OCR COMPUTER SCIENCE NEA: PLATFORMER GAME:

TABLE OF CONTENTS

1: ANALYSIS

1.1: PROBLEM IDENTIFICATION

A platforming game is a type of video game where the primary objective is to navigate a character through a series of levels or stages, jumping or running across platforms, avoiding obstacles, and defeating enemies to reach the end goal. The game typically features a side-scrolling perspective (i.e., the camera moves from left to right with the player), with the player character moving from left to right across the screen.

Some popular examples of platforming games include “Super Mario Bros” and “Sonic the Hedgehog”. Super Mario Bros (SMB) is one of the most popular single player video game franchises of all time. It was a huge commercial success, selling over 40 million copies worldwide and becoming the best-selling video game of all time at the time of its release. It also received critical acclaim for its inventive level design, colourful graphics, and memorable music. The game has since become a cultural icon and has spawned countless sequels and spin-offs, and it is widely considered to be one of the greatest video games of all time. My project will be an object-oriented game using the same format, recreating the mechanics of the original SMB but adding my own levels, graphics, and a few different features.

Platforming games are a simple easy to play game that is very simple to learn and get the hang of. It is therefore not limited by the age of the person playing it, and people of all ages can play and enjoy the game. My version will have a lower difficulty, to allow people who have never played a platforming game to be able to pick it up quickly and enjoy themselves as they play through the levels. However, I am maybe planning to add some form of difficulty or difference that gives more experienced players something to play the game for.

2D platformer games have seen a decrease in production and releases in recent times, as companies believe that many players have shifted their attention to other types of games, such as open-world RPGs, first-person shooters, and “battle-royale” games. Despite this, there are still many fans of 2D platformers:

- As of December 2022, New Super Mario Bros. U Deluxe has had over 14.75 million copies worldwide
- Away from Mario games, as of September 2021, "Celeste" has sold over 3 million copies worldwide across all platforms.
- The success of these games demonstrates that there is still a market for well-designed and innovative 2D platformers, even in a gaming landscape that is increasingly dominated by other genres.
- There is still a group of people who feel nostalgic over the simplicity of the games and still want more and more platformer games to be made. I will highlight more on who will benefit from such a game in my stakeholder’s section.

Although it's not everyone's first choice when looking for an object-oriented language, I am going to being using Python for this project, with massive help from the pygame module.

1.2: WHY USE A COMPUTATIONAL SOLUTION?

I am using a computational approach because it is a video game. There are many complex calculations happening all the time to determine angles and other aspects of the player's movement, and simply put, this would not be feasible in any other way.

1.2.1 COMPUTATIONAL METHODS UTILISED

The problem is suitable for solving by using the following computational methods:

- Decomposition: The task can be broken down using decomposition, as I will be using a modular approach, separating my program into multiple files, and using an OOP programming paradigm.
    - This allows me to work on separate classes and files at a time, allowing me to break down my program into much smaller components which I can then work on easier.
- Object-Oriented Programming (OOP): OOP is a programming paradigm that allows the creation of objects that encapsulate data and behaviour.
    - In my game, I need to create many different objects and entities, like characters, enemies, and blocks which I need to create my levels.
- Inheritance: Inheritance allows new objects to be created based on existing ones, inheriting their properties and behaviours.
    - In my game, this could be used to create a base enemy class which contains most of the base methods and attributes like animating the enemy that all enemy sprites would need, and then have specific enemy classes inherit from this, each with their own appearances, movement, and other features that make them unique.
- Polymorphism: Polymorphism allows objects to take on different forms or behaviours depending on their context.
    - In my game, the player sprite will be animated entirely differently depending on what state it is in (whether it is walking/staying still, looking left/right, jumping/grounded). Sometimes a series of sprites will be played in a look whereas sometimes only one singular sprite will be shown.
- Event-Driven Programming: Event-driven programming allows the game to respond to user input or other external events (like an enemy death, player death, level being completed), triggering actions or changes in the game's state.
    - For example, a player jumping could trigger a change in the character's position, animation, or velocity.
    - This will be partially achieved using a game loop, which manages the game's state and updates the objects and interactions in the game world. This loop runs continuously, updating the game's graphics, physics, and logic based on the player's input and the game's internal state.

1.3: STAKEHOLDERS

Normally, a person who enjoys challenges and has good hand-eye coordination would benefit from a new platforming game. Most platforming games now are made with the perspective that people would likely enjoy the fast-paced action and precision required to navigate through the various obstacles and enemies in each level. I want to make a game that is more accessible to newcomers, player who have a very limited gaming experience.

However, I feel as though the game will also have something to offer for more experienced gamers. Platforming games have a nostalgic appeal to them, as many classic platformers have been popular for decades and continue to be beloved by gamers of all ages. Therefore, I feel as though this game has something for any type of player, regardless of gaming ability and experience.

Overall, anyone who enjoys a fun and challenging gaming experience, with a mix of exploration, puzzle-solving, and action, would likely find a platforming game to be a rewarding and entertaining experience.

Therefore, my stakeholders for the project are going to be people who have either never played the game before or are very experienced players looking to play one of their favourite games again. I want to see their perspective on:

- Any features to add
- What to avoid in my game?
- Any other perspectives/feedback

My stakeholders for the project are as follows:

- Nigel Arun Jacob (school friend): Someone who has never played a platforming game and in general does not really play video games
- Harshal Jain (family member) Someone who has played every edition of the Super Mario Bros series.

1.4: INTERVIEWS

I have 2 stakeholders and I will be asking them the following questions to understand more about the requirements of my game. I will ask them the following questions as part of my research and hope to gather useful information for the design of the game:

- How often do you play games?
- How often do you play platforming games?
- What would you expect from a game/platforming game?
- Are there any features you would like me to include in my game?
- Are there any quality of life features you would like me to include?
- Do you think you would you play my game? Is there anything about it that appeals to you?
- Do you have anything else to add?

1.4.1: QUESTIONS FOR NIGEL

Me: How often do you play games?

Nigel: Not that often, I find more fun in solving maths questions and puzzles.

Me: Have you ever played a platforming game?

Nigel: I don't fully know what you mean by a platforming game.

Me: You control a character to jump through a bunch of obstacles until they reach the end of each level… ring a bell?

Nigel: Nope, I have no idea sorry.

Me: What would you expect from a game/platforming game then if I were to make one for someone with your gaming experience?

Nigel: Simple self-explanatory controls, like it would be possible for me to play and pick it up when I start playing as opposed to being something I would have to spend a long time, like, getting used to the game.

Me: Are there any features you would like me to include in my game?

Nigel: I don't really know, I guess don't make the user interface annoying and irritating to use. Maybe add some puzzle elements or some sort of critical thinking. Make sure you add a screen that clearly states the controls and rules for any beginners like myself.

Me: Do you think you would you play my game? Is there anything about it that appeals to you?

Nigel: From the sounds of it, if you intend on making it beginner friendly, then I don't see why I wouldn't give it a go. With the amount of stress, I get from A-Levels, trying out something new (that hopefully won't

be too difficult and stress me out even more) might be a relaxing break from everything, allowing me to destress and take my mind away from my education for a brief period of time.

Me: Do you have anything else to add?

Nigel: Good luck on making your game!

1.4.2: QUESTIONS FOR HARSHAL

Me: How often do you play games?

Harshal: I pretty much do 2 things, work in the morning and play games with my friends in the evening.

Me: Have you ever played a platforming game?

Harshal: Yes, many, although I haven't really in recent times.

Me: Would you still say there is a space for platforming games in this day and age?

Harshal: I mean I'd say that I kind of miss their simplicity. Having to explain new games to people is usually a tedious activity, they can get pretty complicated sometimes! I remember when I was a kid and I would just want a game to relax, Mario was always the game I could just sit down and play without any prep or required mood, just sit down, and run and jump.

Me: What would you expect from a game/platforming game?

Harshal: "Simple self-explanatory controls, little button delay. Focus on making sure the game runs smoothly, otherwise, what's the point? Maybe a little more difficulty than those games, but I don't really know.

Me: Are there any features you would like me to include in my game?

Harshal: Add a variety of enemies, I remembered being very annoyed at the constant sight of goombas {an enemy in SMB}, and a little variety would make the game more enjoyable.

Me: Do you think you would you play my game? Is there anything about it that appeals to you?

Harshal: It would certainly be a refreshing change of pace. Newer games are more hectic and require a lot more brain power, so trying out a much simpler game with much simpler controls sounds exciting. The nostalgic feeling of playing games similar to the games I played when I was younger is also very appealing and exciting.

Me: Do you have anything else to add?

Harshal: Not really, have fun making your game and I can't wait to play it!

1.4.3: SUMMARY OF FEEDBACK

Overall, the main points I got from these interviews were:

- My chosen stakeholders believe that the game would be suitable for them and their needs.
- Make a clean and easy to use UI
- Make processing speed a priority (if there is any lag or slowness while running the program, then I need to eliminate this.)
- Don't over complicate the controls, keep them simple
- Add a section in the game menus which clearly states the controls and rules of the game for any beginners
- Add a little bit of problem solving or some sort of thinking (not just mindless running and jumping) to offer the game more of its own identity.

1.5: RESEARCH

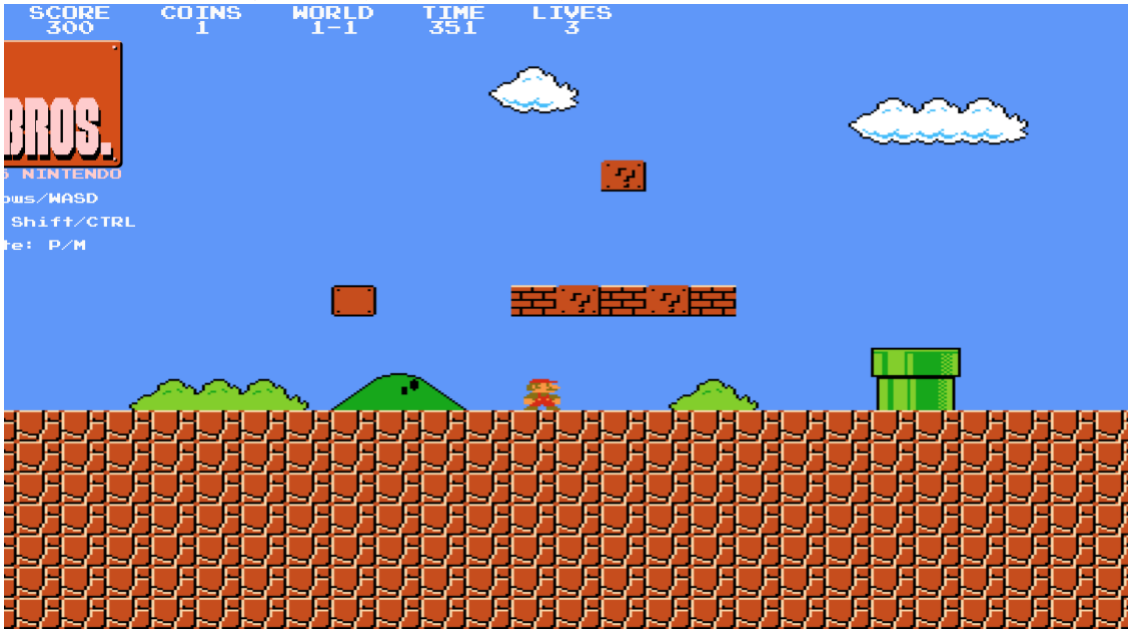
Listed below are the 4 different solutions I will research, with clear justification of what features of their version of the game is relevant to my game.

Solution 1: Super Mario Bros on supermariobros.io website: https://supermariobros.io/

Features I was drawn by:

- It does not fully recreate the original games and their levels system, there's some originality
- The features (jumping, items, enemies) are very accurate and close to the original games
- The game starts off almost instantly as you load the game, there's very little delay.

Features for criticism:

- There's quite a lot of button delay, it's hard to control the main character
- The game runs very slowly, certain movements and jumps are very hard to achieve.
- The sprites and sounds are all taken from the SMB games, which means there is potential copyright infringement if I do the same thing.
- When the screen size is changed, the sprites all get warped:
  - 

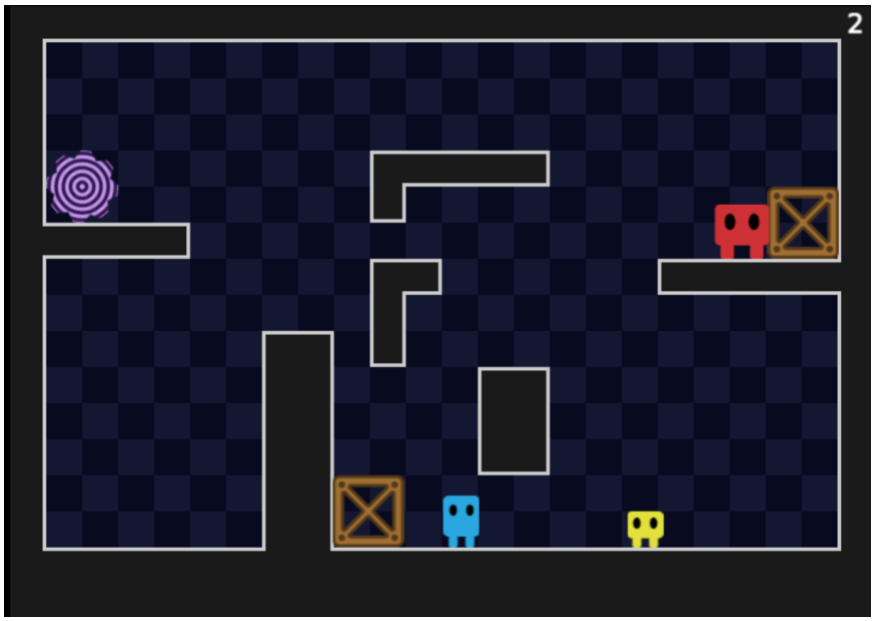
  - The sprites here are slightly squashed.

Solution 2: Big Tall Small: Different style of platforming game: https://www.crazygames.com/game/big-tall-small

Features I was drawn by:

- This game focusses more on a puzzle solving aspect rather than running and jumping.
- The animations are all very smooth, there's no delays or lag.
- The levels are simple and build in complexity over time.
- The UI is very simple and clean.

Features for criticism:

- It's possible to be put in game states where you have trapped yourself and have no option but to reset the level.
  - 

  - Here I am meant to push the block by the red character to the left to help the others reach higher ground, but I have moved the block too far to the right and now I have no choice but to reset the level.
  - Although this maybe fits with the puzzle solving aspect, as care must be given when trying to complete the level, I don't want this in my game, as this ruins the fast-paced nature of a platformer game.

Solution 3: Celeste Classic: https://mattmakesgames.itch.io/celesteclassic

Features I was drawn by:

- I like the simplistic pixel art style of the sprites; I think it will help give that nostalgic feeling that I want for my game.
- I like the varied controls, there is not just running and jumping but also wall jumping and charging.

Features for criticism:

- The game is probably too difficult for someone who's just picked it up, which is not what I would want for my game. I wouldn't consider myself bad at platforming games in general, but I spent 5 minutes and I couldn't get past the first level.
- There is no explanation of the controls anywhere. Considering they are relatively complicated and specific to this game, I would have assumed that there would be some explanation, but it kind of left me having to figure them out myself, which was very confusing and time consuming.

SUMMARY OF RESEARCH

I want to try and implement into my design the following features:

- Not fully recreating another game like SMB; having an original game.
- Smooth animations with no lag or delays.
- The game starting off almost instantly as you load the game, there's very little delay (not a huge, long title screen you must wait for every time)
- Using a simplistic art style for my sprites, to try and get a nostalgic feeling for the game to be more appealing to more experienced gamers.
- Also focussing on a puzzle type aspect, not fully just being mindless running and jumping.

I want to try and avoid the following features:

- Any button delay, or just generally the game lagging and running slowly, impeding the player's experience.
- Using copyrighted sprites and sounds.
- Not warping the sprite sizes and shapes when the screen size is changed.
- Not to be put in a state where the player hasn't lost a life but is still forced to restart the level as they have trapped themselves in a puzzle
    - Therefore, although I want some puzzle aspects, I must find a balance so that I do not lose the quick nature of the running and jumping that the platformer game is meant to bring.
- Making sure the controls are simple and clearly stated in the game.
- Making sure the game is not too difficult for a newcomer or beginner.

1.6: LIMITATIONS OF SOLUTION

As I am working on this by myself and have only 6 months to document and create my solution, a process I must do alongside my A Levels, there are a few things I will have to cut back on and omit from my solution due to time constraints.:

- I do not believe I have the time to realistically learn new a programming language that may be more suited to making a game, like Lua, whilst designing and then coding up the solution.
    - I do have knowledge of python, and a little bit of experience using pygame, and therefore, to ensure that I meet my deadline it is maybe better if I stick to python.
    - This may mean that I must cut down on certain features, or compromise slightly, as python itself is not known for its user interface abilities and is only really used for calculation purposes.
- While I have not planned out what I will be looking to achieve, I can say that features like multiplayer, or having over 60 levels (the typical amount for a SMB game) will most likely not be achievable, due to time limitations.
- I do not have the resources to create my own sprites, images, or music. To avoid copyright, I will be using copyright free resources created by other people, but they will not have been designed for my game and they will be used in other games, so this removes some of the originality that my game could have had with more funding, resources, and time.
- As this will be running on python on a computer, this game is not able to be played on any other device like a game console or a handheld device, and instead can only be played on a computer that runs python and has the pygame module installed.

1.7: REQUIREMENTS

HARDWARE

- A computer capable of running the software
  - The application should not require anything out of the ordinary when it comes to specs. When researching this online, it seems as though almost any computer that can run python should be able to run pygame. People have claimed that many games have worked on their "Raspberry Pi zero", (1GHz single-core CPU, 512MB RAM), so any computer should be fine.
- Standard peripherals (keyboard and mouse)
  - It is not expected that any other peripherals will be needed other than the standard keyboard and mouse, which can be assumed that the user will have, to let the user control the menu screen, and to control the player.

SOFTWARE

- Python
  - As the program will be written in Python, any operating system with the python installed will be able to run the application (macOS, Windows, and Linux being the most common operating systems used).
- PyCharm IDE
  - For development, the PyCharm IDE will be used primarily due to my familiarity and positive experience with its development features (such as auto-complete instructions, auto-import libraries, and ease with things like renaming variables and updating these changes throughout the program.).
- Pygame Library
  - This will be the main library and the basis of the entire program. The program depends on it to run for handling all sprites, and all forms of graphics and interactions between entities like collisions detection and movement.

1.8: SUCCESS CRITERIA

This is the success criteria I have set out for myself to try and achieve during the project.

Success Criteria	Evidence	Justification
Creating a program window	A video/Screenshots of the main file being run and the program window appearing.	We need a window for all the graphics to be displayed on.
Play button on the title screen that plays the first level	Screenshots/Video of the title screen being up, and the play button being clicked leading to the loading of the first level.	To ensure a simple title screen with nothing unnecessarily flashy that takes too long to wait for. To make sure the rules and controls of the game are easily accessible to the player.
Rules menu that is accessible by a button on the title screen and allows play to easily scroll through the controls and rules of the game	Screenshots/A video of the rules button being clicked leading to a rules page which displays the information and allows you to go back to the main menu.	
Base entity parent class	Screenshot of the ground and block entity appearing correctly with the correct sprite image, position, and size.	All entities share many similar attributes and methods (position, height, and width, creating a sprite object from an image file), so they could all inherit this from the same class and remove redundant code.
A way to easily define sprite sheets		As many different spritesheets will need to be imported so I want it to be a quick process.
Ground/Block entities defined (inherited from base entity class)		Blocks and Ground entities are very basic but obviously needed to make the platforms the player runs and jumps across. They are also created here to test the previous two criteria.
Create a map based on a list of strings (almost a 2D array/table like structure)	Screenshots of the blueprint of the map (lists of strings) and the exact creation of the blueprint on the game window.	To make it easy to create levels in the future if you use letters/symbols to represent different entities in a grid like structure (as opposed to individually assigning all sprites (x, y) coordinates) (will make more sense later in “Design”)
Create player sprite	Screenshot of the player sprite appearing where it has been coded to appear	To ensure that entities of different sizes and that use different sprite sheets still load and run as expected.

Add moving functionality to the player (left and right)	Video of the player moving left and right when the appropriate keys are pressed.	The player needs to be able to move left and right in a platforming game.
Create moving camera effect	Video of the "camera" moving, keeping the player in the centre of the screen as they move.	So that the player does not go off the screen.
Animate moving functionality	Video of the player showing a walking animation when moving left or right.	To improve the quality of the visuals of the game.
Add horizontal collisions	Video of the player walking into a block and not walking through it.	Proper collisions required to create any proper form of movement like jumping.
Add vertical collisions	Video of the player moving up/down into a block and not moving through it.	To allow gravity to work (So that the player doesn't just fall through the ground).
Apply gravity to the player's movement	Video of the player falling if in the air.	To allow jumping to work (the player needs to come down)
Add a jump function and animate it	Video of the player jumping up, and falling after a certain duration, after the space/up keys have been pressed.	It is a core function of the game.
Create parent enemy class	Video: Give the generic enemy sprite an appearance and see if it moves on its own, has the correct animations and the correct collision behaviour.	Enemies are required in the game to add a challenge. Apart from a few details, like appearance and potential special abilities, all enemy sprites should act similar, so a base enemy class would save a lot of time.
Allow player to jump on enemies to kill them	Video: see if player jumping and landing right above the enemy results in an enemy death (animation for this must also be created and tested). See if a collision of any other kind produces no result or change.	The player needs a basic way to remove enemies from the game.
Create death function for player	Video: Player collides with enemy (but not when the player is directly above the enemy), and a player death animation has played. The level restarts.	To add difficulty and challenge to the game.
Ability to switch between two maps seamlessly.	Video: Temporary function created to switch current map to two different maps using a key press, while both maps load perfectly with minimal errors.	As the game needs to last a long time and therefore needs multiple maps

Make warp pipes to change maps	Video: Player enters a pipe and exits (both with an animation) wherever the pipe is meant to lead to.	To create a way to move between maps that doesn't rely on a temporary function.
Create end goal for a level	Video: The player walks past the end goal, inciting an animation and loading the next level.	To create a bigger deal when the player finishes a level.
Import actual sprites to be used.	Video: The game functions exactly as it did before, just with entirely new sprites.	I may use copyrighted sprites during development (will be explained in design), but my eventual game will not have them.
Add all types of enemies, with different abilities	Video: The different enemy types are spawned in and animated correctly. Any special abilities are tested to ensure they work.	To make the game more varied, interesting, and potentially more difficult/easier
Add powerups for the player	Video: The player walks into the powerup, changes appearance and can now perform a different ability to potentially kill enemies in more ways.	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	Screenshot of the life counter in the top left. The life counter goes down when the player loses a life. Video: The loading screen shows up every time a new level is loaded (but not when a new map from the same level is loaded)	To add a consequence to dying, and a potential lose condition. The other things are purely to improve the visual aspect of the game.
Add a game over screen when the character reaches 0 lives.	Video: Lose the player's last life and see if the game over screen shows up. There should be a button to go back to the title screen.	To inform the player that they have lost and to give them the ability to restart the game.
Add a victory screen when the player reaches the end of the game.	Video: Reach the goal of the last level and be greeted with the victory screen. There should be a button that redirects the player to the start of the game.	To inform and congratulate the player that they have won.
Add a key that unlocks pipes	Video: The player cannot access a pipe to go forwards unless they have a key. Once the player walks into a key, the key appears in the overlay in the top left corner to inform the player that they have the key. After this, the	To add features to add exploration and problem solving to the game.

	player should be able to access a pipe.	
Add coins that could contribute to a score	Video: The player walks into a coin, and it is collected and disappears from the screen.	
Add blocks that disappear when certain goals have been achieved.	Video: Certain blocks disappear when all enemies have been killed, certain blocks disappear when all coins/keys are collected, and all enemies have been killed.	
Add Background Music	Phone Video (screen recording will not capture audio): Audio changes when level changes	To make the game more immersive and improve the quality of the game. To add memorability to it. To make it more obvious when an event has happened (pipe entered, level changed).
Add Sound Effects	Phone Video: Sound effects are played when certain events like collecting coins occur.	
Create all 8 Levels, with multiple maps	An entire playthrough of the whole game.	To make the final game.

In the post development phase, I will look to do two entire playthroughs of the game, ticking off every box in this following checklist:

Action to test	Expected Outcome	Functioning?	Evidence
(1) Menu	The buttons in the menu lead to the screens they're named to lead to.		
(2) Loading of maps from blueprint to game.	The first map screenshot will match the screenshot of the blueprint for the map		
(3) Horizontal Movement	The player moves left/right when the left/right keys are pressed		
(4) Horizontal Collisions 1	The player tries to move into and through a block walking left but is stopped when it collides with the block.		
(5) Horizontal Collisions 2	The player tries to move into and through a block walking right but is stopped when it collides with the block.		
(6) Vertical Collisions: 1	The player tries to jump up into and through a block but is stopped when it collides with the block. The jump is cancelled and the player lands on the ground.		
(7) Vertical Collisions: 2	The player lands on the ground but is stopped when it collides with the block.		
(8) Jumps	The player jumps when the up arrow/space bar is pressed. The appropriate sound is played.		

(9) Jump Movement	The jump velocity is quick, then slows down as the jump reaches its peak height.		
(10) Gravity: 1	The player falls when in the air due to a jump		
(11) Gravity: 2	The player falls when walking off a platform.		
(12) Gravity: 3	The player falls when hitting the ceiling.		
(13) Enemy Collisions 1: Enemy Death	The player defeats the enemy when jumping and landing directly above the enemy, causing an enemy death, and playing the enemy death animation		
(14) Enemy Death 2	If the player's special abilities are used on the enemy, the enemy death animation is played.		
(15) Enemy Death Animation	The enemy should change sprite and disappear after a small period of time. The appropriate sound is played.		
(16) Enemy Collisions 2: Player Death	If the player collides with an enemy, the player death animation is run.		
(17) Player Death 2: Falling off the map	If the player falls into a huge ditch, the player death animation should run.		
(18) Player Death 3	If the enemy's special abilities are used on the enemy, the player death animation is played.		
(19) Player Death Animation	The player changes into its death sprite image, rises for a small bit of time before moving downwards vertically until it falls off the map. The appropriate sound is played. Every other sprite should be paused (e.g., enemies do not move). The level is then restarted.		
(20) Restarting of level due to player death	The correct level loading screen is shown. The first map of the level is loaded.		
(21) Loading Level Screen	The level number is displayed after the word level, and the lives counter is displayed below that. The number of lives is updated immediately if a player has just lost a life.		
(22) Coin Collection	Coins disappear upon contact with the player and play a sound when collected.		
(23) Key Collection	Keys disappear upon contact with the player and play a sound when collected. The key appears in the top left of the screen. This key image disappears when a pipe has been unlocked		
(24) Pipe Entry 1	Pressing an arrow key next to a pipe in the opposite direction the pipe is facing plays		

	the pipe entry animation. The appropriate map linked to the pipe is loaded.		
(25) Pipe Entry 2	The pipe should only be able to be entered if the player has obtained the key in the map and unlocked the pipe.		
(26) Pipe Entry Animation	The player moves “inside” the pipe. Once the player is no longer visible the next map is loaded. All other sprites are frozen when this is happening.		
(27) Pipe Exit	If the player has entered a map through a pipe, then it should play a pipe exit animation when the map is loaded.		
(28) Pipe Exit Animation	The reverse of the pipe entry animation, the player should come out of the pipe until it is fully visible. All other sprites are frozen when this is happening.		
(29) The Enemy Disappearing Block	This should disappear when all enemies in the map have been defeated.		
(30) The Goal Disappearing Block	This should disappear when all enemies in the map have been defeated, and all coins/keys have been collected.		
(31) Landing on the goal block	The level finished function should run. The player runs off to the next level. All other sprites are frozen during this animation. The next level is loaded. The appropriate sound is played.		
(32) Victory Screen	When the player has completed the last level, a victory screen should be displayed.		
(33) Game Over Screen	When the player has run out of lives, a game over screen should be displayed.		
(34) Background Music	The correct background music is played for each level. It changes for each level.		
(35) Background Colour	The correct background colour is displayed for each level.		

2: DESIGN

2.1: DECOMPOSING THE PROBLEM

In the last section of the analysis stage, I Identified and justified the test data to be used during the iterative development of the solution and identified and justified any further data to be used in the post development phase. Using the success criteria as a list of objectives to achieve, the problem can be divided into 11 main stages of development, where each development focuses on a different aspect:

Success Criteria	Stage of Development
Creating a program window	Development 1: Creating a Program Window
Play button on the title screen that plays the first level	
Rules menu that is accessible by a button on the title screen and allows play to easily scroll through the controls and rules of the game	
Base entity parent class	Development 2: Creating Basic Entity Sprites
A way to easily define sprite sheets	
Ground/Block entities defined (inherited from base entity class)	
Create a map based on a list of strings (almost a 2D array/table like structure)	
Create player sprite	
Add moving functionality to the player (left and right)	Development 3: Horizontal Player Movement
Create moving camera effect	
Animate moving functionality	
Add horizontal collisions	
Add vertical collisions	Development 4: Vertical Player Movement
Apply gravity to the player's movement	
Add a jump function and animate it	
Create parent enemy class	Development 5: Enemy Entities and Death Functions
Allow player to jump on enemies to kill them	
Create death function for player	
Ability to switch between two maps seamlessly.	Development 6: Multiple Maps
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	Development 7: Combat Variety
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	Development 8: Winning and Losing
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	Development 9: Puzzle Like Aspects
Add coins that could contribute to a score	

Add blocks that disappear when certain goals have been achieved.	
Add Background Music	Development 10: Music and Sound
Add Sound Effects	
Create all 8 Levels, with multiple maps	Development 11: All the Levels

However, this will only be used as a guide as at the end of each development, there will be a review to adapt the success criteria if necessary and to set out a clearer path for the following development. I am aware that this is a very large project, however I am willing and ready to create it all and write it all out.

2.2: APPROACH TO TESTING

With each version created during development, a testing stage will follow to check that the feature or change implemented is correctly functioning. This will involve running the program and trying out the changes made to the application. I will lay out the expected outcomes of the program and make alterations to the program until the outcomes have been achieved.

In the last section of the analysis stage, I Identified and justified the test data to be used during the iterative development of the solution and identified and justified any further data to be used in the post development phase. I will be using this test data throughout the program as I test each version individually.

As it's a mainly graphic application, errors may usually be easily spotted from an unexpected graphical behaviour, e.g., the wrong sprite being loaded. In addition, the console will often give a useful indication of what caused the error, such as a syntax or null exception error. However, in areas where it is not obvious what went wrong, print statements will be used to give a console output throughout the program, to try and understand where the mistake is occurring and trace back to the root of the problem.

Screenshots or screen recordings will be used to show the results of testing, showing the resulting outputs whether error or expected behaviour - followed by the steps taken to remedy the issue.

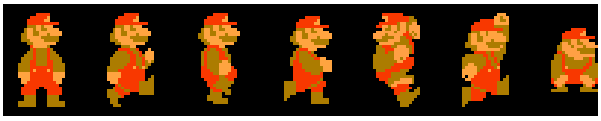
SPRITES USED DURING TESTING:

During most of development, I have used sprites from the Super Mario Bros game. Although these are not copyright free, I spent a lot of time looking for sprites that would suit my game and realised that I was spending too long on such a task, and that I could simply make it so that when I found the correct sprites and sprite sheets, I could easily import them into my code without too many problems.

I was having difficulties before finding good sprite sheets, as I wanted pixelated images of a character, preferably around 32 pixels tall and 16 pixels wide (although these are just rough estimates, and it wouldn't matter if they were much different), which also had a sprite for at least jumping (I didn't want to spend too much time making sprites). However, the sprites I was finding were more for modern games, where all the parts are separated, which helps the whole character be animated much better, but for the type of game I am trying to make, I could only import images and alternate between them, so these were not useful to me.

This an example of the type of sprite sheet I was constantly seeing:

Whereas I wanted a spritesheet that looked more like this:



2.3: DESCRIBING THE SOLUTION

GAME CLASS:

As this game requires a lot of attributes and methods that don't fit into any specific entity (i.e., a method loading a menu screen would not be stored in the player class), I need a class for general methods and updates, hence the game class. Almost every entity will have access to this class, as it will be passed by reference as one of their attributes.

GAME FUNCTIONS METHOD

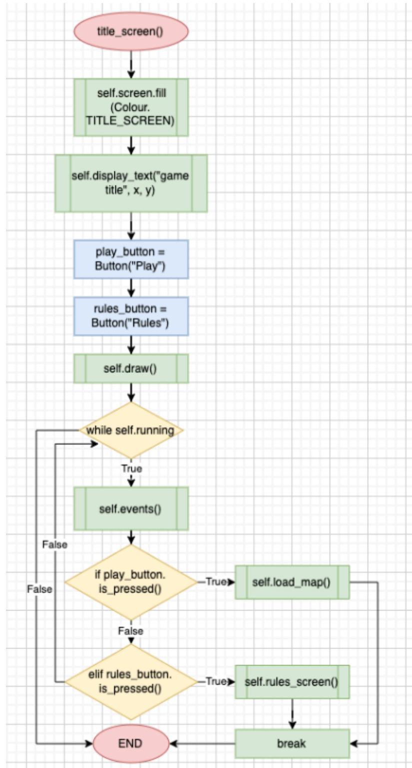
```
function game_functions():
      self.draw()
      self.update()
      self.events()
```

This is the most important function in the entire game. This is the function that is running at all times and helps run the entire game.

- Draw displays everything on the screen
- Events helps detect events like changing the size of the screen or trying to quit the game
- Update runs every single update function of every single entity in the game. This detects any changes to any entity and acts on it (e.g., the move key being detected and the player moving as a result of this). This helps create event driven programming. The more important entities will have their own update functions defined, and you will see a few later on

I would show how this works, but they use pygame specific functions which don't really translate well to general terms outside of what I have described.

