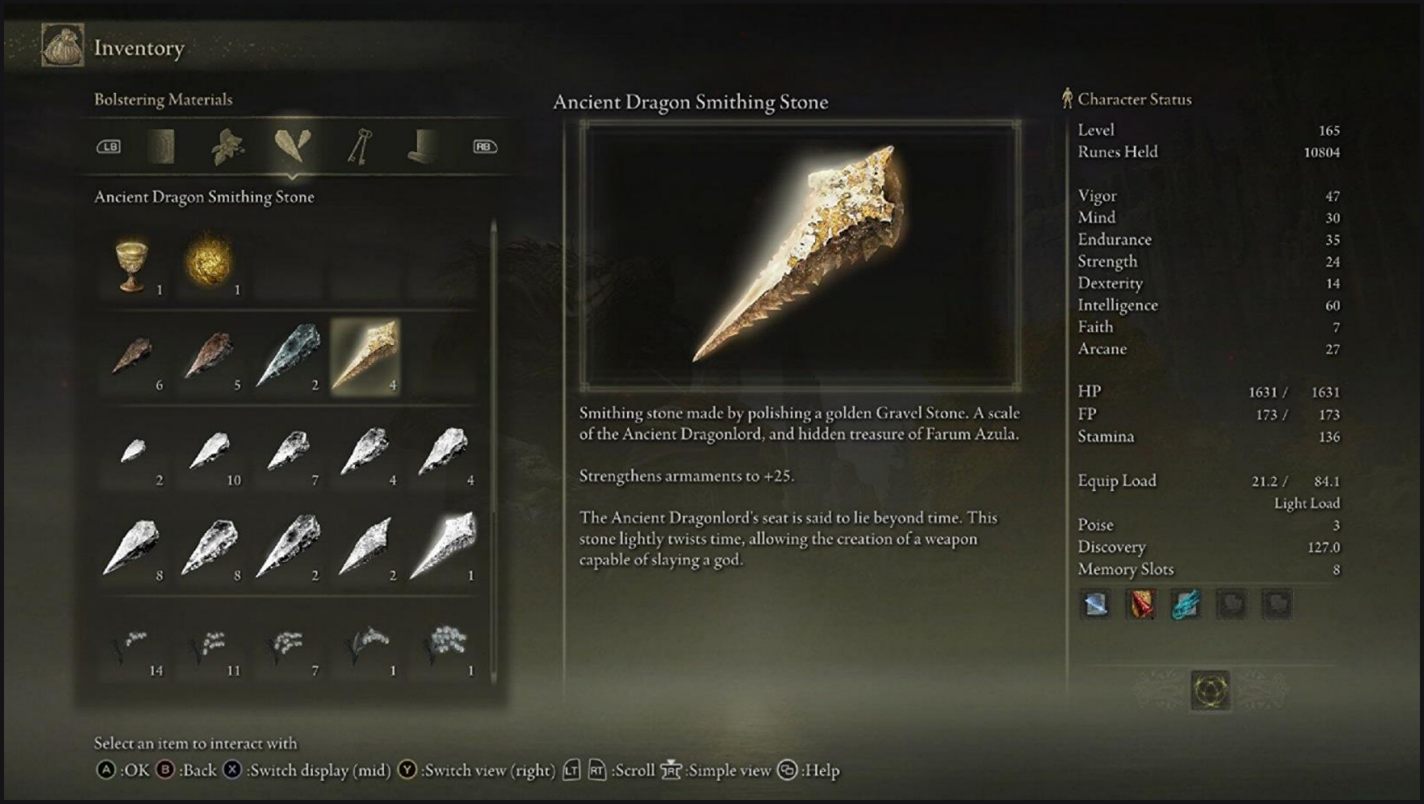
In Holloway, the witcher came upon two friends of years long past - Palmerin de Launfal and Milton de Peyrac-Peyran, who had come to Velen as emissaries of their sovereign duchess.

Game featured: The Witcher 3

<u>Items and inventory.</u> One of the main functions and features of an RPG is the inventory system. Throughout the game, the user will come across a vast number of collectible items that can be used for different purposes within the game to help them progress through the journey. Therefore, RPGs need to provide a mechanism to help the player store, retrieve, and organize the content relevant to their journey.

As the player progresses throughout their journey in an RPG, they interact with the world they are immersed in. The storyline of the game usually forces the player to interact with the surrounding world and other non-player characters. These interactions are usually in the form of some sort of an exchange. Whether this exchange is done through narration, to provide the player with a better sense of the quest, or real exchange, in terms of items, is up to the game designers and developers.

The game needs a way to keep track of all the interactions between the player character and everything and everyone else. One system that is used to keep track of these interactions is the inventory system.



Inventory

Bolstering Materials

Ancient Dragon Smithing Stone

Ancient Dragon Smithing Stone

Smithing stone made by polishing a golden Gravel Stone. A scale of the Ancient Dragonlord, and hidden treasure of Farum Azula.

Strengthens armaments to +25.

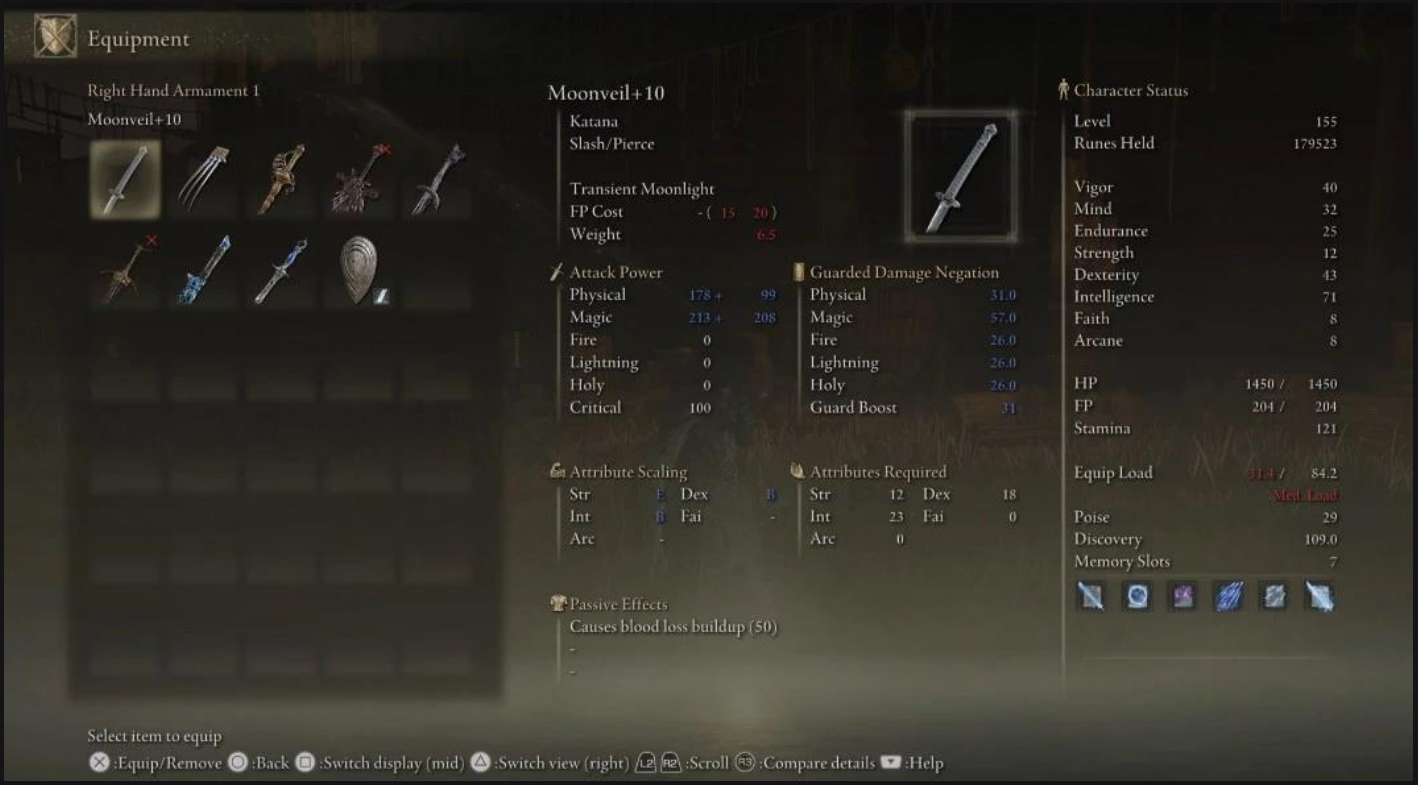
The Ancient Dragonlord's seat is said to lie beyond time. This stone lightly twists time, allowing the creation of a weapon capable of slaying a god.

Character Status

Level	165
Runes Held	10804
Vigor	47
Mind	30
Endurance	35
Strength	24
Dexterity	14
Intelligence	60
Faith	7
Arcane	27
HP	1631 / 1631
FP	173 / 173
Stamina	136
Equip Load	21.2 / 84.1 Light Load
Poise	3
Discovery	127.0
Memory Slots	8

Select an item to interact with

Ⓐ:OK Ⓑ:Back ⓧ:Switch display (mid) Ⓨ:Switch view (right) LT RT :Scroll :Simple view :Help



Equipment

Right Hand Armament 1

Moonveil+10

Moonveil+10

Katana

Slash/Pierce

Transient Moonlight

FP Cost -(15 20)

Weight 6.5

Attack Power			Guarded Damage Negation	
Physical	178 +	99	Physical	31.0
Magic	213 +	208	Magic	57.0
Fire	0		Fire	26.0
Lightning	0		Lightning	26.0
Holy	0		Holy	26.0
Critical	100		Guard Boost	31

Attribute Scaling				Attributes Required			
Str	E	Dex	B	Str	12	Dex	18
Int	B	Fai	-	Int	23	Fai	0
Arc	-			Arc	0		

Passive Effects

Causes blood loss buildup (50)

-

-

Character Status

Level	155
Runes Held	179523
Vigor	40
Mind	32
Endurance	25
Strength	12
Dexterity	43
Intelligence	71
Faith	8
Arcane	8
HP	1450 / 1450
FP	204 / 204
Stamina	121
Equip Load	31.4 / 84.2 Med. Load
Poise	29
Discovery	109.0
Memory Slots	7

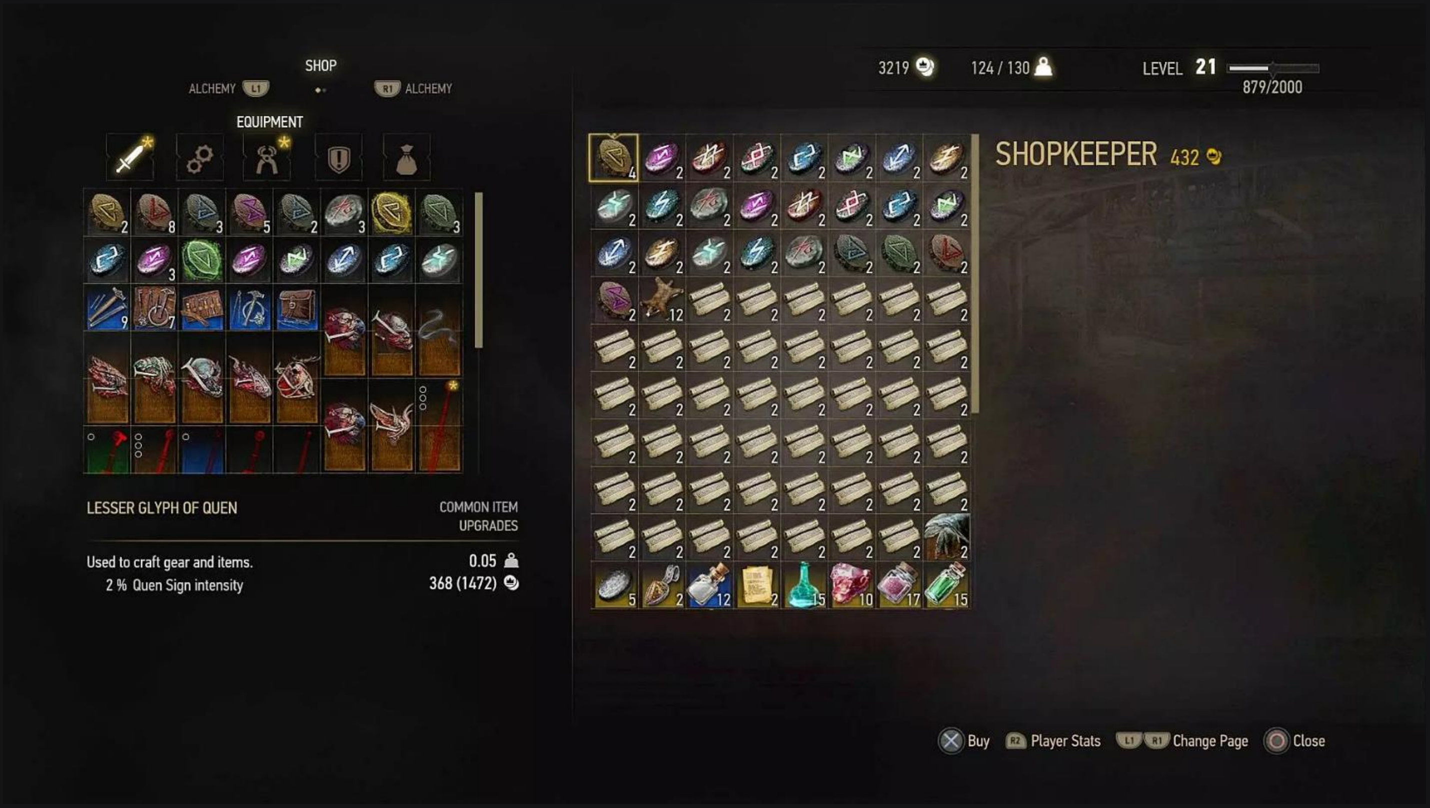
Select item to equip

ⓧ:Equip/Remove ◯:Back ▢:Switch display (mid) △:Switch view (right) L2 R2 :Scroll R3 :Compare details :Help

Game featured: Elden Ring

During the gameplay, players usually start off as a quite simple character and part of the gameplay is to elevate their character by exploring the world and collecting items that will help them increase their skills and abilities.

For instance, a player might start their journey with basic clothes. Throughout the quest, they will either interact with an NPC, such as a merchant who will provide them with a better set of clothes, and/or some sort of a weapon to get them started.



Game featured: The Witcher 3

The simplicity or complexity of the inventory system will be defined by the complexity of the game and the complexity of the characters within the game.

<u>Examples of items in the inventory include:</u>

- Weapons (swords, axes, guns, bows etc.)
- Armour and clothing
- Potions
- Special items (can be quest related)

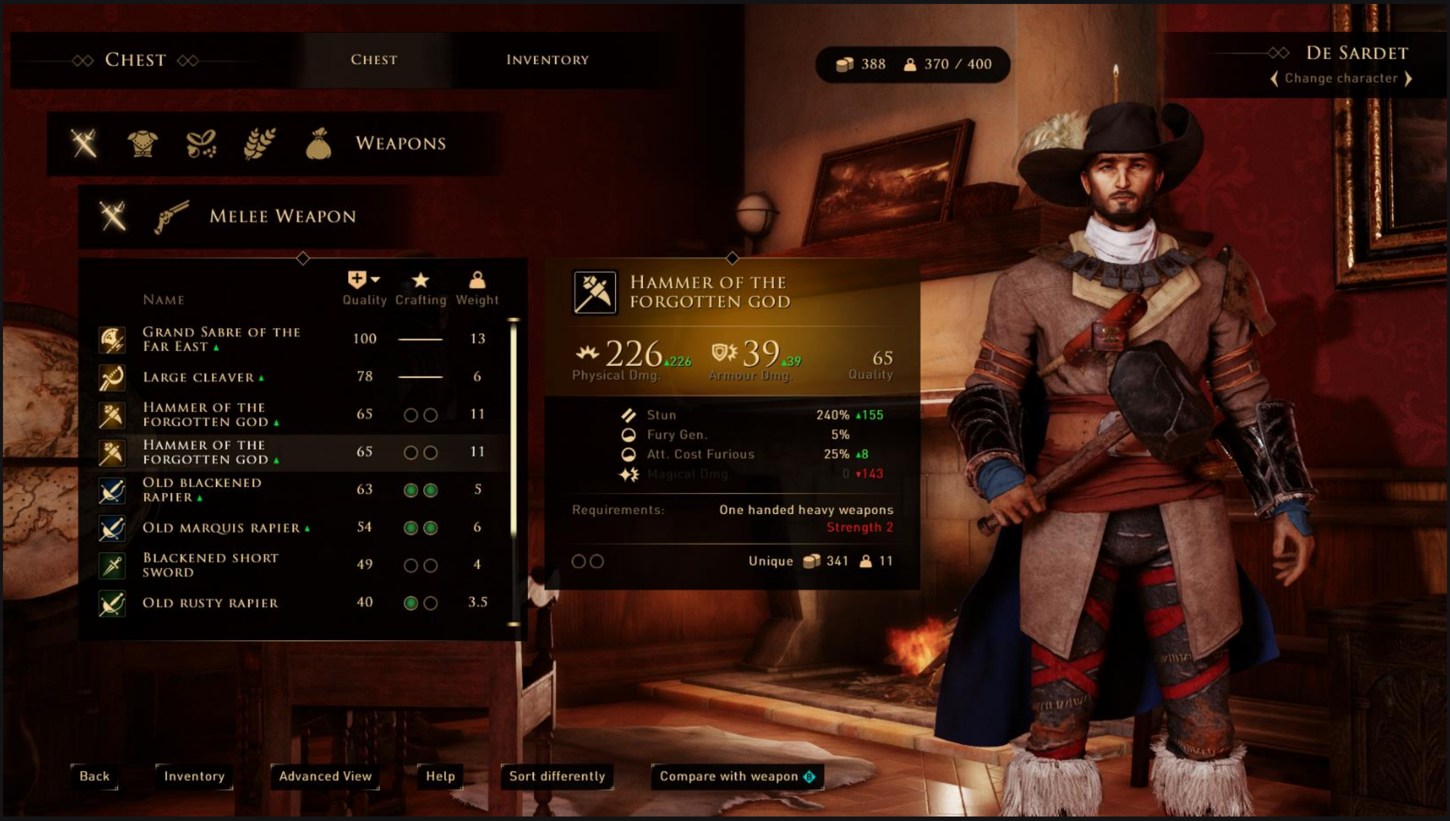
Some of the items are collected or discovered by world exploration, and some of the items are specifically traded throughout the game. If you are setting up a trade system in a game, then you will need to supply the mechanics for the trade. A trade usually takes place while interacting with a non-player character, usually a merchant, and it will use a special window to enable the interaction of the trade to take place.

There is usually a cost associated with any trade. In general, there is a cost associated with everything the player does within the game, and the cost usually either increases the player character's ability and/or experience or decreases it. This can get complex if you dig deeply into it.

The main point to keep in mind is that everything that the player will need to collect and/or manage will be done through the inventory system.

One other element that can be used to enhance the gameplay for the player, and push them to strategize their quest, is limiting the number of items they can carry in their inventory. For instance, in real life, a warrior will have limited ability to carry several types of weapons.

Assuming a warrior can carry a maximum of five different types of weapons at any given time in real life. In the game world, there might be 20 different types of weapons and instead of allowing the player to carry each weapon, so the game, for example, allows the maximum weight capacity of 10



Name	Quality	Crafting	Weight
Grand Sabre of the Far East ▲	100	—	13
Large cleaver ▲	78	—	6
Hammer of the forgotten god ▲	65	○○	11
Hammer of the forgotten god ▲	65	○○	11
Old blackened rapier ▲	63	●●	5
Old marquis rapier ▲	54	●●	6
Blackened short sword	49	○○	4
Old rusty rapier	40	●○	3.5

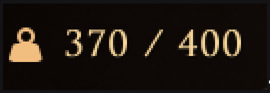
Hammer of the forgotten god

226 ▲226 Physical Dmg. 39 ▲39 Armour Dmg. 65 Quality

Stun	240% ▲155
Fury Gen.	5%
Att. Cost Furious	25% ▲8
Magical Dmg.	0 ▼143

Requirements: One handed heavy weapons Strength 2

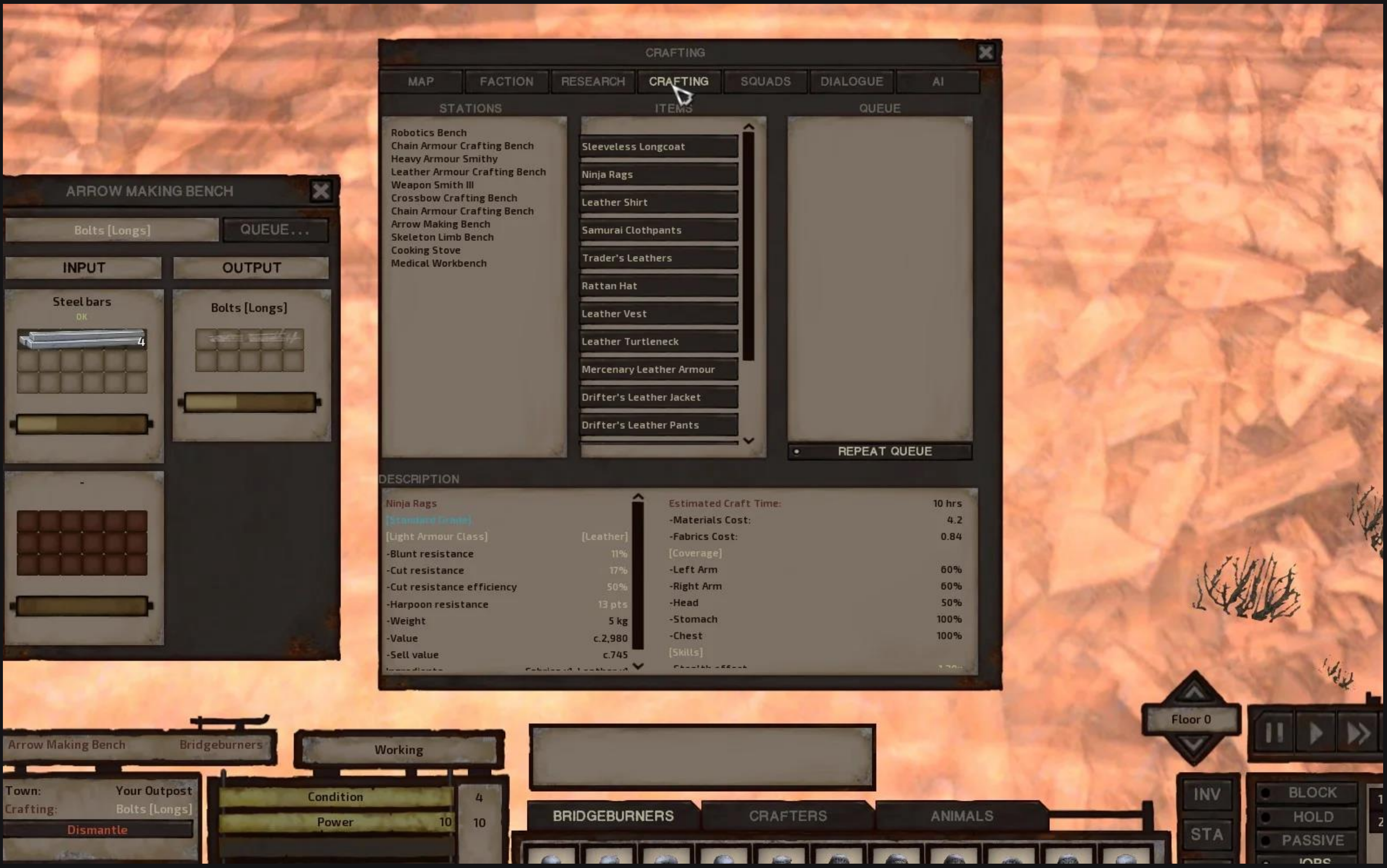
○○ Unique 341 11


370 / 400

- *Weight Limit*

Game featured: GreedFall

Additionally, some RPGs can include a crafting system. The player can create potions, weapons, armour and other special items. The items are usually crafted from resources gathered when exploring the world and/or completing quests.



Ninja Rags
[Standard Grade]
[Light Armour Class] [Leather]
-Blunt resistance 11%
-Cut resistance 17%
-Cut resistance efficiency 50%
-Harpoon resistance 13 pts
-Weight 5 kg
-Value c.2,980
-Sell value c.745

Estimated Craft Time: 10 hrs
-Materials Cost: 4.2
-Fabrics Cost: 0.84
[Coverage]
-Left Arm 60%
-Right Arm 60%
-Head 50%
-Stomach 100%
-Chest 100%
[Skills]

Game Featured: Kenshi

Game combat. Historically, there were three basic types of RPG combat system. The type of combat system will have a significant impact on the game play, as well as the implementation of the game.

The three types of combat system are as follows:

- Traditional turn-based system
- Real-time combat
- Real-time with pause

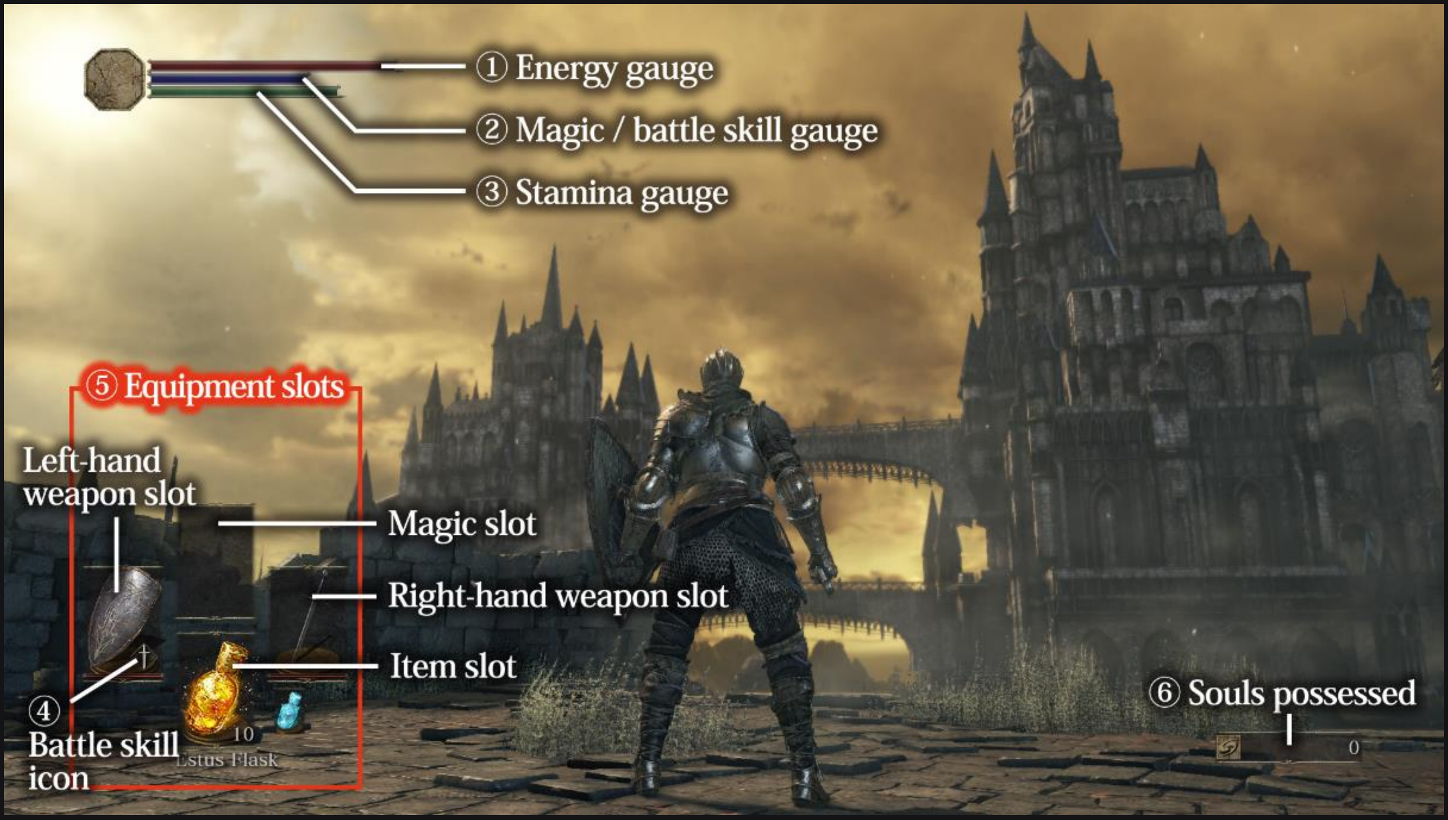
Historically, role-playing games used to implement turn-based combat systems. This type of combat system is as follows: only one character could act at a given time. During this time, all the other characters had to remain still. In other words, they could not take any action. This type of combat system is designed to put more emphasis on rewarding strategic planning.

The next type is the real-time with pause combat system. This type of combat system is also strictly turn-based, however, if the player waits more than a certain period of time to make a move or issue a command, the game will automatically pass on the command to the other player. This will allow the other player, that is, the enemy, to take a turn and attack the player.

Lastly, real-time combat imports features from action games and creates a hybrid action RPG game genre. Action RPG combat systems combine the RPG mechanics of role-playing with the direct, reflex-oriented, arcade-style, real-time combat systems of action games, instead of the more traditional battle systems of RPGs.

User Interactions and Graphics.

Most RPGs use third-person camera for the presentation of the game world. As role-playing games require the player to manage a large amount of information and frequently make use of windowed interfaces to arrange the data for the player. This is usually designed and implemented through a Heads-Up Display (HUD).



Game featured: Dark Souls 3

HUD is frequently used to simultaneously display several pieces of information, including the main character's health, items, and indication of game progression.

The design of the HUD is crucial for RPG games. Typically, there are a few key data elements that are likely to continuously communicate with the player throughout the game play, which are as follows:

- Health
- Energy
- Stamina
- Active weapon
- Active shield
- Special items
- Number of lives
- Access to main menu
- Access to inventory
- Access to skills
- Map

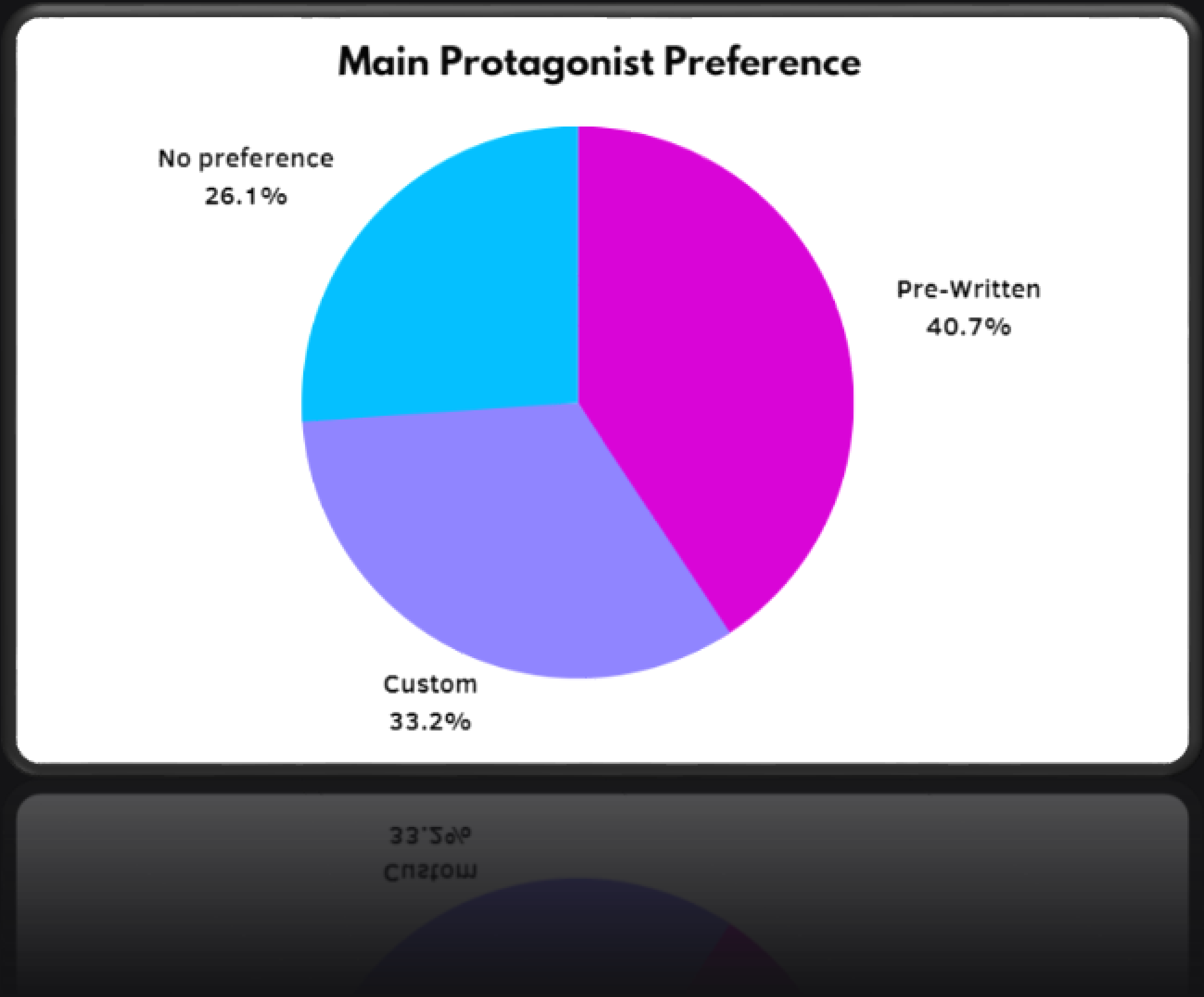
Since most RPGs collect and store substantial amounts of data for the player character, it is especially important to create an easy-to-use, yet clean HUD. HUD is that it should never overpower the screen or become a distraction.

