OCR, A-Level, Programming Project, 2021 – 2023 write-up, including analysis, design, development, and evaluation.

Malachy Moran-Tun

Candidate #6178

Table of Contents

3.1 Analysis of the Problem

3.1.1 Problem Identification

Platformer games are one of the most fundamental gaming genres; everyone has played at least one platformer, whether it's the original Super Mario Bros. on the Nintendo Entertainment System, or a much more recent one, such as Celeste[1]. Nevertheless, the central mechanics of running and jumping, getting round obstacles to explore and/or reach the end of a level, have remained integral to gaming.

The retro-styled, indie, platformer market is initially thought of as over-saturated, since it is the first genre of game that comes to mind for new game developers. Because of this insanely awesome but difficult-to-design genre, many have made the connection between bland gameplay, generic graphics and platformers. However, thankfully, games like Celeste, Spelunky[2], Hollow Knight[3], Shovel Knight[4], VVVVVV[5], and

1 Celeste is a game that is commonly referenced throughout this document. It is a 2018 platformer designed, directed and written by Maddy Thorson, programmed by Maddy and Noel Berry. It regarded as one of the best platformers by many, known for graphics, story, and soundtrack, as well as its high difficulty, while still being fair. It has a 97% rating on steam, and 10/10 on IGN.

2 Spelunky is a 2008 2D platformer made by Derek Yu, mixing roguelike elements into platformers.

3 Hollow Knight is a 2017 2D Metroidvania (non-linear exploration and progression, a portmanteau of Metroid and Castlevania, two classic 2D platformers) developed by Team Cherry.

more have brought precise, tight jumps back into the market's interest, with more coming on the horizon, like Pokey Poke[6] and Hollow Knight: Silksong.

Taking Celeste, for example, its primary platforming focus is the ability to dash: a common mechanic, but it was so integral to the gameplay that now games with a dash mechanic are instantly compared to Celeste. I plan to make my "primary mechanic" be a grapple hook, or rope-swing ability, i.e., the player has the ability to swing off platforms, as well as grab and move objects. Certain terrain / ground types may not be able to be grappled on to, however, in order to maintain the platforming challenge.

I intend to create a game in similar vein to the games mentioned above, with a well-planned out difficulty progression, as created by slowly bringing in platforming concepts, without going too far. Quality of life improvements would include: auto-tiling, pixel-perfect collision detection, sprite animations, as well as more platformer specifics, such as variable jump height, buffering inputs, friction (or some amount of acceleration), mid-air turning etc..

The main gimmick of my game, a "grappling hook", from which the player can swing from to reach different parts of the level, will interact with more than just the ground of the level, for example, the player might need to use it to hit a button then swing from a platform in quick succession. Essentially, it will be the central focus, and the level design will be focused on that, as well as all the enemies / level objects.

4 Shovel Knight is a 2D platformer developed by Yacht Club Games, which focuses more on the action side of platformers.

5 VVVVVV is a 2010 2D puzzle-platformer created by Terry Cavanagh, which focuses on difficult platforming and exploration (as well as having a great soundtrack).

6 Pokey Poke is an upcoming (release TBD) 2D puzzle platformer, focusing on exploration and difficulty, with the main gimmick being the use of a spear for platforming.

3.1.2 Stakeholders

The retro-inspired mechanics, music, sound, and graphics may seem generic in an over-saturated market, at first. However, this is simply because it's much easier to create coherent sprite and gameplay design when keeping it similar to older-styled games. Whilst at first, it may seem difficult to be notable because of this, the majority of the target audience will only play a small selection of these retro-styled games, as well as the majority of triple A games. This means that, if anything, it is much more difficult for a generic 3D game with default assets, rather than a well-defined 2D game, to stick out. I intend to create a game to be played on PC / Mac / Linux machines, primarily because they are open for development, and offer versatility in graphics, sound, and control management.

Due to the more complex controls, as well as the planned difficulty, the target audience is anyone above and around 12 years old; even though, traditionally, a game similar may have a lower age rating, even going to 3+, due to the non-violent and inoffensive graphics, I plan on maintaining a "tough but fair" level design philosophy, similar to that of early Mario games (e.g., SMB3 / SMW) and Celeste.

My stakeholder, Onyx B., has been chosen to match this criteria. They have played a variety of platforming games, including Celeste. Therefore, they should be able to provide applicable feedback to aid in the creation and development of the game.

3.1.3 Solving the Problem using Computational Methods

Computational methods can aid in creating this game and helping with any programming, art, sound, and music constraints I may face. The problem itself can be easily managed by breaking it down into its core elements, and slowly tackling each small part at a time. Moreover, abstracting the problem, and thinking logically and concurrently, will also help me throughout the project.

Thinking Ahead

Thinking ahead is the process of identifying important, core information about the solution, which provides the necessary structure to then create a detailed solution. The points below explain the core elements of my solution:

- The game will be created in the "Godot" game engine, which is an open-source, object-oriented, game engine, coded using GDScript, which is a language syntactically similar to Python, with the flexibility of C#.
- I plan on using both a keyboard and mouse to control the game, with the keyboard controls being able to be "rebound", meaning I will have to create the necessary systems to easily change the controls whilst playing the game.
- The game will output to a monitor, as well as have sound. However, I plan on making sound optional, meaning the game must be designed around primarily visuals, with sound being for additional feedback.
- I plan on implementing a score system that is entirely optional, and community driven, meaning it is up to the players playing the game to create challenges based off the information provided from the game.

Thinking Abstractly

Abstraction is the removal of any unnecessary details and elements of a problem for the purpose of keeping the problem simple and easy to manage, whilst still retaining the core, essential details. This doesn't necessarily mean removing features, but rather designing the game's features around this clean and simple thinking. I plan to design my game with the following abstractions:

- 2D – I have never created a 3D game, meaning restricting it to 2D means I'll be able to understand the rendering of sprite and game objects much more easily. It removes an additional axis, making it much easier to design and program.
- Pixel art – hand drawn art, or pre-rendered 2D sprites, are something I have never created; thus, I will design my game around the classic 1980s / early

1990s look, which has become a staple in modern indie titles. This will make it easy for the player to understand what each part represents, as it will force the graphics to have a simplistic art-style.

- "Hitboxes", or collisions for the ground and enemies will be invisible rectangles, or similar shapes, as is common with many games, to allow it to run on lower spec. machines, as well as make it easier to program.
- Menu – provide customisations, such as control options, and accessibility settings – perhaps including an "assist mode".
- Pause button – allow the player to pause the game to do something else, but also provide the player a way of changing options mid-game, as well as exiting the game.
- Gravity will not work consistently, in order to provide more control over the player whilst on the rope, as compared to on the ground, and jumping.

Thinking Concurrently

Concurrency allows for the processing of multiple instructions at the same time. Crucially, this has to be designed into the software in order to take advantage of it. Concurrent processing is not required for this problem, as the game logic can be handled all in one frame ($\frac{1}{60}^{\text{th}}$ of a second), and then displayed.

Concurrency will be required for displaying multiple parts of the graphics at the same time (e.g., the player and the tiles), as well as to handle any music and sound in the game. However, logic to initialise and control these different parts does not have to be handled concurrently, since all logic can be done in a single frame. Similarly, concurrency will allow for multiple sounds to be played simultaneously, meaning the game can play music as well as sound effects, without removing audio channels in the process.

Thinking Procedurally and Decomposition

Decomposition is the process of breaking down a large problem into a series of more manageable problems; the solutions of which will build up the final solution to the original large problem – in this case, the game.

- Due to the object-oriented nature of the game engine I have chosen to use, Godot, decomposition can occur quite naturally, as the problem is already broken down into separate classes, which can be re-used throughout the game.

- Inheritance will also be used to inherit core methods and attributes, most importantly, the position of various objects in the game. This will allow for one object to contain a sprite, collision information, and animation, all of which will follow the same position.
- The player will also use a "state-machine" to switch between two main states: basic platformer movement, and rope-swing movement. This breaks down the overall task of player movement into two smaller, and subsequently easier to solve problems, as well as provides a larger amount of overall control when creating the player.

Thinking Logically

Logical thinking is an important step in creating any effective final product – in this case, a game. Because the game requires inputs to be fetched every frame, as well as "physics" based code, such as movement, gravity, and the rope-swinging code, thinking logically is imperative to a successful product.

- A "main" game loop needs to be processed every frame. This will use delta time to correctly calculate the game's physics, regardless of the framerate.
    - Inside this main loop will be a "state-machine", which will change what piece of code it's running based on a single "state" variable. This allows for completely separate physics logic for the platformer state, and the rope swing state.
- Several conditions for controls must be added to allow for the game to react to inputs.
- Conditions are required to correctly animate the player, and switching between different player states.
- A condition to test whether the player is on the ground will be necessary for jumping logic.
- There must be logic to stop the player's speed from indefinitely increasing.

3.1.4 Research the Problem

Comparison of Retro-Styled Platformers

As very often mentioned in this document, my main inspirations are games like Hollow Knight, Celeste, the classic Mario games (i.e., SMB, SMB2, SMB3, SMW, SML, SML2), and the classic Sonic "trilogy" (Sonic 1, 2, CD, and 3 & Knuckles, as well as their 8-bit counter parts). There is a significant difference between classic and modern platformers, at first sight (mainly the age, of course), but the core mechanics that are integral to the experience are shared, even 30+ years later.

To simplify the analysis of key mechanics, including: how the user experience feels; the graphical user interface implemented; and communication of important story and/or mechanics, I will focus my analysis on Celeste – mainly concentrating on the following important elements:

- The tutorial, and its communication with the player
- The mechanics in Celeste
- The presentation of Celeste, including the user experience and GUI elements

The Tutorial and its Communication with the Player

Celeste's tutorial is extremely simple, and acts mainly as a story element. Celeste's main mechanic is the dash, however, it pairs well with the wall-jumping and climbing mechanic. Since climbing is much more simple and more common than dashing, Celeste opts for allowing the user to test this out in a small screen just after pressing start.

After this brief introduction into climbing, Celeste introduces the story mechanics. Crucially, my game will not include any story mechanics, making it more similar to early platformers, such as the classic Super Mario Bros. series, in order to make the scope of the game much more reasonable.

Nevertheless, seeing the story elements of Celeste demonstrates another important factor: accessibility. Even though it would be more thematically appropriate to use a pixelated font, due to the large amount of text in Celeste, having an easy to read, vector-based, sans-serif font will avoid any problems for players with visual or reading difficulties (e.g., dyslexia).

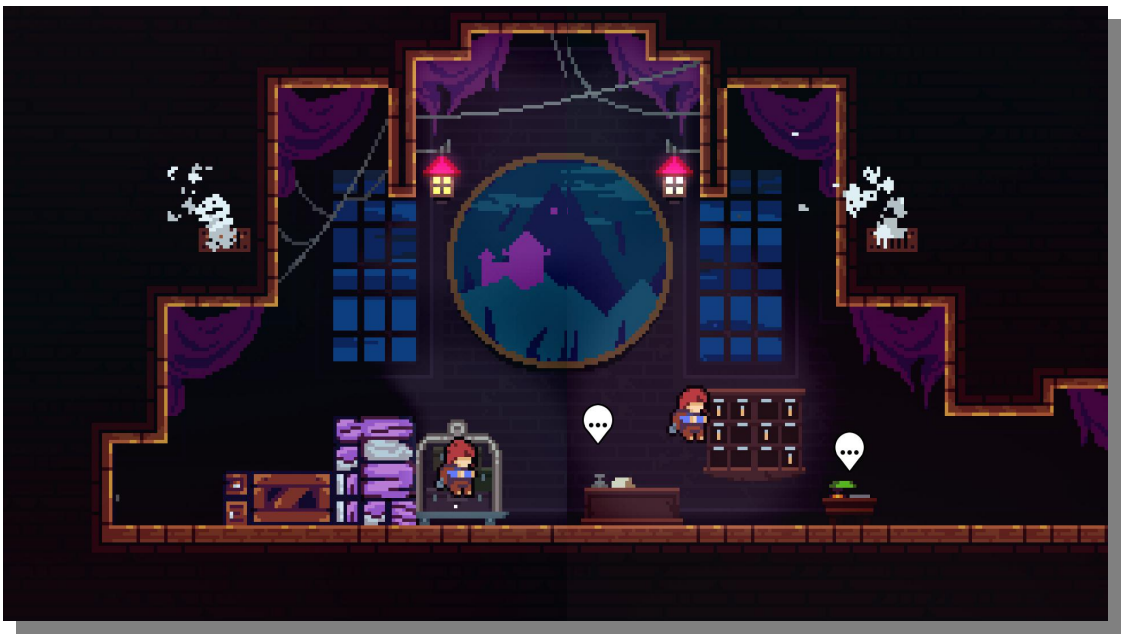
Afterwards, the tutorial continues with a more fast-paced section, involving fairly easy jumps, just to gradually introduce the player to the action side of Celeste's gameplay. Crucially, this is where the dash mechanic is introduced: Celeste's main "gimmick" (just as my game's main gimmick will be the rope-swing mechanic). An important thing to note is that the dash mechanic is disabled until it is required during this short cutscene:

Once this cutscene occurs, the player can finally use the dash mechanic. This short tutorial is well integrated with the story, but also acts as a small level, allowing the player to freely play around with the mechanics until they are comfortable with using them when necessary. It is not until the first chapter of the game where the difficulty increases to a significant level.

The Mechanics in Celeste

Each chapter of Celeste has its own special mechanic, yet, in order to avoid too much scope, I will focus my game on only the main gimmick: the rope swing. However, there are some important general platformer mechanics that I plan on implementing from Celeste (and other platformer games).

The jumping mechanics of Celeste include variable jump height, i.e., holding the jump button for longer means jumping higher. This has been a staple of platforming ever since the original Super Mario Bros., making it extremely prominent in almost all platformer games. Without variable jump height, the lack of control often feels "off".

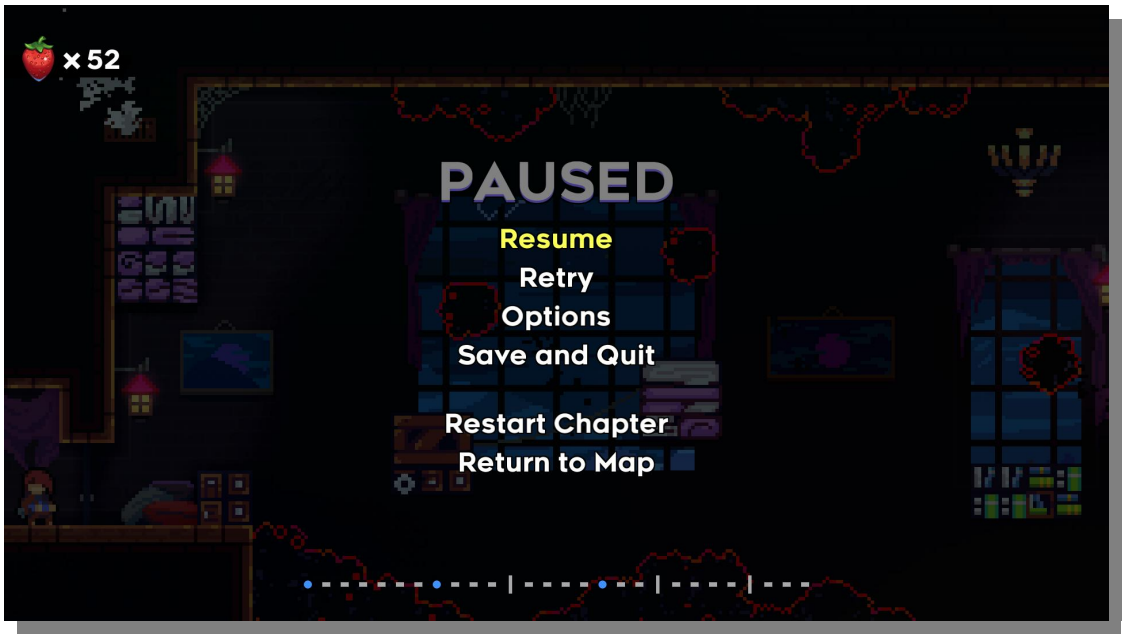


Above shows the minimum and maximum height the protagonist, Madeline, can jump. This is all controlled by the player, depending on how long the jump button is pressed. The immense amount of control this gives to the player cannot be understated. Many more of these important jump mechanics, such as coyote time, a jump buffer, air brake, a higher down gravity etc., can be found in a wide variety of platformer games, however, the majority of this will be explained in the design portion of the document.

The Presentation of Celeste, including the User Experience and GUI Elements

Celeste's visuals are outstanding, particularly due to the balance it achieves from mixing pixel art with modern particles. However, it also keeps a clean look by hiding unnecessary GUI elements from the player, meaning most screenshots (as shown above) contain no GUI at all, meaning only important parts of the GUI are shown when they need to be shown.

The above screenshot shows no GUI at all, meaning all the user has to focus on is the gorgeous visuals, as well as the platforming action. However, this does not mean there is no GUI at all in the entirety of Celeste. For example, if the game is paused, the number of strawberries (the main collectable of Celeste) is shown, as well as a pause menu.



Interview Based on Analysis of Celeste (#1)

To get a good idea on what people, not just myself, like about platformers, I will conduct a 9-question interview on 2 people in my target audience, but still with a variety of ages, to get a wide variety of opinions from a long period of gaming.

- Interviewee 1: Justin T. – Played games mainly in 1980s
- Interviewee 2: Onyx B. – Plays modern games frequently

Question 1: What are you favourite platformer games?

Justin T.:

"[...] one that comes to mind is "Boxer" on the Acorn Electron, since it was the first one I ever played [...] oh! Prince of Persia!"

Onyx B.:

"Personally, I'm a big enjoyer of games such as Celeste and Kirby Super Star"

Question 2: Why do you consider said games your favourites?

Justin T.:

"Well Boxer isn't really memorable for its gameplay, it's memorable since it's my first. [...] Prince of Persia? Erm, well I suppose I like the isometric view (I suppose it isn't very isometric, though). I suppose I like how you can grab things, like... I like how you can climb up and down things, unlike other platformers, I suppose. And how some of the platforms give way, so it opens up another part of the world to explore... and there's also some fighting in it(!). I suppose it's not very much fighting, but it's there's a boss every now and then. I suppose it isn't really a boss, really, but a guard."

Onyx B.:

"Well, Kirby, for example, he's just a little guy. A funky little guy. I mean yeah, I can complete it, and I can play large amounts of it without raging, because I'm not the best at these types of games(!). So, the fact that I can actually succeed on it makes it much more enjoyable. With games like Celeste, the spectacular visuals of the games, the heart-wrenching story of the game really grabs my attention. So, even if I struggle with Celeste, it makes me want to continue trying until I do beat it, no matter how long it takes. I wouldn't call it unfair."

Question 3: In terms of platformer mechanics, what do you consider the most important?

Justin T.:

"Erm, I think having variable jumps is important. Like a small jump. A walking jump, and a running [jump…] that's the most important thing, isn't it? […] Being able to climb up and down is important, since some of the other games didn't have that ability"

Onyx B.:

"I mean wall jumping is an obvious simple extra thing to add more opportunities and depth […] Variable jump height, again it really adds more detail, you know. I feel like it gives me more chances to fail, which sounds odd but I kinda like it."

Question 4: Following your previous answer(s), why are they so important?

Justin T.:

"Well [variable jumps] are important because you can make the world varied, otherwise every single jump would have to be the same distance – the gaps between the platforms would all be the same. Erm, I've played games without an up and down feature, as such, where jump was a way to get up to something, but you couldn't actually go back down, so you couldn't go back to get something, like a collectible, or treasure, or key, or something."

Onyx B.:

"I don't really know what else to say for that [variable jump height]. It's not really noticeable as a massive extra thing, but it just adds more, so you're not doing the same, simple, basic things over and over again […] Wall jumps accomplish the same goal."

Question 5: What platformer mechanics do you see often and dislike?

Justin T.:

"Because I find it difficult to do, I don't like jumping off vertical walls [wall jumping]. In all games really, I just haven't been able to do it. […] There was a game on the BBC [Micro B] called "Wizzy's Mansion". I think you had 5 lives to start with, but at certain places, […] if you fall more than the height of one [screen] […] it was possible to lose all 5 lives in one go.

Onyx B.:

"To a small extent, wall jumps, but I appreciate their existence, nevertheless. Personally, for me, I dislike timed parts of the games. Like parts where I have to react very quickly, and if I miss something for a split-second, that's it."

Question 6: Following your previous answer(s), why do you not enjoy them?

Justin T.:

“Vertical wall jumping, I just found it difficult to do – I just keep falling back down again. Wizzy’s Mansion’s thing was also very frustrating. Especially in a game where you couldn’t save your position.”

Onyx B.:

“I just have, as I mentioned previously, a very slow reaction time, and it’s also a lot easier for me to miss important parts. That’s it really”

Question 7: Are there any interactive mechanics in platformers that have stood out to you?

Justin T.:

“Errm, I suppose I like the puzzle solving element, where you have to find certain objects and use them in another place. Trampolines! They can increase the height jump [that was previously mentioned].”

Onyx B.:

“Blocks where you hop on them, and then after a certain amount of time they fall below you. Yes, they are timed, but they’re still easier to get past. So, it adds more, but with less frustration.”

Question 8: Do you find that the difficulty of platformer games is well-balanced - why?

Justin T.:

“No… I think they can get very difficult very quickly. Some of the older games don’t really have levels that are truly different – they just make it “harder”. So, they just speed things up, don’t they, or just give you less time to do it. And it gets to the point where you have to be a very fast player to beat a level. But then the ones where the levels are different, they’re not necessarily in order of increasing difficulty. They usually start of easy (on level 1), then you could have level 2, which is more difficult than level 3. […] Mineshaft? […] is a single screen per level platformer (which most retro games are). But, on this one, you can skip levels, […] you can just play the levels you like.”

Onyx B.:

“It depends on the game. [...] Kirby is made so that it’s easier to play for those who aren’t good at those kinds of games. On the other hand, Celeste is a lot more difficult, but I think that the difficulty with that is an integral part of it.”

Question 9: Do you find that platformer games are accessible to all types of people - why?

Justin T.:

“It depends on the level of difficulty. [...] Colour blindness I don’t think is generally a problem in most platformers since there is generally a contrasting colour between what’s a solid platform and what isn’t. Erm, I don’t think, erm, a hearing-impaired person would have a problem.”

Onyx B.:

“Well, there’s the obvious issues with sound and deaf people. So... I think it’s useful to not have any parts of the game depend on sound, obviously y’know. Erm, yeah, sound effects are obviously very nice to have – they improve the experience, they stimulate a sense. But those with hearing difficulties shouldn’t be at a disadvantage. [Photosensitive] epilepsy as well. Not necessarily removing all forms of flash, but at least adding a warning. [...] And then also language difficulties, [...] you’ve gotta consider the languages it’s available in and translate it.”

Question 10: Anything else you’d like to add that’s related?

Justin T.:

“The most important part is the controls: if that’s good, the second most important part is level design; if that’s good, you’ve got a good game.”

Onyx B.:

“I want some emotional attachment. I want some sentimentality. I wanna feel connected to it. [...] it could just be in a really simple, small form. But a little bit of sentimental value real raises the game up for me.”

Review of the Interview

From the above interviews, I have ascertained the following information:

- Wall jumps are difficult, particularly if they are not programmed well – they can either be a well appreciated challenge, or a complete turn off from a game.
- Unique platforming mechanics, such as grabbing onto surfaces, are appreciated, and especially help to stand out in the market.
- Boss fights are appreciated where relevant, but platformer games should not focus on them, unless the entire game is planned around it.
- Variable jump height is integral to platformer games; without it, the controls are poor, and the levels become boring and monotonous.
- The difficulty of platformer games is rarely well balanced: either bad programming leads to unfair deaths, or the difficulty curve is irregular or extremely steep.
- Sound and colours should not be essential for a platforming mechanic to work, but be an additional, non-necessary, layer – accessibility options should be provided for any mechanics that may run into this problem.

3.1.5 Specify the Proposed Solution

Interview based on the Review

After giving a brief summary of the main idea of the game, I asked the following questions to one of my interviewees.

Question 1: What types of control options should be available?

Onyx B.:

"Keyboard and mouse – it's a good, simple combination; you don't need a fancy controller or anything extra"

Question 2: What should the primary focus of each level be? For example: reaching the end of the level, defeating all enemies, exploration, etc.

Onyx B.:

"I think reaching the end of the level should be the main focus"

Question 3: Following your previous answer(s), why should each level be like this (if at all)?

Onyx B.:

"Defeating all the enemies is cool, it's fun, but y'know, it's about jumping on platforms: it's a platformer, you gotta focus on the platforms. Enemies are extra... superfluous, as one might say"

Question 4: What type of graphical style should the game be? For example: pixel art, hand-drawn etc.

Onyx B.:

"I want some pixel art – gimme those sweet, sweet pixels"

Question 5: Should a sound effect be played for every player action, e.g., jumping, landing on the ground, or should sound be more subtly used?

Onyx B.:

"More subtly used; I don't wanna have a headache by the end of me playing the game. They're nice, but can be annoying if overused"

Question 6: Should any collectibles be in the game?

Onyx B.:

“Yes”

Question 7: Following your previous answer(s), if yes to collectibles, how important should they be to the gameplay?

Onyx B.:

“Not crucial, but an extra thing to keep you wanting to play more”

Question 8: Should there be some way to compare “scores” between players? For example: time, death count etc.

Onyx B.:

“Yes – death count, I like the nice death count in Celeste”

Question 9: Following your previous answer(s), if yes to scoring, how much of a focus should it be compared to just completing the level?

Onyx B.:

“Not a main focus at all. Just a way to see how well I completed it, and add a bit of competition between me and any other players… and to mock my previous self… for my [bad] playing”

Question 10: Would you like to see any multiplayer options, or should it remain singleplayer?

Onyx B.:

“Singleplayer – I feel like multiplayer doesn’t entirely fit the concept of the game, so it’s nice to compare with other people, but not play at the same time”

Review of the Interview – Establishing the Main Features of the Game

From the interview, I ascertained the following features for the game:

- Controls
  - The controls should be keyboard and mouse, since it is much simpler to control and easier to understand.
- Level
  - The main focus of the level should be to reach the end, therefore, progression through the game is linear, and does not require exploration in order to complete it.
  - Non-linear elements of the level can be introduced using collectibles, but these should not be the main focus.
  - Enemies, and other obstructions, are not necessary, as long as the platforming is diverse enough.
- Graphics and Sound
  - The graphical style should be akin to retro games from the 8 to 16-bit era, i.e., a pixelated style, with bright and vivid colours.
  - Sound should be implemented but used subtly. It should not be necessary to hear for progression, but rather as an added bonus.
- Scoring
  - Scoring should be primarily based on death counts, similar to Celeste. It should not be a primary goal to have low death counts, but rather an additional challenge players can decide to take themselves.
- Multiplayer
  - Multiplayer support will not be added so the singleplayer experience can be more refined

Proposed Solution

Feature	Justification
The player will be based off a two-state state-machine, meaning different mechanics will be used between the "platforming" state, and the "rope-swinging" state	This will allow for much more fine control over the mechanics of the game during its development, as well as make debugging much easier, since a bug will only exist in one state at a time
The player will be able to "shoot" a grapple towards the direction of the mouse. If it hits part of the terrain, the player's state will be changed into the grapple state	This is a simple solution that allows for an precision and control, since the direction of the grapple can be easily changed using the mouse, instead of being locked to 8 directions using the arrow keys
The player will be able to control the angle and the length of the rope after it has "hit", and the state has been changed	This, once again, is all about providing control to the player. This will provide the main part of the platforming challenge, as basic platforming (similar to Mario games) is not overly necessary for the game.
The player will be able to run and jump in the initial "platformer" state	Whilst basic platforming isn't strictly necessary (since the main gimmick could be utilised at all times), it is still a useful feature to have as it allows the levels to become much less limited, from a design perspective, as well as providing another layer of thought for the player
Collectibles, such as coins, or other types of power-ups, should be added to the levels	Whilst not strictly necessary, collectibles provide another layer of depth in the game; instead of just getting to the end of the level, different players could challenge each other with tasks such as "collect as many coins as possible within 2 minutes". This creates a community within the game. Crucially, these are not going to be challenges implemented within the game, due to time limitations.

The game should keep track of how many times the player has died: per level, and overall	This, similar to collectibles, provides the community another layer of depth to challenge each other. A death count is better than a lives system, however, since it eliminates the need to “hunt” for 1-ups, or additional lives, making the game much less frustrating, and something players cannot focus on, if they do not wish to
The “camera” in the game will focus on a central point between the player and the mouse position	This allows for controlling the player to be easier, since moving the mouse will allow looking ahead, and aiming at a piece of terrain far away in the level, providing a larger set of level-design options
When launched, the game should open with a menu, as well as have a sub “options” menu	This will allow the player to not have to immediately start playing the game, as well as change any important options, such as music and sound volume
Pausing the game should bring up the options menu	This will allow the player to change important settings without having to completely restart the game, such as whether the game is fullscreen or not. The pause menu should also offer the ability to exit the game to the main menu, and to the desktop
Controls should be able to be “rebound” to different buttons on the keyboard	This will allow the player to play with their preferred keys. For example, some may prefer the arrow keys, whereas others may prefer using W, A, S, D
Custom background music should be present throughout all levels and menus in the game	This will provide the correct “atmosphere” for the game, since a completely silent game would simply lack any character, feeling very empty and unfinished

Sound effects should be present, but subtly used in the game.	This will provide feedback to the player for performing different actions, which is integral in any sort of game. They should be subtly used in order to not overwhelm, distract, or confuse the player
GUI to display the number of collectibles collected, as well as the death count. The GUI should hide when the player begins moving	This will allow the player to visually see their current statistics. The GUI hides in order to not distract the player, as well as creating a much cleaner look

Hardware Requirements

The minimum hardware requirements are based upon Godot's (the game engine) requirements. Any requirements listed are the minimum, and should, therefore, work with more advanced hardware.

Component	Requirement	Justification
CPU	2.5 GHz Dual core 32-bit	Whilst the game will be coded using single-threaded logic, a dual core CPU is useful for processing collisions and other built-in functions in Godot. 2.5 GHz will be sufficient to run the game, as the logic is very simple, and most of the CPU power will be taken up processing libraries.
GPU	Support for OpenGL 3.3 Integrated Graphics: Intel HD Graphics 2500 Discrete Graphics: AMD Radeon HD 7000 NVIDIA GeForce 8 Series	The game requires the use of OpenGL 3.3 libraries; thus, any graphics cards must support the use of OpenGL 3.3. All modern graphics cards support higher versions of OpenGL, so the requirement is not difficult to meet.

RAM	4GB	The game itself is very small; it and its libraries will occupy a small portion of RAM, however, 4GB is required to allow the operating system to process any functions while the game is running.
Storage	50MB	The game should not occupy more than 50MB of storage. This excludes any pre-installed libraries, such as OpenGL 3.3+.
Input	Keyboard and mouse	The game only supports a keyboard and mouse control scheme; therefore, a keyboard and mouse must be present to play the game.
Output	60Hz Monitor	The game will be designed to where sound is optional, meaning, only a monitor is required to view and play the game. The game will be designed to run at 60 frames-per-second, meaning a 60Hz (or greater) monitor will be required. Speakers or headphones are recommended but not required.

Software Requirements

The minimum software requirements are based upon Godot's (the game engine) requirements. Any requirements listed are the minimum, and should, therefore, work with more advanced software.

Component	Requirement	Justification
Operating System	Windows 7 MacOS 10.10 Linux with X11 or Wayland 32-bit	The game will have both a 32 and 64-bit export, meaning it will run on almost any modern Windows and MacOS machine. All Linux based machines should be able to run it also, although Unix-only machines will not.
OpenGL	OpenGL v3.3	The game's graphical back-end is handled with OpenGL 3.3, thus OpenGL 3.3+ is required to handle the game's graphics

Success Criteria

No.	Requirement	Justification	Reference
1	Player can run and jump using left, right, and jump controls	Basis of the platformer genre, allowing the player to traverse the various levels	Celeste analysis
2	Jumping triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
3	Player can click on a tile to switch to the rope state	Allows the player to use the primary rope-swing gimmick	N/A
4	Switching state triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
5	Player can swing the rope using left and right controls	Allows the player to control the main rope-swing gimmick	N/A
6	Rope is affected by gravity in a “pendulum” effect	Provides some challenge to the rope-swing gimmick, as well as adding a small sense of realism	N/A
7	Player can switch back, from the rope state, to the platformer state using the jump button	Allows the player to switch back into the normal state to make simple movements	N/A
8	Switching back to the platforming state maintains momentum	Allows the player to swing across gaps further than the rope can go	N/A

9	Switching back to the platforming state triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
10	3 unique levels	Allows the player to experience all the mechanics the game has to offer	Interview #1
11	First level should introduce mechanics to the player	Provides a safe level to allow the player to experiment with all the mechanics of the game	Celeste analysis
12	Coin collectibles can be found throughout each level	Allows for optional competition between players	Interview #2
13	Coin-count increases when the coin is collected	Allows players to compare coins collected for said competition	Interview #2
14	Coin-count is saved when the game is exited	Allows the player to continue game at a later point with their progress	Interview #2
15	Collecting a coin triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
16	Checkpoints throughout the level	Allows for larger levels, rather than many small levels, without unfair difficulty	Interview #1
17	Colliding with the checkpoint activates it and deactivates the other(s)	Allows for levels to have multiple checkpoints, rather than just one	Interview #1

18	Colliding with the checkpoint triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
19	Player returns to the checkpoint if they die	This is just how checkpoints work.	Interview #1
20	Death-count increases if the player dies	Allows for optional competition between players	Interview #2
21	Death-count is saved when the game is exited	Allows the player to continue game at a later point with their progress	Interview #2
22	Dying triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
23	GUI showing coin-count in the top left	Allows the player to actually view the coin count	Celeste analysis
24	GUI showing death-count in the top right	Allows the player to actually view the death count	Celeste analysis
25	Respective part of the GUI is shown when the player collects a coin / dies	Keeps the general game clean, whilst still providing important information when necessary	Celeste analysis
26	Transitions when the player enters and exits the level, dies, or the screen changes	Clearly establishes the end of the level, as well as making screen changes look nice	Celeste analysis
27	Main menu allowing the player to start the game or change options	Allows the player to launch the game without immediately playing.	Celeste analysis

28	Options menu allowing the player to go fullscreen and change the volume of the game	Allows the player to change any important options before the game begins	Celeste analysis
29	Pause menu, allowing the game to be temporarily paused, and allowing the player to exit back to the main menu	Allows the player to take a short break, and then resume the game from the previous place	Celeste analysis
30	Player can rebind the up, down, left, right, and jump keys	Allows the player to use any preferred keys or keyboard layouts	Celeste analysis
31	Sprites have a pixel art aesthetic	Keeps consistency throughout the game, as well as matching the retro inspired mechanics	Interview #2

Additional Successes

The success criteria below details any criteria that would be beneficial to add, although is not strictly required for the game to be to a suitable standard. These are to only be added once the previous success criteria is fully complete – essentially optional and non-essential (usually quality-of-life) features.

No.	Requirement	Justification	Reference
1	Options can be changed whilst the game is running via a menu that appears after pressing pause.	Allows the player to change any settings, most likely key-binds, after they have gathered how the functions of the game work.	Celeste analysis
2	Two tilesets: one which the rope can grapple onto, and one which cannot be grappled onto	Provides a broader set of tools to design levels with, as well as more complex level-design.	N/A

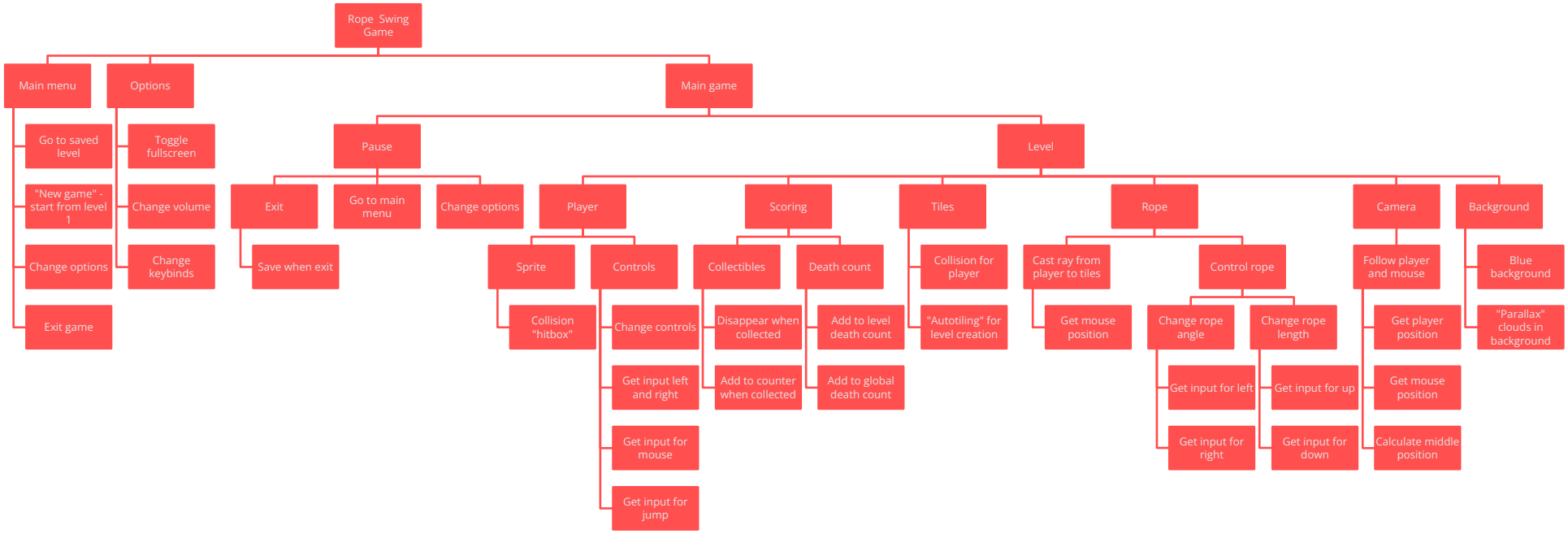
3	Controller support	Additional control methods are always a positive thing to add, particularly as it demonstrates if the game could be suitable for platforms other than PC / Mac.	Celeste analysis
4	Online multiplayer	Whilst co-operative multiplayer is unnecessary and not part of the game's scope, and local split-screen multiplayer would be too difficult to control due to the small screen size, online multiplayer would allow for multiple players on one stage with their own controls and full-sized view.	Celeste modding
5	Pixel-perfect viewport scaling	Creates a more succinct style to the game, as it will scale the game according to a pixel art style, meaning that even rotations will still line up with a pixelated grid.	N/A

6	Unique style	A non-generic style goes a long way for a game's branding and overall identity, so having something that isn't like the first level of a Mario game allows the game to stand out.	Problem identification
7	Social media integration	This allows players to share scores easily	N/A

3.2 Design of the Solution

3.2.1 Decompose the Problem

Systems Diagram



This systems diagram creates a visual representation of the main features of the game to be implemented. Due to the hierarchical nature of the diagram, the solution outlined can easily be transferred into an object-oriented game engine, of which I have decided to used Godot, as mentioned in the “thinking ahead” section of the initial analysis. This means that each module of the diagram will be a separate problem that needs to be tackled during the development of the game. Planning this out initially sets out the problem in an easy-to-understand way, allowing for a systematic approach to creating the final product.

3.2.2 Describe the Solution

Explanation of the Systems Diagram

The table below contains the main features of the above systems diagram, as well as a brief explanation of the core functionality I plan on implementing.

Component	Explanation
Main menu	The main menu will be the first screen the player is greeted with and will offer some necessary functionality. Having this initial menu means the player is not "thrown into action" straight after launching the program, creating a more enjoyable experience for the player. Moreover, the ability to access options is integral in case the defaults are incorrectly set, which is inevitable for some users.
Options	The options menu will be accessible from two points: the main menu, and while pausing the game. This will give the player the ability to change any necessary options, such as whether the game is fullscreen, or the volume, before, and during gameplay.
Pause	The pause functionality will enable the player to briefly stop the game whilst playing. A new pause menu will allow the user to exit the game, as well as change any options, as previously mentioned. This is integral to a positive user experience, as well as creating a much more polished product, as having a formal way to exit the program is superior to clicking on the close button in the windows environment.
Player	The player object is where majority of the game logic will take place. Since the player is a controllable object, a visual "sprite" is necessary to provide feedback to the user. In order to control the player, inputs will be taken and processed, which will then be used elsewhere in the code for running, jumping, and rope-swinging.
Scoring	The game will score the player in two categories: the number of collectibles found, and the death count. Both will have to be stored in global variables that do not reset after a level has been completed. These will be saved when the game is exited, and loaded once the game is started back up.

Tiles	The levels in the game will be made up of individual “tiles”, which the player needs to be able to collide with. Additionally, to create high-quality looking levels, auto-tiling features will be used to “join” tiles together where necessary, creating a seamless look.
Rope	Similar to the player object, the rope will act as a separate state which the player will follow when in said state. This will allow the user to control the rope’s angle and length. The rope also needs to be able to be created, meaning the mouse input and position needs to be taken.
Camera	The camera will try to keep both the player and the mouse on the screen at the same time by taking the mean average of each position. This will allow the levels to be expansive – more than one screen – as well as make it easier for the user to “aim” the rope at platforms in the level.
Background	A background will be added to create a sufficient aesthetic in the game. “Parallax” scrolling will be implemented, which provides pseudo-depth within the game, making the background feel more significant to the player.

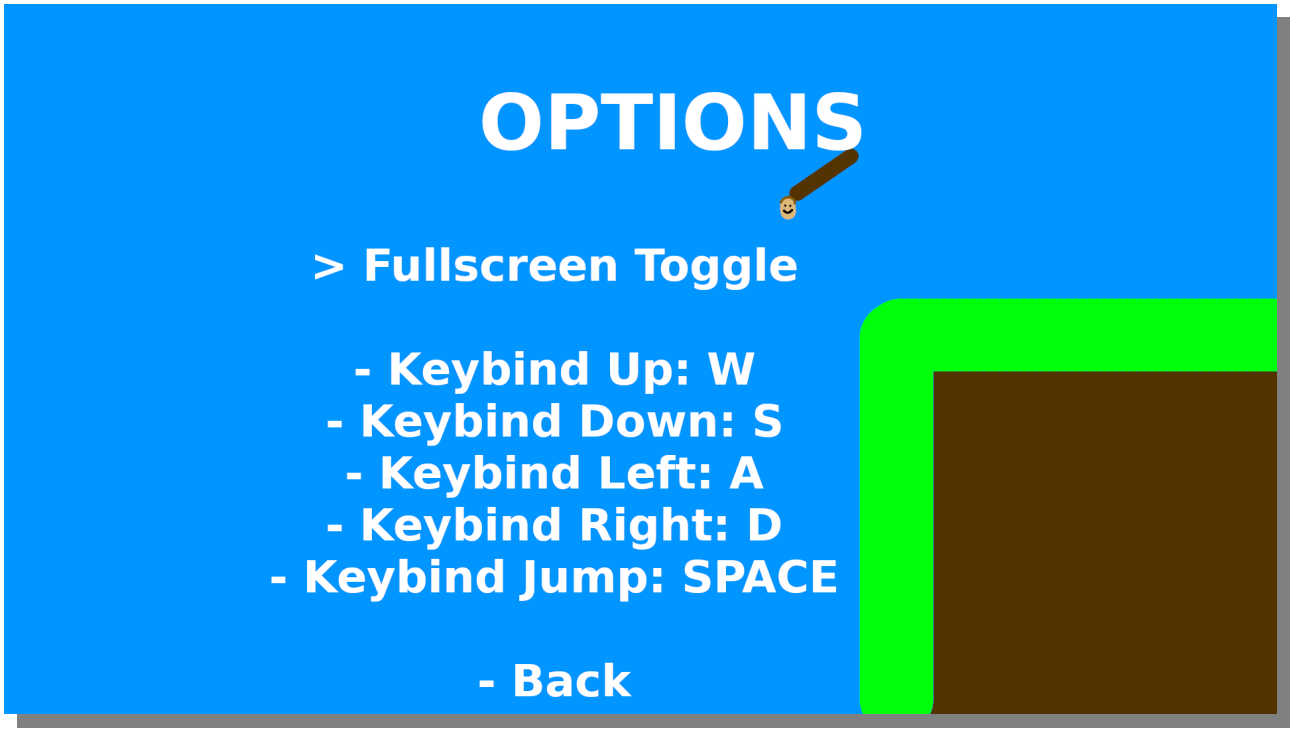
Design of the Menu

When the player first launches the game, they will be greeted with a menu. This is a graphical user interface (GUI), which allows the user to change any necessary settings, as well as play the game. Below is a design mock-up (i.e., very rough and basic version) of the menu design I intend to implement.

The above is a design mock-up of the main menu. The general layout will be replicated; however, the name and overall design is subject to change (obviously). The menu is simplistic to allow the user to easily understand what each option entails and contains a small image of the player sprite to add some additional personality to the menu screen, instead of a blank image. In game sprites will be used in the final game.

The new game button will lead the player into the first level. I plan on having a linear progression, meaning there will not be a "world map" screen. Continue will lead the player into the last level they played, meaning it will be missing when the game is loaded for the first time. The options button will lead to a new menu consisting of basic settings, and exit will end the game. To select an option, the currently bound up and down keys can be used. The ">" character represents which option is currently selected in this mock-up.

Design of the Options Menu



Above is a design mock-up of the options menu. This will be separate from the main menu in order to make it easier to distinguish between. The options menu will have options to change the volume of music and sounds separately, as well as toggle between the window being fullscreen, and in "windowed" mode. The change controls option will lead into another menu, to allow a better visualization of the control

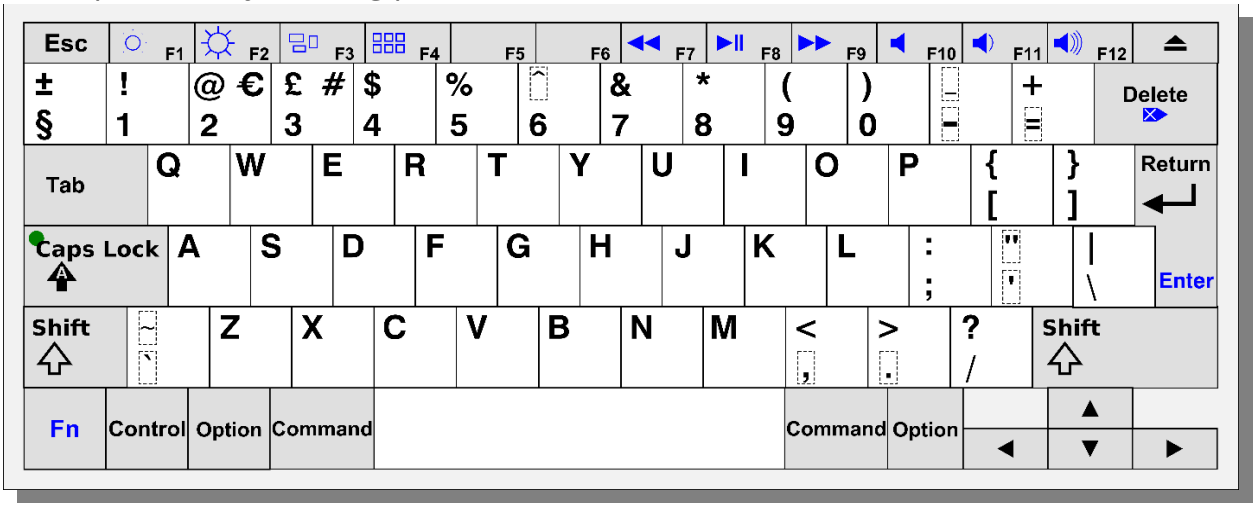
differences. Once again, in game sprites will be used to create a much better looking, more succinct menu.