DIFFERENT SCREENS



All screen loading functions are relatively simple. Text is displayed, buttons are displayed, and if the buttons are clicked then they lead to other menus. I'm not running "game functions" yet, as there are no sprites on the screen at this point, so the function would not work.

CREATING A MAP FROM A BLUEPRINT

This is maybe how a typical blueprint for a map would be like:

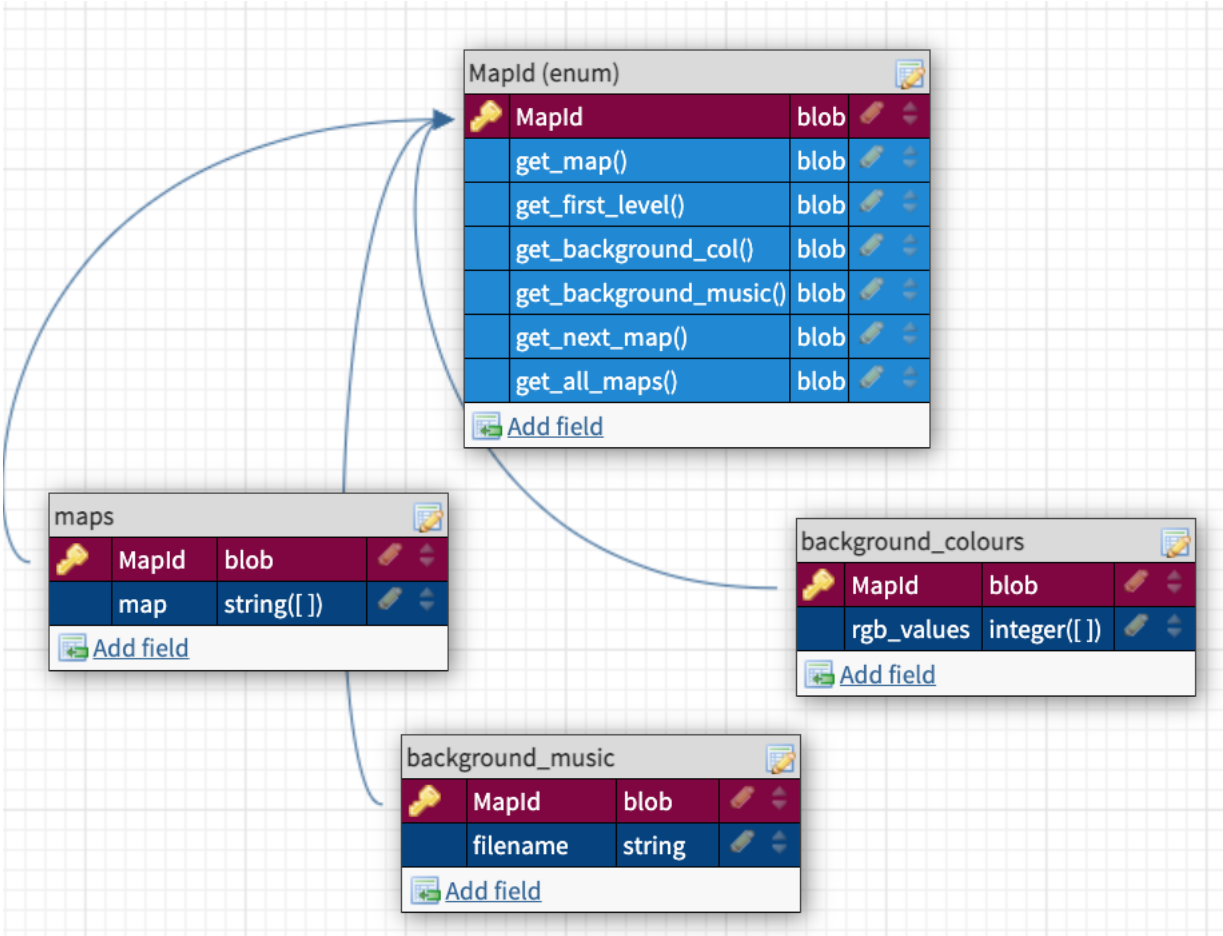
```
["BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
 "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I.....................B..................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I........................................BBBBBBBBBBB",
 ".........I.......P...........B.B.B......K.........BBBBBBBBBBB",
 ".........I.......#.............................h###BBBBBBBBBB",
 ".........I..........................s..........####BBBBBBBBBB",
 "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU",
 "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU"]
```

This may seem like a huge amalgamation of random symbols, but each one of those symbols represents an entity (a dot represents nothing). I will then use a function or method to go through this huge array of lists and add an entity for whatever the symbol represents:

```
function create_map():
for y, row, in enumerate(self.current_map_id.get_map()):
      for x, item, in enumerate(row):
            if item == "B":
                  Block.(self, x, y):
            if item == "P":
                  Player.(self, x, y)
            and so on...
```

As the code letters are all evenly spaced, this allows me to almost see what the level will look like using these blueprints, and once I have all the entities created, I just add them to this create map function, and I add one symbol to the blueprint and the entity is wherever it was intended to be. This will make level making at the end very convenient for me.

MAPS



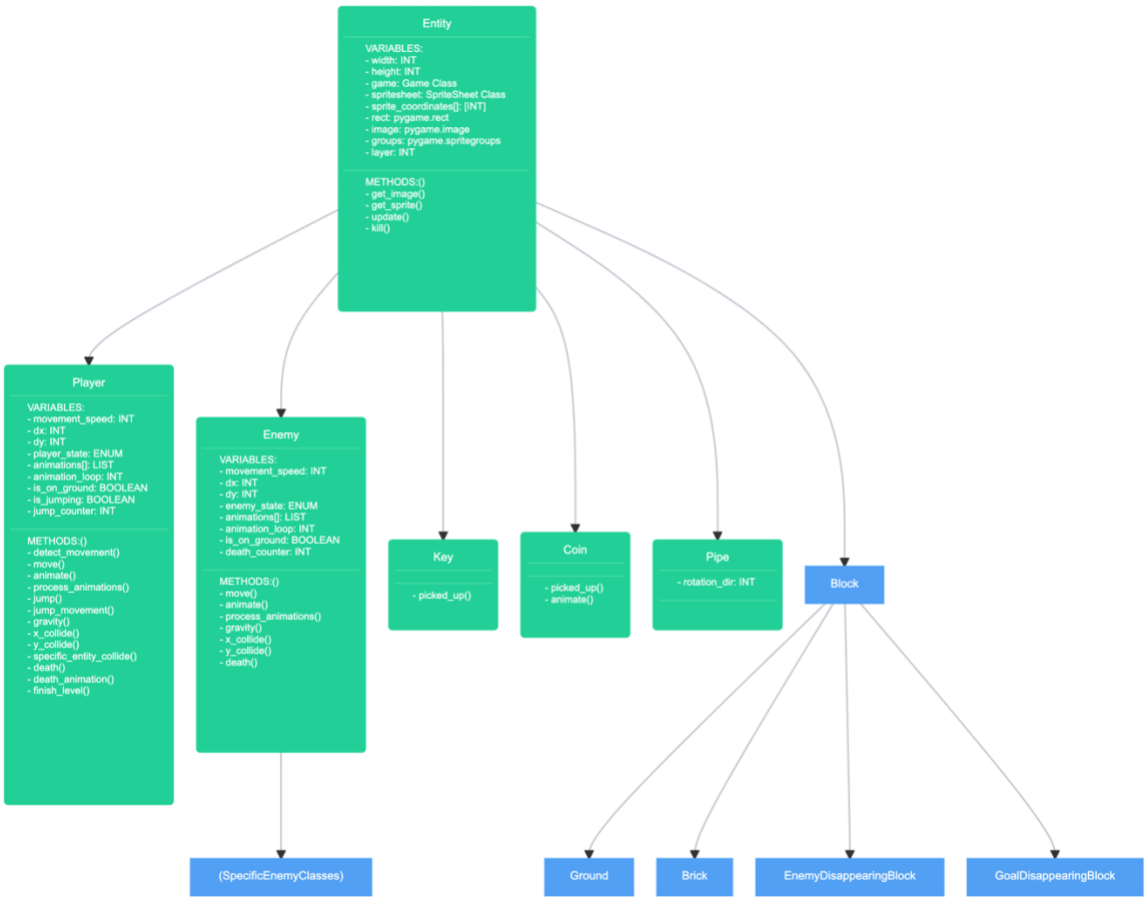
This is probably slightly confusing, but this is how I will organise my maps. I will have an Enum with the map id, which contains the name of the map "e.g., LEVEL_1_M1". The other tables are dictionaries that contain information, but these will be accessed by simple dictionary retrieving functions in the Enum class.

I will also need a function to get the next map in the array. Thanks to how Enum classes work in python, I can get an array of all the map ids. To create the "get next map" function I am going to have to find the of the current map in the huge linked list so I can return "maps[index + 1]". I could use the python built in index function, but this uses linear search. Binary search is a more efficient algorithm for this, and therefore I would like to make use of that to find the index of the current map within the array:

```
binarySearch(arr, x, low, high):
      while low != high:
            mid = (low + high)/2
            if (x == arr[mid]):
                  return mid
            else if (x > arr[mid]):
                  low = mid + 1
            else:
                  high = mid – 1
            endif
      endwhile
```

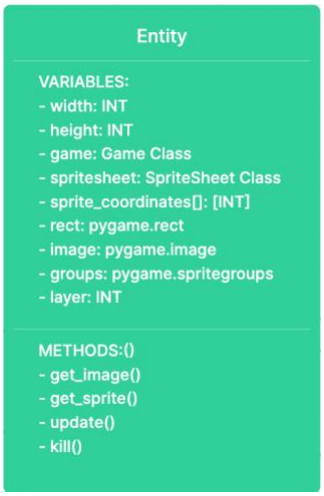
ENTITY CLASSES

The base entity class is defined, as it has basic attributes and methods shared by all entities. Here is a class diagram to show the inheritance:



- The block sprite and its children are mainly there to define different appearances. Instead of stating the appearance every time you want to create a different type of block, you just create a Ground entity, or a Brick entity.
- For the enemy, I have not fully decided what the special enemy abilities will be, nor what they'll look like, as I do not have their sprites yet, however I do know that all enemy sprites will have the same basic functions, and that the different enemies will differ at least by appearance (and therefore, height, width, animations), therefore it is worth making subclasses for each specific enemy type that inherit from a general enemy class.

PARENT ENTITY CLASS



- Game is there as a reference to the main game object, for all entities, so that they can always access all the game's attributes and methods.
- "Kill" destroys the object and removes it from the screen.
- Update is a function which every sprite automatically has (as this entity sprite inherits from a class defined in pygame), however, by default it does nothing. However, for certain sprites I will be using it heavily.
    - All entity update functions will be running at all times.

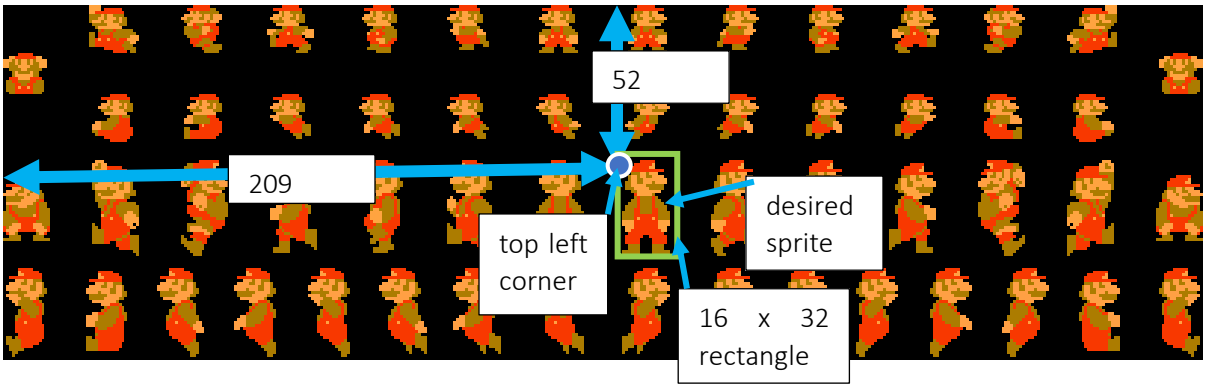
However, almost all these attributes and methods revolve around the "get image" and "get sprite" methods, which are used together to create a sprite object. I will explain how they are used here:

"GET_SPRITE" EXPLANATION

The image of the entity is the visual appearance it has, whereas the rect stores its position and size (x, y, width, height). Each pygame sprite needs these two attributes to be able to be drawn onto the screen. The image is gained from the following process. Say I am using this spritesheet for the character Mario:

At any point in time, I only want one of these sprite images to be displayed. For example, this one, when Mario is standing facing the right direction:

To isolate this image from the rest, the “get_ image” function is used. This passes the sprite coordinates (x, y) of the top left corner of desired image in the sprite sheet. In this case, the top left corner of this sprite is 209 pixels in the x direction and 52 pixels down in the y direction. It then takes this pixel creates a rectangle with the top left corner being the same pixel (209, 52), but its width being the “width” and “height” specified (in this case 16 and 32 respectively.)



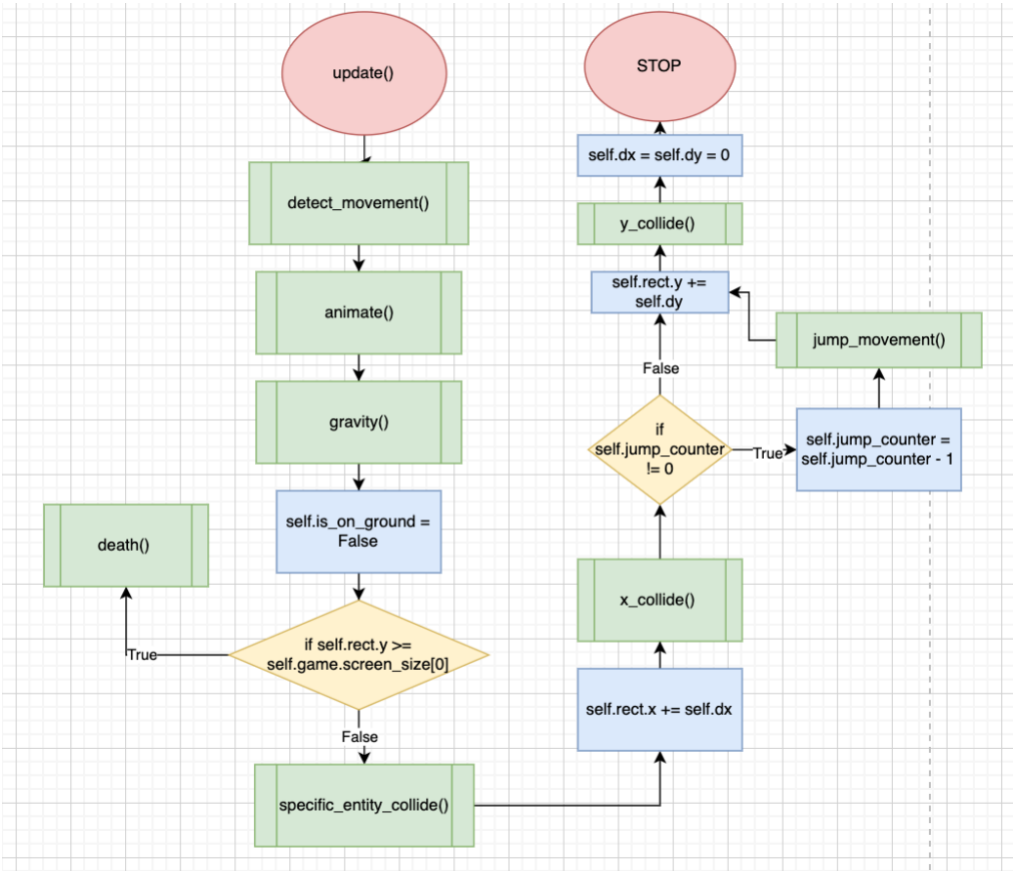
This is what “get image” does. “get sprite” uses the result of the previous function and sets the result as the “image” attribute of the entity class. Then it uses the dimensions and size of the image to create a rect object. Now the entity has the two elements it needs to be displayed and drawn onto the screen.

PLAYER CLASS

The player function is maybe more important than the game function. As the player, you interact with everything, so almost every event, from enemy collisions to block collisions to entering pipes to finishing levels and many more things are done in this class.

The main thing for the player class is its update function. This is where all other functions are run. This function will always be run as all update functions of all entities are run by the game class. This is a very long function, but I will do my best to explain.

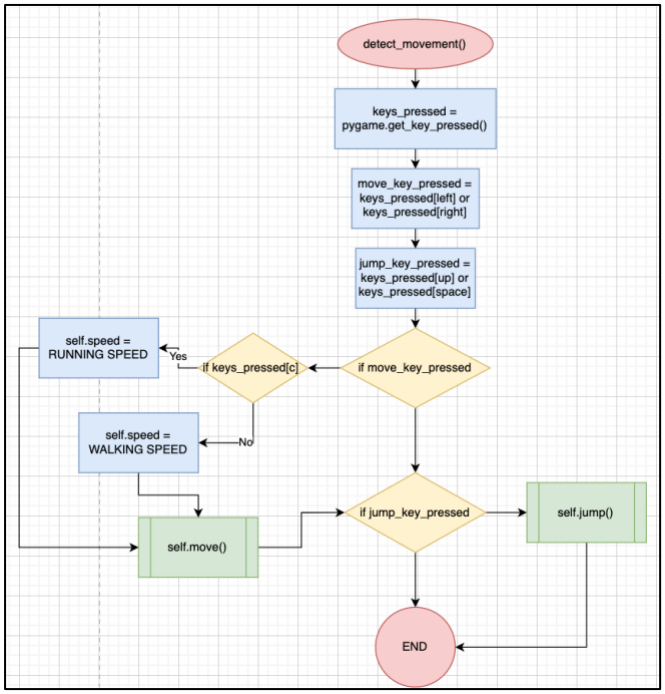
UPDATE METHOD



The order of this function does not matter too much, just as long as the detect movement function is at the beginning, followed by the animate function, and the “x collide”/”y collide” statements are right after the “self.rect.x/y += self.dx/y” statements. Everything else just needs to be run at some point during the function.

I will go through each of these functions right now:

DETECT_MOVEMENT METHOD



This is a relatively simple function to explain.

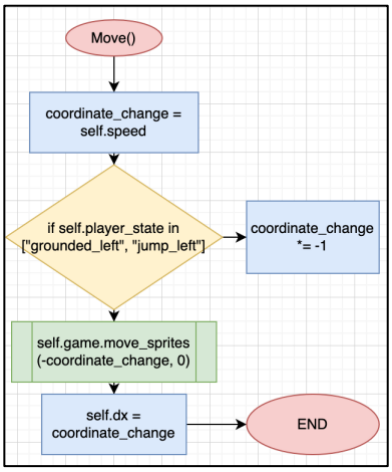
1. It checks the key pressed and sees if it is one of the assigned movement or jump keys.

2. If either of these has been pressed, then the relevant function is run.

3. If the running key ("c") is pressed then the speed of the player changes to the constant value for running speed (which is higher than the value for walking speed)

- I believe the pygame function creates a dictionary of all keys and assigns a Boolean value to each based on whether it has been pressed, so this function should detect if both types of keys have been pressed at the same time.

MOVE METHOD

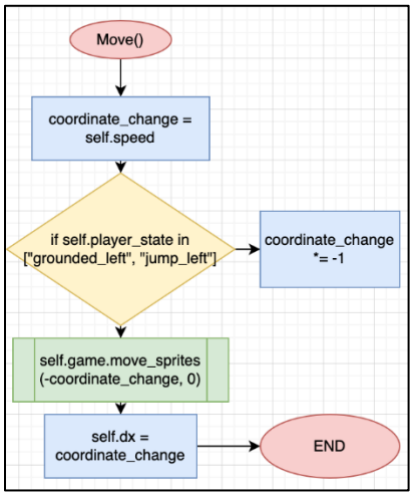


1. Coordinate change is set to the player's movement speed, which was set in the previous function
2. The coordinate change is made negative if the player is facing left.
3. The "move sprites" is to create the illusion of camera movement and keep the player in the centre of the screen even when it moves.
    a. It will move every sprite on screen in the opposite direction whenever the player moves.
    b. While this is happening, say the player will be moving forwards by 3 pixels
    c. However, simultaneously the player will be moved backward 3 pixels by a "move sprites" function.
    d. Therefore, in total the player is technically moving 0 pixels and the player stays in the centre.

e. However, every other sprite has moved backwards 3 pixels, which gives the illusion that the player has moved forwards while the camera has followed them.

4. The dx variable stores any changes to x before its value is eventually added to x at the end.

JUMP METHOD



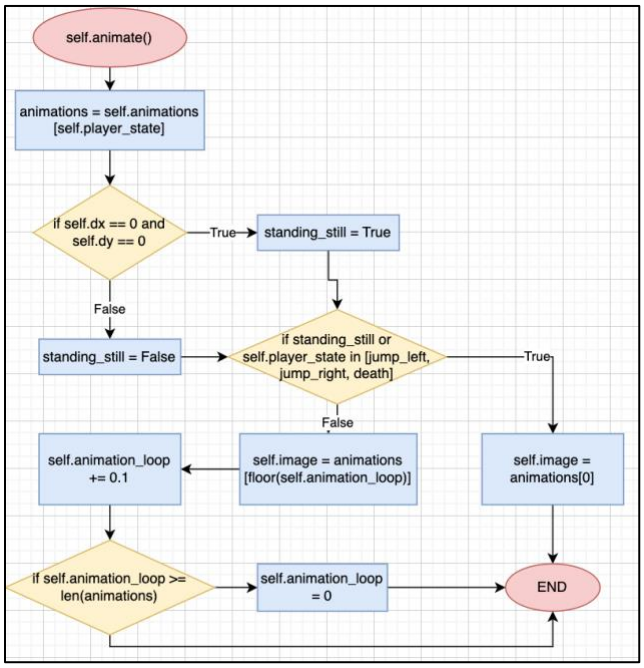
This function simply sets a few flags off and changes player states to inform the program that the player is now jumping.

This is only possible if the player is not currently on the ground.

The actual jump movement is dealt with later in the appropriately.

Now that "detect movement" has been dealt with, the next big method run during the update function is animate:

ANIMATE METHOD:



Firstly, the program retrieves the correct animations for the player due to its player state.

There are two types of animations for the player:

1. There is a singular sprite for the move, e.g., the player jumping in the right direction requires 1 image which stays until the jump ends. This is for when the player is standing still, jumping, or has just lost a life.
2. There is a series of sprites that need to be played in a loop.

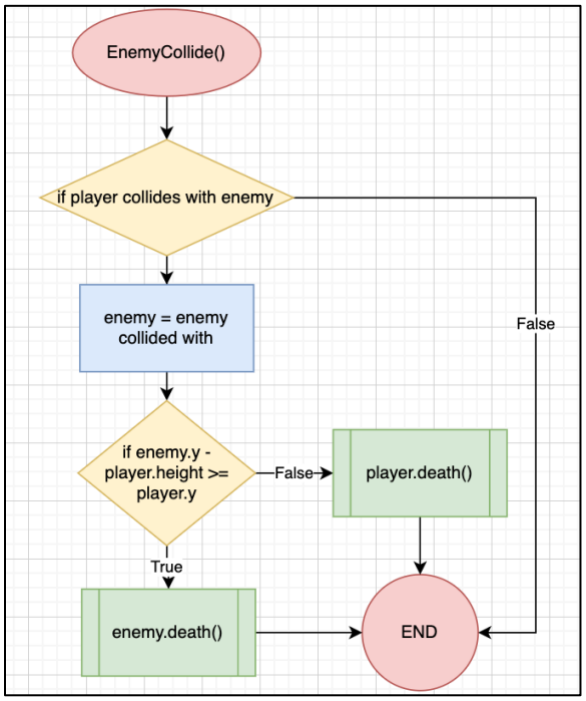
The first two if statements in this method are to determine which type of animation is required.

If the loop is required, then the attribute "animation loop" is required. As the animate function will always be running (as the update function is always running), this attribute slowly increases (and is reset to 0), slowly looping through and displaying each image in the series of images, to form an animation. This will be the case when the player is walking.

THE NEXT FEW PARTS OF THE UPDATE FUNCTION:

1. The gravity function checks to see if the player is in the air (using self.is on ground), if it is then the player falls, otherwise no change is made to dy.
2. The player is always set to not be on the ground at some point in the function, to make sure that it always checks for gravity movement.
    a. Otherwise, if it walks off a block into thin air, the player will walk on thin air as there is no flag telling it that it is no longer on the ground.
    b. If the player is actually on the ground, then this flag will be set to true during the "y collide" function.
3. The if statement checks to see if the player has fallen off the screen. This is a death condition, and therefore if this has happened then the death function is run
    a. The death function makes use of the built in "kill" function for sprites, which removes the sprite from the screen.
4. The specific entity collide function is for collisions with specific entities, like coins, keys, and enemies. The function simply checks if there is a collision with any of them and acts on it if there is.

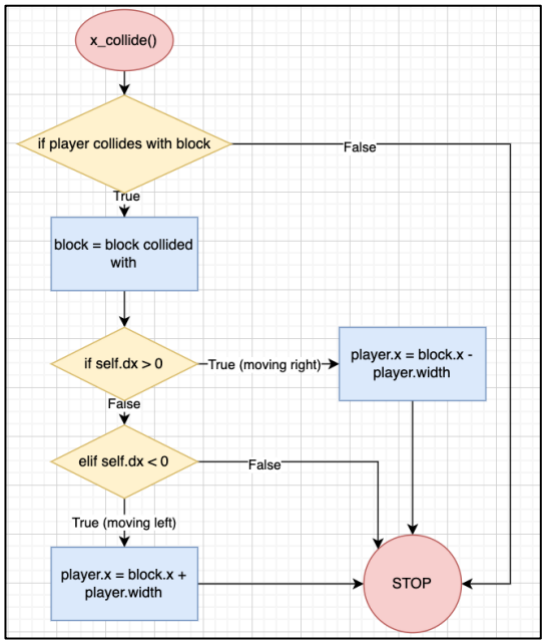
ENEMY COLLISIONS:



I have much more loosely defined this flowchart (it is still a method of the player class).

The second if statement is purely checking if the player is directly above the enemy when the collision happens, as this is the condition for the player to kill the enemy. If any other form of collision happens, then the player is meant to die.

X_COLLIDE



I have also much more loosely defined this code.

This function must be run after the line "self.rect.x += dx", where the potential changes to x are made to the x position of the player. This is to ensure that the new x position of the player won't make it overlap or go inside another entity.

Obviously, a game that relies on proper physics and logic cannot have such an event occurring, so that is what this function does. If the new x position would mean that the player would be too far left/right into a block, then this function simply resets their position to the right/left of the block (done in the statements after the second if statement.)

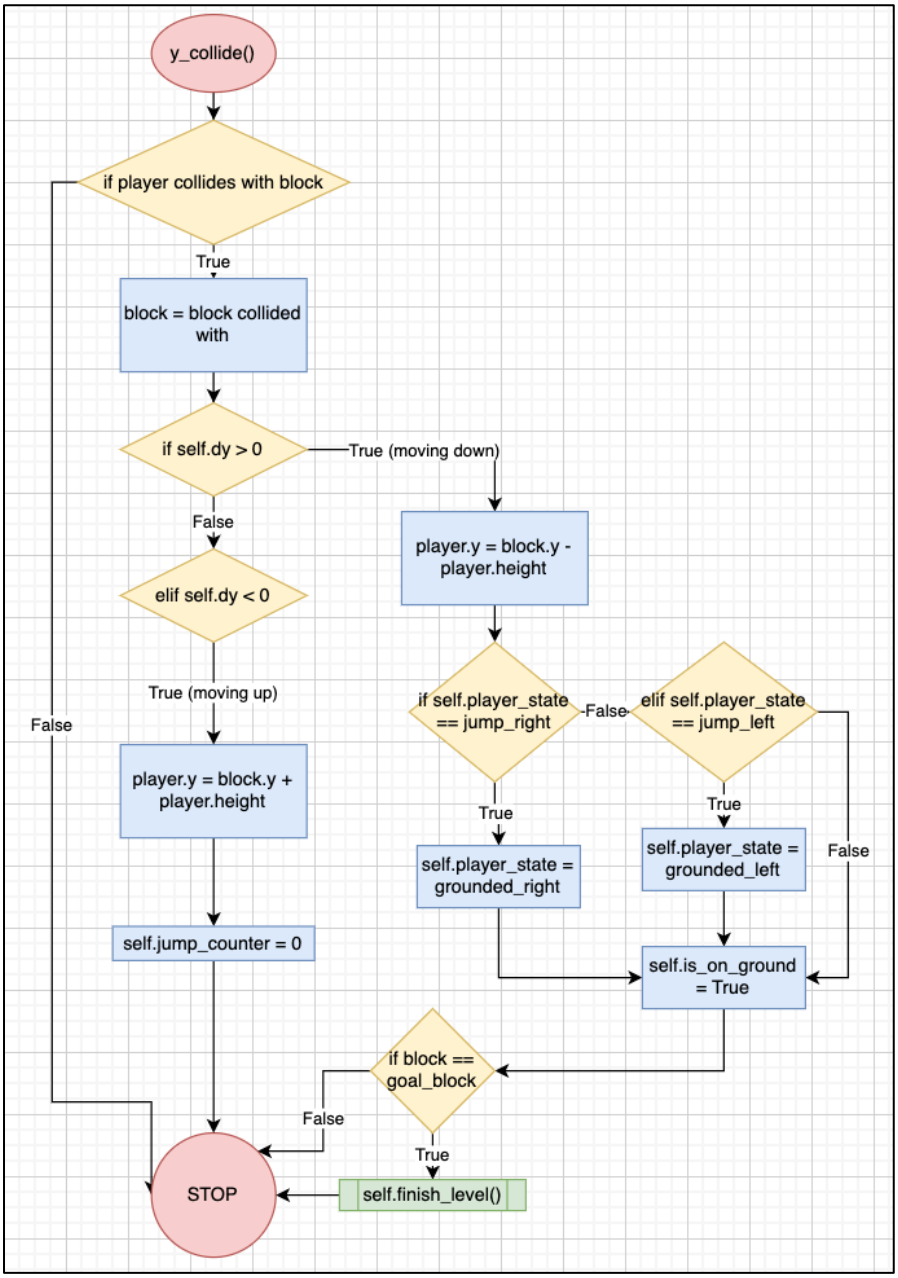
The 3rd if statement is required as the collision could have been vertical and not horizontal, so an extra check is required to see whether it was a collision when the player was moving left.

JUMP_MOVEMENT

```
function jump_movement():
      jump_dy_change = (self.jump_counter/6) * -1
      self.dy = jump_dy_change
```

- This function is very simple, but also required as the actual jump function doesn't make any changes to dy (attribute that stores any changes to y).
- I use the jump counter as part of the calculation, as I want the jump to have a high velocity at first, but slow down as the player gets to the peak of the jump.
- The dy change is made negative as a negative y change means going up.
- A jump movement function is required in the first place as I want the jump to be happening for longer than it takes for the next update function to happen (hence why I have used a jump counter attribute, in the same way I used an animation loop attribute for animations.)

Y_COLLIDE

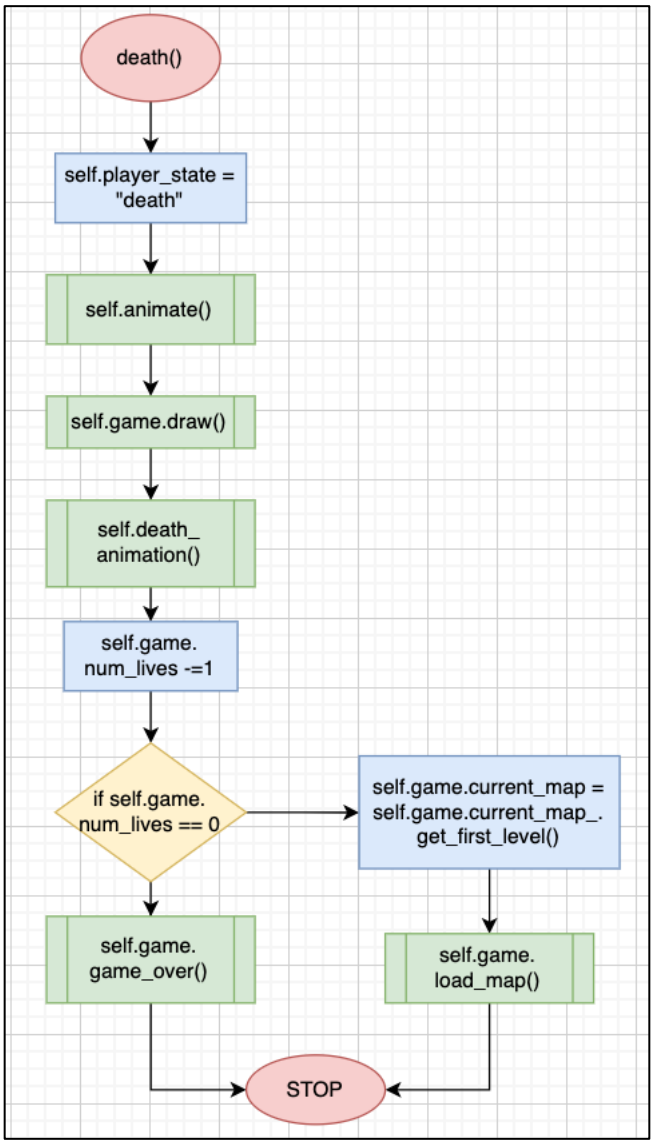


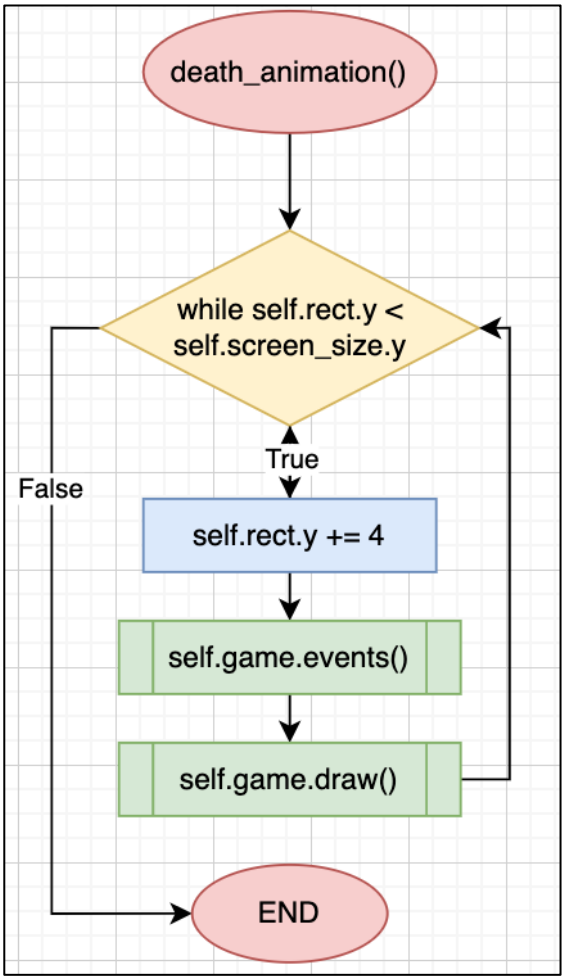
- This function is really similar to the “x collide” function, however there is a little more to do as jumps and the concept of gravity are more complicated than just walking.
- The player must change its state from a jumping one to a grounded one, for the animate function, which happens on the right side of the flowchart
    - An extra check must happen as the player could have just fell onto the ground while being in the “grounded right” player state, and as a result, there needs to be a check to make sure the player is jumping and looking left before turning it to grounded and looking left.
- The jumps need to be cancelled if a vertical collision happens.