The HUD is supposed to simplify the gameplay for the player, and not make it more confusing. Today, many games are moving away from traditional HUDs, leaning more towards cinematic or extremely simplistic experiences during the gameplay. This enables the game designer to immerse the player into the world and not to distract them with a constant, static HUD.

1.3 Audience

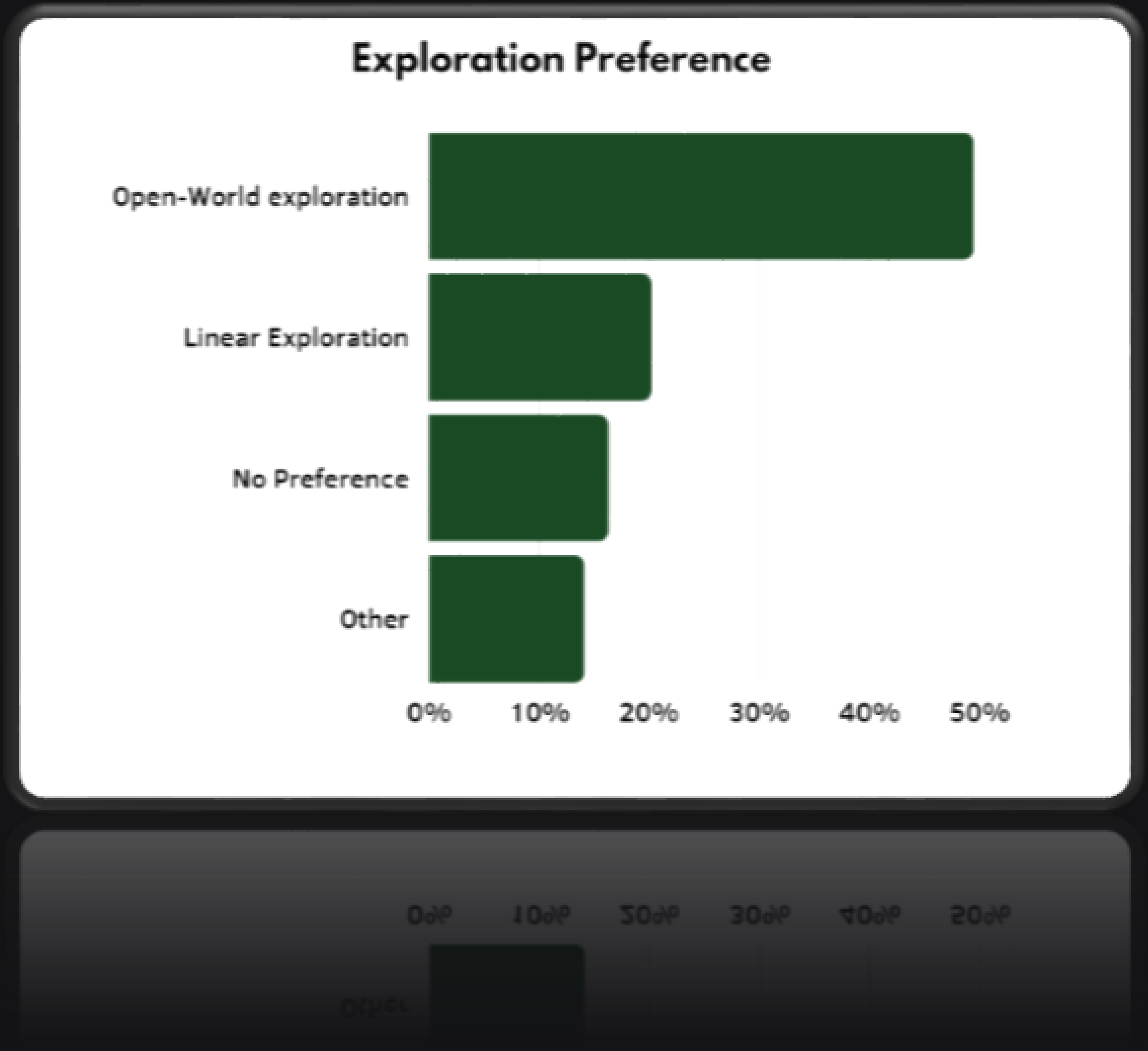
It is not possible to find specific audience for RPGs. Unlike other game genres, RPGs do not have an individual target as they cater to wide range of people. These games are designed to provide players with immersive experiences by allowing them to assume the roles of fictional characters and embark on epic adventures in virtual worlds. The diversity of characters, plotlines, and settings in RPGs ensures that there is something for everyone, making them an attractive choice for gamers of all ages, backgrounds, and interests. Therefore, there is no particular audience for which Neo's Enchanting Adventures is made. However, it can be studied what most people who usually play video games prefer to be implemented.

Source of the survey: Ganymede Games



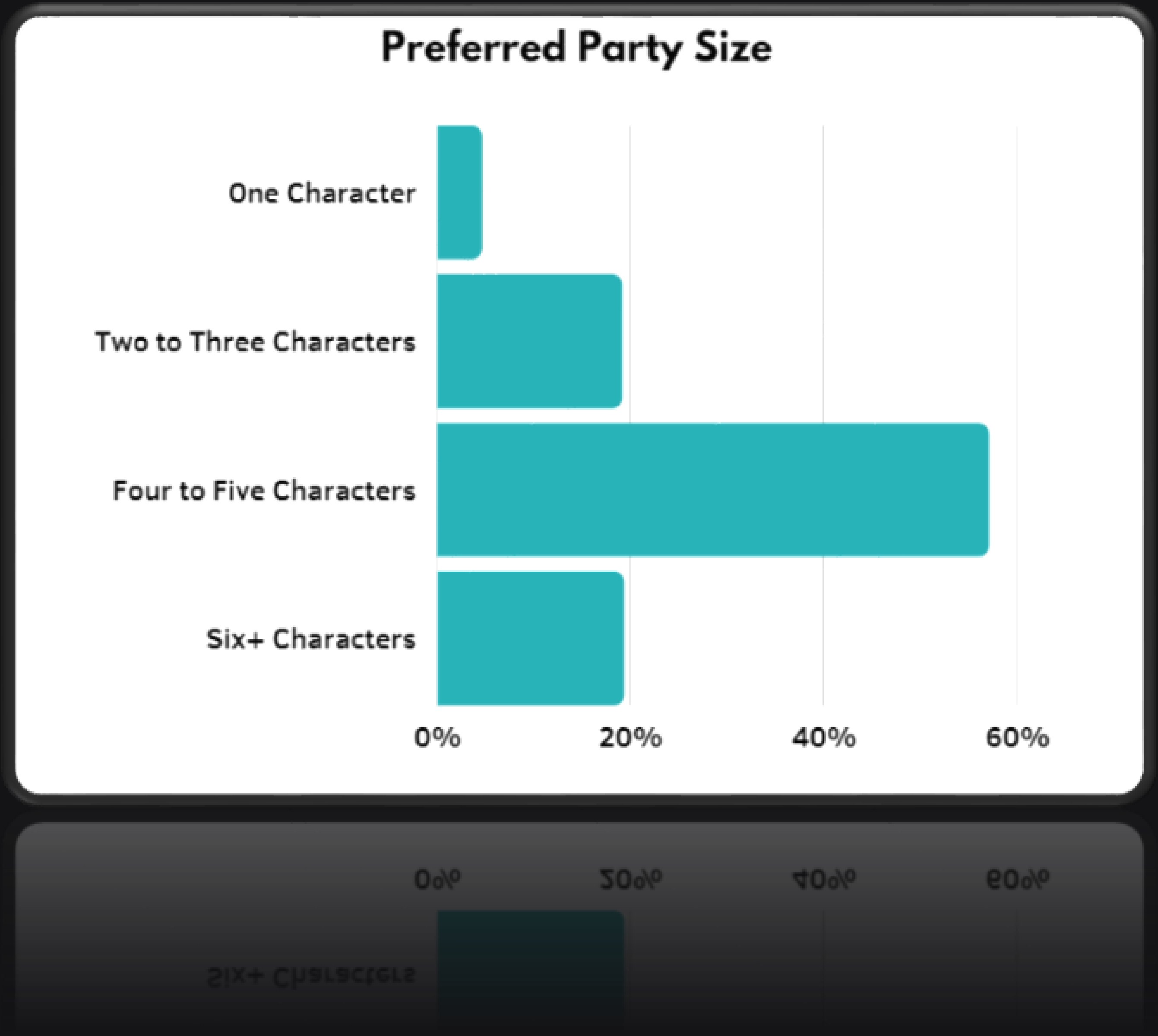
Comment:

Although, the audience seems to prefer Pre-Written Characters (by just 7%), I have decided to implement Custom Character Customisation in my game which is the second preferrable choice.



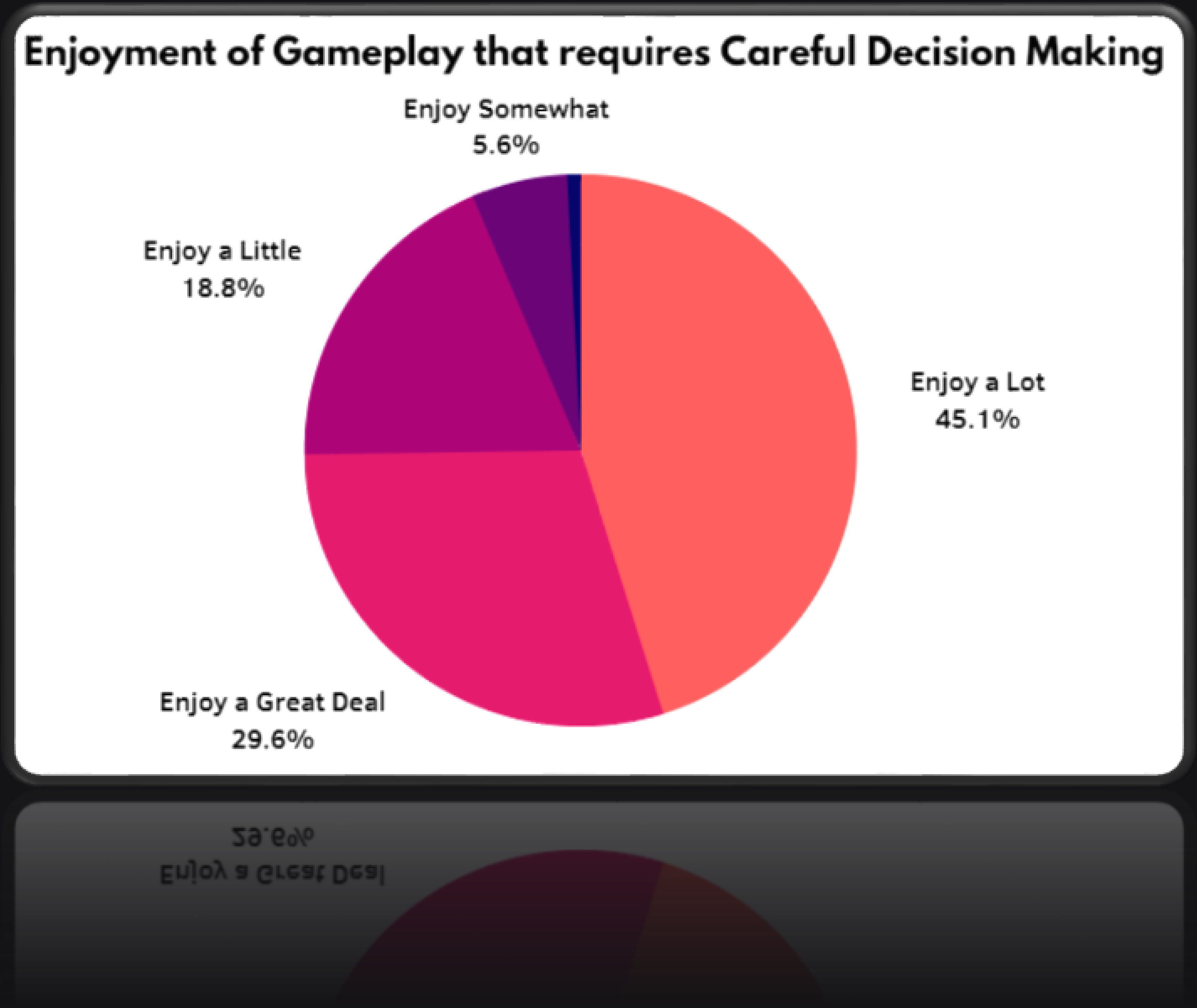
<u>Comment:</u>

My game will include open-world exploration which got the majority of votes.



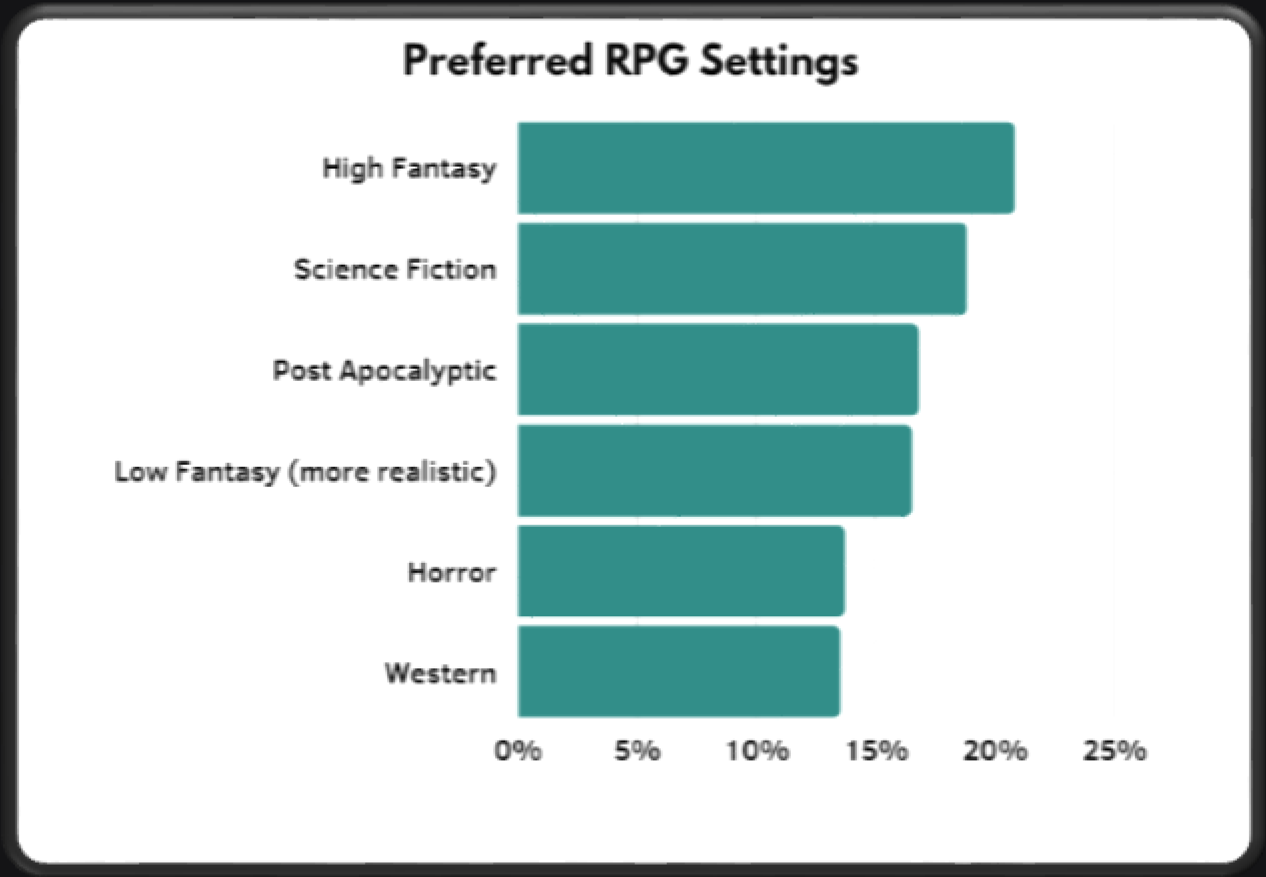
Comment:

Even though the audience prefers four to five playable characters due to the lack of resources this cannot be implemented, and my game will have only one playable character.



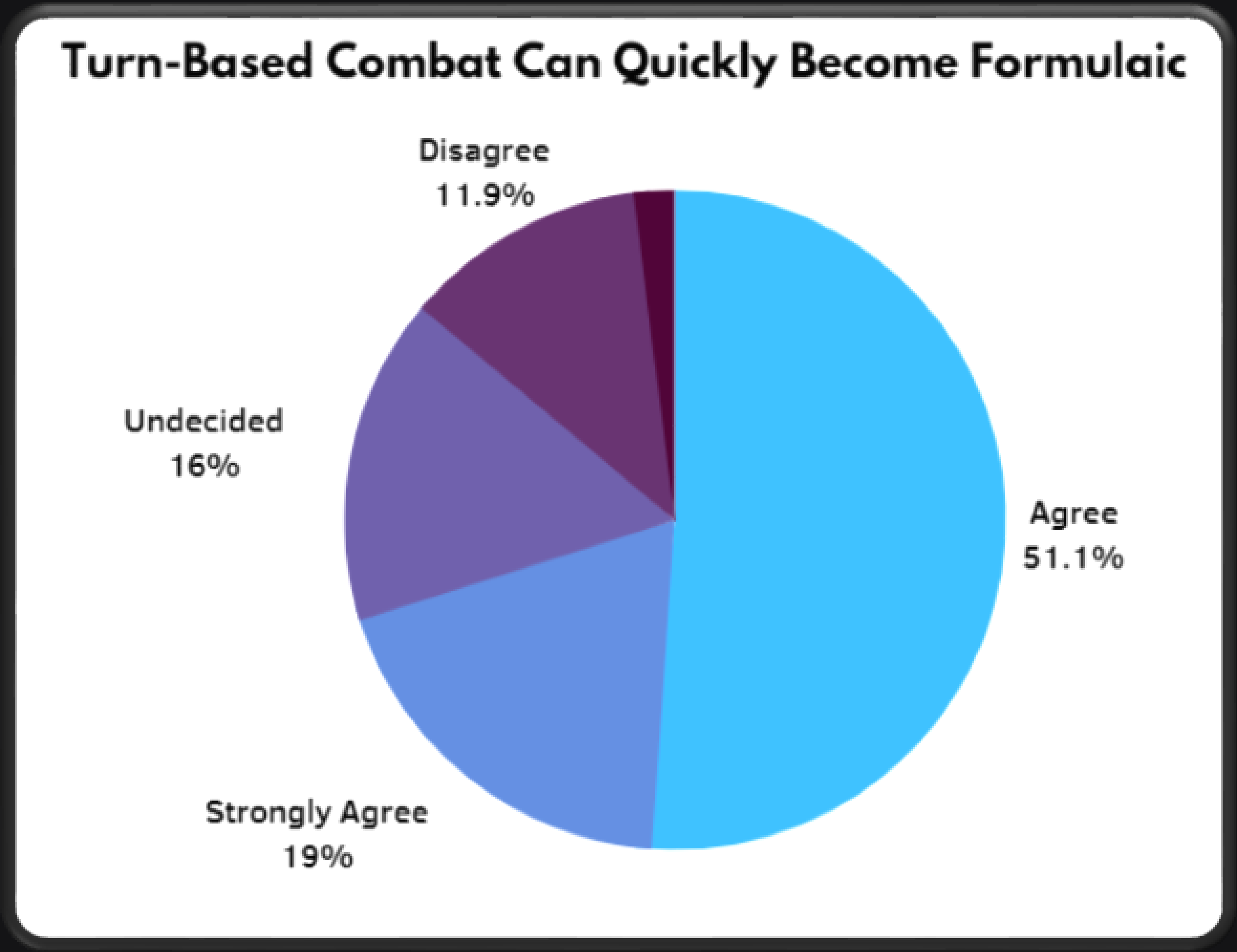
Comment:

Although, the audience enjoys gameplay which requires careful decision making due to the lack of resources this cannot be achieved and, therefore, would not be implemented.



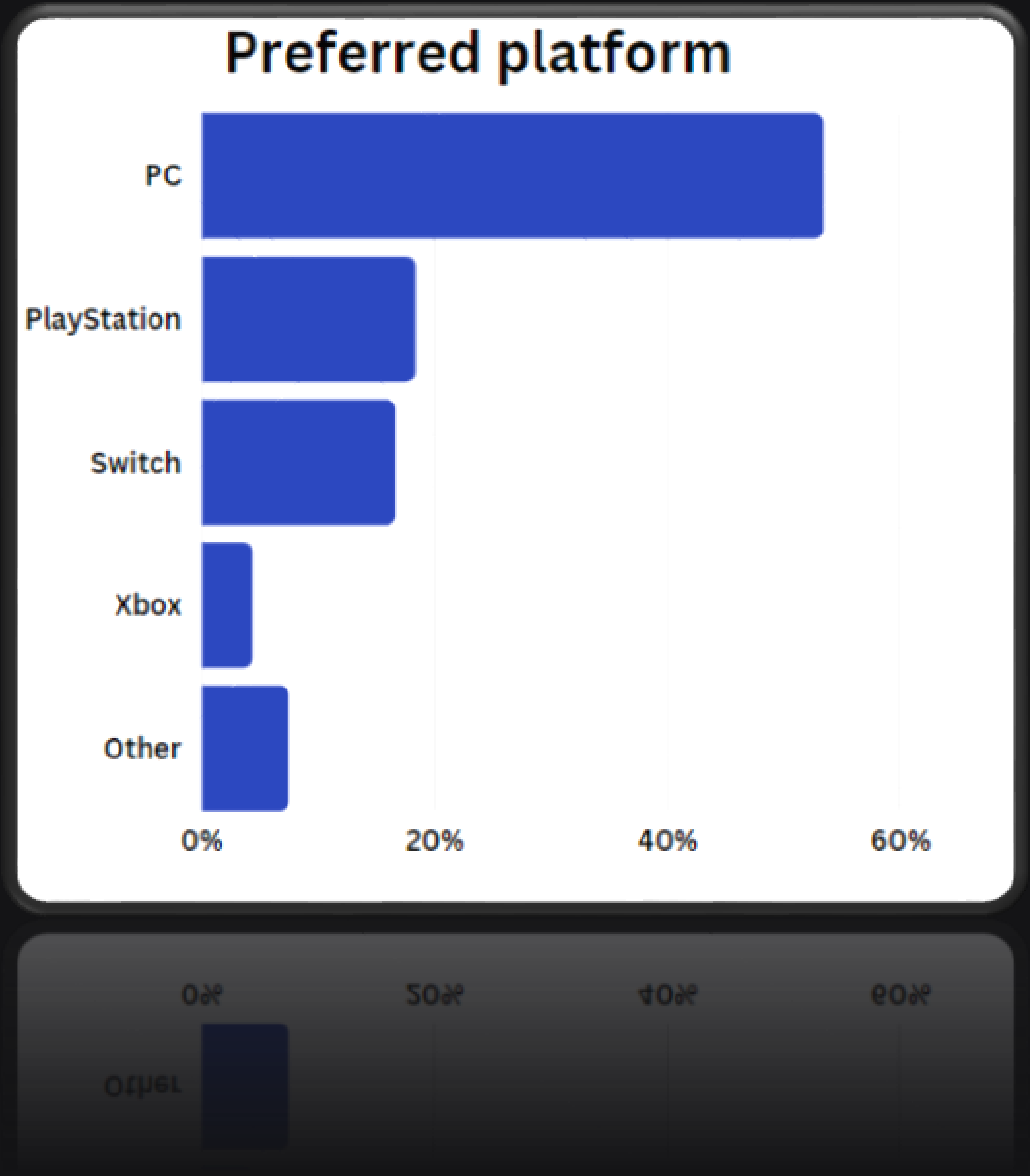
<u>Comment:</u>

As intended my game will be in High Fantasy Setting which also got the majority of votes.



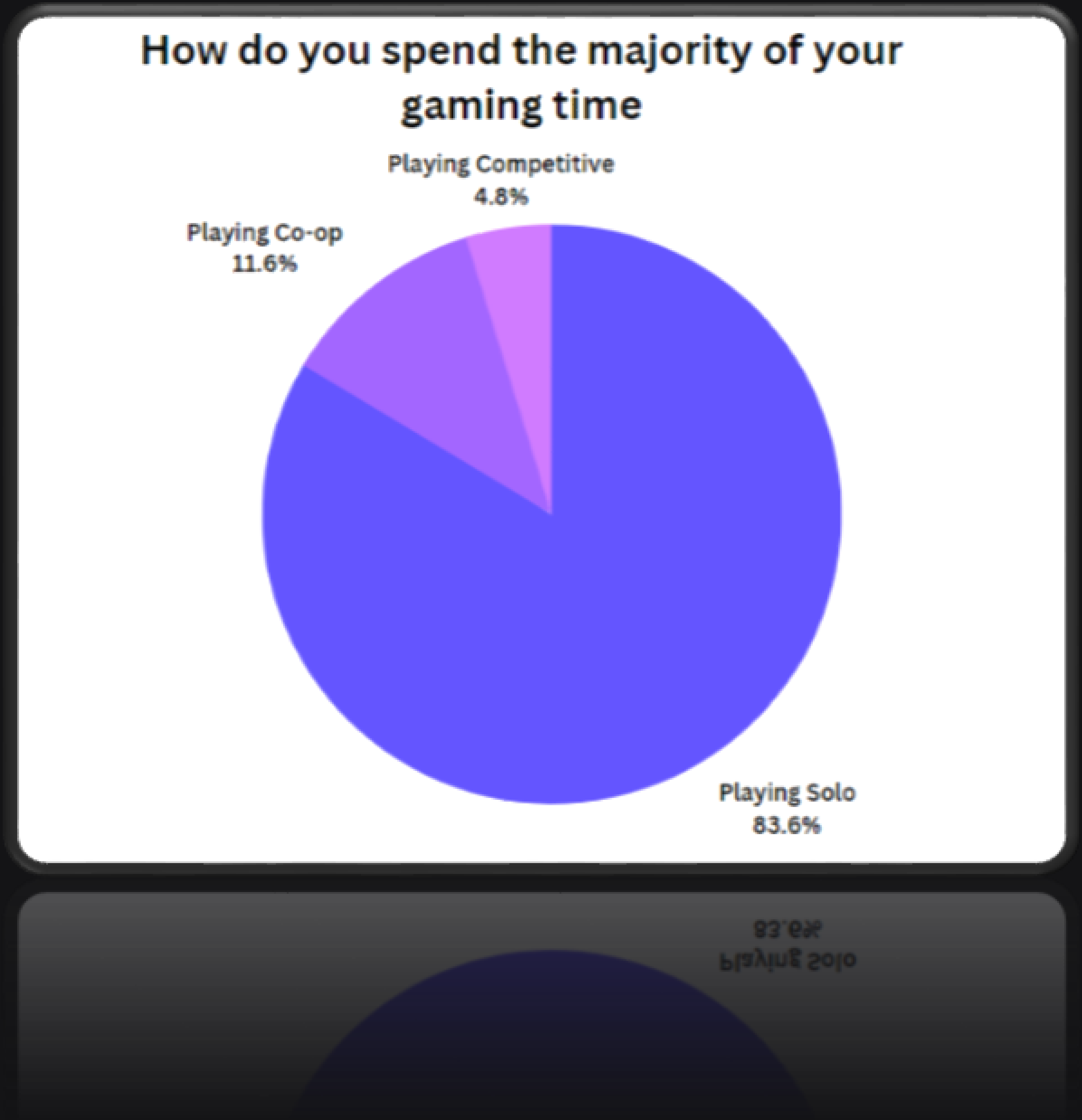
<u>Comment:</u>

Despite the fact that, the audience believes the turn-based combat to be formulaic, my goal is to provide engaging combat by using different types of attacks and dodging techniques.



Comment:

As intended, the game's platform is Desktop which also got the majority of votes.



Comment:

As intended, the game will be single-player which also got the majority of votes.

1.4 Similar Games on the Market

Legend of Zelda

The Legend of Zelda is a classic action-adventure video game series developed and published by Nintendo. The series debuted in 1986 with the release of the first game, The Legend of Zelda, for the Nintendo Entertainment System. Since then, the series has grown to include multiple titles across various Nintendo platforms and has become one of the most popular and critically acclaimed video game franchises of all time.

The gameplay of The Legend of Zelda series typically follows the same basic formula: the player controls a young hero named Link and must journey through a fantasy world filled with monsters, puzzles, and hidden secrets. The player must explore various dungeons, defeat bosses, and collect items and power-ups that will help them progress through the game.

One of the key features of The Legend of Zelda series is its open-world design, which allows players to explore and discover the game world at their own pace. The games typically feature a large overworld map, with multiple smaller areas and dungeons to explore.

Another important feature of the series is its use of puzzles, which are often integrated into the game's dungeons and must be solved in order to progress. These puzzles can take many forms, such as block puzzles, maze-like rooms, and riddles, and often require the use of items that the player has collected.

The series also features a variety of weapons and items that Link can use to defeat enemies and solve puzzles. These include a sword, shield, and bombs, as well as more unique items like a boomerang, hookshot, and magic spells.

Additionally, the game often features a wide range of NPCs (non-playable characters) that provide information, quests and help the player to progress through the game.

The story and setting of The Legend of Zelda series typically revolves around a young hero, Link, and his quest to rescue Princess Zelda and defeat the primary antagonist, Ganon. The series is known for its rich storytelling and memorable characters, as well as its epic and emotional musical score.

Different versions of the Legend of Zelda

<u>**Stardew Valley**</u>

Stardew Valley is a farming simulation role-playing game developed by ConcernedApe and published by Chucklefish. It was first released in 2016 for Windows, and later on for other platforms such as MacOS, Linux, Xbox One, PlayStation 4, Nintendo Switch, iOS and Android.

The game is set in a small farming community called Pelican Town, where the player takes on the role of a character who has inherited their grandfather's old farm. The player's goal is to restore the farm to its former glory by planting crops, raising animals, mining for resources, and interacting with the town's inhabitants.

One of the key features of the game is its farming mechanics. Players can plant a variety of crops, such as wheat, corn, and berries, and take care of them by watering, fertilizing, and harvesting them. Players can also raise animals such as cows, sheep, and chickens, and use their products to create new items.

Another important feature is the exploration and resource-gathering aspect, where players can mine in the caves, fish in the rivers and lakes, and forage in the forest to gather resources to use in crafting and upgrading their farm and equipment.

Stardew Valley also features a social aspect, where players can interact with the town's inhabitants by giving them gifts, completing tasks for them, and even marrying and starting a family.

The game features a sandbox-style open-world gameplay, where the player can progress through the game at their own pace and in their own way, and it also has a dynamic weather system and day/night cycle that affects the gameplay.

The game also features a variety of activities to engage with such as cooking, crafting, and participating in festivals that take place in the town throughout the year.

