Project Monster Hunter: Cartoon

By

Ricky Chen

Contents

Analysis

Problem Identification

Introduction of the game

The game is designed with simplicity in mind, offering players an accessible entry point by tasking them with the initial goal of hunting down and defeating various monsters. These monsters are thoughtfully scattered across different corners of the expansive game world, adding an element of exploration to the gameplay. As players progress, they'll find themselves drawn into a deeper narrative, which unveils an intriguing objective: the quest to discover and catalog all species of monsters. Ultimately, their ultimate aim will be to locate the elusive exit door, offering a tantalizing escape from the perilous confines of this realm.

To facilitate the player's journey, the combat system is intentionally straightforward, consisting of just three distinct actions: Work, Attack, and Magic. The protagonist possesses both health points (HP) and magic points (MP), introducing a layer of strategic decision-making as they must manage these resources wisely to overcome challenges and foes.

In addition to the core gameplay elements, the game features a purchasing system that allows players to acquire unique tools and items. For instance, the coveted "Magic Gourd" tool grants players the ability to capture monsters, adding an exciting layer of complexity and strategy to their monster-hunting endeavors.

Within this dynamic game world, players will encounter a diverse cast of non-playable characters (NPCs) who are essential to the game's immersive experience. Through an intuitive dialogue system, players can engage in meaningful interactions with these NPCs by selecting from a variety of dialogue options, enabling them to shape the narrative and uncover hidden secrets of the game world. This rich blend of gameplay elements ensures that players are in for an engaging and unforgettable gaming experience.

Planning

I have planned my pygame project about what it may include:

- install PyGame environment
- Designing game window (size)
- Design the menu (include buttons and basic images)
- Create an empty environment
- Design the player (sprite)
- Movement development for main character (UP/DOWN.. etc)
- Methods(Health, attacks- (*inventory(later)*)
- Make items including sprites of weapons(*universal item with key stats that change*) and extra items(potion)
- Each item has characteristics(damage or attack speed)
- Enemy npc class (with adjustable characteristics, sprite, health attack, damage, attack speed, name)
- Friendly npc, dialogue system
- Inventory which is in character class
- Level design and impemtaion
- Saving each level
- Ending
- Pause maybe

Hopefully there will be some features or variables that can be used:

- Combat (attack)
- Movement
- Enemies (including bosses)
- NPC (friendly)
- Different levels
- Saving progress
- Health
- Weapons
- Inventory system
- Items (healing)
- Dialogue system (choices)
- Puzzles
- Different endings
- Difficulties
- Pause

Stakeholders

This delightful game boasts characteristics that make it an ideal choice for children to enjoy. With its charming cartoon aesthetic and a simple, engaging storyline, it offers a user-friendly experience that doesn't demand lightning-fast reflexes or complex strategic thinking. The majority of monsters pose little challenge, making it accessible and enjoyable for young players who relish the thrill of collecting special tools and unique monsters as they progress. Additionally, the opportunity to forge connections with the endearing NPCs within the game world adds a layer of social interaction, allowing children to immerse themselves in a heroic role. The ultimate achievement of defeating all monsters and finding the exit door provides a satisfying sense of accomplishment for young players.

However, this game also offers depth for more experienced players. The versatility of customizable maps and varying difficulty levels caters to those seeking a higher level of challenge. Once players have mastered the game's control system, they can experiment with more intricate strategies to conquer the game. For example, opting for a purely magical approach to overcome each stage can prove to be a rewarding tactical choice, introducing a fresh and engaging experience for seasoned players. In essence, this game strikes a balance between accessibility for children and the potential for high-level play, making it a delightful and versatile gaming experience for a wide range of players.

For Developers:

For me, it is a challenge but also a good chance to give other people a really great experience on my game, if they could enjoy, I will be pleased as well.

For the game community:

The players of the game could be passionate and engaged, making them important stakeholders. Their feedback and support influence the direction of the series, and the developer will often take their preferences into account when developing new games.

Research the problem

Similar examples or source I can use

Drawing inspiration from a game like "Hollow Knight" is an excellent approach to refining and enhancing your own game. "Hollow Knight" is well-known for its engaging gameplay, challenging difficulty levels, and rich world-building, which have garnered it a dedicated fanbase. Here are some aspects you might consider adopting or adapting from "Hollow Knight" for your own game:

Atmospheric World Design: "Hollow Knight" is praised for its immersive, interconnected world. Consider creating a captivating game world with interconnected areas, secrets to discover, and a rich lore that players can delve into.

Challenging Gameplay: While you mentioned that your game is suitable for children with easy gameplay, you might introduce higher difficulty levels for more experienced players, akin to how "Hollow Knight" offers progressively challenging encounters.

Upgrade Systems: "Hollow Knight" features an upgrade system where players can enhance their character's abilities. You could implement a similar mechanic to add depth and progression to your game.

Boss Battles: Boss fights are a hallmark of "Hollow Knight." Incorporating challenging and memorable boss battles could add excitement and variety to your game.

Art Style: The hand-drawn, dark, and atmospheric art style of "Hollow Knight" contributes to its unique charm. Consider refining your game's visual style to create a distinctive and appealing look.

Exploration: Encourage players to explore your game world thoroughly, rewarding them with hidden treasures, secrets, or special encounters.

Engaging NPCs: "Hollow Knight" has a cast of intriguing NPCs. Develop your own memorable characters that players can interact with to enrich the narrative and offer quests or assistance.

Dynamic Soundtrack: The music in "Hollow Knight" complements the game's atmosphere. A well-crafted soundtrack can enhance the emotional impact of your game.

Storytelling: "Hollow Knight" tells its story subtly through environmental storytelling and character interactions. Consider weaving an engaging narrative into your game in a similar fashion.

Community Engagement: "Hollow Knight" has a dedicated fan community. Engage with your players, gather feedback, and consider post-launch updates or expansions to keep the game fresh.

Computational solution

Pygame would be a good choice to make this game, as there are following advantages:

Ease of Learning: Pygame's simplicity and clear documentation make it an excellent choice for those new to game development. It provides a great entry point for learning the fundamentals of game programming.

Community and Resources: The Pygame community is active, and there are plenty of tutorials, forums, and resources available to help you along the way. This can be incredibly valuable as you work on your game.

Cross-Platform: Python and Pygame are cross-platform, meaning you can develop games that can run on various operating systems without major modifications.

CSV and Map Creation: As you mentioned, Pygame's capability to work with CSV files can be extremely useful for designing and implementing maps and levels in your game. This simplifies the process of creating and editing game content.

Tiled: Tiled is an excellent choice for creating maps and graphical assets, as it's specifically designed for 2D games. Its ability to export maps to CSV format can seamlessly integrate with your Pygame project.

Pixel Art: If your game has a pixel art style, Pygame is well-suited for rendering pixel graphics and animations. It allows for precise control over pixel placement and movement.

Python's Versatility: Python is a versatile language with a wide range of libraries and frameworks that can complement Pygame. You can use libraries like NumPy for mathematical operations, Pygame's built-in sound and music support, and more to enhance your game.

Community Support: Given Python's popularity, you'll find support and resources not only in the Pygame community but also in the broader Python community. This can be beneficial when dealing with more advanced programming challenges.

Limitations

Python limitations:

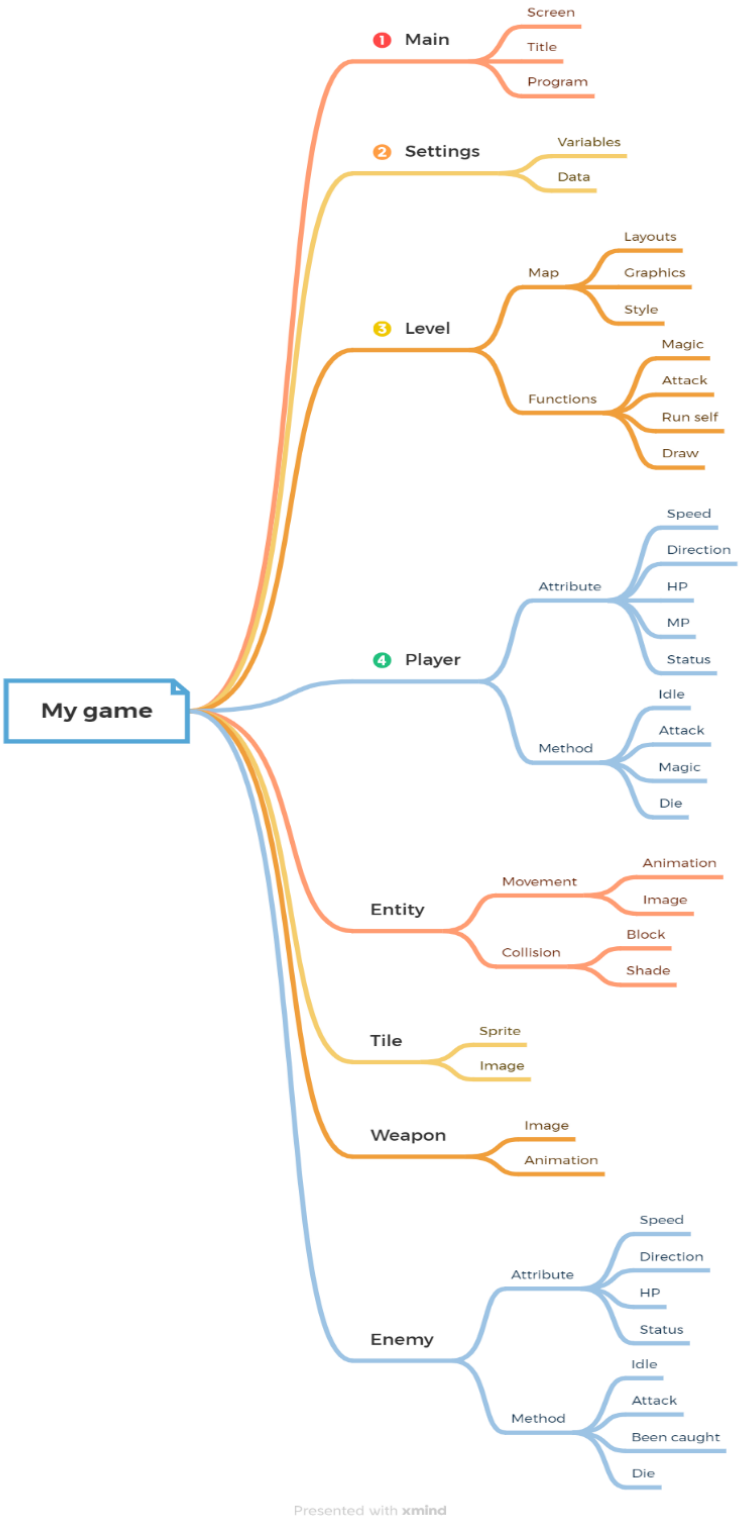
1. **Performance**: Python is an interpreted language, which means it can be slower compared to compiled languages like C++ or C#. This can be a limitation for resource-intensive games that require high frame rates or complex calculations.
2. **GIL (Global Interpreter Lock)**: Python has a GIL, which can limit the ability to effectively use multiple CPU cores for parallel processing. This can impact the performance of multithreaded game code.
3. **Limited Low-Level Access**: Python abstracts many low-level details, which can be a limitation when you need fine-grained control over hardware resources or system-specific optimizations.
4. **Large Memory Footprint**: Python's memory usage can be relatively high, which may be a concern for games with extensive assets or when targeting resource-constrained platforms.
5. **Less Common in AAA Games**: While Python is commonly used in indie and smaller game development projects, it's less common in large AAA game studios due to performance considerations.

Pygame limitations:

1. **2D Graphics Only**: Pygame is primarily designed for 2D game development. If you want to create 3D games, you'll need to use a different game engine or library.
2. **Limited Built-in Features**: Pygame provides a basic set of tools for game development but lacks some of the advanced features and tools found in more specialized game engines.
3. **Less Efficient for Complex Games**: While Pygame is great for simple and moderately complex 2D games, it may not be the best choice for highly complex or graphically intensive games due to performance limitations.
4. **Limited Platform Support**: Pygame's support for platforms other than Windows, macOS, and Linux can be less robust, making cross-platform development more challenging.
5. **Community Size**: While there is an active Pygame community, it is smaller compared to communities around more popular game engines, which may mean fewer resources and plugins available.
6. **Learning Curve**: While Pygame is beginner-friendly, it may not provide the depth of learning and experience that more complex game engines offer, which could limit your ability to transition to larger projects.

Design

Based on the analysis, there are some points that I have to add into my program and following is a logic graph to show them:



Create pygame:

IT iFirst and foremost, it is essential to consider the fundamental requirements for a game. A game inherently necessitates a user interface to function effectively. Consequently, we must develop a screen that can proficiently showcase all the game's contents. This initial phase not only involves the creation of the display screen but also entails assigning a suitable name to the game.

```
Import pygame

Procedure InitializeGame():
    Call PygameInit()
    Set screen = CreatePygameDisplay(WIDTH, HEIGHT)
    SetWindowTitle("Monster Hunter: World (Cartoon)")
    Set clock = CreatePygameClock()
    Set level = CreateLevelObject()
InitializeGame()
```

Run pygame:

Run the game. The previous step is mainly about different settings used. This stage can run the algorithm of the game, plus all the other algorithms in other parts, e.g. level part.

```
Procedure RunGame():
    Repeat Forever:
        For Each Event in PygameEvents:
            If Event is QuitEvent:
                Call PygameQuit()
                Call SysExit()

        Fill Screen with Color "Black"
        Call LevelRun()
        Update Display
        Delay for a Fixed Number of Frames per Second

If Main Program:
    Create Game Object
    Call RunGame()
```

Set basic variables:

A lot of global and private variables will be needed for the game, so listing all variables in a specific module together is necessary. After setting all of them, when I want to use the variables I just need to import the variable module, which I plan to call it “settings”.

Width

Height

FPS

TileSize

Colour

Cooldown

HP

MP

Direction

Status

WeaponType

MonsterType

(Something I may need)

Create the map:

The map of a game is very important because it not only shows the background story of the game, but also decides the way of playing. In my game, I decide to draw maps by using pixels. I find a software which is really helpful in this part called “Tiled”.

Tiled is a free and open source, easy to use, and flexible level editor.

It has a lot of advantages including:

Flexible Object Layers

- Annotate the level with rectangles, ellipses or polygons
- Place, resize and rotate tiles freely
- Avoid repetition with object templates

Efficient Tile Layer Editing

- Multi-layer tile editing
- Easy and fast painting of terrain
- Rule-based tile and object placement
- Supports orthogonal, isometric and hexagonal maps

Edit Large Worlds

- Edit infinite maps that grow as needed
- Organize multiple maps in a world

Powerful Workflow

- Quickly switch between projects
- Jump to any file within a project
- Customizable keyboard shortcuts

Extensible with JavaScript

- Add support for custom file formats

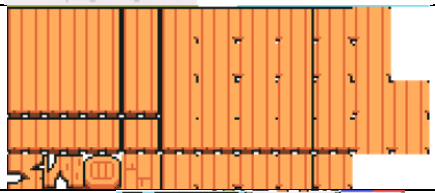
- Write your own actions or tools
- Automate your workflow

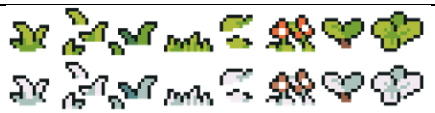
In tiled, I draw maps using the tileset:

You can see there are different blocks on the tileset and every small square stands for a large pixel in the software that can be used to draw the map.

In my plan, there will be three kinds of lands: lawn, desert and snowfield. Sea stands for the block of the map (which means that players can not move over the sea).

Here is a tile table which clearly shows all the tiles needed.

Name of the tile	Function	Picture
Character (unavailable)	To set the original position of the player	
Floor block (unavailable)	To limit the area of players' movement	
Sea besides lawn	To draw the sea besides lawn	
Lawn	To draw the lawn	
Sea besides desert	To draw the sea besides desert	
Desert	To draw the desert	
Snowfield	To draw the snowfield	
Bridge	To draw the bridge	
Temple	To draw the temple, temple is on the snowfield.	
Monsters	To set the position of monsters	

Grass and flower	To draw the grass and flower	
Other objects	To draw other objects	

When adding the tiles, it is noticeable that some tiles need to be set as unavailable, such as the floor blocks, which helps set a barrier for players' movements.

After putting all the tiles at their right position, the map is finished.

```
Procedure CreateMap():
    Define layouts as Dictionary
    Define graphics as Dictionary

    graphics['grass'] = ImportFolder("D:¥CS project¥graphics¥grass")
    graphics['objects'] = ImportFolder("D:¥CS project¥graphics¥objects")

    For Each style, layout in layouts:
        For Each row_index, row in layout:
            For Each col_index, col in row:
                If col is not '-1':
                    Set x to col_index multiplied by TILESIZE
                    Set y to row_index multiplied by TILESIZE

                    If style is 'boundary':
                        Create Invisible Tile at (x, y) and add it to self.obstacle_sprites

                    If style is 'grass':
                        Choose a random_grass_image from graphics['grass']
                        Create a Tile with 'grass' style and random_grass_image at (x, y)
```

```
                Add the Tile to self.visible_sprites and self.obstacle_sprites

            If style is 'object':
                Set surf to graphics['objects'][int(col)]
                Create a Tile with 'object' style and surf at (x, y)
                Add the Tile to self.visible_sprites and self.obstacle_sprites

            If style is 'entities':
                If col is '394':
                    Create a Player at (x, y) with specific parameters
                    Add the Player to self.visible_sprites
                    Add the Player to self.obstacle_sprites

                Else:
                    If col is '390', set monster_name to 'bamboo'
                    Else If col is '391', set monster_name to 'spirit'
                    Else If col is '392', set monster_name to 'raccoon'
                    Else, set monster_name to 'squid'

                    Create an Enemy with monster_name at (x, y)
                    Add the Enemy to self.visible_sprites
                    Add the Enemy to self.obstacle_sprites
    Call CreateMap()
```

Create player:

The player's character may require different kinds of functions and procedures to do various actions, such as walking, attacking and so on. Creating the player has to set their basic variables such as HP and MP, then adding interactions with other objects in the game.

First, creating a specific type of sprites to stand for player with special attributes and methods. The table shows the picture of the player:

State	Picture
Forward	
Backward	
Left	
Right	

```
Class Player Extends Entity:
    Procedure Initialize(pos, groups, obstacle_sprites, create_attack, destroy_attack, create_magic):
        Call Super.Initialize(groups)
        Set image to LoadImage('player.png")
        Set rect to GetRectWithTopLeftPosition(pos) #Adding the player hitbox
        Set hitbox to InflateRect(rect, 0, -26)
```

Set the input keys for the character:

```
Procedure Input():
    Set keys to GetPressedKeys()
```

Moving is the basic part of the game, so the moving key should be easy to use which probably should gather at one location, the four arrow keys are suitable in this case. While the player is moving, the character should not stay at a same state, so when moving it is better using different pictures. After creating the animation, it will show a sense of movement, but now just setting up a different status of the character.

Name	Picture
Move(forward/backward/up/down)	

Moving keys:

```
    If Key UP is Pressed:
        Set self.direction.y to -1
    Else If Key DOWN is Pressed:
        Set self.direction.y to 1
    Else:
        Set self.direction.y to 0


    If Key RIGHT is Pressed:
        Set self.direction.x to 1
    Else If Key LEFT is Pressed:
        Set self.direction.x to -1
    Else:
        Set self.direction.x to 0
```

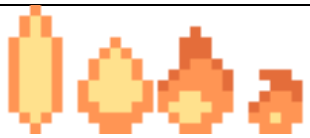
Attack is a method for the player, it allows players to interact with monsters in the game world and the HP of monsters will decrease after successful interaction. The attack picture is just a little bit different from the walking:

Name	Picture
Attack	

Attack:

```
    If Key SPACE is Pressed:
        Set Attacking to True
        Call CreateAttack()
```

Magic should be more amazing and I plan to make two kinds of magic which is healing and fire. The magic will not only interact with player and monsters, it will also effect the objects surrounding.

Name	Function	Picture
Healing	It will increase the HP of the player if HP isn't full.	
Healing interaction	This will produce a kind of animation of healing so the player knows that they are healed.	
Fire	It will make the player shoot a fire ball out and will decrease the HP of enemy.	
Fire interaction	This will produce some fire on the surrounding environment to show that area is burned.	

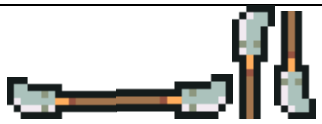
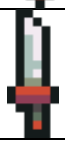
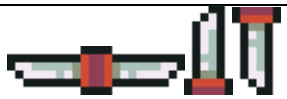
Magic:

```
    If Key U is Pressed:
        Set Attacking to True
        Call CreateMagic(style, strength, cost)
```

Switch Magic:

```
    If Key E is Pressed and CanSwitchMagic:
        Set CanSwitchMagic to False
        Set self.magic_switch_time to GetCurrentTimeInMilliseconds()
```

Weapon is used when player is attacking. There are five types of weapons for the player to use, each weapon will have four kinds of attacking state, standing for forward/backward/left/right attack.

Name	Picture	Attacking state
Axe		
Lance		
Rapier		
Sai		
Sword		

Switch Weapon:

```
    If Key Q is Pressed and CanSwitchWeapon:
        Set CanSwitchWeapon to False
        Set self.weapon_switch_time to GetCurrentTimeInMilliseconds()
```

Get the status of the character (idle, move, attack). The purpose of this step is mainly to decide which kind of animation to show, as the three states have different animation, and interaction with other objects may require the state of the player to work out further reaction:

```
Procedure GetStatus():
  If self.direction.x is 0 and self.direction.y is 0:
    If 'idle' is not in self.status and 'attack' is not in self.status:
      Append '_idle' to self.status

  If Attacking:
    Set self.direction.x to 0
    Set self.direction.y to 0
    If 'attack' is not in self.status:
      If 'idle' is in self.status:
        Replace '_idle' with '_attack' in self.status
      Else:
        Append '_attack' to self.status
  Else:
    If 'attack' is in self.status:
      Replace '_attack' with '' in self.status
```

It requires a data set called status:

```
Procedure ImportPlayerAssets():
  Initialize self.animations as a Dictionary with the following keys:
    'up': List
    'down': List
```

'left': List

'right': List

'right_idle': List

'left_idle': List

'up_idle': List

'down_idle': List

'right_attack': List

'left_attack': List

'up_attack': List

'down_attack': List

Set tiles so that the block of players and monsters would be separated from the normal objects. This is mainly for the HP system, only these two kinds of sprites have HP and have methods of attack, so they need special blocks for further programming on collision between blocks and giving reaction:

```
Class Tile Extends pygame.sprite.Sprite:
  Procedure Initialize(pos, groups, sprite_type, surface = CreateSurface((TILESIZE, TILESIZE))):
    Call Super.Initialize(groups)
    Set self.sprite_type to sprite_type
    Set self.image to surface

    If sprite_type is 'object':
      Set self.rect to GetRectWithTopLeftPosition(pos[0], pos[1] - TILESIZE)
    Else:
      Set self.rect to GetRect(pos)
```

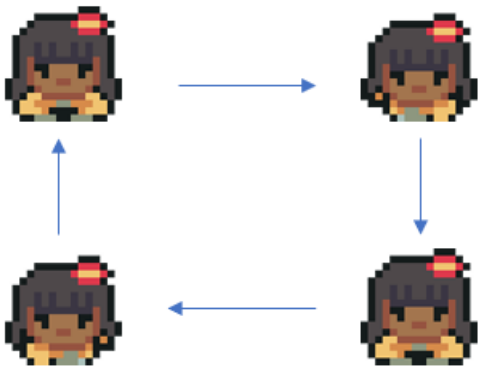
A moving camera should be created as the screen is too small to hold the whole map, the key design is just to make the centre of the screen to move with the character, they will have the same speed, so this speed will be a global variable and the camera will track the player's movement.

```
Class YSortCameraGroup Extends pygame.sprite.Group:
  Procedure Initialize():
    Call Super.Initialize()
    Set self.display_surface to GetDisplaySurface()
    Set self.half_width to GetWidthOfSurface(self.display_surface) // 2
    Set self.half_height to GetHeightOfSurface(self.display_surface) // 2
    Set self.offset to CreateVector2()
    Set self.floor_rect to GetRectWithTopLeftPosition(0, 0)
```

Draw the player on the screen, just display the picture of the player at where we set as point of born:

```
Procedure CustomDraw(player):
    Set self.offset.x to player.rect.centerx - self.half_width
    Set self.offset.y to player.rect.centery - self.half_height

    Set floor_offset_pos to self.floor_rect.topleft - self.offset
    Blit self.floor_surf onto self.display_surface at floor_offset_pos

    For Each sprite in SortedList(self.sprites(), Key=lambda sprite: sprite.rect.centery):
        Set offset_pos to sprite.rect.topleft - self.offset
        Blit sprite.image onto self.display_surface at offset_pos
```

Player moving animation is just a technique of switching pictures very quickly so it looks like moving. Let's use working downward as an example:



The animation is just making these four pictures switching in a very high speed, such as 2 times a second, then the human eye will consider it as a motion.