- For a downwards collision this is done through the “is on ground” flag
- However, for an upwards collision the player will not actually be on the ground, so the jump counter is set to 0 instead to cancel the jump.

- If the player landed on the block (downwards collision), then the block is checked to see if it’s a block that denotes the end of the level. If this is the case, then the finish level method is called.

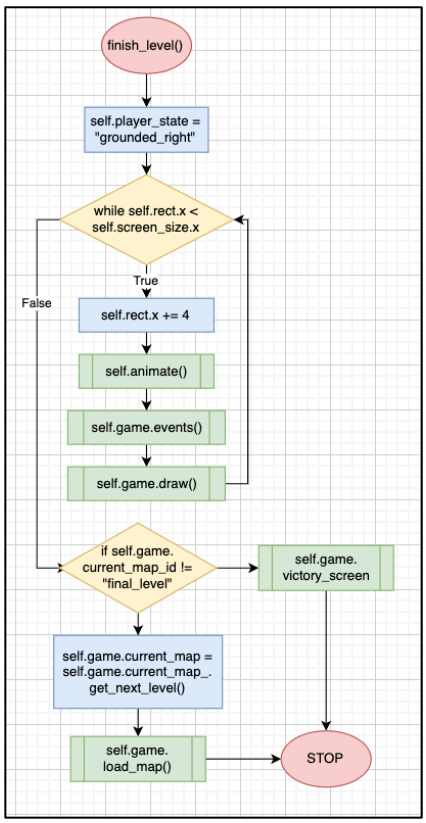
DEATH METHOD





- The first part of the death method is changing the appearance of the player to its death sprite.
- The death animation happens in a while loop as I want everything around the player to be frozen while the animation is happening, therefore I don’t want the game’s update function to run. However, I still need to draw the player every time it moves and catch any events like the player trying to close the window.

FINISH LEVEL METHOD



BASE ENEMY CLASS

The enemy class will be very similar to the player class, except:

- The enemy does not have jump functions
- The enemy does not trigger any other events like the player does (e.g., player has finish level (), collisions with other entities)
- Instead of "detect movement", the enemy just moves all the time.
    - When it collides with a wall, it simply changes direction
    - This also means it only has walking sprites and a death sprite (but no still/jumping sprites).
- In the enemy's death function, it mainly just leaves the screen due to the "kill" method, but it will check to see if it is the last enemy on the screen. If it is, then it will trigger a flag (a game class attribute) to notify the game that the enemies on the screen have been cleared.

Otherwise, everything works very similarly to the player class, and it would be redundant for me to re-explain them (e.g., the animate function would be almost identical).

KEY/COIN CLASSES

These classes work very similar. They have one method, “picked up”. This is run when the player collides with a key/coin. The key or coin will disappear, but they will check to see if they are the last key/coin in the map to be picked up. If this is the case, then this will trigger a flag in the game class (another reason why it is useful for the entities to have access to it) stating that all coins/keys in the map have been collected. These are reset when a new map is loaded. This is useful for what the keys/coins are there for, which you’ll see in the next section.

DISAPPEARING BLOCKS

To add more of a puzzle element, I have decided to add blocks that only disappear when certain conditions are met. The enemy disappearing block will only disappear if all the enemies in the map have been killed, whereas the goal disappearing block will only be used if all of the keys and coins have been collected, and the enemy has been killed.

2.4: KEY VARIABLES AND CLASSES

In my code I have a very large number of variables and classes that help it run, but I will do my best to explain the most important ones.

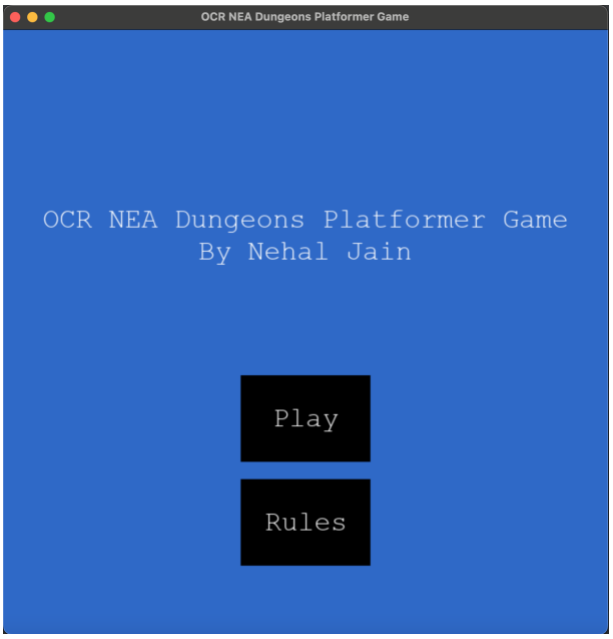
Variable Name	Data Type	Usage
"all sprites group"	pygame <LayeredUpdates> (a fancy array)	Stores all entities that are displayed on screen.
"blocks group"	pygame <LayeredUpdates>	Stores all block types, helps collision detection for player
dx/dy	integer	Stores any changes to the x direction/y direction for players and enemies which helps determine a lot of their movement.
"animation loop"	integer	Helps with the animations for players and enemies where the animation is a series of sprites being shown in a loop.
"player state"/"enemy state"	Enum	Tells program which state the player/enemy is in (grounded/jumping, looking left/right, moving/stationary) so that it can animate and move the sprite accordingly.
"current map id"	Enum	Refers to the current map. Is used as the key for a lot of dictionaries to find information like the next map (helped by binary search), the background music and background colour.
"BLOCK_SIZE_X/BLOCK_SIZE_Y"	integer	These constants are used determining the size of a lot of entities and are used to create a grid like structure for the game.

Class Name	Usage
Game	Stores almost all information to do with the game, the game loop that always runs to update and draw sprites and check for any other events. Has functions to run all the different screens (victory, game over), create the maps based on the blueprints, load the next map
Player	Almost all events in the game are caused by the player, obviously the game would not be able to function without the player. As the player is the one doing the moving, it usually collides with every other sprite, triggering all types of events like a player/enemy death, entering/exiting a pipe, the finishing of a level or the collecting of a key/coins.

Entity	This contains all the base attributes for every entity in the game like their position, size and getting an actual sprite image based on the image file location and a few other variables. Every single entity sprite object inherits from this class.
Animations (Enum)	This contains the key for the dictionary that contains all the coordinates for every single sprite in the game (that requires animation).
MapId (Enum)	Explained in the “current map id” variable above. The “current map id” will store a MapId Enum value.
Spritesheet	Stores all of the information to do with each spritesheet. Simply requires a file location of the image file and it will process the image and convert it into a format that is compatible with pygame.

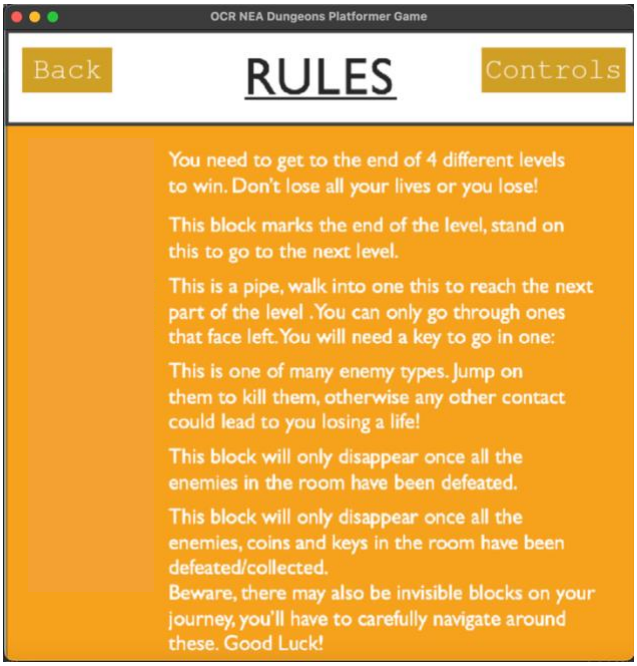
2.5: USER INTERFACE

I want the player to open to a very simplistic title screen. The purpose of the game is to play the levels, so nothing else should be emphasised. An example of what I will be aiming for is something simple like this:

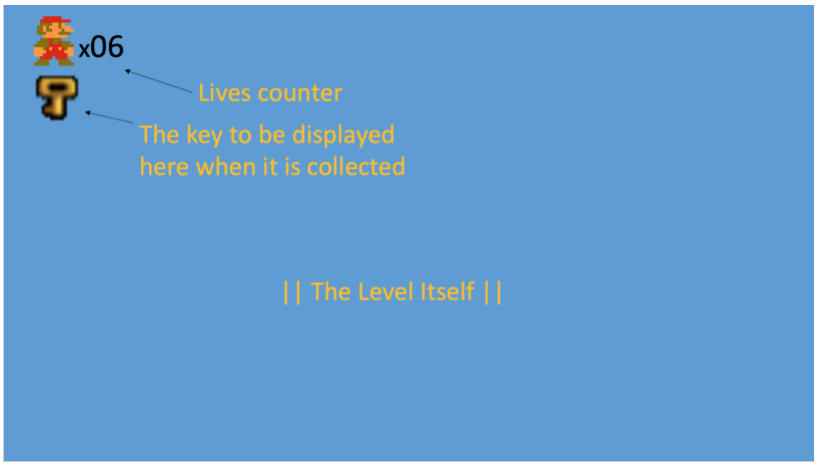


Game title, play button, and a button to navigate to a page showing the rules.

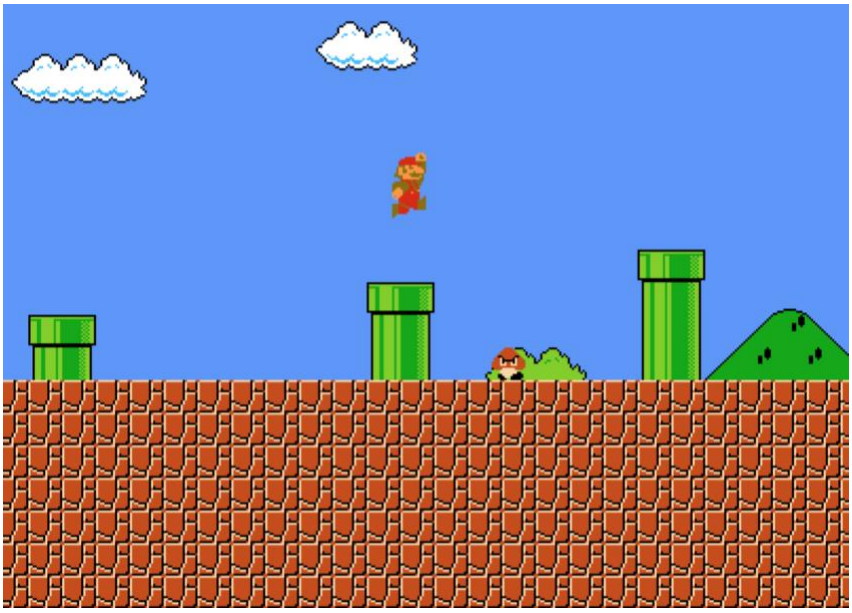
If the rules button is clicked:



I will aim to use templates that look like these. (The rules image has no sprites as I do not have the sprites right now). As for the levels themselves:



I want the overlay of the levels to be really simple. Only the lives counter and the key if it is collected need to be shown. As for the levels themselves, I will use a screenshot from a mario game to demonstrate:



I want the player to be in the centre and the camera to follow the player as this happens. I don't want the camera to move vertically however. There'll be a background colour with maybe one or two decorations (like those clouds and the bushes), and then the other entities also on the large map.

2.6: USABILITY FEATURES

I have split the usability features into the following:

- Learnability:
  - The rules and instructions will be there always when the player starts a game. The user interface is relatively simple to understand as other than a few menu screens the player just sees the game, which, with knowledge of the rules, should be simple to understand.
  - The menus and buttons are clearly labelled and there will be a level loading screen to inform the player what level they have just reached
  - There will be sound effects to help the player understand what event has just happened.
- Memorability:
  - The controls are relatively self-explanatory and therefore should be pretty memorable:
    - Left key to move left. Right key to move right.
    - Up arrow/Space bar to jump (both commonly used for that in games)
- Efficiency:
  - There is not much time wasted in pointlessly long loading screens with tons of animations and waiting time. There is one button to click, and then the program starts very quickly. If the program works as intended, the program should run very smoothly.
- Error reporting/Handling:
  - I will make sure the game is thoroughly tested so that it is very difficult to find. any errors, hence, why I am testing throughout development and at the end. I have also asked the stakeholders I interviewed if they wouldn't mind doing a few tests runs themselves, and they were more than happy to. This allows me to get beta testing as well as alpha testing.

2.7: DATA VALIDATION

- The only ways to input data into the program are to:
  - Click clearly labelled buttons which will always direct the player to another screen
  - Press keys for the player to move
    - Any other key will not do anything as there are if statements to detect which key has been pressed, and if say the "left arrow" has been pressed
    - These keys will only be functional when the player is on the screen and alive as they are contained by the player class.
- Otherwise there is no other way to input data into the program.
- I have added type hints throughout my program in the functions to help with debugging and figuring out whether there was any unexpected data passing into a function.

3: DEVELOPMENT

DEVELOPMENT 1: CREATING AN INTRO SCREEN

My plan is to create a screen to greet the player before they start playing the game. This is a less important issue compared to the making of the actual game itself, but I want to do this now because I want to avoid the possibility of creating it after making the game, accidentally causing huge difficult-to-fix errors in the game (because, for example, the game was dependant on running on launch for it to work). I'm not going to make it too aesthetically pleasing right now however, purely functional. However, before this, I must initialise pygame and all its pygame specific functions to create a window for the game and to make it possible to create a game in the first place. My plan is as follows:

1. Creating a program window
2. Creating a working clickable button

VERSION 1.1: CREATING A PROGRAM WINDOW:

MAIN GAME FILE

```
1   """This file is the one that is to be run to begin the program"""
2   import pygame
3   from game import Game
4
5   if __name__ == "__main__":
6       g = Game()  # creates game object
7
8       while g.running:
9           pass  # will contain a function later on
10
11      pygame.quit()
```

- As said before, I will be using a game class to store all the variables, sprites, maps, as this allows me to run pygame update functions (like draw all sprites onto the screen) to all sprites every tick (which is required).
- This is the file that is run to start the game.
- The game object is created, and during its __init__ function an attribute called "running" is stored in the game object and is made false when the program is meant to stop running. The while loop in lines 8-9 will stop and then line 11 will run.

CONSTANTS.PY

```
"""Contains constants like movement_speed for all sprites"""
STARTING_SCREEN_SIZE = [700, 700]
```

This is a file that will grow over time.

COLOURS.PY

```
"""contains colours enum class which contains rgb values for all colours used in the game"""
from enum import Enum


class Colour(tuple, Enum):
    """contains rgb values for all colours used in the game"""
    RED = (255, 0, 0)
    WHITE = (255, 255, 255)
    BLACK = (0, 0, 0)
    BLUE = (0, 0, 255)
    GREEN = (0, 255, 0)
```

There's nothing much to add here. I added some example colours to test but I have created an Enum class so that colours can be predefined here and accessed easily throughout the program wherever I need them.

GAME.PY

```
"""Game class to store all the variables, sprites, maps, as this allows me to run pygame update functions (like draw all
 sprites onto the screen) to all sprites every tick (which is required)"""
import pygame
from misc.constants import *
from misc.colours import Colour


class Game:
    """Contains the entire game and all variables and main pygame related functions like draw that continuously display
    all sprites every tick"""
    def __init__(self):
        pygame.init()  # starts pygame module and allows all its functions to work.
        self.size = STARTING_SCREEN_SIZE
        self.screen = pygame.display.set_mode(STARTING_SCREEN_SIZE, pygame.RESIZABLE)
        self.clock = pygame.time.Clock()
        self.font = pygame.font.Font("courier.ttf", 32)

        self.running = True
        self.playing = False

        self.intro_screen()

    def intro_screen(self):
        """Creates intro screen"""
        self.screen.fill(Colour.GREEN)
        pygame.display.update()

        while self.running:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.running = False
```

This is the file that contains the game object.

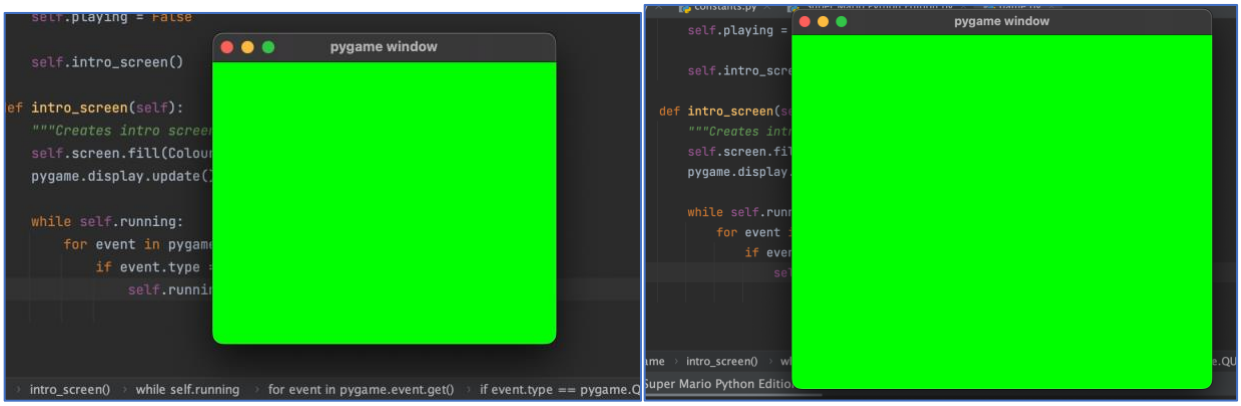
- The initialisation function ("__init__") creates a few pygame specific things that are required for pygame to run like initialising the clock and the screen.
- I have added "pygame.RESIZEABLE" in line 14 so that the user can change the size of the window to their liking.
- "self.running" will become false when the program is exited, while "self.playing" will only become true when the game is actually being played (i.e., they're not just on the menu screen, they're actually playing a level).
- I made a very basic intro screen, which will eventually greet the player when they boot up the game, but as it is right now it is simply a green screen to demonstrate that a working pygame window has been built.
- I've added functionality at the bottom to be able to quit the program if the "x" close button is clicked. As stated earlier, setting "self.running" to equal false will end the while loop (in lines 8-9 in the main program file.) and quit pygame.

TESTING VERSION 1.1

Expected Outcome:

1. A green screen pops up on a new window.
2. The window has the functionality to change size.
3. If I change the colour before running it reflects this change on the window (to see if it is actually responding to changes in my code)
4. If I close the window the program closes.

ATTEMPT 1:



Everything works as intended.

VERSION 1.2: CREATING A CLICKABLE BUTTON

This next step involves creating a button that detects when it has been clicked and sends a signal to the game to move to a different screen.

MINI CHANGES: CONSTANTS.PY AND COLOURS.PY

```
"""Contains constants like movement_speed for all sprites"""
STARTING_SCREEN_SIZE = [700, 700]
FPS = 80
```

Added 'FPS' constant so the program knows what frames per second it should aspire to run at

```
class Colour(tuple, Enum):
    """contains rgb values for all colours used in the game"""
    DULL_BLUE = (47, 105, 199)
    DULL_GREEN = (70, 134, 41)
    BLACK = (0, 0, 0)
    WHITE = (255, 255, 255)
```

Chose less in the face colours for testing purposes and removed unnecessary colours.

GAME.PY: EVENTS

```
26      def events(self):
27          """To always run no matter what to catch events like closing the program which are only specific to the game as
28          a whole"""
29          for event in pygame.event.get():
30              if event.type == pygame.QUIT:
31                  self.playing = self.running = False
32
```

I have added a new 'events' function to the 'game.py' file:

- The for loop to detect whether the program has been closed has been moved to the 'events' function.
- This is all to organise the code a little better, although I believe there'll be more to add to this afterwards.

GAME.PY: INTRO SCREEN

```
33      def intro_screen(self):
34          """Run to create and stay on intro screen until the 'Play' button is clicked. """
35          intro = True
36
37          title = self.font.render("WIP Platformer Game", True, Colour.WHITE)  # This is the actual title
38          title_rect = title.get_rect(x=180, y=10)  # This stores the title's position.
39
40          play_button = Button(x=275, y=250, width=150, height=100, fg_colour=Colour.WHITE,
41                               bg_colour=Colour.BLACK, content="Play", fontsize=32)
42
43          while intro and self.running:
44              self.events()
45
46              # Both of these variables are needed to determine whether the button itself has been clicked.
47              mouse_pos = pygame.mouse.get_pos()
48              mouse_is_pressed = pygame.mouse.get_pressed()  # Returns a list [l_click, r_click, m_click] and shows
49              # whether it has been clicked or not.
50
51              if play_button.is_pressed(mouse_pos, mouse_is_pressed):
52                  intro = False
53                  self.start_game()
54
55              self.screen.fill(Colour.DULL_BLUE)
56              self.screen.blit(title, title_rect)
57              self.screen.blit(play_button.image, play_button.rect)
58              self.clock.tick(FPS)
59              pygame.display.update()
```

I have updated the 'intro screen' function to actually create an intro screen.

- A while loop has been used as I need to constantly check whether the play button has been clicked.
- The 'title' and 'title rect' make use of pygame specific features to simply display white text saying "WIP Platformer game", a placeholder name for the game.
- The button object on line 40 has been defined in a separate file by me (showed later in this document).
- Lines 55-59 are there to display the background, title, and button

GAME.PY: START GAME

```
61      def start_game(self):
62          """Starts the game"""
63          self.playing = True
64          self.screen.fill(Colour.DULL_GREEN)
65          pygame.display.update()
```

Right now, this is just to see if clicking the button will prompt the program to switch screens. The colour will change to green to signify that the screen switched.

BUTTON.PY

```python
import pygame


class Button:
    """The object for creating a clickable button with text"""
    # fgc = foreground colour, bgc = background colour
    def __init__(self, x, y, width, height, fg_colour, bg_colour, content, fontsize):
        self.font = pygame.font.Font("Courier.ttf", fontsize)
        self.width = width
        self.height = height

        # Building the box for the button:
        self.image = pygame.Surface((width, height))
        self.image.fill(bg_colour)
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y

        # Building the text for the button:
        self.content = content  # the words on the button
        self.text = self.font.render(content, True, fg_colour)  # the text being made into a displayable object
        self.text_rect = self.text.get_rect(center=(width/2, height/2))

        self.image.blit(self.text, self.text_rect)  # actually displays them onto the screen

    def is_pressed(self, mouse_pos, pressed) -> bool:
        """Detects whether the button has been clicked or not"""
        if self.rect.collidepoint(mouse_pos):
            if pressed[0]:
                return True
        return False
```

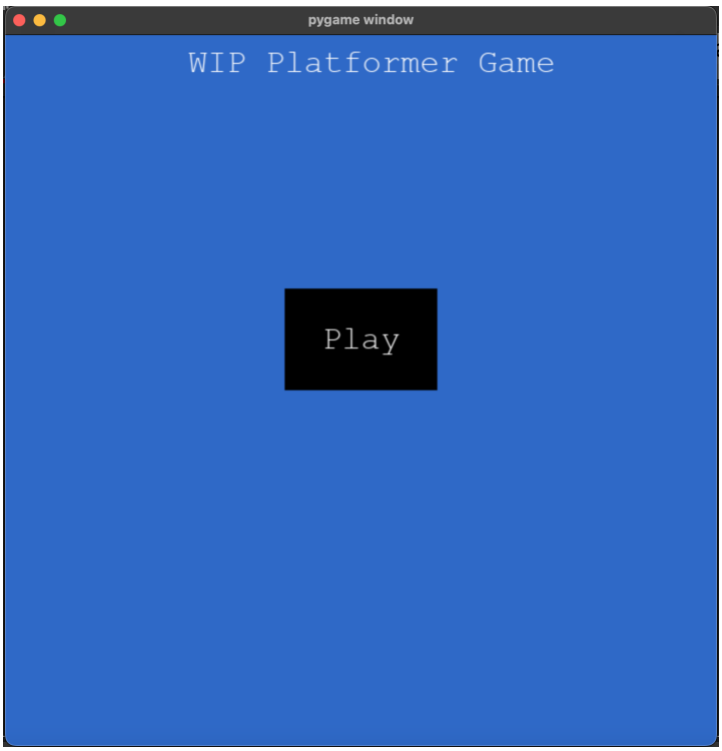
This is the code to create the button. Most of it is explained by the comments or is purely working with pygame specific functions and therefore I cannot explain them in depth.

TESTING VERSION 1.2

Expected Outcome:

1. A blue background displays
2. A black button and a title are displayed
3. When the button is clicked the screen goes empty and turns green.

ATTEMPT 1:

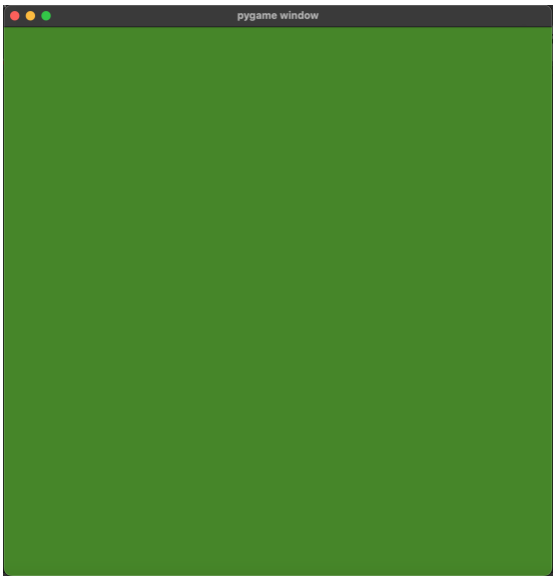


(1) and (2) have been achieved. However, (3) is not working, as the green screen flashes for a second, but it goes back to this screen. Looking back at my old code I noticed that after 'self.start_game()' is run then lines 55-59 are run and the blue background and the button and text are displayed again afterwards. Therefore, I made some changes.

```
33      def intro_screen(self):
34          """Run to create and stay on intro screen until the 'Play' button is clicked. """
35          title = self.font.render("WIP Platformer Game", True, Colour.WHITE)  # This is the actual title
36          title_rect = title.get_rect(x=180, y=10)  # This stores the title's position.
37
38          play_button = Button(x=275, y=250, width=150, height=100, fg_colour=Colour.WHITE,
39                               bg_colour=Colour.BLACK, content="Play", fontsize=32)
40
41          self.screen.fill(Colour.DULL_BLUE)
42          self.screen.blit(title, title_rect)
43          self.screen.blit(play_button.image, play_button.rect)
44          self.clock.tick(FPS)
45          pygame.display.update()
46
47          while self.running:
48              self.events()
49
50              # Both of these variables are needed to determine whether the button itself has been clicked.
51              mouse_pos = pygame.mouse.get_pos()
52              mouse_is_pressed = pygame.mouse.get_pressed()  # Returns a list [l_click, r_click, m_click] and shows
53              # whether it has been clicked or not.
54
55              if play_button.is_pressed(mouse_pos, mouse_is_pressed):
56                  self.start_game()
57                  break # ends intro_screen while loop
```

1. In line 57, I put a break statement to end the for loop at that point, as I want to leave the intro screen when the button is pressed. I am usually sceptical with using break statements as they can be difficult to use sometimes with nested if and while statements, but it works fine and does the job here.
2. I also realised while trying to fix this bug that lines 41-45 do not actually need to be run more than once, so I took them out of the while loop to reduce unnecessary function calls.

ATTEMPT 2:



Now after the change the button works, and it changes screen to what will be the eventual game itself.

A side note is that I realised that the window title says pygame window, and I want it to say the name of the game. This was obviously not big enough to have its own section, but I added this line to my '__init__' function for my game class:

```
pygame.display.set_caption('WIP Platformer Game')
```

THE WINDOW TITLE HAS BEEN SUCCESSFULLY MODIFIED

REVIEW OF DEVELOPMENT 1

This was a short and simple development with limited complexity but still contained critical parts of the final solution. I decided to make the rules page at the end, as I do not have the sprites yet to make a proper rules page (e.g., I can't explain that certain blocks do certain things without their images). I am ready to go into the more difficult and complex development 2.

SUCCESS CRITERIA

Success Criteria	Completed?
Creating a program window	Yes: Dev 1
Play button on the title screen that plays the first level	Yes: Dev 1
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	
A way to easily define sprite sheets	
Define ground and block	
Create a map based on a list of strings	
Create player sprite	
Add horizontal moving functionality to the player	
Create moving camera effect	
Animate moving functionality	
Add horizontal collisions	
Add vertical collisions	
Apply gravity to the player's movement	
Add a jump function and animate it	
Create parent enemy class	
Allow player to jump on enemies to kill them	
Create death function for player	
Ability to switch between two maps seamlessly.	
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 2: CREATING A BASIC MAP

AIMS:

The objective here is to:

- Create a basic entity parent class which all other objects can be derived from
- Create a way to easily import sprite sheets
- Create a ground entity and a block entity
- Create a map based on a list of strings.

VERSION 2.1: CREATING PARENT ENTITY CLASS

GAME.PY: '__INIT__' FUNCTION

```
20          self.all_sprites = pygame.sprite.LayeredUpdates()
21          self.blocks = pygame.sprite.LayeredUpdates()
22          self.warp_points = pygame.sprite.LayeredUpdates()
23          self.doors = pygame.sprite.LayeredUpdates()
24          self.pipes = pygame.sprite.LayeredUpdates()
25          self.enemies = pygame.sprite.LayeredUpdates()
26          self.player = pygame.sprite.LayeredUpdates()
27          self.powerups = pygame.sprite.LayeredUpdates()
```

I defined these sprite groups within the game file. Sprite groups are a part of making a pygame game. They simply categorise all the sprites together into different groups, making it easier for the program to only access certain groups. If I want to move all sprites (which I will most likely want to do for camera movement) then I can access them all using the 'all sprites' group. Or if I want to see if the sprite the player has collided into is a block or an enemy, I can see which group the sprite is in. My usage of them will be seen later. The 'LayeredUpdates()' function simply defines each one of these attributes as groups.

SPRITE_ENTITY.PY: ENTITY_LAYER AND ENTITY_GROUPS ENUM CLASSES

```python
import pygame
from misc.constants import *
from enum import Enum


class EntityLayer(int, Enum):
    """Identifies the layer the sprite is on (to know which sprites to display over other sprites"""
    BACKGROUND = 0
    BLOCK = 1
    PERSON = 2


class EntityGroups(list, Enum):
    """Gives information on the type of entity.
    Values are the sprite groups that these entities would be in."""
    BACKGROUND = []
    BLOCK = ["blocks"]
    DOOR = ["doors", "blocks", "warp_points"]
    PIPE = ["pipes", "blocks", "warp_points"]
    PLAYER = ["player"]
    ENEMY = ["enemies"]
    POWERUP = ["powerups"]
```

This is the beginning of my main file for the base parent entity class that I am to derive all my entities from. Along with their groups, I also must assign other attributes like their layer (if a background entity tile has a layer of 0 and a player has a layer of 2 then the player will be displayed over the background tile). I must also organise my code to add each entity to their respective groups. I have used simple Enum classes to make the code more organised (as opposed to just adding the groups manually when I define each entity class). The lists contain the sprite groups that each entity needs to be added to (other than the "all sprites" group).

SPRITE_ENTITY.PY: ENTITY CLASS

```python
class Entity(pygame.sprite.Sprite):
    """Base class for all entities"""
    def __init__(self, game, ent_groups, x: int, y: int, layer: EntityLayer, spritesheet_name: str, sprite_x: int,
                 sprite_y: int, width=BLOCK_SIZE_X, height=BLOCK_SIZE_Y):
        self.game = game  # So that the entity can have access to the game at all times
        self._layer = layer.value  # The player and the background for example will be on different layers so it knows
        # which sprite to put over the other.
        self.width = width
        self.height = height

        rem_groups = self.load_rem_groups(ent_groups.value)
        self.groups = tuple([self.game.all_sprites] + rem_groups)
        pygame.sprite.Sprite.__init__(self, self.groups)

        self.spritesheet = None
        self.image = None

        # Rect stores the position of the entity, img is just how it appears on screen.
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y
```

```
51      def load_rem_groups(self, rem_groups):
52          """Uses the strings from the 'EntityGroups' class, retrieves the appropriate groups from the game class returns
53          an array with all the groups."""
54          arr = []
55          for group_name in rem_groups:
56              arr.append(getattr(self.game, group_name))
57          return arr
```

This is the base parent entity class that I am to derive all my entities from:

- Line 29: I want all sprites to have access to the main game file
- Line 30: Layers explained above
- Line 32: Sprite sheets will be defined in the next version
- Lines 34-36: Adding the groups from the Enum class (defined above) to the entity class.