The game has a retro pixel art style, and features a calming and soothing soundtrack, which gives the game a relaxing and peaceful atmosphere.

1.5 Justify the features of the problem

Objectives		
	Primary	**Secondary**
1) Basics	1a.1 The camera must follow the player's movement 1a.2 Collision between the character and the objects in the world e.g., trees 1a.3 Graphics 1a.4 Open-World environment	1b.1 The ability to save the game 1b.2 Having multiple save slots

2) The Character	1a.1 Character Movement in 2D. Being able to move left, right, up and down using WASD 1a.2 Character Animation 1a.3 Character and enemy interaction 1a.4 Character ability to attack 1a.5 Ability to interact with the world e.g., gathering resources, talking to NPCs, collecting coins, opening chests 1a.6 Progression and levelling of the character 1a.6 The ability to die	1b.1 Character Customisation. Being able to choose a race of the character at the beginning of the game. Possible options are knight, wizard, archer 1b.2 Character extended combat abilities according to the character chosen at the beginning e.g., attack with a sword, being able to shoot magic spells 1b.3 The ability to upgrade a weapon and armour of the character by buying it from the merchant

3) UI	3a.1 Health bar, active weapon, character level, access to inventory, access to main menu and number of coins are shown as HUD 3a.2 Show a message if the character dies 3a.3 Ability to pause the game 3a.4 Shop menu implemented 3a.5 The user can change the volume of the game i.e., music and sound effects	3b.1 Game menu appears when the player opens the game 3b.2 Settings added. The user can change the volume of the music and gameplay sound. 3b. 3 Once the quest is completed, the message that the quest is completed is shown along with the experience and coins earned 3b. 4 Show mana of the wizard (if the user has chosen a wizard as the character)
4) Enemies	4a.1 Having different classes of enemies with different abilities 4a.2 Classes affect enemies attack damage and a health bar 4a.3 Enemy pathfinding AI 4a.4 Follow enemy AI 4a.5 Shoot and retreat enemy AI 4a.6 Patrol enemy AI 4a.7 Line of sight enemy AI	4b.1 Enemy Animation

5) Inventory	5a.1 Two categories: items, armour and weapons 5a.2 Change character’s weapon and armour	5b.1 Craft items from gathered resources e.g., potions and weapons 5b.2 Eat food form the items category to get health-points faster 5b.3 Items bought in the shop will be added to the player’s inventory 5b.4 The inventory of the player is saved
6) Shop	6a.1 Coin-based system to buy weapons, armour and healing potions 6a.2 Different categories of items e.g., weapons, armour and other	6b.1 Items can be sold to the merchant for coins and more space in the inventory
7) Quests	9a.1 Quest completion gives the player coins and experience 9a.2 Main story quests added	
8) Target Platform	10a.1 Desktop	-

2 Design

2.1 Overview of how the system works

The game is written in **Python** with the use of **Pygame** library, a cross-platform set of Python modules designed for video game development. It provides a set of easy-to-use functions and classes that can be used to create a wide range of games, from simple 2D games to more complex 3D games.

Pygame uses the SDL library, which stands for Simple DirectMedia Layer, to handle the low-level tasks associated with game development, such as drawing graphics to the screen, playing sounds, and handling input from the keyboard and mouse. This allows Pygame developers to focus on the high-level logic of their game, rather than worrying about the underlying technical details.

Initialisation and Modules

The pygame library is composed of a number of Python constructs, which include several different modules. These modules provide abstract access to specific hardware on your system, as well as uniform methods to work with that hardware. For example, display allows uniform access to your video display, while joystick allows abstract control of your joystick.

Displays and Surfaces

In addition to the modules, pygame also includes several Python classes, which encapsulate non-hardware dependent concepts. One of these is the Surface which, at its most basic, defines a rectangular area on which you can draw. Surface objects are used in many contexts in pygame.

The **Sprite** class in Pygame is an essential tool for game developers who are creating 2D games with sprite-based graphics. A sprite is an object in a game that has a graphical representation, such as a character, a weapon, or an enemy. Sprites are typically animated, which means that they have multiple images that are displayed in sequence to create the illusion of movement.

The **Sprite** class in Pygame provides a convenient way to manage sprites in a game. A sprite object is an instance of the **Sprite** class, which contains a surface object that represents the sprite's image. The **Sprite** class provides several methods for manipulating the sprite, such as **update()**, **draw()**, and **kill()**. These methods can be used to update the sprite's position, animate its movement, and remove it from the game world.

One of the key benefits of using the **Sprite** class in Pygame is that it provides a way to efficiently manage large numbers of sprites. Instead of creating individual surface objects for each sprite, you can create a single **Group** object that contains all of the sprites in your game. The **Group** object provides methods for updating and drawing all of the sprites at once, which can save a significant amount of processing time.

The **Sprite** class in Pygame also provides support for collision detection. You can use the **colliderect()** method of a sprite object to check if it collides with another sprite or a rectangular area on the screen. This is essential for implementing game mechanics like combat and object interaction.

Another key feature of the **Sprite** class in Pygame is its support for animation. You can create animations for your sprites using a sequence of images, and then use the **animate()** method of the **Sprite** class to play the animation. The **Sprite** class also provides a range of other methods for manipulating animations, such as **set_animation()** and **stop_animation()**.

In pygame, everything is viewed on a single user-created display, which can be a window or a full screen. The display is created using .set_mode(), which returns a Surface representing the visible part of the window. It is this Surface that you pass into drawing functions like pygame.draw.circle(), and the contents of that Surface are pushed to the display when you call pygame.display.flip().

Images and Rectangles

The image module allows to load and save images in a variety of popular formats. Images are loaded into Surface objects, which can then be manipulated and displayed in numerous ways.

Surface objects are represented by rectangles, as are many other objects in pygame, such as images and windows. Rectangles are so heavily used that there is a special Rect class just to handle them.

Some of the key functionality of the rectangle class in pygame includes:

- Creating rectangles: The **Rect** class provides a simple way to create rectangles with a specified size and position. You can create a **Rect** object using the constructor, passing in the x and y coordinates of the top left corner of the rectangle, as well as the width and height.
- Manipulating rectangles: You can change the position and size of a **Rect** object using its attributes, such as **x**, **y**, **width**, and **height**. You can also use methods like **move()** and **inflate()** to move and resize the rectangle.
- Checking for collisions: The **Rect** class provides several methods for checking if one rectangle collides with another, including **colliderect()**, **collidelist()**, and **collidelistall()**. These methods are useful for detecting collisions between game objects, such as the player and an enemy.
- Clipping surfaces: You can use the **clip()** method of a **Rect** object to create a new surface that only contains the portion of another surface that is within the bounds of the rectangle.
- Drawing rectangles: The **Rect** class provides a **draw()** method that allows you to draw a rectangle onto a surface. This is useful for drawing game objects and UI elements.

2.2 Graphics

All of the graphics map, particle effects, item attributes weapons and armour are stored in Graphics folder. To access the items quicker they are stored in multiple dictionaries in a separate file called **settings.py**.

The *enemies* dictionary defines the enemies of the game. Each enemy type is represented as a key-value pair in the dictionary, where the key is the name of the enemy, and the value is another dictionary containing various attributes of the enemy.

The attributes include:

- <u>health:</u> the amount of health points the enemy has
- <u>exp:</u> the amount of experience points the player can gain by defeating the enemy
- <u>damage:</u> the amount of damage the enemy can deal to the player
- <u>attack_type:</u> the type of attack the enemy uses (e.g. slash, bow, magic)
- <u>attack_sound:</u> the sound effect that plays when the enemy attacks
- <u>speed:</u> the movement speed of the enemy
- <u>resistance:</u> the enemy's resistance to damage
- <u>attack_radius:</u> the radius in which the enemy can attack the player
- <u>notice_radius:</u> the radius in which the enemy can detect the player
- <u>cooldown:</u> the amount of time in milliseconds between the enemy's attacks.

```
enemies = {
    'skeleton_sword':
{'health': 100, 'exp': 15, 'coins': 1, 'damage': 5, 'attack_type' : 'slash',
'attack_sound': '.\\Audio\\Socapex_ new_hits_1.wav','speed': 2, 'resistance': 4,
'attack_radius': 50, 'notice_radius': 360, 'cooldown': 500},

    'skeleton_magic':
{'health': 125, 'exp': 25, 'coins': 5, 'damage': 27, 'attack_type': 'magic',
'attack_sound': '.\\Audio\\Magic_Smite.wav','speed': 4, 'resistance': 3, 'attack_radius':
100, 'notice_radius': 360, 'cooldown': 1800},

    'mage': {'health' : 150, 'exp': 35, 'coins': 10, 'damage': 32, 'attack_type' : 'magic',
'attack_sound': '.\\Audio\\Magic_Smite.wav','speed' : 4, 'resistance' : 4, 'attack_radius':
130, 'notice_radius': 360, 'cooldown': 1500},

    'baldric':
{'health': 200, 'exp': 75, 'coins': 15,'damage': 40, 'attack_type': 'slash',
'attack_sound': '.\\Audio\\Socapex_ new_hits_2.wav','speed': 4, 'resistance': 4,
'attack_radius': 60, 'notice_radius': 360, 'cooldown': 500}
}
```

The *weapons* dictionary contains information about different weapons in a game and is used when the user chooses the player to use swords as weapons.

```
weapons = {

    'base_sword':
{'cooldown': 100, 'damage' : 25,'attack_radius': 10, 'graphic':
'.\\Graphics\\UI\\weapons\\basic_sword_scaled.png'},
    'bow': {'cooldown': 150, 'damage' : 8, 'attack_radius': 35, 'graphic':
'.\\Graphics\\UI\\weapons\\bow.png'},

    'sword_upgrade_1':
{'cooldown': 150, 'damage' : 25, 'attack_radius': 10, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_1.png'},

    'sword_upgrade_2':
{'cooldown': 200, 'damage' : 45, 'attack_radius': 15, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_2.png'},

    'sword_upgrade_3':
{'cooldown': 300, 'damage' : 55, 'attack_radius': 20, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_3.png'},
}
```

Here is what each key-value pair represents:

- cooldown: the amount of time (in milliseconds) the weapon must wait before it can be used again.
- damage: the amount of damage (in units) the weapon deals to its target.
- attack_radius: the distance (in units) from the wielder that the weapon can reach its target.
- graphic: the file path to an image representing the weapon in the user interface.

On the contrast, *magic_spells* dictionary is used when the user chooses the player to use spells as weapons. This dictionary is accessed in *MagicPlayer* class (which inherits from the *Player* class) and *UI* class to draw the graphics of the spells in the bottom-right corner of the screen.

```
magic_spells = {
    'fireball':
{'name': 'Fireball','strength': 10, 'cost': 5, 'audio': '.\\Audio\\fireball.wav',
'graphic': '.\\Graphics\\UI\\spells\\fireball.png'},

    'heal':
```

```
{'name': 'Healing','strength': 15, 'cost': 10, 'audio': '.\\Audio\\heal.wav', 'graphic':
'.\\Graphics\\UI\\spells\\heal.png'},

    'shield':
{'name': 'Shield','strength': 5, 'cost': 15, 'audio': '.\\Audio\\shield.wav', 'graphic':
'.\\Graphics\\UI\\spells\\shield.png'},

    'icicle':
{'name': 'Icicle','strength': 25, 'cost': 35, 'audio': '.\\Audio\\icicle.wav','graphic':
'.\\Graphics\\UI\\spells\\icicle.png'},

    'quake':
{'name': 'Earthquake','strength': 65, 'cost': 50, 'audio': '.\\Audio\\quake.wav','graphic':
'.\\Graphics\\UI\\spells\\quake.png'}
}
```

Here is what each attribute of the dictionary represents:

- name: the name of the spell.
- strength: the strength of the spell. This is used to calculate the amount of damage the spell will do or the amount of healing it will provide.
- cost: the cost of the spell. This is used to deduct the required amount of mana from the player's mana pool.
- audio: the file path to the audio of the spell. The audio is played once the spell is cast
- graphic: the file path to the graphic of the spell. This is used to display the graphic of the spell in the *Inventory* and *UI* classes.

The armour of the enemy is also represented in a dictionary to be quickly accessed, instead of loading it from a file.

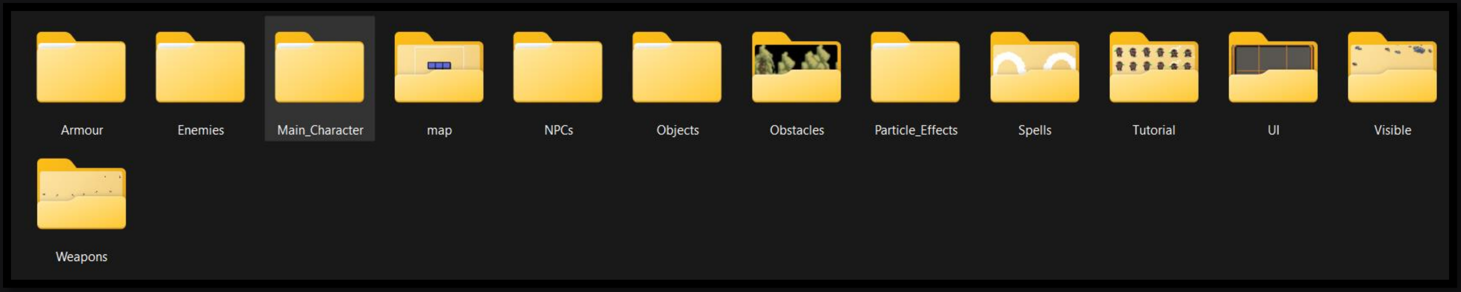
```
armour = {



    'simple_clothes':
    {
     'head': {'resistance': 0, 'graphic': None},
     'top': {'resistance': 2, 'graphic': '.\\Graphics\\Armour\\simple_clothes\\top.png'},
     'bottom': {'resistance': 2, 'graphic':
'.\\Graphics\\Armour\\simple_clothes\\bottom.png'}
    },



    'leather_armour' :
    {
     'head': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\leather_armour\\head.png'},
     'top': {'resistance': 15, 'graphic': '.\\Graphics\\Armour\\leather_armour\\top.png'},
     'bottom': {'resistance': 2, 'graphic':
'.\\Graphics\\Armour\\leather_armour\\bottom.png'}
    },

    'robe':
    {
     'head': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\robe\\head.png'},
     'top': {'resistance': 8, 'graphic': '.\\Graphics\\Armour\\robe\\top.png'},
     'bottom': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\robe\\bottom.png'}
    },


    'chain_armour':
    {
     'head': {'resistance': 15, 'graphic': '.\\Graphics\\Armour\\chain_armour\\head.png'},
     'top': {'resistance': 45, 'graphic': '.\\Graphics\\Armour\\chain_armour\\top.png'},
     'bottom': {'resistance': 15, 'graphic': '.\\Graphics\\chain_armour\\bottom.png'}
    },


    'chain_armour_robe':
    {
     'head': {'resistance': 10, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\head.png'},
     'top': {'resistance': 50, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\top.png'},
     'bottom': {'resistance': 10, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\bottom.png'}
    },


    'plate_armour':
    {
     'head': {'resistance': 55, 'graphic': '.\\Graphics\\Armour\\plate_armour\\head.png'},
     'top': {'resistance': 55, 'graphic': '.\\Graphics\\Armour\plate_armour\\top.png'},
     'bottom': {'resistance': 55, 'graphic':
'.\\Graphics\\Armour\\plate_armour\\bottom.png'}
    }
}
```

Here is what each attribute represents:

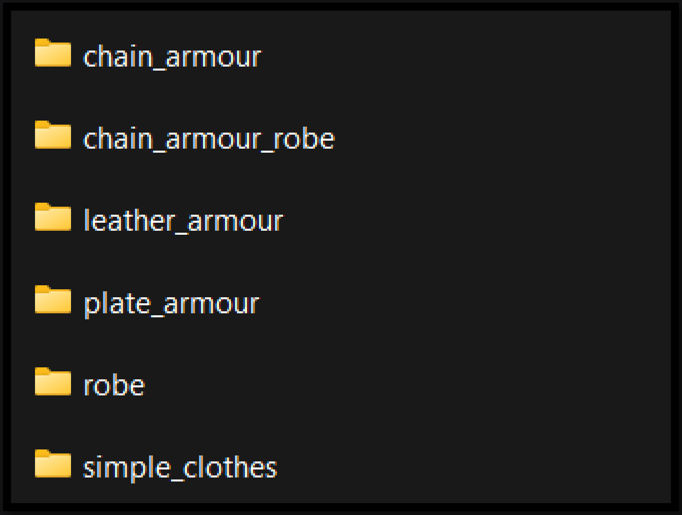
- resistance: represents the amount added to the player's health

- graphic: this is the file path to the graphic that represents each piece the armour. This is the inventory

The character animations are stored in separate folders instead of a dictionary. Each folder is named after the current armour of the character, to display it in the game once it is changed in the inventory.

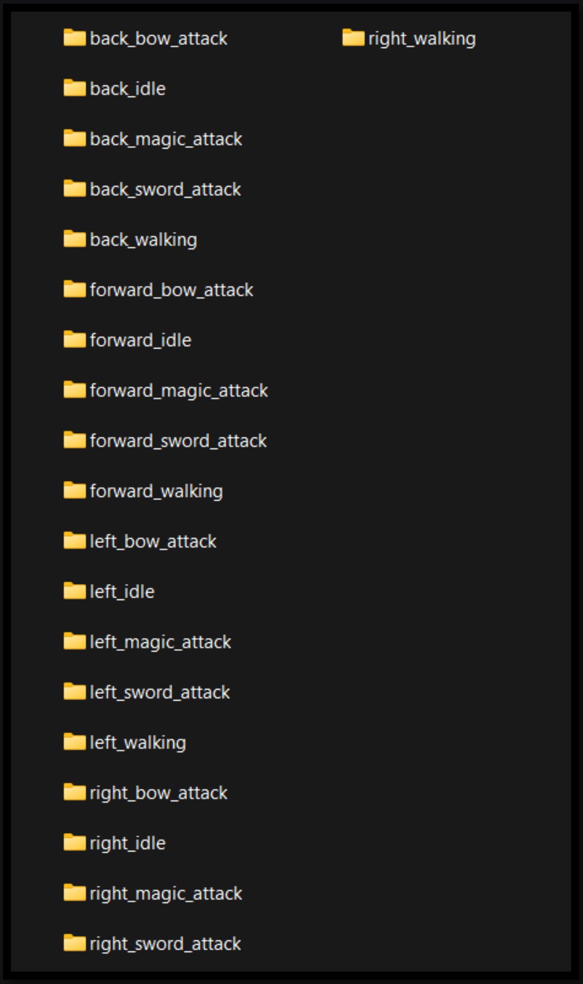


Armour
Enemies
Main_Character
map
NPCs
Objects
Obstacles
Particle_Effects
Spells
Tutorial
UI
Visible
Weapons



chain_armour
chain_armour_robe
leather_armour
plate_armour
robe
simple_clothes

Each folder contains subfolders which are named after the specific animation:



To access those animations they are called in *import_player_assets* function where up a dictionary called "animations" is set. The dictionary contains a list of animation frames for each type of animation the player character can perform. The function loops through each animation type in the "animations" dictionary and imports the animation frames by calling the *import_folder* function with the appropriate file path. The function starts by creating an empty list called "surface_list", which will be used to store all the pygame.Surface objects. It then uses the "walk" function from the "os" library to recursively walk through the directory structure of the specified folder. For each folder it encounters, the function will iterate through all the image files and load each one using the "pygame.image.load" function. The full file path is constructed by concatenating the "path" argument with the file name obtained from "walk", using the "\" character as a path separator.

```
function import_folder(path)
    surface_list = empty list

    for each folder, subfolder, and image_file in walk(path) do
        for each image in image_files do
            full_path = path + '\\' + image
            image_surface = load_image(full_path)
            add image_surface to surface_list
        end for
    end for

    return surface_list
end function
```

The *item_attributes* dictionary is used to access the attributes of the item and use it to set the attributes of the item.

```
item_attributes = {
    'weapon': {
        'base_sword': {'description': 'A basic sword', 'cost': 5, 'graphic': 
'.\\Graphics\\UI\\weapons\\basic_sword.png'},

        'sword_upgrade_1': {'description': 'Upgraded sword', 'cost': 15, 'graphic': 
'.\\Graphics\\UI\\weapons\\sword_1.png'},

        'sword_upgrade_2': {'description': 'Even more upgraded sword', 'cost': 20, 
'graphic': '.\\Graphics\\UI\\weapons\\sword_2.png'},

        'sword_upgrade_3': {'description': 'Ultimate sword upgrade', 'cost': 35, 'graphic': 
'.\\Graphics\\UI\\weapons\\sword_3.png'},
    },

    'armour': {
        'simple_clothes': {'description': 'Basic armour', 'cost': 2, 'graphic': 
'.\\Graphics\\Armour\\simple_clothes\\full_armour.png'},

        'leather_armour': {'description': 'Light and flexible armour', 'cost': 20, 
'graphic': '.\\Graphics\\Armour\\leather_armour\\full_armour.png'},

        'robe': {'description': 'A simple robe for mages', 'cost': 15, 'graphic': 
'.\\Graphics\\Armour\\robe\\full_armour.png'},

        'chain_armour': {'description': 'Heavy chain armour', 'cost': 45, 'graphic': 
'.\\Graphics\\Armour\\chain_armour\\full_armour.png'},

        'chain_armour_robe': {'description': 'Upgraded robe for mages', 'cost': 45, 
'graphic': '.\\Graphics\\Armour\\chain_armour_robe\\full_armour.png'},

        'plate_armour': {'description': 'Thick and heavy armour', 'cost': 70, 'graphic': 
'.\\Graphics\\Armour\\plate_armour\\full_armour.png'},
    }
}
```

The outermost keys represent the categories of items, which are 'weapon' and 'armour'. The values associated with these keys are themselves dictionaries, where the keys represent the names of specific items within that category (e.g., 'basic_sword' for the weapon category), and the values are dictionaries containing information about those items.

Each item's dictionary contains the following information:

- description: A string describing the item.
- cost: An integer representing the cost of the item.
- image_path: A string representing the file path to the image associated with the item to be accessed in the shop and inventory

The attributes are set in the *Item* class and the method to call those attributes is called *set_attributes* method. It accesses the dictionary *item_attributes* from **settings.py** to retrieve the attributes of the item based on its type and name. If any of the attributes are not present in the dictionary, they are set to default values.

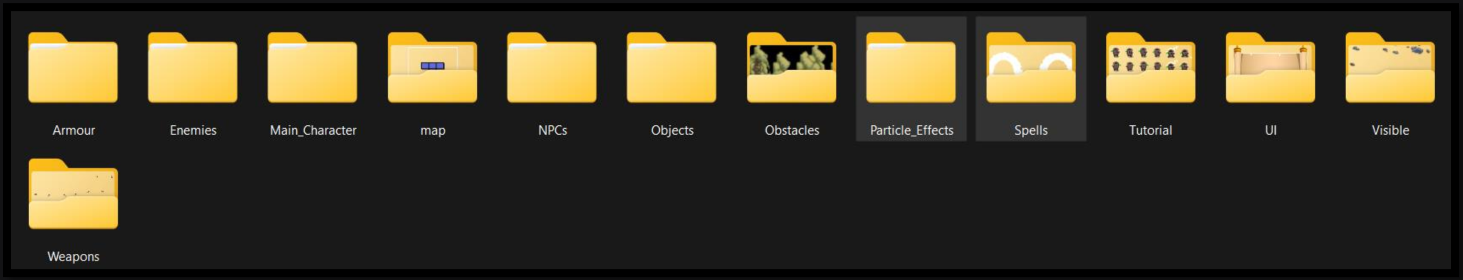
```
import settings

class Item:
    function __init__(self, name, type)
        self.name = name
        self.type = type
        self.cost = 0
        self.description = ""

        self.set_attributes()
end function



    function set_attributes(self)
        item_attributes_access = item_attributes.get(self.type, {}).get(self.name, {})
        self.description = item_attributes_access.get('description', '')
        self.cost = item_attributes_access.get('cost', null)
        self.graphic = item_attributes_access.get('graphic', null)
end function
end class
```

Additionally, setting.py also includes the constants and variables that are frequently used by other classes to minimalize repetition. For example:

```
WIDTH    = 1280
HEIGHT   = 720
FPS      = 60
TILESIZE = 32
MAIN_BACKGROUND = '.\\bg_2.png'
BACKGROUND_2 = '.\\bg.png'
MAP_SIZE = 10240
```

All of those constants are used by more than one class in the game, so by initialising them in setting.py they can be accessed quicker by other classes and methods.

The frames of the particle effects are stored in *Particle Effects* and *Spells.* Particle effects are used to animate spells the player casts and well as death particles when an enemy dies.



Tiles and character models presets:

https://opengameart.org/content/lots-of-free-2d-tiles-and-sprites-by-hyptosis

https://opengameart.org/content/whispers-of-avalon-grassland-tileset

https://opengameart.org/content/zelda-like-tilesets-and-sprites

https://opengameart.org/content/rpg-gui-construction-kit-v10

https://opengameart.org/content/lpc-medieval-fantasy-character-sprites

https://opengameart.org/content/bosses-and-monsters-spritesheets-ars-notoria

UI pre-sets:

https://opengameart.org/content/golden-ui-bigger-than-ever-edition

https://opengameart.org/content/moderna-graphical-interface

https://opengameart.org/content/fantasy-ui-elements-by-ravenmore

https://opengameart.org/content/cartoon-forest-2d-backgrounds

Audio:

https://opengameart.org/content/ambientloading-screen-game-music-pack

https://opengameart.org/content/solemn-war-music

https://opengameart.org/content/win-music-3

https://opengameart.org/content/inventory-sound-effects

https://opengameart.org/content/punches-hits-swords-and-squishes

https://opengameart.org/content/coin-sounds-0

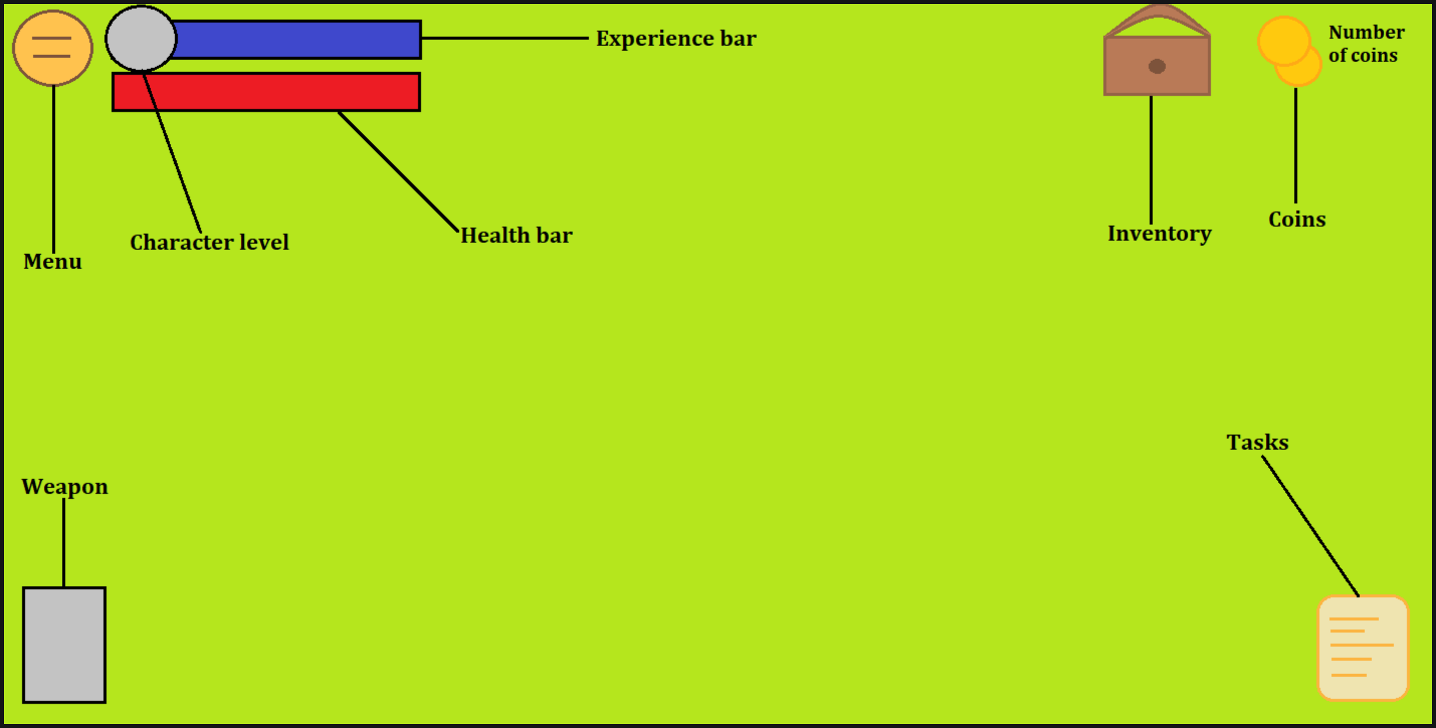
https://opengameart.org/content/magic-smite

https://opengameart.org/content/8-magic-attacks

https://opengameart.org/content/magic-shield

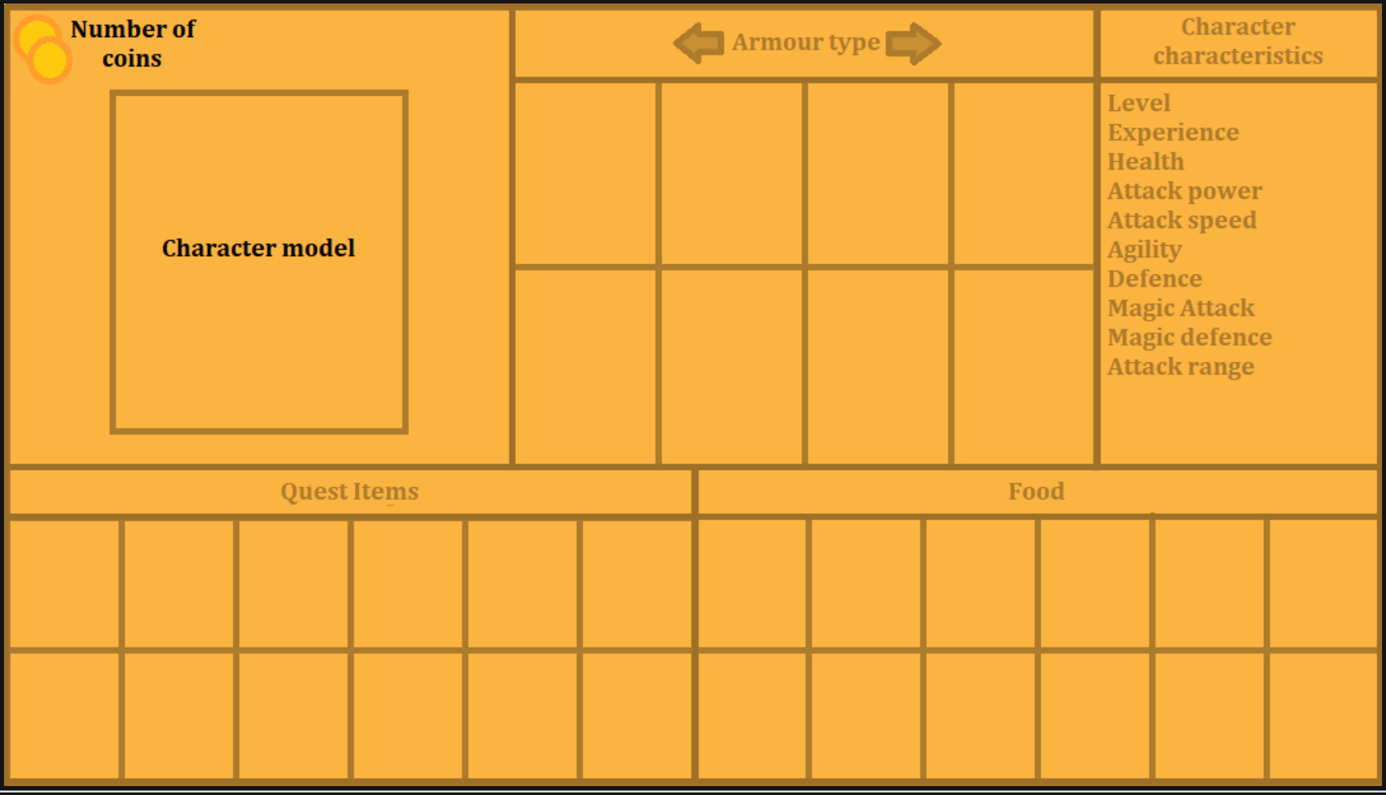
https://opengameart.org/content/3-heal-spells

HUD



An early concept of HUD

Inventory UI



An early concept of player's inventory

2.3 Map

Instead of hardcoding the map, I am going to use *Tiled*, a free open-source level editor.