By pressing the enter key, the user will be able to rebind all controls. However, since a mouse is required, there is no additional setting to change the use of the mouse pointer, only the mouse button.

Controls

One of the most integral parts of user-end design is the default control scheme. Essentially, this will dictate the most frequently used control scheme, as the majority of players will not feel the need to modify it. If this default is poor, it will result in a poor user-experience, therefore it is essential to provide simple to understand and easy to modify controls.z

The aforementioned target audience, that being 12+, are likely to own a netbook sized computer, meaning the keyboard layout is that of the Apple Macintosh Keyboards, or those derived from the minimalistic design. This means that there is a high chance that the arrow keys have a smaller layout, making them more difficult, and frustrating to use, particularly for long periods of time.



Above is a diagram of the current Apple Macintosh Keyboard layout. Due to the restrictions previously mentioned, I have designed the controls to use the W, A, S, and D keys as a substitute for the arrow keys. These full-sized keys are much more suitable for long periods of gaming, as well as are more suitable for right-handed individuals (of which majority of players will be) since it will be much less cramped if playing with little desk space. It is this reason why rebinding controls is still necessary, as it will allow left-handed individuals with full-sized keyboards to more comfortably play the game.

However, default controls must be suitable for all players, thus W, A, S, D will be used.

The game will require additional controls besides four directions. These additional controls will also need to be able to be rebound to other keys, particularly for accessibility and preference. For example, the spacebar will be used to jump, and to "exit" the rope swing. However, many people will prefer to use W or the up-arrow key as jump, so clicking the right mouse button will also "exit" the rope swing.

However, it is impossible to provide the ability to rebind mouse movements, which are integral for the precision required when grappling onto various targets. Because of this, only buttons will be able to be changed in game.

One of the most important points that was raised during the interviews was the problems with the difficulty of platformers. It is integral that I don't add additional artificial difficulty due to bad controls.

Movement

The most important part of a platformer is, as expected, the platforming physics, and movement. This is why it is essential to plan out how the movement code will work ahead of time, as well as to plan out what different aspects of the movement can be programmed at a later time. This creates a natural ranking of the most important parts of platformer physics, especially when making a fun to control, but still suitably difficult platformer.

Many of these small details can be found in the earliest of platformers, including the original Super Mario Bros., all the way up to modern titles, such as Celeste, or Super Meat Boy.

Running

When dealing with the player's movement in a platformer, there are 3 important aspects to consider: the maximum running speed, the acceleration, and the deceleration. Although not entirely realistic, it is extraordinarily helpful to have separate values for acceleration and deceleration; a long acceleration and short deceleration can exaggerate the player's speed, such as in games like Sonic the Hedgehog, whereas a short acceleration and long deceleration can emphasise the "slipperiness". None of these configurations are bad, but they need to be complimentary to the level design, otherwise the movement will not fit with the platformer.

For my game, I plan on basing my movement similar to Celeste's, which includes a fairly short acceleration, and an immensely short deceleration, in order to give the player more control. This emphasis on player control makes Celeste feel "tight" to play, meaning if the player fails, it never feels unfair, which was a requirement of mine based on the interviews. Because I want to emphasise the usefulness of the rope-swing mechanic, the maximum player speed whilst on the ground will be relatively low, allowing swinging on a rope to have a greater prominence.

Jumping

Even though the main USP, or unique selling point, of my game is the rope swing mechanic, it is integral that the jumping is well designed, as much smaller actions will be entirely reliant on an easy-to-use jump, that is well suited to the rest of the level design. Similar to running, there are several variables that control a well created jump in a platformer: the jump height, the jump duration (i.e., the gravity), the air acceleration, air control, air brake, and the down gravity. Many of these are self-explanatory, but some are more subtle.

Air control dictates how easy it is to move the player whilst in air, whereas air brake dictates how easy it is to slow down the already moving player in the air, essentially air deceleration.

For my game, because much of it relies upon chaining inputs together to swing off multiple pieces of the level, the jump height has to be high enough to be able to see your next target, but low enough that the rope doesn't become redundant. Moreover, air movement – i.e., air acceleration, control, and brake – have to be easy enough to manoeuvre but not make the rope controls redundant.

Many newer platformers implement a "down gravity", which is a gravity multiplier that affects the gravity when travelling downwards. This can make jumps much easier to land, particularly if air brake is low. However, due to the main mechanic of the rope, it is important that the player is in the air for as long as possible, therefore this will not be implemented, as it would make chaining rope-swings much more difficult.

Assistance

Platformers can often feel unfair if small and subtle tweaks aren't added to bias the game in the player's favour. These tweaks have to be subtle enough as to not make the game easier, but apparent enough to actually have an impact. These small cheats recognise the player's intention, rather than their exact commands.

Terminal velocity is an important tweak that stops the player from infinitely gaining vertical momentum, making it much easier to control, particularly from high platforms. This value must be high enough that it does not seem like the player "rubber bands", or instantly slows down, but low enough that it actually makes an impact, so the player does not gain too much vertical velocity. This is particularly useful in my game, since a lower-than-average terminal velocity will allow the player to stay in the air for longer, meaning it is easier to control the player and make their next move.

Rounded corners (or capsule collisions) make it so that it is impossible to miss a jump by a few pixels. For example, say the player is below a block by two pixels – this is indistinguishable to the naked eye. If the player jumps, without a rounded collision shape, the player will hit the block and lose their speed. However, with rounded corners, the player is subtly moved in line with the edge of the block, meaning that small and insignificant errors are disregarded entirely.

When making jumps in a platformer, it is very easy to miss pressing the jump button by only a few milliseconds. Coyote time and a jump buffer can remedy this. Coyote time allows the player to jump a few frames after they have fallen off the edge of a block. This is subtle enough that the player does not think the game didn't register their input unfairly, but it does not allow the player to always jump mid-air. A jump buffer is less essential, but still helps to make sure that the game always registers the player's intended input. When falling towards the ground, if the player presses the jump button a few frames early, usually, the game will not register the jump. However, a jump buffer keeps the jump input in mind, and makes the player jump once they reach the ground, even if the jump button is no longer being pressed. These two small details can make a platformer feel much fairer, which is a primary focus, as mentioned in the interviews.

Animations

Not only is making the actual mechanics an important part, but a good practice of game design is making something, that the player will repeatedly do, feel good. This can be done by adding various animations, particle effects, and distortions to the player, which make the game feel much more polished and complete. Subtly adding particles for when the player is running, jumping, and landing can go a long way in giving the player an appropriate amount of feedback whilst platforming. Moreover, shaking the screen when creating a new rope will add useful feedback to the player, rather than just a small visual change.

Sounds

Similar to these animations, small sounds for running, jumping, landing etc. can go a long way to adding much more depth to the character's move set.

Swinging

Of course, the basic platformer mechanics, like described above, is important for a platformer game, but they are not essential for this one; the rope swing mechanic will be front and centre, meaning levels must be designed around the uniqueness of the mechanic.

A swinging mechanic is not as well established as other platformer mechanics, such as running and jumping, however I plan on making rope-swinging fit in with said mechanics. As mentioned previously, to provide precision for the player, the rope's "target" will be controlled by the mouse position on screen. This will allow for extremely precise movement of the mouse, as well as making it easy to quickly and consistently perform sequential rope swings.

The player will be able to control the rope after it has successfully "grappled" on to an appropriate tile. To provide challenge, later levels may include tiles that cannot be grappled on to, as otherwise it would make most obstacles trivial. To control the rope, the player will use the right and left keys, which will increase and decrease the angle respectively. Crucially, to add a suitable degree of realism, whilst also providing a better feeling mechanic, I plan on implementing a "pendulum" system.

The pendulum system will essentially allow the rope to swing from side to side, or settle in the middle, depending on which inevitably feels better and easier to control. This means that, for example, if a player lets go of the right arrow key and the rope was facing towards the right, the rope will be affected by "gravity" and fall back down to a neutral, down-facing position. This is important as it will make the rope feel like the player expects, rather than just a free-floating hovering system. Without a pendulum system, the rope effectively is not affected by gravity, making the game considerably more confusing for new players.

Level Design

Of course, there is no point to good movement physics without good level design, which is a key component of the design portion of this programming project.

Level design doesn't just come down to where to place a platform, or put a collectible, but it also includes integrating game mechanics, such as tutorials, seamlessly, as to add to the player's overall experience.

Level Components

Firstly, it is important to establish the various mechanics I plan on implementing into the level, disregarding the player's own mechanics.

Mechanic	Explanation
Spikes	In order to provide a suitable challenge, there needs to be surfaces that the player cannot land on; this could be via anything from inescapable pits, to pools of a dangerous liquid, to tiles that simply will kill the player. To provide an easy-to-understand mechanic, I plan on using spikes, similar to those seen in games like Celeste, or many Super Mario games. This is because they are instantly recognisable as a hazard, unlike something like a tile, which may not demonstrate danger without an obvious design, such as a skull and crossbones, limiting the art for the game. If the player touches the spikes, they will be killed and reset to a previous point in the level, that being the checkpoint they last touched. This will be one of the many ways the player can die, meaning it also ties into the scoring of the game.
Checkpoints	As I would like longer levels that are not split up into "screens" (unlike many levels seen in Celeste, or older, 1980s platformer games, such as Donkey Kong) a checkpoint system will be a suitable feature. This is because it will allow me to place checkpoints in key parts of the level, allowing the player to not have to complete the level from the start every time they die, which would become frustrating in particularly long levels.
Coins	To also tie in with the scoring system, I plan on implementing coins, since it will provide another layer of scoring that is not essential. This means that those, who find trying to collect as many as possible enjoyable, are free to do so, whereas players who prefer trying to complete the level as fast as possible, or with as few deaths as possible can instead focus on that.

Basic Tropes

There are a few rules that I will follow for all my levels. Following these should make the game feel difficult, but not unfairly frustrating, which is key in level design.

Trope	Explanation
Consistency	When designing levels, it is important to always make sure each mechanic acts consistently and predictably. For example, jumping into water shouldn't let the player swim in one instance, and kill them in another. This is because it allows the player to plan out their actions ahead of time, and it does not unfairly put them at a disadvantage for experimenting. When a new mechanic is introduced, the player can quickly figure out whether it will benefit them, or be a detriment to them, and it will stay that way throughout the entirety of the game. This is also why randomised physics should be avoided, as it can make the game's controls feel inconsistent, as if the player is constantly fighting against them. Crucially, this does not mean that there cannot be any random functions in the game – in fact most games rely upon a small degree of randomisation – but there is a difference between a small amount of randomisation, and so much randomisation that it becomes frustratingly inconsistent.
Screen Space	It is important that the player can plan out their moves, meaning they must be able to see a reasonable number of obstacles ahead of them. This means that, there must never be, for example, a moving platform that comes on and off of the screen, as the player would not know what move to make *until* the platform comes back. Once again, this can feel frustrating as it artificially adds additional waiting time whilst playing the game, which is supposed to be fairly fast paced. However, in order to expand the options in level design, small camera controls can be added, such as following the mouse around, in the case of this game.

Clear Directions	One of the most common, frustrating parts of bad game design is levels which don't provide direction. Even in non-linear games and levels, it's imperative to provide the player with a few (even subtle) hints on where to go. This can be through the use of clues, or more obvious things, such as arrows. I plan on making the levels linear and mostly follow a Mario-esque "just go to the right" technique to provide direction for the player. When designing levels, it is important to ensure that the level-design makes sense, and is easy-to-understand. This can be accomplished through the use of player testing during development.

Difficulty Progression

As mentioned in the interview, an essential part of the level design is the difficulty progression. Many platformer games seem to have a sudden spike in difficulty, that ends up negatively affecting the overall experience, even to the point where some people will give up. It is for this reason why play-testing the levels with a variety of users is integral, as it provides a plethora of feedback to use for later levels, as well as to use if any current levels need redesigning.

The first few levels should be focused on providing the player with a tutorial, which will be expanded upon later. Afterwards, there should be a set of very simplistic levels that first introduce a mechanic, then expand upon it, and finally have a challenge based on the mechanic at the end. Repeat this until all the main mechanics, not introduced in the tutorial levels, have been introduced to the player.

The next set of levels should begin combining these mechanics in more interesting ways, but crucially, they should follow a similar pattern: introduction, expansion, challenge. The introduction portion will be slightly different as it needs to clearly demonstrate how the mechanics can tie into each other in effective and unique ways.

The final set of levels should feel like an ending – they should be much larger, more challenging, but still follow a similar pattern. This will be the last set of levels to make sure that the difficulty is well-balanced; it is more important to focus on the difficulty progression than the number of levels in the game, so I will keep it "short and sweet".

Tutorial Integration

As previously mentioned, another part of the level design is incorporating the tutorial into the first few levels. These levels should follow the same pattern as aforementioned, as it is an effective way of introducing mechanics to the player.

Crucially, the tutorials should take place in a safer environment, where the player is not penalised as much – if at all – for making a mistake. This encourages the player to get used to the controls and various key mechanics, before undertaking the more difficult and challenging levels.

Scoring

Scoring is a major part in many games: how can multiple players compare their progress with each other? This can be as simple as "how far have I got in the game" for an RPG, or "how many levels have I completed" in a platformer. However, a complex game will always have inherently more complex scoring, although it can be made easier by keeping track of some simple values.

It could be easily argued that play-time is a quantifiable way of scoring, as the longer a player has in a certain game, the more skilled they are likely to be. For my project, however, I am not going to use play-time as a scoring method, as a player's skill in platformers is quite difficult to quantify using time; many platformers share a similar skill-set, thus those who have 100 hours in Celeste, for example, and only one hour in my game are more likely to be skilled than someone with 5 hours with my game, but no prior experience.

Instead, I have decided to take a page out of Celeste's book, particularly due to the interviews, and use a death count as the main way of keeping track of score; a death count can be used in a plethora of ways:

- On its own, a death count can be used as a secondary way of seeing how much of the game the player has played: it's not as objective as time, but it still provides a good idea of the amount of time dedicated into the game.
- Alongside knowing how many levels the player has completed, a death count provides a look into the skill of a player; a player with a low death count and a high progression is more skilled than a player with a high death count and low progression.
- Death counts can be used in isolation for user-made challenges, such as "beat this level first try in only 5 deaths", or "complete this level in x amount of time without dying more than y times".

However, I will also keep track of an additional scoring mechanism: the number of coins collected. This provides a useful insight to a more cautious player, who prefers more exploratory, collectible-based platforming action. The two scores can be used in tandem to understand a player's enjoyment, skill, and overall engagement with the game.

Pausing

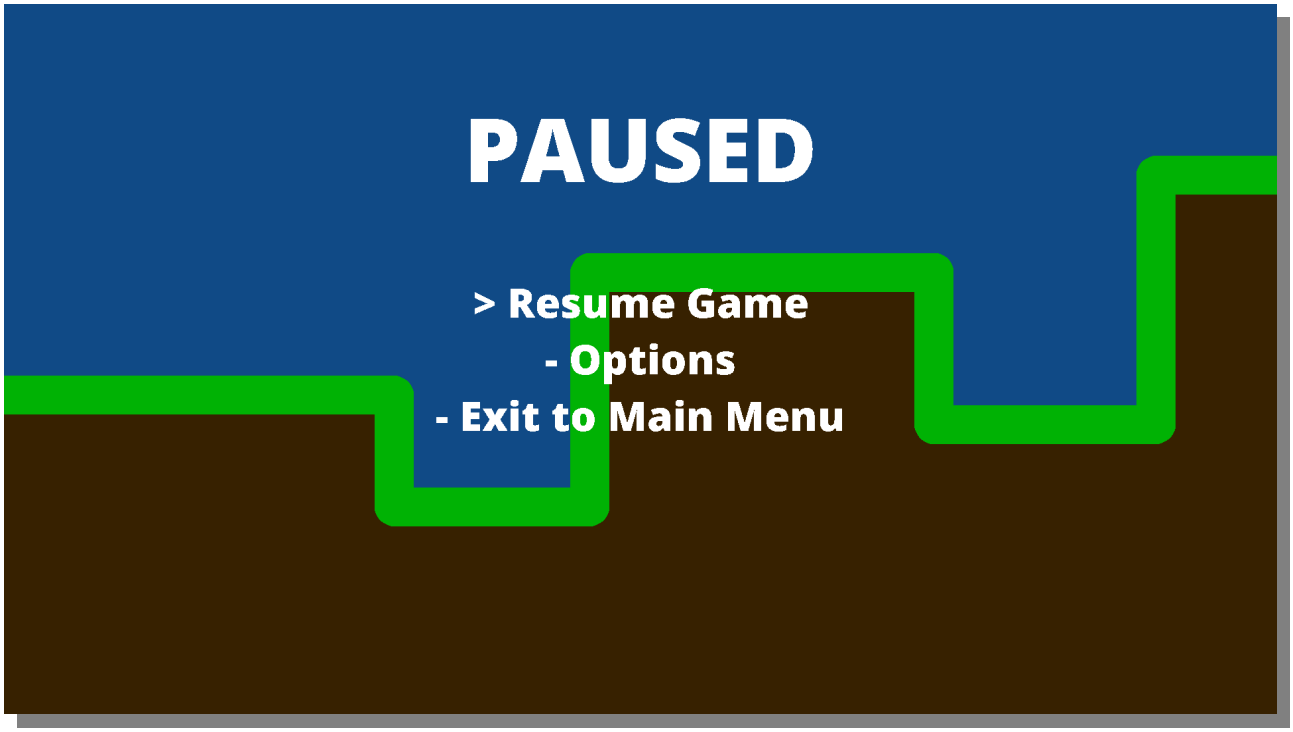
Pausing is necessary to allow the player to have a break, have a Kit-Kat, at any point in the game and continue it whenever. This is essential, particularly in a singleplayer game, for a good user experience.

Exploitation

Pausing the game may seem like a non-trivial task. However, it can lend itself to many exploits if not handled correctly. This is because, when paused – without additional care – timers, momentum, animations etc., can all build up indefinitely. This means that, when the game is resumed, all these increased values are applied, which can cause a multitude of problems. There are a few ways to rectify this. The more complicated way is to keep track of all incrementing variables, and make sure that they stay the same whilst the game is paused. Another way of tackling this is to only allow the player to pause if certain conditions are met: these usually being if the player is on the ground. This means that there will be no additional variables to do with jumping, or moving in the air, and only the horizontal momentum needs to be considered.

Menu

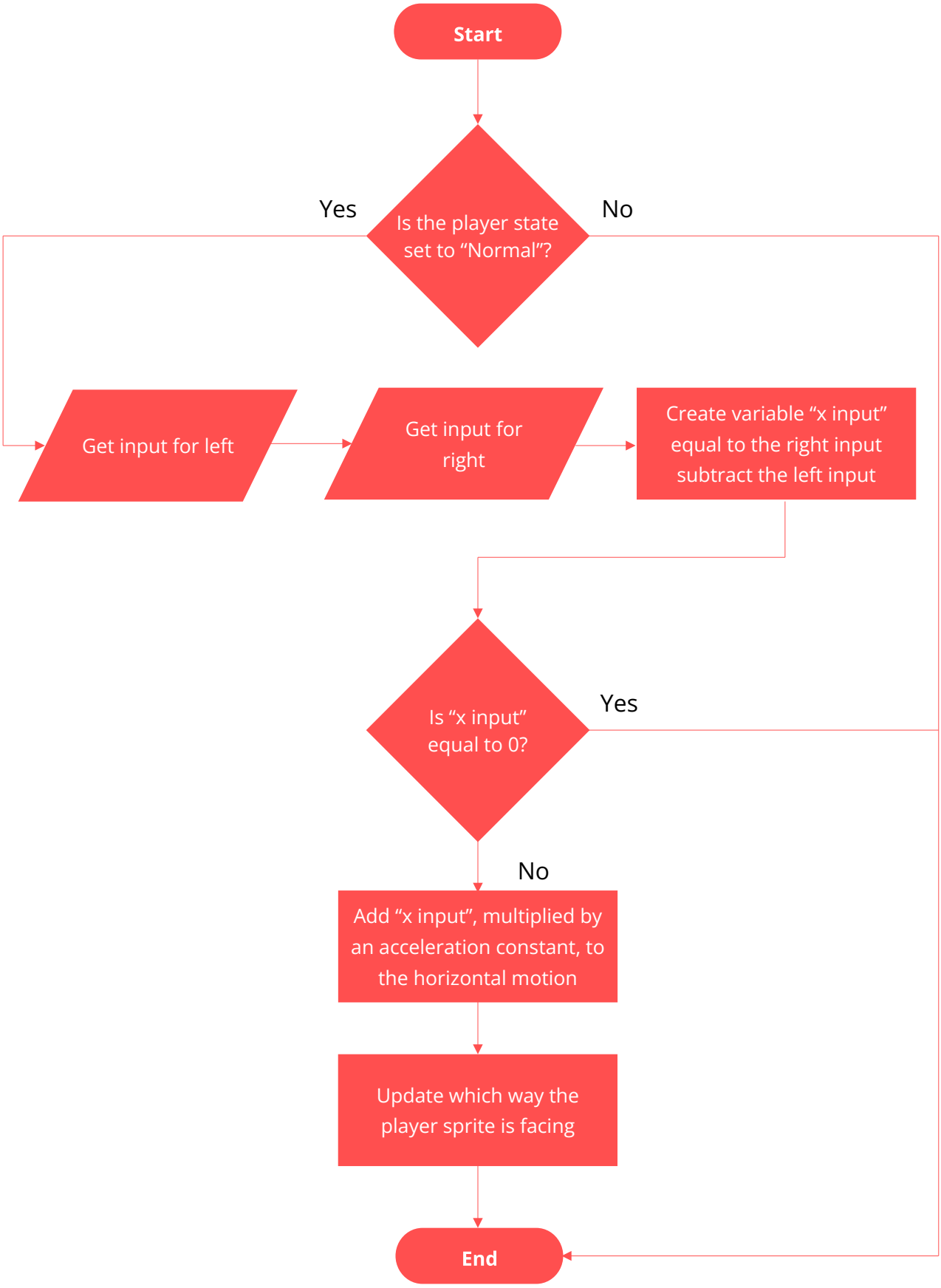
Another important part of pausing is the user interface design. Below is a mock-up for a pause menu.

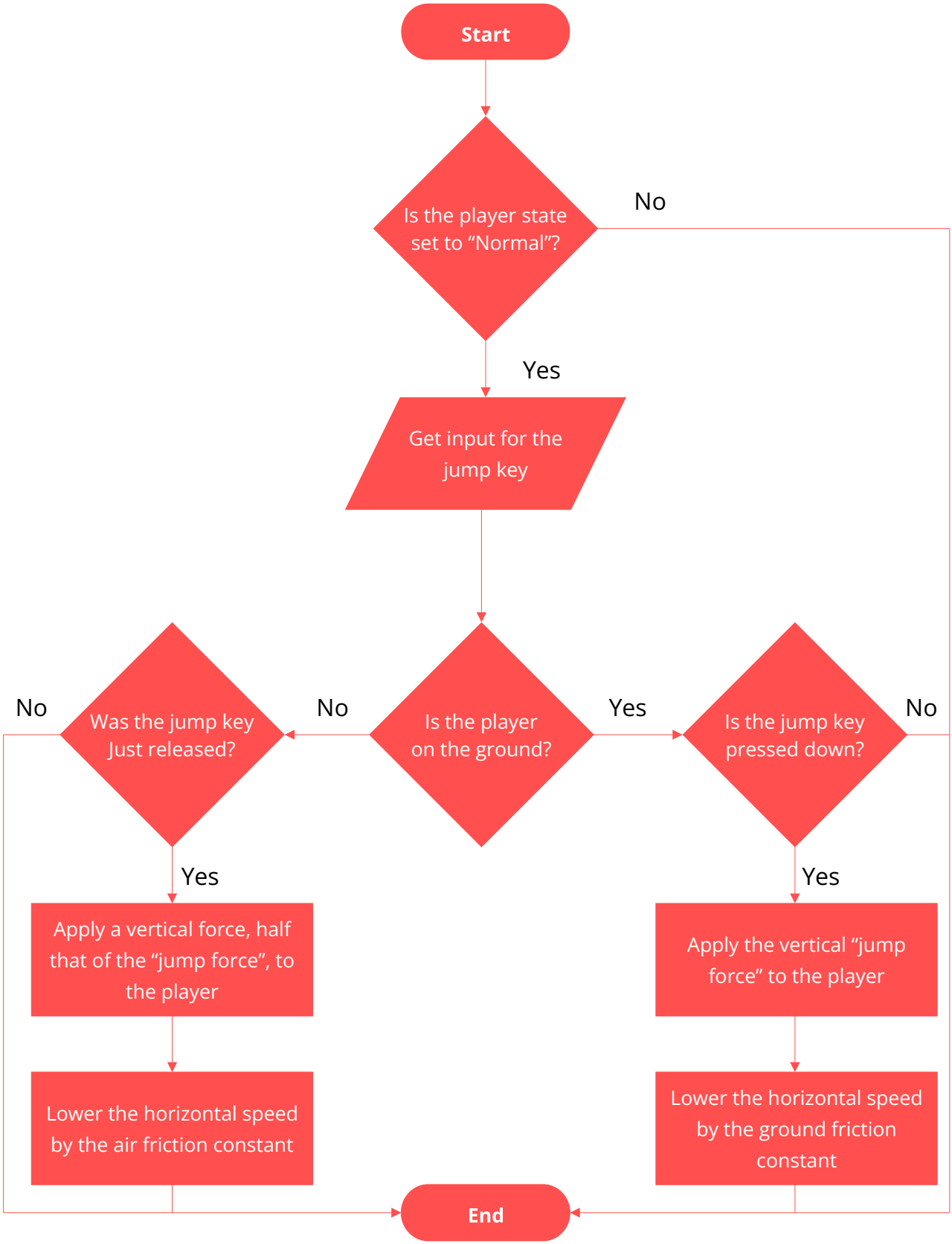


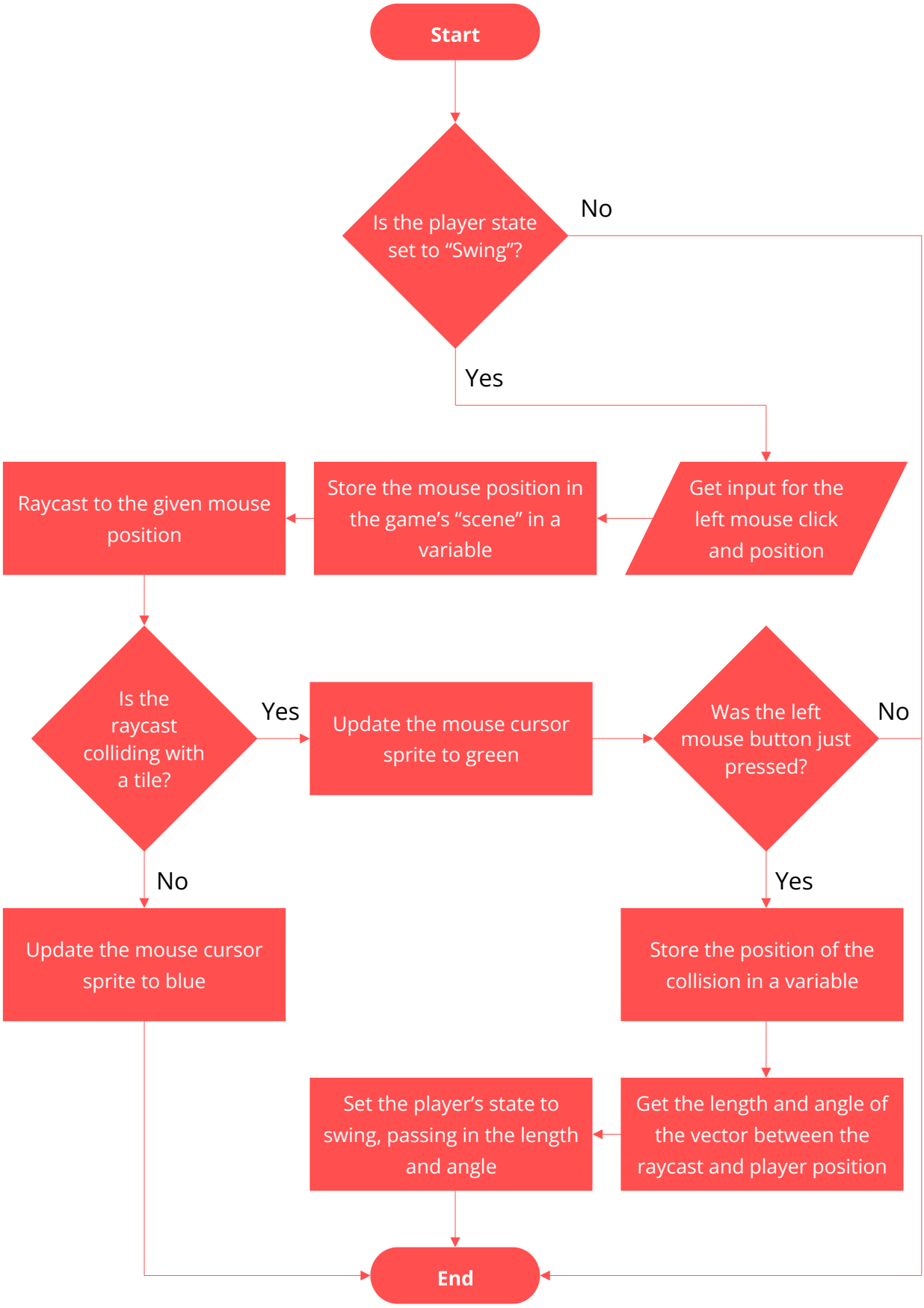
As with the previous mock-ups, this is a massive simplification of the actual UI design, but it does showcase the important elements: a clear title; a translucent, black background over the level currently being played; and three clear options: resume game, go to the options menu (which can then display instead of the pause menu), and exit back to the main menu.

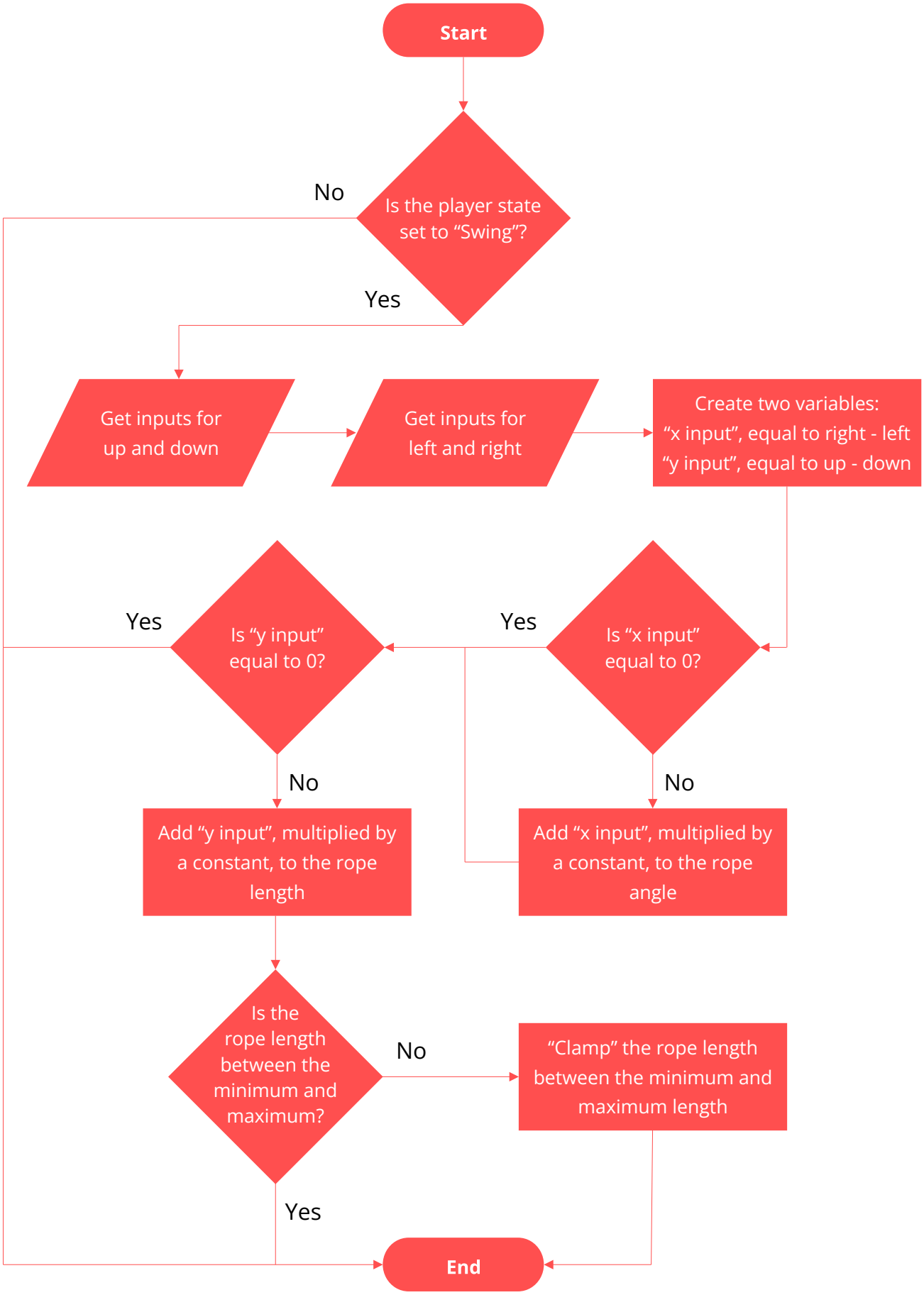
Algorithms

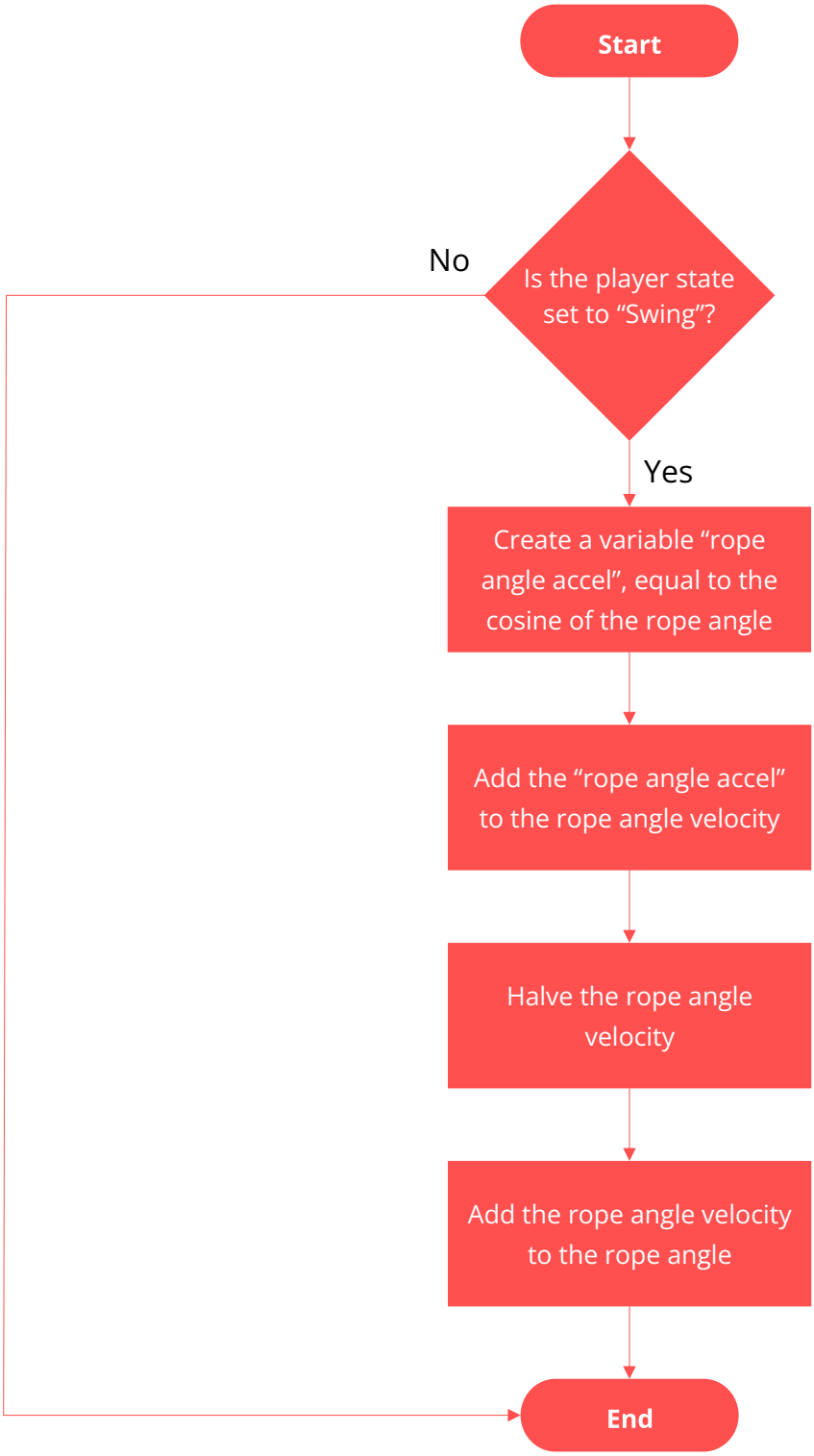
Now that the design for the game has been completed, I can easily begin planning out the various algorithms that will be required to accomplish this solution. The first step is to plan out which parts of the game will require an algorithm. Using the systems diagram as guidance, I can plan out the various algorithms I need in the game. During the game's development process, however, it may turn out that the algorithms need to be slightly modified – this can be done through testing and debugging the initially planned out algorithms. All algorithms below are to be run at the games framerate (i.e., 60 times a second) inside the main game loop, and each algorithm is to be run in a single frame (i.e., all code runs one after the other, then the game updates).


Start
Is the player state set to "Normal"?
Yes
No
Get input for left
Get input for right
Create variable "x input" equal to the right input subtract the left input
Is "x input" equal to 0?
Yes
No
Add "x input", multiplied by an acceleration constant, to the horizontal motion
Update which way the player sprite is facing
End


Start
Is the player state set to "Normal"?
No
Yes
Get input for the jump key
Is the player on the ground?
No
Was the jump key Just released?
No
Yes
Apply a vertical force, half that of the "jump force", to the player
Lower the horizontal speed by the air friction constant
Yes
Is the jump key pressed down?
Yes
No
Apply the vertical "jump force" to the player
Lower the horizontal speed by the ground friction constant
End

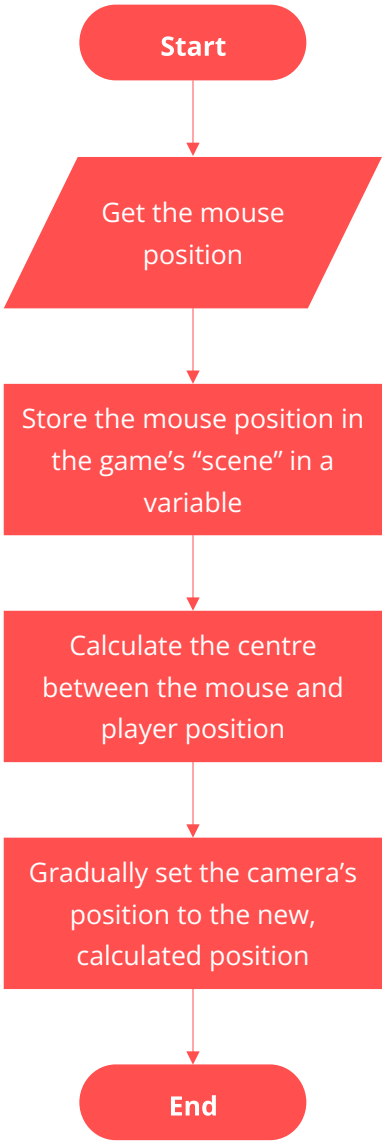

Start
Is the player state set to "Swing"?
No
Yes
Get input for the left mouse click and position
Store the mouse position in the game's "scene" in a variable
Raycast to the given mouse position
Is the raycast colliding with a tile?
Yes
Update the mouse cursor sprite to green
Was the left mouse button just pressed?
No
Yes
No
Update the mouse cursor sprite to blue
Store the position of the collision in a variable
Get the length and angle of the vector between the raycast and player position
Set the player's state to swing, passing in the length and angle
End


Start
Is the player state set to "Swing"?
No
Yes
Get inputs for up and down
Get inputs for left and right
Create two variables:
"x input", equal to right - left
"y input", equal to up - down
Is "y input" equal to 0?
Yes
Is "x input" equal to 0?
Yes
No
No
Add "y input", multiplied by a constant, to the rope length
Add "x input", multiplied by a constant, to the rope angle
Is the rope length between the minimum and maximum?
No
"Clamp" the rope length between the minimum and maximum length
Yes
End

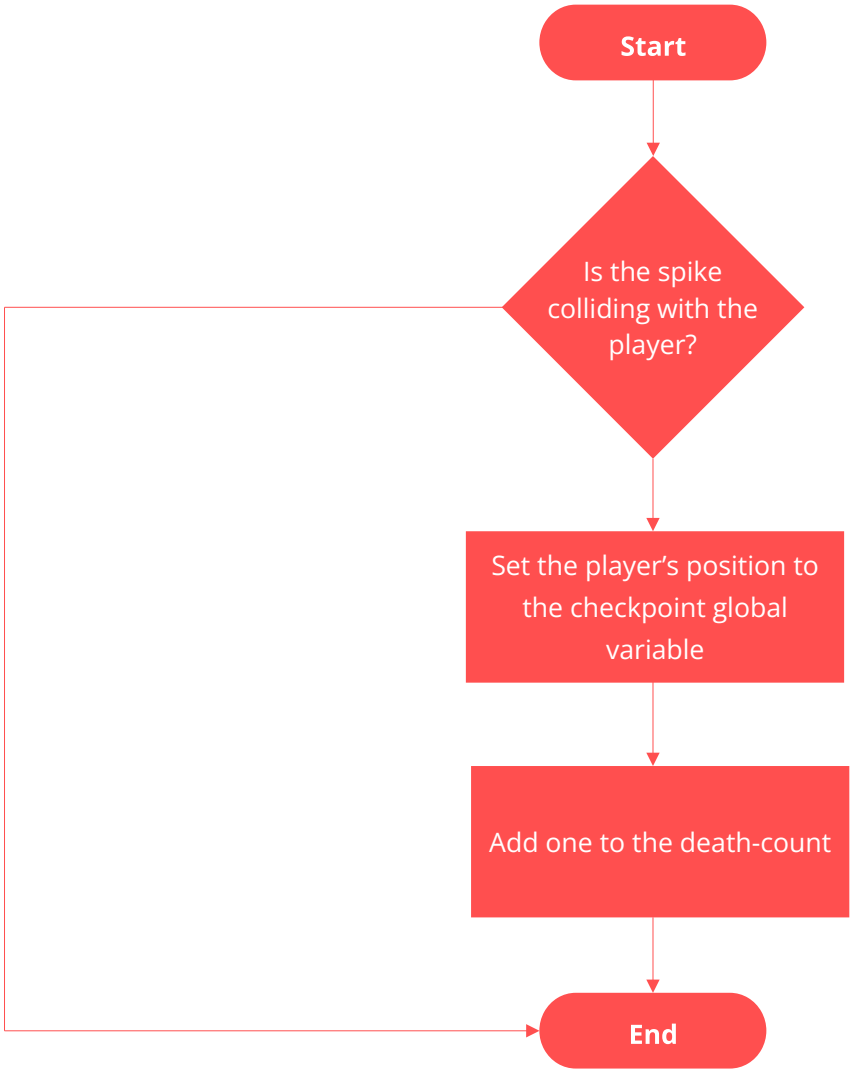

Start
Is the player state set to “Swing”?
No
Yes
Create a variable “rope angle accel”, equal to the cosine of the rope angle
Add the “rope angle accel” to the rope angle velocity
Halve the rope angle velocity
Add the rope angle velocity to the rope angle
End

Rope Line

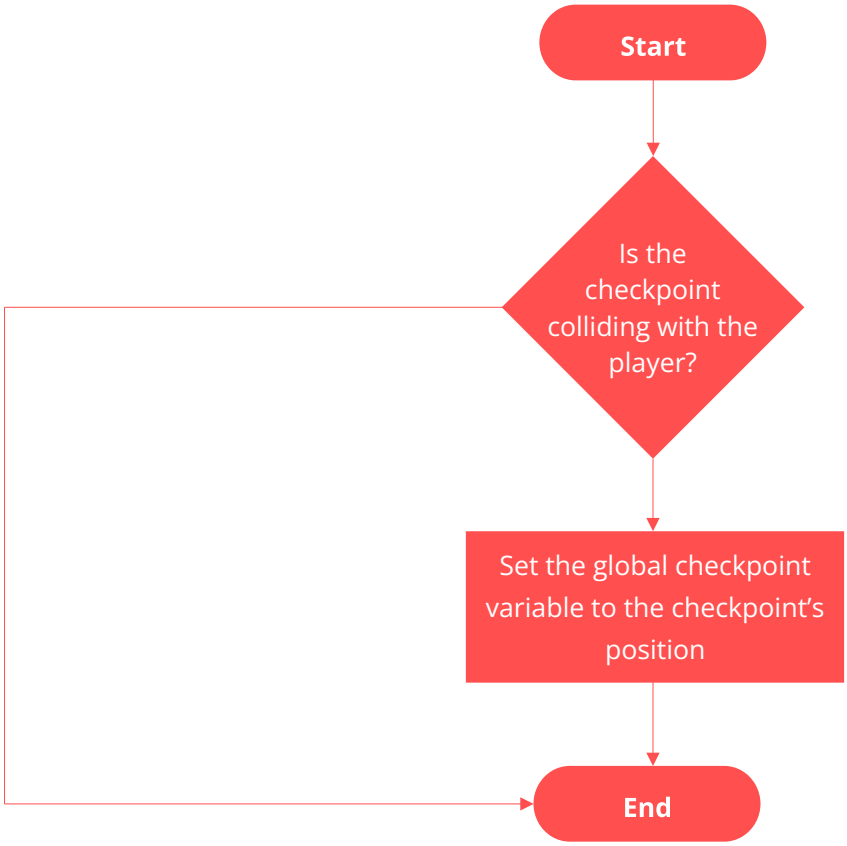

Start
Is the player state set to “Swing”?
No
Yes
Clear the rope line’s points
Add a point for the player’s position
Add a point for the rope position
End


Start
Get the mouse position
Store the mouse position in the game's "scene" in a variable
Calculate the centre between the mouse and player position
Gradually set the camera's position to the new, calculated position
End

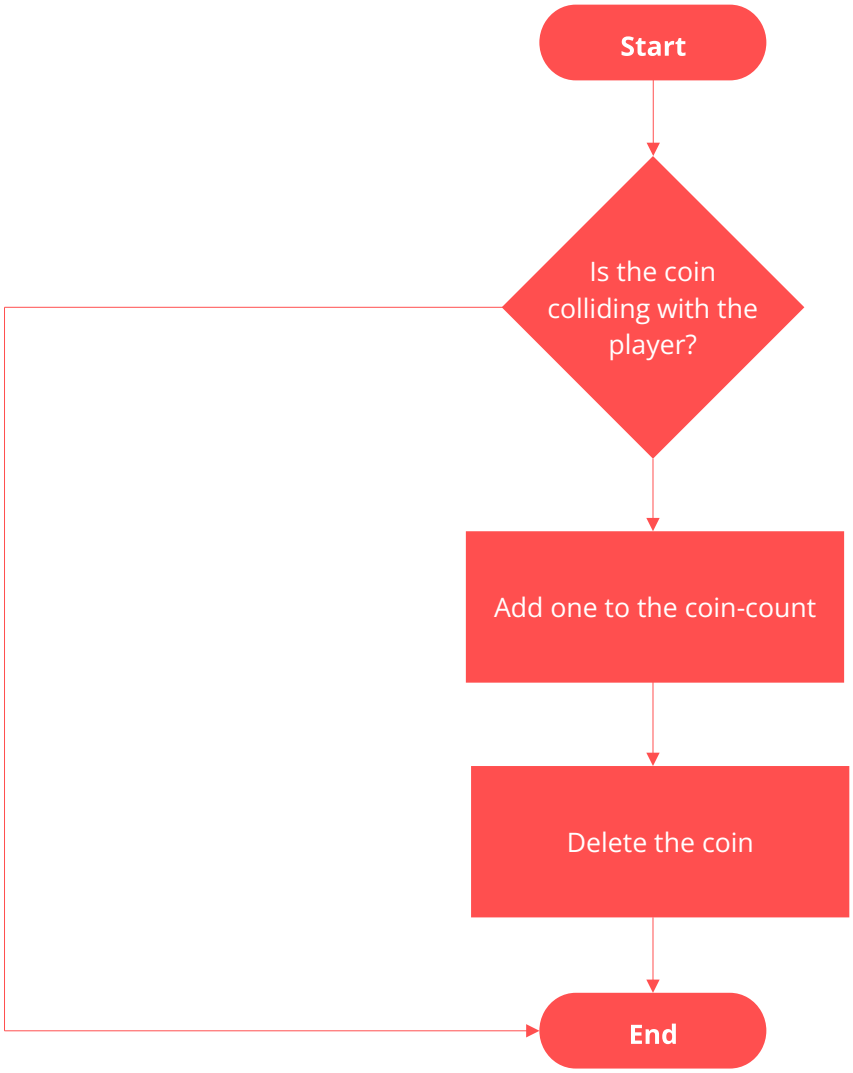
Spikes


Start
Is the spike colliding with the player?
Set the player's position to the checkpoint global variable
Add one to the death-count
End

Checkpoints



Coins


Start
Is the coin colliding with the player?
Add one to the coin-count
Delete the coin
End

Key Functions, Variables, and Data Structures

All Nodes (Objects)

Method	Name	Type	Explanation	Justification
Function	_process()	N/A	In each node (object), this function runs every frame. It will contain majority of the code for dynamic objects, such as the player.	The game needs to run majority of the code every frame to check for user input, and update the game accordingly.
Function	_ready()	N/A	In each node (object), this function runs when loaded. It will contain code that only needs to be run once, majority of which sets-up the node to be used later.	The game needs some code to only be run once, and then never again, unless reloaded. This allows for nodes to be set-up correctly.
Variable	position	Vector2D	Stores the current position of any node that acts in 2D, relative to its parent.	The game needs to keep track of the position of any node that exists in 2D space. Being relative to the parent allows children nodes to follow the parents' position, but also be offset.