Everything else is relatively self-explanatory or makes more sense once the sprite sheet class has been defined.

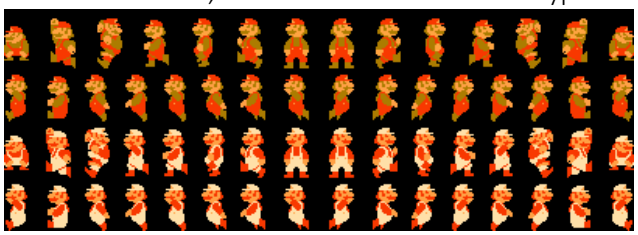
Unfortunately, I cannot test anything in this version right now, as I have not defined any sprite sheets, so I cannot display the entities. I will test this in “Version 2.3”, however as for right now I need to also create a way to import sprite sheets from their “.png” file to the actual game itself.

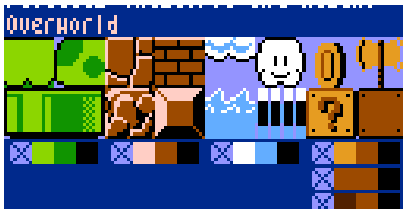
VERSION 2.2: CREATING AND IMPORTING SPRITE SHEETS

AIMS:

1. Define a sprite sheet class that:
    - Takes in a file location parameter and a background colour parameter
    - Loads the image file from the correct location and processes it to become a pygame image
    - Removes the background from the image
    - Merges the pygame image with a pygame surface object to create a complete sprite object.
2. Simply define a sprite sheet in the code by simply using one key variable to access both the file location and background colour of the sprite image.
3. Store all the sprite sheets in a dictionary or array, which can be accessed from the main game class
4. Add spritesheet functionality to game class and entity class

A tile set or sprite sheet (which I will be using to refer to them in this document) essentially contains all the images for a particular sprite image or entity. This can have multiple states of a character, as shown with this Mario one, or all the different types of blocks or sections of blocks in a map:





DEFINE SPRITE SHEET CLASS

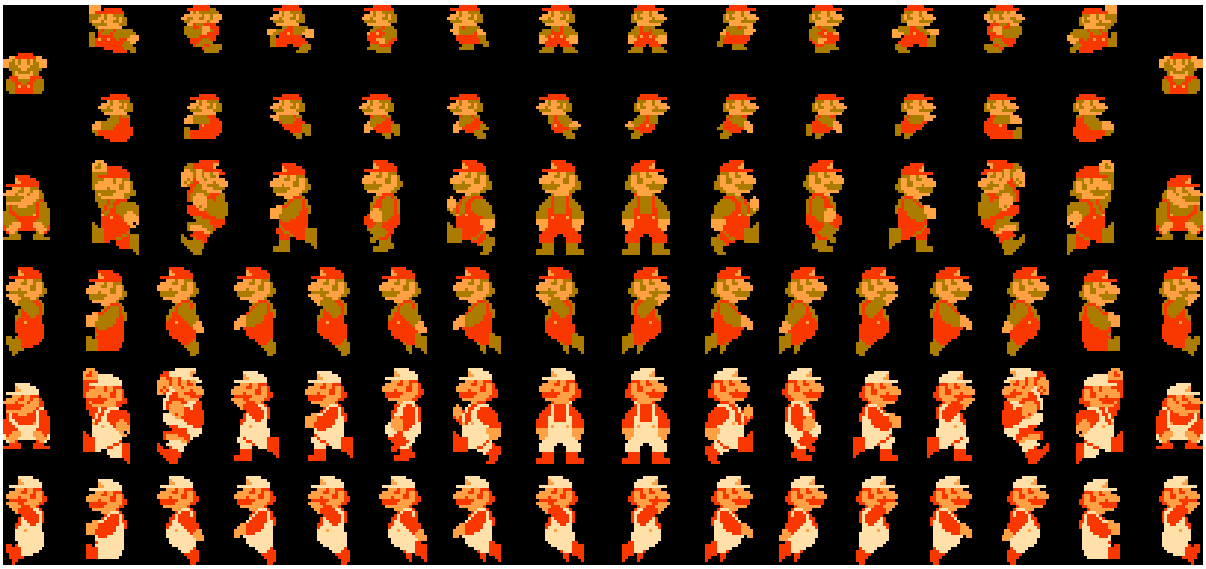
SPRITESHEETS.PY: SPRITESHEET CLASS

```python
"""Contains all spritesheets
character_spritesheet = SpriteSheet("img/main_chr.png", Colour.BLUE_MAIN_CHR) """
import pygame
from misc.colours import Colour
from enum import Enum, auto


class SpriteSheet:
    """Contains info about spritesheet. Example file_location: 'img/main_character.png'."""
    def __init__(self, file_location, bg_colour=Colour.BLACK):
        self.sheet = pygame.image.load(file_location).convert()
        self.name = file_location[4:-4]  # To remove "img/" from the beginning of the file_name and ".png" from the end
        self.bg_colour = bg_colour  # To remove a background colour of the image and make that part transparent.

    def get_specific_sprite(self, x, y, width, height):
        """Searches sprite sheet for desired sprite image using x,y coordinates of top left corner of sprite and its
        width and height. Retrieves this sprite image and makes its background colour transparent. Returns sprite to
        presumably the entity class the spritesheet is of (e.g returns the player sprite to the player class)."""
        sprite = pygame.Surface([width, height])
        sprite.blit(self.sheet, (0, 0), (x, y, width, height))
        sprite.set_colorkey(self.bg_colour)
        return sprite

    def __repr__(self):
        return f"SpriteSheet.{self.name.upper()}"
```

This is the start of the spritesheets file.

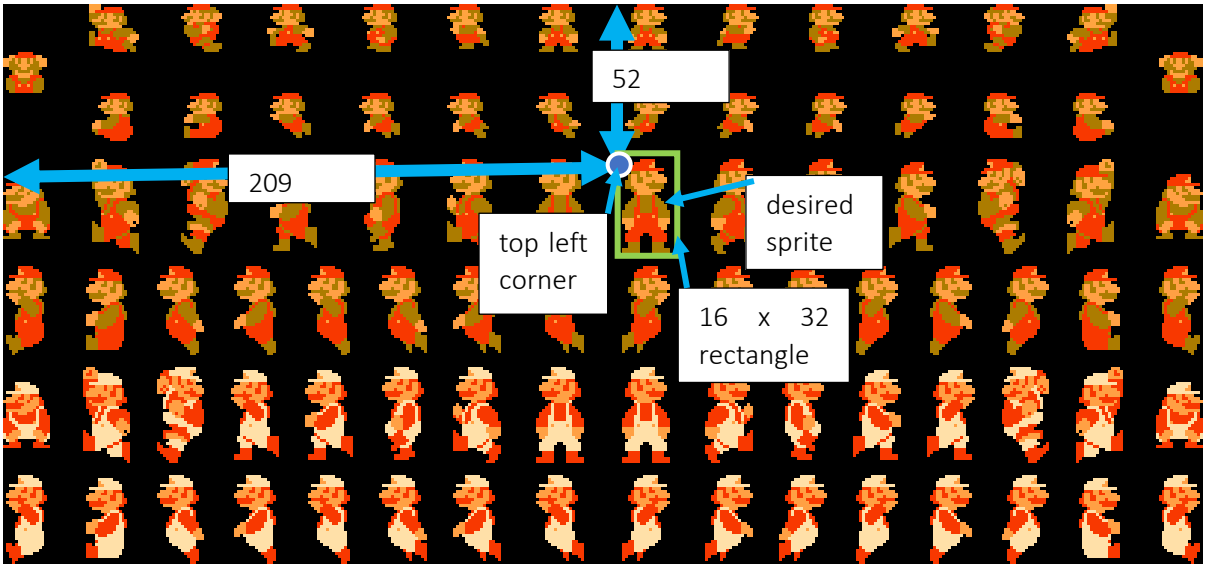
- The sheet attribute is the image of the sprite sheet converted into a file format that can be used with pygame functions.
- I have kept all of my spritesheets right now in a folder called “img” to better organise my file (hence why there is an “img/” at the front of each file location.

The get specific sprite function is used for the following scenario. Say I am using this spritesheet for the character Mario:



This would be the image that the “file location” variable is referring to, and the image that is converted and used for the “sheet” attribute. However, at one time, I only want one of these sprite images to be displayed. For example, this one, when Mario is standing facing the right direction:

To isolate this image from the rest, the “get specific sprite” function is used. This passes the x and y coordinate of the top left corner of desired image in the sprite sheet. In this case, the top left corner of this sprite is 209 pixels in the x direction and 52 pixels down in the y direction. It then takes this pixel creates a rectangle with the top left corner being the same pixel (209, 52), but its width being the “width” and “height” specified (in this case 16 and 32 respectively.)



In get sprite, line 18 creates a pygame sprite object (called a Surface) which has all the attributes of a sprite object (like movement and collision detection) but does not have a displayable image. On line 19, the process above is used to get the image, and then merge it with the imageless sprite object, to create a complete sprite. Line 20 is to remove the black background colour so that there is not a black box around Mario as he is in the game. However, if that specific RGB value is on anywhere in the sprite, then a different colour must be used, hence a variable is used (as opposed to it always using the colour black (0,0,0).

THE "__REPR__" FUNCTION:

The "__repr__" function is simply used during the development stage of the program. I will use it for almost all of my classes. Without it, if the object was printed to the terminal (for debugging purposes) it would display this:

```
<sprites.spritesheets.SpriteSheet object at 0x109747010>
```

Which is confusing and not very informative of the actual object itself. With the "__repr__" function this is printed instead (for a spritesheet with the name "Mario"):

```
SpriteSheet.MARIO
```

This tells us which spritesheet has been printed.

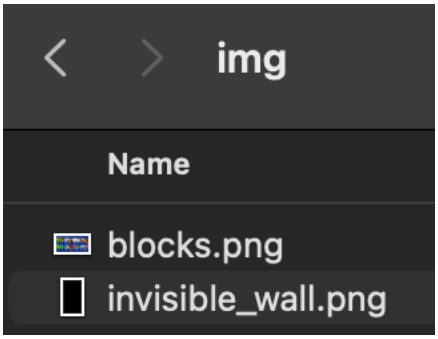
Now that we've defined the spritesheet class, my next plan is to store all the spritesheets in a dictionary or array to then be able to easily access all the spritesheets from the game class. I also want to make it very simple to then define a sprite sheet without having to type up too much code.

CREATE SPRITE SHEET KEY (USING ENUM)

SPRITESHEETS.PY: SPRITESHEETNAME ENUM CLASS

```
27  class SpriteSheetName(Enum):
28      """This enum is originally used to help create the sprite sheets, and then used to refer to them. It stores the
29      parameters required for the SpriteSheet function initialisation. The key is the file name of the sprite sheet and
30      the value is the background colour for the sprite sheet. Each sprite entity will store a 'SpriteSheetName' enum to
31      refer to their sprite sheet."""
32      INVISIBLE_WALL = Colour.BLACK
33      BLOCKS = Colour.BLOCKS_BACKGROUND
34      MARIO = Colour.BLACK
35
36      def get_filename(self):
37          return "img/" + self.name.lower() + ".png"
38
39      def get_background_col(self):
40          return self.value
41
42      def __repr__(self):
43          return "SpriteSheetName." + self.name + ":" + str(self.get_background_col().value)
```

- Firstly, I want a way to refer to the spritesheets in the dictionary.
    - In other words, I would like to create a key for the dictionary.
    - I've decided to use an Enum class, as with PyCharm, it'll show up the options in an autocomplete tab if I type out "SpriteSheetName.".
- I need to store the file names and the background colour for each sprite sheet.
    - I use the filenames as writing "img/" and ".png" around the name would be repetitive and would not be great variable names.
    - The filename is still accessible using the "get filename" function.
- For now, I have added a blocks tile set and an invisible wall spritesheet, which refer to the names of actual files in the "img" folder. You'll see the use of this class in the next function.



CREATE SPRITE SHEET DICTIONARY

SPRITESHEETS.PY: LOAD SPRITESHEETS FUNCTION

```
45  def load_overworld_spritesheets():
46      """Goes through the "SpriteSheetName" enum, creates spritesheet objects for all of them and then puts them all in
47      one dictionary which will be passed onto the 'game' class"""
48      file_names = [name for name in SpriteSheetName]
49      spritesheets = {}
50      for f_name in file_names:  # f_name is a SpriteSheetName enum
51          spritesheets[f_name] = SpriteSheet(f_name.get_filename(), f_name.get_background_col())
52      return spritesheets
```

This function simply collates all the defined keys in the “SpriteSheetName” Enum and uses the data in each Enum as the parameters when initialising the “SpriteSheet” classes. This function will then be used by the game file to store the “spritesheets” dictionary into the “Game” object.

ADD SPRITE SHEET FUNCTIONALITY TO GAME/ENTITY CLASS

GAME.PY: ‘__INIT__’ FUNCTION

```
19          pygame.display.set_caption('WIP Platformer Game')
20
21          self.spritesheets = load_overworld_spritesheets()
22
23          self.all_sprites = pygame.sprite.LayeredUpdates()
```

ENTITY_SPRITE.PY: ENTITY_SPRITE CLASS

```
40          self.spritesheet = game.spritesheets[spritesheet_name]
41          self.image = self.get_sprite(sprite_x, sprite_y)
42
43          # Rect stores the position of the entity, img is just how it appears on screen.
44          self.rect = self.image.get_rect()
45          self.rect.x = x
46          self.rect.y = y
47
48      def get_sprite(self, x: int, y: int):
49          """ In order to shorten long statement (self.spritesheet.get_specific_sprite(x, y, self.width, self.height)"""
50          return self.spritesheet.get_specific_sprite(x, y, self.width, self.height)
```

The changes have been highlighted. Now all the entity sprite object needs to access a specific sprite from its sprite sheet are the x and y coordinates of the desired sprite (how many pixels right and down from the top left corner the sprite is in the entire image file). The “get sprite” function will probably need to be reused later with animations, so I created it to shorten and neaten the code.

SUMMARY

This entire setup in Version 2.2 has allowed me to be able to define a sprite sheet by simply adding one line, “FILENAME = Colour.BackgroundColour”, into the “SpriteSheetName” Enum class (while also making sure the image file is in the “img” folder). This makes it much easier as I’ll have to add many sprite sheets and tile sets for the player, enemies, blocks, and other entities in the game.

Now that sprite sheets can be created, defined, and imported easily, we need to create the specific entities to use these spritesheets. In this next version I am going to create a ground, block, and invisible wall entity. I will obviously need to create many more types of entities; however, I have chosen these to help test features in versions 2.1 and 2.2. The ground and block entities are to ensure that the “get specific sprite” retrieves the correct image based on different coordinates. The invisible wall entity is just to ensure that the background colour of an object is removed, as the images for the block and ground sprites do not actually contain any background colour. I could also test whether the player sprite will display, but there are a few extra variables to that, like a different size and a few other things, so I will do it at the end of the development, almost as a final check that everything is working as intended. After this next version, I will finally be able to test versions 2.1, 2.2 and 2.3.

VERSION 2.3: CREATE GROUND/BLOCK ENTITIES

ENTITY_SPRITE _SHEETS.PY: VARIOUS ENTITY CLASSES

```
68  class Block(Entity):
69      """A block entity sprite"""
70      def __init__(self, game, x, y):
71          super().__init__(game, EntityGroups.BLOCK, x, y, EntityLayer.BLOCK, spritesheet_name=SpriteSheetName.BLOCKS,
72                           sprite_x=48, sprite_y=32)
73
74
75  class Ground(Entity):
76      """A ground entity sprite"""
77      def __init__(self, game, x, y):
78          super().__init__(game, EntityGroups.BLOCK, x, y, EntityLayer.BLOCK, spritesheet_name=SpriteSheetName.BLOCKS,
79                           sprite_x=32, sprite_y=16)
80
81
82  class InvisibleWall(Entity):
83      """An invisible block entity sprite"""
84      def __init__(self, game, x, y):
85          super().__init__(game, EntityGroups.BLOCK, x, y, EntityLayer.BLOCK, spritesheet_name=SpriteSheetName.INVISIBLE_WALL,
86                           sprite_x=1, sprite_y=31)
87
```

TESTING VERSIONS 2.1, 2.2, 2.3

This will be a challenge as I will test 3 different big additions to the program all at once. I could not think of many easier ways to do it without creating a lot of extra unnecessary code.

CHECK 1: PLACE A BLOCK IN THE TOP LEFT CORNER

This will test if almost everything above works. If the block is placed, with the coordinates stated (0,0), with the correct sprite then I have tested almost all of the things I have created.

GAME.PY: START GAME FUNCTIONS

```
70      def start_game(self):
71          """Starts the game"""
72          self.playing = True
73          self.screen.fill(Colour.OVERGROUND_BACKGROUND)
74          pygame.display.update()
75          self.create_tile_map()
76
77      def create_tile_map(self):
78          Block(self, 0, 0)
79          self.all_sprites.draw(self.screen)
80          pygame.display.update()
```

I have added this section of code into the main game file to see if a block can be placed at the top left corner. (I have also changed the background colour to what it will eventually be)

Expected Outcome:

1. The (new) blue/purple background is still there
2. There is a block at the top left corner of the program
3. The block has the correct sprite:

The block has been placed in the top left corner. However, there are still a few more checks to go through to make sure everything is running how I want it to.

CHECK 2: PLACE A BLOCK NEXT TO THE CURRENT BLOCK

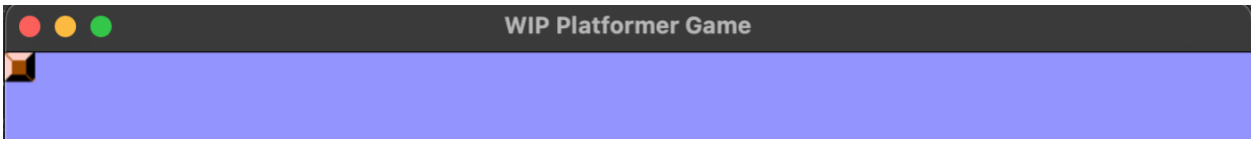
This test is to check that more than one block can be placed, and that the coordinates are indeed working correctly.

GAME.PY: CREATE TILE MAP FUNCTION

```
77      def create_tile_map(self):
78          Block(self, 0, 0)
79          Block(self, 1, 0)
80          Block(self, 0, 1)
81          self.all_sprites.draw(self.screen)
82          pygame.display.update()
```

Expected Outcome:

1. The block in the top left is still there
2. There is a block to the right of the original block and one below the original block.



This is not the desired result. It seems like there is still only one block. I'm not sure whether there can only be one block defined or they've all defaulted to the (0,0) position for some reason. I'm going to change the coordinates of the others to (4,4) and (8,0) to try and get a better picture of what is going on:

As you can see, the blocks have overlapped. After a long look through my code, I realised that although I have changed my blocks x and y coordinate on the screen, I have moved them by 4 or 8 pixels, whereas these images are 16 pixels large. Therefore, if I use multiples of 16, like (16,0) and (0,16), then I should hopefully achieve the desired result:

The check has been complete. However, I don't want to work in multiples of 16, as I have the space to fit in 44 blocks into this screen, so I could be using values up to 704 instead of 44. To fix this:

ENTITY_SPRITE _SHEETS.PY: INSIDE '__INIT__' FUNCTION:

```
44          # Rect stores the position of the entity, img is just how it appears on screen.
45          self.rect = self.image.get_rect()
46          self.rect.x = x * BLOCK_SIZE_X
47          self.rect.y = y * BLOCK_SIZE_Y
```

GAME.PY: CREATE TILE MAP FUNCTION

```
77      def create_tile_map(self):
78          Block(self, 0, 0)
79          Block(self, 1, 0)
80          Block(self, 0, 1)
81          self.all_sprites.draw(self.screen)
82          pygame.display.update()
```

This leads to the same result as the previous test. Therefore, the aims of this test have been achieved.

CHECK 3: PLACE THE OTHER TWO TYPES OF BLOCKS

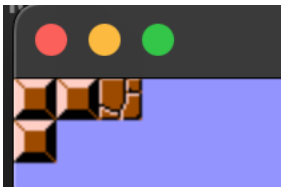
I want to check that not every block looks the same, and that the ground block looks like a ground block. I also want to see if the invisible wall entity is invisible, as this checks whether the function to remove the background colour is working. If it doesn't work, then the invisible wall block should appear black instead of, well, no appearance.

GAME.PY: CREATE TILE MAP FUNCTION

```
77      def create_tile_map(self):
78          Block(self, 0, 0)
79          Block(self, 1, 0)
80          Block(self, 0, 1)
81          Ground(self, 2, 0)
82          InvisibleWall(self, 3, 0)
83          self.all_sprites.draw(self.screen)
84          pygame.display.update()
```

Expected Outcome:

1. The ground block is placed to the right of the two blocks in the top row that have already been placed.
2. The “invisible wall” block to the right of this ground block does not appear black and rather has no appearance.



Both results have been achieved.

SUMMARY

So far everything is working as intended. We now have a base entity class to derive all our blocks, ground tiles, players, enemies, powerups and anything else on the map. We now have a spritesheet class to process a “.png” image into a pygame image. We can now take different sprites from the same file using different (x, y) coordinates. We can now easily define and add new spritesheets. We can remove the background colour of different sprites. We can merge the entities with the images to create sprites. We can place the sprites on the map using simple x and y coordinates. We’ve achieved all of our aims so far.

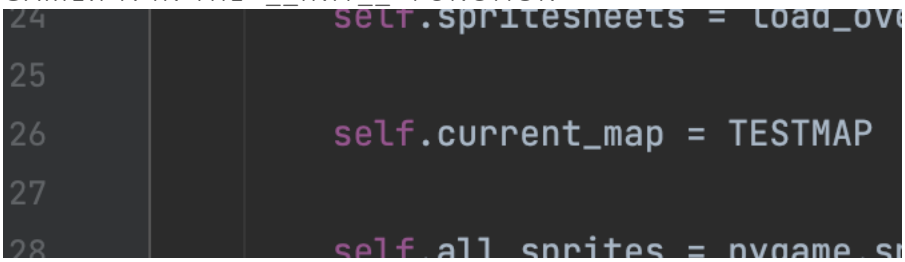
However, if you look at the create tile map function, it looks repetitive and tedious. For 5 blocks, it’s no problem, but with a map with thousands of blocks there will have to be thousands of lines of similar looking code. Having to make a small tweak to the map could take hours of searching and changing. Thus, the next version will be there to fix that.

VERSION 2.4: CREATE A MAP BASED ON A BLUEPRINT

The idea, from the design stage, is to have a huge row of letters that denote different blocks. A “G” could represent a ground tile whereas a “B” could represent a generic block. This could be stored in multiple long strings in an array, which can be iterated through to create a 2d grid. It is a little hard to explain but you’ll understand when you see the solution:

I’ve created an empty array called “TESTMAP” in a file called “maps.py”. This is purely to test all features until I actually make the levels.

GAME.PY: IN THE ‘__INIT__’ FUNCTION



I’ve created a new attribute called current map that stores the map currently on the screen.

GAME.PY: CREATE_TILE_MAP FUNCTION

```
def create_tile_map(self):
    for y, row, in enumerate(self.current_map):
        for x, item, in enumerate(row):
            if item == "B":
                Block(self, x, y)
            elif item == "G":
                Ground(self, x, y)
            elif item == "I":
                InvisibleWall(self, x, y)
    self.all_sprites.draw(self.screen)
    pygame.display.update()
```

This will iterate through the array and place a block wherever there is a B, and the same logic for the others. The array I will be using for this:

MAPS.PY: TESTMAP ARRAY:

```
TESTMAP = ["..............",
           "..............",
           "B............B",
           "B............B",
           "GGGGGGGGGGGGGG"]
```

The dots are for empty space, where there is no block or entity of any kind. This should create a map where the blocks are where the "B"s are, and the ground tiles are where the "G"s are.

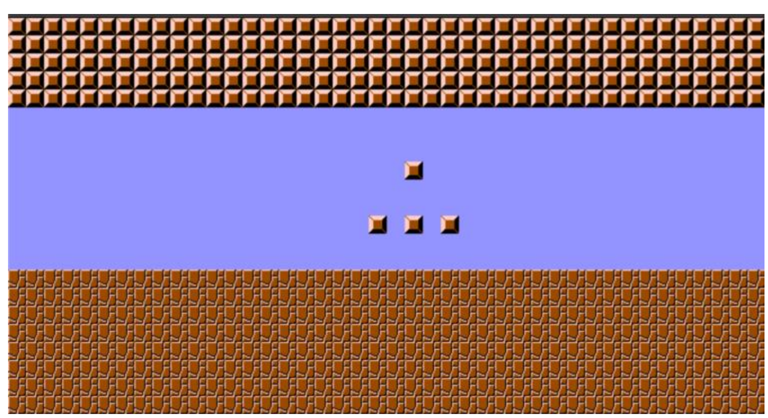
TESTING VERSION 2.4

Expected Outcome: A map is created where the blocks are where the "B"s are, and the ground tiles are where the "G"s are (from the test map array shown above). There should not be anything where the dots are you should just be able to see the background colour:

ATTEMPT 1:

This was the expected outcome. However just to be sure I am going to try and create a slightly more impressive map:

```
TESTMAP = ["BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
           "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
           "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
           "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
           "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
           "I........................................I",
           "I........................................I",
           "I........................................I",
           "I.....................B..................I",
           "I........................................I",
           "I........................................I",
           "I...................B.B.B................I",
           "I........................................I",
           "I........................................I",
           "BGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
           "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG"]
```



There are a few things I am yet to test out, which is what I am going to do in the next version, when I attempt to add a player sprite. This will allow me to test whether the program works despite having a sprite that is not a 16x16 square, or a sprite where removing the background colour removes some background colour (unlike the block and ground images) but not the entire image (like the invisible wall image). I believe it should all run smoothly, and this should be more of a final check, so hopefully that is the case.

VERSION 2.5: CREATING A PLAYER SPRITE

AIMS:

The objectives in this version are to

1. Define a player class as a child class of the entity class
2. Ensure that it is twice as tall as a block
3. Give it the correct sprite
4. Be able to spawn a player into the map using the “create tile map” function.

SPRITE_PLAYER.PY: PLAYER CLASS

```
14  class Player(Entity):
15      """General class for Player/NPC"""
16
17      def __init__(self, game, x, y):
18          super().__init__(game, EntityGroups.PLAYER, x, y, EntityLayer.PERSON, SpriteSheetName.MARIO, sprite_x=209,
19                           sprite_y=56)
20
21          self.height = BLOCK_SIZE_Y * 2  # player is twice as tall as a regular block
22          self.rect.height = BLOCK_SIZE_Y * 2
```

Due to everything achieved so far in development 2, this should be relatively straightforward. This should achieve aims 1 2 and 3. The “SpriteSheetName” has been set to the Mario sprite sheet, which I defined in the “SpriteSheetName” Enum class.

GAME.PY: CREATE_TILE_MAP FUNCTION

```
82      def create_tile_map(self):
83          for y, row, in enumerate(self.current_map):
84              for x, item, in enumerate(row):
85                  if item == "B":
86                      Block(self, x, y)
87                  elif item == "G":
88                      Ground(self, x, y)
89                  elif item == "I":
90                      InvisibleWall(self, x, y)
91                  elif item == "P":
92                      Player(self, x, y)
93          self.all_sprites.draw(self.screen)
94          pygame.display.update()
```

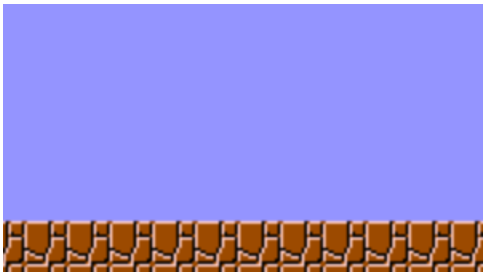
Lines 91 and 92 should achieve aim 4. I have added a “P” to the test map

TESTING VERSION 2.5

Expected Outcome:

- This Mario image appears:
- Ensure that the sprite is twice as tall as a block
- The player sprite shows where the P has been placed
- The background colour (black) is not visible when the sprite is displayed.

ATTEMPT 1:



Mario should have been in this spot. However, while his surface object may be there, the image has not displayed.

I tried to figure out a way to understand what had gone wrong. I tried seeing whether maybe something had gone wrong when the background colour was removed. I tried to test whether the background colour was removed in the last development, but there is a chance that doing that with an image where everything was meant to be removed didn't fully check whether it would work when not everything was to be removed.

```
35    MARIO = Colour.BLACK
```
→
```
35    MARIO = Colour.WHITE
```

I made this change, to try and not remove the actual background colour of the image (black), to see if that would make Mario appear.

ATTEMPT 2:

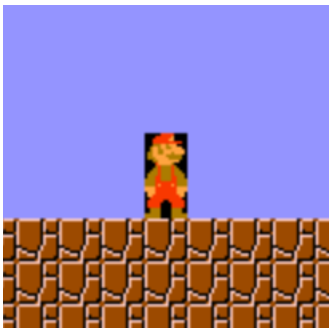
Mario is now indeed in the correct spot. Although he is only a 16x16 pixel square when I want him to be a 32x16 rectangle. The height has not changed despite lines 21 and 22 in the player class shown above. The background colour removal also did not work.

I realised that if you look at the code for the entity class initialisation, the height and width can be attributes passed into the function, but they are by default set to the constants "BLOCK_SIZE_Y" and "BLOCK_SIZE_X" (which are both set to 16). This was completely fine for the other blocks where these were indeed their dimensions, however for Mario I want the dimensions to be something else. As the height is now 16 during the whole initialisation function, the "get specific sprite" function only returned a 16x16 square, so changing the height and rect height afterwards made no impact on the image. Therefore, to change this I should be able to pass in a height of 32 to the "super()" function (which inherits the init function from the entity class), which then means that an image of 32x16 is retrieved.

ATTEMPT 3:

SPRITE_PLAYER.PY: PLAYER CLASS

```
class Player(Entity):
    """General class for Player/NPC"""
    def __init__(self, game, x, y):
        super().__init__(game, EntityGroups.PLAYER, x, y, EntityLayer.PERSON, spritesheet_name=SpriteSheetName.MARIO,
                         sprite_x=209, sprite_y=52, height=BLOCK_SIZE_Y*2)  # player is twice as tall as a regular block
```



As you can see, we now have the full Mario image. I am used to using default variables as they help reduce errors by ensuring that a variable is always passed in no matter what, however I've now come to the realisation that this promotes weak typing and could lead to an error like the one I've just encountered, where I simply forgot that the height and width were indeed parameters of the entity "__init__" function. I went and removed the default variables in the function and then manually added the heights and widths to each object, as there will be many more entities later that do not use a height or a width, and I think it looks more descriptive or informative in the class definition if specifically define the width and height for each one.

As a side note, I am insistent on using the block size constants, as if I want to enlarge every sprite later, all I have to do is increase the block size constant values, as opposed to changing every single height, width, and x/y value in every entity.

However, we still have this background colour problem to deal with. I changed Mario's background colour back to black to try and figure out what was going on.

After placing a few print statements throughout the "sprite_sheets.py" file, I came to realise a very confusing conclusion. The list of filenames it was going through seemed to only be the first two Enum values defined:

RESULT OF PRINTING THE ARRAY "FILENAMES" IN THE FUNCTION "LOAD_OVERWORLD_SPRITES":

```
[SpriteSheetName.INVISIBLE_WALL:(0, 0, 0), SpriteSheetName.BLOCKS:(0, 41, 140)]
```

Which is odd because changing the background colour for Mario to white seemed to make a difference and make the image appear, insinuating that Mario's spritesheet had indeed loaded then. However, when the colour is set to black it didn't load. I tried to see what spritesheet was loaded for Mario when the player class was defined, and this is what I got:

SPRITE_ENTITY: CHECKING WHICH SPRITESHEET MARIO HAS

```
41          self.spritesheet = game.spritesheets[spritesheet_name]
42          if spritesheet_name == SpriteSheetName.MARIO:
43              print(self.spritesheet)
44          self.image = self.get_sprite(sprite_x, sprite_y)
```

```
SpriteSheet.INVISIBLE_WALL
SpriteSheet.INVISIBLE_WALL
SpriteSheet.INVISIBLE_WALL
SpriteSheet.INVISIBLE_WALL
SpriteSheet.INVISIBLE_WALL
SpriteSheet.INVISIBLE_WALL
```

This is the result I got in my terminal. I didn't show all of them, but there are 19 of these. I have 18 invisible wall blocks on my map, insinuating that Mario and the invisible walls have the invisible wall spritesheet. However, it should surely only be printing the spritesheet if the spritesheet name is Mario's. I then realised that I may have completely misunderstood how an Enum works, and that it may treat "SpriteSheetName.MARIO" to be the same as "SpriteSheetName.INVISIBLE_WALL", as although the key is different, their value is the same ("Colour.BLACK", as they have the same background colour). I did an extra check to see if this was the case:

```
print(SpriteSheetName.MARIO == SpriteSheetName.INVISIBLE_WALL)
```

This resulted in a true statement being returned. However, if I now change Mario's background colour in the code to "Colour.WHITE", it returns false. Therefore, there is a fundamental flaw in the structure of my code. I would still like to use an Enum to refer to each spritesheet, as I feel like they look very neat and are easy to read and understand in code, and when I type "SpriteSheetName.", it will show up a list of the options, as an autocomplete tab, whereas if I used strings with a dictionary, this would not happen, and I could accidentally unknowingly make a spelling error. However, I cannot solely use the Enum class, as different keys with the same values are treated as the same, which is not what I want. So, I decided to make an extra dictionary to store the background colours and set each key in the Enum to a different number.