Tiled is a 2D level editor that helps you develop the content of your game. Tiled is free and open-source, meaning it can used it without any cost and customise it as per your need. Its primary feature is to edit tile maps of various forms, but it also supports free image placement as well as powerful ways to annotate your level with extra information used by the game. Tiled focuses on general flexibility while trying to stay intuitive.

Tiled has a user-friendly interface that makes it easy for users to create and edit maps. The interface is intuitive and easy to navigate, which makes it an ideal choice for those who are new to map-making or those who don't have a lot of experience with game development tools. This allows you to focus on creating your game's world, instead of struggling with complex tools.

Tiled allows you to create your own custom tile sets, which can be easily adjusted to fit the theme and style of your game. This feature is particularly useful for RPG games, as it allows you to create unique and immersive worlds that are tailored to the game's story and characters. The ability to customize the tile sets also allows you to create maps that are visually consistent, which can be important for maintaining the player's immersion in the game.

Tiled supports multiple layers, which can be used for different elements of the map, such as terrain, objects, and events. This feature allows for a more organized and efficient map-making process. The ability to work with multiple layers can save a lot of time and effort when creating maps, as it allows to easily make changes to one layer without affecting the others.

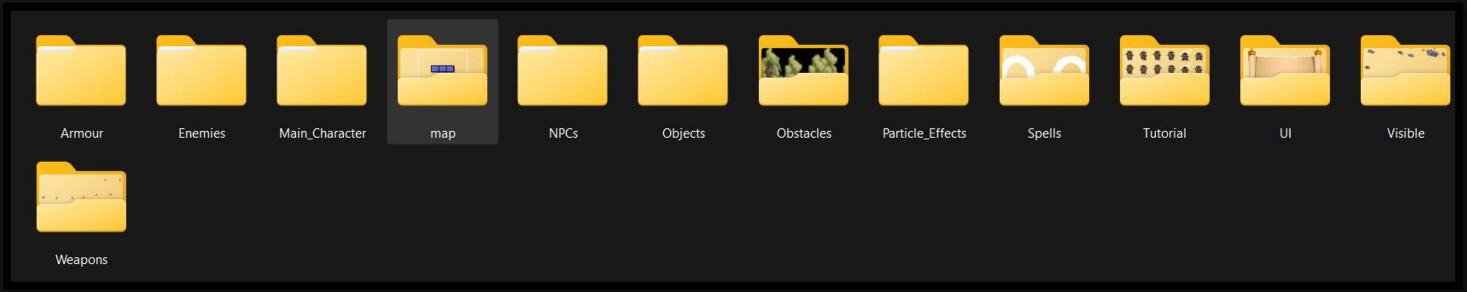
Tiled is available for Windows, MacOS, and Linux, which makes it accessible to a wide range of users. This cross-platform compatibility is essential for game developers as it allows them to work on their maps regardless of their preferred operating system.

Tiled maps can be easily exported and imported into game engines such as Unity and Godot, allowing for seamless integration with your game. This feature is particularly useful for game developers, as it allows them to quickly and easily incorporate the maps they create into their games.

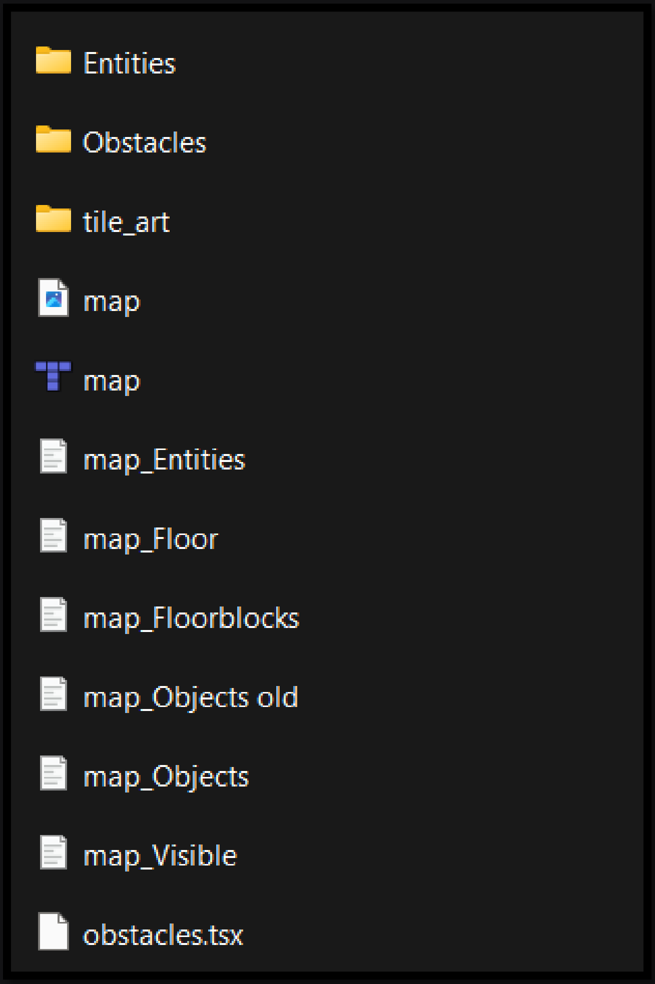
In short, Tiled is an excellent choice for creating maps for RPG games. Its user-friendly interface, customizable tile sets, and layer support make it a powerful and efficient tool for building immersive game worlds. Its cross-platform compatibility and easy integration with game engines make it a versatile tool that can be used by game developers of all experience levels. And lastly, its open-source nature makes it accessible to everyone, regardless of their budget.

In terms of tile maps, it supports straight rectangular tile layers, but also projected isometric, staggered isometric and staggered hexagonal layers. A tile-set can be either a single image containing many tiles, or it can be a collection of individual images. In order to support certain depth faking techniques, tiles and layers can be offset by a custom distance and their rendering order can be configured.

Tiled also supports object, image and group layers.



The map and its layers as well as tile-sets are stored in the Graphics folder of the program.


Entities
Obstacles
tile_art
map
map
map_Entities
map_Floor
map_Floorblocks
map_Objects old
map_Objects
map_Visible
obstacles.tsx

Object Layers

Object layers are useful because they can store many kinds of information that would not fit in a tile layer. Objects can be freely positioned, resized and rotated. They can also have individual custom properties. There are many kinds of objects:

- **Rectangle** - for marking custom rectangular areas
- **Ellipse** - for marking custom ellipse or circular areas
- **Point** - for marking exact locations (since Tiled 1.1)
- **Polygon** - for when a rectangle or ellipse doesn't cut it (often a collision area)
- **Polyline** - can be a path to follow or a wall to collide with
- **Tile** - for freely placing, scaling and rotating your tile graphics
- **Text** - for custom text or notes (since Tiled 1.0)

All objects can be named, in which case their name will show up in a label above them (by default only for selected objects). Objects can also be given a *class*, which is useful since it can be used to customize the colour of their label and the available custom properties for this object. For tile objects, the class can be inherited from their tile.

For most map types, objects are positioned in plain pixels. The only exception to this are isometric maps (not isometric staggered). For isometric maps, it was deemed useful to store their positions in a projected coordinate space. For this, the isometric tiles are assumed to represent projected squares with both sides equal to the *tile height*. If you're using a different coordinate space for objects in your isometric game, you'll need to convert these coordinates accordingly.

The object width and height are also mostly stored in pixels. For isometric maps, all shape objects (rectangle, point, ellipse, polygon and polyline) are projected into the same coordinate space described above. This is based on the assumption that these objects are generally used to mark areas on the map.

Object layers are used to implement the Interaction between the player and the world. An example of it is the shop system. The code checks if the player is in the area and then calls a function to create the Blacksmith class.

Image Layers

Image layers provide a way to quickly include a single image as foreground or background of your map. They currently have limited functionality and you may consider adding the image as a Tile-set instead and place it as a Tile Object. This way, you gain the ability to freely scale and rotate the image.

However, image layers can be repeated along the respective axes through their *Repeat X* and *Repeat Y* properties.

Image layer, Objects, is used to find the position of where to draw obstacles on the map and then the graphics of the items are accessed to draw the images of the obstacles on the map. All of the images are stored in one folder called 'Obstacles'.

Group Layer

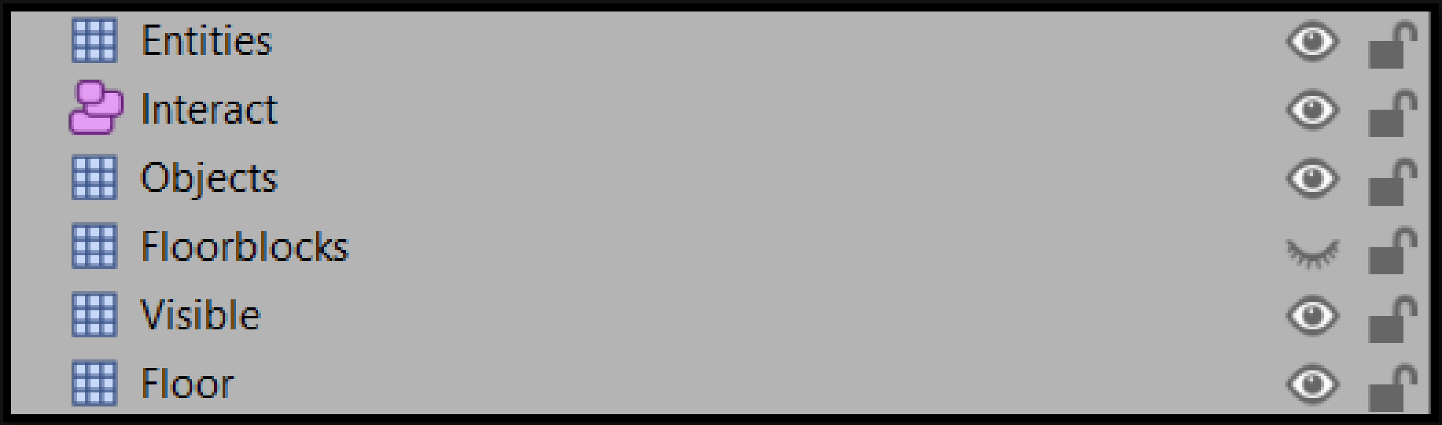
Group layers work like folders and can be used for organizing the layers into a hierarchy. This is mainly useful when your map contains a large amount of layers.

The visibility, opacity, offset, lock and tint colour of a group layer affects all child layers.

Layers can be easily dragged in and out of groups with the mouse. The Raise Layer / Lower Layer actions also allow moving layers in and out of groups.

Neo’s Enchanting Adventures are using orthogonal orientation map.

All of the layers used in the game:



Layers used to import layouts:

- Floorblocks
- Objects
- Entities

Floorblocks layer defines the boundary of the map beyond which the player unable to transverse.

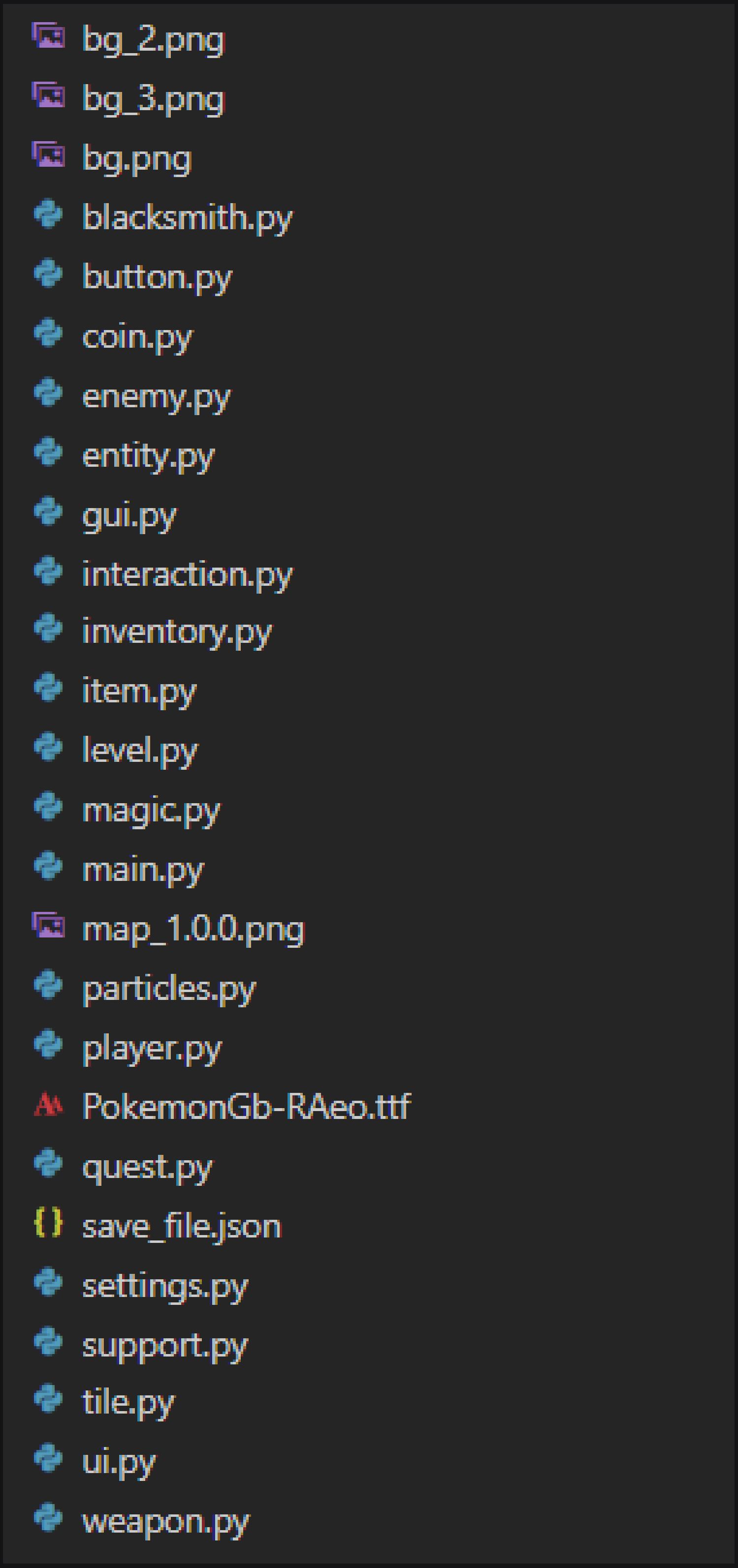
Objects layer defines the obstacles place on the map.

Entities layer defines the locations of the enemies and the player on the map.

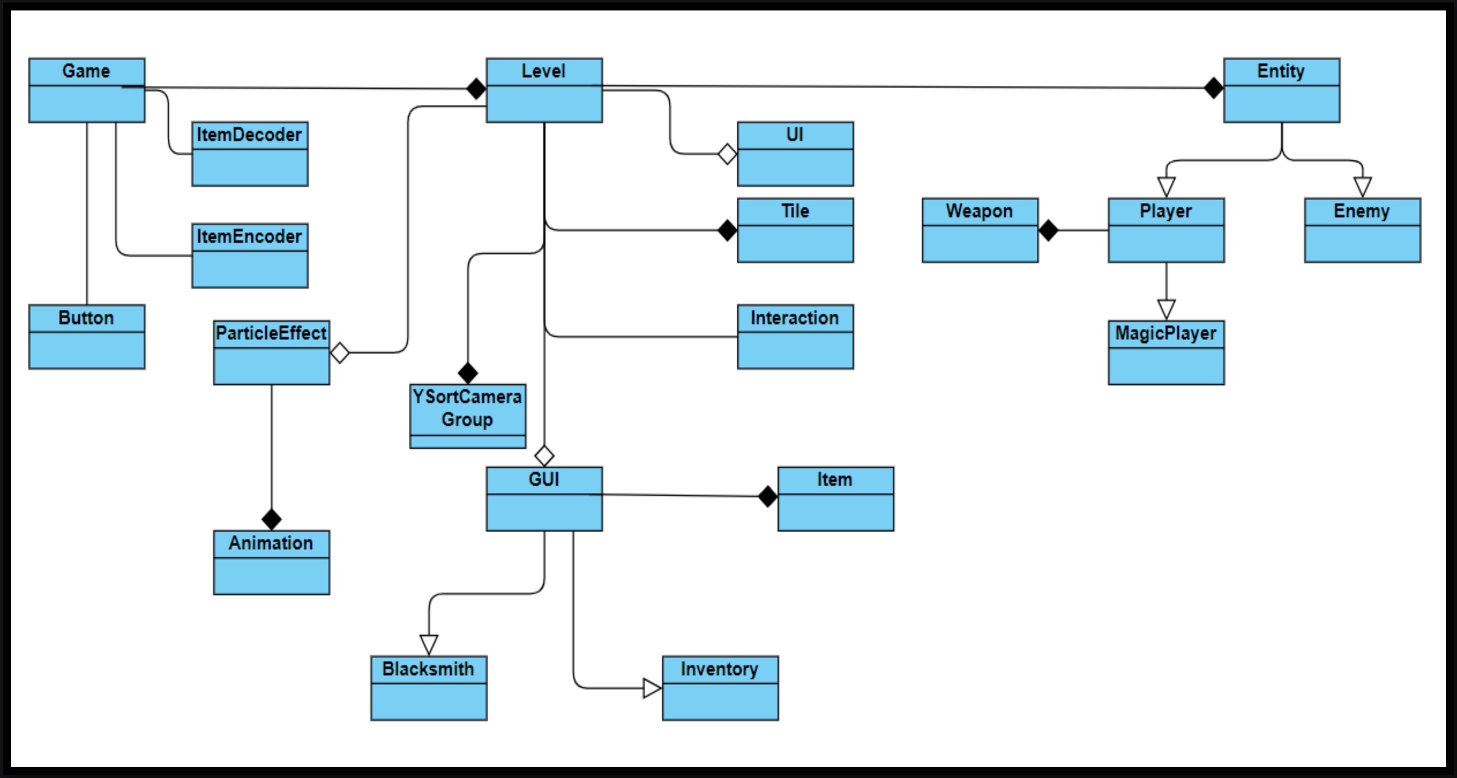
Floor and visible layers are used as a background image of the map. Both are imported from *Tiled* as an image and the backround of the game is filled with this image.

2.4 Code

The program will consist of 21 python files. Each file contains a class (sometimes two) or the methods used frequently in the game.


bg_2.png
bg_3.png
bg.png
blacksmith.py
button.py
coin.py
enemy.py
entity.py
gui.py
interaction.py
inventory.py
item.py
level.py
magic.py
main.py
map_1.0.0.png
particles.py
player.py
PokemonGb-RAeo.ttf
quest.py
save_file.json
settings.py
support.py
tile.py
ui.py
weapon.py

Class Diagram



Brief explanation of each of the classes:

❑ Game

has several methods, including **__init__**, **get_font**, **play_music**, **main_menu**,

run, **choose_character**, and **death_screen**. It imports several modules, including **pygame**, **pygame_gui**, **sys**, **json**, and a few others. It also imports several constants from other modules, including **settings.py**, **level.py**, **button.py**, and **support.py**.

The **__init__** method initializes several attributes of the **Game** object, including the game screen, the clock, and several elements related to the game's user interface (UI). It also loads the first level of the game and sets the initial volume for the game's music.

The **get_font** method returns a font object for the specified font size and font file. The **play_music** method plays the specified music file and sets the volume of the music to the previously specified volume.

The **main_menu** method displays the game's main menu, which includes several buttons for starting a new game, loading a game, accessing options, and quitting the game. Each button has its own **Button** object, which is defined in the **button.py** module. The **main_menu** method waits for the user to click one of the buttons and responds accordingly.

The **run** method runs the game loop and handles user input. It calls the **run** method of the **Level** object, which is defined in the **level.py** module, to update the game state and draw the game screen.

The **choose_character** method displays a screen that prompts the user to choose a character before starting the game. The user can choose between a sword-wielding character and a magic-wielding character.

The **death_screen** method displays a screen that informs the user that they have died in the game and prompts them to start a new game or quit.

- ❑ **Level**

is responsible for managing and displaying the game's current level. It contains a variety of methods that handle tasks such as creating and updating the game map, managing sprites, handling player attacks, displaying the UI, and more.

Upon initialization, the **Level** class sets up various properties and sprite groups. It creates the game map by loading a TMX file and importing CSV files containing layout data for the map's boundaries, objects, and entities. It also imports graphics for objects and creates sprite group s for visible sprites, obstacle sprites, and interaction sprites.

The **Level** class allows the player to select their character type, either sword or magic, which affects the type of attack and graphics displayed. The player's attack method is created based on their character type, and the UI is customized accordingly.

The **Level** class also handles various UI displays, including the inventory, blacksmith forge, alchemy station, and quest log. The appropriate display is toggled based on user input.

The **Level** class manages player attacks by creating and destroying attack sprites when appropriate. It also detects collisions between the player's attacks and attackable sprites, such as enemies.

The **Level** class handles the collection of coins by the player, adding them to a list of coins and updating the UI accordingly.

- ❑ **YSortCameraGroup**

is a subclass of the **pygame.sprite.Group** class, used for grouping sprites together in Pygame. This class has additional functionality for handling camera movement and drawing sprites in the correct order based on their vertical position.

The **__init__** method initializes the **YSortCameraGroup** object by calling the **__init__** method of its superclass, **pygame.sprite.Group**. It then sets the **display_surface** attribute to the Pygame surface object returned by **pygame.display.get_surface()**. It calculates the **half_width** and **half_height** of the display surface, and sets the **offset** attribute to a Pygame Vector2 object.

The **floor_surf** attribute is initialized by loading an image from a file path, and converting it to a Pygame surface object using the **convert()** method. The **floor_rect** attribute is initialized to a rectangle with its top-left corner at (0,0) and its size matching the **floor_surf**.

The **custom_draw** method is used to draw the sprites in the group onto the display surface, with the player object being used to determine the camera position. The **offset** attribute is updated to the difference between the center of the player object's rect and the **half_width** and **half_height** values. The floor is drawn first by subtracting the **offset** from the **topleft** attribute of the **floor_rect**, and then blitting the **floor_surf** onto the display surface at that position.

Next, the sprites in the group are sorted by their vertical position, with the **sorted()** function using a lambda function to sort the sprites by their **centery** attribute. Each sprite's position is then updated by subtracting the **offset** from its **topleft** attribute, and the sprite's image is blitted onto the display surface at that position.

The **enemy_update** method is used to update the enemy sprites in the group. It first creates a list of all sprites in the group that have a **sprite_type** attribute equal to **'enemy'**, using a list comprehension. It then loops over each sprite in the list and calls its **enemy_update** method with the player object as an argument. This method is expected to update the sprite's position and behavior based on the player's position.

❑ <u>**Entity**</u>

inherits from **pygame.sprite.Sprite**. It contains several methods for moving the entity, detecting collisions with other sprites, and calculating a wave value.

The **__init__** method initializes some attributes of the entity, including the index of the current animation frame, the animation speed, and the entity's direction vector.

The **move** method updates the position of the entity based on its direction and speed. It first normalizes the direction vector to ensure that the entity moves at a consistent speed, then moves the entity horizontally and checks for collisions with other sprites. If there is a collision, the entity is moved back to the edge of the obstacle. The same process is repeated for vertical movement.

The **collision** method checks for collisions with other sprites in the obstacle group, and adjusts the entity's position if a collision is detected.

The **wave_value** method calculates a value based on a sine wave that varies over time. If the sine wave is greater than or equal to 0, the method returns 255. Otherwise, it returns 0.

Note that the **obstacle_sprites** attribute is not defined in this code, so it is unclear where this attribute is coming from and what sprites it contains.

❑ Player

inherits from the **Entity** class. The **Player** class initializes the player sprite, handles the player's movement and input, and defines the player's stats and abilities.

The **Player** class has the following attributes:

- **armour_name**: a string representing the player's armor type
- **image**: a Pygame Surface object representing the player's sprite
- **rect**: a Pygame Rect object representing the player's position and size on the screen
- **hitbox**: a Pygame Rect object representing the player's hitbox size and position
- **status**: a string representing the player's current animation status
- **attacking**: a boolean indicating if the player is currently attacking
- **attack_cooldown**: an integer representing the cooldown time between attacks
- **attack_time**: an integer representing the time when the player last attacked
- **can_attack**: a boolean indicating if the player can currently attack
- **obstacle_sprites**: a Pygame sprite group representing the obstacles on the screen
- **interaction**: a Pygame sprite group representing the interactive objects on the screen
- **create_attack**: a function that creates an attack sprite when the player attacks
- **destroy_attack**: a function that destroys an attack sprite when it collides with an obstacle or enemy
- **animation_arrow**: a Pygame Surface object representing the arrow sprite used in the player's bow attack animation
- **weapon_list**: a dictionary representing the weapons available to the player
- **weapon_index**: an integer representing the index of the player's currently equipped weapon
- **sword_index**: an integer representing the index of the player's sword weapon
- **weapon**: a string representing the player's currently equipped weapon
- **allowed_to_switch**: a boolean indicating if the player can currently switch weapons
- **weapon_switch_time**: an integer representing the time when the player last switched weapons
- **switch_duration_cooldown**: an integer representing the cooldown time between weapon switches

- **toggle_forge**: a function that toggles the forge menu
- **toggle_alchemy**: a function that toggles the alchemy menu
- **toggle_quest**: a function that toggles the quest menu
- **stats**: a dictionary representing the player's current stats (health, energy, attack, magic, speed, and level)
- **max_stats**: a dictionary representing the player's maximum stats
- **health**: an integer representing the player's current health
- **energy**: an integer representing the player's current energy
- **exp_to_level_up**: an integer representing the amount of experience needed to level up
- **exp**: an integer representing the player's current experience
- **speed**: an integer representing the player's current speed
- **level**: an integer representing the player's current level
- **coins**: an integer representing the player's current number of coins
- **armour**: a dictionary representing the available armor types and their stats
- **current_armour**: a dictionary representing the player's currently equipped armor and its stats
- **health_before_armour**: an integer representing the player's health before equipping armor
- **vulnerable**: a boolean indicating if the player can currently take damage
- **hurt_time**: an integer representing the time when the player last took damage
- **invulnerable_duration**: an integer representing the duration of the player's invulnerability after taking damage
- **shielded**: a boolean indicating if the player is currently shielded
- **last_interaction_time**: an integer representing the time of the player's last interaction with an interactive object on the screen

The **Player** class has several methods:

- **__init__(self,position,groups,obstacle_sprites,interaction,toggle_forge,toggle_alchemy,toggle_quest,create_attack,destroy_attack,animation)**: This is the constructor method for the **Player** class. It takes in several parameters, such as the player's starting position, sprite groups, obstacle sprites, interaction sprites, and more. It initializes the player's stats, current weapon, current armor, animations, and more.
- **import_player_assets(self)**: This method loads the player's animations from the appropriate directory.
- **input(self)**: This method handles player input. It checks for key presses and updates the player's position and status accordingly. It also handles attacking, switching weapons, interacting with NPCs, and more.
- **update(self)**: This method updates the player's position, hitbox, and animations. It also handles collision detection with obstacles and interaction sprites.

- **get_health(self, amount)**: This method increases the player's health by a specified amount.
- **take_damage(self, damage)**: This method decreases the player's health by a specified amount. It also sets the player to be invulnerable for a short period of time.
- **heal(self, amount)**: This method heals the player by a specified amount.
- **get_energy(self, amount)**: This method increases the player's energy by a specified amount.
- **lose_energy(self, amount)**: This method decreases the player's energy by a specified amount.
- **switch_weapon(self, weapon)**: This method switches the player's current weapon to the specified weapon.
- **switch_armour(self, armour)**: This method switches the player's current armour to the specified armour.
- **update_stats(self)**: This method updates the player's stats based on their current armour and level.
- **attack(self)**: This method handles the player's attack animation and collision detection with enemies.
- **create_attack(self)**: This method creates a new attack sprite and adds it to the appropriate sprite group.
- **destroy_attack(self)**: This method destroys the player's current attack sprite.
- **toggle_forge(self)**: This method toggles the forge menu.
- **toggle_alchemy(self)**: This method toggles the alchemy menu.
- **toggle_quest(self)**: This method toggles the quest menu.
- **draw_stats(self, screen)**: This method draws the player's stats to the screen.
- **draw_interactions(self, screen)**: This method draws the interaction text to the screen.

MagicPlayer

has additional methods and attributes specific to a player character that uses magic attacks.

Here are the methods defined in the **MagicPlayer** class:

- **__init__(self,position,groups,obstacle_sprites,interaction,toggle_forge, toggle_inventory,toggle_quest,create_magic_attack,destroy_attack,animation)**: Initializes a **MagicPlayer** instance. It calls the **__init__** method of the **Player** class to set up basic player attributes, and then adds additional attributes specific to magic players. These include an animation for magic attacks, a method for creating a magic attack, a list of magic spells, and a boolean attribute for whether the player is shielded or not.
- **import_player_assets(self)**: Imports the character graphics for the magic player and sets up the **animations** dictionary to hold different animation states for the player.

- **input(self)**: Handles player input. It checks for arrow key input to move the player, the shift key to speed up the player, the "e" key to initiate a magic attack, the "n" key to heal the player, the "z" key to switch between magic spells, and the "return" key to interact with objects like a blacksmith. It also sets the **attacking** attribute to True if the player initiates a magic attack.
- **get_status(self)**: Updates the player's status based on their current direction and whether they are attacking or not.
- **cooldown(self)**: Manages the cooldown times for the player's abilities. It sets **attacking** to False if the magic attack cooldown has elapsed, and it sets **allowed_to_switch** to True if the weapon switch cooldown has elapsed. It also sets **vulnerable** to True if the invulnerability duration has elapsed.
- **energy_recovery(self)**: Handles the player's energy recovery, which is used to power magic attacks.
- **full_magic_damage(self)**: Calculates the total damage of a magic attack, which includes the player's base attack and the strength of the selected magic spell.
- **fireball(self,strength,cost)**: Creates a fireball magic attack with the specified strength and energy cost.

❑ <u>**Enemy**</u>

is a subclass of **Entity** and represents an enemy in a game. It has the following attributes:

- **sprite_type**: a string representing the type of sprite, which is set to "enemy".
- **image**: a pygame surface representing the current image of the enemy.
- **animations**: a dictionary containing pygame surfaces for each animation (e.g., idle, walking, attacking).
- **status**: a string representing the current status of the enemy (e.g., idle, walking, attacking).
- **frame_index**: an integer representing the current frame index of the animation.
- **start_position**: a tuple representing the starting position of the enemy.
- **rect**: a pygame rect representing the hitbox of the enemy.
- **hitbox**: a pygame rect representing the hitbox of the enemy with a slight adjustment for visual purposes.
- **obstacle_sprites**: a sprite group representing the obstacles in the game.

- **name**: a string representing the name of the enemy.
- **health**: an integer representing the current health of the enemy.
- **max_health**: an integer representing the maximum health of the enemy.
- **exp**: an integer representing the amount of experience points the enemy gives when defeated.
- **coins**: an integer representing the amount of coins the enemy drops when defeated.
- **speed**: an integer representing the speed of the enemy.
- **attack_damage**: an integer representing the amount of damage the enemy deals to the player when attacking.
- **resistance**: an integer representing the resistance of the enemy to attacks.
- **attack_radius**: an integer representing the radius within which the enemy can attack the player.
- **notice_radius**: an integer representing the radius within which the enemy notices the player.
- **attack_type**: a string representing the type of attack used by the enemy.
- **attack_sound**: a pygame sound object representing the sound effect played when the enemy attacks.
- **can_attack**: a boolean indicating whether the enemy can currently attack the player.
- **attack_time**: a float representing the time of the last attack.
- **attack_cooldown**: an integer representing the cooldown period between attacks.
- **damage_player**: a function representing the damage dealt to the player when attacked.
- **attack_particles**: a sprite group representing the particles emitted when the enemy attacks.
- **death_particles**: a sprite group representing the particles emitted when the enemy dies.
- **coin_spawn**: a function representing the coins dropped when the enemy dies.
- **add_exp**: a function representing the experience points gained when the enemy dies.
- **vulnerable**: a boolean indicating whether the enemy can currently take damage.
- **hit_time**: a float representing the time of the last hit.
- **invulnerable_duration**: an integer representing the duration of the invulnerability period after being hit.
- **moving**: a boolean indicating whether the enemy is currently moving.
- **last_position**: a tuple representing the previous position of the enemy.
- **spells**: a list containing the different spells the enemy can cast.
- **graph**: a dictionary representing the graph of nodes and edges connecting the obstacles in the game.
- **movement_speed**: a float representing the speed of the enemy's movement.
- **movement_radius**: an integer representing the radius within which the enemy can move.