Variable	global_position	Vector2D	Stores the current position of any node that acts in 2D, relative to the active scene.	The game needs to keep track of the position of any node that exists in 2D space. Being relative to the scene allows the node to have its position detached from the parent.

Player

Method	Name	Type	Explanation	Justification
Constant	accel	Integer	Stores the rate at which the player's speed accelerates horizontally.	The player does not instantly reach top speed, thus requires a value of acceleration.
Constant	max_air_spd	Integer	Stores the maximum horizontal speed for the player whilst in the air.	The player's speed needs to have a maximum, as to not accelerate ad infinitum. This is different than the ground speed to allow the player to be faster whilst in the air.

Constant	max_ground_spd	Integer	Stores the maximum horizontal speed for the player whilst on the ground.	The player's speed needs to have a maximum, as to not accelerate ad infinitum. This is different than the air speed to allow the player to be easier to control whilst on the ground.
Constant	term_vel	Integer	Stores the maximum vertical speed for the player (whilst in the air).	The player's speed needs to have a maximum, as to not accelerate ad infinitum.
Constant	grav	Integer	Stores the rate at which the player accelerates vertically.	The player does not instantly reach top speed, thus requires a value of acceleration.
Constant	ground_frict	Real	Stores the rate at which the player's horizontal speed decelerates whilst on the ground.	The player needs to slow down if a movement key is not pressed. This is different than the air friction as to allow the player to be easier to control whilst on the ground.

Constant	air_frict	Real	Stores the rate at which the player's horizontal speed decelerates whilst in the air.	The player needs to slow down if a movement key is not pressed. This is different than the ground friction as to allow the player to travel further in the air, useful for when swinging on the rope.
Constant	jump_force	Integer	Stores the initial force given to the player when jumping.	When jumping, a force is applied to the player upwards. This allows the player to jump.
Constant	max_rope_len	Integer	Stores the maximum length the rope can possibly go.	The rope should not be able to infinitely extend as it would become too easy to exploit and too hard to control.
Constant	min_rope_len	Integer	Stores the minimum length the rope can possibly go.	The rope should not be able to infinitely contract as it would eventually go into negative values, which is logically unsound (to say the least).

Constant	max_rope_spd	Integer	Stores the maximum speed the rope can swing from side to side.	The rope should not be able to swing at an infinite speed as this would become too easy to exploit and too hard to control.
Enum	state	Enum	Stores any possible player states.	Instead of having the player state just be a magic number, an enumerator allows it to be easily read in-code.
Variable	player_state	Integer	Stores the current player state.	The game needs to keep track of whether the player is in the rope state, or the normal platforming state, and run appropriate code for each one. This separates them into distinct and clear algorithms.
Variable	rope_len	Integer	Stores the current rope length.	The game needs to keep track of how long the rope is in order to be able to calculate where the player should be while swinging, as well as how fast the rope should move.

Variable	rope_angle	Real	Stores the current rope angle.	The game needs to keep track of the angle at which the rope is with respect to the horizontal axis in order to allow the player to change this angle and, in turn, swing.
Variable	rope_pos	Vector2D	Stores the current position the rope is “grappled” on to.	The game needs to know where the origin of the rope is so it can calculate where the player should end up (in a circular motion). This is essentially the centre of the circle of movement.
Variable	motion	Vector2D	Stores the current player motion.	The game needs to keep track of how fast the player is moving in the x and y direction, in order to calculate collisions and the like.
Variable	colliding	Boolean	Stores whether the player is colliding with a tile or not.	The game needs to know if the player is colliding with a tile to alter the rope movement (e.g., “bounce” off a tile if swung into it).

Variable	coyote_on_floor	Boolean	Stores whether the player is on the floor, or if the coyote-time timer is still active.	The game needs to allow the player to jump even if they are not technically on the floor – this is known as coyote-time (see previous sections for explanation).
Function	horizontal_movement()	N/A	Contains the code for the player's horizontal movement.	The player needs to be able to move horizontally based on what keys the player is pressing.
Function	gravity()	N/A	Contains the code for the player's vertical movement in the platformer state.	The player needs to act according to gravity; whilst not on the rope, the player must be able to fall.
Function	ground_spd_modifier()	N/A	Contains the code that modifies the player's horizontal movement, i.e., limiting it to the maximum speed, and adding ground friction.	The game needs to be able to modify the horizontal movement to keep the player from infinitely accelerating, and allowing the player to stop.

Function	air_spd_modifier()	N/A	Contains the code that modifies the player's vertical movement, i.e., limiting it to the maximum speed, and adding air friction.	The game needs to be able to modify the vertical movement to keep the player from infinitely accelerating, and allowing the player to stop.
Function	rope_angle_changes()	N/A	Contains the code that allows the player to control the rope's angle.	This allows the player to swing on the rope, by changing the rate at which the rope moves from side-to-side.
Function	reset_rope()	N/A	Contains the code that resets the player back to the platformer state.	The player needs to be exit the rope-swing state back to the normal platforming state. This allows them to "fling" themselves across the level.
Function	initialise_rope()	N/A	Contains the code that sets-up the rope initially.	The rope needs to be set-up each time the player begins a swing. This includes finding out where the mouse cursor is.

Function	rope_animation()	N/A	Contains the code that modifies where the rope-line is drawn.	The rope-line needs to be constantly updated according to its angle, and the player's position, as well as its origin point.
Function	die()	N/A	Contains the code that resets the player back to the active checkpoint.	The player needs to be able to re-attempt a portion, as well as face a challenge (e.g., colliding with spikes).

Global Variables

Method	Name	Type	Explanation	Justification
Variable	checkpoint_pos	Vector2D	Stores the position of the active checkpoint	This allows the game to "respawn" the player at the correct checkpoint. It needs to be global so it can be updated when a new checkpoint is activated
Variable	coin_count	Integer	Stores the total number of coins collected.	This allows for scoring of the total amount of coins in the game. It needs to be global so it can be updated each time a coin is collected.

Variable	death_count	Integer	Stores the total number of deaths.	This allows for scoring by the number of deaths a player has. It needs to be global so it can be updated each time the player collides with a spike.
Variable	level_to	String	Stores the path of the level resource to go to if “Continue Game” is pressed.	This allows the player to continue at the level they previously left off at. This needs to be global so it can be modified when the player goes to the next level, as well as reset when “New Game” is pressed.
Variable	window_fullscreen	Boolean	Stores whether the game is fullscreen or not.	This allows the screen to be toggled into fullscreen or windowed mode. This needs to be global so it can be modified by the options menu or if the user presses F11.

Variable	master_vol	Real	Stores the master volume.	This allows all sounds to be made louder or quieter. This needs to be global so it can be accessed by any nodes playing audio, as well as modified by the options menu.
Variable	music_vol	Real	Stores the music volume.	This allows all music to be made louder or quieter. This needs to be global so it can be accessed by any nodes playing music, as well as modified by the options menu.
Variable	sfx_vol	Real	Stores the sound-effect volume.	This allows all sound effects to be made louder or quieter. This needs to be global so it can be accessed by any nodes playing sound effects, as well as modified by the options menu.

Variable	button_up	Integer	Stores the scan-code of the button bound to “move up”.	This allows the key-bind for moving up to be changed. This needs to be global so it can be accessed anywhere in code, to detect key presses.
Variable	button_down	Integer	Stores the scan-code of the button bound to “move down”.	This allows the key-bind for moving down to be changed. This needs to be global so it can be accessed anywhere in code, to detect key presses.
Variable	button_left	Integer	Stores the scan-code of the button bound to “move left”.	This allows the key-bind for moving left to be changed. This needs to be global so it can be accessed anywhere in code, to detect key presses.
Variable	button_right	Integer	Stores the scan-code of the button bound to “move right”.	This allows the key-bind for moving right to be changed. This needs to be global so it can be accessed anywhere in code, to detect key presses.

Variable	button_jump	Integer	Stores the scan-code of the button bound to “jump”.	This allows the key-bind for jumping to be changed. This needs to be global so it can be accessed anywhere in code, to detect key presses.
Variable	data	Dictionary	Stores the data to be saved into a JSON file.	This allows the global variables to be saved and loaded when the game is closed or opened, respectively.

Camera

Method	Name	Type	Explanation	Justification
Variable	mouse_pos	Vector2D	Stores the position of the mouse relative to the active scene.	This allows the camera to follow the mouse, which is important as it allows the player to look ahead before swinging.
Variable	game_size	Vector2D	Stores the size of the game window.	This allows the camera to scale down the mouse position if the window is scaled up.

Variable	interpolate_val	Real	Stores the weight of linear interpolation.	This allows the camera to move quicker depending on certain circumstances, such as the player's speed
Constant	min_interpolate_val	Real	Stores the minimum weight of linear interpolation.	This ensures that, even if the player is not moving, the camera can still move with the mouse.

Rope Raycast

Method	Name	Type	Explanation	Justification
Variable	mouse_pos	Vector2D	Stores the position of the mouse relative to the active scene.	This allows the raycast to cast to the mouse position, initialising the player swing

Sprites

Method	Name	Type	Explanation	Justification
Constant	player_atlas	PNG	Contains all sprites for the player in an “atlas”.	Allows for the player to have an image associated with it, and be animated.
Constant	checkpoint_atlas	PNG	Contains all sprites for the checkpoints in an “atlas”.	Allows for the checkpoints to have an image associated with them, and be animated.
Constant	spike_atlas	PNG	Contains all sprites for the spikes in an “atlas”.	Allows for the spikes to have an image associated with them, and be animated.
Constant	coin_sprite	PNG	Contains the sprite for the coins.	Allows for the coins to have an image associated with them.
Constant	rope_cursor_sprite	PNG	Contains the sprite for the mouse cursor if it can grapple onto a tile.	Allows for the mouse cursor to dynamically change appearance if it can grapple onto a tile.
Constant	no_rope_cursor_sprite	PNG	Contains the sprite for the mouse cursor if it can’t grapple onto a tile.	Allows for the mouse cursor to dynamically change appearance if it can’t grapple onto a tile.

Constant	normal_cursor_sprite	PNG	Contains the sprite for the mouse cursor if it is not above a tile.	Allows for the mouse cursor to dynamically change appearance if it is not above a tile.
Constant	tileset_atlas	PNG	Contains all sprites for the tiles in an “atlas”.	Allows for the levels to be built out of individual “tiles”.
Constant	background_sprite	PNG	Contains the sprite for the background.	Allows for the levels to have a static background.
Constant	foreground_sprite	PNG	Contains the sprite for the foreground of the background.	Allows the background to have multiple layers, with parallax scrolling, to provide depth.
Constant	icon_32x	PNG	Contains a 32 × 32 image for the games icon.	Allows for the game to have multiple icon sizes, so the icon is correctly scaled in Windows, MacOS, and Linux. A Windows “.ico” file contains sizes 32 – 256 pixels, whereas a MasOS “.icns” file contains sizes 32 – 512 pixels. Linux uses “.png” files for its icons.
Constant	icon_64x	PNG	Contains a 64 × 64 image for the games icon.	
Constant	icon_128x	PNG	Contains a 128 × 128 image for the games icon.	
Constant	icon_256x	PNG	Contains a 256 × 256 image for the games icon.	
Constant	icon_512x	PNG	Contains a 512 × 512 image for the games icon.	

Constant	logo_no_outline_pixel	PNG	Contains a pixel-perfect sprite for the logo.	Allows for the game to have its own logo in-game
Constant	logo_outline_pixel	PNG	Contains a pixel-perfect sprite for the logo with a white outline.	Allows for the in-game logo to be seen on a dark background.
Constant	logo_no_outline_large	PNG	Contains a 20× scaled sprite for the logo.	Allows for the logo to be displayed outside the game.
Constant	logo_outline_large	PNG	Contains a 20× scaled sprite for the logo with a white outline.	Allows for the displayed-outside logo to be seen on a dark background.

Additional Files

Method	Name	Type	Explanation	Justification
Variable	save_file	JSON	Contains the save data for the game.	This allows the global variables to be saved and loaded when the game is closed or opened, respectively.

Acceptance Testing

No.	Requirement	Input	Expected Outcome
1	The highlighted menu option changes	Valid: up arrow key, down arrow key, “W” key, or “S” key Invalid: “X” key	Valid: the menu option is highlighted red Invalid: the highlighted menu option stays the same
2	The highlighted menu option changes to where the mouse cursor is	Valid: mouse movement Invalid: mouse scroll	Valid: the menu option under the mouse cursor is highlighted red Invalid: the highlighted menu option stays the same
3	If the highlighted menu option is “New Game”, and is selected, the game loads the first level	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the first level Invalid: the game stays on the menu screen
4	If the highlighted menu option is “Continue”, and is selected, the game loads the saved level	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the saved level Invalid: the game stays on the menu screen

5	If the highlighted menu option is “Options”, and is selected, the game loads the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the menu screen
6	If the highlighted menu option is “Exit”, and is selected, the game exits	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to black and exits Invalid: the game stays on the menu screen
7	If the highlighted options menu button is “Back”, and is selected, the game reverts back to the main menu / pause menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the main menu or pause menu Invalid: the game stays on the options menu
8	If the highlighted options menu button is “Fullscreen”, and is selected, the game toggles fullscreen	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game toggles whether it is fullscreen or not Invalid: the game remains in its fullscreen / windowed state

9	If the highlighted options menu button is “Controls”, and is selected, the game loads the controls menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the controls menu Invalid: the game stays on the options menu
10	If the master volume slider is highlighted, and the value has changed, change the master volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the master volume is changed Invalid: the master volume stays the same
11	If the music volume slider is highlighted, and the value has changed, change the music volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the music volume is changed Invalid: the music volume stays the same
12	If the sound effects volume slider is highlighted, and the value has changed, change the sound effects volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the sound effects volume is changed Invalid: the sound effects volume stays the same

13	If the highlighted controls menu button is “Back”, and is selected, the game reverts back to the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the controls menu
14	If the highlighted controls menu button is “Up”, and is selected, the next key pressed becomes the key-bind for moving up.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving up changes Invalid: the key-bind remains the same
15	If the highlighted controls menu button is “Down”, and is selected, the next key pressed becomes the key-bind for moving down.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving down changes Invalid: the key-bind remains the same
16	If the highlighted controls menu button is “Left”, and is selected, the next key pressed becomes the key-bind for moving left.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving left changes Invalid: the key-bind remains the same

17	If the highlighted controls menu button is “Right”, and is selected, the next key pressed becomes the key-bind for moving right.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving right changes Invalid: the key-bind remains the same
18	If the highlighted controls menu button is “Jump”, and is selected, the next key pressed becomes the key-bind for jumping.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for jumping changes Invalid: the key-bind remains the same
19	The player can pause and un-pause the game.	Valid: “Escape” key Invalid: “G” key	Valid: the game’s pause state is toggled Invalid: the game’s pause state remains the same
20	If the highlighted pause menu button is “Resume”, and is selected, the game un-pauses and resumes	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game resumes and the options menu is hidden Invalid: the game stays pauses
21	If the highlighted pause menu button is “Options”, and is selected, the game loads the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the pause menu

22	If the highlighted pause menu button is “Exit”, and is selected, the game saves and exits back to the main menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the main menu and saves the game Invalid: the game stays on the pause menu
23	If the mouse is moved, the in-game cursor also moves	Valid: mouse movement Invalid: mouse scroll	Valid: the in-game cursor sprite matches the mouse cursor’s position Invalid: the in-game cursor sprite does not move
24	If the rope can grapple on to a tile, the in-game cursor turns green	Valid: mouse movement (over a tile) Invalid: mouse scroll	Valid: the in-game cursor changes to green Invalid: the in-game cursor stays the same
25	If the rope will collide on a “non-ropeable” tile, the in-game cursor turns red	Valid: mouse movement (over a non-ropeable tile) Invalid: mouse scroll	Valid: the in-game cursor changes to red Invalid: the in-game cursor stays the same

26	If the rope will not collide with anything, the in-game cursor turns blue	Valid: mouse movement (over no collidable tile) Invalid: mouse scroll	Valid: the in-game cursor changes to blue Invalid: the in-game cursor stays the same
27	The rope grapples on to the tile underneath the mouse cursor	Valid: left mouse button Invalid: middle mouse button	Valid: the player enters the “swing” state and a rope appears Invalid: the player’s state remains the same
28	The player can exit the rope state back to the normal platformer state	Valid: spacebar (while in rope state) Invalid: “M” key	Valid: the player exits the rope state and enters the normal platforming state Invalid: the player remains in the rope state
29	While in the rope state, the player can move the rope clockwise	Valid: the left arrow key, or the “A” key Invalid: the “#” key	Valid: the rope turns clockwise Invalid: the rope stays where it is (or falls due to gravity)

30	While in the rope state, the player can move the rope anti-clockwise	Valid: the left arrow key, or the “D” key Invalid: the “#” key	Valid: the rope turns anti-clockwise Invalid: the rope stays where it is (or falls due to gravity)
31	While in the rope state, the player can extend the rope	Valid: the down arrow key, or the “S” key Invalid: the “#” key	Valid: the rope extends (until it reaches the maximum length) Invalid: the rope stays the same length
32	While in the rope state, the player can contract the rope	Valid: the up arrow key, or the “W” key Invalid: the “#” key	Valid: the rope contracts (until it reaches the minimum length) Invalid: the rope stays the same length
33	While in the platformer state, the player can move right	Valid: the right arrow key, or the “D” key Invalid: the “=” key	Valid: the player moves towards the right (unless colliding with a wall) Invalid: the player does not move

34	While in the platformer state, the player can move left	Valid: the left arrow key, or the “A” key Invalid: the “=” key	Valid: the player moves towards the left (unless colliding with a wall) Invalid: the player does not move
35	While in the platformer state, the player can jump	Valid: the spacebar, or “W” key, or up arrow key Invalid: the “=” key	Valid: the player jumps off the ground Invalid: the player stays on the ground
36	The player falls if not in the rope state	N/A	The player falls to the ground, and stops descending once hit the ground
37	The rope swings downwards and side-to-side (like a pendulum)	Valid: not the “A” key, “D” key, left arrow key, or right arrow key Invalid: the “A” key, “D” key, left arrow key, or right arrow key	Valid: the rope acts like a pendulum according to gravity, and swings towards the ground Invalid: the rope moves according to the previous requirements

38	The player stops moving if colliding with a tile	Valid: the player is colliding with a tile Invalid: the player is not colliding with a tile	Valid: the player stops moving, but can still move in the opposite direction (away from the tile) Invalid: the player can continue moving
39	The player dies if colliding with a spike	Valid: the player is colliding with a spike Invalid: the player collides with a coin	Valid: the player dies, the screen transitions to black, and 1 is added to the death count Invalid: the player collects the coin
40	The player respawns at the activated checkpoint	Valid: the player dies Invalid: the player collides with a coin	Valid: the player is teleported to the checkpoint while the screen is black, then the screen transitions back Invalid: the player collects the coin

41	The checkpoint activates if the player collides with it	Valid: the player collides with a deactivated checkpoint Invalid: the player collides with an activated checkpoint	Valid: the deactivated checkpoint activates, and its animation changes to a waving flag. All other checkpoints are deactivated Invalid: the checkpoint remains activated
42	The player collects a coin if it collides with one	Valid: the player collides with a coin Invalid: the player collides with a spike	Valid: the coin disappears and 1 is added to the coin count Invalid: the player dies
43	If a value displayed in the GUI is updated, it briefly shows said value	Valid: the player collides with a spike or with a coin Invalid: the player collides with a checkpoint	Valid: the GUI appears and displays the newly updated value Invalid: the GUI stays hidden

44	The player can display the GUI whenever wanted	Valid: left control key Invalid: the “Y” key	Valid: the GUI briefly shows all values Invalid: the GUI stays hidden
45	A sound effect if played whenever the player jumps	Valid: the spacebar, or “W” key, or up arrow key Invalid: the “=” key	Valid: the sound effect is played Invalid: the sound effect is not played
46	A sound effect is played whenever the player dies	Valid: the player collides with a spike Invalid: the player collides with the ground	Valid: the sound effect is played Invalid: the sound effect is not played
47	A sound effect is played whenever the player collects a coin	Valid: the player collides with a coin Invalid: the player collides with the ground	Valid: the sound effect is played Invalid: the sound effect is not played

48	A sound effect is played whenever the player activates a checkpoint	Valid: the player collides with a deactivated checkpoint Invalid: the player collides with an activated checkpoint	Valid: the sound effect is played Invalid: the sound effect is not played
49	A sound effect is played whenever the player is falling	Valid: the player is falling Invalid: the player is on the ground	Valid: the sound effect is played Invalid: the sound effect is not played
50	A sound effect is played whenever the player lands on the ground	Valid: the player collides in the ground (from previously being in the air) Invalid: the player is in the air	Valid: the sound effect is played Invalid: the sound effect is not played
51	Whenever a new level is started, unique music begins playing.	N/A	Each level plays its own unique music.
52	While in the main menu, unique music begins playing.	N/A	The main menu has its own unique music.

3.3 Developing the Solution

3.3.1 Iterative Development Process

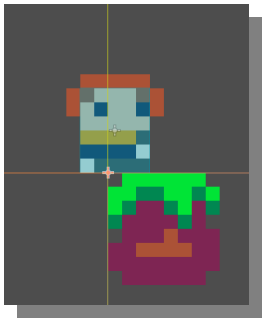
Setting Up the Player Node and Environment

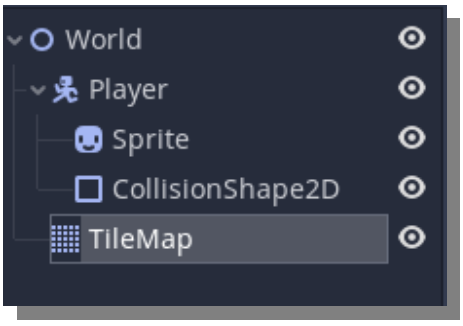
The first step for development was to setup the nodes in Godot; nodes are objects based on a variety of classes that end up creating an overall scene, or environment. Firstly, I needed to choose suitable nodes for my player, and the tiles in the world. I learned, by reading the documentation, that a "KinematicBody2D" would be best suited for my game, as it would allow me to move the player in the scene, based on a movement vector I give it.

The player, "KinematicBody2D", node also needs to be able to collide with any tiles around it – tiles which I will assign a collision to using Godot's built-in collision manager. To do this, I added a child node to the player called "CollisionShape2D". This will give my player node collision data. I initially set this to be a rectangle. The player also needed a child "Sprite" node. This will contain an image of the player, which I can then animate later. I made the collision fit around the sprite, so it would look like the sprite collides with the environment around it.

Finally, I added the aforementioned tileset using the "TileMap" node. Similar to the player, this will be a child class of the current scene – but not a child class of the player. For now, I simply assigned one tile to this, but later I will set up "autotiling" to allow tiles to seamlessly connect to each other.

After all the initial setup, the scene has the nodes as seen to the left, as well as the player and tiles.





Adding Movement Code to the Player

Now that the player node is set-up, I bundled it into a separate "scene". This will allow me to re-use the player in separate levels, without duplicating the node itself. The next thing I needed to do was to add the player's movement code. To do this, I assigned a script to the player node.

Because I knew that the player would need two separate states – one for normal platformer movement, and one for when the player is on a rope – I created an enumerator to store my states.

Next, I wanted to get keyboard inputs. To test this, I assigned the motion vector's x-value to my left and right value, multiplied by delta-time, to make sure movement stays consistent, regardless of the game's framerate, which I learned after researching the built in "_physics_process()" method of the "KinematicBody2D" class.

When I went to test this functionality, the player did not move. After re-examining the code, I realised that I forgot to set the player state, meaning no code was being executed. After correcting this, the behaviour worked as expected, and the player can now move left and right.

```
#Enumerator to initialise states
enum state {
    normal,
    swing
}

var player_state = state.normal

#A vector - magnitude and direction (essentially velocity)
var motion = Vector2.ZERO

#onready makes sure that the nodes have been initialised and loaded into the scene
onready var sprite = $Sprite
onready var animation = $AnimationPlayer

#Built in function from KinematicBody2D
func _physics_process(delta):
    match player_state:
        state.normal:
            var x_input = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
            var y_input = Input.get_action_strength("ui_up") - Input.get_action_strength("ui_down")

            motion.x += x_input * accel * delta     #Multiply by delta since occuring every frame
            motion.x = clamp(motion.x, -max_spd, max_spd)

        state.swing:
            pass


    motion = move_and_slide(motion, Vector2.UP) #Moves the player node by the vector + automatically collides
                                                #Also returns left over motion, meaning if collided
                                                #It will return no movement, and stop for the next frame

```

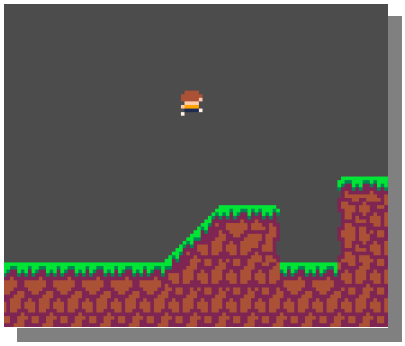
Adding Jumping and Gravity

Now that my inputs and player states were working, I needed to fully flesh out the platformer mechanics. To do this, I first went about adding gravity to the game, by applying a downwards vertical force to the y-value of the motion vector. Now the player does not "fly" away, depending on whatever direction it moves.

To allow the player to jump, I needed to add an upwards vertical force to the y-value, except only if the jump button is pressed:

```
38              motion.y += grav * delta    #Multiply by delta since occuring every frame
39
40              if is_on_floor():
41                  if Input.is_action_just_pressed("ui_up"):
42                      motion.y = -jump_force
```

In order to test this, I used Godot's auto-tiling feature to create a tileset that will collide with the player's "CollisionShape2D" node and created a basic test scene to allow me to test gravity and jumping.



The gravity and jumping worked very well, but there was a new problem with the code: the player would not stop moving horizontally. To fix this, I put the horizontal movement code in an if statement, so it would only accelerate the player if the key was held down, and an additional piece of code to linearly interpolate the motion towards 0 if the player was on the ground. If the player was in the air, a smaller force would be applied.

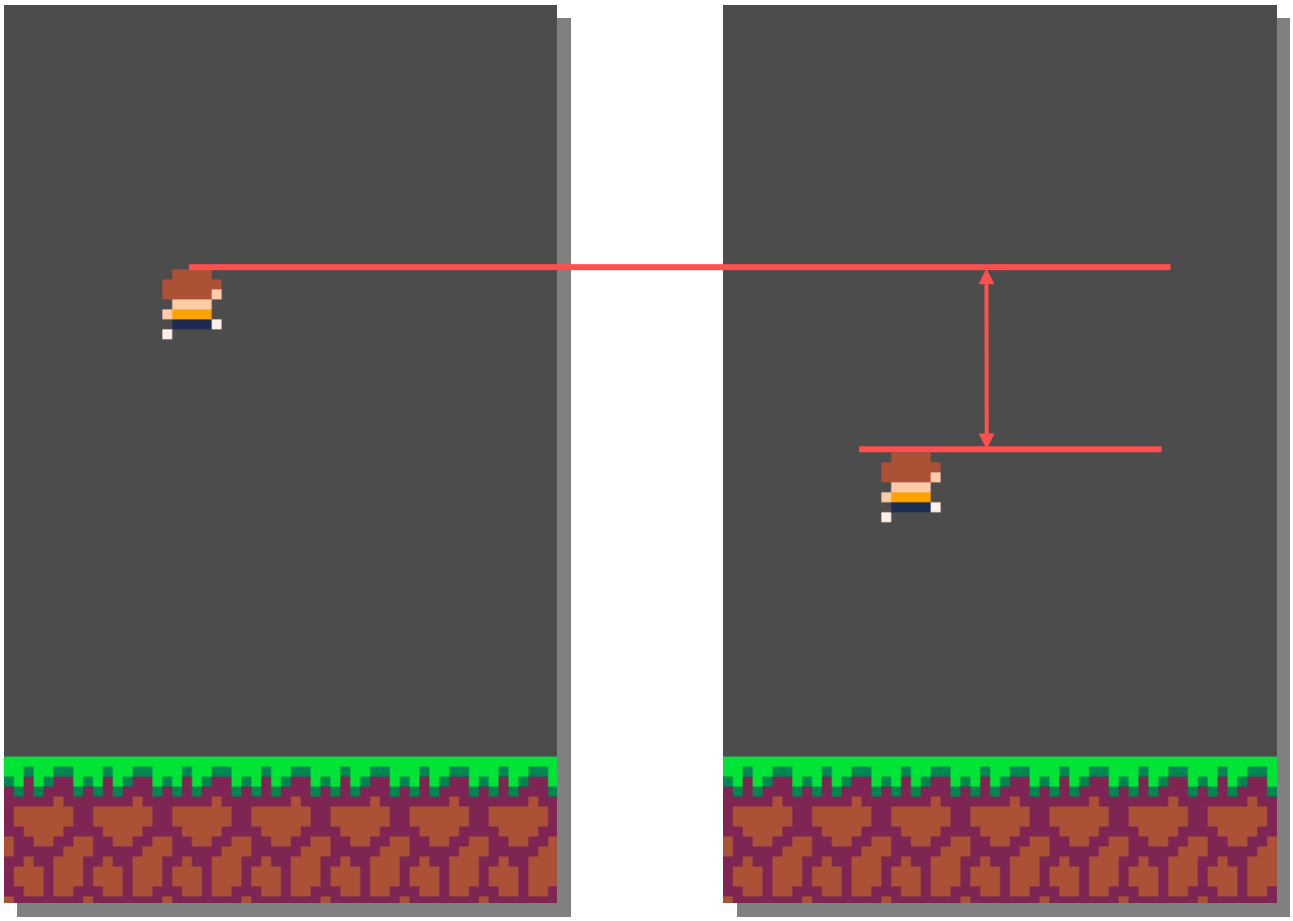
```
if x_input != 0:
    motion.x += x_input * accel * delta
    motion.x = clamp(motion.x, -max_spd, max_spd)
```

```
if is_on_floor():
    if Input.is_action_just_pressed("ui_up"):
        motion.y = -jump_force

    if x_input == 0:
        motion.x = lerp(motion.x, 0, ground_frict)
```

Finally, I wanted to add variable jump height, as was mentioned in the interview process. This will allow the player to have more control over the height of the jump, rather than just being a fixed height. To achieve this, I checked if the jump key was released, and if it was, I would set the vertical motion to half the jump force. This meant that gravity would pull the player down quicker, effectively reducing the height of the jump.

Unfortunately, whilst this allowed variable jump height, it would also allow the player to jump mid-air, indefinitely. To rectify this, I added an additional check to make sure that the player was moving up at a reasonably quick speed (half the jump height). This resolved the issue.

```
else:                                                   #checks that we're moving up quickly
    if Input.is_action_just_released("ui_up") and motion.y < -jump_force / 2: #variable jump height
        motion.y = -jump_force / 2

    if x_input == 0:
        motion.x = lerp(motion.x, 0, air_frict)
```



Interview of Features 1

To get feedback on the current features I have developed, I asked my stakeholder, Onyx B., about their opinion of the product so far.

Question 1: How do the controls feel for the player's movement?

"The controls are a bit slippery feeling. Like, the player doesn't stop very easily, but otherwise the controls feel very nice, responsive, and easy to understand."

Question 2: Are there any additional controls that you feel would improve the game?

"I think the controls are perfect for the game and adding more would over complicate it.

I appreciate the variable jump height, as it allows me to move the player with more precision. The game should allow the user to change the buttons used for controls, as some people may prefer to use the W, A, S, D keys, rather than the arrow keys."

Question 3: Are there any additional features, generally, that you feel are needed in the next stages of the game?

"The game now needs the rope mechanics, as well as some small animation for the player, as currently, the player sprite does not change, which would provide more feedback to the user."

I used the information from the interview to inform my next steps of development, making sure not to overcomplicate the controls when adding the rope mechanics, as well as adding more user-feedback, such as animation.

Raycasting from the Player

To allow the player to swing on a rope, I needed to perform a raycast from the player's position. This sends out a "ray" which returns the information of the tile it collides with. In this case, this will be the end of the rope for the player. After researching in Godot's documentation, I found a "RayCast2D" node, which would enable me to raycast from the player's position. I created a new child node for the player and assigned a script to it.

I created a custom function in the raycast node which would return the point the raycast collides with. To test this initial feature, I edited the player's script to detect when the left mouse button was clicked and used the location of the mouse as an argument in the function. It would then print this location to the output window.

```
119 func _unhandled_input(event):
120     if event is InputEventMouseButton:
121         if event.button_index == BUTTON_LEFT and event.pressed:
122             #Initialise rope swing
123             cast = rope_cast.cast_to_coordinate(get_global_mouse_position())
124 
125             print(cast)
```

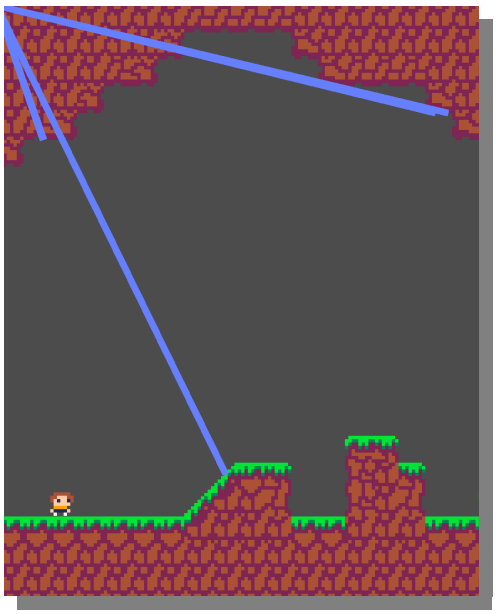
```
Output:
--- Debugging process started ---
Godot Engine v3.4.4.stable.official.419e713a2 - https://godotengine.
OpenGL ES 3.0 Renderer: NVIDIA GeForce GTX 960/PCIe/SSE2
OpenGL ES Batching: ON

-1
(125.170998, 24)
(25.749472, 32.000008)
```

```
1  extends RayCast2D
2
3  var point = Vector2(0, 0)
4
5  func cast_to_coordinate(coordinate):
6      var the_position = to_local(coordinate)
7
8      set_cast_to(the_position)
9      set_enabled(true)
10
11     if is_colliding():
12         if get_collider() is TileMap:
13             point = get_collision_point()
14
15             return point
16     else:
17         return -1
18
19
20     set_enabled(false)
```

Thanks to the output, I now knew that the raycast was:

- Colliding with a tile
- Returning -1 if the mouse was not over a tile
- Colliding with a different tile if the mouse moves

My next step was to draw a line to represent the initial raycast. This required the use of a “Line2D” node, which allowed me to display a line on the screen. I added some additional code to set the line’s points to the position of the raycast (which is the same as the player’s position, as it is a child node, so inherits position values) as well as the point of collision.

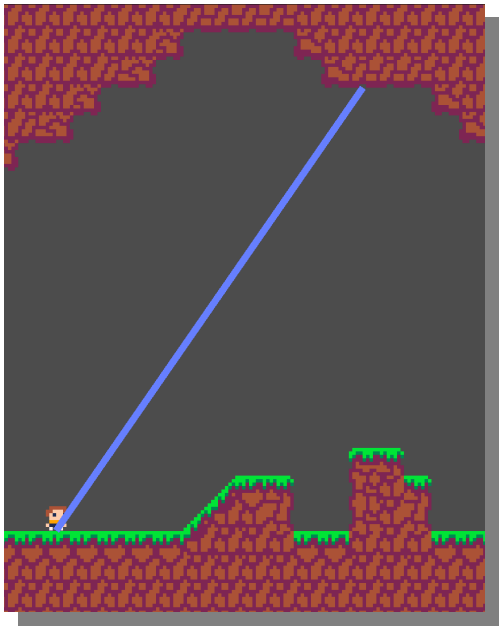


Unfortunately, this did not result in the expected outcome. The line was drawn to a seemingly arbitrary point on the screen, and each time I clicked, multiple lines were draw.

To resolve this, I reset the line’s points each time the mouse was clicked, which resolved the problem with multiple lines appearing.

```
line.add_point(position)
line.add_point(point)
```

I was initially unsure about what was causing the other issue, so I researched online into how Godot deals with positions. It quickly became apparent that the “Line2D” node requires the use of global positions, whereas the position of the player was local. Global positions are stored as vectors from the top-left corner of the screen, whereas local positions are stored as vectors from the position of the parent node.



Finally, to resolve the issue, I converted the position to a global one, and the line now draws the raycast correctly.

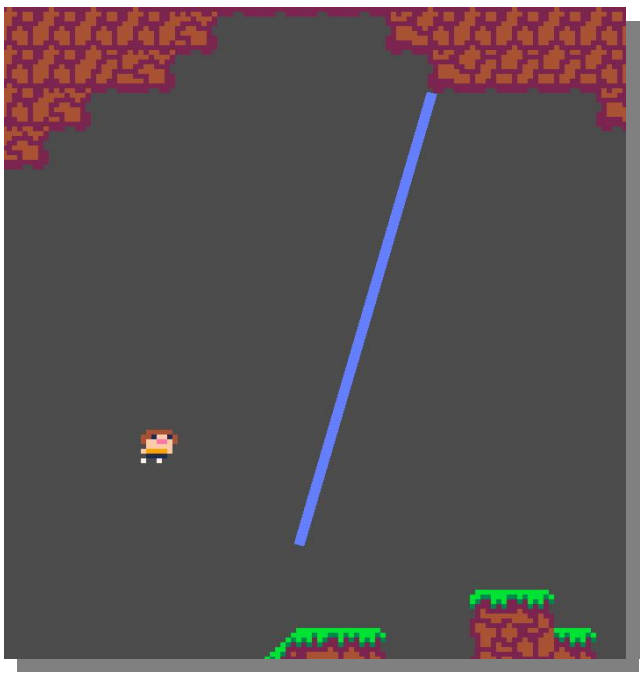
Moving the Player with the Rope

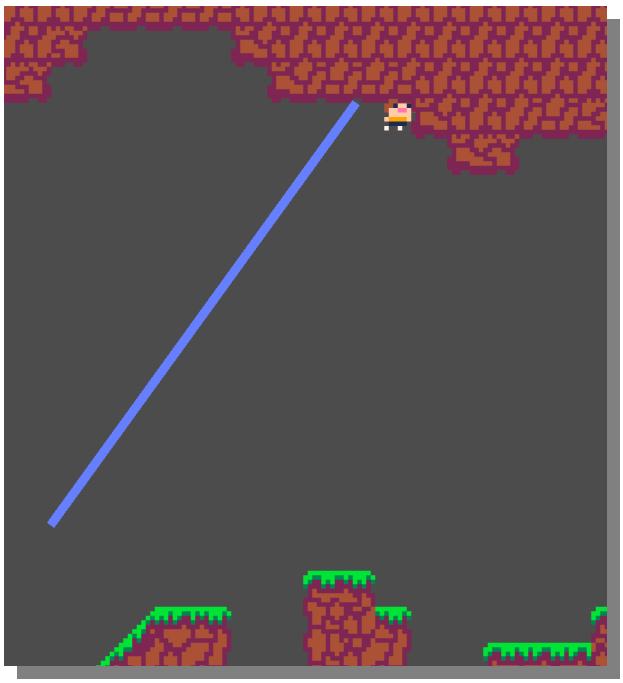
The next step to implementing the rope-swing mechanics was to make the player move with the rope. The first step to this was initialising the rope-swing state, which meant getting the angle and the length of the rope, to be used in calculations later. Finally, the player state variable was modified.

Now the player was in the swing state, I first wanted to make the rope move in a circle, with the player following it, but not yet controlling it. My first idea to achieve this was to create a new rope vector, which would have the same magnitude (length) as the initial raycast, but was rotated by an angle, incrementing by 1° each frame. This would have hopefully moved the player around in a circle; however, this did not end up happening.

```
131     if event is InputEventMouseButton:
132         if event.button_index == BUTTON_LEFT and event.pressed:
133             #Initialise rope swing
134             cast = rope_cast.cast_to_coordinate(get_global_mouse_position())
135             if typeof(cast) == TYPE_VECTOR2:
136                 local_cast = to_local(cast)
137 
138                 rope_angle = local_cast.angle()
139                 rope_len = local_cast.length()
140 
141                 player_state = state.swing
```

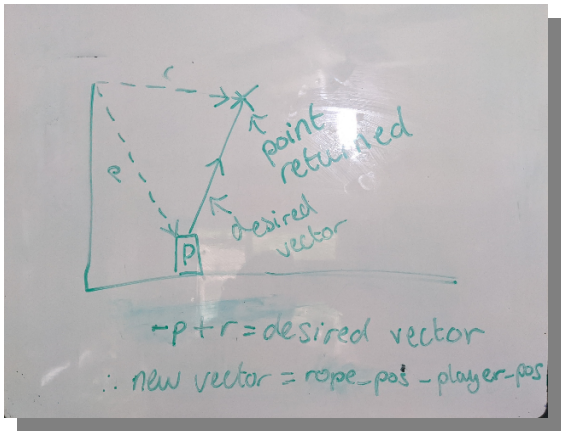
```
105             rope_angle += deg2rad(1)
106 #           rope_pos = Vector2(rope_len, 0).rotated(rope_angle)
107             new_rope = Vector2(rope_len, 0).rotated(rope_angle)
108             print(rope_pos, Vector2(rope_len, 0).rotated(rope_angle))
109 #           motion.x = rope_pos[0] - position[0]
110 #           motion.y = rope_pos[1] - position[1]
111 #           print(new_cast)
112             motion = local_cast - new_rope
113 #           print(motion)
```





Instead, the player seemed to move in a sporadic, and seemingly random way. If the increment angle was increased, it would “gravitate” towards the point of rotation, but just shake around it. Moreover, the line position did not update, however I was not concerned about this, as it should be simple to fix.