If making it as a long chain, it should be more clear.

```
For Each animation In KeysOf(self.animations):
    Set full_path to character_path + animation
    Set self.animations[animation] to ImportFolder(full_path)
Procedure Animate():
    Set animation to self.animations[self.status]
    Increment self.frame_index by self.animation_speed
    If self.frame_index is greater than or equal to LengthOfList(animation):
        Set self.frame_index to 0
    Set self.image to animation[IntegerPartOf(self.frame_index)]
    Set self.rect to GetRectWithCenter(self.hitbox.center)
```

‘Entity’ is defined as a class that extends pygame.sprite.Sprite. This means instances of this class can be treated as sprites and managed by Pygame's sprite system.

This procedure initializes a new instance of the Entity class.

It takes a parameter groups which seems to be a list of sprite groups to which this entity will be added.

It first calls the Initialize method of the superclass (Super.Initialize(groups)), which presumably initializes the sprite part of the entity.

It sets some initial attributes of the entity:

self.frame_index is set to 0, likely used for animation purposes.

self.animation_speed is set to 0.15, possibly representing the speed of animation.

self.direction is set to a vector with x and y components initialized to 0.

Procedure Move:

This procedure is responsible for moving the entity.

It takes a parameter speed, which presumably indicates how fast the entity should move.

It first checks if the magnitude (length) of the self.direction vector is not 0. If it's not 0, it normalizes the vector (Normalize(self.direction)), ensuring that the entity moves at a constant speed regardless of its direction.

It updates the x position of the entity's hitbox (self.hitbox.x) by adding the product of self.direction.x and speed. This effectively moves the entity horizontally.

It then calls the Collision procedure with the argument 'horizontal', likely to handle collisions with obstacles in the horizontal direction.

It updates the y position of the entity's hitbox (self.hitbox.y) by adding the product of self.direction.y and speed. This moves the entity vertically.

It calls the Collision procedure again with the argument 'vertical', presumably to handle collisions in the vertical direction.

Finally, it updates the center of the entity's rectangle (self.rect.center) to match the center of its hitbox (self.hitbox.center), which is necessary for rendering the entity correctly on the screen.

Here is the pseudocode:

```
Class Entity Extends pygame.sprite.Sprite:
    Procedure Initialize(groups):
        Call Super.Initialize(groups)
        Set self.frame_index to 0
        Set self.animation_speed to 0.15
        Set self.direction to CreateVector2()

    Procedure Move(speed):
        If MagnitudeOf(self.direction) is not 0:
            Set self.direction to Normalize(self.direction)

        Set self.hitbox.x to self.hitbox.x + self.direction.x * speed
        Call Collision('horizontal')
        Set self.hitbox.y to self.hitbox.y + self.direction.y * speed
        Call Collision('vertical')
        Set self.rect.center to self.hitbox.center
```

The Collision procedure takes a parameter direction, which is expected to be either 'horizontal' or 'vertical'.

If the direction is 'horizontal', it iterates through each sprite in self.obstacle_sprites (assuming self refers to an object or instance of a class).

For each sprite, it checks if its hitbox (a rectangular area representing the sprite's collision area) collides with self.hitbox (presumably representing the collision area of the object calling this procedure).

If a collision is detected and the object's horizontal movement (self.direction.x) is positive (self.direction.x > 0), it adjusts the object's hitbox to the left edge of the colliding sprite's hitbox (Set self.hitbox.right to sprite.hitbox.left). If the movement is negative (self.direction.x < 0), it adjusts the object's hitbox to the right edge of the colliding sprite's hitbox (Set self.hitbox.left to sprite.hitbox.right).

If the direction is 'vertical', it follows a similar process, but this time it checks and adjusts the vertical position of the object based on its self.direction.y and the colliding sprite's hitbox.

Here is the pseudocode:

```
Procedure Collision(direction):
  If direction is 'horizontal':
    For Each sprite in self.obstacle_sprites:
      If sprite.hitbox collides with self.hitbox:
        If self.direction.x > 0:
          Set self.hitbox.right to sprite.hitbox.left
        ElseIf self.direction.x < 0:
          Set self.hitbox.left to sprite.hitbox.right

  If direction is 'vertical':
    For Each sprite in self.obstacle_sprites:
      If sprite.hitbox collides with self.hitbox:
```

```
            If self.direction.y > 0:
                Set self.hitbox.bottom to sprite.hitbox.top
            ElseIf self.direction.y < 0:
                Set self.hitbox.top to sprite.hitbox.bottom
```

Weapon module takes two parameters: player (presumably an instance of the player class) and groups (a list of sprite groups to which this weapon will be added).

It first calls the Initialize method of the superclass (Super.Initialize(groups)), which presumably initializes the sprite part of the weapon.

It extracts the direction of the weapon (direction) from the player's status using the first part of a string split by underscore.

It constructs the full_path of the weapon's image file based on the player's weapon and direction.

It loads the image of the weapon (self.image) from the constructed full_path.

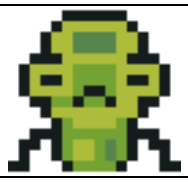
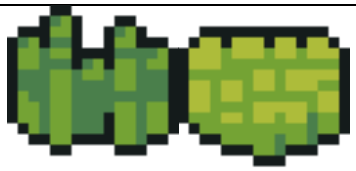
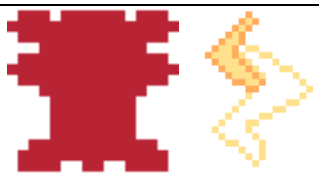
Depending on the direction, it sets the position (self.rect) of the weapon relative to the player's position.

Here is the pseudocode:

```
Class Weapon Extends pygame.sprite.Sprite:
  Procedure Initialize(player, groups):
    Call Super.Initialize(groups)
    Set direction to FirstPartOfSplitString(player.status, '_')

    Set full_path to file(weapon) + player.weapon + direction + ".png"
    Set self.image to LoadImage(full_path).convert_alpha()

    If direction is 'right':
      Set self.rect to GetRectWithMidLeftPosition(player.rect.midright + CreateVector2(0, 16))
    ElseIf direction is 'left':
      Set self.rect to GetRectWithMidRightPosition(player.rect.midleft + CreateVector2(0, 16))
```

```
    ElseIf direction is 'down':
        Set self.rect to GetRectWithMidTopPosition(player.rect.midbottom + CreateVector2(-
10, 0))
    Else:
        Set self.rect to GetRectWithMidBottomPosition(player.rect.midtop + CreateVector2(-
10, 0))
```

Enemy is an important part of the game. I create different kinds of monsters to work as enemies of players.

Name	Picture	Attack
Bamboo		
Raccoon		 
Spirit		
Squid		

The enemy module takes several parameters including monster_name (name of the enemy), pos (position of the enemy), groups (sprite groups), and obstacle_sprites (sprites representing obstacles).

It first calls the Initialize method of the superclass (Super.Initialize(groups)), which presumably initializes the sprite part of the enemy.

It sets sprite_type to 'enemy'.

It imports graphics/animations for the enemy using the ImportGraphics method.

It sets the initial status of the enemy to 'idle' and updates its image based on the current status and frame index.

It sets the position (self.rect) and hitbox (self.hitbox) of the enemy.

Various attributes related to the enemy's stats, such as health, speed, etc., are initialized based on the monster_info fetched from some data source (monster_data).

It initializes attributes related to enemy attacks (can_attack, attack_time, attack_cooldown).

ImportGraphics(name):

This procedure imports graphics/animations for the enemy from specified folders and stores them in self.animations.

GetPlayerDistanceDirection(player):

This procedure calculates the distance and direction vector between the enemy and the player.

GetStatus(player):

This procedure determines the status of the enemy based on the distance from the player.

Actions(player):

This procedure determines the actions of the enemy based on its status.

Animate():

This procedure animates the enemy based on its status and updates its image and rect.

Here is the pseudocode:

```
Class Enemy Extends Entity:
  Procedure Initialize(monster_name, pos, groups, obstacle_sprites):
    Call Super.Initialize(groups)
    Set self.sprite_type to 'enemy'

    Call ImportGraphics(monster_name)
    Set self.status to 'idle'
    Set self.image to self.animations[self.status][IntegerPartOf(self.frame_index)]

    Set self.rect to GetRectWithTopLeftPosition(pos)
    Set self.hitbox to InflateRect(self.rect, 0, -10)
    Set self.obstacle_sprites to obstacle_sprites

    Set self.monster_name to monster_name
    Set monster_info to monster_data[self.monster_name]
    Set self.health to monster_info['health']
    Set self.exp to monster_info['exp']
    Set self.speed to monster_info['speed']
    Set self.attack_damage to monster_info['damage']
    Set self.resistance to monster_info['resistance']
    Set self.attack_radius to monster_info['attack_radius']
    Set self.notice_radius to monster_info['notice_radius']
    Set self.attack_type to monster_info['attack_type']
```

Set self.can_attack to True

Set self.attack_time to None

Set self.attack_cooldown to 400

```
Procedure ImportGraphics(name):
    Initialize self.animations as a Dictionary with keys 'attack', 'idle', and 'move'
    Set main_path to file(monster)+ name + "/"
    For Each animation in KeysOf(self.animations):
        Set self.animations[animation] to ImportFolder(main_path + animation)

Procedure GetPlayerDistanceDirection(player):
    Set enemy_vec to CreateVector2(self.rect.center)
    Set player_vec to CreateVector2(player.rect.center)
    Set distance to MagnitudeOf(player_vec - enemy_vec)

    If distance is greater than 0:
        Set direction to Normalize(player_vec - enemy_vec)
    Else:
        Set direction to CreateVector2()

    Return (distance, direction)

Procedure GetStatus(player):
    Set distance to GetPlayerDistanceDirection(player)[0]

    If distance is less than or equal to self.attack_radius and self.can_attack:
        If self.status is not 'attack':
            Set self.frame_index to 0
```

```
        Set self.status to 'attack'
    ElseIf distance is less than or equal to self.notice_radius:
        Set self.status to 'move'
    Else:
        Set self.status to 'idle'
```

```
Procedure Actions(player):
    If self.status is 'attack':
        Set self.attack_time to GetCurrentTimeInMilliseconds()
        Print('attack')
    ElseIf self.status is 'move':
        Set self.direction to GetPlayerDistanceDirection(player)[1]
    Else:
        Set self.direction to CreateVector2()

Procedure Animate():
    Set animation to self.animations[self.status]

    Increment self.frame_index by self.animation_speed
    If self.frame_index is greater than or equal to LengthOfList(animation):
        If self.status is 'attack':
            Set self.can_attack to False
        Set self.frame_index to 0

    Set self.image to animation[IntegerPartOf(self.frame_index)]
    Set self.rect to GetRectWithCenter(self.hitbox.center)
```

Player cool down:

Set current_time to GetCurrentTimeInMilliseconds(): This retrieves the current time in milliseconds.

If Attacking: Checks if the player is currently in an attacking state.

If current_time - self.attack_time >= self.attack_cooldown: Checks if enough time has passed since the last attack based on the attack cooldown.

Set Attacking to False: Resets the attacking state to false.

Call DestroyAttack(): Performs some action to destroy or reset the attack.

If Not CanSwitchWeapon: Checks if the player is currently unable to switch weapons.

If current_time - self.weapon_switch_time >= self.switch_duration_cooldown: Checks if enough time has passed since the last weapon switch based on the switch duration cooldown.

Set CanSwitchWeapon to True: Allows the player to switch weapons.

If Not CanSwitchMagic: Checks if the player is currently unable to switch magic.

If current_time - self.magic_switch_time >= self.switch_duration_cooldown: Checks if enough time has passed since the last magic switch based on the switch duration cooldown.

Set CanSwitchMagic to True: Allows the player to switch magic.

Here is the pseudocode:

```
Procedure Cooldowns():
  Set current_time to GetCurrentTimeInMilliseconds()

  If Attacking:
    If current_time - self.attack_time >= self.attack_cooldown:
      Set Attacking to False
      Call DestroyAttack()
```

```
If Not CanSwitchWeapon:
    If current_time - self.weapon_switch_time >= self.switch_duration_cooldown:
        Set CanSwitchWeapon to True

If Not CanSwitchMagic:
    If current_time - self.magic_switch_time >= self.switch_duration_cooldown:
        Set CanSwitchMagic to True
```

Enemy cool down:

If Not CanAttack: Checks if the enemy is currently unable to attack.

Set current_time to GetCurrentTimeInMilliseconds(): Retrieves the current time in milliseconds.

If current_time - self.attack_time >= self.attack_cooldown: Checks if enough time has passed since the last attack based on the attack cooldown.

Set CanAttack to True: Allows the enemy to attack again.

Here is the pseudocode:

```
Procedure Cooldown():
  If Not CanAttack:
    Set current_time to GetCurrentTimeInMilliseconds()
    If current_time - self.attack_time >= self.attack_cooldown:
      Set CanAttack to True
```

For the game to run successfully and make it playable, there are four ranks of requirements: Must, Should, Could an Would.

Must:

1. The game must have a basic interface for the player to play, which means that at least a screen must be displayed and it shows some contents.
2. The game must have basic input keys for the user to use. For example, in my game the player must be able to use the direction keys on the keyboard to control the character to move.
3. The game must have attack keys as the player must have certain interactions with the game world, such as the monsters and the land.
4. The game must have sprites other than the player, so that it fills up the game world and makes the game playable.
5. The game must be able to close. Otherwise it will never end and keep running, which may crash the device.
6. The game must have weapons as it is one of the most important parts of the game.
7. The game must have a complete map which provides a place for the player to hung around and enjoy the game.
8. The game must have monsters, as it is the central part of the game 'Monster Hunter'.
9. The game must have colliding system so that the interactions with monsters and the map can be achieved.
10. The game must have health system.

Should:

1. The game should have fluent animation because it could provide a virtual experience for the player.
2. The game should have sound effect so that it is more playable and the player could enjoy the game in both virtual and vocal ways.
3. The game should have different kinds of weapons so the player can have more choices, not always playing one of them, which might be boring.
4. The game should have a moving camera which always keep the player at the centre, so that the map will not be limited by the size of the game window.
5. The game should have both physical attack and magic so that it is not boring for the player.
6. The game should show the attack process, such as the animation of weapons or the movement of magic bonds.
7. The game should have a basic UI to show the health and the magic points. Therefore the player knows the conditions of the character.
8. The game should have different kinds of monsters with various appearance and attacking methods.
9. The monsters should be able to move randomly on the map so that it increases the possibility of interacting.
10. The monsters should be able to rebirth so that the player can always find monsters to fight.
11. The monsters should also have inner health point to count damage and kill.
12. Most of the other sprites on the map should not be breakable or movable, such as rocks and walls, they should be fixed at their positions.
13. There should be an invisible wall at the edge of the map, in my game I will display it as the sea and the player can not work through. This could prevent unlimited increase of the game file.

Could:

1. The game could make different kinds of particles as special effects, such as the attacking particles or flames of magic. This will increase the playability of the attack system.
2. The game could have special death animation for monsters, which it will provide a better respond to the player when they kill a monster.
3. The game could have good attracting system, which means that the monsters will automatically follow the player's working path until they can attack.
4. The attack of monsters and the player could all have inner cooldown, which will prevent some special crashes such as too many inputs and responses.
5. The weapons and magic could make some kinds of effect on the environment, such as destroying grasses.

Would:

1. The game would be able to control the volume of health and magic point by a central setting menu.
2. The game would have an experience system which then the player can update inside games and get higher attack damage, higher health and magic points.
3. The game would have a user interface to control the sound and brightness.
4. The game would have both success and failure for the player.

Index	Name of test	Purpose	Result
1	Initializing	Check if the game initializes without any errors.	
2	Pygame check	Ensure that pygame initializes successfully.	
3	Game window	Verify that the game window is created with the correct dimensions.	
4	Title	Confirm that the game caption is set to ‘Monster Hunter: Cartoon’	
5	Clock	Test if the clock object is created.	
6	Sound	Check if the main sound loads and plays correctly.	
7	Initial volume of sound	Test if the volume of the main sound is set to 0.5	
8	Sound loop	Verify that the main sound loops continuously	
9	QUIT	Simulate a QUIT event and ensure that the game quits without errors.	
10	Keydown	Simulate a KEYDOWN event with the 'm' key and verify if the toggle_menu method of the Level object is called.	
11	Game loop	Ensure that the game loop runs continuously.	
12	Screen filled	Check if the screen is filled with the correct color defined in ‘WATER_COLOR’	
13	Run method	Verify that the run method of the Level object is called within the game loop.	
14	Display	Test if the display updates correctly.	
15	FPS check	Check if the frame rate is limited to the specified value (‘FPS’)	

16	Functionality	Test the overall functionality by running the game and checking for any unexpected behavior.	
17	User	Interact with the game manually to ensure that it responds correctly to user input and events.	
18	Error Handling	Test how the game handles unexpected inputs or events, such as invalid key presses or window resizing.	
19	Source	Verify that all required resources (such as audio files) are loaded correctly.	
20	Missing source	Test what happens if a resource is missing or cannot be loaded	
21	Overall performance	Evaluate the performance of the game, including frame rate and resource usage, to ensure smooth gameplay.	
22	Variable check	Check if the WIDTH and HEIGHT constants have the correct values. Ensure that TILESIZE, BAR_HEIGHT, HEALTH_BAR_WIDTH, ENERGY_BAR_WIDTH, and ITEM_BOX_SIZE are set to appropriate values.	
24	Colour	Verify that all color constants are defined correctly and in the correct format (e.g., hexadecimal or named colors).	
25	Colour suited	Check if colors are visually distinguishable and suitable for their respective purposes (e.g., UI	

		colors, health and energy bar colors)	
26	UI font	Check if the path to the UI font file (UI_FONT) is correct.	
27	UI elements	Ensure that the font size (UI_FONT_SIZE) is appropriate for the UI elements.	
28	Data	Ensure that the weapon_data, magic_data, and monster_data dictionaries contain the expected keys and values.	
29	Data entries	Test if weapon, magic, and monster data entries have all required attributes (e.g., cooldown, damage, graphic path).	
30	File path	Verify that all file paths for graphics, audio, and other resources are correct and accessible.	
31	Missing path	Check if missing or incorrect file paths are handled gracefully in the code.	
32	Hitbox	Check if HITBOX_OFFSET values are appropriate for their corresponding entities	
32	Cool down logic	Test the logic related to weapon cooldowns, damage calculations, magic strength, enemy health, etc., using sample data from the dictionaries.	
33	Calculation	Verify that all calculations and logic related to the game entities are functioning as expected.	
34	Level initialization	Verify that the Level object initializes without any errors.	

35	groups	Ensure that all necessary attributes and sprite groups are initialized correctly.	
36	Map creation	Test if the map creation logic (create_map method) generates the map with the correct tiles and entities.	
37	Objects on map	Check if different types of tiles (grass, objects) are created correctly with their respective graphics.	
38	Create attack	Test if the create_attack method creates a new weapon sprite correctly.	
39	Remove weapon	Verify that the destroy_attack method removes the current weapon sprite without errors.	
40	Interaction(attack)	Test the player attack logic (player_attack_logic method) to ensure that attack sprites interact correctly with attackable sprites.	
41	Damage	Verify that attack sprites cause damage to attackable sprites and trigger appropriate effects.	
42	Player being damaged	Test the damage_player method to ensure that player health decreases correctly when taking damage from enemies.	
43	Invulnerable after damage	Check if the player becomes invulnerable for a certain duration after taking damage.	
44	Death particles	Verify that the trigger_death_particles method creates death particles at the correct position and with the correct type.	

45	Create particles	Test if the create_particles method creates particles with the correct type and at the correct position.	
46	Experience point	Test the add_exp method to ensure that player experience points increase correctly when enemies are defeated.	
47	Menu	Test the toggle_menu method to ensure that the game pause state toggles correctly when the menu is opened or closed.	
48	Camera group	Test if the YSortCameraGroup class functions correctly in rendering sprites with correct depth sorting.	
49	Following camera	Verify that sprites are drawn relative to the player's position and the camera offset.	
50	Update on enemy	Test if the enemy_update method updates enemy sprites correctly, considering the player's position and behavior.	
51	Player initialization	Check if the player object initializes without any errors	
52	Player attributes	Verify that all necessary attributes are initialized correctly.	
53	Player image	Test if the player image is loaded correctly.	
54	Transparency	Ensure that the image has the correct transparency settings (convert_alpha).	
55	Input keys	Test various key inputs (movement, attack, switch weapon/magic) to ensure they are correctly detected.	

56	Direction and status	Check if the player's direction and status are updated based on the input.	
57	Attack of player	Test if the create_attack method is called when the attack input key is pressed.	
58	Magic of player	Test if the create_magic method is called when the magic input key is pressed.	
59	Switch weapon/magic	Test the switching of weapons and magic to ensure it cycles through the available options correctly.	
60	Switch cooldown	Verify that the cooldown between switching is enforced correctly.	
61	Damage calculation	Test the get_full_weapon_damage and get_full_magic_damage methods to ensure they calculate damage correctly.	
62	Calculation base	Verify that the calculated damage includes both base and weapon/magic damage.	
63	Status update	Test if the player's status (idle, attack) updates correctly based on movement and attacking states.	
64	Animation	Test if player animations are displayed correctly based on the player's status and direction.	
65	Animation loop	Ensure that animations loop correctly and smoothly.	
66	Player stats	Test if player stats (health, energy) are updated correctly over time (e.g., energy recovery).	