- **target_position**: a pygame vector representing the position the enemy is moving towards.

The methods of the Enemy class include:

- The **init** method initializes the various attributes of the enemy object, such as its graphics, movement, stats, player interaction, vulnerability, and magic (for certain types of enemies). It also creates a graph of the game world's obstacles and their connections to enable pathfinding, and sets a target position for the enemy to move towards. The method takes in various parameters, such as the enemy's name, position, groups, obstacle_sprites, **damage_player, attack_particles, death_particles, coin_spawn, and add_exp**, to set up the object.
- The **create_graph** method creates a graph of the game world's obstacles and their connections, which is used for pathfinding by the enemy object.
- The **get_neighbors** method returns a list of obstacle sprites that are adjacent to a given sprite, which is used in creating the graph.
- The idle method moves the enemy object towards a randomly generated target position within a certain radius, if the enemy is not engaged in combat.
- The **import_graphics** method imports the graphics files for the various animations of the enemy object, such as forward_idle, left_idle, right_idle, back_idle, forward_walking, left_walking, right_walking, back_walking, forward_attack, left_attack, right_attack, and back_attack.
- The **get_direction_distance_player** method calculates the distance and direction between the enemy object and the player object, and returns them as a tuple.
- The **get_status** method determines the status of the enemy object, based on its distance from the player object and the direction it is facing. If the enemy is within attacking range and able to attack, its status is set to an attacking animation. If the enemy is within attacking range but not able to attack, its status is set to an idle animation. If the enemy is within noticing range, its status is set to a walking animation towards the player object. If the enemy is outside of noticing range, its status is set to an idle animation. The method also updates the frame index of the animation, depending on the status of the enemy.
- Additionally, to make the game more realistic the enemy moves around when the enemy is within noticeable range. Dijkstra's algorithm is used to add all of the obstacles on the map to **obstacle_graph** of the enemy, so the enemy is aware where the obstacles are and is less likely to collide with them.

❑ UI

class constructor initializes the display surface, font, and the position and dimensions of three bars (exp bar, health bar, and energy bar). It also initializes an empty list to store the weapon graphics.

The "import_graphics" method takes a dictionary of weapon graphics as an argument, loads the graphics from the given paths, and appends them to the "weapon_graphics" list.

The "get_font" method returns a Pygame font object with the given size.

The "display_stats_image" method loads and displays an image on the screen.

The "display_bar" method takes the current and maximum values for a stat, the background rectangle of the bar, and the color of the bar as arguments, and displays a bar on the screen with the current value represented by a colored rectangle.

The "display_level" method takes the player's level as an argument, renders the level as text using the font object, and displays it on the screen.

The "display_coins" method takes the number of coins the player has as an argument, loads a coin icon, renders the number of coins as text using the font object, and displays both the icon and the text on the screen.

The "selection_box" method takes the position, size, and whether the box is currently selected or not as arguments, and displays a rectangular box with a border on the screen. If the box is currently selected, it uses a different color for the border.

The "weapon_overlay" method takes the index of the current weapon and whether the player is allowed to switch weapons as arguments, displays a selection box, and displays the graphic for the current weapon in the center of the box.

Finally, the "display" method takes the player object as an argument, calls several of the above methods to display the player's stats, level, coins, and weapon, and displays them all on the screen.

- ❑ **GUI**

handles the drawing of inventory slots and items, as well as their interaction. The GUI class has various attributes, such as item size, slot size, inventory slot image, font, last click and last remove times, and methods, such as drawing slots and items, adding and removing items, and equipping or selling items.

The **draw_slots_and_items** method takes two optional arguments, **is_inventory** and **is_in_shop**, which default to True and False, respectively. This method is used by both Inventory and Blacksmith classes and depeding on the parameters set when the fucntion is called the method performs different functionality. The default is s that the method is used by inventory and the inventory of the player is not in shop. The functionality will differ when the **Inventory** is called in **Blacksmith** class. The **is_in_shop** value is set to True and by presin gthe items with left mouse button in player’s inventory they are sold to the blacksmith. On the contrary, if the method is called by **Blacksmith** class, then by hovering and clicking on the item it is bought if the player has enough coins.

The **add_item** method adds an item to the inventory if there is room for it, otherwise it returns False.

The **remove_item** method removes an item from the inventory at the given index and sets the corresponding item slot to None.

- ❑ **Inventory**

inherits from the GUI class. The Inventory class is responsible for drawing the player's inventory, equipment, weapons, and spells on the game screen.

The class has an **init** method that initializes the Inventory object with an image for the inventory background, a surface to display the inventory, a player object, the inventory capacity, a list of items, equipped armour and weapon items, fonts for the main display and item descriptions, and various dictionaries for armour, weapons, and spells.

The class has several methods for drawing different parts of the inventory screen. The **draw_armour_pieces** method draws the armour items that the player is currently wearing. It checks to see if the player has an item in each armour slot and, if so, draws the item's graphic and displays a description of the item when the player hovers over it.

The **draw_current_weapons** method draws the weapons that the player is currently holding. It checks to see which weapons the player has and, if the player has one or two weapons, it draws the weapon graphics and displays the weapon's damage and attack radius when the player hovers over it.

The **draw_coins** method draws the player's current amount of coins on the screen. It loads a coin graphic, scales it, and draws it on the screen. It then renders the amount of coins the player has and displays it next to the coin graphic.

The **draw_player_stats** method displays the player's level, experience, health, and mana on the screen. It renders each statistic using the main font and displays it at a specified position on the screen.

- ❑ **Blacksmith**

inherits from **GUI** class. It has some attributes such as **background**, **display_surface**, **capacity**, **items**, **item_slots**, **player_inventory**, **start_x**, and **start_y**.

In the **__init__** method, the **background** attribute is loaded from an image file, the **capacity** is set to 36, **items** and **item_slots** are initialized as empty lists with a length of **capacity**, **player_inventory** is set to an instance of **Inventory and start_x** and **start_y** are set to specific values. In the **for** loop inside the **__init__** method, items are created and added to the **items** attribute if they are not already in the **player_inventory**.

The **remove_item** method takes an **item_index** and sets the corresponding item and item slot to **None**.

The **draw_player_inventory** method calls the **draw_slots_and_items** and **draw_coins** methods of **player_inventory** and passes **True**, **True**, and **self** as arguments.

The **sell_item** method takes an **item_index**, and if the player has enough coins, it adds the corresponding item to their inventory, deducts the cost of the item from their coins, and removes the item from the **items** and **item_slots** attributes.

The **display** method displays the blacksmith GUI by first displaying the **background**, then calling the **draw_slots_and_items** method of **GUI** (which **Blacksmith** inherits from), passing **False** as an argument, and calling **draw_player_inventory**.

❑ Item

a constructor method that takes in a **name** and **type** parameter and initializes instance variables for **cost** and **description** to 0 and an empty string, respectively. The **set_attributes** method retrieves the attributes for the given **name** and **type** combination from the **item_attributes** dictionary defined in **settings.py**, and sets the **description**, **cost**, and **graphic** attributes of the **Item** instance accordingly. If the **cost** attribute is not specified in **item_attributes**, it is set to **None**.

❑ Weapon

inherits from **pygame.sprite.Sprite**, which means it can be added to sprite groups and drawn on the screen.

The **__init__** method takes two arguments: **player**, which is an instance of the **Player** class, and **groups**, which is a list of sprite groups that the weapon should be added to.

The first thing the method does is call the **__init__** method of the **pygame.sprite.Sprite** class to set up the sprite. It also sets the **sprite_type** attribute to "weapon", which will be used later to determine what kind of sprite it is.

Next, it determines the direction the player is facing by splitting the player's **status** attribute (which has the format "direction_state") and taking the first part. For example, if the player's **status** is "left_idle", **direction** will be "left".

It sets the position of the weapon's **rect** attribute based on the direction the player is facing. If the player is facing forward, the weapon is placed above the player's head; if the player is facing back, the weapon is placed below the player's feet; if the player is facing left, the weapon is placed to the left of the player; and if the player is facing right, the weapon is placed to the right of the player.

❑ <u>**Interaction**</u>

defines a sprite that represents an interactable object in the game. The constructor takes in **pos** and **size** parameters to determine the object's position and size, respectively. It also takes in **groups** which represents the sprite groups that the interaction sprite belongs to, and **name** which represents the name of the interactable object.

Inside the constructor, the **image** attribute is set to a **pygame.Surface** with size **size**. The **rect** attribute is then set to the rectangle bounding the **image** and positioned at **pos**. The **hitbox** attribute is also set to a copy of the **rect** that is slightly smaller, presumably to make the object easier to interact with.

The **name** attribute is set to the value of the **name** parameter passed in.

❑ <u>**Quest**</u>

is used to manage quests in a game. The class has several methods that are used to add and manage quests, display quest information, and give rewards to the player.

The **__init__** method initializes the class with several attributes, including a reference to the **Player** object, the background image for the quest display, and font settings for displaying text. It also initializes several attributes related to managing quests, including a list of available quests, the current quest being undertaken by the player, and a list of completed quests.

The **add_quest** method is used to add a new quest to the player's quest log. It loops through the available quests and checks if the current quest is completed or if there is no current quest. If there is no current quest and the quest is not completed, the current quest is set to this quest. If the quest is completed and not in the completed list, the quest is added to the completed list and the current quest is set to **None**.

The **set_completed** method checks if the player has completed the current quest by checking if they have met the quest's objective. If the player has completed the quest, the quest is marked as completed and the **play_sound** function is called to play a sound effect.

The **get_rewards** method gives the player any rewards for completing the quest, such as experience points and coins.

The **draw_available_quests** method is used to display information about the current quest, including its name, objective, description, and rewards.

The **draw_complete_quests** method is used to display a list of completed quests.

Finally, the **display** method is used to display the quest log on the game screen by calling the various drawing methods and setting the completed status of the current quest.

❑ Tile

is used to create tiles in a game using the Pygame library. The **Tile** class inherits from the **pygame.sprite.Sprite** class and has several attributes and methods.

The **__init__** method initializes the class with several attributes, including the tile's position, the sprite groups it belongs to, the sprite type (e.g. grass, dirt, etc.), and a surface to represent the tile. It also sets the image and hitbox of the tile, with the hitbox being slightly larger than the image to allow for collision detection.

The **image** attribute is the actual image that will be displayed on the screen. It is set to the surface passed in as an argument, which by default is a **pygame.Surface** object with dimensions of **TILESIZE** by **TILESIZE**. This attribute is used to display the tile on the screen.

The **rect** attribute is a **pygame.Rect** object that represents the tile's position and dimensions. It is set to the position passed in as an argument and the dimensions of the tile's image.

The **hitbox** attribute is also a **pygame.Rect** object that represents the tile's collision detection area. It is set to the same position as the tile's rect, but with a height that is slightly larger than the tile's image. This allows for more accurate collision detection with the player sprite.

❑ ParticleEffect

used for creating particle effects in Pygame. It inherits from the Pygame sprite class and takes in the following parameters in its constructor:

- **pos**: the position of the particle effect on the screen
- **animation_frames**: a list of images representing the animation frames for the particle effect
- groups: a list of Pygame sprite groups that the particle effect should be added to
- **sprite_type**: a string representing the type of particle effect (e.g. "death", "fireball", "arrow", etc.)

The **ParticleEffect** class has an animate() method that updates the particle effect's **frame_index** and image, and an **update** method that calls the **animate** method. When the particle effect's **frame_index** goes beyond the number of frames in its animation, the particle effect is removed from all sprite groups and killed.

The **animation_speed** property determines how quickly the particle effect's frames are cycled through. It is different for each **sprite_type**, with values chosen empirically to achieve the desired animation speed.

❑ Animation

loads sprite sheets for various animations (such as spells, arrows, and particle effects), and a **create_particles** method that creates a **ParticleEffect** object using the appropriate animation frames for the given **animation_type**.

The **import_folder** function is likely a custom function that imports all the image files in a given folder and returns them as a list of **pygame.Surface** objects.

❑ Coin

inherits from **pygame.sprite.Sprite**. The class defines a sprite object that represents a coin in a game. The class has several methods:

- **__init__**: This method initializes the coin object with its position, amount, and groups to which it belongs. It also loads the frames for the animation of the coin.
- **animate**: This method animates the coin by updating its image to the next frame in the animation sequence.
- **check_collide**: This method checks if the coin collides with a player by comparing the player's rectangle to the coin's rectangle. If there is a collision, it returns **True**, otherwise it returns **False**.
- **update**: This method calls the **animate** method to update the coin's image.

The **Coin** class is meant to be used with the Pygame library to create a sprite object that can be added to a group of sprites and updated in a game loop. The class can be instantiated with a position, group, and amount to create a coin object that can be used in a game. The **update** method should be called each frame to update the coin's animation, and the **check_collide** method can be used to detect collisions with a player.

❑ Button

The class has several methods:

- **__init__**: This method initializes the button object with its image, position, text, font, base colour, and hovering colour. It also creates a rectangle object for the button.
- **update**: This method updates the button by blitting the button's image and text onto the screen.
- **checkForInput**: This method checks if the mouse cursor is within the button's rectangle by checking if the cursor's x and y positions are within the left/right and top /bottom

boundaries of the rectangle. If the cursor is within the button's rectangle, it returns **True**, otherwise it returns **False**.

- **changeColour**: This method changes the colour of the button's text when the mouse cursor is hovering over the button. If the cursor is within the button's rectangle, the text colour is changed to the hovering colour, otherwise it is changed back to the base colour.
- **drawRectAround**: This method draws a rectangle around the button if the cursor is hovering over it.

The **Button** class can be used in a Pygame game to create a button object with an image, text, and position. The class can be instantiated with the necessary parameters, and then its **update** method should be called each frame to update the button on the screen. The **checkForInput** method can be used to detect if the button has been clicked, and the **changeColour** method can be used to change the colour of the button's text when the mouse cursor is hovering over it. Finally, the **drawRectAround** method can be used to draw a rectangle around the button when the cursor is hovering over it.

❑ **ItemEncoder**

inherits from **json.JSONEncoder**. This class is used to encode instances of a custom class named **Item** as JSON objects.

The **default** method is overridden to handle instances of the **Item** class. If the object being encoded is an instance of the **Item** class, the method returns the dictionary representation of the object's attributes using the **__dict__** method. Otherwise, it returns the result of the **JSONEncoder** class's default method to handle the object.

The **ItemEncoder** class is used when encoding **Item** objects to JSON. This is done by creating an instance of the **ItemEncoder** class and using its **encode** method to convert the **Item** object to a JSON string. The resulting JSON string will contain the attributes of the **Item** object in a JSON format. If any of the attributes of the **Item** object are themselves custom classes, those classes will need to be encoded separately using their own custom encoders.

❑ **ItemDecoder**

inherits from **json.JSONDecoder**. This class is used to decode JSON objects that were encoded from instances of a custom class named **Item**.

The **__init__** method overrides the **__init__** method of the parent class and specifies the **object_hook** argument as the **object_hook** method of the **ItemDecoder** class.

The **object_hook** method is used to parse the JSON objects and return the appropriate Python objects. If the dictionary has a key **__type__** with the value **'item'**, the method returns a new instance of the **Item** class using the dictionary as its keyword arguments. If the dictionary does not have the key **__type__** with the value **'item'**, the method returns the original dictionary.

When decoding JSON that was encoded from an **Item** object, this **ItemDecoder** class is used by calling its **decode** method with the JSON string as the argument. The resulting object will be the original **Item** object that was encoded. If there were any other custom classes used as attributes in the **Item** object, their corresponding decoders should also be used to decode the JSON strings for those attributes.

3 Technical Solution

Main

```
from pygame import *
import pygame_gui
import pygame, sys
from settings import *
from level import Level
from button import *
import json
from support import *


class Game:
    def __init__(self):


        #general setup
        pygame.init()
        self.screen = pygame.display.set_mode((WIDTH,HEIGHT))
        pygame.display.set_caption('NEA')
        self.clock = pygame.time.Clock()


        self.level = Level()


        self.gui_manager = pygame_gui.UIManager((WIDTH, HEIGHT))
        self.music_slider = 
pygame_gui.elements.UIHorizontalSlider(relative_rect=pygame.Rect((400, 190), (450, 
40)),start_value=50,value_range=(0, 100), manager = self.gui_manager)
        self.sound_effects = 
pygame_gui.elements.UIHorizontalSlider(relative_rect=pygame.Rect((400, 350), (450, 
40)),start_value=50,value_range=(0, 100), manager = self.gui_manager)


        self.music = None
        self.music_volume = 0.5



    def get_font(self, size):
        return pygame.font.Font(UI_FONT, size)

    def play_music(self,path):
        if self.music is not None:
            self.music.stop()
        self.music = pygame.mixer.Sound(path)
        self.music.set_volume(self.music_volume)
        self.music.play()

    def main_menu(self):
        background = pygame.image.load(MAIN_BACKGROUND)
        self.play_music(MENU_AUDIO)


        while True:
            self.screen.blit(background, (0,0))
            menu_mouse_pos = pygame.mouse.get_pos()


            menu_Text = self.get_font(45).render("NEO'S ENCHANTING ADVENTURES", True, 
"#b68f40")
            menu_Rect = menu_Text.get_rect(center =(640, 100))


            LOAD_BUTTON = Button(image = None, pos =(640, 225),
            text_input = "PLAY", font = self.get_font(52), base_colour = MENU_TEXT_COLOUR, 
hovering_colour = "white")
            NEW_GAME_BUTTON = Button(image = None, pos =(640, 350),
            text_input = "NEW GAME", font = self.get_font(52), base_colour = 
MENU_TEXT_COLOUR, hovering_colour = "white")
            OPTIONS_BUTTON = Button(image = None, pos =(640, 475),
            text_input = "OPTIONS", font = self.get_font(52), base_colour = 
MENU_TEXT_COLOUR, hovering_colour = "white")
            QUIT_BUTTON = Button(image = None, pos =(640, 600), 
            text_input = "QUIT", font = self.get_font(52), base_colour = MENU_TEXT_COLOUR, 
hovering_colour = "white")


            self.screen.blit(menu_Text, menu_Rect)


            for button in [LOAD_BUTTON, NEW_GAME_BUTTON, OPTIONS_BUTTON, QUIT_BUTTON]:
                button.changeColour(menu_mouse_pos)
                button.update(self.screen)



            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if LOAD_BUTTON.checkForInput(menu_mouse_pos):
                        self.play_music(MENU_SELECT_AUDIO)
                        self.load()
                    if NEW_GAME_BUTTON.checkForInput(menu_mouse_pos):
                        self.play_music(MENU_SELECT_AUDIO)
                        self.choose_character()
                    if OPTIONS_BUTTON.checkForInput(menu_mouse_pos):
                        self.play_music(OPTIONS_AUDIO)
                        self.options()
                    if QUIT_BUTTON.checkForInput(menu_mouse_pos):
                        pygame.quit()
                        sys.exit()


            pygame.display.update()



    def run(self):
        self.play_music(GAMEPLAY_AUDIO)
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.KEYDOWN:
                    if event.key == K_ESCAPE:
                        self.pause()
                    if event.key == pygame.K_i:
                        self.level.toggle_inventory()
                    elif event.key == pygame.K_q:
                        self.level.toggle_quest()


            self.screen.fill('pink')
            if self.level.player_type == None:
                self.choose_character()
            if self.level.player.check_death():
                self.death_screen()
            self.level.run()
            pygame.display.update()
            self.clock.tick(FPS)


    def choose_character(self):
        background = pygame.image.load(MAIN_BACKGROUND)
        sword_icon = pygame.image.load(SWORD_ICON)
        magic_icon = pygame.image.load(MAGIC_ICON)


        while True:
            mouse_pos = pygame.mouse.get_pos()
            self.screen.blit(background, (0,0))


            choose_text = self.get_font(35).render('YOU MUST CHOOSE A CHARACTER FIRST', 
True, 'white')
            choose_text_rect = choose_text.get_rect(center = (640,175))
            self.screen.blit(choose_text, choose_text_rect)


            SWORD_BUTTON = Button(image = sword_icon, pos =(400, 375), 
            text_input = "", font = self.get_font(50), base_colour = MENU_TEXT_COLOUR, 
hovering_colour = 'white')
            MAGIC_BUTTON = Button(image = magic_icon, pos =(850, 375), 
            text_input = "", font = self.get_font(50), base_colour = MENU_TEXT_COLOUR, 
hovering_colour = 'white')


            for button in [SWORD_BUTTON, MAGIC_BUTTON]:
                button.drawRectAround(self.screen,mouse_pos)
                button.update(self.screen)
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if SWORD_BUTTON.checkForInput(mouse_pos):
                        self.play_music(MENU_SELECT_AUDIO)
                        self.level.set_player_type("sword")
                        self.run()
                    if MAGIC_BUTTON.checkForInput(mouse_pos):
                        self.play_music(MENU_SELECT_AUDIO)
                        self.level.set_player_type("magic")
                        self.run()

            pygame.display.update()


    def save(self):
        self.play_music(MENU_SELECT_AUDIO)
        game_data = {
            'player': {
                'type': self.level.player_type,
                'inventory': self.level.inventory.items,
                'level': self.level.player.level,
                'health': self.level.player.health,
                'energy': self.level.player.energy,
                'coins': self.level.player.coins,
                'exp': self.level.player.exp,
                'armour_name': self.level.player.armour_name,
                'sword_index': self.level.player.sword_index,
            },
            'quest': {
                'current_quest': self.level.quest.current_quest,
                'completed': self.level.quest.completed,
            }
        }

        with open('save_file.json', 'w') as file:
            json.dump(game_data, file, cls=ItemEncoder)


    def load(self):
        with open('save_file.json', 'r') as file:
            game_data = json.load(file, cls=ItemDecoder)
        if 'type' in game_data['player']:
            self.level.player_type = game_data['player']['type']
        else:
            self.choose_character()
        Self.level.inventory.items = game_data['player']['inventory']
        self.level.player.level = game_data['player']['level']
        self.level.player.health = game_data['player']['health']
        self.level.player.energy = game_data['player']['energy']
        self.level.player.coins = game_data['player']['coins']
        self.level.player.exp = game_data['player']['exp']
        self.level.player.armour_name = game_data['player']['armour_name']
        self.level.player.sword_index = game_data['player']['sword_index']


        self.level.quest.current_quest = game_data['quest']['current_quest']
        self.level.quest.completed = game_data['quest']['completed']


        self.run()



    def options(self):
        background = pygame.image.load(BACKGROUND_2)
        self.play_music(OPTIONS_AUDIO)
        clock = pygame.time.Clock()

        while True:
            time_delta = clock.tick(60) / 1000.0
            self.screen.blit(background, (0,0))
            options_mouse_pos = pygame.mouse.get_pos()


            music_text = self.get_font(45).render("Music Volume", True, MENU_TEXT_COLOUR)
            music_rect = music_text.get_rect(center=(640,140))
            self.screen.blit(music_text, music_rect)


            effects_text = self.get_font(45).render("Sound Effects Volume", True,
MENU_TEXT_COLOUR)
            effects_rect = effects_text.get_rect(center=(640,290))
            self.screen.blit(effects_text, effects_rect)
            OPTIONS_BACK = Button(image = None, pos =(640,460), text_input = "BACK", font =
self.get_font(50), base_colour = MENU_TEXT_COLOUR, hovering_colour = 'white')
            OPTIONS_BACK.changeColour(options_mouse_pos)
            OPTIONS_BACK.update(self.screen)


            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if OPTIONS_BACK.checkForInput(options_mouse_pos):
                        return None

                self.gui_manager.process_events(event)


            self.gui_manager.update(time_delta)
            self.gui_manager.draw_ui(self.screen)
            self.music_volume = self.music_slider.get_current_value() / 100
            effects_volume = self.sound_effects.get_current_value() / 100
            pygame.display.update()



    def pause(self):
        background = pygame.image.load(BACKGROUND_2)
        self.play_music(OPTIONS_AUDIO)

        while True:
            self.screen.blit(background, (0,0))
            load_mouse_pos = pygame.mouse.get_pos()



            RESUME_BUTTON = Button(image = None, pos =(640, 120),
            text_input = "RESUME", font = self.get_font(35), base_colour =
MENU_TEXT_COLOUR, hovering_colour = 'white')
            SAVE_BUTTON = Button(image = None, pos =(640, 245),
            text_input = "SAVE", font = self.get_font(35), base_colour = MENU_TEXT_COLOUR,
hovering_colour = 'white')
            HELP_BUTTON = Button(image = None, pos =(640, 370),
            text_input = "HELP", font = self.get_font(35), base_colour = MENU_TEXT_COLOUR,
hovering_colour = 'white')
            OPTIONS_BUTTON = Button(image = None, pos =(640, 495),
            text_input = "OPTIONS", font = self.get_font(35), base_colour =
MENU_TEXT_COLOUR, hovering_colour = 'white')
```

```
            QUIT_BUTTON = Button(image = None, pos =(640, 620),
            text_input = "EXIT TO MAIN MENU", font = self.get_font(35), base_colour =
MENU_TEXT_COLOUR, hovering_colour = 'white')


            for button in [RESUME_BUTTON, SAVE_BUTTON,HELP_BUTTON, OPTIONS_BUTTON,
QUIT_BUTTON]:
                button.changeColour(load_mouse_pos)
                button.update(self.screen)


            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if RESUME_BUTTON.checkForInput(load_mouse_pos):
                        self.play_music(GAMEPLAY_AUDIO)
                        self.run()
                    if SAVE_BUTTON.checkForInput(load_mouse_pos):
                        self.save()
                    if HELP_BUTTON.checkForInput(load_mouse_pos):
                        self.help()
                    if OPTIONS_BUTTON.checkForInput(load_mouse_pos):
                        self.options()
                    if QUIT_BUTTON.checkForInput(load_mouse_pos):
                        self.play_music(MENU_AUDIO)
                        self.main_menu()
                if event.type == pygame.KEYDOWN:
                    if event.key == K_ESCAPE:
                        return None


            pygame.display.update()


    def help(self):
        background = pygame.image.load(BACKGROUND_2)
        while True:
            self.screen.blit(background, (0,0))
            mouse_pos = pygame.mouse.get_pos()


            key_bindings = ["ATTACK: E","SWITCH WEAPON: Z","INVENTORY: I", "QUESTS: Q",
"BLACKMISMITH INTERACTION: ENTER"]
            y = 120
            for key in key_bindings:
                text = self.get_font(30).render(key, True, MENU_TEXT_COLOUR)
                rect = text.get_rect(center=(640, y))
                self.screen.blit(text, rect)
                y += 100

            BACK = Button(image = None, pos =(640,620), text_input = "BACK", font =
self.get_font(50), base_colour = MENU_TEXT_COLOUR, hovering_colour = 'white')
            BACK.changeColour(mouse_pos)
            BACK.update(self.screen)


            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if BACK.checkForInput(mouse_pos):
                        return None


            pygame.display.update()


    def death_screen(self):
        self.play_music(DEATH_AUDIO)


        while True:
            self.screen.fill('black')
            mouse_pos = pygame.mouse.get_pos()


            dead_text = self.get_font(60).render("YOU DIED", True, "red")
            dead_rect = dead_text.get_rect(center=(640,220))
            self.screen.blit(dead_text,dead_rect)


            LOAD = Button(image = None, pos =(640,340), text_input = "LOAD RECENT SAVE
FILE", font = self.get_font(30), base_colour = MENU_TEXT_COLOUR, hovering_colour = "white")
            BACK = Button(image = None, pos =(640,460), text_input = "QUIT TO MAIN MENU",
font = self.get_font(30), base_colour = MENU_TEXT_COLOUR, hovering_colour = "white")


            for button in [LOAD, BACK]:
                button.changeColour(mouse_pos)
                button.update(self.screen)



            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if LOAD.checkForInput(mouse_pos):
                        self.load()
                    if BACK.checkForInput(mouse_pos):
                        self.main_menu()

            pygame.display.update()




if __name__ == '__main__':
    game = Game()
    game.main_menu()
```