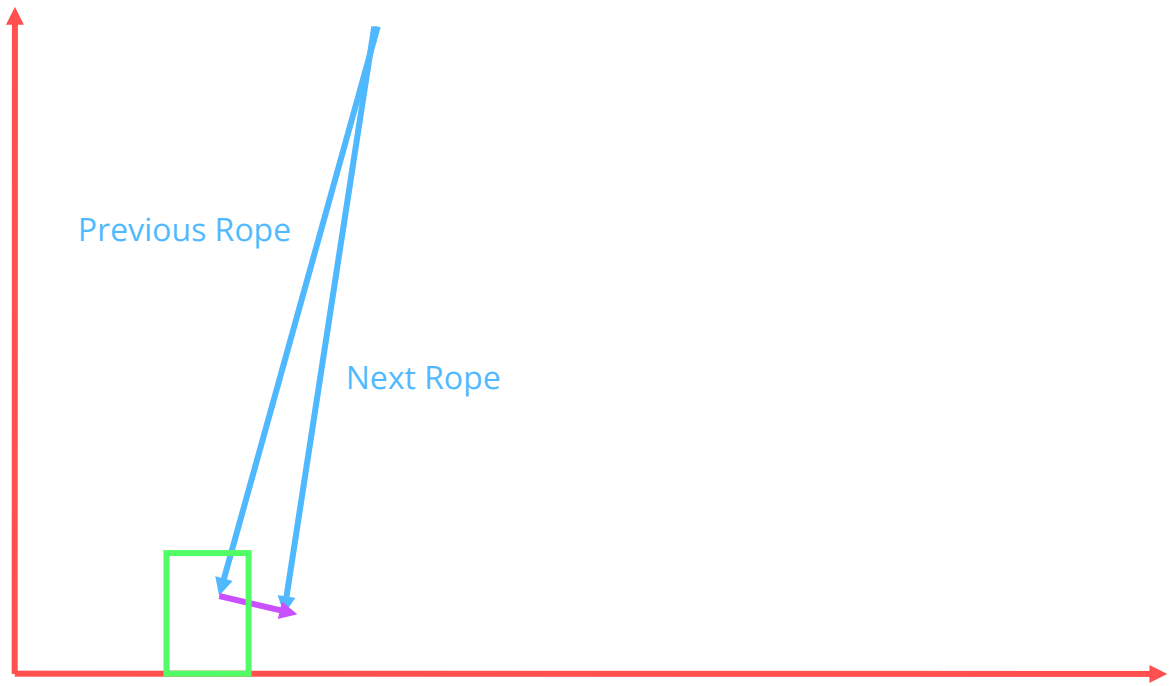
My initial thought was that the vector I supplied was simply incorrect. To help me visualise this, I created a diagram of what was needed:



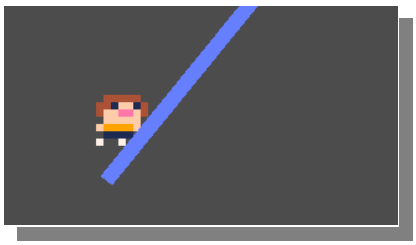
Based on my previous problem with the line, I assumed that a global position was being used, instead of a local one. To rectify this, I used vector maths to figure out how to obtain the desired vector. Unfortunately, this did not resolve the issue either and the same behaviour was seen.

```
var new_vector = rope_pos - position
rope_angle += deg2rad(100)
new_rope = Vector2(new_vector.length(), 0).rotated(rope_angle)
motion = local_cast - new_rope
```

My new solution was to find the vector between the previous rope position and the next one, and then apply this vector to the player’s motion:

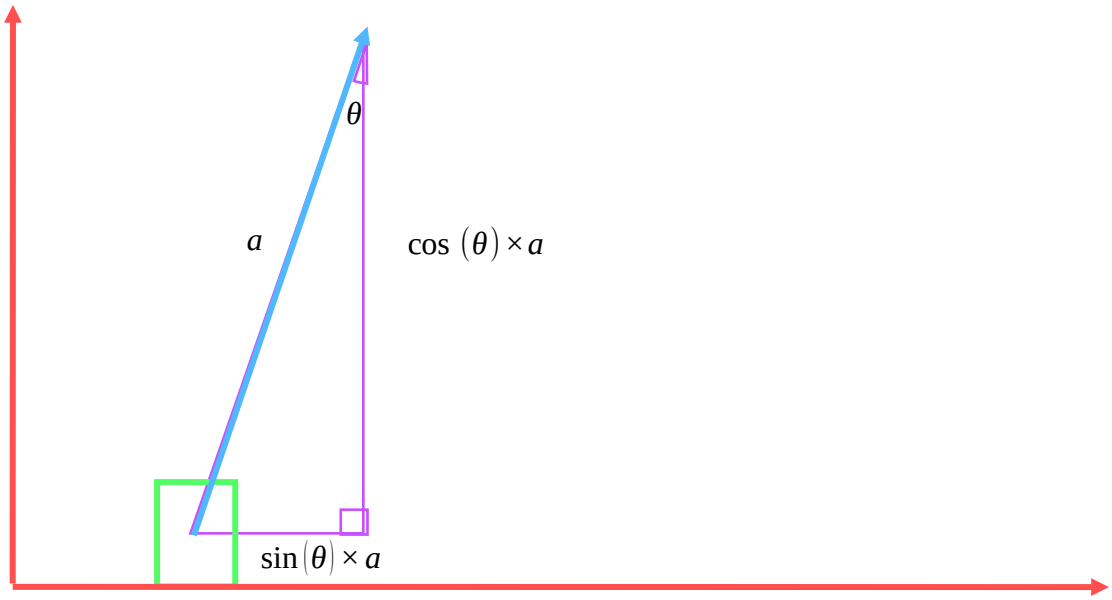


Unfortunately, this did not resolve this issue either. Although, it provided a much better visualisation of what the problem may be, as the player was rotating in a circle around the base of the vector.



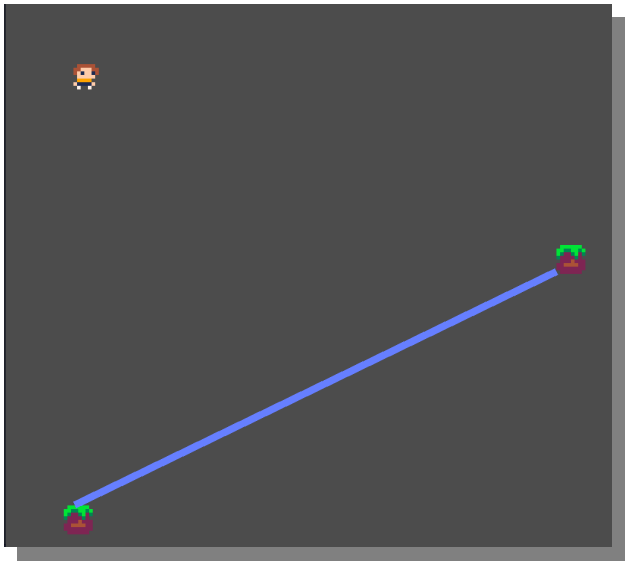
```
var local_cast_prev = local_cast
local_cast = -local_cast
local_cast = local_cast.rotated(deg2rad(10))
motion = local_cast + local_cast_prev

local_cast = -local_cast
```

My initial thought was that the “rotated()” method was rotating the vector from the centre, rather than from one of the ends. My new solution was to manually create a new position vector (rather than a motion vector) for the player, and manually assign both the x-values and y-values to a new rotation using trigonometry:

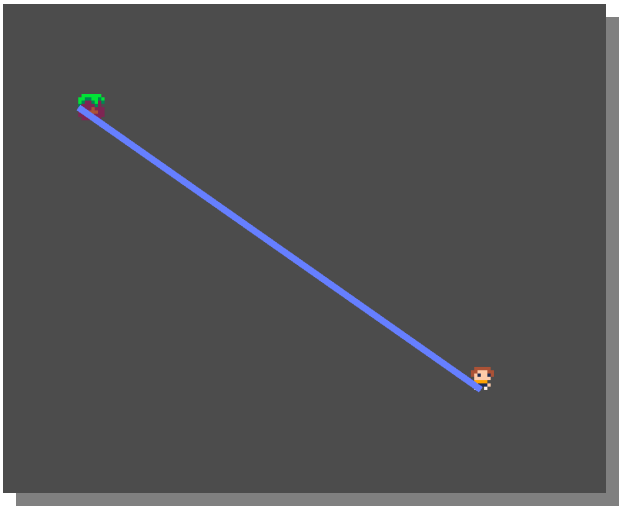


To rotate around the point, I simply set the angle, in the diagram, to a new angle, dependent on the delta-time, so it would update every frame. This was much closer to the desired effect; however, the player was rotating in the top left corner of the screen, rather than around the desired point.

To resolve this, I simply needed to add the position of the raycast, in order to offset the rotation, and the player now moves around the correct point. Finally, I added code to update the line's points.



```
85          state.swing:
86  #           local_cast_prev = local_cast
87
88              d += delta
89              var radius = rope_len
90              var speed = 0.5
91
92              position = Vector2(
93                  sin(d * speed) * radius,
94                  cos(d * speed) * radius
95              )
96
97  #           motion = local_cast - local_cast_prev
```



```
position = Vector2(
    sin(d * speed) * radius,
    cos(d * speed) * radius
) + cast

line.clear_points()
line.add_point(position)
line.add_point(local_cast)
```

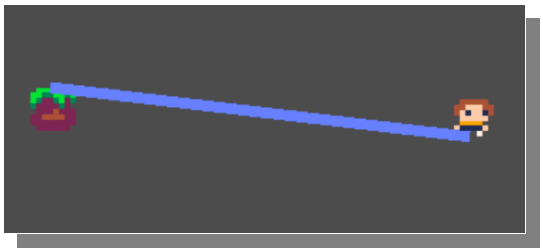
Allowing the Player to Control the Rope

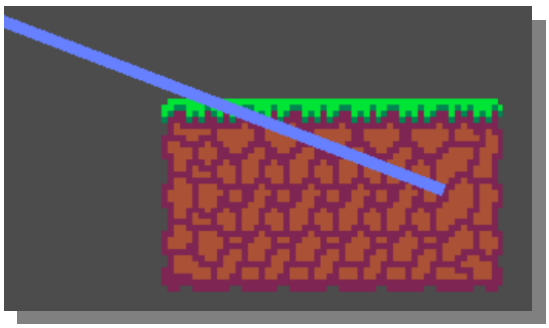
Now that the player was correctly "attached" to the rope, the final step was to allow the user to control the player whilst on the rope. To do this, I needed to modify the previous code to instead take a new angle based on the inputs for the platformer movement.

In order to change the length of the rope, only one variable needed to be modified: the radius (now renamed to "rope_len"). In order to make sure that the length of the rope was never too short or too long, I "clamped" this value to stop it from going over the maximum, or under the minimum value.

```
#Changes the length of the rope based on up / down
rope_len = clamp(rope_len + ((input_down - input_up) * 5), 50, 500)

#Calculates new rope position
var position_to = Vector2(
    sin(angle_to),
    cos(angle_to)
) * rope_len + rope_pos
```

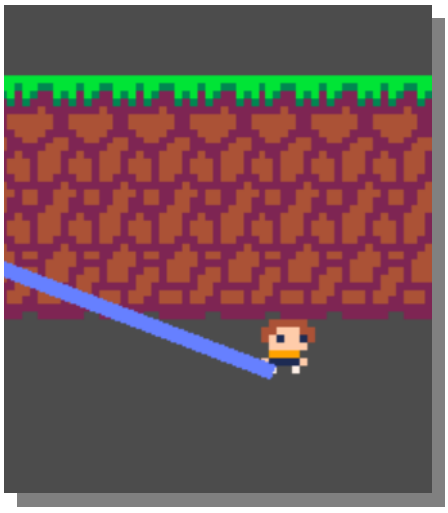
Now the player could control the length of the rope, at a fixed angle, I needed to apply the same logic to the “angle_to” variable, which would change the angle the new position vector is to be at. Unfortunately, at this point I realised that there were no longer any collisions with the tiles.





This was because I was manually updating the player’s position, instead of passing a motion vector through Godot’s built-in “move_and_slide()” method. To rectify this, I used the same logic that I tried previously, and created a new motion vector using vector maths. I then passed this into the player’s “motion” variable, and collisions worked.

```
#Changes the length of the rope based on up / down
rope_len = clamp(rope_len + ((input_down - input_up) * 5), 50, 500)

angle_to = angle_to + ((input_right - input_left) * 0.05)

#Calculates new rope position
var position_to = Vector2(
    sin(angle_to),
    cos(angle_to)
) * rope_len + rope_pos

#Adds rope points to the line
line.clear_points()
line.add_point(position)
line.add_point(rope_pos)

#Calculates the movement from the current position to the new one
motion = position_to - position
```



However, when colliding with the tiles, the movement “snapped” into places. I theorised that this was a build-up of momentum due to the “angle_to” variable still being increased, even if the player was colliding.

To rectify this, if the player was colliding with anything, I would reverse the angle added to the variable, which meant that the player would simply "slide" across the tiles, rather than continually building up momentum.

```
#If not colliding with something
if get_slide_count() == 0:
    #Changes the angle of the rope based on left / right
    angle_to = angle_to + ((input_right - input_left) * 0.05)
else:
    #Inverse the change of angle while colliding
    angle_to = angle_to - ((input_right - input_left) * 0.07)
```

Now all that was necessary was the ability to "let go" of the rope. This was a simple as resetting the player's state back to the initial "platformer" state, as well as clearing any rope lines.

```
#Resets player state out of rope state
if Input.is_action_just_released("ui_select"):
        line.clear_points()
        player_state = state.normal
```

Interview of Features 2

Now that the basic rope mechanics were implemented, I once again conducted an interview with my stakeholders.

Question 1: Compared to the last build, how do the controls of the platforming feel now?

"The controls have improved due to the added friction for the player, meaning it is now much easier to perform jumps between platforms"

Question 2: How do the new rope mechanics feel, especially compared to the platforming mechanics?

"The rope mechanics feel very well integrated into the game, and it is not jarring to switch between the two. I think that the rope mechanics now need to have more resistance against the player, and to gradually fall back down, like a pendulum, in order to provide a more realistic experience"

Question 3: Are there any other features that should be added at this stage of development?

“It can be a bit hard when swinging between ropes to hit the next platform. It should be easier to aim the rope mid-air.”

Thanks to this interview, I knew that I now needed to implement pseudo-gravity mechanics into the rope state, as well as allow the mouse to follow the camera. This is what I began to implement in the next stages of development.

Moving the Camera with the Mouse

As per the interview, a major problem was that it was very difficult to “aim” the rope at anything, since the camera always followed the player, and did not account for where the mouse was. If the camera followed the player and the mouse, always keeping both on screen, it would allow for much easier aiming.

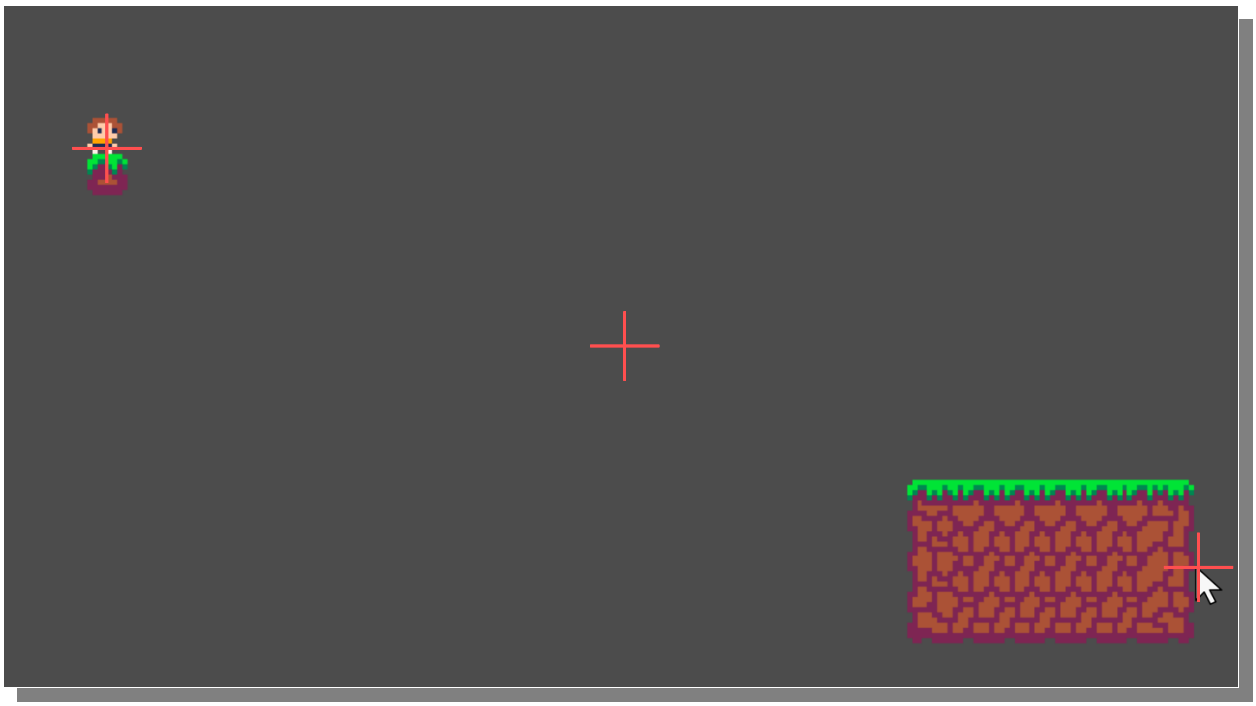
The first step to this was adding a new “Camera2D” node. This would give me much more control over the camera. Crucially, I needed this to not be a child of any other node in the room, as it needed to have its own position that I could manually calculate and update. This was done by assigning a script to the node, with a function that would update every frame.

I first wanted to get the camera to follow the player, but smoothly in order to not make quick jumps and fast-paced sections jarring, and difficult to follow. To achieve this, I linearly interpolated the camera’s position to the player’s position, which allowed it to move smoothly.

```
5  var interpolate_val = 2
6
7  func _process(delta):
8
9      var target = player.get_global_position()
10
11     global_position = lerp(global_position, target, interpolate_val * delta)
12
```

Next, I wanted to incorporate the centre position between the player and the mouse. This was as simple as taking the mean of the x and y component of their vectors, and then using that in the linear interpolation. This meant that the camera now kept both the player and the mouse on screen, while making it much easier to aim.

```
var mid_x = (target.x + get_global_mouse_position().x) / 2
var mid_y = (target.y + get_global_mouse_position().y) / 2

global_position = lerp(global_position, Vector2(mid_x, mid_y), interpolate_val * delta)
```



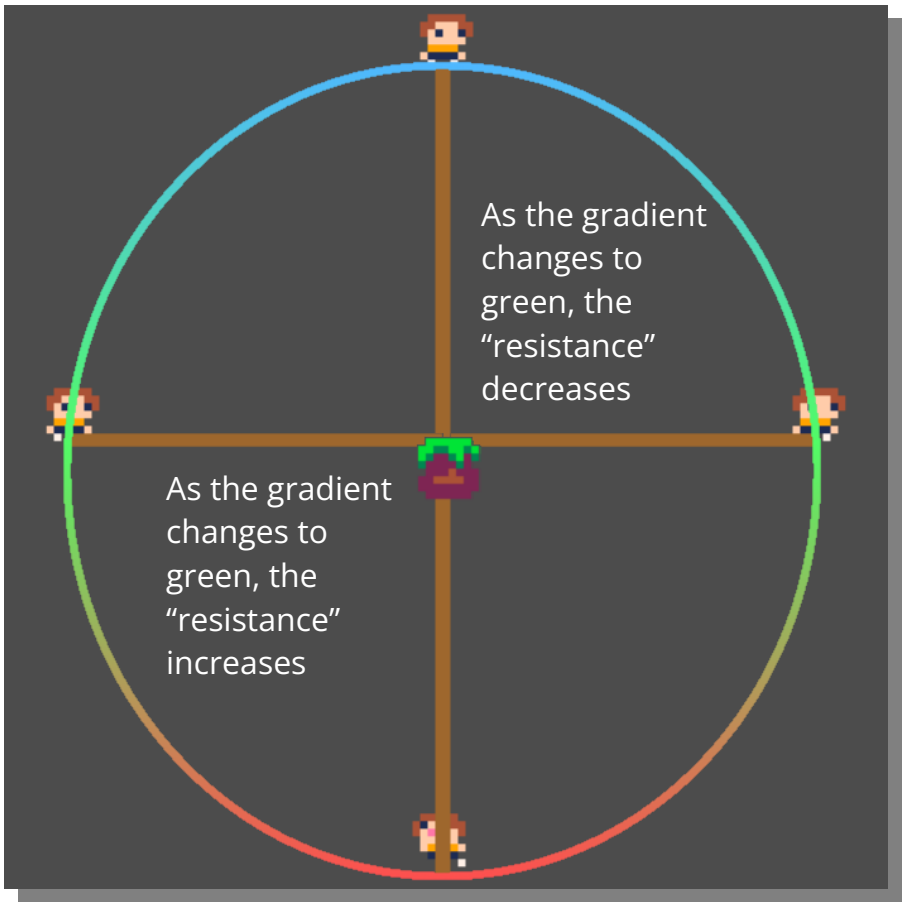
Finally, I wanted to make the camera move quicker if the player was moving quicker, allowing for more precision, as well as making sure the player is never off screen. Thanks to the linear interpolation, this is as simple as modifying the interpolation weight, making sure this only changes if the player's velocity is sufficiently high.

```
#Use player's velocity as the lerp value to stop player from going off screen
#But keep the minimum at 2 (any lower velocity will still allow camera to move)
var player_vel = player.motion.length() * 0.03
if player_vel > 2:
    interpolate_val = player_vel
else:
    interpolate_val = 2
```

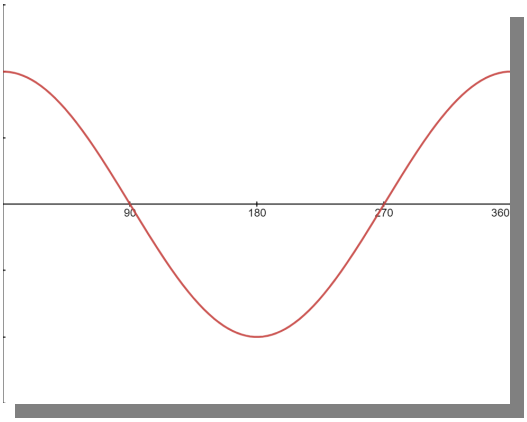
Adding "Gravity" to the Rope

Now everything was working, although the rope physics felt a bit off. As said in the interview, this was because there was no "gravity" on the rope, i.e., it was just as easy to move the rope at the bottom of the "circle" that the rope goes around, as it was at the top.

To rectify this, I needed to apply an additional “change in rope angle”, known as the “rope angle velocity”. This new “rope angle velocity” would also diminish by a half each frame, in order to add a perceived effect of friction around the rope’s pivot. Most importantly, this “rope angle velocity” had to be higher if the player was at the sides of the circle, as this is where gravity would affect the player the most.



The hardest part was figuring out how to implement this. I figured the easiest way was to use a sine or cosine curve; if the angle was put into a trigonometric function, it will return a value between -1 and 1, which will change, gradually, depending on how far around the circle the player is in 90° increments. This was perfect for what I needed.



The function I needed ended up being cosine, since the angles in Godot start at 0° to the “east” / right. The function needs to return the highest value, i.e., 1, at this point, and the lowest value, i.e., -1, at the opposite point (180°). This can then be added to the “rope angle velocity”, which is then halved (every frame) and added on to the new rope angle.

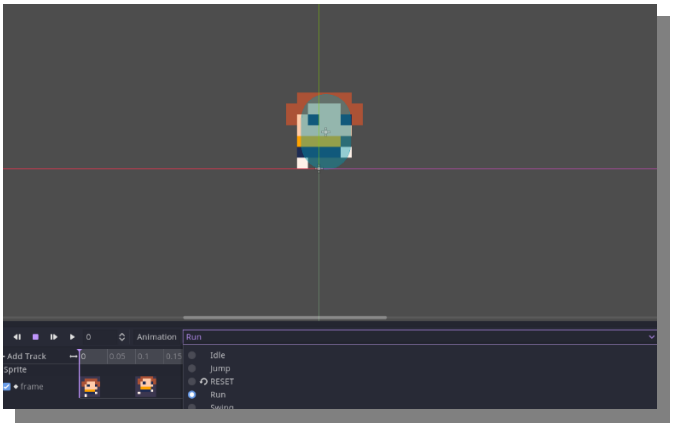
This will successfully add gravity to the rope physics, adding a suitable degree of realism.

Now that the mathematics were figured out, finally implementing this system was an easy task.

```
var rope_angle_accel = 0.03 * cos((rope_pos - position).angle())
rope_angle_vel += rope_angle_accel
rope_angle_vel *= 0.5

angle_to += rope_angle_vel
```

Adding Animations

Now for an easy part: adding animations. To do this, I used Godot's built-in animation editor by adding an "Animation" node to the player "scene". Here, I could add animations for standing idle - although I kept just a single frame – running, jumping, and swinging on the rope.



Once the animations were added, I added code to run them at the appropriate time.

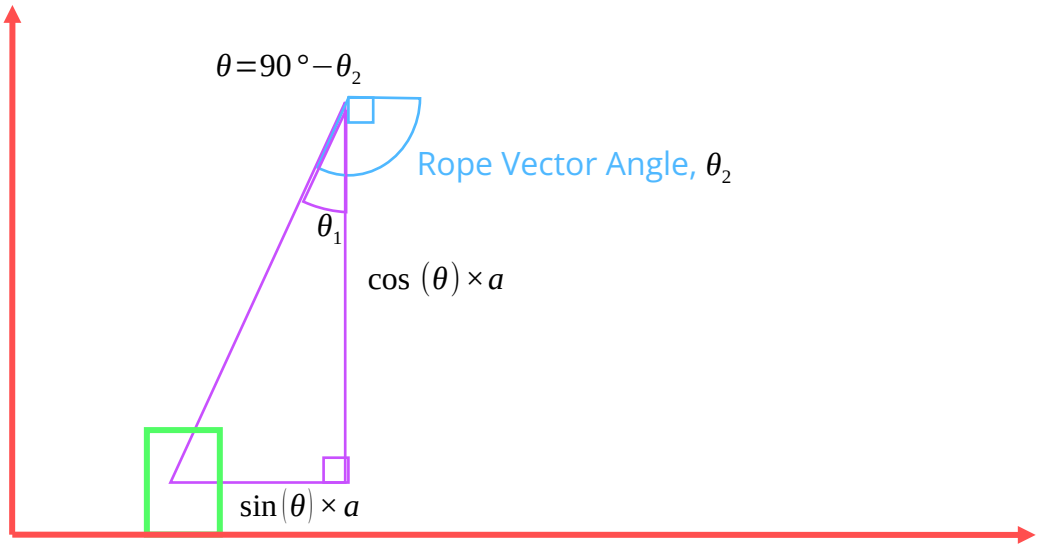
Now the game felt much more "alive". An example piece of code is below, although most animations are handled elsewhere.

```
if floor(abs(motion.x)) == 0:
    animation.play("Idle")
else:
    animation.play("Run")
```

Fixing the Raycast

Whilst testing out the new rope functionality, I noticed a new bug that could have been present for a while. Essentially, the rope would "jitter" or "fling" the player across the level when it was initialised. Because the movement was totally fine afterwards, I knew this was a problem with the calculation of either the rope length, or the rope angle. To figure out which one it was, I temporarily disabled the additional "gravity" code, so I could see the rope without any additional forces.

This allowed me to see that the rope angle was being incorrectly calculated, as it needed to be offset by a certain amount. This required an extensive amount of vector maths, which in the end, gave me this diagram.:



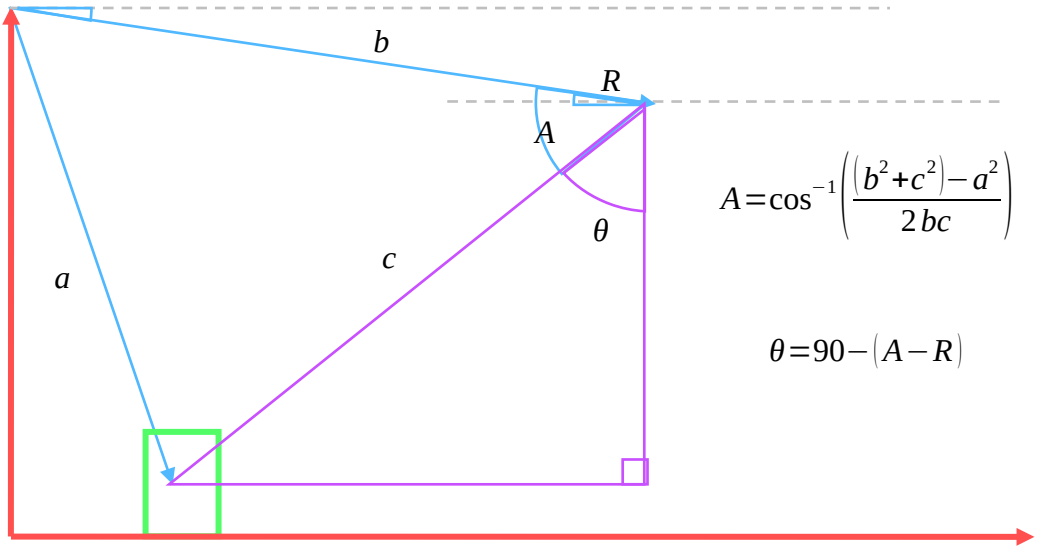
Essentially, the angle returned by Godot is not the angle needed in the calculations later, so it was always offset by 90°, which meant it tried to correct this and “snap” back into place.

Implementing the fix to the code is extremely simple and resolves any problems with the rope’s angle.

```
rope_pos = cast
rope_len = (position - rope_pos).length()

angle_to = deg2rad(90) - (position - rope_pos).angle()
```

This was not an easy answer to come by, however. It took many hours of reasoning to decipher why it was not working. Originally, I theorised that 180° had to be taken away, instead of 90°, since the vector angle method may have been returning the opposite side of the vector.

I did find a solution before the one above, but it required the use of the cosine rule, which I felt was not the most efficient, and also easy to understand, solution:



This worked, but I was unhappy with the complexity of the maths, as well as how the value was slightly different for each 90° "quadrant" of the rope's angle. Implementing this into code made me realise that there must be a simpler solution.

```
var a = position.length()
var b = rope_pos.length()
var c = rope_len

var A = acos(((pow(b, 2) + pow(c, 2)) - pow(a, 2))/(2 * b * c))
var R = rope_pos.angle()

angle_to = deg2rad(90) - A + R
```

Interview of Features 3

Now that all the features requested from the previous interview have been implemented, I decided to conduct an additional interview to figure out the final features that are necessary to be implemented in the game.

Question 1: How do the new additions to the rope mechanics feel to control?

“The rope mechanics are now exactly as I hope they would be. There is a ‘tightness’ to the control, which provides a lot of control when making complex manoeuvres between both the platformer and the rope swing state.”

Question 2: How do the new visual features (e.g., the additional animations) feel from a player’s perspective?

“The new animations provide two key details for the game: a personality for the character, which creates an important aspect of sentimentality, which I requested in earlier interviews before development; as well as feedback to the player when changing states and making jumps. This makes any movement in the game feel much more impactful, making it more enjoyable to control.”

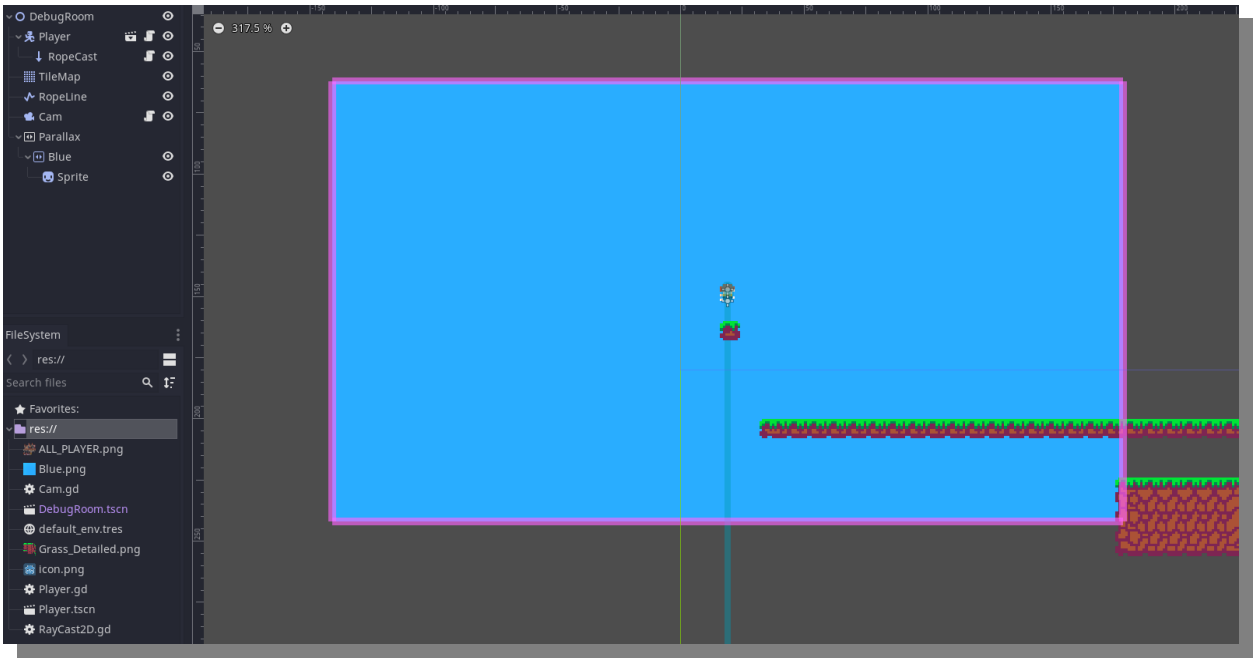
Question 3: What additional features do you feel should be implemented?

“The game now needs some visual polish – which can be created with small details, such as a background – as well as some utility features, such as a GUI, a main menu, and a pause menu. It also needs some features to make it more into a platforming game, rather than a ‘sandbox’ to swing about in.”

This interview allowed me to plan out my next stages of development, which would involve adding more graphical polish, as well as important features, such as pausing, scoring, and a main menu. Crucially, I decided I needed to add a checkpoint system, and also level hazards, to allow for more in-depth level design.

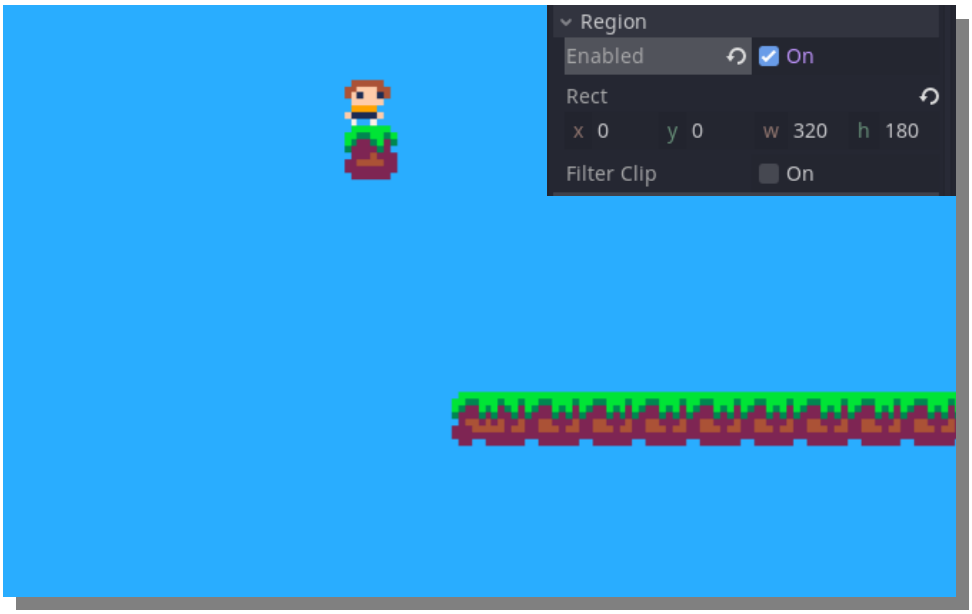
Adding a Background

The first objective I decided to tackle was adding a background. Once all my initial features are finished, I would like to implement parallax scrolling, which splits the background into several layers that scroll at different speeds than each other, which provides the player with a sense of depth. However, this is not necessary for my game, so I will focus on more important features first.



Adding a standard background is a simple task in Godot, due to the various nodes. The first node I needed was a “ParallaxBackground” node, which will allow me to add parallax scrolling in the near future. I then added a child “Background” node, which finally needs a child “Sprite” node, similar to the player’s sprite.

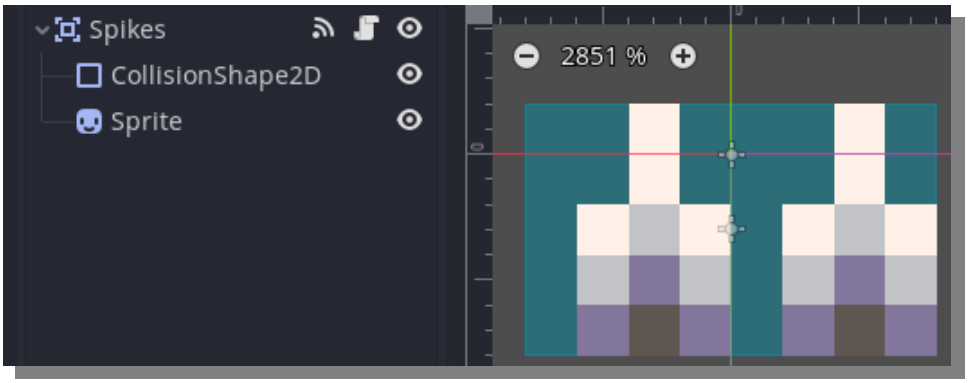
Now that the nodes were set-up, I simply needed to make the background repeat itself indefinitely, rather than just in a small rectangle. To do this, I needed to add a region that was the same width and height as the camera. This makes the background “tile” itself, which is perfect for what I need. Doing this is as simple as modifying the attributes in the sprite node, and once that was done, I had a background in my game.



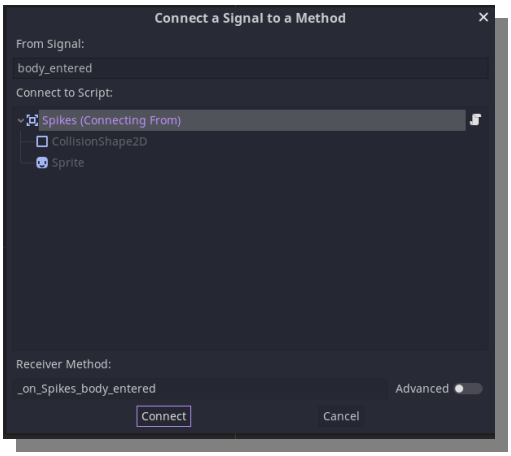
Adding Level Hazards

My next plan of action was to add level hazards. This is important to allow for a variety of level designs, as the only current hazard is falling. My plan was to add generic spikes that would kill the player if it collided with them.

The first step was to add another node – an “Area2D” node, since I only need to know if the player has collided, and not act as a solid object, like the tiles. I gave the “Area2D” node two child nodes: a “CollisionShape2D”, and a “Sprite” node. This will allow me to assign the region where the player will collide with it and put a sprite on top to provide a visual for the player. I set the “CollisionShape2D” node to be a rectangle that surrounded the sprite, and then converted the entire “Area2D” node to a scene. This will allow me to re-use the spikes throughout the level, without having to duplicate them.



Once my collision was set-up correctly, I needed to add a signal. Signals are events that attach to functions, which will run once the event happens. In this case, I set-up an event that triggers whenever a body (i.e., the player) enters the spikes’ collision rectangle, and attach this to a function named “_on_Spikes_body_entered()”.



This then meant I could call a custom procedure inside the player node which will run the code for the player’s death.

```
3  onready var player = get_parent().get_node("Player")
4
5  func _on_Spikes_body_entered(body):
6      player.die()
```

To test this, I temporarily assigned the “die()” function to print a message. This function will then be updated once the checkpoint system is implemented.

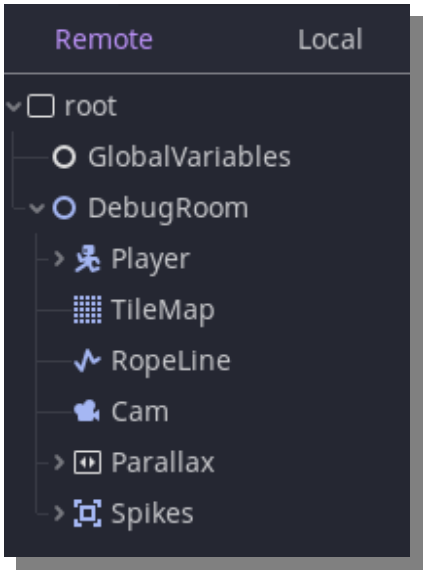
```
197 func die():
198     print("die!")
```

```
Output:
--- Debugging process started ---
Godot Engine v3.4.4.stable.official.419e
OpenGL ES 3.0 Renderer: Intel(R) Iris(R)
OpenGL ES Batching: ON

die!
die!
```

Adding a Checkpoint System

The checkpoint system will allow the player to repeatedly attempt a difficult portion of the level, without having to restart from the beginning. To implement this, I first needed to create a global 2D Vector to store the position of the checkpoint. This can be done using a “singleton”, which auto-loads a script, or scene, and keeps it loaded whilst the game is running.

Firstly, I created a new script called “GlobalVariables”. Unlike all other scripts so far, this was not attached to any node to begin with. I then created a new singleton called “GlobalVariables”, and assigned it said script. This can be seen when the game is loaded.



The node tree shows a “GlobalVariables” node outside of the “DebugRoom” node (where all other nodes are a child of). This means that when a new scene is loaded, or the current scene is reloaded, the global variables node is not reloaded, which is perfect for this use-case.

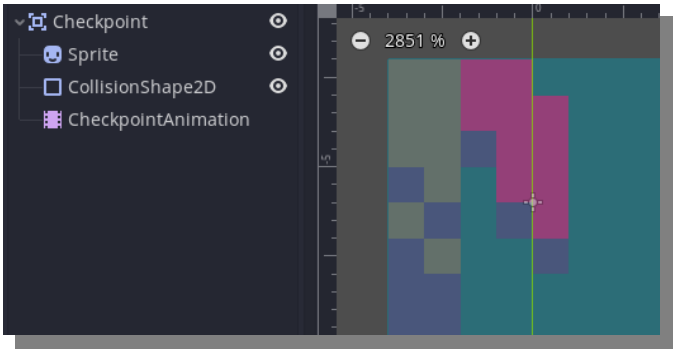
Inside the attached script, I added a variable called “checkpoint_pos” and set it to a 2D vector at (0, 0) as a default.

```
1  extends Node
2
3  var checkpoint_pos = Vector2.ZERO
```

Finally, I modified the “die()” function to update the player’s position to this new global position, and tested the functionality by running into the spikes. This teleported the player to the (0, 0) position.

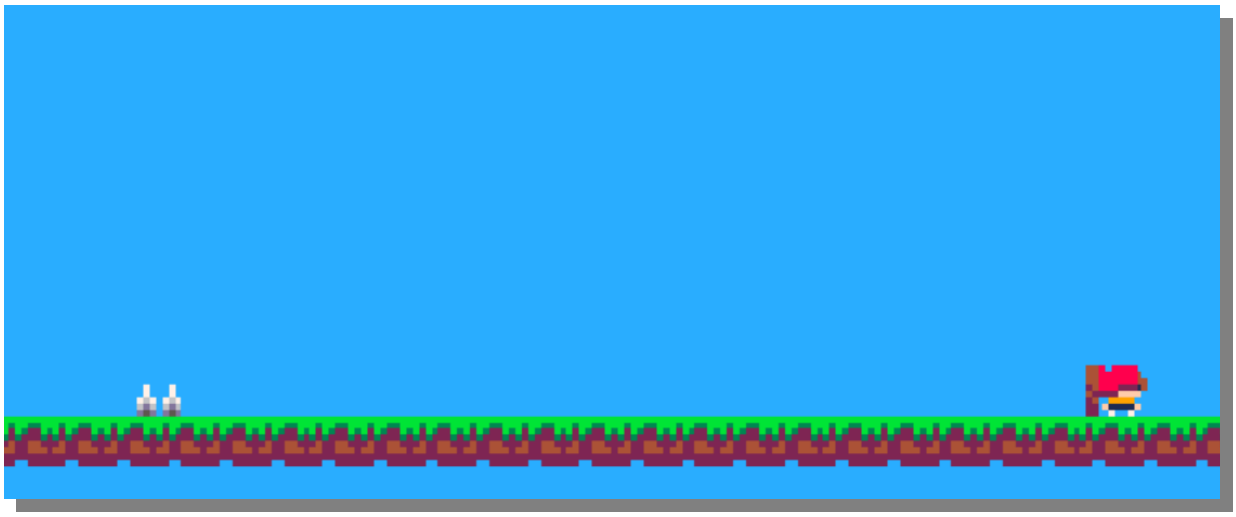
```
197 func die():
198     position = GlobalVariables.checkpoint_pos
```

Similar to the spikes, I added a new “Area2D” node, with a “CollisionShape2D” and “Sprite” node as children. Additionally, I added an “AnimationPlayer” node, which would control the animation for the flag, which will be used to indicate whether the checkpoint was active or not.



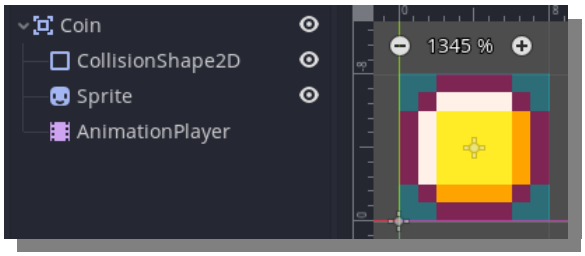
Once this was done, I added another signal to the checkpoint flag, which would update the animation and the global variable if collided with. This meant that if the player collided with the spikes, they would be teleported to the checkpoint. Implementing this and testing this out demonstrated that it indeed worked.

```
3  onready var animation = $CheckpointAnimation
4
5  func _process(delta):
6      if GlobalVariables.checkpoint_pos != position:
7          animation.play("RedStill")
8
9  func _on_Checkpoint_body_entered(body):
10     animation.play("RedActive")
11     GlobalVariables.checkpoint_pos = position
```

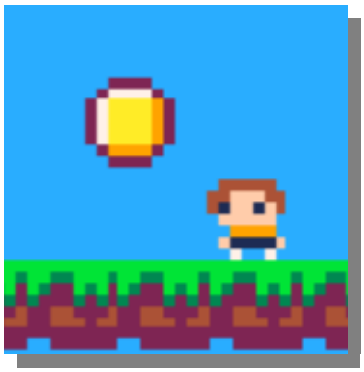


Adding Collectibles

Now that I had the basis for nodes that could be collided with, and also global variables, it was not difficult to extend this to add collectibles. To do this, I once again created a new “Area2D” node, with “CollisionShape2D”, “Sprite”, and “AnimationPlayer” nodes as children. I created a function to run on the “body_entered” signal and made this increment the global “coin_count” variable.




```gdscript
func _ready():
    animation.play("CoinAnimation")

func _on_Coin_body_entered(body):
    GlobalVariables.coin_count += 1
    print(GlobalVariables.coin_count)
    queue_free()    #deletes the instance

```



```
Output:
--- Debugging process started ---
Godot Engine v3.4.4.stable.official.419e713
OpenGL ES 3.0 Renderer: Intel(R) Iris(R) Pl
OpenGL ES Batching: ON

1
--- Debugging process stopped ---
```

I tested the functionality by printing the coin count to the screen, which successfully printed out “1”.

