Ninja Platformer Game

By

Sam Cai

Contents

Analysis

Project Outline

For this project, I intend to use Python, together with a library called Pygame to create a platformer game. The game will feature a map where the player can navigate horizontally, and vertically. Players must dodge enemy attacks and traverse the map to defeat all enemies before moving on to the next level. The ultimate objective is to obtain the highest score possible, which is determined by an algorithm involving time used and the amount of additional items collected by the player.

This game will be entirely in 2D and use pixel art as its art style. The reason being that 2D pixel art offers a nostalgic and retro aesthetic that resonates with many gamers, especially those who grew up during the era of 8-bit and 16-bit consoles. Also on the technical side, it's simpler to program.

Below is a screenshot from a platformer game available on Steam. The image showcases a dynamic character set within an interactive map. The character is standing on solid ground which suggests there is a physics simulation. Beyond the core gameplay elements, the game has impressive visuals, using decorative backgrounds and assets like trees and fences. Aside from the static assets, there are real-time animations like falling leaves and flowing water.

Figure 1. Screenshot of a platformer game demonstrated by Youtuber DaFluffyPotato

Even though this is an amazing game idea, it is easier said than done. A game like this has a lot of different components, such as physics simulation. For instance, how do we know if the player should be standing on the ground or not? What about gravity and speed?

Developing a game with complex mechanics that allow for player interaction, character development, and game progression requires computational approaches. This involves creating algorithms to handle game mechanics such as character attributes, game physics, and AI-controlled NPCs (non-player characters). Also, games rely heavily on player input and interaction, which can be handled efficiently through computational methods. This includes user interfaces, and input devices such as mouses and keyboards that allow for real-time and responsive gameplay.

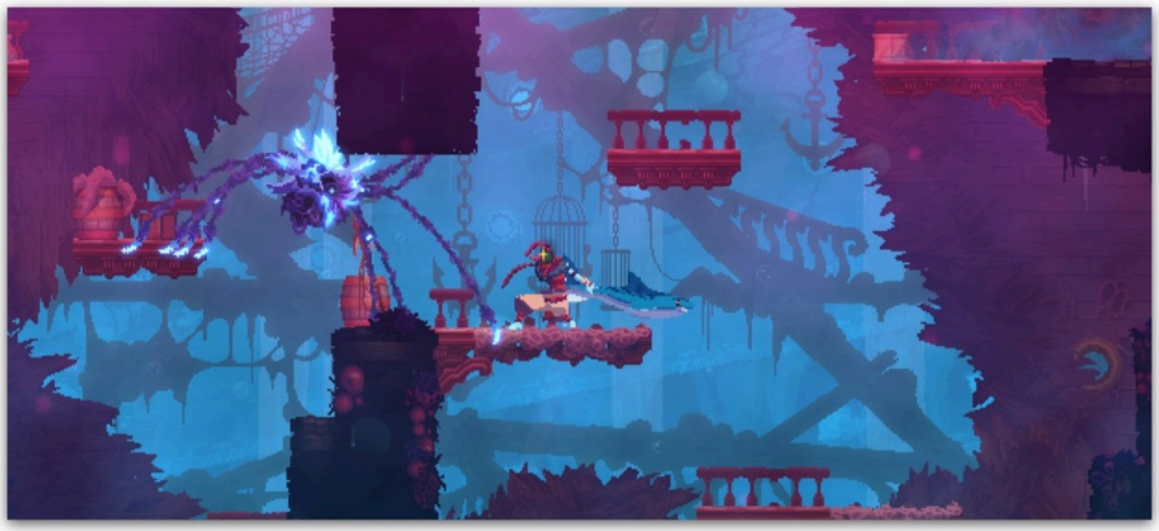


Figure 2.Screenshot from 'Dead Cells', a dungeon-themed open world game by Motion Twin, published in 2018

Aside from the pure technical aspects of this game, we also need to realize the significance of graphics and animation. Obviously, visuals are one of the most important aspects of a game, and humans naturally gravitate towards appealing aesthetics. Computational methods such as modeling, animation software, and graphic rendering engines allow us to create immersive and visually stunning game worlds. Existing rendering techniques such as shaders, global illumination and path tracing are great examples.

Lastly, which is also the most important step of game development, is testing and optimization. Things like simulating gameplay scenarios, testing performance on various hardware configurations, and optimizing game elements for better performance and user experience could overall improve the quality of the final product.

Creating a game like this is a difficult task as it involves a complex set of rules, mechanics, and interactions between the player, environment, and characters. Through the use of algorithms, data structures, and programming techniques, making the game becomes possible.

Throughout the design and development stage of my game, I will be constantly referencing other existing platformer games out there, like Dead Cells, Hollow Knight, and Super Mario Bros just to name a few, these games set the bar for this genre of game. By studying these, I hope to incorporate elements that will enhance the gameplay experience in my own creation, while also bringing in unique features that set my game apart.

Stakeholders

The game I'm developing is designed with accessibility in mind - it doesn't involve complex controls or too much genre-specific content. This means even someone with basic computer skills can enjoy the gaming experience easily. The primary aim is to cater to a broad audience, making the general public our target players. Basically, this game is for everyone with an interest in computer games. Recognizing the diverse preferences of various players, I am planning to integrate a customizable system into the gameplay - as people have unique preferences, especially when it comes to elements like character design or art styles. Players can modify aspects ranging from the aesthetic, like the level's color palette and character appearances, to more functional components. This approach not only provides a sense of ownership to the player but also ensures that the game feels personal and tailored to each individual's taste.

I have many friends who enjoy playing computer games in their spare time, including my roommate Daniel Dun, who is an avid gamer and enjoys games of various genres. He has collected over 200 games in his Steam library and clearly, he is a better gamer than I am. Therefore, I will seek his assistance when designing this game, as he has developed a strong sense of what makes a good game in terms of aesthetics and playability. I will conduct an interview and collect his opinion, and I think this will be very useful when finalizing the specifications of the game. This ensures that my game will be comparable to all the other good games out there.

Software Choice

I've chosen to work with Pygame as the foundational platform for this project, rather than other popular game engines like Unity or Unreal Engine. While these engines could offer much better graphics and intricate game mechanics, they come pre-equipped with ‘too-many’ features. Elements like character control, camera dynamics, and player interactivity (including mouse, keyboard, and UI controls) are already built-in. This provides plenty of room for game design but limits hands-on coding experiences.

My reason for choosing Python as the programming language is because this is the language I have a deep understanding of, this is the first programming language I have learnt and it uses minimal syntax, unlike the family of C languages, which requires frequent use of semicolons and curly brackets, types are optional in python so I don't have to declare them separately, having to write less code means it allows me to spend more time coding my ideas instead of trying to fix the syntax of the code.

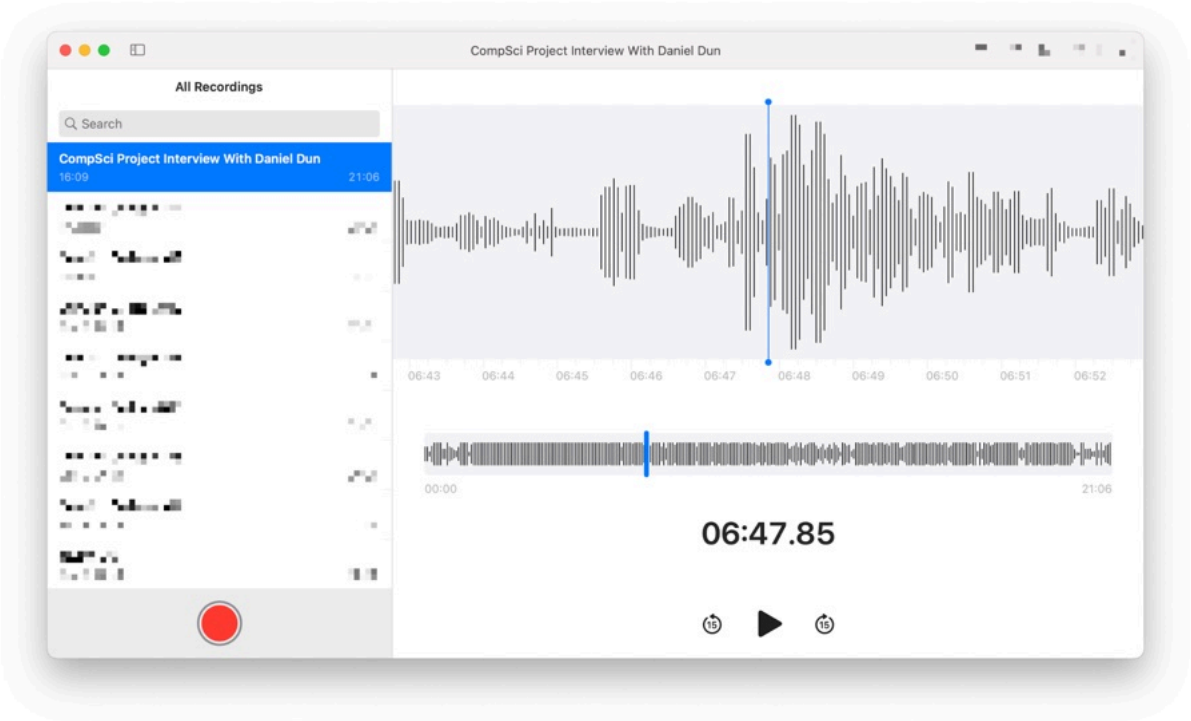
The Python interpreter is available on multiple platforms, including Windows PCs, Apple Macs, Linux, and even Android phones. It supports various processor architectures such as x86 and arm, making it suitable to run on a wide range of devices. This means that once the game is finished, it would be easy to distribute a copy to end users. They would only need to follow a few simple steps to install and run the game.

Performance-wise, because Python is an interpreted language, it is inherently slower than compiled languages like Rust, C++, or Java. but it shouldn't be a huge issue when running a simple game, the game should ideally run at 60FPS (frames per second) so as long as the game can finish a round of calculation within 1/60 or 0.0167 seconds, the gameplay should remain smooth and free from stutters.

The game will be developed using a game library, Pygame, which provides tools and libraries for game development based on the Python programming language. I will use a separate Python virtual environment for development, for reasons which I will explain later on.

The development machine uses an ARM architecture CPU running on MacOS, and I have two more additional test machines running on x86 Windows as well as Ubuntu Linux.

Interview



I conducted an in-depth interview with my friend Daniel and in this 20-minute conversation we discussed various things about my game. Snippets of our conversation transcripts are provided below.

Colour-Coded:
(Me): Red
(Daniel Dun): Blue
(INTRO SECTION)

Me: Hey Daniel, so you know what I've been making a game in pygame right? I'm kinda stuck right now so can you give me some recommendations or advice on what to do next? Also, I'm going to ask you a bunch of questions about game design and player controls because I know you've played lots of games in the past and you are an expert on this. This is gonna be very valuable information in later stages and here I have prepared 20 questions for you.

(Daniel): Oh ok sure.
(Me): The first section is on Game Design and narrative.

(Q1): What themes or settings captivate you most in platform games (e.g., futuristic, medieval, jungle, urban)?

(A1): Well I've been playing the game Dead Cells for quite some time now and it's my no.1 favorite platformer game. I mean sure you can classify its style as 'medieval' and magic

fantasy-like, it's got tons of ancient castles as the map and uses potions a lot. So I guess it's my favourite theme.

(Q2): Which type of challenges or obstacles are the most enjoyable for you in platformers (e.g., puzzles, enemies, platforming sequences)?
(A2): Definitely the enemies, I don't like puzzles because it makes me lose brain cells and I am not a fan of platforming sequences cuz I just don't like jumping around randomly. I feel like with enemies you can do a lot, like the types of combats, the skills and special weapons you can use, the possibility is literally unlimited on this one and I feel like you definitely can add lots of different types of enemies in the game.

(Q3): How would you like to be rewarded as you progress in a game (e.g., power-ups, new abilities, story reveals)?
(A3): Well the most obvious one is that you can get more scores, but new abilities are also nice, for example you can jump higher or deal more damage to the enemies, it helps you when the levels are getting increasingly harder.

(Q4): Moving on to the technical side of things. How do you feel about the controls? Do you prefer them to be simple or more complex? Arrow Keys or WASD?
(A4): Personally, I like controls that start simple but have depth. For instance, initially, you'd just jump and move, but as you progress, you unlock more abilities for the character. If you wanna add more abilities in the future I think it's better to keep the movement keys as arrow keys as WASD gets in the way of other ability keys.

(Q5): Got it. And what do you say about on-screen information? Minimal UI or a detailed stats display?
(A5): I lean towards a minimal UI during gameplay, you can have a few but not too many things on the main UI, the most important ones like player HP or remaining enemy count but perhaps a separate menu or pause screen where players can deep dive into stats if they wish.

(Q6): How forgiving should the game be? Like, how many hits or mistakes before you lose or have to restart?
(A6): A moderate difficulty is ideal. Maybe 2-3 hits? But don't make it too easy though.

(Q7): Speaking of pacing, how do you feel about save points or checkpoints?
(A7): Regular checkpoints are good, especially in challenging sections. It can be frustrating to redo long portions of a game so maybe you can save the game whenever you move onto the next level.

(Q8): Camera perspective. Static, following, or dynamic?
(A8): I prefer a following camera for platformers. It feels more immersive.

(Q9): Thoughts on difficulty levels?
(A9): Multiple difficulty options are great. Maybe change enemy patterns, fewer checkpoints, or add time limits for added challenges.

(Q10): Should I include tutorials or hints?
(A10): Yes, definitely, because I have played games in the past that don't include a tutorial level and the game just throws me into the main level and I had no idea what's going on. At least do a controls tutorial.

(Q11): Any particular power-ups or abilities you'd love to see in a game?
(A11): Maybe something that manipulates time or gravity? It can introduce some cool mechanics.

(Q12): Let's talk visuals. Any art styles you're particularly fond of?
(A12): I love pixel art, it's the most common among platformer games, and I just love the retro feeling of pixel arts.

(Q13): Music and sound effects?
(A13): Engaging background music is crucial. Retro chiptunes or atmospheric tracks would be great. Sound effects should be crisp and distinct and give player feedback. For example the SFX needs to be loud enough so that the player knows that it's got hit.

(Q14): How do you feel about the structure of levels? Linear or explorative?
(A14): Just keep it simple, a linear level structure would be enough.

(Q15): Pace of the game? Fast action or slow exploration?
(A15): Come on, it's a platformer not an open-world RPG, it needs to be fast

(Q16): Multiplayer options?
(A16): Single-player narratives are great, but a co-op mode can be a fun addition. Be aware of the internet lag that happens in multiplayer modes though, it can really ruin the experience so do think about it.

(Q17): Preferred color palettes or visual styles And animations?
(A17): I prefer darker, moody palettes. But it should align with your game's theme.Fluid animations make a game feel polished, especially character reactions or ambient animations.

(Q18): Are there platformers that, in your opinion, are benchmarks in the genre?
(A18): Besides Dead Cells, classics like Super Mario or even indie gems like Hollow Knight.

(Q19): Optimal gaming session length?
(A19): 5 minutes per level is plenty, players can get quite tired physically if a level is too long.

(Q20): PvP competitions or leaderboards?
(A20): yes, because I am very competitive in games. I play a rhythm-game called Arcaea competitively for three hours a day, just to be better in this game than all of my friends, and that is the only motivation for me to play this game, so yes. I need to feel good about this.

(Q21): Easter eggs or secrets?
(A21): Absolutely! One of my favourite.

Specify the proposed solution

So, after doing some preliminary research work and conducting the interview, I have consolidated the points and we now have a proposed solution to develop a game that immerses players in a detailed and interactive game world.

I have included measurable success criteria for the proposed solution below, as well as the required hardware and software configuration.

Requirement specification (success criteria)

Hardware requirements:

- A modern processor (no requirement for multi-core CPU as pygame doesn't natively support multi-threading)
- Sufficient RAM to support the game's graphics and processing requirements.
- Input devices such as a keyboard (for character movement and control)
- A display with a resolution higher than (640 x 360px)
- No requirement for a dedicated/integrated GPU as everything runs on the CPU (including rendering)
- Approx. 200MB of free space on storage devices for the pygame library, the Python interpreter and the game data files.

Software requirements:

- Python Interpreter with version 3.0+ (as syntax incompatible is with 2.x interpreter)
- Any ARM/x86 operating system distribution(Windows/MacOS/Linux)
- up-to-date pygame or pygame-ce library
- the PIL (Python Imaging Library) for handling images

A. Game Mechanics:

1.1. Input Requirements:

1.1.1. The player can use arrow keys to move around the character

1.1.2. The character shall be able to **dash through enemies** to defeat them

1.1.3. The character shall be able to **jump to avoid obstacles** and reach higher platforms.

1.1.4. Customizable Controls: The game offers a **flexible control scheme**, allowing players to adjust keybindings for optimal comfort and responsiveness.

1.2. The In-Game Menu is accessed by pressing ESC in the game.

1.3. The Player should not go through walls, but through the decoration items.

2. Graphics Requirements:

2.1. Player sprite animates in the direction of the player movement.

2.2. Responsive and intuitive UI that is **easy to navigate.**

2.3. The game should maintain a **consistent art style and level design**

3. Game's technical requirements

3.1. Debug Menu: A **handy debug menu** is available for troubleshooting and game insights, aiding in smooth gameplay.

3.2. The game should be running at **60FPS+** and the controls need to be **responsive**

3.3. Bug-Free: Efforts should be made to ensure that there are **no game-breaking bugs** or glitches. The game **should not exit** unless the player wants to.

3.4. Load Times: Game levels and assets should load swiftly, keeping **wait times minimal** to maintain player immersion.

3.5. Optimization: **Efficient use of resources** to ensure that the game doesn't overheat devices or consume **excessive battery** (for mobile games) or system resources (for PC/console games).

3.6. Save Mechanism: A **reliable save system** that ensures players don't lose their progress. **Auto-save features** can also be beneficial.

3.7. **Controller support** (Xbox/Dualshock controller).

4. Game Design:

4.1. The dash has a cool-down duration, which means the player has to wait for a certain time to use the dash again.

4.2. The player is given **3 lives at the start**, and the game concludes when these lives are depleted.

4.3. The player dies when they falls into the void

4.4. Level progression when all enemies are defeated by dashing

4.5. Player health taking damage from enemies when it has been hit.

4.6. Personalized Identity: Players can **input their name** for a personalized touch in leaderboards and/or multiplayer sessions.

4.7. A **pause, resume, and menu system** so players can navigate and take breaks..

4.8. **Sound effects and music** that match the game's theme and enhance the gameplay experience.

4.9. The difficulty of levels **gradually gets harder** as the player progresses (e.g., tougher enemies, intricate map designs, or traps that require special skills to bypass).

4.10. Incorporate a **tutorial level to familiarize** players with the game mechanics and dashing techniques.

4.11. The game shall have a **high score system** allowing players to **compete** with each other.

4.12. User-Created Levels: Players can use a **built-in editor to design and share their own levels**, promoting creativity and community engagement.

4.13. In-Game Player Statistics: Access key performance metrics, such as hits taken or enemies defeated, through the **in-game menu to track and improve gameplay**.

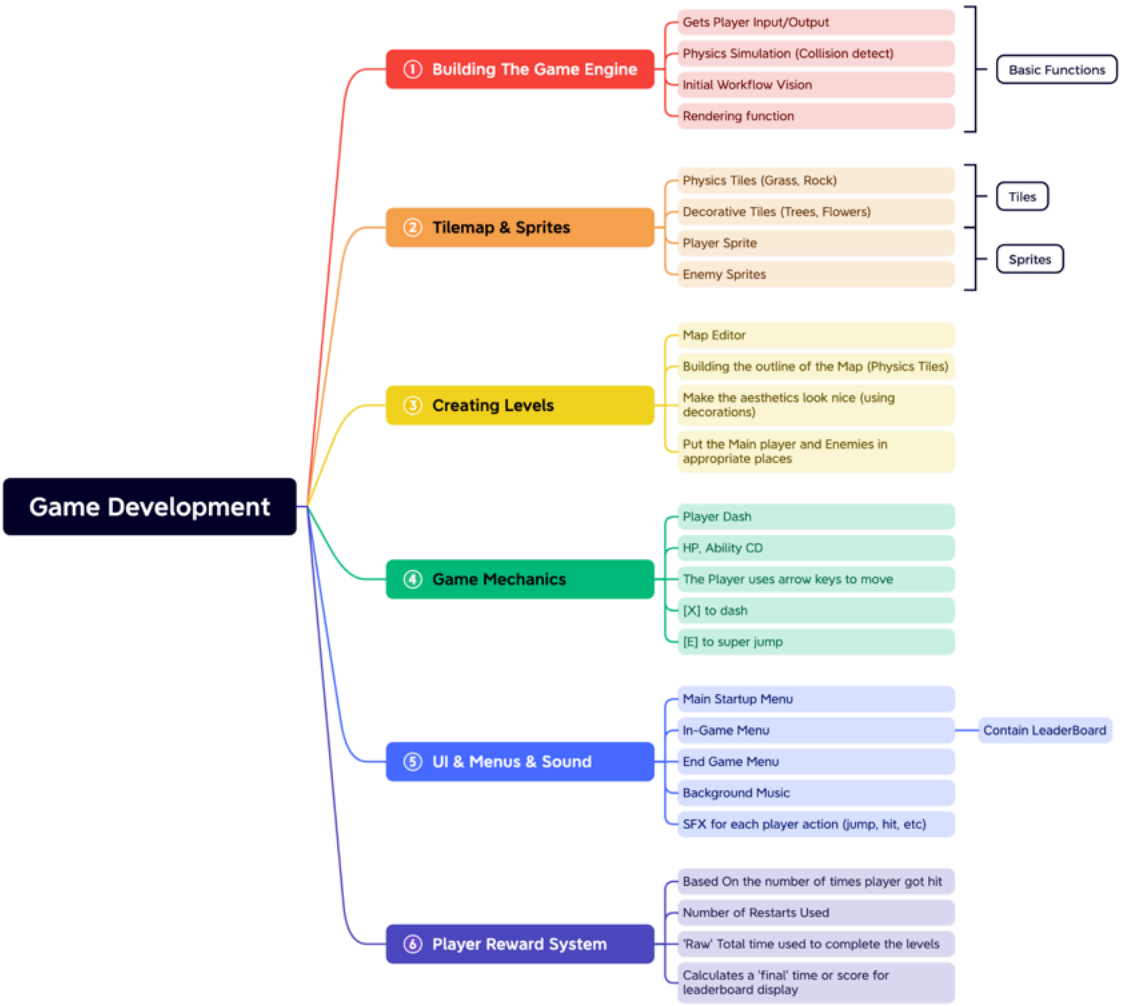
4.14. After the player beats the game, an information panel is shown at the end with all the information (time taken, hits taken, restart count)

Limitations of the proposed solution

The proposed solution does come with its limitations, whether it's the game design or the tech aspects of my game. In the last section of my success criteria specification, I have included a number of optional requirements. These are some changes that could be made, but it is very challenging under the development time constraints. There are always some features that a particular user won't like, so it makes it very challenging to design a game that appeals to absolutely everyone. But the requirement should satisfy 99% of the users.

Design

Before we start any actual coding, it is important to lay out a detailed plan for what is there to be done. Below is a mindmap showing the most important components of my game development project.

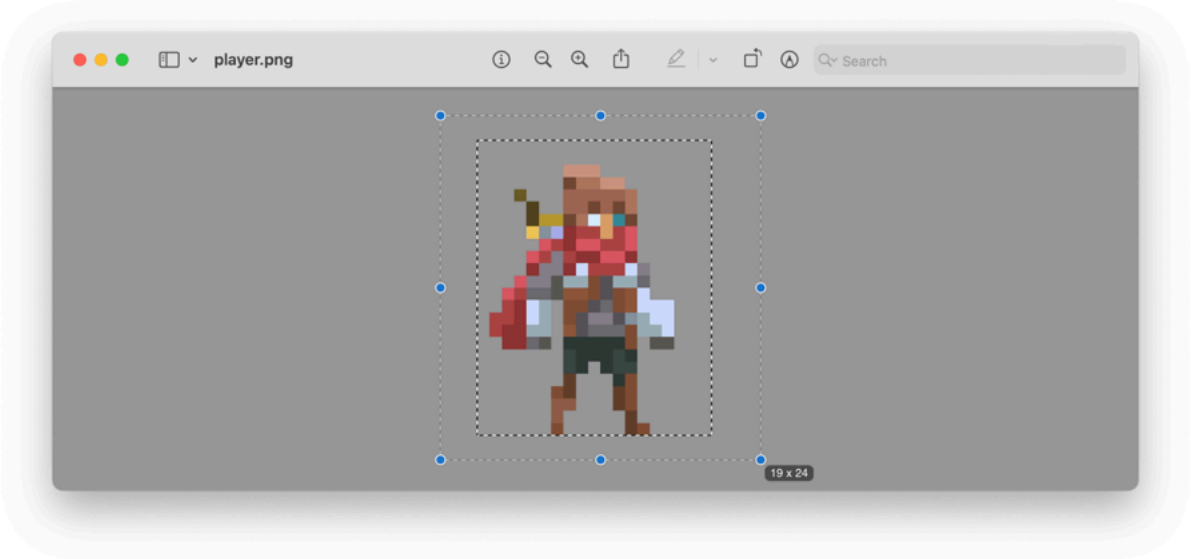


The development process is split into six distinct sections with their own tasks and objectives, this helps me to focus on the task I am currently working on as well as gives me a clear idea of what features can be added/are missing.

There will be some overlaps between the design stage and the implementation stage, however each section will have its ‘exclusive’ content too. Such as visual designs and code debugging.

The Game Engine

Player Entity



This is the player sprite we will be using, it is 23px tall and 19px wide, we will be using the pygame built-in Rect function to create a Rect object for the player for collision and physics simulation.

Below is a list of skills the player can do.

Player Skill	Key Bind	Potential Implementaiton	Notes
Move	Left Right Keys **[←] [→]**	Set Horizontal Constant Velocity	Still possible in midair
Jump	Up Arrow Key **[↑]**	Sudden Upwards Velocity Increase	1s CD, disable jump in mid air (so no double jumps)
Big Jump	**[Z]**	Similar to Jump, but with a higher velocity	3s of CD
Dash	**[X]**	Sudden Increase in the Horizontal Velocity	1s CD

To implement these functions I need to assign certain variables to the entity to control its movement. Position and velocity is represented by a list [x,y] with corresponding coordinates. Later in the section we will figure out a way to get player input and move the character accordingly. But this movement mechanism relies on the position vector to be updated every frame using velocity.

Below is the formal definition of the game coordinates and movement.

Let $\mathcal{P}$ be a 2-dimensional vector space over the field $\mathbb{R}$ of real numbers. Elements of $\mathcal{P}$ are ordered pairs represented as $\mathbf{p} = [x, y]$, where $x, y \in \mathbb{R}$. In the context of motion in a plane, $\mathbf{p}(t)$ represents the position vector of a particle at time t.
The operations of vector addition and scalar multiplication in $\mathcal{P}$ are defined as:

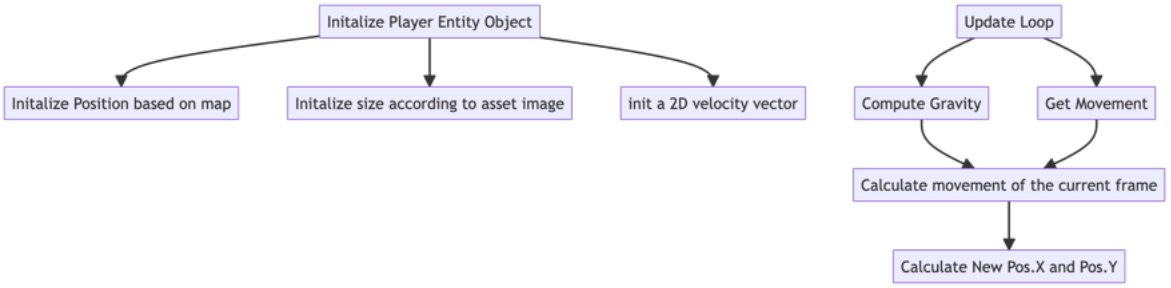
1. Vector Addition: For any two vectors $\mathbf{p}_1 = [x_1, y_1]$ and $\mathbf{p}_2 = [x_2, y_2]$ in $\mathcal{P}$, their sum is given by: $\mathbf{p}_1 + \mathbf{p}_2 = [x_1 + x_2, y_1 + y_2]$

2. Scalar Multiplication: For any scalar $\alpha \in \mathbb{R}$ and vector $\mathbf{p} = [x, y]$ in $\mathcal{P}$, the scalar product is: $\alpha\mathbf{p} = [\alpha x, \alpha y]$

Using the tools of linear algebra, we can define a basis for $\mathcal{P}$, typically chosen as the standard basis $\{\mathbf{e}_1, \mathbf{e}_2\}$ where $\mathbf{e}_1 = [1, 0]$ and $\mathbf{e}_2 = [0, 1]$. Any vector $\mathbf{p}$ in $\mathcal{P}$ can be expressed as a linear combination of these basis vectors.

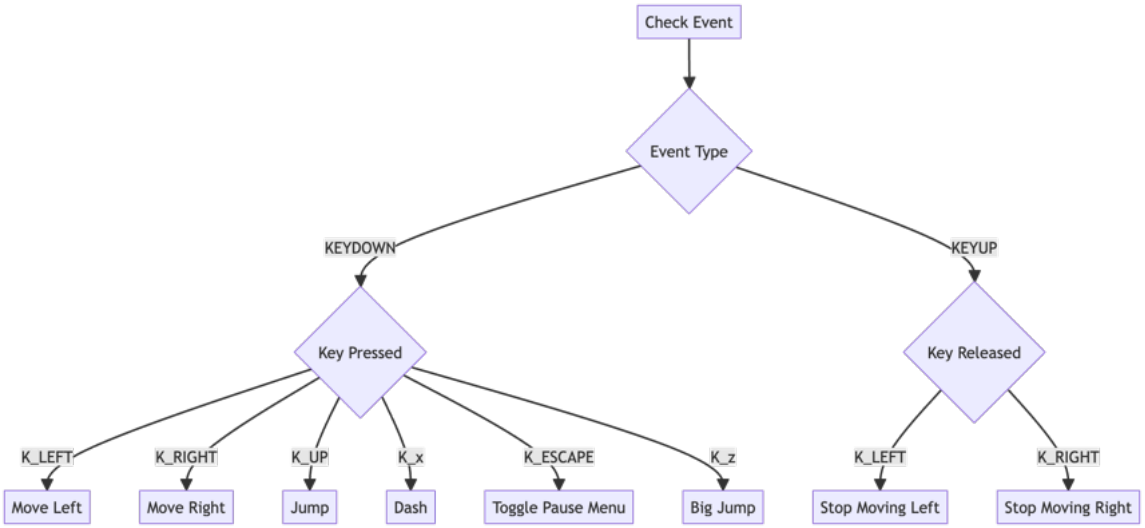
Now, considering the position vector $\mathbf{p}(t)$ as a function of time t, the derivative of $\mathbf{p}$ with respect to t gives the velocity vector $\mathcal{V}(t)$: $\mathcal{V}(t) = \frac{d\mathbf{p}}{dt}$
This velocity vector $\mathcal{V}(t)$ captures the instantaneous rate of change of position with respect to time.

With that in mind, we can start designing the player object. We will have a position vector as describe above, and we will have a velocity vector as well in order to deal with movement and gravitational acceleration.



Player Control & Movement

Right now the player just sits on the map and doesn't do anything, and it needs to be moving rather than stationary. We can incorporate algorithms that deal with keyboard input to move the player around. Below is a flow chart.



Note that I have two types of events, one is KEYDOWN, and one is KEYUP, you might think by just using KEYDOWN is enough, but that creates a bug, for example, when the left arrow is pressed, the player moves left, great, but when the key is released, the player is still moving right. Therefore we need a KEYUP event to stop the player from moving if the keys are no longer being pressed.

Other than the arrow keys which controls the movement, we also have two player abilities, namely big jump and dash which are triggered by the X and Z key.

Below is the pesudocode.

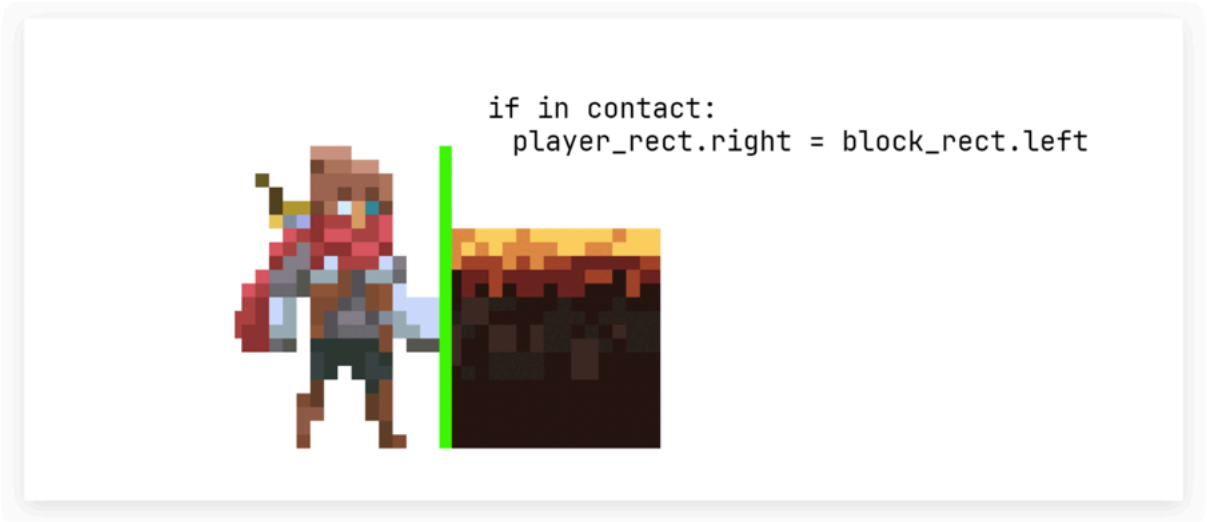
```
GET ALL THE INPUTS FROM PYGAME EVENTS:
    IF KEY IS PRESSED DOWN:
        IF THE PRESSED KEY IS THE LEFT ARROW:
            CALL MoveLeft()
        IF THE PRESSED KEY IS THE RIGHT ARROW:
            CALL MoveRight()
        IF THE PRESSED KEY IS THE UP ARROW:
            CALL the Jump function for the player

        IF THE PRESSED KEY IS X:
            CALL the Dash function for the player
        IF THE PRESSED KEY IS ESC:
            OPEN the in-game menu
        IF THE PRESSED KEY IS Z:
            CALL the BigJump function for the player
```

```
IF KEY IS RELEASED:
    IF IT IS THE LEFT ARROW:
        CALL StopMovingLeft()
    IF IT IS THE RIGHT ARROW:
        CALL StopMovingRight()
```

Entity Physics Simulation

Although the tilemap system is not designed yet (but will be in later stages), we can start thinking about how can we implement the physics simulation between entities.

The physics simulation only happens between the entities and blocks, so we need to a way for the entities to know if it's in contact with the blocks and in which direction. Luckily, pygame comes with the rect.colliderect() function and it turns a Boolean value indicating whether there is collision happening between two objects.

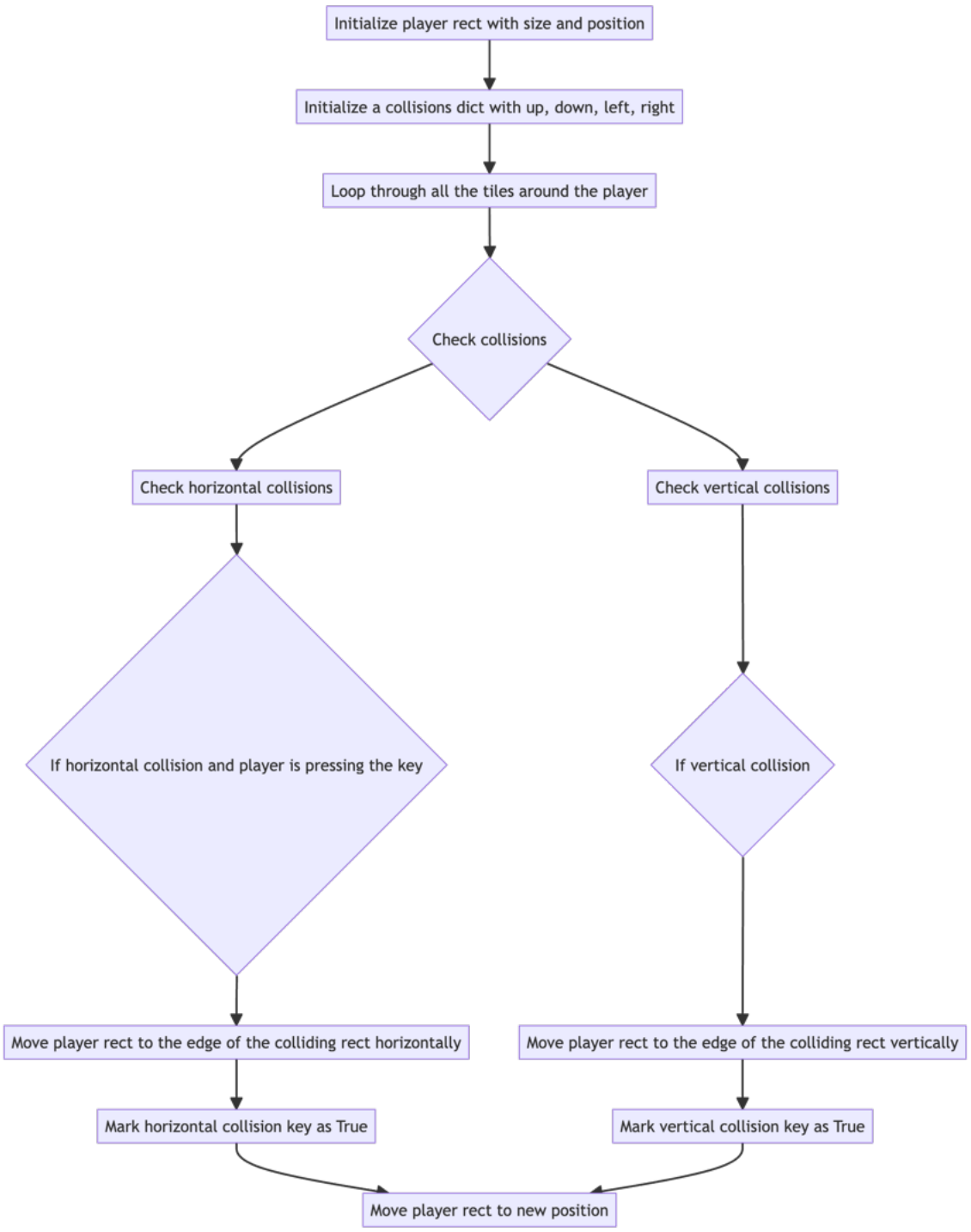


The physics system works like this, whenever an object collides with another object, the engine 'corrects' the player coordinate to the edge of the block coordinate so even if the player tries to run into the direction, player's position will not change, therefore creating a illusion that there is something blocking the way.

Above is an example where a block on the player's right, so even if the player tries to go right, the right edge of the player will always be 'corrected' to the block's left edge, therefore it can't move.

Similarly, when the player is standing on the ground, which means the player rect's bottom is in contact with block's top. The player's y coordinate will be corrected so it doesn't sink or fall through the ground.

Flowchart for collision checking:

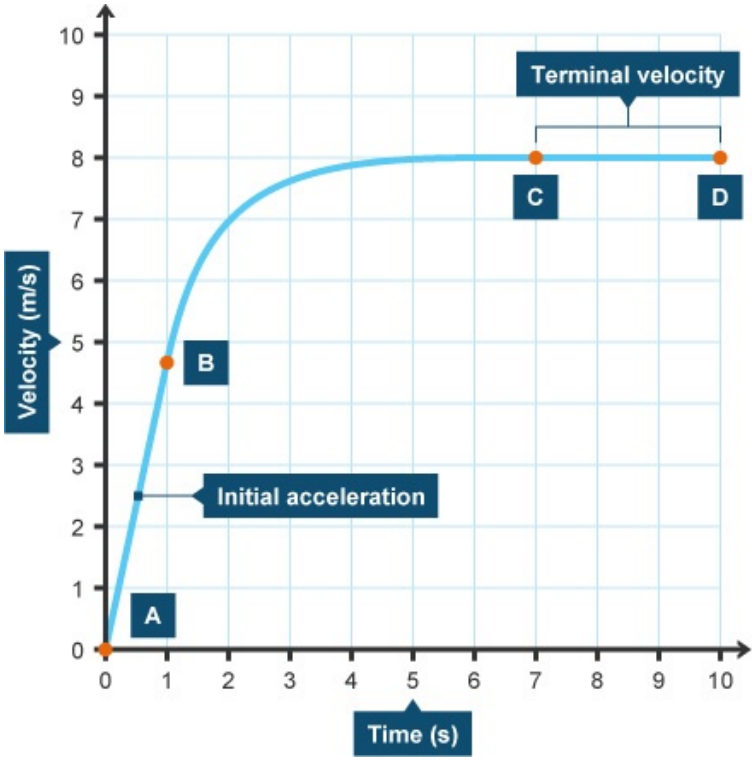

Initialize player rect with size and position
Initialize a collisions dict with up, down, left, right
Loop through all the tiles around the player
Check collisions
Check horizontal collisions
Check vertical collisions
If horizontal collision and player is pressing the key
If vertical collision
Move player rect to the edge of the colliding rect horizontally
Move player rect to the edge of the colliding rect vertically
Mark horizontal collision key as True
Mark vertical collision key as True
Move player rect to new position

Gravity & Terminal Velocity

Gravity on the other hand is actually pretty straight forward to implement, we can simply add a downward value to the velocity every frame and that could be something as simple as

```
gravity = 0.1
players velocity[y] += gravity
```

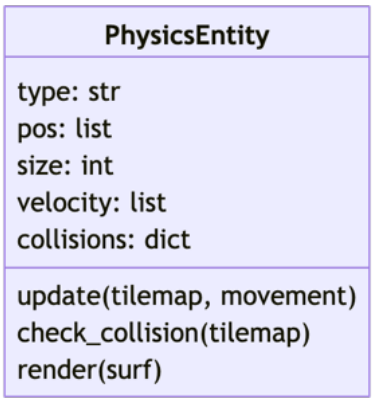
However, there is a issue where the downward velocity will increase indefinitely and the speed will increase at an exponential rate, which is definitely not normal, so what went wrong here?



While our acceleration calculations are accurate, there's a crucial oversight in our implementation—air resistance. There is this concept in physics called the terminal velocity, where an object in free fall reaches a fixed velocity after a certain period of time, even though the gravity is accelerating the object down, there's an upward opposing air resistance force slowing down the acceleration, as the object falls quicker, the air resistance also increases. When these forces are equal, acceleration becomes zero and the object reaches terminal velocity.

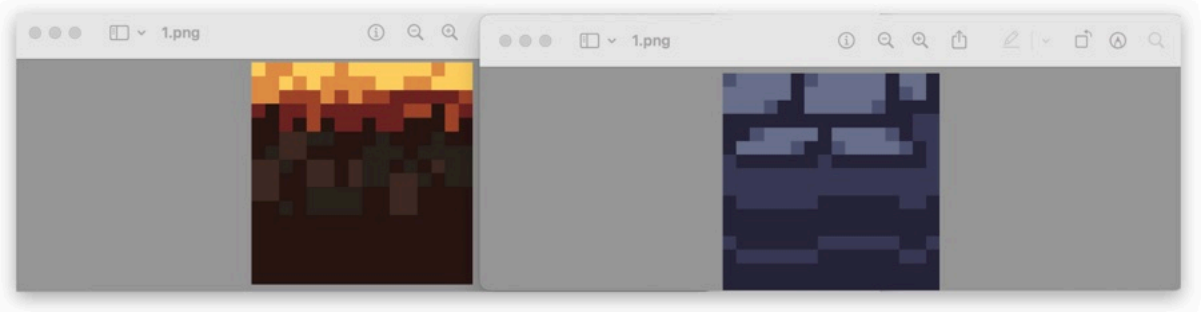
```
self.velocity[1] = min(5, self.velocity[1] + 0.1)
```

This one line of code is able to fix it instantly.

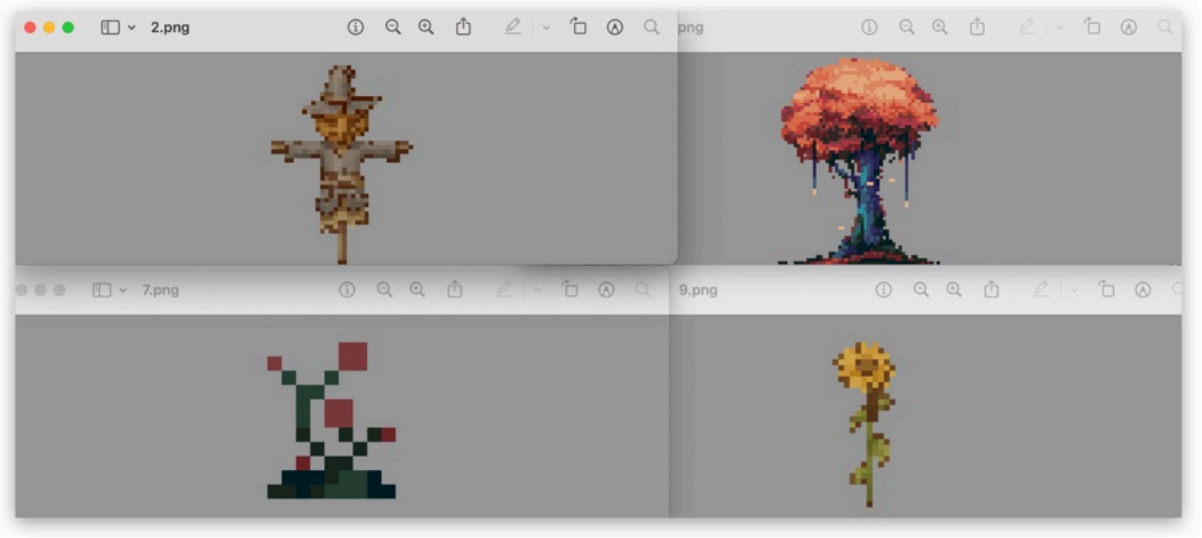

PhysicsEntity
type: str
pos: list
size: int
velocity: list
collisions: dict
update(tilemap, movement)
check_collision(tilemap)
render(surf)

Tilemap & Blocks

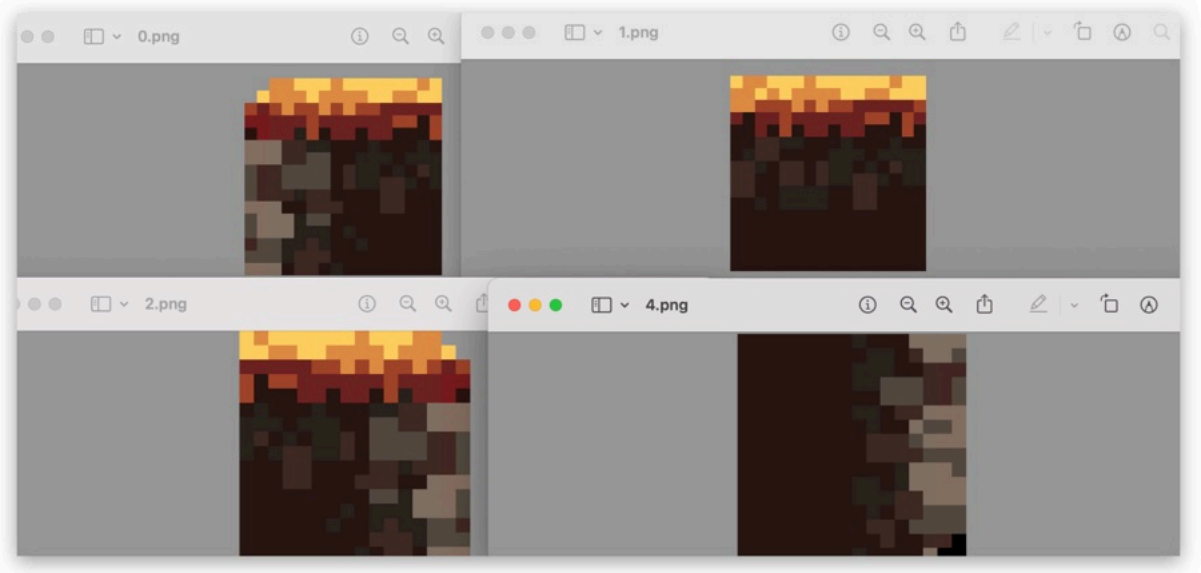
Tilemaps, which are also known as blocks, is the building block of my level maps. There will be two types of tiles: the physics tiles which the player and enemy entities can interact with (to stand on it, collision with the tiles etc.), and the other type of decorative tiles purely for visual effects with no physics interaction.



To add more variety to my game, I am going to have two physics tiles, namely grass and rock. Below are some of the decorative objects I am going to use in my map and there are plenty more.



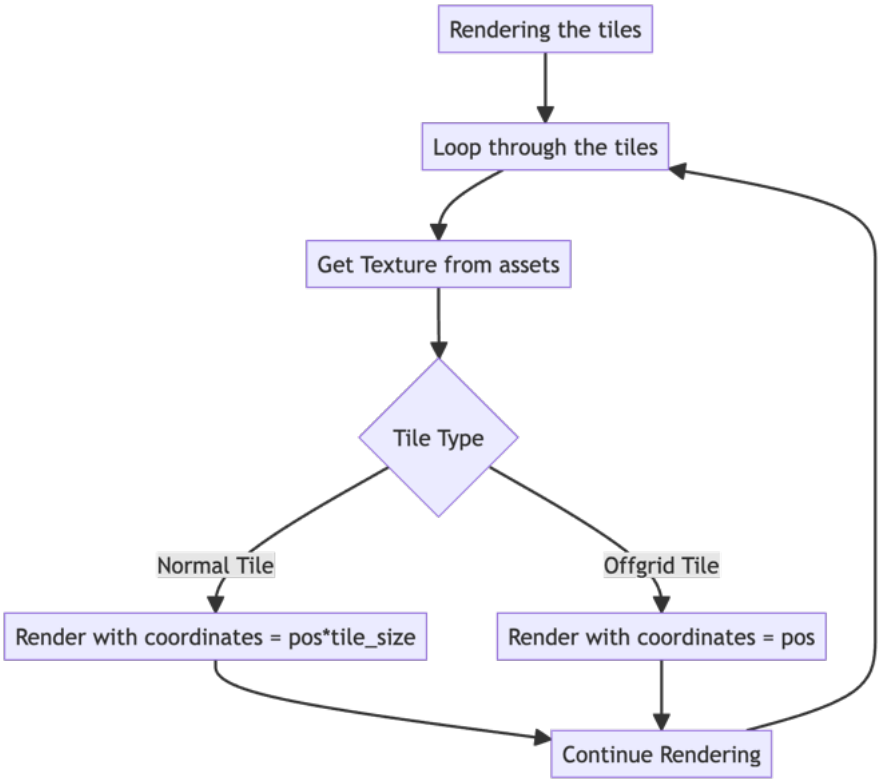
For the physics tiles, they are going to have few variants to it, why? Because a grass block has different textures depending on if it's next to a block, or nothing for example, to produce a natural look.



And here is how the tilemap system is going to work. We will be using a dictionary to store the map using a coordinate system. Each tile has two properties, its tile type and variant. A single tile will look something like this, with the key representing the tile location and its properties, such as type and variant.

For more usability, I will actually have two seperate tilemaps, one is for the tiles that are squares, while the second is for decorative blocks that are positioned 'offgrid'. These decorative blocks are not spaced apart by whole multiples of the block width, which in this context is set to 16 pixels. The tiles will have a fixed size to make the rendering work, so I am adding that parameter into the class variable as well.

We can add a render function to render the desired tiles onto the screen, by calculating the screen pixel position based on the tile coordinates using simple math.

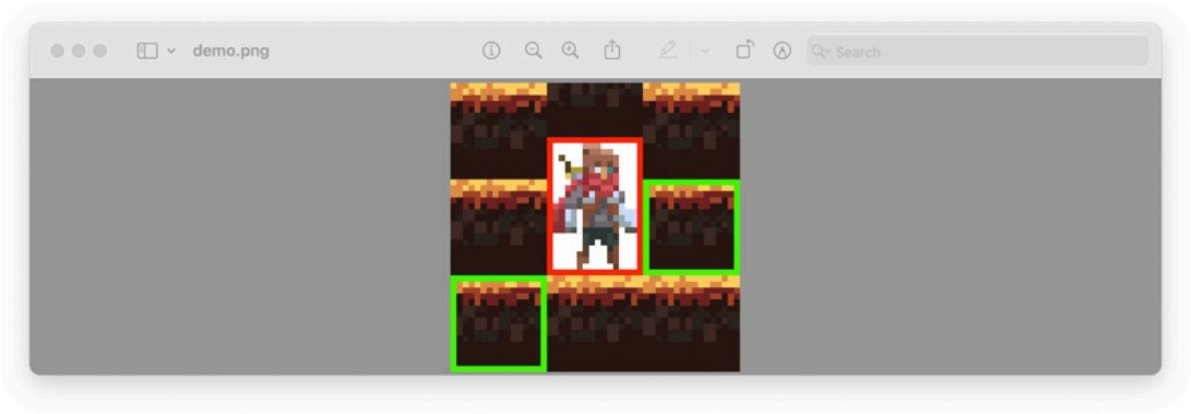



```
PROCEDURE render(surface to be rendered on):
    FOR offgrid tiles:
        surf.render(correct_image_texture, pos)
    FOR normal tiles:
        surf.render(correct_asset_according_to_type_and_variant, pos*tile_size)
```

Now, the tiles should be correcting displaying onto the screen but they don't have any physics interaction, it is purely visual. Let's consider how should the tile interact with the player.

This is pretty simple, since we already have our player entity and the player has a rect object, we can run some collision check if the player is in contact with the blocks. Each block is going to have their Rect object and pygame has a colliderect method to check that. While we can be continuously checking collision between the player and all of the blocks, this is not efficient at all on the hardware resources. It would be fine if we have a small map, As the map size increases, CPU power usage for collision checks also rises, but much of this is

inefficient since the player will never interact with certain blocks.

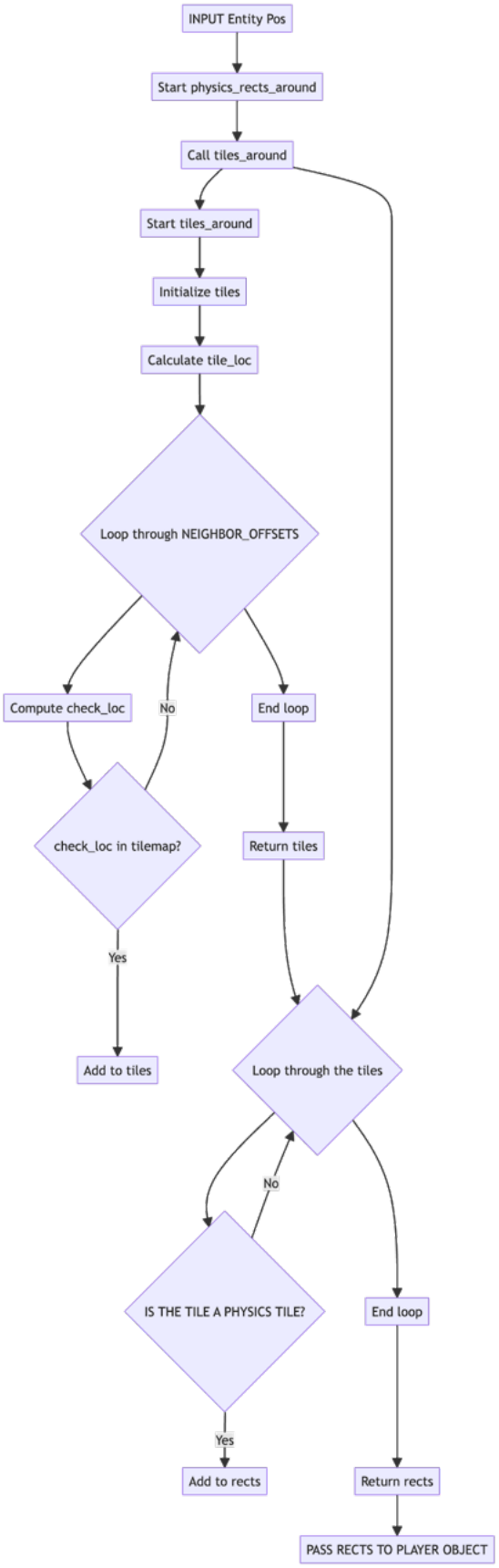


Instead, we can just check the tiles that are nearest to the player, namely, if we picture the player as a centre block, we just need to check the eight blocks surrounding the player. Which reduces the computational requirement but a lot.

Using coordinate geometry we can write the offset of the blocks as:

```
NEIGHBOR_OFFSETS :=
[(-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (-1, 1), (0, 1), (1, 1)]
```

If we think about it, the physics collision only applies to entities such as player and enemies, so we just need to pass the coordinates of these objects into the tilemap.



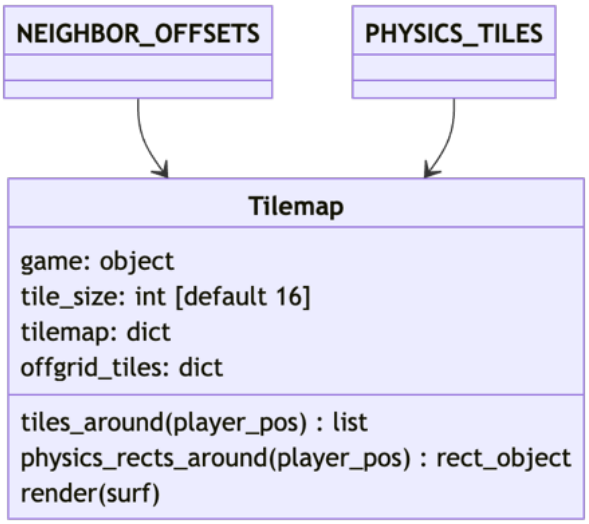
This is the steps for checking physics collision in between the player and the tilemap.

Initially, the player position is passed into the map and we loop through all of theneighboring tiles surrounding the player, which are the eight blocks in all directions.

I there is a tile in the range, check the tile type to seee if it's one of hte physics tiles(grass/stone). If so, we get the Rect object of the tiles and return it to the player object to stop it from moving.

The player rect will then update its location based on the tile rect, which creates this 'illusion' that there is actually physics simulation going on.

This is the class diagram for the Tilemap Class



Rendering & Camera

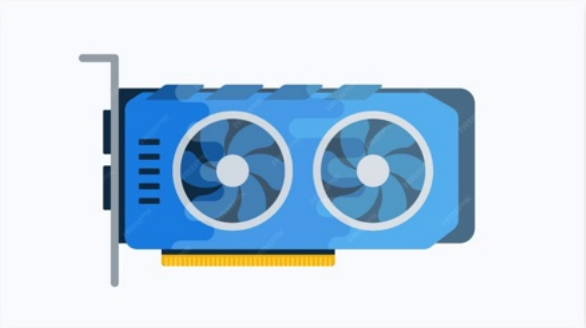
For rendering the gameplay onto the screen, I am actually going to have two surfaces to work with, the first is a render layer which has a fixed size of (640x360) and the other is the screen layer which actually shows up on our computer. In this case I have chosen a multiple of 2 to be the screen resolution, which is (1280x720). Note that the width of screen and height of screen and width of render and height of render needs to be same ratio, otherwise weird scaling issue appears and the image

The reason why we have two screen is because we can ensure that my game looks consistent across various devices and screen sizes. We can control the quality and the details of our game without being restricted by the actual screen resolution.

The render layer, being of fixed size, acts as our canvas where all the game elements, characters, backgrounds, and other assets are drawn. This layer has a predefined resolution which ensures that every element is rendered consistently, irrespective of the device's display resolution.

Now, once we have our game looking perfect on the render layer, the next step is to display it on the screen layer. By simply scaling up (or down) our render layer to fit the screen, we can achieve it perfectly without losing any details or causing any graphical distortions.

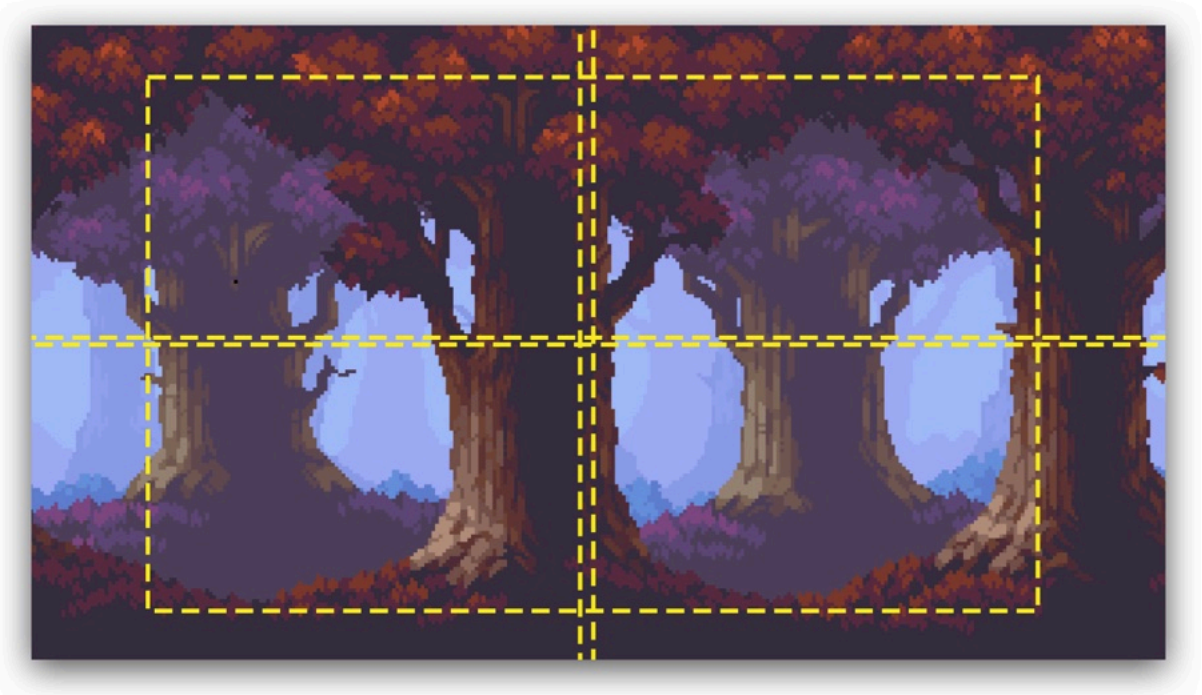
This system also provides us with a lot of flexibility. For instance, if in the future we decide to support higher screen resolutions, we don't have to modify the original game assets or redesign the gameplay. Instead, we can just adjust the scaling factor to fit the new resolution.



Another advantage of this two-layer system is performance. By rendering the game at a fixed resolution, we can optimize the performance for that specific resolution. This means that the game can run smoothly on lower-end devices, as the GPU only has to handle the fixed resolution of the render layer.

```
GAME LOOP:
update_everything() //stuff like characters, enemies, particle animations etc
display_everything => onto the render_surface
repeat and tick_FPS

KEEP LOOPING
```

Also, we need to make sure the camera follows the character wherever the player goes.



The camera should correct its position in real time. There are two types of corrections, and I am going to call them hard correction and soft correction. Soft correction ensures the player is centered at the middle and when the player stops its motion, there will be some easing animation so it looks natural to the eyes.

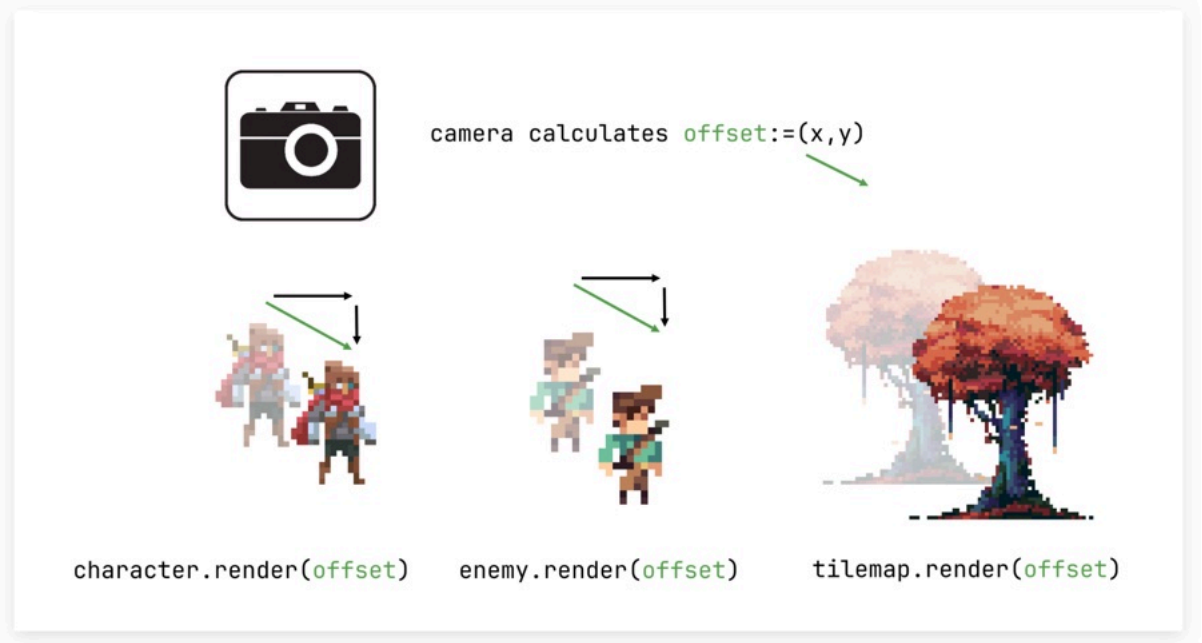
Hard correction is applied when the player touches the rectangular boundary as shown above, which are the four edges of the shape. This triggers immediate correction of the camera position and does not have easing animations.