Level

```
import pygame
from pytmx.util_pygame import load_pygame
from settings import *
from tile import Tile
from player import Player
from magic import MagicPlayer
from support import *
from weapon import *
from ui import *
from enemy import Enemy
from particles import Animation
from coin import Coin
from inventory import *
from interaction import Interaction
from blacksmith import Blacksmith
from quest import Quest
```

```
class Level:
    def __init__(self):


        self.player_type = None


        self.display_surface = pygame.display.get_surface()
        self.inventory_display = False
        self.forge_display = False
        self.alchemy_display = False
        self.quest_display = False


        # sprite group setup
        self.visible_sprites = YSortCameraGroup()
        self.obstacle_sprites = pygame.sprite.Group()
        self.interaction_sprites = pygame.sprite.Group()


        self.current_attack = None
        self.attack_sprites = pygame.sprite.Group()
        self.attackable_sprites = pygame.sprite.Group()


        #particles
        self.animation = Animation()


        # sprite setup
        self.create_map()


        self.ui = UI()

        self.coins = []



    def create_map(self):
        tmx_data = load_pygame('.\\Graphics\\map\\map.tmx')

        layouts = {
            'boundary': import_csv_layout('.\\Graphics\\map\\map_Floorblocks.csv'),
            'object': import_csv_layout('.\\Graphics\\map\\map_Objects.csv'),
            'entities' : import_csv_layout('.\\Graphics\\map\\map_Entities.csv')
            }
        graphics = {
            'objects': import_folder('.\\Graphics\\Obstacles')
        }


        for style, layout in layouts.items():
            for row_index, row in enumerate(layout):
                for column_index, column in enumerate(row):
                    if column != '-1':
                        x = column_index * TILESIZE
                        y = row_index * TILESIZE
                        if style == 'boundary':
                            Tile((x,y),[self.obstacle_sprites],'floor_blocks')
                        if style == 'object':
                            surface = graphics['objects'][int(column)]
                            Tile((x,y),[self.visible_sprites,
self.obstacle_sprites],'object', surface)
                        if style == 'entities':
                            if column == '1':
                                pass
                            else:
                                if column == '2' : name = 'mage'
                                elif column == '3' :  name = 'skeleton_sword'
                                elif column == '4' :  name = 'baldric'
                                elif column == '7' : name = 'skeleton_magic'
                                else: name = 'mage'
                                Enemy(name, (x,y),
[self.visible_sprites,self.attackable_sprites],

self.obstacle_sprites,self.damage_player,self.enemy_attack_particles,self.death_particles,s
elf.coin_spawn,self.add_exp)



        for object in tmx_data.get_layer_by_name('Interact'):
            if object.name == 'Blacksmith':
                Interaction((object.x,object.y), (object.width,object.height),
self.interaction_sprites, object.name)


    def set_player_type(self,player_type):
        self.player_type = player_type
        if self.player_type == "magic":
            self.player =
MagicPlayer((2850,2400),[self.visible_sprites],self.obstacle_sprites,self.interaction_sprit
es,
            self.toggle_forge,self.toggle_alchemy,self.toggle_quest,
self.create_magic_attack,self.destroy_attack,self.animation)

            self.ui.import_graphics(magic_spells)

        elif self.player_type == "sword":
            self.player =
Player((2850,2400),[self.visible_sprites],self.obstacle_sprites,self.interaction_sprites,
            self.toggle_forge, self.toggle_alchemy, self.toggle_quest,
self.create_attack,self.destroy_attack,self.animation)

            self.ui.import_graphics(weapons)


        self.inventory = Inventory(self.player)
        self.forge = Blacksmith(self.inventory)
        self.quest = Quest(self.player)


    def run(self):
        self.visible_sprites.custom_draw(self.player)
        self.ui.display(self.player)
        self.quest.add_quest()

        if self.inventory_display and not self.forge_display and not self.alchemy_display
and not self.quest_display:
            self.inventory.display()
        elif self.forge_display:
            self.visible_sprites.update()
            self.forge.display()
        elif self.alchemy_display:
            self.visible_sprites.update()
            self.alchemy.display()
        elif self.quest_display:
            self.quest.display()
        else:
            self.visible_sprites.update()
            self.visible_sprites.enemy_update(self.player)
            self.player_attack()
            self.coin_pickup()


    def create_attack(self):
        self.current_attack = Weapon(self.player,[self.attack_sprites])
        if self.player.weapon_index == 1:
            self.player.bow_attack([self.visible_sprites,self.attackable_sprites])
    def create_magic_attack(self,style,strength,cost):
        if style == 'fireball':
            self.player.fireball(strength,cost,[self.visible_sprites,self.attack_sprites])
        elif style == 'heal':
            self.player.heal(strength,cost,self.visible_sprites)
        elif style == 'shield':
            self.player.shield(strength,cost,self.visible_sprites)
        elif style == 'icicle':
            self.player.icicle(strength,cost,[self.visible_sprites,self.attack_sprites])
        elif style == 'quake':
            self.player.quake(strength,cost,[self.visible_sprites,self.attack_sprites])


    def destroy_attack(self):
        if self.current_attack:
            self.current_attack.kill()
        self.current_attack = None


    def player_attack(self):
        if self.attack_sprites:
            for attack_sprite in self.attack_sprites:
                collision_sprites = 
pygame.sprite.spritecollide(attack_sprite,self.attackable_sprites, False)
                if collision_sprites:
                    for target_sprite in collision_sprites:
                        if target_sprite.sprite_type == 'enemy':
                            target_sprite.get_damage(self.player,attack_sprite.sprite_type)


    def damage_player(self,amount,attack_type):
        if self.player.vulnerable and not self.player.shielded:
            self.player.health -= amount
            self.player.vulnerable = False
            self.player.hurt_time = pygame.time.get_ticks()


    def enemy_attack_particles(self,particle_type,position,sprite_type):

self.animation.create_particles(particle_type,position,[self.visible_sprites,self.attackabl
e_sprites],sprite_type)


    def death_particles(self,particle_type,position):
```

```
self.animation.create_particles(particle_type,position,self.visible_sprites,'death')


    def toggle_inventory(self):
        play_sound(OPEN_INVENTORY_AUDIO)
        self.inventory_display = not self.inventory_display


    def toggle_forge(self):
        play_sound(SWORD_SOUND)
        self.forge_display = not self.forge_display


    def toggle_alchemy(self):
        self.alchemy_display = not self.alchemy_display


    def toggle_quest(self):
        self.quest_display = not self.quest_display


    def coin_spawn(self,position,amount):
        coin = Coin(position,self.visible_sprites,amount)
        self.coins.append(coin)


    def coin_pickup(self):
        if len(self.coins) == 0:
            pass
        else:
            for coin in self.coins:
                if coin.check_collide(self.player.rect):
                    coin.kill()
                    play_sound(COIN_PICKUP_AUDIO)
                    self.coins.remove(coin)
                    self.player.coins += coin.amount


    def add_exp(self,amount):
        self.player.exp += amount


    def add_item(self,item):
        self.inventory.add_item(item)
```

```
class YSortCameraGroup(pygame.sprite.Group):
    def __init__(self):


        super().__init__()
        self.display_surface = pygame.display.get_surface()
        self.half_width = self.display_surface.get_size()[0] // 2
        self.half_height = self.display_surface.get_size()[1] // 2
        self.offset = pygame.math.Vector2()


        # creating the floor
        self.floor_surf = pygame.image.load(".\\Graphics\\map\\map.png").convert()
        self.floor_rect = self.floor_surf.get_rect(topleft=(0,0))


    def custom_draw(self,player):
        #get offset
        self.offset.x = player.rect.centerx - self.half_width
        self.offset.y = player.rect.centery - self.half_height

        #drawing the floor
        floor_offset_position = self.floor_rect.topleft - self.offset
        self.display_surface.blit(self.floor_surf,floor_offset_position)


        for sprite in sorted(self.sprites(), key = lambda sprite: sprite.rect.centery):
            offset_position = sprite.rect.topleft - self.offset
            self.display_surface.blit(sprite.image,offset_position)


    def enemy_update(self,player):
        enemy_sprites = [sprite for sprite in self.sprites() if hasattr(sprite,
'sprite_type') and sprite.sprite_type == 'enemy']
        for sprite in enemy_sprites:
            sprite.enemy_update(player)
```

Enity

```
import pygame
from math import sinclass Entity(pygame.sprite.Sprite):

    def __init__(self,groups):
        super().__init__(groups)
        self.frame_index = 0
        self.animation_speed = 0.15
        self.direction = pygame.math.Vector2()


    def move(self,speed):
        if self.direction.magnitude() != 0:
            self.direction = self.direction.normalize()


        self.hitbox.x += self.direction.x * speed
        self.collision('horizontal')
        self.hitbox.y += self.direction.y * speed
        self.collision('vertical')
        self.rect.center = self.hitbox.center


    def collision(self,direction):
        if direction == 'horizontal':
            for sprite in self.obstacle_sprites:
                if sprite.hitbox.colliderect(self.hitbox):
                    if self.direction.x > 0:
                        self.hitbox.right = sprite.hitbox.left
                    if self.direction.x < 0:
                        self.hitbox.left = sprite.hitbox.right


        if direction == 'vertical':
            for sprite in self.obstacle_sprites:
                if sprite.hitbox.colliderect(self.hitbox):
                    if self.direction.y > 0:
                        self.hitbox.bottom = sprite.hitbox.top
                    if self.direction.y < 0:
                        self.hitbox.top = sprite.hitbox.bottom


    def wave_value(self):
        if sin(pygame.time.get_ticks()) >= 0:
            return 255
        else:
            return 0
```

Player

```
import pygame
from settings import *
from support import *
from entity import Entity


class Player(Entity):
    def 
__init__(self,position,groups,obstacle_sprites,interaction,toggle_forge,toggle_alchemy,togg
le_quest,create_attack,destroy_attack,animation):
        super().__init__(groups)

        self.armour_name = 'simple_clothes'
        self.image = 
pygame.image.load(f'.\\Graphics\\Main_Character\\{self.armour_name}\\forward_idle\\forward_
idle.png').convert_alpha()
        self.rect = self.image.get_rect(topleft = position)
        self.hitbox = self.rect.inflate(-5, HITBOX_SIZE['player'])


        #graphics setup
        self.import_player_assets()
        self.status = 'forward'


        #movement
        self.attacking = False
        self.attack_cooldown = 400
        self.attack_time = None
        self.can_attack = True


        self.obstacle_sprites = obstacle_sprites
        self.interaction = interaction


        #weapon
        self.create_attack = create_attack
        self.destroy_attack = destroy_attack
        self.animation_arrow = animation
        self.weapon_list = weapons
        self.weapon_index = 0
        self.sword_index  = 0
        self.weapon = list(weapons.keys())[self.weapon_index]
        self.allowed_to_switch = True
        self.weapon_switch_time = None
        self.switch_duration_cooldown = 200


        #forge and alchemy
        self.toggle_forge = toggle_forge
        self.toggle_alchemy = toggle_alchemy
        self.toggle_quest = toggle_quest


        #stats
        self.stats = {'health': 100, 'energy': 60, 'attack': 10, 'magic': 5, 'speed': 5,
'level': 1}
        self.max_stats = {'health': 200, 'energy': 100, 'attack': 20, 'magic': 15, 'speed':
8}
        self.health = self.stats['health']
        self.energy = self.stats['energy']
        self.exp_to_level_up = 100
        self.exp = 0
        self.speed = self.stats['speed']
        self.level = self.stats['level']
        self.coins = 0
        self.armour = armour
        self.current_armour = self.armour[self.armour_name]
        self.health_before_armour = self.stats['health']
        for piece in self.current_armour:
            self.stats['health'] += self.armour[self.armour_name][piece]['resistance']


        self.vulnerable = True
        self.hurt_time = None
        self.invulnerable_duration = 600

        self.shielded = False
        self.last_interaction_time = 0


    def import_player_assets(self):
        character_path = f'.\\Graphics\\Main_Character\\{self.armour_name}\\'
        self.animations = {'forward_walking': [], 'back_walking': [], 'left_walking': [],
'right_walking': [],
        'right_idle': [], 'left_idle': [], 'forward_idle': [], 'back_idle': [],
        'right_sword_attack': [], 'left_sword_attack': [], 'back_sword_attack': [],
'forward_sword_attack': [],
        'back_bow_attack' : [], 'forward_bow_attack': [], 'right_bow_attack':[],
'left_bow_attack':[]
        }


        for animation in self.animations.keys():
            full_path = character_path + animation
            self.animations[animation] = import_folder(full_path)



    def input(self):
        if not self.attacking:
            keys = pygame.key.get_pressed()
            current_time = pygame.time.get_ticks()

            if keys[pygame.K_UP] or keys[pygame.K_w]:
                self.direction.y = -1
                self.status = 'back_walking'
            elif keys[pygame.K_DOWN] or keys[pygame.K_s]:
                self.direction.y = 1
                self.status = 'forward_walking'
            else:
                self.direction.y = 0


            if keys[pygame.K_RIGHT] or keys[pygame.K_d]:
                self.direction.x = 1
                self.status = 'right_walking'
            elif keys[pygame.K_LEFT] or keys[pygame.K_a]:
                self.direction.x = -1
                self.status = 'left_walking'
            else:
                self.direction.x = 0


            #speed up
            if keys[pygame.K_LSHIFT]:
                if 'walking' in self.status:
                    self.speed = self.stats['speed'] + 2
            else:
                self.speed = self.stats['speed']

            #attack input
            if keys[pygame.K_e]:
                self.attacking = True
                self.attack_time = pygame.time.get_ticks()
                self.create_attack()
                play_sound(SWORD_ATTACK_SOUND)

            #switching weapons
            if keys[pygame.K_z] and self.allowed_to_switch:
                self.allowed_to_switch = False
                self.weapon_switch_time = pygame.time.get_ticks()
                if self.weapon_index != 1:
                    self.weapon_index = 1
                    self.animation_speed = 0.10
                else:
                    self.weapon_index = self.sword_index
                    self.animation_speed = 0.15
                play_sound(CHANGE_WEAPON)
                self.weapon = list(weapons.keys())[self.weapon_index]


            #interaction
            if keys[pygame.K_RETURN]:
                collided_interaction_sprite = pygame.sprite.spritecollide(self,
self.interaction, False)
                if collided_interaction_sprite:
                    if collided_interaction_sprite[0].name == 'Blacksmith':
                        if current_time - self.last_interaction_time > 300:
                            self.toggle_forge()
                            self.last_interaction_time = current_time



    def get_status(self):
        #idle status
        if self.direction.x == 0 and self.direction.y == 0:
            if 'walking' in self.status:
                self.status = self.status.replace('_walking', '_idle')
            elif not 'idle' in self.status and not 'attack' in self.status:
                self.status = self.status + '_idle'
        # sword attack status
        if self.attacking:
            self.direction.x = 0
            self.direction.y = 0
            if self.weapon_index == 0:
                if not 'attack' in self.status:
                    if 'idle' in self.status:
                        self.status = self.status.replace('_idle', '_sword_attack')
                    elif 'walking' in self.status:
                        self.status = self.status.replace('_walking', '_sword_attack')
                    else:
                        self.status = self.status + '_sword_attack'
            if self.weapon_index == 1:
                if not 'attack' in self.status:
                    if 'idle' in self.status:
                        self.status = self.status.replace('_idle', '_bow_attack')
                    elif 'walking' in self.status:
                        self.status = self.status.replace('_walking', '_bow_attack')
                    else:
                        self.status = self.status + '_bow_attack'
        else:
            if 'attack' in self.status and self.weapon_index == 0:
                self.status = self.status.replace('_sword_attack', '_walking')
            elif 'attack' in self.status and self.weapon_index == 1:
                self.status = self.status.replace('_bow_attack', '_walking')



    def cooldown(self):
        current_time = pygame.time.get_ticks()


        if self.attacking:
            if current_time - self.attack_time >= self.attack_cooldown +
weapons[self.weapon]['damage']:
                self.attacking = False
                self.destroy_attack()

        if not self.allowed_to_switch:
            if current_time - self.weapon_switch_time >= self.switch_duration_cooldown:
                self.allowed_to_switch = True


        if not self.vulnerable:
            if current_time - self.hurt_time >= self.invulnerable_duration:
                self.vulnerable = True


    def animate(self):
        animation = self.animations[self.status]


        self.frame_index += self.animation_speed
        if self.frame_index >= len(animation):
            self.frame_index = 0
        self.image = animation[int(self.frame_index)]
        self.rect = self.image.get_rect(center = self.hitbox.center)


        #flicker
        if not self.vulnerable:
            alpha = self.wave_value()
            self.image.set_alpha(alpha)
        else:
            self.image.set_alpha(255)


    def bow_attack(self,groups):
        if 'forward' in self.status:
            self.animation_arrow.create_particles('arrow_forward',self.rect.center + 
pygame.math.Vector2(0,64),groups,'weapon')
        elif 'back' in self.status:
            self.animation_arrow.create_particles('arrow_back',self.rect.center + 
pygame.math.Vector2(0,-64),groups,'weapon')
        elif 'left' in self.status:
            self.animation_arrow.create_particles('arrow_left',self.rect.center + 
pygame.math.Vector2(-64,0),groups,'weapon')
        elif 'right' in self.status:
            self.animation_arrow.create_particles('arrow_right',self.rect.center + 
pygame.math.Vector2(64,0),groups,'weapon')

    def get_health(self,amount):
        if self.health < self.stats['health']:
            self.health += amount
        if self.health >= self.stats['health']:
            self.health = self.stats['health']


    def health_recovery(self):
        if self.health < self.stats['health']:
            self.health += 0.01
        else:
            self.health = self.stats['health']


    def check_death(self):
        if self.health <= 0:
            return True
        else:
            return False
    def check_level(self):
        return self.level

    def check_for_level_up(self,exp):
        if exp >= self.exp_to_level_up:
            self.level +=1
            self.exp = exp - self.exp_to_level_up
            self.exp_to_level_up *= 1.05
            self.stats['health'] *= 1.02
            self.health_before_armour *= 1.02
            self.stats['energy'] *= 1.01
            self.stats['attack'] *= 1.005
            self.stats['speed'] += 1
            self.stats['magic'] *= 1.005


            if self.stats['health'] >= self.max_stats['health']:
                self.stats['health'] = self.max_stats['health']
            if self.stats['energy'] >= self.max_stats['energy']:
                self.stats['energy'] = self.max_stats['energy']
            if self.stats['attack'] >= self.max_stats['attack']:
                self.stats['attack'] = self.max_stats['attack']
            if self.stats['speed'] >= self.max_stats['speed']:
                self.stats['speed'] = self.max_stats['speed']
            if self.stats['magic'] >= self.max_stats['magic']:
                self.stats['magic'] = self.max_stats['magic']


    def get_weapon_damage(self):
        if self.can_attack:
            full_damage = self.stats['attack'] + weapons[self.weapon]['damage']
        else:
            full_damage = 0
        return full_damage

    def update_armour(self,name):
        self.current_armour = self.armour[name]
        for piece in self.current_armour:
            self.stats['health'] += self.armour[self.armour_name][piece]['resistance']
        self.import_player_assets()


    def update(self):
        self.check_death()
        self.check_for_level_up(self.exp)
        self.health_recovery()
        self.input()
        self.cooldown()
        self.get_status()
        self.animate()
        self.move(self.speed)
```

Magic

```
import pygame
from player import Player
from support import *
from settings import *


class MagicPlayer(Player):
    def __init__(self,position,groups,obstacle_sprites,interaction,toggle_forge,
toggle_inventory,toggle_quest,create_magic_attack,destroy_attack,animation):
        Player.__init__(self,position,groups,obstacle_sprites,interaction,toggle_forge,
toggle_inventory,toggle_quest,create_magic_attack,destroy_attack,animation)

        self.animation_spells = animation
        self.create_magic_attack = create_magic_attack
        self.weapon_list = magic_spells
        self.weapon = list(magic_spells.keys())[self.weapon_index]


        self.shielded = False


    def import_player_assets(self):
        character_path = f'.\\Graphics\\Main_Character\\{self.armour_name}\\'
        self.animations = {
        'forward_walking': [], 'back_walking': [], 'left_walking': [], 'right_walking': [],
        'right_idle': [], 'left_idle': [], 'forward_idle': [], 'back_idle': [],
        'right_magic_attack': [],'left_magic_attack': [], 'back_magic_attack': [],
'forward_magic_attack': []
        }
        for animation in self.animations.keys():
            full_path = character_path + animation
            self.animations[animation] = import_folder(full_path)


    def input(self):
        if not self.attacking:
            keys = pygame.key.get_pressed()
            current_time = pygame.time.get_ticks()


            if keys[pygame.K_UP] or keys[pygame.K_w]:
                self.direction.y = -1
                self.status = 'back_walking'
            elif keys[pygame.K_DOWN] or keys[pygame.K_s]:
                self.direction.y = 1
                self.status = 'forward_walking'
            else:
                self.direction.y = 0


            if keys[pygame.K_RIGHT] or keys[pygame.K_d]:
                self.direction.x = 1
                self.status = 'right_walking'
            elif keys[pygame.K_LEFT] or keys[pygame.K_a]:
                self.direction.x = -1
                self.status = 'left_walking'
            else:
                self.direction.x = 0


            #speed up
            if keys[pygame.K_LSHIFT]:
                if 'walking' in self.status:
                    self.speed = self.stats['speed'] + 2
            else:
                self.speed = self.stats['speed']


            #attack input
            if keys[pygame.K_e]:
                self.attacking = True
                self.attack_time = pygame.time.get_ticks()
                style = list(magic_spells.keys())[self.weapon_index]
                strength = list(magic_spells.values())[self.weapon_index]['strength']
                cost = list(magic_spells.values())[self.weapon_index]['cost']
                self.create_magic_attack(style,strength,cost)
            if keys[pygame.K_n]:
                self.get_health(5)


            #switch spells
            if keys[pygame.K_z] and self.allowed_to_switch:
                self.allowed_to_switch = False
                self.weapon_switch_time = pygame.time.get_ticks()
                if self.weapon_index >= len(list(magic_spells.keys())) - 1 :
                    self.weapon_index = 0
                else:
                    self.weapon_index += 1
                play_sound(CHANGE_WEAPON)
                self.weapon = list(magic_spells.keys())[self.weapon_index]


            #forge
            if keys[pygame.K_RETURN]:
                collided_interaction_sprite = pygame.sprite.spritecollide(self,
self.interaction, False)
                if collided_interaction_sprite:
                    if collided_interaction_sprite[0].name == 'Blacksmith':
                        if current_time - self.last_interaction_time > 300:
                            self.toggle_forge()
                            self.last_interaction_time = current_time


    def get_status(self):
        #idle status
        if self.direction.x == 0 and self.direction.y == 0:
            if 'walking' in self.status:
                self.status = self.status.replace('_walking', '_idle')
            elif not 'idle' in self.status and not 'attack' in self.status:
                self.status = self.status + '_idle'

        #magic attack
        if self.attacking:
            self.direction.x = 0
            self.direction.y = 0
            if not 'attack' in self.status:
                if 'idle' in self.status:
                    self.status = self.status.replace('_idle', '_magic_attack')
                elif 'walking' in self.status:
                    self.status = self.status.replace('_walking', '_magic_attack')
                else:
                    self.status = self.status + '_magic_attack'


        else:
            if 'attack' in self.status:
                self.status = self.status.replace('_magic_attack', '_walking')


    def cooldown(self):
        current_time = pygame.time.get_ticks()


        if self.attacking:
            if current_time - self.attack_time >= self.attack_cooldown +
magic_spells[self.weapon]['strength']:
                self.attacking = False
                self.destroy_attack()


        if not self.allowed_to_switch:
            if current_time - self.weapon_switch_time >= self.switch_duration_cooldown:
                self.allowed_to_switch = True


        if not self.vulnerable:
            if current_time - self.hurt_time >= self.invulnerable_duration:
                self.vulnerable = True


        if self.shielded:
            if not self.attacking:
                self.shielded = False



    def energy_recovery(self):
        if self.energy < self.stats['energy']:
            self.energy += 0.01 * self.stats['magic']
        else:
            self.energy = self.stats['energy']


    def full_magic_damage(self):
        return self.stats['attack'] + magic_spells[self.weapon]['strength']


    def fireball(self,strength,cost,groups):
        self.shielded = False
        if self.energy >= cost:
            self.energy -= cost
            if 'forward' in self.status:
                self.animation_spells.create_particles('fireball_forward',self.rect.center
+ pygame.math.Vector2(0,32),groups,'magic')
            elif 'back' in self.status:
                self.animation_spells.create_particles('fireball_back',self.rect.center +
pygame.math.Vector2(0,-32),groups,'magic')
            elif 'left' in self.status:
                self.animation_spells.create_particles('fireball_left',self.rect.center +
pygame.math.Vector2(-32,2),groups,'magic')
            elif 'right' in self.status:
                self.animation_spells.create_particles('fireball_right',self.rect.center +
pygame.math.Vector2(32,2),groups,'magic')


            play_sound(self.weapon_list['fireball']['audio'])


    def heal(self,strength,cost,groups):
        self.shielded = False
        if self.energy >= cost:
            self.health += strength
            self.energy -= cost
            if self.health >= self.stats['health']:
                self.health = self.stats['health']
            self.animation_spells.create_particles('heal',self.rect.center +
pygame.math.Vector2(0,-5),groups,'magic')

            play_sound(self.weapon_list['heal']['audio'])


    def shield(self,strength,cost,groups):
        if self.energy >= cost:
            self.energy -= cost
            self.shielded = True
            self.animation_spells.create_particles('shield',self.rect.center +
pygame.math.Vector2(0,-5),groups,'magic')
            play_sound(self.weapon_list['shield']['audio'])
        else:
            self.shielded = False

    def icicle(self,strength,cost,groups):
        self.shielded = False
        if self.energy >= cost:
            self.energy -= cost
            if 'forward' in self.status:
                self.animation_spells.create_particles('icicle_forward',self.rect.center +
pygame.math.Vector2(0,38),groups,'magic')
            elif 'back' in self.status:
                self.animation_spells.create_particles('icicle_back',self.rect.center +
pygame.math.Vector2(0,-38),groups,'magic')
            elif 'left' in self.status:
                self.animation_spells.create_particles('icicle_left',self.rect.center +
pygame.math.Vector2(-38,2),groups,'magic')
            elif 'right' in self.status:
                self.animation_spells.create_particles('icicle_right',self.rect.center +
pygame.math.Vector2(38,2),groups,'magic')

            play_sound(self.weapon_list['icicle']['audio'])


    def quake(self,strength,cost,groups):
        self.shielded = False
        if self.energy >= cost:
            self.energy -= cost
            if 'forward' in self.status:
                self.animation_spells.create_particles('quake',self.rect.center +
pygame.math.Vector2(0,80),groups,'magic')
            elif 'back' in self.status:
                self.animation_spells.create_particles('quake',self.rect.center +
pygame.math.Vector2(0,-72),groups,'magic')
            elif 'left' in self.status:
                self.animation_spells.create_particles('quake',self.rect.center +
pygame.math.Vector2(-95,0),groups,'magic')
            elif 'right' in self.status:
                self.animation_spells.create_particles('quake',self.rect.center +
pygame.math.Vector2(95,0),groups,'magic')


            play_sound(self.weapon_list['quake']['audio'])


    def get_damage(self, amount):
        if not self.shielded:
            if self.health > 0:
                self.health -= amount
            if self.health <= 0:
                self.health = 0


    def update(self):
        self.check_death()
        self.check_for_level_up(self.exp)
        self.health_recovery()
        self.energy_recovery()
        self.input()
        self.cooldown()
        self.get_status()
        self.animate()
        self.move(self.speed)
```

Enemy

```
import pygame
from settings import *
from entity import Entity
from support import *
import random


class Enemy(Entity):

    def
__init__(self,name,position,groups,obstacle_sprites,damage_player,attack_particles,death_pa
rticles,coin_spawn,add_exp):

        super().__init__(groups)
        self.sprite_type = 'enemy'


        #graphics setup
        self.import_graphics(name)
        self.status = 'forward_idle'
        self.image = self.animations[self.status][self.frame_index]


        #movement
        self.start_position = position
        self.rect = self.image.get_rect(topleft = position)
        self.hitbox = self.rect.inflate(0, -10)
        self.obstacle_sprites = obstacle_sprites


        #stats
        self.name = name
        enemy_info = enemies[self.name]
        self.health = enemy_info['health']
        self.max_health = enemy_info['health']
        self.exp = enemy_info['exp']
        self.coins = enemy_info['coins']
        self.speed = enemy_info['speed']
        self.attack_damage = enemy_info['damage']
        self.resistance = enemy_info['resistance']
        self.attack_radius = enemy_info['attack_radius']
        self.notice_radius = enemy_info['notice_radius']
        self.attack_type = enemy_info['attack_type']
        self.attack_sound = enemy_info['attack_sound']


        #player interaction
        self.can_attack = True
        self.attack_time = None
        self.attack_cooldown = enemy_info['cooldown']
        self.damage_player = damage_player
        self.attack_particles = attack_particles
        self.death_particles = death_particles
        self.coin_spawn = coin_spawn
        self.add_exp = add_exp


        self.vulnerable = True
        self.hit_time = None
        self.invulnerable_duration = 400


        self.moving = False
        self.last_position = position

        #magic enemy
        self.spells = ['dark_bolt','fire_bomb','lightning']

        self.graph = self.create_graph()
        self.movement_speed = 1.5
        self.movement_radius = 100
        self.target_position = pygame.math.Vector2(random.randint(-50, 50),
random.randint(-50, 50)) + self.hitbox.center


    def create_graph(self):
        graph = {}
        for sprite in self.obstacle_sprites:
            if sprite.sprite_type == 'object':
                node = (sprite.rect.center[0], sprite.rect.center[1])
                graph[node] = []
                for neighbor in self.get_neighbors(sprite):
                    if neighbor.sprite_type == 'object':
                        neighbor_node = (neighbor.rect.center[0], neighbor.rect.center[1])
                        distance = ((node[0] - neighbor_node[0])**2 + (node[1] -
neighbor_node[1])**2)**0.5
                        graph[node].append((neighbor_node, distance))
        return graph


    def get_neighbors(self, sprite):
        neighbours = []
        for obstacle in self.obstacle_sprites:
            if sprite != obstacle:
                if sprite.rect.colliderect(obstacle.rect):
                    neighbours.append(obstacle)
        return neighbours


    def idle(self):
        distance_to_target = self.target_position - self.hitbox.center
        if distance_to_target.magnitude() < 5:
            x = random.uniform(-self.movement_radius, self.movement_radius)
            y = random.uniform(-self.movement_radius, self.movement_radius)
            self.target_position = self.hitbox.center + pygame.math.Vector2(x, y)

        self.direction = (self.target_position - self.hitbox.center).normalize()
        self.move(self.movement_speed)


    def import_graphics(self,name):
        self.animations = {'forward_idle': [], 'left_idle': [], 'right_idle':
[],'back_idle': [],
        'forward_walking': [], 'left_walking': [], 'right_walking': [], 'back_walking': [],
        'forward_attack' : [], 'left_attack': [], 'right_attack': [], 'back_attack': [] }
        main_path = f'.\\Graphics\\Enemies\\{name}\\'


        for animation in self.animations.keys():
            self.animations[animation] = import_folder(main_path + animation)


    def get_direction_distance_player(self, player):
        enemy_vec = pygame.math.Vector2(self.rect.center)
        player_vec = pygame.math.Vector2(player.rect.center)
        distance = (player_vec - enemy_vec).magnitude()


        if distance > 0:
            direction = (player_vec - enemy_vec).normalize()
        else:
            direction = pygame.math.Vector2()

        return(distance,direction)

    def get_status(self,player):
        distance = self.get_direction_distance_player(player)[0]
        direction = self.get_direction_distance_player(player)[1]


        if distance <= self.attack_radius and self.can_attack:
            if not 'attack' in self.status:
                self.frame_index = 0
            if round(direction.y) == -1:
                self.status = 'back_attack'
            elif round(direction.y) == 1:
                self.status = 'forward_attack'
            elif round(direction.x) == -1:
                self.status = 'left_attack'
            elif round(direction.x) == 1:
                self.status = 'right_attack'


        elif distance <= self.attack_radius and not self.can_attack:
            if round(direction.y) == -1:
                self.status = 'back_idle'
            elif round(direction.y) == 1:
                self.status = 'forward_idle'
            elif round(direction.x) == -1:
                self.status = 'left_idle'
            elif round(direction.x) == 1:
                self.status = 'right_idle'


        elif distance <= self.notice_radius:
            if round(direction.y) == -1:
                self.status = 'back_walking'
            elif round(direction.y) == 1:
                self.status = 'forward_walking'
            elif round(direction.x) == -1:
                self.status = 'left_walking'
            elif round(direction.x) == 1:
                self.status = 'right_walking'


        elif distance <= 1300 and distance > self.notice_radius:
            self.idle()


        else:
            if 'walking' in self.status:
                self.status  = self.status.replace('_walking', '_idle')
            elif 'attack' in self.status:
                self.status = self.status.replace('_attack','_idle')



    def action(self,player):
        if 'attack' in self.status and self.health > 0:
            self.attack_time = pygame.time.get_ticks()
            if self.attack_type == 'magic':
                self.magic_attack(player)
            else:
                self.damage_player(self.attack_damage,self.attack_type)
            play_sound(self.attack_sound)
        elif 'walking' in self.status and self.health > 0:
            self.direction = self.get_direction_distance_player(player)[1]
        else:
            self.direction = pygame.math.Vector2()


    def magic_attack(self,player):
        if not self.check_movement() and self.can_attack:
            spell = self.spells[random.randint(0,2)]


            if spell == 'dark_bolt':
                self.attack_particles(spell, player.rect.center + pygame.math.Vector2(0,-
20), 'magic')
            elif spell == 'fire_bomb':
                self.attack_particles(spell, player.rect.center, 'magic')
            else:
                self.attack_particles(spell, player.rect.center + pygame.math.Vector2(0,-
50), 'magic')


            self.can_attack = False
            self.damage_player(self.attack_damage,self.attack_type)


    def animate(self):
        animation = self.animations[self.status]


        self.frame_index += self.animation_speed
        if self.frame_index >= len(animation):
            if 'death' in self.status:
                self.kill()
            elif 'attack' in self.status:
                self.can_attack = False
            self.frame_index = 0


        self.image = animation[int(self.frame_index)]
        self.rect = self.image.get_rect(center = self.hitbox.center)


        if not self.vulnerable:
            alpha = self.wave_value()
            self.image.set_alpha(alpha)
        else:
            self.image.set_alpha(255)

    def cooldown(self):
        current_time = pygame.time.get_ticks()
        if not self.can_attack:
            if current_time - self.attack_time >= self.attack_cooldown:
                self.can_attack = True


        if not self.vulnerable:
            if current_time -self.hit_time >= self.invulnerable_duration:
                self.vulnerable = True


    def get_damage(self,player,attack_type):
        if self.vulnerable:
            self.direction = self.get_direction_distance_player(player)[1]
            if attack_type == 'weapon' or attack_type == 'bow':
                self.health -= player.get_weapon_damage()
            elif attack_type == 'magic':
                self.health -= player.full_magic_damage()
        self.hit_time = pygame.time.get_ticks()
        self.vulnerable = False
    def check_death(self,player):
        if self.health <= 0:
            self.kill()
            self.add_exp(self.exp)
            self.death_particles('death',self.rect.center)
            self.coin_spawn(self.rect.center,self.coins)


    def hit_reaction(self):
        if not self.vulnerable:
            self.direction *= -self.resistance


    def check_movement(self):
        if self.rect.topleft == self.last_position:
            self.last_position = self.rect.topleft
            return False
        else:
            self.last_position = self.rect.topleft
            return True


    def update(self):
        self.hit_reaction()
        self.move(self.speed)
        self.animate()
        self.cooldown()


    def enemy_update(self, player):
        self.check_death(player)
        self.get_status(player)
        self.action(player)
```