67	Stats limit	Verify that stats are capped at their maximum values.	
68	Upgrade cost	Test if the upgrade costs for player stats are retrieved correctly from the upgrade_cost dictionary.	
69	Enemy initialization	Verify that the enemy object initializes without any errors.	
70	Enemy attributes	Ensure that all necessary attributes are initialized correctly, including graphics, stats, sounds, and interaction parameters.	
71	Image	Test if the enemy graphics are imported correctly from the specified folder based on the monster name.	
72	Animation dictionary	Ensure that the animations dictionary contains the correct paths to the animation frames.	
73	Distance and direction	Test the get_player_distance_direction method to ensure it calculates the distance and direction between the enemy and the player correctly.	
74	Status	Test the get_status method to ensure it correctly determines the status (idle, move, attack) of the enemy based on the player's distance.	
75	Action	Verify that the actions method executes the correct action (attack or move) based on the enemy's status.	
76	Correct action based on range	Test if the enemy attacks the player when in attack range and	

		moves towards the player when in notice range.	
77	Animation changes	Test if the enemy animation changes correctly based on its status (idle, move, attack).	
78	Animation loop	Ensure that animation loops correctly and smoothly.	
79	Monster cooldown	Test if the attack cooldown and invincibility duration are managed correctly.	
80	Cooldown error	Verify that the enemy cannot attack or take damage during the cooldown periods.	
81	Damage	Test the get_damage method to ensure it calculates the correct damage to the enemy based on the player's attack type.	
82	Been damaged	Ensure that the enemy's health decreases correctly when taking damage.	
83	Death	Verify that the enemy is killed when its health reaches zero.	
84	Death display	Test if death triggers the correct particle effects, experience points addition, and plays the death sound.	
85	Hit reaction	Test if the enemy's hit reaction (pushback) is applied correctly when taking damage.	
86	Entity initialization	Verify that the Entity object initializes without any errors.	
87	Values	Ensure that the initial values of frame_index, animation_speed, and direction are set correctly.	
88	Move update	Test the move method to ensure it correctly updates the position	

		of the entity based on its direction and speed.	
89	Obstacles	Verify that the entity moves smoothly in both horizontal and vertical directions without passing through obstacles.	
90	Collision	Test the collision method to ensure it detects collisions with obstacle sprites correctly in both horizontal and vertical directions.	
91	Reaction of collision	Verify that the entity's movement is stopped or adjusted appropriately when colliding with obstacles.	
92	Wave value	Test the wave_value method to ensure it returns values between 0 and 255 based on the sine function of the current time.	
93	Oscillation	Verify that the returned value oscillates smoothly between 0 and 255 as time progresses.	
94	Magic initialization	Verify that the MagicPlayer object initializes without any errors.	
95	Sound of magic	Ensure that the sounds dictionary contains the correct sound objects for 'heal' and 'flame' spells.	
96	Heal	Test the heal method to ensure it heals the player correctly when the player has enough energy.	
97	HP increase	Verify that the player's health increases by the specified strength and energy decreases by the cost of the spell.	

98	Sound of healing	Check if the 'heal' sound is played and if particles are created at the player's position.	
99	Not enough energy for healing	Test the heal method when the player's energy is insufficient to cast the spell.	
100	Flame	Test the flame method to ensure it creates flame particles in the correct direction and positions.	
101	Cost of flame	Verify that the player's energy decreases by the cost of the spell.	
102	Sound of flame	Check if the 'flame' sound is played and if flame particles are created in the correct positions.	
103	Not enough cost for flame	Test the flame method when the player's energy is insufficient to cast the spell.	
104	Flame failed to play	Verify that the player's energy remains unchanged, and no flame particles are created.	
105	Weapon initialization	Verify that the Weapon object initializes without any errors.	
106	Sprite type of weapon	Ensure that the sprite_type attribute is set correctly to 'weapon'.	
107	Image of weapon	Check that the image attribute is loaded with the correct weapon graphic based on the player's direction.	
108	Placement based on direction	Test the placement of the weapon sprite based on different player directions.	
109	Position	For each direction ('right', 'left', 'down', 'up'), verify that the weapon sprite is positioned	

		correctly relative to the player's rect.	
110	Position relative to player	Ensure that the weapon sprite is placed at the appropriate position relative to the player's center point.	
111	Edge cases	Test edge cases such as when the player's direction is not one of the expected values ('right', 'left', 'down', 'up').	
112	Reaction	Verify that the weapon sprite placement handles unexpected player directions gracefully without causing errors or unexpected behavior.	
113	Loading graphics	Check that the loaded weapon graphic is displayed correctly on the screen when blitted.	
114	Upgrade initialization	Verify that the Upgrade object initializes without any errors.	
115	Attributes for upgrade	Check that the required attributes and parameters are correctly set during initialization.	
116	Item	Ensure that the Item objects are created and stored correctly within the Upgrade object.	
117	Input key	Test keyboard input handling in the input method of the Upgrade class.	
118	Selection index	Verify that the selection index changes correctly when pressing the left and right arrow keys.	
119	Bounds of index	Check that the selection index does not go out of bounds.	

120	Selection cooldown	Test the cooldown functionality in the selection_cooldown method of the Upgrade class.	
121	Trigger	Test the trigger functionality in the trigger method of the Item class.	
122	Selected attributes	Verify that the selected attribute of the player is correctly upgraded when triggering an item.	
123	Experience reaction	Check that the player's experience and attribute values are updated correctly after triggering an item.	
124	Display of Item	Test the rendering of items on the screen in the display method of the Item class.	
125	Name, cost and attribute	Verify that the item names, costs, and attribute bars are displayed correctly.	
126	Highlight	Check that the selected item is highlighted differently from others	
127	UI initialization	Verify that the UI object initializes without any errors.	
128	Health bar	Test the show_bar method to ensure that it correctly displays the player's health bar.	
129	Health width	Verify that the health bar width is adjusted based on the current health and maximum health values.	
130	Colour	Check that the correct colors are used for the health bar and its border.	

131	Energy bar	Similar to the health bar test, verify that the show_bar method correctly displays the player's energy bar.	
132	Energy width	Ensure that the energy bar width is adjusted based on the current energy and maximum energy values.	
133	Colour	Check that the correct colors are used for the energy bar and its border.	
134	Experience	Test the show_exp method to ensure that it correctly displays the player's experience points.	
135	Position of experience	Verify that the experience points are rendered at the correct position on the screen.	
136	Weapon overlay display	Test the weapon_overlay method to ensure that it correctly displays the player's selected weapon.	
137	Magic overlay display	Similar to the weapon overlay test, verify that the magic_overlay method correctly displays the player's selected magic.	

Implementation

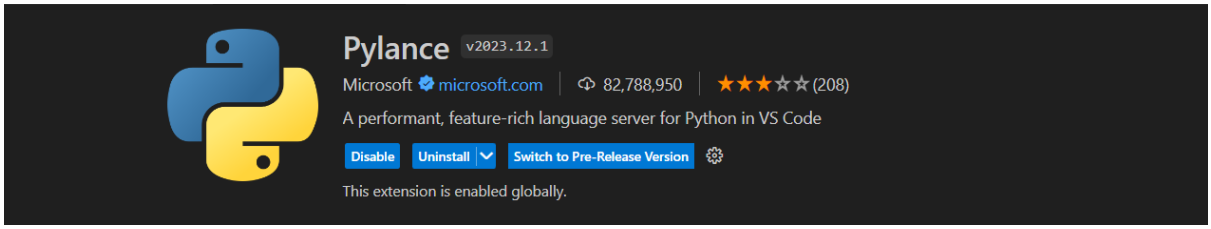
I have divided the implementation of my game into 11 stages, they are:

1. Project setup
2. Level setup
3. Creating the player
4. Creating the camera
5. Graphics
6. Player animations
7. Weapons
8. UI
9. Magic
10. Enemy creation
11. Player-enemy interaction

Stage 1: Project Setup:

Step 1 : Set up a pygame environment

Based on the pseudocode given, I will create this project in VScode using python and pygame language. The first step of this will be creating python environment.



Now the environment has been installed so that the computer can understand what I'm writing.

After that, I just try to import the pygame, while it does not work. The message looks like:

```
Traceback (most recent call last):
  File "c:\CS\import pygame.py", line 1, in <module>
```

I realize that I haven't installed the pygame so the computer consider the "pygame" as a module in C disk. It is not hard to fix this error:

```
PS C:\Users\Rshen5878> pip install pygame
Collecting pygame
  Obtaining dependency information for pygame from https://files.pythonhosted.org/packages/82/61/93ae7afbd931a70510cfdf0a7bb0007540020b8d80bc1d8762ebdc46479b/pygame-2.5.2-cp311-cp311-win_amd64.whl.metadata
  Downloading pygame-2.5.2-cp311-cp311-win_amd64.whl.metadata (13 kB)
Downloading pygame-2.5.2-cp311-cp311-win_amd64.whl (10.8 MB)
     ---------------------------------------- 10.8/10.8 MB 14.9 MB/s eta 0:00:00
Installing collected packages: pygame
Successfully installed pygame-2.5.2
```

Now, the pygame works, it shows that our basic game environment has been set up.

Step 2 : Create a window for the game

The next step is to create a window for the game. The window should have a width, a height and a background colour.

```
WIDTH = 1200
HEIGHT = 900
class Game:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode(WIDTH, HEIGHT)


    def run(self):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

            self.screen.fill('black')
            pygame.display.update()

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

However, there's an error found:

```
TypeError: size must be two numbers
```

The width and height are both numbers, so the error is not just about numbers.

After adding a second bracket, the bug is solved:

```
pygame.display.set_mode((WIDTH,HEIGHT))
```

Now the window has been set up, but a new problem comes:

If in a normal case, the window should have a title. However, now it only has a black background.

After debugging, I just find out it is a very small problem, which is that the height of the screen is too large so the title is hided. Now, I change the height into 720 and then the title is displayed.

```
HEIGHT = 720
```

The game requires an inner clock working as FPS so the cool down and animation can be implemented, so I create it.

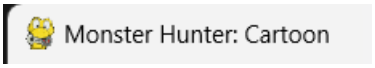
```
self.clock = pygame.time.Clock()
```

```
self.clock.tick(FPS)
```

Step 3: Change the title

Here comes the final step of this stage. As I do not want the title of the game to be “pygame window”, it should have another name. The only thing I need is “set_caption”.

```
pygame.display.set_caption('Monster Hunter: Cartoon')
```



I want to call it “Monster Hunter: Cartoon”, which after testing the title is changed correctly.

Now Stage 1 has been finished.

Stage 2: Level Setup

Before I start creating my levels, I just find out that my codes should be separated into different sections, which makes it much clearer and it will be easier for me to modify or test. As a result, I will split my project into 13 sections:

Main

Settings

Level

Player

Particles

Tile

Enemy

Weapon

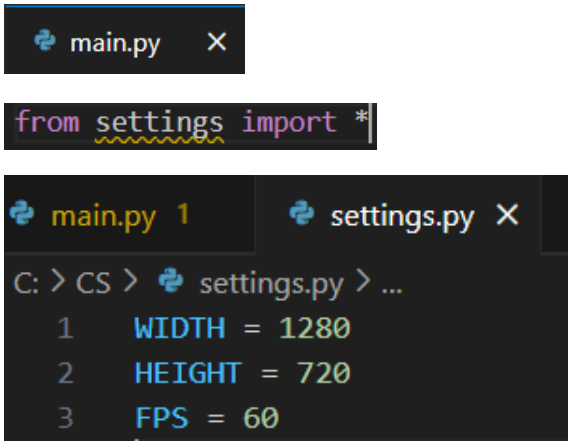
Magic

Entity

Upgrade

Ui

Support



Now I have renamed the basic setup as “main”, and the variables that I set, such as ‘WIDTH’, ‘HEIGHT’ and ‘FPS’ should all be moved into the “settings”. Then just import them from the settings.

Although it shows erroring, after testing it works.

Step 1: It is the start of level creation. Firstly, as my design mentioned, I should create different classes for different sprites (player, enemies, map …) and also deal with their interactions.

To manage them in a more efficient way, I will separate different groups. The largest two groups are 'visible_sprites' and 'obstacles_sprites'.

All the sprites that are visible belongs to the 'visible_sprites' class and sprites that the player can collide are in 'obstacles_sprites' class.

Then, it's time for coding.

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

        self.display_surface = pygame.display.get_surface()
        self.visible_sprites = pygame.sprite.Group()
        self.obstacles_sprites = pygame.sprite.Group()
```

In the new file called "level", pygame is imported and the two groups of class are created.

However, it is not enough just to create those classes, I need to define a function which could update and draw the game.

```
    def run(self):
        pass
```

At this moment there's no need to put anything inside, so just put pass which is more convenient for testing.

Step 2: Draw the detailed map.

As it is now still testing stage, I will not directly bring my final map. Instead, I will draw a pretty easy one for testing and debugging.

```
WORLD_MAP = [
['x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x'],
['x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ','p',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ','x','x','x','x','x',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ','x'],
['x',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x','x','x',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ','x',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ','x','x','x','x','x',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ','x','x','x',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ',' ','x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','x'],
['x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x'],
]
```

Above is the map designed for testing purpose. In the map, ‘x’ stands for stones, ‘ ’ stands for free space and ‘p’ stands for the player.

To create the basic map, I actually need to create those sprites that may appear in the map. As a result, the new section “tile” is used.

```
import pygame

class Tile(pygame.sprite.Sprite):
    def __init__(self,pos,groups):
        super().__init__(groups)
        self.image = pygame.image.load('"C:\CS\graphics\test\rock.png"').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)
```

Here comes the tiles. I create the sprite for the rocks which import the image from my C disc called “rock.png”. Then it will print the image at the position I will enter. Then, I just need to first link this file with the “level” and then create a new function which will print the image at the position given ‘x’ on the world map. After those it should works.

```
import pygame

class Player(pygame.sprite.Sprite):
    def __init__(self,pos,groups):
        super().__init__(groups)
        self.image = pygame.image.load('"C:\final game\graphics\test\player.png"').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)
```

The codes to create the player is very similar at this stage as I just need the image to be printed here. The animation and motion are all in the behind stages.

```
        self.create_map()

    def create_map(self):
        for row in WORLD_MAP:
            print(row)
```

After creating the tiles, I could then write the function of creating the map at the "level" section.

```
def __init__(self):
    pygame.init()
    self.screen = pygame.display.set_mode((WIDTH,HEIGHT))
    self.clock = pygame.time.Clock()
    pygame.display.set_caption('Monster Hunter: Cartoon')

    self.level = Level()
```

```
def run(self):
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        self.screen.fill('black')

        self.level.run()

        pygame.display.update()
        self.clock.tick(FPS)
```

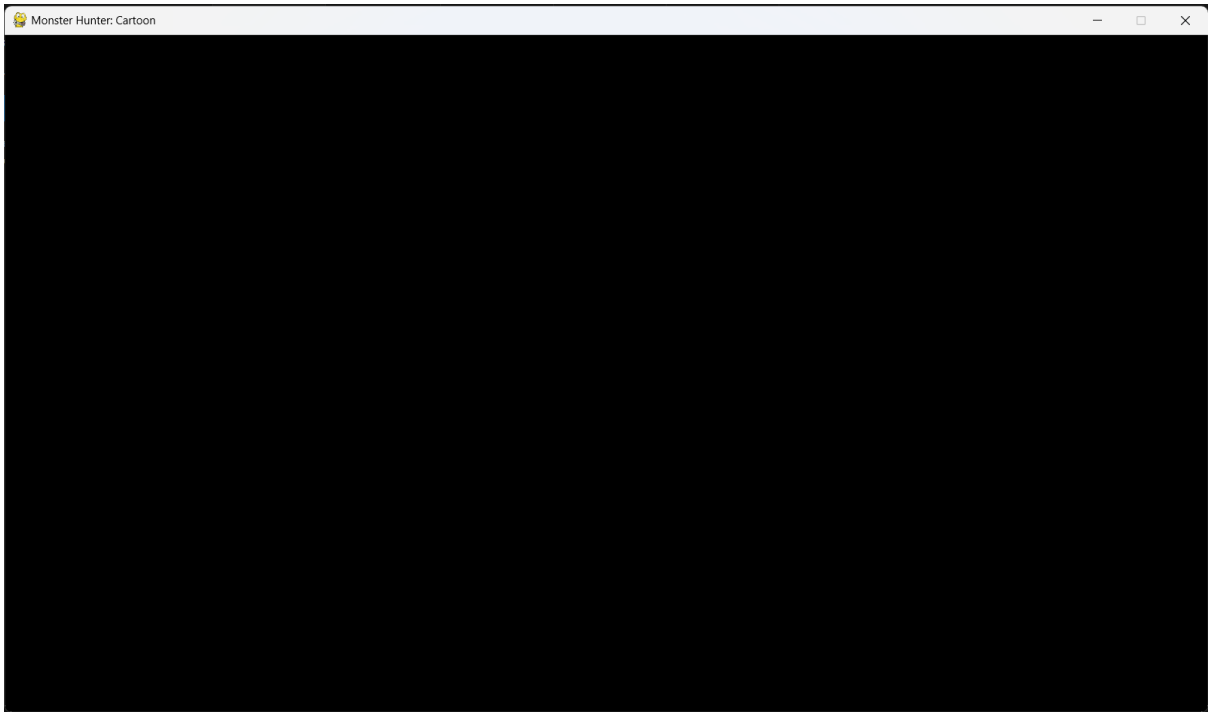
Then import the class Level from the "level" to the "main" to run it. The next step is to test it.

```
  File "c:\CS\level.py", line 10, in __init__
    self.create_map()
  File "c:\CS\level.py", line 13, in create_map
    for row in WORLD_MAP:
               ^^^^^^^^^
NameError: name 'WORLD_MAP' is not defined
```

Here comes the error, 'WORLD_MAP' is not defined. I'm sure that the WORLD_MAP is drawn inside the "settings" section, so there might be problems in the "level" section.

```
import pygame
```

The error is found. I do not import anything from the settings so the “level” section does not know anything about WORLD_MAP, just adding: from settings import * will fix it.



```
  File "c:\CS\level.py", line 10, in __init__
    self.create_map()
  File "c:\CS\level.py", line 13, in create_map
    for row in WORLD_MAP:
               ^^^^^^^^^
NameError: name 'WORLD_MAP' is not defined
```

```
pygame 2.5.2 (SDL 2.28.3, Python 3.11.8)
Hello from the pygame community. https://www.pygame.org/contribute.html
['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', 'p', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x', 'x', 'x', 'x', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', 'x', 'x', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', 'x', 'x', 'x', 'x', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', 'x', 'x', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'x']
['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']
```

Now it is perfect for this section as the game window runs and the map is printed in the print sheet.

The next step will be putting the right images on the right position, which I need to give a definition to every box of the map using row and column numbers. When meeting an 'x' it will print the image of stone; when meeting a 'p' it will print the image of the player; when meeting ' ', just leaves it blank.

```
def create_map(self):
    for row_index, row in enumerate(WORLD_MAP):
        for col_index, col in enumerate(row):
            x = col_index * 64
            y = row_index * 64
            if col == 'x':
                Tile((x,y),[self.visible_sprites,self.obstacle_sprites])
            if col == 'p':
                Player((x,y),[self.visible_sprites])
```

```
  File "c:\CS\level.py", line 19, in create_map
    Tile((x,y),[self.visible_sprites,self.obstacle_sprites])
    ^^^^
NameError: name 'Tile' is not defined
```

Here is the code to draw the map. After testing, an error is found which is just because I forget to import Tile class from tile to the level. As a result, just import it to fix this error. So does the Player class.

```
  File "c:\CS\level.py", line 21, in create_map
    Tile((x,y),[self.visible_sprites,self.obstacle_sprites])
                                     ^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Level' object has no attribute 'obstacle_sprites'. Did you mean: 'obstacles_sprites'?
```

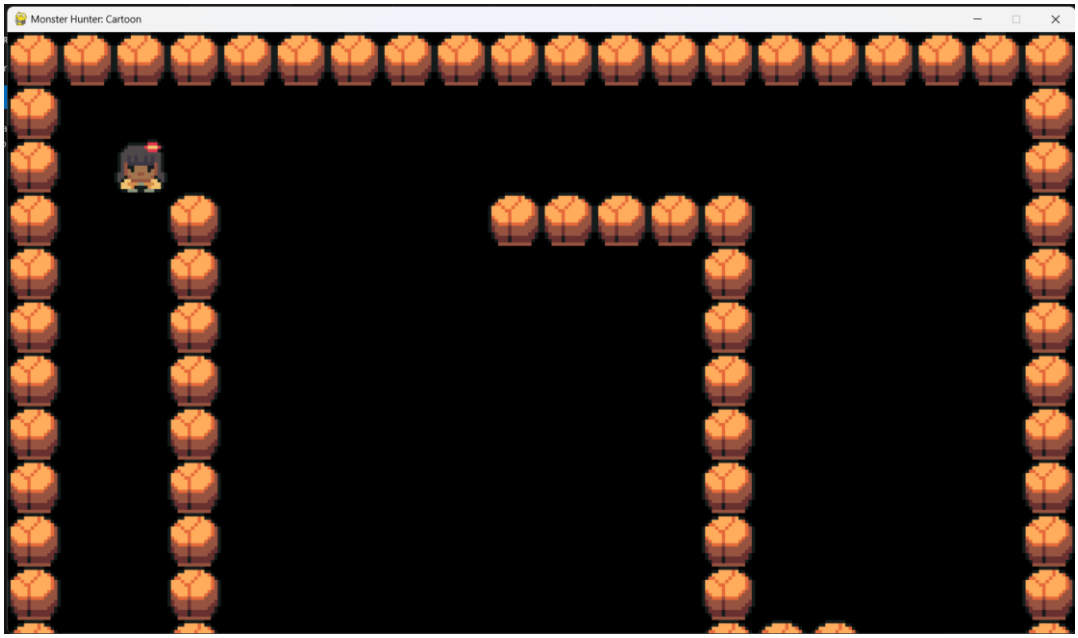
After another test, a new spelling mistake is found which I forget an ‘s’ at the end of ‘obstacle’. After adding it:

Nothing appears, there should be something wrong which is not normal bug, it might be a logic error.

```
def run(self):
    pass
```

The issue is found. I just create the map, but I do not run any function so it is just in the list, not printing out the images. As a result, I need to draw the images out following the map created in the create_map function.

```
def run(self):
    self.visible_sprites.draw(self.display_surface)
```



Now it looks much better than before and you can see all the images are placed in the right position.

Then, the whole level setup has been finished. If I want a more complex map I just need to import the map from the software ‘Tiled’ which I will mention afterwards. The basic logic and functions have been finished setting up.

Stage 3: Creating the player

Step 1: The player needs to move around the map by using their keyboards for inputs. As a result, I need to first create the keyboards input of up, down, left and right.

```
def input(self):
    keys = pygame.key.get_pressed()

    if keys[pygame.K_UP]:
        self.direction.y = -1
    elif keys[pygame.K_DOWN]:
        self.direction.y = 1
    else:
        self.direction.y = 0

    if keys[pygame.K_RIGHT]:
        self.direction.x = 1
    elif keys[pygame.K_LEFT]:
        self.direction.x = -1
    else:
        self.direction.x = 0
```

The keys are pretty easy to set up as just define the direction of the player and then input different x or y for different directions. In this case, using vectors will be even more convenient to define the direction.

```
def __init__(self,pos,groups):
    super().__init__(groups)
    self.image = pygame.image.load('C:\CS\graphics\Test\Player.png').convert_alpha()
    self.rect = self.image.get_rect(topleft = pos)
    self.direction = pygame.math.Vector2()
```

The direction is then defined as a vector which is pretty easy for mathematical calculations. The difficulty that I find is that it is pretty hard to test if this does work or not, so I decide to directly go on until there is certainly motion happening.