Based on the current player position, the camera will create an offset with this formula

```
offset.x += (player_asset_width - display.width() / 2 - offset.x)
offset.y += (player_asset_height - display.height() / 2 - offset.y])
```

it positions the offset to the center of the screen to the correct position by halving the resolution, and it gradually ‘catches up’ to the target value by subtracting the differences from the original value.

Using the offset variable, we can integrate it into our object classes and the map. This allows us to render them with a specified offset on the screen, creating a dynamic camera movement.

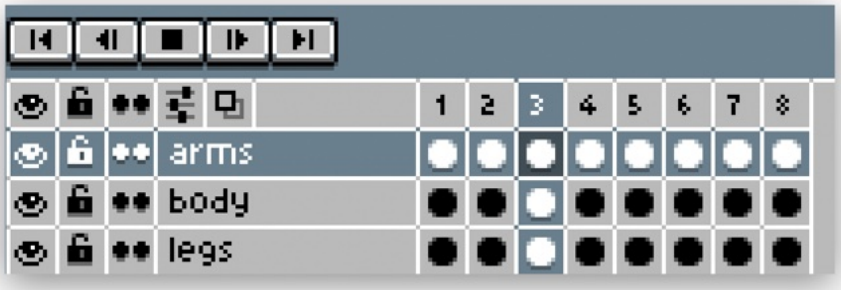


We can also adjust the speed of this adjustment by multiplying the action with a speed constant, to create smoother motion.

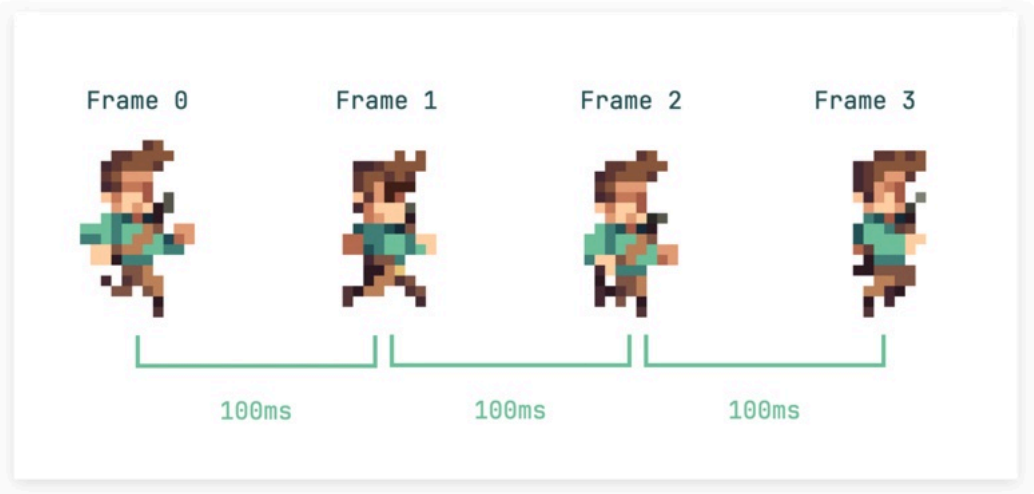
```
offset.x += update*speed
offset.y += update*speed
```

Sprite Animation & States

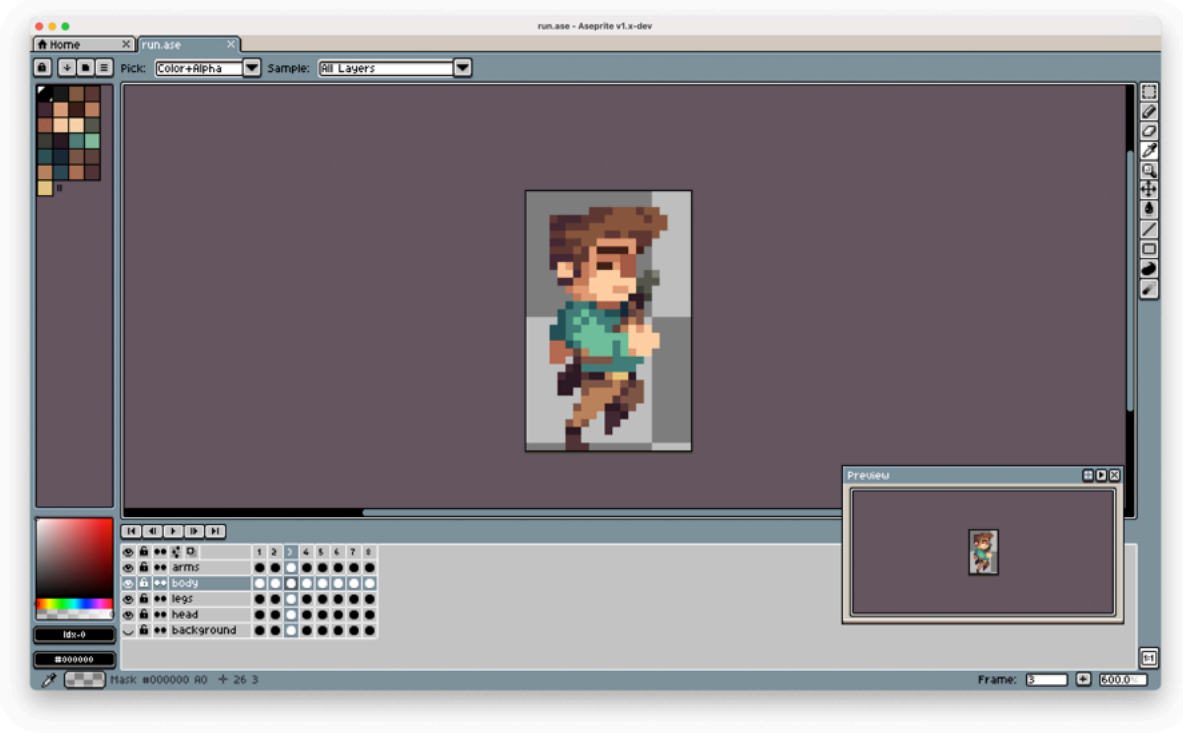
When discussing animated images, the .gif format is often the first that comes to mind. It is capable of doing short-looping animations and it is widely used on the internet.
Very unfortunately, pygame doesn't actually have support for GIF images, so we have to write an animation engine on our own.



To be able to do that, we need to realize that videos, or any animation is made of still images, called frames. The indiviual images are moving fast enough so it tricks our brain into thinking the images is moving.



The individual frames are played with a time gap, which is the reciprocal value of the framerate. As the animation reaches the final frame, it is looped back to the initial frame.



The frames are separately made with software ‘aseprite’ and we need to save them into a normal image format like png to be loaded ito the game engine.

The player character will have these animations triggered by certain conditions, called states.

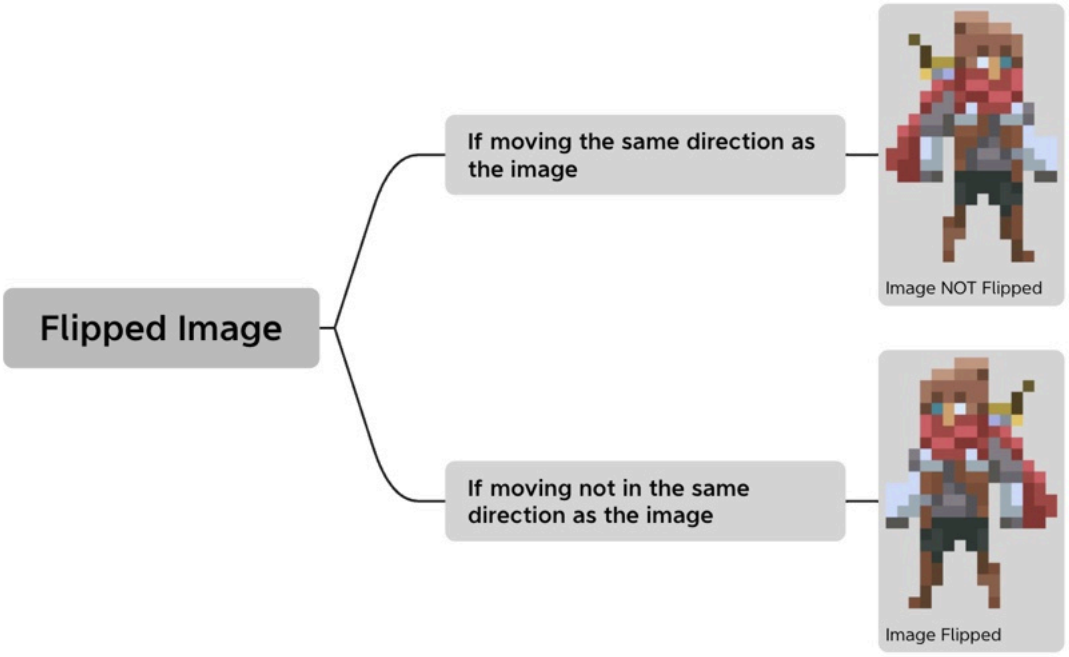
State Name	**Description & Condition**	**Images**
idle	if the character is not moving and standing on the ground, it is considered idling.	 (4 frames)

jump	When the player is off the ground. The jumping state only has one frame because the player does not move mid-air.	 (1 frame)
run	The running animation, triggered by the player's x-axis velocity. Which means the player is moving horizontally on the ground.	 (6 frames)
wall_slide	The player is able to 'hang' on walls, it falls slower than the gravity as there is friction on the walls. Triggered when the player is only in contact with left or right tiles and not on ground.	 (1 frames)

Enemy sprites' state animations follow similar logic.

Image Reflection

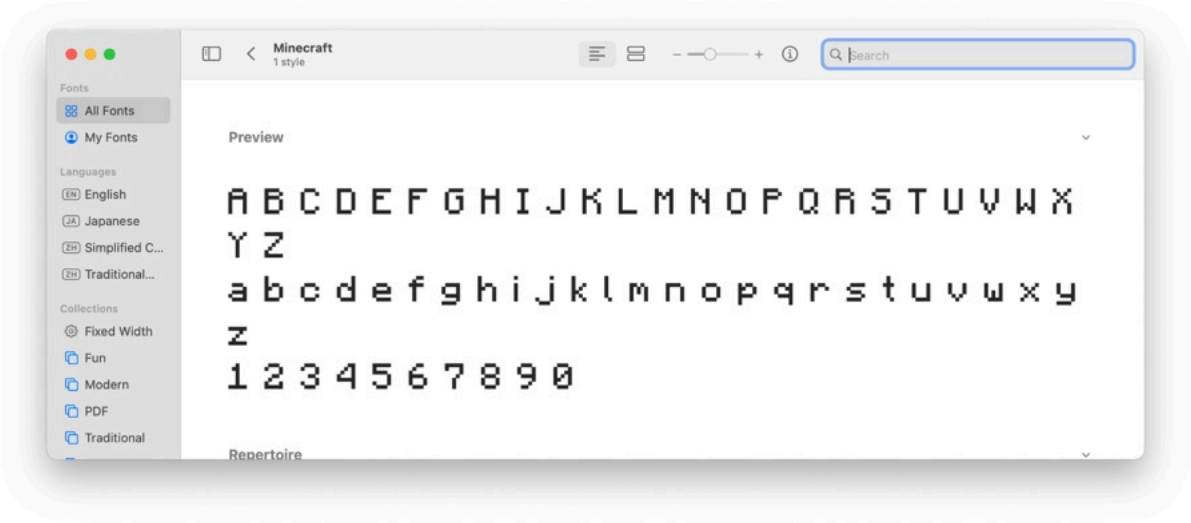
You may have noticed that the animation images are only facing right, and it doesn't look right when the character moves in the other direction. To overcome this, I am going to introduce a new boolean variable called 'flipped' to indicate if the character is moving in the direction that is the opposite of the image direction. So I can create a flipped version of the image for rendering.



Text Rendering System

Using text is very important in any game. It is used for displaying the score, the health of the player, the name of the player, the name of the level, etc. In this section, we will create a text rendering system that will allow us to display text on the screen.

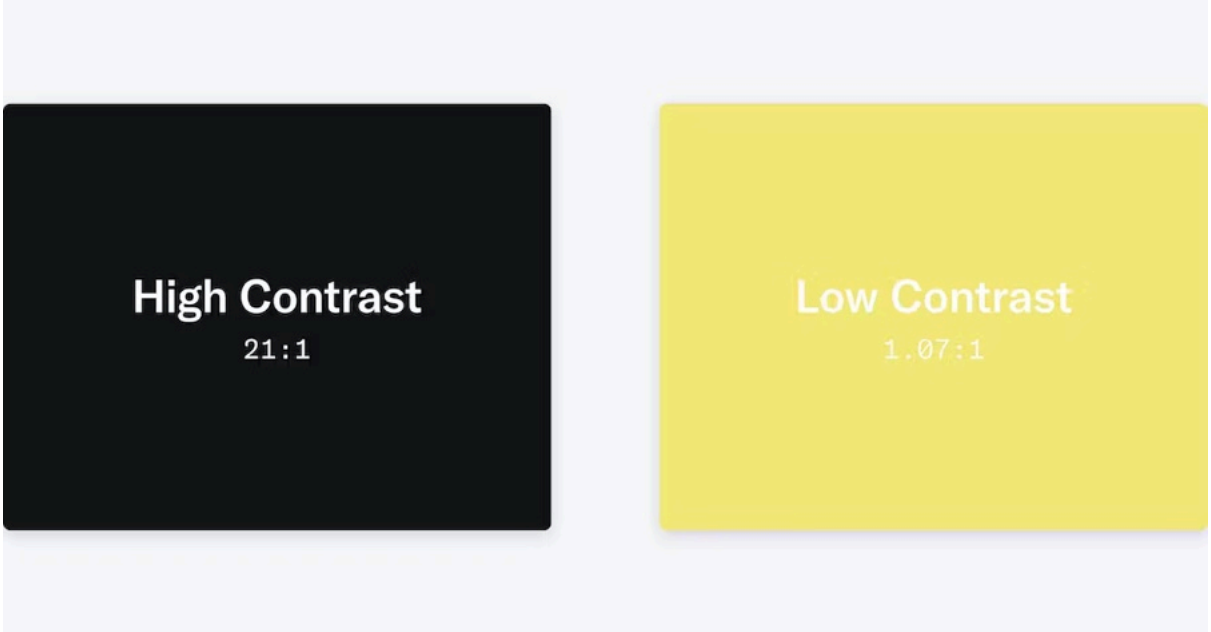
If we want to display texts in pygame, we follow this procedure, which is very similar to the one we used for displaying images, first we need to select and load a font we want to use, since I am going for a retro look in this game, I will use a 8-bit inspired font, called Minecraft.Otf.



Once we have loaded the font into the game, we pass the text into the font with the correct size and colour, then we render it into an image, and finally we blit it onto the screen with the correct coordinates.

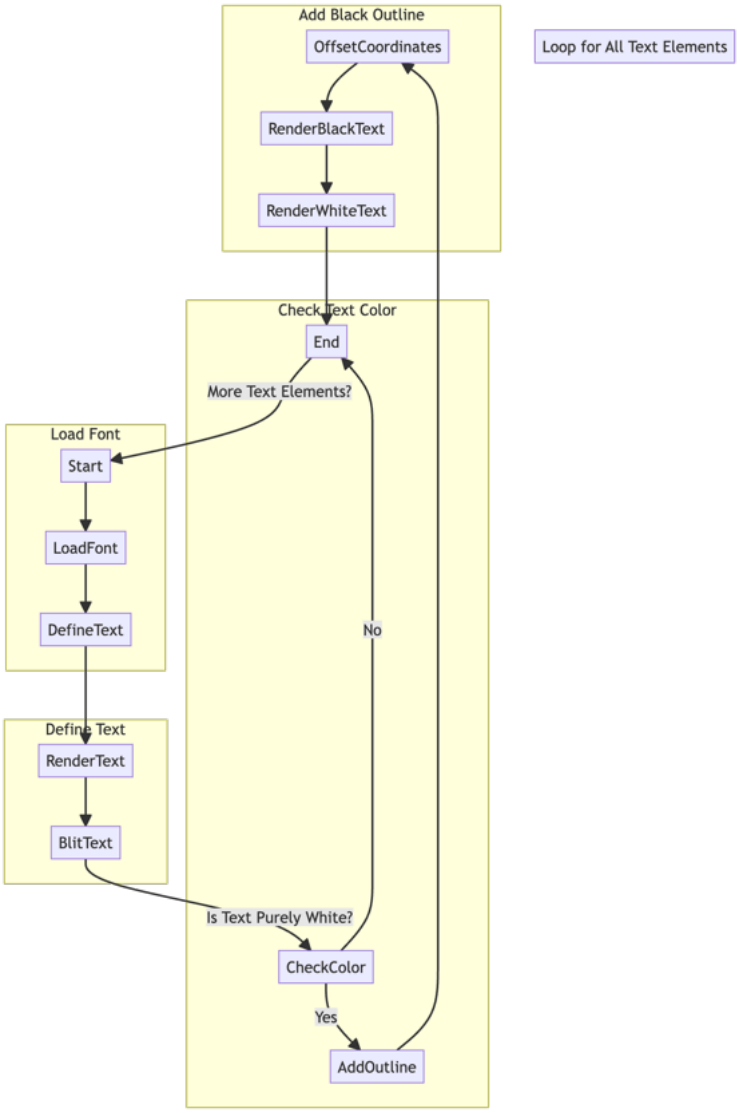
In order to keep the art style consistent, I will use the same font for all the text in the game, as well as the colour scheme.

However, if the text is purely white, you might notice a problem, it is not really readable with the background I am using, there is this rule in UI design where between the text and background, you should keep a certain contrast ratio, so that the text is readable, and the background is not too distracting.



In order to solve this problem, I will add a black outline to the text, so that it is more readable while not affecting the art style too much.

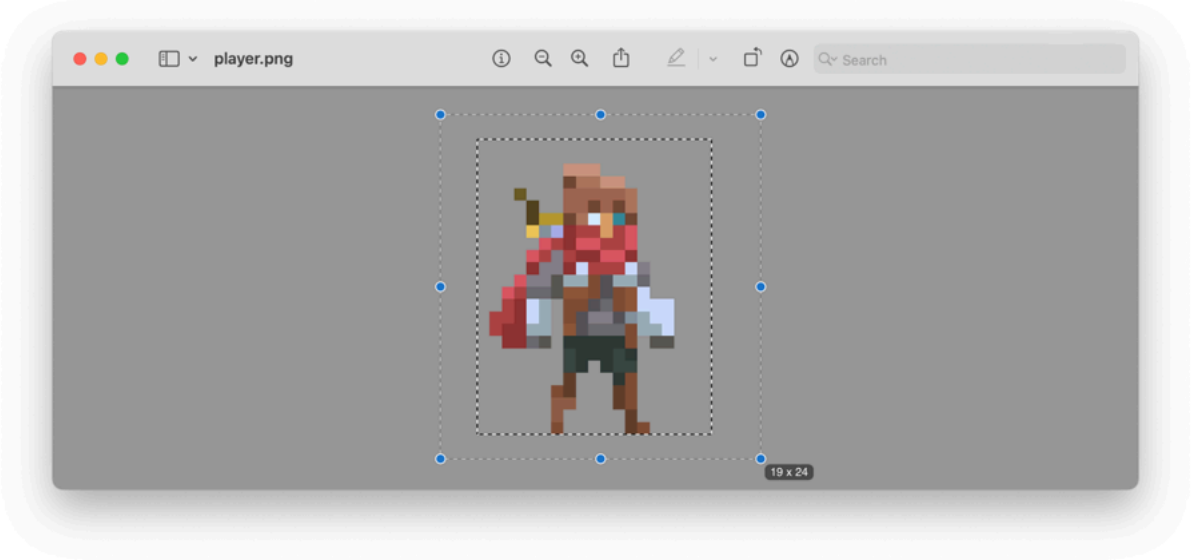
This is a lot more readable and looks more polished. Therefore I will use this method for all the text in the game - when there is a white text, I will first render a black text with the same font size, but with the x and y coordinates offset by 1 pixel, and then render the white text on top of it, achieving the shadow effect.




```
LOAD the desired font LOAD_FONT("Minecraft.Otf")
# RENDER the text into an image
text_surface = RENDER_TEXT(font, text_content, text_size, text_color)

BLIT the text image onto the screen at specified coordinates ( x, y)

IF text_color is purely white THEN
ADD a black outline (shadow) for readability
    shadow_surface = RENDER_TEXT_WITH_OUTLINE(font, text_content, text_size,
text_color)
    BLIT(shadow_surface, x + 1, y + 1)  # Offset by 1 pixel
ENDIF

REPEAT the above steps for all text elements in the game

END
```

Game Design

Main Character

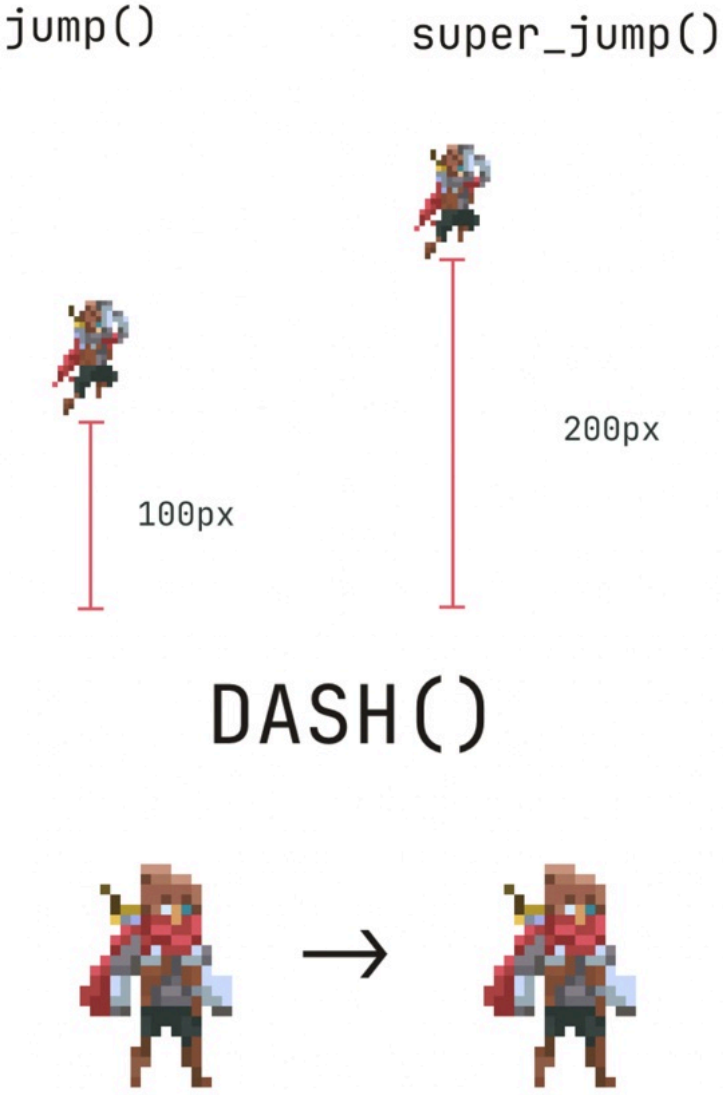


This is the player sprite we will be using, it is 23px tall and 19px wide, we will be using the pygame built-in Rect function to create a Rect object for the player for collision and physics simulation.

Below is a list of skills the player can do.

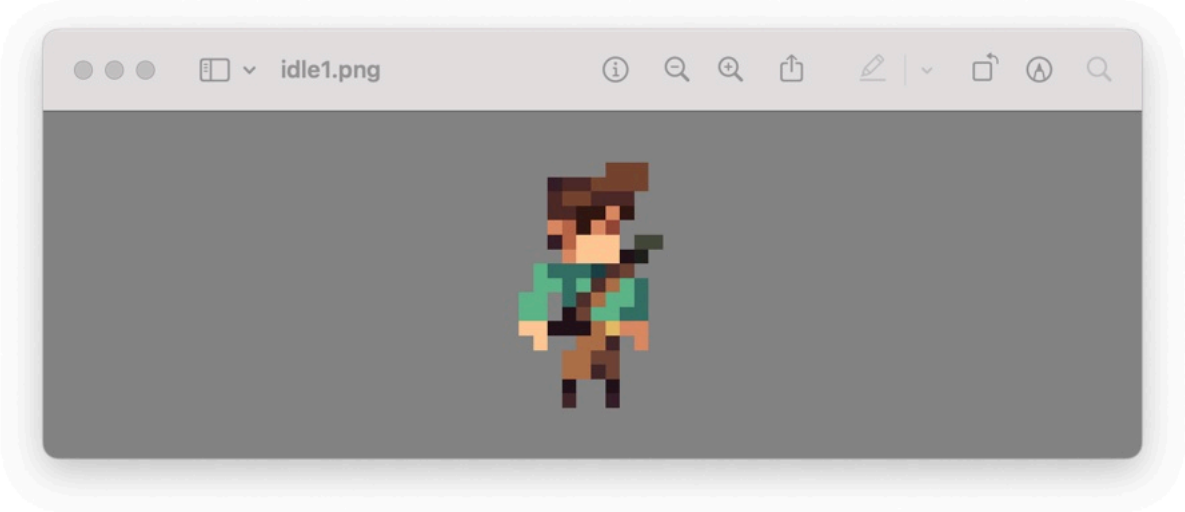
Player Skill	Key Bind	Potential Implementaiton	Notes
Move	Left Right Keys **[←] [→]**	Set Horizontal Constant Velocity	Still possible in midair
Jump	Up Arrow Key **[↑]**	Sudden Upwards Velocity Increase	1s CD, disable jump in mid air (so no double jumps)
Big Jump	**[Z]**	Similar to Jump, but with a higher velocity	3s of CD
Dash	**[X]**	Sudden Increase in the Horizontal Velocity	1s CD

Main Character Abilities (Dashing & Super-Jumping)

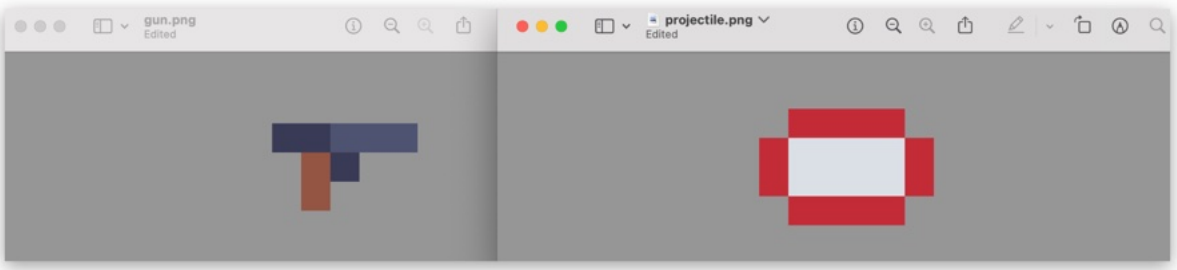


Ability	**Description**	**Image**
Dashing & Super-Jumping	Dashing will make the the player dash in the direction it is looking at, and super jumping can jump a lot higher than normal jumps, but requires more CD. (explained later)	

Enemies



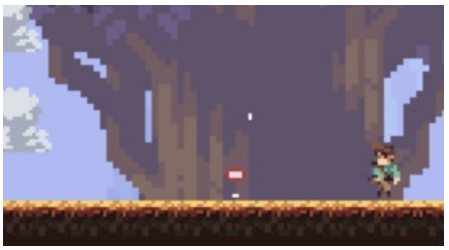
In the game, enemies will be spawned at specific locations on the map. The enemies have a weapon which they can use to fire on the player, and they have unlimited ammo.



The enemies are a bit shorter than the main character, being 17 pixels tall, so it actually makes it harder for the player to hit the enemies.

Enemy Logic

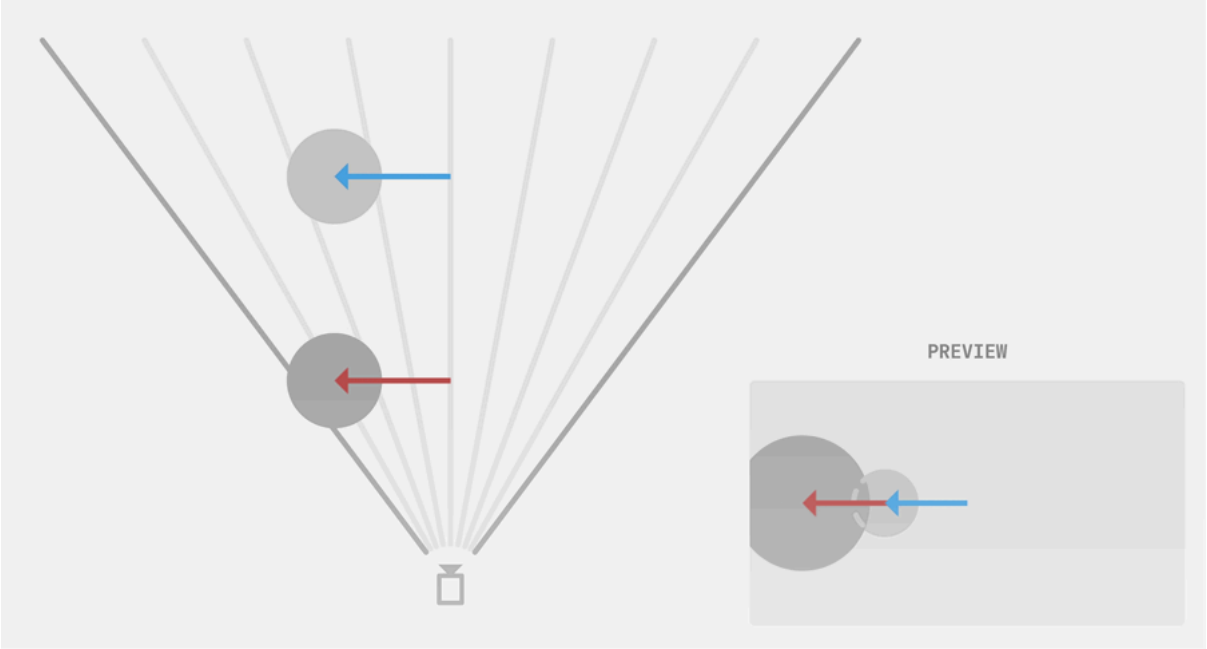
The enemies have two states, which are explained in the table below.

State	**Description & Condition**	**Images**
Wandering	An enemy walks around the platform, but they will not be able to leave the platform its spawned on. (which prevents them from falling into the void)	
Shoot	2 ways to implement it: either the enemy shoots randomly at given time intervals, or it shoots when the player is in sight. The second method would be harder to implement, as it requires player sprite detection.	

Visual Effects

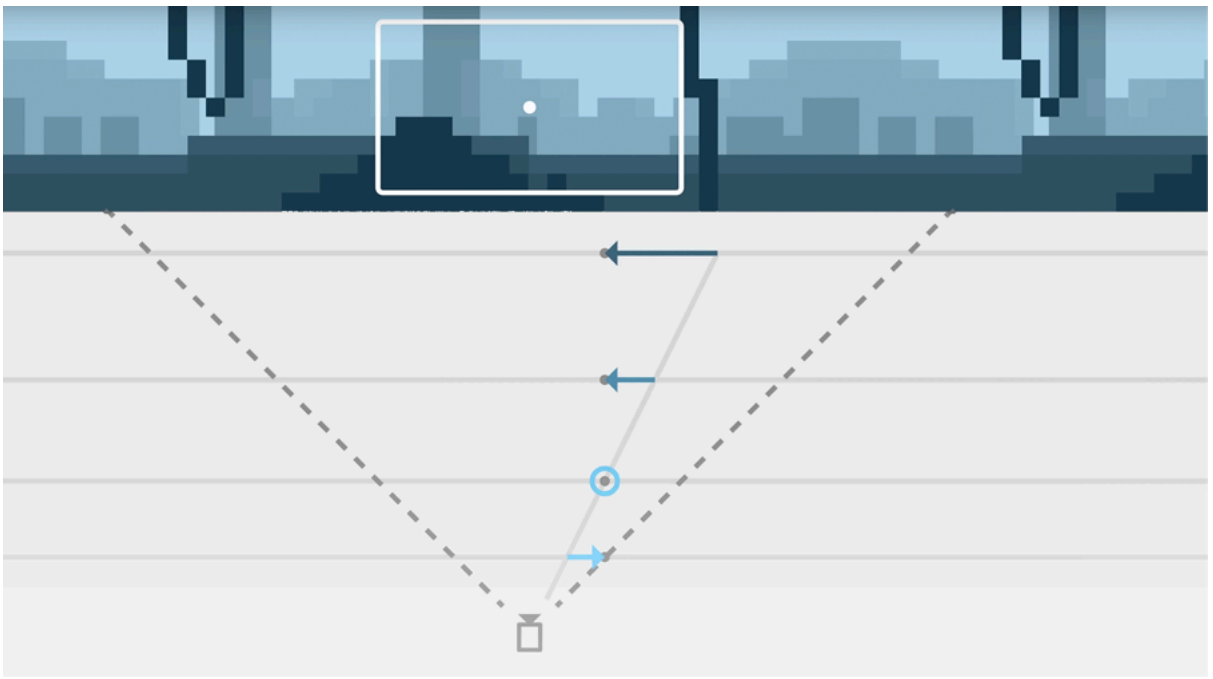
Parallex Scrolling & Clouds

Let's explain parallax scrolling, if you have ever been on a moving train, you have probably experienced this effect. This happens when you are looking at something far and something near you at the same time. Even though you are travelling at the same speed relative to these objects, from your perspective, it is going to look like these objects are moving at different velocities.

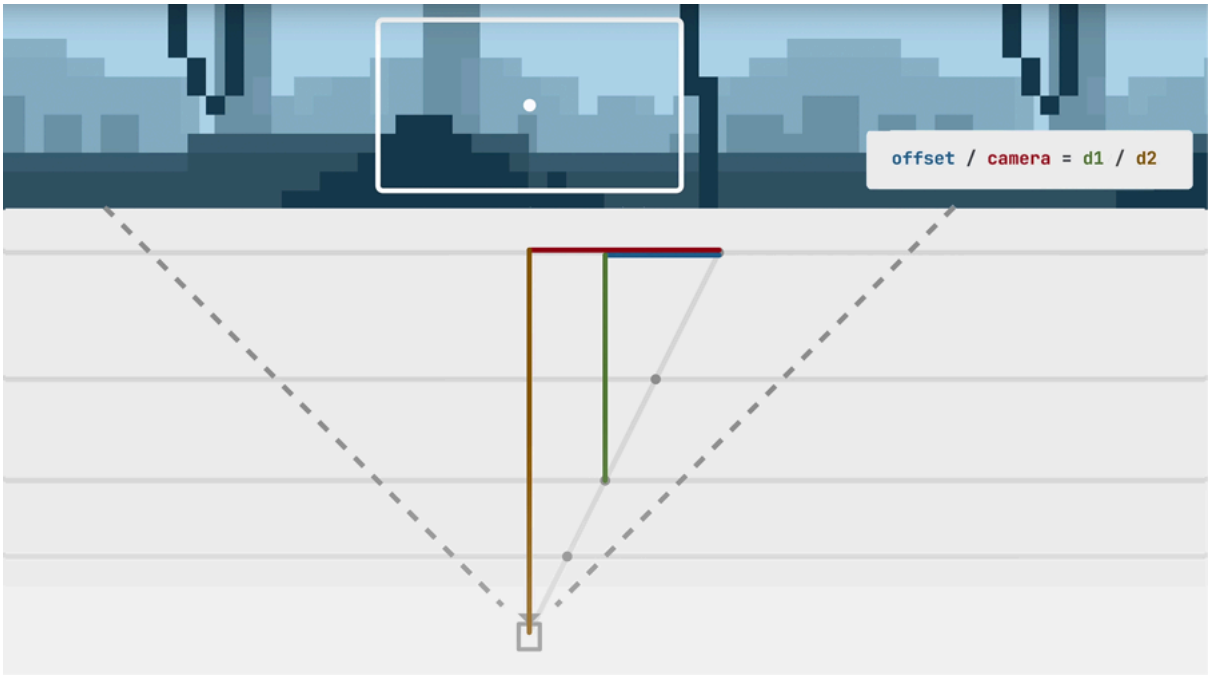


We can see using this visualization, where our perspective stationary at the camera and the dark grey & light grey ball represents objects at different distance. The red & blue arrows have moved them by the same length but to our eyes (rendered in the preview tab), they have moved differently.

We have use this technique to create some parallax animations, which is going to help to make our game look more realistic. One way to do that is by creating multiple layers, which represents objects at different distances. Normally it would be a background with distant mountains and scenaries, and we have have a road the player can walk on which is near us.



To do this, we can simply pick a layer to be fixed and compute the angle using the camera pos and the other coordiante. A line is fixed by two points.

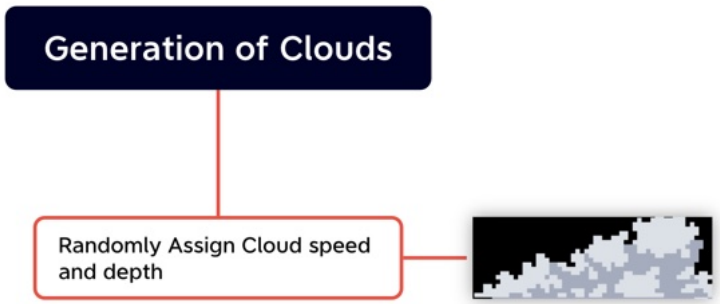


You will notice that we have similar triangles and that’s exactly what we are gonna use, the ratio of these layers lengths are the same.

Even though we are not going to use parallax scrolling on our background, as it would be too chaotic visually, we are still going to implement this techniques with clouds. To create a sense of depth.

Let's imagine if there is two clouds, we can move the y axis value to change the height of our camera. From the camera's perspective, the clouds will move different heights due to parallax scrolling.

Upon the generation of clouds, the individual cloud object will be assigned a depths value, which affects the way of rendering as the player moves around.

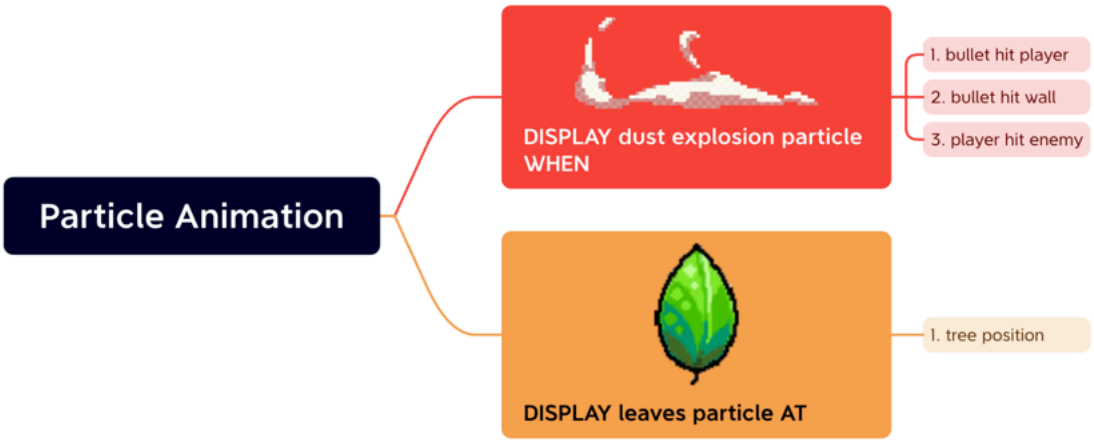


Cloud
- pos: list - img: any - speed: float - depth: float
+__init__(pos: tuple, img: any, speed: float, depth: float) : : void +update() : : void +render(surf: any, offset: tuple) : : void

Clouds
- clouds: list
+__init__(cloud_images: list, count: int) : : void +update() : : void +render(surf: any, offset: tuple) : : void

Particles

There are several ways to trigger the particles, it’s quite straightforward.

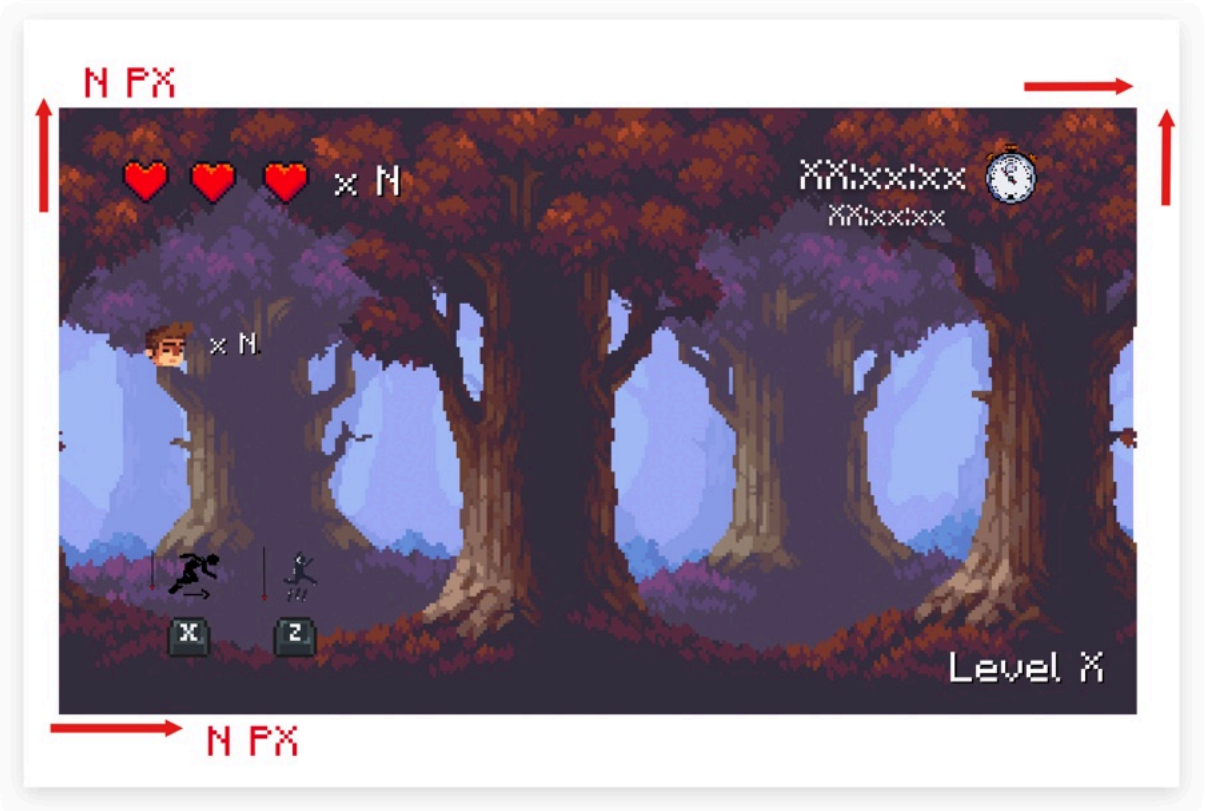


It triggers the particle animation whenever these conditions are met.

Spark
+pos: list +angle: float +speed: float
+Spark(pos: list, angle: float, speed: float) +update() : : bool +render(surf: pygame.Surface, offset: tuple=(0, 0)) : : void

This is the diagram for the Spark class for now but we will go over the exact details of the particle animations in the implementation section.

Screenshake



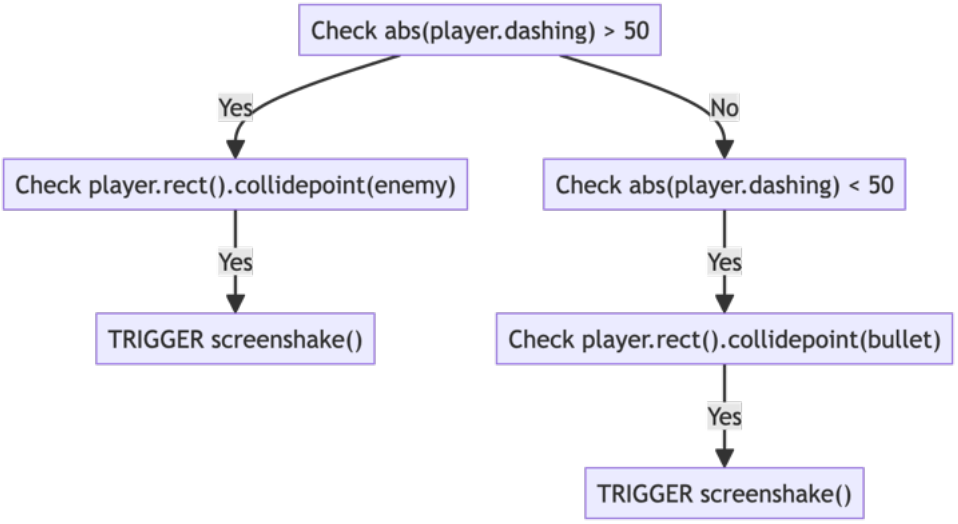
I want an effect where whenever the character gets hit and dashes into the enemy, the screen shakes and player is given visual feedback.

The screen shakes by moving in the x and y directions for a few pixels and wobbles around the centre, then finally resetting to (0,0). The offset is generated by a random number in the range of [0,1] and multiplied by a constant, then the screenshake is halved every time so it goes back to 0.

```
screenshake_offset =
(random.random() * screenshake - screenshake / 2,
 random.random() * screenshake - screenshake / 2)
```

therefore it adds the offset when displaying the game display onto the screen.

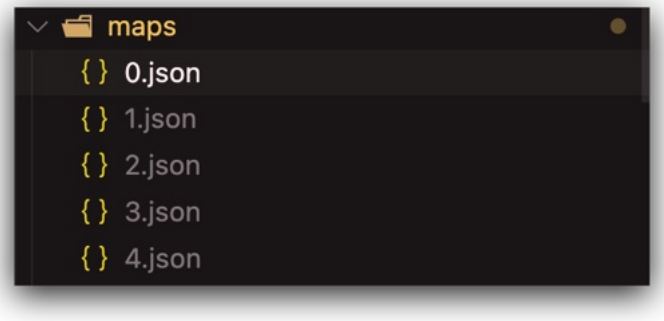
```
screen.blit(pygame.transform.scale(display, screen.get_size()), screenshake_offset)
```


Check abs(player.dashing) > 50
Yes
No
Check player.rect().collidepoint(enemy)
Check abs(player.dashing) < 50
Yes
Yes
TRIGGER screenshake()
Check player.rect().collidepoint(bullet)
Yes
TRIGGER screenshake()

Maps & Level Editor

Loading levels

We have already defined the structure of the tilemap system, and we need to package a specific arrangement of tiles into a level map.



As discussed earlier, it is easy to save the tile map data into a json file, so we can save and load maps into actual files instead of just a string of characters.

```
1. PROCEDURE load_level(map_id):
   tilemap LOAD the correct MAP
   leaf_spawners = []
   enemies = []


   2.1: LOOP tree in tilemap.extract([('large_decor', 2)]):
       TAKE the  RECT and add to leaf_spawners LIST


   2.2: LOOP spawner in tilemap.extract([('spawners', 0), ('spawners', 1)]):
       if spawner['variant'] == PLAYER:
           player.pos = spawner['pos']


       2.3: else:
           TAKE the position and put it into the ENEMY LIST
```

This is the procedure to load a level from a json file, we are passing in a map_id into the function and the raw data will be retrieved and processed as a dictionary object. Then we iterate through the elements in the dictionary and this is the portion of the algorithm that deals with entity objects such as the player, enemies, and trees. If so, these objects are going to be converted into pygame rects and added to a list to keep track of the instances.

Notice that the tile blocks will be stored in the tilemap object, and the render function inside the class is going to take care of the blocks, because the blocks need minimal processing.

Additionally, there are some things to reset whenever a level starts.

Reset item	**Description**	**Image**
Player Movement	This is more of a debug thing but the player movement needs to be reset so it doesn't move around using input from previous levels.	
Current level timer	Whenever the player moves on the next level/reload level. Current level timer needs to be reset for correctly calculating the total run time, which will be explained in depth in later sections	
Player Status (only when restarting)	Only when the player has pressed the restart button in the menu, player status such as HP will be reset as the level loads.	

Tutorial Level

Since I have the level variable to track which level has been loaded, I will add a tutorial level at the start for the player to familiarize themselves with the controls.

There will be a floating text showcasing the controls on top of the screen. The level will consist a flat ground and a single enemy to help the user to know what to do.

This is the level in the level editor.

Pseucode for rendering the prompts would be trivial.

Level Editor

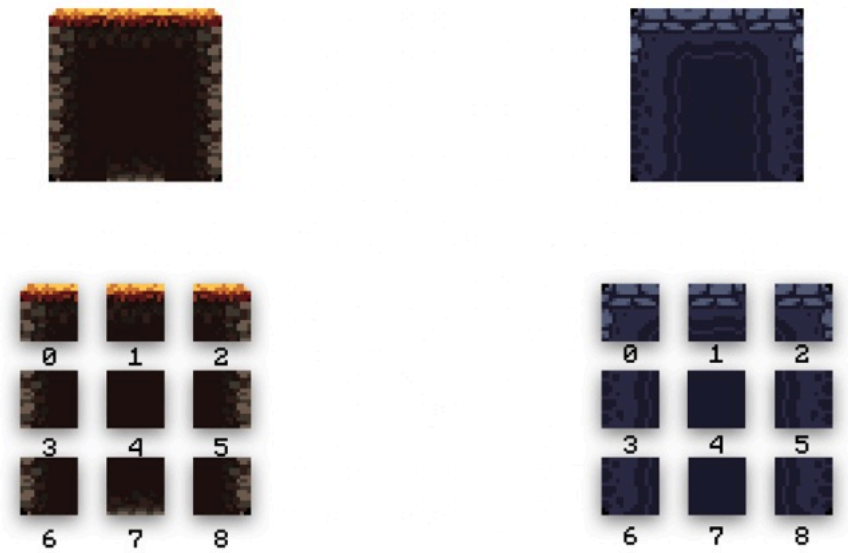
For my game, rather then using an external level editor such as LDtk, I am going to create my own level editor. Given that my level design is not overly complex, I believe that many of the advanced features in LDtk might not be useful. therefore I can make a minimal level editor myself.

This is going to be built on top of my game engine, so I don't have to rewrite all the rendering mechanisms myself, the physics gravity simulation stuff will go away but i will use make use of the tile systems.

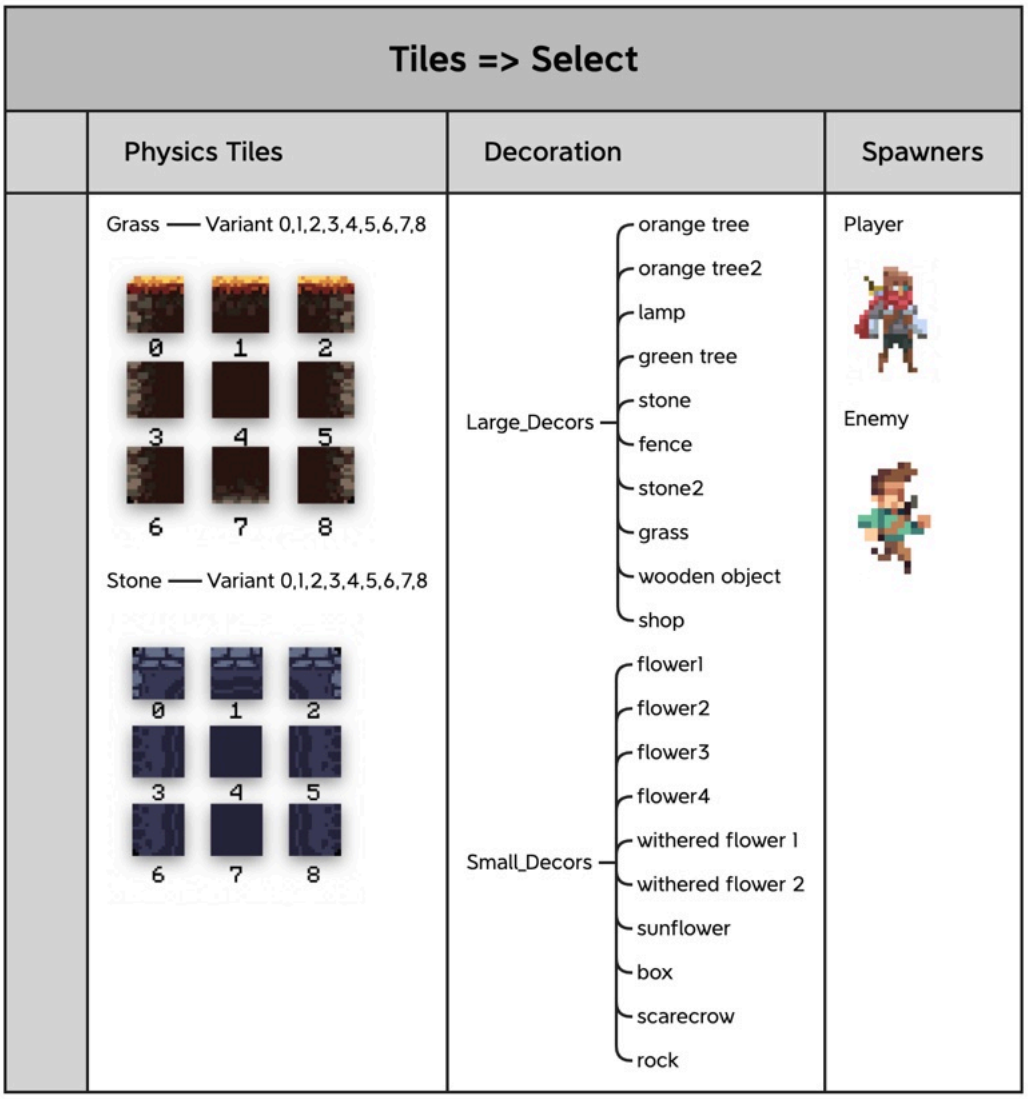
Before we start building the editor, let's review what assets do we have to work with.

decor_others

stone & grass tiles



So that is all of the tiles we have to work with. I am going to implement a menu system where the user can select a specific item.



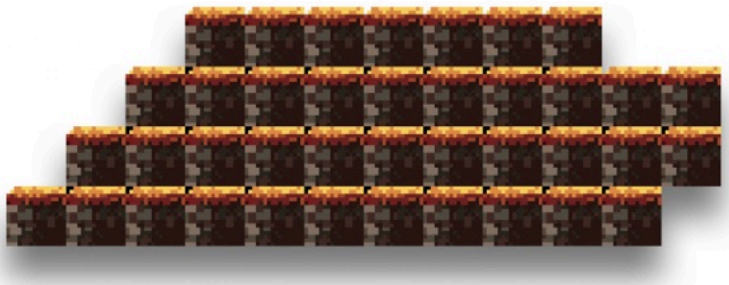
Now we have a categorized system, and we can use it in our level editor, in addition, we need to define some keybinds.

Key	Action	Description
	User Movement	It moves the user around to navigate the map

	Places a tile	By clicking the left mouse button, it places a tile onto the map
	Delete A tile	By clicking the right mouse button, it deletes a tile from the map
	Scroll through the tile categories	By scrolling the middle mouse button, we can change the different tile categories, namely, Physics tiles, Decorations, and Spawners.
 + 	Scroll through the tile variants	Inside the tile categories there are varients of the same tile type. For example, the grass and stone types each have 9 variants. I have included all the decorations under one big decor group for convenience.
	Output/Saves the map	When the key is pressed, the game saves the map into a .json format
	Toggles Offgrid/Ongrid	As explained earlier in the tilemap system, tiles can exist in absolute locations rather than using the coordinate system.
	Auto Tiling	It changes the variant of the tiles into the correct one using the autotiling algorithm, which will be explained in the next section

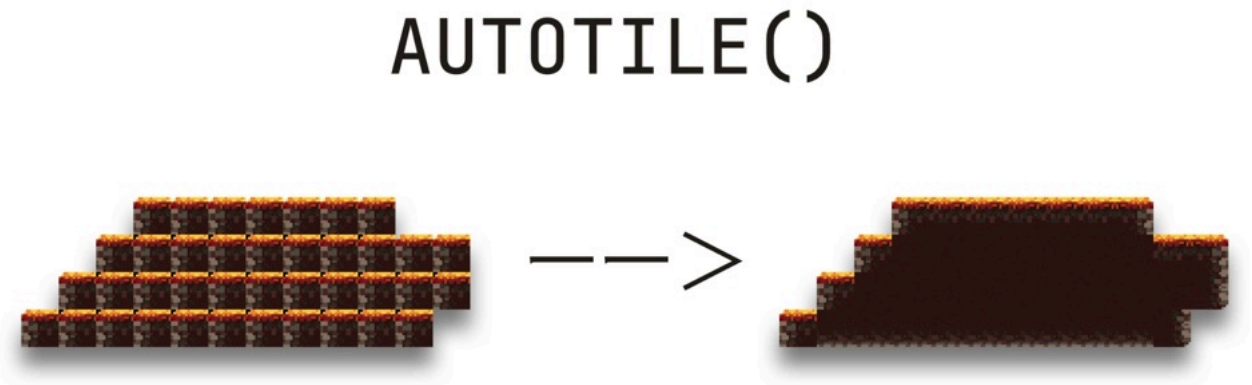
Autotiling (Algorithm)

Let’s say you are creating a new island on your map and you have created something that looks like this, looks a bit odd doesn’t it?



This is because the blocks are not obeying the laws of nature, for example, you won’t expect grass growing underneath a block of dirt, and this is why it look so wrong. Fortunately, tile variants are made to fix this, which are essentially different types of the same grass block. While we can fix every single block by hand but it would be tedious and take a long time.

So instead, we want to create an autotile function that fix the orientation of the tiles based on an algorithm, that fixes it for us, the effect is shown in the image blow.



The exact algorithm will be explained below in the implementation section, as it relies on data structures, which is hard to explain in words.

Sound FX & Music

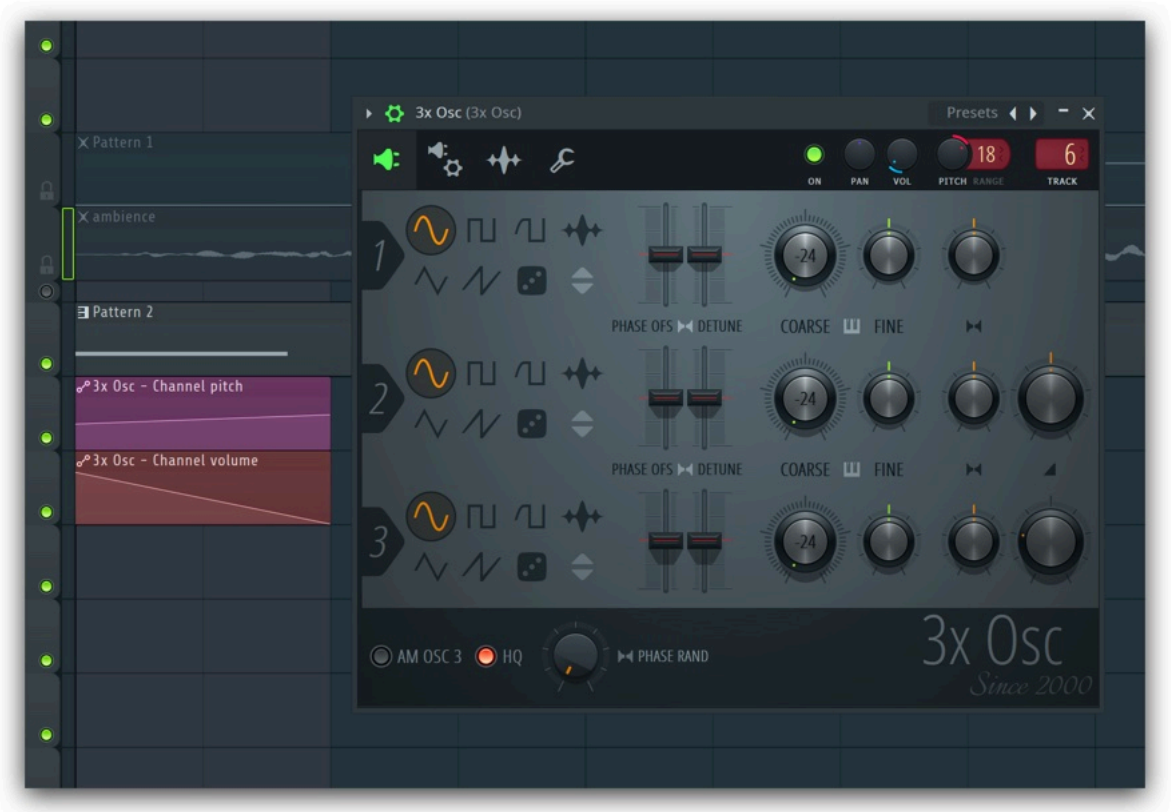
The game has no sound or auditory feedback now and we need to add a background music and SFXs to the game, to give the player a more immersive experience. I will be making my own sound effects as well as music from scratch in FL Studio.

For every user input or character movement, there has to be a sound.

Event	**What does that sound like?**
Player Jump.wav	A sine wave with pitch automation
Player Gets Hit.wav	Sound with high transient at the start
Player Dashes.wav	A swoosh sound, maybe automating volume a white noise?
Enemy shoots.wav	Bullet release sound
Toggle/Scroll through Menu.wav	Can be the same as the player Jump audio

I will be using the .wav format as it records raw audio information and it is universally compatible, it does not require a special decoder like MP3 does so it makes the compatibility better.

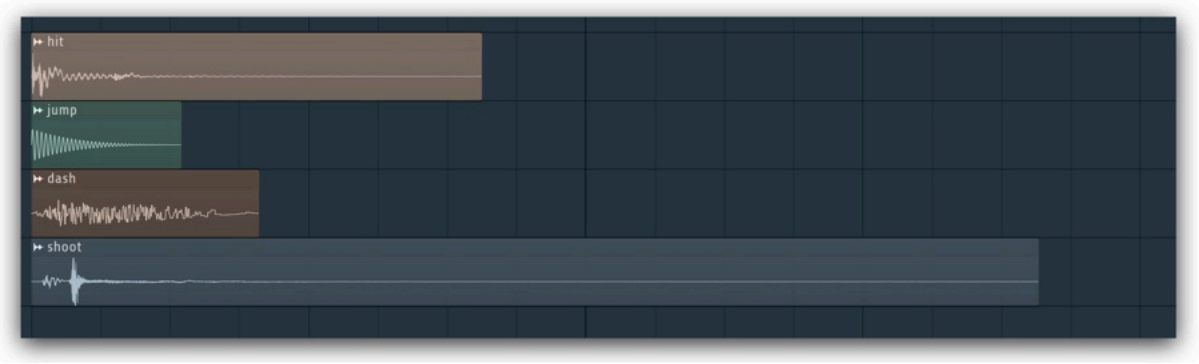
Jump Sound



The Jumping sound is made from a pure sine frequency and by automating down the pitch and volume, to create a smooth transition. It only uses one oscillator because I want to keep the sound simple.

As for the sound when the player gets hit, I used a combination of foleys and percussion hits, mixed with a delay effect. The sound has a pretty big transient which indicate the hit is sudden.

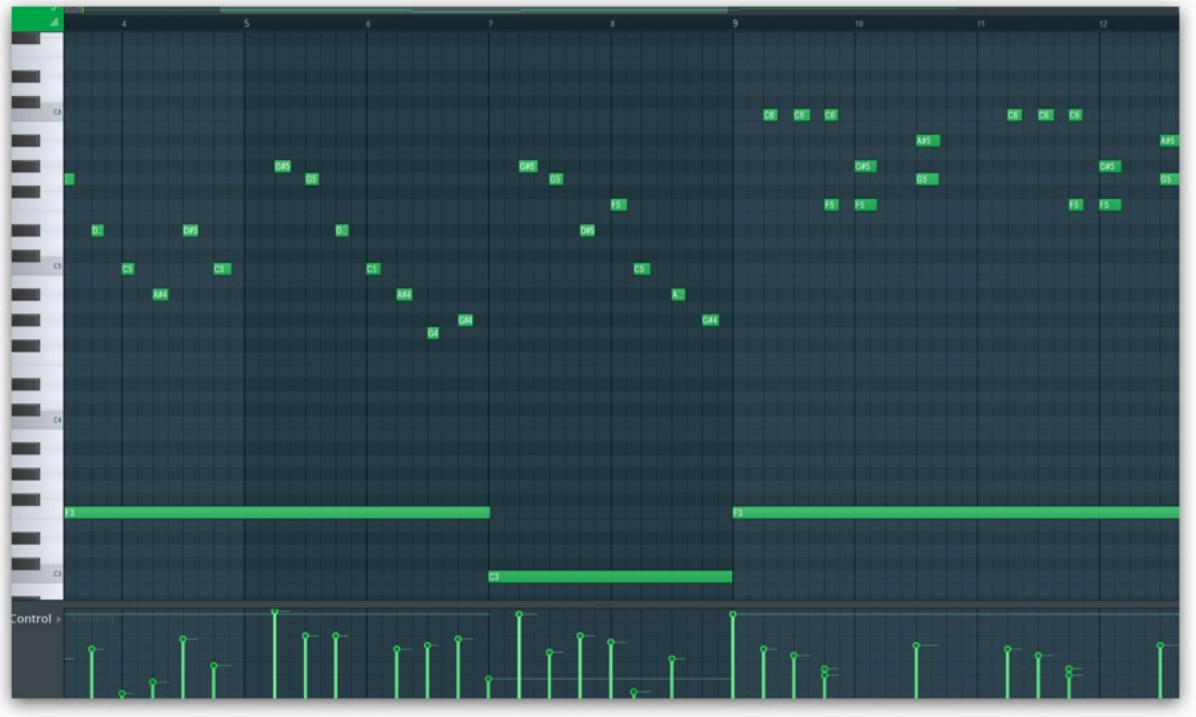
The other sound effects are made with a similar approach, so I won't show every one of them, here are four rendered sound effects in the WAV format, ready to be used in the game.