GUI

```
import pygame
from button import Button
from settings import *
class GUI:
    def __init__(self):
        self.items = []
        self.item_size = 55
        self.slot_size = 70
        self.inventory_slot_image = pygame.image.load(SLOT_IMAGE)
        self.inventory_slot_image = pygame.transform.scale(self.inventory_slot_image, 
(self.slot_size, self.slot_size))
        self.start_x = 150
        self.start_y = 150
        self.slot_spacing = 5
        self.main_font = pygame.font.Font(UI_FONT,25)
        self.description_font = pygame.font.Font(UI_FONT, 15)
        self.last_click = 0
        self.last_remove = 0


    def draw_slots_and_items(self, is_inventory = True, is_in_shop = False, shop = None):
        if is_inventory:
            desc_box_offset = 50
            items_surface = self.main_font.render('Items', True, FONT_COLOUR)
            items_rect = items_surface.get_rect(topleft=(325, 105))
            self.display_surface.blit(items_surface, items_rect)
        elif not is_inventory:
            desc_box_offset = 200
            items_surface = self.main_font.render('Shop', True, FONT_COLOUR)
            items_rect = items_surface.get_rect(topleft=(965, 65))
            self.display_surface.blit(items_surface, items_rect)


        for i in range(self.capacity):
            x_offset = self.slot_spacing + ((self.slot_spacing + self.slot_size) * (i % 6)) 
+ self.start_x
            y_offset = self.slot_spacing + ((self.slot_spacing + self.slot_size) * (i // 
6)) + self.start_y


            slot_rect = self.inventory_slot_image.get_rect(center=(x_offset + 
self.slot_size // 2, y_offset + self.slot_size // 2))
            self.display_surface.blit(self.inventory_slot_image, slot_rect)


            if i < len(self.items) and self.items[i] is not None:
                item_graphic = pygame.image.load(self.items[i].graphic)
                item_graphic = pygame.transform.scale(item_graphic, (self.item_size, 
self.item_size))
                self.item_slots[i] = item_graphic
                item_rect = item_graphic.get_rect(center = slot_rect.center)
                self.display_surface.blit(item_graphic, item_rect)


                mouse_x, mouse_y = pygame.mouse.get_pos()
                if item_rect.collidepoint(mouse_x, mouse_y):
                    desc_rect = pygame.Rect(x_offset-desc_box_offset, y_offset-100, 400,
80)
                    desc_surface = pygame.Surface(desc_rect.size)
                    desc_surface.fill(DESC_BOX_COLOUR)
                    name = self.description_font.render(self.items[i].name.title(), True,
DESC_FONT_COLOUR)
                    cost = self.description_font.render(f"Cost: {self.items[i].cost}",
True, DESC_FONT_COLOUR)
                    description = self.description_font.render(self.items[i].description,
True, DESC_FONT_COLOUR)
                    desc_surface.blit(name, (10, 10))
                    desc_surface.blit(cost,(10,30))
                    desc_surface.blit(description,(10,50))
                    self.display_surface.blit(desc_surface, desc_rect)
                    pygame.draw.rect(self.display_surface, UI_BORDER_COLOUR_2, desc_rect,
2)



                    if pygame.mouse.get_pressed()[0]:
                        if pygame.time.get_ticks() - self.last_click < 500:
                            if is_in_shop:
                                shop.add_item(self.items[i])
                                self.coins += self.items[i].cost
                                self.remove_item(i)
                            elif not is_inventory:
                                self.sell_item(i)
                            elif is_inventory and not is_in_shop:
                                if self.items[i].type == 'armour':
                                    self.equip_armour(i)
                                elif self.items[i].type == 'weapon':
                                    self.equip_weapon(i)
                        self.last_click = pygame.time.get_ticks()
            elif i < len(self.items) and self.items[i] is None:
                if pygame.time.get_ticks() - self.last_remove > 500:
                    self.items = [item for item in self.items if item is not None]
                    self.last_remove = pygame.time.get_ticks()



    def add_item(self,item):
        if len(self.items) < self.capacity:
            self.items.append(item)
            return True
        else:
            return False


    def remove_item(self,index):
        self.items[index] = None
        self.item_slots[index] = None
```

Inventory

```
import pygame
from gui import GUI
from settings import *
from player import Player
from magic import MagicPlayer
from item import Item
from support import play_sound


class Inventory(GUI):
    def __init__(self, player):
        super().__init__()
        self.background = pygame.image.load(INVENTORY_BG)
        self.display_surface = pygame.display.get_surface()
        self.player = player
        self.capacity = 30
        self.items = []
        self.equipped_armour = Item(self.player.armour_name,'armour')
        self.equipped_weapon = Item('base_sword', 'weapon')
        self.main_font = pygame.font.Font(UI_FONT, 22)
        self.description_font = pygame.font.Font(UI_FONT, 15)


        self.armour = armour
        self.weapons = weapons
        self.weapon_key_list = list(weapons.keys())
        self.spells = magic_spells
        self.item_slots = [None] * self.capacity
        self.start_x = 150
        self.start_y = 150


        self.coins = player.coins


    def draw_armour_pieces(self):
        armour_surface = pygame.image.load(ARMOUR_SLOT)
        armour_surface = pygame.transform.scale(armour_surface, (280,350))
        armour_rect = armour_surface.get_rect(topleft = (900,80))
        self.display_surface.blit(armour_surface,armour_rect)


        rect_sizes = [
        (100, 84), # mid-top rectangle
        (100, 88), # center rectangle
        (100, 84) # mid-bottom rectangle
        ]


        x_offset = 1001
        y_offset = [160, 248, 336]


        for index, (slot, item) in enumerate(self.player.current_armour.items()):
            if item and item['graphic'] is not None:
                item_graphic = pygame.image.load(item['graphic'])
                item_graphic = pygame.transform.scale(item_graphic, (80,80))
                rect = pygame.Rect(x_offset,y_offset[index], rect_sizes[index][0],
rect_sizes[index][1])
                self.display_surface.blit(item_graphic, rect)


                mouse_x, mouse_y = pygame.mouse.get_pos()
                if rect.collidepoint(mouse_x, mouse_y):
                    x = x_offset - item_graphic.get_width() - 20
                    y = y_offset[index] - index * 80
                    desc_rect = pygame.Rect(x, y, 245, 35)
                    desc_surface = pygame.Surface(desc_rect.size)
                    desc_surface.fill(DESC_BOX_COLOUR)
                    res_info = self.description_font.render(f"Resistance:
{item['resistance']}", True, DESC_FONT_COLOUR)
                    desc_surface.blit(res_info, (10, 10))
                    self.display_surface.blit(desc_surface, desc_rect)
                    pygame.draw.rect(self.display_surface, UI_BORDER_COLOUR_2, desc_rect, 
2)



        if isinstance(self.player, Player) and not isinstance(self.player, MagicPlayer):
            self.draw_current_weapons()


    def draw_current_weapons(self):
        current_weapons = []
        current_weapons.append(self.weapons[list(weapons.keys())[self.player.sword_index]])
        current_weapons.append(self.weapons[list(weapons.keys())[1]])

        rect_sizes = [
            (81, 88),  # mid-left rectangle
            (78, 88)  # mid-right rectangle
        ]


        x_offset = [923, 1100]
        y_offset = 260


        for i in range(len(rect_sizes)):
            if i < len(current_weapons) and current_weapons[i]['graphic'] is not None:
                image = pygame.image.load(current_weapons[i]['graphic'])
                image = pygame.transform.scale(image, (60,60))
                rect = pygame.Rect(x_offset[i],y_offset, rect_sizes[i][0], 
rect_sizes[i][1])
                self.display_surface.blit(image, rect)


            mouse_x, mouse_y = pygame.mouse.get_pos()
            if rect.collidepoint(mouse_x, mouse_y):
                x = x_offset[i] - 275
                y = y_offset - image.get_height() - i * 80
                desc_rect = pygame.Rect(x, y, 275, 55)
                desc_surface = pygame.Surface(desc_rect.size)
                desc_surface.fill(DESC_BOX_COLOUR)
                damage_info = self.description_font.render(f"Attack damage: 
{current_weapons[i]['damage']}", True, DESC_FONT_COLOUR)
                radius_info = self.description_font.render(f"Attack radius: 
{current_weapons[i]['attack_radius']}", True, DESC_FONT_COLOUR)
                desc_surface.blit(damage_info, (10, 10))
                desc_surface.blit(radius_info,(10,30))
                self.display_surface.blit(desc_surface, desc_rect)
                pygame.draw.rect(self.display_surface, UI_BORDER_COLOUR_2, desc_rect, 2)


    def draw_coins(self):
        icon = pygame.image.load(COIN_ICON_1)
        icon = pygame.transform.scale(icon,(50,50))
        icon_rect = icon.get_rect(topright = (1270,5))
        self.display_surface.blit(icon,icon_rect)
        text_surface = self.description_font.render(str(self.player.coins), False,
FONT_COLOUR)
        text_rect = text_surface.get_rect(topleft = (1180,20))
        self.display_surface.blit(text_surface,text_rect)


    def draw_player_stats(self):
        start_pos = 445
        stats = [
            (f"Level: {str(self.player.level)}", (0, 25)),
            (f"Exp: {round(self.player.exp)}/{round(self.player.exp_to_level_up)}", (0,
50)),
            (f"Health: {round(self.player.health)}/{round(self.player.stats['health'])}",
(0, 75)),
            (f"Energy: {round(self.player.energy)}/{round(self.player.stats['energy'])}",
(0, 100)),
            (f"Speed: {round(self.player.stats['speed'])}", (0, 125))
            ]
        for i, (stat_text, offset) in enumerate(stats):
            stat_surface = self.description_font.render(stat_text, False, FONT_COLOUR)
            stat_rect = stat_surface.get_rect(topleft=(905, start_pos + i * 25))
            self.display_surface.blit(stat_surface, stat_rect)


        if isinstance(self.player, Player) and not isinstance(self.player, MagicPlayer):
            player_attack = self.description_font.render( f"Attack:
{round(self.player.stats['magic'])}", False, FONT_COLOUR)
            player_attack_rect = player_attack.get_rect(topleft = (905,570))
            full_damage = self.description_font.render(f"Full attack damage:
{round(self.player.get_weapon_damage())}", False, FONT_COLOUR)
            full_damage_rect = full_damage.get_rect(topleft = (905, 595))
        else:
            player_attack = self.description_font.render( f"Magic Attack:
{round(self.player.stats['magic'])}", False, FONT_COLOUR)
            player_attack_rect = player_attack.get_rect(topleft = (905,570))
            full_damage = self.description_font.render(f"Full attack damage:
{round(self.player.full_magic_damage())}", False, FONT_COLOUR)
            full_damage_rect = full_damage.get_rect(topleft = (905, 595))
        self.display_surface.blit(player_attack,player_attack_rect)
        self.display_surface.blit(full_damage,full_damage_rect)


    def draw_spells(self):
        self.spell_rects = {}
        x = 125
        y = 585
        for spell_name, spell_info in self.spells.items():
            rect = pygame.Rect(x, y, 80, 80)
            self.spell_rects[spell_name] = rect
            pygame.draw.rect(self.display_surface, UI_BG_COLOUR,rect)
            pygame.draw.rect(self.display_surface, UI_BORDER_COLOUR_2, rect,2)
            image = pygame.image.load(spell_info['graphic'])
            image_rect = image.get_rect(center = rect.center)
            self.display_surface.blit(image, image_rect)
            x += 105

    def draw_equipped_armour(self):
        text = self.main_font.render('Armour', False, FONT_COLOUR)
        text_rect = text.get_rect(topleft =(660,215))
        self.display_surface.blit(text,text_rect)


        inventory_image = pygame.transform.scale(self.inventory_slot_image,(80,80))
        inventory_item_rect = inventory_image.get_rect(x = 685, y = 260,
width=self.slot_size, height=self.slot_size)
        self.display_surface.blit(inventory_image, inventory_item_rect)


        armour_graphic = pygame.image.load(self.equipped_armour.graphic)
        armour_graphic = pygame.transform.scale(armour_graphic,(70,70))
        armour_rect = armour_graphic.get_rect(x = 685, y = 260, center =
inventory_item_rect.center + pygame.math.Vector2(5,6))
        self.display_surface.blit(armour_graphic, armour_rect)


    def equip_armour(self,index):
        armour_to_equip = self.items[index]
        self.items[index] = self.equipped_armour
        self.equipped_armour = armour_to_equip
        self.player.armour_name = self.equipped_armour.name
        self.player.update_armour(self.equipped_armour.name)
        play_sound(ARMOUR_EQUIP_AUDIO)
    def equip_weapon(self,index):
        if isinstance(self.player, Player) and not isinstance(self.player, MagicPlayer):
            previuos_weapon = self.equipped_weapon
            self.equipped_weapon = self.items[index]
            equipped_weapon_index = self.weapon_key_list.index(self.items[index].name)
            self.player.sword_index = equipped_weapon_index
            self.items[index] = previuos_weapon
            play_sound(SWORD_SOUND)


    def draw_spells_description(self):
        mouse_pos = pygame.mouse.get_pos()


        for spell_name, rect in self.spell_rects.items():
            if rect.collidepoint(mouse_pos):
                desc_rect = pygame.Rect(rect.right + 10, rect.top, 200, 80)


                spell = magic_spells[spell_name]
                name = spell['name']
                strength = spell['strength']
                cost = spell['cost']


                desc_surface = pygame.Surface(desc_rect.size)
                desc_surface.fill(DESC_BOX_COLOUR)
                name_text = self.description_font.render(f"{name}", True, DESC_FONT_COLOUR)
                strength_text = self.description_font.render(f"Strength: {strength}", True, 
DESC_FONT_COLOUR)
                cost_text = self.description_font.render(f"Cost: {cost}", True, 
DESC_FONT_COLOUR)
                desc_surface.blit(name_text, (10, 10))
                desc_surface.blit(strength_text, (10, 30))
                desc_surface.blit(cost_text, (10, 50))


                self.display_surface.blit(desc_surface, desc_rect)
                pygame.draw.rect(self.display_surface, UI_BORDER_COLOUR_2, desc_rect, 2)


    def display(self):
        self.display_surface.blit(self.background, (0,0))

        self.draw_slots_and_items()
        self.draw_equipped_armour()
        self.draw_armour_pieces()
        self.draw_coins()
        self.draw_player_stats()


        if not (isinstance(self.player, Player) and not isinstance(self.player, 
MagicPlayer)):
            self.draw_spells()
            self.draw_spells_description()

```

Blacksmith

```
import pygame
from gui import GUI
from inventory import Inventory
from item import Item
from settings import *
from support import play_sound


class Blacksmith(GUI):
    def __init__(self, inventory):
        super().__init__()
        self.background = pygame.image.load(BLACKSMITH_BG)
        self.display_surface = pygame.display.get_surface()
        self.capacity = 36
        self.items = []
        self.item_slots = [None] * self.capacity
        self.player_inventory = inventory
        self.start_x = 780
        self.start_y = 120


        for item_type, items in item_attributes.items():
            for name, attributes in items.items():
                item = Item(name, item_type)
                if not (name == 'simple_clothes' or name == 'base_sword'):
                    if item not in self.player_inventory.items:
                        self.items.append(item)

    def remove_item(self, item_index):
        self.items[item_index].cost = round(self.items[item_index].cost * 0.85)
        self.items[item_index] = None
        self.item_slots[item_index] = None
    def draw_player_inventory(self):
        self.player_inventory.draw_slots_and_items(True, True, self)
        self.player_inventory.draw_coins()


    def sell_item(self, item_index):
       item = self.items[item_index]
       if self.player_inventory.player.coins >= item.cost:
            self.player_inventory.add_item(item)
           self.player_inventory.player.coins -= item.cost
            self.remove_item(item_index)
            self.last_click = 0
            play_sound(SELL_AUDIO)
       return None


    def display(self):
        self.display_surface.blit(self.background, (0,0))
        self.draw_slots_and_items(False)
        self.draw_player_inventory()
```

Item

```
from settings import *


class Item:
    def __init__(self,name,type):
        self.name = name
        self.type = type
        self.stockable = type != 'weapon' and type != 'armour'
        self.cost = 0
        self.description = ""


        self.set_attributes()




    def set_attributes(self):
        item_attributes_access = item_attributes.get(self.type, {}).get(self.name, {})
        self.description = item_attributes_access.get('description', '')
        self.cost = item_attributes_access.get('cost', None)
        self.graphic = item_attributes_access.get('graphic', None)
```

Quest

```
import pygame
from settings import *
from support import play_sound


class Quest:
    def __init__(self,player):
        self.player = player
        self.background = pygame.image.load(QUEST_BACKGROUND)
        self.display_surface = pygame.display.get_surface()
        self.font = pygame.font.Font(UI_FONT, 15)
        self.quests = quests
        self.current_quest = None
        self.completed = []



    def add_quest(self):
        for quest_key, quest_value in quests.items():
            if self.current_quest == None and not
self.quests[quest_key]['objective']['completed']:
                self.current_quest = quest_key
            elif self.quests[quest_key]['objective']['completed'] and quest_key not in
self.completed:
                self.completed.append(quest_key)
                self.current_quest = None


    def set_completed(self):
        if self.current_quest == 'first_steps':
            if self.player.coins >= 5:
                self.quests[self.current_quest]['objective']['completed'] = True
                play_sound(QUEST_COMPLETED)
        elif self.current_quest == 'protection':
            if self.player.armour_name == 'leather_armour':
                self.quests[self.current_quest]['objective']['completed'] = True
                play_sound(QUEST_COMPLETED)
    def get_rewards(self):
        for reward_key, reward_value in quests['first_steps']['reward'].items():
            if reward_key == 'exp':
                self.player.exp += reward_value
            elif reward_key == 'coins':
                self.player.coins += reward_value

    def draw_avaliable_quests(self):
        headline = self.font.render('Current quest', True, QUEST_HEADLINE_COLOUR)
        headline_rect = headline.get_rect(topleft=(340, 250))
        self.display_surface.blit(headline, headline_rect)


        name = self.font.render(self.quests[self.current_quest]['name'], True,
QUEST_FONT_COLOUR)
        name_rect = name.get_rect(topleft = (340,295))
        self.display_surface.blit(name, name_rect)


        lines = ["Objective:", self.quests[self.current_quest]['objective']['text'],
self.quests[self.current_quest]['description'],
        "Reward:", f"{self.quests[self.current_quest]['reward']['exp']} exp,
{self.quests[self.current_quest]['reward']['coins']} coins"]
        line_height = 40
        y = name_rect.bottom + 20


        for line in lines:
            text_surface = self.font.render(line, True, QUEST_FONT_COLOUR)
            text_rect = text_surface.get_rect(topleft=(340, y))
            self.display_surface.blit(text_surface, text_rect)
            y += line_height



    def draw_complete_quests(self):
        x_offset = 735
        y_offset = 250
        headline = self.font.render('Completed quests', True, QUEST_HEADLINE_COLOUR)
        headline_rect = headline.get_rect(topleft=(x_offset, y_offset))
        self.display_surface.blit(headline, headline_rect)


        for quest in self.completed:
            y_offset += 30
            name = self.font.render(self.quests[quest]['name'], True, QUEST_FONT_COLOUR)
            name_rect = name.get_rect(topleft = (x_offset,y_offset))
            self.display_surface.blit(name,name_rect)



    def display(self):
        self.display_surface.blit(self.background, (250,85))
        self.draw_avaliable_quests()
        self.draw_complete_quests()
        self.set_completed()
```

UI

```
import pygame
from settings import *
from support import play_sound


class UI:
    def __init__(self):

        self.display_surface = pygame.display.get_surface()
        self.font = pygame.font.Font(UI_FONT, UI_FONT_SIZE)


        #bar setup
        self.exp_bar_rect = pygame.Rect(96,20,BAR_WIDTH,BAR_HEIGHT)
        self.health_bar_rect = pygame.Rect(96,40,BAR_WIDTH,BAR_HEIGHT)
        self.energy_bar_rect = pygame.Rect(96,60,BAR_WIDTH,BAR_HEIGHT)

        self.weapon_graphics = []

    def import_graphics(self,type):
        for graphic in type.values():
            path = graphic['graphic']
            weapon = pygame.image.load(path).convert_alpha()
            self.weapon_graphics.append(weapon)


    def get_font(self, size):
        return pygame.font.Font(UI_FONT, size)


    def display_stats_image(self):
        image = pygame.image.load(STATS_IMAGE)
        image_rect = image.get_rect(topleft = (20,15))


        self.display_surface.blit(image,image_rect)


    def display_bar(self,current,max_amount,bg_rect,color):
        pygame.draw.rect(self.display_surface,UI_BG_COLOUR,bg_rect)


        #stat to pixel
        ratio = current/ max_amount
        current_width = bg_rect.width * ratio
        current_rect = bg_rect.copy()
        current_rect.width = current_width


        pygame.draw.rect(self.display_surface,color,current_rect)


    def display_level(self,level):
        text_surface = self.font.render(str(level), False, TEXT_COLOUR)
        text_rect = text_surface.get_rect(center =(57,48))


        self.display_surface.blit(text_surface,text_rect)


    def display_coins(self,amount):
        icon = pygame.image.load(COIN_ICON)
        icon_rect = icon.get_rect(topright = (1250,25))
        text_surface = self.font.render(str(amount),False, TEXT_COLOUR)
        text_rect = text_surface.get_rect(topright = (1210,35))


        self.display_surface.blit(icon,icon_rect)
        self.display_surface.blit(text_surface,text_rect)


    def selection_box(self,left,top,box_size,switched):
        bg_rect = pygame.Rect(left,top,box_size,box_size)
        pygame.draw.rect(self.display_surface,UI_BG_COLOUR,bg_rect)
        pygame.draw.rect(self.display_surface,UI_BORDER_COLOUR,bg_rect,2)
        if switched:
            pygame.draw.rect(self.display_surface,UI_BORDER_COLOUR_ACTIVE,bg_rect,2)
        else:
            pygame.draw.rect(self.display_surface,UI_BORDER_COLOUR,bg_rect,2)
        return bg_rect


    def weapon_overlay(self,weapon_index,switched):
        bg_rect = self.selection_box(20,630,80,switched)
        weapon_surface = self.weapon_graphics[weapon_index]
        weapon_rect = weapon_surface.get_rect(center = bg_rect.center)


        self.display_surface.blit(weapon_surface,weapon_rect)


    def display(self, player):
        self.display_stats_image()
        self.display_bar(player.exp, player.exp_to_level_up,self.exp_bar_rect, EXP_COLOUR)

self.display_bar(player.health,player.stats['health'],self.health_bar_rect,HEALTH_COLOUR)

self.display_bar(player.energy,player.stats['energy'],self.energy_bar_rect,ENERGY_COLOUR)


        self.display_level(player.level)
        self.weapon_overlay(player.weapon_index,player.allowed_to_switch)
        self.display_coins(player.coins)
```

Weapon

```
import pygame


class Weapon(pygame.sprite.Sprite):
    def __init__(self,player,groups):
        super().__init__(groups)


        self.sprite_type = 'weapon'
        #player direction
        direction = player.status.split('_')[0]


        #graphics
        full_path = f'.\\Graphics\\Weapons\\{player.weapon}\\{direction}.png'
        self.image = pygame.Surface((20,20))
        #placement
        if direction == 'forward':
            self.rect = self.image.get_rect(midtop = player.rect.midbottom +
pygame.math.Vector2(0,-10))
        elif direction == 'back':
            self.rect = self.image.get_rect(midbottom = player.rect.midtop +
pygame.math.Vector2(0,10))
        elif direction == 'right':
            self.rect = self.image.get_rect(midleft = player.rect.midright)
        elif direction == 'left':
            self.rect = self.image.get_rect(midright = player.rect.midleft)
```

Interaction

```
import pygame


class Interaction(pygame.sprite.Sprite):
    def __init__(self, pos, size, groups, name):
        super().__init__(groups)
        self.image = pygame.Surface(size)
        self.rect = self.image.get_rect(topleft=pos)
        self.hitbox = self.rect.copy().inflate(-self.rect.width * 0.2, -self.rect.height *
0.75)
        self.name = name
```

Support

```
from csv import reader
from os import walk
import pygame
from settings import *
import json
from item import Item




def import_csv_layout(path):
    terrain_map = []


    with open(path) as level_map:
        layout = reader(level_map, delimiter = ',')
        for row in layout:
            terrain_map.append(list(row))
        return terrain_map


def import_folder(path):
    surface_list = []


    for folder,subfolders,image_files in walk(path):
        for image in image_files:
            full_path = path + '\\' + image
            image_surface = pygame.image.load(full_path).convert_alpha()
            surface_list.append(image_surface)


    return surface_list


def play_sound(path):
    music = pygame.mixer.Sound(path)
    music.set_volume(effects_volume)
    music.play()



class ItemEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Item):
            return obj.__dict__
        return json.JSONEncoder.default(self, obj)

class ItemDecoder(json.JSONDecoder):
    def __init__(self, *args, **kwargs):
        super().__init__(object_hook=self.object_hook, *args, **kwargs)


    def object_hook(self, dct):
        if '__type__' in dct and dct['__type__'] == 'item':
            return Item(**dct)
        return dct
```

Tile

```
import pygame
from settings import *


class Tile(pygame.sprite.Sprite):
    def __init__(self, position, groups, sprite_type, surface = 
pygame.Surface((TILESIZE,TILESIZE))):
        super().__init__(groups)
        self.sprite_type = sprite_type
        self.image = surface
        y_offset = HITBOX_SIZE[sprite_type]
        self.rect = self.image.get_rect(topleft = position)
        self.hitbox = self.rect.inflate(0,y_offset)
```

Particles

```
import pygame
from support import import_folder


class Animation:
    def __init__(self):
        self.frames = {
            #spells
            'fireball_forward': import_folder('.\\Graphics\\Spells\\fireball\\forward'),
            'fireball_back': import_folder('.\\Graphics\\Spells\\fireball\\back'),
            'fireball_left': import_folder('.\\Graphics\\Spells\\fireball\left'),
            'fireball_right': import_folder('.\\Graphics\\Spells\\fireball\\right'),
            'heal':import_folder('.\\Graphics\\Spells\\heal'),
            'shield' : import_folder('.\\Graphics\\Spells\\shield'),
            'icicle_forward' : import_folder('.\\Graphics\\Spells\\icicle\\forward'),
            'icicle_back' : import_folder('.\\Graphics\\Spells\\icicle\\back'),
            'icicle_left' : import_folder('.\\Graphics\\Spells\\icicle\\left'),
            'icicle_right' : import_folder('.\\Graphics\\Spells\\icicle\\right'),
            'quake': import_folder('.\\Graphics\\Spells\\quake'),
            #arrow
            'arrow_forward': 
import_folder('.\\Graphics\\Particle_Effects\\arrow\\forward'),
```

```python
            'arrow_back': import_folder('.\\Graphics\\Particle_Effects\\arrow\\back'),
            'arrow_left': import_folder('.\\Graphics\\Particle_Effects\\arrow\\left'),
            'arrow_right': import_folder('.\\Graphics\\Particle_Effects\\arrow\\right'),
            #other
            'death': import_folder('.\\Graphics\\Particle_Effects\\death'),
            #enemy spells
            'dark_bolt' : import_folder('.\\Graphics\\Particle_Effects\\dark_bolt'),
            'fire_bomb' : import_folder('.\\Graphics\\Particle_Effects\\fire_bomb'),
            'lightning' : import_folder('.\\Graphics\\Particle_Effects\\lightning')
        }


    def create_particles(self,animation_type,pos,groups,sprite_type):
        animation_frames = self.frames[animation_type]
        ParticleEffect(pos,animation_frames,groups,sprite_type)



class ParticleEffect(pygame.sprite.Sprite):
    def __init__(self,pos,animation_frames,groups,sprite_type):
        super().__init__(groups)
        self.sprite_type = sprite_type
        self.frame_index = 0
        if sprite_type == 'death':
            self.animation_speed = 0.29
        elif 'fireball' in self.sprite_type:
            self.animation_speed = 0.18
        elif 'arrow' in self.sprite_type:
            self.animation_speed = 0.09
        else:
            self.animation_speed = 0.15
        self.frames = animation_frames
        self.image = self.frames[self.frame_index]
        self.rect = self.image.get_rect(center = pos)


    def animate(self):
        self.frame_index += self.animation_speed
        if self.frame_index >= len(self.frames):
            self.kill()
        else:
            self.image = self.frames[int(self.frame_index)]


    def update(self):
        self.animate()
```

Coin

```python
import pygame
from support import *
class Coin(pygame.sprite.Sprite):
    def __init__(self,position,groups,amount):
        super().__init__(groups)
        self.frames = import_folder('.\\Graphics\\Particle_Effects\\coin')
        self.frame_index = 0
        self.animation_speed = 0.15
        self.image = self.frames[self.frame_index]
        self.rect = self.image.get_rect(center = position)
        self.amount = amount




    def animate(self):
        self.frame_index += self.animation_speed
        if self.frame_index >= len(self.frames):
            self.frame_index = 0
        else:
            self.image = self.frames[int(self.frame_index)]


    def check_collide(self,player_rect):
        if player_rect.colliderect(self.rect):
            return True
        else:
            return False


    def update(self):
        self.animate()
```

Button

```python
import pygame
class Button():
    def __init__(self, image, pos, text_input, font, base_colour, hovering_colour):
        self.image = image
        self.x_pos = pos[0]
        self.y_pos = pos[1]
        self.font = font
        self.base_colour, self.hovering_colour = base_colour, hovering_colour
        self.text_input = text_input
        self.text = self.font.render(self.text_input, True, self.base_colour)
        if self.image is None:
            self.image = self.text
        self.rect = self.image.get_rect(center=(self.x_pos, self.y_pos))
        self.text_rect = self.text.get_rect(center=(self.x_pos, self.y_pos))

    def update(self, screen):
        if self.image is not None:
            screen.blit(self.image,self.rect)
        screen.blit(self.text, self.text_rect)

    def checkForInput(self, pos):
        if pos[0] in range(self.rect.left, self.rect.right)and pos[1] in
range(self.rect.top, self.rect.bottom):
            return True
        return False


    def changeColour(self, pos):
        if pos[0] in range(self.rect.left, self.rect.right)and pos[1] in
range(self.rect.top, self.rect.bottom):
            self.text = self.font.render(self.text_input, True, self.hovering_colour)
        else:
            self.text = self.font.render(self.text_input, True, self.base_colour)


    def drawRectAround(self,screen,pos):
        if self.image is not None:
            if pos[0] in range(self.rect.left, self.rect.right)and pos[1] in
range(self.rect.top, self.rect.bottom):
                pygame.draw.rect(screen,'white',self.rect,3)
        else:
            pass
```