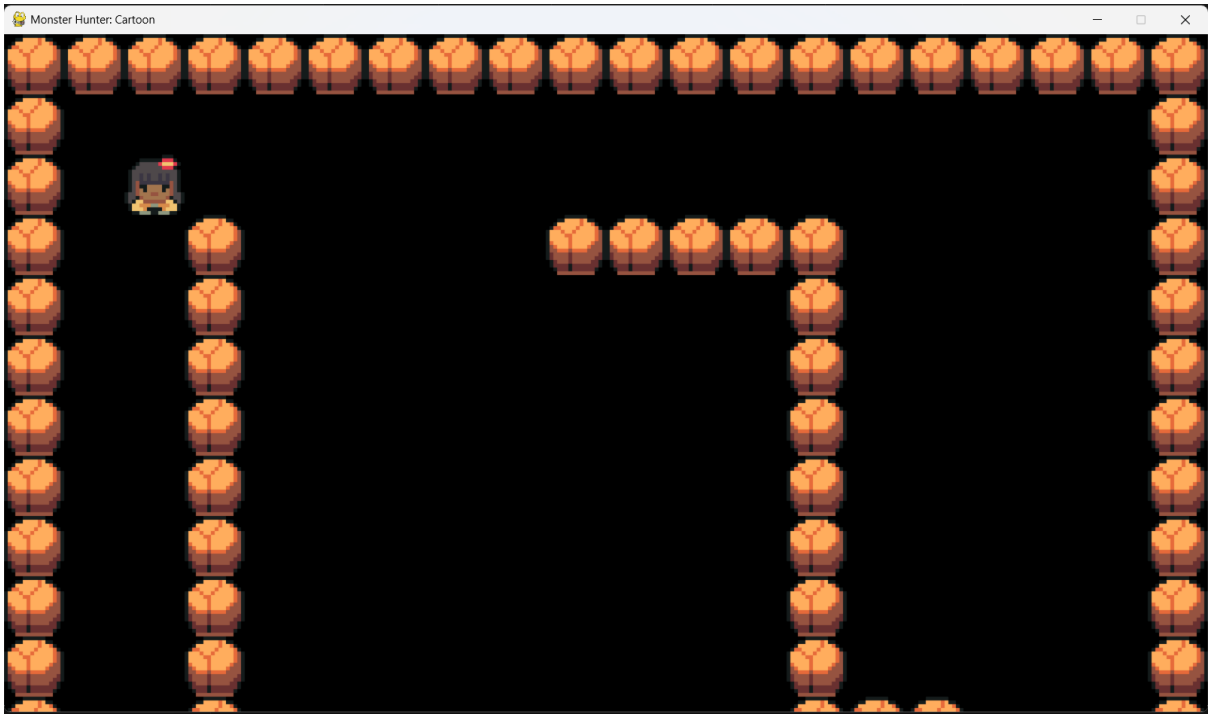
Step 2: After entering the keys and define the directions, the player just requires a speed to really move.

```
    self.speed = 5
```

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

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

Therefore, I just give a speed of 5 and then define a function called 'move' which could change the x and y coordinates of the player to show a motion. Here I create a new function called 'collision'. The purpose of this function is mainly to create a real experience for the player of colliding with other objects. The key inside the 'collision' is the obstacle boxes I give to everything I create before. The player and stones will have collision box to produce the effect of collision, while something like grass or flowers will not have boxes to collide. I just set when their sides of boxes touch each other, the touched edge of rectangle of the player will always equal to that of the stones or something else, which create an effect of stopping.

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

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



I still haven't written the move function to the level side, but it is necessary to test if there is errors. After testing, it seems that tail now it still works.

The final step of moving is to update all the functions to the level side, I just create a function called update which include all the functions required for motion, and then add to the level side to see if it works.

```
def run(self):
    self.visible_sprites.draw(self.display_surface)
    self.visible_sprites.update()
```

```
def run(self):
    self.visible_sprites.draw(self.display_surface)
    self.visible_sprites.update()
```

```
    for sprite in self.obstacles_sprites:
                  ^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Player' object has no attribute 'obstacles_sprites'
```

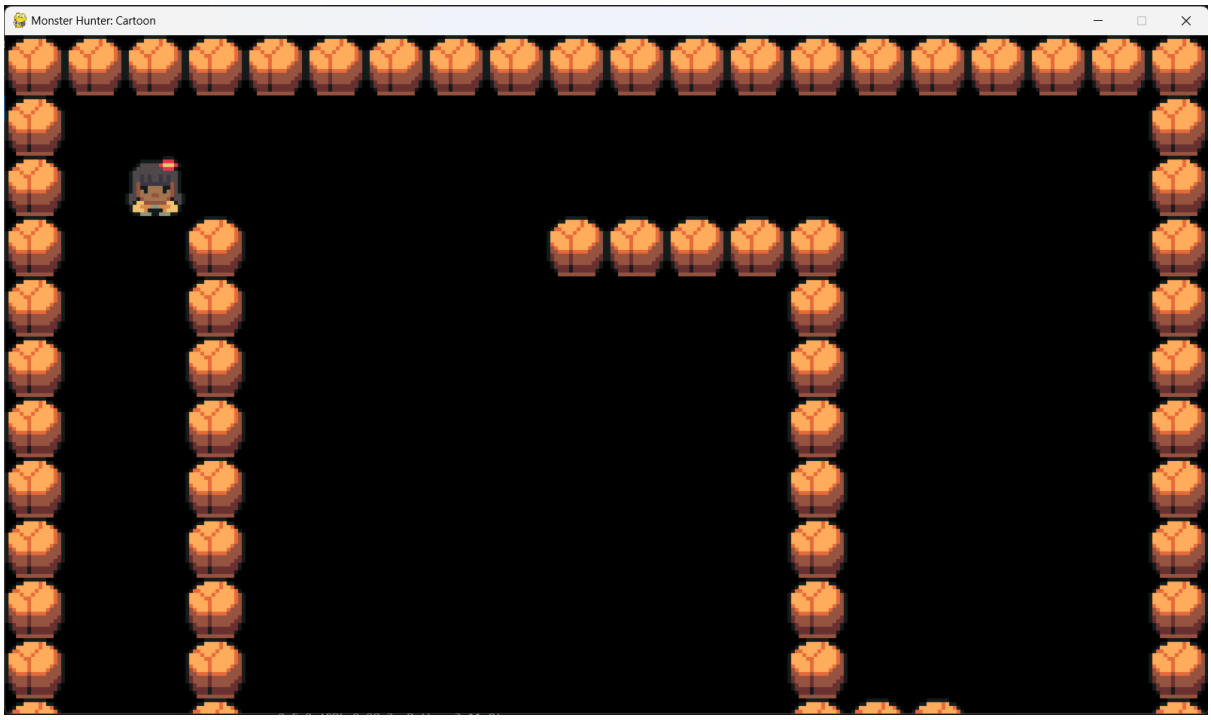
Here comes the error, there's no attribute called 'obstacles_sprites'. I go back to the class definition which find out that I lack the self definition.

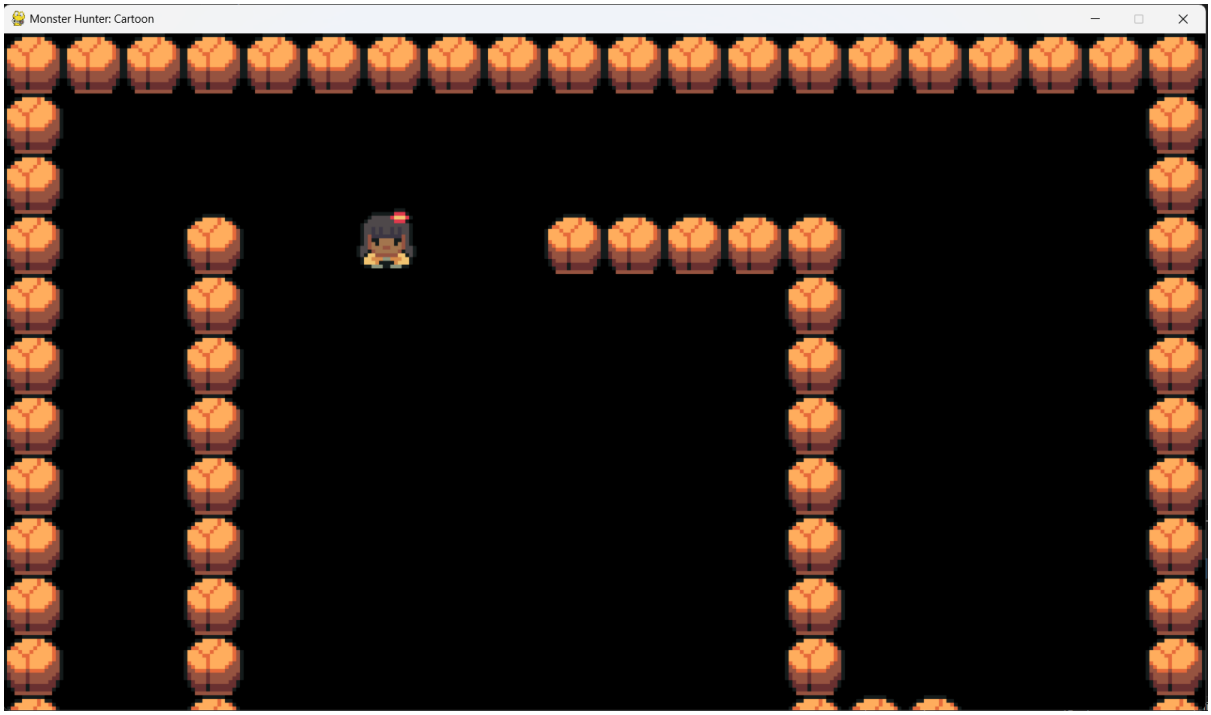
```
self.obstacles_sprites = obstacles_sprites
```

The ‘obstacles_sprites’ is defined in level section as a group of sprites, so I can directly inherit it to the player section. Then I test again, a new problem comes. It says in the level section there is no obstacles_sprites.

```
TypeError: Player.__init__() missing 1 required positional argument: 'obstacles_sprites'
```

```
    if col == 'x':
        Tile((x,y),[self.visible_sprites,self.obstacles_sprites])
    if col == 'p':
        Player((x,y),[self.visible_sprites],self.obstacles_sprites)
```





I add it to the Player function and run it, it works, the player can move now. This means that the stage 3 is finished.

Stage 4: Creating the camera

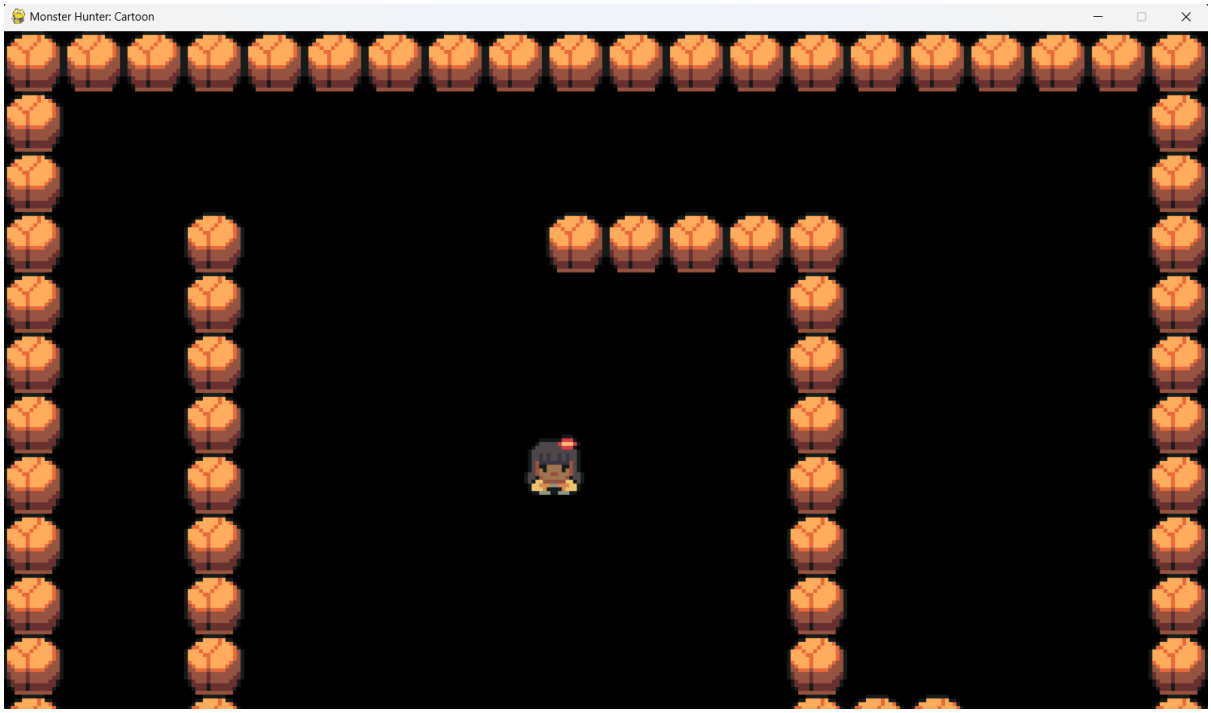
Step 1: The player can now move, but the screen do not follow the player. In this case, when the map is large, the player can not see anything if they move far away. As a result, it is necessary to create a camera which follows the sight of the character.

```
class CameraGroup(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()
```

The purpose of the code is mainly to always keep the sprite of the player at the center of the screen. As a result, it creates a similar effect as a following camera.

Now the camera has been defined and created, but it still needs a function to draw it, so I write a new function called 'custom_draw' to draw out the sights of the camera.

```
def custom_draw(self,Player):

    self.offset.x = Player.rect.centerx - self.half_width
    self.offset.y = Player.rect.centery - self.half_height

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



However, when testing, the sight does not really move, which means there's something missing. After checking the whole level section, the reason of the error is that my 'visible_sprites' group should now equal the 'CameraGroup' so that all the codes about the camera can be implemented.

```
self.visible_sprites = CameraGroup()
```

```
def run(self):
    self.visible_sprites.custom_draw(self.Player)
    self.visible_sprites.update()
```

```
  File "c:\CS\level.py", line 28, in run
    self.visible_sprites.custom_draw(self.Player)
                                     ^^^^^^^^^^^
AttributeError: 'Level' object has no attribute 'Player'
```

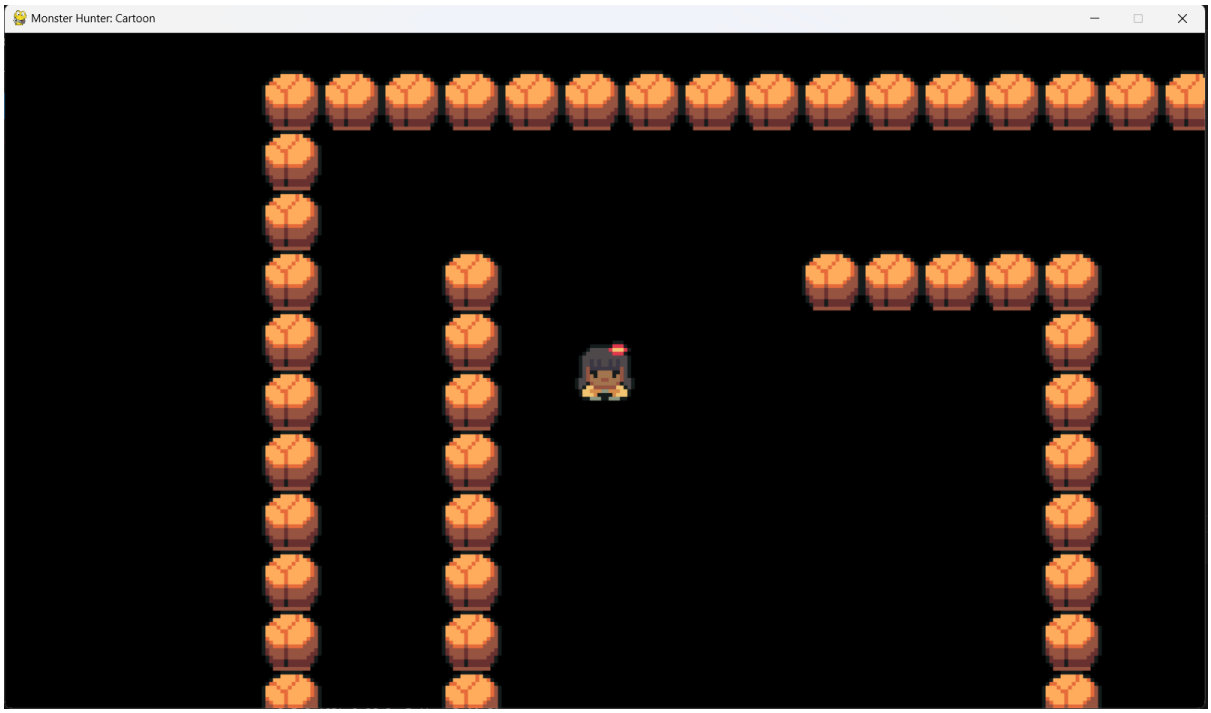
It still shows error code which the error is that I do not use an instance from the group 'Player', instead I use the whole group 'Player', and the group 'Player' can not be a parameter for custom_draw, so it shows an error.

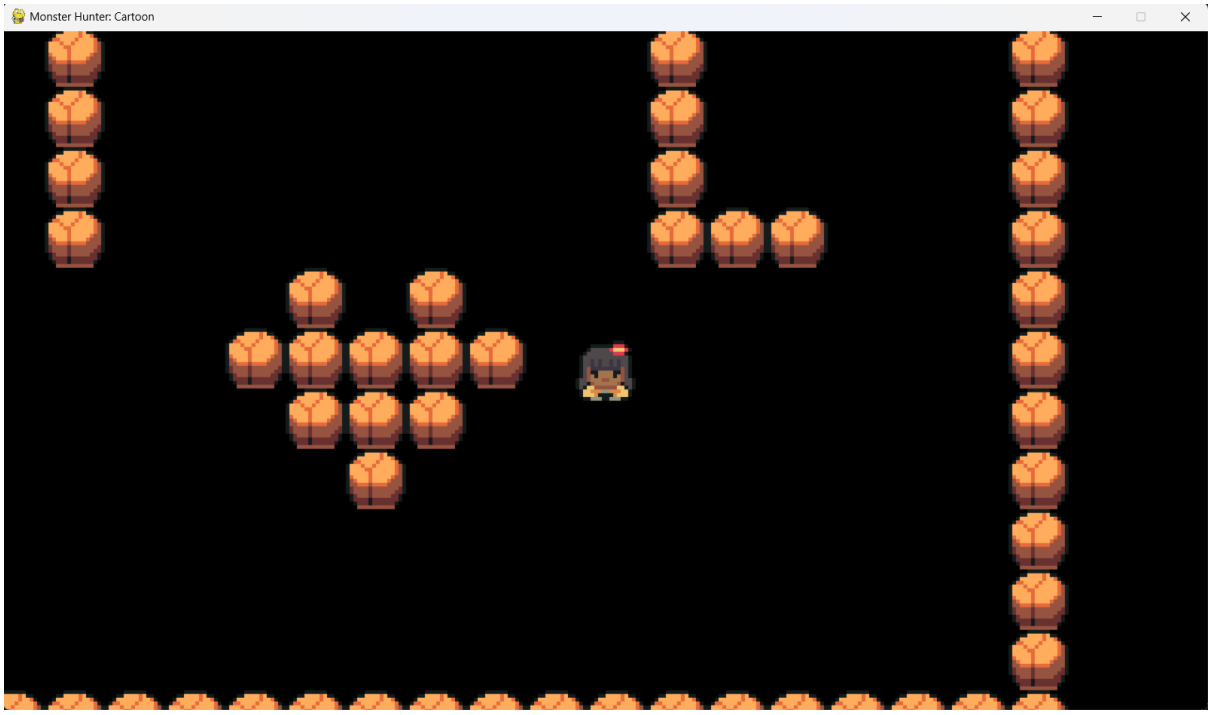
```
if col == 'p':
    self.player = Player((x,y),[self.visible_sprites],self.obstacles_sprites)
```

```
self.offset.x = player.rect.centerx - self.half_width
self.offset.y = player.rect.centery - self.half_height
```

```
def run(self):
    self.visible_sprites.custom_draw(self.player)
    self.visible_sprites.update()
```

These three places are where I used ‘Player’ instead of ‘player’, and the first screen shoot shows the process of creating an instance of ‘Player’. Following that, I test again.





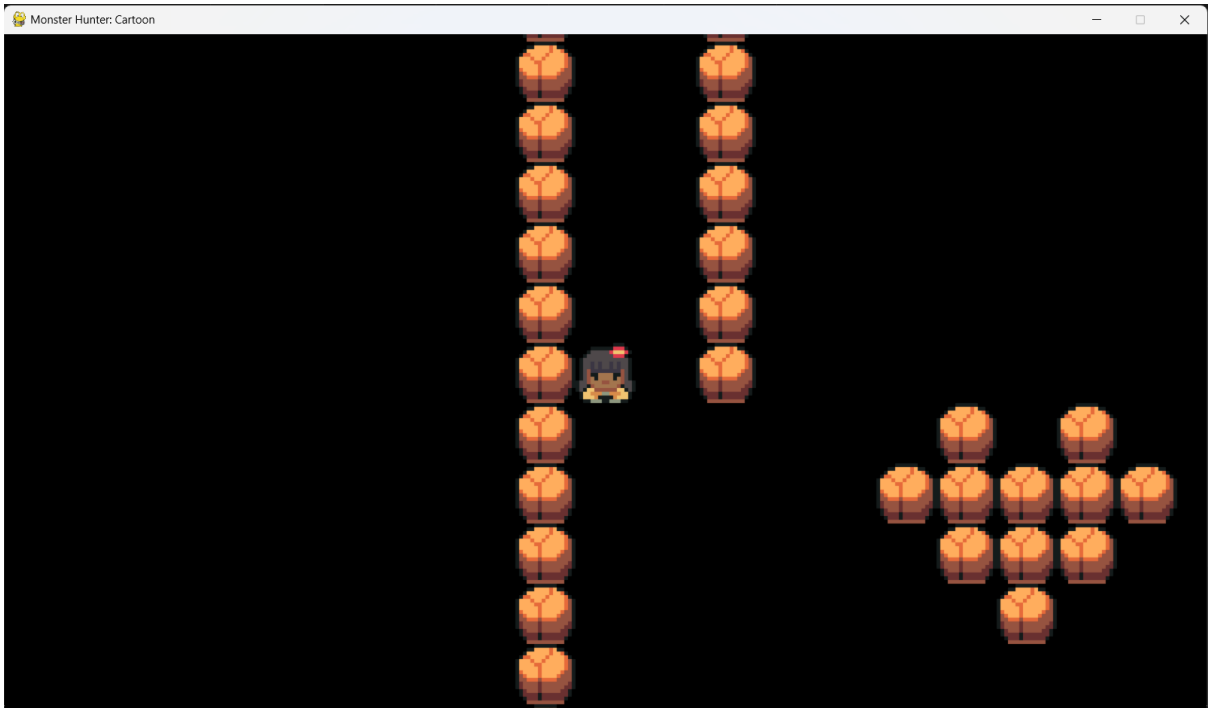
Now the camera is working successfully, so this step has been finished.

Step 2: I want to make the game more realistic, so I decide to create overlapping between players and stones, so that the player may be able to hide behind trees or stones when they play the game.

The key to create the effect of overlapping is to create hitboxes. Hitboxes are boxes which show the real volumn of the sprites. For example, if the box of the player is a square, the hitbox of it might be a rectangle with smaller area. After creating hitboxes for different sprites, the collision function can then use hitbox to decide whether to stop the sprite instead of considering the box before.