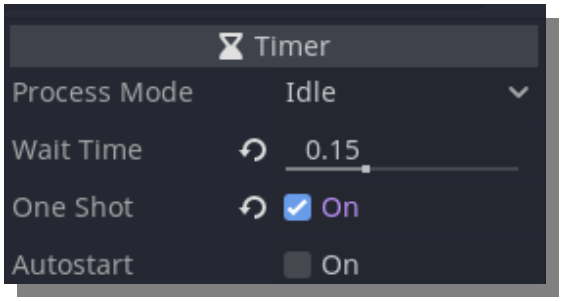
Adding Coyote Time and Jump Buffer

“Wile E. Coyote, super genius.” Now that all other in-game mechanics were sorted, I decided to tackle the lesser-needed, but quality-of-life mechanics, such as coyote time and a jump buffer. As mentioned in the design section, coyote time allows the player to jump even if they are not on the floor, as long as they only left the floor a few milliseconds before. Similarly, a jump buffer will register the player’s jump even if they pressed the jump key a few milliseconds before they were on the ground. Both methods recognise the players intention.

The first step to adding coyote time was to add a timer child node to the player – this will handle timing the few milliseconds where the player can still jump.

I set this timer to 0.15 seconds in Godot, and made sure to enable “One Shot”, meaning the timer will not repeat.

The next step was to add in the functionality via code. To do this, I added a new variable that temporarily stored whether the player was on the ground in the previous frame. Then, after the player’s position updates, I check to see if the player is no longer on the floor, if so, the timer starts.

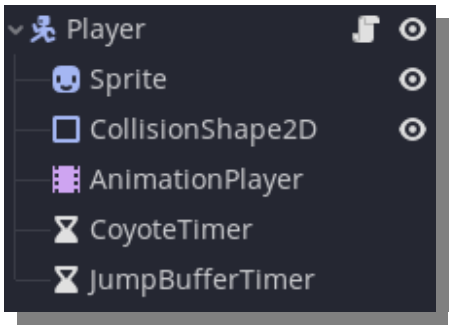



```
# Checks to see if the player is on the floor before the move_and_slide
# function updates the player's position
var floor_before_move = is_on_floor()

motion = move_and_slide(motion, Vector2.UP) # Moves the player node by the vector + automatically collides
                                            # Also returns left over motion, meaning if collided
                                            # It will return no movement, and stop for the next frame

# If the player's new position is not on the floor, but it was the previous frame
if floor_before_move && !is_on_floor():
    coyote_timer.start()
```

Finally, I modified the existing jumping code to check if either the player was on the floor, or the timer was still running.

```
var coyote_on_floor = is_on_floor() || !coyote_timer.is_stopped()

if coyote_on_floor:
    # Floor movement code
```

The jump buffer was added in a similar way. First I added a new timer node to the player, with the same settings as the coyote timer.

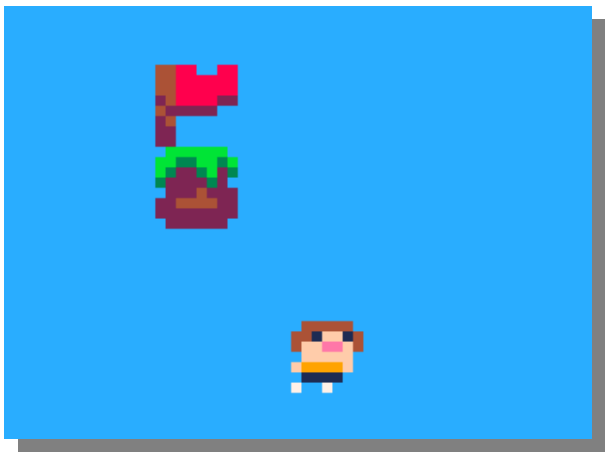


Next I needed to add the jump buffer into code. To do this, I modified the existing code to test whether the jump buffer timer had not stopped. If it hadn’t, then the player would jump, and the timer would stop to prevent multiple jumps. Above this, I checked whether the player pressed the jump button, and if so, the timer would start.

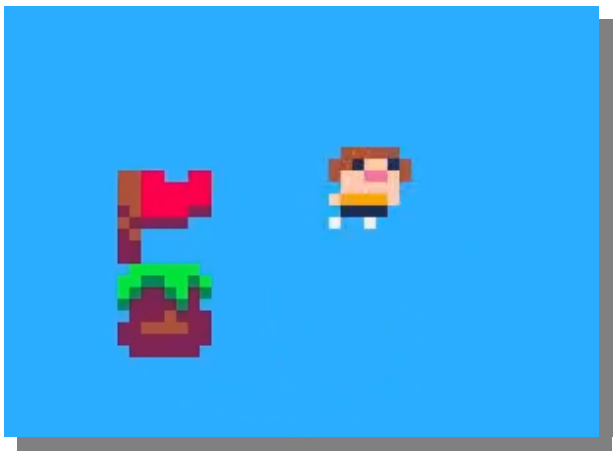
```
if input_jump:
    jump_buffer_timer.start()

if coyote_on_floor:
    #Floor movement code
    motion.x = clamp(motion.x, -max_ground_spd, max_ground_spd)

    if !jump_buffer_timer.is_stopped():
        motion.y = -jump_force
        jump_buffer_timer.stop()
```

In the diagram below, you can easily see coyote time in effect, as the player has jumped whilst not on the floor. Unfortunately, the jump buffer cannot be shown via an image, as it looks as if the player has jumped normally, as intended.

Without Coyote Time



With Coyote Time

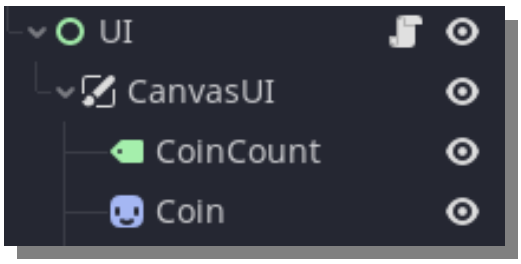


Adding a GUI

Now that all the game mechanics were added, it was time to add a GUI (graphical user interface). A GUI will allow me to display various statistics that the player needs to know in game, such as the coin and death count. To give the game a clean aesthetic, I want these counts to only show when the coin or death count is updated, or if the player presses the down button in the normal state.

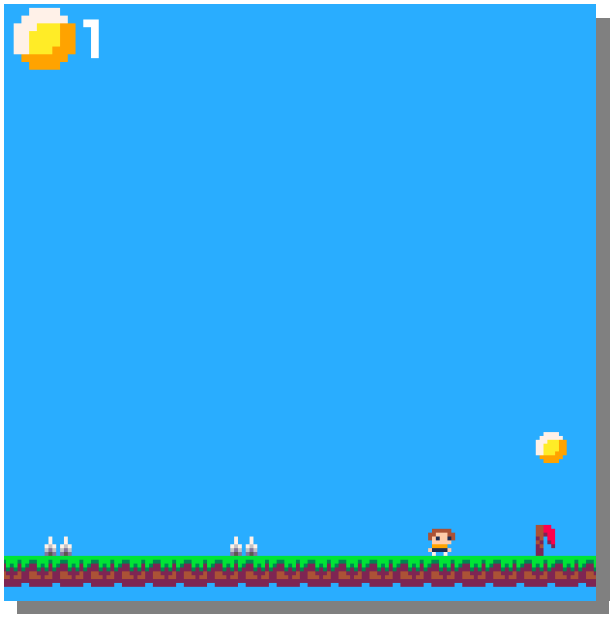
To start, I added a new “Control” node to the Godot scene. The control node allows me to control various parts of the game’s screen, unlike the “Node2D” node, which controls various parts of the game’s world. All previous nodes, apart from the Timer nodes, have been children of Node2D, since everything affected / was affected by the game’s world.

I then added a child “CanvasUI” node, which ensures that any child nodes will stay on screen, and not move away if the camera does. This is important as the UI should always be visible when the player needs it. I added two more child nodes to the CanvasUI – a sprite and a label. The label allows me to display text, which will be the actual count, and the sprite allows me to display an icon, such as a coin, to show what the label is displaying.

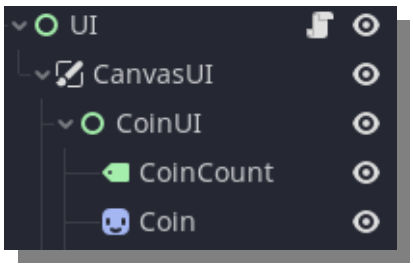


Next, I attached a script to the UI Control node, which will contain any code to modify the label, and the UI’s position. To do this, I used the coin count global variable, and updated the label every frame, ensuring that it will be always up to date.

```
37 func _process(_delta):
38     coin_count.text = str(GlobalVariables.coin_count)
39
```

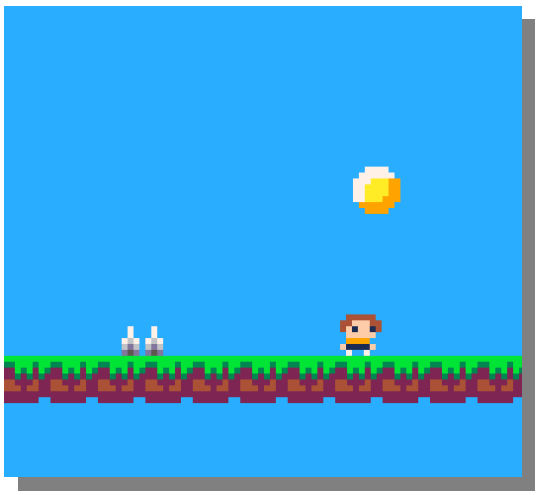


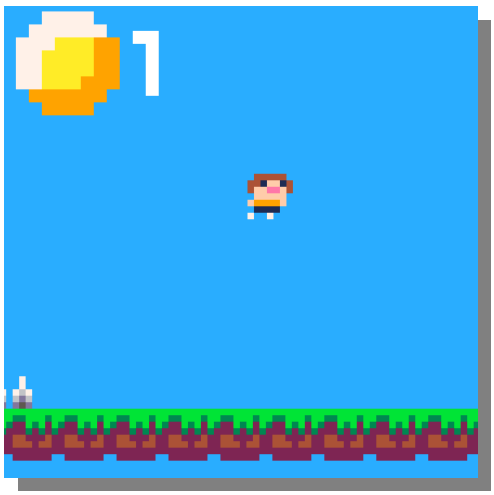
Now that the coin count was updating, I decided to add the the functionality to allow the UI to appear and disappear. This can be done by modifying the y value of the label and sprite. To make this easier, I created another control node, and made both the label and the sprite a child of it.



I then added a timer node, as a child of the “CoinUI” node, which would keep track of how long the UI should be on screen. I set this up to be 2.5 seconds, meaning every time a coin was collected, the UI would show for 2.5 seconds, and then hide again.

To get this functionality in code, I created a variable to keep track of the previous coin count; if this previous count was not the same as the current count, meaning a coin was collected, I would start the timer. Then, I check to see if the timer was still counting down, and if so, the CoinUI node would linearly interpolate to the on-screen position. To get the on-screen position, I simply save the value as the game is loading. If the timer was not counting down, i.e., had stopped, the CoinUI node would instead linearly interpolate off-screen. This worked: now the UI only shows up once a coin has been collected, and smoothly appears and disappears by "sliding" on and off-screen .





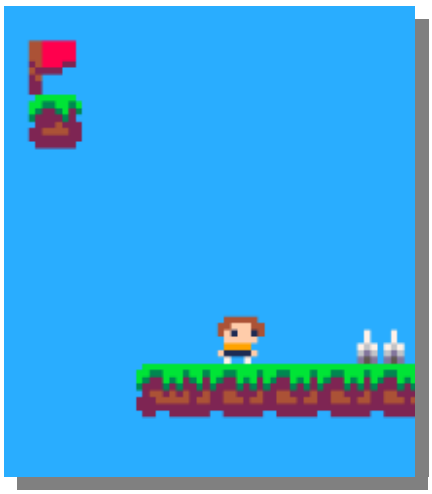

```gdscript
func _process(_delta):
    coin_count.text = str(GlobalVariables.coin_count)

    if previous_coin != coin_count:
        coin_timer.start()

    if !coin_timer.is_stopped():
        previous_coin = coin_count

        coin_ui.rect_position.y = lerp(coin_ui.rect_position.y, ui_onscreen, 0.1)
    else:
        coin_ui.rect_position.y = lerp(coin_ui.rect_position.y, ui_offscreen, 0.1)
```

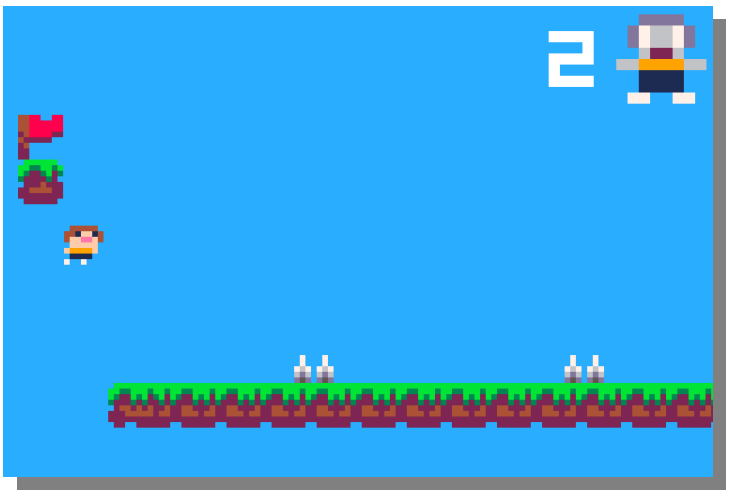
```gdscript
func _ready():
    ui_onscreen = coin_ui.rect_position.y
    ui_offscreen = -coin_count.rect_size.y
```

To replicate this functionality for the death count, all I had to do was duplicate the CoinUI node, rename appropriately, change the sprite, and copy the code. This worked, but it made the code hard to read, so I modified it to use a function instead.

Initially, this was not working – the UI would stay on-screen and not go away; however, I figured out that this was because I was trying to update the previous variable. To rectify this, I simply returned the previous variable, and updated the variable there.

```
22 func update_count(count_node, count_var, ui_node, timer_node, previous):
23     count_node.text = str(count_var)
24 
25     if previous != count_var:
26         timer_node.start()
27 
28     if !timer_node.is_stopped():
29         previous = count_var
30 
31         ui_node.rect_position.y = lerp(ui_node.rect_position.y, ui_onscreen, 0.1)
32     else:
33         ui_node.rect_position.y = lerp(ui_node.rect_position.y, ui_offscreen, 0.1)
34 
35     return previous
36 
37 func _process(_delta):
38 
39     previous_coin = update_count(coin_count, GlobalVariables.coin_count, coin_ui, coin_timer, previous_coin)
40 
41     previous_death = update_count(death_count, GlobalVariables.death_count, death_ui, death_timer, previous_death)
42 
```



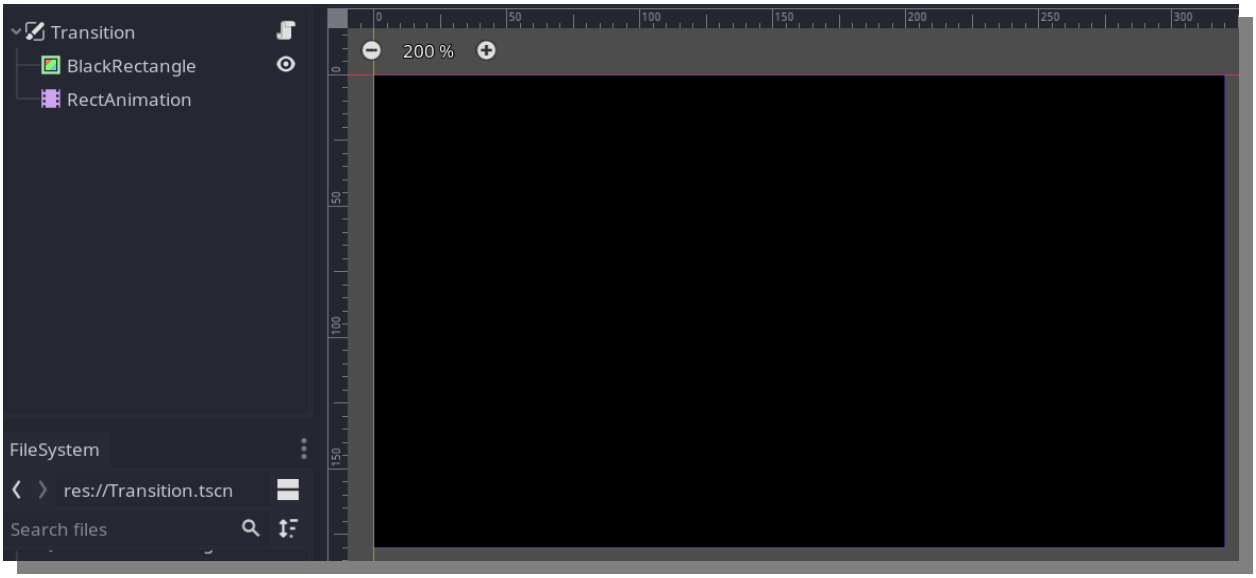


Adding Level and Scene Transitions

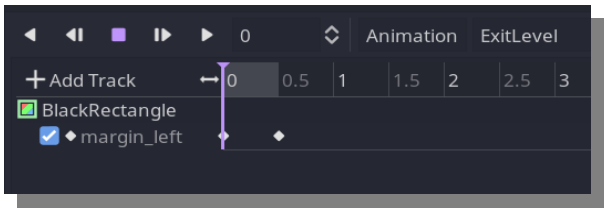
Since important GUI elements were now added, I decided to focus on more aesthetic overlays – specifically transitions between levels, scenes (e.g., the menu), and also between deaths (as currently the player would just teleport).

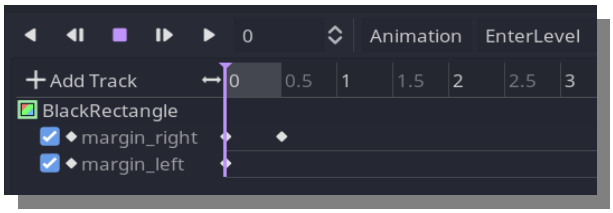
To achieve this, I created a new scene that could be instantiated from anywhere, which would subsequently run a script. This script would draw a rectangle to cover the screen, then uncover it to reveal the new level (or camera placement in the level).

I added two child nodes to the parent: a “ColorRect” node (to be the black rectangle), and an “AnimationPlayer” node (to control the size of the rectangle).



Inside the animation editor, I set-up two animations: one for entering the level (i.e., the rectangle covering the screen), and one for exiting the level (i.e., the rectangle revealing the screen). This will provide me more control if necessary, such as if the transition needs to last longer than one frame.





Then I created two procedures which can be called from anywhere when the transition is needed:

The next step was to use these transition procedures to allow me to:

```
5  func exit_level_transition():
6      rect_animation.play("ExitLevel")
7
8  func enter_level_transition():
9      rect_animation.play("EnterLevel")
10
```

1. Start the transition
2. Wait until the rectangle covers the screen
3. Run some code (usually just to change the scene)
4. End the transition

Luckily, in Godot, this can be done using yield(). To set this up, I created a unique signal (i.e., something the node sends out when specified – some come automatically with nodes, but custom ones can be created for this purpose).

This signal will be sent out when either animation is complete (although all we need is the beginning, "covering" animation).

```
5  signal transition_completed
6
7  func exit_level_transition(): ...
9
10 func enter_level_transition(): ...
12
13
14 func _on_RectAnimation_animation_finished(anim_name):
15     emit_signal("transition_completed")
16
```

To allow the transition node to be accessed from anywhere in the game, a singleton is used once more. Godot allows for scenes to be "autoloaded", making them globally accessible throughout the game. Once the transition scene is added to the autoload list, it can be easily accessed in code.

Whenever a transition is required to be sent out, the node first needs to call the procedure to begin the first animation, then yield. This waits until the specified scene outputs a signal until continuing the function it is run inside. Crucially, this does not mean the node does not run any code, just that inside the function. Once the signal is received, the rest of the code can be run, which happens whilst the screen is covered. The player's death procedure is an example.

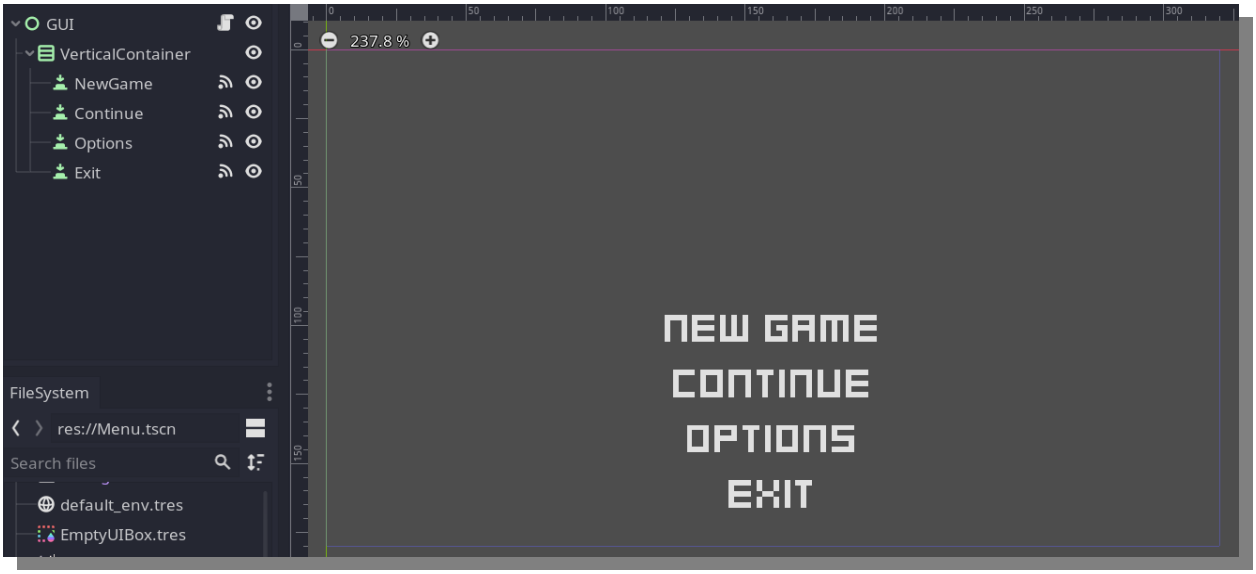
```
138 func die():
139     Transition.exit_level_transition()
140
141     # Stops the current function until the signal
142     # is emitted from the specified scene
143     # yeild(Scene, signal)
144     yield(Transition, "transition_completed")
145
146     position = GlobalVariables.checkpoint_pos
147     GlobalVariables.death_count += 1
148
149     reset_rope()
150
151     Transition.enter_level_transition()
```

Adding a Menu

Now that transitions were added, I decided to add a menu, in which the transitions could be used between it and the levels of the game. I planned on only having 4 options for the menu: "New Game", "Continue", "Options", and "Exit". Implementing a continue and options function would require saving and loading, so I ensured that the system would work with these, even though this was not currently set up.

I created a new scene with the parent node “GUI”, and added a “VBoxContainer” node to the bottom half of the screen.

This node automatically stacks child “Button” (or other user-interface) nodes vertically, and allows them to be selected with the mouse or keyboard whilst highlighting the selected node.



Once this was added, each button was given its own unique signal. This could then be used inside the menu script. When GUI node had loaded, the “New Game” node had to be selected (i.e., a default selection). Each signal was assigned its own function, which ran code when the enter key was pressed, or when the user left-clicked the button.

```
1  extends Control
2
3  func _ready():
4      $VerticalContainer/NewGame.grab_focus()
5
6
7  func _on_NewGame_pressed():
8      Transition.exit_level_transition()
9      yield(Transition, "transition_completed")
10
11     get_tree().change_scene("res://DebugRoom.tscn")
12
13     Transition.enter_level_transition()
14
15 func _on_Continue_pressed():
16     pass # Replace with function body.
17
18 func _on_Options_pressed():
19     pass # Replace with function body.
20
21 func _on_Exit_pressed():
22     get_tree().quit()
23
```

The continue and options buttons were left empty, as to accommodate when said functionality was added. Finally, the “New Game”, and “Exit” button’s functionality was added, to either use the transition system to change scenes, or to exit the game, respectively.

Then some minor aesthetic nodes were added, such as a tileset and background sprite.

The space left over can be used for a logo.

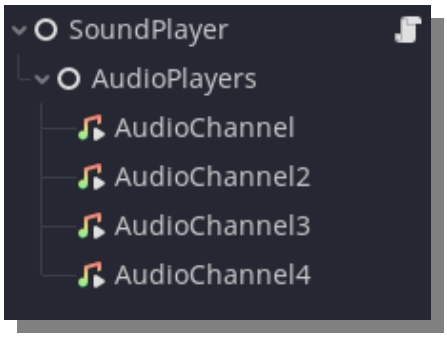
Adding Sounds

Now that the menu was complete, there was one last “in-level” task that needed to be sorted: sounds. In Godot, sounds are played through “AudioStreamPlayer” nodes. This means that sounds can be easily played by simply adding one of these nodes to the active scene.

However, I needed a slightly more complex system, as there was not just going to be a single sound playing at one time. Moreover, I didn’t want one node for every single possible sound, as that would be redundant, take up much more memory than necessary, and mean playing a specific sound twice simultaneously would be impossible. To solve this problem, I created a sound system that utilises “AudioStreamPlayer” nodes as sound channels, meaning up to 4 sounds can be played simultaneously, including the same specific sound.

To do this, I first created a new scene called “SoundPlayer”. The root node did not have to have any specific functionality, but rather just be able to run a script, thus all I needed was a default “Node” node. For organisation, I added child node called “AudioPlayers”, which will contain the 4 “AudioStreamPlayer” nodes.

This is useful because the code goes through all children of the “AudioPlayers” node, meaning if I wanted to add a non-audio related node (for whatever reason), it must be outside the “AudioPlayers” node.



Now that the node tree was set up, I needed to generate some sounds. I organised a list of sounds I needed, and then used “jfxr”, an online sound generator inspired by bfxr (which was inspired by sfxr). The sounds that I required were: collecting a checkpoint; collecting a coin; taking damage; using the grapple; jumping off the ground; soaring in the air; landing on the ground; and interacting with the menu.

Now that my sounds were generated, I attached a script to the root node. The first thing that I needed to do was preload all the sounds. This allows them to be played in the “AudioStreamPlayer” nodes whenever necessary. Without preloading, the sounds would not be played at the correct time.

```
3  const Checkpoint = preload("res://Sounds/Checkpoint.wav")
4  const Coin = preload("res://Sounds/Coin.wav")
5  const Damage = preload("res://Sounds/Damage.wav")
6  const Grapple = preload("res://Sounds/Grapple.wav")
7  const Jump = preload("res://Sounds/Jump.wav")
8  const Menu = preload("res://Sounds/Menu.wav")
9  #const Swing = preload("res://Sounds/Swing.wav")
10 #const Swing = preload("res://Sounds/White Noise.wav")
11 #const Swing = preload("res://Sounds/Pink Noise.wav")
12 const Swing = preload("res://Sounds/Brown Noise.wav")
13 #const Swing = preload("res://Sounds/Whiter Noise.wav")
14 const Land = preload("res://Sounds/Land.wav")
```

So, how was the system going to work? Whenever a function was called, which takes in an argument for what sound to play. It then loops through all the “AudioStreamPlayer” nodes, and finds if one is available (i.e., not playing a sound). If there is one, then it changes the available channel’s audio to be the sound specified in the function, and plays it. Once the sound has finished playing, the channel will be become free again.

```
18 func play_sound(sound):
19     # Loops through each audio channel
20     for channel in audio_player.get_children():
21         # Plays the specified sound in the
22         # First available audio channel
23         if !channel.playing:
24             channel.stream = sound
25             channel.play()
26             break
```

However, it’s also important to be able to know if a specific sound is playing (e.g., we only want a sound to play once if a player has collided with a spike). Moreover, it is useful to be able to stop a sound mid-playing. This can use the function that detects whether a sound is playing.

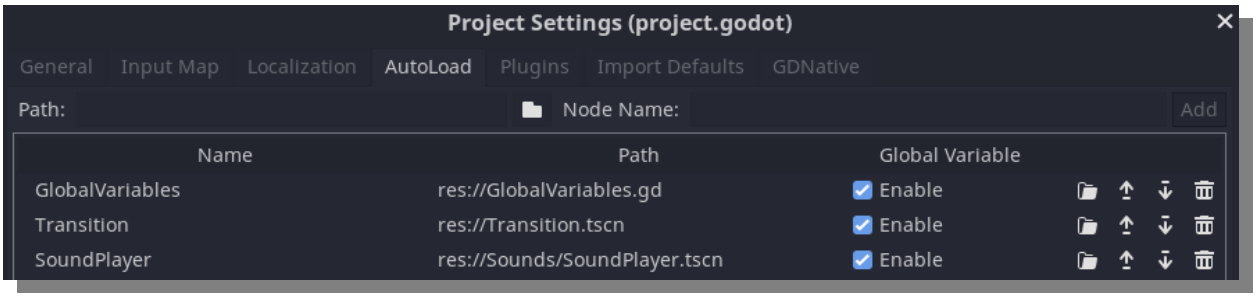
```
28 func is_playing(sound):
29     # Loops through each audio channel
30     for channel in audio_player.get_children():
31         if channel.playing && channel.stream == sound:
32             return channel
33
34     return null
```

```
36 func stop_sound(sound):
37     var channel = is_playing(sound)
38
39     if channel:
40         channel.stop()
41         return true
42
43     return false
```

The above code loops through all audio channels to check if the specified sound is playing. If it is, it will return the channel it is playing in, otherwise it returns null.

This can then be used in the second function, which will stop the sound, but only if it is already playing – ensuring that there will be no errors using the function.

Finally, it is important that these functions can be called from anywhere in the game. To do so, I added the scene to the autoload list, meaning it can, like the transitions, be accessed from anywhere via code. Below is an example of the three functions in action.




```
148 func die():
149     animation.play("Die")
150
151     SoundPlayer.stop_sound(SoundPlayer.Swing)
152     SoundPlayer.play_sound(SoundPlayer.Damage)
```

Adding a Wind Noise

One of the many important subtleties of game design is small, immersive, in-game sounds. Now that the sound system was fully set-up, I decided create a more dynamic swing sound, i.e., it changes pitch and loudness depending on the player's speed.

This is where the is_playing() function came in handy; I used it to check that the swing sound is playing, and then modified said sound, regardless of which channel it is playing on. All I needed to do was decide on a formula for the sound's pitch. This was essentially a bunch of trial and error, and changed multiple times. One of the first I settled on was the pitch scale (e.g., 1x is the original sound and 2x is twice the sound's frequency) being equal to $(\frac{v}{100})^{1.01}$, where v is the player's velocity. I put this at the end of the player's _process() function, meaning it runs after all the other code, every frame – this ensures that the sound updates straight away, instead of a frame late.

```
269     # Swing sound
270     var sound_channel = SoundPlayer.is_playing(SoundPlayer.Swing)
271     if sound_channel && SoundPlayer.get_channel_pitch_scale(sound_channel):
272         SoundPlayer.set_channel_pitch_scale(sound_channel, (pow(motion.length() / 100, 1.01)))
```

However, I later realised that I wanted the sound to also play while the player was falling – regardless if the player had just swung on the rope, or rather just jumped off a platform. This required some slight modifications to the functionality.

To do this, I decided that the swing sound should always be playing when the player is not in the air. Luckily, this was easy to do as I already had an if statement testing for such.

```
270     # If the player's new position is not on the floor, but it was the previous frame
271     if floor_before_move && !is_on_floor():
272         var channel = SoundPlayer.play_sound(SoundPlayer.Wind)
273         SoundPlayer.set_channel_vol(channel, -100)
274         coyote_timer.start()
```

It was important to change the volume down to -100dB, as this effectively mutes it, so it can then be gradually increased as the player's velocity does. Otherwise, it would play a loud sound for one frame, and then suddenly change to the correct volume.

Volume is logarithmic in base 10, so originally I created a function to allow for such, as in Godot, only natural logarithms are natively implemented.

```
54  func log_base(value, base):
55      return log(value) / log(base)
```

However, after modifying the wind sound, I realised that ignoring how decibels are logarithmic made the wind sound more natural, thus I used a formula more akin to the pitch scale. I also slightly modified the pitch scale formula to lower the pitch slightly.

```
166 func wind_noise():
167     # Gets the channel in which the wind sound is playing
168     var sound_channel = SoundPlayer.is_playing(SoundPlayer.Wind)
169     var vel = motion.length()
170     if sound_channel && SoundPlayer.get_channel_pitch_scale(sound_channel) && vel > 40:
171         SoundPlayer.set_channel_pitch_scale(sound_channel, 0.6 * pow(vel / 100, 1.01))
172         SoundPlayer.set_channel_vol(sound_channel, -100 / pow(4, vel / 100))
```

So, now the formulae are as follows:

$$p = 0.6\left(\frac{v}{100}\right)^{1.01} \qquad d = \frac{-100}{(4)^{\frac{v}{100}}}$$

These values came from about 30 mins of trial and error – slowly modifying values until it sounded correct. Occasionally, I used the graphing software, Desmos, to visualise the current formula, and what I wanted to change.

However, I kept running into random crashes, and I couldn't quite tell why. Luckily, I realised that the custom procedures I was using to set the pitch and volume were causing the issue. This is because I had neglected to ensure that the sound was playing, and, even though I was, in fact, checking before I ran the functions, if the sound ended on the *exact* frame, the game would crash.

To rectify this was simple, however:

```
52 func get_channel_pitch_scale(channel):
53     if channel.is_playing():
54         return channel.get_pitch_scale()
55
56 func set_channel_pitch_scale(channel, value):
57     if channel.is_playing():
58         channel.set_pitch_scale(value)
59
60 func get_channel_vol(channel):
61     if channel.is_playing():
62         return channel.get_volume_db()
63
64 func set_channel_vol(channel, value):
65     if channel.is_playing():
66         channel.set_volume_db(value)
```

Interview of Features 4

Are the following features well-suited for the game: the spikes, checkpoints, coins?

"Absolutely! These features go a long way into adding more depth into the game, making it feel much more like a fleshed-out platformer game, rather than a 'sandbox' game with no end goal or clear challenge. The spikes provide this clear challenge, while the checkpoints clearly demonstrate a goal or target for the player. The coins are a nice touch for players to compete with each other"

Are there any more "niche" features that need adding, similar to coyote-time and the jump buffer?

"Coyote-time and the jump buffer were the most important, less well-known features that were needed for a good platformer game. Now that these are added, I think it's more important to focus on features that match the game's main gimmick of rope swinging, such as adding tiles that the player cannot grapple onto, which adds a further sense of challenge."

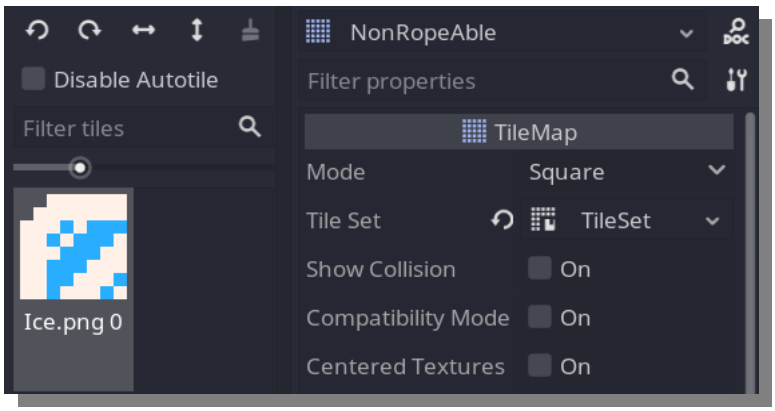
Are there any features that would make the game stand out from other platformer games?

"I think the main gimmick is enough to let the game stand out on its own, however, having a more unique style for the sprites and overall aesthetic may improve the overall individuality of the game – but this is only important once the other features I mentioned are fully set-up, and the game is fully functional."

Adding "Non-Ropeable" Tiles

Now that majority of the features were developed, I decided to tackle one of the "additional successes" as detailed in the analysis sections: tiles that the player would collide with, but cannot grapple on to. This is also what was requested in the previous interview.

To do this, I first created a new TileMap node, and inside it, created another tileset. Instead of grass, this time it would be ice, for two reasons: there is slight logic that it would be harder to grapple on to ice, and it is of a clear contrast (colour and texture wise) to the grass.

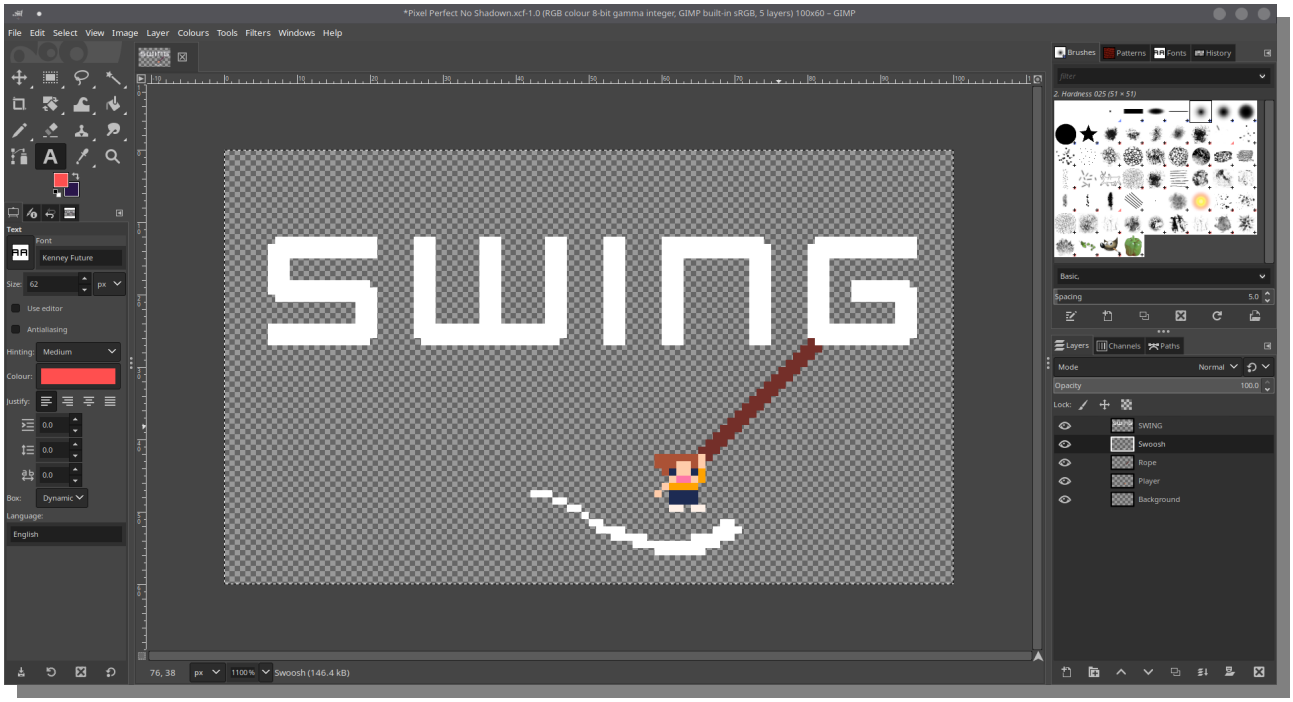


From here, all I needed to do was to modify the raycast code to get both tileset nodes (stored in a variable), and then, via a switch statement (called match in Godot), run specific code – either get the collision, or do nothing (for now).

```
17      if is_colliding():
18          var tile = get_collider()
19          if tile is TileMap:
20              # switch statement depending on the tile type
21              match tile:
22                  rope_able:
23                      point = get_collision_point()
24
25                  non_rope_able:
26                      pass;
```

Designing a Logo

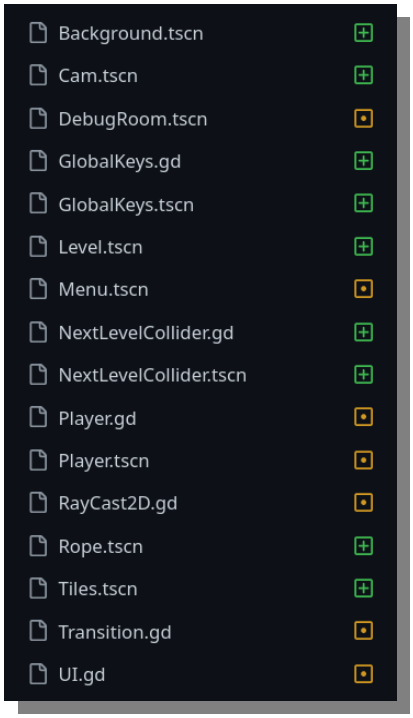
For a bit of change of pace, I decided to quickly whip up a logo for the game (seen at the beginning of this document). To do this, I used GIMP once more, and Kenney's pixel fonts (https://kenney.nl). I did this with a small canvas size of 100px by 60px – to keep the pixel art aesthetic – and then scaled it up by 20 times (this version can be used outside of the game without bilinear filtering affecting the look).

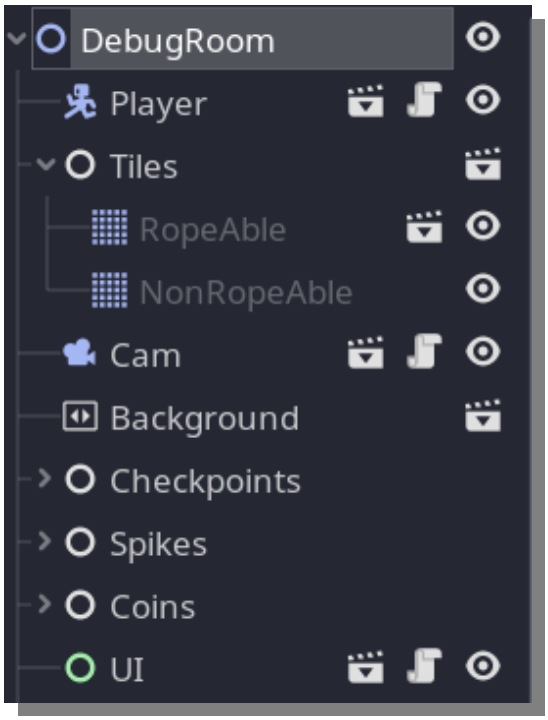


Setting Up Level Inheritance

An important part of any platformer game is multiple levels, so it was important to make each node in the level into a separate scene, which allows me to re-use them multiple times. This can be done easily in Godot by right-clicking each node, and selecting “Save Branch as Scene”.

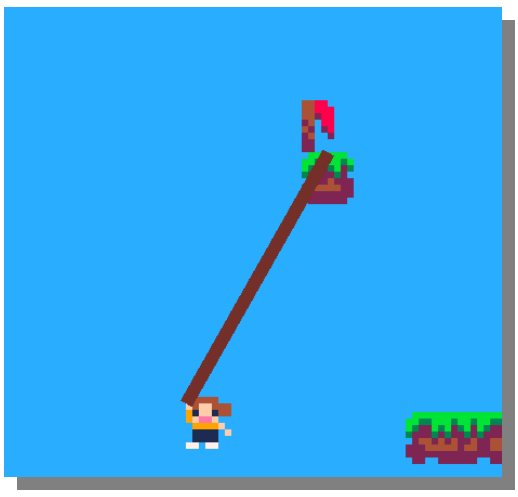
Below are all the new scenes I added, as well as the node tree.





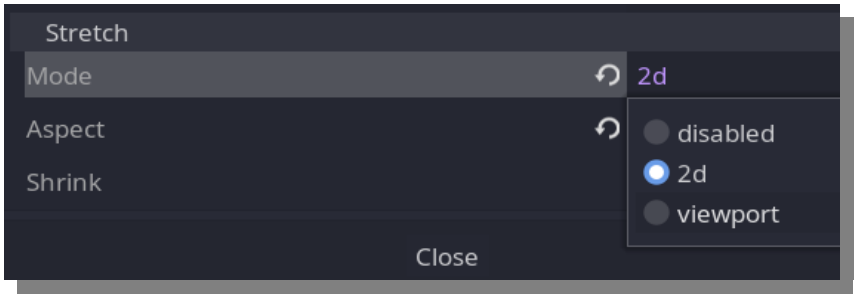
Pixel-Perfect Viewports

Another one of the “additional successes” is pixel-perfect scaling. As detailed before, this allows for a much more succinct style. Below is a comparison of the before and after:



This also allowed for the UI to display higher-resolution text, which can be useful for accessibility, as some fonts may be unsuitable for people with disabilities such as dyslexia.

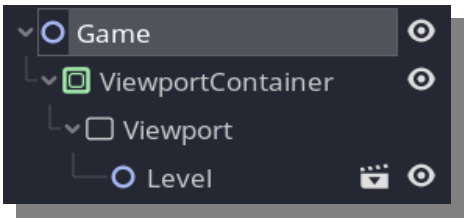
My first attempt at this was simple, but it did not provide me with enough control. In Godot, it is trivial to change the scaling mode to "Viewport" inside project settings – exactly what I want to do! However, this also meant that the camera snapped to pixels in a 320 × 180 resolution, instead of 1920 × 1080, meaning the whole experience felt jittery and disorienting (although this cannot be conveyed via screenshots).



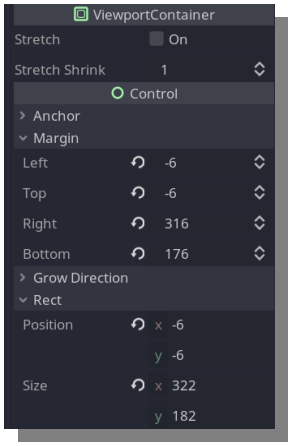
So, what can I do instead? Well, Godot allows for manual control of viewports, which is essentially a surface, or screen, onto which the game is drawn on to.