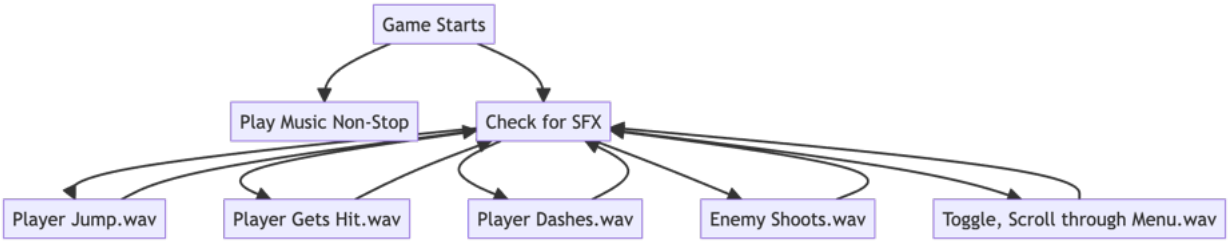


Now that the sound effects are finished, it's time to make the music, I listened to various game's BGM with a similar theme and decided which type of music best suits my game.

I discovered a preset called 'tabla dance' in Kontakt, and it has a mysterious quality that ideally suits the background music for my game.

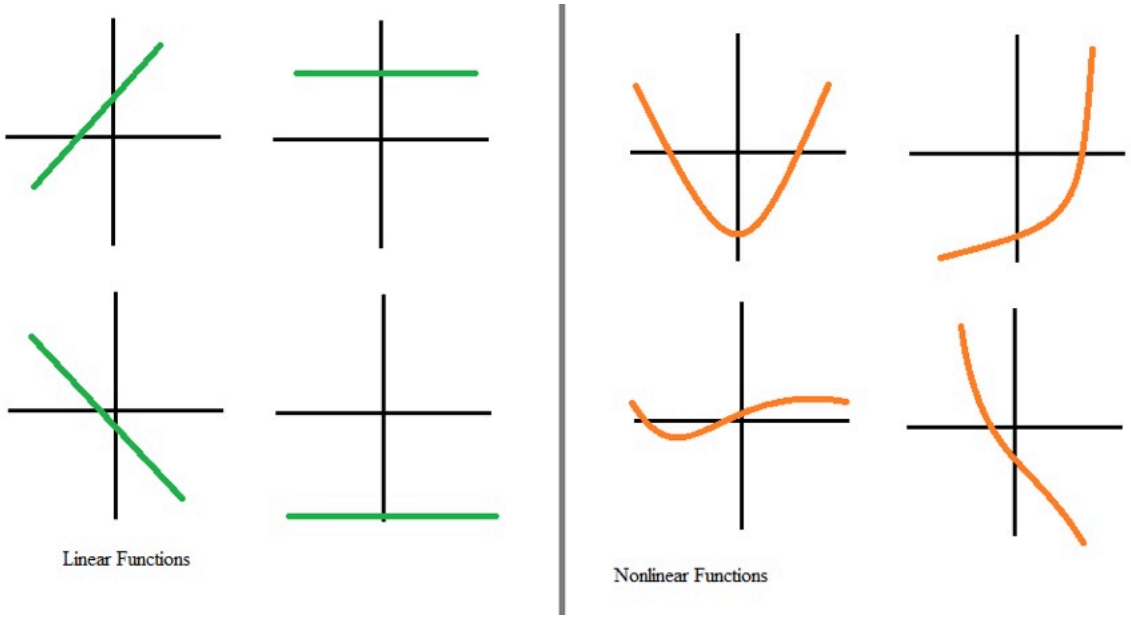
Then I used some melody arpeggios for more embellishment, making it more polished. It centres around the F minor tonality with the usage of Fmin7 chords and occasionally to the V chord of the key, which is C7, creating a V-I perfect cadence, indicating the end/beginning of a new section.




Game Starts
Play Music Non-Stop
Check for SFX
Player Jump.wav
Player Gets Hit.wav
Player Dashes.wav
Enemy Shoots.wav
Toggle, Scroll through Menu.wav

UI & Menus

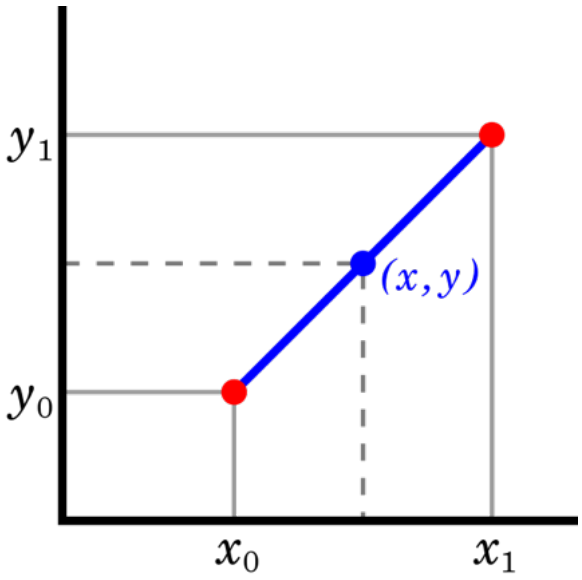
Bézier curves & Nonlinear Animation



Animations can be linear or non-linear. A linear animation means that the rate of change is constant over time. Non-linear animations, on the other hand, vary their rate of change, which can make them look more dynamic or natural. Linearity is a term borrowed from mathematics, which simply describe if a function is a straight line or not.

To make the game look more polished, I am going to add non-linear animation to all of my UI elements, such as in-game pause menu and transitions.

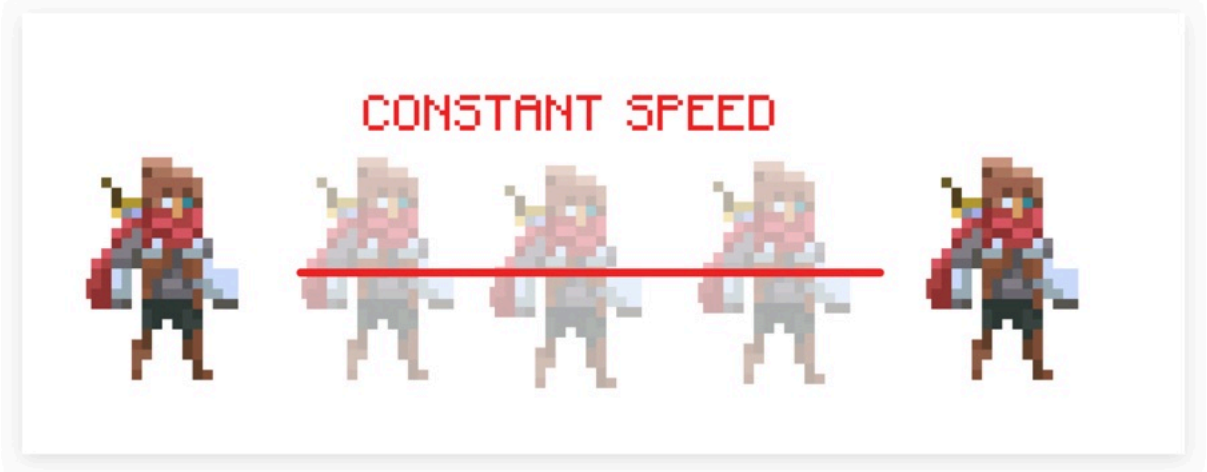
Normally when we want to move an object from a place to somewhere else, we would set a starting position, an ending position, and a duration. How would the computer know where the object is supposed to be for the frames in between? The answer would be linear interpolation.



The linear interpolation (lerp) function $L(x)$ can be defined as:
Given two points, (x_0, y_0) and (x_1, y_1), linear interpolation finds a value y at a given x within the interval $[x_0, x_1]$.

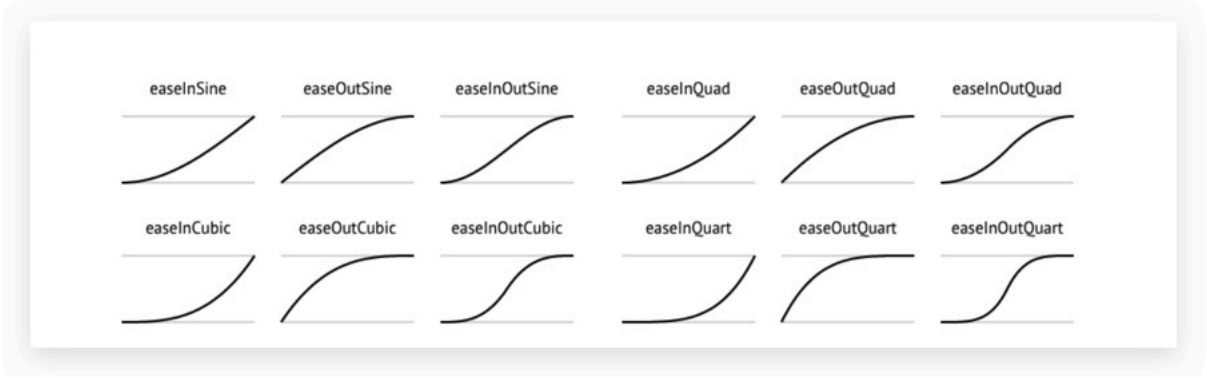
$$L(x) = y_0 + \frac{x - x_0}{x_1 - x_0}(y_1 - y_0)$$

For $x_0 \leq x \leq x_1$, $L(x)$ yields a value within the interval $[y_0, y_1]$, and the result is a linear mapping between the two points.



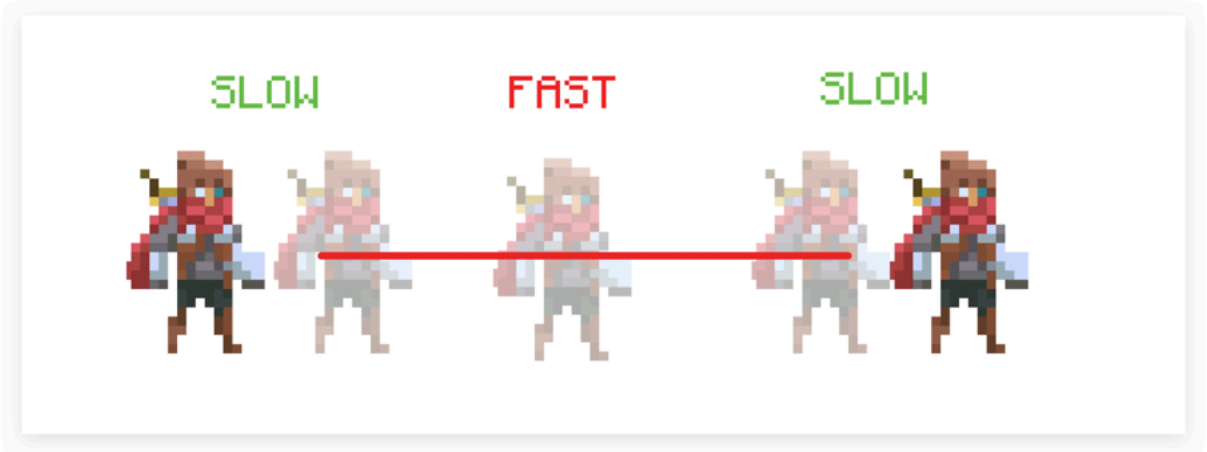
As a result, when lerp is applied, the object moves at a constant speed from start to end, which looks quite boring.

However, we do have tools such as easing functions and bezier curves to add non-linearality to the animation. Easing functions specify the speed of an animation at each point in time, making them accelerate or decelerate in specific ways.



Examples of Easing:
Ease-in: Starts slow and ends fast.
Ease-out: Starts fast and ends slow.
Ease-in-out: Starts slow, gets faster in the middle, and ends slow.
These easing functions can be represented using Bezier curves,

Non-linear animations are often preferred over linear ones because they tend to look more natural. Think of a bouncing ball – it doesn't move at a constant speed - it accelerates due to gravity and decelerates when it bounces. This non-linear motion is more realistically represented using non-linear animation.



I will use the quadratic easing out function in my game, to create a gradual slowing down effect.

```
FUNCTION ease_out_quad(t):
    return -t * (t - 2)
```

Pseudocode for rendering the non-linear animation: (Applies to all objects)

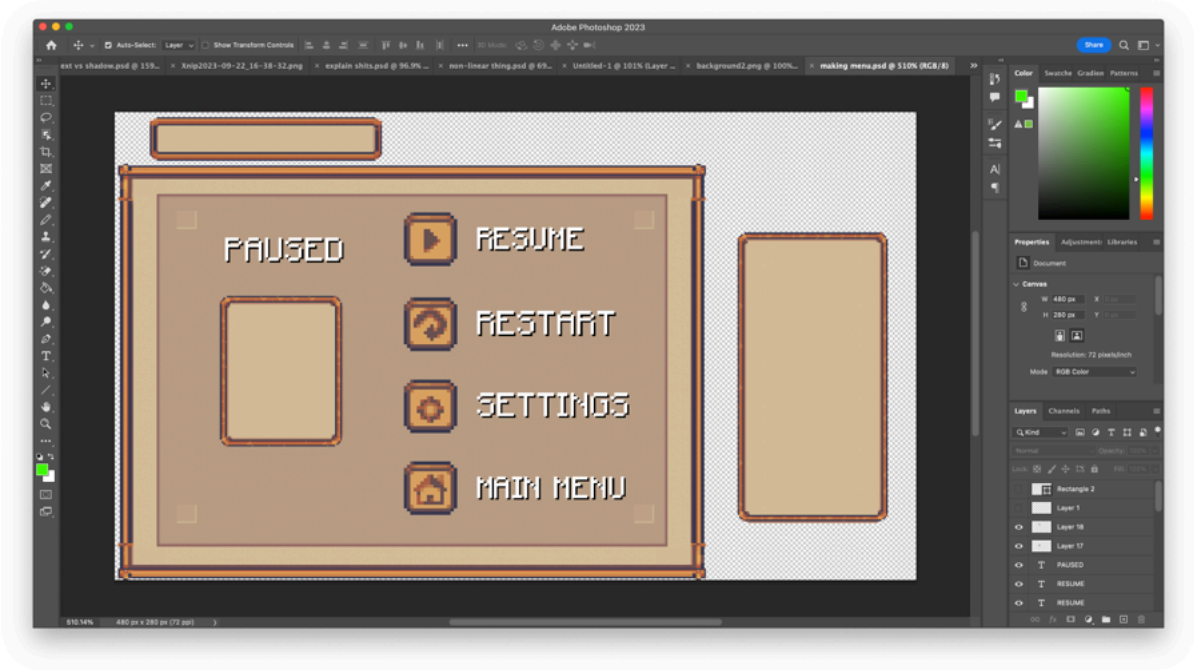
```
FUNCTION OBJECT_NON_LINEAR_RENDER:

    // Define the target position for the panel
    CONSTANT TARGET_POS
    CALCULATE difference as (TARGET_POS - object_current_POS)
    CALCULATE normalized_difference as (ABSOLUTE VALUE of difference)

    // Ensure normalized_difference stays within the range [0, 1]
    IF normalized_difference > 1 THEN
        SET normalized_difference TO 1
    ELSE IF normalized_difference < 0 THEN
        SET normalized_difference TO 0 END IF

    // Use easing function to compute progress
    progress = CALCULATE EasingFunction(normalized_difference)

    // Determine direction to adjust the end panel position
    IF difference is positive THEN
        INCREASE object_current_POS by (progress)
        IF object_current_POS exceeds TARGET_POS THEN
            SET object_current_POS TO TARGET_POS
        END IF
    ELSE
        DECREASE object_current_POS by (progress)
        IF object_current_POS is less than TARGET_POS THEN
            SET object_current_POS TO TARGET_POS
        END IF
    END IF

    // Calculate the position to render the game end panel
    render_position = (X_POSITION, object_current_POS)

    DISPLAY OBJECT at render_position

END FUNCTION
```

In-Game Menu

We are going to design the in-game menus first, it would be user to set the loop logic as it runs with in the main game loop. There will be two menus within the game loop, which are the pause menu and the player status panel.

Pause Menu

The pause menu is self explanatory, it pauses the game by temporality exiting the main game loop and redirecting to the pause menu panel. It is triggered by the [ESC] button on the keyboard.



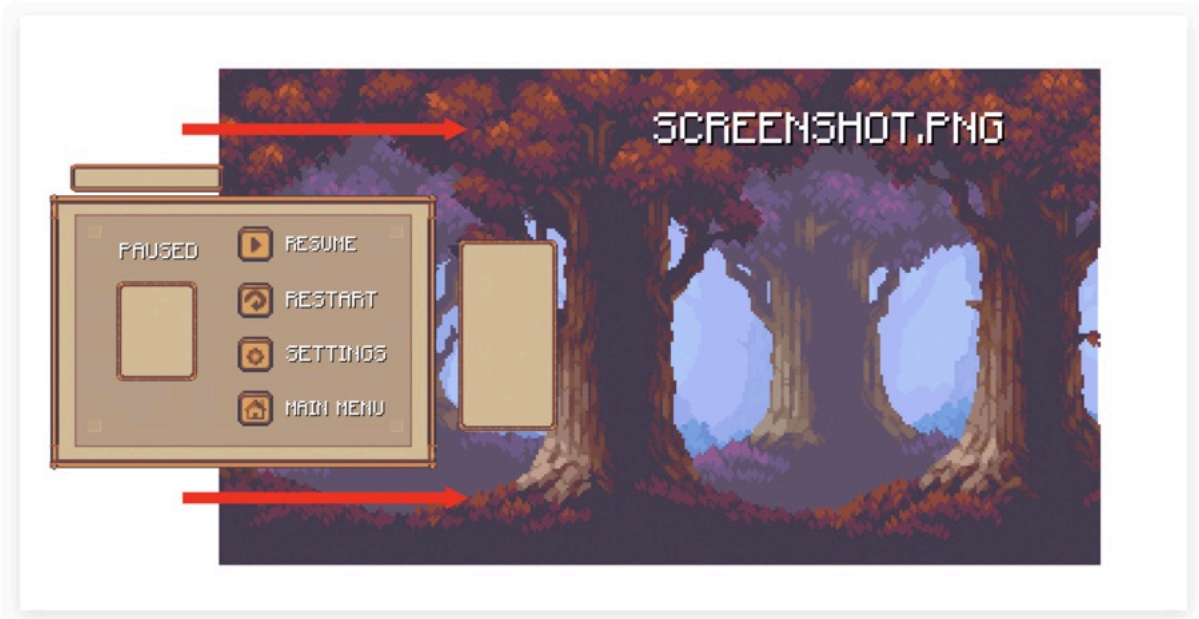
This is the design of the pause menu I made in photoshop, it has four buttons which are [RESUME], [RESTART], [SETTINGS] and [MAIN MENU].

BUTTON	**FUNCTION**
[RESUME]	Exits the menu and resume the main game loop
[RESTART]	Restarts **CURRENT** level while saving previous progress
[SETTINGS]	Triggers the debug menu by setting the show_debug flag to true.
[MAIN MENU]	Exit the game loop and reset the progress to 0, enters main menu loop

It has a box under the text ‘PAUSED’ which plays an animation of the character idling, which adds more life to the UI. On top there is another box which shows the name of the player, giving a more personalized touch to the game.

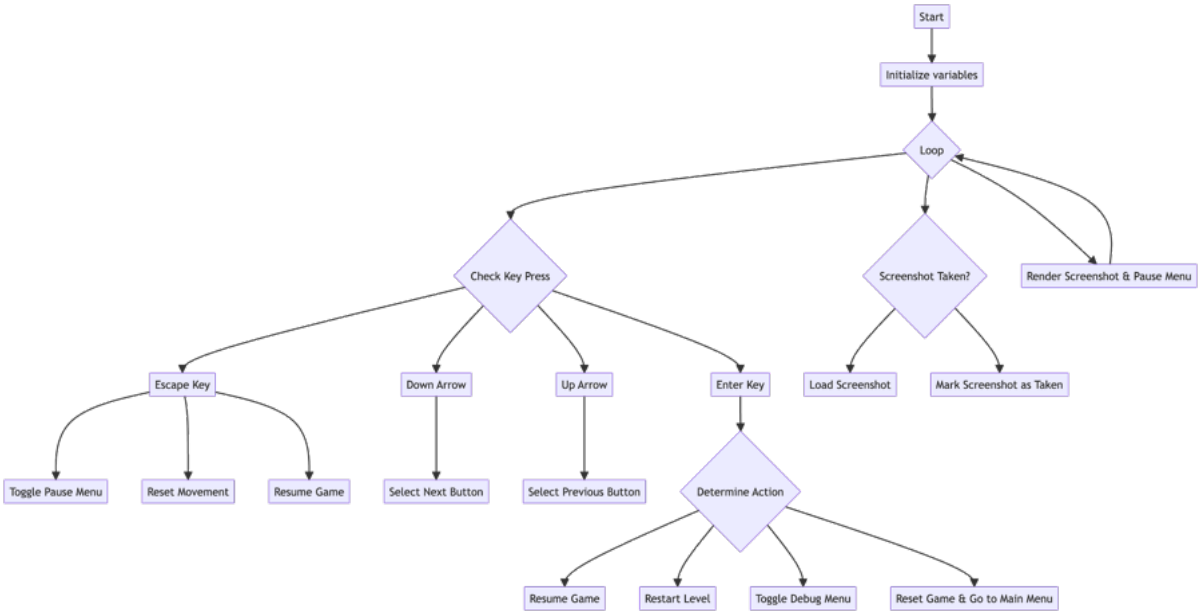
On the right we have another box which displays various player stats that is not displayed in the main game.

Stat Type	Description	IMAGE
Player Get Hit	For each bullet that hits the player, the variable is incremented	
Restart	Records the number of times the player has restarted any level.	



When the menu button is triggered, the menu slides from the left to the main game by using the smooth non-linear transition function we've talked about above.

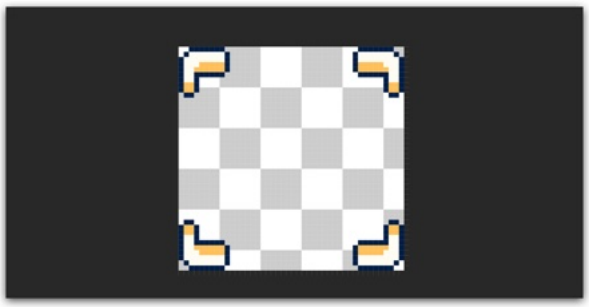
To display the game layer underneath the menu, I am going to use a trick, so instead of actually pausing the game, which means I need to pause the enemies, the physics simulation and the timer, this is a lot of work. Instead, I can just take a screenshot of the gameplay when it is paused and display the image as the background, which will save a lot of time while programming this feature.



Button Selector

My game uses keyboard input and so are the UIs, the user toggles around the menu with the up and down arrow keys and I will create a variable for the mouse selector to keep track of which button is selected.

Each button is going to be labelled with an ID and the index starts from 0, this is good because I can use some modular arithmetic to reset the selector when the selected value exceeds the number of buttons.



To make it more user-friendly, I will also add a button selector which sits on top of the buttons, indicating which button has been selected. It uses the mouse selector variable to calculate its position (as the buttons are equally spaced apart).

```
FUNCTION ingame_menu:

    SET screenshot_taken TO False
    SET button_selector TO 0
    SET menu_buttons TO ['Resume', 'Restart', 'Options', 'Main Menu']

    WHILE True DO:

            IF event IS KEY_PRESSED:
                IF key IS ESCAPE:
                    TOGGLE show_pause_menu
                    RESET movement
                    CALL main_game

                IF key IS DOWN_ARROW:
                    INCREMENT button_selector
                    MODULO button_selector BY length of menu_buttons

                IF key IS UP_ARROW:
                    DECREMENT button_selector
                    IF button_selector IS LESS THAN 0:
                        SET button_selector TO last index of menu_buttons
                    MODULO button_selector BY length of menu_buttons

                IF key IS ENTER:
                    IF selected_button IS "Resume":
                        HIDE pause_menu
                        RESET movement
                        CALL main_game
                    ELSE IF selected_button IS "Restart":
                        CALL restart_level
                    ELSE IF selected_button IS "Options":
                        TOGGLE debug_menu

                    ELSE IF selected_button IS "Main Menu":
                        RESET entire_game
                        CALL main_menu

        IF screenshot NOT taken:
            LOAD screenshot

            screenshot
            SET screenshot_taken TO True

        RENDER screenshot on display
        RENDER pause_menu

END FUNCTION
```

Timer & Player Status UI

Item	Description	Image
Ability CD visualizer	Calculates the CD of player abilities, such as dash and super Jump	
Timer	Records the time spent in the current level, as well as the total cumulative time across the entire run.	
Enemy Count	The len(enemies) variable tracks the number of remaining enemies, providing the player with an indicator of how many enemies still need to be defeated	

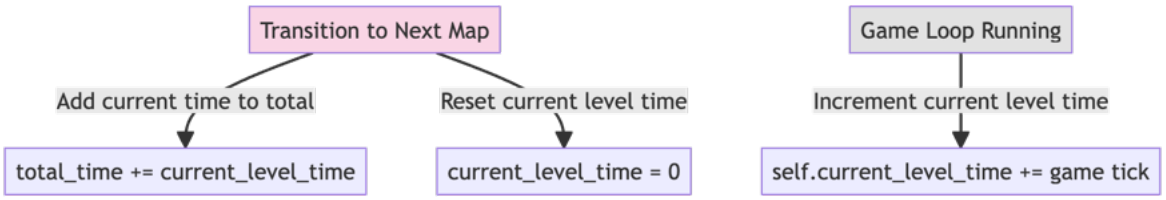
Player Health	Displays the Player HP using the number of hearts rather than a number text, which is hard to read during a moving battle.	
Level Tracker	Displays the level number in the bottom right corner, allowing the player to know their current progression	

The timer initializes 2 variables when the game starts: current_level_time and global_total_time. Each time we move onto the next level, current_level_time gets recorded by the global time and is reset. The timer records time in milliseconds, which need to be converted and rounded into minutes: seconds format using some modulo arithmetic.

```
Conversion of Milliseconds to Seconds:
total_seconds = INT(global_total_time / 1000)

Extraction of Minutes from Total Seconds:
minutes = INT(global_total_time / 60)

Extraction of Remaining Seconds:
seconds = total_seconds % 60 The final time is then represented as:
minutes: seconds
```

whenever it transitions to the next MAP:

```
total_time += current_level_time
current_level_time = 0
```



Player Score Calculation

To enhance the game's competitive edge, I plan to introduce a performance-based scoring system. Instead of traditional numerical scores, this system will measure time, incorporating penalties for errors. This approach could even pave the way for speedrun challenges.

As previously detailed, the game records the global time upon successful completion. The final time will be derived from a range of data, factoring in time penalties for mistakes such as taking damage from enemies or restarting the game.

Time Penalty:

Penalty Type	**Description**	**Time Added**
Player Get Hit	For each bullet that hits the player, a penalty is added.	**1s**
Restart	To stop the player from manipulating the time and create the best run by restarting the level, a time penalty is added each time the level restarts, so the player need to try their best to complete the 7 levels in one go.	**5s**

Therefore the final time can be calculated with this formula:

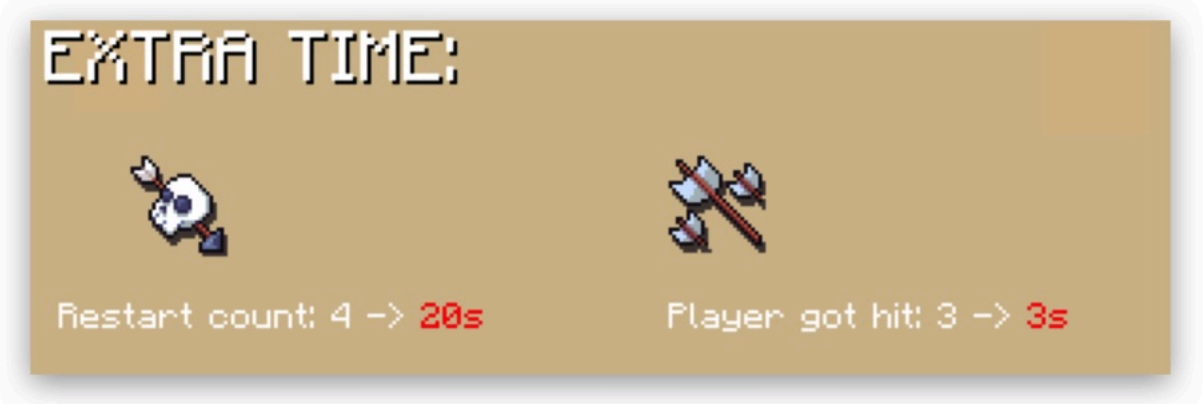
```
FINAL_TIME = RAW_TIME + PLAYER_GET_HIT * 1 + RESTART_COUNT * 5
```

End Game Panel

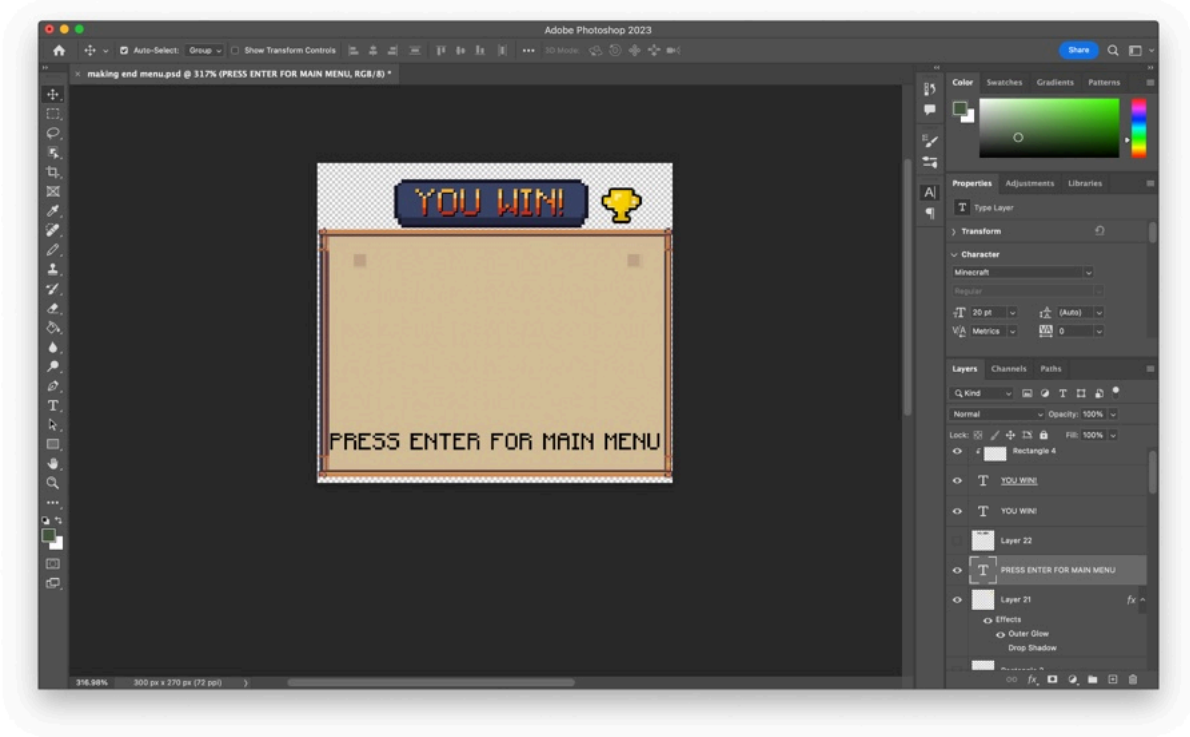
The Game is lacking a transition between the end of the game and the main menu so I will have to add that.

I am planning to add a fruit-ninja inspired game panel, so when all the enemies are eliminated and the game ends, a panel slides in from above and show the player the relevant stats. The bonuses/penalties are added incrementally in order to create anticipation for the player. Which leads to a final score at the end.

As detailed in the previous section, my game calculates the total time by combining the actual time taken and any time penalties. Beneath the image, there will be a text description indicating the category of the time penalty.

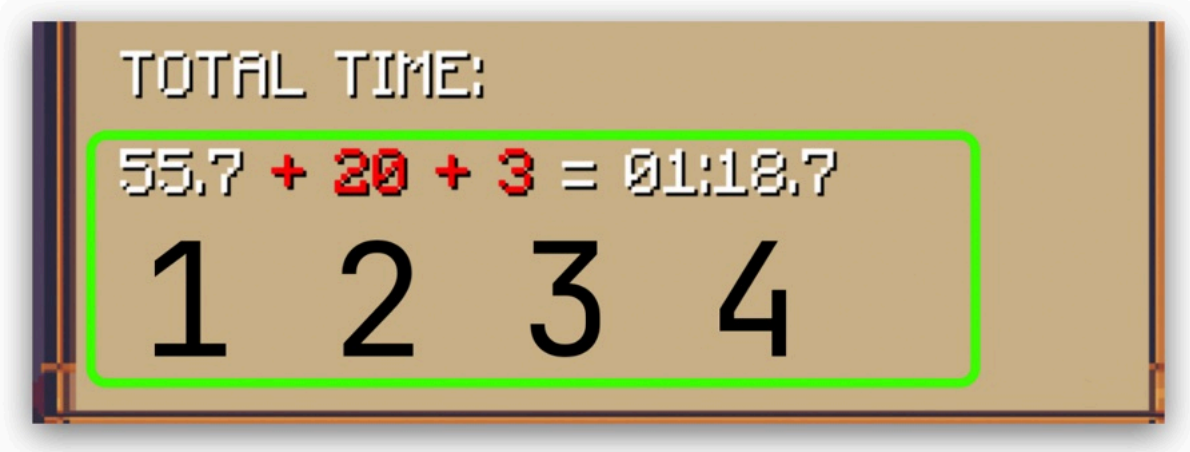


The time penalty will be highlighted in red so it is easier for the player to see.



The photoshop design of the end game panel includes a text that says ‘YOU WIN!’ in vibrant, red and orange colour, together with a trophy on the right.

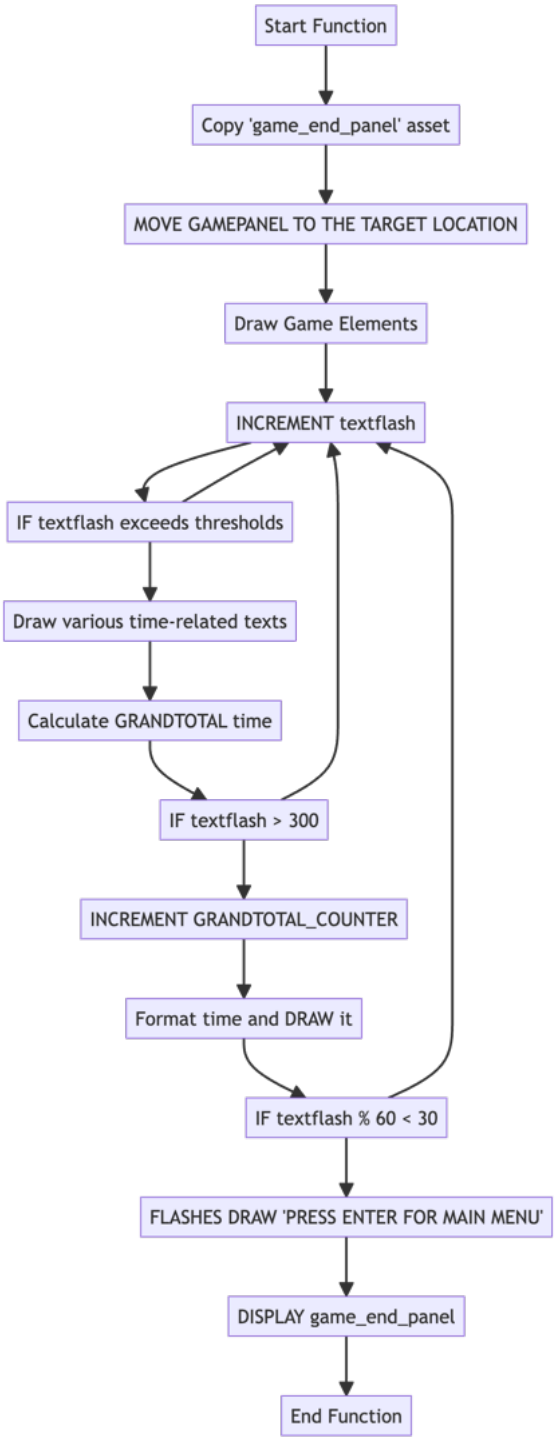
I will also add a flashing text on the bottom that says ‘PRESS ENTER FOR MAIN MENU’ so imply it requires user action.



To build anticipation, the total time will be unveiled incrementally, culminating in a timer that starts from 00:00:00. When the game panel appears, a timer runs and when the timer hits a certain value, the time segment is revealed.

```
FUNCTION render_end_game_panel:

    COPY 'game_end_panel' asset to game_end_panel


    MOVE GAMEPANEL TO THE TARGET LOCATION ease_out_quad function on
```

```
distance_difference

    DRAW 'EXTRA TIME:' text on game_end_panel
    DRAW 'Player got hit' text and count on game_end_panel
    DRAW arrow image on game_end_panel
    DRAW 'Restart count' text and count on game_end_panel
    DRAW 'skeleton' image on game_end_panel
    DRAW 'TOTAL TIME:' text on game_end_panel

    INCREMENT textflash

    IF textflash exceeds certain thresholds:
        DRAW various time-related texts on game_end_panel
    ENDIF

    CALCULATE GRANDTOTAL time
    IF textflash exceeds 300:
        INCREMENT GRANDTOTAL_COUNTER based on conditions
        FORMAT time and DRAW it on game_end_panel
    ENDIF

    IF textflash % 60 < 30:
        FLASHES DRAW 'PRESS ENTER FOR MAIN MENU' text on game_end_panel
    ENDIF

    DISPLAY game_end_panel on display at render_pos
END FUNCTION
```


Start Function
Copy 'game_end_panel' asset
MOVE GAMEPANEL TO THE TARGET LOCATION
Draw Game Elements
INCREMENT textflash
IF textflash exceeds thresholds
Draw various time-related texts
Calculate GRANDTOTAL time
IF textflash > 300
INCREMENT GRANDTOTAL_COUNTER
Format time and DRAW it
IF textflash % 60 < 30
FLASHES DRAW 'PRESS ENTER FOR MAIN MENU'
DISPLAY game_end_panel
End Function

1. the following is a text explanation of the subroutine.
2. Copy 'game_end_panel' asset: The asset named 'game_end_panel' is copied to a variable called game_end_panel.

3. Move game panel to the target location: The game panel is moved to its target position. The movement is smoothed out using the ease_out_quad function, which takes the difference between the current position and the target position as an input.

4. Draw Game Elements: Several game elements are drawn on the game_end_panel:
- EXTRA TIME text
- Player got hit text along with the count of times the player got hit
- An arrow image
- Restart count text along with the count of restarts
- A skeleton image
- TOTAL TIME: text

5. Increment textflash: The textflash variable is incremented. This variable seems to be used as a timer or counter to determine when certain actions should take place.

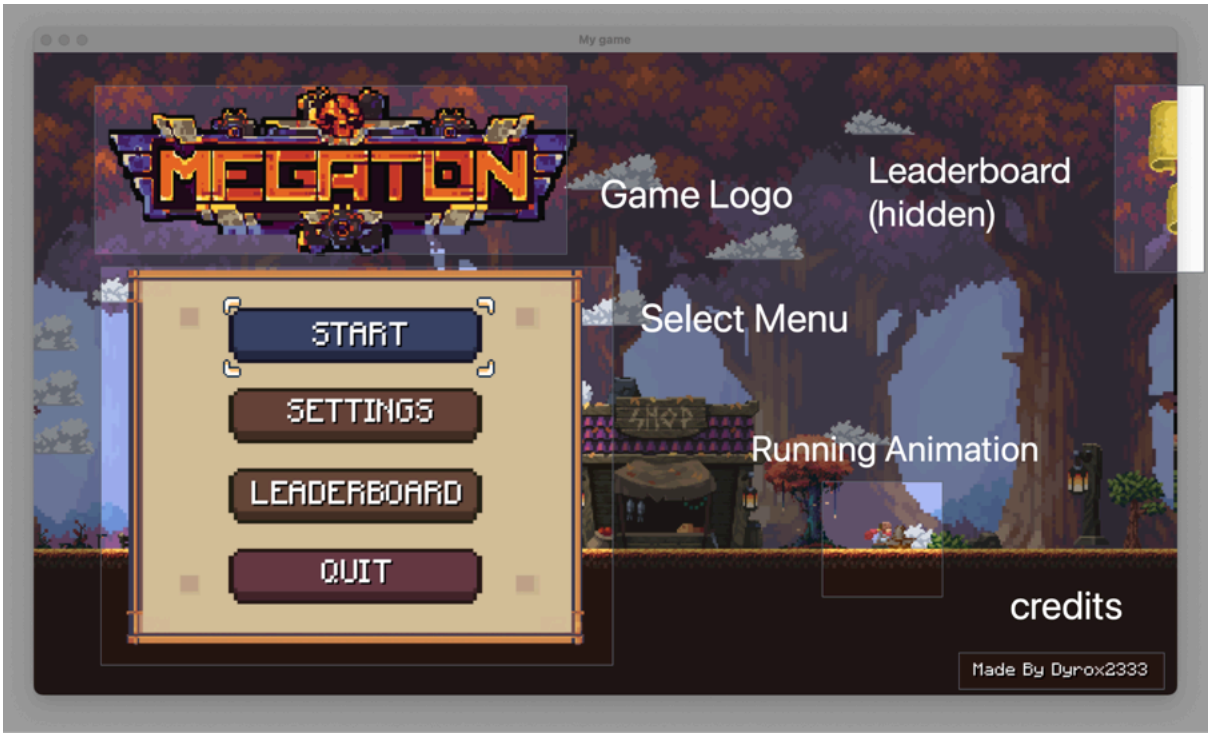
6. IF textflash exceeds thresholds: If the textflash value crosses certain predefined thresholds, various time-related texts are drawn on the game_end_panel.

7. Calculate GRANDTOTAL time: The total time, GRANDTOTAL, is calculated. This seems to be a combination of the actual game time and any penalties or bonuses.

8. IF textflash > 300: If the textflash value exceeds 300:
- The GRANDTOTAL_COUNTER is incremented based on certain conditions.
- The total time is then formatted and drawn on the game_end_panel.

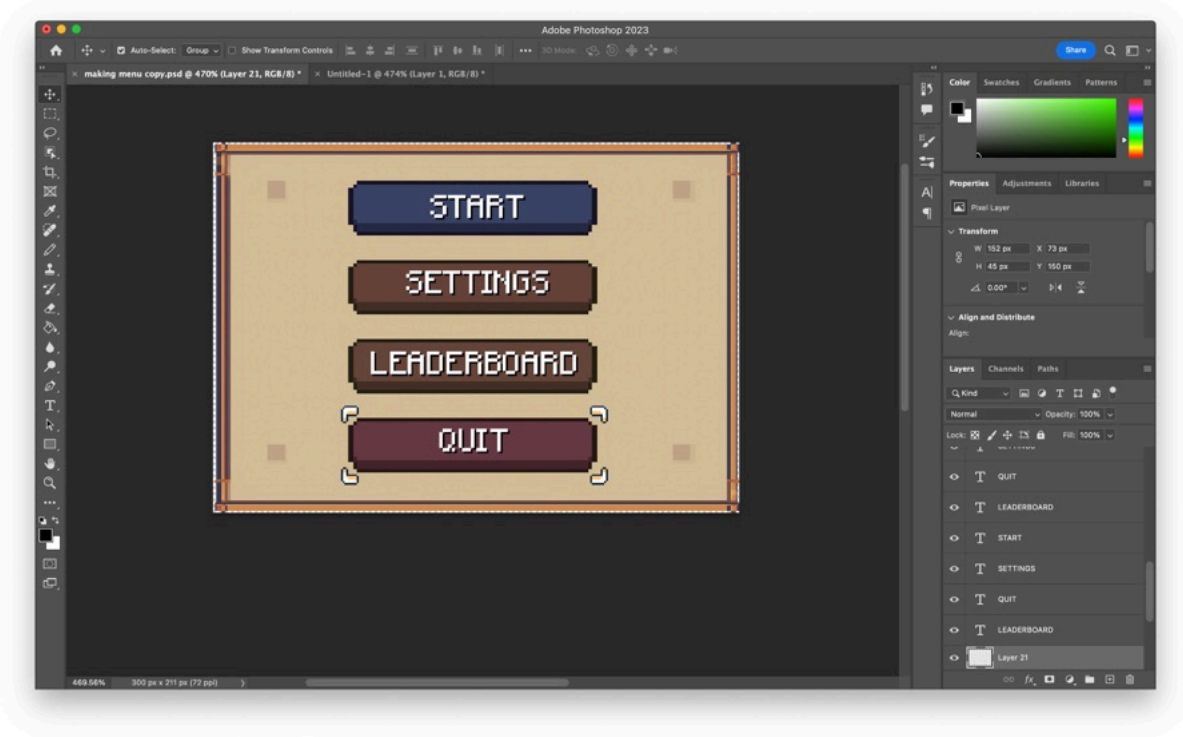
9. IF textflash % 60 < 30: If the remainder of textflash divided by 60 is less than 30:
- The text 'PRESS ENTER FOR MAIN MENU' flashes on the game_end_panel.
10. DISPLAY game_end_panel: Finally, the game_end_panel is displayed on the main game display at the specified render position.

Main Screen



The main screen serves the entry point to the game and the user is greeted with the main screen. There will be four main components in the main screen:

1. Graphics and logo that presents the game, preferably with animations.
2. The main menu that serves as the entry point to gameplay, which uses the start button
3. A leaderboard which the player can see past player high scores.
4. Various utility functions such as quit game or toggle settings.



This is four functional buttons on the main screen, relatively straightforward.

```
IF PRESS START:
        RUN MAIN_GAME()
IF PRESS SETTINGS:
        TOGGLE DEBUG_MENU()
IF HOVER LEADERBOARD:
        SHOW LEADERBOARD()
IF PRESS QUIT:
        QUIT GAME()
```

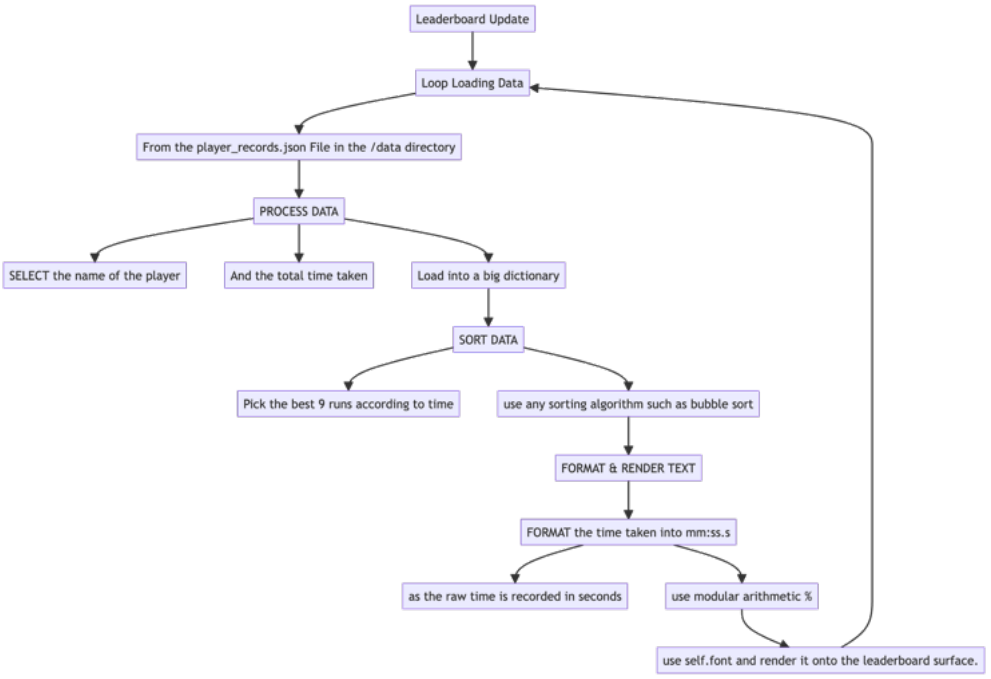
Next, we need to add the leaderboard.

Leaderboard

This is a prototype of a leaderboard design I created in Photoshop, featuring a dark-green colour palette reminiscent of a forest's mysterious ambience. It consists of a centred text that says ‘LEADERBOARDS’ and ‘BEST SCORES’, sitting on top of a scroll, reinforcing the mystery vibe. It has a total of nine empty spots, distinguished alternately by light and dark green colours. The leaderboard acts like its own surface and will be rendered onto the main screen.

Now, in order to display the best scores onto the screen, we need another algorithm to format and sort the data as required. We load the raw player_records.json data into a dictionary and we need to perform the following steps.

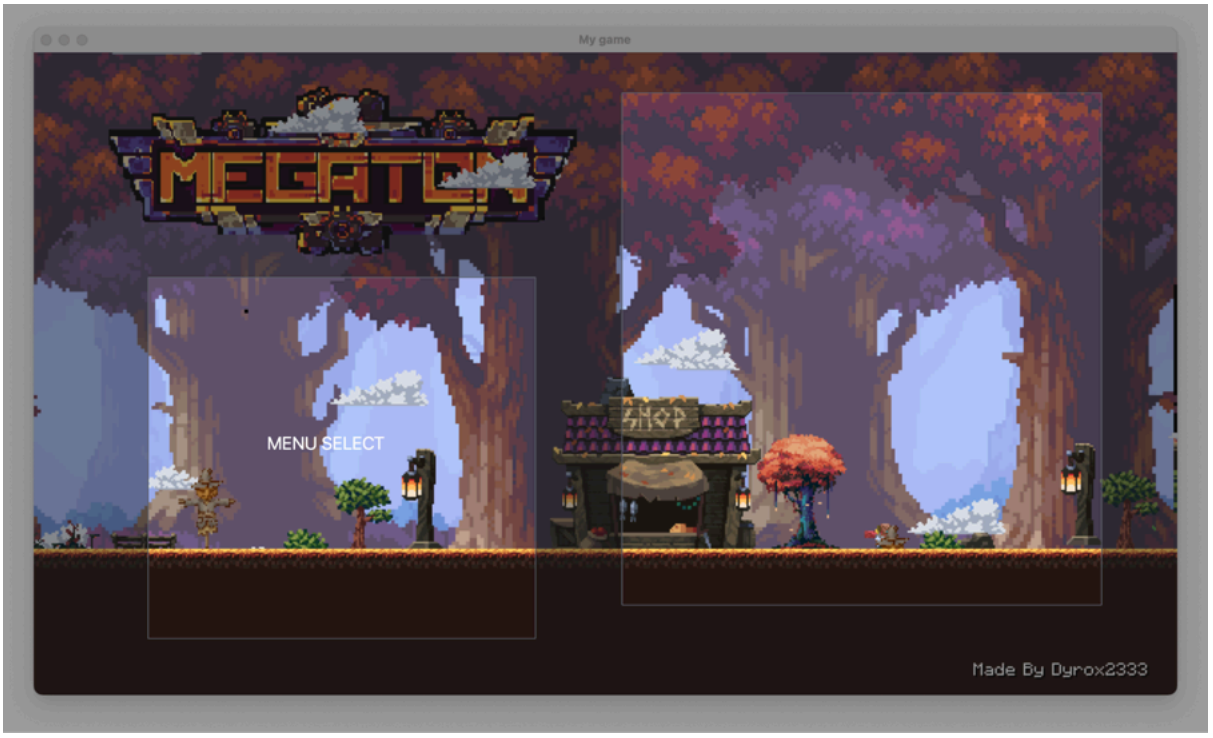
```
1. LOAD all data from player_records.json
2. BIND the time data to the player name
3. SORT the data by time, ascending
4. SELECT the top 9 players records
5. FORMAT the time data to be in minutes and seconds
6. DISPLAY the data in a table
```


Leaderboard Update
Loop Loading Data
From the player_records.json File in the /data directory
PROCESS DATA
SELECT the name of the player
And the total time taken
Load into a big dictionary
SORT DATA
Pick the best 9 runs according to time
use any sorting algorithm such as bubble sort
FORMAT & RENDER TEXT
FORMAT the time taken into mm:ss.s
as the raw time is recorded in seconds
use modular arithmetic %
use self.font and render it onto the leaderboard surface.


LEADERBOARDS
Player Records
(rendered indepentantly)
1. Dyrox2 2:31.3
2. Dyrox3 2:48.3
3. Dyrox 2:52.6
4. 6lits6 3:54.4
5. mjanmarski_davez_2 4:02.1
6. warcomong 4:27.7
7. paris man 4:35.4
8. rshen 5:20.5
9. mjanmarski_davez 5:47.7

Background Scrolling Animation

As we can see on the image, there is a player character running in the background and is running towards the right, on a seemingly endless road. This is a special level in our level map named main_menu.json. It is basically just a M by N rectangular grass block collection with some decorations on the surface, such as houses, trees and fences. I have used a trick to make this work, which will be explained in the implementation section.



Implementation

Setting up the development environment

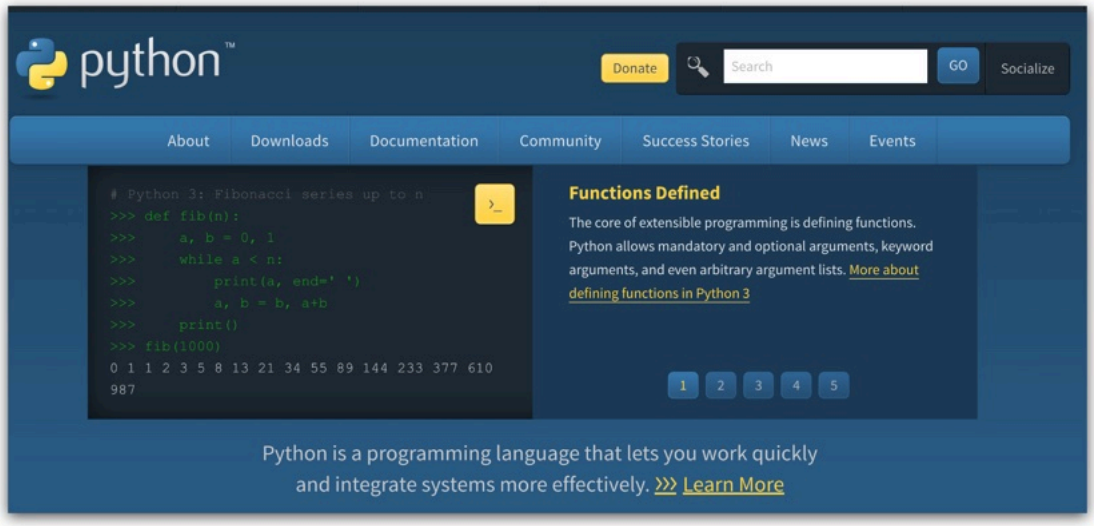


fig.29 Screenshot of the Python homepage

Now, it's time to set up my project, as explained in earlier chapters, I'll be using the Python programming language. First, I'll make sure Python is installed on my computer. Once that's done, I can use package managers like pip to install Pygame and any other libraries I might need. This will give me all the necessary resources to build my game.

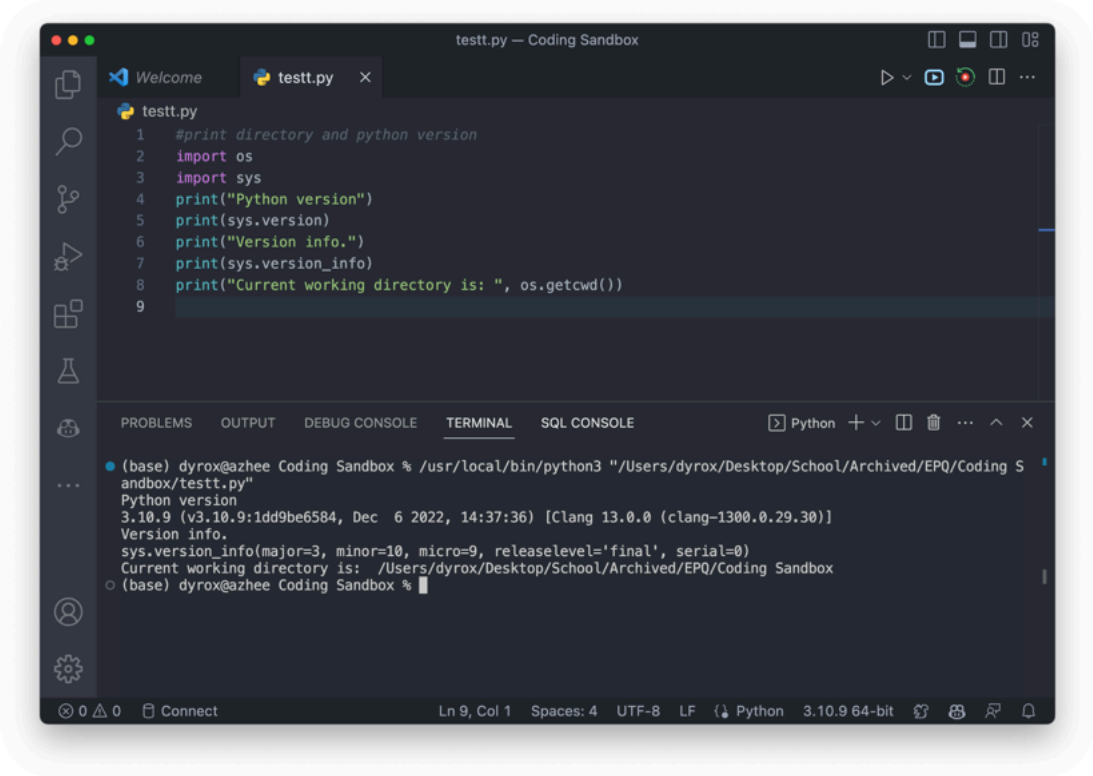


fig.30 Code that outputs the current version of Python

I'll use a user-friendly integrated programming environment (IDE), such as Visual Studio Code. The programming process is streamlined by these tools' useful capabilities including code autocompletion and debugging.

Version conflicts | Virtual Environment used

There is one problem though, and it’s the version conflict between my python projects. I have some other project that use a specific version of Python, and I wanna ruin that. It could be solved quite easily with virtual environments, which means I can have multiple Python environments on my system simultaneously and this is very convenient meaning I do not have to uninstall the previous version of Python when I install a new one. We can picture virtual environments as containers with each of its contents being one specific version of Python and specific versions of libraries.

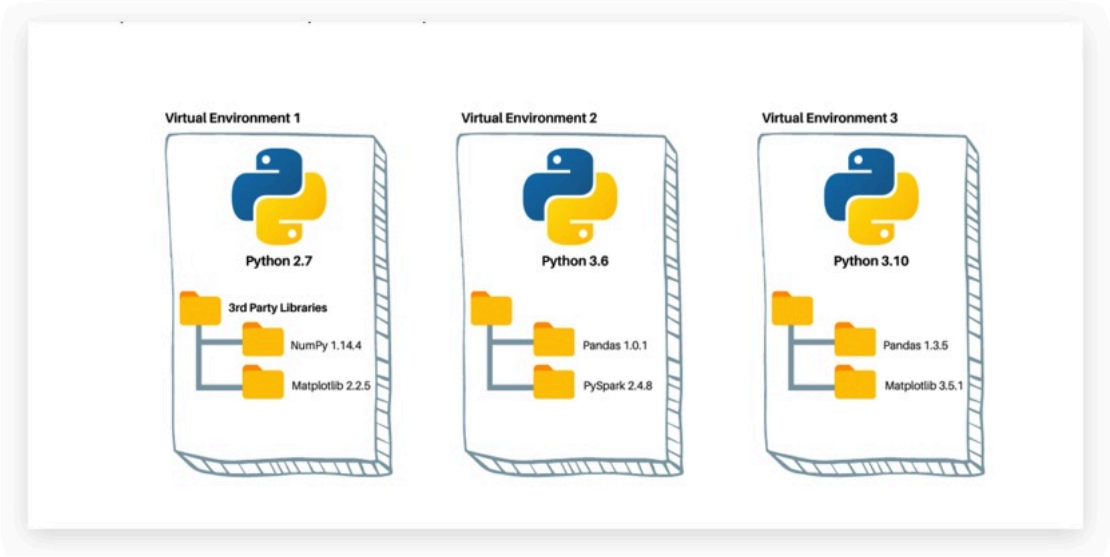


fig.32 Abstracted diagram showing different virtual Enviroments

A solution to this is the Anaconda virtual environment, alongside that, we also get conda, which is an open-source package management system and environment management system, which means in addition to pip, which is the default python package-management system, we also get to install software that is not written in python, such as FFmpeg, which is used to encode/decode media files and Libsndfile which is widely used reading and writing audio files. Note that they are both written in the programming language C, not Python so they cannot be installed with pip, using pip and conda together can make a much smoother user experience.

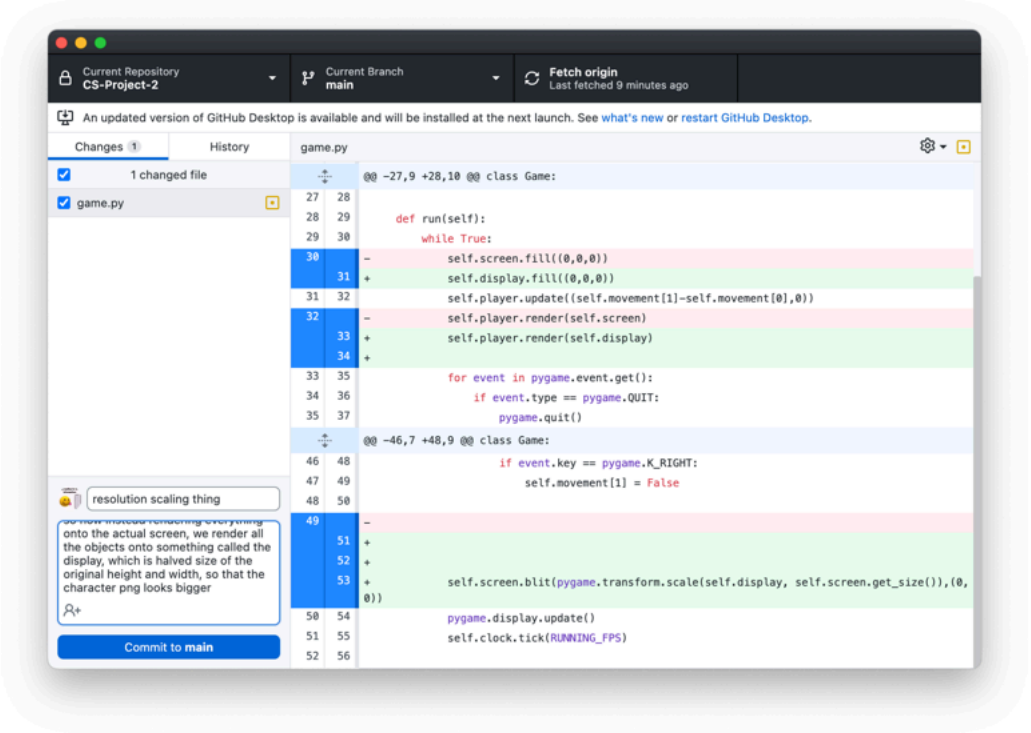
fig.33 Setting up a virtual environment in the system terminal

Note that in the image, the libraries have directories ending in ‘osx-arm64’, this means my operating system is MacOS and has the processor architecture arm64, the conda package manager is able to successfully identify my computer environment and allocate the corresponding packages, which saves a lot of trouble - because once the problem is compiled for one specific architecture, the same compiled machine code cannot be run other machines with a different architecture.

I will quickly explain how the architectures differ and how they are going to impact my workflow. The new Apple M1 chips are using the arm64 architecture and they are RISC processors (which stands for reduced instruction set computer), they have a lot fewer instructions in their assembly code but they are executed much faster, as one instruction only takes one CPU clock cycle to complete.

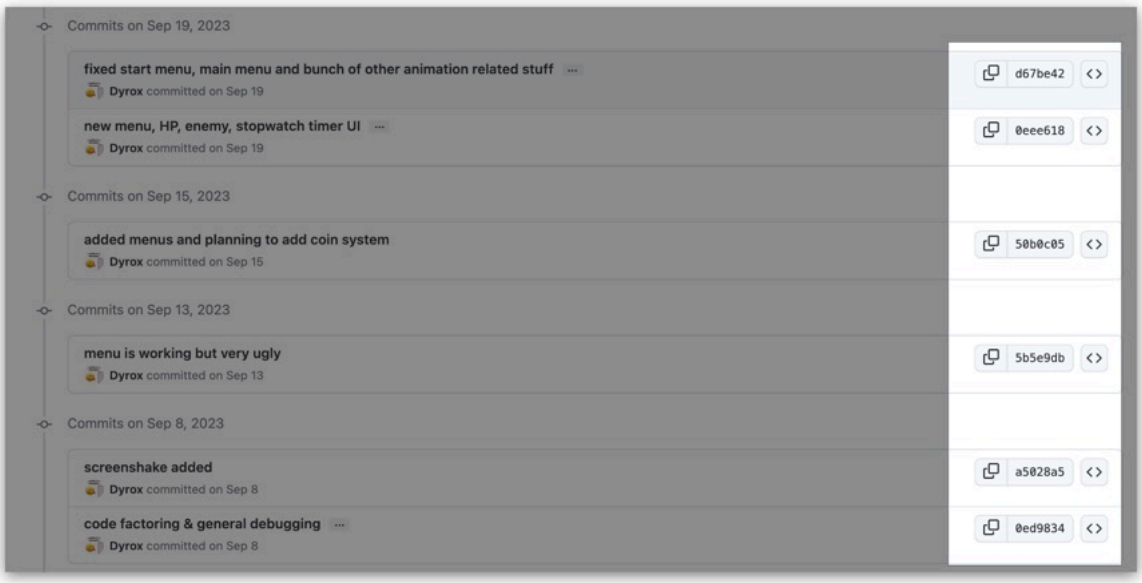
However, older software usually has only x86 compiled machine code as the market is dominated by Intel and AMD, which uses CISC processors (Complex instruction set computers), we can use several approaches to emulate the x86 environment such as virtual machines and dynamic binary translators such as Apple’s Rosetta 2.