```
class Tile(pygame.sprite.Sprite):
    def __init__(self,pos,groups):
        super().__init__(groups)
        self.image = pygame.image.load('C:\CS\graphics\Test\Rock.png').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)
        self.hitbox = self.rect.inflate(0,-10)
```

I first write the hitbox of the 'Tile' class, as there are fixed on the map and it is easier for testing.

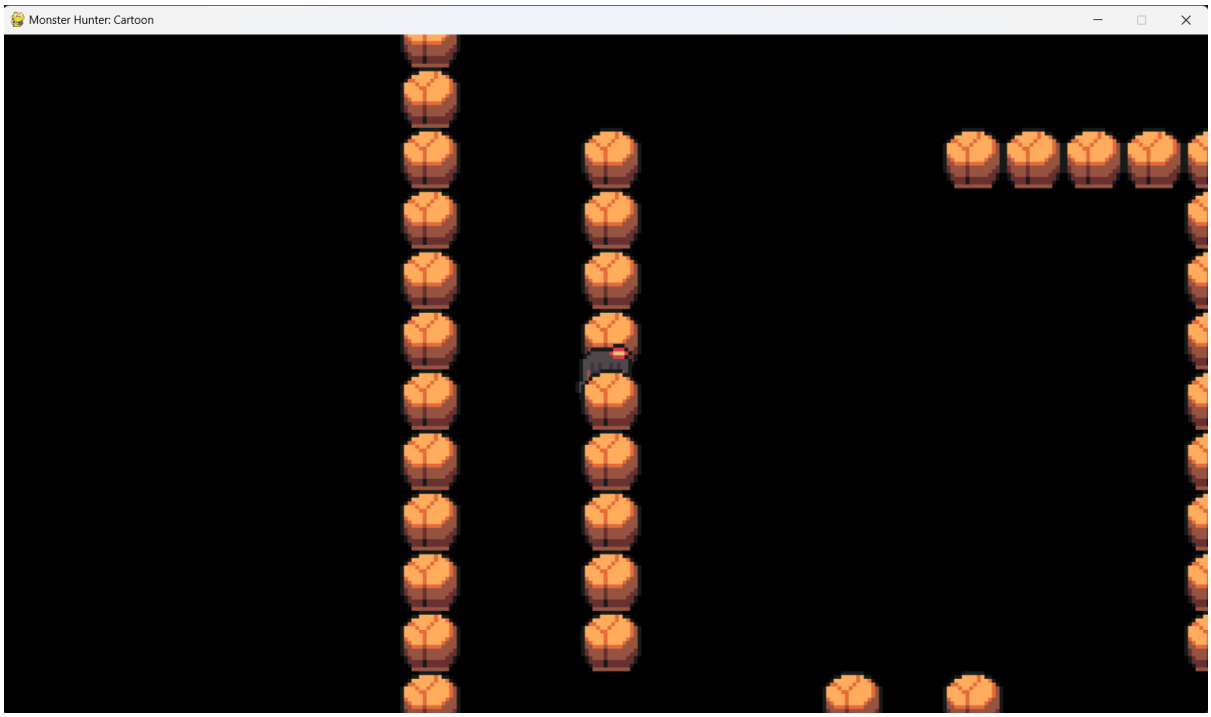


Although it doesn't work now, there should currently be no visible bugs in the tile section. As a result, the next step should be creating the hitbox for the player and then implementing them in the level section.

```
class Player(pygame.sprite.Sprite):
    def __init__(self,pos,groups,obstacles_sprites):
        (parameter) self: Self@Player
                                    'C:\CS\graphics\Test\Player.png').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)
        self.hitbox = self.rect.inflate(0,-26)
```

```
def collision(self,direction):
    if direction == 'horizontal':
        for sprite in self.obstacles_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.obstacles_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
```

These two steps I first create a hitbox for the player, then change all the rectangle in the collision to hitbox as the rectangle before should not be used for collision calculations. Then I test it.



It goes wrong which the hitbox doesn’t work and the rectangle before doesn't work either.

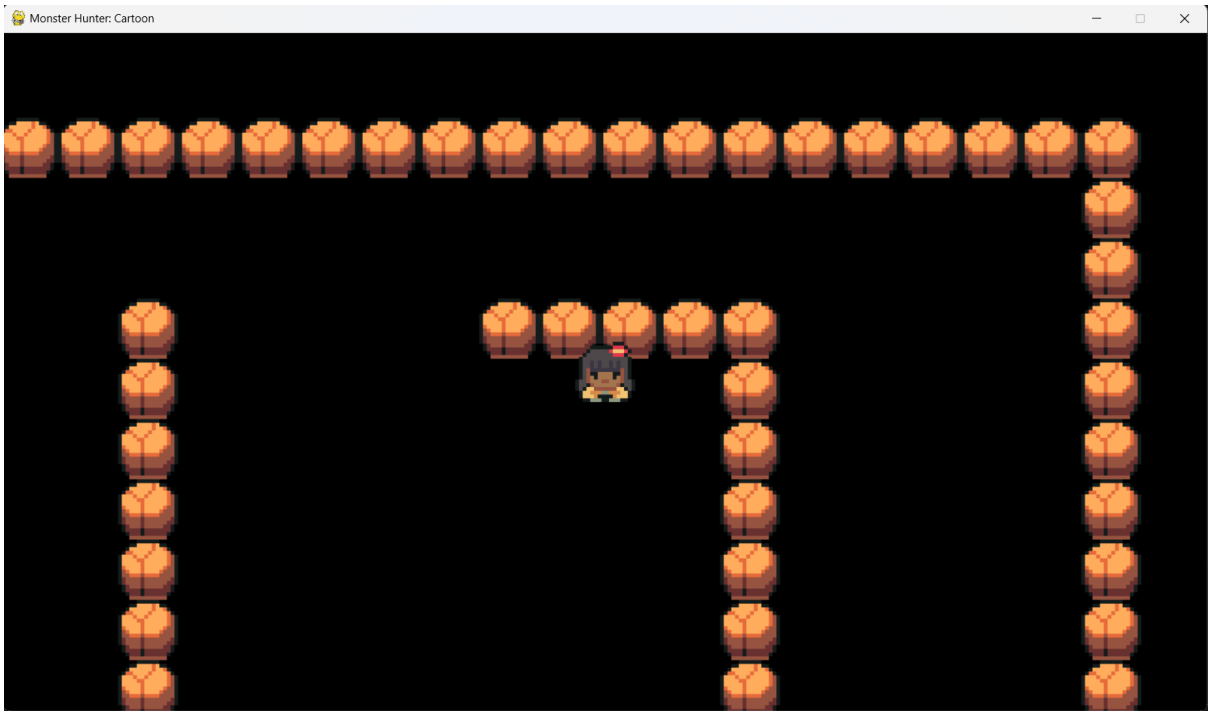
```
self.rect.x += self.direction.x * speed
self.collision('horizontal')
self.rect.y += self.direction.y * speed
self.collision('vertical')
```

Here comes the logic error. The hitboxes now apply on both the player and the stones, but the collision speed and stopping behaviours are still considering the rectangles before, which those rectangles now do not work. Therefore, just change all the 'rect' to hitbox so that the speed and collision can count normally.

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

New problem comes. The player can not move now. I think the hitbox should work correctly, but the player's motion seems to be affected by the rectangles, so the rectangles should move with the hitbox, which means they should have the same center. As a result, I try to make their centers to be the same point all the time.

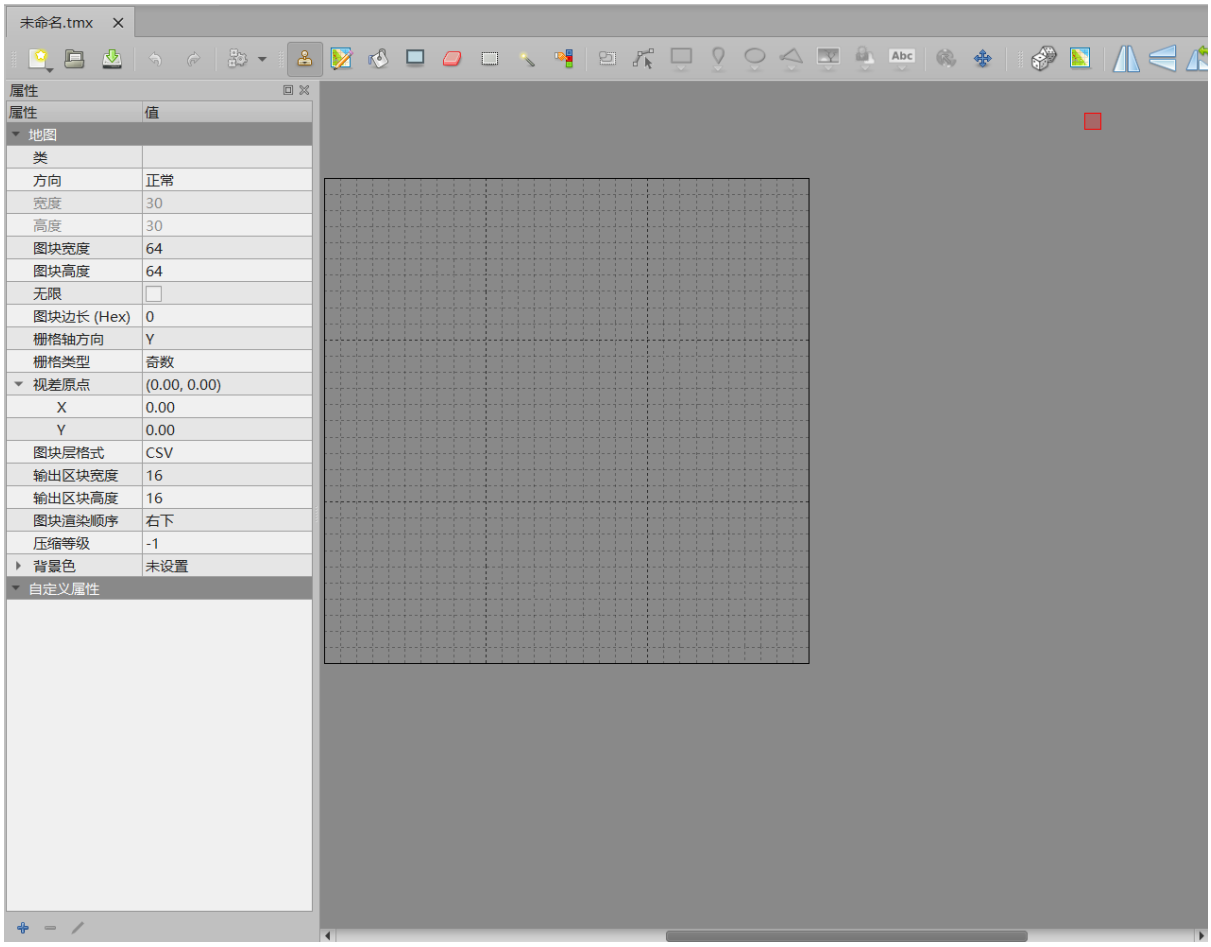
```
self.hitbox.x += self.direction.x * speed
self.collision('horizontal')
self.hitbox.y += self.direction.y * speed
self.collision('vertical')
self.rect.center = self.hitbox.center
```



Now overlap does work! When the player is behind stones, they can hide; when they are in front of stones, they will cover the stones. This is what I want to achieve and this stage is finished.

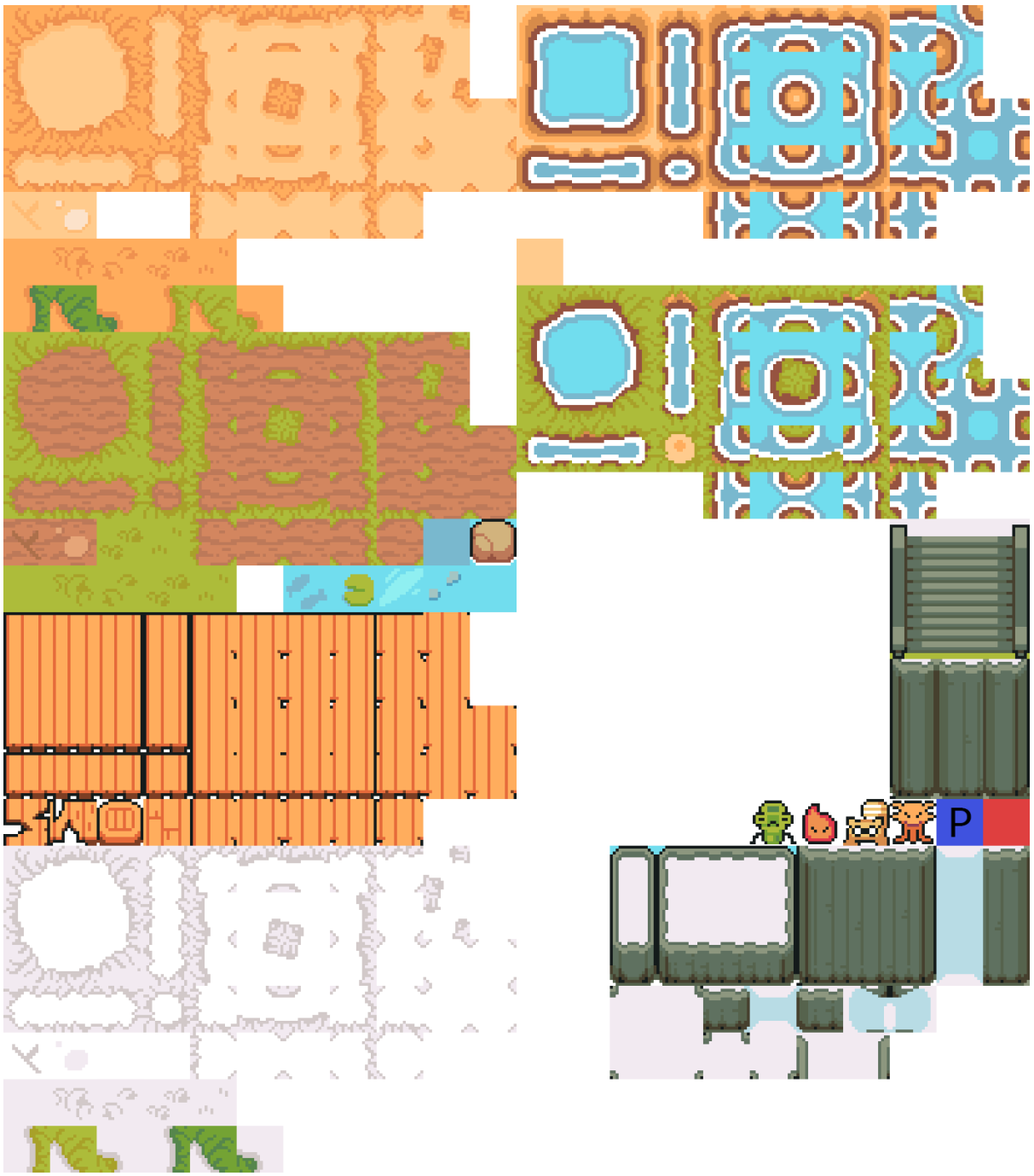
Stage 5: Graphics

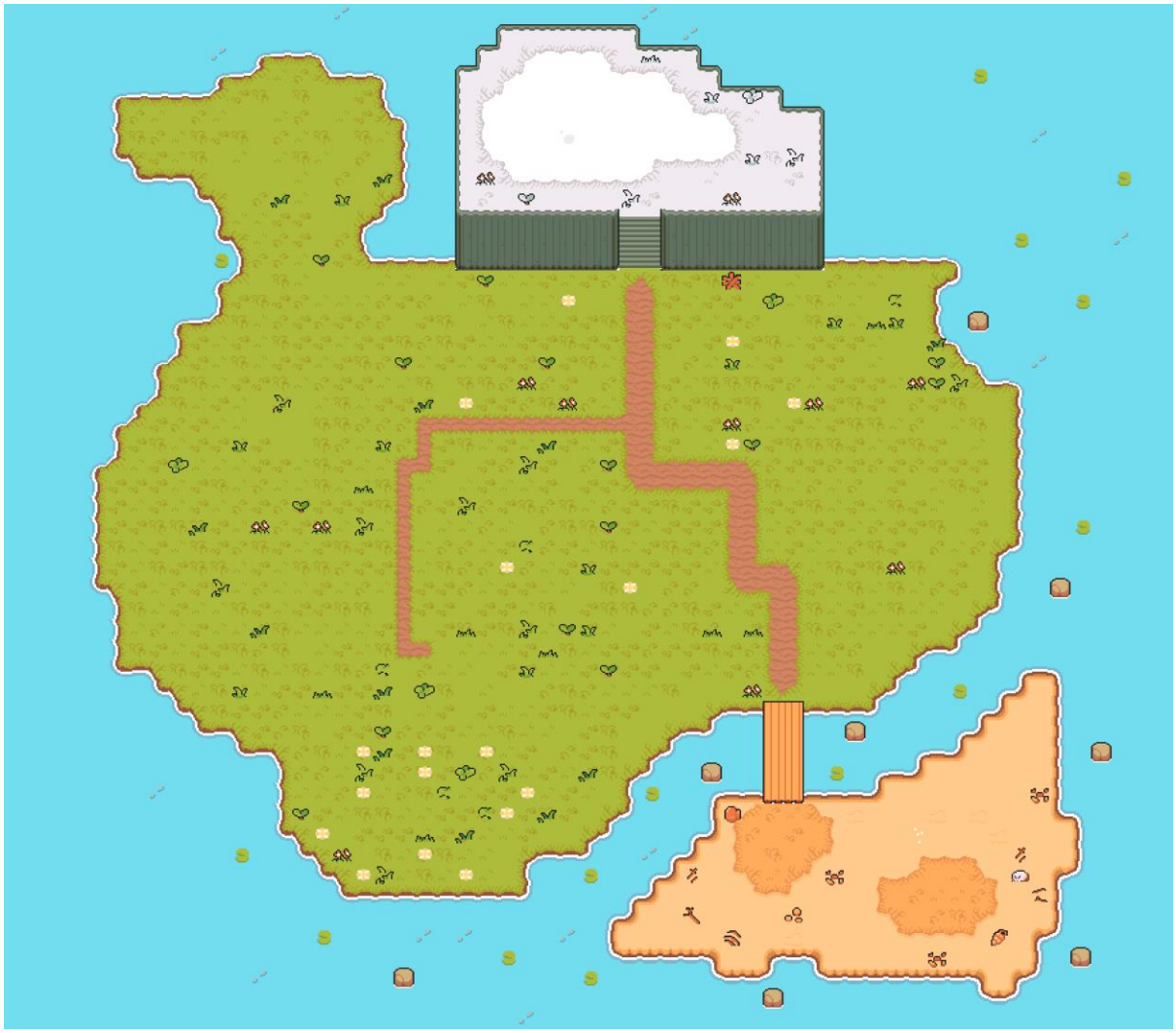
Step 1: Learn to use a software called “Tiled”.



My software is in Chinese, but it is easy to introduce all the functions and abilities of this software. It is first to create a large map with fixed tile size, the length and width are also set at the start.

I will not show the process of drawing the map here, which I can just show the ‘tiles’ I used for the map and the final product.


P



Then the map is designed which I do not need the WORLD_MAP I designed before for just testing purpose. As a result, there's very large changes on my code, which mainly change the map (putting the map picture on it as a background) and define all the things now on the map as visible_sprites (which the player can walk through).

```
self.floor_surf = pygame.image.load('C:\CS\graphics\Tilemap\ground.png').convert()
self.floor_rect = self.floor_surf.get_rect(topleft = (0,0))
```

Step 2: Tail now, the map is still not shown, it is because I haven't imported the file from the Tiled as a csv file, which csv file can mark all the objects I create on the map and then tell the computers where they are and what they are, which is the key of making this game map. At the same time, I should also create different groups of sprites to identify them as visible or obstacles so that they can behave like what I want them to, such as stopping the player.

If I need to use csv files, I should first import the csv into python and then my computer can read the statements about the csv files. To make it clear, I create a new section called 'support', which includes all the import and necessary settings of using csv files.

```
from csv import reader
from os import walk
import pygame
```

Then I need to define two functions of using csv files, which one is used for importing single layout and one is used for importing folders.

```
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 _,__,img_files in walk(path):
        for image in img_files:
            full_path = path + '/' + image
            image_surf = pygame.image.load(full_path).convert_alpha()
            surface_list.append(image_surf)

    return surface_list
```

After finishing the support section, I can pass to the level section to identify different objects.

```
def create_map(self):
    layouts = {
        'boundary': import_csv_layout('C:\CS\map\map_FloorBlocks.csv'),
        'grass': import_csv_layout('C:\CS\map\map_Grass.csv'),
        'object': import_csv_layout('C:\CS\map\map_Objects.csv'),
    }

    graphics = {
        'grass': import_folder('C:\CS\graphics\grass'),
        'objects': import_folder('C:\CS\graphics\objects')
    }
```

Now there are just two kinds of objects used (as I do not know if it works, if there is anything wrong, there's no need for large amount of modification). They are all imported from different locations for different purposes.

Although it still does not show anything, the importing of csv files works, which means I can continue working on this direction.

```
    for style,layout in layouts.items():
        for row_index,row in enumerate(layout):
            for col_index, col in enumerate(row):
                if col != '-1':
                    x = col_index * 64
                    y = row_index * 64

                    if style == 'boundary':
                        Tile((x,y),[self.obstacles_sprites],'invisible')

                    if style == 'grass':
                        random_grass_image = choice(graphics['grass'])
                        Tile((x,y),[self.visible_sprites,self.obstacles_sprites],'grass',random_grass_image)

                    if style == 'object':
                        surf = graphics['objects'][int(col)]
                        Tile((x,y),[self.visible_sprites,self.obstacles_sprites],'object',surf)

    self.player = Player((2000,1430),[self.visible_sprites],self.obstacles_sprites)
```

The code is pretty similar as before but just adding some new objects listed on the last screen shoot. Also, because of the large area of the map, the initial coordinates of the player is also changed.

```
  File "c:\CS\level.py", line 15, in __init__
    self.create_map()
  File "c:\CS\level.py", line 37, in create_map
    Tile((x,y),[self.obstacles_sprites],'invisible')
TypeError: Tile.__init__() takes 3 positional arguments but 4 were given
```

After testing, error is found. The error mainly comes from that the range of choosing random images is not defined, so when it takes a number which there is not an image which is at that number, then the tile can not be found and it outputs an error code.

```
class Tile(pygame.sprite.Sprite):
    def __init__(self,pos,groups,sprite_type,surface = pygame.Surface((64,64))):
        super().__init__(groups)
        self.sprite_type = sprite_type
        self.image = surface
        if sprite_type == 'object':
            self.rect = self.image.get_rect(topleft = (pos[0],pos[1] - 64))
        else:
            self.rect = self.image.get_rect(topleft = pos)
        self.hitbox = self.rect.inflate(0,-10)
```

Now, after defining the range of the images, the tiles are shown. However, the background is not shown which means there must be some other issue in the level stage.