By default, there is a root viewport node that is created when the game starts, which is scaled depending on said project setting. However, since my game does not have a static camera, but rather needs to move smoothly using sub-pixels, I need to be able to take manual control of a viewport.

To go about doing so, I created another scene called "Game", which is to contain 3 nodes: a "ViewportContainer" node (which allows for the display of a "Viewport" node), said "Viewport" node, and a node representing the level. Essentially, the level scene is embedded within the game scene, inside the viewport.



I set-up the viewport to the resolution of 322 × 182, instead of 320 × 180 – this was because the camera needed to be able to be in-between pixels on all sides of the screen, i.e., it had to be able to be up to 1 pixel ahead to the left, right, top, or bottom of the actual, in-game view.



In addition, the viewport had to be scaled up by 6 times – from the "resolution" of 320 × 180 to 1920 × 1080. This was, in fact, one of the initial reasons I chose 320 × 180 – it is a resolution often used for pixel art games, as it perfectly scales up with common HD resolutions: 720p, 1080p, 1440p, and 4K.

Whilst I am only going to support 1080p, since it is the most common, and it is easier to only support one resolution, it should not be much more work to add support for all 4 of the aforementioned resolutions in the future.

Due to the viewport resolution being 322 × 182, I needed to offset it by 1 pixel to centre it – however, now that it is scaled up 6×, this offset, in actuality, becomes 6.

With this setup, I now have manual control over the viewport in-game, as I can directly access the methods and attributes of the viewport node. This will be useful when calculating the camera's new position.

After researching how to create a smooth scrolling camera, I found out that the easiest way to do so was by keeping the camera inside the level scene, and simply adding a shader to it. Shaders allow for much closer control over the GPU. They are usually used for 3D rendering, although can be used here to control how pixels are drawn onto the screen – exactly what I needed.

I have little experience writing shaders, so I used a simple one detailed online. This shader will simply allow me to move the "ViewportContainer" node by sub-pixels (although since the "ViewportContainer" is a parent of the viewport, it can display these sub-pixels as regular pixels for a 1080p resolution. This means that the shader needs only one parameter, being a 2D vector of the offset.

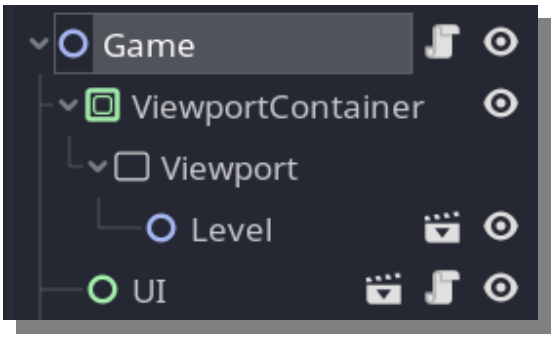
```
1  shader_type canvas_item;
2
3  uniform vec2 camera_offset = vec2(0.0, 0.0);
4
5  void vertex()
6  {
7      VERTEX += camera_offset;
8  }
```

So, how does this help stop the camera from snapping to a 320 × 180 resolution? It doesn't. Instead, the "actual" position (i.e., in sub-pixels) the camera should be is stored in a variable. The fractional / decimal part of this variable is what the shader parameter is set to – this means that the viewport will become offset by a few pixels in 1080p space. When the camera is set to its non-fractional position, snapping to the 320 × 180 grid, the viewport offset means that it actually is displaying those sub-pixels, scaled up into 1080p space.

Implementing this into code is actually fairly trivial: the hardest part is definitely figuring out how to allow the viewport to move correctly with the camera.

```
29      var pos = viewport.get_mouse_position() / window_scale - (game_size / 2) + player.global_position
30
31      actual_cam_pos = lerp(actual_cam_pos, pos, interpolate_val * delta)
32
33      var subpixel_pos = actual_cam_pos.round() - actual_cam_pos
34
35      viewpoint_container.material.set_shader_param("camera_offset", subpixel_pos)
36
37      global_position = actual_cam_pos.round()
```

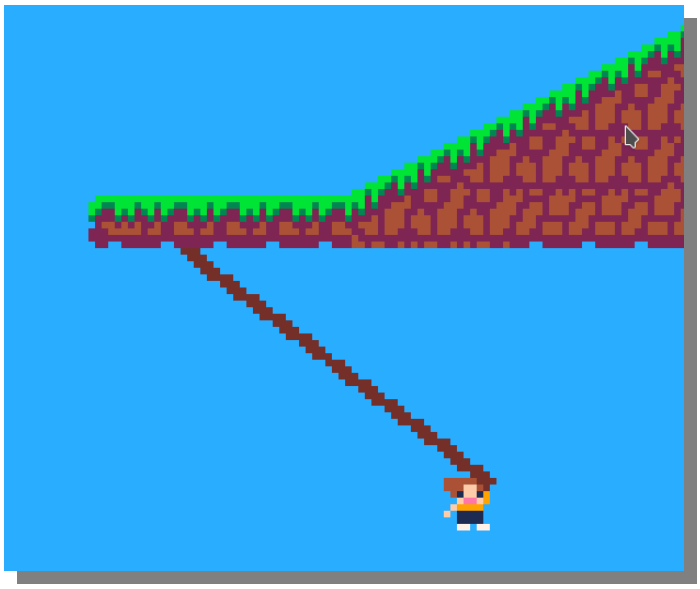
Once this was done, I moved the UI elements outside of the level scene, as this allows them to smoothly move down and back up, without snapping to the 320 × 180 grid too. Overall, this change makes the game feel much more polish and finalised, although it did come with some hurdles to overcome...



Mouse Issues with Viewport Scaling

Scaling up the viewport meant that the game was no longer reading mouse inputs correctly, but rather it seemed like it would be (seemingly randomly) offset. This offset didn't even stay constant once the mouse cursor was moved around, either, which made this much more difficult to solve. These mouse issues were easily the most difficult, headache inducing, horrendously unruly bugs in the entire project, and I am certain that years have been lost off my life because of it. The fixing of such detrimental difficulties is a tall tale, demonstrating treacherous trials and tribulations, terrible treason upon the heart, tears, and most importantly: achievement. A wretched week of wondrous work, and war: a struggle of the century. So, let's begin.

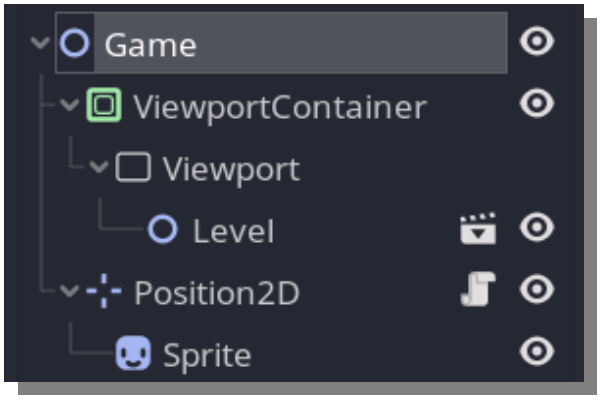
The story starts with relentless refrainment of rotten Reddit, abstinence from seeking the help of others. I could not for the life of me figure out why the mouse was offset, and it began to take a toll on my mental health.

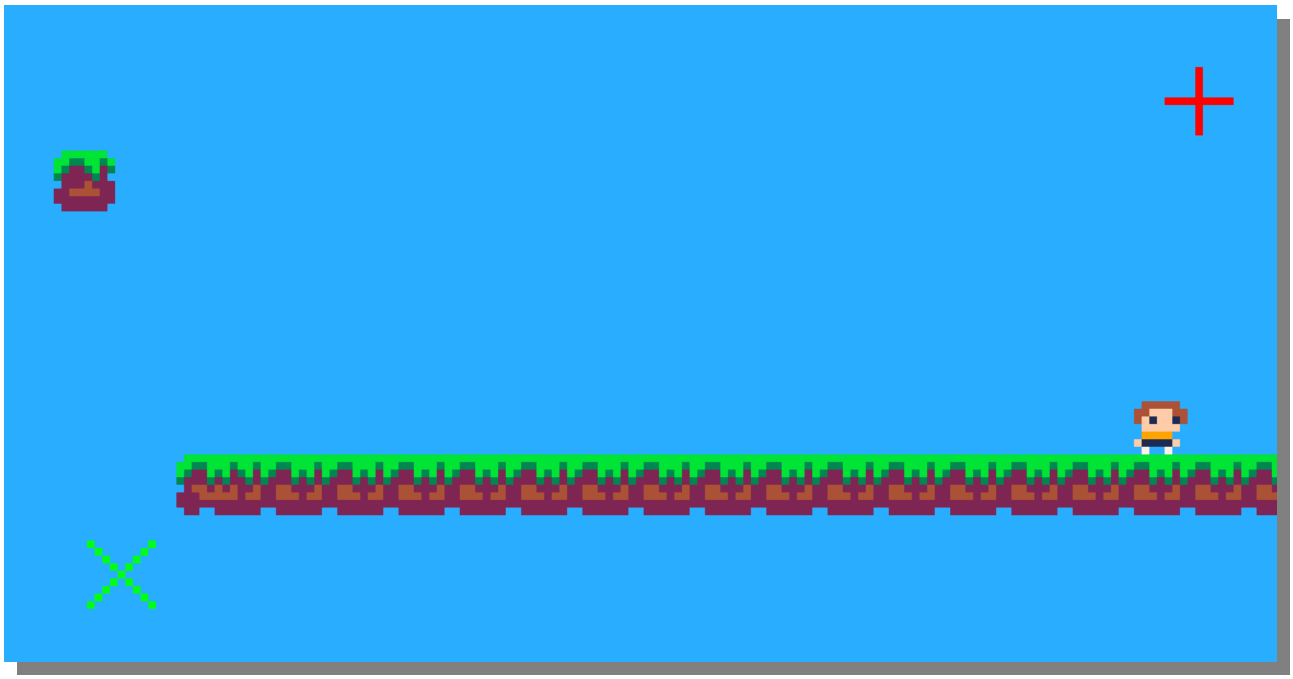


As demonstrated on the left, the mouse cursor's position seemed irrelevant to where the raycast was colliding – something was very wrong.

To aid my debugging, I added a sprite to outside the viewport, allowing me to keep track of where the mouse cursor should be. Additionally, I added a different sprite to where the raycast thought the mouse position was.

One of my first attempts to solve this was to pass the co-ordinates from outside the viewport, although dividing the x and y value by 6 (due to the scale). To do this, I created a function inside the raycast to update the target position, and then accessed this function from the Position2D node (which is how I was displaying the sprite for the correct mouse position). Moreover, I temporarily disabled the camera, as that moved with the mouse, making debugging more difficult.





The green cross represents where the mouse cursor's position *should* be. Whereas the red cross represents where the game thinks the mouse is[7].

Another one of my attempts was to invert the transformation done by the viewport. This can be done using an inverse transformation matrix, which can be easily calculated in Godot using the built-in affine_inverse() function. I attached a script to the viewport container node, as this has a function that allows for the viewport's transformation matrix to be returned, calculated the inverted position, and passed it through to the raycast.

```
1  extends ViewportContainer
2
3  onready var raycast = $Viewport/Level/RopeCast
4  onready var viewport = $Viewport
5
6  func _process(_delta):
7      var local_to_viewport = get_viewport_transform() * get_global_transform()
8      var viewport_to_local = local_to_viewport.affine_inverse()
9
10     var mouse_position_viewport = get_local_mouse_position()
11     var mouse_position_local = viewport_to_local * mouse_position_viewport
12
13     raycast.update_pos(mouse_position_local * 6)
```

7 The presence of a red cross in a video game actually violates the Geneva conventions. "The red cross and red crescent emblems are protected symbols under international humanitarian law and national laws. Any use that is not expressly authorized by the Geneva Conventions and their Additional Protocols constitutes a misuse of the emblem. Use of these emblems by unauthorized persons is strictly forbidden." See https://www.icrc.org/en/copyright-and-terms-use.

This, unfortunately, gave the same result; clearly inverting the matrix is simply just scaling it down, too. Obviously, something else had to be done.

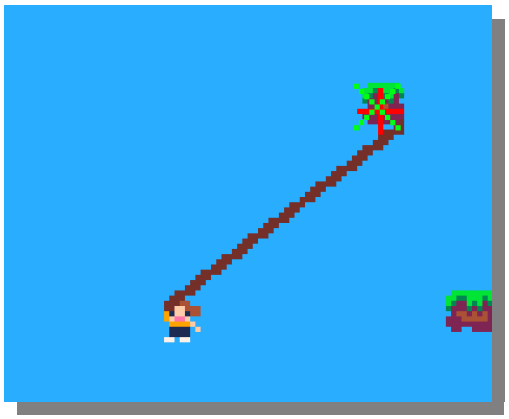
Some of my other attempts involved getting the global mouse position from the viewport container, subtracting the viewport position, and using that as the mouse position, but this ended up not changing anything. Additionally, I tried treating the 2D vector as separate x and y positions, in case dividing a vector didn't work properly (even though this is not the case elsewhere in the game). This, again, did not help.

```
#   Attempt
    var global_pos = get_global_mouse_position()
    var viewport_pos = viewport.global_position
    var local_pos = global_pos - viewport_pos
    raycast.update_pos(local_pos)

#   Attempt 2
    var pos  = get_local_mouse_position()
    raycast.update_pos(pos)

#   Attempt 3
    var mouse_pos = get_viewport().get_mouse_position()
    mouse_pos.x = mouse_pos.x / 6
    mouse_pos.y = mouse_pos.y / 6

    raycast.update_pos(mouse_pos / 6)
```

I was very quickly running out of ideas. Perhaps inverting the matrix had to happen inside the level node?

As you can probably guess, it made absolutely no difference.

This was truly a fantastical fight, causing fear and frightful feelings to my, now, battered, broken brain. At this point, I was genuinely up past midnight, promising myself – pretending that I would persist and prevail against all the purported probability of my failure against this nonsensical noggin-bending bug.

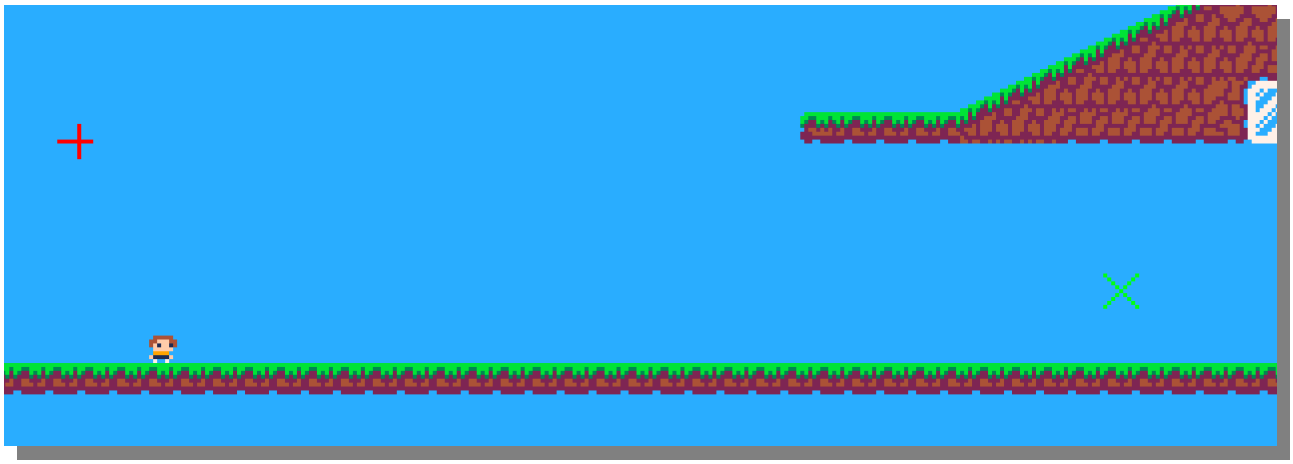


"I've absolutely had enough," I screamed to myself; it was time to end the eternal excruciation encountered by myself. I didn't want to, but I decided to add some magic numbers[8] and just add an arbitrary offset to the mouse position – surely *this* ought to work.

8 In programming, magic numbers are unique, often arbitrary, unexplained values which should be replaced with a constant.

Well yes, it *did* work. Except it didn't, because my misfortune could not end so soon. Once I re-enabled the camera, hell broke loose once more, and my reluctance to an arbitrary value was here once-more.

```
27      var val = viewport.get_mouse_position()
28      var offset = Vector2(850, -370) #+ cam.position
29      val -= offset
30      print(val)
31      sprite.global_position = val / 6
32      set_cast_to(to_local(val / 6))
```



In fact, this was obviously so much worse. Unfortunately, I realised that my radical Reddit refusal must be, reluctantly, retired. And in a frenzy, I recorded a video of my problems, and sent it off to the aether that is r/godot.

5

Posted by u/PrimedPixel **Programusicigner** 18 days ago

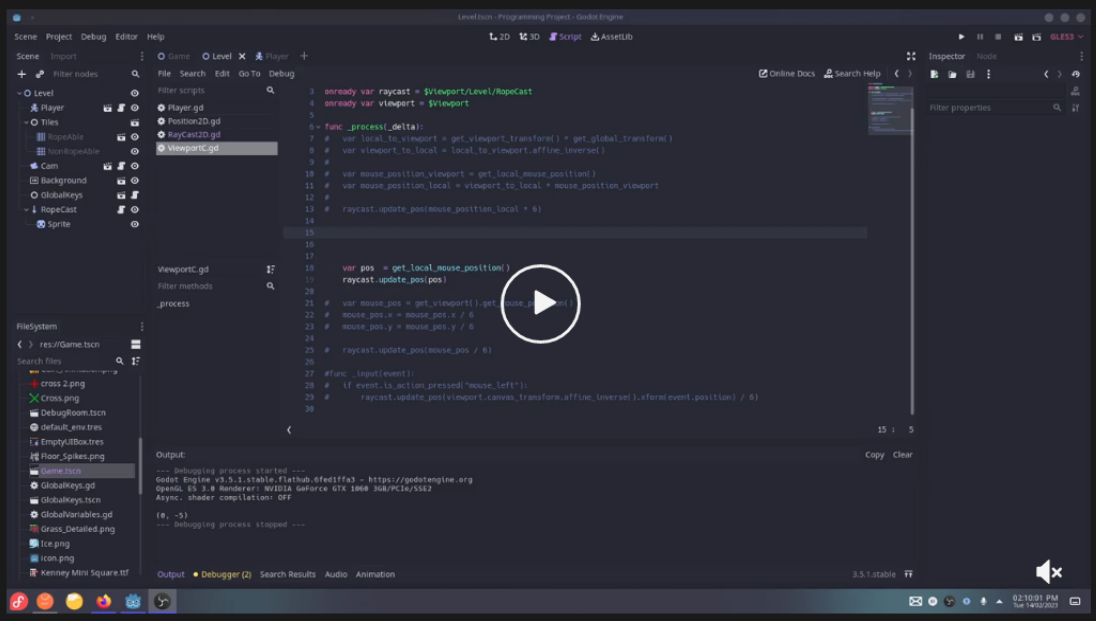
Mouse Offset Due to Viewport

Help

Hiya all,I can't seem to get the correct mouse position ever since I added a viewport (scaled 6x).

I need the mouse position to allow me to update a raycast's "cast_to", yet whatever I've tried has either led it to being completely off, or instead at a seemingly arbitrary offset (see video).

The red cross represents where the mouse position is being calculated (adjusted for the raycast), whereas the green cross represents a position node with a sprite - outside of the viewport:



Code to update the raycast (inside the Viewport Container and has changed a lot while testing, but I got the same result as far as I can tell):

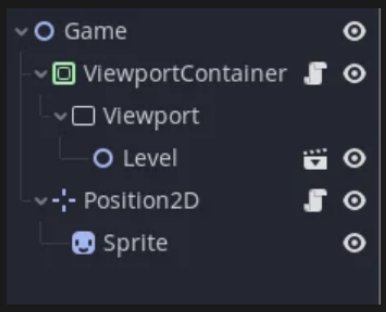
```
extends ViewportContainer

onready var raycast = $Viewport/Level/RopeCast
onready var viewport = $Viewport

func _process(_delta):
	var pos  = get_local_mouse_position()
	raycast.update_pos(pos)
```

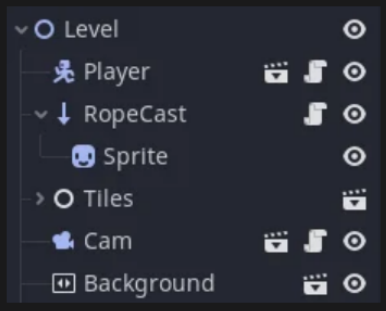
Code for the Raycast "update_pos()" function

```
func update_pos(pos):
	sprite.global_position = pos

	set_cast_to(to_local(sprite.global_position))
```

(I'm aware I could just do `set_cast_to(sprite.position)` or an alternative, I've just been testing everything)

Node Trees



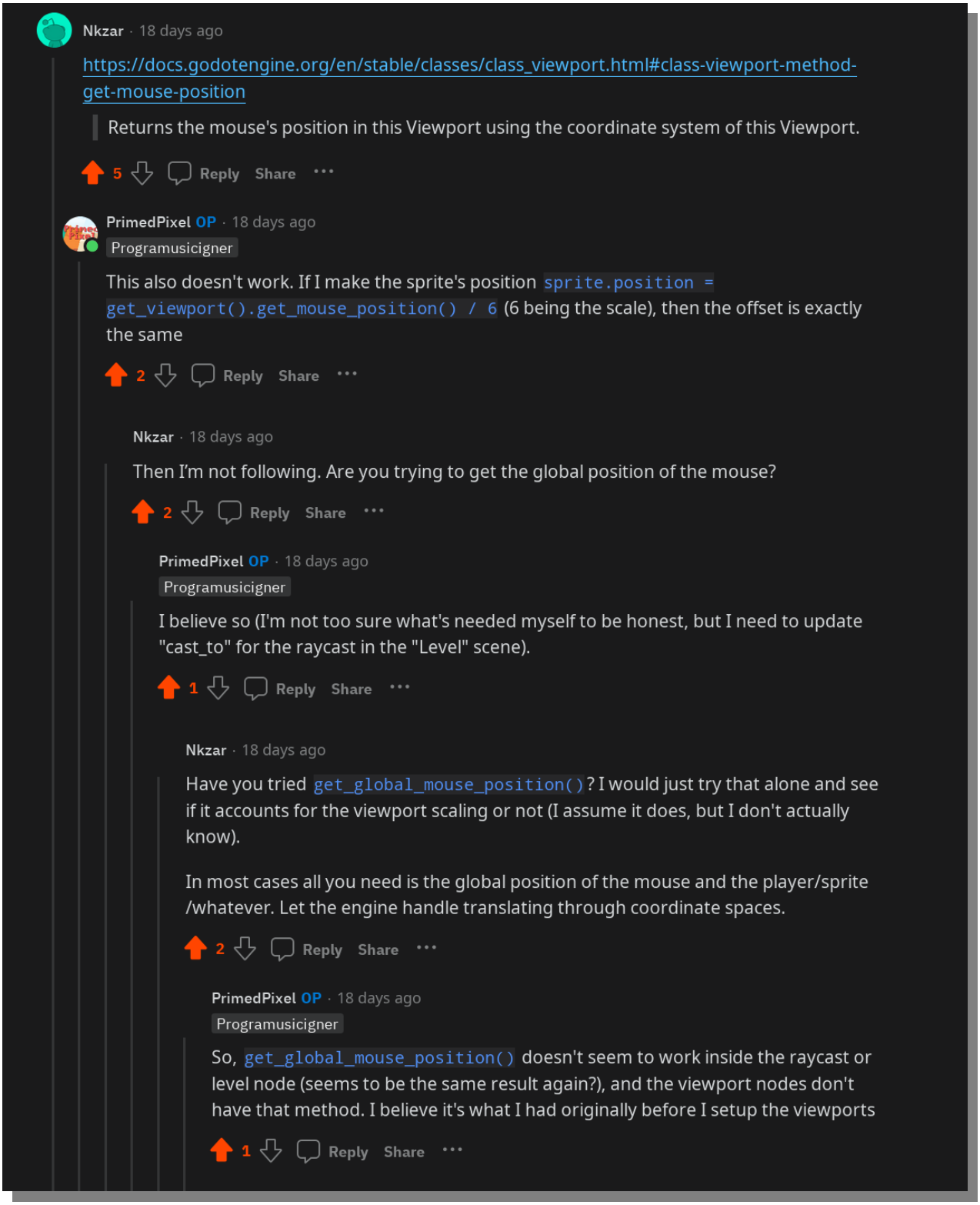


I've (temporarily) disabled the camera script, as it follows the mouse, making debugging much more difficult

I believe that's everything, but lmk if there's any more info needed,

Thanks in advance

And what gleeful comments did I receive? And were they helpful at all?



Nkzar · 18 days ago

https://docs.godotengine.org/en/stable/classes/class_viewport.html#class-viewport-method-get-mouse-position

> Returns the mouse's position in this Viewport using the coordinate system of this Viewport.

5 Reply Share ···

PrimedPixel OP · 18 days ago

Programusicigner

This also doesn't work. If I make the sprite's position `sprite.position = get_viewport().get_mouse_position() / 6` (6 being the scale), then the offset is exactly the same

2 Reply Share ···

Nkzar · 18 days ago

Then I'm not following. Are you trying to get the global position of the mouse?

2 Reply Share ···

PrimedPixel OP · 18 days ago

Programusicigner

I believe so (I'm not too sure what's needed myself to be honest, but I need to update "cast_to" for the raycast in the "Level" scene).

1 Reply Share ···

Nkzar · 18 days ago

Have you tried `get_global_mouse_position()`? I would just try that alone and see if it accounts for the viewport scaling or not (I assume it does, but I don't actually know).

In most cases all you need is the global position of the mouse and the player/sprite /whatever. Let the engine handle translating through coordinate spaces.

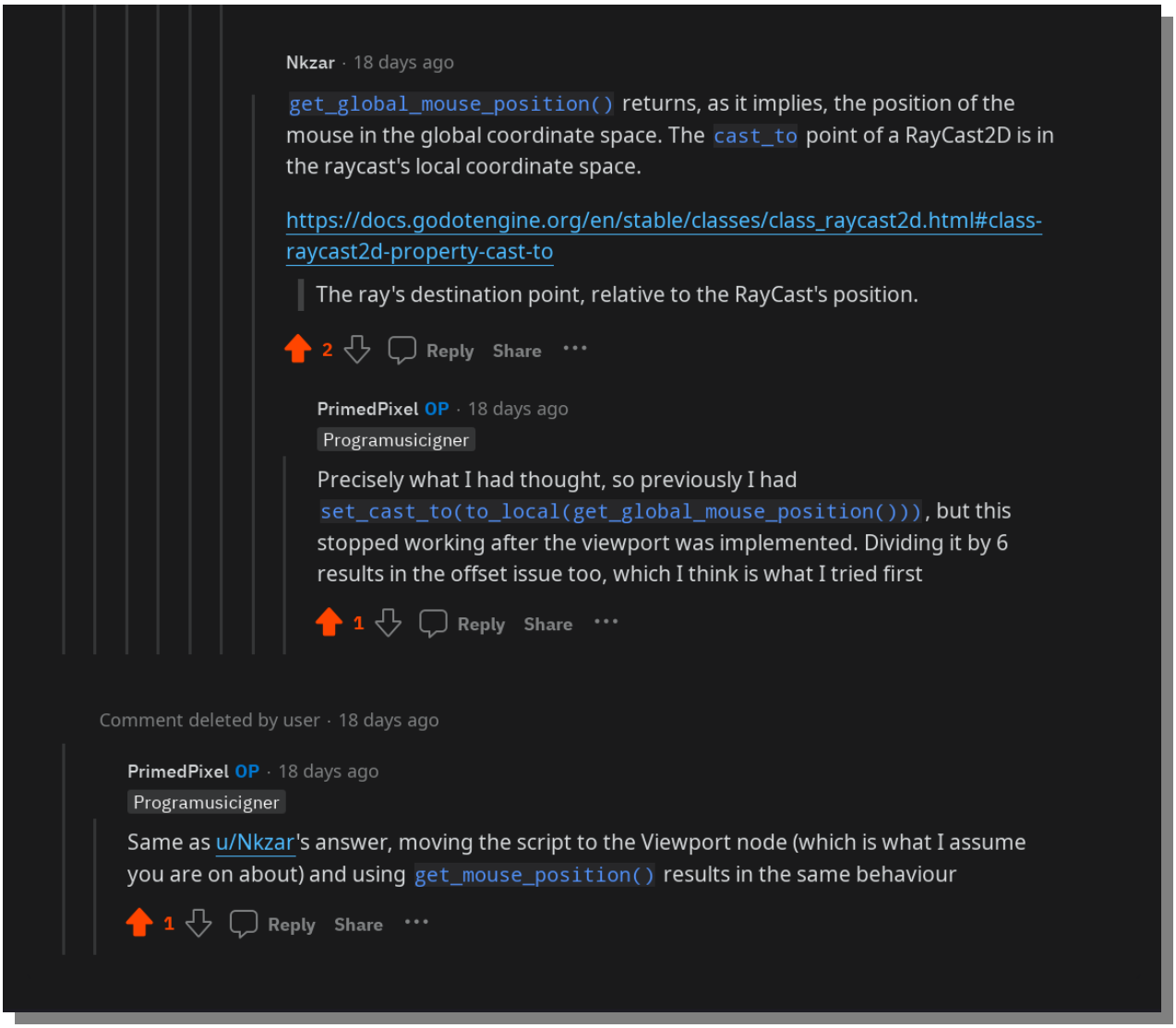
2 Reply Share ···

PrimedPixel OP · 18 days ago

Programusicigner

So, `get_global_mouse_position()` doesn't seem to work inside the raycast or level node (seems to be the same result again?), and the viewport nodes don't have that method. I believe it's what I had originally before I setup the viewports

1 Reply Share ···



So, no. They were not helpful. One person actually tried to help, and another was so embarrassed by their inability to provide useful information that they deleted their comment only 3 minutes after my reply. Thanks Reddit, very cool!

Sarcastic sadness aside, I tried a few more attempts: copying and pasting anything I saw that even *slightly* related to the issue I was having. Although it was unlikely that anything would work.

```
35      var view_container = viewport.get_parent()
36      var mouse_position = view_container.get_local_mouse_position()
37
38      var screen_pos = viewport.get_mouse_position()
39      var ray_origin = cam.global_transform[2].normalized() * cam.global_transform[2].length()
40      var ray_target = cam.get_global_transform().affine_inverse().xform(to_global(screen_pos))
41      var ray_direction = (ray_target - ray_origin).normalized()
42      ray_target = ray_origin + ray_direction * 1000
43      var local_pos = to_local(ray_target) - sprite.global_transform.origin
44      sprite.global_position = to_global(ray_target)
45      set_cast_to(to_local(local_pos))
```

Desperately, in a programming mania, I thoroughly scoured the Internet for anything else that even slightly related to my issue. And I found something beautiful. Something that could end all my suffering. Something big.

Just 11 months ago, the most beautiful human in the world, who goes by the username "u/Tianmaru" on Reddit, posted their troubles with viewports from their Ludum Dare[9] 50 game. And down at the bottom was the most beautiful piece of code[10] one could ever lay eyes on. A resplendent relic, cautiously concealed within the caliginous crevasse of confusion that is Reddit.

The mouse position seems to be off inside my stretched viewport. If your projects stretch mode is set to 2d or viewport, things start to get really messy. As far as I understand it, your viewport contents get basically transformed twice: The first time to fit the ViewportContainer, the second time to fit the screen. Somehow, this causes get_global_mouse_position() to stop functioning as intended, and you would need the inverse transformation of the main port to fix it. I adjusted the fix from ArdaE to fit this problem. In the scene that is displayed in your sub-viewport, use InputEventMouseMotion events to update the mouse position. Use the inverse canvas transform to get world space coordinates.

```
var mouse_pos := Vector2()

func _unhandled_input(event):
    if event is InputEventMouseMotion:
        mouse_pos =  get_canvas_transform().affine_inverse() * event.position
```

So, as far as I can tell, having Godot set to "2D" transformation in project settings means that it essentially gets scaled twice. So, indeed, the inverse matrix needs to be applied – I was correct earlier. But I need to get the mouse position from inside the raycast using an input event. Doing this means that one of the two transformations have been undone, and then it's necessary to undo the transformation once-more.

```
36  # Sets the raycast to the mouse_position
37  # The viewport mode in Godot transforms
38  # The mouse position twice, so it's necessary
39  # To use an inverse matrix first
40  func _unhandled_input(event):
41      if event is InputEventMouseMotion:
42          mouse_pos =  get_canvas_transform().affine_inverse() * event.position
43
44          set_cast_to(to_local(mouse_pos))
```

9 Ludum Dare is a game jam, in which participants must create a game around a designated theme in only 48 or 72 hours. At the end of the jam, all entries are voted for and ranked. I have participated in the past for Ludum Dare 46, with my game "Time is of the Essence", in which I came... 1,635th. Oh well.

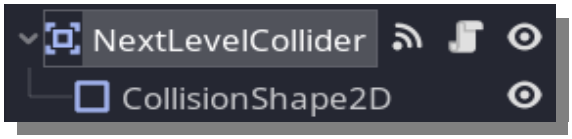
10 See https://www.reddit.com/r/godot/comments/tx9x2b/a_guide_to_mouse_events_in_subviewports/ for the full post.

This concludes a tale that 'twas terrifically terrifying to tell. A shocking story, stocked with sheer sadness, surprises, and satisfaction. Anyway, how does one change levels in my game?

Changing Levels

Now that the viewports were set-up, I couldn't just change the main game scene, as I was previously, to change between levels; I would have to have two scenes per-level, which is inefficient at best. To rectify this, I needed to create some code that deleted the level node, and replaced it with the next specified one. Additionally, I needed an object that allowed me to change levels when colliding with it.

To start, I created a new scene, with the root node being an "Area2D" – this will allow me to check for collisions, in a similar vein to the other collidables (i.e., coins, checkpoints, and spikes). Additionally, I added a "CollisionRectangle2D" child node: this can be resized as and when the scene is being re-used inside levels, meaning any time the player collides with this invisible collider, the level will change.



Similar to the collidables, I used a signal to detect whether a "body" entered the next level, and ensured that this "body" was the player.

```
8  func _on_NextLevelCollider_body_entered(body):
9      if !(body is Player):
10         return
```

Luckily for me, the level node is simply a child of the viewport, so I simply need to get the viewport node, which can be easily done using an built-in function, get the first (and only) child node of the viewport, and delete it. Afterwards, I can create a new node of the new level scene.

However, implementing this into code and running occasionally gave an error: it was possible to have multiple levels at the same time, which caused a plethora of issues. To resolve this, I simply needed to use yield() again – this time, to ensure that an "idle frame" has passed, since deleting a node happens on the next frame after being called. Once this was added, there were no more crashes!

```gdscript
func _on_NextLevelCollider_body_entered(body):
    if !(body is Player):
        return

    Transition.exit_level_transition()
    yield(Transition, "transition_completed")

    var viewport = get_viewport()
    var viewport_child = viewport.get_child(0)
    viewport_child.queue_free()

    # Waits until the scene has been deleted (1 frame)
    yield(get_tree(), "idle_frame")

    var new_scene = target_level_path.instance()
    viewport.add_child(new_scene)

    Transition.enter_level_transition()
```

Setup for "Continue Game"

Allowing the game to be continued was a similar process. I needed to change the current scene to the game, from the menu, and then change the level node (after waiting an idle frame). This was extremely simple to implement, as the code was essentially already written. However, since saving was not yet implemented, I first had to create a global variable, which would eventually store what level to go to, and then set this to level 2 (as a temporary measure).

```gdscript
func _on_Continue_pressed():
    Transition.exit_level_transition()
    yield(Transition, "transition_completed")

    GlobalVariables.level_to = level_2
    get_tree().change_scene("res://Game.tscn")

    Transition.enter_level_transition()
```

Once the game node had loaded, the following code runs:

```
5  func _ready():
6      if GlobalVariables.level_to != null:
7          # Changes the default level in the "Game" scene to the new one
8          var viewport_child = viewport.get_child(viewport.get_child_count() -
9          viewport_child.queue_free()
10
11         # Waits until the scene has been deleted (1 frame)
12         yield(get_tree(), "idle_frame")
13
14         # Creates the new scene
15         var new_scene = GlobalVariables.level_to.instance()
16         viewport.add_child(new_scene)
17
```

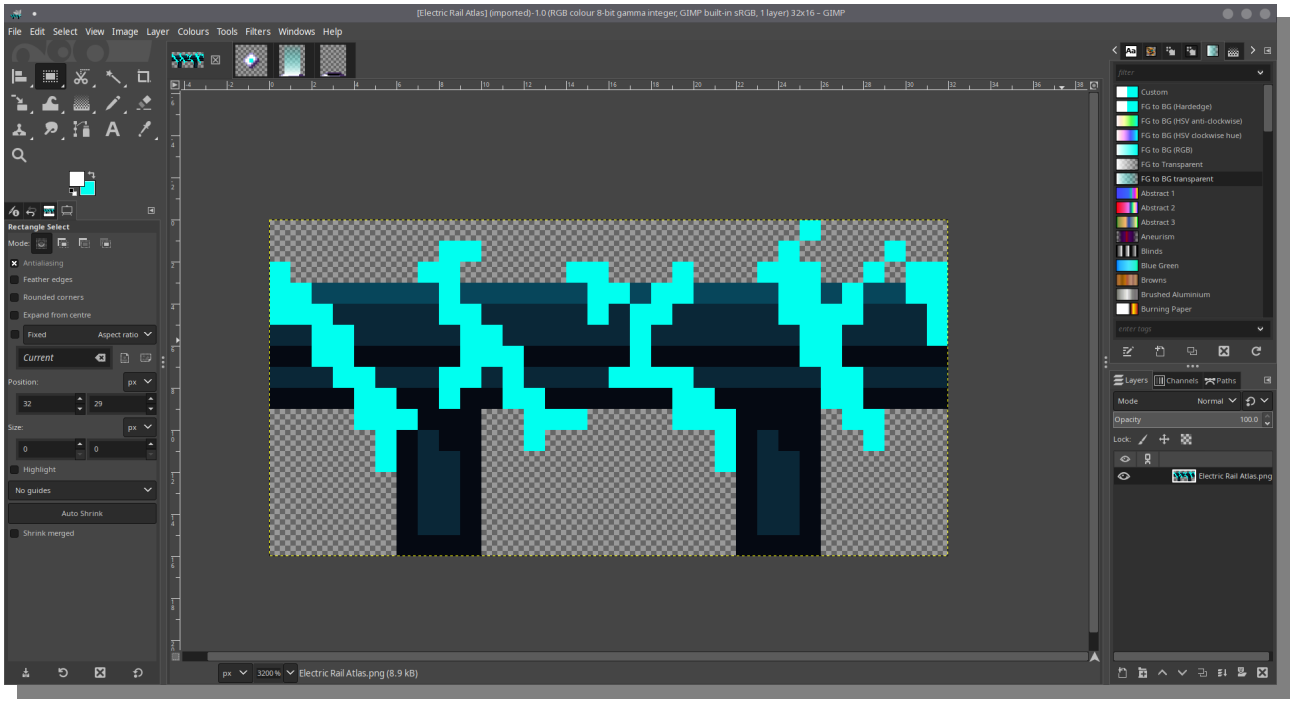
XΞKONIFICATION

At this point, most of the work had been done for the game, but I felt like it was really generic; the sprites, while good, didn't have their own character. Moreover, I wanted to return to a style I previously used for the GMTK Game Jam, in 2021.

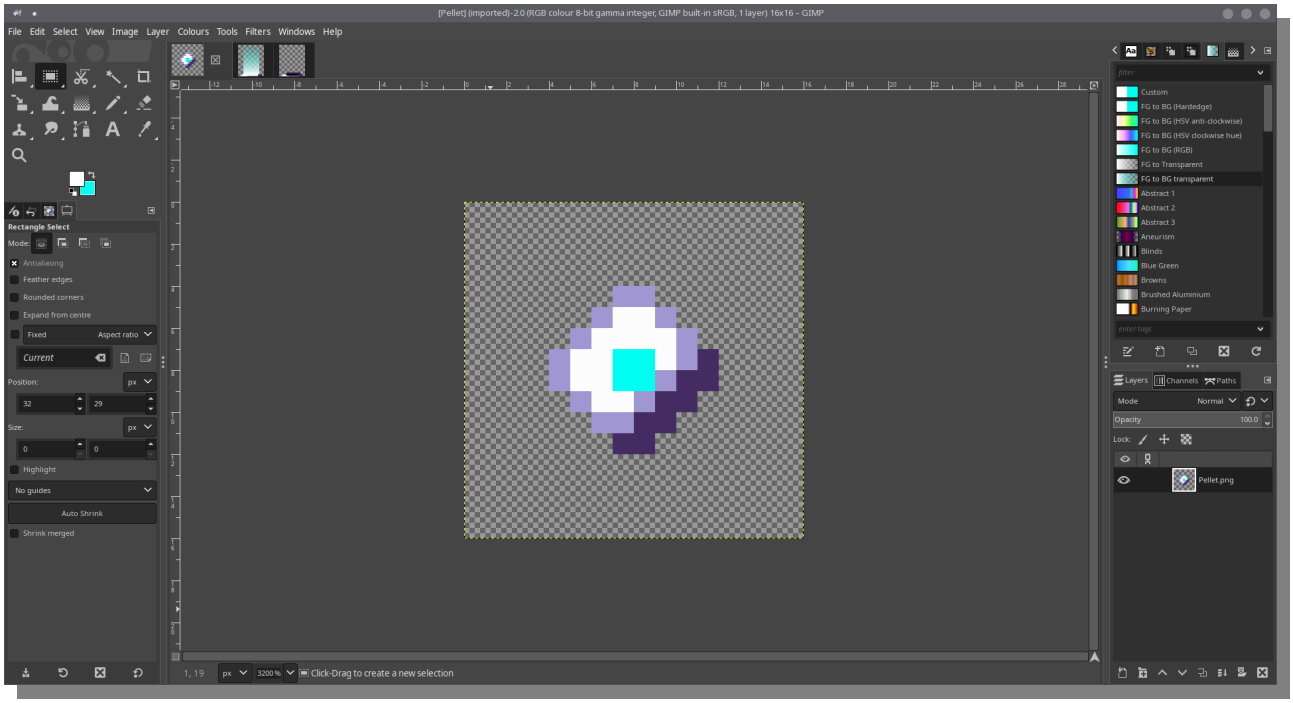
For said game jam, I created XΞKINO, pronounced /zeh-kin-no/, (as mentioned at the beginning of this document) and whilst the branding and overall identity of that game were great – probably the best I have ever created – I did not think the overall concept was that great. So, not wanting to forever have XΞKINO as a failed game jam project, I thought I could re-use said branding, sprites, and aesthetic.

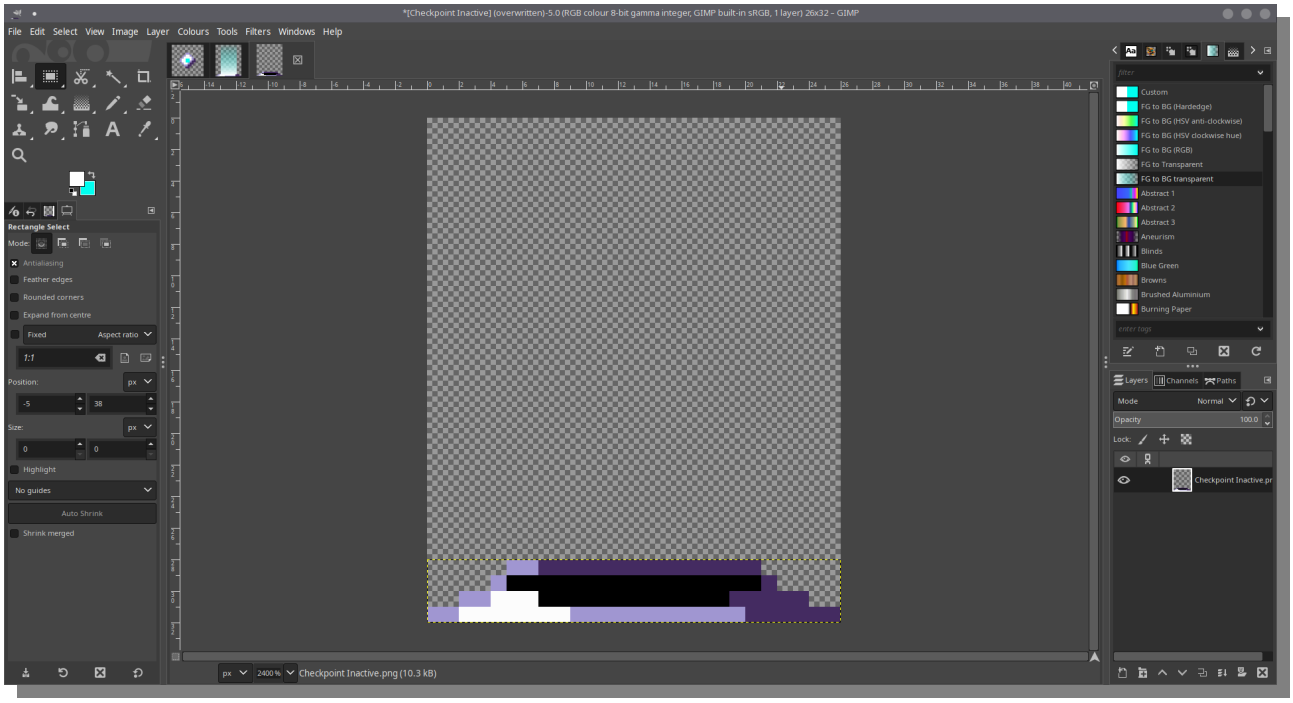
The artwork for XΞKINO was created by “ansimuz”[11], so my first step was to re-download the assets. These were 16 × 16 sprites, unlike the 8 × 8 I was originally working with. This meant that I could either have a more zoomed in game, or zoom-out the game to a resolution of 640 × 360, instead of 320 × 180. I chose the latter, mainly because the game requires the player to be able to look far ahead, which simply was not possible if the game was zoomed in more.