Using Git (Version control)



After creating my repository, I realised it is important to keep track of the changes to my code. Using version control tools like git and Github allows me to backup and keep track of the current progress. The changes are recorded and the contents are labelled with colours, red means the code deleted and green means the code added. By doing this, we can ensure that the code is always up to date and we can always go back to the previous version if we need to.

Each version is called a commit, and each commit has a unique ID. We can use the ID to go back to the previous version, in case there is a bug.



Implementing the game

As the Pygame library is built on top of the Simple DirectMedia Layer (SDL) library, it comes with minimum basic features for building a game. Unlike other game engines like Unity or Unreal Engine. Pygame does not have advanced functions such as animation systems, menu designers, physics simulation or even support for 3D. Everything has to be written from scratch in 2D, and this is a big challenge.

In the beginning, I did think about making a 3D game but quickly realized it was not really feasible, the most basic 3D rendering requires the use of rotation matrices and trigonometric functions hundreds of times a second, which is very expensive to compute - and we are not even talking about shaders and lights, this is merely for rendering the geometry and vertices. There is indeed a way to compute this in parallel with GPU acceleration but this is quite complex and requires the use of the C++ programming language, and would make optimizing the game a pain. Consequently, the graphics performance will suffer, making it more suitable for me to focus on 2D games.

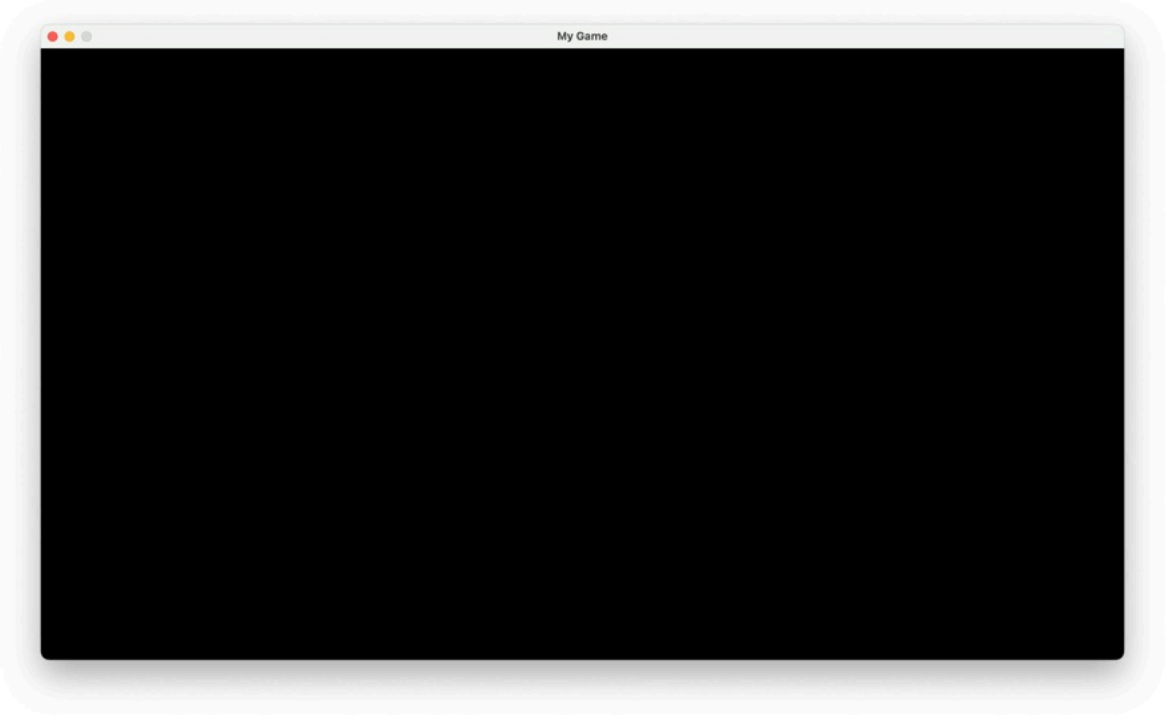
Prototype 1: Basic Game Engine

Boilerplate code + basic functions

```python
import sys

import pygame
FPS = 60

class Game:
    def __init__(self):
        pygame.init()

        pygame.display.set_caption('my game game')
        self.screen = pygame.display.set_mode((1280,720))

        self.clock = pygame.time.Clock()

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

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

Game().run()
```

To create my game instance, I will need some boilerplate code, which are the necessary initialization functions to create a game window. It creates a window under a certain resolution and we define a Clock running at a constant FPS, which refreshes the screen that number of times a second, also we need to put everything inside of a loop so it runs indefinitely.

Notice that I have put everything inside of a class so it eliminates the scope problem, which means all the components of my game can use the same variable throughout the whole game, so that I don't have to use the global keyword inside of every function.



An empty display is rendered.

Framerate Independence

```
FPS = 120
DESIRED_FPS = 60

prev_time = time.time()

def refresh_dt():
    global dt
    global prev_time
    dt = time.time() - prev_time
    dt*= DESIRED_FPS
    prev_time = time.time()
```

```
while True:

    refresh_dt()

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

Figure 8.

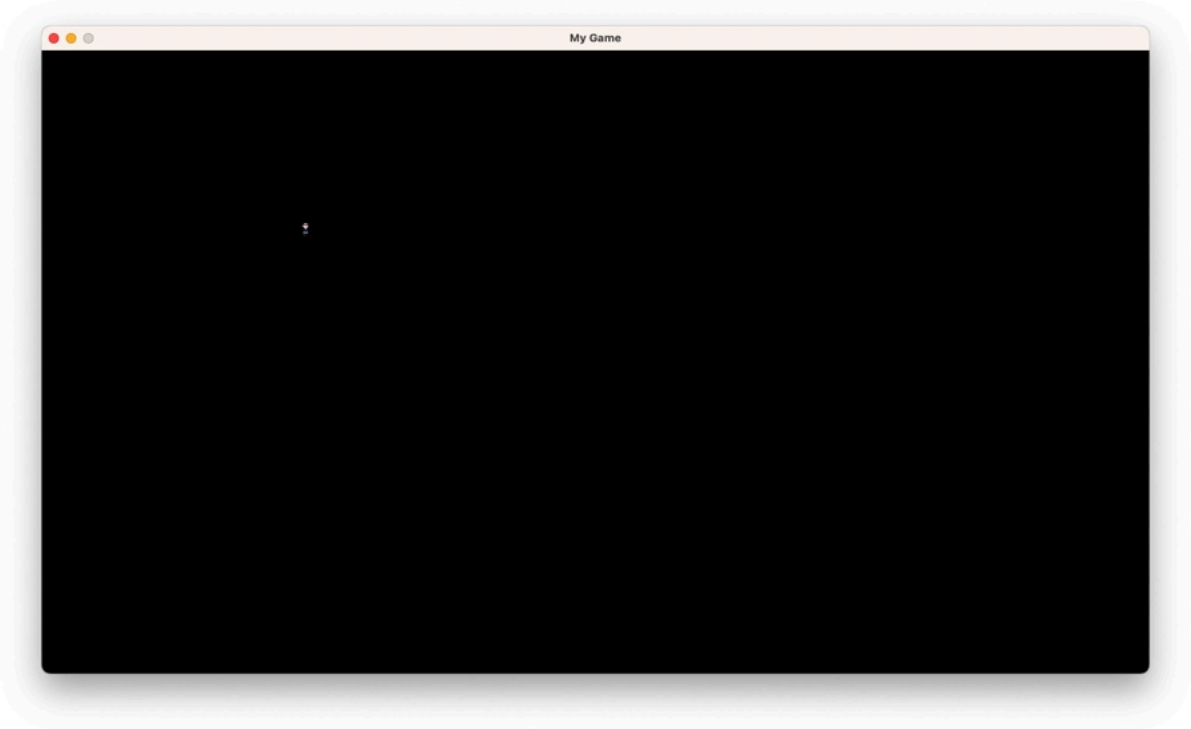
```
def move(rect,movement,tiles): # movement = [5,2]
    rect.x += movement[0]*dt
    collisions = collision_test(rect,tiles)
    for tile in collisions:
        if movement[0] > 0:
```

Figure 9.

Framerate Independence – It makes sure The game runs at the same speed regardless of the framerate, so we can change the rendered FPS and not affect the speed of the game.
In this case, we have the variable `FPS = 120` and `DESIRED_FPS = 60`, meaning that the game will render internally at the pace of 60 times per second, while rendering displaying 120 images to the screen every second.

Notice that in the game loop, the game is updated every 1/120 of a second but the movement or the overall effect is scaled down by the dt, because if we call the `refresh_dt()` function more frequently, the value of dt gets smaller.

Key Input & Character Movement



We have successfully spawned our main character and it is displaying on the screen, it is literally a rectangle doing nothing, we need to add key controls to make it move around.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_LEFT:
        self.movement[0] = True
    if event.key == pygame.K_RIGHT:
        self.movement[1] = True
```

This block of code means whenever these keys (in this case the left and right arrow keys) are pressed, certain actions are triggered or variables are set. We set the characters movement vector to be ‘True’, which means it has a positive numerical value and will start moving. However, we have noticed a problem, that the character never stops moving even we let ggo of the keys and we aren’t able to control it. We need to add a boundary condition which we need to set the velocity to 0 when the key is lifted up.

```
        if event.type == pygame.K_LEFT:
            if event.key == pygame.K_UP:
                self.movement[0] = True
            if event.key == pygame.K_RIGHT:
                self.movement[1] = True

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_UP:
                self.movement[0] = False
            if event.key == pygame.K_DOWN:
                self.movement[1] = False
```

```
self.velocity[1] = min(5, self.velocity[1] + 0.1 )
```

Entity’s gravity simulation can be implemented using just this one line.

The player character is done for now, but the game is very empty, we need to create a map so that the main character can navigate around.

Tilemap and blocks

```
import pygame

BASE_IMG_PATH = 'data/images/'

def load_image(path):
	img = pygame.image.load(BASE_IMG_PATH + path).convert()
	img.set_colorkey((0, 0, 0))
return img
```

In order to implement for the tilemaps, we need to import the textures first, I will create a load_image function to take the raw file and convert it into a usable pygame object. It has a base_img_path which eliminates the need to write the prefix path every time. Also, it sets the black part of the image transparent, in case the raw image does not have an alpha channel.

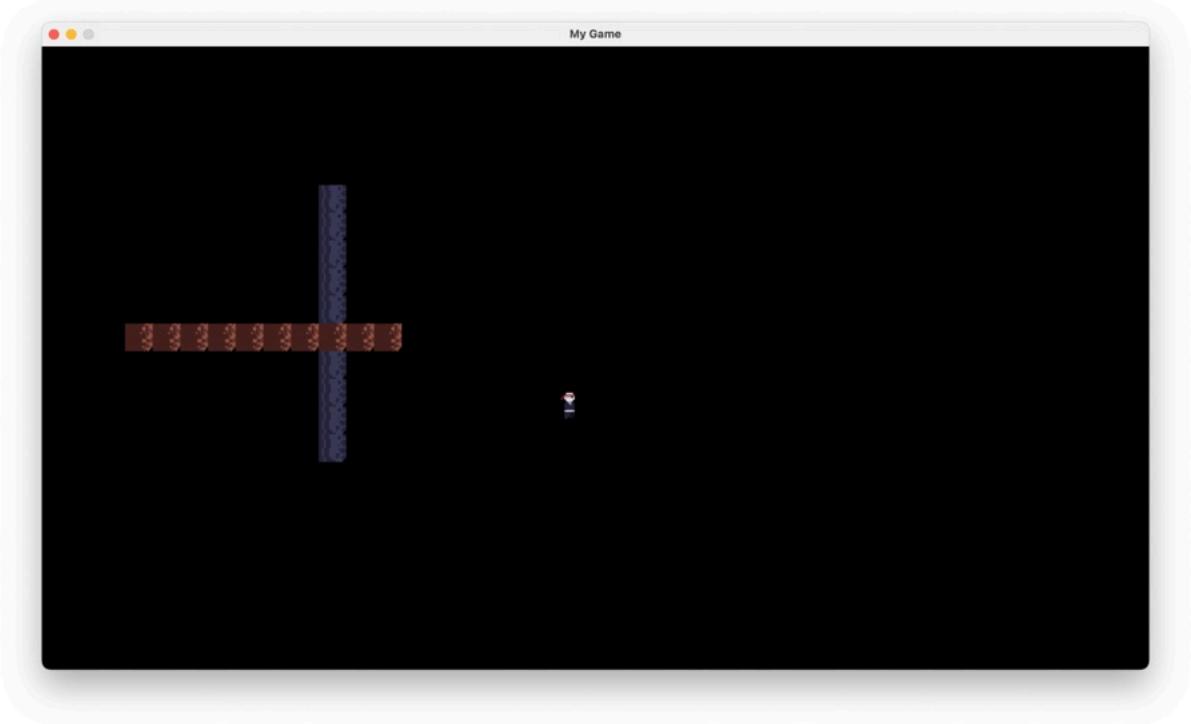
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···        Python

_ISSetPhysicalKeyboardCapsLockLED Inhibit
(base) dyrox@azhee CS Project 2 % /usr/local/bin/python3 "/Users/dyrox/Desktop/School/CS Proj
ect 2/game.py"
pygame-ce 2.3.0 (SDL 2.26.4, Python 3.10.9)
2023-08-08 11:08:11.608 Python[11393:501247] TSM AdjustCapsLockLEDForKeyTransitionHandling -
_ISSetPhysicalKeyboardCapsLockLED Inhibit
(base) dyrox@azhee CS Project 2 % /usr/local/bin/python3 "/Users/dyrox/Desktop/School/CS Proj
ect 2/game.py"
pygame-ce 2.3.0 (SDL 2.26.4, Python 3.10.9)
Traceback (most recent call last):
  File "/Users/dyrox/Desktop/School/CS Project 2/game.py", line 64, in <module>
    Game().run()
  File "/Users/dyrox/Desktop/School/CS Project 2/game.py", line 40, in run
    self.tilemap.render(self.display)
  File "/Users/dyrox/Desktop/School/CS Project 2/scripts/tilemap.py", line 20, in render
    tile_image = self.game.assets[tile['type']][tile['variant']]
KeyError: 'type'
(base) dyrox@azhee CS Project 2 %
```

But when I tried to render a few testing blocks onto the screen, it wouldn't let me. It throws a 'type' KeyError which is quite confusing.

```
for i in range(10):
    self.tilemap[str(3+i)+';10'] = {type: 'grass', 'variant': '1', 'pos':
    self.tilemap['10;'+str(5+i)] = {type: 'stone', 'variant': '1', 'pos':
```

I realized that in the dictionary for setting up the testing blocks, there was a field type which no quotation marks, since we are not using the Python type object in this case. This is a customized key field, I need to define my keys as strings, which needs quotation marks.

```
for i in range(10):
    self.tilemap[str(3+i)+';10'] = {'type': 'grass', 'variant': '1',
    self.tilemap['10;'+str(5+i)] = {'type': 'stone', 'variant': '1',
```



Now the blocks are rendered onto the screen, but there is something odd about them, even though the variants are set to be 1 in this case, the blocks are not facing up, which they should.

```
for i in range(10):
    self.tilemap[str(3+i)+';10'] = {'type': 'grass', 'variant': '1', 'pos': (3+i,10)}
    self.tilemap['10;'+str(5+i)] = {'type': 'stone', 'variant': '1', 'pos': (10,5+i)}
```

We realize that when the blocks are loaded into the array, they have a weird order, the array indices does not match the corresponding filenames, we can solve it quickly by sorting the array.

```
[<Surface(16x16x32, colorkey=(0, 0, 0, 255))>, <Surface(16x16x32, colorkey=(0, 0, 0, 255
(0, 0, 0, 255))>]
[<Surface(16x16x32, colorkey=(0, 0, 0, 255))>, <Surface(16x16x32, colorkey=(0, 0, 0, 255
(0, 0, 0, 255))>, <Surface(16x16x32, colorkey=(0, 0, 0, 255))>]
['8.png', '4.png', '5.png', '7.png', '6.png', '2.png', '3.png', '1.png', '0.png']
[<Surface(16x16x32, colorkey=(0, 0, 0, 255))>]
```

```python
def load_images(path):
    images = []
    for img_name in sorted(os.listdir(BASE_IMG_PATH + path)):
        images.append(load_image(path + '/' + img_name))
    return images
```

the images are loaded with the sorted names, which means they should display the correct textures now.

Tile Collision Physics

After finishing implementing the tilemap, we immediately notice a problem, the player just goes through the wall, there is no physics simulation between the player and the blocks, so we need to add that. As discussed in our design section, the way that this works is when the player is in contact with a block, the block essentially just resets the player's coordinate so it can't move forward.

```
def update(self, tilemap, movement = (0,0)):
    frame_movement = (movement[0] + self.velocity[0], movement[1] + self.velocity[1])



    self.pos[0] += frame_movement[0]
    entity_rect = self.rect()
    for rect in tilemap.physics_rects_around(self.pos):
        if entity_rect.colliderect(rect):
            if frame_movement[0] > 0:
                entity_rect.right = rect.left
            if frame_movement[0] < 0:
                entity_rect.left = rect.right
            self.pos[0] = entity_rect.x


    self.pos[1] += frame_movement[1]
    for rect in tilemap.physics_rects_around(self.pos):
        if entity_rect.colliderect(rect):
            if frame_movement[1] > 0:
                entity_rect.bottom = rect.top
            if frame_movement[1] < 0:
                entity_rect.top = rect.bottom
            self.pos[1] = entity_rect.y

    self.velocity[1] = min(5, self.velocity[1] + 0.1 )
```

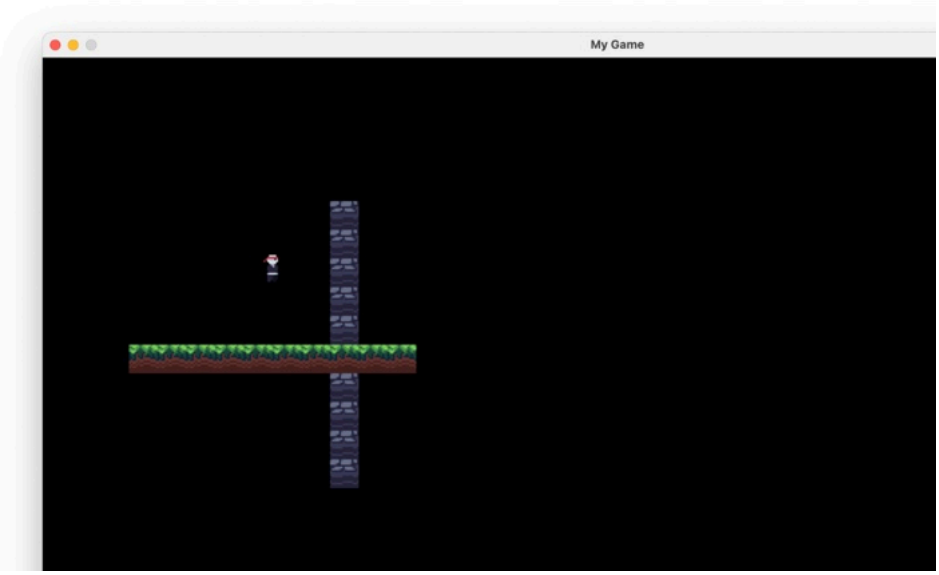
The highlighted bit is the code responsible for handling collision.

However, another problem arises, everything is okay when the player walks on the floor, but as soon as it touches the wall, it magically teleports. Now this took me quite a while to find

out what went wrong, but the problem lies in the two dimension system, we need to take account that this game has both x and y coordinates so we need to check for collision twice with a new variable, because the inner-frame movement could be hard to detect as it is constantly changing.

```
def update(self, tilemap, movement=(0, 0)):

    frame_movement = (movement[0] + self.velocity[0], movement[1] + self.velocity[1

    self.pos[0] += frame_movement[0]
    entity_rect = self.rect()
    for rect in tilemap.physics_rects_around(self.pos):
        if entity_rect.colliderect(rect):
            if frame_movement[0] > 0:
                entity_rect.right = rect.left
            if frame_movement[0] < 0:
                entity_rect.left = rect.right
            self.pos[0] = entity_rect.x

    self.pos[1] += frame_movement[1]
    entity_rect = self.rect()
    for rect in tilemap.physics_rects_around(self.pos):
        if entity_rect.colliderect(rect):
            if frame_movement[1] > 0:
                entity_rect.bottom = rect.top
            if frame_movement[1] < 0:
                entity_rect.top = rect.bottom
            self.pos[1] = entity_rect.y

    self.velocity[1] = min(5, self.velocity[1] + 0.1)

def render(self, surf):
    surf.blit(self.game.assets['player'], self.pos)
```

Jumping



Additionally, the character now has the ability to jump.

```
def jump(self):
    if self.wall_slide:
        if self.flip and self.last_movement[0] < 0:
            self.velocity[0] = 3.5
            self.velocity[1] = -2.5
            self.air_time = 5
            self.jumps = max(0, self.jumps - 1)
            return True
        elif not self.flip and self.last_movement[0] > 0:
            self.velocity[0] = -3.5
            self.velocity[1] = -2.5
            self.air_time = 5
            self.jumps = max(0, self.jumps - 1)
            return True

    elif self.jumps:
        self.velocity[1] = -3.5
        self.jumps -= 1
        self.air_time = 5
        return True
```

This doesn’t really have to explained line by line but the core idea here is we use some algorithm to restrict the time interval so the player doesn’t keep jumping.

Camera & Scroll

In a 2D platformer game, we would expect a camera to move around with the player, so the player is always in frame. We use a core idea of render offset, the render offset is calculated using the relative position of the player in the screen and we can use this variable, which is a just a pair coordinates, to render all the other elements with this offset, as if they are moving with the player.

```
    self.display.fill((0,0,0))
    self.scroll[0]+=(self.player.entity_rect.centerx-self.display.get_width()/2-self.scroll[0])/RUNNING_FPS
    self.scroll[1]+=(self.player.entity_rect.centery-self.display.get_height()/2-self.scroll[1])/RUNNING_FPS


    self.tilemap.render(self.display,offset=self.scroll)

    self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0))
    self.player.render(self.display,offset=self.scroll)
```

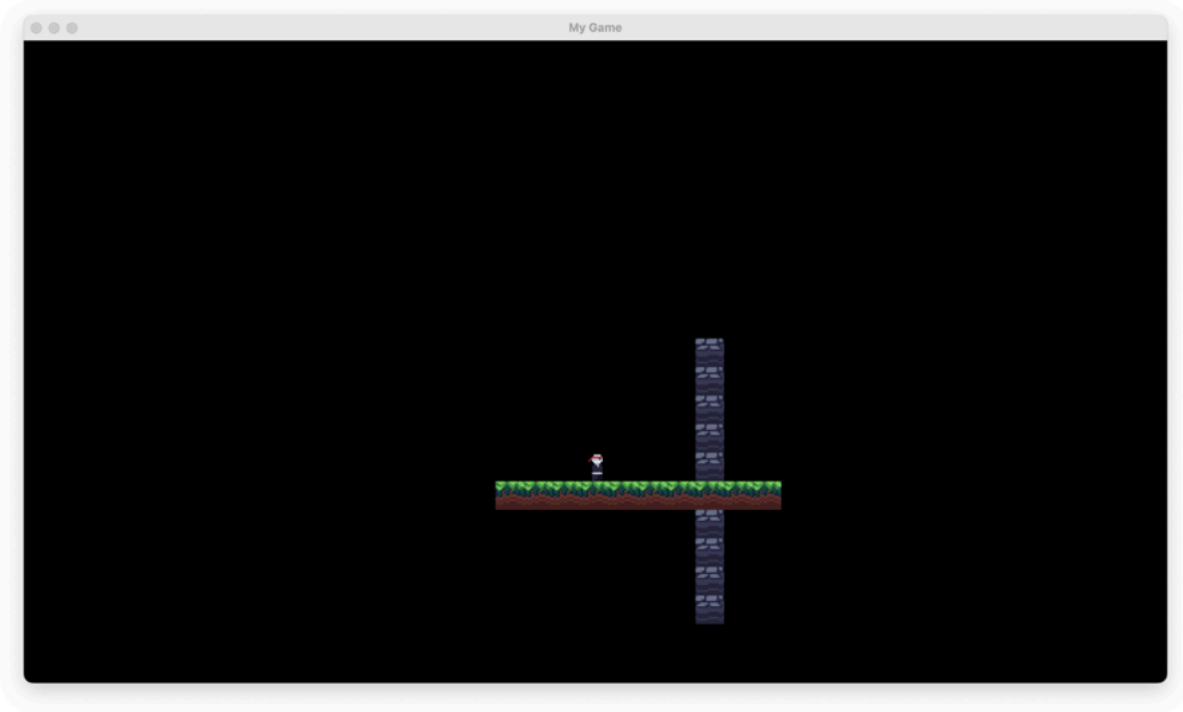
Our aim here is to center the player in the middle, so there are plenty of diving by halves here, the /RUNNING_FPS is for controlling the responsiveness of the correction, so that it doesn't look too robotic. However, the graphics look a bit wobbly, which means the when the camera settles to the target position, instead of stopping, it moves in randomly direction about 2pxs, which is quite annoying.

```
while True:
    self.display.fill((0,0,0))
    self.scroll[0]+=(self.player.entity_rect.centerx-self.display.get_width()/2-self.scroll[0])//RUNNING_FPS
    self.scroll[1]+=(self.player.entity_rect.centery-self.display.get_height()/2-self.scroll[1])//RUNNING_FPS


    self.tilemap.render(self.display,offset=self.scroll)

    self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0))
    self.player.render(self.display,offset=self.scroll)
```

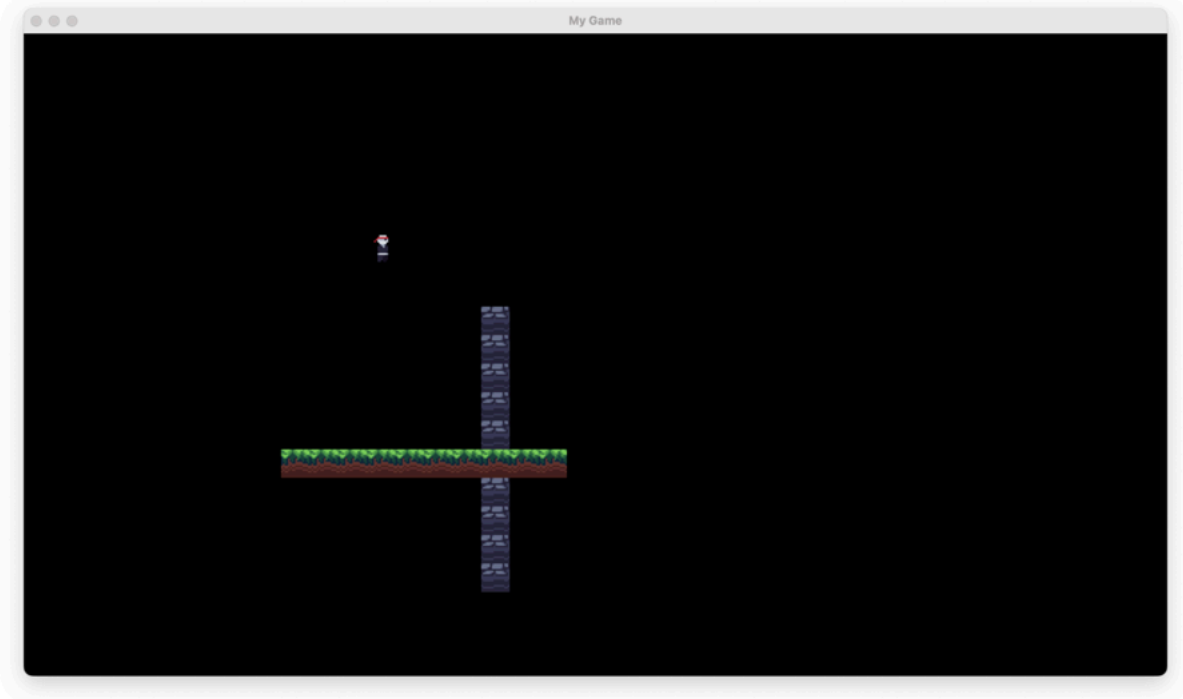
However, this is also easily fixed with a integer division operator //, so it rounds down to the nearest integer, eliminating glitches caused by floating point division.

```
def entity_render_update(self):
    if not self.dead:
        self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0))
        self.player.render(self.display, offset=self.render_scroll)

    for enemy in self.enemies.copy():
        kill = enemy.update(self.tilemap, (0, 0))
        enemy.render(self.display, offset=self.render_scroll)
        if kill:
            self.enemies.remove(enemy)
```

Then, all the sprites (players, tilemaps) will be rendered with the render_scroll.



Now we can see as the player character go out of the bound



The camera follows it!

Prototype 2: Game with added graphical elements

Backdrop and Clouds

It is finally ready for the visual elements, frankly, a pure black background looks quite dull and it is not visually appealing at all, so we can add a custom background to our game.

```
self.display.blit(self.assets['background'], (0, 0))
self.EVERYTHING_render_update()
```

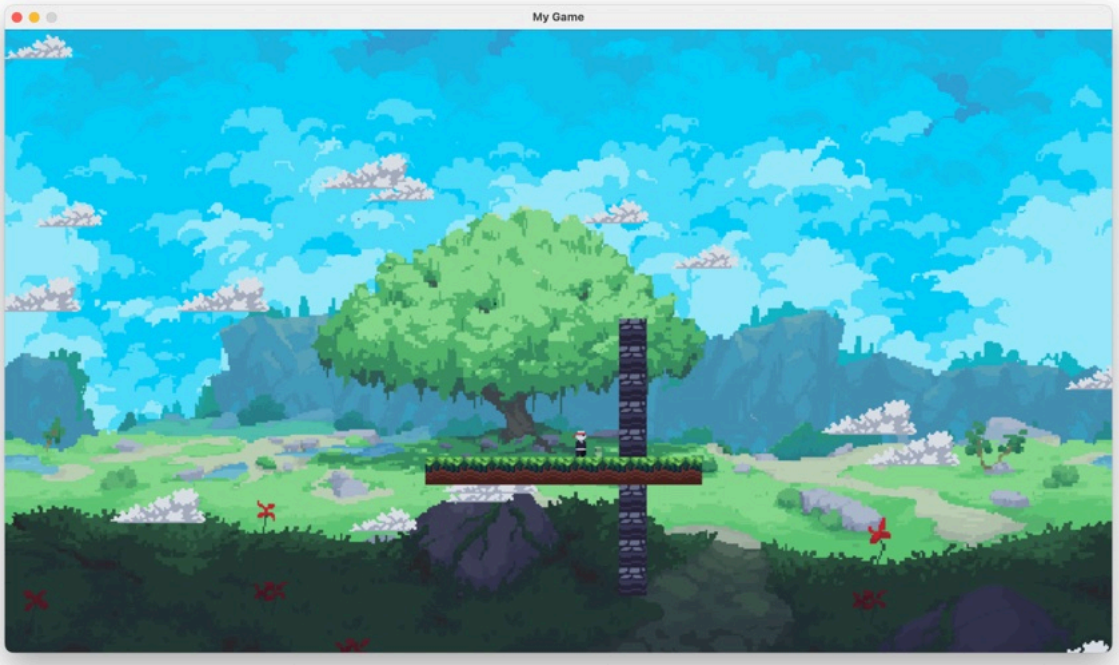
This is as simple as just displaying the image at (0,0), then we refreshing everything else on top of it so the elements don't overlap.

```
You, 2 months ago | 1 author (You)
class Cloud:
    def __init__(self, pos, img, speed, depth):
        self.pos = list(pos)
        self.img = img
        self.speed = speed
        self.depth = depth

    def update(self):
        self.pos[0] += self.speed

    def render(self, surf, offset=(0, 0)):
        render_pos = (self.pos[0] - offset[0] * self.depth, self.pos[1] - offset[1] * self.depth)
        surf.blit(self.img, (render_pos[0] % (surf.get_width() + self.img.get_width()) - self.img.get_width(), render_pos[1] % (surf.get_

You, 2 months ago | 1 author (You)
class Clouds:
    def __init__(self, cloud_images, count=16):
        self.clouds = []

        for i in range(count):
            self.clouds.append(Cloud((random.random() * 99999, random.random() * 99999), random.choice(cloud_images), random.random() * 0

        self.clouds.sort(key=lambda x: x.depth)

    def update(self):
        for cloud in self.clouds:
            cloud.update()

    def render(self, surf, offset=(0, 0)):
        for cloud in self.clouds:
            cloud.render(surf, offset=offset)
```

Then we should also add some clouds to the game, using the parallax effect we mentioned in the design section.

```
self.clouds = Clouds(self.assets['clouds'], count=16)
```

This spawns 16 clouds onto the screen.


My Game

Character State & Animations

Currently, when the player jumps or moves, the character sprite doesn't change at all and it looks odd. To give our character more 'character' (pun not intended), we can define the character states and give them animation.

```
'player/idle': Animation(load_images('entities/player/idle'), img_dur = 6),
'player/run': Animation(load_images('entities/player/run'), img_dur = 4),
'player/jump': Animation(load_images('entities/player/jump')),
'player/slide': Animation(load_images('entities/player/run')),
'player/wall_slide': Animation(load_images('entities/player/run'))
```

For example, we would have a set of idling player animation, and we need to animate the player differently if it is moving.

```
if not self.wall_slide:
    if self.air_time > 4:
        self.set_action('jump')
    elif movement[0] != 0:
        self.set_action('run')
    else:
        self.set_action('idle')
```

For example, this piece of code determines if the player is jumping (if it's off the ground), or running (when the movement doesn't equal zero), else it's just idle.

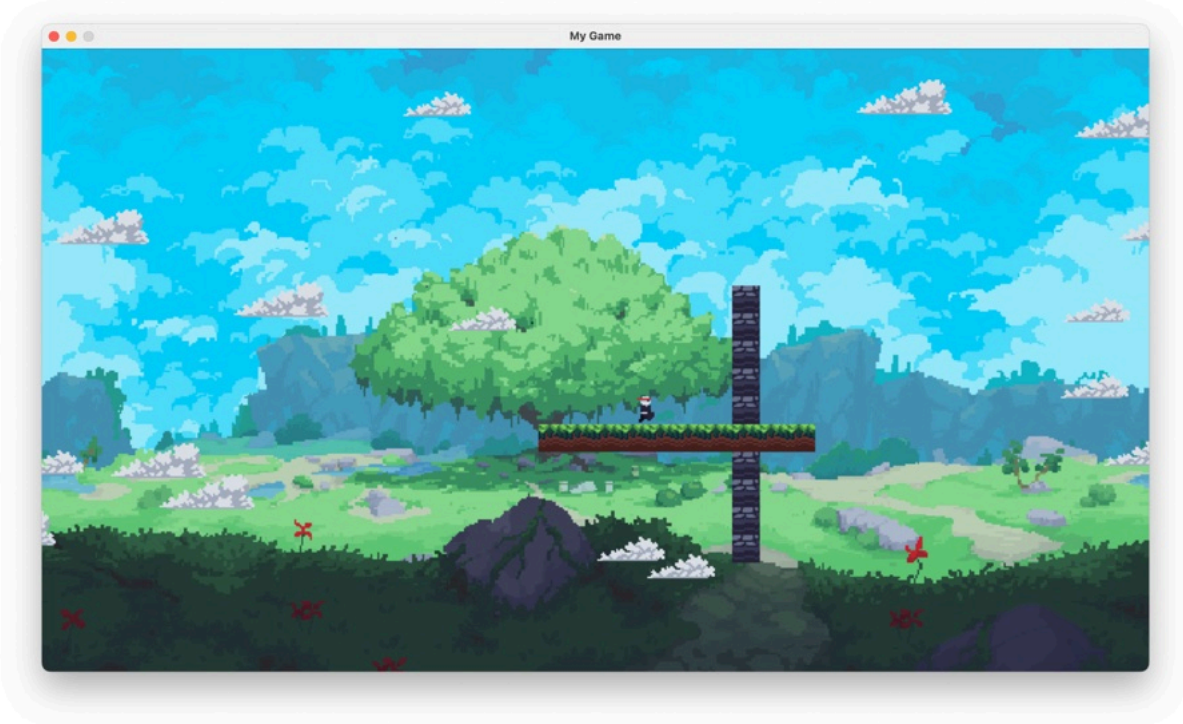
```
def set_action(self, action):
    if action != self.action:      You, 3 months ago • dashing …
        self.action = action
        self.animation = self.game.assets[self.type + '/' + self.action].copy()
```

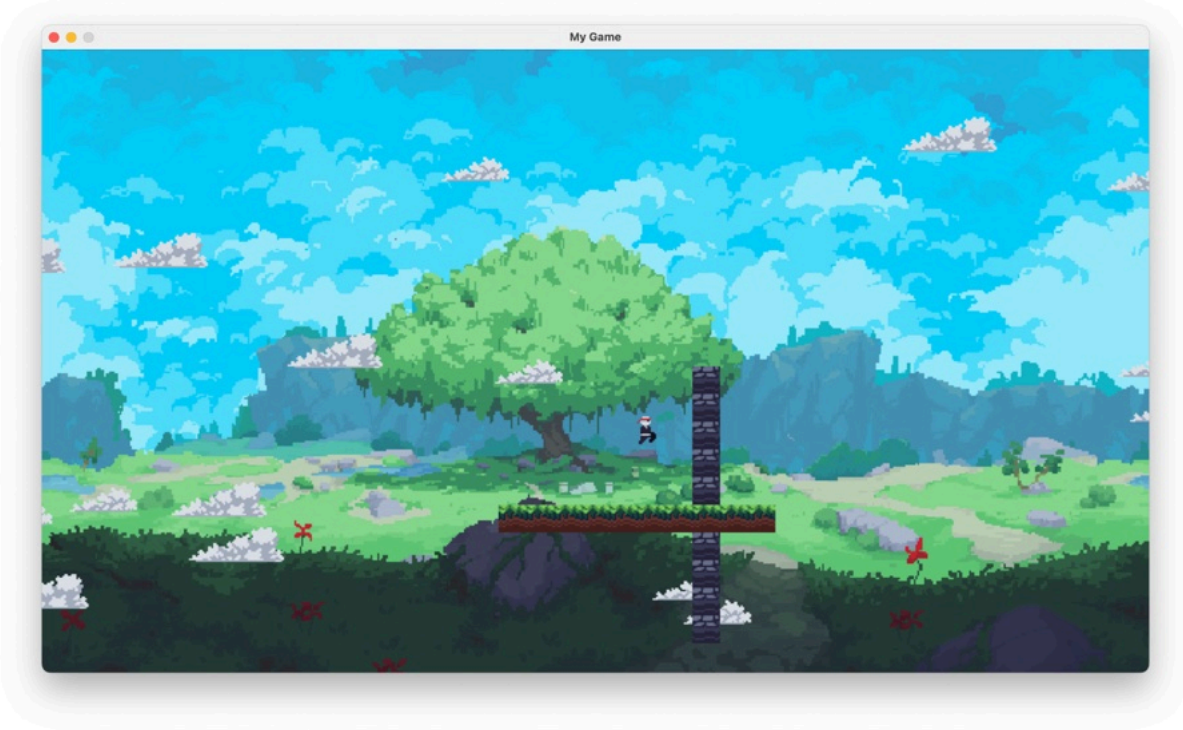
Then we would select the corresponding images to animate, the set of images will be loaded as an array.

sprite_list =[]

```python
class Animation:
    def __init__(self,images,img_dur=5,loop=True):
        self.images = images
        self.img_duration = img_dur
        self.loop = loop
        self.done = False
        self.frame = 0

    def copy(self):
        return Animation(self.images, self.img_duration, self.loop)

    def update(self):
        if self.loop:
            self.frame = (self.frame + 1) % (len(self.images) * self.img_duration)
        else:
            self.frame = min(self.frame + 1, len(self.images) * self.img_duration - 1)
            if self.frame >= len(self.images) * self.img_duration - 1:
                self.done = True

    def img(self):
        return self.images[int(self.frame/self.img_duration)]
```

Then it loops through the array to animate, resetting the index to 0 if the it is greater the image count.





The player now shoes a ‘jumping’ animation when the jump button is triggered.

Implementing the Level Editor

The basic game engine is pretty much playable now, but we need to expand on the maps.I did think about using existing level editors out there such as LDtk, but I thought it would be more fun to write my own level editor, and it turned out to be easier than expected. The editor is going to be based off the main game as I have already implemented the logic to render tilemaps, I just have to add the ability to add and delete tiles from a tilemap, then it can be saved onto the disk as a file.

```
RENDER_SCALE = 2.0

RUNNING_FPS = 60
DISPLAYING_FPS = 120

class Editor:
    def __init__(self):
        pygame.init()
        pygame.display.set_caption('editor')

        window_size = (1280,720)
        self.screen = pygame.display.set_mode(window_size)
        self.display = pygame.Surface((640,360))
        # self.display = pygame.Surface((1280,720))

        self.clock = pygame.time.Clock()
        self.movement = [False,False]

        self.assets = {
            'decor': load_images('tiles/decor'),
            'grass': load_images('tiles/grass'),
            'stone': load_images('tiles/stone'),
            'large_decor': load_images('tiles/large_decor'),
        }
```

As explained in the design section, I will have categories of tile called types and variants, which are basically subgroups of a bigger group.

```
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                self.clicking = True
            if event.button == 3: #right mouse
                self.right_clicking = True
```

The event listener listens for mouse clicking actions, the clicking variables are set to True accordingly to which button is pressed.

```
if self.shift: #change variant
    if event.button == 4: #scroll up
        self.tile_variant = (self.tile_variant -1)%len(self.assets[self.tile_list[self.tile_group]])
    if event.button == 5: #scroll down
        self.tile_variant = (self.tile_variant +1)%len(self.assets[self.tile_list[self.tile_group]])
else: #change block type
    if event.button == 4: #scroll up
        self.tile_group = (self.tile_group -1)%len(self.tile_list)
    if event.button == 5: #scroll down
        self.tile_group = (self.tile_group +1)%len(self.tile_list)
```

To make the menu easier to navigate, the user can scroll on the mouse wheel to go back and forth in the tile group, and to delete redundant controls, I incorporated the changing variant with the shift button, so when the shift button is pressed, the variant will be cycled instead of the tile type.
Previous we have talked about the on-grid and off-grid elements, the mouse position is translate from actual coordinates to pixel coordinate depending if ongrid is toggled on.

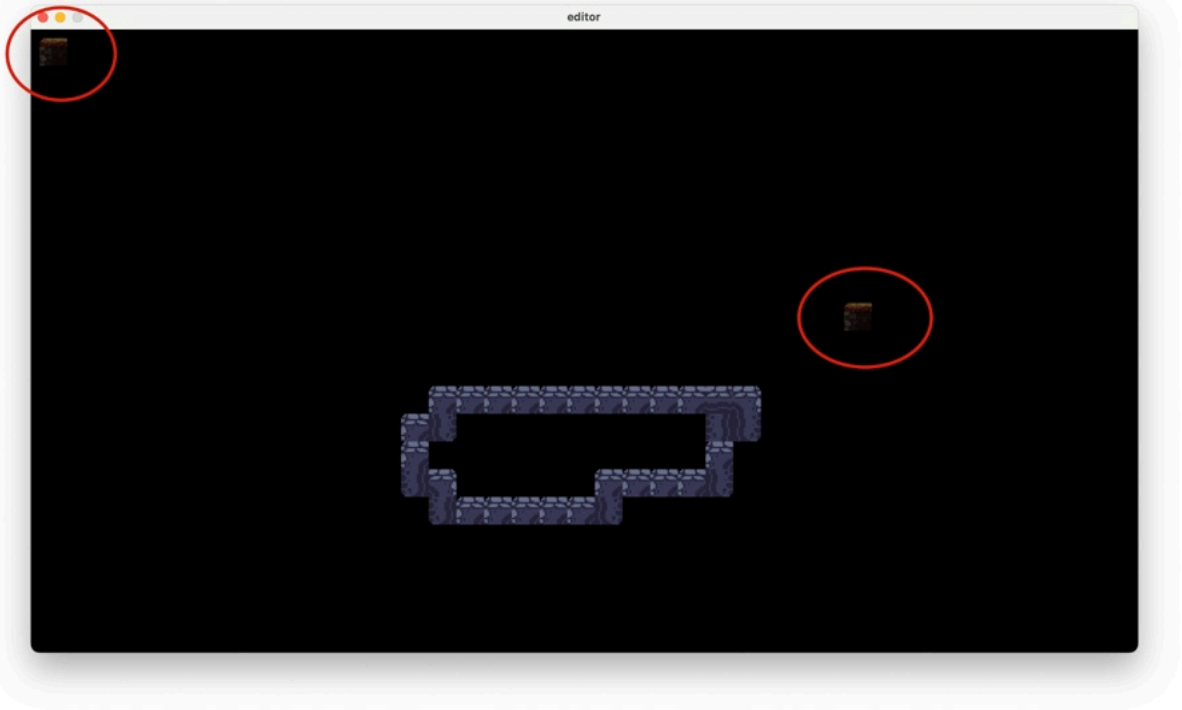
```
    mpos = pygame.mouse.get_pos()
    mpos = (mpos[0] / RENDER_SCALE, mpos[1] / RENDER_SCALE)
    tile_pos = (int((mpos[0] + self.scroll[0]) // self.tilemap.tile_size),
                int((mpos[1] + self.scroll[1]) // self.tilemap.tile_size))

    if self.ongrid:
        self.display.blit(current_tile_img, (tile_pos[0] * self.tilemap.tile_size - self.scroll[0],
                                             tile_pos[1] * self.tilemap.tile_size - self.scroll[1]))
    else:
        self.display.blit(current_tile_img, mpos)
```

This is the main mechanics for adding/deleting tiles, the tiles are saved to the tilemap dictionary object, then rendered onto the screen to reflect the changes.

```
if self.clicking and self.ongrid:
    self.tilemap.tilemap[str(tile_pos[0]) + ';' + str(tile_pos[1])] = {
        'type': self.tile_list[self.tile_group], 'variant': self.tile_variant, 'pos': tile_pos}
if self.right_clicking:
    tile_loc = str(tile_pos[0]) + ';' + str(tile_pos[1])
    if tile_loc in self.tilemap.tilemap:
        del self.tilemap.tilemap[tile_loc]
    for tile in self.tilemap.offgrid_tiles.copy():
        tile_img = self.assets[tile['type']][tile['variant']]
        tile_r = pygame.Rect(tile['pos'][0] - self.scroll[0], tile['pos'][1] - self.scroll[1],
                             tile_img.get_width(), tile_img.get_height())
        if tile_r.collidepoint(mpos):
            self.tilemap.offgrid_tiles.remove(tile)
```

To let the user know which tile has been selected, we render a semi-transparent image of the current tile next to the mouse pointer.

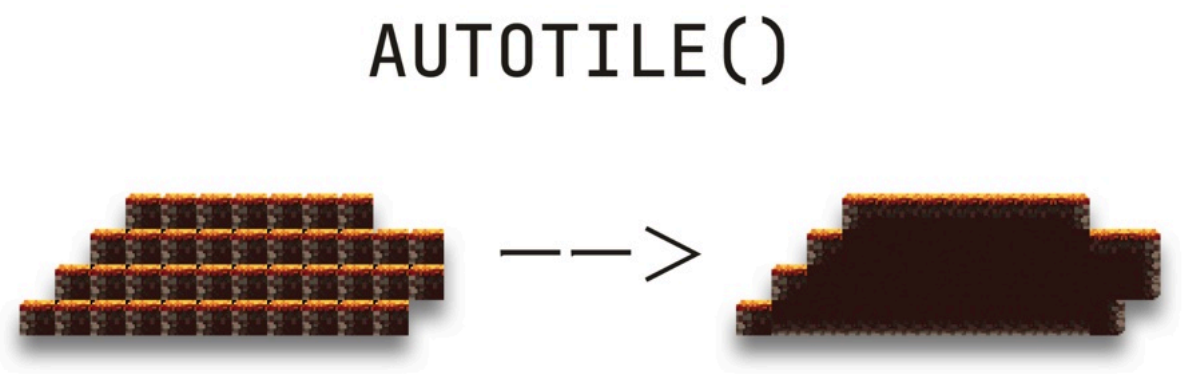
```
if self.ongrid:
    self.display.blit(current_tile_img, (tile_pos[0] * self.tilemap.tile_size - self.scroll[0],
                                         tile_pos[1] * self.tilemap.tile_size - self.scroll[1]))
else:
    self.display.blit(current_tile_img, mpos)
```



To ease for usability, texts are added to the left and right corner of the window to list out the corresponding functions and keybinds.

Much better.

The Autotiling Algorithm



I will reuse the diagram from the design stage to show you what it does to a chunk of blocks.

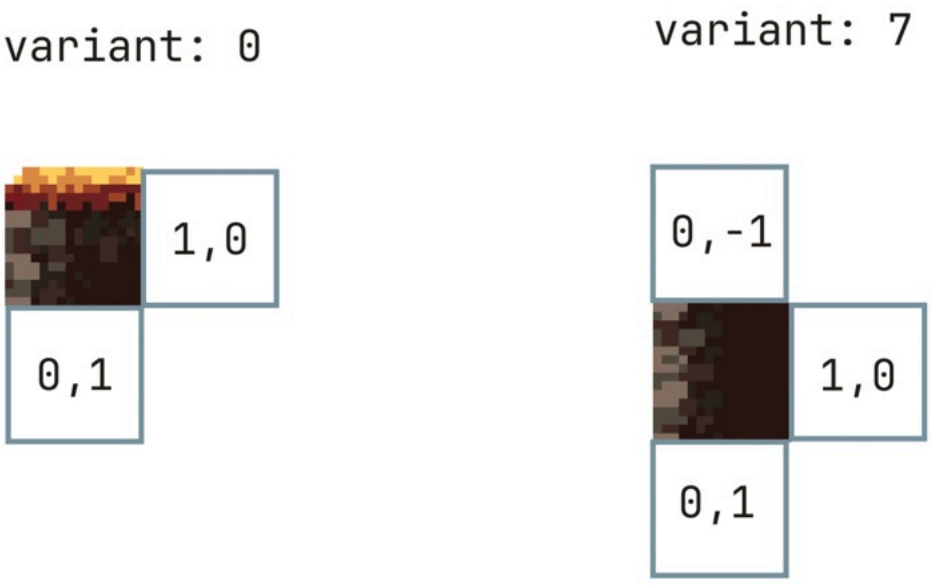
This algorithm can take a chunk of blocks, purely on its shape, to give every single block its correct texture. All the variants are displayed below using the index 0 - 8, a total of 9 blocks representing a 3x3 block. We can invent the algorithm by examining some simpler cases, for example all the blocks 'inside' has the variant 4 and what do the blocks have in common?

They are neighboured from above, below, left and right. Basically in all directions. We can use this idea of neighboured blocks to determine the variant of target blocks.

```
AUTOTILE_MAP = {
    tuple(sorted([(1, 0), (0, 1)])): 0,
    tuple(sorted([(1, 0), (0, 1), (-1, 0)])): 1,
    tuple(sorted([(-1, 0), (0, 1)])): 2,
    tuple(sorted([(-1, 0), (0, -1), (0, 1)])): 3,
    tuple(sorted([(-1, 0), (0, -1)])): 4,
    tuple(sorted([(-1, 0), (0, -1), (1, 0)])): 5,
    tuple(sorted([(1, 0), (0, -1)])): 6,
    tuple(sorted([(1, 0), (0, -1), (0, 1)])): 7,
    tuple(sorted([(1, 0), (-1, 0), (0, 1), (0, -1)])): 8,
}
```

By creating an autotile_map like this, we can determine the variant of the block according to its neighbours, for example, a block which has variant 0 means it has a neighbour underneath it and to its right, a block that has variant 7 means it has up, down and right neighbours.

The algorithm will look from the centre block to determine the variant.



```
def autotile(self):
    for loc in self.tilemap:
        tile = self.tilemap[loc]
        neighbours = set()
        for shift in [(1,0),(-1,0),(0,1),(0,-1)]:
            check_loc = str(tile['pos'][0] + shift[0]) + ';' + str(tile['pos'][1] + shift[1])
            if check_loc in self.tilemap:
                if self.tilemap[check_loc]['type'] == tile['type']:
                    neighbours.add(shift)
        neighbors = tuple(sorted(neighbors))
        if (tile['type'] in AUTOTILE_TYPES) and (neighbors in AUTOTILE_MAP):
            tile['variant'] = AUTOTILE_MAP[neighbors]       You, 1 second ago • Uncommitted changes
```

All of the tiles are located into a huge dictionary first, and then we are going to check all of their neighbours by iterating through a fixed array, which are [(1, 0), (-1, 0), (0, -1), (0, 1)], corresponding to the neighbours. These neighbours are then put together to compare with the autotile_map above, to determine the variant.

Implementing this function into the level editor means I can turn a level like this into this instantly, with the click of a button, saving me a lot of time.


editor

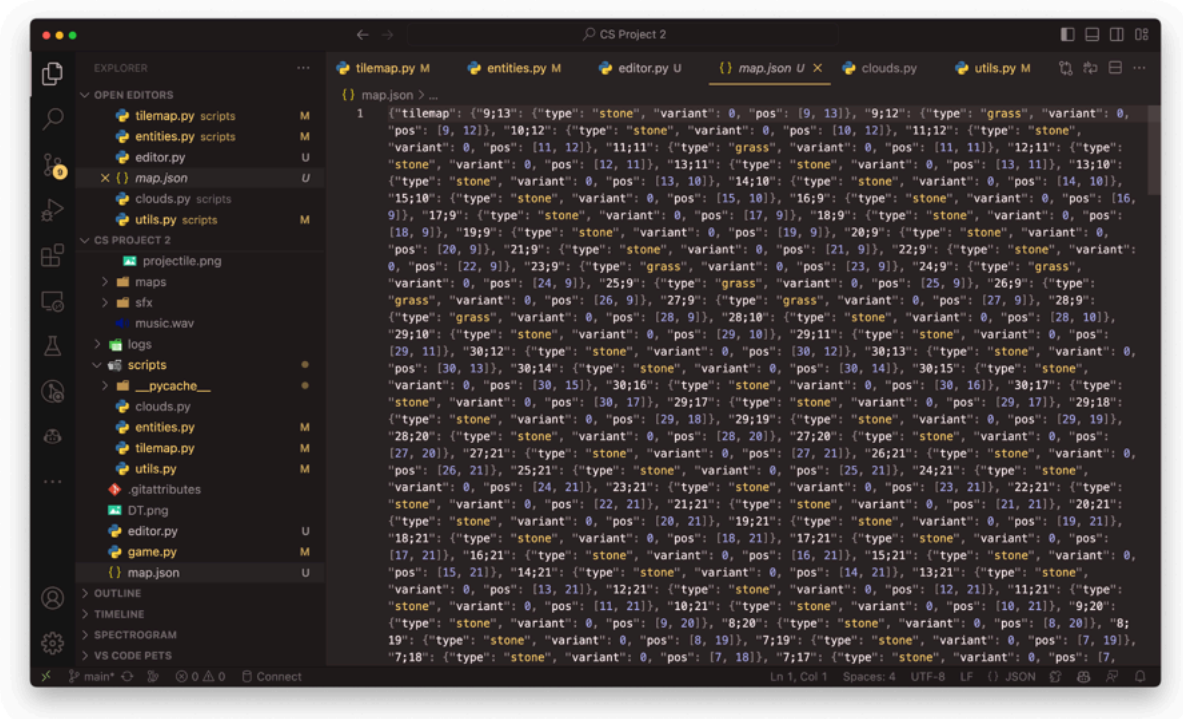
Saving & Loading Maps

Since our map uses a dictionary format, it would be appropriate to save them into json files as it can be directly read by python.

```
if event.key == pygame.K_o:
    self.tilemap.save('map.json')
```

```
def save(self, path):
    f = open(path, 'w')
    json.dump(
        {'tilemap': self.tilemap,
        'tile_size': self.tile_size,
        'offgrid': self.offgrid_tiles}
        ,f)
```

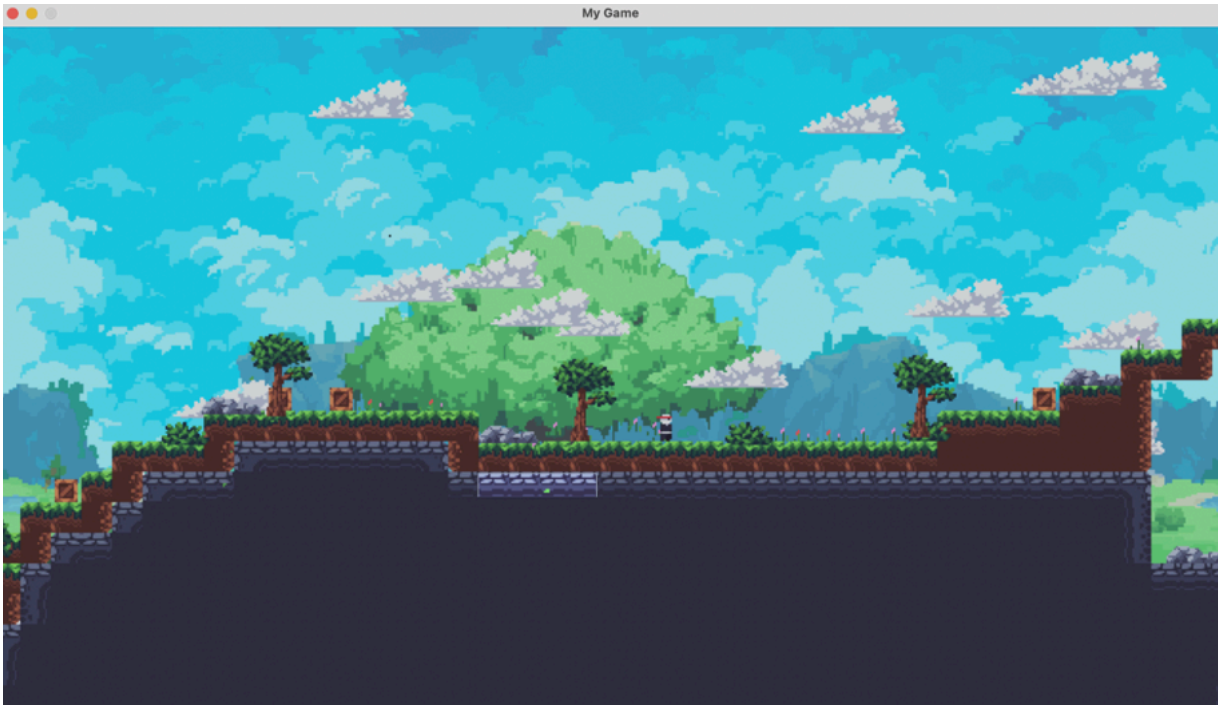
When the key ‘o’ is pressed in the level editor, a copy of the map is saved to a json file.



So the map looks something like this, we don’t have to read it as it is left for the program to decode, so we need to introduce a new function to load the file and render is as a map in game.

```
def load(self, path):
    f = open(path, 'r')
    data = json.load(f)
    f.close()       You, 1 second ago • Uncommitted changes
    self.tilemap = data['tilemap']
    self.tile_size = data['tile_size']
    self.offgrid_tiles = data['offgrid']
```

Loading the previous map we just made in the level editor, we get a map that looks like this.



I have added a lot of variations in the map, such as random crates, rocks and trees, usage of different grass and rock blocks, as well as a diverse landscape.

Giving Life to Leaves! (Sine Functions)

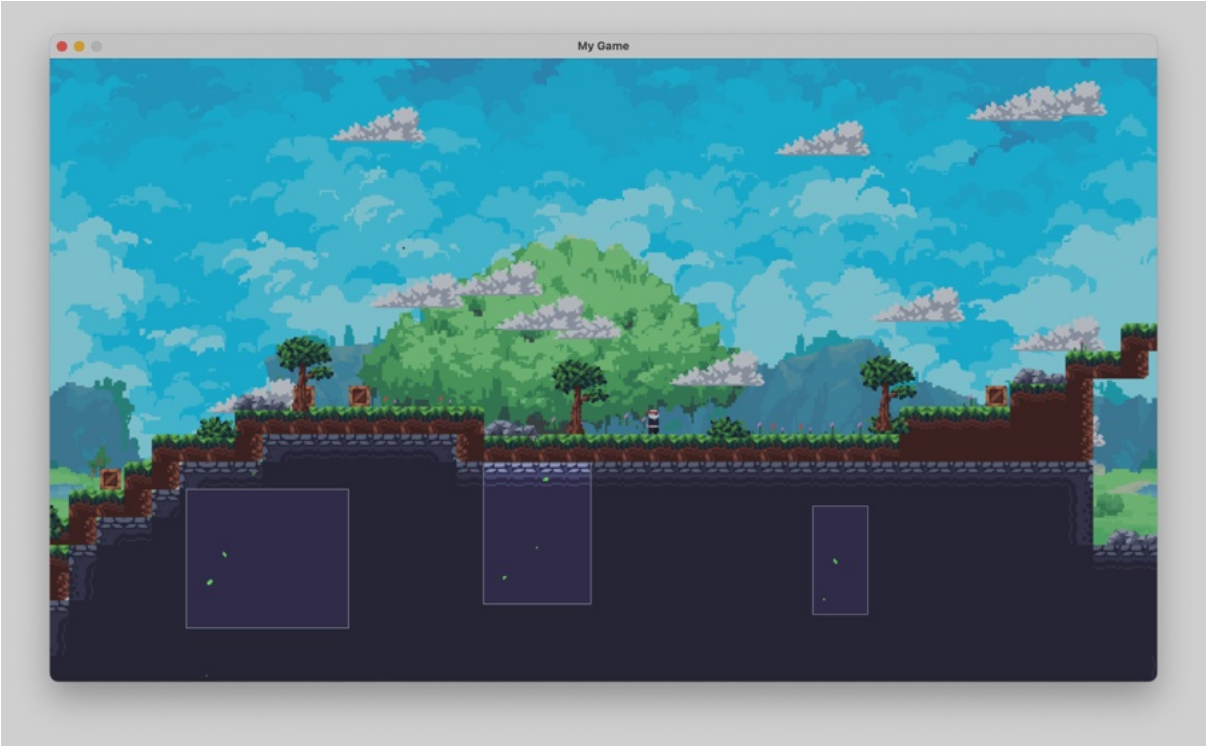


Figure 3.

We could enhance the scene's realism by introducing more dynamic visual elements. Currently, the leaves fall directly downwards, which feels somewhat lifeless and unnatural. In reality, leaves often drift left and right as they fall, because they are carried by the wind. Implementing a mechanism to simulate this wind-driven movement could significantly improve the scene's authenticity.

A smart way would be to use a sine function to automate the movement. By applying the sine function, we can simulate the natural, oscillating motion of leaves as they fall, influenced by the wind. The sine wave's peaks and troughs can represent the left and right drifts of the leaves, respectively.

The mathematical equation looks like this:

$$x = Asin(\omega t + \varepsilon)$$

with x representing the horizontal location (coordinate) of the leaves, A is amplitude of the movement, ω angular frequency and ε phase shift.

The graph looks a little something like this below:

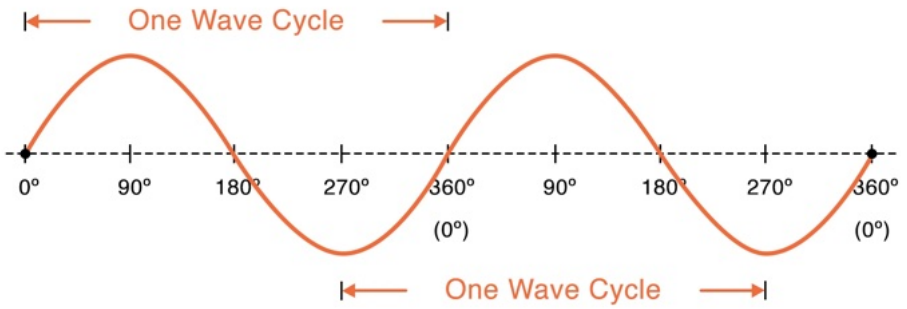


Figure 4.

Implementing this in code would be:

```
for particle in self.particles.copy():
    kill = particle.update()
    particle.render(self.display, offset=self.scroll)
    if particle.type == 'leaf':
        particle.pos[0] += math.sin(particle.animation.frame*0.035)*0.3
    if kill:
        self.particles.remove(particle)
```

Figure 5. Sine function implemented with the Python built-in math module

with pos[0] represents the x-axis

Figure 6.

This proves to be very effective and instantly gave life to the leaves

Prototype 3: Character ability system added

Wall Jumps

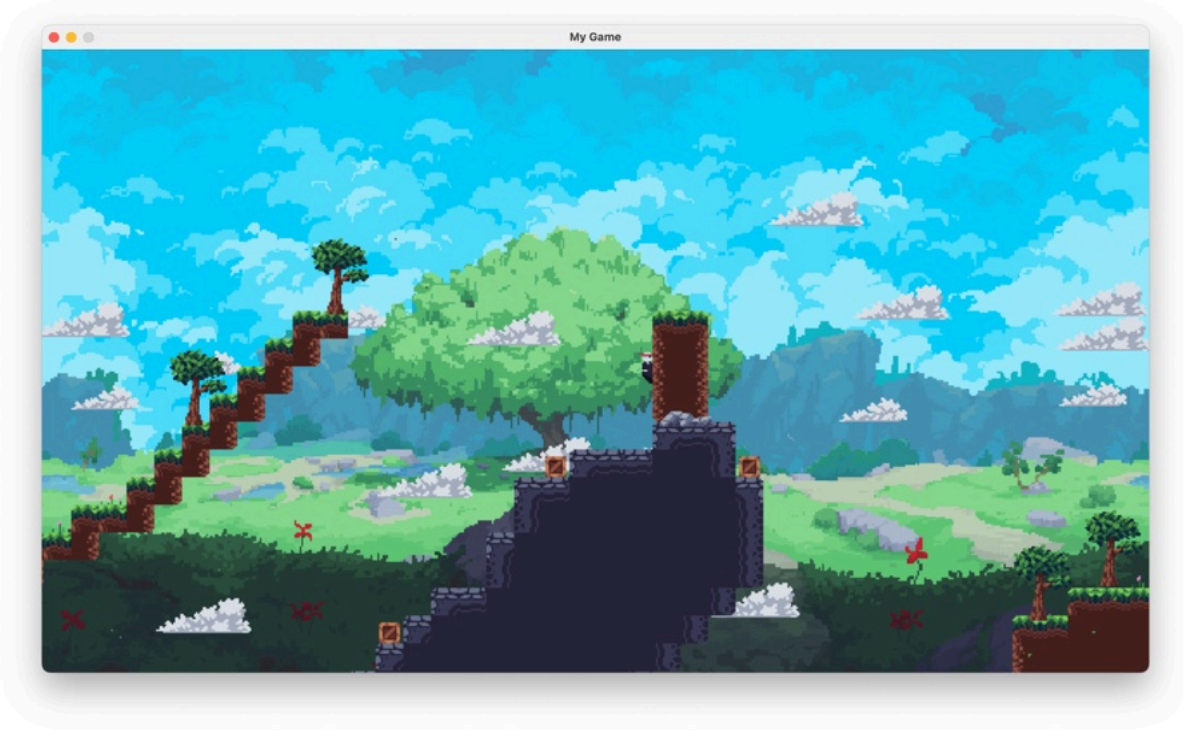
Besides the 'normal' jumps, the player can also perform wall jumps, which is jumping from the walls and giving them an extra boost upwards, so there exists a way to move up in the y direction.