```
def custom_draw(self,player):

    self.offset.x = player.rect.centerx - self.half_width
    self.offset.y = player.rect.centery - self.half_height

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

The issue is found. In the custom_draw function, the floor is not drawn, which means the background map is not updated. To solve this problem, just add statements to draw the floor that I create before.

```
floor_offset_pos = self.floor_rect.topleft - self.offset
self.display_surface.blit(self.floor_surf,floor_offset_pos)
```


Monster Hunter: Cartoon
Monster Hunter: Cartoon

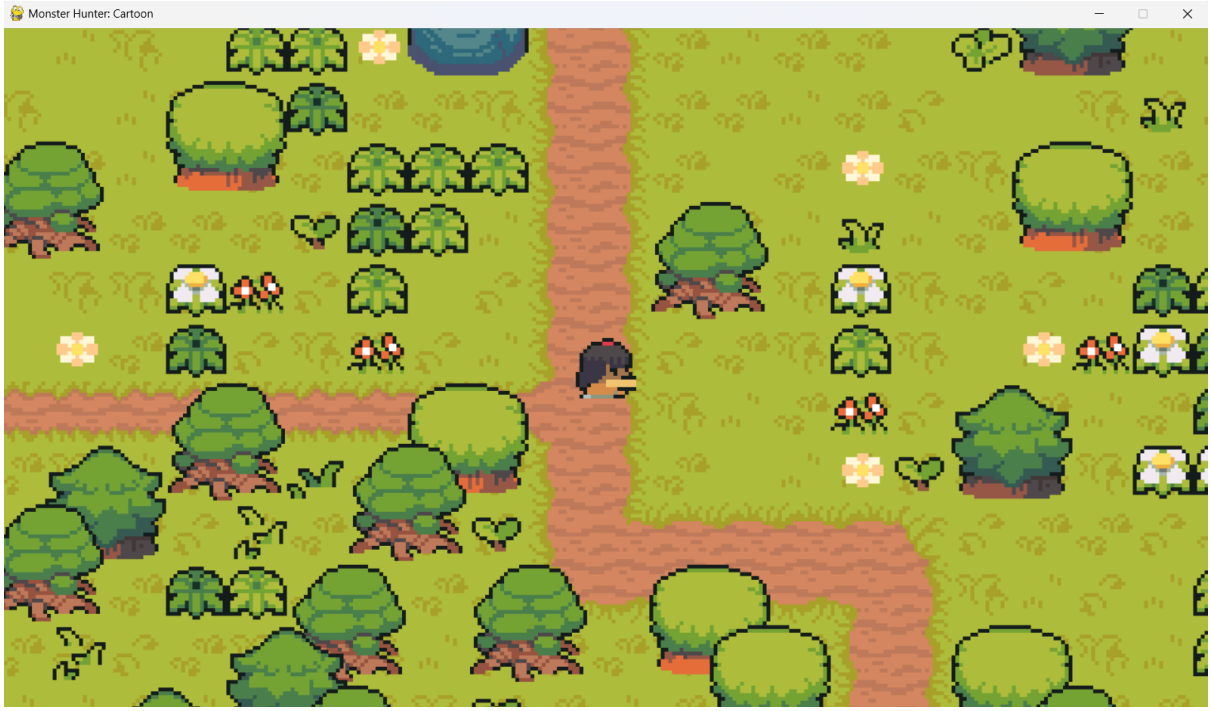
After writing the statement about the floor, the whole map is loaded and the game looks pretty successful until this stage. After testing, all the things before (moving, collision, overlap or camera) still keep working which is great.

Stage 6: Player animations

Step 1: To start with, let me create some basic inputs of the player's motion, such as attack and magic.

```
if keys[pygame.K_SPACE]:
    self.attacking = True
    print('Attack')
```

```
if keys[pygame.K_LCTRL]:
    self.attacking = True
    print('magic')
```



Attacking is a kind of status which it will be used in the steps afterwards for animation. I just give attack the SPACE key and give magic the left CTRL key. It is pretty easy to use them. After testing, pressing both SPACE and CTRL the player will show the attack image.

Step 2: Then I would like to start working on the facing direction of the player. My target is that when walking left the player is facing left and so on, so the direction of the player is the same as the direction of motion.

To create this, I need to create a group of images which contain the images of the player facing every direction.

```
def import_player_assets(self):
    character_path = 'C:\CS\graphics\player'
    self.animations = {'up': [],'down': [],'left': [],'right': [],
        'right_idle':[],'left_idle':[],'up_idle':[],'down_idle':[],
        'right_attack':[],'left_attack':[],'up_attack':[],'down_attack':[]}

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

Those images are all named and put in the right folder for the function. However, the computer will not know what status the player is at, so I have to create a variable called ‘status’ so that the algorithm can identify which image to use by considering the ‘status’. In the design, I have said I will give 3 kinds of status, which are ‘idle’ , ‘attack’ and ‘move’. Those 3 status are 3 different situations and the image will be different, which now in the animations group there are 12 images for use as each status will have 4 directions of images.

```
def input(self):
    if not self.attacking:
        keys = pygame.key.get_pressed()

        if keys[pygame.K_UP]:
            self.direction.y = -1
            self.status = 'up'
        elif keys[pygame.K_DOWN]:
            self.direction.y = 1
            self.status = 'down'
        else:
            self.direction.y = 0

        if keys[pygame.K_RIGHT]:
            self.direction.x = 1
            self.status = 'right'
        elif keys[pygame.K_LEFT]:
            self.direction.x = -1
            self.status = 'left'
        else:
            self.direction.x = 0
```

```
def get_status(self):

    if self.direction.x == 0 and self.direction.y == 0:
        if not 'idle' in self.status and not 'attack' in self.status:
            self.status = self.status + '_idle'

    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','_attack')
            else:
                self.status = self.status + '_attack'
    else:
        if 'attack' in self.status:
            self.status = self.status.replace('_attack','')
```

After the status has been created, now the images can be loaded by different status.

Step 3: Now it's time to use those images to create animations. As I mentioned in the design stage, my animation is just to switch between different images in very short period of time, which if fast enough it looks like the image is moving.

```
self.import_player_assets()
self.status = 'down'
self.frame_index = 0
self.animation_speed = 0.15

self.direction = pygame.math.Vector2()
self.speed = 5
self.attacking = False
self.attack_cooldown = 400
self.attack_time = None

self.obstacle_sprites = obstacle_sprites
```

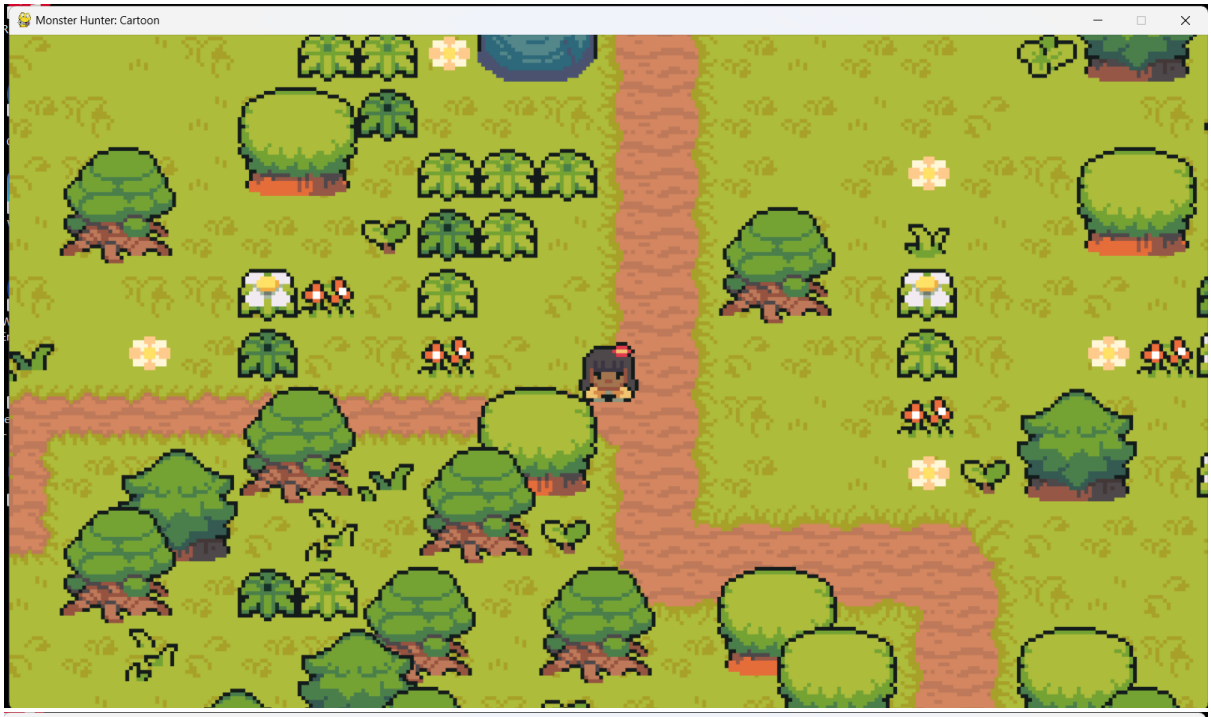
I will repeat switching 4 images and I've given them indexes so that it is pretty convenient for them to switch in order.

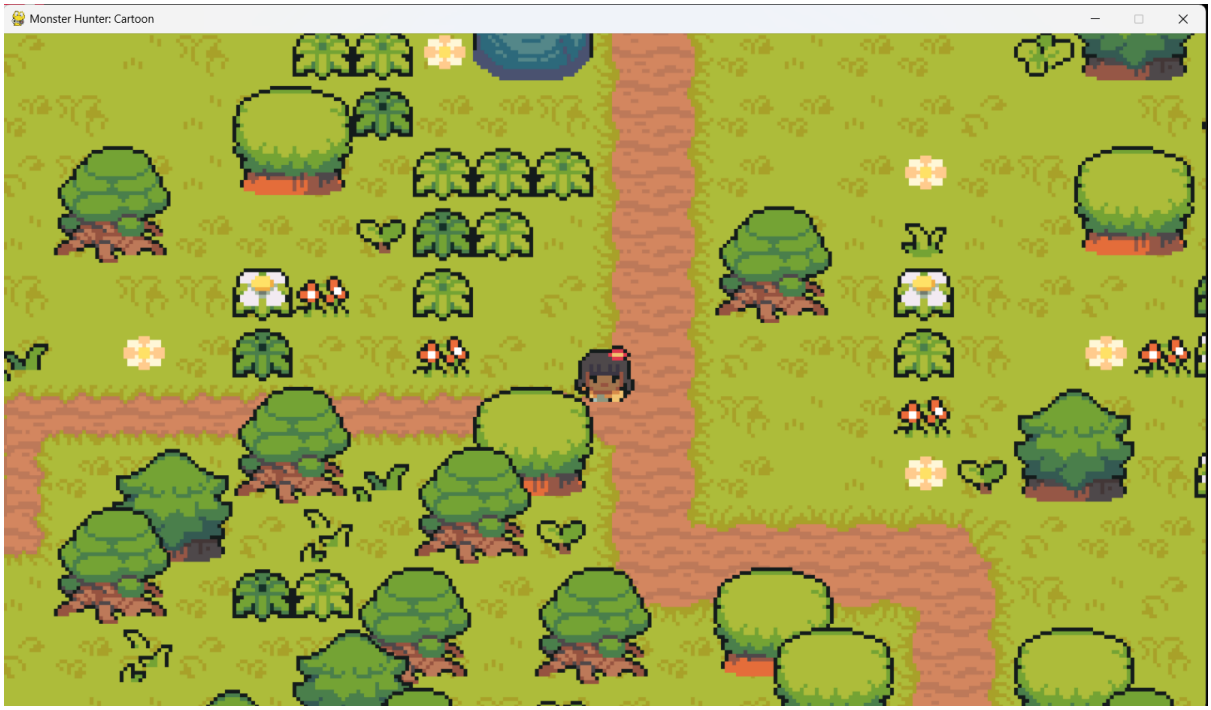
```
def animate(self):
    animation = self.animations[self.status]

    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)
```

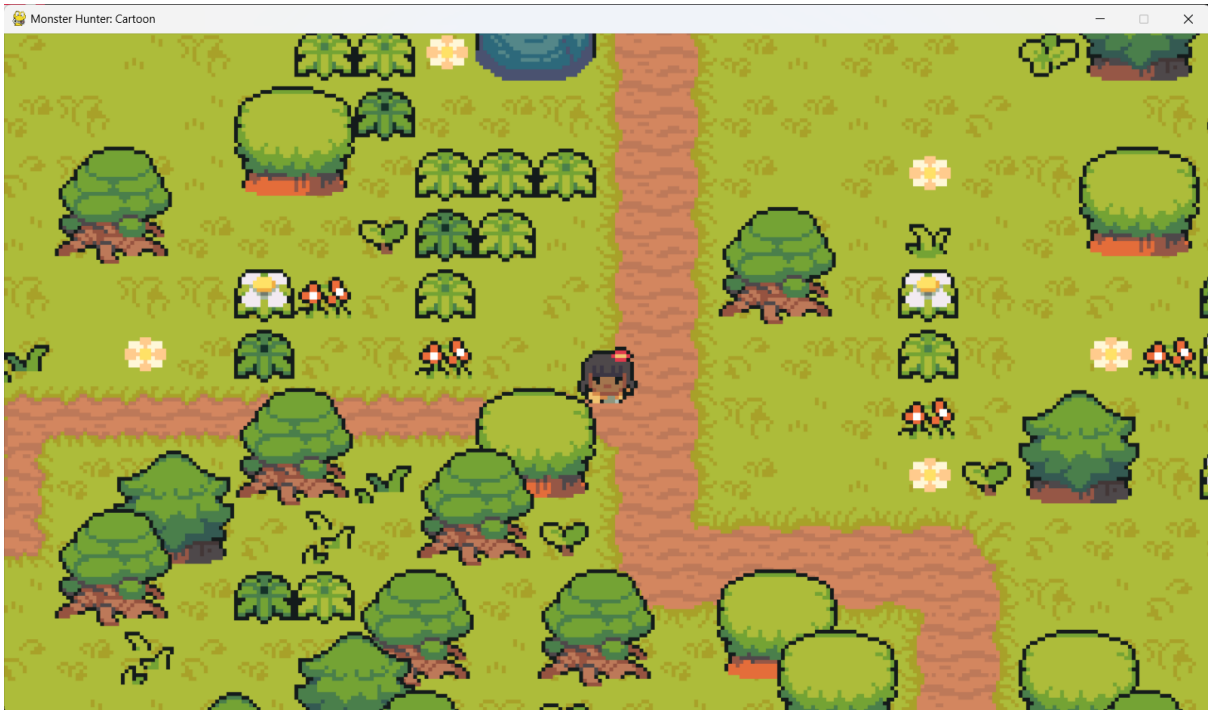
```
File "c:\CS\player.py", line 126, in animate
  self.image = animation[int(self.frame_index)]
               ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^
IndexError: list index out of range
```

That is the code for the animation, which I call it 'animate'. However, when testing the error comes out. The problem is mainly saying that the image of that path is not found so it outputs an index error.

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


Monster Hunter: Cartoon


Monster Hunter: Cartoon



After I add the '/', the path is full and all the images can be found in the folder. After testing, from the screen shoot we can see the character is actually changing her position in fixed period of time, which achieves the effect of animation.

Step 4: The game should also have an inner cool down system which will count different cool downs (attack, magic, maybe switch weapon and magic afterwards). This is pretty important as if no cool down the game will be pretty easy to crash if the input is too frequent.

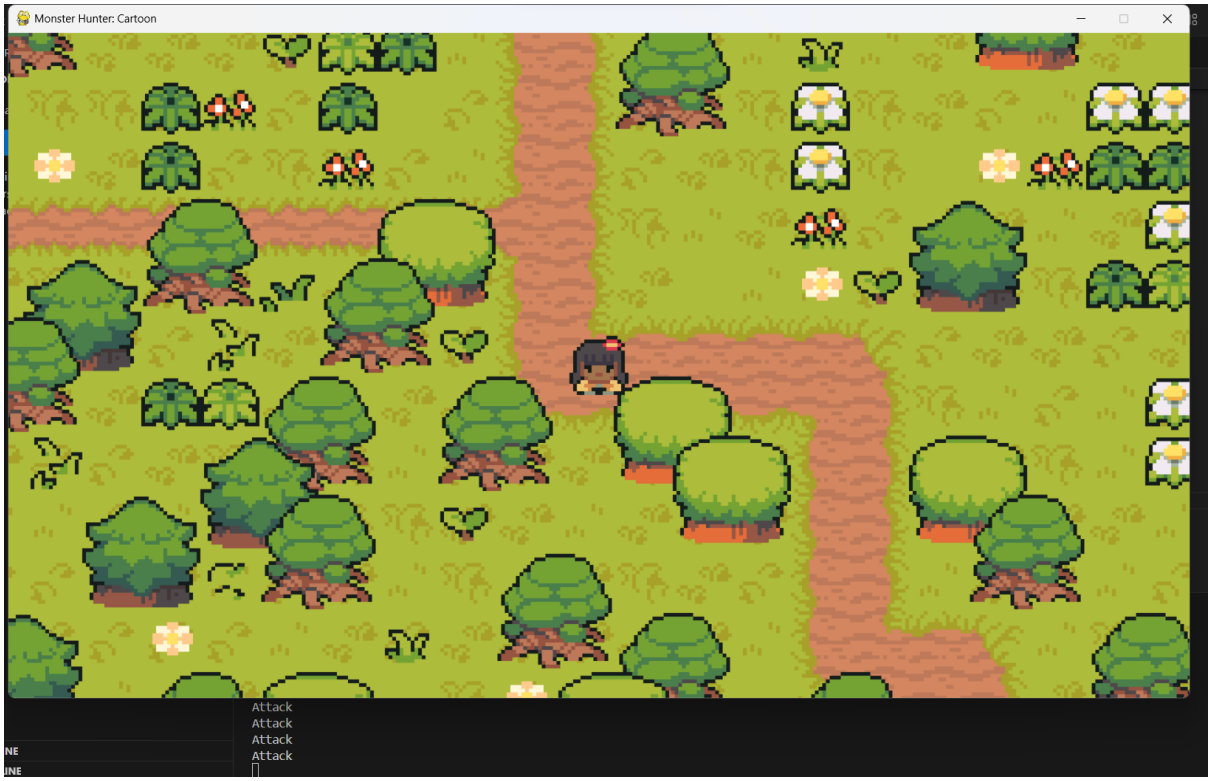
```
self.attack_cooldown = 400
self.attack_time = None
```

```
if keys[pygame.K_SPACE]:
    self.attacking = True
    self.attack_time = pygame.time.get_ticks()
    print('Attack')
```

```
if keys[pygame.K_LCTRL]:
    self.attacking = True
    self.attack_time = pygame.time.get_ticks()
    print('magic')
```

```
def cooldowns(self):
    current_time = pygame.time.get_ticks()

    if self.attacking:
        if current_time - self.attack_time >= self.attack_cooldown:
            self.attacking = False
```

The cool down system is not complex at all, as the key inside is to reset the self.attacking to False when it is during cool down period. Because of the False of attacking, the player will keep the status as 'idle', which will then not show attack and the damage can not apply.



When testing, I repeat taping the space, but the attack is printed with clear interval and the animation is not shown during the interval, which means the cool down system is successful.

When the cool down is finished, it is the end of the whole stage 6.

Stage 7: Weapons

Step 1: At the start of this stage, it is necessary to first upload the images of the weapons and then show them in the game.

```
import pygame

class Weapon(pygame.sprite.Sprite):
    def __init__(self,player,groups):
        super().__init__(groups)
        direction = player.status.split('_')[0]

        full_path = 'C:\CS\graphics\weapons\{player.weapon}\{direction}.png'
        self.image = pygame.image.load(full_path).convert_alpha()

        if direction == 'right':
            self.rect = self.image.get_rect(midleft = player.rect.midright + pygame.math.Vector2(0,16))
        elif direction == 'left':
            self.rect = self.image.get_rect(midright = player.rect.midleft + pygame.math.Vector2(0,16))
        elif direction == 'down':
            self.rect = self.image.get_rect(midtop = player.rect.midbottom + pygame.math.Vector2(-10,0))
        else:
            self.rect = self.image.get_rect(midbottom = player.rect.midtop + pygame.math.Vector2(-10,0))
```

To start with, I create a new section called 'weapon' which is only used for weapon settings. It imports all the images of the weapons and also give out the direction of the weapons. It also creates the interaction rectangles for the weapons, so they can interact with other objects such as monsters. It is the preparation for the following codes about damage and enemies.

These are the five weapons I designed for the player, the player can switch between them and they have different attack range and damage.

```
weapon_data = {
    'sword': {'cooldown': 100, 'damage': 15,'graphic':'C:\CS\graphics\weapons\Sword\Full.png'},
    'lance': {'cooldown': 400, 'damage': 30,'graphic':'C:\CS\graphics\weapons\Lance\Full.png'},
    'axe': {'cooldown': 300, 'damage': 20, 'graphic':'C:\CS\graphics\weapons\Axe\Full.png'},
    'rapier':{'cooldown': 50, 'damage': 8, 'graphic':'C:\CS\graphics\weapons\Rapier\Full.png'},
    'sai':{'cooldown': 80, 'damage': 10, 'graphic':'C:\CS\graphics\weapons\Sai\Full.png'}}
```

In the settings section, I add the cooldown, damage and image paths of those weapons so that in the following parts those data can be directly imported from the settings, which means not need to repeat typing the paths or data.

Step 2: Next should be adding the weapon interaction into the attack function. As the interaction boxes have been created, the thing I need to do is just to build functions in the weapon section and then import them in the level section.

```
self.current_attack = None
```

```
def create_attack(self):
    self.current_attack = Weapon(self.player,[self.visible_sprites])

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

Firstly, I define two new functions called create_attack and destroy_attack. Create_attack is mainly to interact with other objects, so that damage can be applied. Destroy_attack is a function to define the status of enemies, to see if they are killed so that the destroy animation can be shown.

```
def __init__(self,pos,groups,obstacle_sprites,create_attack,destroy_attack):
```

```
self.direction = pygame.math.Vector2()
self.speed = 5
self.attacking = False
self.attack_cooldown = 400
self.attack_time = None
self.obstacle_sprites = obstacle_sprites

self.create_attack = create_attack
self.destroy_attack = destroy_attack
self.weapon_index = 0
self.weapon = list(weapon_data.keys())[self.weapon_index]
self.can_switch_weapon = True
self.weapon_switch_time = None
self.switch_duration_cooldown = 200
```

Next, I add the two new methods to the Player class, as they are both player's behaviours. Furthermore, I add the cooldown system for the weapons, which include attacking cooldown and switching cooldown.

```
if self.attacking:
    if current_time - self.attack_time >= self.attack_cooldown:
        self.attacking = False
        self.destroy_attack()

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

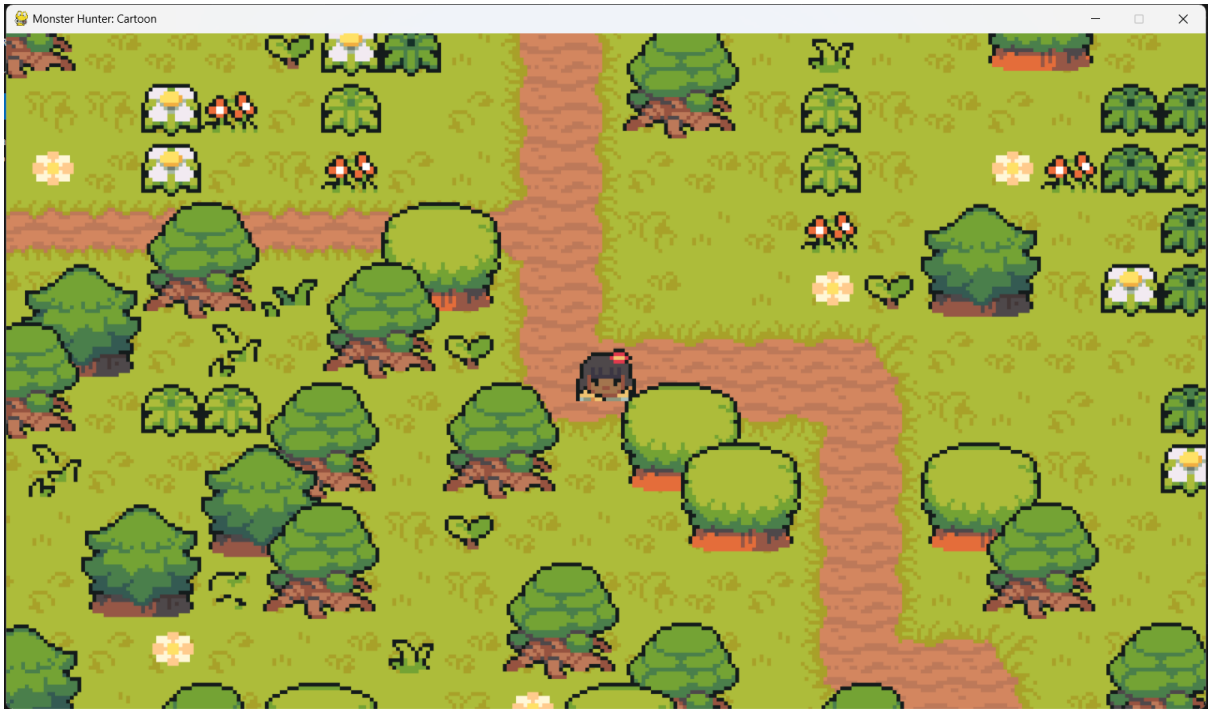
I also add some statements in the cooldown function before, which is destroy_attack for interaction. The purpose of the third and fourth if is mainly to limit the timing of switching weapon, it only allows the player to switch weapon when they are not attacking.