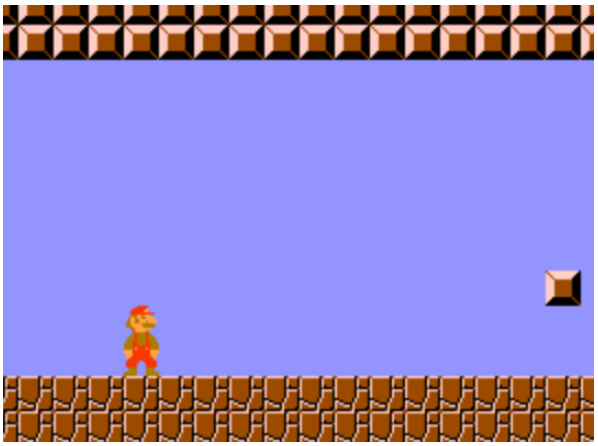
SPRITE_SHEETS.PY:

```
56  background_colours = {SpriteSheetName.MARIO: Colour.BLACK,
57                        SpriteSheetName.BLOCKS: Colour.BLOCKS_BACKGROUND,
58                        SpriteSheetName.INVISIBLE_WALL: Colour.BLACK}
```

```
27  class SpriteSheetName(Enum):
28      """This enum is originally used to help create the sprite sheets, and then used to refer to them. It stores the
29      parameters required for the SpriteSheet function initialisation. The key is the file name of the sprite sheet and
30      the value is the background colour for the sprite sheet. Each sprite entity will store a 'SpriteSheetName' enum to
31      refer to their sprite sheet."""
32      BLOCKS = auto()
33      INVISIBLE_WALL = auto()
34      MARIO = auto()
35
36      def get_filename(self):
37          return "img/" + self.name.lower() + ".png"
38
39      def get_background_col(self):
40          return background_colours[self]
```

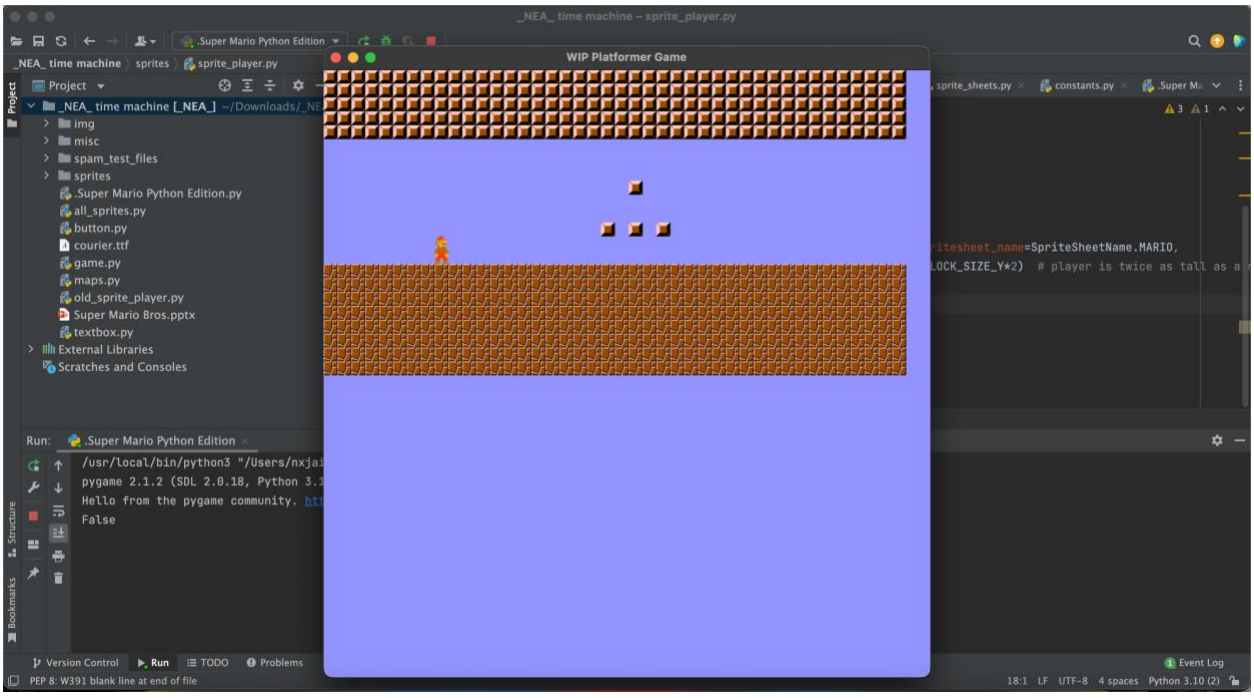
"auto()" is a simple function that automatically assigns a different number to each Enum value so that they are all unique. I have also changed the "get background col" function to utilise the dictionary to retrieve the relevant background colour. This should now hopefully work:

ATTEMPT 4:



All the expected outcomes have now finally been reached. Although the way to define a new spritesheet is a little more convoluted, it is still only a few extra lines, so overall there is not much issue.

I now have a slight issue with the program that I did not account for when designing this game:



This is (almost) my entire screen. The sprites I am using were big enough for a mini-Gameboy screen (or any other console used to play the original Mario bros) but for a big computer screen it's not quite as I would have hoped. I've added in an extra version to this development to try and resize everything without messing up the sprites I already have.

VERSION 2.6: CHANGING THE SIZE OF SPRITES

AIMS:

The objective in this version is to be able to change the size of all sprites by simply changing one constant variable. I don't want to have to go and change every single height, width, x, and y coordinate if I change the size once and then decide that the variables are too large. This seems like too small a task to give its own version, but I feel like a lot could potentially go wrong, especially with the "get specific sprite" function, so I gave it its own section in this report.

My original plan would be simply to create a constant in the constants file and equal it to some scalar multiplier value, say 2. Then I would go to the entity class and multiply everything by this scalar multiplier and modify anything that goes wrong.

CONSTANTS.PY

```
8       SCALE_UP = 2
```

ENTITY_SPRITE.PY: INSIDE THE "__INIT__" FUNCTION OF THE ENTITY SPRITE CLASS

```
self.rect.x = x * BLOCK_SIZE_X * SCALE_UP
self.rect.y = y * BLOCK_SIZE_Y * SCALE_UP
```

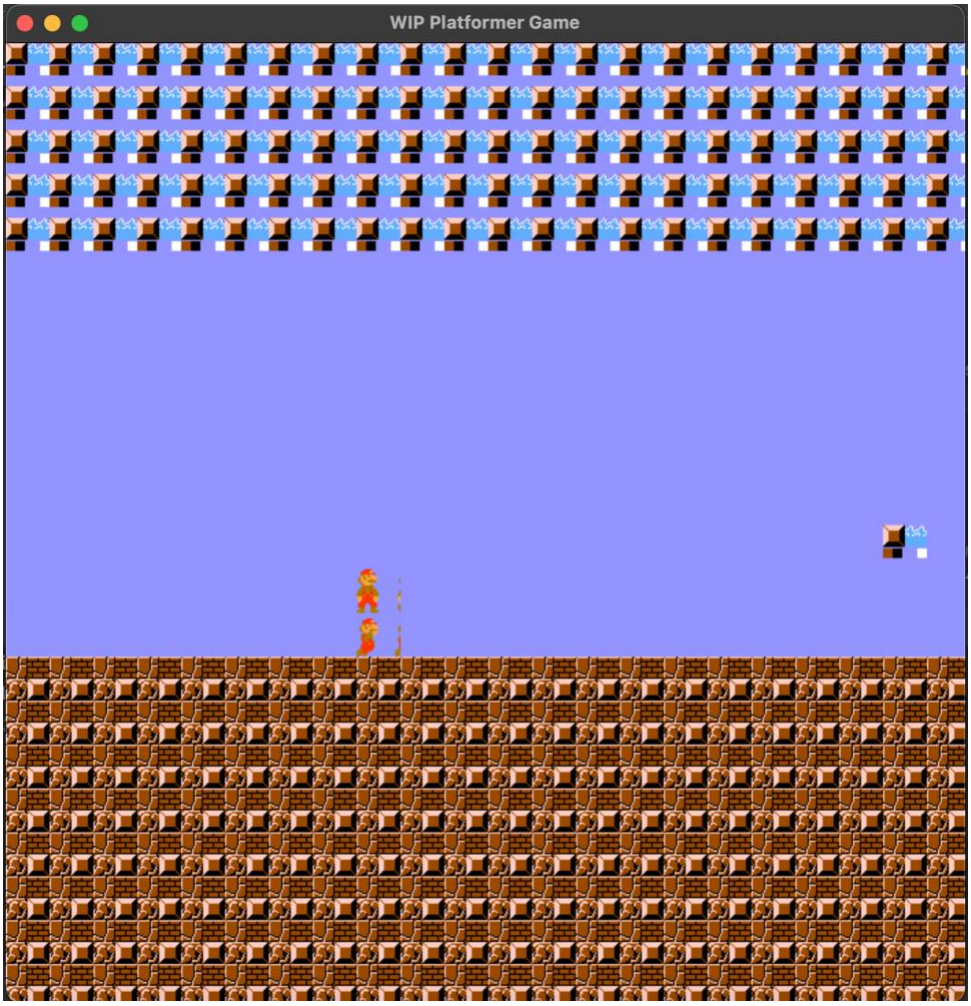
```
self.width = width * SCALE_UP
self.height = height * SCALE_UP
```

TESTING VERSION 2.6

Expected Outcome:

1. All the sprites have enlarged by a scale factor of 2
2. The sprites still maintain their same appearance from before.
3. The sprites will change by the scale factor of any value “SCALE_UP” is set to.

ATTEMPT 1:



So, on one hand, expected outcome 1 seems to have occurred, as the sprites now fill the entire screen. However, the sprites are all messed up. Each sprite seems to have the original desired image, but now also more of the spritesheet that I do not want. For example, if I take the ground tile:

Desired Image:	Actual Image:
	

SECTION OF BLOCKS SPRITESHEET:

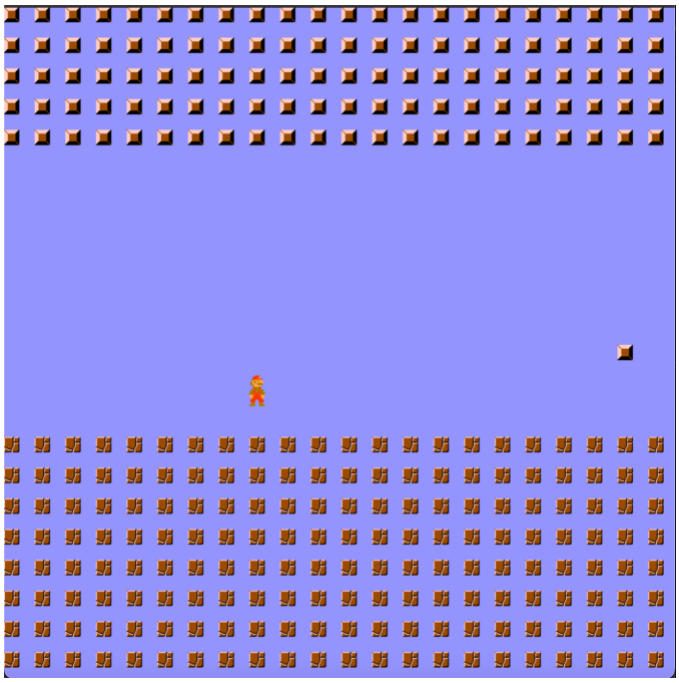
If we look at the spritesheet itself, it seems to have gotten the right top left corner, but then taken a 32x32 image instead of a 16x16 image, as the number of pixels selected has also scaled up. I still want the eventual sprite to be 32x32, but when I am selecting the specific image from the sprite sheet, I only want it to take the image that is 16x16, meaning that scaling up does not affect this process. The variables that are used to determine how many pixels are selected are "self.width" and "self.height", so I need to remove the scaling up from these.

ENTITY_SPRITE.PY: GET_SPRITE FUNCTION OF THE ENTITY SPRITE CLASS

```
self.width = width
self.height = height
```

Now the scaling up for the height and width have been removed.

ATTEMPT 2:



So, it's a little hard to tell here, but the sprites are technically all in the right place. And only the correct images have now been selected. But the images themselves are still the same size. If I check the size of, say one of the blocks:

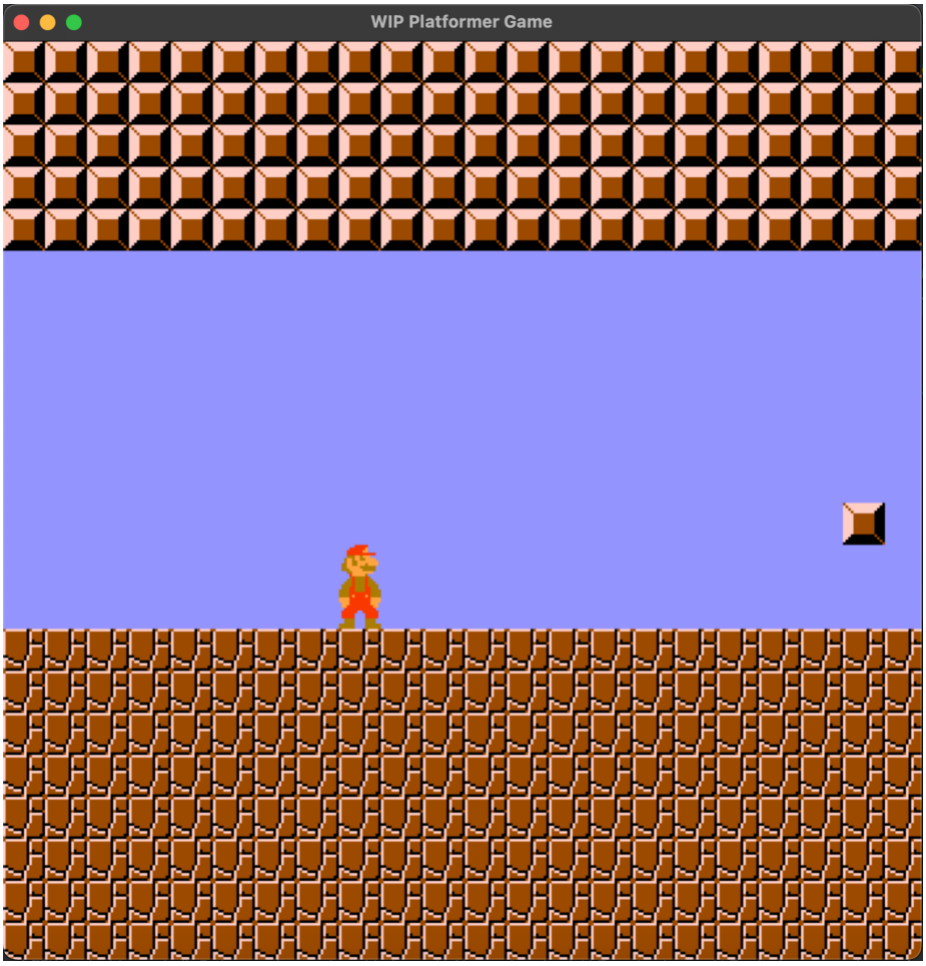
```
print(self.rect.height, self.rect.width)
```

This outputs 16x16. I need the rect to also be scaled up as these monitor the position of the sprite itself, and therefore are used to determine whether two sprites have collided. The rect is still 16x16, as the rect

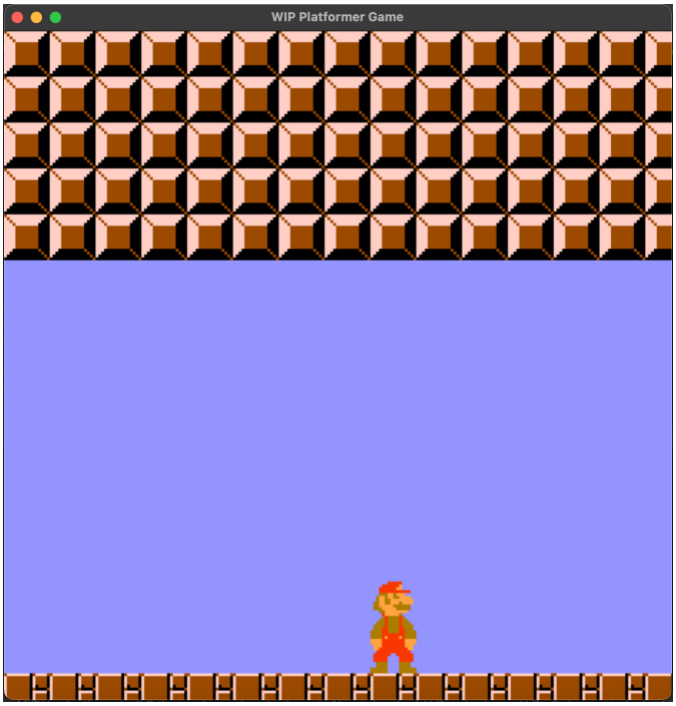
will be created using the dimensions of the image. Therefore, to hopefully solve all these problems, we need to scale up the image after it has been retrieved from the file, which should hopefully be done by this code here:

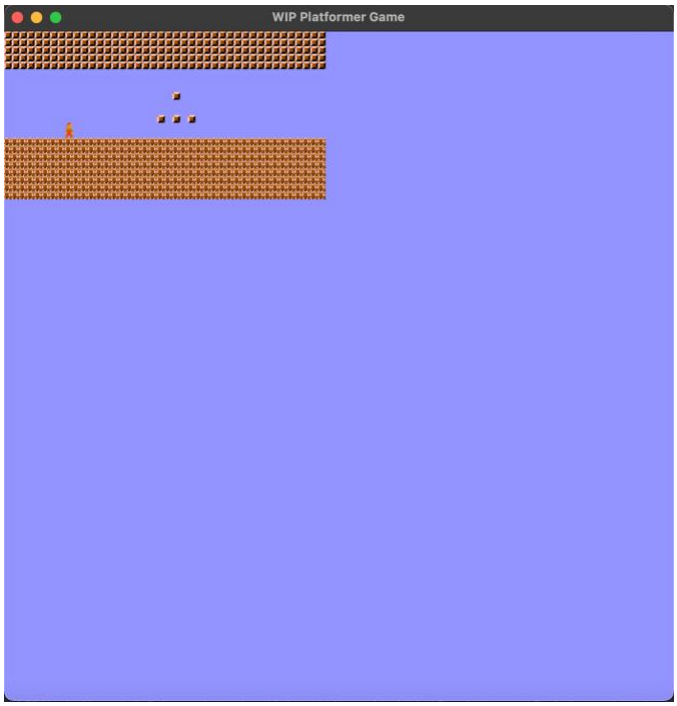
```
self.image = pygame.transform.scale(self.image, (width * SCALE_UP, height * SCALE_UP))
```

ATTEMPT 3:



Aims 1 and 2 have now been achieved. This is a pretty ideal size for all of these sprites, but just to make sure this all changes when “SCALE_UP” is changed, I’m going to change it to 3 and then 0.5 to check the last aim.





As you can see, all the sprites are still relatively in the same place, all the images have now been correctly selected and they're all the correct size. I'm going to move the scale up value back to 2 however, but it is good to see that a lot of the program can be changed correctly by just changing only one constant.

REVIEW OF DEVELOPMENT 2

This was a very critical development. I have now created infrastructure that allows me to easily derive new types of blocks, spritesheets, and then relatively easily put them on the map.

There were a few hiccups, especially with version 2.5. However, I am relatively confident that everything should work as intended and will allow for defining different entity classes of different sizes and appearances much easier in the future.

I will make a note for my future self to not make the same mistakes with default variables and Enum classes in python. Default variables are useful as they help reduce errors by ensuring that a variable is always passed in no matter what, however I've now come to the realisation that this promotes weak typing, which could lead to logic errors that are discovered much later. Enum classes are great for code organisation and clarity, however I must remember that I cannot use them if two keys have the same value, as they will be treated the same.

I am also happy that all the sprites in the program can be resized by the changing of one variable, as opposed to having to manually change many different variables with the potential errors, as it shows that my code is robust and is easy to tweak.

I have highlighted "resize everything using one variable" in this success criteria in a blue colour, as I have added it midway through the project.

However, now comes the beginning of most likely the most difficult part of the entire project. Programming the player's movement and creating collisions mechanic seemed like such a long process that I have split it into 2 parts, firstly in Version 3 I will tackle horizontal movement, whereas in Version 4 I will try to add vertical movement.

SUCCESS CRITERIA

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	
Create moving camera effect	
Animate moving functionality	
Add horizontal collisions	
Add vertical collisions	
Apply gravity to the player's movement	
Add a jump function and animate it	

Create parent enemy class	
Allow player to jump on enemies to kill them	
Create death function for player	
Ability to switch between two maps seamlessly.	
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 3: HORIZONTAL MOVEMENT FOR THE PLAYER

AIMS:

The objectives in the third development are to:

1. Add the ability to move left and right with the player.
2. Add a camera moving effect
3. Animate the moving functionality
4. Add horizontal collisions

VERSION 3.1: HORIZONTAL MOVEMENT

AIMS:

The objectives in this version are to:

1. Move the player sprite right when the right arrow key is pressed
2. Move the player sprite left when the left arrow key is pressed
3. Move the player sprite quicker when a “running key” is pressed.

For starters, the player is not going to be the only entity moving, enemies and powerups will as well. So, I’ve set a constant for the movement speed.

CONSTANTS.PY

```
WALKING_SPEED = 3
RUNNING_SPEED = 8
```

ANIMATIONS.PY: ENTITY_STATE ENUM

```
4   class EntityState(Enum):
5       """Describes which state the player is in to help the program decide which movement to undertake and which
6       animations to play"""
7       GROUNDED_LEFT = auto()
8       GROUNDED_RIGHT = auto()
```

I also need to create this little Enum function that will be expanded on later. This will also be used in animations, also with entities other than the player like enemies, so I have created an “animations.py” file and left it in there. It’s relevance to us now is to help the program figure out whether to move the sprite left or right. This will become clearer with its implementation shown later.

SPRITE_PLAYER.PY: "__INIT__" FUNCTION

```
class Player(Entity):
    """General class for Player/NPC"""
    def __init__(self, game, x, y):
        super().__init__(game, EntityGroups.PLAYER, x, y, EntityLayer.PERSON, spritesheet_name=SpriteSheetName.MARIO,
                         sprite_x=209, sprite_y=52, width=BLOCK_SIZE_X, height=BLOCK_SIZE_Y*2)
        # player is twice as tall as a regular block

        self.speed = WALKING_SPEED
        self.dx = 0
        self.player_state = EntityState.GROUNDED_RIGHT
```

I've initialised these variables for later use:

- Speed is the number of pixels that will be moved either left or right when the left or right key is pressed
- "dx" will store any changes to the x coordinate of the player.

SPRITE_PLAYER.PY: UPDATE FUNCTION

```
23  def update(self):
24      """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
25      self.detect_movement()
26      self.rect.x += self.dx  # Updates position of player
27      self.dx = 0
```

This function will be always run by the game class and will actually run any function to do with movement (and later jumping and collision functions). The detect movement function will potentially make a change to "dx", either setting it to the walking speed or running speed depending on whether the player is walking or running, and then making it negative if this movement is leftwards. This "dx" value will then be added onto the sprite position in line 26. If there was no movement, then "dx" would have stayed at 0 so no change happens to the sprite position. "dx" is then reset to 0 at the end of the function so that the player only moves when the key is pressed.

SPRITE_PLAYER.PY: DETECT_MOVEMENT FUNCTION

```
29  def detect_movement(self):
30      """Determines whether a movement is happening and potentially changes movement speed, the entity state and runs
31      the move() or jump() functions if the appropriate keys have been pressed."""
32      key_pressed = pygame.key.get_pressed()  # Creates huge array of each key and a boolean value of whether they
33      # were pressed
34
35      if key_pressed[pygame.K_c]:  # RUN
36          self.speed = RUNNING_SPEED
37      else:
38          self.speed = WALKING_SPEED
39
40      move_key_pressed = key_pressed[pygame.K_LEFT] or key_pressed[pygame.K_RIGHT]
41
42      if move_key_pressed:
43          if key_pressed[pygame.K_LEFT]:  # GROUNDED_LEFT
44              self.player_state = EntityState.GROUNDED_LEFT
45
46          elif key_pressed[pygame.K_RIGHT]:  # GROUNDED_RIGHT
47              self.player_state = EntityState.GROUNDED_RIGHT
48
49          self.move()
```

- Line 32 will detect which (if any) keys have been pressed.
- Lines 35-38 will detect whether the “c” key (the designated running button) has been pressed, and if so, will change the movement speed to the running speed.
- Line 40 detects whether a left or right arrow has been pressed.
- Lines 43-47 indicate a change in the player’s direction, which will be used in the move function.
- Line 49 will only run if a move key has been pressed

SPRITE_PLAYER.PY: MOVE FUNCTION

```
51      def move(self):
52          """Actually changes dx value"""
53          coordinate_change = self.speed
54
55          if self.player_state == EntityState.GROUNDED_LEFT:
56              coordinate_change *= -1  # Makes number negative if the person's moving left
57
58          setattr(self, "dx", coordinate_change)  # Changes dx value
```

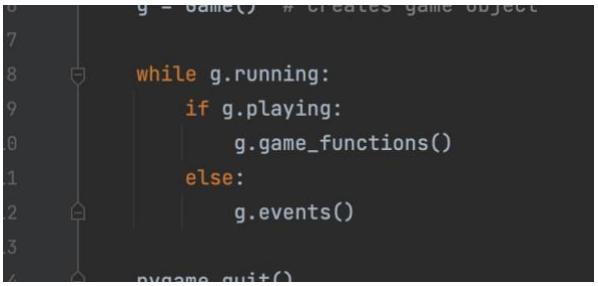
This should be relatively self-explanatory. As explained above, once “dx” changes, this will be added to the player’s “self.rect.x” value, meaning the player sprite will move.

GAME.PY: UPDATE AND “GAME_FUNCTIONS” FUNCTION

```
42      def events(self):
43          """To always run no matter what to catch events like closing the program which are only specific to the game as
44          a whole"""
45          for event in pygame.event.get():
46              if event.type == pygame.QUIT:
47                  self.playing = self.running = False
48
49      def update(self):
50          """Runs all update functions for all sprites which updates them during the game.
51          Only used while 'self.playing = True'."""
52          self.all_sprites.update()
53
54      def game_functions(self):
55          """Runs 'events' and 'update' functions which need to be run every second to update the screen and catch
56          hold of events like closing the program which are only specific to the game as a whole"""
57          self.events()
58          self.update()
```

All sprites in pygame have a built-in update function, although I’m not sure of its purpose. But I need to always check for updates to these sprites, hence the making of the update function, and then made a function to always run while the game is playing (game functions). These functions have purely been created for code organisation and clarity reasons, as they have just collated or shortened other functions.

MAIN GAME FILE:



```
g = Game() # creates game object

while g.running:
    if g.playing:
        g.game_functions()
    else:
        g.events()

pygame.quit()
```

This is how the update and events functions will always be running. I need the events function to always run as no matter what the game state is in, it is always needed to be able to detect when the program wants to be closed and quit. However, I haven't allowed update to be run when the game is not being played, as when the game is not being played the sprites have not been created, so there is nothing to update (as "all sprites" will be empty)

TESTING VERSION 3.1

Expected Outcome:

1. The player sprite moves right when the right arrow key is pressed
2. The player sprite moves left when the left arrow key is pressed
3. The player sprite moves quicker when the “c” key is pressed.

ATTEMPT 1:

I’ve added a block to insinuate Mario’s starting position, so that when Mario does move, the player sprite will not be directly below the block, and you can see that he has moved. Unfortunately, despite the keys being pressed, Mario is not moving.

My first thought would be to check if the rect position is actually changing due to key presses. I added a little debug function in the “detect movement” function to print the rect position of the sprite whenever the character “p” is pressed. If nothing is printed, then either the “detect movement” function did not run, or the key detection is not working.

```
if key_pressed[pygame.K_p]:
    print(self.rect.x)
```

→

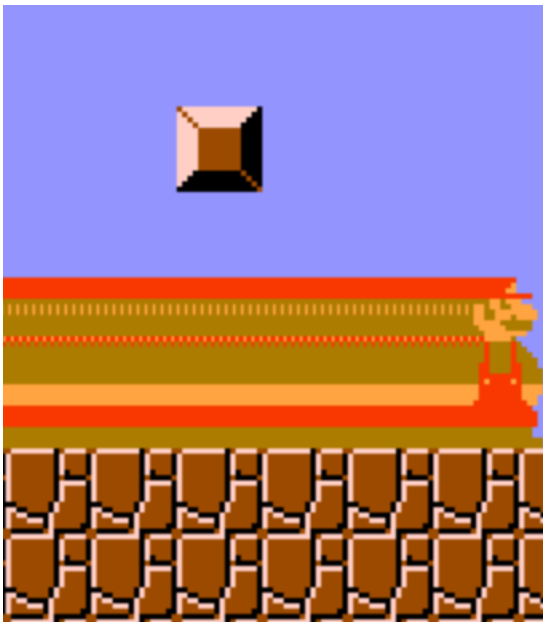
```
256
1285
```

The information in the green box is the terminal output. The x value is indeed changing, the image is just simply not being updated.

I had a potential idea about the draw function needing to be run every time an update happened, as opposed to just once. I ran it after every move function to see whether this was the case.

```
setattr(self, "dx", coordinate_change)  # cha

self.game.all_sprites.draw(self.game.screen)
pygame.display.update()
```

ATTEMPT 2:



Well movement has happened, but it has not removed the previous iterations of the sprite. I looked at this for a long time, and I could not figure out a way to improve this, without doing something which is slightly processor intensive. The only way to remove a sprite on pygame is either to use the “kill” function it or to display something over it. However, the first option removes the object and requires me to recreate it again, meaning I must then go through the whole “__init__” function again. This sounds more strenuous on the computer than displaying everything on the page every second. This still sounds bad in terms of the number of functions and processes being run all the time, but this is still a very small and simple program, where going through a process like that shouldn’t be a problem for almost all computers. I would still prefer if I could find a less processor intensive solution, but I don’t want to dwell on this too long for now, and would rather find a solution that works, finish the rest of the program, and see if I can find an alternate solution at the end.

Anyways, the new way to do this would be to create the following “draw” function, and add it to “game functions” in the game class, so that it is always run:

GAME.PY: UPDATE AND “GAME_FUNCTIONS” FUNCTION

```
55      def draw(self):
56          """Draws all the sprites onto the screen, which has to be done every tick.
57          Only used while 'self.playing = True'."""
58          self.screen.fill(self.background_colour)
59          self.all_sprites.draw(self.screen)
60          self.clock.tick(FPS)
61          pygame.display.update()
62
63      def game_functions(self):
64          """Runs 'events', 'update' and 'draw' functions which need to be run every second to update the screen and catch
65          hold of events like closing the program which are only specific to the game as a whole"""
66          self.events()
67          self.update()
68          self.draw()
```

The draw function will not be run in the intro screen, as there are no sprites and there is no need for a clock to tick.

It should also be noted that the movement was incredibly quick. While loops and for loops are obviously run at very high speeds, so one press of the button sent Mario travelling very quickly. If you look at the difference in x values shown in testing attempt 1, there was a difference of over 1000 pixels just from one press of the right arrow. Hopefully, with line 60, this high speed is reduced significantly.

ATTEMPT 3:

Left Key Pressed:	Original Position	Right Key Pressed:
		

As you can see, when the left key is pressed, the sprite moves left, but if the right key is pressed, the sprite moves right

X values of sprite when walking	X values of sprite when running
	

These are the x values of the sprite when walking right and running right. The left one increments by 3 whereas the right one increments by 8 every tick. The aims have been achieved.

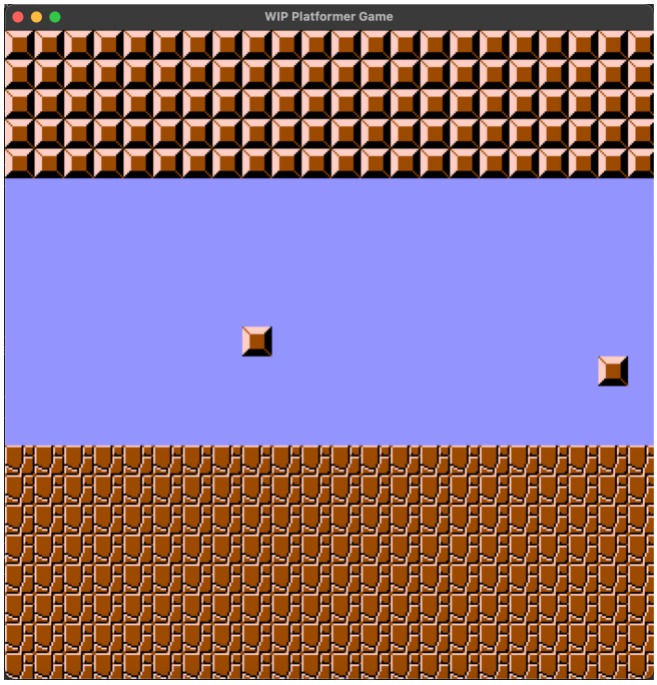
VERSION 3.2: CAMERA MOVEMENT

AIMS:

The objectives in this version are to:

1. Let the camera follow the player when the player moves.
2. Make sure that the player stays centred when the camera moves.
3. Make sure that are still centred when the window is resized

Now the next problem is this. If the character moves too far to the left or right, then they simply just go off screen:



Not only do I want it to be possible to always see the character, but I also want to be able to create a level that is far bigger than one screen. Therefore, I want the player to be always in the centre of the screen, and for this to hold true when the screen is resized.

However, my first problem is that pygame doesn't have any mechanism for the camera to follow the player. So, I must get creative. My solution is to:

- Move every sprite on screen in the opposite direction whenever the player moves.
- While this is happening, say the player will be moving forwards by 3 pixels
    - However, simultaneously the player will be moved backward 3 pixels by a "move sprites" function.
    - Therefore, in total the player is technically moving 0 pixels and the player stays in the centre.
- However, every other sprite has moved backwards 3 pixels, which gives the illusion that the player has moved forwards while the camera has followed them.