```
def update(self, tilemap,movement=(0,0)):
    super().update(tilemap,movement=movement)
    self.air_time += 1

    if self.collisions['down']:
        self.air_time=0
        self.jumps = 1
    You, 1 minute ago • Uncommitted changes
    if self.air_time > 4:
        self.set_action('jump')
    elif movement[0]!=0:
        self.set_action('run')
    else:
        self.set_action('idle')

def jump(self):
    if self.jumps:
        self.velocity[1]=-3
        self.jumps-=1
        self.air_time = 5
```

When the player touches the floor, when the collision['down'] is True, we reset the jump count the player is able to perform.

```
self.wall_slide = False
if (self.collisions['right']or self.collisions['left']) and self.air_time > 4:
    self.wall_slide=True
    self.velocity[1] = min(0.5, self.velocity[1])
    if self.collisions['right']:      You, 27 seconds ago • Uncommitted changes
        self.flip = False
    else:
        self.flip = True
    self.set_action('wall_slide')
```

This block of code checks if the player is hanged onto a wall, when the collision is on the left or right, but not on the ground, which implies the player is in the air, hence setting wall_slide to be True.



This is a example of what does wall jump/wall slide looks like.

Dashing

Here comes the most important element of the game, which is the ability to dash

```
            self.player.jump()
        if event.key == pygame.K_x:
            self.player.dash()
```

We define a method dash() to perform the action when the X key is pressed.

```
def dash(self):
    if not self.dashing:
        if self.flip:
            self.dashing = -60
        else:
            self.dashing = 60
```

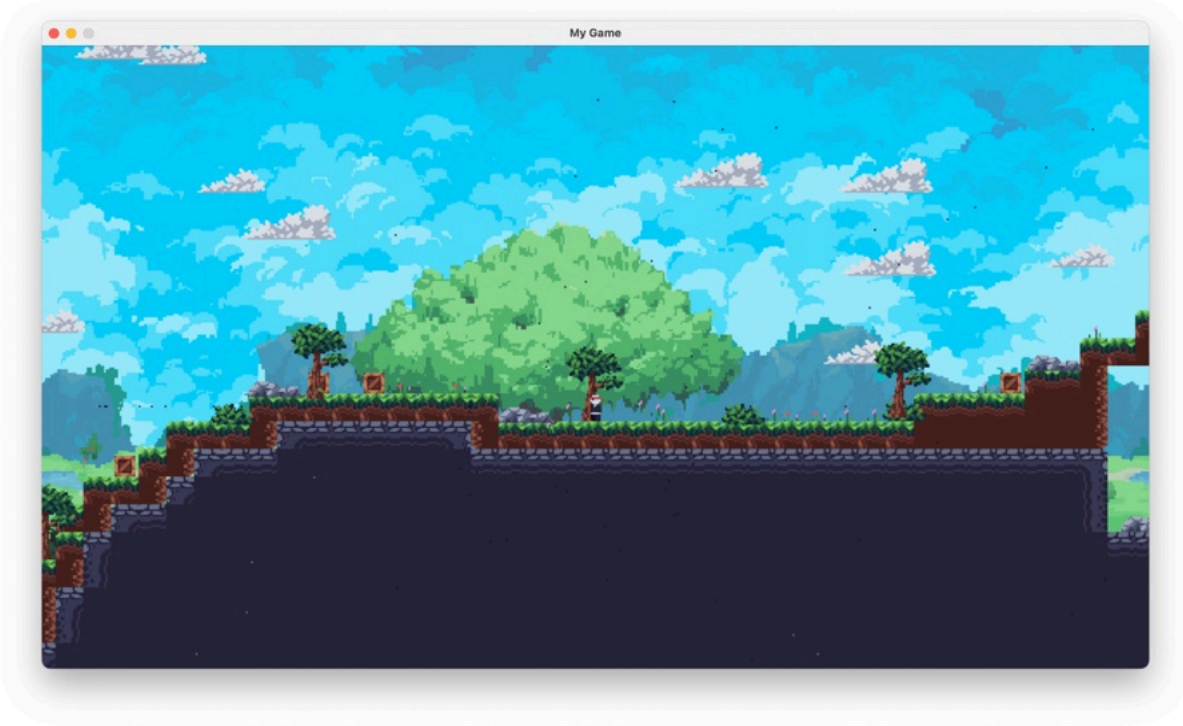
Here we determine of the dashing variable based on the direction the player is facing, negative if the character is flipped.

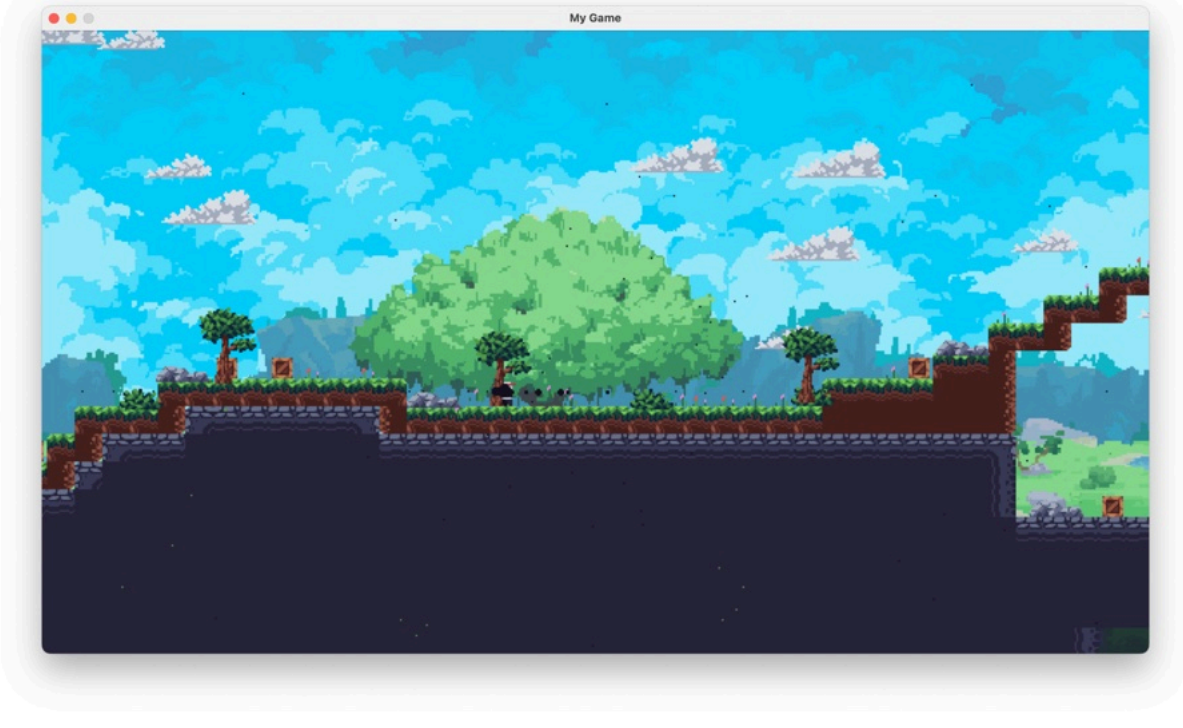
```
if self.dashing >0:
    self.dashing = max(0,self.dashing-1)
if self.dashing <0:
    self.dashing = min(0,self.dashing+1)
if abs(self.dashing)>50:
    self.velocity[0] = abs(self.dashing)/self.dashing *8
    if abs(self.dashing) == 51:
        self.velocity[0] *= 0.1 #divide velocity by 10
```

This is the core mechanics that deal with the dashing action, the top 4 lines basically resets the dashing variable to 0 when it's triggered, so it doesn't dash forever.
Else, the dashing speed is added onto the velocity and the update function takes the velocity value and updates the position.

```
if abs(self.dashing) in {60, 50}:
    for i in range(20):
        angle = random.random() * math.pi * 2
        speed = random.random() * 0.5 + 0.5
        pvelocity = [math.cos(angle) * speed, math.sin(angle) * speed]
        self.game.particles.append(Particle(self.game, 'particle', self.rect().center,
                                            velocity=pvelocity, frame=random.randint(0, 7)))
```

Other than that, we will also add a visual indicator whenever the player has teleported, it spawns exactly 20 particles with random angle and speed, on the player Rect. And it looks something like this.





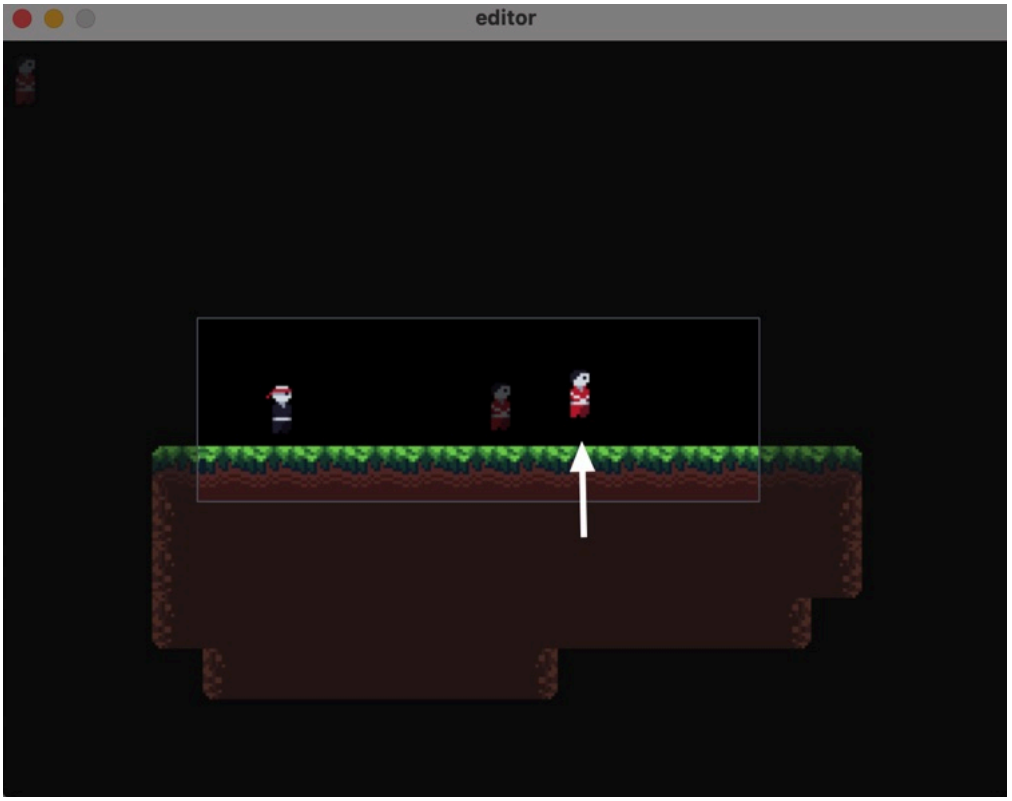
Level Spawner Debug

This is indeed quite eccentric, but for some reason the newly made levels in the level editor will not spawn.

It loads perfectly in the level editor but if I try to load the map into the main game, the following error appears.

```
Traceback (most recent call last):
  File "/Users/dyrox/Desktop/School/FINAL STUFF/game.py", line 225, in <module>
    Game().run()
  File "/Users/dyrox/Desktop/School/FINAL STUFF/game.py", line 70, in __init__
    self.load_level(self.level)
  File "/Users/dyrox/Desktop/School/FINAL STUFF/game.py", line 82, in load_level
    for spawner in self.tilemap.extract([('spawners', 0), ('spawners', 1)]):
  File "/Users/dyrox/Desktop/School/FINAL STUFF/scripts/tilemap.py", line 36, in extract
    for loc in self.tilemap:
RuntimeError: dictionary changed size during iteration
```

The dictionary has changed size during iteration, which is a weird error because we didn't really do that, but upon experiments I have found a fix for this problem.



So instead of saving the characters onto the grid, I am going to spawn them as offgrid elements.

```
496            10
497          ]
498        }
499      },
500      "tile_size": 16,
501      "offgrid": [
502        {
503          "type": "spawners",
504          "variant": 0,
505          "pos": [
506            89,
507            100
508          ]
509        },
510        {
511          "type": "spawners",
512          "variant": 1,
513          "pos": [
514            188,
515            106
516          ]
517        }
518      ]
519    }
```

Now the characters are spawned with no issues.

Screenshake

When the player hits the emey or gets hit, I want to add a screenshake to the display as a visual indicator.

```
elif abs(self.player.dashing) < 50:
    if self.player.rect().collidepoint(projectile[0]):
        self.projectiles.remove(projectile)
        self.HP -= 1
        self.player_got_hit_count += 1
        self.sfx['hit'].play()
        self.screenshake = max(16, self.screenshake)      You, 2 months ago • menu is working but very u
        for i in range(30):
            angle = random.random() * math.pi * 2
            speed = random.random() * 5
            self.sparks.append(Spark(self.player.rect().center, angle, 2 + random.random()))
            self.particles.append(Particle(self, 'particle', self.player.rect().center,
                                           velocity=[math.cos(angle + math.pi) * speed * 0.5,
                                                     math.sin(angle + math.pi) * speed * 0.5],
                                           frame=random.randint(0, 7)))
```

For example, when both dashing AND collision occurs, we know that the player has dashed into an enemy, so we set the screenshake to 16.

```
self.screenshake = max(0, self.screenshake - 1)
self.screenshake_offset = (random.random() * self.screenshake - self.screenshake / 2, random.random() * self.screenshake - self.screenshake / 2)

self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), self.screenshake_offset)
```

Then, this code is responsible for taking the screenshake and turning into a x,y coordinate offset we can use for display, also it's hard see from still images, a video of this will be included.

Loading Levels

Previously, we have talked about loading maps into the game, but we need to determine when should the player progress onto the next level.

```
self.tilemap = Tilemap(self, til
self.level = 0
self.load_level(self.level)
```

Basically we have a level variable which indicates the current level the player is in, and the variable is incremented when the player beats the level, when len(self.enemies) == 0, which means all enemies are killed.

```
if not len(self.enemies):
    self.transition +=1
    if self.transition >30:
        self.level = min(self.level+1,len(os.listdir('data/maps')))
        self.load_level(self.level)
if self.transition <0:
    self.transition +=1
```

If the player dies in the level, the level is re-loaded.

```
if self.dead:
    self.dead += 1
    if self.dead >= 10:
        self.transition = min(30, self.transition)
    if self.dead > 40:
        self.load_level(self.level)
```

Prototype 4: UI refined

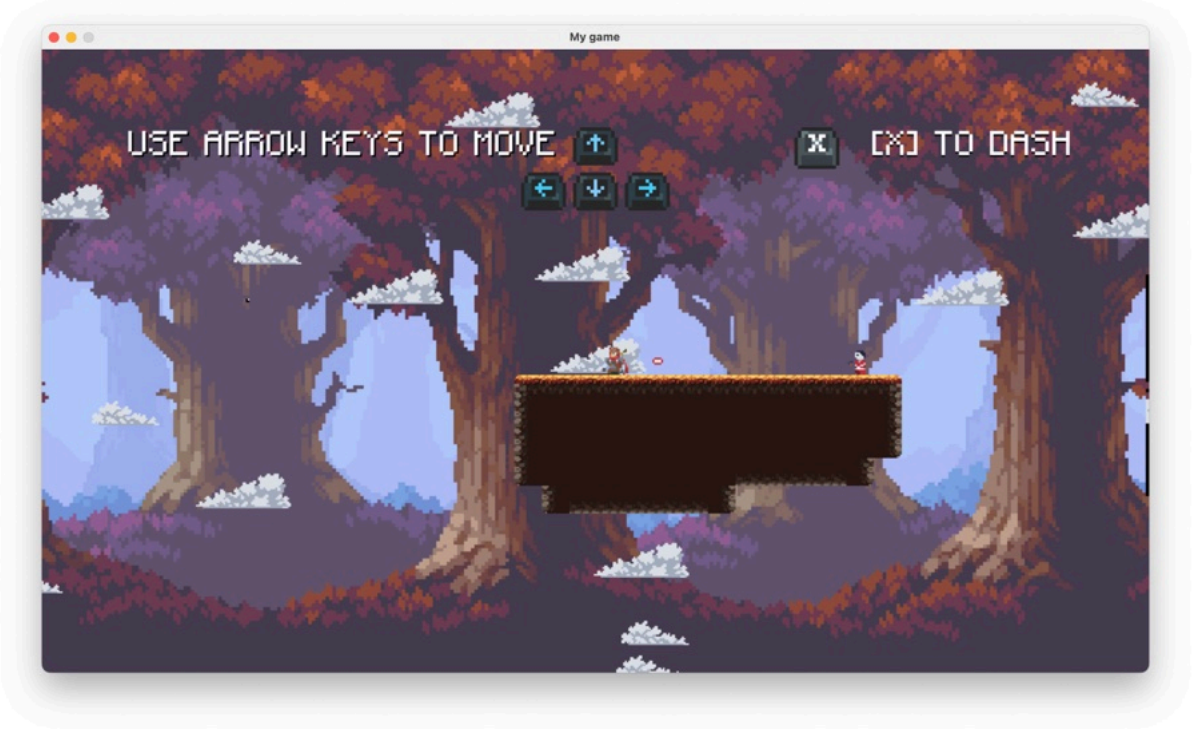
Text Rendering System

As discussed previously in the design stage, we will be using a unique text rendering system that render the text in two layers, one is the filled with the solid colour of the text and one black duplicate acting as the shadow, in order to improve the readability of the text.

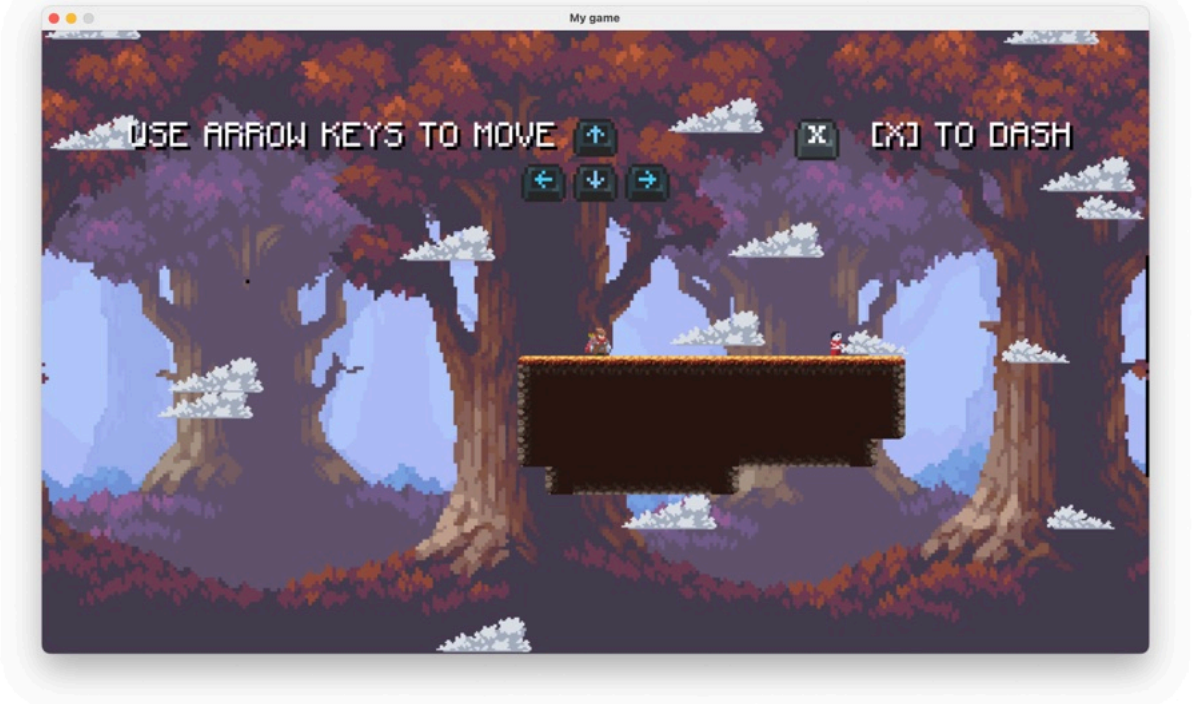
In this case, we can see that a single layer of white text is not so readable, the background as well as the clouds use the same color, the text can blend in and create a camouflage effect.


USE ARROW KEYS TO MOVE

While adding to a shadow of 1px offset does the text look better, we can even improve it further.



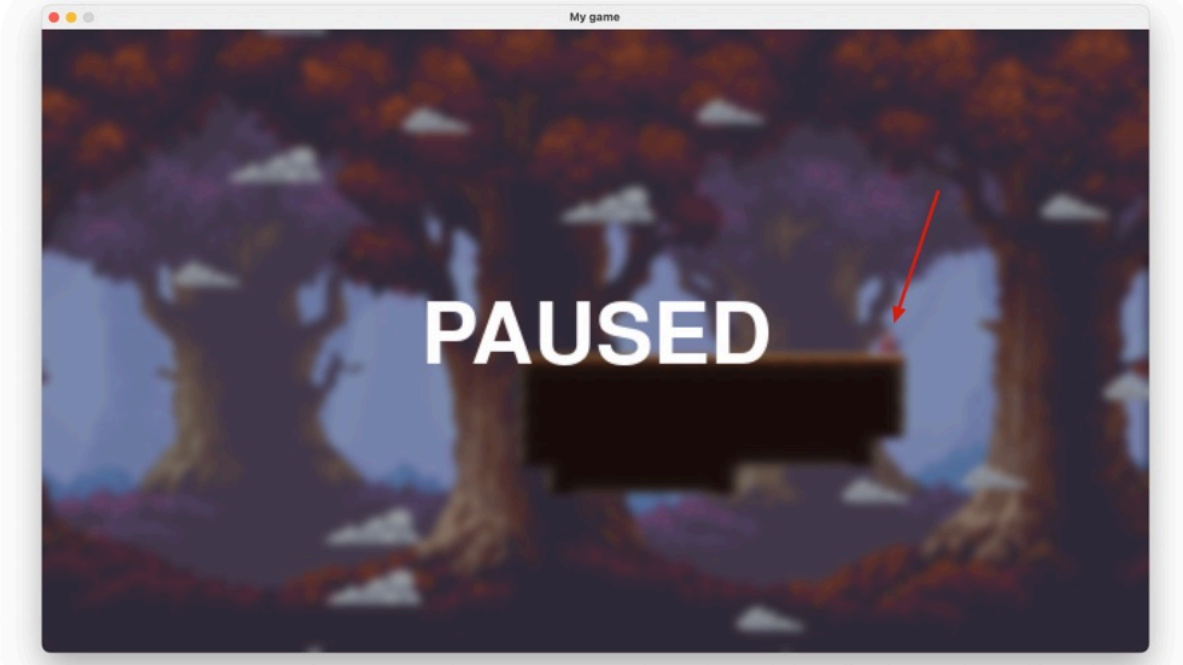
by creating a large offset of (-2,-2) both in the x and y direction, we essentially have a ‘thicker’ shadow and makes the text more readable, even with the fast moving graphics in the background.



#Pause Menu (Scrapped)

Note: This is a scrapped, unused version of the pause menu from the game's development phase. It was not included in the final build and is presented here only to demostrate the progression of the game-making process.

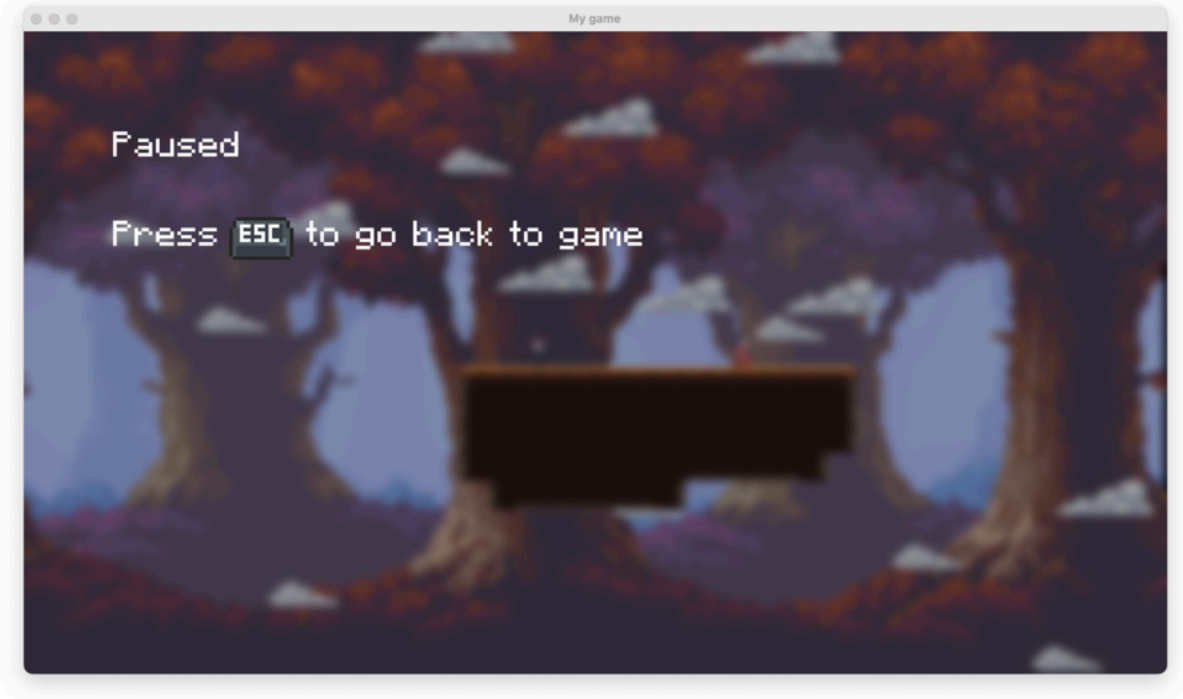
This is the alpha stage of the pause menu, the text and the background are just there as a placeholder, which aren’t very aesthetically pleasing.



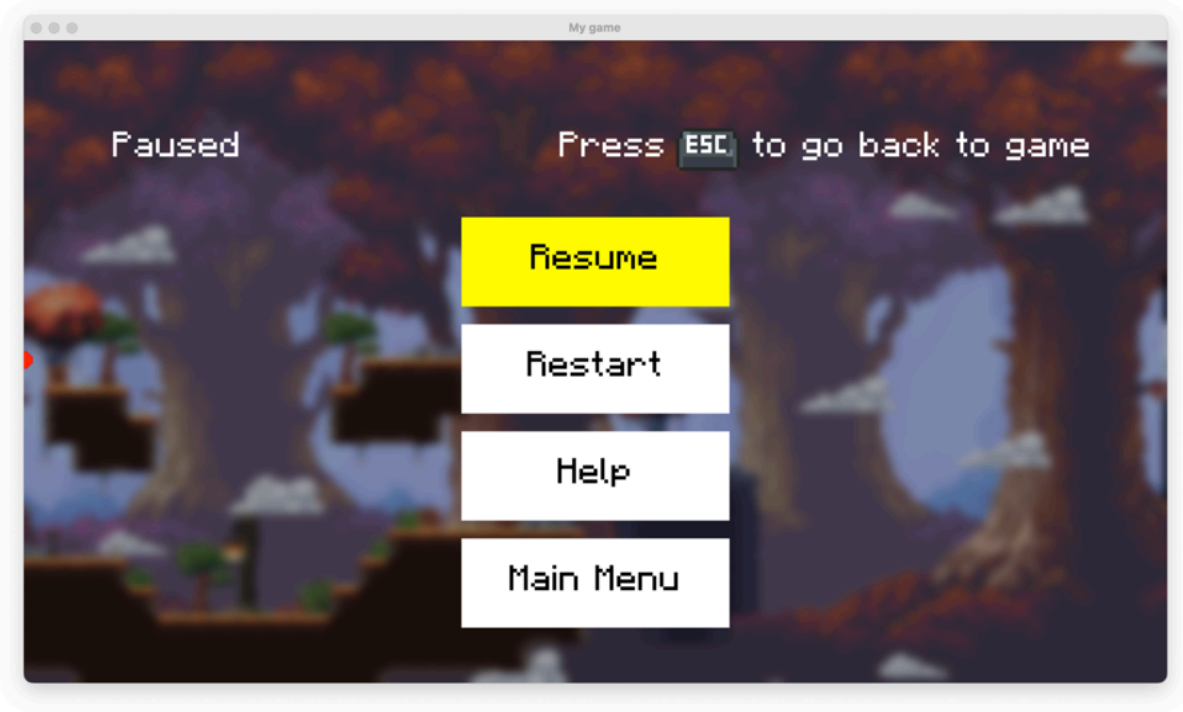
In the second draft, I have included the blurred background underneath as another layer but we definitely need a better font as well as layout.

```
if not self.blurred:
    pygame.image.save(self.display, 'screenshot.png')
    OriImage = Image.open('screenshot.png')

    blurImage = OriImage.filter(ImageFilter.GaussianBlur(2.5))
    #make blurimage darker
    enhancer = ImageEnhance.Brightness(blurImage)
    blurImage = enhancer.enhance(0.7)
    blurImage.save('simBlurImage.png')

    blurImageBG = pygame.image.load('simBlurImage.png')



    self.blurred = True

self.display.blit(blurImageBG, [0, 0])
```

The blurred background is created by screenshotting the previous frame and saving onto the disk, then processed using a series of filters such as gaussian blur and darkening.



Here I have used a stylized, pixelated font as well as a pixel art image that indicates the ESC key of the keyboard.



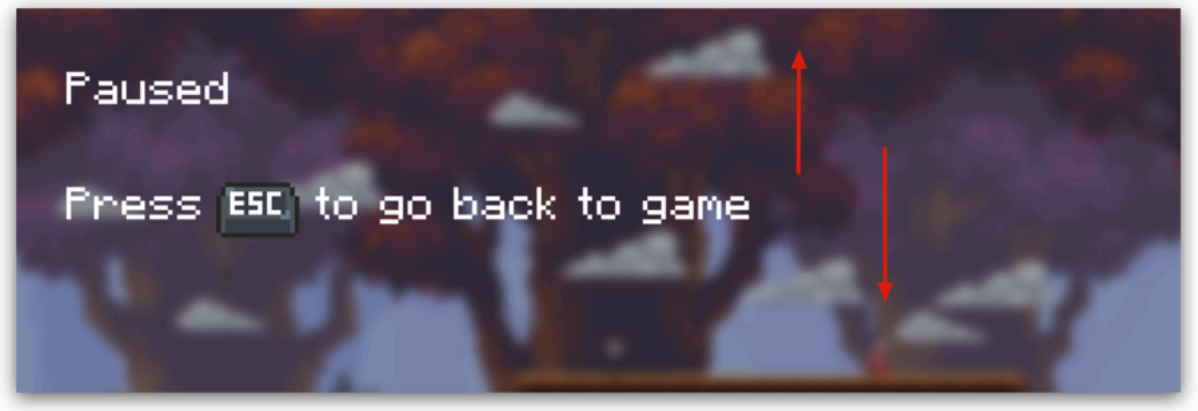
Next, we have a menu that is much more functional, it has four different options rather than just one.

```
self.button_selector = 0
self.menu_buttons = ['Resume', 'Restart', 'Help', 'Main Menu']

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

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                print('game resumed')
                self.reset_movement()
                self.main_game()
            if event.key == pygame.K_DOWN:
                self.button_selector += 1
                self.button_selector %= len(self.menu_buttons)
            if event.key == pygame.K_UP:
                self.button_selector -= 1
                if self.button_selector < 0:
                    self.button_selector = len(self.menu_buttons)-1
                self.button_selector %= len(self.menu_buttons)

    print(f'{self.button_selector} is selected')
```

I have a couple of functions to keep track which option has been selected, although the menu layout has been scrapped, this implementation of the button selector still remains the same in the final build.

Menu Float

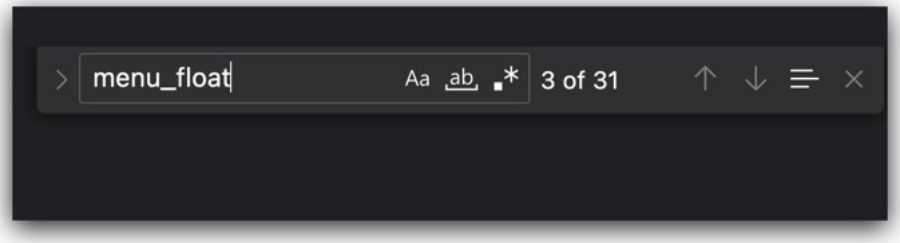
This is a purely visual element, I will explain what it is because it is badly named.



All UI elements, such as buttons, status menus follows a sinusoid shape animation in the y-direction, to give it a ‘breathing’ animation, as if the menu is alive.

```
self.x += 0.05
self.x %= math.pi * 2

self.menu_float = 5*math.sin(self.x)
self.draw_text('Paused', pygame.font.Font('minecraft.otf', 20), 'white', self.display, (50, 50+self.menu_float))
#display this image onto display, assets/keyboard_ESC.png, also 2x bigger
self.display.blit(pygame.transform.scale(pygame.image.load('assets/keyboard_ESC.png'), (int(2*pygame.image.load('assets/keyboard_ESC.png

self.draw_text('Press      to go back to game', pygame.font.Font('minecraft.otf', 20), 'white', self.display, (50, 100+self.menu_float)
self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), self.screenshake_offset)
pygame.display.update()
self.clock.tick(RUNNING_FPS)
```

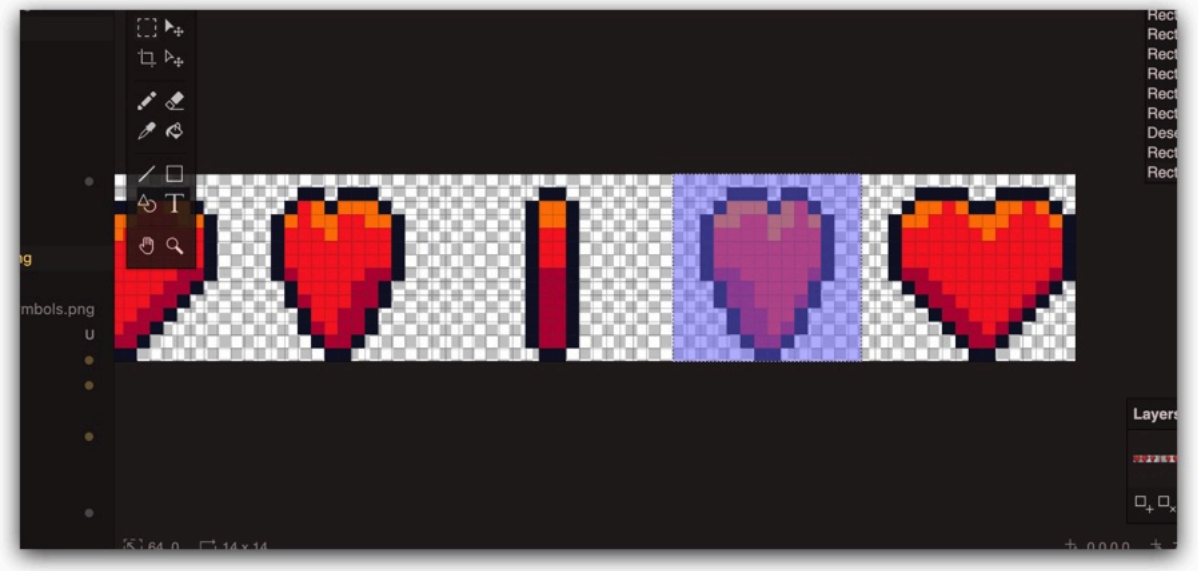
The main implementation is done with the sin function found in the python math library, and the menu_float variable can be globally accessed by any components of the game.



Approximately 31 elements in my game have used this animation.

In-game Player Menu

Using the mockup design, we can start implementing the UI elements accordingly. Immediately, we notice in the bottom left, that we should use a visual element to display the remaining HP available, instead of a text, which is hard to read during a moving battle, for example.



I've chosen to use a spinning heart animation to display the player's HP. To implement this, I've created a new animation class specifically for visual effects. While we already have an animation class for sprites, many functions in it are not relevant for this type of animation, like character states, which don't apply to a looping animation.

Hearts

```
class ObjectAnimation:
    def __init__(self, pos, images):
        self.images = images
        self.index = 0
        self.image = self.images[self.index]
        self.rect = self.image.get_rect(topleft=pos)
        self.animation_time = pygame.time.get_ticks()


    def animate(self, dt):
        if pygame.time.get_ticks() - self.animation_time > 100:  # change 100 to
adjust speed
            self.index = (self.index + 1) % len(self.images)
            self.image = self.images[self.index]
            self.animation_time = pygame.time.get_ticks()

    def draw(self, surface):
        surface.blit(self.image, self.rect.topleft)
```

Firstly, when an animation object is created, all the frames are loaded as a list of images, then the image are accessed from the list using indices.

```
self.images = images
self.image = self.images[self.index]
```

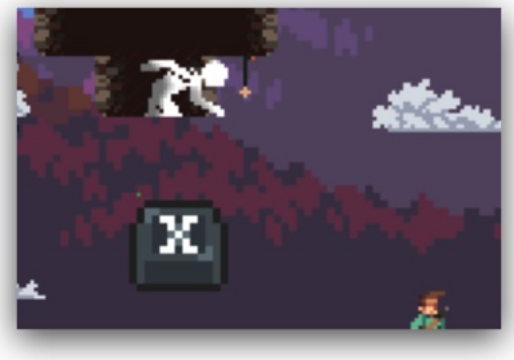
The `animate()` function is called every frame to update the frame position, again, modular arithmetic is used to take care of the looping.

```
for i in range(self.HP):      You, 17 seconds ago • Uncommitted changes
    pos = (i * (self.raw_assets['hearts'][0].get_width()*1.5) + 20, self.display_HEIGHT//15)
    self.heart_animations.append(ObjectAnimation(pos, self.raw_assets['hearts']))
```

For example, this is the code for generating the hearts, the self.HP variable keeps track the player HP, which is used to determined how many hearts are there to be rendered, since this information is updated every frame, hearts are deleted automatically when the player's HP decreases.

CD Block

Now that the hearts are displaying correctly on the screen, we can add a CD duration animation to make it look more interactive. The original plan was to use a progress bar on top of the character's head, but I plan to add another ability to the character later on (super jump), two progress bars would look silly so I need to place the CD animation somewhere else.



The CD animation is now placed at the bottom left corner of the screen, and the key which corresponds to the action is placed again for clarity.

```python
def display_CD_block(self):      You, 1 minute ago • Uncommitted changes
    dash_CD = self.player.get_dash_CD()
    CD_fraction = dash_CD / 60


    BLOCK_SIZE = 32  # For example, adjust this value as needed.

    CD_block = pygame.Surface((BLOCK_SIZE, BLOCK_SIZE * CD_fraction))
    pygame.draw.rect(CD_block, (0, 0, 0), pygame.Rect(0, 0, BLOCK_SIZE, BLOCK_SIZE), )


    CD_block.fill((255, 255, 255))

    CD_block.set_alpha(255)
    self.display.blit(self.keyboard_x, (40, self.display_HEIGHT - 40))

    dash_UI_icon = self.raw_assets['dash_UI_icon']

    CD_block.blit(dash_UI_icon, (0,0), special_flags=pygame.BLEND_RGBA_MULT)

    self.display.blit(CD_block, (35, self.display_HEIGHT - 80))
```

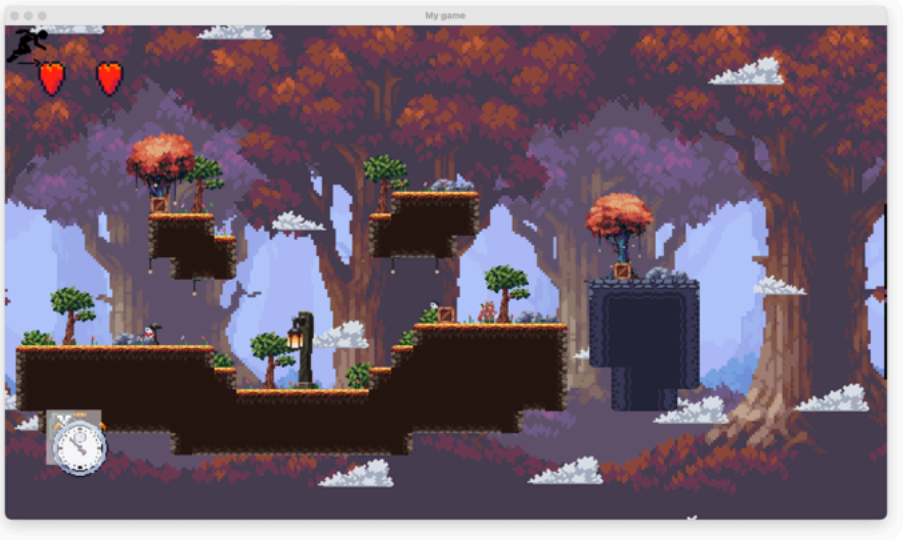
The exact CD value is returned from the player sprite and the CD animation is calculated by a fraction, which provides a visual indicator of how long do we have to wait to use the next skill.

```python
def get_dash_CD(self):
    return 60 - abs(self.dashing)      You, no
```

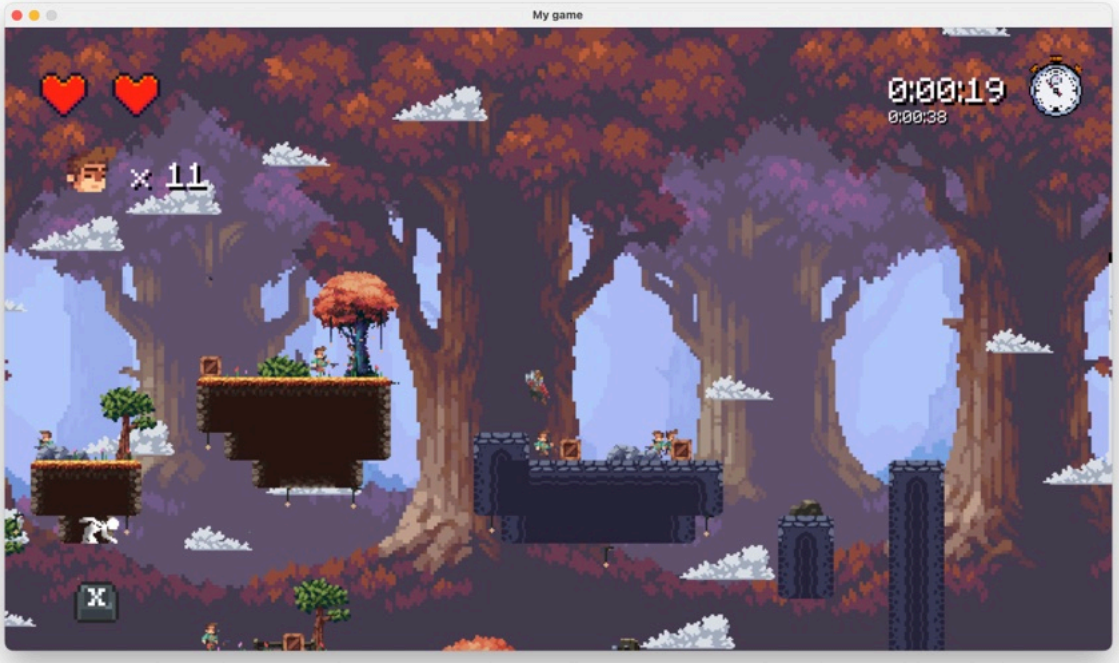
Enemy Count

```
def show_enemy_count(self):
    self.display.blit(self.assets['enemy_head'], (self.display_WIDTH//20*1,self.display_HEIGHT//20*4+self.m
    self.draw_text(f'x {len(self.enemies)}', pygame.font.Font('minecraft.otf', 20), 'black', self.display,
    self.draw_text(f'x {len(self.enemies)}', pygame.font.Font('minecraft.otf', 20), 'white', self.display,
```

This part is similar to the heart animation, enemy's head will be shown on the left to indicate the number of enemy needs to be beat to progress to the next level.



Below is the re-ordered UI.



Timer

```
def running_timer(self):
    self.current_level_time += 1/RUNNING_FPS

    time_text = str(datetime.timedelta(seconds=round(self.current_level_time, 0)))
    self.draw_text(time_text, pygame.font.Font('minecraft.otf', 20), 'black', self.display, (self.display_WIDTH//20
    self.draw_text(time_text, pygame.font.Font('minecraft.otf', 20), 'white', self.display, (self.display_WIDTH//20

    #also display the total time+current level time, in a smaller font, at the top right corner, below the current_
    total_time_text = str(datetime.timedelta(seconds=round(self.total_time+self.current_level_time, 0)))
    self.draw_text(total_time_text, pygame.font.Font('minecraft.otf', 10), 'black', self.display, (self.display_WID

    self.draw_text(total_time_text, pygame.font.Font('minecraft.otf', 10), 'white', self.display, (self.display_WID


    self.display.blit(self.raw_assets['stopwatch'], (self.display_WIDTH//20*19-self.raw_assets['stopwatch'].get_wid
```

The logic behind the timer is a little bit more involved, it consists of a current level time and total time, the total time is the previous's levels time added to the current level time, which is a useful metric if the player is trying to speedrun the game.

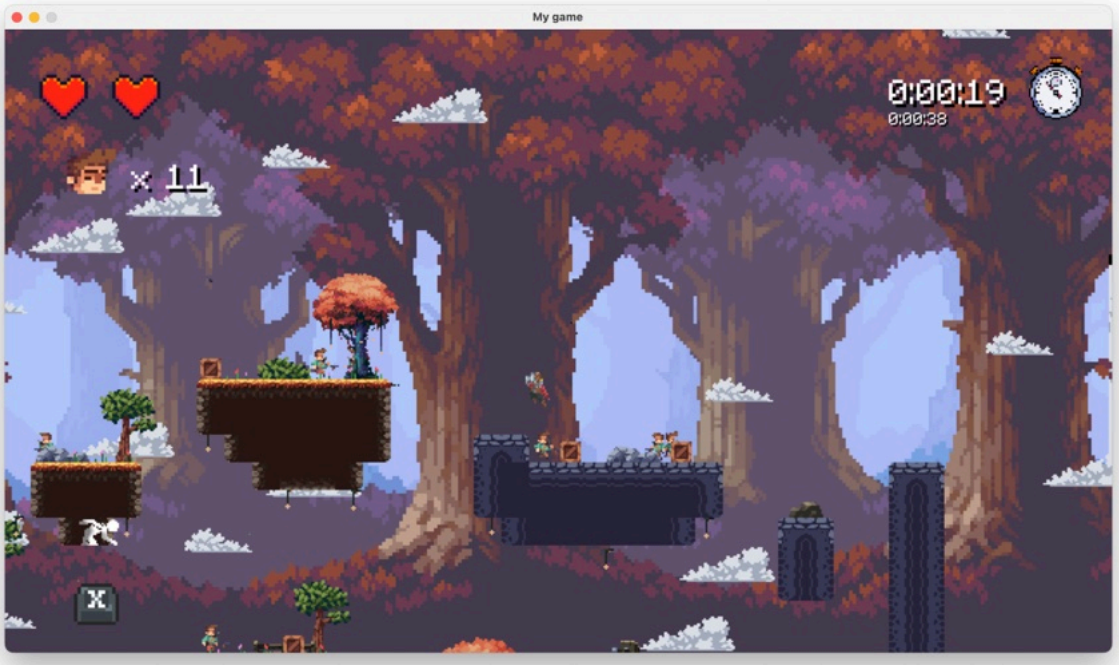
```
def save_global_time(self):
    self.total_time += self.current_level_time      You, 18 seconds ago • Uncommitted
    self.current_level_time = 0

def check_level_loading(self):
    if not len(self.enemies):
        self.transition +=1
        if self.transition >30:
            self.level = min(self.level+1,len(os.listdir('data/maps')))
            self.load_level(self.level)
            self.save_global_time()
    if self.transition <0:
```

(variable) level: int

This is the exact mechanics to calculate global time and current level time.

```
self.total_time = 0
self.current_level_time = 0
```

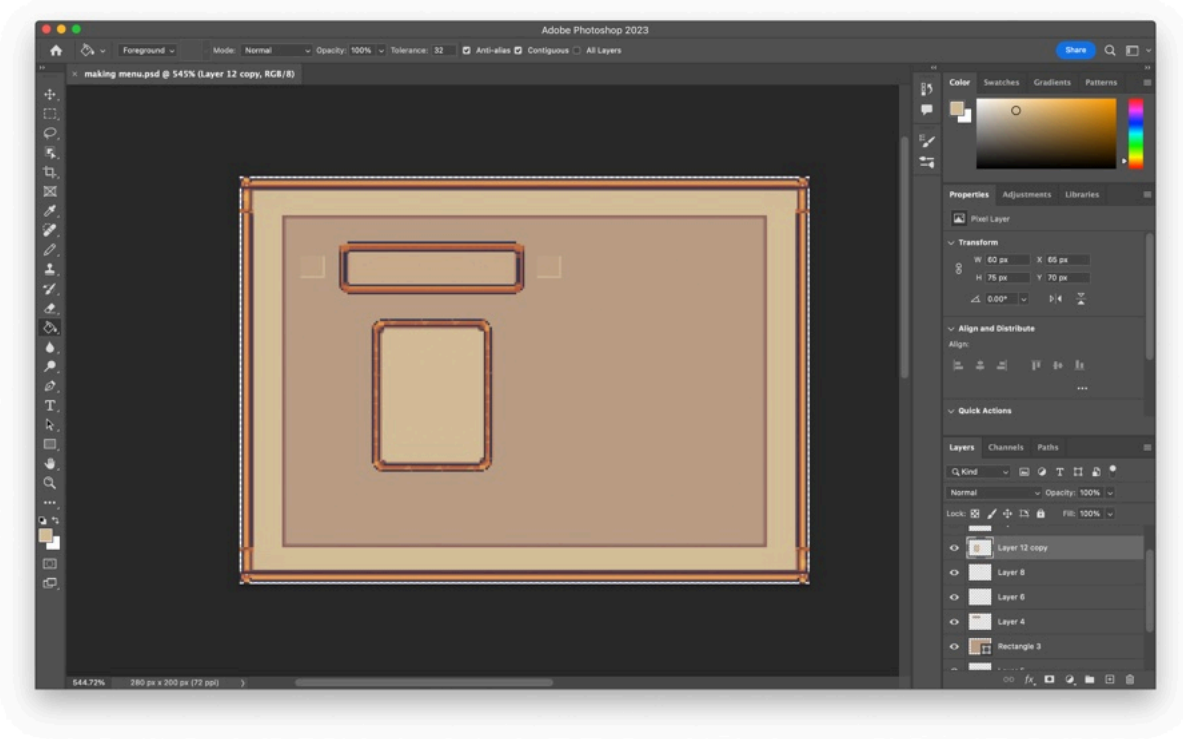
These variables are initialized at the start of the game.


My game
0:00:19
0:00:38
× 11
X

Pause Menu

The Pause menu is a big highlight of my game, it uses a complex method for the animation which makes it look natural.

This is the base image I am working with and the image animations, texts are added in the code.

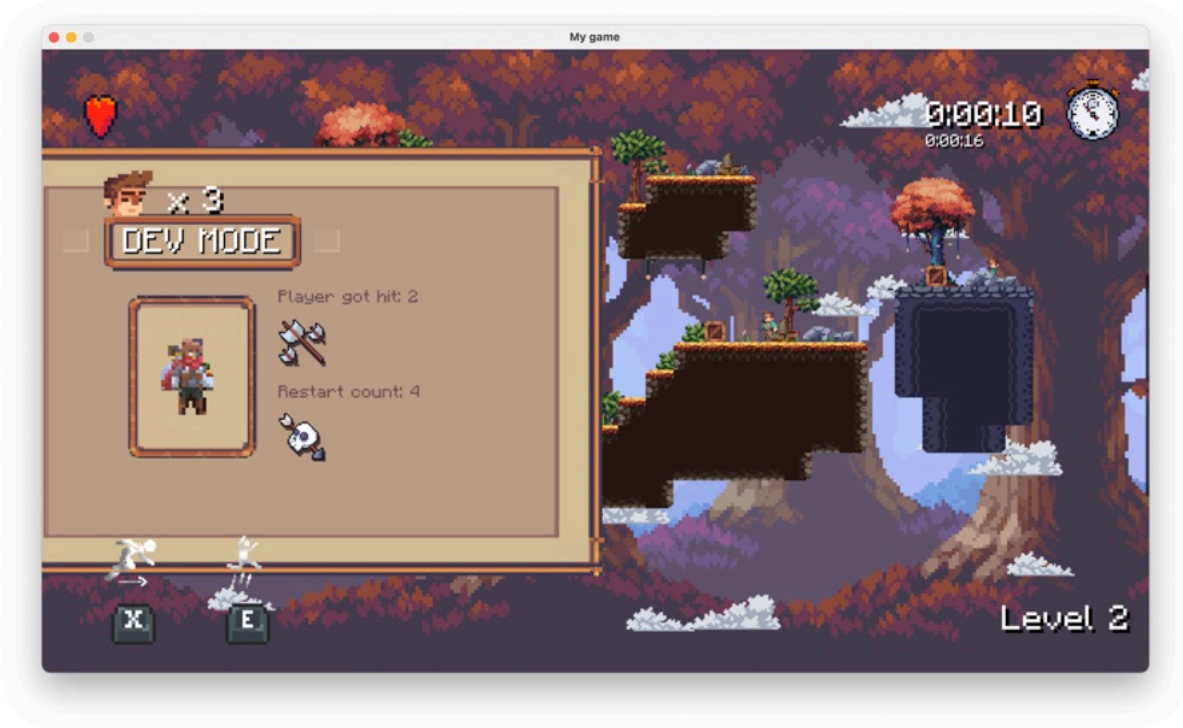


The pause menu is actually it's own game loop, and it's triggered from the main game loop, by setting up two loops, I don't have to explicitly pause the physics simulation in the background, which saves a lot of work.

main_game():

```
if event.key == pygame.K_ESCAPE:
    # print('menu triggered')
    self.show_pause_menu = not self.show_pause_menu
    self.ingame_menu()
```

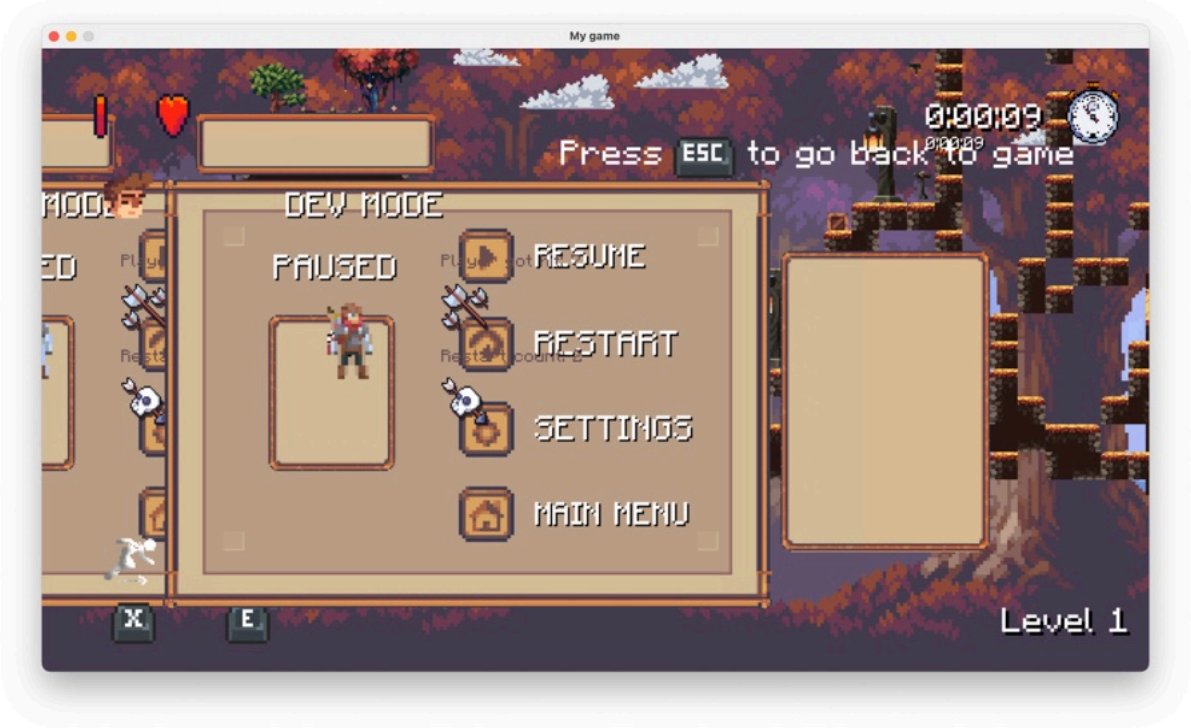
That's the entry point to the pause menu and the pause menu loop takes over from there.



When the ESC key is pressed, the menu slides in from the left side of the screen, but the animation moves linearly and it looks a bit unnatural at the moment, it will be improved later on. The first layout prototype looks like this and it is a bit crowded, it has no room for the buttons so we need to move the objects to somewhere else.

This layout does look better and I just have to need to move the text to where they should be.

However, there is actually something wrong with the background, when I implemented this, the game screenshots the previous frame (the moment when ESC is pressed) and saves it onto the disk, to display behind the menu, otherwise it will end up with a black background.



But things starts to go wrong with when you press the menu button too quickly, the menu from the previous frames are also treated as the background so we end up with a duplicate of the image on the screen. While we could create two layers for the main game and the menu, and we only screenshot the game, but it would be too much work and not worth the time. This could be fixed with a quick trick.

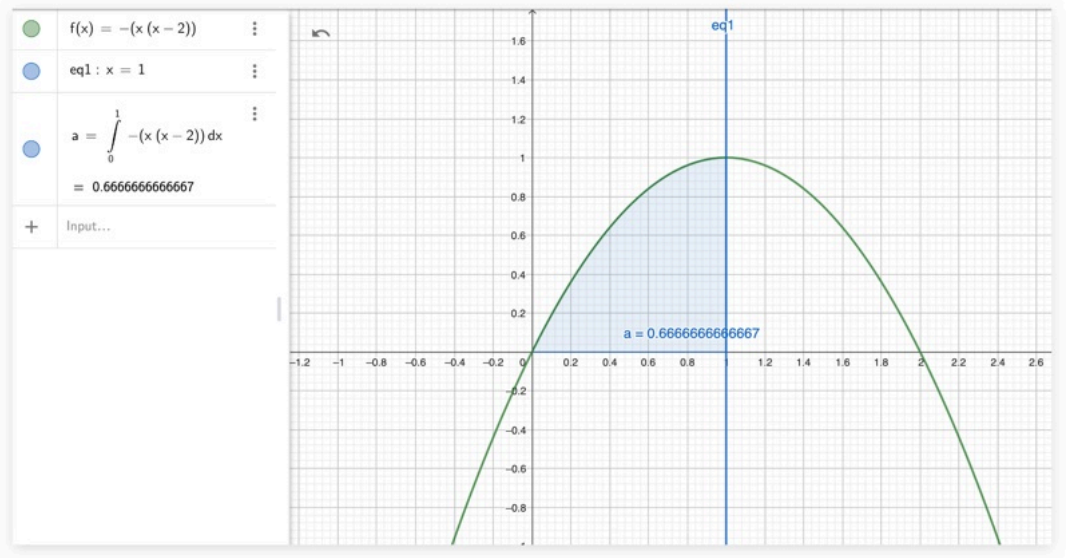
```
if not self.blurred:
    if self.pause_menu_position < -490:      You, now • Uncommitted changes
        pygame.image.save(self.display, 'screenshot.png')
    OriImage = Image.open('screenshot.png')
    screenshot_no_blur = pygame.image.load('screenshot.png')
    blurImage = OriImage.filter(ImageFilter.GaussianBlur(2.5))
    #make blurimage darker
    enhancer = ImageEnhance.Brightness(blurImage)
    blurImage = enhancer.enhance(0.7)
    blurImage.save('simBlurImage.png')

    blurImageBG = pygame.image.load('simBlurImage.png')
    self.blurred = True
```

We only take the screenshot when the menu is to the very left so that whenever it screenshot the image, it does not contain our pause menu.

Sliding Animation

This is the part that I'm the most proud with this project, It uses the bezier and accelerated movement talked about in our design stage to give a smooth animation. The basic idea behind the ease_out_quad function is that it should produce an eased output between 0 and 1 when given an input between 0 and 1.

```
def ease_out_quad(t):
    """Quadratic easing out - decelerating to zero velocity."""
    return -t * (t - 2)
```



The easing animation has this shape, the region in blue is the acceptable range of inputs

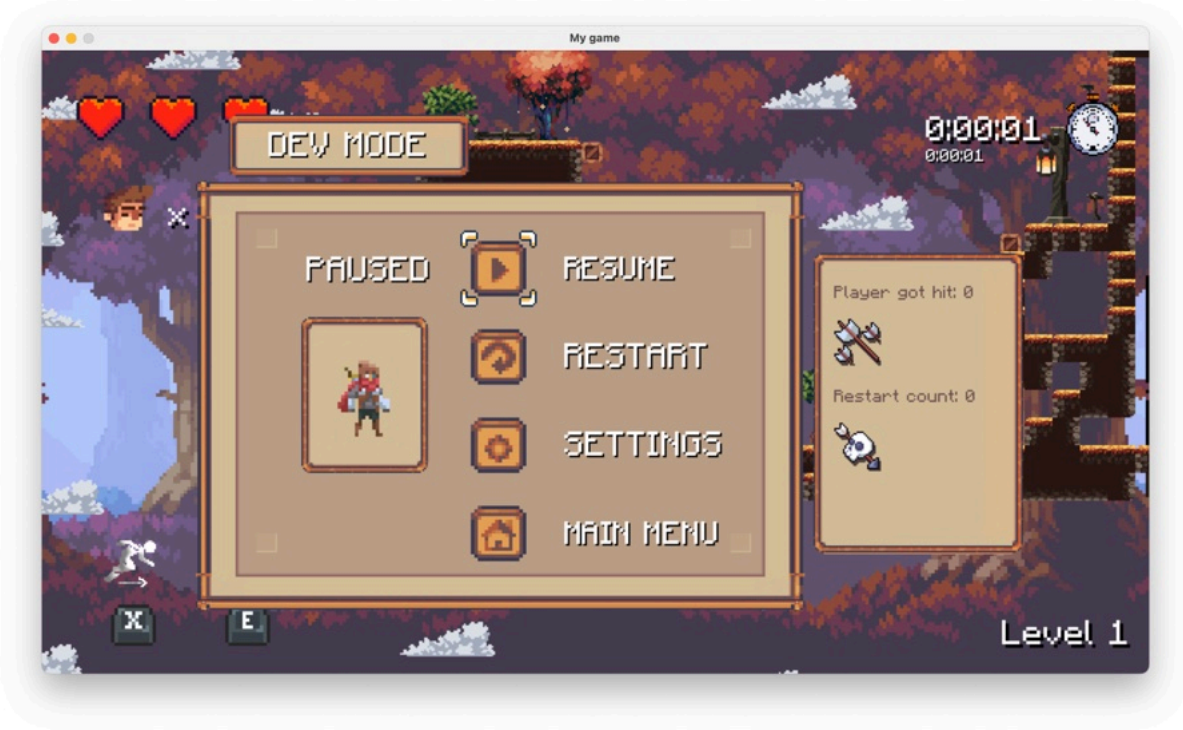
```
self.show_status_menu = False
self.status_menu_position = self.display_WIDTH  # Start off-screen to the right
self.status_menu_target_position = (self.display_WIDTH - self.raw_assets['status_menu'].get_width()) // 2
```

```
def render_status_menu(self):
    speed = 10
    if self.show_status_menu:
        progress = ease_out_quad((self.status_menu_position - self.status_menu_target_position) / 100)
        self.status_menu_position -= speed * progress
        # Cap the position to the target if it overshoots due to the easing function
        if self.status_menu_position < self.status_menu_target_position:
            self.status_menu_position = self.status_menu_target_position
    else:
        progress = ease_out_quad((self.display_WIDTH - self.status_menu_position) / 100)      You, 1 second ago • Uncommitted changes
        self.status_menu_position += speed * progress
        # Cap the position to the screen width if it overshoots due to the easing function
        if self.status_menu_position > self.display_WIDTH:
            self.status_menu_position = self.display_WIDTH

    # Use the status_menu image, and display it at the status_menu_position, display_height/2
    self.display.blit(self.raw_assets['status_menu'], (self.status_menu_position, self.display_HEIGHT/2-self.raw_assets['status_menu'].ge
```

This is hard to demonstrate with still images, as it's an animation, but I will try to include a video demo.

Button Selector

We need a visual indicator to show which option has been selected, I will add the same easing animation to the button selector as I did for the sliding animation.



I will use modular arithmetic to keep track of the currently selected button, the button resets to the top once it reaches the bottom and key down is pressed.