```
  self.player = Player((2000,1430),[self.visible_sprites],self.obstacle_sprites)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ypeError: Player.__init__() missing 2 required positional arguments: 'create_attack' and 'destroy_attack'
```

The error comes, it says I miss the two methods in the player instance of the level section.

```
self.player = Player((2000,1430),[self.visible_sprites],self.obstacle_sprites,self.create_attack,self.destroy_attack)
```

It is easy to solve this issue, just add the two methods to the instance and then the game can run.



However, the new problem comes. There is no image of weapons loaded in the game, which means that it is unsuccessful to implement.

```
super().__init__(groups)
direction = player.status.split('_')[0]
print(direction)
```

After testing, I just find the error. When I want to print the direction of the weapon, it prints nothing, meaning that I do not get the status of the player, so the path of the image of weapons is not completed and they can be loaded.

```
good = self.status.split('_')[0]
print(good)
```

```
Down
Down
Down
Down
Down
Down
Down
Down
Down
```

When I print it in the player section, it is actually working.

Stage 8: UI

Step 1: I need to give the initial data I set for the player, including health, energy, attack, magic and speed, I would call them all as stats.

```
self.stats = {'health': 100,'energy':60,'attack': 10,'magic': 4,'speed': 5}
```

```
self.direction = pygame.math.Vector2()
self.speed = 5
```

```
self.direction = pygame.math.Vector2()

self.attacking = False
```

This is what I set for the player to born with. They do have clear data which will be pretty helpful for further coding. As speed is given in stats, there's no need to set it other places, so I will delete the one I use before.

Step 2: Next it is time to work on the real UI for the data.

```
BAR_HEIGHT = 20
HEALTH_BAR_WIDTH = 200
ENERGY_BAR_WIDTH = 140
ITEM_BOX_SIZE = 80
UI_FONT = '../graphics/font/joystix.ttf'
UI_FONT_SIZE = 18

WATER_COLOR = '#71ddee'
UI_BG_COLOR = '#222222'
UI_BORDER_COLOR = '#111111'
TEXT_COLOR = '#EEEEEE'

HEALTH_COLOR = 'red'
ENERGY_COLOR = 'blue'
UI_BORDER_COLOR_ACTIVE = 'gold'
```

Here are some simple terms or variables that I will need for creating the UI, for example the bar things are used to create the UI for health and energy and those colours are mainly for drawing. After preparing these, I can then pass to the stage of coding.

```
import pygame
from settings import *

class UI:
    def __init__(self):

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

        self.health_bar_rect = pygame.Rect(10,10,HEALTH_BAR_WIDTH,BAR_HEIGHT)
        self.energy_bar_rect = pygame.Rect(10,34,ENERGY_BAR_WIDTH,BAR_HEIGHT)

        self.weapon_graphics = []
        for weapon in weapon_data.values():
            path = weapon['graphic']
            weapon = pygame.image.load(path).convert_alpha()
            self.weapon_graphics.append(weapon)
```

This is the new section that I create called UI, it defines the class UI and create them in this step. It might be hard to test for this step as it is not drawing the UI, so I will directly pass to the next step of drawing.

Step 3: Now it's time to draw the bars and tags.

```
def show_bar(self,current,max_amount,bg_rect,color):

    pygame.draw.rect(self.display_surface,UI_BG_COLOR,bg_rect)

    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)
    pygame.draw.rect(self.display_surface,UI_BORDER_COLOR,bg_rect,3)
```

```
def display(self,player):
    self.show_bar(player.health,player.stats['health'],self.health_bar_rect,HEALTH_COLOR)
    self.show_bar(player.energy,player.stats['energy'],self.energy_bar_rect,ENERGY_COLOR)
```

Here is the code to show the health and energy bar. It passes the test. Then I need to test when the health and energy is not full, it shows black area, so I will times them by different rate to see if it shows the lossof health and energy.

```
self.health = self.stats['health'] * 0.5
self.energy = self.stats['energy'] * 0.8
```

It is pretty clear that it shows the loss. Then, I should work on the display of exp and weapon/magic box.

```
def show_exp(self,exp):
    text_surf = self.font.render(str(int(exp)),False,TEXT_COLOR)
    x = self.display_surface.get_size()[0] - 20
    y = self.display_surface.get_size()[1] - 20
    text_rect = text_surf.get_rect(bottomright = (x,y))

    pygame.draw.rect(self.display_surface,UI_BG_COLOR,text_rect.inflate(20,20))
    self.display_surface.blit(text_surf,text_rect)
    pygame.draw.rect(self.display_surface,UI_BORDER_COLOR,text_rect.inflate(20,20),3)
```

```
def selection_box(self,left,top, has_switched):
    bg_rect = pygame.Rect(left,top,ITEM_BOX_SIZE,ITEM_BOX_SIZE)
    pygame.draw.rect(self.display_surface,UI_BG_COLOR,bg_rect)
    if has_switched:
        pygame.draw.rect(self.display_surface,UI_BORDER_COLOR_ACTIVE,bg_rect,3)
    else:
        pygame.draw.rect(self.display_surface,UI_BORDER_COLOR,bg_rect,3)
    return bg_rect
```

```
def weapon_overlay(self,weapon_index,has_switched):
    bg_rect = self.selection_box(10,630,has_switched)
    weapon_surf = self.weapon_graphics[weapon_index]
    weapon_rect = weapon_surf.get_rect(center = bg_rect.center)

    self.display_surface.blit(weapon_surf,weapon_rect)
```

They are all shown correctly, which means the UI stage has been finished.

Stage 8: Magic

Step 1: I need to create a dictionary for the magic in the settings section, so that when I need to use any of their data, I can directly import from the settings.

```
magic_data = {
    'Flame': {'strength': 5,'cost': 20,'graphic':'C:\CS\graphics\particles\Flame\Fire.png'},
    'Heal' : {'strength': 20,'cost': 10,'graphic':'C:\CS\graphics\particles\Heal\Heal.png'}
    }
```

Here is the magic data, I give them strength to measure the amount of damage or healing, cost for energy and their images.

Step 2: Next I need to use the magic data to create the input of the magic.

```
def create_magic(self,style,strength,cost):
    print(style)
    print(strength)
    print(cost)
```

```
self.player = Player(
    (2000,1430),
    [self.visible_sprites],
    self.obstacle_sprites,
    self.create_attack,
    self.destroy_attack,
    self.create_magic)
```

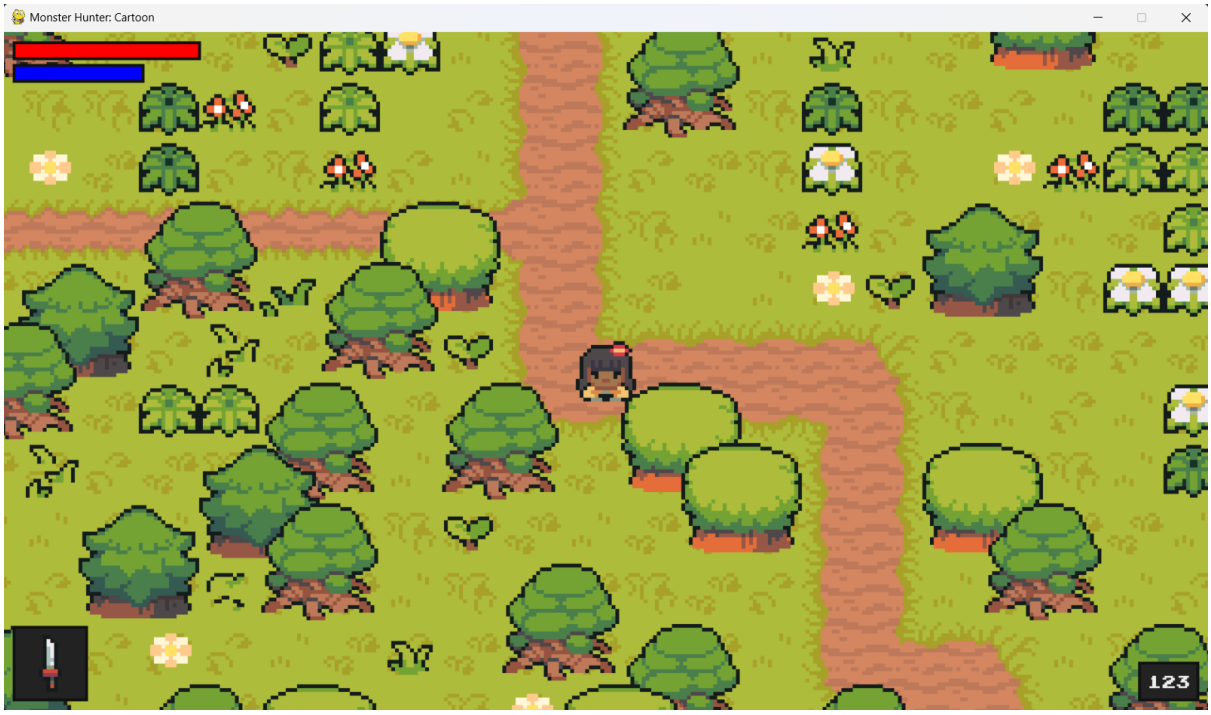
I create the new function called create magic in level section and then use it in the player section to create the input of magic.

```
def __init__(self,pos,groups,obstacle_sprites,create_attack,destroy_attack,create_magic):
```

```
self.create_magic = create_magic
self.magic_index = 0
self.magic = list(magic_data.keys())[self.magic_index]
self.can_switch_magic = True
self.magic_switch_time = None
```

```
if keys[pygame.K_LCTRL]:
    self.attacking = True
    self.attack_time = pygame.time.get_ticks()
    style = list(magic_data.keys())[self.magic_index]
    strength = list(magic_data.values())[self.magic_index]['strength'] + self.stats['magic']
    cost = list(magic_data.values())[self.magic_index]['cost']
    self.create_magic(style,strength,cost)
```

Here are the codes for the magic input, it is pretty similar to the weapon input as just import the current magic data from the dictionary I create in settings and then print them.




```
Flame
9
20
```

The game is actually working and also the data of magic is printed correctly.

```
if keys[pygame.K_e] and self.can_switch_magic:
    self.can_switch_magic = False
    self.magic_switch_time = pygame.time.get_ticks()

    if self.magic_index < len(list(magic_data.keys())) - 1:
        self.magic_index += 1
    else:
        self.magic_index = 0

    self.magic = list(magic_data.keys())[self.magic_index]
```

I also create the input of switching magic, which it is also very similar to switch weapon and I just give it a key of ‘e’.

Step 3: I need to then renew the UI of magic in the ui section. It is still pretty similar to the weapon.

```
self.magic_graphics = []
for magic in magic_data.values():
    magic = pygame.image.load(magic['graphic']).convert_alpha()
    self.magic_graphics.append(magic)
```

```
def magic_overlay(self,magic_index,has_switched):
    bg_rect = self.selection_box(80,635,has_switched)
    magic_surf = self.magic_graphics[magic_index]
    magic_rect = magic_surf.get_rect(center = bg_rect.center)

    self.display_surface.blit(magic_surf,magic_rect)
```

```
self.magic_overlay(player.magic_index,not player.can_switch_magic)
```

The game is running and the UI is shown correctly. Then the magic stage is finished.

Stage 10: Creating the enemies

Step 1: The data of different enemies need to be imported in the settings.

```
monster_data = {
    'squid': {'health': 100,'exp':100,'damage':20,'attack_type': 'slash', 'attack_sound':'C:\CS\audio\attack\slash.wav', 'speed': 
    'raccoon': {'health': 300,'exp':250,'damage':40,'attack_type': 'claw',  'attack_sound':'C:\CS\audio\attack\slash.wav','speed':
    'spirit': {'health': 100,'exp':110,'damage':8,'attack_type': 'thunder', 'attack_sound':'C:\CS\audio\attack\slash.wav', 'speed'
    'bamboo': {'health': 70,'exp':120,'damage':6,'attack_type': 'leaf_attack', 'attack_sound':'C:\CS\audio\attack\slash.wav', 'spee
    }
```

This is the information about all the monsters that I create, they have their own attack method and different sound.

Step 2: I should then create a new group called ‘Entity’ to stand for all the sprites that can attend interactions, such as the player and the enemies.

```
class 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
```

This class only needs move and collision now as I have not implemented the interactions yet. As a result, just keep it simple now.

Step 3: Then all the preparations for creating the enemies have been finished, it's time to start working out them. Firstly, I create a new section called 'enemies' which all their methods and attributes will be included there.

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

```
class Enemy(Entity):
    def __init__(self,monster_name,pos,groups,obstacle_sprites):

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

Here are the basic imports and definition of Enemy. The sprite type is set as 'enemy', because the enemy will have health and interaction with the player, so they can not be 'object', they need to be a new type.

```
self.import_graphics(monster_name)
self.status = 'idle'
self.image = self.animations[self.status][self.frame_index]
```

```
    self.rect = self.image.get_rect(topleft = pos)
    self.hitbox = self.rect.inflate(0,-10)
    self.obstacle_sprites = obstacle_sprites
```

Then I create the basic graphics and movement for them.

```
self.monster_name = monster_name
monster_info = monster_data[self.monster_name]
self.health = monster_info['health']
self.exp = monster_info['exp']
self.speed = monster_info['speed']
self.attack_damage = monster_info['damage']
self.resistance = monster_info['resistance']
self.attack_radius = monster_info['attack_radius']
self.notice_radius = monster_info['notice_radius']
self.attack_type = monster_info['attack_type']
```

```
    self.can_attack = True
    self.attack_time = None
    self.attack_cooldown = 400
```

There are lots of statements in their stats as there will be many interactions with the player, they do require such many of stats to behave differently for various interactions.

```
def import_graphics(self,name):
    self.animations = {'idle':[],'move':[],'attack':[]}
    main_path = f'C:\CS\graphics\monsters\{name}'
    for animation in self.animations.keys():
        self.animations[animation] = import_folder(main_path + '/' + animation)
```

This function is pretty similar to the image loading of the player, it is used to import the images of the monsters.

```
def get_status(self, player):
    distance = self.get_player_distance_direction(player)[0]

    if distance <= self.attack_radius and self.can_attack:
        if self.status != 'attack':
            self.frame_index = 0
        self.status = 'attack'
    elif distance <= self.notice_radius:
        self.status = 'move'
    else:
        self.status = 'idle'
```

```
def actions(self,player):
    if self.status == 'attack':
        self.attack_time = pygame.time.get_ticks()
        print('attack')
    elif self.status == 'move':
        self.direction = self.get_player_distance_direction(player)[1]
    else:
        self.direction = pygame.math.Vector2()
```

These two functions define a dangerous distance for monsters and also their behaviours for different distances.

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

    self.frame_index += self.animation_speed
    if self.frame_index >= len(animation):
        if self.status == 'attack':
            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)
```

This achieves the animation of the monsters by frequently switching images, which is very close to the animation of the player.

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

The enemies also need cooldown. Otherwise, the player will always be attacked which the game can not go on successfully.

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

def enemy_update(self,player):
    self.get_status(player)
    self.actions(player)
```

Finally, all the functions are updated and the player interactions are also updated. In this steo it is pretty hard to test as the enemies are drawn on the map, those drawing should be down in the level section, so I will pass to the next step and then test together.

Step 4: I will add the entity csv file data into the layouts and print all the entities (the player and monsters).

```
layouts = {
    'boundary': import_csv_layout('C:\CS\map\map_FloorBlocks.csv'),
    'grass': import_csv_layout('C:\CS\map\map_Grass.csv'),
    'object': import_csv_layout('C:\CS\map\map_Objects.csv'),
    'entities': import_csv_layout('C:\CS\map\map_Entities.csv')
}
```

```
if style == 'entities':
    if col == '394':
        self.player = Player(
            (x,y),
            [self.visible_sprites],
            self.obstacle_sprites,
            self.create_attack,
            self.destroy_attack,
            self.create_magic)
    else:
        if col == '390': monster_name = 'bamboo'
        elif col == '391': monster_name = 'spirit'
        elif col == '392': monster_name ='raccoon'
        else: monster_name = 'squid'
        Enemy(monster_name,(x,y),[self.visible_sprites],self.obstacle_sprites)
```

Then the csv file data has been loaded, the only thing left is to update the monsters at the places that the csv file gives as monsters.

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

```
def run(self):
    self.visible_sprites.custom_draw(self.player)
    self.visible_sprites.update()
    self.visible_sprites.enemy_update(self.player)
    self.ui.display(self.player)
```

The function enemy update has been created and after running all the enemies are at their right positions. They will also attack the player after getting too close, which means this stage is finished.

Stage 11: Player & enemy interactions

Step 1: I would like to start with player damaging the enemy. To achieve this, I need to check if the weapon/magic collides with the enemy. For this purpose, two more groups need to be created, I call them 'attack sprites' and 'attackable sprites'.

```
self.attack_sprites = pygame.sprite.Group()
self.attackable_sprites = pygame.sprite.Group()
```

The two groups are created. The idea of checking if there is attack achieved is just to see if there is collision between these two groups.

Now I just want to start with player to enemy interaction, so in this case the enemy will act as attackable sprites and the weapon will be attack sprites.

```
Enemy(monster_name,(x,y),[self.visible_sprites,self.attackable_sprites],self.obstacle_sprites)
```

```
def create_attack(self):
    self.current_attack = Weapon(self.player,[self.visible_sprites,self.attack_sprites])
```

```
if style == 'grass':
    random_grass_image = choice(graphics['grass'])
    Tile((x,y),[self.visible_sprites,self.obstacle_sprites,self.attackable_sprites],'grass',random_grass_image)
```

Now both the monsters and the weapon are in their right groups, it is time to build up the interaction.

```
def player_attack_logic(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 == 'grass':
                        target_sprite.kill()
                    else:
                        target_sprite.get_damage(self.player,attack_sprite.sprite_type)
```

This is the interaction logic between the player and the enemy. If there is no collision, then nothing happens; if there is collision happening, then it will consider the type of sprites. When it's grass, just destroy it; when it's a monster, give a damage.

```
def get_damage(self,player,attack_type):
    if self.vulnerable:
        self.direction = self.get_player_distance_direction(player)[1]
        if attack_type == 'weapon':
            self.health -= player.get_full_weapon_damage()
        else:
            pass

        self.hit_time = pygame.time.get_ticks()
        self.vulnerable = False
```

```
def get_full_weapon_damage(self):
    base_damage = self.stats['attack']
    weapon_damage = weapon_data[self.weapon]['damage']
    return base_damage + weapon_damage
```

The first function is used to decrease health after interaction happens, and the second is used to import the damage of the current weapon and add it to the basic damage of the player.

```
def check_death(self):
    if self.health <= 0:
        self.kill()
```

After decreasing the health, if the health is lower than 0, then the monster should be 'killed' which it should display some death animation, but now I'm just focus on the interaction so I just consider it as been killed, but no further action.

After running it, although it is not yet implemented in the level section, at least it is not going wrong currently.

```
    self.vulnerable = True
    self.hit_time = None
    self.invincibility_duration = 300
```

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

I just forget this which this is used to limit the damaging times. If without this, for every second the monster will be damaged by 60 times as it is the FPS of the game.

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

When the monsters are getting damaged but not dead, they should also have some kinds of behaviours to show that they are damaged, which is just this function hit reaction.

Step 2: Now I need to create enemy to player interaction, it is pretty similar but the difference is that the damage of the enemy should be counted by the attack type.

```
def damage_player(self,amount,attack_type):
    if self.player.vulnerable:
        self.player.health -= amount
        self.player.vulnerable = False
        self.player.hurt_time = pygame.time.get_ticks()
```

```
    self.vulnerable = True
    self.hurt_time = None
    self.invulnerability_duration = 500
```

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

The system is almost the same but just with player as an attackable sprite and the monsters will be attack sprites. However, as the target of the monsters are all the player, nothing else, so I create a function called damage player, which will simplify this problem a little bit.

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

This will be the animation of the interaction of the player.