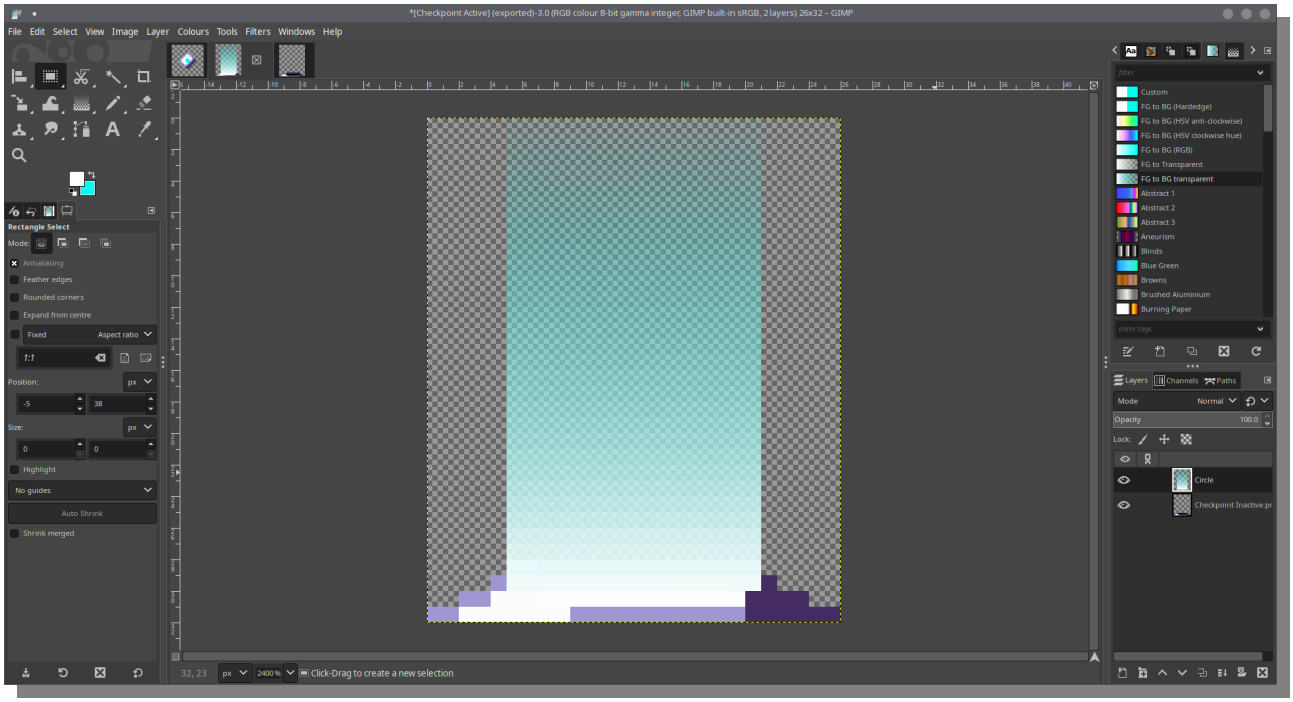
To do this, I simply modified the viewports and viewport code to scale 3×, rather than 6×. Afterwards, I did the same to the menu screen. Now I could update the sprites and tileset. This was fairly simple, as majority of what I needed to do was import new sprites, then tell Godot to work in 16 × 16. Sometimes, there wasn’t spritework for what I needed, so GIMP came in handy once more. Below are the sprites that I created (to match the same aesthetic).

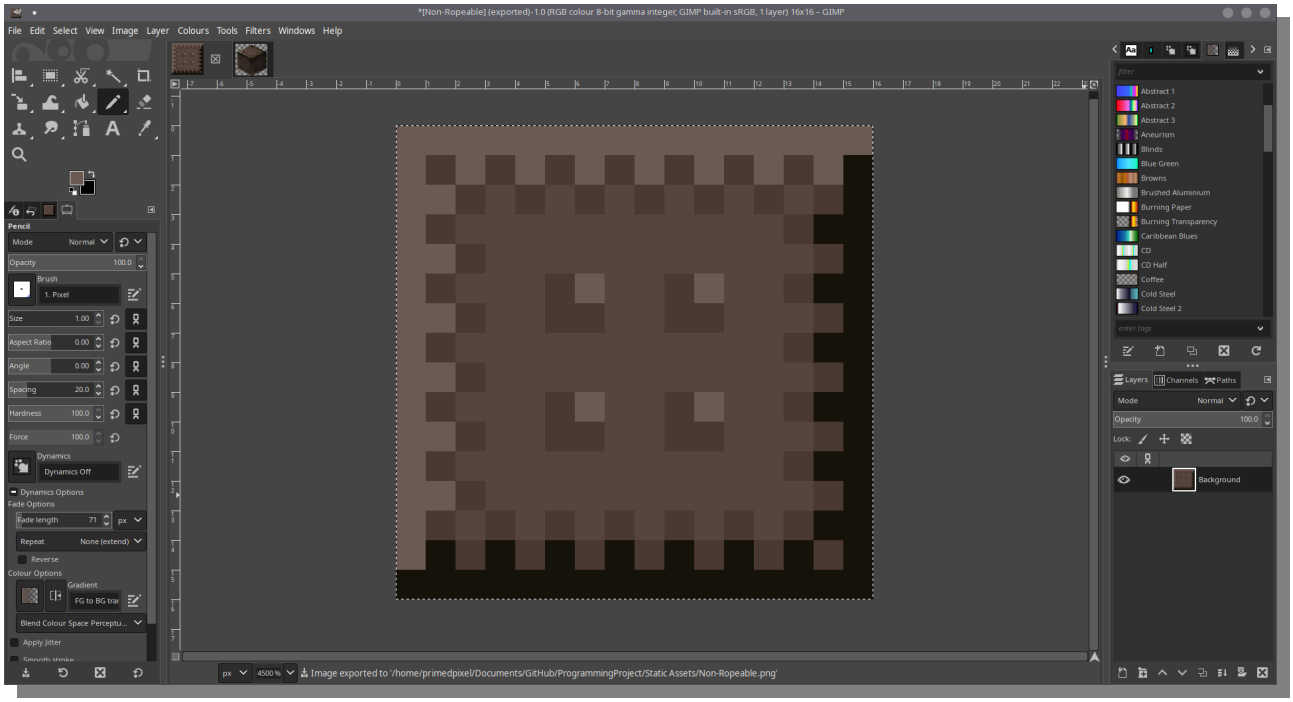


11 The artwork can be found via itch.io. It is part of the “warped collection”. https://itch.io/c/463790/warped, https://ansimuz.itch.io.


[Pellet] (imported)-2.0 (RGB colour 8-bit gamma integer, GIMP built-in sRGB, 1 layer) 16x16 – GIMP
File Edit Select View Image Layer Colours Tools Filters Windows Help
Rectangle Select
Pellet.png
Click-Drag to create a new selection

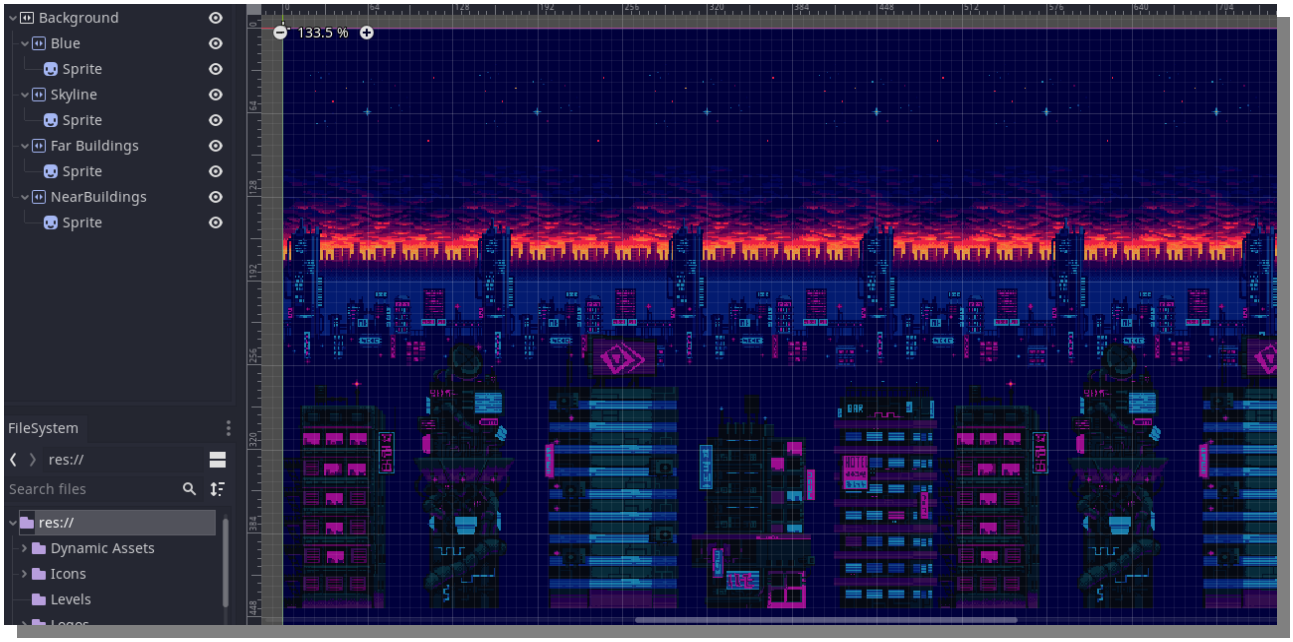

*[Checkpoint Inactive] (overwritten)-5.0 (RGB colour 8-bit gamma integer, GIMP built-in sRGB, 1 layer) 26x32 – GIMP
File Edit Select View Image Layer Colours Tools Filters Windows Help
Rectangle Select
Checkpoint Inactive.pr
Checkpoint Inactive.png (10.3 kB)


*[Checkpoint Active] (exported)-3.0 (RGB colour 8-bit gamma integer, GIMP built-in sRGB, 2 layers) 26x32 – GIMP
File Edit Select View Image Layer Colours Tools Filters Windows Help
Rectangle Select
Click-Drag to create a new selection

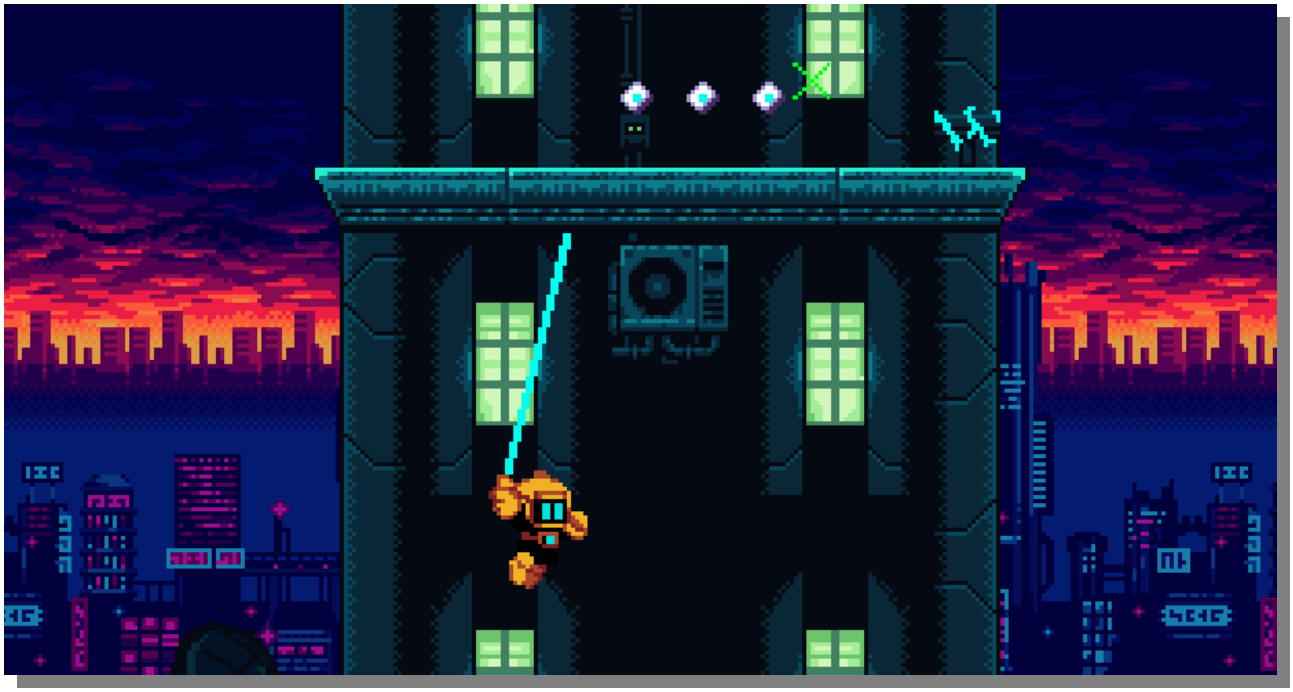

*[Non-Ropeable] (exported)-1.0 (RGB colour 8-bit gamma integer, GIMP built-in sRGB, 1 layer) 16x16 – GIMP
File Edit Select View Image Layer Colours Tools Filters Windows Help
Pencil
Image exported to '/home/primedpixel/Documents/GitHub/ProgrammingProject/Static Assets/Non-Ropeable.png'

Once all the animations were re-done, which was essentially just changing a bunch of values, I decided to add parallax scrolling with multiple background layers.

Doing this was fairly easy: Godot's background node has built-in support for parallax, so it was as simple as setting each node's "region" to the correct size, and ensuring the nodes were in the correct order.



Finally, the game had some proper style!



Interview of Features 5

Does the new aesthetic for the game match the more "unique" style requested in the previous interview?

"Absolutely! This game now feels much less generic, and much more like a well-crafted and polished indie game. The only thing that is needed to complete the style is suitable music, but other than that, it's perfect. Moreover, the name is much more unique and unusual (in a good way) than 'Swing', which didn't entice the player with much."

What features are required to fully complete the game?

"The game needs to be able to pause, to allow players to take a quick break (as it's a singleplayer game). Oh, and the game needs is a fully functional options menu that can be accessed from the main menu and while paused. It would be useful to save the game as well, allowing the player to return to where they were after a previous play-session."

Saving the Game

As said in the interview, an integral part to all games is saving; saving allows for much longer games, as progress is not reset every time the game is closed, as well as saving things like high-scores, or in this case, coin and death counts.

In Godot, the best way to do this is using JSON, as it allows for more readability than XML, and more organisation than INI. Moreover, INI files (and similar) are subject to ACE, or arbitrary code execution. This means that if a player downloads a save file from the Internet, it could contain malicious code that could run in the background, therefore it is imperative to either sterilise the data, or (as is much easier) only allow specific formats, such as JSON.

Luckily, all the variables that needed saving were already stored in the "GlobalVariables" singleton. This meant that I could add the code to save there, which allows for the saving and loading function to be called from anywhere in the game, providing much more flexibility.

Inside the "GlobalVariables" script, I first set-up a file to write to. Next, I needed functions that allows for checking if the file exists, writing the save-game, reading / loading the save-game, and deleting the save-game.

```gdscript
func save_exists():
    return save_file.file_exists(save_file_path)
```

Checking if the file exists is as simple as using the built-in function.

The purpose of creating a custom function is to remove the need to type in the path manually.

Next, I needed a way to write said save-game. This required the following:

- Check that the file exists (i.e., it can be written to)
- Store the data in a dictionary
- Save the data into a JSON string
- Save the JSON string into a file

Each of these stages can be clearly created in code:

```gdscript
func write_savegame():
    # Checks that the file exists
    var error = save_file.open(save_file_path, File.WRITE)

    if error != OK:
        printerr("Could not open save file for writing!")
        return

    # For whatever reason (I suspect since we're running from a singleton
    # get_viewport() does not work, so it must be manually get
    var current_level_path = get_tree().get_current_scene().get_node("ViewportConta

    # Stores the data in a dictionary
    var data = {
        "checkpoint_position":
        {
            "x": checkpoint_pos.x,
            "y": checkpoint_pos.y,
        },

        "stats":
        {
            "coin_count": coin_count,
            "death_count": death_count,
        },

        "continue_level": current_level_path,
    }

    # Saves the data into a JSON formatted string, then writes said string
    var json_str = JSON.print(data)
    save_file.store_string(json_str)
    save_file.close()
```

Loading is quite similar, just in reverse:

- Open the file
- Check that the file exists (i.e., can be read from)
- Load the data into a variable
- Parse the JSON file (i.e., decode the JSON data)
- Apply the data to the appropriate variables

```
49 func load_savegame():
50     # Opens the file
51     var error = save_file.open(save_file_path, File.READ)
52 
53     # Checks the file is readable
54     if error != OK:
55         printerr("Could not open save file for reading!")
56         return
57 
58     # Loads the data into a varialbe and closes the file
59     var file_data = save_file.get_as_text()
60     save_file.close()
61 
62     # Parses (decodes) the JSON data
63     var data = JSON.parse(file_data).result
64 
65     # Applies the data to the variables
66     checkpoint_pos = Vector2(data.checkpoint_position.x, data.checkpoint_position.y)
67     coin_count = data.stats.coin_count
68     death_count = data.stats.death_count
69 
70     level_to = data.continue_level
```

Finally, to delete the file (which is useful when selecting new game), it's a simple as getting the save file directory, and deleting the file. It's then important to reset the variables.

```
72 func delete_savegame():
73     if save_exists():
74         var dir = Directory.new()
75         dir.remove(save_file_path)
76 
77     # Perhaps there's a way to do this without repeating?
78     checkpoint_pos = Vector2.ZERO
79     coin_count = 0
80     death_count = 0
81     level_to = -1
82 
```

Once this was all implemented, it's as simple as loading the save-game when the game starts, and also allowing for the player to save and exit the game. This will be added alongside...

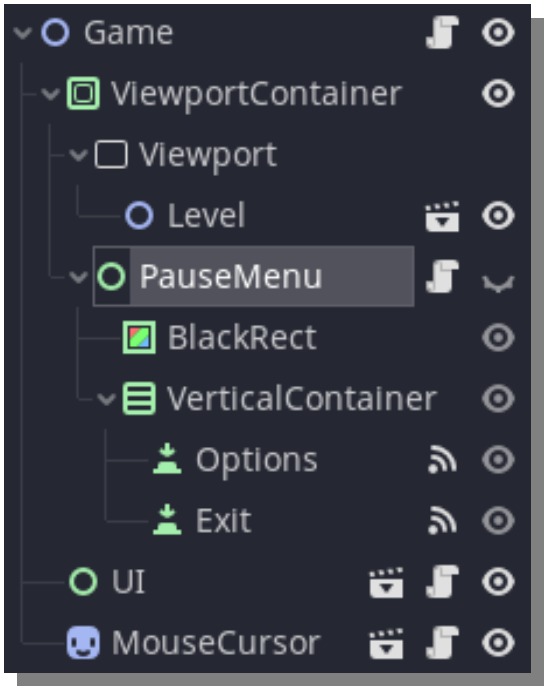
Pausing

Luckily, pausing in Godot is already built-in! This means that all that is needed is a pause menu, which can be built in a similar way to the main menu.

Inside the viewport container node (*not* the viewport node, as this can only contain the level), I added a "Control" node named "PauseMenu". This allows me to put not only just a "VerticalContainer" node, but also a translucent, black rectangle to help indicate that the game is paused.

Inside the "VerticalContainer" node are just two buttons: options and exit. For now, options does not do anything, but exit saves the game (simply using the write_savegame() method inside "GlobalVariables"), and then changes back to the menu.

In order to allow the game to switch between paused and unpaused states, I simply check whether the escape key has been pressed, and if it has, the "PauseMenu" node's visibility is toggled. This is useful in two ways: it shows and hides the pause menu and rectangle, but also the buttons cannot be interacted with if the node is not visible, meaning the menu can only be interacted with if the game is paused.



```gdscript
func _process(_delta):
    var key_pause = Input.is_action_just_pressed("button_pause")

    # Inverts the pause state (i.e., true -> false, and false -> true)
    if key_pause:
        visible = !visible
        get_tree().paused = visible
        $VerticalContainer/Options.grab_focus()


func _on_Options_pressed():
    pass # Replace with function body.


func _on_Exit_pressed():
    Transition.exit_level_transition()
    yield(Transition, "transition_completed")

#   music.fade_out()
#   yield(fade_out, "tween_completed")
    get_tree().paused = false

    GlobalVariables.write_savegame()
    get_tree().change_scene("res://Menu.tscn")

    Transition.enter_level_transition()
```

However, when I went to run this code, everything paused, but I seemingly couldn't interact with anything. Luckily, I realised that this was because I had neglected to set the node's "pause mode". In Godot, nodes can either stop or process while paused. It's important to set all nodes that need to run while paused to "process", whereas all others need to be set to "stop". Once all the nodes were changed – especially remembering to change the transition node, as otherwise the yield() function would never end, and the mouse cursor node – pausing, indeed worked.

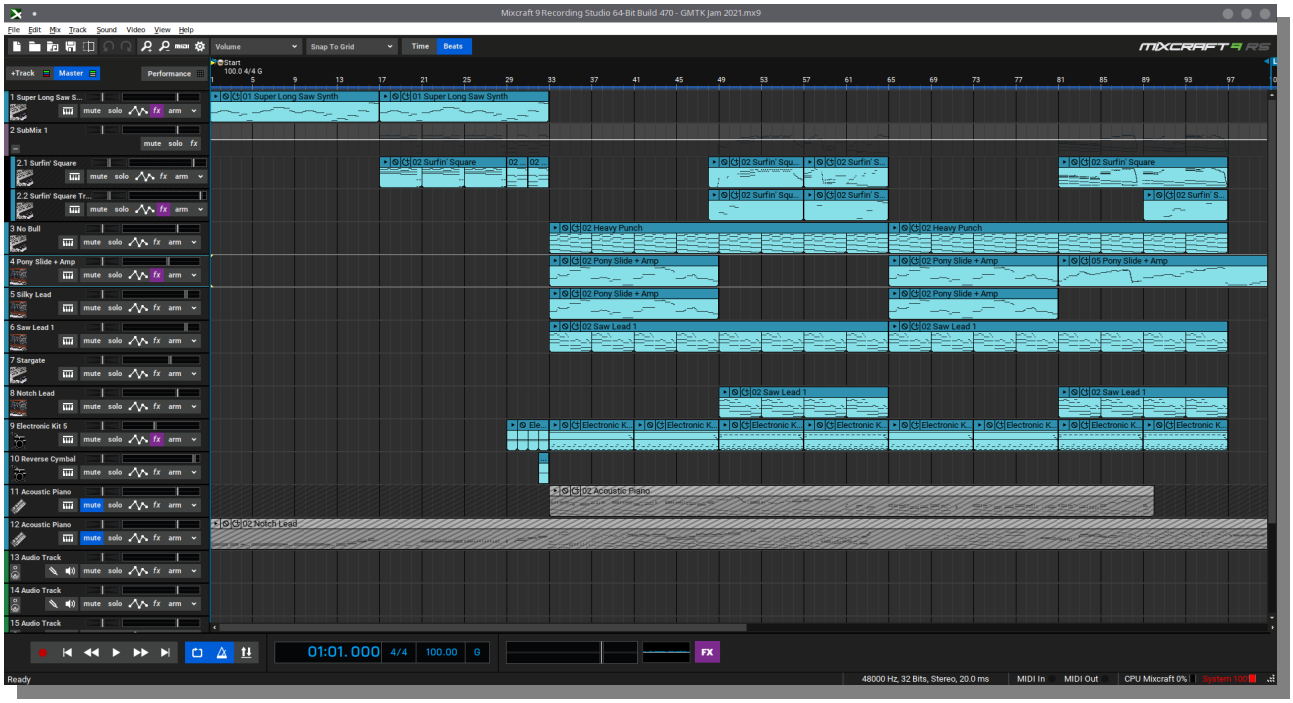
Adding Music

Now that the overall aesthetic of the game was settled, it was music time! To compose music, I use the DAW (digital audio workstation) Mixcraft.

I already had two songs fully composed (and released on Spotify[12]), however I decided to write some more so that each level could have its own piece. I went about doing this fairly sporadically, as I usually do when writing music. In fact, two of the three compositions were partially written before this project was even started. However, majority of the production was during the project.

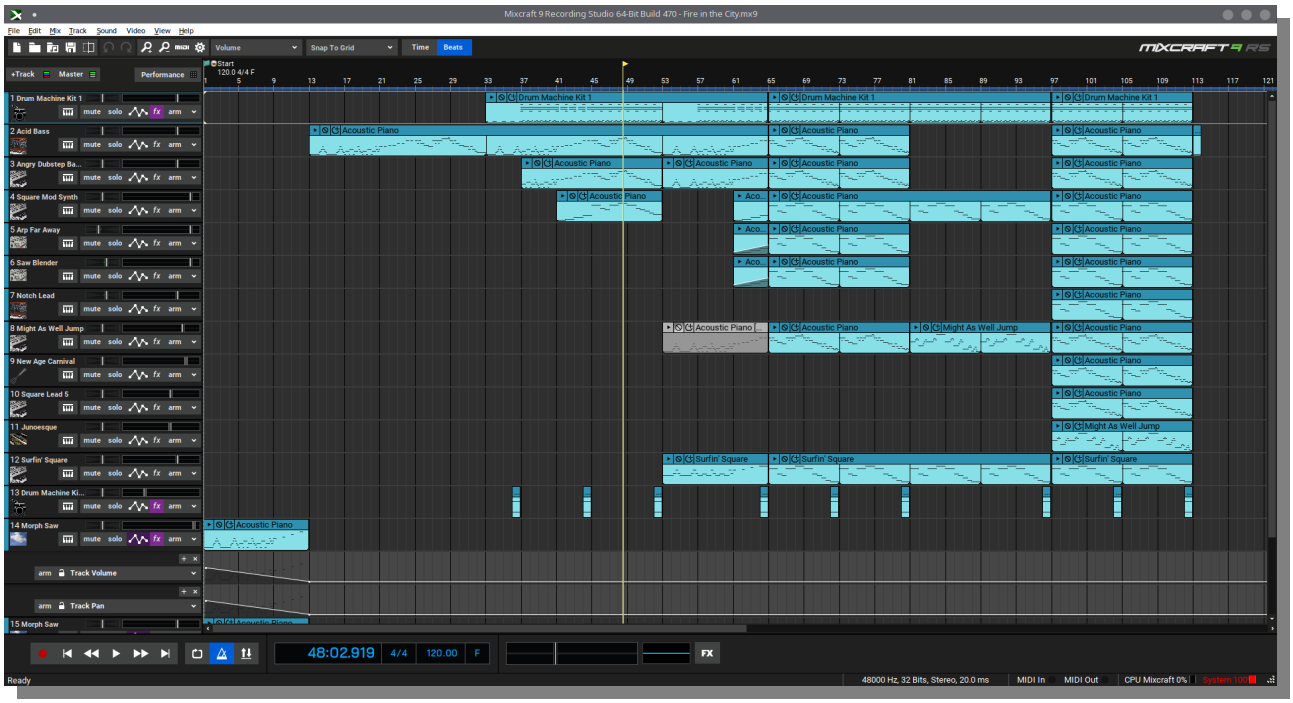
I used my keyboard as MIDI input to Mixcraft, and played the lead track. Chords, bass, and other tracks were added manually using a piano roll. Since this is a *programming* project, and not a music project, the rest of this section will just contain a screenshot of the Mixcraft project, as well as a few facts about each piece.

12 My music is available on many platforms, including Spotify: https://open.spotify.com/artist/4vae5dm3sk6jtWNcBPXVxU, and many other music platforms, under my online alias “Primed Pixel”.



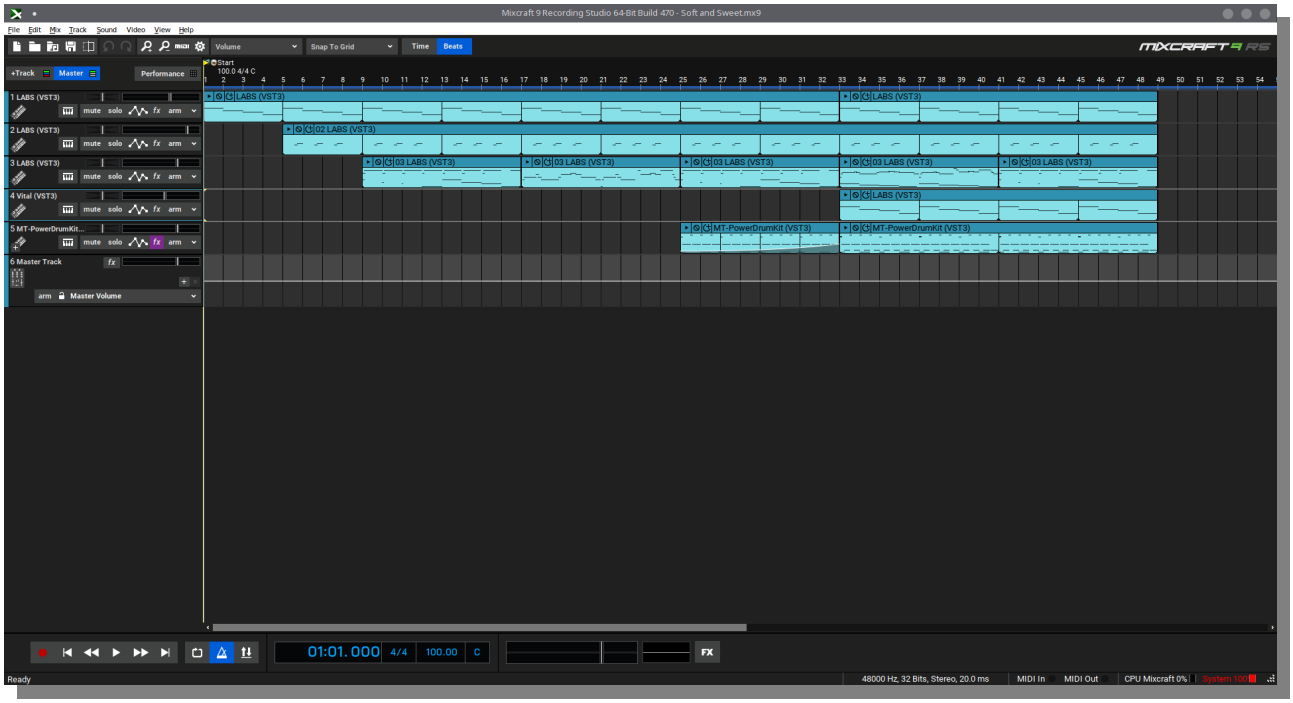
- The title of this composition is a direct reference to the GMTK Game Jam 2021 version of XΞKINO, which had a slight bit of story written… in the last hour of the jam.
- This is the first song I wrote using a MIDI keyboard – the original “recording” can be seen at the bottom, greyed out. All my previous compositions (a full album, two singles, and a variety of other, non-released projects) were completely written using a mouse.
- I was completely obsessed with G-minor. Fun fact.

Fire in the City



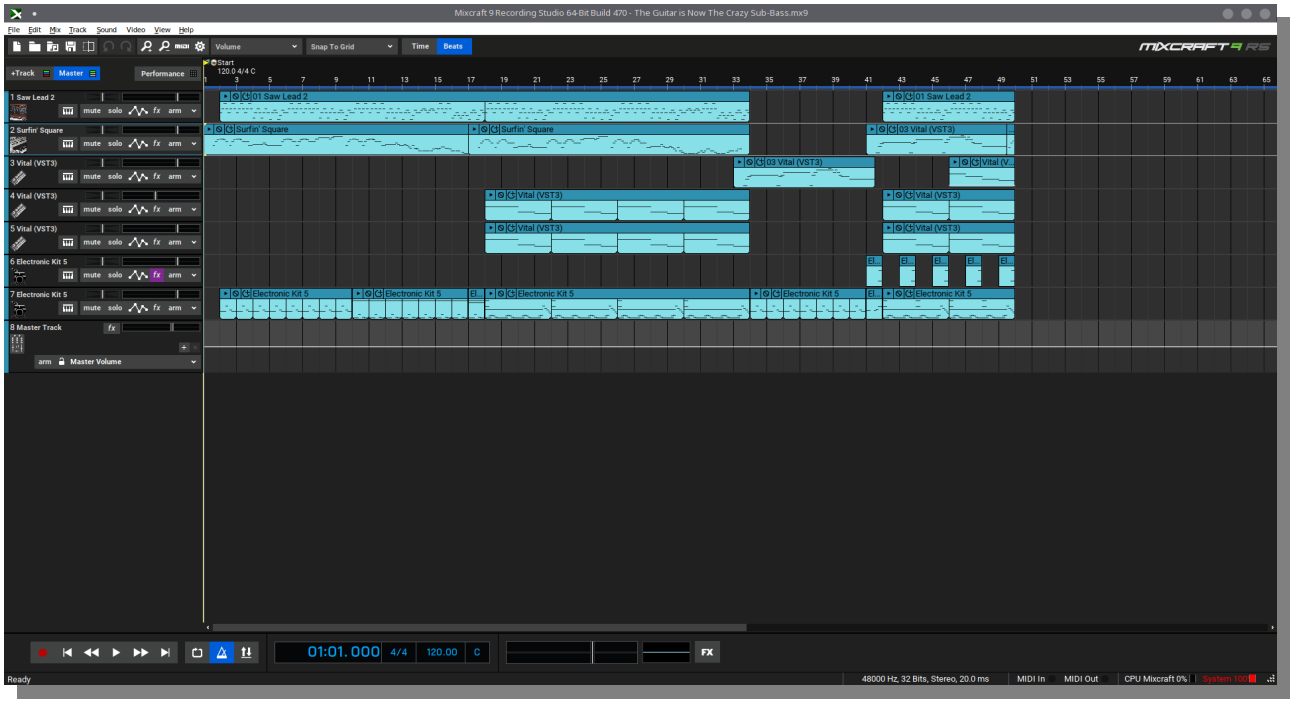
- This piece was written on a guitar, but then manually transcribed by hand
- The original guitar piece was created at the beginning of 2020. The only reason I remember this is because 2020 was a horrible year afterwards. Maybe this cursed it.
- The name for this composition was created before the final level in the GMTK Game Jam 2021 game, in which the player must traverse the cityscape as fast as possible, away from a wall of fire. I felt the name matched the piece (for whatever reason) which then inspired the game's final level.
- Because this was written on guitar, it's not in G-Minor, otherwise it probably would be.

“Soft and Sweet” (Working Title)



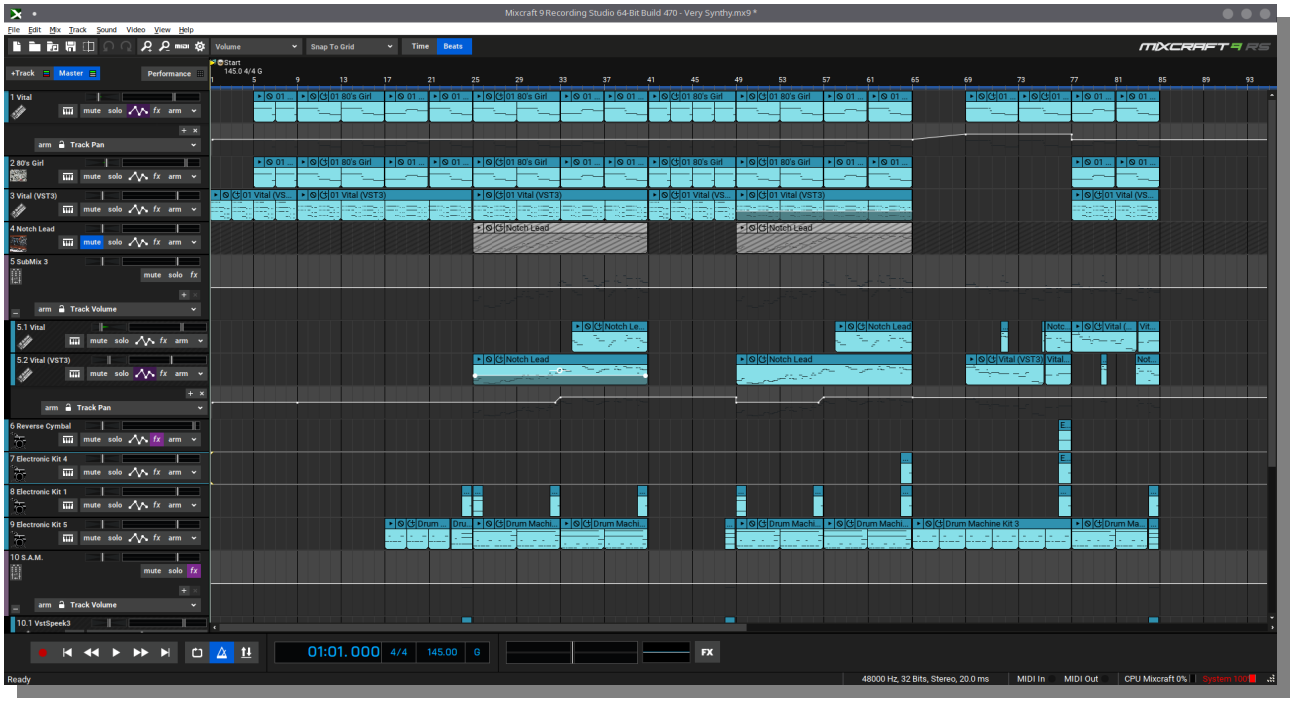
- This was the first song completed that wasn’t in the original GMTK Game Jam 2021 game.
- It was originally written for the more generic aesthetic (although it really didn’t fit it, but that’s why it’s not in minor)
- Unlike many of my other compositions, the synths are mostly made by Spitfire Audio, in “LABS”, and the drums are sampled, not electronic, drum-machine-esque.
- I’m really bad at titling pieces – most of my previous compositions were named using a random noun and adjective. Such favourites include “Space Cows”, “Purple Mangos”, “Electric Diamond”, “Pink Puppy”.

“The Guitar is now the Crazy Sub-Bass” (Working Title)



- This is a piece I constantly played on guitar randomly, so I have absolutely no clue when it was first written. There is a chance it is the oldest of the five pieces, and I suspect that is the case.
- It was originally in E-minor, however because I’m obsessed with G-minor, it was transposed when I played it through the MIDI keyboard.
- The sub-bass part was done by complete accident – the pitch shift was supposed to be much quicker – yet is probably the most defining part.

“Very Synthy” (Working Title)



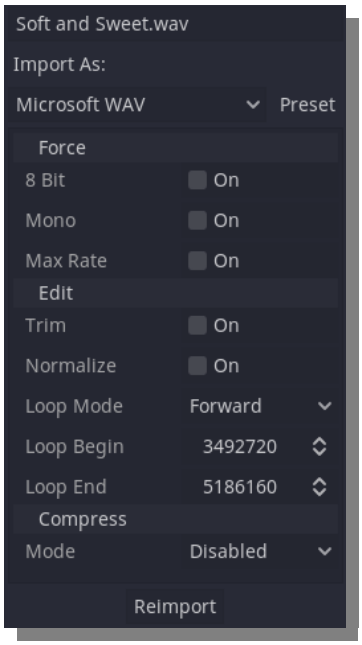
- This has the worst title out of the bunch. My guess is that I completely ignored this project file because of its name, assuming it was just a short improvisation I had done. This was not the case, and half of the composition was already done once I re-discovered it for this project.
- In the piece, a robotic “1, 2, 3, 4” can be heard. This is actually a software emulation of Commodore S.A.M. – software automatic mouth – a voice-synthesis cartridge for the Commodore 64. If I had enough money, I would buy the cartridge and actually record it from my Commodore.
- Wow, it’s in G-minor. What a surprise

So, how did I go about adding the music into Godot? The first thing that I needed to do was to find the loop points, as Godot can use these to loop the song at the correct point, so it doesn’t just end after being played once.

To do this accurately, I needed to find out the exact sample that the loop points were at. Luckily, I knew the exact sample-rate, beats-per-minute, and bars where the loop points were. To calculate the point, I used the following formula:

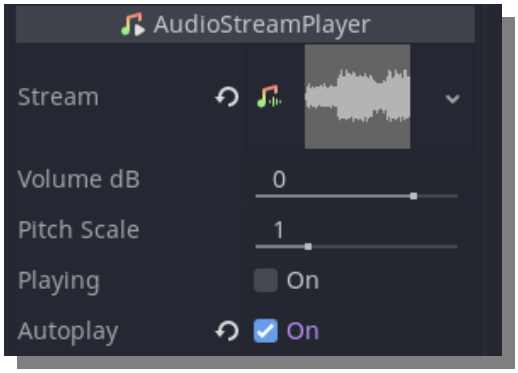
$$\frac{tm}{\left(\frac{b}{60}\right)} \times s$$

where t is the first number in the *time signature* (i.e., how many beats in a bar), m is how many bars (measures) in the loop point is, b is the BPM, and s is the sample-rate in hertz.



Once I had these values for each loop point, I imported the files into Godot, and set each one appropriately.

Finally, it was as simple as adding another "AudioStreamPlayer" node, with "Autoplay" enabled, for each scene that had music.



Adding an Options Menu

There is one last, crucial thing that I need to add before I can focus on level-design: an options menu. This can be created using the same process as the main menu. Since an options menu is more complicated, it's a good idea to plan it out before. In the design section, this planned design can be seen. Using this, I planned out which nodes I would need. This is important as the options menu needs to be able to be used from the pause and main menu.

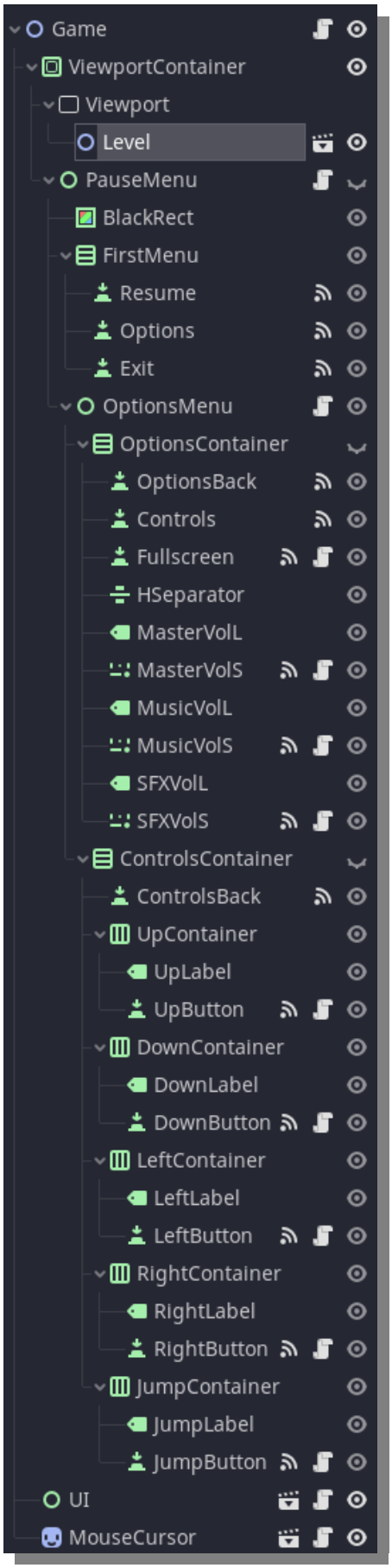
The planned node tree I used is as follows:

- Options menu node (this is necessary as this node will be turned into a scene)
  - Options container (contains all the buttons for the options menu)
    - Back button
    - Controls button
    - Fullscreen button
    - Horizontal separator
    - Master volume slider
    - Music volume slider
    - Sound effects slider
  - Controls container (contains all the buttons for changing controls)

- Back button
- Up container
    - Up label
    - Up button
- Down container
    - Down label
    - Down button
- Left container
    - Left label
    - Left button
- Right container
    - Right label
    - Right button
- Jump container
    - Jump label
    - Jump button

So, what exactly are these nodes? The buttons, slider, and horizontal separator are self-explanatory, but what about the containers for the controls menu? These are so the controls menu can display two things: the actual action being changed (e.g., jump, move right etc.) and the button it is currently bound to. The containers simply allow them to be horizontally “tiled”, and vertically aligned with each other.

Now that these nodes were planned out, I added them into Godot, alongside the already set-up pause menu. Crucially, in order to allow the options menu to be used in two places, I renamed the pause menu “VerticalContainer” node to “FirstMenu”, and did the same on the main menu.

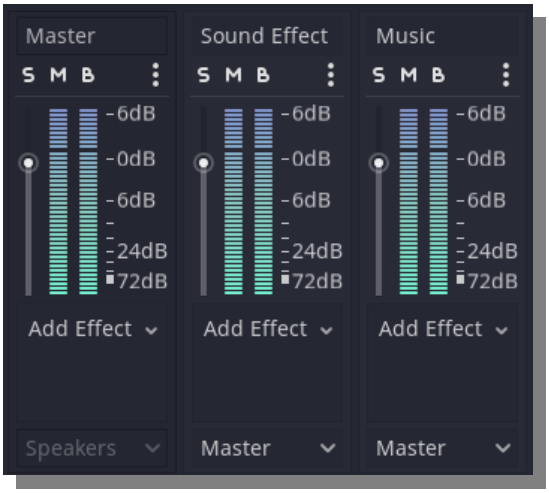


The first step was to show the options menu when the “Options” button was pressed. Luckily, this is fairly simple, and just involves setting certain nodes to be visible / invisible, as well as “grabbing the focus” of the first node in that menu (i.e., which button should be selected first).

```
25 func _on_Options_pressed():
26     pause_container.visible = false
27     options_container.visible = true
28
29     options_container_back.grab_focus()
```

Next, I attached a script to the “OptionsMenu” node, which contains all the logic for both the options and controls menu. Many of these functions are self-explanatory: the back button returns back to the pause menu; the controls button goes to the controls menu; the fullscreen button toggles fullscreen; the volume sliders change the volume of their respective audio bus.

An audio bus is essentially a virtual audio channel that is all eventually mixed into the master bus. I set these up in Godot in such a way that the music gets played through the music bus, and the sound effects through the sound effect bus.



Because decibels are a logarithmic scale, it's important to convert this back into a linear scale when using audio sliders. This can be done in Godot using the built-in function "linear2db()", and "db2linear()" respectively.

```
35 func _on_Controls_pressed():
36     controls_container.visible = true
37     options_container.visible = false
38
39     controls_container_back.grab_focus()
40
41
42 func _on_Fullscreen_pressed():
43     OS.window_fullscreen = !OS.window_fullscreen
44     fullscreen_button.update_text()
45
46
47 func _on_MasterVolS_value_changed(value):
48     if master_bus != null:
49         AudioServer.set_bus_volume_db(master_bus, linear2db(value))
50
51
52 func _on_MusicVolS_value_changed(value):
53     if music_bus != null:
54         AudioServer.set_bus_volume_db(music_bus, linear2db(value))
55
56 func _on_SFXVolS_value_changed(value):
57     if sfx_bus != null:
58         AudioServer.set_bus_volume_db(sfx_bus, linear2db(value))
```

However, what if the game starts with only 50% audio volume, instead of the default 100%? The slider won't display the right value. To fix this, when the slider nodes load, they simply change their value to the respective volume already set. This volume will be loaded alongside the save.

```
3  onready var master_bus = AudioServer.get_bus_index("Master")
4
5  # Called when the node enters the scene tree for the first time.
6  func _ready():
7      value = db2linear(AudioServer.get_bus_volume_db(master_bus))
```

What's more complicated is the controls menu, as this needs to display two values, and have the ability to stop inputs whilst the controls are being changed.