```
if event.key == pygame.K_DOWN:
    button_selector += 1
    button_selector %= len(menu_buttons)
if event.key == pygame.K_UP:
    button_selector -= 1
    if button_selector < 0:
        button_selector = len(menu_buttons) - 1
    button_selector %= len(menu_buttons)
```

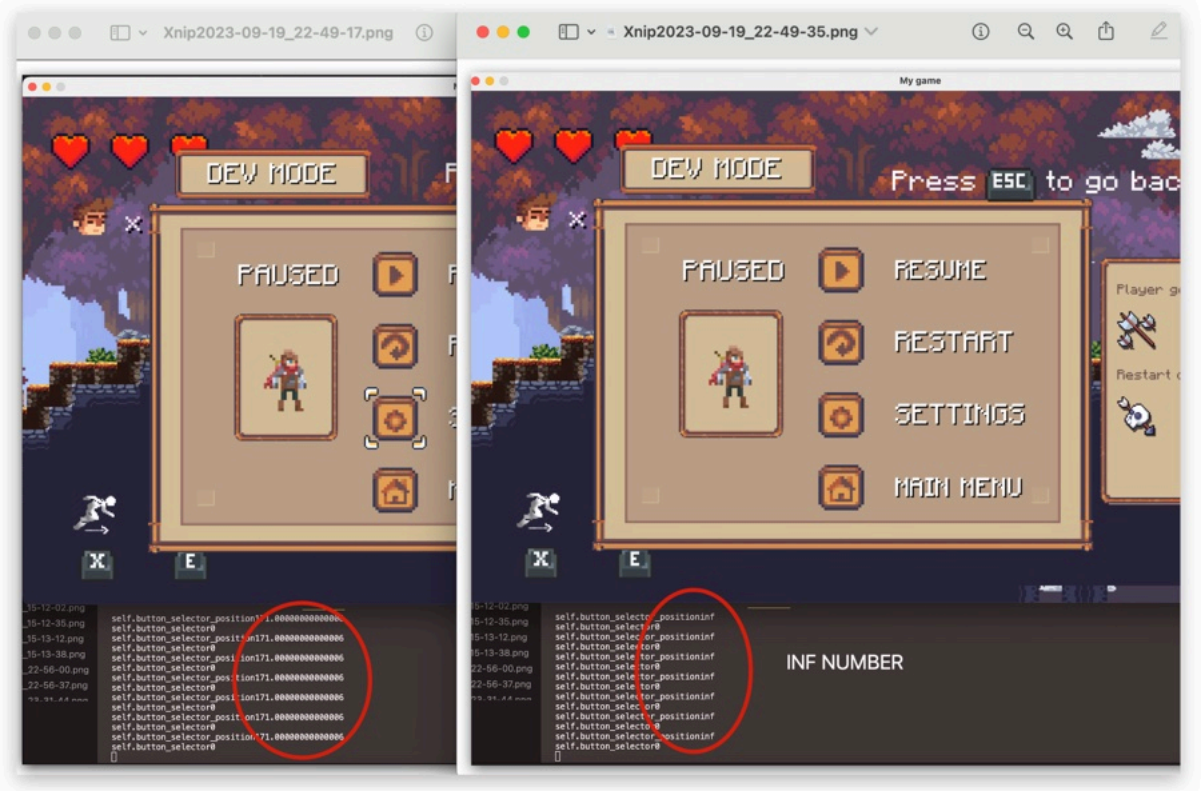
Button Selector ID	Button	Function called
0	Resume	Exits Menu, main_game() called
1	Restart	restart_level()
2	Settings	show_debug_menu change state
3	Main Menu	main_menu()

Code for calling the events upon pressing:

```
if event.key == pygame.K_RETURN:
    if self.menu_buttons[self.button_selector] == "Resume":
        self.show_pause_menu = False
        self.reset_movement()
        self.main_game()
    elif self.menu_buttons[self.button_selector] == "Restart":
        self.restart_level()
    elif self.menu_buttons[self.button_selector] == "Options":
        self.show_debug_menu = not self.show_debug_menu
    elif self.menu_buttons[self.button_selector] == "Main Menu":
        self.entire_game_reset()
        self.main_menu()
```

Code for animating the button selector, non-linearly.

```
button_selector_target_position = 69 + button_selector * 51
difference = button_selector_target_position - button_selector_position
# Normalize the difference for the easing function
normalized_difference = abs(difference) / 51.0
# Calculate progress using the easing function
progress = ease_out_quad(normalized_difference)
if difference > 0:
    button_selector_position += button_selector_speed * progress
    if button_selector_position > button_selector_target_position:
        button_selector_position = button_selector_target_position
else:
    button_selector_position -= button_selector_speed * progress
    if button_selector_position < button_selector_target_position:
        button_selector_position = button_selector_target_position
```



However, there seems to be an issue where the button selector just flies out of the screen when I loop through the options. When the difference between the target and current position is small, the progress variable should converge to 0, slowing down the movement. Instead, it appears that it's increasing without bounds. This happens because when updating the difference, numerical errors can appear.

```
normalized_difference = abs(difference) / 51.0
```

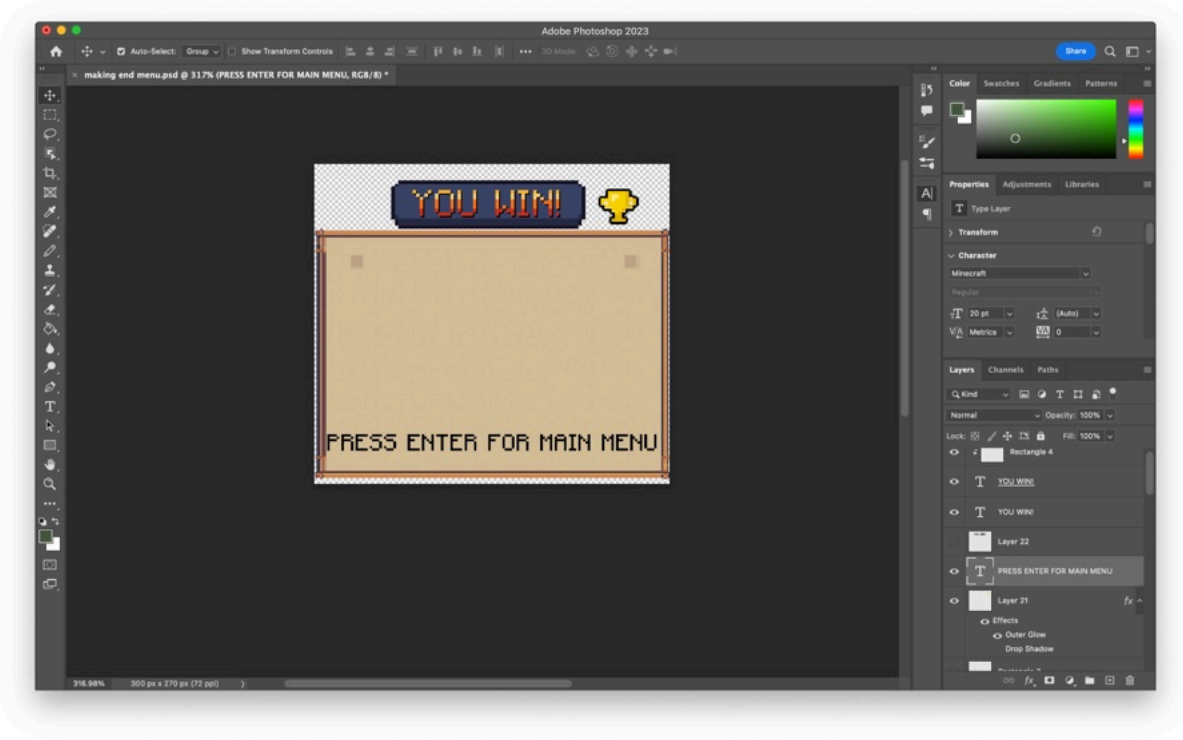
We are using the division operating to calculate the normalized difference, as we know, computers are not good at floating point division when it comes to precision. It may go over 1 or go into negative the values, in which case doesn't make any sense to the `ease_out_quad` function, as it only accepts input values from 0 -> 1. But this could be fixed quite easily with a boundary check, implemented below.

```
# If the normalized difference is out of bounds, correct it
if normalized_difference > 1:
   normalized_difference = 1
elif normalized_difference < 0:
   normalized_difference = 0
```

There is a still a subtle thing that I should be aware of, when I loop the selector from top to bottom OR bottom to top. The animation appeared to be very slow as it is designed to navigate between adjacent blocks, but not from top to bottom. Therefore, when travelling a distance that is larger than the gap between adjacent blocks, the selector speed gets increased so it looks like they are reaching the target with the same time.

```
if normalized_difference > 2:
   button_selector_speed = 20
else:
   button_selector_speed = 7
```

Game End Screen & Saving Player Scores



The ending screen displayed to the player after completing their run includes information such as the time elapsed and any penalties incurred. It contains several images to display it more graphically.

```
    self.textflash += 1
    if self.textflash > 60:
        basetime = round(self.total_time, 1)
        self.draw_text(f'{basetime}', self.font, 'black', game_end_panel, (text_pos[0],
227))
        self.draw_text(f'{basetime}', self.font, 'white', game_end_panel, (text_pos[0] - 1,
226))
```

Although hard to showcase using still images, but the texts have a flashing animation open showing to the user.

```
    if self.textflash % 60 < 30:
        text, text_width, text_pos = 'PRESS ENTER FOR MAIN MENU', self.font.size(text)[0],
(game_end_panel.get_width() // 2 - text_width // 2, game_end_panel.get_height() - 50)
        self.draw_text(text, self.font, 'black', game_end_panel, (text_pos[0] + 1,
text_pos[1] + 1))
        self.draw_text(text, self.font, 'white', game_end_panel, text_pos)

    self.display.blit(game_end_panel, render_pos)
```

Since the self.textflash is keep incrementing, I want it to show on the screen only for half of the time, thus creating a ‘blinking’ effect.

After the End screen has been shown, a copy of the current player data is written to a file called player_records.json.

It has 6 different attributes:

1. player_name
2. total_time, which equals the ‘raw’ time the player took to beat the level plus the time penalty.
3. raw_time, player_got_hit_count and restart_count just to keep track of data as a ‘backup’
4. the date and time which the player completed this run.

```
player_record_entry = {
    'player name': self.player_name,
    'total time': round(self.total_time + 5 * self.restart_count + self.player_got_hit_count, 1),
    'raw time': round(self.total_time, 5),
    'got hit count': self.player_got_hit_count,
    'restart count': self.player_got_hit_count,
    'completed datetime': f'{today} @ {current_time}'
}

# Proper way to update the JSON file
with open('data/player_records.json', 'r') as f:
    try:
        data = json.load(f)
    except json.decoder.JSONDecodeError:  # In case the file is empty or has invalid JSON format
        data = []

data.append(player_record_entry)
```

I have chosen to store these data in a dictionary first because the python dictionary has a very similar structure to the .json file format, so it is quite easy to convert between the files.

```
  File "/Users/dyrox/Desktop/School/CS Project 2/game.py", line 986, in <module>
    Game().main_menu()
  File "/Users/dyrox/Desktop/School/CS Project 2/game.py", line 958, in main_menu
    self.render_leaderboard()
  File "/Users/dyrox/Desktop/School/CS Project 2/game.py", line 871, in render_leaderboard
    data = json.load(f)
  File "/Users/dyrox/anaconda3/lib/python3.10/json/__init__.py", line 293, in load
    return loads(fp.read(),
  File "/Users/dyrox/anaconda3/lib/python3.10/json/__init__.py", line 346, in loads
    return _default_decoder.decode(s)
  File "/Users/dyrox/anaconda3/lib/python3.10/json/decoder.py", line 340, in decode
    raise JSONDecodeError("Extra data", s, end)
json.decoder.JSONDecodeError: Extra data: line 8 column 2 (char 177)
```

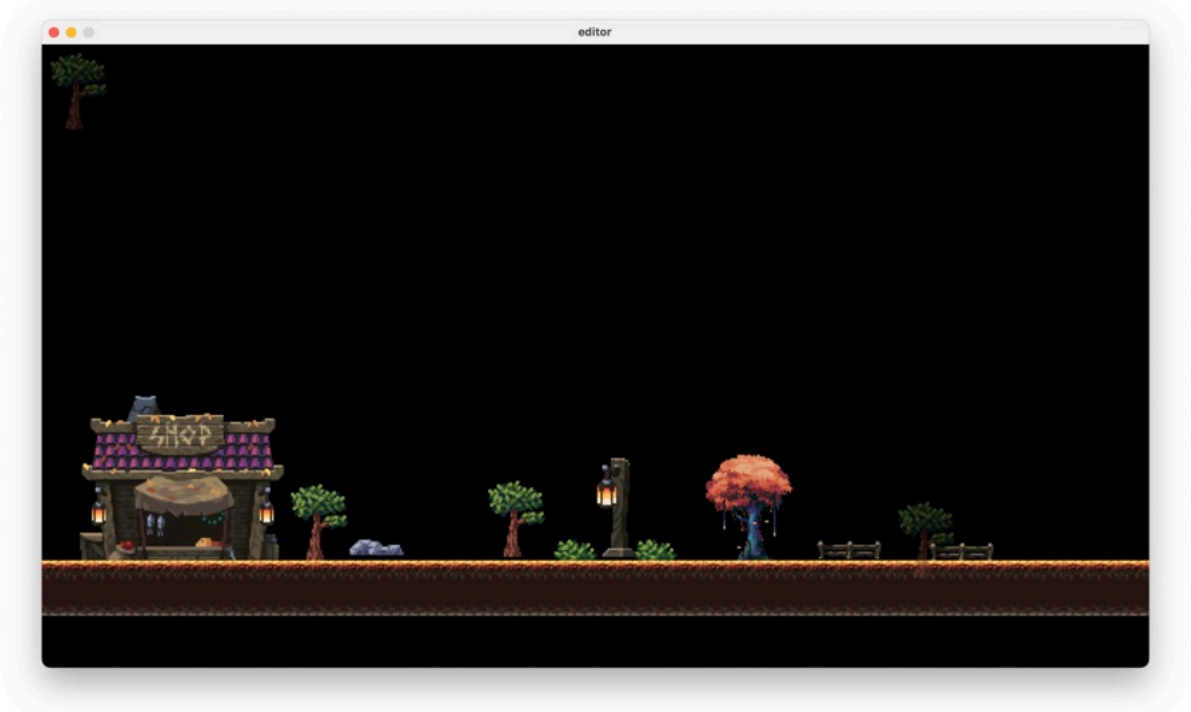
Also, if the .json file hasn’t been created or damaged, the game recognizes it automatically and creates a new file to prevent it from crashing.

```python
def game_end(self):
    today = datetime.date.today()
    current_time = datetime.datetime.now().strftime('%H:%M')

    player_record_entry = {
        'player name': self.player_name,
        'total time': round(self.total_time + 5 * self.restart_count + self.player_got_hit_count,1),
        'raw time': round(self.total_time,5),
        'got hit count': self.player_got_hit_count,
        'restart count': self.player_got_hit_count,
        'completed datetime': f'{today} @ {current_time}'
    }

    # Proper way to update the JSON file
    with open('data/leaderboard.json', 'r') as f:
        try:
            data = json.load(f)
        except json.decoder.JSONDecodeError: # In case the file is empty or has invalid JSON format
            data = []

    data.append(player_record_entry)

    with open('data/leaderboard.json', 'w') as f:
        json.dump(data, f, indent=4)
```

Main Menu

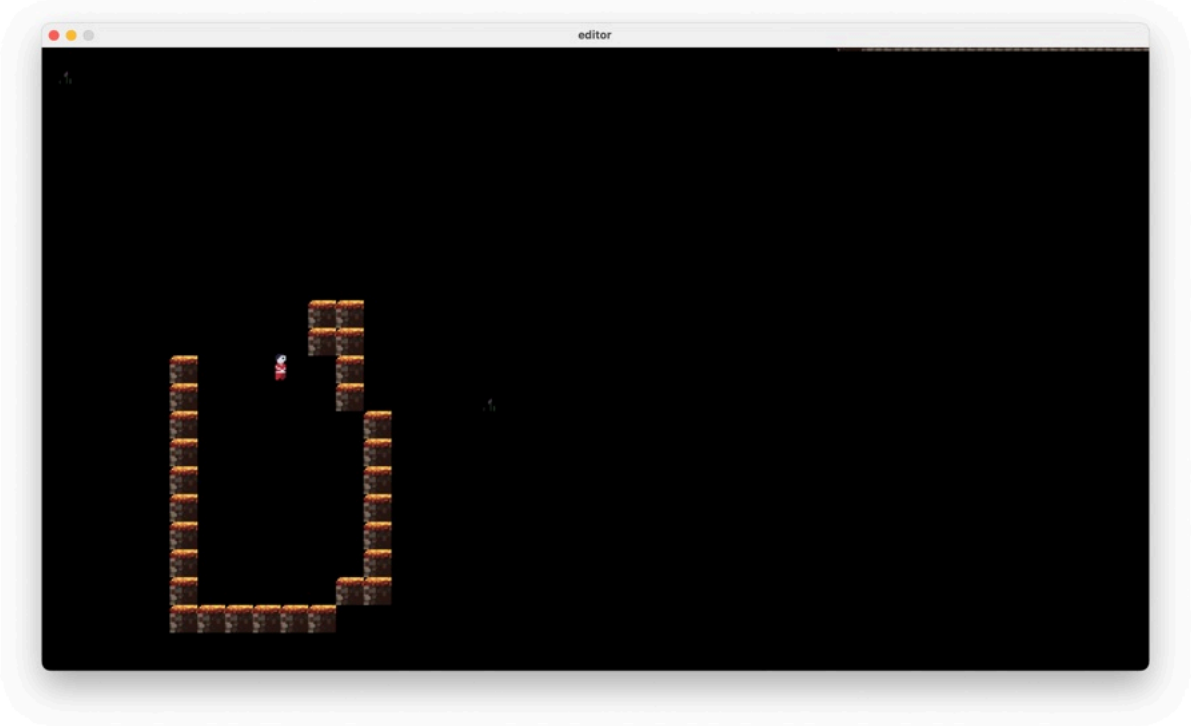
Scrolling Animation

For the main menu, instead of just using a stationary image for the background, which would look okay but quite boring, I decided to incorporate a moving animation in the background, much like the game Minecraft.



With the existing main game engine, there is no need to write the scrolling background animation from scratch, I can simply create a level of flat ground and a few decorations. This

runs like a normal level but the character inputs are disabled and a few other things are overlaid on the screen (such as the menu panel and leaderboard), to make it look like a proper main menu.



Because the main menu map still runs like a normal level, it does have update functions that check if all the enemies have been defeated to move onto the next level, but I don't want to do that, instead, I used a sneaky trick by placing the enemy into a empty container far away that is not visible to the user, the enemy will be not defeated and the level won't process, which is what we want

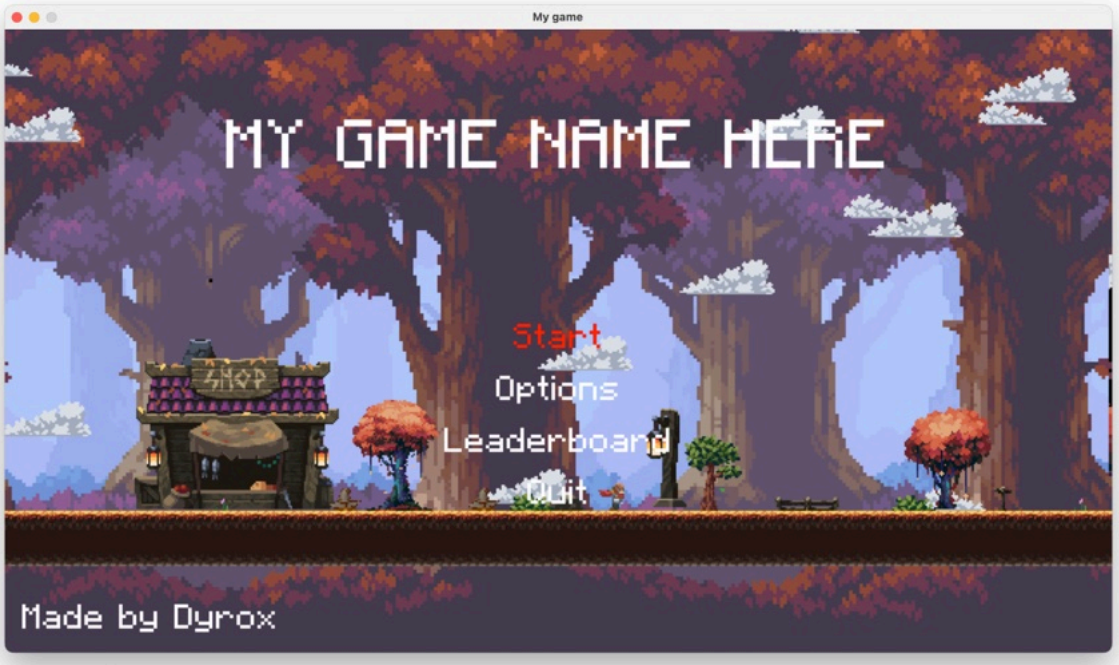
```
def main_menu(self):

    self.game_finished = False
    self.in_main_menu = True
    self.textflash = 0

    self.leaderboard_position = self.assets['leaderboard'].get_width() + self.display_WIDTH
    self.leaderboard_target_position = self.display_WIDTH // 20 * 15 - self.assets['leaderboard'].get_width() // 2

    self.load_level('main_menu_map')
    # Load background image (Make sure you have a `background.jpg` in your assets)      You, 6 days ago • fixed start menu, ma
    background = pygame.image.load('data/images/background.png').convert()
    background = pygame.transform.scale(background, (self.display_WIDTH, self.display_HEIGHT))

    # Define the buttons
    self.menu_buttons = ["Start", "Options", "Leaderboard", "Quit"]
    self.button_selector = 0
```

The entry point is the main_menu() loop, which is called when the Game object is initialized, so the user will be directed to the main menu first.

All the animations are working correctly, but the layout is a little strange, from my experience of doing UI designs, I need to move the animation away from the user's sight and let them focus on the menu buttons.

```
def scroll_update(self):      You, 2 weeks ago • menu is working but very ugly …

    # if in_main_menu: the camera should be higher
    if self.in_main_menu:
        self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2 - self.scroll[0]) // 30 - 4
        self.scroll[1] += (self.player.rect().centery - self.display.get_height() / 2 - self.scroll[1]) // 30 - 3
    else:
        self.scroll[1] += (self.player.rect().centery - self.display.get_height() / 2 - self.scroll[1]) // 30
        self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2 - self.scroll[0]) // 30

    self.render_scroll = [int(self.scroll[0]), int(self.scroll[1])]
```

So I added this script to shift the camera if the user is in the main menu, making room for the buttons. This was done by adding a specific value


My game
MY GAME NAME HERE
Start
Options
Leaderboard
Quit
Made by Dyrox

Leaderboard

As we have decided previously in the design stage, we will have a leaderboard that shows up on the main menu, indicating which player has the best run.

Loading a blank leaderboard image is pretty easy, and let's think about how to display the correct data on the leaderboard. (P.S. I have shifted the position of the buttons and logo away from the centre to make space for the leaderboard)

```
1088        with open("data/player_records.json", "r") as f:
1089            data = json.load(f)
1090
1091        # 2. Select the Best 9 Entries
1092        sorted_data = sorted(data, key=lambda x: x["total time"])[:9]
```

We'll start by loading the leaderboard data from the data/player_records.json file. And We'll sort the data based on the total time in ascending order (as lower time is better) and pick the top 9.

```
text = f"{index}. {entry['player name']} {total_time_formatted}"
text_surface = font.render(text, True, (255, 255, 255))  # Render in white color, adjust as needed
text_rect = text_surface.get_rect(topleft=(start_x, start_y + index * gap))
# add another black shadow layer underneath the text
text_surface_shadow = font.render(text, True, (0, 0, 0))
text_rect_shadow = text_surface_shadow.get_rect(topleft=(start_x + 1, start_y + index * gap + 1))
leaderboard.blit(text_surface_shadow, text_rect_shadow)

leaderboard.blit(text_surface, text_rect)
```

We'll iterate over the selected entries and render them on the leaderboard using leaderboard.blit, although the text rendered where they were supposed to be with the correct starting position and gaping, but there is a slight issue though.

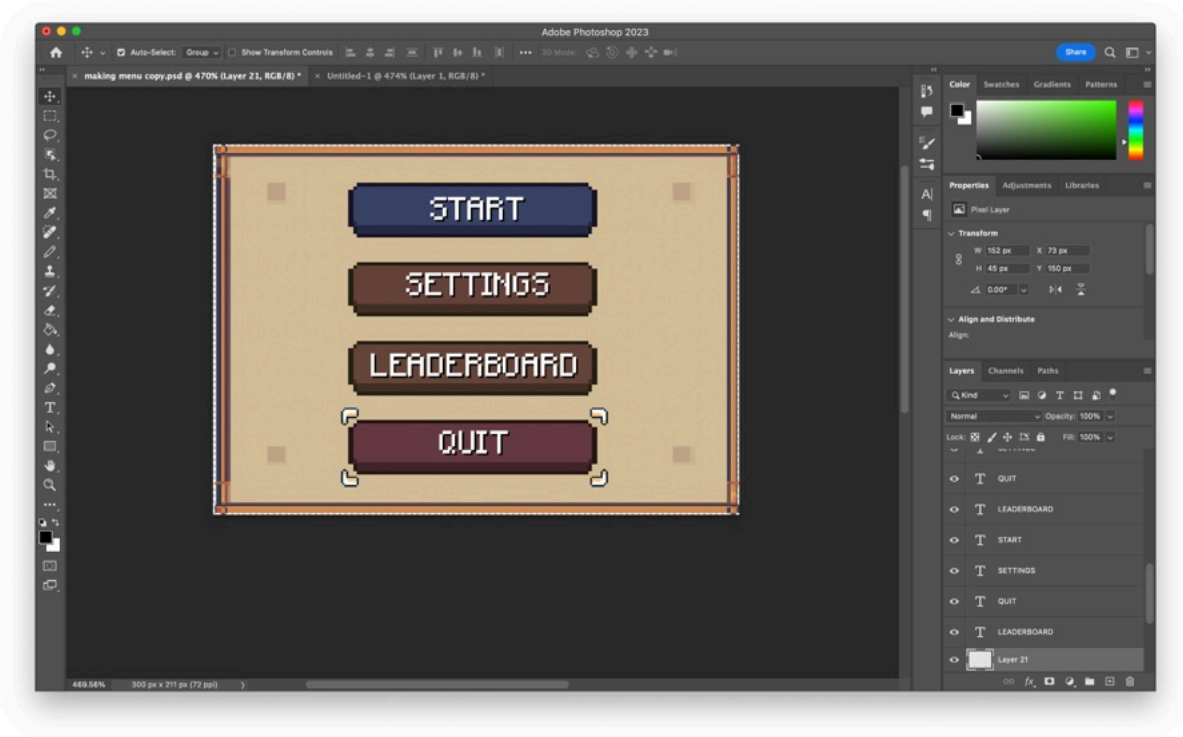
When we have a runtime that is longer than a minute, the score is still displayed in purely seconds, but we want a mm:ss.s format because it is much easier to read, which can be fixed quickly using modular arithmetic.

```
# 3. Display the Entries on the Leaderboard
for index, entry in enumerate(sorted_data, 1):  # Start index from 1
    total_time_seconds = entry['total time']
    # format the time into minutes and seconds
    if total_time_seconds >= 60:
        if total_time_seconds % 60 < 10:
            total_time_formatted_seconds = f'0{int(total_time_seconds % 60)}'
        else:
            total_time_formatted_seconds = int(total_time_seconds % 60)
        fraction_part = round(total_time_seconds - int(total_time_seconds),1)
        fraction_part = str(fraction_part)[2:]
        total_time_formatted = f'{int(total_time_seconds // 60)}:{total_time_formatted_seconds}.{fraction_part}'
    else:
        total_time_formatted = total_time_seconds
```

instant fix.

Main Menu Panel

Similar to the in-game menu. It consists of two parts, one is the menu panel pre-made in Photoshop, and the other is the button selector which moves according to which button is selected.



To implement it into the game, I need to list to keep track of which button is selected.

```
# Define the buttons
self.menu_buttons = ["Start", "Options", "Leaderboard", "Quit"]
self.button_selector = 0

self.main_menu_button_selector_position = 16
self.main_menu_button_selector_target_position = 16*self.button_selector
```

As well as to deal with the keyboard actions when ENTER is pressed, by pressing start, it sets the self.in_main_menu flag to false to stop rendering things that are not related to the main game (such as leaderboard), it resets both the game and the player status so the progress starts from scratch again.
Pressing options launches the debug menu.

```
            if event.key == pygame.K_RETURN:
                if self.menu_buttons[self.button_selector] == "Start":
                    self.load_level(self.level)
                    self.in_main_menu = False
                    self.reset_movement()
                    self.reset_player_status()
                    self.show_status_menu = False
                    self.main_game()

                    # print(self.level)

                elif self.menu_buttons[self.button_selector] == "Options":
                    self.show_debug_menu = not self.show_debug_menu
                    print('debug menu show has been set to', self.show_debug_menu)
                elif self.menu_buttons[self.button_selector] == "Leaderboard":
                    # Implement the Leaderboard functionality here
                    pass
                elif self.menu_buttons[self.button_selector] == "Quit":

                    pygame.quit()
                    sys.exit()
```

When the proper menu panel replaces the white pixel text, the main menu is now looking a lot more polished now.

Debug Menu

```
player.air_time is 2
player.air_time is 3
player.air_time is 4
player.air_time is 0
player.air_time is 1
player.air_time is 2
player.air_time is 3
player.air_time is 4
player.air_time is 0
player.air_time is 1
player.air_time is 2
player.air_time is 3
player.air_time is 4
```

Whenever I want to debug my code, I have to manually output variables to the terminal to see their values, which is quite inconvenient considering that I have to switch windows every time I do that. Instead, I wrote a debug menu that directly runs within the pygame process and displays the values I want, there are two benefits to this, first I don't have to switch in between windows which saves time, and second, the variables replace their values and overwrites the old value, which is much easier to read compared to the huge wall of text the terminal produces.

```python
def update_debug_stuff(self):
    self.debug_surface = pygame.Surface((self.screen.get_width(), self.screen.get_height()), pygame.SRCALPHA)

    currentFPS = f'FPS: {self.clock.get_fps():.0f}'

    if self.show_debug_menu:
        # display different variables

        self.draw_text(currentFPS, self.pygame.font.Font(None, 30), 'white', self.debug_surface,
                       (self.debug_surface.get_width() - 100, 0))

        self.draw_text(f'self.level: {self.level}', self.pygame.font.Font(None, 30), 'white', self.debug_surface, (0, 0))
        self.draw_text(f'self.pause_menu_position: {self.pause_menu_position}', self.pygame.font.Font(None, 30), 'white',
                       self.debug_surface, (0, 25))
        self.draw_text(f'self.transition: {self.transition}', self.pygame.font.Font(None, 30), 'white', self.debug_surface, (0, 50))
        self.draw_text(f'self.dead: {self.dead}', self.pygame.font.Font(None, 30), 'white', self.debug_surface, (0, 75))
        self.draw_text(f'self.restart_count: {self.restart_count}', self.pygame.font.Font(None, 30), 'white', self.debug_surface,
                       (0, 100))
        self.draw_text(f'self.player_got_hit_count: {self.player_got_hit_count}', self.pygame.font.Font(None, 30), 'white',
```

(method) def get_height() -> int

This function updates the debug information displayed on the screen by creating a new surface to display the debug information and draws various variables related to the game state onto it. Variables like the current, frames per second, the player's position and status, the level's state, and various other game variables. This method is called every frame by the main game loop when the debug menu is enabled, which can be triggered either through the in-game menu setting button or the settings on the main menu.

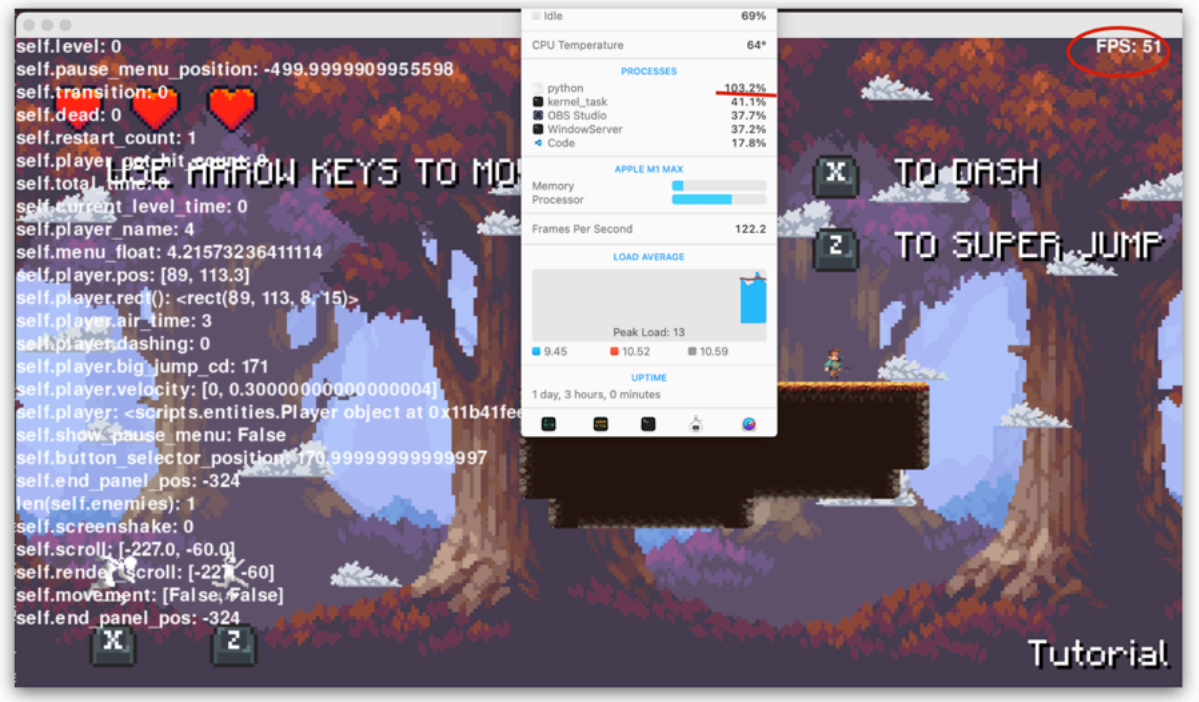
Great, now we have the debug menu up and running, however, there is an issue, when the debug menu runs, the FPS drops significantly and slows the game, what should I do to improve the efficiency? I did some experiments and found out that it was because I was re-initializing the font every time I render a text object, since there is a large amount of text, it slows the game drastically. Instead of using pygame.font.Font(None, 30) for all text objects, I initialized the font at the start of the game and just use the object whenever needed.

```
self.debug_font = pygame.font.Font(None, 30)
```

```
if self.show_debug_menu:
    # display different variables

    self.draw_text(currentFPS, self.debug_font, 'white', self.debug_surface,
                   (self.debug_surface.get_width() - 100, 0))

    self.draw_text(f'self.level: {self.level}', self.debug_font, 'white', self.debug_surface, (0, 0))
    self.draw_text(f'self.pause_menu_position: {self.pause_menu_position}', self.debug_font, 'white',
                   self.debug_surface, (0, 25))
    self.draw_text(f'self.transition: {self.transition}', self.debug_font, 'white', self.debug_surface, (0, 50))
    self.draw_text(f'self.dead: {self.dead}', self.debug_font, 'white', self.debug_surface, (0, 75))
```

Code Refactoring, Profiling & Optimization

With the font out of the way, we realize that the game is running still with some level of lag. Even though our Python interpreter is running at its full speed (100% of the single core is used).



Luckily for us, we have a library that analyses the performance of the code called pstats. It is able to run the program while tracing the resources that each function uses.

```
if __name__ == "__main__":
    cProfile.run('Game().main_menu()', 'profile_results.prof')


import pstats

p = pstats.Stats('profile_results.prof')
p.sort_stats('cumulative').print_stats(10)  # This will sort by cumulative time and print the top 10 functions.
```

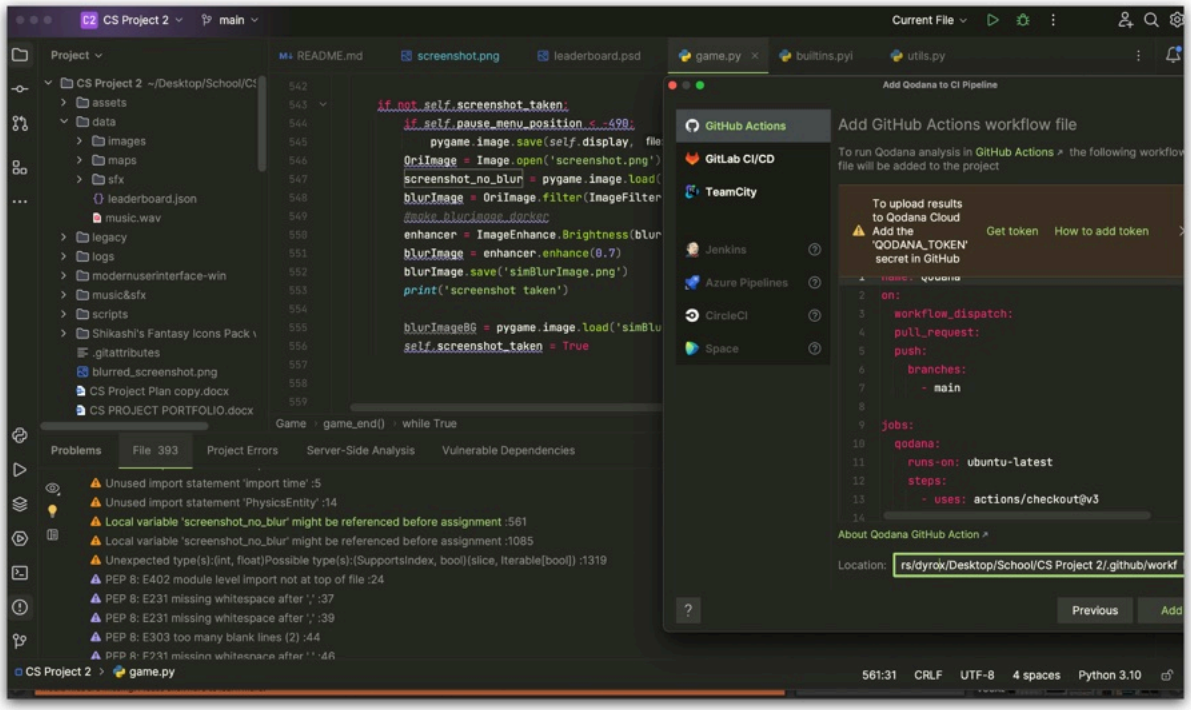
In the graph below, we can see a report is ordered by cumulative time, and the list has been reduced to show the top 20 function calls out of a total of 687 due to the restriction in place.

```
         4738487 function calls (4736430 primitive calls) in 48.592 seconds

   Ordered by: cumulative time
   List reduced from 687 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     21/1    0.000    0.000   48.592   48.592 {built-in method builtins.exec}
      2/1    0.424    0.212   46.879   46.879 /Users/dyrox/Desktop/School/CS Project 2/game.py:1165(main_menu)
      3/1    0.098    0.033   45.370   45.370 /Users/dyrox/Desktop/School/CS Project 2/game.py:751(main_game)
        1    0.008    0.008   43.430   43.430 /Users/dyrox/Desktop/School/CS Project 2/game.py:963(ingame_menu)
1746/1549    0.054    0.000   39.338    0.025 /Users/dyrox/Desktop/School/CS Project 2/game.py:658(EVERYTHING_render_update)
1746/1549    0.031    0.000   18.332    0.012 /Users/dyrox/Desktop/School/CS Project 2/game.py:306(check_level_loading)
        1    0.093    0.093   18.216   18.216 /Users/dyrox/Desktop/School/CS Project 2/game.py:473(game_end)
     1461    0.151    0.000   17.926    0.012 /Users/dyrox/Desktop/School/CS Project 2/game.py:582(displayUI_EVERYTHING)
    98602    0.054    0.000    9.432    0.000 /Users/dyrox/Desktop/School/CS Project 2/game.py:178(draw_text)
   593084    9.320    0.000    9.320    0.000 {method 'blit' of 'pygame.surface.Surface' objects}
   103732    9.111    0.000    9.111    0.000 {method 'render' of 'pygame.font.Font' objects}
      864    5.018    0.006    8.937    0.010 /Users/dyrox/Desktop/School/CS Project 2/game.py:349(render_end_game_panel)
     1510    4.270    0.003    7.226    0.005 /Users/dyrox/Desktop/School/CS Project 2/game.py:868(render_pause_menu)
     1342    3.604    0.003    4.597    0.003 /Users/dyrox/Desktop/School/CS Project 2/game.py:536(running_timer)
     1461    1.944    0.001    2.941    0.002 /Users/dyrox/Desktop/School/CS Project 2/game.py:567(show_level_number)
     1342    1.792    0.001    2.332    0.002 /Users/dyrox/Desktop/School/CS Project 2/game.py:559(show_enemy_count)
     2658    2.050    0.001    2.050    0.001 {method 'tick' of 'pygame.time.Clock' objects}
     2658    1.117    0.000    2.039    0.001 /Users/dyrox/Desktop/School/CS Project 2/game.py:595(update_debug_stuff)
        1    0.014    0.014    1.712    1.712 /Users/dyrox/Desktop/School/CS Project 2/game.py:27(__init__)
     2373    1.681    0.001    1.681    0.001 {built-in method pygame.display.update}
```

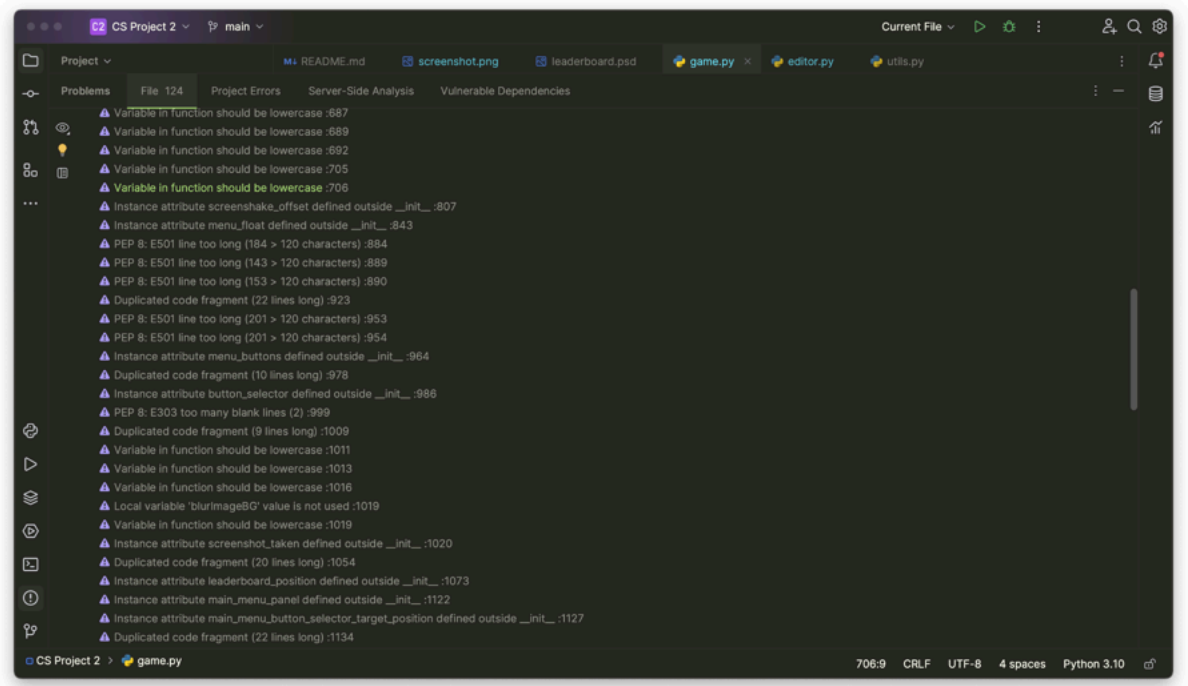
From the provided data, the builtins.exec() function is where the most cumulative time is spent, which is expected as it represents the entry point of the script execution. The main_menu and main_game functions also show significant cumulative times, indicating they are central to the game's execution and may be good targets for optimization. The function EVERYTHING_render_update() on line 658 has a considerable cumulative time, suggesting that the rendering updates are particularly resource-intensive.

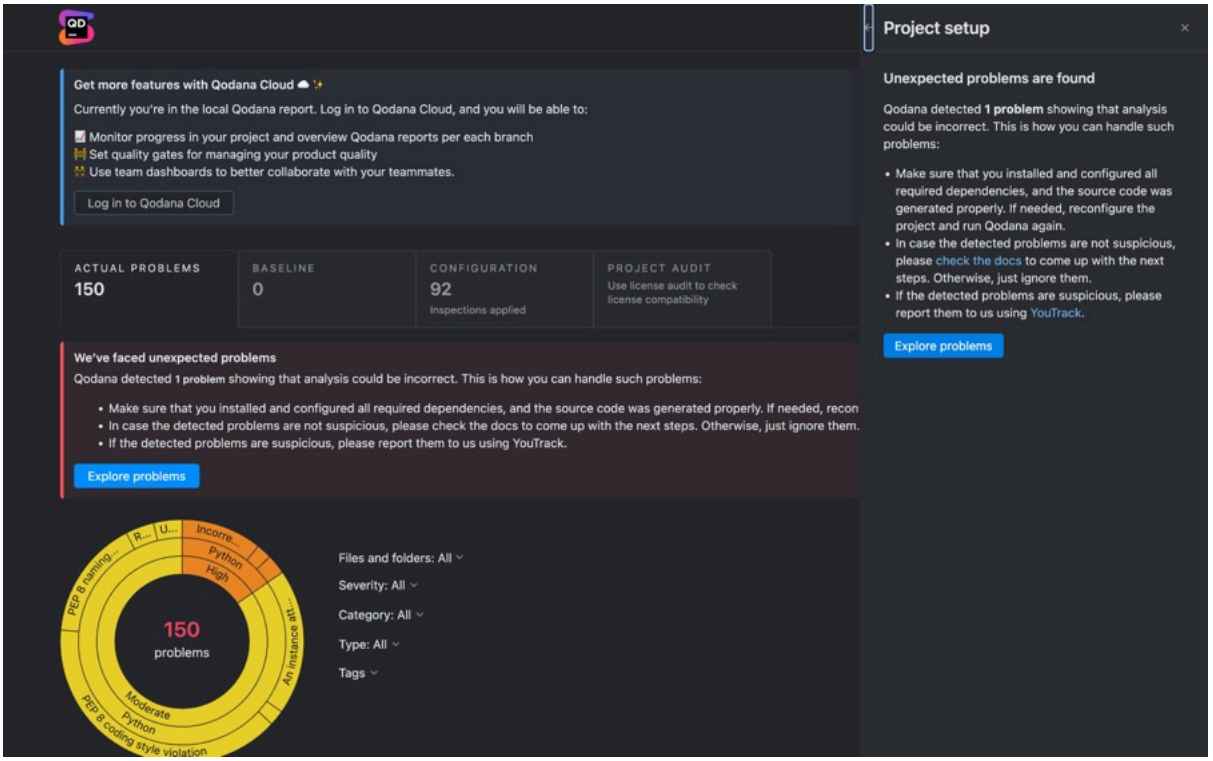
The check_level_Loading and game_end functions also stand out, indicating that level loading and the game-ending sequences are areas where performance could potentially be improved.

While working on the project, I discovered another IDE named PyCharm, which functions similarly to VSCode but with some more advanced features. It includes a built-in tool that assesses the quality of my code; it examines not just the structure but also analyses all variables, determining their efficiency and identifying potential logical errors. Since Python does not reveal errors until the code is executed, this tool is great as it can detect errors before running the code.



These are all some of the suggestions that it provided, it has identified duplicated code blocks and unused variables. Which then I can remove to make the code more elegant. For example, there is an unused variable blurImage, this exists in the previous implementation of the menu pause, but it has been changed and not removed.





This IDE has a page that lists out all the errors so we can debug them quite efficiently.

Optimized tiles rendering

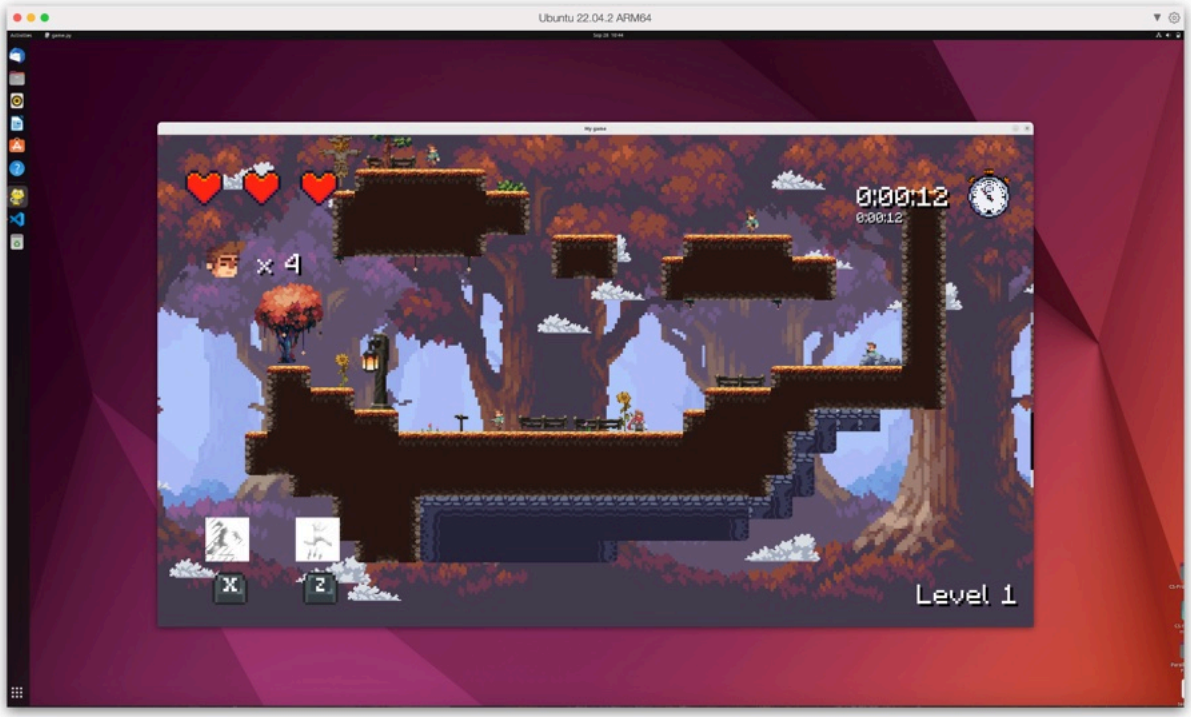
```python
def render(self, surf, offset=[0,0]):