This seems slightly processor intensive, but once again, with such great computers, this should not actually be a problem (I also don't really have any other option). To do this, we first need a function to move the sprites:

GAME.PY: "MOVE_SPRITES" FUNCTION

```
58    def move_sprites(self, x, y):
59        """Moves all sprites (used for camera movement)"""
60        for sprite in self.all_sprites:
61            sprite.rect.x += x
62            sprite.rect.y += y
```

I put it in the game function as this has access to all the sprites. Now we need to actually run this function when the player moves.

SPRITE_PLAYER.PY: MOVE FUNCTION:

```
54    def move(self):
55        """Actually changes dx value"""
56        coordinate_change = self.speed
57
58        if self.player_state == EntityState.GROUNDED_LEFT:
59            coordinate_change *= -1  # Makes number negative if the person's moving left
60
61        # Camera Movement:
62        self.game.move_sprites(-coordinate_change, 0)  # minus is there to move the objects in opposite direction
63
64        setattr(self, "dx", coordinate_change)  # Changes dx value
```

This should be relatively straightforward to understand. This all now means that the player will stay in the centre while everything else moves around it.

However, if the window is resized, then the character could go off screen, as none of the sprites are moving. So, if the character is in x position 209, but the screen now only shows from 0 to 200, then the player will be off screen. I need to change this:

GAME.PY: EVENTS FUNCTION

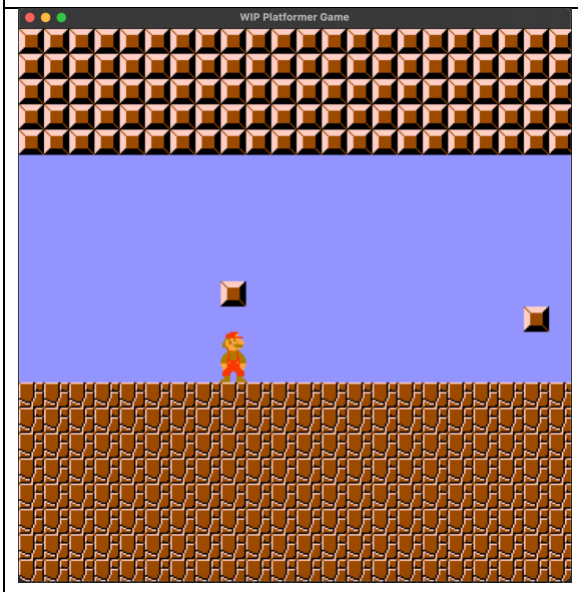
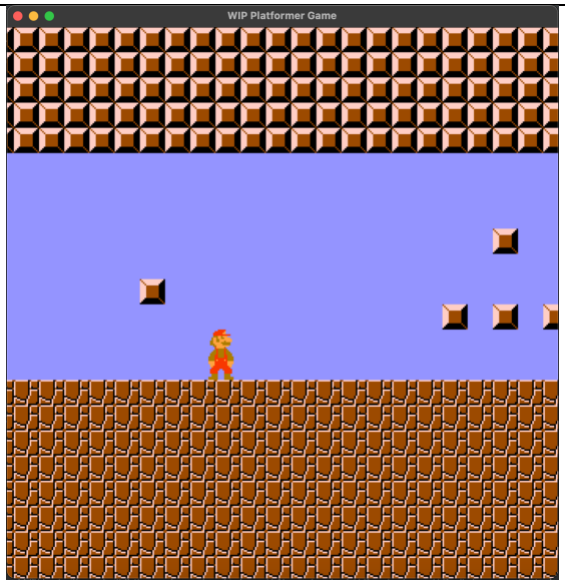
```
78    def events(self):
79        """To always run no matter what to catch events like closing the program or resizing the screen which are only
80        specific to the game as a whole"""
81        for event in pygame.event.get():
82            if event.type == pygame.QUIT:
83                self.playing = self.running = False
84            elif event.type == pygame.VIDEORESIZE:  # Moves camera so everything is in the middle.
85                dx = self.screen.get_size()[0] - self.size[0]  # self.screen.get_size()[0] is the new width,
86                # self.size[0] is the old width
87                dy = self.screen.get_size()[1] - self.size[1]  # same here but with height
88
89                self.size = self.screen.get_size()  # sets the size variable equal to the current size.
90                self.move_sprites(dx / 2, dy / 2)
```

TESTING VERSION 3.2

Expected Outcome:

1. The camera follows the player when the player moves.
2. The player stays centred when the camera moves.
3. Everything is still centred when the window is resized

ATTEMPT 1:

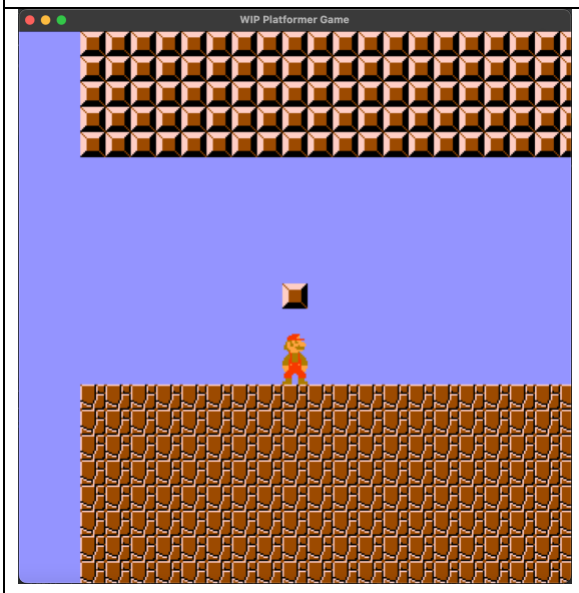
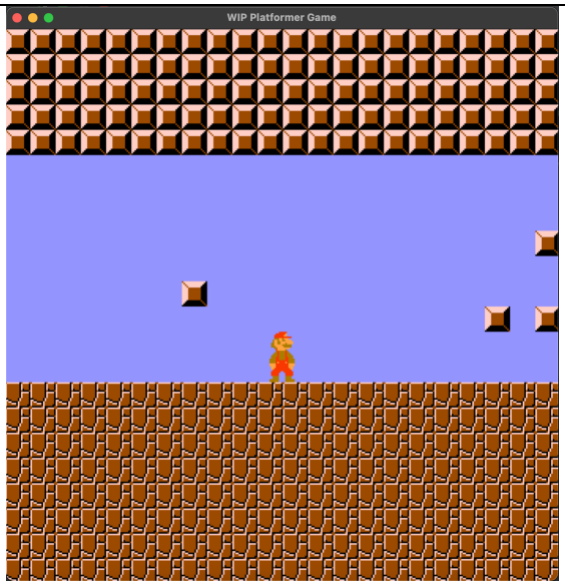
Before Movement	After Movement
	

The player will stay in the centre while everything else moves around it. However, if the player does not start in the centre, then it will never be in the centre and will stay wherever it started the map at. You can see that the "camera has moved", allowing you to see more of the block formation to the right, but the player sprite is still not centred, it's slightly to the left. I could always make sure I spawn the player in the middle every time, but this seems like a brute force method that will only lead to more problems. I want this code to be more flexible, and to make it possible for Mario to be centred even if he enters from a door on the edge of the map. Therefore, I need to create a player centring function that I run as the map is created:

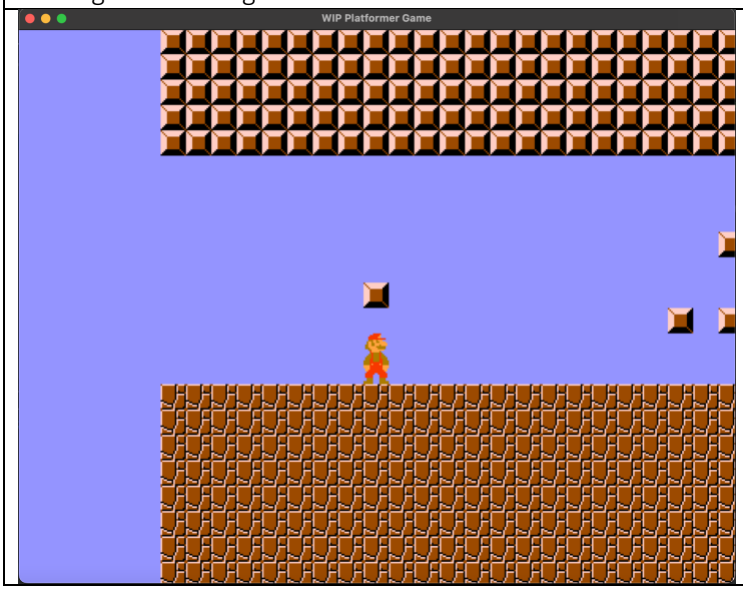
GAME.PY: CENTRE_CHARACTER

```
119     def centre_character(self):
120         """Run whenever a map is newly loaded to centre the character in the camera view"""
121         player = [_ for _ in self.player][0]  # retrieves player sprite
122 
123         desired_x = self.size[0] // 2 - (player.rect.width / SCALE_UP)  # I only want to change x as I don't want
124         # vertical camera movement, only lateral camera movement.
125 
126         # Moves all sprites accordingly until player is in right place
127         while player.rect.x != desired_x:
128             if player.rect.x < desired_x:
129                 self.move_sprites(1, 0)
130             else:
131                 self.move_sprites(-1, 0)
```

ATTEMPT 2:

Before Movement	After Movement
	

I will have to add some blocks to the right to ensure that Mario cannot fall off, but for now the character is now centred. If I try to resize the screen:

Making window larger	Making window smaller
	

The character is still centred. The expected outcomes have been achieved.

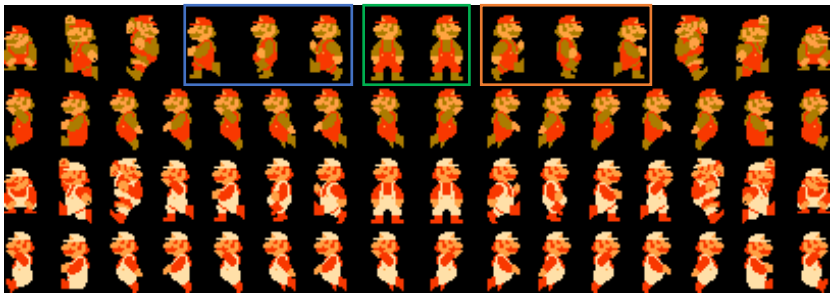
VERSION 3.3: ANIMATE MOVING FUNCTIONALITY

AIMS:

The objectives in this version are to:

1. Import the animation sprites using lists of x and y coordinates
2. The player is animated when walking left and right

So right now, we have a moving sprite. However, when it moves, Mario is stuck looking right, regardless. I want the main character to be able to move left and right and look like they're walking left and right. If you go back to the Mario spritesheet, you'll see that the sprites are there to give a walking animation, they just need to be played in a loop.



The ones in the blue/orange box are the ones I want to use when Mario is walking left/right. The ones in green are the two when he is stationary and either looking left or right. There are other sprites here, but a reminder that I will be using a different sprite sheet later, and I only want to create a base for where it is easy for me to import that spritesheet without too many complications.

ANIMATIONS.PY: ANIMATIONS ENUM

```
class Animations(Enum):
    """Stores the x,y coordinates for the sprites. For the grounded sprites, the second item in the array contains the
    walking animations to create a loop."""
    MAIN_CHARACTER = {EntityState.GROUNDED_LEFT: [[180, 52], [[90, 51], [120, 52], [150, 51]]],
                      EntityState.GROUNDED_RIGHT: [[209, 52], [[299, 51], [269, 52], [239, 51]]]}
```

This Enum will store a dictionary which will be loaded into each character. Depending on what state the player is in, there will be different animations played.

The values to each entity state may seem like a horrible amalgamation of numbers in a 3d list, but the idea is that:

1. Each list of 2 numbers contains an [x, y] coordinate like that.
2. These will each be fed into the "get sprite" function to return a sprite object.
3. The list will then look like [A, [B, C, D]], where the letters are different sprite objects.
4. A will be the sprite when the character is standing. [B, C, D] will be the loop of sprites played to create a walking animation.

This way, all you have to do is enter the x and y coordinate to add new animations/still sprites. To enact step 2 in that list of steps, I need to create a few slightly difficult to explain functions.

SPRITE_PLAYER.PY: PROCESSING ANIMATION FUNCTIONS

```
70      # ____________________________________ANIMATION FUNCTIONS____________________________________
71      def process_animations(self, animations_enum: Animations):
72          """Takes list of coordinates for a spritesheet and returns pygame sprites"""
73          animations = {}
74
75          for player_state in animations_enum.value:
76              animation_list = self.get_animation_sprites(animations_enum.value[player_state])
77              animations[player_state] = animation_list
78
79          return animations
```

```
81      def get_animation_sprites(self, coordinates_list: list):
82          """Helper function for process animations: takes each list and applies get_sprite function onto each x,y
83          coordinate pairs and returns the list in the same format and size as it was before"""
84          new_list = []
85          x_coord = None
86
87          for item in range(len(coordinates_list)):
88              if type(coordinates_list[item]) is list:  # If we've encountered a pair of coordinates or more, as opposed
89                  # to just a single coordinate.
90                  new_list.append(self.get_animation_sprites(coordinates_list[item]))  # Recursively calls algorithm until
91                  # the for loop is iterating through a pair of coordinates and not a multidimensional list
92
93              else:  # This is when the algorithm is iterating through a pair of coordinates and coordinates_list[item] is
94                  # an x/y coordinate
95                  if x_coord is None:  # If this is true then coordinates_list[item] = the x coordinate.
96                      x_coord = coordinates_list[item]
97                  else:  # Otherwise, coordinates_list[item] = the y coordinate.
98                      new_list.append(self.get_sprite(x_coord, coordinates_list[item]))
99          if len(new_list) == 1:  # This is during the recursive stage when new_list = [SurfaceObj], but I want the
100             # SurfaceObj by itself so the next line takes it out of the list.
101             new_list = new_list[0]
102         return new_list
```

“get animation sprites” is used to turn those complicated looking multidimensions lists of image coordinates into less complicated lists of sprites themselves. It’s a little hard to explain but it is a recursive algorithm, and I tried my best to explain using the comments. But essentially a list like [[180, 52], [[90, 51], [120, 52], [150, 51]]] turns into [A, [B, C, D]], where the letters are pygame surface objects. Process animations then creates a dictionary and reassigns each entity state with their respective animation loops.

SPRITE_PLAYER.PY: IN THE “__INIT__” FUNCTION

```
23              self.animations = self.process_animations(Animations.MAIN_CHARACTER)
24              self.animation_loop = 1
```

All this is stored in an attribute “Animations”. “Animation loop” is used to help the animating process, which I will show now:

SPRITE_PLAYER.PY: ANIMATE FUNCTION:

```
103     def animate(self):
104         """Selects sprites to display for whatever the player is doing. If moving there is an animation loop played"""
105         animations = self.animations[self.player_state]
106 
107         standing_still = (self.dx == 0)
108 
109         if standing_still:
110             self.image = animations[0]
111 
112         else:  # Plays walking/running animation sprites in a loop.
113             self.image = animations[1][floor(self.animation_loop)]
114             self.animation_loop += 0.1
115             if self.animation_loop >= 3:
116                 self.animation_loop = 0
117 
118         self.image = pygame.transform.scale(self.image, (16 * SCALE_UP, 32 * SCALE_UP))  # As these images have not
119         # been scaled
```

- Line 105 is why storing the current player's state is useful

Now, to run this function constantly I need to add it to the update function.

TESTING VERSION 3.3

Expected Outcome:

1. The player has the standing still sprite when stationary
2. The player is looking in the left/right direction when the left/right key is pressed.
3. The player is correctly animated when walking left and right

[Check the "Testing – Version 3.3" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION 3.4: HORIZONTAL COLLISIONS

AIMS:

The objectives in this version are to:

1. Detect when a horizontal collision happens between the player and a block
2. Stop the player walking through a block:

Right now, if the player sprite walks into a block, it'll walk right through it. This is obviously not the desired result. I need to make the player stop before it walks through the block, so that the sprites never overlap like they do in this image. Pygame offers some help to this, as if the "rect" attributes of each object overlap or touch then it detects it. I can use this to help create the desired collision mechanic:

SPRITE_PLAYER: X_COLLIDE FUNCTION:

```
122     # ____________________________________________COLLISION FUNCTIONS_____________________________________________
123     def x_collide(self):
124         """Detects a horizontal collision and acts on it to stop player going through objects/entities."""
125         hit_block = pygame.sprite.spritecollide(self, self.game.blocks, False)
126         # hit_block[0] will refer to the block hit
127 
128         # If collision is horizontal
129         if hit_block:
130             if self.dx > 0:
131                 self.rect.x = hit_block[0].rect.left - self.rect.width  # Moves char just to the left of the other object
132 
133             elif self.dx < 0:
134                 self.rect.x = hit_block[0].rect.right  # Moves char just to the right of the other object
```

- Line 125 detects whether the player sprite has collided with a block. This is why the groups mechanic is very useful: "self.blocks" will include all block type tiles, pipes, doors, and others.
- Line 131/134: Makes sure the x coordinate of the player sprite is just to the left of the block/right of the block.

As usual, I need to add this to the update function so that it runs:

SPRITE_PLAYER: UPDATE FUNCTION.

```
26      def update(self):
27          """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick""
28          self.detect_movement()
29          self.animate()
30 
31          self.rect.x += self.dx  # Updates position of player
32          self.x_collide()
33 
34          self.dx = 0
```

It needs to be after line 31, as if the player is walking right, then dx is going to have a positive value, so the "x_collide" function needs to quickly reset "rect.x" back to the value that places the player to the left of the block (and vice versa).

TESTING VERSION 3.4

Expected Outcome:

1. When walking right, into a block, the player stops right before the block and is unable to walk through the block.
2. When walking left, into a block, the player stops right before the block and is unable to walk through the block.

ATTEMPT 1:

[Check the "Testing – Version 3.4 A1" video to see the results]

Although technically the expected outcomes have occurred, a slightly unexpected error has come about. It was a slight oversight, however, although the collide function stops the player from moving any further, the move function still registers the player as attempting to move, and therefore the camera moves in the opposite direction to try and keep the player in the centre of the screen (as achieved before in version 3.2). As collide runs after the movement function in the update function, my only solution to this is to move every single sprite in the opposite direction they're trying to be moved in, to cancel out the original "camera" movement and to keep the sprites in the same place:

SPRITE_PLAYER: INSIDE X_COLLIDE FUNCTION:

```
if hit_block:
    if self.dx > 0:
        self.rect.x = hit_block[0].rect.left - self.rect.width  # Moves char just to the left of the other object

        # To Reverse Camera Movement:
        self.game.move_sprites(self.speed, 0)

    elif self.dx < 0:
        self.rect.x = hit_block[0].rect.right  # Moves char just to the right of the other object

        # To Reverse Camera Movement:
        self.game.move_sprites(-self.speed, 0)
```

Expected Outcome:

1. When walking right, into a block, the player stops right before the block and is unable to walk through the block.
2. When walking left, into a block, the player stops right before the block and is unable to walk through the block.
3. The camera stays still when the sprite collides with a block

ATTEMPT 2:

[Check the "Testing – Version 3.4 A2" video to see the results]

This change fixed the problem. The camera is now still when the player collides with a block. All the expected outcomes have occurred. The aims for this version have been achieved.

REVIEW OF DEVELOPMENT 3

This section of the development was a very challenging part of the project. It was much easier to implement than I originally thought however, which I must attribute to good designs made beforehand. The occasional oversight did occur; however, I feel as though overall I am pleased with how smoothly the implementation occurred.

There were a few processor-intensive tasks, for example the camera movement, the cancelling of the camera movement and the fact that I must draw the background and the sprite at every second, however, I'm not sure if there is another way to achieve the same things. The program has technically not slowed down as it is still a relatively simple program, so if I make no changes to any of these processes then it isn't the biggest issue. I am not sure whether I am fully pleased with my solution to the things I mentioned, so if I have time at the end, I will look at potentially improving those and making them more efficient and less processor intensive.

However regardless, I am pleased with the fact that everything functionally works, and I have created simple ways to import new animations and apply them to the program, for when I use different sprite sheets. My aim is also to create a flexible program where it is easy to adapt and add new content, so although I am not using the correct sprite sheets right now, I will then hopefully be able to integrate the correct ones later on, without many issues, which will test these aspects of my program.

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	
Apply gravity to the player's movement	
Add a jump function and animate it	
Create parent enemy class	

Allow player to jump on enemies to kill them	
Create death function for player	
Ability to switch between two maps seamlessly.	
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 4: VERTICAL MOVEMENT FOR THE PLAYER

AIMS:

The objectives in the third development are to:

1. Add vertical collisions
2. Add gravity effect to the player's movement
3. Add the ability to jump and animate it

VERSION: 4.1: VERTICAL COLLISIONS

AIMS:

The objectives in this version are to:

1. Detect when a vertical collision happens between the player and a block
2. Stop the player falling through a block (create an expendable function to test this)

Now that we've dealt with horizontal collisions, it's time to deal with vertical collisions. Although there is no way to move in a vertical direction right now, I will need this to work to then be able to test out and create gravity which then all allows me to create a successful jump function.

As I have already created a horizontal collision function, a vertical one should not be too difficult. I will need to modify this function when I am creating jumps, but for now I at least want the player to stop when it collides with a block vertically, so that it does not fall through:

SPRITE_PLAYER.PY: INSIDE "__INIT__" FUNCTION:

```
19          self.speed = WALKING_SPEE
20          self.dx = self.dy = 0
21          self.player_state = Entit
```

Firstly, I will need to initialise a "dy" attribute, to deal with changes to the y attribute, like what we had with the x attribute. This value will be later changed by the jump function and gravity function.

SPRITE_PLAYER: Y_COLLIDE FUNCTION:

```
148     def y_collide(self):
149         """Detects a vertical collision and acts on it to stop player going through objects/entities."""
150         hit_block = pygame.sprite.spritecollide(self, self.game.blocks, False)
151 
152         if hit_block:
153             if self.dy > 0:
154                 self.rect.y = hit_block[0].rect.top - self.rect.height  # Moves char just below the other object
155 
156             elif self.dy < 0:
157                 self.rect.y = hit_block[0].rect.bottom  # Moves char just above the other object
```

This is very similar to the “x_collide” function and should be relatively self-explanatory.

SPRITE_PLAYER: UPDATE FUNCTION:

```
26     def update(self):
27         """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
28         self.detect_movement()
29         self.animate()
30 
31         self.rect.x += self.dx  # Updates position of player
32         self.x_collide()
33 
34         self.rect.y += self.dy
35         self.y_collide()
36 
37         self.dx = self.dy = 0
```

I need to mirror what I did with the x coordinate with the y coordinate:

1. Every change to y, which is stored in “dy”, must be added to the y position of the sprite every update function
2. The “y_collide” function is then run to ensure that the player sprite has not travelled too far and has not overlapped with a block in a vertical direction, by checking to see if it has collided with a block (in the vertical direction)
3. The dy attribute must be reset to 0 to ensure that the same change is not added to the y attribute in the next update.

However, as I said before, “dy” will only store any changes to y, which only happens with the jump function and gravity function. I have not created those yet, so I need a quick bit of code to test this version that I can just remove later:

SPRITE_PLAYER.PY: INSIDE “DETECT_MOVEMENT” FUNCTION

```
53         if key_pressed[pygame.K_UP]:
54             self.dy -= self.speed
55 
56         elif key_pressed[pygame.K_DOWN]:
57             self.dy += self.speed
```

TESTING VERSION 4.1

Expected Outcome:

1. When moving up into a block, the player stops right before the block and is unable to move through the block.
2. When moving down into a block, the player stops right before the block and is unable to move through the block.
3. When moving into a block at an angle (both horizontal and vertical movement), the player stops right before the block and is unable to move through the block.

ATTEMPT 1:

[Check the "Testing – Version 4.1 A1" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION: 4.2: ADD GRAVITY

AIMS:

The objectives in this version are to:

1. Add a gravity function to always make the player fall when the player is in mid-air.
2. Ensure that this works with the vertical collisions created and tested in version 4.1

This should be a relatively straight forward version, as the function I am about to create should become more in depth in the next version once I add jumping. However, I cannot add jumps until there is also a mechanism that allows the player to naturally fall.

To add this mechanic, I need to know when the player is in the air and when the player is not in the air. I have created a flag attribute called "is on ground" to do this. This attribute is set to true when the player has collided with a block in a downwards direction (i.e., when the player lands on a block, and when "dy" is greater than 0), and false when the player has collided with a block in an upwards direction (i.e., when "dy" is smaller than 0). It will also be set false when the player jumps, but that is a problem for the next version. After implementing this, I simply need to add a value to "dy" when the player is not on the ground.

SPRITE_PLAYER.PY: INSIDE "__INIT__" FUNCTION, INITIALISING FLAG ATTRIBUTE:

```
self.is_on_ground = False
```

SPRITE_PLAYER.PY: Y_COLLIDE FUNCTION:

```
157     def y_collide(self):
158         """Detects a vertical collision and acts on it to stop player going through objects/entities."""
159         hit_block = pygame.sprite.spritecollide(self, self.game.blocks, False)
160 
161         if hit_block:
162             if self.dy > 0:
163                 self.rect.y = hit_block[0].rect.top - self.rect.height  # Moves char just below the other object
164                 self.is_on_ground = True
165 
166             elif self.dy < 0:
167                 self.rect.y = hit_block[0].rect.bottom  # Moves char just above the other object
168                 self.is_on_ground = False
```

SPRITE_PLAYER.PY: GRAVITY FUNCTION:

```
def gravity(self):
    """Adds 5 to the y position if the player is in the air (like actual gravity)"""
    if not self.is_on_ground:
        self.dy = 5
```

SPRITE_PLAYER.PY: UPDATE FUNCTION:

```
def update(self):
    """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
    self.detect_movement()
    self.animate()
    self.gravity()

    self.rect.x += self.dx  # Updates position of player
    self.x_collide()

    self.rect.y += self.dy
    self.y_collide()

    self.dx = self.dy = 0
```

TESTING VERSION 4.2

Expected Outcome:

1. The player falls immediately on start-up.
2. The player stands on the ground and doesn't fall through (collision mechanic from 4.1 still works)
3. If the player is moved up and hits a ceiling the player falls to the ground. (Doesn't have to fall if it's in mid-air as right now the flag is only set to false when it collides upwards).

ATTEMPT 1:

[Check the "Testing – Version 4.2 A1" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION: 4.3: ADD A JUMP FUNCTION

AIMS:

The objective in this version is to create a successful jump function. There are many things to consider in this jump function:

1. A jump must last through multiple update functions, as it is too long of an event to last just one.
    i. I will create a "jump counter", which will start at some value when a jump has begun, and slowly decrease every update function, until it reaches 0, when the jump ends.
2. A jump must only be possible if on the ground.
    i. This can be done using the "is on ground" attribute.
3. The gravity function should not run when the player is jumping:
    i. Add an extra check to make sure the jump counter is equal to 0
4. A jump should cancel when hitting a ceiling.
    i. The "jump counter" can be set to 0 when an upwards collision happens
5. The jump speed is quicker and slows down when the jump reaches its peak.
    i. The jump speed can be based on the jump counter, which will be slowly decreasing over the course of the jump anyways.
6. The jumps have different animations that need to be imported and played at the right time.
    i. I am going to add entity states for when the player is jumping left and right.
    ii. I will also add a flag to see if the character is jumping, to differentiate between jumping and standing states, and make it easier to switch between the two animations:

To put this all into code:

SPRITE_PLAYER.PY: INSIDE "__INIT__" FUNCTION: INITIALISE NEW ATTRIBUTES:

```
self.jump_counter = 0  # Keeps track of time length of jump, allows jump to last longer than the time taken to
# run update once
self.is_jumping = False
```

ANIMATIONS.PY:

```
class EntityState(Enum):
    """Describes which state the player is in to help the program decide which animations to play""
    GROUNDED_LEFT = auto()
    GROUNDED_RIGHT = auto()

    JUMP_LEFT = auto()
    JUMP_RIGHT = auto()
```

```
class Animations(Enum):
    MAIN_CHARACTER = {EntityState.GROUNDED_LEFT: [[180, 52], [[90, 51], [120, 52], [150, 51]]],
                      EntityState.GROUNDED_RIGHT: [[209, 52], [[299, 51], [269, 52], [239, 51]]],
                      EntityState.JUMP_RIGHT: [359, 52],
                      EntityState.JUMP_LEFT: [30, 52]}
```

Now I've created these entity states, but I need to make it possible to switch between the standing/jumping states. I will do this in the jump function and the "y_collide" function:

SPRITE_PLAYER.PY: "Y_COLLIDE" FUNCTION:

```
def y_collide(self):
    """Detects a vertical collision and acts on it to stop player going through objects/entities."""
    hit_block = pygame.sprite.spritecollide(self, self.game.blocks, False)

    if hit_block:
        if self.dy > 0:  # Player hits ground
            if self.player_state == EntityState.JUMP_RIGHT:
                self.player_state = EntityState.GROUNDED_RIGHT
            elif self.player_state == EntityState.JUMP_LEFT:
                self.player_state = EntityState.GROUNDED_LEFT

            self.rect.y = hit_block[0].rect.top - self.rect.height  # Moves char just below the other object
            self.is_on_ground = True

        elif self.dy < 0:  # Upwards collision, player hits ceiling
            self.rect.y = hit_block[0].rect.bottom  # Moves char just above the other object
            self.is_on_ground = False
            self.jump_counter = 0  # cancels jump
```

- When the player hits the ground there will be a switch from a jumping state to a standing state.
- You can also see that the jump will be cancelled when the player hits the ceiling on the last line.

SPRITE_PLAYER.PY: "JUMP" FUNCTION:

```
185     def jump(self):
186         """Enacts a jump (from the 'detect_movement' function)"""
187         if self.is_on_ground:
188             self.is_on_ground = False
189             self.is_jumping = True
190             self.jump_counter = 35
191
192             if self.player_state == EntityState.GROUNDED_RIGHT:
193                 self.player_state = EntityState.JUMP_RIGHT
194             elif self.player_state == EntityState.GROUNDED_LEFT:
195                 self.player_state = EntityState.JUMP_LEFT
```

- Line 187: Ensures Aim 2
- Lines 192-195 allow the switch from a standing state to a jumping state. (Aim 5)

The jump function will be run in the detect movement function:

```
            self.speed = WALKING_SPEED

        move_key_pressed = key_pressed[pygame.K_LEFT] or key_pressed[pygame.K_RIGHT]
        jump_key_pressed = key_pressed[pygame.K_SPACE] or key_pressed[pygame.K_UP]

        if move_key_pressed:
            if key_pressed[pygame.K_LEFT]:  # GROUNDED_LEFT
                if self.is_jumping:
                    self.player_state = EntityState.JUMP_LEFT
                else:
                    self.player_state = EntityState.GROUNDED_LEFT

            if key_pressed[pygame.K_RIGHT]:  # GROUNDED_RIGHT
                if self.is_jumping:
                    self.player_state = EntityState.JUMP_RIGHT
                else:
                    self.player_state = EntityState.GROUNDED_RIGHT

            self.move()

        if jump_key_pressed:
            self.jump()
```

This function essentially starts off the jump, but to actually add something to dy each update function I've used a different function:

SPRITE_PLAYER.PY: "JUMP_MOVEMENT" FUNCTION:

```
    def jump_movement(self):
        """Actually adds a value to dy every update function"""
        jump_dy_change = (self.jump_counter / 6) * -1  # needs to be negative as negative y change = going up
        setattr(self, "dy", jump_dy_change)
```

Now to put this all together in the update function where everything runs:

SPRITE_PLAYER.PY: "UPDATE" FUNCTION:

```
    def update(self):
        """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
        self.detect_movement()
        self.animate()
        self.gravity()

        self.rect.x += self.dx  # Updates position of player
        self.x_collide()

        if self.jump_counter != 0:
            self.jump_counter -= 1
            self.jump_movement()

        self.rect.y += self.dy
        self.y_collide()

        self.dx = self.dy = 0
```

I also need to alter the if statement in the animate function to include when the player is jumping:

```
131         standing_still = (self.dx == 0 and self.dy == 0)
132
133         if standing_still or self.is_jumping:
134             self.image = animations[0]
```

I didn't show it, but I also did the code for Aim 3 (to reduce space in this document). I also removed the code created earlier to test versions 4.1 and 4.2, where the up and down arrows could be used to slowly increase/decrease "dy".

TESTING VERSION 4.3

Expected Outcome:

1. A jump will only be possible if on the ground.
2. The gravity function will not run when the player is jumping.
3. A jump should cancel when hitting a ceiling.
4. The jump speed is quicker and slows down when the jump reaches its peak.
5. The jumps are animated.
6. Moving and jumping also work together

ATTEMPT 1:

I was about to run and record the testing and came across an immediate error:

```
    self.image = animations[0]
TypeError: 'pygame.Surface' object is not subscriptable
```

Looking into the code itself, I realised that, as there is only 1 sprite for the jumping animations, the entity state key is linked to the surface object by itself, as opposed to a list with the surface object. Due to how the walking animations are structured, I need to be able to take the first item out of a list. I want a general case for all animations where there is just one possible sprite, so this is my solution:

```
94      def process_animations(self, animations_enum: Animations):
95          """Takes list of coordinates for a spritesheet and returns pygame sprites"""
96          animations = {}
97
98          for player_state in animations_enum.value:
99              animation_list = self.get_animation_sprites(animations_enum.value[player_state])
100             if type(animation_list) is not list:
101                 animation_list = [animation_list]
102             animations[player_state] = animation_list
103
104         return animations
```

ATTEMPT 2:

[Check the "Testing – Version 4.3 A2" video to see the results]

Everything seems to be working fine, until the player stays in the jumping state and doesn't move out of it. This is because I forgot to change the "is jumping" attribute when the player hits the ground. This attribute is mainly there to help the program know which state to put the player in when the player is turning ("jumping left" or "grounded left"). To fix this, I have made the following fix

SPRITE_PLAYER.PY: INSIDE "Y_COLLIDE" FUNCTION:

```
if hit_block:
    if self.dy > 0:  # Player hits ground
        if self.player_state == EntityState.JUMP_RIGHT:
            self.player_state = EntityState.GROUNDED_RIGHT
        elif self.player_state == EntityState.JUMP_LEFT:
            self.player_state = EntityState.GROUNDED_LEFT

        self.rect.y = hit_block[0].rect.top - self.rect.height  # Mo
        self.is_on_ground = True
        self.is_jumping = False
```

ATTEMPT 3:

[Check the "Testing – Version 4.3 A3" video to see the results]

Now the problem is that the player seems to be moving to the right when running and jumping. This is also because of a little oversight which I've fixed here:

SPRITE_PLAYER.PY: INSIDE "MOVE" FUNCTION:

```
def move(self):
    """Actually changes dx value"""
    coordinate_change = self.speed

    if self.player_state in [EntityState.GROUNDED_LEFT, EntityState.JUMP_LEFT]:
        coordinate_change *= -1  # Makes number negative if the person's moving left
```

ATTEMPT 4:

[Check the "Testing – Version 4.3 A4" video to see the results]

This change fixed the problem. All the expected outcomes have now occurred. The aims for this version have been achieved.