Settings

```
# game setup
WIDTH    = 1280
HEIGHT   = 720
FPS      = 60
TILESIZE = 32
MAIN_BACKGROUND = '.\\bg_2.png'
BACKGROUND_2  = '.\\bg.png'
```

```
MAP_SIZE = 10240


HITBOX_SIZE = {
    'player': -3.5,
    'object': -20,
    'floor_blocks': 0
}


#UI
INVENTORY_BG = '.\\bg_3.png'
BLACKSMITH_BG = '.\\Graphics\\UI\\blacksmith_bg.png'
ARMOUR_SLOT = '.\\Graphics\\UI\\armour.png'
SLOT_IMAGE = '.\\Graphics\\UI\\slot_1.png'
MENU_TEXT_COLOUR = '#d7fcd4'
CURSOR= '.\\Graphics\\UI\\cursor.png'
SWORD_ICON = '.\\Graphics\\UI\\icons\\sword_underline_scaled.png'
MAGIC_ICON = '.\\Graphics\\UI\\icons\\smoke_scaled.png'
BAR_HEIGHT = 12
BAR_WIDTH = 104
ITEM_BOX_SIZE = 80
UI_FONT = "PokemonGb-Raeo.ttf"
UI_FONT_SIZE = 18
STATS_IMAGE = '.\\Graphics\\UI\\health_exp_mana_lvl_1.png'
INVENTORY_ICON = '.\\Graphics\\UI\\inventory_icon.png'
COIN_ICON = '.\\Graphics\\UI\\coin_icon_1.png'
COIN_ICON_1 = '.\\Graphics\\UI\\coin_icon.png'
QUEST_BACKGROUND = '.\\Graphics\\UI\\quest_bg.png'


UI_BG_COLOUR = '#4e4a4e'
UI_BORDER_COLOUR = '#111111'
UI_BORDER_COLOUR_2 = '#2e2b2b'
TEXT_COLOUR = '#eeeeee'
EXP_COLOUR = '#597dce'
HEALTH_COLOUR = '#d04648'
ENERGY_COLOUR = '#812cee'
UI_BORDER_COLOUR_ACTIVE = '#a6680a'
DESC_BOX_COLOUR = '#413e41'
FONT_COLOUR = '#ffd1a4'
DESC_FONT_COLOUR = '#eec7a0'
QUEST_HEADLINE_COLOUR = '#351b11'
QUEST_FONT_COLOUR = '#4f2b1c'


UI_SLIDER_START_VALUE = 50
```

```
UI_SLIDER_LENGTH = 300
#audio
MENU_AUDIO = '.\\Audio\\booya_b.wav'
OPTIONS_AUDIO = '.\\Audio\\monoliths.wav'
GAMEPLAY_AUDIO = '.\\Audio\\To_Be_Tacribian.wav'
DEATH_AUDIO = '.\\Audio\\TragicDeath.wav'
MENU_SELECT_AUDIO = '.\\Audio\\MENU_Select.wav'
QUEST_COMPLETED = '.\\Audio\\win_music.wav'
COIN_PICKUP_AUDIO = '.\\Audio\\hjm-coin_clicker_1.wav'
OPEN_INVENTORY_AUDIO = '.\\Audio\\leather_inventory.wav'
ARMOUR_EQUIP_AUDIO = '.\\Audio\\cloth_inventory.wav'
SELL_AUDIO = '.\\Audio\\sell_buy_item.wav'
CHANGE_WEAPON = '.\\Audio\\MENU_Pick.wav'
SWORD_SOUND = '.\\Audio\\metal_clash.wav'
SWORD_ATTACK_SOUND = '.\\Audio\\Socapex _new_hits_3.wav'


effects_volume = 0.3


weapons = {
    'base_sword': {'cooldown': 100, 'damage' : 25,'attack_radius': 10, 'graphic':
'.\\Graphics\\UI\\weapons\\basic_sword_scaled.png'},
    'bow': {'cooldown': 150, 'damage' : 8, 'attack_radius': 35, 'graphic':
'.\\Graphics\\UI\\weapons\\bow.png'},
    'sword_upgrade_1': {'cooldown': 150, 'damage' : 25, 'attack_radius': 10, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_1.png'},
    'sword_upgrade_2': {'cooldown': 200, 'damage' : 45, 'attack_radius': 15, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_2.png'},
    'sword_upgrade_3': {'cooldown': 300, 'damage' : 55, 'attack_radius': 20, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_3.png'},
}


magic_spells = {
    'fireball': {'name': 'Fireball','strength': 10, 'cost': 5, 'audio':
'.\\Audio\\fireball.wav', 'graphic': '.\\Graphics\\UI\\spells\\fireball.png'},
    'heal' : {'name': 'Healing','strength': 15, 'cost': 10, 'audio': '.\\Audio\\heal.wav',
'graphic': '.\\Graphics\\UI\\spells\\heal.png'},
    'shield': {'name': 'Shield','strength': 5, 'cost': 15, 'audio': '.\\Audio\\shield.wav',
'graphic': '.\\Graphics\\UI\\spells\\shield.png'},
    'icicle': {'name': 'Icicle','strength': 25, 'cost': 35, 'audio':
'.\\Audio\\icicle.wav','graphic': '.\\Graphics\\UI\\spells\\icicle.png'},
    'quake': {'name': 'Earthquake','strength': 65, 'cost': 50, 'audio':
'.\\Audio\\quake.wav','graphic': '.\\Graphics\\UI\\spells\\quake.png'}
}
```

```
armour = {


    'simple_clothes':
    {
     'head': {'resistance': 0, 'graphic': None},
     'top': {'resistance': 2, 'graphic': '.\\Graphics\\Armour\\simple_clothes\\top.png'},
     'bottom': {'resistance': 2, 'graphic':
'.\\Graphics\\Armour\\simple_clothes\\bottom.png'}
    },


    'leather_armour' :
    {
     'head': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\leather_armour\\head.png'},
     'top': {'resistance': 15, 'graphic': '.\\Graphics\\Armour\\leather_armour\\top.png'},
     'bottom': {'resistance': 2, 'graphic':
'.\\Graphics\\Armour\\leather_armour\\bottom.png'}
    },

    'robe' :
    {
     'head': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\robe\\head.png'},
     'top': {'resistance': 8, 'graphic': '.\\Graphics\\Armour\\robe\\top.png'},
     'bottom': {'resistance': 5, 'graphic': '.\\Graphics\\Armour\\robe\\bottom.png'}
    },


    'chain_armour':
    {
     'head': {'resistance': 15, 'graphic': '.\\Graphics\\Armour\\chain_armour\\head.png'},
     'top': {'resistance': 45, 'graphic': '.\\Graphics\\Armour\\chain_armour\\top.png'},
     'bottom': {'resistance': 15, 'graphic': '.\\Graphics\\chain_armour\\bottom.png'}
    },


    'chain_armour_robe':
    {
     'head': {'resistance': 10, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\head.png'},
     'top': {'resistance': 50, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\top.png'},
     'bottom': {'resistance': 10, 'graphic':
'.\\Graphics\\Armour\\chain_armour_robe\\bottom.png'}
    },
    'plate_armour':
    {
     'head': {'resistance': 55, 'graphic': '.\\Graphics\\Armour\\plate_armour\\head.png'},
     'top': {'resistance': 55, 'graphic': '.\\Graphics\\Armour\plate_armour\\top.png'},
     'bottom': {'resistance': 55, 'graphic':
'.\\Graphics\\Armour\\plate_armour\\bottom.png'}
    }
}


enemies = {
    'skeleton_sword' : {'health' : 100, 'exp': 15, 'coins': 1, 'damage': 5, 'attack_type' :
'slash', 'attack_sound': '.\\Audio\\Socapex_ new_hits_1.wav','speed' : 2, 'resistance' : 4,
'attack_radius': 50, 'notice_radius': 360, 'cooldown': 500},
    'skeleton_magic' : {'health' : 125, 'exp': 25, 'coins': 5, 'damage': 27, 'attack_type'
: 'magic', 'attack_sound': '.\\Audio\\Magic_Smite.wav','speed' : 4, 'resistance' : 3,
'attack_radius': 100, 'notice_radius': 360, 'cooldown': 1800},
    'mage': {'health' : 150, 'exp': 35, 'coins': 10, 'damage': 32, 'attack_type' : 'magic',
'attack_sound': '.\\Audio\\Magic_Smite.wav','speed' : 4, 'resistance' : 4, 'attack_radius':
130, 'notice_radius': 360, 'cooldown': 1500},
    'baldric' : {'health' : 200, 'exp': 75, 'coins': 15,'damage': 40, 'attack_type' :
'slash', 'attack_sound': '.\\Audio\\Socapex_ new_hits_2.wav','speed' : 4, 'resistance' : 4,
'attack_radius': 60, 'notice_radius': 360, 'cooldown': 500}
}



quests= {
    'first_steps': {'name': 'First Steps', 'objective': {'text': 'Collect 5 coins',
'completed': False}, 'description' : 'Money is money', 'reward': {'exp': 20, 'coins': 0}},
    'protection': {'name': 'Protection', 'objective': {'text': 'Buy leather armour',
'completed': False}, 'description' : 'Increase resistance', 'reward': {'exp': 35, 'coins':
5}}


}


item_attributes = {
    'weapon': {
        'base_sword': {'description': 'A basic sword', 'cost': 5, 'graphic':
'.\\Graphics\\UI\\weapons\\basic_sword.png'},
        'sword_upgrade_1': {'description': 'Upgraded sword', 'cost': 15, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_1.png'},
        'sword_upgrade_2': {'description': 'Even more upgraded sword', 'cost': 20,
'graphic': '.\\Graphics\\UI\\weapons\\sword_2.png'},
        'sword_upgrade_3': {'description': 'Ultimate sword upgrade', 'cost': 35, 'graphic':
'.\\Graphics\\UI\\weapons\\sword_3.png'},
    },

    'armour': {
        'simple_clothes': {'description': 'Basic armour', 'cost': 2, 'graphic':
'.\\Graphics\\Armour\\simple_clothes\\full_armour.png'},
        'leather_armour': {'description': 'Light and flexible armour', 'cost': 20,
'graphic': '.\\Graphics\\Armour\\leather_armour\\full_armour.png'},
        'robe': {'description': 'A simple robe for mages', 'cost': 15, 'graphic':
'.\\Graphics\\Armour\\robe\\full_armour.png'},
        'chain_armour': {'description': 'Heavy chain armour', 'cost': 45, 'graphic':
'.\\Graphics\\Armour\\chain_armour\\full_armour.png'},
        'chain_armour_robe': {'description': 'Upgraded robe for mages', 'cost': 45,
'graphic': '.\\Graphics\\Armour\\chain_armour_robe\\full_armour.png'},
        'plate_armour': {'description': 'Thick and heavy armour', 'cost': 70, 'graphic':
'.\\Graphics\\Armour\\plate_armour\\full_armour.png'},
    }
}
```

4 Testing

Link to the videos with game footage:

https://youtu.be/yDPE1SJq3dI

https://www.youtube.com/watch?v=zAEtHycA7OA

Target methods	Expected Output	Output	Successful
Game: main_menu	Display the main menu of the game and change the colour of the button once the user hovers over the button. Once the button is pressed the	Main menu scree is displayed, and buttons change colour once the user hovers over them. The screen changes depending on	**Yes**

	screen changes to the appropriate user choice	which button is pressed	
Game: options	Sliders of music volume and sound effects volume are displayed and the volume of the music changes in the game	Sliders work and intended and the volume of the music and sound effects changes in the program	Yes
Game: choose_character	Rectangles are drawn on top of button images and the class of the player depends on the choice of the user	Rectangles are draws on top of buttons and the player's class is established by user's choice	Yes
Game: save	The data of the player is saved in a dictionary. Sound is made every time the user presses save button	Works as expected	Yes
Game:load	Load values of the dictionary saved in save() and initialise them to the appropriate game values	A glitch has been found where loading works, but not every time	No
Game: pause	Pauses the game and displays buttons. Music in the program changes	Works as intended	Yes
Game: death_screen	Display death screen once the player dies and load previous saved file if 'load previous save file' button is pressed. Returns to main menu if 'exit to main_menu' button is pressed	Displays the death screen and load and main menu buttons work as expctected	Yes

Level: create_map	Draw obstacles, entities and interaction sprites on the map	Draws obstacles entities, interaction sprites on the map	Yes
Level: set_player_type	Initialise the player with the appropriate class depending on the user choice	Works as intended	Yes
Level: player_attack	Check collision between player’ weapon/spell sprites and enemy sprite, if occurs deal damage to the enemy	Works as expected	
Level: coin_pickup	Check if the sprite of the coin collided with the sprite of the player, if so kill coin sprite and increase player’s coins by the amount of the coin picked	Works as expected	Yes
Level: display_inventoty, display_forge, display_quest	Display inventory, quest and shop UI depending on the user input	Works as intended	Yes
YSortCameraGroup: enemy_update	Update the methods in the enemy class that need to be interacted with the player	Works as intended	Yes
YSortCameraGroup: custom_draw	Draw the floor surface and the sprites	Works as intended	Yes
MagicPlayer: all of the spell methods	Create a particle which direction will be dependent on the direction of the player	Works as intended	
Quest: set_completed	Set quest to completed is objective(s) met	The quest is set to complete once the	Yes

		player opens the quest display	
Quest: add_quest	Adds the next avaliable quest in the dictionary	The quest is added from the dictionary	Yes
Quest: get_rewards	Add certain amount of exp and coins to the player if the quest complete	Exp and coins are added to the player once the quest is complete	Yes
Blacksmith: sell_item	Check if the player has enough coins to buy th item, if so add item to the player’s inventory and remove from the Blacksmith class. Additionally, decrease the item’s cost by 15%	Works as intended	Yes
Invetory: equip_armour	Update armour of the player and draw an image of the armour on top of the armour slot	Works as intended	Yes
Inventory: equip_weapon	Draw an image of the weapon in one of the armour slots and set the player’s weapon to the equipped the index of the key in the weapons dictionary	Works as intended	
UI: weapon_overlay	Display current chosen weapon or spell as HUD	Slight glitch when buying new weapon and replacing the old one since if old weapon is currently chosen it is displayed as HUD. Once the user siwtches the weapons the image of	Yes

		the old weapon dissapears	
ItemEncoder	Encode values of Item class in the Inventory list	Works as intended	Yes
ItemDecoder	Decode values of Item class in Inventory list	Works as intended	Yes
GUI:draw_slots_and_items	Display an appropriate inventory depending on where and when the method is called	Works as intended	Yes

5 Evaluation

The most important lesson I have learn from doing this project is that lack of resources was my biggest struggle. Not only did I have to code the game, but I also had to work on graphics too. Even though most of the graphics were taken from other resources, I had to create some of the graphics on my own because I could not find free graphics for specific things. Most of the stuff I did myself included user interface, HUD and some of the particle animations. Lack of resources also includes lack of time. RPG games generally have a wide range of functionality. This includes enemies, quests, inventories, NPCs etc. Trying to code all of those in a limited amount of time required a lot of effort and time put in the game.

Primary objective Category	**Completion**	**Comment**
1) Basics	85%	Every basic element is satisfied except having multiple save-slots and saving sometimes gives an error. An algorithm could have been used to save and load the game
2) The Character	100%	One of the first elements created. All of the requirements are met. Only two

		choices in the beginning of the game due to lack of resources.
3) UI	100%	One of the most time-consuming pats of the project. The reason is simply because of searching for the right assets and making them look good in the game and because of creation of multiple classes such as UI or Inventory.
4) Enemies	100%	Also, one of the first elements created. All requirements are met. More enemies could have been added if not for the lack of resources.
5) Inventory	100%	All of the requirements are met. Everything works as intended. More weapons and armour could have been added. Another type of Item could have been added such as ‘potion’ because only MagicPlayer has the ability to heal.
6) Shop	100%	Works as intended. More weapons and armour options could have been added. Weapons could have had durability as an attribute and the durability could have been fixed when visiting Blacksmith. Another shop class, Alchemist, could have been added to buy potions.

7) Quests	100%	Works as intended. More quests could have been added. Different types of quests could have been added e.g. Main and Side.

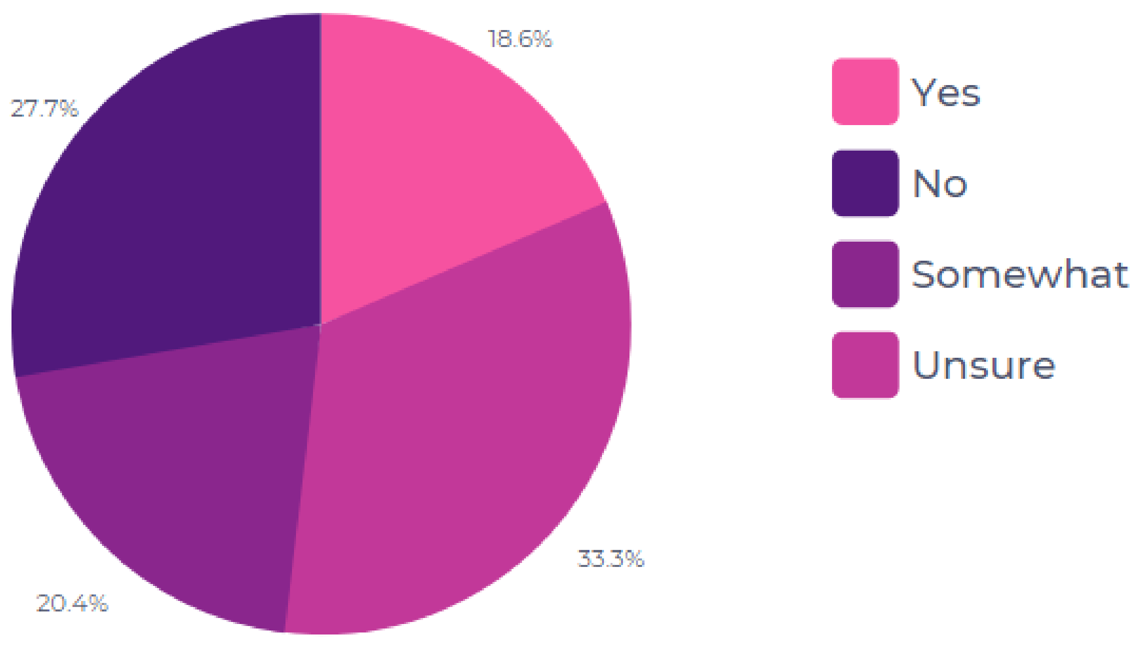
User Feedback

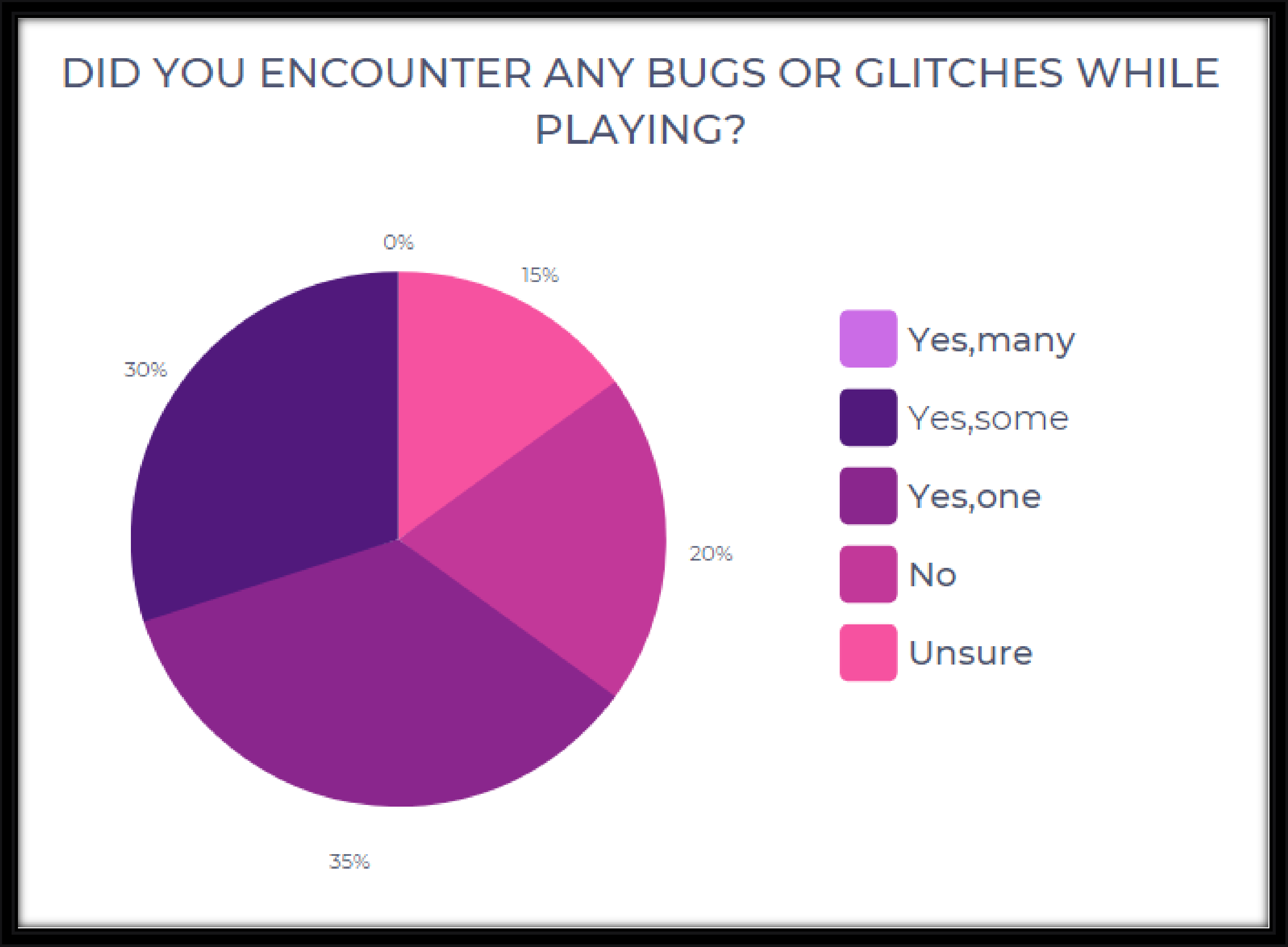
I asked 20 people of different backgrounds (those who have played an RPG before and those who didn't) for feedback once they have played the game.

Questions asked:

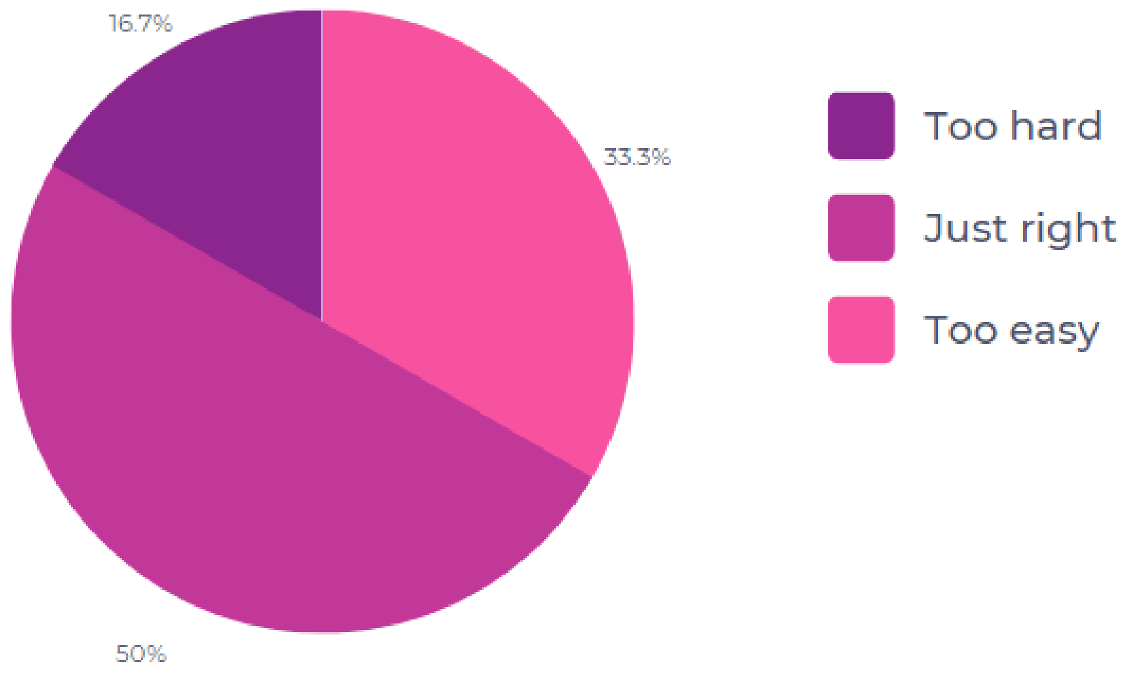
- Were the rules clear and easy to understand, or were they confusing and hard to follow?
- Did you encounter any bugs or glitches while playing?
- Was the difficulty level appropriate, or was the game too easy or too hard?
- Were there any particular features or mechanics that you enjoyed or didn't enjoy?
- Did you feel like the game balanced combat and non-combat elements well?
- Were there any parts of the game that felt repetitive or tedious?
- Did you feel like the game had a clear sense of progression, or did it feel stagnant?
- Would you recommend this game to a friend? If so, why? If not, why not?

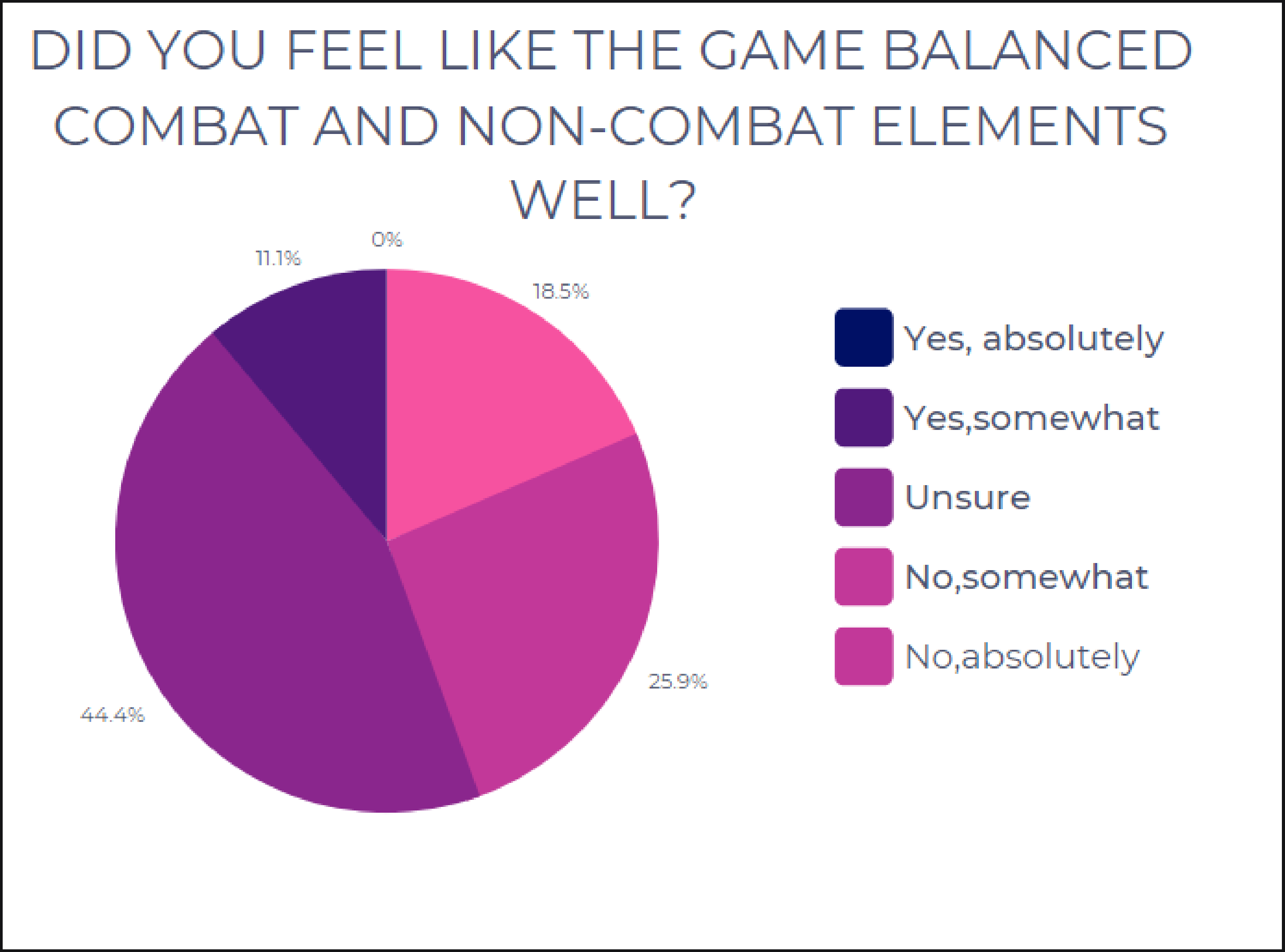
WERE THE RULES CLEAR AND EASY TO UNDERSTAND, OR WERE THEY CONFUSING AND HARD TO FOLLOW?

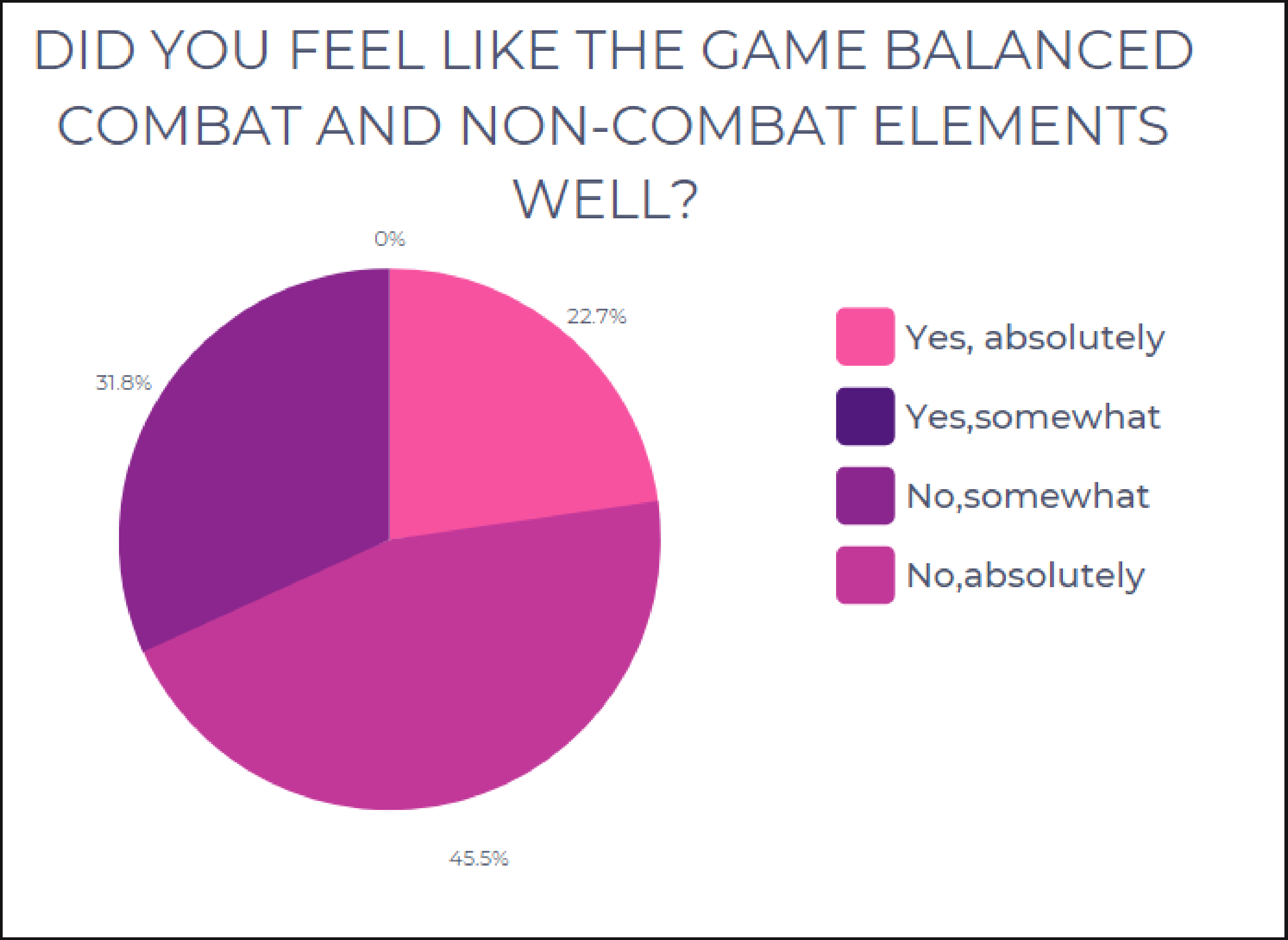



DID YOU ENCOUNTER ANY BUGS OR GLITCHES WHILE PLAYING?
0%
15%
30%
20%
35%
Yes,many
Yes,some
Yes,one
No
Unsure

WAS THE DIFFICULTY LEVEL APPROPRIATE, OR WAS THE GAME TOO EASY OR TOO HARD?




DID YOU FEEL LIKE THE GAME BALANCED COMBAT AND NON-COMBAT ELEMENTS WELL?
0%
11.1%
18.5%
25.9%
44.4%
Yes, absolutely
Yes,somewhat
Unsure
No,somewhat
No,absolutely



Were there any particular features or mechanics that you enjoyed or didn't enjoy?

Most users said they enjoyed the game; however, they wished the game had more quests to do to feel like the progress present in RPGs has been made.

Were there any parts of the game that felt repetitive or tedious?

Some users suggested that the combat system is not very interesting and becomes boring quite quickly

Did you feel like the game had a clear sense of progression, or did it feel stagnant?

Most users agreed that the game feels like it has some sort of progression, however most wiched to have more progression.

Would you recommend this game to a friend? If so, why? If not, why not?

Most users would recommend the game to their friend(s) and some even wished to play the game together and suggested multiplayer system.

My Comments:

Overall, the feedback is expected because most RPG have a lot of game functionality that keeps the player going, although most users agreed that Neo's Enchanting Adventures had good RPG elements

Sourses used:

https://www.youtube.com/watch?v=QU1pPzEGrqw&list=PLoYazmRdF4YGYs-YqAG3Y09j_eBKF4VXx&index=52

i