    for x in range(int(offset[0]//self.tile_size),int((offset[0]+surf.get_width())//self.tile_size+1)):
        for y in range(int(offset[1]//self.tile_size),int((offset[1]+surf.get_height())//self.tile_size+1)):
            loc = f'{x};{y}'
            if loc in self.tilemap:
                tile = self.tilemap[loc]
                tile_image = self.game.assets[tile['type']][tile['variant']]
                tile_position = (tile['pos'][0] * self.tile_size - offset[0], tile['pos'][1] * self.tile_size - offset[1])
                surf.blit(tile_image, tile_position)
            You, 18 seconds ago • Uncommitted changes
```

There are some improvements that can be made to the tile rendering system, we can notice that not all tiles need to be rendered because we simply cannot see those outside of the screen, so we only render the tiles which are in our view.

We center the player's coordinates and put boundaries around the hitbox, which is the screenwidth and screenheight, we divide the actual coordinates with a factor of tile size to get the pixel coordinates of the tile and we will render them only.

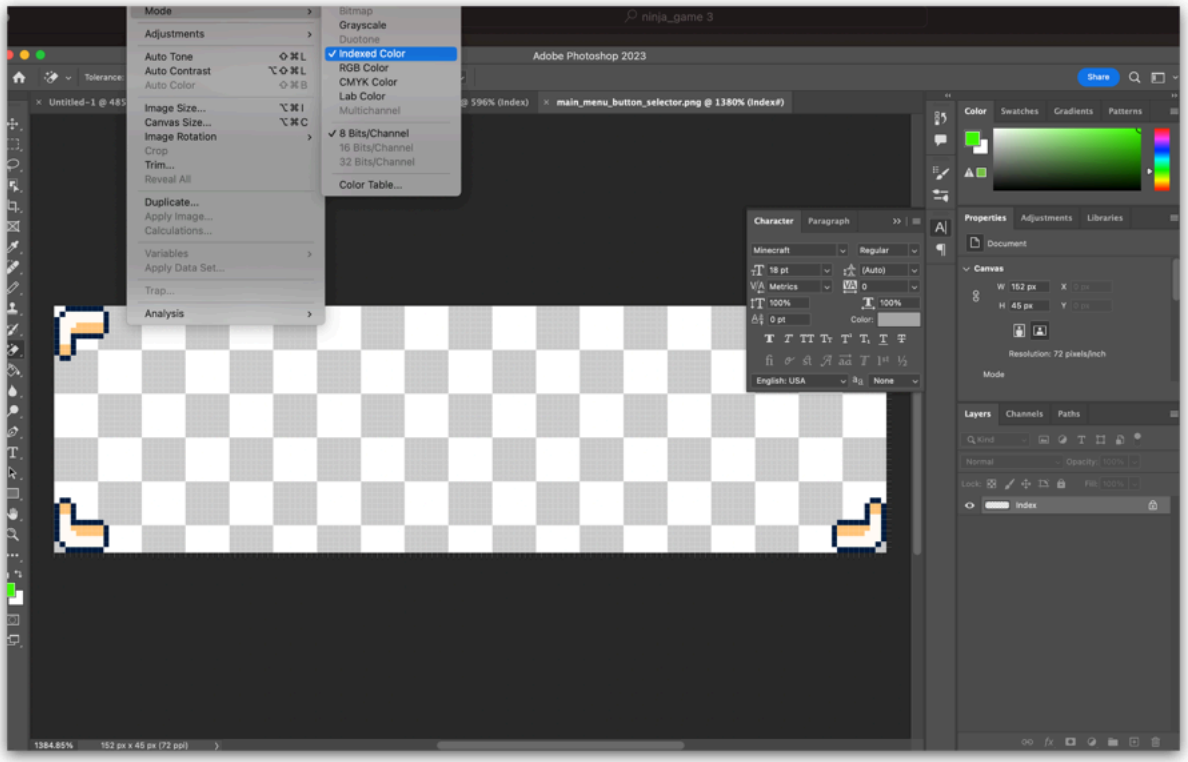
RGB vs Index Colour format

When I tested my game on macOS, the images displayed correctly; however, on other operating systems like Linux or Windows, there were issues. There are quite a few differences, such as path names, character encoding, and image formats, between these platforms. For instance, running my game on my Linux virtual machine results in texture issues with the CD blocks in the left corner, where the background is white instead of transparent

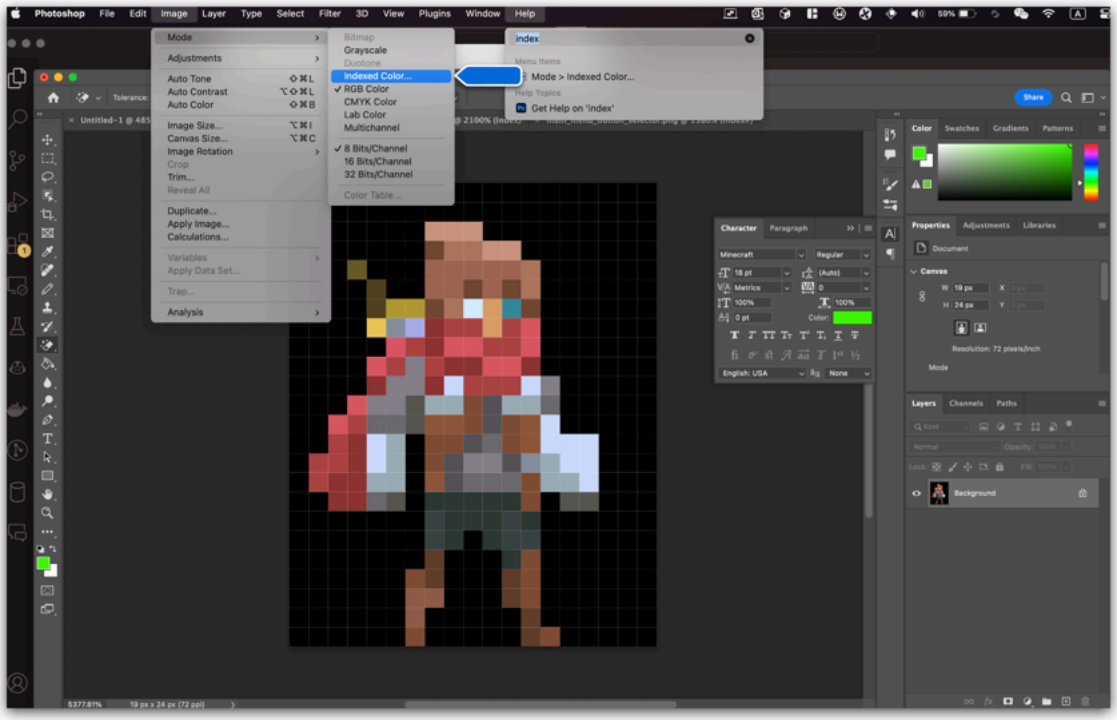


A similar error also happens on the main screen, where the button selector just becomes a white block.

However, most of the other images don't seem to have this problem - upon closer inspection, I found the images that are working correctly have the index colour format, instead of RGB. Index colours only has 256 fixed colours to choose from, but it's fine for 8bit styled games like this.



The format conversion probably happened when I edited the image with photoshop, photoshop automatically converts all images to rgb.

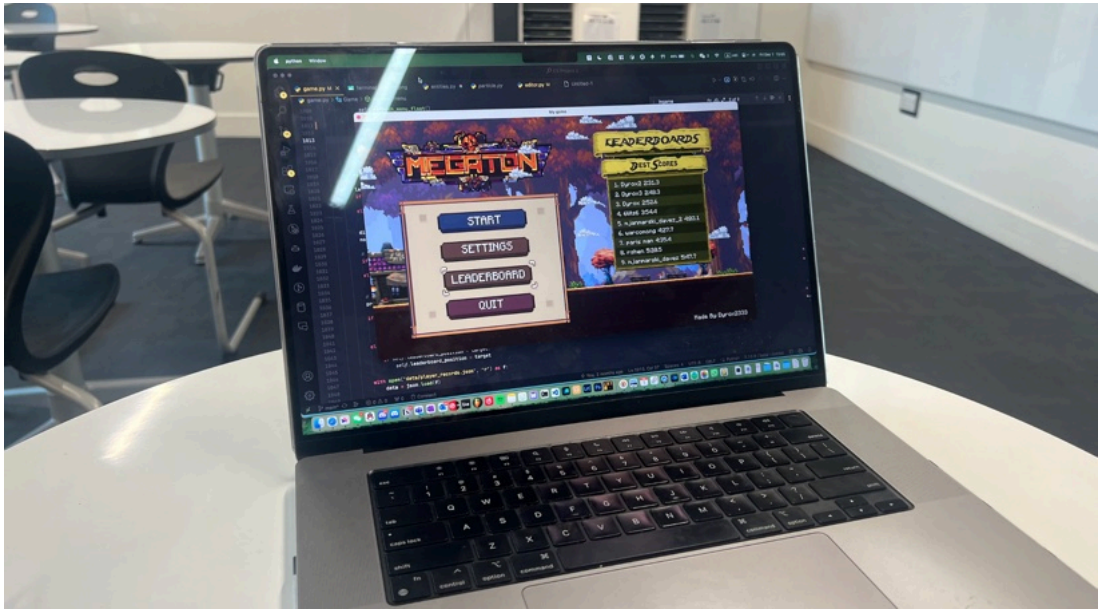

Photoshop File Edit Image Layer Type Select Filter 3D View Plugins Window Help
Mode
Adjustments
Auto Tone
Auto Contrast
Auto Color
Image Size...
Canvas Size...
Image Rotation
Crop
Trim...
Reveal All
Duplicate...
Apply Image...
Calculations...
Variables
Apply Data Set...
Trap...
Analysis
Bitmap
Grayscale
Duotone
Indexed Color...
RGB Color
CMYK Color
Lab Color
Multichannel
8 Bits/Channel
16 Bits/Channel
32 Bits/Channel
Color Table...
index
Mode > Indexed Color...
Get Help on 'index'
Color
Swatches
Gradients
Patterns
Properties
Adjustments
Libraries
Document
Canvas
Resolution: 72 pixels/inch
Mode
Character
Paragraph
Layers
Channels
Paths
Background

Evaluation

Testing

At this stage, we are finished with the implementation, which means the codebase will remain as it is from now on. The finished product will be tested against the criteria we have listed out in the requirements section, to see if it has met the goal.
Generally, the solution meets the specified requirements, successfully addressing each aspect of the success criteria. Furthermore, I've extensively played the game to identify potential errors during gameplay.

All the testing will be performed on the same machine of which the game is developed. Which is my Mac laptop computer, which is running an Apple M1 Max CPU with 64GB of ram, on the arm64 architecture, notte that this computer has a different architecture than most x86 based Windows PC out there, so eventually I would have to test my solution on different platforms too.



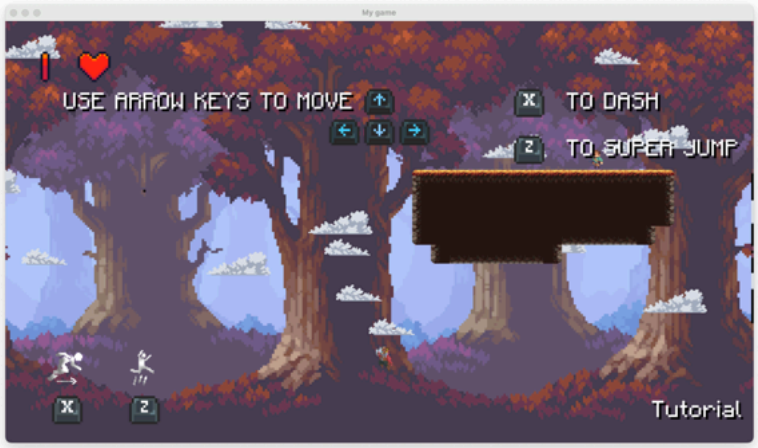
Testing against requirements and success criteria

Table 1: Game Mechanics

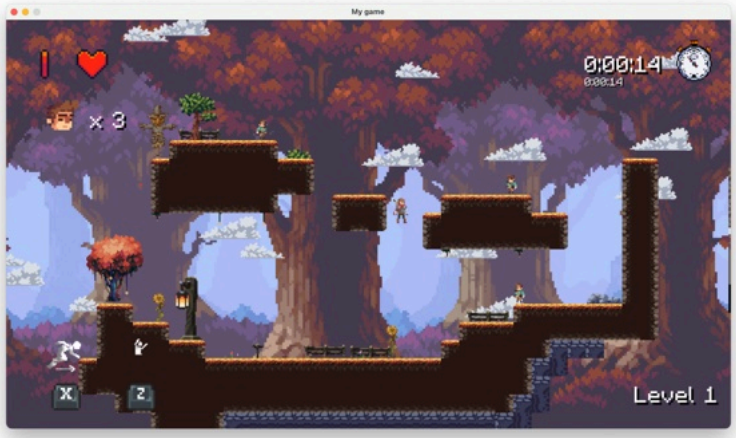
Spec points	Works?	Item Tested	Test	Expected Result	Actual Result
1.1.1	✅	Player stands on the ground	Leaving the player character on the ground	Player stands on the ground and does not fall through.	

1.1.2	✅	Player moves left	Pressing Left Arrow	Player moves left	

1.1.3	✅	Player moves right	Pressing Right Arrow	Player moves right	

1.1.4	✅	Player moves down	Walk off an edge	Player moves down	

1.1.5	✅	Player jump	Pressing Up arrow	Player jumps up	

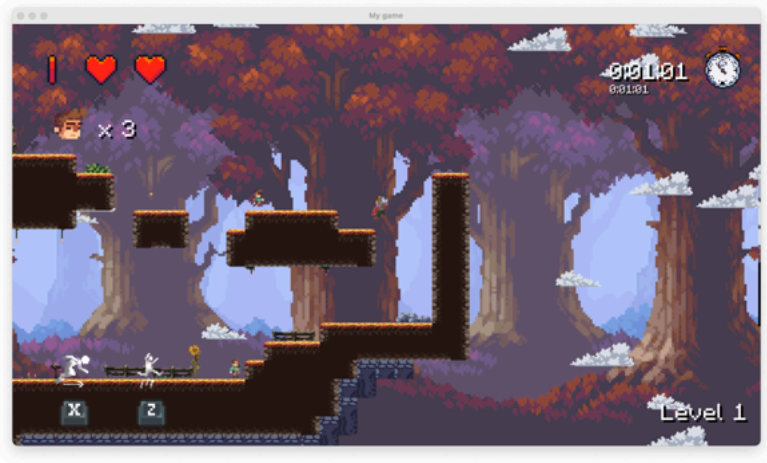
1.1.6	✅	Big jump	Pressing Z	Player performs big jump	

1.1.7	✅	Fall speed	Player falls after jumping	Player accelerates downwards until landing	

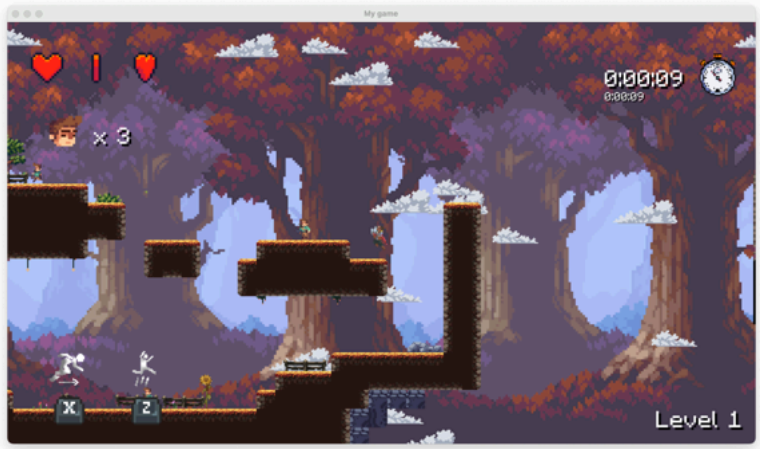
1.1.8	✅	Dash cooldown	Dashing and then immediately trying to dash again	Player can't dash until cooldown completes	

1.1.9	✅	Following Camera	The Camera follows the player so that the character is always in frame	The player moves around the map and see if the camera follows it.	

1.1.10	✅	Screenshake	The screen ‘shakes’ when the player is hit by enemy or we dash into enemies	We press X near an enemy to dash into it or get hit.	

1.2.1	✅	Wall jump	While the player hangs onto a wall, the player can jump from the wall to create a horizontal dash	The player hits the up arrow key while on a wall	 (a bit hard to see but it does work)

1.2.2	✅	Input Requirements	Character can dash through enemies to defeat them	Character defeats enemies when dashing through them	

1.2.3	✅	Input Requirements	Character can jump to avoid obstacles and reach platforms	Character avoids obstacles and reaches higher platforms	
1.2.4	❌	Input Requirements	Customizable Controls: Flexible control scheme	Players can adjust keybindings for comfort and responsiveness	

1.2.5	✅	Input Requirements	In-Game Menu Access (ESC key)	In-Game Menu is accessible by pressing the ESC key	

1.2.6	✅	Game Mechanics	Collision with walls and decoration items	Player cannot go through walls but can go through decoration items	

Table 2: Graphics Requirements

Test Number	Works?	Item Tested	Test	Expected Result	Actual Result
2.1	✅	Player sprite animation	Player sprite animates in the direction of movement	Player sprite animation matches player's movement	

2.2	✅	Responsive UI	UI is easy to navigate	UI is intuitive and easy for players to navigate	
2.3	✅	Consistent Art Style and Level Design	Game maintains consistent art style and level design	Game maintains a consistent visual style throughout and level design	

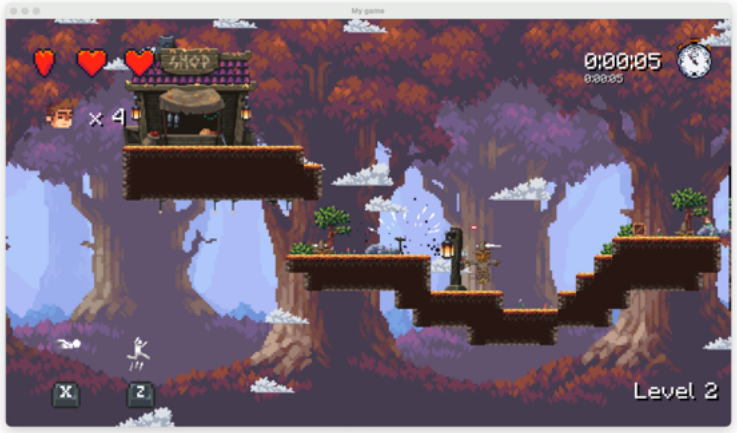
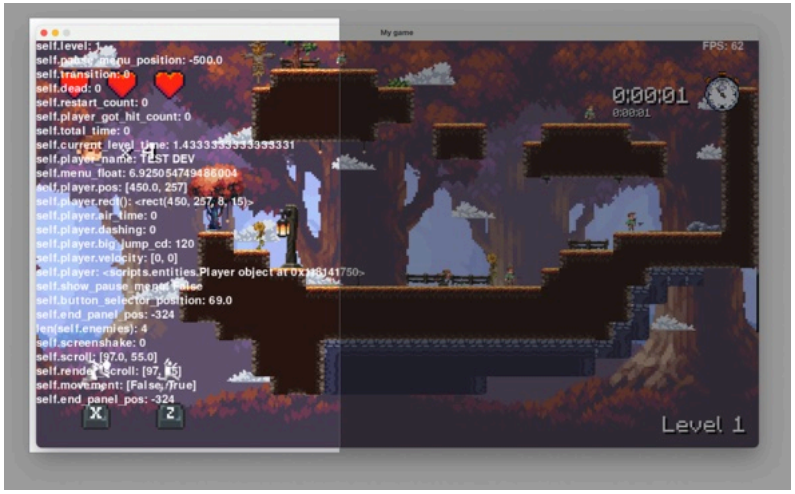
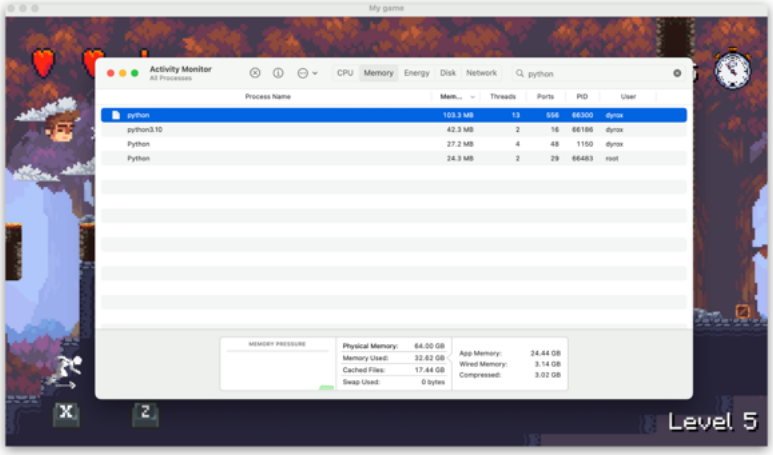
2.4	✅	Particle Animation	We dash into the enemy to observe if particles are produced.	A bunch of white particles are spawned around the enemy.	
2.5	✅	Non-linear animation	All the elements in my game has a non-linear animation	The animation are triggered as we move in between elements, such as scrolling and selecting.	

Table 3: Game's Technical Requirements

Test Number	Works ?	Item Tested	Test	Expected Result	Actual Result
3.1	✅	Debug Menu	Debug menu availability	Debug menu is available for troubleshooting and insights	

3.2	✅	Game Performance (FPS and Controls)	Game runs at 60FPS+ with responsive controls	Game runs smoothly at 60FPS or higher with responsive controls	
3.3	✅	Bug-Free	No game-breaking bugs or glitches	Game is free from game-breaking bugs and glitches and doesn't exit unexpectedly	

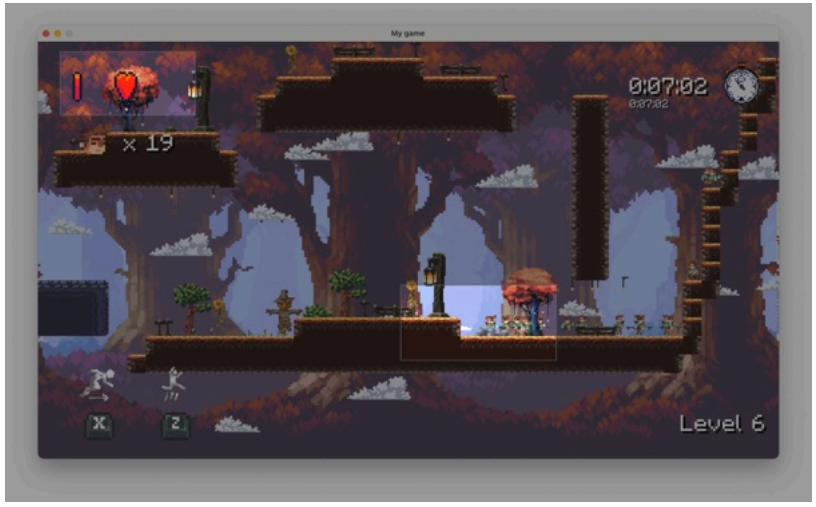
3.4	✅	Load Times	Swift loading times with minimal wait	Game levels and assets load swiftly with minimal wait times	
3.5	✅	Optimization	Efficient resource usage	Game efficiently uses resources, avoiding overheating or excessive resource consumption	

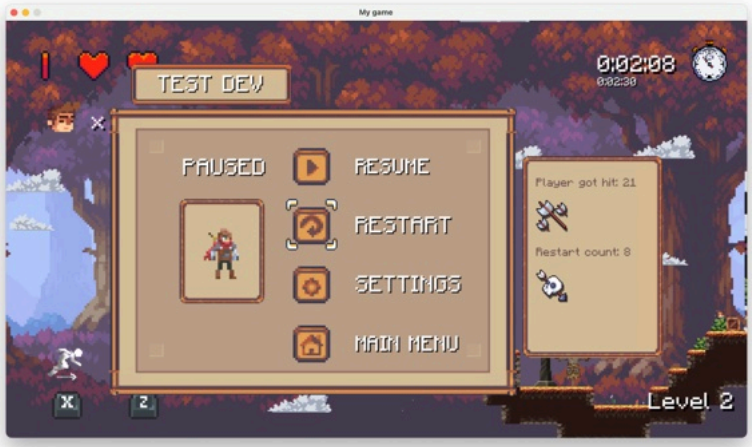
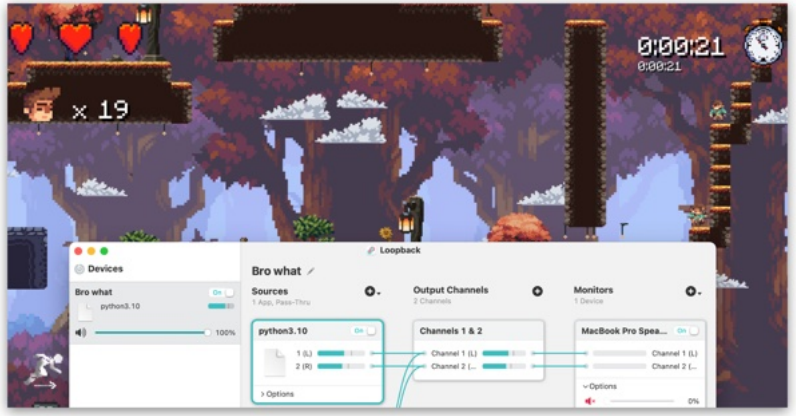
3.6	✅	Save Mechanism	Reliable save system with auto-save (if applicable)	Players' progress is reliably saved, with auto-save features if applicable	
3.7	❌	Controller Support (Xbox/Dualshock)	Support for Xbox and Dualshock controllers	Game supports Xbox and Dualshock controllers	

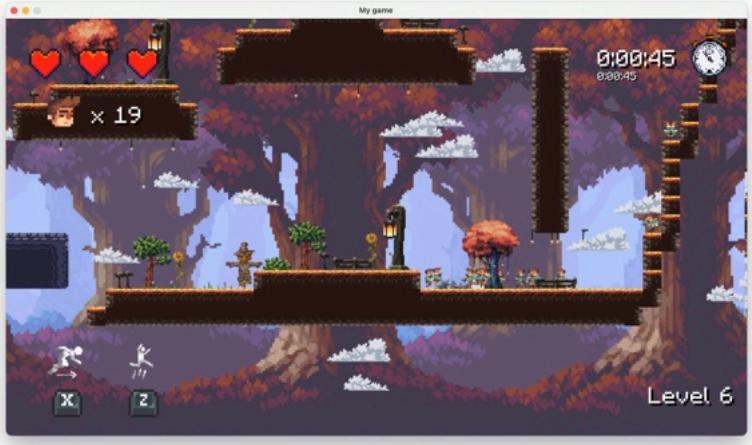
Table 4: Game Design

Test Number	Works?	Item Tested	Test	Expected Result	Actual Result
4.1	✅	Dash Cooldown	Dash has a cooldown duration	Player must wait for a certain time to use the dash again	
4.2	✅	Player Lives	Player starts with 3 lives, game ends on depletion	Player starts with 3 lives, and the game ends when lives are depleted	

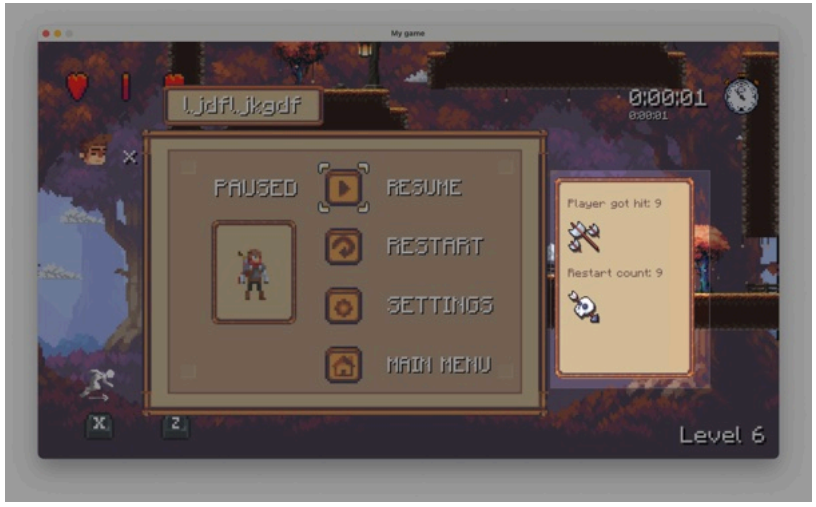
4.3	✅	Player Falling into Void	Player dies when falling into the void	Player dies when they fall into the void	
4.4	✅	Level Progression with Defeated Enemies (Dashing)	Progression when all enemies defeated by dashing	Progress to the next level when all enemies are defeated by dashing	

4.5	✅	Player Health	Player's health decreases when hit by enemies	Player's health decreases when hit by enemies	
4.6	✅	Personalized Identity	Players can input their names for leaderboards	Players can input their names for personalized leaderboards and/or multiplayer sessions	

4.7	✅	Pause, Resume, and Menu System	Pause, resume, and menu functionality	Players can pause, resume, and navigate through the menu	
4.8	✅	Sound Effects and Music	Matching sound and music	Sound effects and music match the game's theme and enhance the gameplay	

4.9	✅	Gradual Difficulty Increase	Difficulty increases as the player progresses	Game gradually becomes harder as player progresses (tougher enemies, intricate maps, traps)	
4.10	✅	Tutorial Level	Incorporate a tutorial level	Tutorial level is included to familiarize players with game mechanics and dashing techniques	

4.11	✅	High Score System	High score system for player competition	Game has a high score system for player competition	
4.12	✅	User-Created Levels	Built-in editor for designing levels	Players can design and share their own levels through a built-in editor	

4.13	✅	In-Game Player Statistics	Access performance metrics in-game menu	Players can access key performance metrics (hits taken, enemies defeated) through the in-game menu	
4.14	✅	End-of-Game Information Panel	Information panel after game completion	After beating the game, show an information panel with details (time taken, hits taken, restart count)	

While most of the required specifications are met, a small number of them are not, these could be due to technical difficulty, or lack of time for development. These are mainly optional specs which are not part of the main game so they can be emitted.

Analysing the Success Criteria

After the game has been tested against, we need to compare the success criteria against the performance.

SECTION 1 - Input Requirements:

1.1 The player can use arrow keys to move around the character.

The character movement is controlled by the arrow keys, it is able to do horizontal as well as vertical movements based on the user's action. The [x,y] coordinate pair of the player location is updated in real time and the image is rendered at the location every new frame, hence creating the effect that the player can be controlled with external keyboard control.

1.2 The character shall be able to dash through enemies to defeat them.

The dash function is implemented inside the player's sprite, and the collision check runs every frame to detect if the player has dashed into an enemy, if so, the game eliminates the enemy and play a series of animation (such as smoke and particle generation) to indicate the player has successfully defeated the enemy.

1.3 The character shall be able to jump to avoid obstacles and reach higher platforms.

The player has the ability to jump 2 tiles high and it is enough to avoid obstacles and reach higher platforms. Since each level has a different terrain, the player basically needs to do a bit of 'parkour' to get around places on the map. The enemy's bullets can also be dodged with jumping as they are sprites and has physical hitbox, so the jumping ability becomes very useful.

1.4 Customizable Controls: The game offers a flexible control scheme, allowing players to adjust keybindings for optimal comfort and responsiveness.

The keyboard binding can be change with minimal code adjustment, as we have a separate system of receiving keystrokes and binding these keystrokes to actual actions (such as jumping, moving etc.) Some people prefer different layouts, for example, some may prefer WASD over arrow keys and some prefer the other way round, so both of these group of players can be satisfied. Other than that, players may even have different hand sizes and they would prefer to put the skill buttons somewhere else, to avoid two hands bumping together.

1.5 The In-Game Menu is accessed by pressing ESC in the game.

The In-Game Menu can be called by pressing the ESC in the main game loop, and the menu loop takes over the screen. The game basically has many 'states' and each screen can run independently of each other. For example, the main game loop runs independently of the main screen loop and it ensures the no computing power is wasted in the background. The

in game menu has statistics that are updated in real time, such as enemy kills and restart counts, so the player can know there own progress. Also you can navigate to the main screen or restart the current level using the menu.

1.6 The Player should not go through walls, but through the decoration items.

There are two different types of tiles in the game, one is physical tiles and has hitboxes, and the other type of tile are purely for visual effect and do not have hitboxes. In the game engine development section, I have labelled the two different types of tiles using attributes, so the collision detection system ignores those without the hitbox tag.

SECTION 2 - Graphics Requirements:
2.1 Player sprite animates in the direction of the player movement.
This is achieved with a Boolean variable, as the original player sprite faces to the right, so when the velocity is negative (moving to the left), the image is flipped so it looks that the player is facing the correct direction.

2.2 Responsive and intuitive UI that is easy to navigate.
The UI is made minimal to ease usage and the UI animations are updated upon user input, which makes it responsive.

2.3 The game should maintain a consistent art style and level design.
The art uses a pixelated style, with bright colours and minimalist design. The level design is consistent with similar terrain, difficulty level as well as number of enemies etc.

SECTION 3 - Game's Technical Requirements:

3.1 Debug Menu: A handy debug menu is available for troubleshooting and game insights, aiding in smooth gameplay.
The debug menu can be accessed both through the main menu as well as the in-game pause menu, enabling developers to view important information while debugging or testing the game. The debug menu contains a variety of variables such as player coordinates, alive enemy count etc.

3.2 The game should be running at 60FPS+ and the controls need to be responsive.
The game uses minimal processing power as it renders in 2D, the map rendering system is also optimized so that the blocks outside of the player's camera are not rendered in order to save resources. Even though the Python interpreter only uses a single thread to process, the work load is small enough for one CPU core to render the graphics at 60FPS and responsive.

3.3 Bug-Free: Efforts should be made to ensure that there are no game-breaking bugs or glitches. The game should not exit unless the player wants to.
The only exit button is on the main menu and the native Python quit() function is called upon closing the game, the game has been thoroughly tested for an extended period of time and it has not yet shown signs that it would quit randomly.

3.4 Load Times: Game levels and assets should load swiftly, keeping wait times minimal to maintain player immersion.
The levels usually just load instantly as the map data are minimal

3.5 Optimization: Efficient use of resources to ensure that the game doesn't overheat devices or consume excessive battery (for mobile games) or system resources (for PC/console games).
As previously stated, it only uses a single thread on the CPU process so it does not overheat the device at all, the battery life is also great, using less than 5W on my Apple M1 processor.

3.6 Save Mechanism: A reliable save system that ensures players don't lose their progress. Auto-save features can also be beneficial.
Every time the player progresses onto a new level, a level variable is set so that when the player dies, it is still in the same level when it respawns.

3.7 Controller support (Xbox/Dualshock controller).
Not implemented.

SECTION 4 - Game Design:

4.1 The dash has a cool-down duration, which means the player has to wait for a certain time to use the dash again.
The dash cool-down is tracked with a CD variable, and the value is updated using a tick counter, so it refreshes every 1.5 second (or 90 TICKS) before the player can use the power-up again.

4.2 The player is given 3 lives at the start, and the game ends when these lives are depleted.
Tracked using a integer variable, decremented every time the player gets hit by the enemy's bullet and the restart_level() function is called when the player dies.

4.3 The player dies when they fall into the void.
This is achieved with the air_time variable, when the air_time variable exceeds a certain value, that means the player has been falling for too long, which indicates that it has fallen into the void and the restart_level() function is called.

4.4 Level progression when all enemies are defeated by dashing.
The enemies are all stored inside a list, so we use the len() function to get the length of the list, which is also the number of enemy still alive, when that number decreases to 0, all enemies are defeated and the level_counter is incremented by 1.

4.5 Player health taking damage from enemies when it has been hit.
The particles has a hitbox and collision detection system running every tick to make sure the HP variable is updated accordingly.

4.6 Personalized Identity: Players can input their name for a personalized touch in leaderboards and/or multiplayer sessions.

The player_name is entered upon entering the game and it is displayed both on the in-game menu and the leaderboard on the main screen. The leaderboard saves the name of the player as well as their record in the file system.

4.7 A pause, resume, and menu system so players can navigate and take breaks.
When the pause menu is called, the game is paused and the states are saved - enemy and player will stay where they are and all the variables are frozen. and this will be resumed when the player presses the continue button.

4.8 Sound effects and music that match the game's theme and enhance the gameplay experience.
I have composed the background music as well as the sound effects to enhance the gameplay experience. There mainly 4 different types of SFX, the sound plays when the player scrolls through the menu. the bullet hit sound, jumping sound and dash sound. They suit the

4.9 The difficulty of levels gradually gets harder as the player progresses (e.g., tougher enemies, intricate map designs, or traps that require special skills to bypass).
I have designed the levels in a way that it increases difficulty as the player progresses, the first ones are very easy with minimal enemies, (2-3), and in later levels, both the terrain and the enemy count goes up, making the level challenging and engaging to the player.

4.10 Incorporate a tutorial level to familiarize players with the game mechanics and dashing techniques.
In the tutorial level, I've presented players with an image of keyboard controls alongside explanatory text, clarifying their available actions. Additionally, a test enemy has been strategically positioned to allow players to apply and refine their newly acquired skills.

4.11 The game shall have a high score system allowing players to compete with each other.
The leaderboard records the length of a run and the highest are displayed on the leaderboard, the player are able to compare their run to exist wants to make this competitive.

4.12 User-Created Levels: Players can use a built-in editor to design and share their own levels, promoting creativity and community engagement.
The built-in editor is easy to use and maps can be exported as .json files, which can be read by my game, added as custom maps and be played.

4.13 In-Game Player Statistics: Access key performance metrics, such as hits taken or enemies defeated, through the in-game menu to track and improve gameplay.
The UI provides a comprehensive player stats, hits taken and enemy defeated can be viewed in the pause menu.

4.14 After the player beats the game, an information panel is shown at the end with all the information (time taken, hits taken, restart count).
The penalties (such as restarts and death counts) are added to the final time of the run and that is the ‘score’ the player gets, the player would see these penalties and the score is then saved into the leaderboard.

Changes made in the development process

Since my development of the implementation use an iterative approach, some of othe specs/design points are changed along the way, these changes are made to improve the quality of the game in general, such as a better layout not covered in the design section. Here are some of the things that has changed in the development stage:

- Introduced an interactive menu with animation and player stats
- Player, and tiles textures changed
- Introduced multiple other decoration items (lamps, fences, trees)
- Added three lives rather than just one HP
- Added a new system to calculate player score (using hit counts and restart counts.)
- Added spinning animation for the hearts in the UI
- A new animation object class rather than just objects to create standalone animations for display only, no physics.
- An enemy counter on the UI
- Changed size of enemy & player (to create a bigger hitbox)
- Changed the rendering mechanism of tiles (only renders those in frame).
- Added a super jump to cope with the height difference in platforms.
- Shortened CD of super jump.

Usability Features

The game has been optimised for usability, it includes the following features such as:

- On screen control with keys displayed (such as the [X] and [Z] keys written under the ability CD block)
- A tutorial level where the game introduces the mechanics as well as basic controls to the user.
- The texts are big enough to be seen from a far distance
- The levels are saved so when the player dies they respawn from the current level tather from the very start.
- The in-game UI lists out everything the player needs to know (which level is he on, how many enemies left, how any HP left).
- The in-game timer is displayed real-time in the top right corner, so it could be useful when speedrunning.

Possible Future Improvements

As previously outlined, while the current iteration of the game successfully meets a majority of the predefined specifications, there remains room for enhancement in future versions of the game, I just need time to make all the changes. This could be creating more features to the game, such as adding plots and more types of creatures, as well as porting the game to other platforms like iOS and Android. This would be quite difficult as the game is running on Python with Pygame and Pygame uses some really low-level system subroutines so it might not be compatible with mobile systems. The following are some possible improvements that can be made to the game in the future.

- Minimap in game to show the map layout as well as the enemy locations.
- Ability to choose any level from the menu.
- Selection of Characters - Players can select characters with unique dash abilities and traits
- Overarching Storyline, Game includes an overarching storyline with different terrains and NPCs
- Boss Fights, Include boss fights at the end of each level.
- Varied Attack Mechanics, Players can unlock and use various attack methods, adding depth to combat strategies.
- Real-Time Multiplayer, Players can engage in real-time battles or co-op challenges with friends or matched opponents
- Diverse Enemies and Obstacles, Game includes a diverse range of enemies and obstacles for the player to overcome
- Game Shop, Players can spend in-game currency (obtained in the game) to buy upgrades

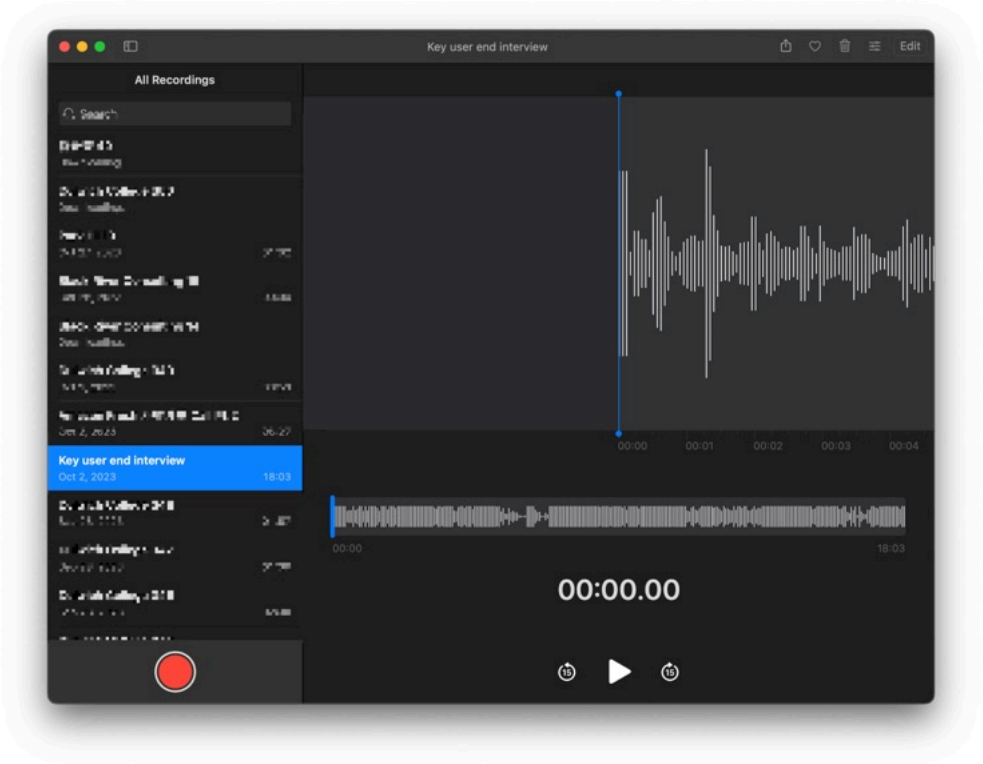
Maintenance

While the development phase of my game has concluded, the game maintenance should still be active. To ensure its continued evolution, I have made the code repository public on Github, so anyone is welcome to submit issues, suggestions, or even create fork from my game. Users are welcomed to leave comments as well as feedback to my game, and to my code to improve efficieny and speed. Since the only active developer is just myself, I need a lot of beta testers to test my game. For example, when I tested the game on my Macbook, everything went fine, but when I tested it on my Linux virtual machine, I encountered several texture issues regarding colour codecs, so this highlights the importance to test my program on different platforms.

On top of the usual maintenance aspects, I really encourage players getting creative and sharing their own levels. if we have a playerbase that’s large enough, we can transform this into a community based game where people can upload their levels and configuration to a server and everyone can access them. They can be new levels, new enemy types and new character abilities.

Git has made the maintenance process a lot easier as it includes time tracking with different iterations.

Key user end interview



The subjective assessments of the spec points are finished, and it's time to interview my key user, to hear what he thinks about my game.

What do you think about the game overall? What's your favourite thing from the game?
Daniel: It's really good actually, the game control is really smooth and as a big fan of Roguelike games this is a huge plus. By the way I really liked the little easter egg you put in the final level of the game.

What improvements do you think can be made to the game?
Daniel: Hmm… Let me think about it, everything is pretty good except for the wall jump, the control is so weird, why does the character jump to the right when you press the left arrow button? Makes no sense to me
(Sam: I mean this is how it is supposed to work? I don't see a problem.)
Daniel: This is terrible design, you might need to reconsider your life choices.

Do you find the levels challenging enough?
Daniel: I'm not sure if I can even beat the game… they seem too hard, maybe you should make them easier instead.
(Sam: No it's just you're bad at the game, get good, easy.)
Daniel: Ok sure.

Are the game's usability features good? If it is easy to use, the onscreen help gives enough information to get to understand how to play it etc.
Daniel: I mean I'd give it a solid 8/10, but why can't you use WASD for movement like a normal human? The arrow keys are literally so hard to use.

(Sam: Hey look, I used the arrow keys in the beginning of the development and all the tutorial animations are based on the arrow keys, it'd be a huge hassle if I had to make the animation from scratch again).
Daniel: So you're saying that you prefer WASD too?
(Sam: I mean yeah, any sensible gamer would have the same opinion.).

Bibliography

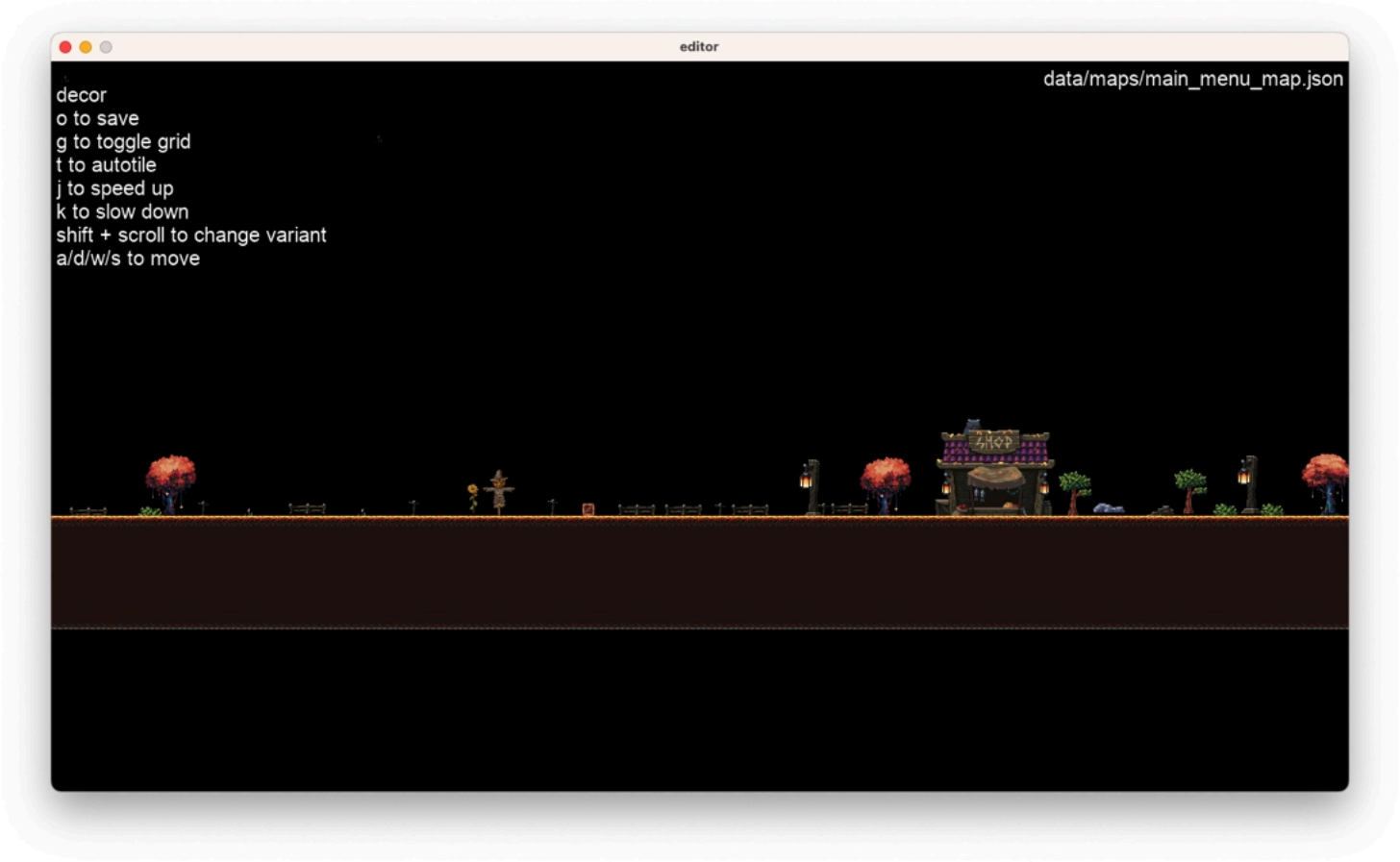
Tools & Software:
https://www.python.org/
https://pyga.me/
https://github.com/bbor/stingray-easing-functions
https://media.istockphoto.com/id/626205158/photo/portrait-of-young-man-with-shocked-facial-expression.jpg?s=612x612&w=0&k=20&c=0SDJDt1crElYppk8F8Qw-MNeBp3Sr8G7cCs4PAHfEH0=
https://www.aseprite.org/
https://dafluffypotato.com/assets/pg_tutorial

Game Assets:
https://dreammix.itch.io/keyboard-keys-for-ui
https://cainos.itch.io/pixel-art-platformer-village-props
https://bdragon1727.itch.io/basic-pixel-health-bar-and-scroll-bar
https://brullov.itch.io/oak-woods
https://rvros.itch.io/animated-pixel-hero
https://paperhatlizard.itch.io/cryos-mini-gui
https://greatdocbrown.itch.io/coins-gems-etc?download
https://trixelized.itch.io/starstring-fields?download
https://theflavare.itch.io/mondstadt-theme-background-pixel-art

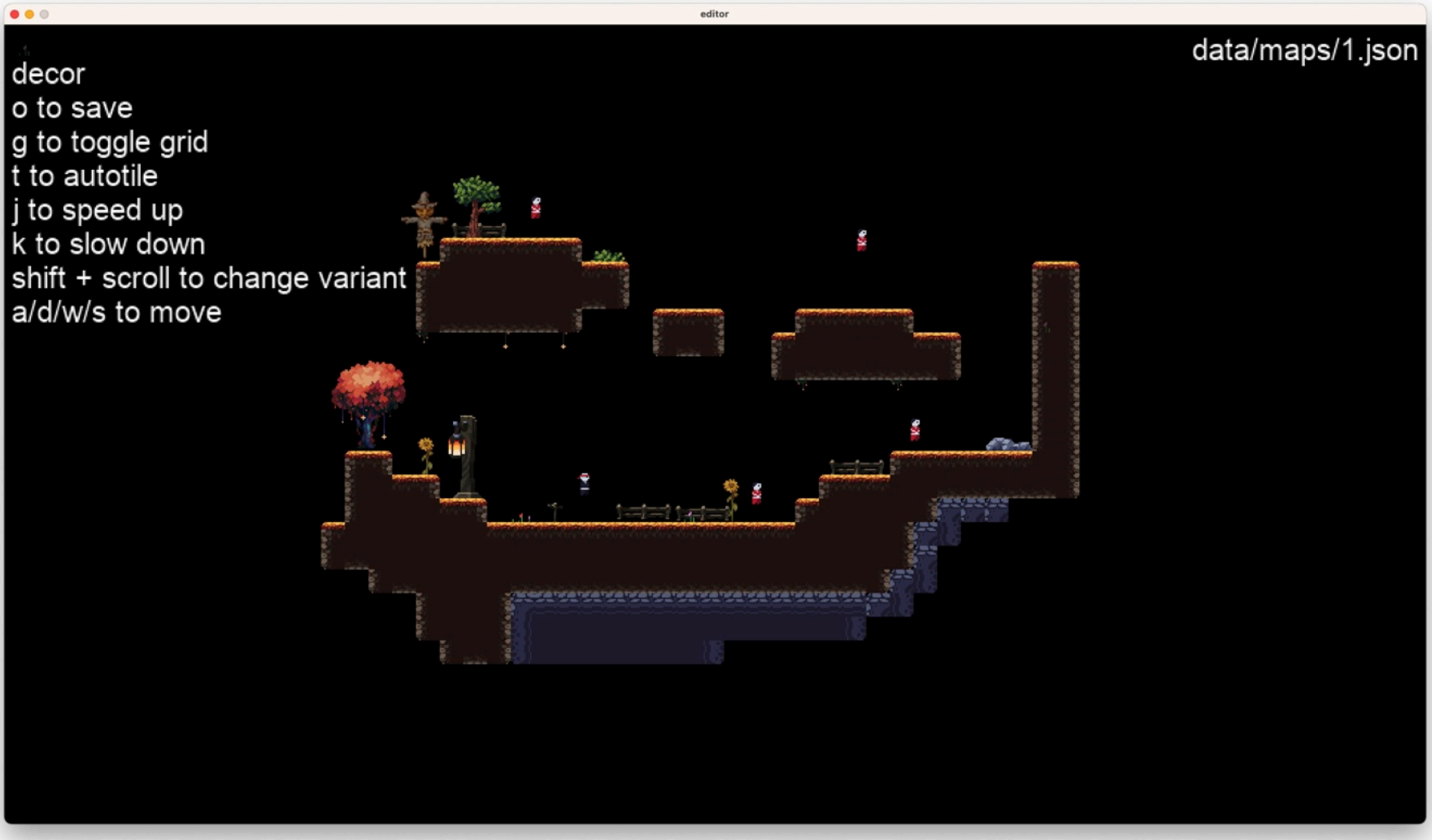
APPENDIX

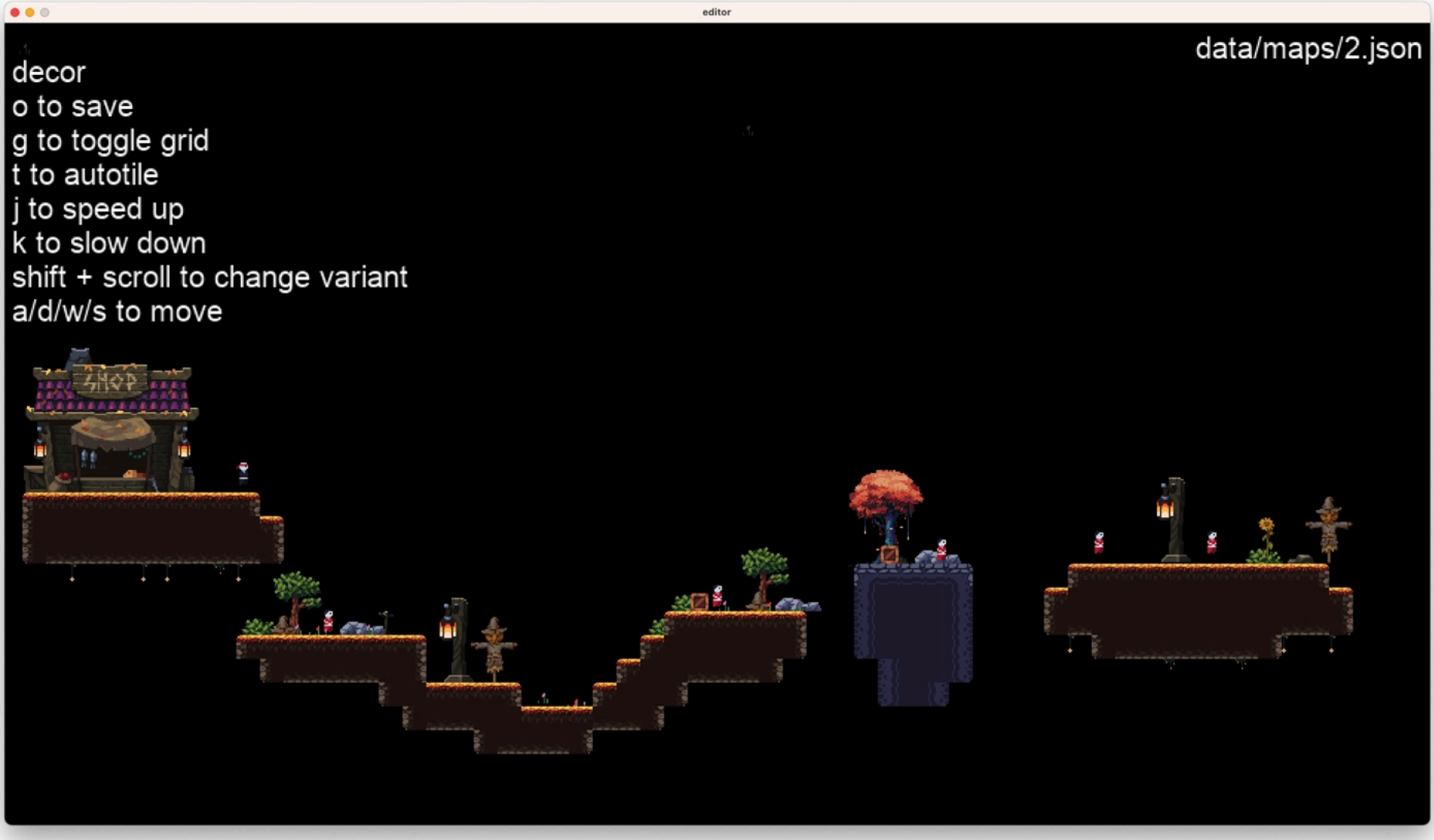
Level Maps

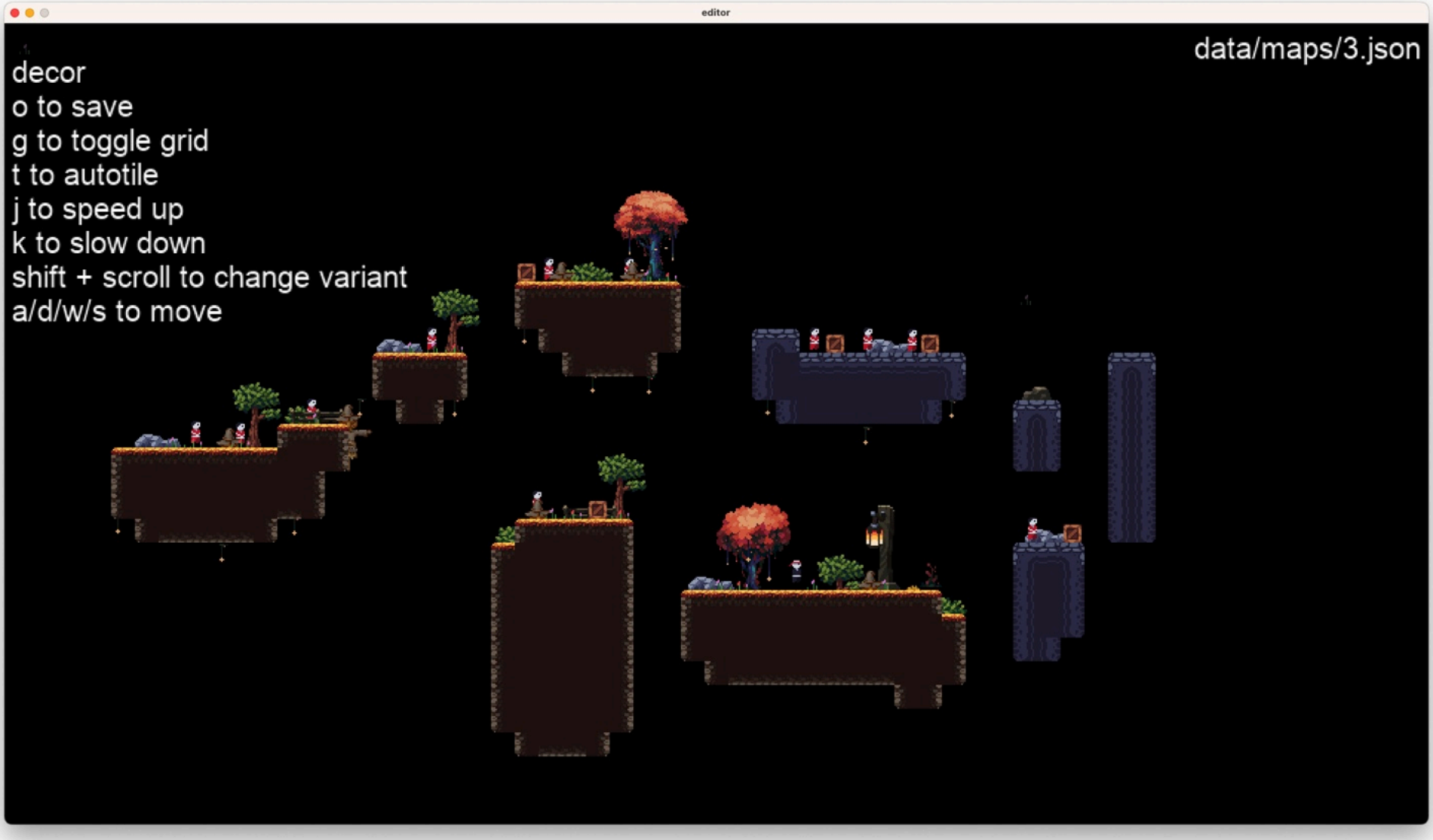
These are all self-made maps stored in .json files, below are screenshots of them rendered in my level editor.

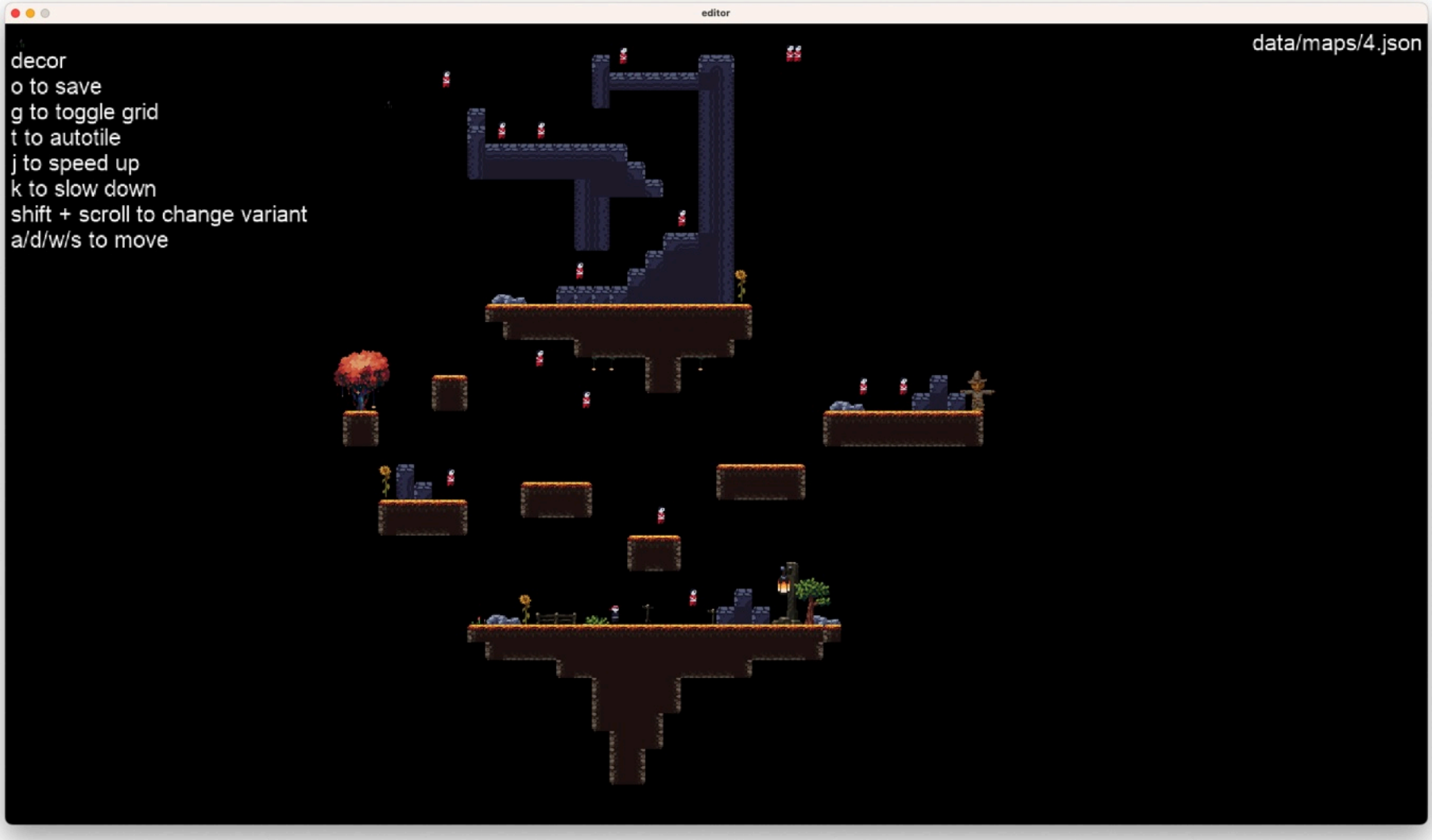



editor
data/maps/0.json
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move

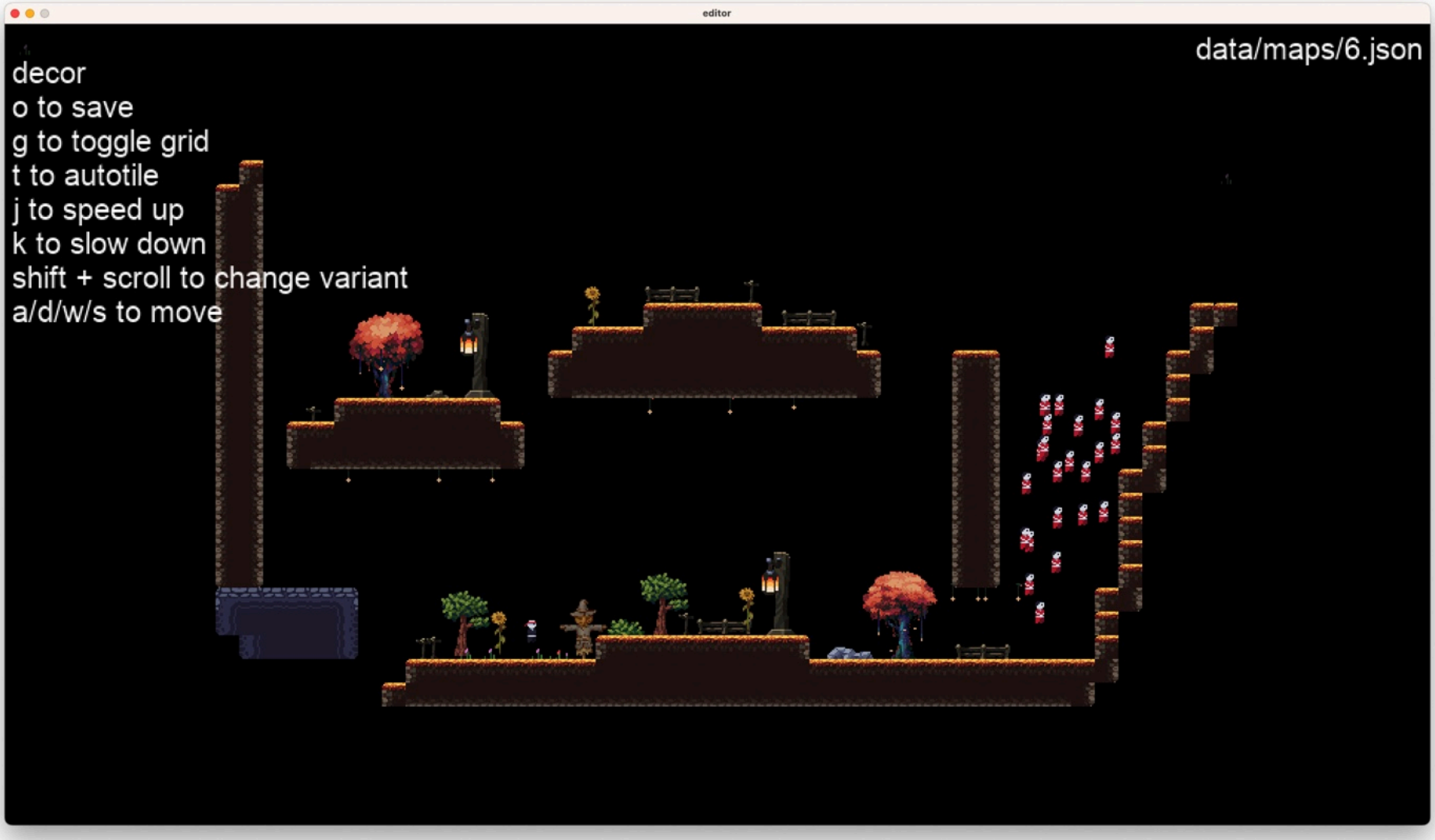

editor
data/maps/1.json
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move


editor
data/maps/2.json
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move


editor
data/maps/3.json
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move


editor
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move
data/maps/4.json


editor
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move
data/maps/5.json


editor
data/maps/6.json
decor
o to save
g to toggle grid
t to autotile
j to speed up
k to slow down
shift + scroll to change variant
a/d/w/s to move

Level Editor Code

```
import sys
import pygame

from scripts.utils import load_images
from scripts.tilemap import Tilemap

RENDER_SCALE = 2.0
RUNNING_FPS = 60
DISPLAYING_FPS = 120


class Editor:
    def __init__(self):
        pygame.init()
        pygame.display.set_caption('editor')

        window_size = (1280,720)
        self.screen = pygame.display.set_mode(window_size)
        self.display = pygame.Surface((640,360))
        self.clock = pygame.time.Clock()
        self.speed = 1
        self.assets = {
            'decor': load_images('tiles/decor'),
            'grass': load_images('tiles/grass'),
            'large_decor': load_images('tiles/large_decor'),
            'stone': load_images('tiles/stone'),
            'spawners': load_images('tiles/spawners'),
        }

        self.movement = [False, False, False, False]

        self.tilemap = Tilemap(self, tile_size=16)

        map_name = input("Map name: ")
        self.file_path = f'data/maps/{map_name}.json'
        if self.file_path:
            try:
                self.tilemap.load(self.file_path)
            except FileNotFoundError:
                pass
        else:
            print("No file selected!")
            sys.exit()

        self.scroll = [0, 0]

        self.tile_list = list(self.assets)
        self.tile_group = 0
        self.tile_variant = 0

        self.clicking = False
        self.right_clicking = False
        self.shift = False
```

```
        self.ongrid = True

    def run(self):

        while True:
            self.display.fill((0, 0, 0))

            self.scroll[0] += (self.movement[1] - self.movement[0]) * 2
            self.scroll[1] += (self.movement[3] - self.movement[2]) * 2
            render_scroll = (int(self.scroll[0]), int(self.scroll[1]))

            self.tilemap.render(self.display, offset=render_scroll)

            current_tile_img =
self.assets[self.tile_list[self.tile_group]][self.tile_variant].copy()
            current_tile_img.set_alpha(100)

            mpos = pygame.mouse.get_pos()
            mpos = (mpos[0] / RENDER_SCALE, mpos[1] / RENDER_SCALE)
            tile_pos = (int((mpos[0] + self.scroll[0]) // self.tilemap.tile_size),
                        int((mpos[1] + self.scroll[1]) // self.tilemap.tile_size))

            if self.ongrid:
                self.display.blit(current_tile_img, (tile_pos[0] *
self.tilemap.tile_size - self.scroll[0],
                                                     tile_pos[1] *
self.tilemap.tile_size - self.scroll[1]))
            else:
                self.display.blit(current_tile_img, mpos)

            if self.clicking and self.ongrid:
                self.tilemap.tilemap[str(tile_pos[0]) + ';' + str(tile_pos[1])] = {
                    'type': self.tile_list[self.tile_group], 'variant':
self.tile_variant, 'pos': tile_pos}
            if self.right_clicking:
                tile_loc = str(tile_pos[0]) + ';' + str(tile_pos[1])
                if tile_loc in self.tilemap.tilemap:
                    del self.tilemap.tilemap[tile_loc]
                for tile in self.tilemap.offgrid_tiles.copy():
                    tile_img = self.assets[tile['type']][tile['variant']]
                    tile_r = pygame.Rect(tile['pos'][0] - self.scroll[0], tile['pos'][1]
- self.scroll[1],
                                         tile_img.get_width(), tile_img.get_height())
                    if tile_r.collidepoint(mpos):
                        self.tilemap.offgrid_tiles.remove(tile)

            self.display.blit(current_tile_img, (5, 5))

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

                if event.type == pygame.MOUSEBUTTONDOWN:
                    if event.button == 1:
                        self.clicking = True
```

```
                        if not self.ongrid:
                            self.tilemap.offgrid_tiles.append(
                                {'type': self.tile_list[self.tile_group], 'variant':
self.tile_variant,
                                 'pos': (mpos[0] + self.scroll[0], mpos[1] +
self.scroll[1])})
                    if event.button == 3:
                        self.right_clicking = True
                    if self.shift:
                        if event.button == 4:
                            self.tile_variant = (self.tile_variant - 1) % len(
                                self.assets[self.tile_list[self.tile_group]])
                        if event.button == 5:
                            self.tile_variant = (self.tile_variant + 1) % len(
                                self.assets[self.tile_list[self.tile_group]])
                    else:
                        if event.button == 4:
                            self.tile_group = (self.tile_group - 1) %
len(self.tile_list)
                            self.tile_variant = 0
                        if event.button == 5:
                            self.tile_group = (self.tile_group + 1) %
len(self.tile_list)
                            self.tile_variant = 0
                if event.type == pygame.MOUSEBUTTONUP:
                    if event.button == 1:
                        self.clicking = False
                    if event.button == 3:
                        self.right_clicking = False

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_a:
                        self.movement[0] = self.speed
                    if event.key == pygame.K_d:
                        self.movement[1] = self.speed
                    if event.key == pygame.K_w:
                        self.movement[2] = self.speed
                    if event.key == pygame.K_s:
                        self.movement[3] = self.speed
                    if event.key == pygame.K_g:
                        self.ongrid = not self.ongrid
                    if event.key == pygame.K_t:
                        self.tilemap.autotile()
                    if event.key == pygame.K_o:
                        self.tilemap.save(self.file_path)
                    if event.key == pygame.K_LSHIFT:
                        self.shift = True
                    if event.key == pygame.K_j:
                        self.speed = 5
                    if event.key == pygame.K_k:
                        self.speed = 1
                if event.type == pygame.KEYUP:
                    if event.key == pygame.K_a:
                        self.movement[0] = False
                    if event.key == pygame.K_d:
                        self.movement[1] = False
```

```
                    if event.key == pygame.K_w:
                        self.movement[2] = False
                    if event.key == pygame.K_s:
                        self.movement[3] = False
                    if event.key == pygame.K_LSHIFT:
                        self.shift = False

            # write text: tile type
            font = pygame.font.SysFont('Arial', 20)
            text = font.render(self.tile_list[self.tile_group], True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height()))
            # write text: o to save
            text = font.render('o to save', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height()))
            # write text: g to toggle grid
            text = font.render('g to toggle grid', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 2))
            # write text: t to autotile
            text = font.render('t to autotile', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 3))
            # write text: j to speed up
            text = font.render('j to speed up', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 4))
            # write text: k to slow down
            text = font.render('k to slow down', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 5))
            # write text: shift + scroll to change variant
            text = font.render('shift + scroll to change variant', True, (255, 255,
255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 6))
            # write text: a/d/w/s to move
            text = font.render('a/d/w/s to move', True, (255, 255, 255))
            self.display.blit(text, (5, 5 + current_tile_img.get_height() +
text.get_height() * 7))

            #show level name on the top right
            text = font.render(self.file_path, True, (255, 255, 255))
            self.display.blit(text, (self.display.get_width() - text.get_width() - 5,
5))



            self.screen.blit(pygame.transform.scale(self.display,
self.screen.get_size()), (0, 0))
            pygame.display.update()
            self.clock.tick(60)


Editor().run()
```

Scripts Codes

clouds.py

```
import random

class Cloud:
    def __init__(self, pos, img, speed, depth):
        self.pos = list(pos)
        self.img = img
        self.speed = speed
        self.depth = depth

    def update(self):
        self.pos[0] += self.speed

    def render(self, surf, offset=(0, 0)):
        render_pos = (self.pos[0] - offset[0] * self.depth, self.pos[1] - offset[1] *
self.depth)
        surf.blit(self.img, (render_pos[0] % (surf.get_width() + self.img.get_width()) -
self.img.get_width(), render_pos[1] % (surf.get_height() + self.img.get_height()) -
self.img.get_height()))

class Clouds:
    def __init__(self, cloud_images, count=16):
        self.clouds = []

        for i in range(count):
            self.clouds.append(Cloud((random.random() * 99999, random.random() * 99999),
random.choice(cloud_images), random.random() * 0.05 + 0.05, random.random() * 0.6 +
0.2))

        self.clouds.sort(key=lambda x: x.depth)

    def update(self):
        for cloud in self.clouds:
            cloud.update()

    def render(self, surf, offset=(0, 0)):
        for cloud in self.clouds:
            cloud.render(surf, offset=offset)
```

entities.py

```
import math
import random

import pygame

from scripts.particle import Particle
from scripts.spark import Spark

class PhysicsEntity:
    def __init__(self, game, e_type, pos, size):
        self.game = game
        self.type = e_type
        self.pos = list(pos)
        self.size = size
        self.velocity = [0, 0]
        self.collisions = {'up': False, 'down': False, 'right': False, 'left': False}

        self.action = ''
        self.anim_offset = (-3, -3)
        self.flip = False
        self.set_action('idle')

        self.last_movement = [0, 0]

    def rect(self):
        return pygame.Rect(self.pos[0], self.pos[1], self.size[0], self.size[1])

    def set_action(self, action):
        if action != self.action:
            self.action = action
            self.animation = self.game.assets[self.type + '/' + self.action].copy()

    def update(self, tilemap, movement=(0, 0)):
        self.collisions = {'up': False, 'down': False, 'right': False, 'left': False}

        frame_movement = (movement[0] + self.velocity[0], movement[1] +
self.velocity[1])

        self.pos[0] += frame_movement[0]
        entity_rect = self.rect()
        for rect in tilemap.physics_rects_around(self.pos):
            if entity_rect.colliderect(rect):
                if frame_movement[0] > 0:
                    entity_rect.right = rect.left
                    self.collisions['right'] = True
                if frame_movement[0] < 0:
                    entity_rect.left = rect.right
                    self.collisions['left'] = True
                self.pos[0] = entity_rect.x

        self.pos[1] += frame_movement[1]
        entity_rect = self.rect()
        for rect in tilemap.physics_rects_around(self.pos):
            if entity_rect.colliderect(rect):
```

```
                if frame_movement[1] > 0:
                    entity_rect.bottom = rect.top
                    self.collisions['down'] = True
                if frame_movement[1] < 0:
                    entity_rect.top = rect.bottom
                    self.collisions['up'] = True
                self.pos[1] = entity_rect.y

        if movement[0] > 0:
            self.flip = False
        if movement[0] < 0:
            self.flip = True

        self.last_movement = movement

        self.velocity[1] = min(5, self.velocity[1] + 0.1)

        if self.collisions['down'] or self.collisions['up']:
            self.velocity[1] = 0

        self.animation.update()

    def render(self, surf, offset=(0, 0)):
        surf.blit(pygame.transform.flip(self.animation.img(), self.flip, False),
(self.pos[0] - offset[0] + self.anim_offset[0], self.pos[1] - offset[1] +
self.anim_offset[1]))

class Enemy(PhysicsEntity):
    def __init__(self, game, pos, size):
        super().__init__(game, 'enemy', pos, size)

        self.walking = 0

    def update(self, tilemap, movement=(0, 0)):
        if self.walking:
            if tilemap.solid_check((self.rect().centerx + (-7 if self.flip else 7),
self.pos[1] + 23)):
                if (self.collisions['right'] or self.collisions['left']):
                    self.flip = not self.flip
                else:
                    movement = (movement[0] - 0.5 if self.flip else 0.5, movement[1])
            else:
                self.flip = not self.flip
            self.walking = max(0, self.walking - 1)
            if not self.walking:
                dis = (self.game.player.pos[0] - self.pos[0], self.game.player.pos[1] -
self.pos[1])
                if (abs(dis[1]) < 16):
                    if (self.flip and dis[0] < 0):
                        self.game.sfx['shoot'].play()
                        self.game.projectiles.append([[self.rect().centerx - 7,
self.rect().centery], -1.5, 0])
                        for i in range(4):
                            self.game.sparks.append(Spark(self.game.projectiles[-1][0],
random.random() - 0.5 + math.pi, 2 + random.random()))
                    if (not self.flip and dis[0] > 0):
```