REVIEW OF DEVELOPMENT 4

I am pleased that I now have working vertical movement for my player. I am surprised it has such an easy implementation. I am ready to move on with this project

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	Yes: Version 4.1
Apply gravity to the player's movement	Yes: Version 4.2
Add a jump function and animate it	Yes: Version 4.3
Create parent enemy class	
Allow player to jump on enemies to kill them	
Create death function for player	
Ability to switch between two maps seamlessly.	
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 5: CREATING AN ENEMY CLASS

VERSION 5.1: CREATING BASIC MOVING ENEMY

AIMS:

The objectives in this version are to:

1. Add an enemy class
2. Create movement that does not rely on key presses
3. Adapt the player animation functions for the enemy.
4. Make the enemy change direction if the enemy collides with a wall.

I am going to simply create one enemy class before I import different sprites, as they will all act very similar. A lot of the code is like the player sprite, like the processing animations, but I will show some of the functions that are different.

ANIMATIONS.PY:

```
class Animations(Enum):
    """Stores the x,y coordinates for the sprites. For the grounded sprites, the second item in the array contains the
    walking animations to create a loop."""
    GOOMBA = {EntityState.GROUNDED_LEFT: [[0, 4], [30, 4]],
              EntityState.GROUNDED_RIGHT: [[30, 4], [0, 4]],
              EntityState.DEATH: [60, 0]}
```

I have not added a death function for either player or enemy yet, that is a function for later, but I have still added its sprite. The enemies are only going to be moving, so I have added the sprites that are going to be playing in a loop.

SPRITE_ENEMY.PY: "__INIT__" FUNCTION

```
class Enemy(Entity):
    """General class for enemies"""
    def __init__(self, game, x, y):
        super().__init__(game, EntityGroups.ENEMY, x, y, EntityLayer.PERSON, spritesheet_name=SpriteSheetName.ENEMIES,
                         sprite_x=0, sprite_y=0, width=BLOCK_SIZE_X, height=BLOCK_SIZE_Y)
        # player is twice as tall as a regular block

        self.speed = E_WALKING_SPEED  # a constant that is slower than the player walking speed
        self.dx = self.dy = 0
        self.enemy_state = EntityState.GROUNDED_LEFT

        self.animations = self.process_animations(Animations.GOOMBA)
        self.animation_loop = 1

        self.is_on_ground = False
```

SPRITE_ENEMY.PY: UPDATE AND MOVE FUNCTION

```
def update(self):
    """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
    self.move()
    self.animate()
    self.gravity()

    self.rect.x += self.dx  # Updates position of player
    self.x_collide()

    self.rect.y += self.dy
    self.y_collide()

    self.dx = self.dy = 0

def move(self):
    """Actually changes dx value"""
    coordinate_change = self.speed

    if self.enemy_state == EntityState.GROUNDED_LEFT:
        coordinate_change *= -1  # Makes number negative if the person's moving left

    setattr(self, "dx", coordinate_change)  # Changes dx value
```

I need the enemy to always be moving, and not wait for a key press, so I have removed the detect movement function, and just always run the move function.

SPRITE_ENEMY.PY: ANIMATE FUNCTION

```
def animate(self):
    """Selects sprites to display for whatever the player is doing. If moving there is an animation loop played"""
    animations = self.animations[self.enemy_state]

    self.image = animations[floor(self.animation_loop)]
    self.animation_loop += 0.1
    if self.animation_loop >= 2:
        self.animation_loop = 0

    self.image = pygame.transform.scale(self.image, (16 * SCALE_UP, 16 * SCALE_UP))  # As these images have not
    # been scaled
```

There are only walking animations that play, so all I need is for the walking animation sprites to loop over and over.

SPRITE_ENEMY.PY: INSIDE "X_COLLIDE" FUNCTION

```
if hit_block:
    if self.dx > 0:
        self.rect.x = hit_block[0].rect.left - self.rect.width  # Moves char just to the left of the other object
        self.enemy_state = EntityState.GROUNDED_LEFT  # Changes direction

    elif self.dx < 0:
        self.rect.x = hit_block[0].rect.right  # Moves char just to the right of the other object
        self.enemy_state = EntityState.GROUNDED_RIGHT  # Changes direction
```

I need the enemy to change direction when walking into and colliding with a wall or block.

TESTING VERSION 5.1

Expected Outcome:

1. An enemy will spawn and appear
2. The enemy will be automatically walking at a speed slower than the player's
3. The enemy will be animated correctly
4. The enemy will change direction when colliding with a wall or a block.

ATTEMPT 1:

[Check the "Testing – Version 5.1 A1" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION 5.2: ENEMY DEATH FUNCTION

AIMS:

The objectives in this version are to:

1. Detect when the player collides with enemy
2. Remove the enemy from the screen
3. Play a death animation beforehand.
    a. Enemy changes sprite to death sprite
    b. Enemy stays there for a few seconds before disappearing.

SPRITE_PLAYER.PY: ENEMY COLLIDE FUNCTION

```
193     def enemy_collide(self):
194         """Determines whether an enemy collision has occurred. Determines whether this is a death for the player or for
195         the enemy."""
196         hit_enemy = pygame.sprite.spritecollide(self, self.game.enemies, False)
197         # hit_enemy will create a list of sprites in the enemies group that have collided with the player.
198         # It will have an empty list if there are no collisions, hence the "if hit_enemy" statement below will not run
199         # hit_enemy[0] will be the enemy that the player collides with.
200 
201         if hit_enemy:
202             enemy = hit_enemy[0]
203             if (enemy.rect.y - self.rect.height) < self.rect.y and enemy.enemy_state is not EntityState.DEATH:
204                 self.kill()
205             else:
206                 enemy.death()
```

[EXPLANATION FOR LINE 203]

- Line 203 checks if the player is not directly on top of the enemy (which would be the condition for the player to kill the enemy), if this is the case then the player is killed.
- It also checks to see if the enemy is not already dead, as the dead enemy sprite will stay there for a few seconds as part of the animation, so this could the program and trigger line 204 when this is not wanted.
- I've added line 204 as a placeholder for the next version when I create a player death function.

SPRITE_ENEMY.PY: DEATH FUNCTION

```
146     def death(self):
147         """Sets enemy state into the death state which changes its sprite image to the death one and sets off the
148         death timer in the update function."""
149         self.enemy_state = EntityState.DEATH
150         self.animate()
```

This may seem like such a simple function, however, as I don't want the enemy sprite to disappear instantly, I need to do the actual death in the update function. This simply signals to the program that the death animation is to commence.

SPRITE_ENEMY.PY: INSIDE ANIMATION FUNCTION

```
        if self.enemy_state is not EntityState.DEATH:
            self.image = animations[floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 2:
                self.animation_loop = 0

        else:
            self.image = animations[0]
```

As the death animation only has one sprite image that does not need to be played in a loop.

SPRITE_ENEMY.PY: UPDATE FUNCTION

```
    def update(self):
        """Runs any update to the player sprite like a movement and then the subsequent animation. Runs every tick"""
        if self.enemy_state is not EntityState.DEATH:
            self.move()
            self.animate()
            self.gravity()

            self.rect.x += self.dx  # Updates position of player
            self.x_collide()

            self.rect.y += self.dy
            self.y_collide()

            self.dx = self.dy = 0
        else:
            if self.death_counter != 0:
                self.death_counter -= 1
            else:
                self.kill()
```

Death counter is an attribute that has been set to 20 in the “__init__” function, to allow the animation to last longer than 1 update function.

TESTING VERSION 5.2

Expected Outcome:

1. When the player jumps on the enemy sprite, the player remains alive
2. When this happens, the enemy sprite stops moving.
3. The enemy sprite changes to the death sprite image.
4. The enemy sprite stays in this death sprite image for a few seconds before disappearing.

ATTEMPT 1

[Check the "Testing – Version 5.2 A1" video to see the results]

The player has been removed instead of the enemy sprite. The logic above should make sense, but maybe the program recognises the collision when the sprites overlap and not when they touch, so there is an offset of 1 or 2 pixels that I need to account for. To check this:

```
print(enemy.rect.y - self.rect.height, self.rect.y)
```

I then jumped vertically on top of the goomba, so that these two values in theory should be equal

```
352 355
```

There is an offset of 3, so I have reflected on this in the program:

SPRITE_PLAYER.PY: INSIDE ENEMY COLLIDE FUNCTION

```
if hit_enemy:
    enemy = hit_enemy[0]
    if (enemy.rect.y - self.rect.height + 3) < self.rect.y and enemy.enemy_state is not EntityState.DEATH:
        self.kill()
    else:
        enemy.death()
```

ATTEMPT 2

[Check the "Testing – Version 5.2 A2" video to see the results]

It works, but I'm not fully happy with how my animation looks. It works how I wanted before, but I'd like the main character to have a little jump as a result of the kill. I want it to be shorter in duration to a regular jump, so I need to tweak the jump function a little:

SPRITE_PLAYER.PY: PART OF JUMP FUNCTION:

```
def jump(self, counter):
    """Enacts a jump (from the 'detect_movement' function)"""
    if self.is_on_ground:
        self.is_on_ground = False
        self.is_jumping = True
        self.jump_counter = counter
```

Now when regular jumps are to be done, I have passed in 35 to the counter, but for this smaller jump when a player jumps on an enemy, I have used a smaller value:

SPRITE_PLAYER.PY: PART OF COLLIDE ENEMY FUNCTION:

```
else:
    enemy.death()
    self.is_on_ground = True  # Otherwise, jump won't happen
    self.jump(20)  # Smaller jump than usual
```

ATTEMPT 3

[Check the “Testing – Version 5.2 A3” video to see the results]

I am now happy with the animation. All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION 5.3: PLAYER DEATH FUNCTION

AIMS:

The objectives in this version are to:

1. Call a player death function when the player hits the enemy (but is not on top of the enemy)
2. This death function plays an animation where the player image changes to the death sprite.
3. In the animation, the player sprite rises for a bit before falling off the map.
4. Pause every other sprite when the player death animation is occurring.
5. Wait a few seconds before restarting the level
6. Call the function when the player falls off the screen (i.e., drops into a gap in the ground and falls off the screen.

I have added the animation to the animations Enum class, but now I must add this check to the animate function file:

SPRITE_PLAYER.PY: INSIDE ANIMATE FUNCTION:

```
if standing_still or self.is_jumping or self.player_state == EntityState.DEATH:
    self.image = animations[0]
```

SPRITE_PLAYER.PY: DEATH FUNCTION:

```
234     # ----------------------------------------PLAYER LEAVES MAP FUNCTIONS----------------------------------------
235     def death(self):
236         """Player death animation. Changes into death sprite, waits a little, rises a little and then falls off the
237         map. All other sprites are paused during this."""
238         print("Death function run flag")
239         self.player_state = EntityState.DEATH
240         self.animate()
241         self.game.draw()
242         sleep(0.25)
243         while self.rect.y < 700:
244             if self.death_counter != 0:  # When the player moves upwards in the death animation
245                 self.rect.y -= 3
246                 self.game.events()
247                 self.game.draw()
248                 self.death_counter -= 1
249             else:
250                 self.rect.y += 4
251                 self.game.events()
252                 self.game.draw()
253         print("Player died")
254         sleep(2)
255         self.game.start_game()
```

This is the death function itself.

- The entire death animation and the rest of the aims will happen in this function, so “game functions” will not run, and therefore the player “update function” and the game “draw” function will also not run. Therefore lines 240 and 241 are there to compensate for this.

- I added sleep statements in lines 242 and 254 to add little delays in the animation (taken from the time module).
- I have used a while loop in line 243 as opposed to using the update function every tick so that everything is completed in this function, which means that all other sprites remain still and do not get updated.
- Death counter is an attribute I have initialised earlier with the sole purpose of demonstrating aim 3, in lines 244-248.
- For the case where the player falls off the screen, which causes the death, the animation will not be visible as the player will already be off screen, so I have added line 253 to confirm this for now.
    - Later, this type of death will be obvious as I will play a sound when the death occurs.

Now I need to call this when the player is meant to be hit by the enemy:

TESTING VERSION 5.3

Expected Outcome:

1. The death function will run when the player touches an enemy but (but is not directly on top of the enemy)
2. The enemy still dies if the player jumps on the top of the enemy.
3. The death animation plays
4. The correct death animation plays: the player sprite rises for a bit before falling off the map.
5. Every other sprite pauses when the player death animation is occurring.
6. The program waits for a few seconds before the level is restarted.

ATTEMPT 1:

[Check the "Testing – Version 5.3 A1" video to see the results]

Although almost all the aims have been achieved, there was a random pause of two seconds when the player sprite respawns. The terminal box also printed the "player died" flag on two occasions each death. This made me think that the death function was being run more than once. I added a few more print statements to try and confirm this. I then started the program and walked into the enemy, which should call the death function once. The game then restarted, and this is what the terminal printed during this:

```
gamestartedflag
Death function run flag
Player died

gamestartedflag
Death function run flag
Player died

gamestartedflag
```

I would have expected the game started flag, which is printed when the "start game" function is run, to have been printed twice. However, it has printed on three different occasions. The second block of text printed in between loading the map again, so the function has been called again at this point. Trying to find out why is the next step.

I theorised that the "kill" function, which I originally thought entirely removed a sprite from existence, did not actually do that, so I instead had multiple player objects created, and the "deleted" Player sprite may be still running the function and causing errors. I checked this by adding all instances of the player function to an array. I then printed this array out after 1 death and 1 respawn:

```
[<Player Sprite(in 0 groups)>, <Player Sprite(in 0 groups)>, <Player Sprite(in 2 groups)>]
```

So, there is also a random 3rd player sprite here causing issues, yet I do not know why it is created. I've realised now that if "kill" doesn't entirely delete an object, but rather just removes them from the sprite groups (thus they do not get drawn), then the structure of my whole code needs to be reworked.

I originally thought that this is how my program would work with pygame:

1. A block needs to be placed in a certain location → I create a new block object and pass the coordinates in.
2. To remove the block from the screen, I use the "kill" function → Destroys the object from existence (hence it is no longer on the screen)
3. Therefore, to place the same object again, I must create an entirely new object and place it in with the same coordinates, as my old object was destroyed.

This is how I thought a pygame game was meant to be made, so I designed my game with this all-in mind. As this is not the case, I need to redesign parts of my code.

This will be a very large project and as such I am halting this version and finishing it later.

VERSION 5.4: CREATE NEW ENTITY CONSTRUCTOR

AIMS:

The objectives in this version are to:

1. Create an array which tracks which objects have been created but are currently not on screen.
2. Create an alternative to "kill"
3. Create an entity constructor that checks the unused objects array to see if an existing object is there
    a. If there is one, use it, and create a function to display the object on the screen again.
    b. Otherwise, create a new instance of said object.

So essentially, if we use the enemy as an example, when it dies (assume there is only one enemy on the screen):

1. Remove the enemy sprite from the "all sprites" group, so it will stop being drawn or updated.
2. Add it to the unused sprites array, for later use.
3. When the level restarts and the enemy must be spawned in its specific place, an enemy sprite will have to be retrieved from somewhere.
4. The program will check the unused sprites array to see if there is an enemy that has been created but is not being used.
    a. If there is, it returns the unused old enemy sprite.
    b. Otherwise, it creates a new enemy sprite
5. If the old enemy sprite has been retrieved, most variables should be the same, but a new x and y coordinate must be set, and the entity state must be reset to its default (along with any other variables)

The following screenshots are all from the "Entity" class in the "sprites_entity.py" file.

```
class Entity(pygame.sprite.Sprite):
    unused_instances = []
```

First, I must define this array of unused sprites: I have done it as a class variable to the Entity class, so that all child classes and instances of these classes have access to it.

```
84      @classmethod
85      def find_unused_instance(cls):
86          """To be used with every child class of entity, not entity itself."""
87          for instance in cls.unused_instances:
88              if isinstance(instance, cls):
89                  return instance
90          return False
```

Now I must be able to filter through the array of unused entity types and see if there is a specific unused object. Say this is being run by the “Player” class, “cls” will refer to the player class. Line 88 will check to see if any of the instances are an instance of a “Player” class. If one is found, it is returned, otherwise false is returned.

```
92      @classmethod
93      def create_instance(cls, game, x, y):
94          """The constructor variable for every child class of entity. Checks if an instance of the variable is available
95          i.e. has been created but is not being displayed on screen, if so, this unused instance is retrieved, otherwise
96          it creates a new instance."""
97          instance_found = cls.find_unused_instance()
98          if instance_found:
99              instance_found.show_on_screen(x, y)
100             return instance_found
101         return cls(game, x, y)  # if unused instance is not found then a new instance is created.
```

The function above is used in this function, which is the new constructor function for any instance of an entity sprite. The docstring explains what the function does. The “show on screen” function will be shown here:

```
65      def show_on_screen(self, new_x: int, new_y: int):
66          """When an unused sprite which has been removed off the screen is being shown again."""
67          self.game.all_sprites_group.add(self)  # If it's in the all sprites group, then the sprite will be drawn/updated
68          self.unused_instances.remove(self)  # The sprite is now being used.
69          self.reset_variables()  # All entities have this, only player and enemy ones actually do something
70          self.change_coordinates(new_x, new_y)
71
72      def remove_from_screen(self):
73          """Alternative to self.kill(), it removes the sprite from the all_sprites group, which stops the sprite from
74          being drawn and updated, and adds it to the unused instances array, so it can be reused again later on."""
75          self.game.all_sprites_group.remove(self)
76          self.change_coordinates(0, 0)  # In case it could have interfered with something
77          self.unused_instances.append(self)
78
79      def change_coordinates(self, new_x: int, new_y: int):
80          """Used mainly when an instance of a class is being reused, as it needs new coordinates to be set."""
81          self.rect.x = new_x * BLOCK_SIZE_X * SCALE_UP
82          self.rect.y = new_y * BLOCK_SIZE_X * SCALE_UP
```

These functions should also be explained by their docstrings.

- Line 69: I need to reset a few variables like entity state (the player sprite would be in its death state so this needs to be changed to a grounded state before it is shown again otherwise it will start off with the death state) and the death counter needs to be reset.
- The remove from screen function has replaced all uses of “kill”, which would have been used to empty the screen before loading a new map, or to remove a player/enemy from the screen.

Now I just need to use the new constructor functions:

GAME.PY: INSIDE "CREATE MAP" FUNCTION

```
if item == "B":
    Block.create_instance(self, x, y)
elif item == "G":
    Ground.create_instance(self, x, y)
elif item == "I":
    InvisibleWall.create_instance(self, x, y)
elif item == "P":
    Player.create_instance(self, x, y)
elif item == "E":
    Enemy.create_instance(self, x, y)
```

If all goes well, this version should have meant that no extra sprites are made, removed from the screen, and then left unused while an almost duplicate has been created for no reason, taking up a lot of extra memory to store the unused objects and also processor time to create them in the first place.

To test version 5.4, I will add a temporary counter to the entity class, which will increase by 1 every time an entity is created. This number should equal the number of sprites in "all sprites" when everything is on the screen (no deaths have happened yet). I will test this before and after a death and restart happen, to see if the numbers are the same. I will also see if the number sprites in the "all sprites" group decreases by 1 when an enemy death happens, to see if the remove from screen function works as intended.

TESTING VERSIONS 5.3 AND 5.4

Expected Outcome (V 5.3):

1. The death function will run when the player touches an enemy but (but is not directly on top of the enemy)
2. The enemy still dies if the player jumps on the top of the enemy.
3. The death animation plays
4. The correct death animation plays: the player sprite rises for a bit before falling off the map.
5. Every other sprite pauses when the player death animation is occurring.
6. The program waits for a few seconds before the level is restarted.
7. Between the starting of the level, the death of the player and the restarting of the level:
    i. "Death function flag" is printed once
    ii. "Player died" is printed once
    iii. "Game Started Flag" is printed twice
    iv. The game only freezes for a few seconds once (just before the level is restarted)

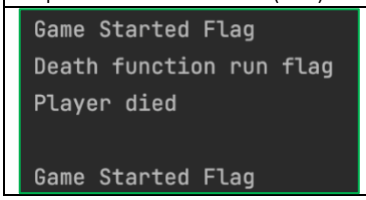
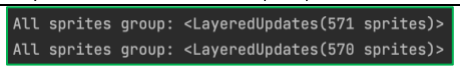
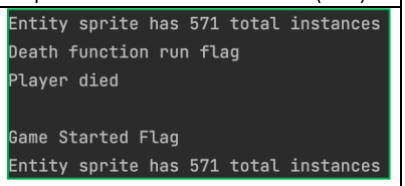
Expected Outcome (V 5.4):

1. The number of entities made before and after a death happen are the same.
2. The number of entities on the screen is the same as the number of entities created overall (when all sprites created are displayed on the screen)
3. The number of sprites in the "all sprites" group decreases by 1 when an enemy dies.

ATTEMPT 1

[Check the "Testing – Version 5.3/5.4 A1" video to see the results]

Expected Outcomes 1-6 have been achieved by this video.

Expected Outcome 7 (5.3)	Expected Outcome 3 (5.3)	Expected Outcomes 1 + 2 (5.3)
		

All the expected outcomes have occurred. The aims for this version have been achieved.

REVIEW OF DEVELOPMENT 5

This development was meant to be a relatively simple one, as I believed I had created a good robust framework and basis to then create an enemy class based on my player class and gained a good enough understanding of collisions to the point where the only proper complexity would have been the death animations. To an extent, that was almost the case, however, a huge flaw with my design and structure of my program meant that I needed to make a huge, unexpected change to it, which took a lot of time.

The problem wasn't that my design was fully wrong, more that I based my code around my knowledge of how pygame worked, which I have now found to be wrong. If the way I thought the sprites were handled was indeed correct, then my code would have been fine.

I originally thought that this is how my program would work with pygame:

1. A block needs to be placed in a certain location → I create a new block object and pass the coordinates in.
2. To remove the block from the screen, I use the "kill" function → Destroys the object from existence (hence it is no longer on the screen)
3. Therefore, to place the same object again, I must create an entirely new object and place it in with the same coordinates, as my old object was destroyed.

However, this was not the case, and the "destroyed" objects were still in the code, just left unused. If I had left the program the way it was, not only would there have been potential bugs (as seen in testing attempt 1 for version 5.3), there would have also been lots of wasted memory space and wasted processing time creating redundant objects. So, although there was a lot of extra time taken, I have now learned how to better create pygame games for the future, and I have greatly optimised my code, and my design is still robust.

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	Yes: Version 4.1
Apply gravity to the player's movement	Yes: Version 4.2
Add a jump function and animate it	Yes: Version 4.3
Create parent enemy class	Yes: Version 5.1
Allow player to jump on enemies to kill them	Yes: Version 5.2

Create death function for player	Yes: Version 5.3
Create new entity constructor	Yes: Version 5.4
Ability to switch between two maps seamlessly.	
Make warp pipes to change maps	
Create end goal for a level	
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 6: MULTIPLE LEVELS

AIMS:

The objectives in the sixth development are to:

1. Add the ability to have change maps
2. Add working warp pipes that transfer you between maps

VERSION 6.1: MULTIPLE MAPS

AIMS:

The objectives in this version are to:

1. Add a second map
2. Be able to switch between maps without errors. (For now, this will be with certain temporary key presses)

MAPS.PY:

```
class MapId(Enum):
    """This is the key for the maps to refer to each map in the game"""
    LEVEL_1_1 = auto()
    LEVEL_1_2 = auto()
    LEVEL_2_1 = auto()
    LEVEL_3_1 = auto()
    LEVEL_4_1 = auto()
    LEVEL_5_1 = auto()
    LEVEL_5_2 = auto()

    def get_map(self):
        return maps[self]
```

```
maps = {MapId.LEVEL_1_1:[...],
        MapId.LEVEL_1_2: ["BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB",
                          "BBBBBBBBBBBBBB............BBBBBBBBBBBBBBB",
                          "BBBBBBBBBBBBBB.P..........BBBBBBBBBBBBBBB",
                          "BBBBBBBBBBBBBB........E...BBBBBBBBBBBBBBB",
                          "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"]}
```

This is how I've chosen to organise my maps. I've given them an enum id, which links to a map. I've then created a dictionary of maps ("LEVEL_1_1" is the test map I showed before), which the map id can retrieve. I might change the naming of the ids, but as for now, the first number is the overall level number whereas the second number is the map number for that level.

GAME.PY: INSIDE "__INIT__" FUNCTION

```
self.map_ids = [Id for Id in MapId]
self.current_map_id = MapId.LEVEL_1_2
```

GAME.PY: INSIDE "CREATE MAP" FUNCTION

```
def create_map(self):
    """Creates the map from the blueprints given in the 'maps.py' file."""
    for y, row, in enumerate(self.current_map_id.get_map()):
```

Now to test if they work, I'll add some key presses to switch levels (for now):

SPRITE_PLAYER.PY: INSIDE DETECT MOVEMENT FUNCTION:

```
if key_pressed[pygame.K_o]:
    self.game.current_map_id = MapId.LEVEL_1_1
    self.game.start_game()

if key_pressed[pygame.K_p]:
    self.game.current_map_id = MapId.LEVEL_1_2
    self.game.start_game()
```

TESTING VERSION 6.1

Expected Outcome:

1. When the "p" key is pressed, the level switches to a map where the player is trapped in a small box
2. When the "o" key is pressed, the level switches to a map where the player is in the larger test map seen before
3. It is possible to switch back and forth with no errors whatsoever

ATTEMPT 1:

[Check the "Testing – Version 6.1 A1" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

VERSION 6.2: WARP PIPES

AIMS:

The objectives in this version are to:

1. Add a warp pipe and import the correct sprite.
2. Add the ability to rotate the pipe sprite.
3. Add the warp function to allow the player to move through the pipe and move to a different map
4. Add a pipe entering/exiting animation.

I am going to split the testing of this version up, as it is quite a big version and a lot can go wrong.

DEVELOPING/TESTING VERSION 6.2.1: CREATE PIPE SPRITE

DEVELOPMENT:

The warp pipe sprite needs to first be defined:

SPRITE_ENTITY.PY: PIPE ENTITY

```
164     class Pipe(Entity):
165         """A pipe sprite object"""
166
167         def __init__(self, game, x, y):
168             super().__init__(game, x, y, EntityGroups.PIPE, EntityLayer.PIPE, spritesheet_name=SpriteSheetName.PIPES,
169                              sprite_x=2, sprite_y=42, width=BLOCK_SIZE_X * 4, height=BLOCK_SIZE_Y * 2)
```

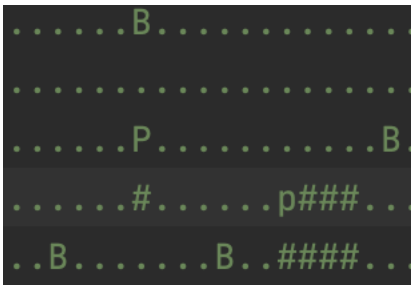
- I have given it an entity layer which is one above the player, as this will be useful for the pipe animation where the player goes underneath the pipe (to give the illusion that they went through the pipe)

Now to add it to the map:

GAME.PY: INSIDE "CREATE_MAP" FUNCTION:

```
elif item == "P":
    Player.create_instance(self, x, y)
elif item == "E":
    Enemy.create_instance(self, x, y)
elif item == "p":  # facing left
    Pipe.create_instance(self, x, y)
```

GAME.PY: PART OF THE MAP FOR LEVEL 1 MAP 1



I have started using hashtags for the space that will be taken up by an entity that is bigger than 1 grid square.

TESTING:

Expected Outcome:

1. The pipe appears to the right of the block formations, facing left
2. The collisions for the pipe are the same as a regular block (stops the entity before they walk through the pipe)
    a. For the player
    b. For the enemy

ATTEMPT 1

[Check the "Testing – Version 6.2.1 A1" video to see the results]

All the expected outcomes have occurred. This aim has been achieved.

DEVELOPING/TESTING VERSION 6.2.2: ROTATE PIPE SPRITE

DEVELOPMENT:

Firstly, I've created this Enum to make the later code clearer.

SPRITE_ENTITY.PY: ROTATION DIRECTION ENUM:

```
26  class RotationDirection(int, Enum):
27      """Values used to rotate any sprite as long (as you assume that they are looking left)"""
28      LEFT = 0
29      DOWN = 90
30      RIGHT = 180
31      UP = 270
```

The values here should signify which way the entrance/exit to the pipe is facing. Next, I need to make it so that I can change the image of a sprite after the sprite has originally been created (as pipes will be reused, but they may have to be rotated):

SPRITE_ENTITY.PY: INSIDE "INIT" FUNCTION, GET IMAGE FUNCTION OF ENTITY SPRITE

```
52          pygame.sprite.Sprite.__init__(self, self.groups)
53
54          self.spritesheet = game.spritesheets[spritesheet_name]
55          self.sprite_coordinates = [sprite_x, sprite_y]
56
57          self.get_image(x, y)
58
59      def get_image(self, x, y, rotation_dir: RotationDirection = None):
60          """Gets the image and creates the object sprite."""
61          self.image = self.get_sprite(self.sprite_coordinates[0], self.sprite_coordinates[1])
62          self.image = pygame.transform.scale(self.image, (self.width * SCALE_UP, self.height * SCALE_UP))
63          if rotation_dir is not None:
64              self.image = self.image = pygame.transform.rotate(self.image, rotation_dir)
65
66          # Rect stores the position of the entity, img is just how it appears on screen.
67          self.rect = self.image.get_rect()
68          self.rect.x = x * BLOCK_SIZE_X * SCALE_UP
69          self.rect.y = y * BLOCK_SIZE_Y * SCALE_UP
```

Not everything needs to be able to rotate, hence why line 63 is there. Now I need to edit the "create instance" constructor function so that it can accommodate sprites that rotate:

SPRITE_ENTITY.PY: CREATE INSTANCE FUNCTION:

```
110     @classmethod
111     def create_instance(cls, game, x, y, rotation_dir: RotationDirection = None):
112         """The constructor variable for every child class of entity. Checks if an instance of the variable is available
113         i.e. has been created but is not being displayed on screen, if so, this unused instance is retrieved, otherwise
114         it creates a new instance."""
115         instance_found = cls.find_unused_instance()
116         if instance_found:
117             instance_found.show_on_screen(x, y)
118 
119             if rotation_dir is not None:  # for pipes or other rotatable objects.
120                 instance_found.get_image(x, y, rotation_dir)
121                 instance_found.set_rotation_dir(rotation_dir)
122 
123             return instance_found
124 
125         if rotation_dir is None:
126             return cls(game, x, y)  # if unused instance is not found then a new instance is created.
127         return cls(game, x, y, rotation_dir)  # for pipes or other rotatable objects.
```

- Lines 119-121, 125, and 127 should accommodate for sprites that can rotate (pipes), while not bothering for sprites that never need to rotate (blocks, player).

Now I need to be able to add these different pipes to the map:

GAME.PY: INSIDE "CREATE_MAP" FUNCTION:

```
elif item == "v":  # facing up
    Pipe.create_instance(self, x, y, RotationDirection.UP)
elif item == "V":  # facing down
    Pipe.create_instance(self, x, y, RotationDirection.DOWN)
elif item == "h":  # facing left
    Pipe.create_instance(self, x, y, RotationDirection.LEFT)
elif item == "H":  # facing right
    Pipe.create_instance(self, x, y, RotationDirection.RIGHT)
```

This may look confusing, but:

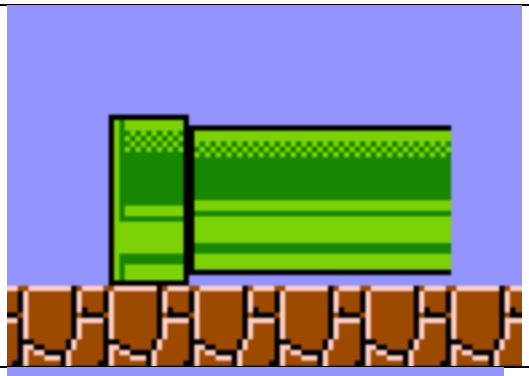
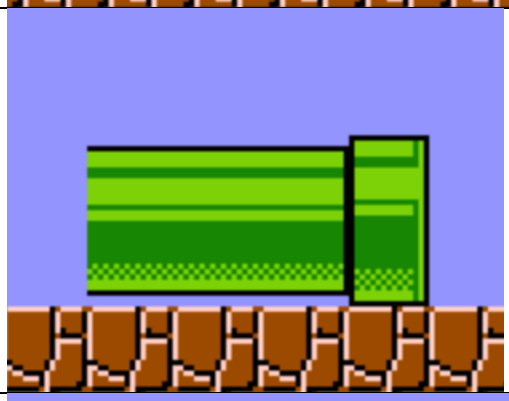
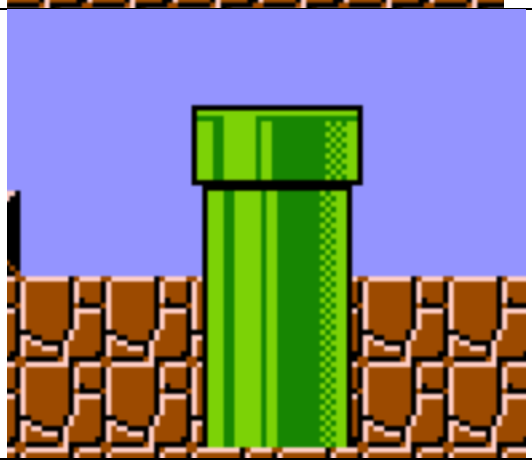
- H is for a horizontal pipe while V is for a vertical pipe
- Capital letters denote the positive "xy" directions (right, down) whereas lowercase letters denote the negative "xy" directions (left, up)

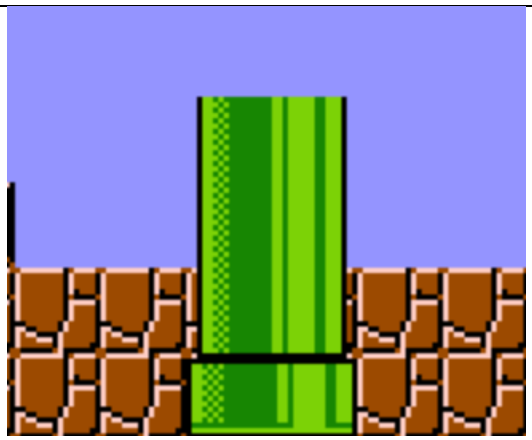
TESTING:

Expected Outcome:

1. All 4 sprites are displayed depending on the letter in the map:

ATTEMPT 1

Pipe Entrance Facing	Letter	Result
Left	h	
Right	H	
Up	v	

Down	V	

All the expected outcomes have occurred. This aim has been achieved.