My full project file:

[Rshen23/Programming-project: Two-year project of programming in Computer Science (github.com)]

Testing

Index	Name of test	Purpose	Result (See the video testing)
1	Initializing	Check if the game initializes without any errors.	The game does not crashes.
2	Pygame check	Ensure that pygame initializes successfully.	~(this means same above)
3	Game window	Verify that the game window is created with the correct dimensions.	~
4	Title	Confirm that the game caption is set to ‘Monster Hunter: Cartoon’	~
5	Clock	Test if the clock object is created.	~
6	Sound	Check if the main sound loads and plays correctly.	The background music in the video testing shows this.
7	Initial volume of sound	Test if the volume of the main sound is set to 0.5	~
8	Sound loop	Verify that the main sound loops continuously	~
9	QUIT	Simulate a QUIT event and ensure that the game quits without errors.	14:00 tail the end
10	Keydown	Simulate a KEYDOWN event with the 'm' key and verify if the toggle_menu method of the Level object is called.	All across the video
11	Game loop	Ensure that the game loop runs continuously.	The game does not crash.
12	Screen filled	Check if the screen is filled with the correct color defined in ‘WATER_COLOR’	~

13	Run method	Verify that the run method of the Level object is called within the game loop.	~
14	Display	Test if the display updates correctly.	~
15	FPS check	Check if the frame rate is limited to the specified value ('FPS')	It is 60 FPS.
16	Functionality	Test the overall functionality by running the game and checking for any unexpected behavior.	All across the video
17	User	Interact with the game manually to ensure that it responds correctly to user input and events.	~
18	Error Handling	Test how the game handles unexpected inputs or events, such as invalid key presses or window resizing.	~
19	Source	Verify that all required resources (such as audio files) are loaded correctly.	~
20	Missing source	Test what happens if a resource is missing or cannot be loaded	~
21	Overall performance	Evaluate the performance of the game, including frame rate and resource usage, to ensure smooth gameplay.	~
22	Variable check	Check if the WIDTH and HEIGHT constants have the correct values. Ensure that TILESIZE, BAR_HEIGHT, HEALTH_BAR_WIDTH, ENERGY_BAR_WIDTH, and ITEM_BOX_SIZE are set to appropriate values.	At the start, it is a 1600:900 screen, which is enough

24	Colour	Verify that all color constants are defined correctly and in the correct format (e.g., hexadecimal or named colors).	All across the video, there's no colour lost.
25	Colour suited	Check if colors are visually distinguishable and suitable for their respective purposes (e.g., UI colors, health and energy bar colors)	~
26	UI font	Check if the path to the UI font file (UI_FONT) is correct.	6:12 ~ 6:33
27	UI elements	Ensure that the font size (UI_FONT_SIZE) is appropriate for the UI elements.	~
28	Data	Ensure that the weapon_data, magic_data, and monster_data dictionaries contain the expected keys and values.	All across the video the weapons are normally used, meaning that it does not have any error.
29	Data entries	Test if weapon, magic, and monster data entries have all required attributes (e.g., cooldown, damage, graphic path).	~
30	File path	Verify that all file paths for graphics, audio, and other resources are correct and accessible.	~
31	Missing path	Check if missing or incorrect file paths are handled gracefully in the code.	~
32	Hitbox	Check if HITBOX_OFFSET values are appropriate for their corresponding entities	2:33 ~ 2:57
32	Cool down logic	Test the logic related to weapon cooldowns, damage calculations,	11:01

		magic strength, enemy health, etc., using sample data from the dictionaries.	
33	Calculation	Verify that all calculations and logic related to the game entities are functioning as expected.	~
34	Level initialization	Verify that the Level object initializes without any errors.	All across the video
35	groups	Ensure that all necessary attributes and sprite groups are initialized correctly.	~
36	Map creation	Test if the map creation logic (create_map method) generates the map with the correct tiles and entities.	00:27
37	Objects on map	Check if different types of tiles (grass, objects) are created correctly with their respective graphics.	~
38	Create attack	Test if the create_attack method creates a new weapon sprite correctly.	1:11
39	Remove weapon	Verify that the destroy_attack method removes the current weapon sprite without errors.	~
40	Interaction(attack)	Test the player attack logic (player_attack_logic method) to ensure that attack sprites interact correctly with attackable sprites.	~
41	Damage	Verify that attack sprites cause damage to attackable sprites and trigger appropriate effects.	~
42	Player being damaged	Test the damage_player method to ensure that player health decreases correctly when taking damage from enemies.	~

43	Invulnerable after damage	Check if the player becomes invulnerable for a certain duration after taking damage.	~
44	Death particles	Verify that the trigger_death_particles method creates death particles at the correct position and with the correct type.	3:14
45	Create particles	Test if the create_particles method creates particles with the correct type and at the correct position.	~
46	Experience point	Test the add_exp method to ensure that player experience points increase correctly when enemies are defeated.	~
47	Menu	Test the toggle_menu method to ensure that the game pause state toggles correctly when the menu is opened or closed.	10:16
48	Camera group	Test if the YSortCameraGroup class functions correctly in rendering sprites with correct depth sorting.	2:33
49	Following camera	Verify that sprites are drawn relative to the player's position and the camera offset.	~
50	Update on enemy	Test if the enemy_update method updates enemy sprites correctly, considering the player's position and behavior.	9:52
51	Player initialization	Check if the player object initializes without any errors	All across the video
52	Player attributes	Verify that all necessary attributes are initialized correctly.	~

53	Player image	Test if the player image is loaded correctly.	~
54	Transparency	Ensure that the image has the correct transparency settings (convert_alpha).	~
55	Input keys	Test various key inputs (movement, attack, switch weapon/magic) to ensure they are correctly detected.	0:52
56	Direction and status	Check if the player's direction and status are updated based on the input.	0:52
57	Attack of player	Test if the create_attack method is called when the attack input key is pressed.	1:11
58	Magic of player	Test if the create_magic method is called when the magic input key is pressed.	1:32
59	Switch weapon/magic	Test the switching of weapons and magic to ensure it cycles through the available options correctly.	6:12
60	Switch cooldown	Verify that the cooldown between switching is enforced correctly.	~
61	Damage calculation	Test the get_full_weapon_damage and get_full_magic_damage methods to ensure they calculate damage correctly.	All across the video
62	Calculation base	Verify that the calculated damage includes both base and weapon/magic damage.	~
63	Status update	Test if the player's status (idle, attack) updates correctly based	4:18

		on movement and attacking states.	
64	Animation	Test if player animations are displayed correctly based on the player's status and direction.	2:33
65	Animation loop	Ensure that animations loop correctly and smoothly.	~
66	Player stats	Test if player stats (health, energy) are updated correctly over time (e.g., energy recovery).	~
67	Stats limit	Verify that stats are capped at their maximum values.	~
68	Upgrade cost	Test if the upgrade costs for player stats are retrieved correctly from the upgrade_cost dictionary.	9:52
69	Enemy initialization	Verify that the enemy object initializes without any errors.	All across the video
70	Enemy attributes	Ensure that all necessary attributes are initialized correctly, including graphics, stats, sounds, and interaction parameters.	~
71	Image	Test if the enemy graphics are imported correctly from the specified folder based on the monster name.	~
72	Animation dictionary	Ensure that the animations dictionary contains the correct paths to the animation frames.	10:03
73	Distance and direction	Test the get_player_distance_direction method to ensure it calculates the distance and direction between the enemy and the player correctly.	~

74	Status	Test the get_status method to ensure it correctly determines the status (idle, move, attack) of the enemy based on the player's distance.	~
75	Action	Verify that the actions method executes the correct action (attack or move) based on the enemy's status.	~
76	Correct action based on range	Test if the enemy attacks the player when in attack range and moves towards the player when in notice range.	~
77	Animation changes	Test if the enemy animation changes correctly based on its status (idle, move, attack).	7:09
78	Animation loop	Ensure that animation loops correctly and smoothly.	~
79	Monster cooldown	Test if the attack cooldown and invincibility duration are managed correctly.	11:01
80	Cooldown error	Verify that the enemy cannot attack or take damage during the cooldown periods.	~
81	Damage	Test the get_damage method to ensure it calculates the correct damage to the enemy based on the player's attack type.	All across the video
82	Been damaged	Ensure that the enemy's health decreases correctly when taking damage.	~
83	Death	Verify that the enemy is killed when its health reaches zero.	9:58
84	Death display	Test if death triggers the correct particle effects, experience points	~

		addition, and plays the death sound.	
85	Hit reaction	Test if the enemy's hit reaction (pushback) is applied correctly when taking damage.	3:00
86	Entity initialization	Verify that the Entity object initializes without any errors.	All across the video
87	Values	Ensure that the initial values of frame_index, animation_speed, and direction are set correctly.	~
88	Move update	Test the move method to ensure it correctly updates the position of the entity based on its direction and speed.	~
89	Obstacles	Verify that the entity moves smoothly in both horizontal and vertical directions without passing through obstacles.	~
90	Collision	Test the collision method to ensure it detects collisions with obstacle sprites correctly in both horizontal and vertical directions.	5:37
91	Reaction of collision	Verify that the entity's movement is stopped or adjusted appropriately when colliding with obstacles.	~
92	Wave value	Test the wave_value method to ensure it returns values between 0 and 255 based on the sine function of the current time.	All across the video
93	Oscillation	Verify that the returned value oscillates smoothly between 0 and 255 as time progresses.	~
94	Magic initialization	Verify that the MagicPlayer object initializes without any errors.	1:32

95	Sound of magic	Ensure that the sounds dictionary contains the correct sound objects for 'heal' and 'flame' spells.	~
96	Heal	Test the heal method to ensure it heals the player correctly when the player has enough energy.	1:44
97	HP increase	Verify that the player's health increases by the specified strength and energy decreases by the cost of the spell.	~
98	Sound of healing	Check if the 'heal' sound is played and if particles are created at the player's position.	~
99	Not enough energy for healing	Test the heal method when the player's energy is insufficient to cast the spell.	~
100	Flame	Test the flame method to ensure it creates flame particles in the correct direction and positions.	1:32
101	Cost of flame	Verify that the player's energy decreases by the cost of the spell.	~
102	Sound of flame	Check if the 'flame' sound is played and if flame particles are created in the correct positions.	~
103	Not enough cost for flame	Test the flame method when the player's energy is insufficient to cast the spell.	~
104	Flame failed to play	Verify that the player's energy remains unchanged, and no flame particles are created.	~
105	Weapon initialization	Verify that the Weapon object initializes without any errors.	1:11
106	Sprite type of weapon	Ensure that the sprite_type attribute is set correctly to 'weapon'.	~

107	Image of weapon	Check that the image attribute is loaded with the correct weapon graphic based on the player's direction.	~
108	Placement based on direction	Test the placement of the weapon sprite based on different player directions.	~
109	Position	For each direction ('right', 'left', 'down', 'up'), verify that the weapon sprite is positioned correctly relative to the player's rect.	0:53
110	Position relative to player	Ensure that the weapon sprite is placed at the appropriate position relative to the player's center point.	~
111	Edge cases	Test edge cases such as when the player's direction is not one of the expected values ('right', 'left', 'down', 'up').	3:44
112	Reaction	Verify that the weapon sprite placement handles unexpected player directions gracefully without causing errors or unexpected behavior.	All across the video
113	Loading graphics	Check that the loaded weapon graphic is displayed correctly on the screen when blitted.	All across the video
114	Upgrade initialization	Verify that the Upgrade object initializes without any errors.	~
115	Attributes for upgrade	Check that the required attributes and parameters are correctly set during initialization.	~
116	Item	Ensure that the Item objects are created and stored correctly within the Upgrade object.	~

117	Input key	Test keyboard input handling in the input method of the Upgrade class.	~
118	Selection index	Verify that the selection index changes correctly when pressing the left and right arrow keys.	~
119	Bounds of index	Check that the selection index does not go out of bounds.	~
120	Selection cooldown	Test the cooldown functionality in the selection_cooldown method of the Upgrade class.	~
121	Trigger	Test the trigger functionality in the trigger method of the Item class.	~
122	Selected attributes	Verify that the selected attribute of the player is correctly upgraded when triggering an item.	~
123	Experience reaction	Check that the player's experience and attribute values are updated correctly after triggering an item.	~
124	Display of Item	Test the rendering of items on the screen in the display method of the Item class.	~
125	Name, cost and attribute	Verify that the item names, costs, and attribute bars are displayed correctly.	~
126	Highlight	Check that the selected item is highlighted differently from others	~
127	UI initialization	Verify that the UI object initializes without any errors.	6:12
128	Health bar	Test the show_bar method to ensure that it correctly displays the player's health bar.	All across the video

129	Health width	Verify that the health bar width is adjusted based on the current health and maximum health values.	~
130	Colour	Check that the correct colors are used for the health bar and its border.	~
131	Energy bar	Similar to the health bar test, verify that the show_bar method correctly displays the player's energy bar.	~
132	Energy width	Ensure that the energy bar width is adjusted based on the current energy and maximum energy values.	~
133	Colour	Check that the correct colors are used for the energy bar and its border.	~
134	Experience	Test the show_exp method to ensure that it correctly displays the player's experience points.	~
135	Position of experience	Verify that the experience points are rendered at the correct position on the screen.	~
136	Weapon overlay display	Test the weapon_overlay method to ensure that it correctly displays the player's selected weapon.	1:11
137	Magic overlay display	Similar to the weapon overlay test, verify that the magic_overlay method correctly displays the player's selected magic.	1:32

Evaluation

For evaluation, there are numerous aspects that warrant discussion. Initially, I will delve into an analysis of the success and shortcomings concerning the professionalism and playability of my game, encapsulating these findings into a conclusive assessment. This examination will scrutinize various elements, including the game's user interface, overall polish, adherence to design principles, and the fluidity of gameplay mechanics. By dissecting these components, I aim to identify areas of strength and weakness, highlighting aspects that contribute positively to the player experience while pinpointing areas that may require refinement or enhancement. Following this evaluation, I will shift focus to the ongoing maintenance of my game. This involves addressing any existing bugs or technical issues, implementing necessary updates, and ensuring compatibility with evolving hardware and software environments. Additionally, I will explore potential avenues for future development, brainstorming innovative features, expansions, or adaptations that could further enrich the gaming experience. By proactively considering these possibilities, I aim to chart a roadmap for the continued growth and evolution of my game, fostering its longevity and relevance in an ever-changing landscape.

Professionality

Success:

1. The game is run successfully in the python without bugs available and the player's input could all be reacted. (See all the screen shoots the game is running without crash)
2. Monsters have their special design and they have their basic behaviour system which means the code of monsters are completed and working successfully. (See page 40 and 127, the game runs and the monsters are idling or moving on the map successfully)
3. The player has enough interaction with both monsters and other objects on the map, which means the interactions are implemented successfully. (Still see page 127, the monsters will track the player and attack, the player can also attack them)
4. The player's motion is fluent and there's no display issue happening, which means the animation of the player is built well. (See page 105 and 106, the animation is working and there are clear changes on the player's character over time)
5. The monsters' animation and random motion have all been built successfully, which they are pretty fluent and do not show any error in this field. (See page 127, the monsters will not stop at the same place, they will randomly move and will track the player with animation)
6. The distance system is built successfully which the monsters will automatically track the player after getting close. (See page 127, the monster will only track the player when the player enters a certain distance)
7. The UI system is built clearly and simple, but it is understandable and it does not have any display error. (See page 119 and 127)
8. The cooldown system of the game is performing well. (See the cooldown design in all the sprites including player attack, weapon, magic, damage to player and damage to monster)

9. The map is built successfully, which all the sprites in the csv files are in their right directions and there's no display error on the floor. (See all the screen shoots, the floor is printed in the correct way)

Failure:

1. The weapon and magic system is not very successful as it does not reach the optimised performance. The switching of weapon and magic is failed because of technique problems.
2. The particles of the game is not implemented because of lack of code learning.
3. Attacking sound of monsters are not implemented because of file issue.
4. UI of switching weapon and magic is not great, because of failure on the weapon/magic system.
5. The interaction between enemies and the player does not work on the health and energy.
6. The exp system is not completed.

Playability

Success:

1. The image of the map, the player and the monsters are beautiful and close to 2D cartoon style, which could provide great virtual experience.
2. The game world has four types of fields with different monsters on different fields, which could fill up the player's experience.

Failure:

1. It does not provide pause system, so that may cause bad experiences.
2. It does not provide a control interface to control the sound or brightness of the game.
3. It does not provide a full background story for the game, so the player does not know what to do in the game.
4. It does not provide a central goal, so the player does not have any target during their game experience.
5. It does not provide great virtual or acoustical effect, which the game might be a bit boring.

Usability:

1. Learnability:

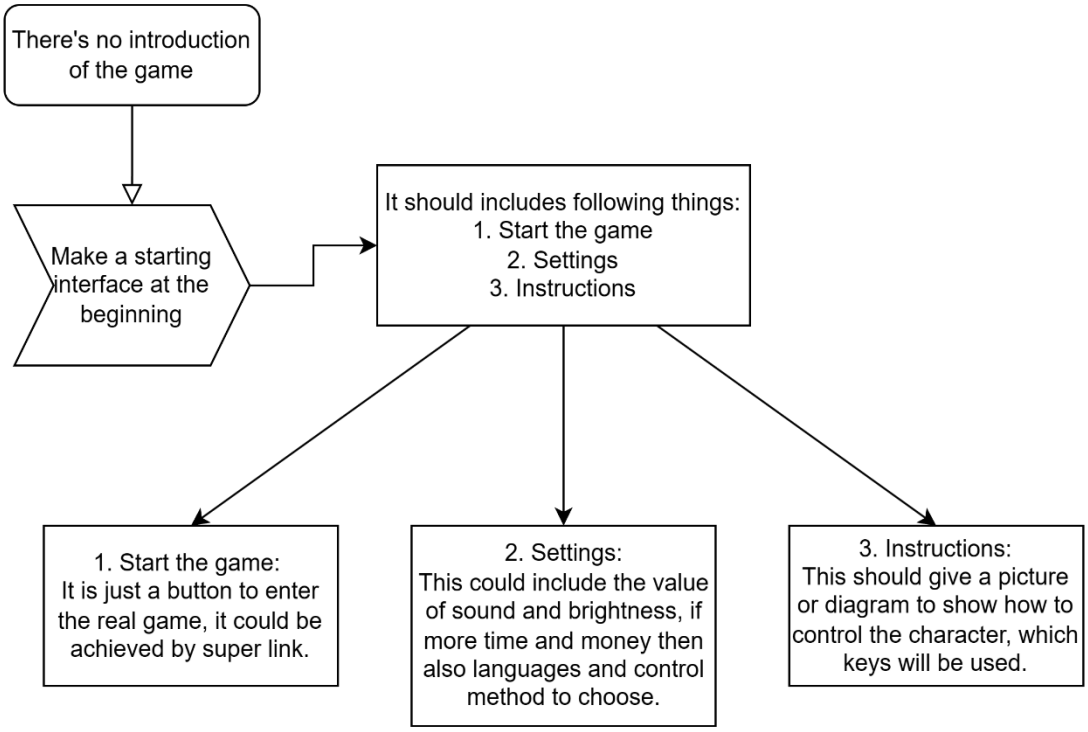
   Learnability partially shows the level of difficulty of the game, which it describes how hard it is for a new player to learn the game.

   In this section, the advantage of my game is that there are very few keys that the player will need to use, so it is pretty easy for them to learn to control.

   Also, there's no complex system in my game, just walking around and use the weapon to hunt the monsters, so the contents of my game are very simple.

   However, what I do badly is that there's no introduction for the player at the beginning of the game, so the player has to try to find the right keys on their own, which might cause very poor experiences.

   Here is the flow chart of how to create the user interface if time and money is enough to do so.



2. Responsiveness:

   It describes the speed and accuracy with which the game responds to player input.

   In my game, almost all the inputs will give out feedback very quickly as there are no complex ones.

   The only thing that I need to notice here is that I need to be careful about the future things. If there are some kinds of things such as dialog systems or purchasing systems in my game, then the speed of responding might become an issue.

3. Accessibility:

   It is the degree which the game is playable by people with varying abilities, including those with disabilities.

   In my game it is not shown, but because this is a game with fighting and actions, so I would try my best to build up a system for players with disabilities.

   The first thing I would implement would be eye tracker, as it could support players with issues on their arms or hands to play my game.

   The next thing I think off is a visual impairment model. For example, I would set a thing called heart bit system, which when the monsters are closer, the heart bit will be louder. Another would be a guide which has sound to direct the correct way to escape or reach the target. This will provide a choose for blind people to play my game.

4. Feedback

It shows how well the game provides feedback to players regarding their actions, outcomes, and game status.

Unfortunately, this part is not really shown in my game. It seems that my game does not have a clear aim of playing and sometimes players just do not know what to do next even after they have did a lot of things.

As a result, in the future I should try to set a clear target for the player to catch up and better to have a mission system.

In the Mission system, the player will be guided to the targeted place so that they know what to hunt and where to find it; rewards will also be provided if they hand in special mission items. As a result of this, the player would be more motivated to do the tasks and play the game, as they get more feedbacks from the process.

Maintenance

The game is designed to have both advantages and limitations on maintenance, which I will discuss both sides and try to give solutions for limitations.

Advantages:

1. The code is divided into different sections. For example, if I want to make changes on the enemies, I just go to the enemies section and add it to the level section, then the change can be implemented.
2. The documents used in the algorithm are all separated clearly in different folders, so it is pretty easy to build up the dictionaries in the settings section. Also, it is easy to add new data into the algorithm. It could be an advantage when new creatures are imported into
3. Classes are clear and understandable, some classes like Entity, and some groups like obstacle sprites or visible sprites are all pretty convenient for creating new creatures or characters in the game. Those methods and attributes are there to be inherited and they are very basic which is probably be useful.
4. Many of the variables are set in the settings section with understandable names, so it is easy to change their values, which is helpful for fixing unreasonable values given (e.g. too high starting health or too high monsters health). It is a large advantage when trying to improve players' experiences.

Limitations:

1. The game does not have a tutorial at the start, so it might be hard for the player to learn how to play this game. The best solution for this will be building up a ‘setting’ user interface, which could include a tutorial teaching the player the keys to use for different behaviours such as walking and attacking.
2. The game does not have any control interface on the sound or brightness, which may not provide a good experience to specific users. The solution for this is the same as the last problem, just build a ‘setting’ interface which includes all of these control settings inside.
3. The game is still using local files and the path names are also local addresses, which this may not work for people downloading it. The solution is to build up an external file which could work as a library for all the other sections. When the user downloads the file, they could directly import all the game data from that file so there will not be any kind of path or import issue.

Future development

1. **Gameplay Mechanics:** Tweaking and refining the core gameplay mechanics can significantly enhance the player experience. This could involve improving controls, balancing difficulty, adding new features, or fine-tuning existing ones based on player feedback.

2. **Graphics and Visuals:** Enhancing the visual quality of the game can make it more appealing and immersive. This might involve upgrading textures, implementing advanced lighting effects, optimizing performance for smoother framerates, or ensuring consistency in art style throughout the game.

3. **Audio and Sound Design:** Improving the audio elements of the game, including music, sound effects, and voice acting, can greatly enhance the atmosphere and immersion. This could involve composing new music tracks, refining sound effects, or integrating dynamic audio systems to respond to gameplay events.

4. **User Interface (UI) and User Experience (UX):** Streamlining the UI and improving overall UX can make the game more intuitive and enjoyable to play. This might include redesigning menus, optimizing HUD elements, enhancing accessibility options, and providing clear feedback to the player.

5. **Content Expansion:** Adding new content such as levels, characters, items, quests, or storylines can provide players with more to explore and experience within the game world. This keeps the game fresh and engaging, encouraging players to continue playing and returning for more.

6. **Bug Fixes and Optimization:** Addressing bugs, glitches, and performance issues is crucial for ensuring a smooth and stable gaming experience. This involves thorough testing and debugging to identify and fix any technical issues that may arise during gameplay.

7. **Community Engagement and Feedback:** Actively soliciting feedback from players and engaging with the game's community can provide valuable insights into areas that need improvement. Developers can use this feedback to prioritize updates and enhancements that align with player preferences and expectations.

8. **Narrative and Storytelling:** Enhancing the narrative elements of the game, including character development, dialogue, and plot twists, can deepen the player's emotional engagement and investment in the game world. This could involve hiring professional

writers, expanding lore, or implementing branching story paths to offer more player choice.

9. **Multiplayer and Social Features:** Improving multiplayer modes and adding social features such as leaderboards, achievements, or online matchmaking can extend the game's longevity and foster a sense of community among players.

10. **Accessibility and Inclusivity:** Making the game more accessible to a wider range of players, including those with disabilities, can greatly improve its overall appeal and inclusivity. This might involve adding customizable control options, subtitles, colorblind modes, or other features to accommodate diverse player needs.

Bibliography

www.pixilart.com

www.mapeditor.org

www.geeksforgeeks.org

www.pygame.org/wiki

www.youtube.com/watch?v=4-k6j9g5Hzk

www.hollowknight.com

www.stardewvalley.net

www.zeldaclassic.com