```
                        self.game.sfx['shoot'].play()
                        self.game.projectiles.append([[self.rect().centerx + 7,
self.rect().centery], 1.5, 0])
                        for i in range(4):
                            self.game.sparks.append(Spark(self.game.projectiles[-1][0],
random.random() - 0.5, 2 + random.random()))
        elif random.random() < 0.01:
            self.walking = random.randint(30, 120)

        super().update(tilemap, movement=movement)

        if movement[0] != 0:
            self.set_action('run')
        else:
            self.set_action('idle')

        if abs(self.game.player.dashing) >= 50:
            if self.rect().colliderect(self.game.player.rect()):
                self.game.screenshake = max(16, self.game.screenshake)
                self.game.sfx['hit'].play()
                for i in range(30):
                    angle = random.random() * math.pi * 2
                    speed = random.random() * 5
                    self.game.sparks.append(Spark(self.rect().center, angle, 2 +
random.random()))
                    self.game.particles.append(Particle(self.game, 'particle',
self.rect().center, velocity=[math.cos(angle + math.pi) * speed * 0.5, math.sin(angle +
math.pi) * speed * 0.5], frame=random.randint(0, 7)))
                self.game.sparks.append(Spark(self.rect().center, 0, 5 +
random.random()))
                self.game.sparks.append(Spark(self.rect().center, math.pi, 5 +
random.random()))
                return True

    def render(self, surf, offset=(0, 0)):
        super().render(surf, offset=offset)

        if self.flip:
            surf.blit(pygame.transform.flip(self.game.assets['gun'], True, False),
(self.rect().centerx - 4 - self.game.assets['gun'].get_width() - offset[0],
self.rect().centery - offset[1]))
        else:
            surf.blit(self.game.assets['gun'], (self.rect().centerx + 4 - offset[0],
self.rect().centery - offset[1]))

class Player(PhysicsEntity):


    def __init__(self, game, pos, size):
        super().__init__(game, 'player', pos, size)
        self.air_time = 0
        self.jumps = 1
        self.wall_slide = False
        self.dashing = 0
        self.big_jump_cd = 0  # Initial cooldown is 0 (available to use)
        self.big_jump_max_cd = 120  # Example: 5 seconds (assuming 60 frames per second
```

```
    def get_dash_CD(self):
        return 60 - abs(self.dashing)

    def update(self, tilemap, movement=(0, 0)):

        super().update(tilemap, movement=movement)

        self.air_time += 1

        if self.air_time > 150:
            if not self.game.dead:
                self.game.screenshake = max(16, self.game.screenshake)
            self.game.dead += 1
            self.game.sfx['hit'].play()

        if self.collisions['down']:
            self.air_time = 0
            self.jumps = 1

        self.wall_slide = False
        if (self.collisions['right'] or self.collisions['left']) and self.air_time > 4:
            self.wall_slide = True
            self.velocity[1] = min(self.velocity[1], 0.5)
            if self.collisions['right']:
                self.flip = False
            else:
                self.flip = True
            self.set_action('wall_slide')

        if not self.wall_slide:
            if self.air_time > 4:
                self.set_action('jump')
            elif movement[0] != 0:
                self.set_action('run')
            else:
                self.set_action('idle')

        if abs(self.dashing) in {60, 50}:
            for i in range(20):
                angle = random.random() * math.pi * 2
                speed = random.random() * 0.5 + 0.5
                pvelocity = [math.cos(angle) * speed, math.sin(angle) * speed]
                self.game.particles.append(Particle(self.game, 'particle',
self.rect().center, velocity=pvelocity, frame=random.randint(0, 7)))
        if self.dashing > 0:
            self.dashing = max(0, self.dashing - 1)
        if self.dashing < 0:
            self.dashing = min(0, self.dashing + 1)
        if abs(self.dashing) > 50:
            self.velocity[0] = abs(self.dashing) / self.dashing * 8
            if abs(self.dashing) == 51:
                self.velocity[0] *= 0.1
            pvelocity = [abs(self.dashing) / self.dashing * random.random() * 3, 0]
            self.game.particles.append(Particle(self.game, 'particle',
self.rect().center, velocity=pvelocity, frame=random.randint(0, 7)))
```

```
        if self.velocity[0] > 0:
            self.velocity[0] = max(self.velocity[0] - 0.1, 0)
        else:
            self.velocity[0] = min(self.velocity[0] + 0.1, 0)


        if self.big_jump_cd < self.big_jump_max_cd:
            self.big_jump_cd += 1

    def render(self, surf, offset=(0, 0)):

        if abs(self.dashing) <= 50:
            super().render(surf, offset=offset)




    def jump(self):
        if self.wall_slide:
            if self.flip and self.last_movement[0] < 0:
                self.velocity[0] = 3.5
                self.velocity[1] = -2.5
                self.air_time = 5
                self.jumps = max(0, self.jumps - 1)
                return True
            elif not self.flip and self.last_movement[0] > 0:
                self.velocity[0] = -3.5
                self.velocity[1] = -2.5
                self.air_time = 5
                self.jumps = max(0, self.jumps - 1)
                return True

        elif self.jumps:
            self.velocity[1] = -3.5
            self.jumps -= 1
            self.air_time = 5
            return True

    def dash(self):
        if not self.dashing:
            self.game.sfx['dash'].play()
            if self.flip:
                self.dashing = -60
            else:
                self.dashing = 60
    def get_big_jump_CD(self):
        """Return the current cooldown value for the big jump."""
        return self.big_jump_cd

    #write a big jump function with bigger CD, independent of normal jumps, also anothe
function which returns a CD value for the UI
    def big_jump(self):
        if self.big_jump_cd == self.big_jump_max_cd:
            self.velocity[1] = -5  # Bigger upward velocity for a bigger jump
            self.big_jump_cd = 0
```

```
            self.air_time = -150
```

object.py

```
import pygame
class ObjectAnimation:
    def __init__(self, pos, images):
        self.images = images
        self.index = 0
        self.image = self.images[self.index]
        self.rect = self.image.get_rect(topleft=pos)
        self.animation_time = pygame.time.get_ticks()


    def animate(self, dt):
        if pygame.time.get_ticks() - self.animation_time > 100:  # change 100 to adjust
speed
            self.index = (self.index + 1) % len(self.images)
            self.image = self.images[self.index]
            self.animation_time = pygame.time.get_ticks()

    def draw(self, surface):
        surface.blit(self.image, self.rect.topleft)
```

particle.py

```
class Particle:
    def __init__(self, game, p_type, pos, velocity=[0, 0], frame=0):
        self.game = game
        self.type = p_type
        self.pos = list(pos)
        self.velocity = list(velocity)
        self.animation = self.game.assets['particle/' + p_type].copy()
        self.animation.frame = frame

    def update(self):
        kill = False
        if self.animation.done:
            kill = True

        self.pos[0] += self.velocity[0]
        self.pos[1] += self.velocity[1]

        self.animation.update()

        return kill

    def render(self, surf, offset=(0, 0)):
        img = self.animation.img()
        surf.blit(img, (self.pos[0] - offset[0] - img.get_width() // 2, self.pos[1] -
offset[1] - img.get_height() // 2))
```

spark.py

```
import math
import pygame

class Spark:
    def __init__(self, pos, angle, speed):
        self.pos = list(pos)
        self.angle = angle
        self.speed = speed

    def update(self):
        self.pos[0] += math.cos(self.angle) * self.speed
        self.pos[1] += math.sin(self.angle) * self.speed

        self.speed = max(0, self.speed - 0.1)
        return not self.speed

    def render(self, surf, offset=(0, 0)):
        render_points = [
            (self.pos[0] + math.cos(self.angle) * self.speed * 3 - offset[0],
self.pos[1] + math.sin(self.angle) * self.speed * 3 - offset[1]),
            (self.pos[0] + math.cos(self.angle + math.pi * 0.5) * self.speed * 0.5 -
offset[0], self.pos[1] + math.sin(self.angle + math.pi * 0.5) * self.speed * 0.5 -
offset[1]),
            (self.pos[0] + math.cos(self.angle + math.pi) * self.speed * 3 - offset[0],
self.pos[1] + math.sin(self.angle + math.pi) * self.speed * 3 - offset[1]),
            (self.pos[0] + math.cos(self.angle - math.pi * 0.5) * self.speed * 0.5 -
offset[0], self.pos[1] + math.sin(self.angle - math.pi * 0.5) * self.speed * 0.5 -
offset[1]),
        ]

        pygame.draw.polygon(surf, (255, 255, 255), render_points)
```

tilemap.py

```
import json
import pygame

AUTOTILE_MAP = {
    tuple(sorted([(1, 0), (0, 1)])): 0,
    tuple(sorted([(1, 0), (0, 1), (-1, 0)])): 1,
    tuple(sorted([(-1, 0), (0, 1)])): 2,
    tuple(sorted([(-1, 0), (0, -1), (0, 1)])): 3,
    tuple(sorted([(-1, 0), (0, -1)])): 4,
    tuple(sorted([(-1, 0), (0, -1), (1, 0)])): 5,
    tuple(sorted([(1, 0), (0, -1)])): 6,
    tuple(sorted([(1, 0), (0, -1), (0, 1)])): 7,
    tuple(sorted([(1, 0), (-1, 0), (0, 1), (0, -1)])): 8,
}

NEIGHBOR_OFFSETS = [(-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (0, 0), (-1, 1), (0,
1), (1, 1)]
PHYSICS_TILES = {'grass', 'stone'}
PICKUP_TILES = {'heart','coin'}
AUTOTILE_TYPES = {'grass', 'stone'}

class Tilemap:
    def __init__(self, game, tile_size=16):
        self.game = game
        self.tile_size = tile_size
        self.tilemap = {}
        self.offgrid_tiles = []

    def extract(self, id_pairs, keep=False):
        matches = []
        for tile in self.offgrid_tiles.copy():
            if (tile['type'], tile['variant']) in id_pairs:
                matches.append(tile.copy())
                if not keep:
                    self.offgrid_tiles.remove(tile)

        for loc in self.tilemap:
            tile = self.tilemap[loc]
            if (tile['type'], tile['variant']) in id_pairs:
                matches.append(tile.copy())
                matches[-1]['pos'] = matches[-1]['pos'].copy()
                matches[-1]['pos'][0] *= self.tile_size
                matches[-1]['pos'][1] *= self.tile_size
                if not keep:
                    del self.tilemap[loc]
        return matches

    def tiles_around(self, pos):
        tiles = []
        tile_loc = (int(pos[0] // self.tile_size), int(pos[1] // self.tile_size))
        for offset in NEIGHBOR_OFFSETS:
            check_loc = str(tile_loc[0] + offset[0]) + ';' + str(tile_loc[1] +
offset[1])
            if check_loc in self.tilemap:
```

```
                tiles.append(self.tilemap[check_loc])
        return tiles

    def save(self, path):
        f = open(path, 'w')
        json.dump({'tilemap': self.tilemap, 'tile_size': self.tile_size, 'offgrid':
self.offgrid_tiles}, f)
        f.close()

    def load(self, path):
        f = open(path, 'r')
        map_data = json.load(f)
        f.close()

        self.tilemap = map_data['tilemap']
        self.tile_size = map_data['tile_size']
        self.offgrid_tiles = map_data['offgrid']

    def solid_check(self, pos):
        tile_loc = str(int(pos[0] // self.tile_size)) + ';' + str(int(pos[1] //
self.tile_size))
        if tile_loc in self.tilemap:
            if self.tilemap[tile_loc]['type'] in PHYSICS_TILES:
                return self.tilemap[tile_loc]

    def physics_rects_around(self, pos):
        rects = []
        for tile in self.tiles_around(pos):
            if tile['type'] in PHYSICS_TILES:
                rects.append(pygame.Rect(tile['pos'][0] * self.tile_size, tile['pos'][1]
* self.tile_size, self.tile_size, self.tile_size))
        return rects

    def pickup_check(self, pos):
        tile_loc = str(int(pos[0] // self.tile_size)) + ';' + str(int(pos[1] //
self.tile_size))
        if tile_loc in self.tilemap:
            if self.tilemap[tile_loc]['type'] in PICKUP_TILES:
                return self.tilemap[tile_loc]


    def render(self, surf, offset=(0, 0)):
        for tile in self.offgrid_tiles:
            surf.blit(self.game.assets[tile['type']][tile['variant']], (tile['pos'][0] -
offset[0], tile['pos'][1] - offset[1]))

        for x in range(offset[0] // self.tile_size, (offset[0] + surf.get_width()) //
self.tile_size + 1):
            for y in range(offset[1] // self.tile_size, (offset[1] + surf.get_height())
// self.tile_size + 1):
                loc = str(x) + ';' + str(y)
                if loc in self.tilemap:
                    tile = self.tilemap[loc]
                    surf.blit(self.game.assets[tile['type']][tile['variant']],
(tile['pos'][0] * self.tile_size - offset[0], tile['pos'][1] * self.tile_size -
offset[1]))
```

```
    def autotile(self):
        for loc in self.tilemap:
            tile = self.tilemap[loc]
            neighbors = set()
            for shift in [(1, 0), (-1, 0), (0, -1), (0, 1)]:
                check_loc = str(tile['pos'][0] + shift[0]) + ';' + str(tile['pos'][1] +
shift[1])
                if check_loc in self.tilemap:
                    if self.tilemap[check_loc]['type'] == tile['type']:
                        neighbors.add(shift)
            neighbors = tuple(sorted(neighbors))
            if (tile['type'] in AUTOTILE_TYPES) and (neighbors in AUTOTILE_MAP):
                tile['variant'] = AUTOTILE_MAP[neighbors]
```

util.py

```
import os
import pygame

BASE_IMG_PATH = 'data/images/'

def load_image(path):

   img = pygame.image.load(BASE_IMG_PATH + path).convert()
   img.set_colorkey((0, 0, 0))
   return img

def load_images(path):
   images = []
   for img_name in sorted(os.listdir(BASE_IMG_PATH + path)):
       images.append(load_image(path + '/' + img_name))
   return images

class Animation:
   def __init__(self, images, img_dur=5, loop=True):
       self.images = images
       self.loop = loop
       self.img_duration = img_dur
       self.done = False
       self.frame = 0

   def copy(self):
       return Animation(self.images, self.img_duration, self.loop)

   def update(self):
       if self.loop:
           self.frame = (self.frame + 1) % (self.img_duration * len(self.images))
       else:
           self.frame = min(self.frame + 1, self.img_duration * len(self.images) - 1)
           if self.frame >= self.img_duration * len(self.images) - 1:
               self.done = True

   def img(self):
       return self.images[int(self.frame / self.img_duration)]

def APPLE_FILE_CLEAR():
   for dirpath, dirnames, filenames in os.walk('.'):
       if '.DS_Store' in filenames:
           file_path = os.path.join(dirpath, '.DS_Store')
           try:
               os.remove(file_path)
               print(f"Removed {file_path}")
           except Exception as e:
               print(f"Error removing {file_path}: {e}")

def ease_out_quad(t):
   """Quadratic easing out - decelerating to zero velocity."""
   return -t * (t - 2)
```

main_game.py

```
import sys
import math
import random
import os
import datetime
import json
import pygame

from PIL import Image, ImageFilter, ImageEnhance

from scripts.utils import load_image, load_images, Animation, APPLE_FILE_CLEAR,
ease_out_quad
from scripts.entities import PhysicsEntity, Player, Enemy
from scripts.tilemap import Tilemap
from scripts.clouds import Clouds
from scripts.particle import Particle
from scripts.spark import Spark
from scripts.object import ObjectAnimation

RUNNING_FPS = 60



# DISPLAYING_FPS = 120

class Game:
    def __init__(self):
        self.player_name = input('enter your name:')
        pygame.init()

        APPLE_FILE_CLEAR()
        self.x = 0

        pygame.display.set_caption('My game')
        self.window_size = (1280, 720)
        # self.window_size = (1920, 1080)
        self.screen = pygame.display.set_mode(self.window_size)
        self.display = pygame.Surface((640, 360), pygame.SRCALPHA)
        self.show_debug_menu = False
        self.textflash = 0

        self.display_WIDTH = self.display.get_width()
        self.display_HEIGHT = self.display.get_height()
        self.display_2 = pygame.Surface((640, 360))

        self.heart_animations = []
        self.total_time = 0
        self.current_level_time = 0
        self.restart_count = 0
        self.player_got_hit_count = 0
        self.in_main_menu = True
        self.render_scroll = [0, 0]

        self.level_keys = {
```

```
                    pygame.K_0: 0,
                    pygame.K_1: 1,
                    pygame.K_2: 2,
                    pygame.K_3: 3,
                    pygame.K_4: 4,
                    pygame.K_5: 5,
                    pygame.K_6: 6,
                    pygame.K_7: 7,
                }

        # Debug variables
        self.debug_timer = 0
        self.debug_font = pygame.font.Font(None, 30)
        self.debug_info_updated = False
        self.GRANDTOTAL_COUNTER = 0

        self.clock = pygame.time.Clock()

        self.movement = [False, False]

        self.font = pygame.font.Font('minecraft.otf', 20)

        self.assets = {
            'decor': load_images('tiles/decor'),
            'grass': load_images('tiles/grass'),
            'large_decor': load_images('tiles/large_decor'),
            'stone': load_images('tiles/stone'),
            'player': load_image('entities/player.png'),
            'background': load_image('background2.png'),
            'clouds': load_images('clouds'),
            'enemy/idle': Animation(load_images('entities/enemy/idle'), img_dur=6),
            'enemy/run': Animation(load_images('entities/enemy/run'), img_dur=4),
            'player/idle': Animation(load_images('entities/player/idle'), img_dur=6),
            'player/run': Animation(load_images('entities/player/run'), img_dur=7),
            'player/jump': Animation(load_images('entities/player/jump')),
            'player/slide': Animation(load_images('entities/player/slide')),
            'player/wall_slide': Animation(load_images('entities/player/wall_slide')),
            'particle/leaf': Animation(load_images('particles/leaf'), img_dur=20,
loop=False),
            'particle/particle': Animation(load_images('particles/particle'), img_dur=6,
loop=False),
            'gun': load_image('gun.png'),
            'projectile': load_image('projectile.png'),
            'arrow_keys_diagram': load_image('上下左右.png'),
            'x_key_diagram': load_image('objects/keyboard_X.png'),
            'enemy_head': load_image('objects/enemy_head.png'),
            'e_key_diagram': load_image('objects/keyboard_Z.png'),
            'blurred_background': load_image('blurred_background.png'),
            'game_logo': load_image('objects/game_logo.png'),
            'in_game_pause_menu_status': pygame.image.load(
                'data/images/objects/in_game_pause_menu_status.png').convert_alpha(),
            'button_selector': load_image('objects/button_selector.png'),
            'leaderboard': load_image('objects/leaderboard.png'),
            'main_menu_buttons':
pygame.image.load('data/images/objects/main_menu_buttons.png').convert_alpha(),
            'main_menu_button_selector':
```

```
load_image('objects/main_menu_button_selector.png').convert_alpha(),
            'game_end_panel':
pygame.image.load('data/images/objects/game_end_panel.png').convert_alpha(),

        }

        self.raw_assets = {
            'hearts': load_images('objects/hearts'),
            'dash_UI_icon':
pygame.image.load('data/images/objects/dash_UI_icon.png').convert_alpha(),
            'jump_UI_icon':
pygame.image.load('data/images/objects/jump_UI_icon.png').convert_alpha(),
            'stopwatch':
pygame.image.load('data/images/objects/stopwatch.png').convert_alpha(),
            'status_menu':
pygame.image.load('data/images/objects/status_menu.png').convert_alpha(),
            'menuidleplayer': load_images('entities/player/idle2x'),
            'skeleton':
pygame.image.load('data/images/objects/skeleton.png').convert_alpha(),
            'arrows':
pygame.image.load('data/images/objects/arrows.png').convert_alpha(),

        }

        self.clouds = Clouds(self.assets['clouds'], count=16)

        self.sfx = {
            'jump': pygame.mixer.Sound('data/sfx/jump.wav'),
            'dash': pygame.mixer.Sound('data/sfx/dash.wav'),
            'shoot': pygame.mixer.Sound('data/sfx/shoot.wav'),
            'hit': pygame.mixer.Sound('data/sfx/hit.wav'),
            'ambience': pygame.mixer.Sound('data/sfx/ambience.wav'),
        }

        self.HP = 3
        self.sfx['jump'].set_volume(0.7)
        self.sfx['dash'].set_volume(0.3)
        self.sfx['shoot'].set_volume(0.4)
        self.sfx['hit'].set_volume(0.8)
        self.sfx['ambience'].set_volume(0.2)

        self.take_screenshot = False

        self.player = Player(self, (50, 50), (8, 15))

        self.tilemap = Tilemap(self, tile_size=16)

        self.load_level('main_menu_map')
        self.level = 0

        self.screenshake = 0

        self.paused = False

        pygame.mixer.music.load('data/music.wav')
        pygame.mixer.music.set_volume(0.5)
```

```
        pygame.mixer.music.play(-1)

        self.keyboard_x = self.assets['x_key_diagram']
        player_box_pos = (100, 140)  # Center of the player box
        self.playermodel = ObjectAnimation(player_box_pos,
self.raw_assets['menuidleplayer'])

        self.button_selector_position = 69
        self.button_selector_target_position = 69
        self.button_selector_speed = 7

        self.end_panel_pos = -self.assets['game_end_panel'].get_height()
        self.end_panel_target_pos = (self.display_HEIGHT -
self.assets['game_end_panel'].get_height()) // 2

        # if data/player_records.json doesnt exist, create a blank json file
        if not os.path.exists('data/player_records.json'):
            with open('data/player_records.json', 'w') as f:
                f.write('[]')

        for i in range(self.HP):
            pos = (i * (self.raw_assets['hearts'][0].get_width() * 1.5) + 20,
                   self.display_HEIGHT // 15)  # Change 32 and 10 accordingly
            self.heart_animations.append(ObjectAnimation(pos,
self.raw_assets['hearts']))

        self.show_pause_menu = False
        self.pause_menu_position = -self.assets['in_game_pause_menu_status'].get_width()
# Start off-screen to the left
        self.pause_menu_target_position = (self.display_WIDTH - self.assets[
            'in_game_pause_menu_status'].get_width()) // 2  # Middle of the screen minus
half the width of the status menu

    def draw_text(self, text, font, color, surface, pos):
        textobj = font.render(text, 1, color)
        textrect = textobj.get_rect()
        textrect.topleft = pos
        surface.blit(textobj, textrect)

    def load_level(self, map_id):
        self.tilemap.load('data/maps/' + str(map_id) + '.json')

        self.leaf_spawners = []
        for tree in self.tilemap.extract([('large_decor', 2)], keep=True):
            self.leaf_spawners.append(pygame.Rect(4 + tree['pos'][0], 4 +
tree['pos'][1], 23, 13))

        self.enemies = []
        for spawner in self.tilemap.extract([('spawners', 0), ('spawners', 1)]):
            if spawner['variant'] == 0:
                self.player.pos = spawner['pos']
                self.player.air_time = 0
            else:
                self.enemies.append(Enemy(self, spawner['pos'], (8, 15)))

        self.projectiles = []
```

```
        self.particles = []
        self.sparks = []

        self.scroll = [0, 0]
        self.dead = 0
        self.transition = -50

    def projectile_render_update(self):
        for projectile in self.projectiles.copy():
            projectile[0][0] += projectile[1]
            projectile[2] += 1
            img = self.assets['projectile']
            self.display.blit(img, (projectile[0][0] - img.get_width() / 2 -
self.render_scroll[0],
                                    projectile[0][1] - img.get_height() / 2 -
self.render_scroll[1]))
            if self.tilemap.solid_check(projectile[0]):
                self.projectiles.remove(projectile)
                for i in range(4):
                    self.sparks.append(
                        Spark(projectile[0], random.random() - 0.5 + (math.pi if
projectile[1] > 0 else 0),
                              2 + random.random()))
            elif projectile[2] > 360:
                self.projectiles.remove(projectile)
            elif abs(self.player.dashing) < 50:
                if self.player.rect().collidepoint(projectile[0]):
                    self.projectiles.remove(projectile)
                    self.HP -= 1
                    self.player_got_hit_count += 1
                    self.sfx['hit'].play()
                    self.screenshake = max(16, self.screenshake)
                    for i in range(30):
                        angle = random.random() * math.pi * 2
                        speed = random.random() * 5
                        self.sparks.append(Spark(self.player.rect().center, angle, 2 +
random.random()))
                        self.particles.append(Particle(self, 'particle',
self.player.rect().center,
                                                       velocity=[math.cos(angle +
math.pi) * speed * 0.5,
                                                                 math.sin(angle +
math.pi) * speed * 0.5],
                                                       frame=random.randint(0, 7)))

    def spark_particle_render_update(self):
        for spark in self.sparks.copy():
            kill = spark.update()
            spark.render(self.display, offset=self.render_scroll)
            if kill:
                self.sparks.remove(spark)

        for particle in self.particles.copy():
            kill = particle.update()
            particle.render(self.display, offset=self.render_scroll)
            if particle.type == 'leaf':
```

```
                particle.pos[0] += math.sin(particle.animation.frame * 0.035) * 0.3
            if kill:
                self.particles.remove(particle)

    def cloud_leaf_render_update(self):
        for rect in self.leaf_spawners:
            if random.random() * 49999 < rect.width * rect.height:
                pos = (rect.x + random.random() * rect.width, rect.y + random.random() *
rect.height)
                self.particles.append(Particle(self, 'leaf', pos, velocity=[-0.1, 0.3],
frame=random.randint(0, 20)))

        self.clouds.update()
        self.clouds.render(self.display, offset=self.render_scroll)

    def scroll_update(self):

        # if in_main_menu: the camera should be higher
        if self.in_main_menu:
            self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2
- self.scroll[0]) // 30 - 4
            self.scroll[1] += (self.player.rect().centery - self.display.get_height() /
2 - self.scroll[1]) // 30 - 3
        else:
            self.scroll[1] += (self.player.rect().centery - self.display.get_height() /
2 - self.scroll[1]) // 30
            self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2
- self.scroll[0]) // 30

        self.render_scroll = [int(self.scroll[0]), int(self.scroll[1])]

    def entity_render_update(self):
        if not self.dead:
            self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0))
            self.player.render(self.display, offset=self.render_scroll)

        for enemy in self.enemies.copy():
            kill = enemy.update(self.tilemap, (0, 0))
            enemy.render(self.display, offset=self.render_scroll)
            if kill:
                self.enemies.remove(enemy)

    def reset_current_level_timer(self):
        self.current_level_time = 0

    def restart_level(self):

        self.restart_count += 1
        self.load_level(self.level)
        self.reset_movement()
        self.reset_player_status()
        self.reset_current_level_timer()
        self.main_game()

    def draw_hearts(self):
        for heart in self.heart_animations[:self.HP]:
```

```
            heart.animate(pygame.time.get_ticks())
            heart.draw(self.display)

    def save_global_time(self):
        self.total_time += self.current_level_time
        self.current_level_time = 0

    def check_level_loading(self):

        if self.level == len(os.listdir('data/maps')) - 2 and not len(self.enemies):
            self.save_global_time()
            # wait 2 seconds, then go to game_end

            self.game_finished = True
            self.game_end()

        if not len(self.enemies):

            self.transition += 1
            if self.transition > 40:
                self.level += 1

                self.save_global_time()

                if self.level == len(os.listdir('data/maps')) - 1:
                    print('GAME FINISHED')
                else:

                    self.load_level(self.level)

        if self.transition < 0:
            self.transition += 1

        if self.HP <= 0:
            self.dead += 1
        if self.dead:
            self.dead += 1
            if self.dead >= 10:
                self.transition = min(40, self.transition)
            if self.dead > 40:
                self.restart_level()

        if self.transition:
            transition_surf = pygame.Surface(self.display.get_size())
            pygame.draw.circle(transition_surf, (255, 255, 255),
                               (self.display.get_width() // 2, self.display.get_height()
// 2),
                               (45 - abs(self.transition)) * 8)
            transition_surf.set_colorkey((255, 255, 255))
            self.display.blit(transition_surf, (0, 0))

    def render_end_game_panel(self):

        game_end_panel = self.assets['game_end_panel'].copy()
        print('rendering end game panel')
```

```
        target = self.end_panel_target_pos

        difference = target - self.end_panel_pos
        normalized_difference = abs(difference) / game_end_panel.get_width()

        if normalized_difference > 1:
            normalized_difference = 1
        elif normalized_difference < 0:
            normalized_difference = 0

        # Calculate progress using the easing function
        progress = ease_out_quad(normalized_difference)

        if difference > 0:
            self.end_panel_pos += 13 * progress
            if self.end_panel_pos > target:
                self.leaderboard_position = target
        else:
            self.end_panel_pos -= 13 * progress
            if self.end_panel_pos < target:
                self.end_panel_pos = target

        render_pos = (self.display_WIDTH // 2 - game_end_panel.get_width() // 2,
self.end_panel_pos + self.menu_float)

        # draw text that says extra timef
        self.draw_text('EXTRA TIME:', pygame.font.Font('minecraft.otf', 20), 'black',
game_end_panel, (27, 89))
        self.draw_text('EXTRA TIME:', pygame.font.Font('minecraft.otf', 20), 'white',
game_end_panel, (26, 88))
        # self.draw_text(f'Player got hit: {self.player_got_hit_count} ->
{self.player_got_hit_count*1}s', pygame.font.Font('minecraft.otf', 10), 'black',
game_end_panel, ( 201,  166))
        self.draw_text(f'Player got hit: {self.player_got_hit_count} -> ',
pygame.font.Font('minecraft.otf', 10),
                       'white', game_end_panel, (200, 165))
        self.draw_text(f'{self.player_got_hit_count * 1}s',
pygame.font.Font('minecraft.otf', 10), 'red',
                       game_end_panel, (200 + pygame.font.Font('minecraft.otf',
10).size(
                f'Player got hit: {self.player_got_hit_count} -> ')[0], 165))

        # Draw arrow image a bit below the text
        game_end_panel.blit(self.raw_assets['arrows'], (200, + 125))

        # Draw the text "Restart count:" and the associated count, below the arrow image
        # self.draw_text(f'Restart count: {self.restart_count} ->
{self.restart_count*5}s', pygame.font.Font('minecraft.otf', 10), 'black',
game_end_panel, (31,  + 166))
        self.draw_text(f'Restart count: {self.restart_count} -> ',
pygame.font.Font('minecraft.otf', 10), 'white',
                       game_end_panel, (30, + 165))
        # {self.restart_count*5}s is red
        self.draw_text(f'{self.restart_count * 5}s', pygame.font.Font('minecraft.otf',
10), 'red', game_end_panel, (
            30 + pygame.font.Font('minecraft.otf', 10).size(f'Restart count:
```

```
{self.restart_count} -> ')[0], 165))

        game_end_panel.blit(self.raw_assets['skeleton'], (50, 125))

        # total total time
        self.draw_text(f'TOTAL TIME:', self.font, 'black', game_end_panel, (27, 196))
        self.draw_text(f'TOTAL TIME:', self.font, 'white', game_end_panel, (26, 195))

        self.textflash += 1

        if self.textflash > 60:
            basetime = round(self.total_time, 1)
            # draw a black
            self.draw_text(f'{basetime}', self.font, 'black', game_end_panel, (27, 227))
            self.draw_text(f'{basetime}', self.font, 'white', game_end_panel, (26, 226))

        if self.textflash > 120:
            # draw a black

            self.draw_text(f'+ {self.restart_count * 5}', self.font, 'black',
game_end_panel,
                           (26 + self.font.size(f'{round(self.total_time, 1)} ')[0] + 1,
227))
            self.draw_text(f'+ {self.restart_count * 5}', self.font, 'red',
game_end_panel,
                           (26 + self.font.size(f'{round(self.total_time, 1)} ')[0],
226))

        if self.textflash > 180:
            # draw a black
            self.draw_text(f'+ {self.player_got_hit_count}', self.font, 'black',
game_end_panel, (
                26 + self.font.size(f'{round(self.total_time, 1)} + {self.restart_count
* 5} ')[0] + 1, 227))
            self.draw_text(f'+ {self.player_got_hit_count}', self.font, 'red',
game_end_panel,
                           (26 + self.font.size(f'{round(self.total_time, 1)} +
{self.restart_count * 5} ')[0], 226))

        if self.textflash > 240:
            # add a = sign
            self.draw_text(f'=', self.font, 'black', game_end_panel, (26 +
self.font.size(
                f'{round(self.total_time, 1)} + {self.restart_count * 5} +
{self.player_got_hit_count} ')[0] + 1, 227))

            self.draw_text(f'=', self.font, 'white', game_end_panel, (26 +
self.font.size(
                f'{round(self.total_time, 1)} + {self.restart_count * 5} +
{self.player_got_hit_count} ')[0], 226))

        GRANDTOTAL = round(self.total_time + 5 * self.restart_count +
self.player_got_hit_count, 1)
        if self.textflash > 300:
            if GRANDTOTAL - self.GRANDTOTAL_COUNTER > 1:
```

```
                self.GRANDTOTAL_COUNTER += 1
            elif GRANDTOTAL - self.GRANDTOTAL_COUNTER <= 1:
                self.GRANDTOTAL_COUNTER += 0.1

            if self.GRANDTOTAL_COUNTER > GRANDTOTAL:
                self.GRANDTOTAL_COUNTER = GRANDTOTAL

            td = datetime.timedelta(seconds=int(self.GRANDTOTAL_COUNTER))
            fractional_seconds = round(self.GRANDTOTAL_COUNTER % 1, 1)
            time_formatted = f"{(td.seconds // 60) % 60:02}:{td.seconds % 60:02}.{int(10
* fractional_seconds)}"

            # increment to the grandtotal
            self.draw_text(f'{time_formatted}', self.font, 'black', game_end_panel, (26
+ self.font.size(
                f'{round(self.total_time, 1)} + {self.restart_count * 5} +
{self.player_got_hit_count} = ')[0] + 1,

227))
            self.draw_text(f'{time_formatted}', self.font, 'white', game_end_panel, (26
+ self.font.size(
                f'{round(self.total_time, 1)} + {self.restart_count * 5} +
{self.player_got_hit_count} = ')[0], 226))

        # put a flashing text at the buttom of the panel, say PRESS ENTER FOR MAIN MENU

        # print(self.textflash)
        if self.textflash % 60 < 30:
            # but the text middled at the buttom of the panel
            self.draw_text('PRESS ENTER FOR MAIN MENU', self.font, 'black',
game_end_panel, (
                game_end_panel.get_width() // 2 - self.font.size('PRESS ENTER FOR MAIN
MENU')[0] // 2 + 1,
                game_end_panel.get_height() - 49))
            self.draw_text('PRESS ENTER FOR MAIN MENU', self.font, 'white',
game_end_panel, (
                game_end_panel.get_width() // 2 - self.font.size('PRESS ENTER FOR MAIN
MENU')[0] // 2,
                game_end_panel.get_height() - 50))

        self.display.blit(game_end_panel, render_pos)

    def game_end(self):
        self.screenshot_taken = False
        today = datetime.date.today()
        current_time = datetime.datetime.now().strftime('%H:%M')

        player_record_entry = {
            'player name': self.player_name,
            'total time': round(self.total_time + 5 * self.restart_count +
self.player_got_hit_count, 1),
            'raw time': round(self.total_time, 5),
            'got hit count': self.player_got_hit_count,
            'restart count': self.player_got_hit_count,
            'completed datetime': f'{today} @ {current_time}'
        }
```

```
        # Proper way to update the JSON file
        with open('data/player_records.json', 'r') as f:
            try:
                data = json.load(f)
            except json.decoder.JSONDecodeError:  # In case the file is empty or has
invalid JSON format
                data = []

        data.append(player_record_entry)

        with open('data/player_records.json', 'w') as f:
            json.dump(data, f, indent=4)

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

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_RETURN:
                        self.entire_game_reset()
                        self.main_menu()

            if not self.screenshot_taken:
                if self.pause_menu_position < -490:
                    pygame.image.save(self.display, 'screenshot.png')
                OriImage = Image.open('screenshot.png')
                screenshot_no_blur = pygame.image.load('screenshot.png')
                blurImage = OriImage.filter(ImageFilter.GaussianBlur(2.5))
                # make blurimage darker
                enhancer = ImageEnhance.Brightness(blurImage)
                blurImage = enhancer.enhance(0.7)
                blurImage.save('simBlurImage.png')
                print('screenshot taken')

                blurImageBG = pygame.image.load('simBlurImage.png')
                self.screenshot_taken = True

            self.display.blit(screenshot_no_blur, (0, 0))
            self.refresh_menu_float()
            self.update_debug_stuff()

            self.render_end_game_panel()

            self.screen.blit(pygame.transform.scale(self.display,
self.screen.get_size()), self.screenshake_offset)
            if self.show_debug_menu:
                self.screen.blit(self.debug_surface, (0, 0))
            pygame.display.update()
            self.clock.tick(RUNNING_FPS)

    def running_timer(self):
        #use
        self.current_level_time += 1 / RUNNING_FPS
```

```
        time_text = str(datetime.timedelta(seconds=round(self.current_level_time, 0)))
        self.draw_text(time_text, pygame.font.Font('minecraft.otf', 20), 'black',
self.display, (
            self.display_WIDTH // 20 * 16.5 - self.raw_assets['stopwatch'].get_width()
// 2 + 4, 34 + self.menu_float))
        self.draw_text(time_text, pygame.font.Font('minecraft.otf', 20), 'white',
self.display, (
            self.display_WIDTH // 20 * 16.5 - self.raw_assets['stopwatch'].get_width()
// 2 + 2, 32 + self.menu_float))

        # also display the total time+current level time, in a smaller font, at the top
right corner, below the current_level_time timer
        total_time_text = str(datetime.timedelta(seconds=round(self.total_time +
self.current_level_time, 0)))
        self.draw_text(total_time_text, pygame.font.Font('minecraft.otf', 10), 'black',
self.display, (
            self.display_WIDTH // 20 * 16.5 - self.raw_assets['stopwatch'].get_width()
// 2 + 3, 53 + self.menu_float))

        self.draw_text(total_time_text, pygame.font.Font('minecraft.otf', 10), 'white',
self.display, (
            self.display_WIDTH // 20 * 16.5 - self.raw_assets['stopwatch'].get_width()
// 2 + 2, 52 + self.menu_float))

        self.display.blit(self.raw_assets['stopwatch'], (
            self.display_WIDTH // 20 * 19 - self.raw_assets['stopwatch'].get_width() //
2, 20 + self.menu_float))

    def show_enemy_count(self):
        self.display.blit(self.assets['enemy_head'],
                          (self.display_WIDTH // 20 * 1, self.display_HEIGHT // 20 * 4 +
self.menu_float))
        self.draw_text(f'x {len(self.enemies)}', pygame.font.Font('minecraft.otf', 20),
'black', self.display,
                       (self.display_WIDTH // 20 * 2.3 + 2, self.display_HEIGHT // 20 *
4.5 + self.menu_float + 2))
        self.draw_text(f'x {len(self.enemies)}', pygame.font.Font('minecraft.otf', 20),
'white', self.display,
                       (self.display_WIDTH // 20 * 2.3, self.display_HEIGHT // 20 * 4.5
+ self.menu_float))

    def show_level_number(self):
        # show level number at the buttom right corner of the screen
        if self.level == 0:
            level_name = 'Tutorial'
        else:
            level_name = f'Level {self.level}'

        self.draw_text(level_name, pygame.font.Font('minecraft.otf', 20), 'black',
self.display, (
            self.display_WIDTH // 20 * 18.5 - self.font.size(level_name)[0] // 2 + 2,
            self.display_HEIGHT // 20 * 18.5 - self.font.size(level_name)[1] // 2 + 2 +
self.menu_float))
```

```
        self.draw_text(level_name, pygame.font.Font('minecraft.otf', 20), 'white',
self.display, (
            self.display_WIDTH // 20 * 18.5 - self.font.size(level_name)[0] // 2,
            self.display_HEIGHT // 20 * 18.5 - self.font.size(level_name)[1] // 2 +
self.menu_float))

    def displayUI_EVERYTHING(self):
        self.refresh_menu_float()
        self.render_pause_menu()
        self.draw_hearts()
        if self.level == 0:
            self.display_welcome()
        if self.level:
            self.running_timer()
            self.show_enemy_count()
        self.display_CD_block()

        self.show_level_number()

    def update_debug_stuff(self):

        self.debug_surface = pygame.Surface((self.screen.get_width(),
self.screen.get_height()), pygame.SRCALPHA)

        currentFPS = f'FPS: {self.clock.get_fps():.0f}'

        if self.show_debug_menu:
            # display different variables

            self.draw_text(currentFPS, self.debug_font, 'white', self.debug_surface,
                           (self.debug_surface.get_width() - 100, 0))

            self.draw_text(f'self.level: {self.level}', self.debug_font, 'white',
self.debug_surface, (0, 0))
            self.draw_text(f'self.pause_menu_position: {self.pause_menu_position}',
self.debug_font, 'white',
                           self.debug_surface, (0, 25))
            self.draw_text(f'self.transition: {self.transition}', self.debug_font,
'white', self.debug_surface, (0, 50))
            self.draw_text(f'self.dead: {self.dead}', self.debug_font, 'white',
self.debug_surface, (0, 75))
            self.draw_text(f'self.restart_count: {self.restart_count}', self.debug_font,
'white', self.debug_surface,
                           (0, 100))
            self.draw_text(f'self.player_got_hit_count: {self.player_got_hit_count}',
self.debug_font, 'white',
                           self.debug_surface, (0, 125))
            self.draw_text(f'self.total_time: {self.total_time}', self.debug_font,
'white', self.debug_surface,
                           (0, 150))
            self.draw_text(f'self.current_level_time: {self.current_level_time}',
self.debug_font, 'white',
                           self.debug_surface, (0, 175))
            self.draw_text(f'self.player_name: {self.player_name}', self.debug_font,
'white', self.debug_surface,
```

```
                            (0, 200))
            self.draw_text(f'self.menu_float: {self.menu_float}', self.debug_font,
'white', self.debug_surface,
                            (0, 225))
            self.draw_text(f'self.player.pos: {self.player.pos}', self.debug_font,
'white', self.debug_surface,
                            (0, 250))
            self.draw_text(f'self.player.rect(): {self.player.rect()}', self.debug_font,
'white', self.debug_surface,
                            (0, 275))
            self.draw_text(f'self.player.air_time: {self.player.air_time}',
self.debug_font, 'white',
                            self.debug_surface, (0, 300))
            self.draw_text(f'self.player.dashing: {self.player.dashing}',
self.debug_font, 'white', self.debug_surface,
                            (0, 325))
            self.draw_text(f'self.player.big_jump_cd: {self.player.big_jump_cd}',
self.debug_font, 'white',
                            self.debug_surface, (0, 350))
            self.draw_text(f'self.player.velocity: {self.player.velocity}',
self.debug_font, 'white',
                            self.debug_surface, (0, 375))

            self.draw_text(f'self.player: {self.player}', self.debug_font, 'white',
self.debug_surface, (0, 400))
            self.draw_text(f'self.show_pause_menu: {self.show_pause_menu}',
self.debug_font, 'white',
                            self.debug_surface, (0, 425))
            self.draw_text(f'self.button_selector_position:
{self.button_selector_position}', self.debug_font, 'white',
                            self.debug_surface, (0, 450))
            self.draw_text(f'self.end_panel_pos: {self.end_panel_pos}', self.debug_font,
'white', self.debug_surface,
                            (0, 475))
            self.draw_text(f'len(self.enemies): {len(self.enemies)}', self.debug_font,
'white', self.debug_surface,
                            (0, 500))
            self.draw_text(f'self.screenshake: {self.screenshake}', self.debug_font,
'white', self.debug_surface,
                            (0, 525))
            self.draw_text(f'self.scroll: {self.scroll}', self.debug_font, 'white',
self.debug_surface, (0, 550))
            self.draw_text(f'self.render_scroll: {self.render_scroll}', self.debug_font,
'white', self.debug_surface,
                            (0, 575))
            self.draw_text(f'self.movement: {self.movement}', self.debug_font, 'white',
self.debug_surface, (0, 600))
            self.draw_text(f'self.end_panel_pos: {self.end_panel_pos}', self.debug_font,
'white', self.debug_surface,
                            (0, 625))

            # self.display.blit(pygame.transform.scale(self.debug_surface,
self.display.get_size()), (0,0))
            self.debug_info_updated = True

    def EVERYTHING_render_update(self):
```

```
        self.scroll_update()
        self.projectile_render_update()
        self.spark_particle_render_update()
        self.cloud_leaf_render_update()
        self.entity_render_update()

        self.tilemap.render(self.display, offset=self.render_scroll)

        if self.in_main_menu:

            self.render_main_menu_panel()
            self.render_leaderboard()

        if not self.in_main_menu:
            self.displayUI_EVERYTHING()

        self.check_level_loading()

        # Assuming you're measuring delta time each frame with dt
        if self.show_debug_menu:
            self.update_debug_stuff()

    def display_CD_block(self):
        dash_CD = self.player.get_dash_CD()
        CD_fraction = dash_CD / 60

        big_jump_CD = self.player.get_big_jump_CD()
        big_jump_CD_fraction = big_jump_CD / self.player.big_jump_max_cd
        # print(CD_fraction)
        # print(big_jump_CD)

        BLOCK_SIZE = 32

        JUMP_CD_BLOCK = pygame.Surface((BLOCK_SIZE, BLOCK_SIZE * big_jump_CD_fraction))
        pygame.draw.rect(JUMP_CD_BLOCK, (0, 0, 0), pygame.Rect(0, 0, BLOCK_SIZE,
BLOCK_SIZE), )

        DASH_CD_block = pygame.Surface((BLOCK_SIZE, BLOCK_SIZE * CD_fraction))
        pygame.draw.rect(DASH_CD_block, (0, 0, 0), pygame.Rect(0, 0, BLOCK_SIZE,
BLOCK_SIZE), )

        DASH_CD_block.fill((255, 255, 255))
        JUMP_CD_BLOCK.fill((255, 255, 255))

        DASH_CD_block.set_alpha(255)
        JUMP_CD_BLOCK.set_alpha(255)
        self.display.blit(self.keyboard_x, (40, self.display_HEIGHT - 40))

        self.display.blit(self.assets['e_key_diagram'],
                          (80 + self.assets['e_key_diagram'].get_width(),
self.display_HEIGHT - 40))

        dash_UI_icon = self.raw_assets['dash_UI_icon']
        jump_UI_icon = self.raw_assets['jump_UI_icon']
```

```
        JUMP_CD_BLOCK.blit(jump_UI_icon, (0, 0), special_flags=pygame.BLEND_RGBA_MULT)

        DASH_CD_block.blit(dash_UI_icon, (0, 0), special_flags=pygame.BLEND_RGBA_MULT)

        self.display.blit(DASH_CD_block, (35, self.display_HEIGHT - 80))
        self.display.blit(JUMP_CD_BLOCK, (75 + self.assets['e_key_diagram'].get_width(),
self.display_HEIGHT - 80))

    def reset_movement(self):
        self.movement[0] = False

        self.movement[1] = False

    def display_welcome(self):
        arrow_keys = self.assets['arrow_keys_diagram']
        arrow_keys = pygame.transform.scale(arrow_keys, (arrow_keys.get_width() * 2,
arrow_keys.get_height() * 2))

        self.display.blit(arrow_keys,
                          (self.display.get_width() // 2 - arrow_keys.get_width() // 2,
60 + self.menu_float))

        self.draw_text('USE ARROW KEYS TO MOVE', pygame.font.Font('minecraft.otf', 20),
'black', self.display,
                       (52, 62 + self.menu_float))
        self.draw_text('USE ARROW KEYS TO MOVE', pygame.font.Font('minecraft.otf', 20),
'white', self.display,
                       (50, 60 + self.menu_float))

        self.display.blit(self.keyboard_x,
                          (self.display.get_width() / 4 * 3 - arrow_keys.get_width() //
2, 60 + self.menu_float))
        self.draw_text('TO DASH', pygame.font.Font('minecraft.otf', 20), 'black',
self.display,
                       (self.display.get_width() / 4 * 3 + 2, 62 + self.menu_float))
        self.draw_text('TO DASH', pygame.font.Font('minecraft.otf', 20), 'white',
self.display,
                       (self.display.get_width() / 4 * 3, 60 + self.menu_float))

        # e to super dash

        self.display.blit(self.assets['e_key_diagram'],
                          (self.display.get_width() / 4 * 3 - arrow_keys.get_width() //
2, 100 + self.menu_float))
        self.draw_text('TO SUPER JUMP', pygame.font.Font('minecraft.otf', 20), 'black',
self.display,
                       (self.display.get_width() / 4 * 3 + 2, 102 + self.menu_float))
        self.draw_text('TO SUPER JUMP', pygame.font.Font('minecraft.otf', 20), 'white',
self.display,
                       (self.display.get_width() / 4 * 3, 100 + self.menu_float))

    def main_game(self):
        self.GRANDTOTAL_COUNTER = 0
        self.end_panel_pos = -self.assets['game_end_panel'].get_height()

        while True:
```

```
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_LEFT:
                        self.movement[0] = True
                    if event.key == pygame.K_RIGHT:
                        self.movement[1] = True
                    if event.key == pygame.K_UP:
                        if self.player.jump():
                            self.sfx['jump'].play()
                    if event.key == pygame.K_x:
                        self.player.dash()
                    if event.key == pygame.K_ESCAPE:
                        # print('menu triggered')
                        self.show_pause_menu = not self.show_pause_menu
                        self.ingame_menu()
                    if event.key == pygame.K_z:
                        if self.player.big_jump():
                            self.sfx['jump'].play()
                    if event.key == pygame.K_TAB:
                        print('bruh')
                    if event.key == pygame.K_RETURN:
                        if self.game_end:
                            self.entire_game_reset()
                            self.main_menu()



                    if event.key in self.level_keys:
                        self.level = self.level_keys[event.key]
                        self.load_level(self.level)
                        self.reset_movement()
                        self.reset_player_status()
                        self.reset_current_level_timer()
                        self.main_game()


                if event.type == pygame.KEYUP:
                    if event.key == pygame.K_LEFT:
                        self.movement[0] = False
                    if event.key == pygame.K_RIGHT:
                        self.movement[1] = False

            self.display.blit(self.assets['background'], (0, 0))
            self.EVERYTHING_render_update()

            self.screenshake = max(0, self.screenshake - 1)
            self.screenshake_offset = (random.random() * self.screenshake -
self.screenshake / 2,
                                       random.random() * self.screenshake -
self.screenshake / 2)
            self.screen.blit(pygame.transform.scale(self.display,
self.screen.get_size()), self.screenshake_offset)
            if self.show_debug_menu:
```

```
                self.screen.blit(self.debug_surface, (0, 0))
            pygame.display.update()

            self.clock.tick(RUNNING_FPS)


    def refresh_menu_float(self):
        self.x += 0.05
        self.x %= math.pi * 2
        self.menu_float = 7 * math.sin(self.x)

    def reset_player_status(self):
        self.show_pause_menu = False
        # reset CD
        self.player.big_jump_cd = 0
        self.HP = 3

    def global_stats_reset(self):
        self.player_got_hit_count = 0
        self.restart_count = 0
        self.total_time = 0

    def entire_game_reset(self):
        self.reset_player_status()
        self.global_stats_reset()

        self.current_level_time = 0
        self.level = 1
        # print(self.level)
        self.reset_movement()

    def render_pause_menu(self):
        speed = 30
        if self.show_pause_menu:
            progress = ease_out_quad((self.pause_menu_target_position -
self.pause_menu_position) / self.assets[
                'in_game_pause_menu_status'].get_width())
            self.pause_menu_position += speed * progress
            # Cap the position to the target if it overshoots due to the easing function
            if self.pause_menu_position > self.pause_menu_target_position:
                self.pause_menu_position = self.pause_menu_target_position
        else:
            progress = ease_out_quad(
                (self.pause_menu_position +
self.assets['in_game_pause_menu_status'].get_width()) / self.assets[
                    'in_game_pause_menu_status'].get_width())
            self.pause_menu_position -= speed * progress
            # Cap the position to negative width of the menu if it overshoots due to the
easing function
            if self.pause_menu_position <
-self.assets['in_game_pause_menu_status'].get_width():
                self.pause_menu_position =
-self.assets['in_game_pause_menu_status'].get_width()
        # print(self.pause_menu_position)
        status_menu = self.assets['in_game_pause_menu_status'].copy()
        # also add a render of player sprite, on the player box, the player box is a
```

```
82x94 box within the status_menu, with its left top corner at (61,81), onto the
status_menu surface

        self.playermodel.animate(pygame.time.get_ticks())
        self.playermodel.draw(status_menu)

        self.draw_text(self.player_name, (font := pygame.font.Font('minecraft.otf',
20)), 'black', status_menu,
                       (108 - font.size(self.player_name)[0] // 2 + 1, 20 -
font.size(self.player_name)[1] // 2 + 1))
        self.draw_text(self.player_name, (font := pygame.font.Font('minecraft.otf',
20)), 'white', status_menu,
                       (108 - font.size(self.player_name)[0] // 2, 20 -
font.size(self.player_name)[1] // 2))
        # Define the top-left coordinates of the stats panel
        stats_panel_x, stats_panel_y = 383, 88

        # Draw the text "Player got hit:" and the associated count
        self.draw_text(f'Player got hit: {self.player_got_hit_count}',
pygame.font.Font('minecraft.otf', 10), '#6b5153',
                       status_menu, (stats_panel_x + 6, stats_panel_y + 12))

        # Draw arrow image a bit below the text
        status_menu.blit(self.raw_assets['arrows'], (stats_panel_x + 6, stats_panel_y +
32))

        # Draw the text "Restart count:" and the associated count, below the arrow image
        self.draw_text(f'Restart count: {self.restart_count}',
pygame.font.Font('minecraft.otf', 10), '#6b5153',
                       status_menu, (stats_panel_x + 6, stats_panel_y + 72))

        status_menu.blit(self.raw_assets['skeleton'], (stats_panel_x + 6, stats_panel_y
+ 92))

        # draw the button_selector onto the buttons according the variable
self.button_selector

        self.button_selector_target_position = 69 + self.button_selector * 51

        difference = self.button_selector_target_position -
self.button_selector_position

        # Normalize the difference for the easing function
        normalized_difference = abs(difference) / 51.0
        if normalized_difference > 2:
            self.button_selector_speed = 20
        else:
            self.button_selector_speed = 7

        # If the normalized difference is out of bounds, correct it
        if normalized_difference > 1:
            normalized_difference = 1
        elif normalized_difference < 0:
            normalized_difference = 0

        # Calculate progress using the easing function
```

```
        progress = ease_out_quad(normalized_difference)

        if difference > 0:
            self.button_selector_position += self.button_selector_speed * progress
            if self.button_selector_position > self.button_selector_target_position:
                self.button_selector_position = self.button_selector_target_position
        else:
            self.button_selector_position -= self.button_selector_speed * progress
            if self.button_selector_position < self.button_selector_target_position:
                self.button_selector_position = self.button_selector_target_position

        # print(f'self.button_selector_position{self.button_selector_position}')

        button_selector_loc = (173, self.button_selector_position)
        status_menu.blit(self.assets['button_selector'], button_selector_loc)

        # Draw skeleton image a bit below the second text

        # self.draw_text(self.player_name, pygame.font.Font('minecraft.otf', 20),
'black', status_menu, (68+102//2-self.font.size(self.player_name)[0]//2,
47+10-self.font.size(self.player_name)[1]//2))
        # self.draw_text(self.player_name, pygame.font.Font('minecraft.otf', 20),
'white', status_menu, (67+102//2-self.font.size(self.player_name)[0]//2,
46+10-self.font.size(self.player_name)[1]//2))

        self.display.blit(status_menu, (
            self.pause_menu_position,
            self.display_HEIGHT / 2 -
self.assets['in_game_pause_menu_status'].get_height() / 2))

    def ingame_menu(self):

        self.screenshot_taken = False
        self.button_selector = 0
        self.menu_buttons = ['Resume', 'Restart', 'Options', 'Main Menu']

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

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        # print('game resumed')
                        self.show_pause_menu = not self.show_pause_menu
                        self.reset_movement()
                        self.main_game()
                    if event.key == pygame.K_DOWN:
                        self.sfx['jump'].play()
                        self.button_selector += 1
                        self.button_selector %= len(self.menu_buttons)
                    if event.key == pygame.K_UP:
                        self.sfx['jump'].play()
                        self.button_selector -= 1
                        if self.button_selector < 0:
```

```
                            self.button_selector = len(self.menu_buttons) - 1
                        self.button_selector %= len(self.menu_buttons)

                    if event.key == pygame.K_RETURN:
                        if self.menu_buttons[self.button_selector] == "Resume":
                            self.show_pause_menu = False
                            self.reset_movement()
                            self.main_game()
                        elif self.menu_buttons[self.button_selector] == "Restart":

                            self.restart_level()

                        elif self.menu_buttons[self.button_selector] == "Options":
                            self.show_debug_menu = not self.show_debug_menu
                            print('debug menu show has been set to',
self.show_debug_menu)
                            pass
                        elif self.menu_buttons[self.button_selector] == "Main Menu":
                            self.entire_game_reset()

                            self.main_menu()

            if not self.screenshot_taken:
                if self.pause_menu_position < -490:
                    pygame.image.save(self.display, 'screenshot.png')
                OriImage = Image.open('screenshot.png')
                screenshot_no_blur = pygame.image.load('screenshot.png')
                blurImage = OriImage.filter(ImageFilter.GaussianBlur(2.5))
                # make blurimage darker
                enhancer = ImageEnhance.Brightness(blurImage)
                blurImage = enhancer.enhance(0.7)
                blurImage.save('simBlurImage.png')

                blurImageBG = pygame.image.load('simBlurImage.png')
                self.screenshot_taken = True

            # self.display.blit(blurImageBG, [0, 0])

            self.display.blit(screenshot_no_blur, (0, 0))

            self.render_pause_menu()

            self.update_debug_stuff()
            # print(f'{self.button_selector} is selected')

            # self.mouse_pos = pygame.mouse.get_pos()
            # self.mouse_x = self.mouse_pos[0]//2
            # self.mouse_y = self.mouse_pos[1]//2

            self.refresh_menu_float()


            self.screen.blit(pygame.transform.scale(self.display,
self.screen.get_size()), self.screenshake_offset)
            if self.show_debug_menu:
                self.screen.blit(self.debug_surface, (0, 0))
```

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

    def render_leaderboard(self):

        leaderboard = self.assets['leaderboard'].copy()
        if self.menu_buttons[self.button_selector] == "Leaderboard":
            target = self.leaderboard_target_position
        else:
            target = self.display_WIDTH - 20

        difference = target - self.leaderboard_position
        normalized_difference = abs(difference) / leaderboard.get_width()

        # If the normalized difference is out of bounds, correct it
        if normalized_difference > 1:
            normalized_difference = 1
        elif normalized_difference < 0:
            normalized_difference = 0

        # Calculate progress using the easing function
        progress = ease_out_quad(normalized_difference)

        if difference > 0:
            self.leaderboard_position += 13 * progress
            if self.leaderboard_position > target:
                self.leaderboard_position = target
        else:
            self.leaderboard_position -= 13 * progress
            if self.leaderboard_position < target:
                self.leaderboard_position = target

        with open("data/player_records.json", "r") as f:
            data = json.load(f)

        # 2. Select the Best 9 Entries
        sorted_data = sorted(data, key=lambda x: x["total time"])[:9]

        # Define a starting position for the entries
        start_x = 35  # Adjust as needed
        start_y = 80  # Adjust as needed

        # Define a gap between entries for spacing
        gap = 16  # Adjust as needed

        # Use a font to render the entries
        font = pygame.font.Font('minecraft.otf', 10)

        # 3. Display the Entries on the Leaderboard
        for index, entry in enumerate(sorted_data, 1):  # Start index from 1
            total_time_seconds = entry['total time']
            # format the time into minutes and seconds
            if total_time_seconds >= 60:
                if total_time_seconds % 60 < 10:
                    total_time_formatted_seconds = f'0{int(total_time_seconds % 60)}'
                else:
```

```
                    total_time_formatted_seconds = int(total_time_seconds % 60)
                fraction_part = round(total_time_seconds - int(total_time_seconds),1)
                fraction_part = str(fraction_part)[2:]
                total_time_formatted = f'{int(total_time_seconds //
60)}:{total_time_formatted_seconds}.{fraction_part}'
            else:
                total_time_formatted = total_time_seconds

            text = f"{index}. {entry['player name']} {total_time_formatted}"
            text_surface = font.render(text, True, (255, 255, 255))  # Render in white
color, adjust as needed
            text_rect = text_surface.get_rect(topleft=(start_x, start_y + index * gap))
            # add another black shadow layer underneath the text
            text_surface_shadow = font.render(text, True, (0, 0, 0))
            text_rect_shadow = text_surface_shadow.get_rect(topleft=(start_x + 1,
start_y + index * gap + 1))
            leaderboard.blit(text_surface_shadow, text_rect_shadow)

            leaderboard.blit(text_surface, text_rect)


        # BLIT IMAGE of LEADERBOARD , on the right-hand side with updated position
        self.display.blit(leaderboard, (self.leaderboard_position, self.display_HEIGHT
// 20 * 1 + self.menu_float))

    def render_main_menu_panel(self):

        self.main_menu_panel = self.assets['main_menu_buttons'].copy()
        self.main_menu_button_selector_target_position = 16 + self.button_selector * 45


        difference = self.main_menu_button_selector_target_position -
self.main_menu_button_selector_position
        normalized_difference = abs(difference) / 45

        if normalized_difference > 2:
            self.button_selector_speed = 20
        else:
            self.button_selector_speed = 7

        # If the normalized difference is out of bounds, correct it
        if normalized_difference > 1:
            normalized_difference = 1
        elif normalized_difference < 0:
            normalized_difference = 0

        # Calculate progress using the easing function
        progress = ease_out_quad(normalized_difference)

        if difference > 0:
            self.main_menu_button_selector_position += self.button_selector_speed *
progress
            if self.main_menu_button_selector_position >
self.main_menu_button_selector_target_position:
                self.main_menu_button_selector_position =
self.main_menu_button_selector_target_position
```

```
        else:
            self.main_menu_button_selector_position -= self.button_selector_speed *
progress
            if self.main_menu_button_selector_position <
self.main_menu_button_selector_target_position:
                self.main_menu_button_selector_position =
self.main_menu_button_selector_target_position

        self.main_menu_panel.blit(self.assets['main_menu_button_selector'],
                                  (74, self.main_menu_button_selector_position))
        self.display.blit(self.main_menu_panel,
                          (self.display_WIDTH // 20 * 1, self.display_HEIGHT // 20 * 7 +
self.menu_float))

    def main_menu(self):

        self.game_finished = False
        self.in_main_menu = True
        self.textflash = 0

        self.leaderboard_position = self.assets['leaderboard'].get_width() +
self.display_WIDTH
        self.leaderboard_target_position = self.display_WIDTH // 20 * 15 -
self.assets['leaderboard'].get_width() // 2

        self.load_level('main_menu_map')
        # Load background image (Make sure you have a `background.jpg` in your assets)
        background = pygame.image.load('data/images/background.png').convert()
        background = pygame.transform.scale(background, (self.display_WIDTH,
self.display_HEIGHT))

        # Define the buttons
        self.menu_buttons = ["Start", "Options", "Leaderboard", "Quit"]
        self.button_selector = 0

        self.main_menu_button_selector_position = 16
        self.main_menu_button_selector_target_position = 16

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

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_DOWN:
                        self.sfx['jump'].play()
                        self.button_selector += 1
                        self.button_selector %= len(self.menu_buttons)
                    if event.key == pygame.K_UP:
                        self.sfx['jump'].play()
                        self.button_selector -= 1
                        if self.button_selector < 0:
                            self.button_selector = len(self.menu_buttons) - 1
                        self.button_selector %= len(self.menu_buttons)
```

```
                    if event.key == pygame.K_RETURN:
                        if self.menu_buttons[self.button_selector] == "Start":
                            self.load_level(self.level)
                            self.in_main_menu = False
                            self.reset_movement()
                            self.reset_player_status()
                            self.show_status_menu = False
                            self.main_game()

                            # print(self.level)

                        elif self.menu_buttons[self.button_selector] == "Options":
                            self.show_debug_menu = not self.show_debug_menu
                            print('debug menu show has been set to',
self.show_debug_menu)
                        elif self.menu_buttons[self.button_selector] == "Leaderboard":
                            # Implement the Leaderboard functionality here
                            pass
                        elif self.menu_buttons[self.button_selector] == "Quit":

                            pygame.quit()
                            sys.exit()

            # Rendering
            self.display.blit(self.assets['background'], (0, 0))

            self.refresh_menu_float()
            self.display.blit(self.assets['game_logo'],
                              (self.display_WIDTH // 20 * 1, self.display_HEIGHT // 20 *
1 + self.menu_float))

            self.EVERYTHING_render_update()


            self.movement[1] = 0.8

            self.draw_text('Made By Dyrox2333', pygame.font.Font('minecraft.otf', 10),
'black', self.display, (
                self.display_WIDTH // 20 * 19.5 - self.font.size('Made By Dyrox2333')[0]
// 2 + 1,
                self.display_HEIGHT // 20 * 19.5 - self.font.size('Made By
Dyrox2333')[1] // 2 + 1))
            self.draw_text('Made By Dyrox2333', pygame.font.Font('minecraft.otf', 10),
'white', self.display, (
                self.display_WIDTH // 20 * 19.5 - self.font.size('Made By Dyrox2333')[0]
// 2,
                self.display_HEIGHT // 20 * 19.5 - self.font.size('Made By
Dyrox2333')[1] // 2))

            self.screen.blit(pygame.transform.scale(self.display,
self.screen.get_size()), (0, 0))

            if self.show_debug_menu:
                self.screen.blit(self.debug_surface, (0, 0))
            pygame.display.flip()
            self.clock.tick(RUNNING_FPS)
```

```
Game().main_menu()
```