DEVELOPING/TESTING VERSION 6.2.3: CREATE WARP FUNCTION

DEVELOPMENT:

So due to time constraints, I am going to have to use the backup plan in the design section. I am now going to have to

- Only make 1 entry pipe per map.
- The entry pipes are always facing left.
- Any exit pipes are facing right/down but cannot be entered back into (no backtracking in a level).

This means that each map only has one way to leave the map, and only one destination map after the player leaves the original map. I can just create a huge array of all map ids in order, and then when the player goes in an entry pipe, I can make the player go to the map next in the array.

As the map ids are in an Enum, I need a way to:

- Create an array of all maps in order
- Find the position of the current map in the list using a binary search
- Return the map id of the map after the current one:

MAPS.PY: BINARY_SEARCH FUNCTION

```
4   def binary_search_enums(arr: [Enum], x: Enum):
5       """Used to search for the index of the enum in a list of enums"""
6       low = 0
7       high = len(arr) - 1
8
9       while low <= high:
10          mid = (high + low) // 2
11          if arr[mid].value < x.value:  # If x is greater, ignore left half
12              low = mid + 1
13
14          elif arr[mid].value > x.value:  # If x is smaller, ignore right half
15              high = mid - 1
16
17          else:# means x is present at mid
18              return mid
19      return -1  # If we reach here, then the element was not present
```

- ".value" is being used so that the Enums can be compared.
- Line 19 should never run, is more of a fail-safe.

MAPS.PY: INSIDE "MAP_ID" ENUM CLASS

```
41      @classmethod
42      def get_maps_list(cls):
43          """Creates list of all map_ids"""
44          return [map_id for map_id in cls]
45
46      def get_next_map(self):
47          """returns map id of next map"""
48          maps_list = self.get_maps_list()
49          index = binary_search_enums(maps_list, self)
50          return maps_list[index + 1]
```

Now I need to create the ability for the player to enter a pipe so that the above functionality can then be used. First, I need to see whether a pipe is an entry pipe or not.

SPRITE_ENTITY.PY: PIPE ENTITY: "IS_ENTRY_PIPE" FUNCTION

```
def is_entry_pipe(self) -> bool:
    return self.rotation_dir is RotationDirection.LEFT
```

Now I need to detect whether a player is walking in the right direction into a pipe facing left.

SPRITE_PLAYER.PY: INSIDE "X_COLLIDE" FUNCTION

```
160         hit_pipe = pygame.sprite.spritecollide(self, self.game.pipes_group, False)
```

```
178     if hit_pipe:
179         pipe = hit_pipe[0]
180         if self.dx > 0 and pipe.is_entry_pipe():  # Can only enter pipe that faces left
181             self.enter_pipe(pipe)
```

Now to combine everything and make entering a pipe led the player to the next map:

SPRITE_PLAYER.PY: ENTER PIPE FUNCTION:

```
263     def enter_pipe(self, pipe):
264         self.game.current_map_id = self.game.current_map_id.get_next_map()
265         self.game.start_game()
```

TESTING:

Expected Outcome:

1. Walking right into the pipe should transport the player into the next map (the enclosed box in the sky)
2. Walking into a pipe facing right should do nothing.

ATTEMPT 1

[Check the "Testing – Version 6.2.3 A1" video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

DEVELOPING/TESTING VERSION 6.2.4: PIPE ANIMATIONS

DEVELOPMENT:

AIMS:

1. A pipe entering animation is player when the player enters a pipe
    a. The player is in its standing sprite.
    b. The pipe slowly disappears into the pipe while everything else freezes
2. A pipe exiting animation is played when the player enters a map through a pipe.

SPRITE_PLAYER.PY: PIPE ENTER ANIMATION FUNCTION:

```
276     def pipe_enter_animation(self, pipe):
277         """The animation when a player enters a pipe"""
278         self.player_state = EntityState.GROUNDED_RIGHT
279         self.dx = 0
280         self.animate()
281
282         desired_player_x = pipe.rect.x+1
283         while self.rect.x != desired_player_x:
284             self.rect.x += 1
285             self.game.events()
286             self.game.draw()
```

This should achieve the animation set out in my aims. To run this function, I need to put it in the enter pipe function:

SPRITE_PLAYER.PY: ENTER PIPE FUNCTION:

```
269     def enter_pipe(self, pipe):
270         """The entering of a pipe by a player. Warped to next map in game"""
271         self.pipe_enter_animation(pipe)
272         sleep(2)
273         self.game.current_map_id = self.game.current_map_id.get_next_map()
274         self.game.start_game()
```

Now I need to make a mechanism for when the player enters a map through a pipe. My plan is to:

1. Place the player inside the pipe so that they overlap when the level begins
    a. Which indicates to the program that the player is meant to come out of the pipe.
2. Potentially detect this at the very beginning of the map being created, which will then trigger the pipe exiting animation:

SPRITE_PLAYER.PY: EXIT PIPE FUNCTIONS:

```
288     def if_exit_pipe(self):
289         """Run at the beginning of a map loading to see if the player is meant to come out of a pipe to get into the new map"""
290         hit_pipe = pygame.sprite.spritecollide(self, self.game.pipes_group, False)
291         if hit_pipe and not hit_pipe[0].is_entry_pipe():
292             self.exit_pipe_animation(hit_pipe[0])
293 
294     def exit_pipe_animation(self, pipe):
295         """The animation when a player exits a pipe"""
296         if pipe.rotation_dir is RotationDirection.DOWN:
297             desired_player_y = pipe.rect.y + pipe.rect.height + 1
298             while self.rect.y != desired_player_y:
299                 self.rect.y += 1
300                 self.game.events()
301                 self.game.draw()
302 
303         if pipe.rotation_dir is RotationDirection.RIGHT:
304             desired_player_x = pipe.rect.x + pipe.rect.width + 1
305             while self.rect.x != desired_player_x:
306                 self.rect.x += 1
307                 self.game.events()
308                 self.game.draw()
```

Now I just need to run this at the beginning of the loading of a map:

GAME.PY: LOAD_MAP FUNCTION (FORMERLY CALLED “START GAME”):

```
def load_map(self):
    """Loads a map"""
    self.empty_screen()
    self.playing = True
    self.create_map()
    self.centre_character()
    player = [_ for _ in self.player_group][0]
    player.if_exit_pipe()
```

TESTING:

Expected Outcome:

1. When the player enters a pipe, the appropriate animation is played.
2. When the player enters the other map, the appropriate animation is played for:
    a. A pipe facing right
    b. A pipe facing down

ATTEMPT 1:

[Check the “Testing – Version 6.2.4 A1” video to see the results]

All the expected outcomes have occurred. The aims for this version have been achieved.

REVIEW OF DEVELOPMENT 6

This development marks the end of any new features really being added. Although I still have a lot to add, I have now created the basis and infrastructure to be able to do all of that very easily. For example:

- Now that I have created a working pipe that transports you from one map to the other, creating a finish line flag at the end of a level would use the same mechanism. The animation for that will also be very similar to the pipe animation, in the way that I used a while loop to freeze everything else.
- Now that I have made it easy to import any sprite, I can now add a variety of enemies, with different appearances, sizes and maybe movement speeds. All I need to do is:
    - Write down the new spritesheet name and its background colour
    - Find the x, y coordinates for all the animation sprites and store that in an appropriately named animations Enum class
    - Put that information in a new enemy class derived from the base enemy class and a new enemy is formed.
    - With this, I could make a new type of enemy in around 5 or 10 minutes, thanks to the infrastructure I have been trying to build this whole project.
- If I wanted to implement the disappearing blocks in my success criteria, I now have a reliable "remove from screen" function, and I can add if statement checks to the block's update function to trigger this when the required event has occurred

Because of this, I am not going to detail the further development as much, as all the code documented so far has hopefully meant that the implementation of any new features by this point should be trivial.

I have also concluded that a lot of things in my success criteria for this project will simply not be possible due to time constraints.

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	Yes: Version 4.1
Apply gravity to the player's movement	Yes: Version 4.2
Add a jump function and animate it	Yes: Version 4.3
Create parent enemy class	Yes: Version 5.1
Allow player to jump on enemies to kill them	Yes: Version 5.2

Create death function for player	Yes: Version 5.3
Create new entity constructor	Yes: Version 5.4
Ability to switch between two maps seamlessly.	Yes: Version 6.1
Make warp pipes to change maps	Yes: Version 6.2
Create end goal for a level	Later Development
Import actual sprites to be used.	
Add all types of enemies, with different abilities	
Add powerups for the player	
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	
Add a game over screen when the character reaches 0 lives.	
Add a victory screen when the player reaches the end of the game.	
Add a key that unlocks pipes	
Add coins that could contribute to a score	
Add blocks that disappear when certain goals have been achieved.	
Add Background Music	
Add Sound Effects	
Create all 8 Levels, with multiple maps	

DEVELOPMENT 7: CREATING ALL LEVELS AND FINISHING DEVELOPMENT

As stated in the review for development 6, I will not be extensively documenting and showing the testing for the final stage of this project. Mainly because all of it worked on the first attempt of testing, but also because a lot of what I added is a very simple extension on what I created beforehand. I will instead give a summary of everything that has been added in this development. You will be able to see all the components working in the final testing phase:

- Change sprites to copyright free ones
    - Now that I have completed most of the development phase and mostly completed all the entity sprite behaviours, it is no longer beneficial for me to use the spritesheets that I will not be using in the final game. It will be easier for me to code the later parts of the project.
    - I made it simple for me to import sprite sheets, by just creating an Enum key with the sprite sheet's name so this was a very seamless process.
- Add multiple types of enemies
    - As importing new spritesheets was very easy, and I already created a base enemy class, this was also a seamless process. All I had to do was find the sprite coordinates and add them to the animations Enum.
- Add a finish goal
    - I achieved this by:
        - Creating a new type of block called a goal block
        - Gave it its own sprite group
        - When the player collides with the goal block group, the finish level function is run.
- Add an animation for finishing a level
    - The player runs off the screen
    - I use a while loop until the player is off the screen so that nothing else is moving, similar to the death animation and pipe animation.
- Add a victory screen
- Add a lives counter to the top left
    - I simply made sprites that resided in the top left copied the player's movement entirely so it would stay there.
- Add a screen before a level begins showing the life count and the level number (only a level, not a new map).
    - I just moved the life counter to the centre of the screen.
- Add a key and make sure the pipe only works if the key is selected
- Move the key to the top left of the screen when it is picked up.
    - I made it copy the motion of the player similar to the lives counter so it would stay in the top left. I gave it a higher layer than other entities so it would appear over them.
- Add animated coins that get picked up when colliding with the player
    - The animated part of these coins is identical to the animate function for the player/enemy (except there is only one state, so there is only one set of animations)
- Add enemy disappearing blocks
- Add enemy/coin/key disappearing blocks

    - My explanation for how I did these blocks is in the design section.
- Create 4 Levels
    - This was originally going to be 8, each with many maps, but I only had time for 4.
- Add menu screens to show rules and controls
- Add a game over screen
- Add a different background colour depending on the level.
    - This is linked to the map id using a dictionary.

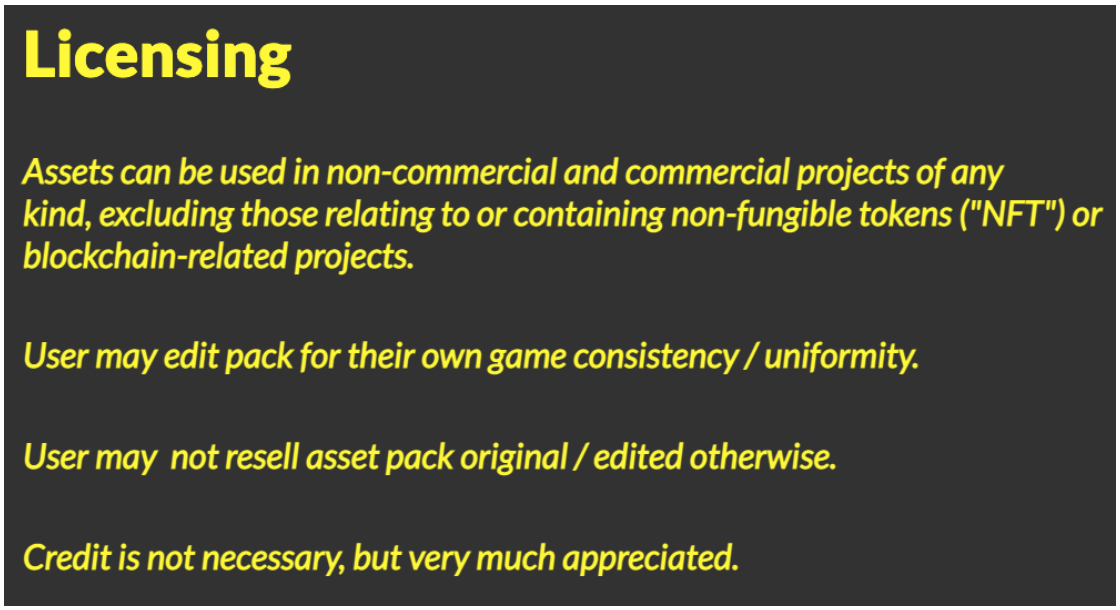
NEW SPRITES

The sprites I found and used were free and the creator has given permission for anyone to use his work:

https://jonathan-so.itch.io/creatorpack

> Do you need graphics or music for your game? **The Game Creator's Pack** has you covered!
>
> This Creative-Commons-Zero [CC-0] asset package contains authentic retro-style graphic assets and music for creators of all skill levels and needs.

https://elvies.itch.io/platform-grow-set-tiles-and-character-pack?download



I recorded the footage of the testing maps when I first imported the new sprites, to compare the looks before and after the switch:

[Check the "Testing – Version 7: New Sprites" video]

REVIEW OF DEVELOPMENT 7

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Later Development
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	Yes: Version 4.1
Apply gravity to the player's movement	Yes: Version 4.2
Add a jump function and animate it	Yes: Version 4.3
Create parent enemy class	Yes: Version 5.1
Allow player to jump on enemies to kill them	Yes: Version 5.2
Create death function for player	Yes: Version 5.3
Create new entity constructor	Yes: Version 5.4
Ability to switch between two maps seamlessly.	Yes: Version 6.1
Make warp pipes to change maps	Yes: Version 6.2
Create end goal for a level	Later Development
Import actual sprites to be used.	Development 7
Add all types of enemies, ~~with different abilities~~	Partially: Development 7
Add powerups for the player	Not Completed
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	Development 7
Add a game over screen when the character reaches 0 lives.	Development 7
Add a victory screen when the player reaches the end of the game.	Development 7
Add a key that unlocks pipes	Development 7
Add coins that ~~could contribute to a score~~	Partially: Development 7
Add blocks that disappear when certain goals have been achieved.	Development 7
Add Background Music	Not Completed
Add Sound Effects	Not Completed
Create all ~~8~~ 4 Levels, with multiple maps	Partially: Development 7

4: FINAL TESTING

USABILITY/BETA TESTING

For the final testing, the program was given to a few other users without being given instructions on how to use it as a form of beta testing. This allows me to receive any feedback and constructive criticism about the game, and to also discover any bugs that may have been overlooked during development.

The positive feedback received from the users:

- Perceived as fun
- Animations ran smoothly
- Game ran without lag or delays
- The collision detection was impressive
- The controls were simple to understand.

However, in terms of constructive criticism:

- It was sometimes not clear when something was happening, suggestion to add sound effects to the game to make certain things clear (when an enemy has been defeated, when a pipe has been entered, when a coin has been collected)
- The game felt a little empty without any background music.
- Some of the levels were too difficult. I didn't notice this myself as I had unknowingly memorised the game levels from designing it and testing it, so any common pitfalls or areas of difficulty were more difficult for me to spot.
    - This was especially emphasized for the level with an invisible wall maze.
    - I went to the levels that got the most attention and altered them. On a second playthrough, the users said the difficulty was much fairer.

As for the music and sound effects, I realised that even though I was under a lot of time pressure, music and sound effects were just too important to miss out. So, I added them. They were a lot easier to add than I expected, as pygame simply requires the file name of the sound. I added an Enum where the key was just the filename, and then added the function to play the sounds, and then coded in the functions when specific sounds needed to be played. The sounds were taken from the "Game Creator's Pack" where I got most of my sprites. [See the "Full Playthrough" video to see the working sounds]

In addition, a few bugs/oversights were discovered:

- There were two ways to get yourself stuck:
    - Enemy disappearing blocks (removed when all enemies are eliminated) would not be able to be removed sometimes if the player put themselves in a position where they could no longer reach an enemy.
        - I redesigned these levels so that enemies were always reachable if an enemy disappearing block was also in the level.

    - Coins that were needed to unlock the gate to the exit of the level were also out of the player's reach, unless they precisely timed their jump, which was too unreliable and took away some of the fun
        - I also made sure that these coins were now accessible without too much repetitive trial and error from the player.
- The player's jumping state would remain when the next map was loaded. If the jump key was pressed before the player went into the pipe, then the jump would happen as the player left the pipe.
    - I simply reset jump counter to 0 in the "reset variables" function.
    - [See the "Beta Testing – Jump Error Fixed" video]
- In the menus, the "play again" button in the victory/game over screen would sometimes send the player straight back to playing, instead of sending them back to the title screen.
    - This is because the "play" button which appears in the title screen overlapped with the "play again" button, and the way the buttons are programmed, there was no way around this except for putting the buttons in entirely different spots.
    - [See the "Usability Testing – Button Error Fixed" video]

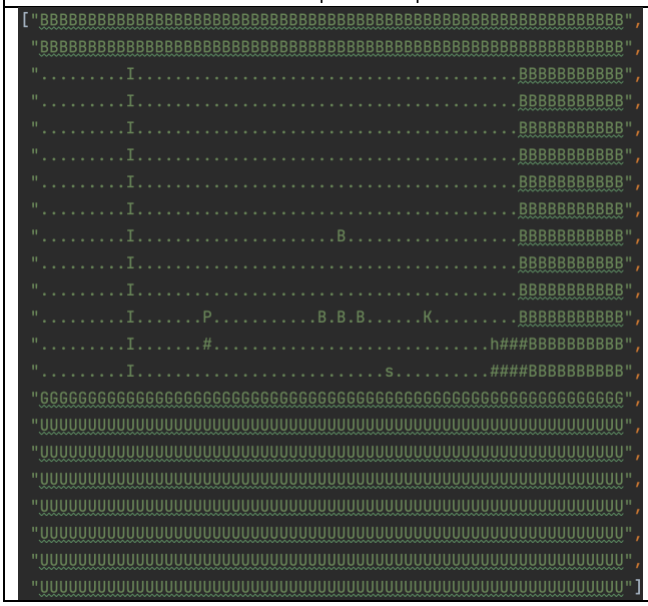
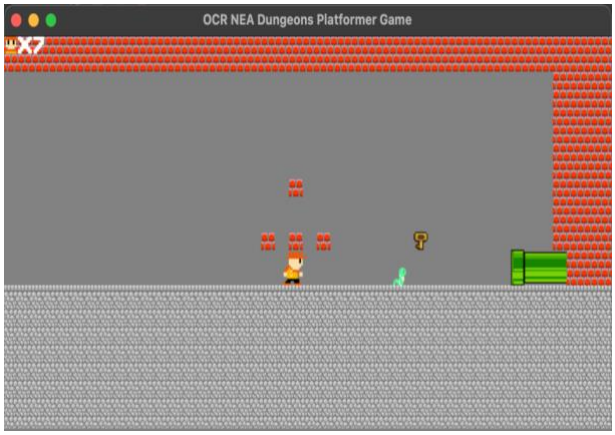
FINAL TESTING CHECKLIST

Action to test	Expected Outcome	Functioning?	Evidence
(1) Menu	The buttons in the menu lead to the screens they're named to lead to.	Yes	Final Testing - Menu Screens
(2) Loading of maps from blueprint to game.	The first map screenshot will match the screenshot of the blueprint for the map	Yes	Below This Table
(3) Horizontal Movement	The player moves left/right when the left/right keys are pressed	Yes	Final Testing 1 – 0:03
(4) Horizontal Collisions 1	The player tries to move into and through a block walking left but is stopped when it collides with the block.	Yes	Final Testing 3 – 0:08
(5) Horizontal Collisions 2	The player tries to move into and through a block walking right but is stopped when it collides with the block.	Yes	Final Testing 3 – 0:04
(6) Vertical Collisions: 1	The player tries to jump up into and through a block but is stopped when it collides with the block. The jump is cancelled and the player lands on the ground.	Yes	Final Testing 4 – 0:01
(7) Vertical Collisions: 2	The player lands on the ground but is stopped when it collides with the block.	Yes	Final Testing 1 – 0:04
(8) Jumps	The player jumps when the up arrow/space bar is pressed. The appropriate sound is played.	Yes	Final Testing 1 – 0:03
(9) Jump Movement	The jump velocity is quick, then slows down as the jump reaches its peak height.	Yes	Final Testing 1 – 0:03
(10) Gravity: 1	The player falls when in the air due to a jump	Yes	Final Testing 1 – 0:04
(11) Gravity: 2	The player falls when walking off a platform.	Yes	Final Testing 2 – 0:00
(12) Gravity: 3	The player falls when hitting the ceiling.		
(13) Enemy Collisions 1: Enemy Death	The player defeats the enemy when jumping and landing directly above the enemy, causing an enemy death, and playing the enemy death animation	Yes	Final Testing 1 – 0:06
~~(14) Enemy Death 2~~	~~If the player's special abilities are used on the enemy, the enemy death animation is played.~~	~~n/a~~	~~n/a~~
(15) Enemy Death Animation	The enemy should change sprite and disappear after a small period of time. The appropriate sound is played.	Yes	Final Testing 1 – 0:06
(16) Enemy Collisions 2: Player Death	If the player collides with an enemy, the player death animation is run.	Yes	Final Testing 4 – 0:06

(17) Player Death 2: Falling off the map	If the player falls into a huge ditch and off the map, the player death function is called	Yes	Final Testing 2 – 0:00
~~(18) Player Death 3~~	~~If the enemy's special abilities are used on the enemy, the player death animation is played.~~		
(19) Player Death Animation	The player changes into its death sprite image, rises for a small bit of time before moving downwards vertically until it falls off the map. The appropriate sound is played. Every other sprite should be paused (e.g., enemies do not move). The level is then restarted.	Yes	Final Testing 4 – 0:06
(20) Restarting of level due to player death	The correct level loading screen is shown. The first map of the level is loaded.	Yes	Final Testing 4 – 0:06
(21) Loading Level Screen	The level number is displayed after the word level, and the lives counter is displayed below that. The number of lives is updated immediately if a player has just lost a life.	Yes	Final Testing 4 – 0:06
(22) Coin Collection	Coins disappear upon contact with the player and play a sound when collected.	Yes	Final Testing 2 – 0:01
(23) Key Collection	Keys disappear upon contact with the player and play a sound when collected. The key appears in the top left of the screen. This key image disappears when a pipe has been unlocked	Yes	Final Testing 1 – 0:15
(24a) Pipe Entry 1	Pressing a right arrow key next to a pipe facing left ~~in the opposite direction the pipe is facing~~ plays the pipe entry animation. The appropriate map linked to the pipe is loaded.	Yes	Final Testing 1 – 0:14
(24b) Pipe Entry 2	A pipe facing in any other direction should not be able to be entered.	Yes	Final Testing 3 – 0:01
(25) Pipe Entry 3	The pipe should only be able to be entered if the player has obtained the key in the map and unlocked the pipe.	Yes	Final Testing 1 – 0:14
(26) Pipe Entry Animation	The player moves "inside" the pipe. Once the player is no longer visible the next map is loaded. All other sprites are frozen when this is happening.	Yes	Final Testing 1 – 0:16
(27) Pipe Exit	If the player has entered a map through a pipe, then it should play a pipe exit animation when the map is loaded.	Yes	Final Testing 1 – 0:18
(28) Pipe Exit Animation	The reverse of the pipe entry animation, the player should come out of the pipe until it is fully visible. All other sprites are frozen when this is happening.	Yes	Final Testing 1 – 0:18

(29) The Enemy Disappearing Block	This should disappear when all enemies in the map have been defeated.	Yes	Final Testing 3 – 0:12
(30) The Goal Disappearing Block	This should disappear when all enemies in the map have been defeated, and all coins/keys have been collected.	Yes	Final Testing 3 – 0:16
(31) Landing on the goal block	The level finished function should run. The player runs off to the next level. All other sprites are frozen during this animation. The next level is loaded. The appropriate sound is played.	Yes	Final Testing 3 – 0:23
(32) Victory Screen	When the player has completed the last level, a victory screen should be displayed.	Yes	Full Playthrough – 2:25
(33) Game Over Screen	When the player has run out of lives, a game over screen should be displayed.	Yes	Final Testing 5 – 0:03
(34) Background Music	The correct background music is played for each level. It changes for each level.	Yes	See Full Playthrough
(35) Background Colour	The correct background colour is displayed for each level.	Yes	See Full Playthrough

Success Criteria 2)

Level 1 Map 1 Blueprint	Level 1 Map 1 Game
	

EVALUATION

SUCCESS OF THE SOLUTION

Key:
White: Criteria Met
Red: Criteria Missed
Yellow: New Criteria (modified from original) Met
Blue: New Criteria Met.

Success Criteria	Completed?
Creating a program window	Yes: Version 1.1
Play button on the title screen that plays the first level	Yes: Version 1.2
Rules button that shows a screen explaining the rules and controls.	Yes: Development 7
Base entity parent class	Yes: Version 2.1
A way to easily define sprite sheets	Yes: Version 2.2
Define ground and block	Yes: Version 2.3
Create a map based on a list of strings	Yes: Version 2.4
Create player sprite	Yes: Version 2.5
Resize everything using one variable	Yes: Version 2.6
Add horizontal moving functionality to the player	Yes: Version 3.1
Create moving camera effect	Yes: Version 3.2
Animate moving functionality	Yes: Version 3.3
Add horizontal collisions	Yes: Version 3.4
Add vertical collisions	Yes: Version 4.1
Apply gravity to the player's movement	Yes: Version 4.2
Add a jump function and animate it	Yes: Version 4.3
Create parent enemy class	Yes: Version 5.1
Allow player to jump on enemies to kill them	Yes: Version 5.2
Create death function for player	Yes: Version 5.3
Create new entity constructor	Yes: Version 5.4
Ability to switch between two maps seamlessly.	Yes: Version 6.1
Make warp pipes to change maps	Yes: Version 6.2
Create end goal for a level	Yes: Development 7
Import actual sprites to be used.	Yes: Development 7
Add all types of enemies, ~~with different abilities~~	Partially: Development 7
Add powerups for the player	Not Completed
Add lives, a life counter, and a loading screen before each level showing the number of lives left.	Yes: Development 7
Add a game over screen when the character reaches 0 lives.	Yes: Development 7
Add a victory screen when the player reaches the end of the game.	Yes: Development 7
Add a key that unlocks pipes	Yes: Development 7
Add coins that ~~could contribute to a score~~	Partially: Development 7
Add blocks that disappear when certain goals have been achieved.	Yes: Development 7

Add Background Music	Yes: Beta Testing
Add Sound Effects	Yes: Beta Testing
Create all ~~8~~ 4 Levels, with multiple maps	Partially: Development 7

Although some of the functionality had to be scrapped due to time constraints, almost all of the functionalities set out in analysis has been achieved - along with the addition of a few new ones.

The most important features: vertical movement, horizontal movement, collisions, and event handling by the player sprite have all been successfully implemented. In addition, the program is fast and has very few delays. The features removed would've maybe added more to the game in terms of variety and skill required, but I am still happy with the project I have created instead.

USABILITY FEATURES

- There are sound effects to indicate to the player when an action happens.
- The title screen has buttons with the appropriate descriptive words of what their function is and what menu they will send the player to.
    - However, the title screen isn't very aesthetically pleasing and if I had more time, or could ask someone with graphic design ability to create graphics, I would do this to create menus that are not just purely functional
- I added a lives counter at the top left to indicate when the player loses a life, and how many lives they have left before the game ends.
- I show the key in the top left when it has been collected to indicate that the player can enter a pipe.

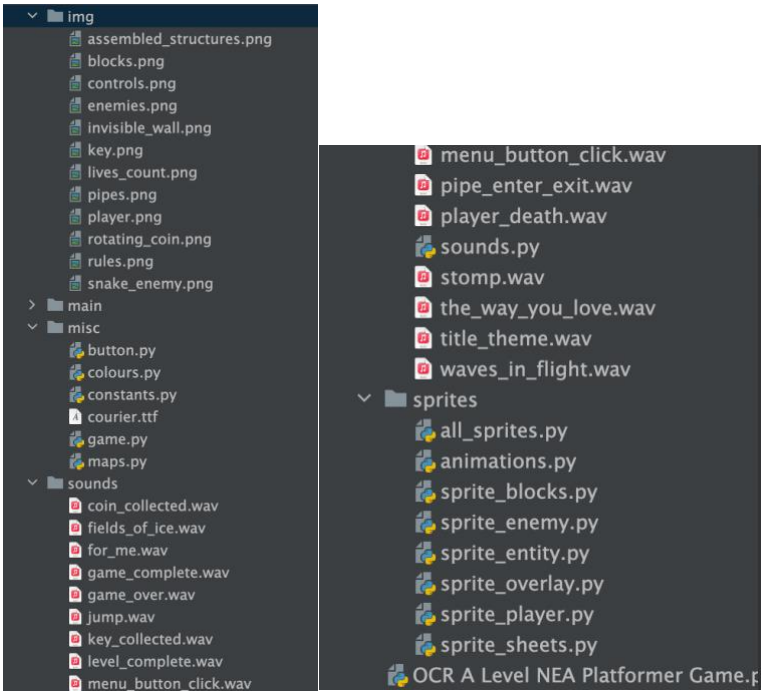
All the evidence for this is in the testing videos.

MAINTENANCE

- The code can be relatively well maintained as comments and docstrings have been used throughout to explain what each section of the code does.
- I have used properly named variables and methods like "get sprite" and "player state" so it is clear what my program is doing.
    - I have also used Enum classes instead of strings or variables to refer to most data like images and sprite sheets, so that the data is more rigidly defined, and it is easier to spot logic errors.
- In addition, the program is split into modules, using separate classes and methods to help better understand the function of the code, as well as to better isolate any errors.
    - The modulation also allows for any new functionality to be added with relative ease, should I want to add any more features.
- I have used type hints for most of my variables in functions to indicate to the programmer what data type the parameters are meant to be.

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

def detect_movement(self):...

def move(self):...

# ____________________________________ANIMATION FUNCTIONS____________________________________
def process_animations(self, animations_enum: Animations):...

def get_animation_sprites(self, coordinates_list: list):...

def animate(self):...

# ____________________________________COLLISION FUNCTIONS____________________________________
def x_collide(self):...

def y_collide(self):...

def specific_entity_collide(self):...

def enemy_collide(self):...

def key_collide(self):...

def coin_collide(self):...
# ____________________________________JUMP FUNCTIONS____________________________________
```

- I have made it very easy for me to import new sprite sheets, enemies, sounds and other entities due to:
    - My Enum classes for sprite sheets and sounds meaning that all I need to do is create an Enum with the file name
    - My base entity and enemy classes meaning that all I need to do is specify the sprite sheet and sprite coordinates.
- I have organised my files and images into different folders for any other programmers to understand the structure of my code.



FURTHER DEVELOPMENT

I believe there is a lot of room for further development. I could:

- Add more levels, which I would implement by:
    - creating a new map id
    - assigning the id, a background colour
    - background music
    - a blueprint
- Adding more abilities to the player to move or kill enemies
    - Adding a power up entity
    - Making specific power up entities
    - Add a “collision with powerup” function to the player
    - Add more player states and sprite coordinates to indicate the change in appearance.
    - Add a key detection for whichever key is selected as the key to activate the power up.
    - Add extra functions to the update function depending on what the power up does.
- Adding more abilities to the enemy to move or kill the player.
    - Adding an extra function or two to the new enemy depending on their ability
    - Adding these functions to the enemy update function to run them.
    - Add collision functions to the player which will lead to a player death.
- Creating a more vibrant home screen/game over screen/victory screen.
- Adding more animations, which I would do in a similar fashion to the ones I already have
- Potentially add infinite world generation
- Add signs or characters in the game that display text when interacted with.