To go about this, I added two variables, which keep track of whether a key-bind is being changed, and what key that is. Once a button is pressed to change a key-bind, all nodes lose their "focus" functionality – this means that the "focus" cannot be changed, i.e., it will not deselect the node, even if the up / down keys are pressed (which can be used to navigate the menu otherwise).

```
81 func _on_UpButton_button_down():
82     changing = true
83     changing_key = "button_up"
84 
85     disable_controls_focus()
86 
87     up_button.set_text("Press Key")
```

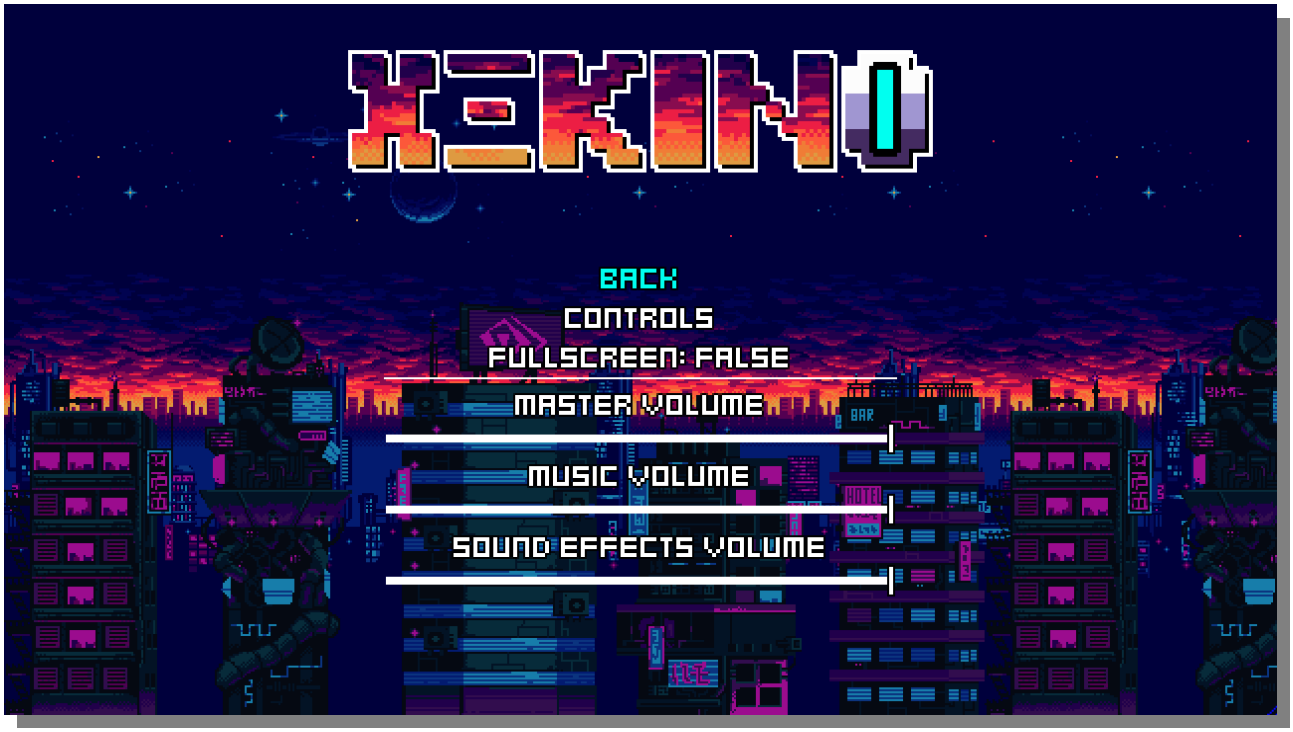
```
126 func disable_controls_focus():
127     up_button.focus_mode = Control.FOCUS_NONE
128     down_button.focus_mode = Control.FOCUS_NONE
129     left_button.focus_mode = Control.FOCUS_NONE
130     right_button.focus_mode = Control.FOCUS_NONE
131     jump_button.focus_mode = Control.FOCUS_NONE
```

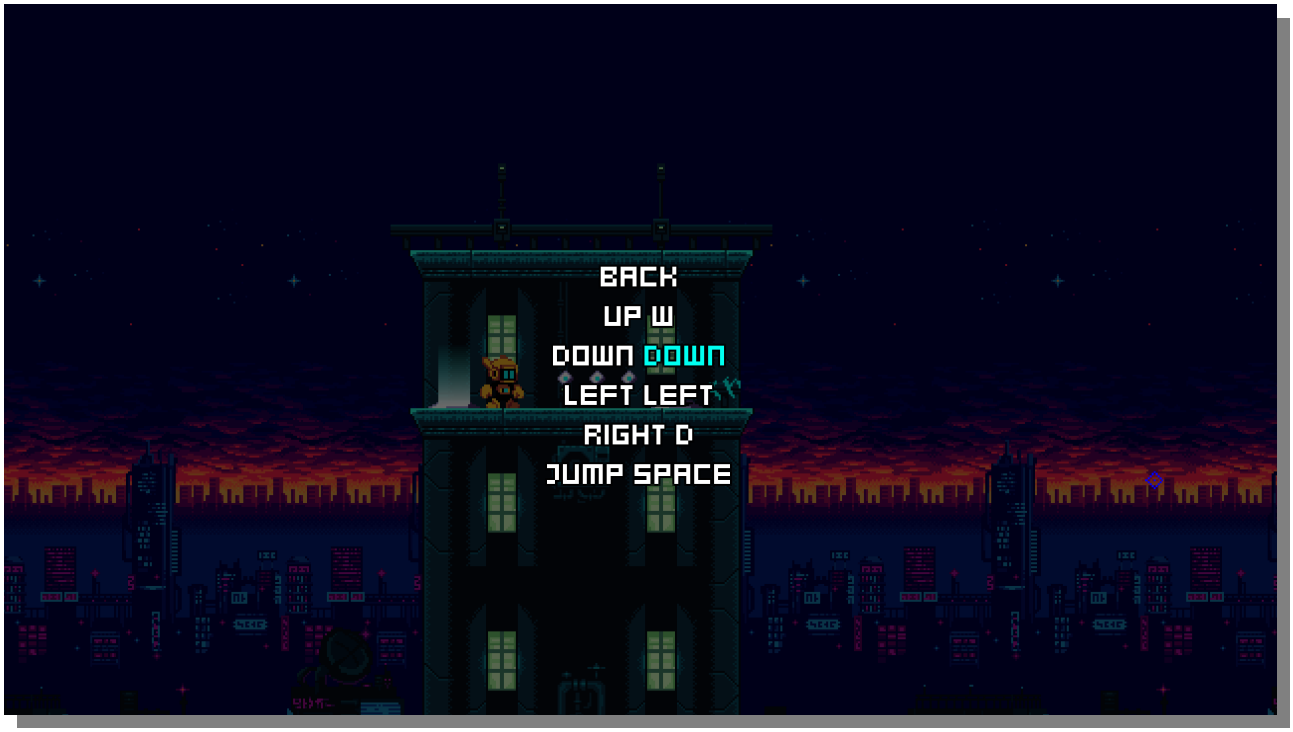
The game then checks for any keys to be pressed, and makes sure that the key is not enter (as this would then disable the button). If it's not enter, then the game changes the respective key to whichever was pressed down, updates the text of the correct button, and re-focuses the node.

```
69 func change_key(map_key, new_key):
70     # Delete key of pressed button
71     if !InputMap.get_action_list(map_key).empty():
72         InputMap.action_erase_event(map_key, InputMap.get_action_list(map_key)[0])
73 
74     # Add new Key
75     InputMap.action_add_event(map_key, new_key)
```

```
134 func _input(event):
135     if event is InputEventKey:
136         if InputMap.action_has_event("ui_accept", event):
137                 return
138 
139         if changing:
140             change_key(changing_key, event)
141             changing = false
142 
143             yield(get_tree(), "idle_frame")
144 
145             up_button.focus_mode = Control.FOCUS_ALL
146             down_button.focus_mode = Control.FOCUS_ALL
147             left_button.focus_mode = Control.FOCUS_ALL
148             right_button.focus_mode = Control.FOCUS_ALL
149             jump_button.focus_mode = Control.FOCUS_ALL
150 
151             match changing_key:
152                 "button_up":
153                     up_button.update_text()
154                     up_button.pressed = false
155                     up_button.grab_focus()
156                 "button_down":
157                     down_button.update_text()
158                     down_button.pressed = false
159                     down_button.grab_focus()
160                 "button_left":
161                     left_button.update_text()
162                     left_button.pressed = false
163                     left_button.grab_focus()
164                 "button_right":
165                     right_button.update_text()
166                     right_button.pressed = false
167                     right_button.grab_focus()
168                 "button_jump":
169                     jump_button.update_text()
170                     jump_button.pressed = false
171                     jump_button.grab_focus()
```

Now all that is needed is to convert the node to a scene, and add it to both menus.


BACK
CONTROLS
FULLSCREEN: FALSE
MASTER VOLUME
MUSIC VOLUME
SOUND EFFECTS VOLUME


BACK
UP W
DOWN DOWN
LEFT LEFT
RIGHT D
JUMP SPACE

Interview of Features 6

Are there any more features required for the game to be considered fully-functional?

“I don’t think so – there are obviously additional platformer techniques that could be added if there were no time constraints, but otherwise, the game has enough to be considered done. All that is needed now is a few levels, perhaps enough to match all the music.”

Debug Menu

Before I began designing levels, I knew it was imperative that I created some tools to allow me to debug different aspects of the game, as well as keep track of things such as node counts, player speed, and checkpoint position. To do this, I created a new node inside the “Game” scene, and added a “Label” node to it. The label node will contains a lot of helpful information, as seen below:

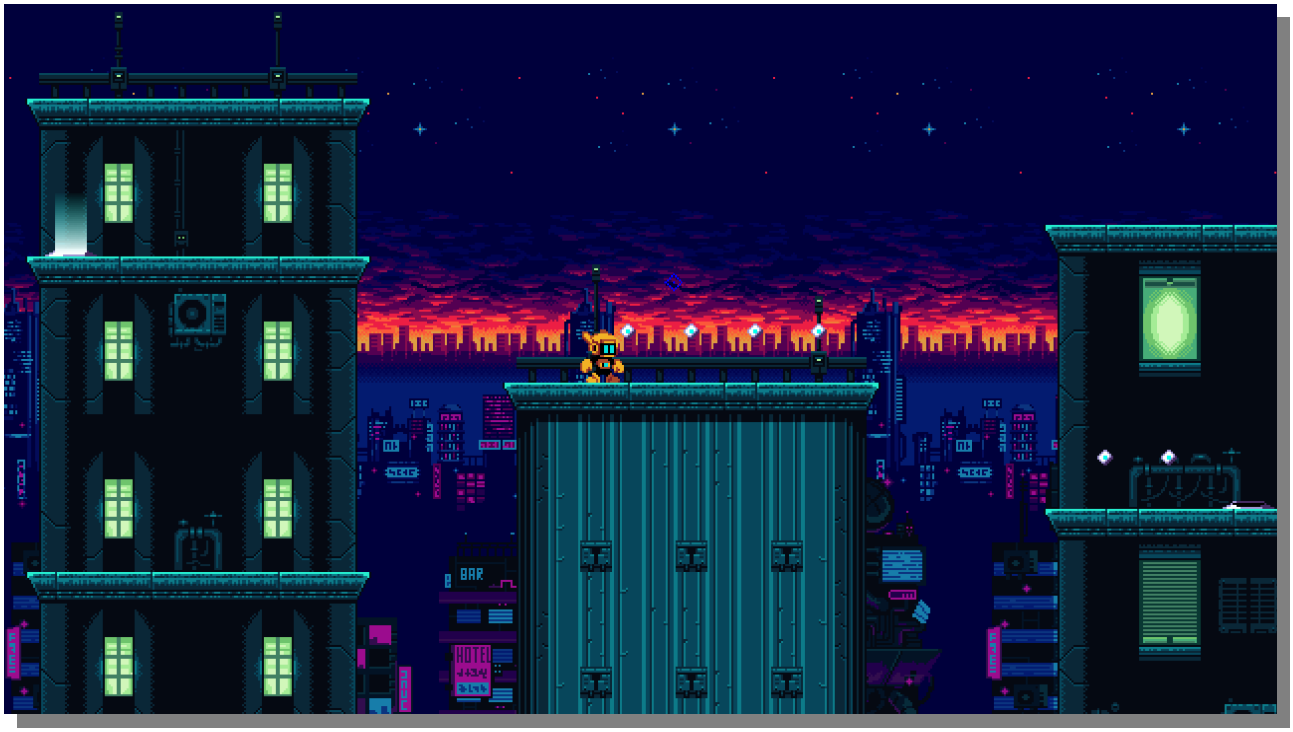
```
44          var rope_angle = fmod(player.angle_to, (2 * PI))
45
46          label.text = "FPS: " + str(Engine.get_frames_per_second()) +                \
47              "\nFullscreen: " + str(OS.window_fullscreen) +                          \
48              "\nMaster Vol: " + str(master_vol) + "db, " + str(db2linear(master_vol)) + \
49              "\nMusic Vol: " + str(music_vol) + "db, " + str(db2linear(music_vol)) + \
50              "\nSFX Vol: " + str(sfx_vol) + "db, " + str(db2linear(sfx_vol)) +     \
51                                                                                     \
52              "\n\nPlayer Velocity: " + str(player.motion) +                         \
53              "\nPlayer Speed: " + str(player.motion.length()) +                     \
54              "\nPlayer Position: " + str(player.position) +                         \
55              "\nPlayer Global Position: " + str(player.global_position) +           \
56              "\nPlayer State: " + str(player.player_state) +                        \
57              "\nPlayer Collision: " + str(player.colliding) +                       \
58              "\nPlayer is_on_floor: " + str(player.is_on_floor()) +                 \
59              "\nPlayer is_on_wall: " + str(player.is_on_wall()) +                   \
60              "\nPlayer is_on_ceil: " + str(player.is_on_ceiling()) +               \
61                                                                                     \
62              "\n\nRope Pos Player: " + str(player.rope_pos) +                       \
63              "\nRope Pos Caset: " + str(player.cast) +                              \
64              "\nRope Length: " + str(player.rope_len) +                             \
65              "\nRope Angle: " + str(rope_angle) + ", " + str(rad2deg(rope_angle)) + "°" + \
66                                                                                     \
67              "\n\nCamera Position: " + str(cam.position) +                          \
68              "\nCamera Global Position: " + str(cam.global_position) +              \
69              "\nCamera Interp: " + str(cam.interpolate_val) +                       \
70                                                                                     \
71              "\nSpike Node Count: " + str(spikes.get_child_count()) +              \
72              "\nDeath Count: " + str(GlobalVariables.death_count) +                \
73              "\nCheckpoint Node Count: " + str(checkpoints.get_child_count()) +    \
74              "\nCheckpoint Position: " + str(GlobalVariables.checkpoint_pos) +     \
75              "\nCoin Node Count: " + str(coins.get_child_count()) +                 \
76              "\nCoin Count: " + str(GlobalVariables.coin_count)
77
78          if fire != null:
79              label.text += "\nFire Distance: " + str(fire.fire_distance) +        \
80                            "\nFire Speed: " + str(fire.fire_speed)
```

Designing Levels

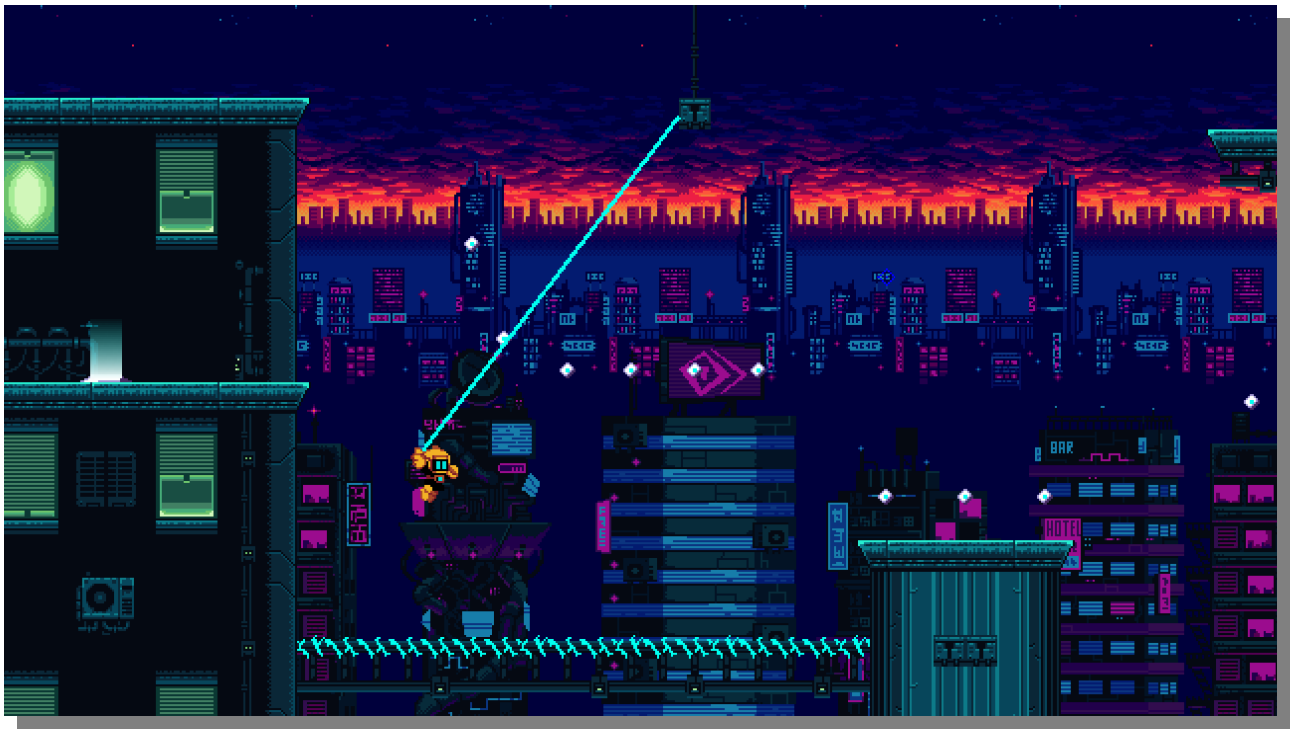
At last, the programming is complete! And just as the interview says, the final part of the development process is to design levels – 4 in total, to match the 4 pieces of music I've created.

As majority of the reasoning behind general level-design was already detailed in the design section, the following will be general insights into certain portions of each level which reflect these decisions.

Level #1

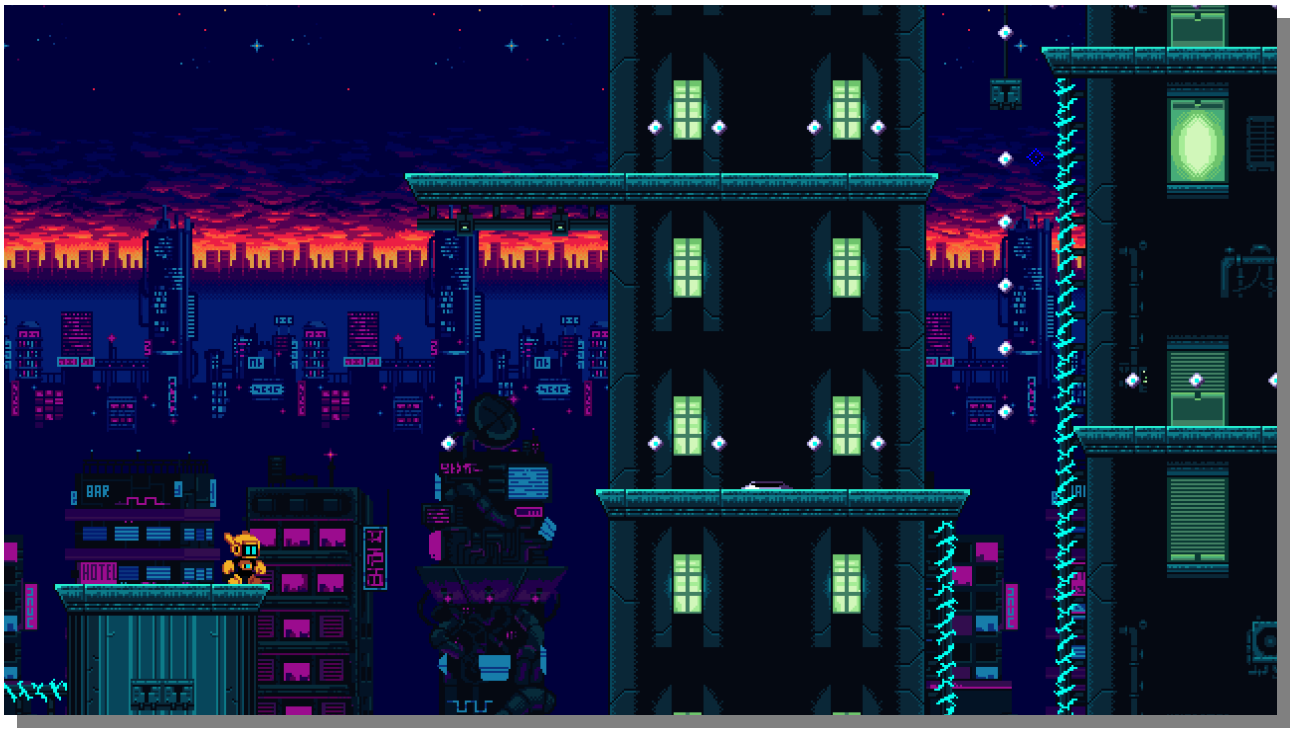


The first section of this level is simply there to introduce the player to the basic platforming mechanics. Coins are used to guide the player to where they need to go, and very simple and basic jumps are given, as to allow the player to quickly get used to the generic platforming mechanics.



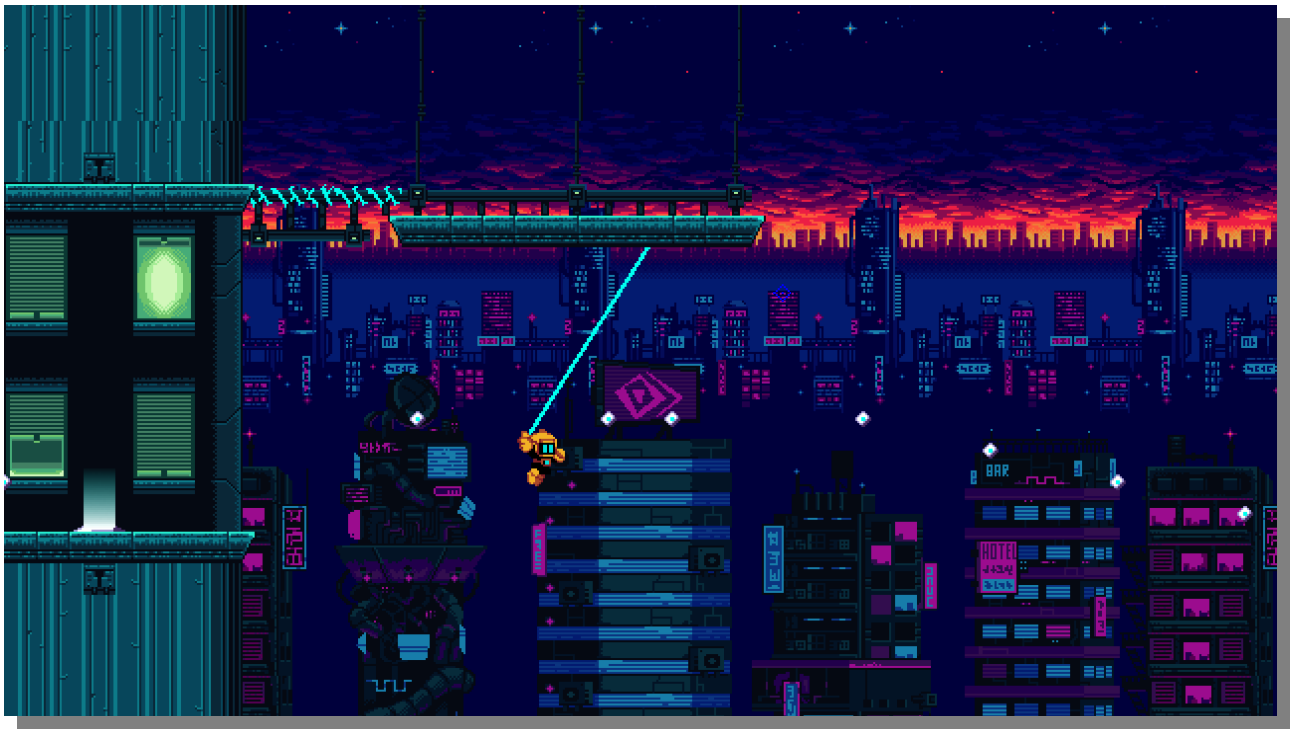
From there, a checkpoint is given, while the first obvious hazard is introduced to the user. This isn't there to provide a challenge, as getting past it is relatively simple, but rather to demonstrate to the user that the game isn't just a basic platformer; if they attempt to reach the next platform with just a jump, they will fail. To encourage the player to try other things, a single block is placed around some coins, as to draw attention to it – the shape of the coins should subtly indicate that this block can indeed be grappled onto.

However, it is integral that the player understands that this is not the only type of block that can be grappled on to, thus the section immediately after requires the player to grapple onto a different tile. This should, hopefully, indicate that majority of tiles can be used with the rope.



Afterwards, another hanging block is clearly visible. This is to demonstrate the rope extension and contraction mechanics, as all previous swings were possible without them. The player must grapple onto the block and bring themselves upwards.

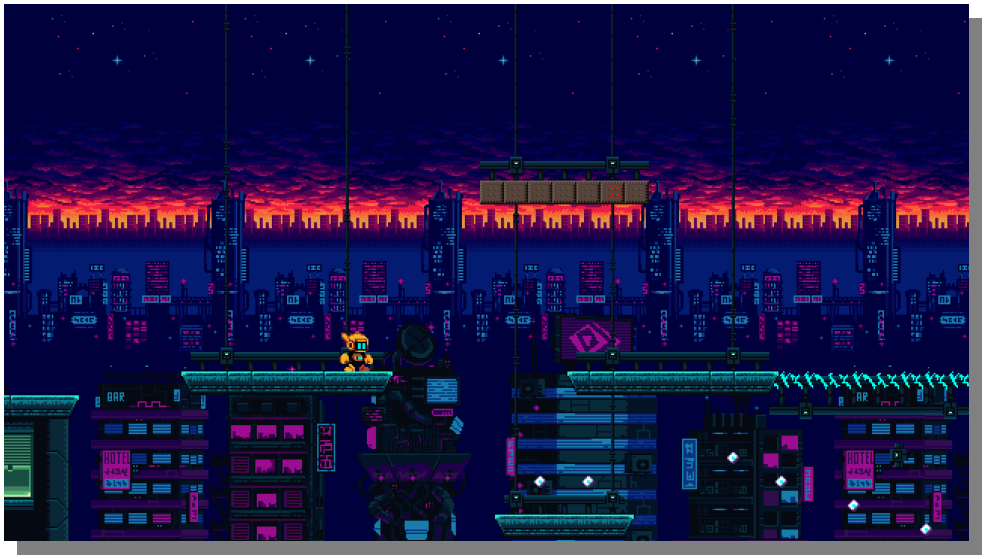
Once again, more coins are used as a way to guide the player on where to go after. For example, the player must jump on the hanging block to progress onwards; therefore, there is a coin on top of said block to guide the player to do this.



Once again, a different platform is used to demonstrate where the rope can be used. In this instance, the player needs to perform multiple swings to traverse across. Finally, the player needs to propel themselves forwards at the end of the platform, which is indicated by the coin trail once-more. This should cover all the main mechanics for the rope. Thus, the level ends.

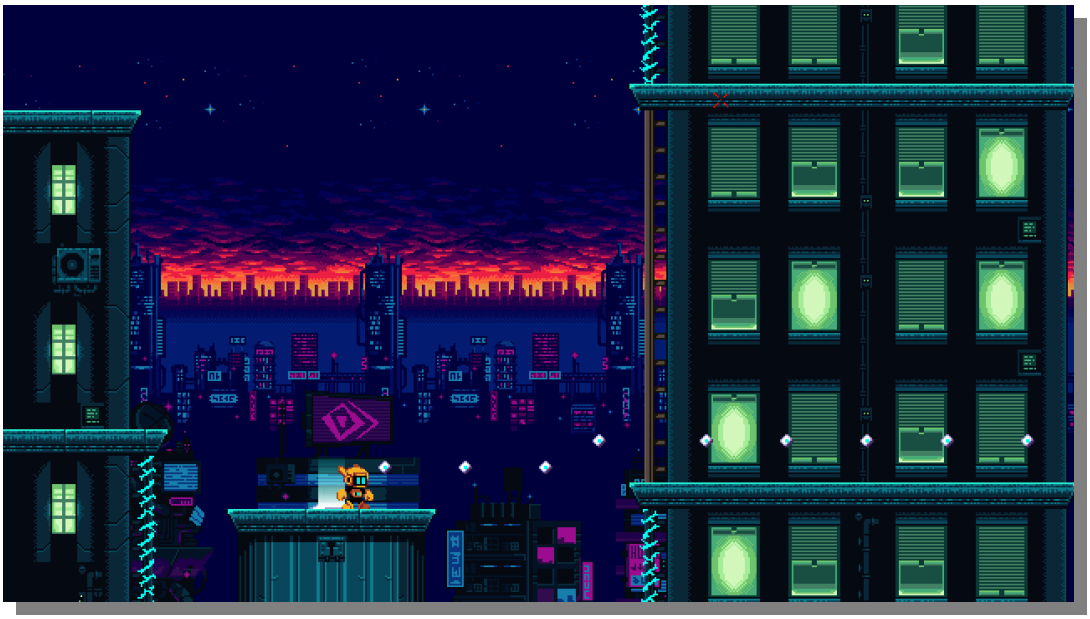
Level #2

As with the first level, the second level will focus on one specific portion of the gameplay: the “non-ropeable” tiles. There’s two types: one that the player collides with, and one that the player can go through.

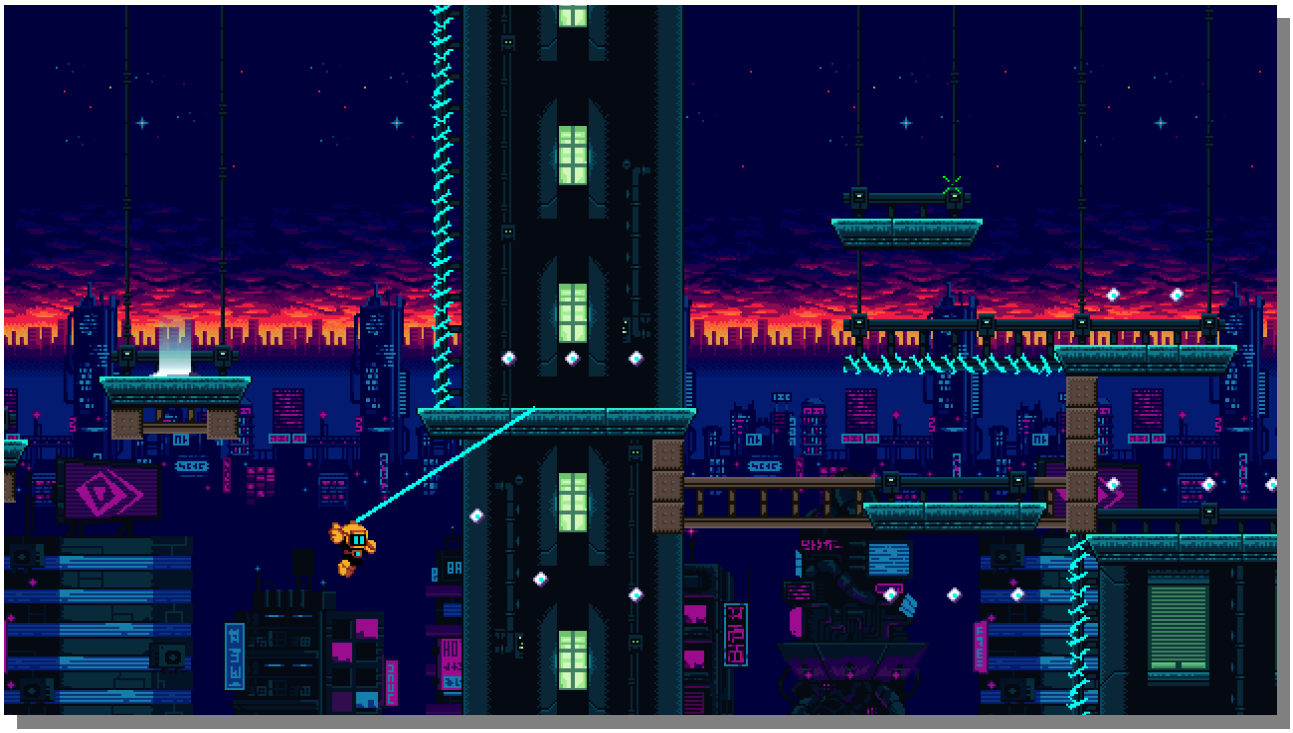


The first section of the level starts with (seemingly) two paths. This is to encourage the player to try and rope on to the top section. If they do this, two things happen: the cursor changes to red (now re-designed to not violate the Geneva conventions!), and the player cannot rope if they attempt to. This demonstrates the non-ropeable mechanics.

The next portion begins with a maximum size jump (i.e., a jump that is as big as it can possibly be). This is to encourage the player to grapple onto the building instead – however, if they decide to do this, they will quickly realise this is not possible, along with the cursor being red. With no other option, the player can only jump forwards, where they will then learn that you do not collide with these tiles.



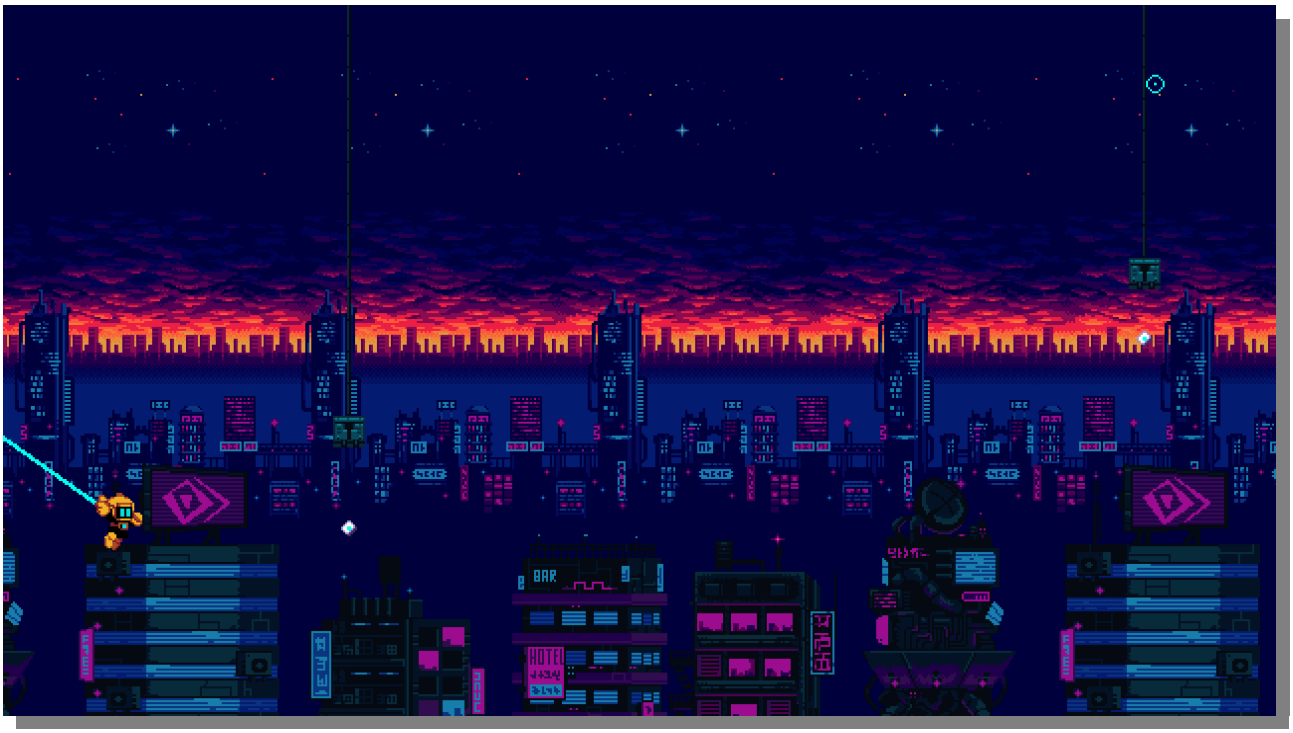
Afterwards, there is a more complicated section involving the tiles. This clearly shows an essential part of the level-design: a concept is introduced in a safe(r) and easy environment first, then a more difficult and complicated section involving said mechanic comes afterwards.



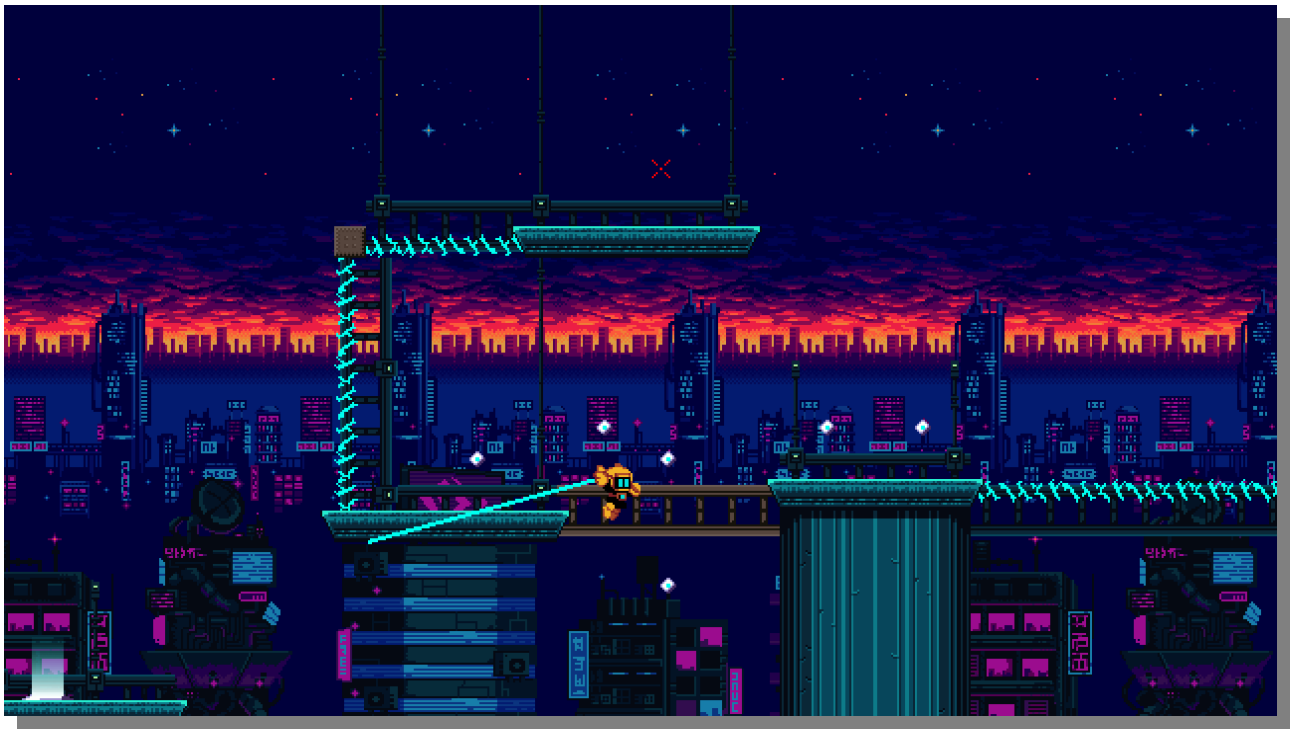
Finally, the level ends with a simple (but far) rope swing to the endpoint.

Level #3

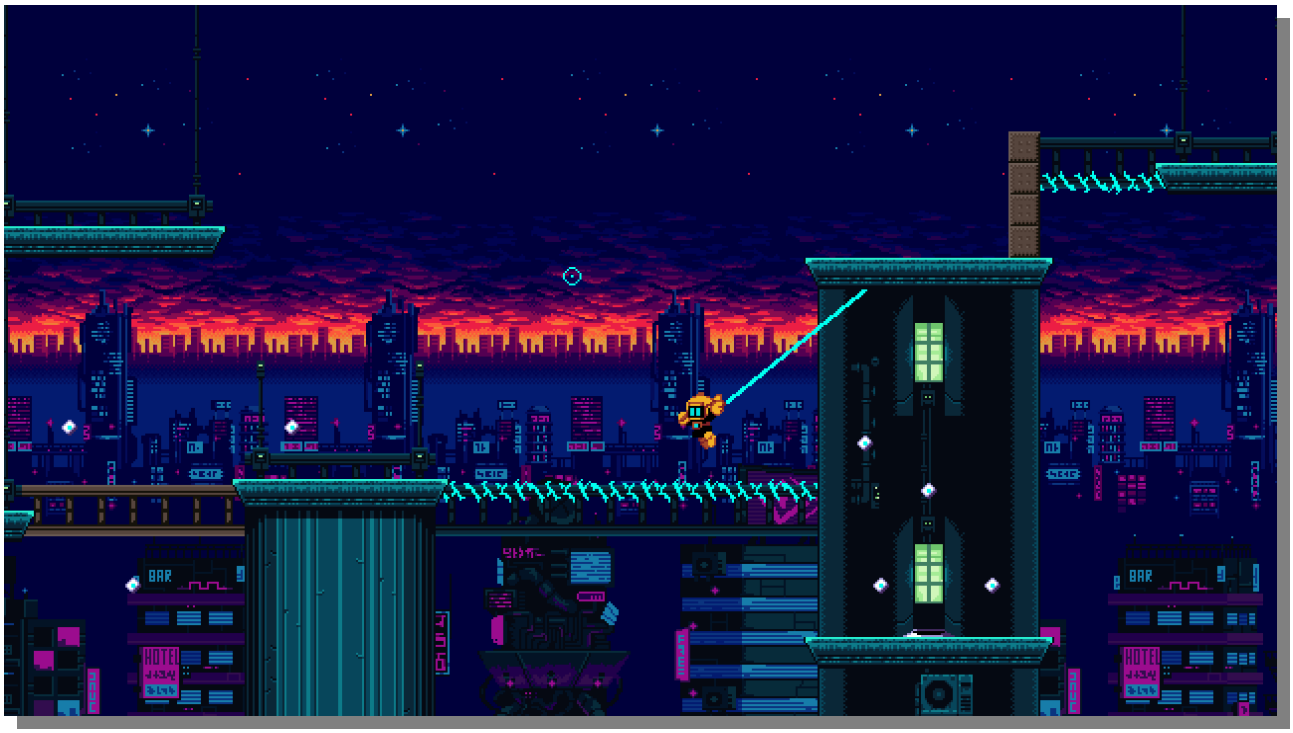
The third level will take the mechanics introduced in the past two “tutorial-esque” levels, and create a complicated environment for the player.



The first section contains some lengthy rope-swings, many of which cannot be made without “ungrappling” first, further increasing the difficulty compared to the past levels.

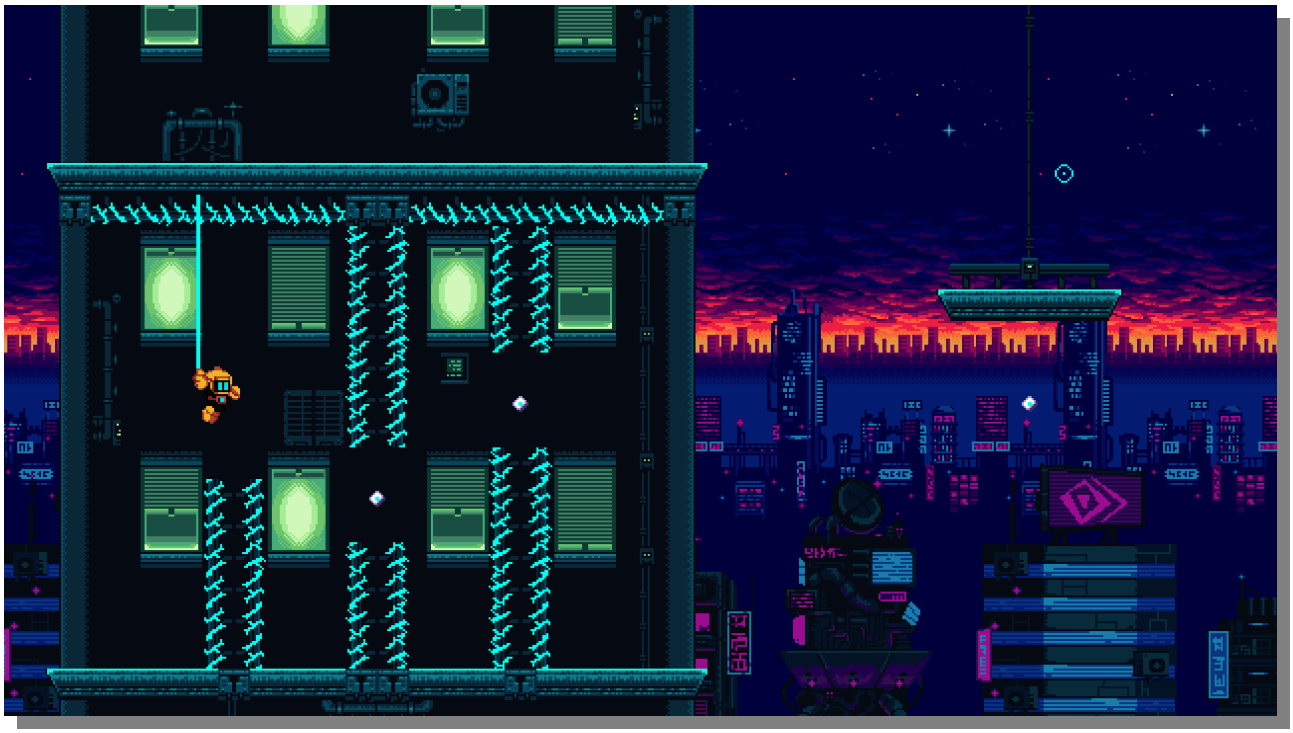


The next section further develops the non-ropeable tiles, with the player having to fully use the extent of the rope control to get through.



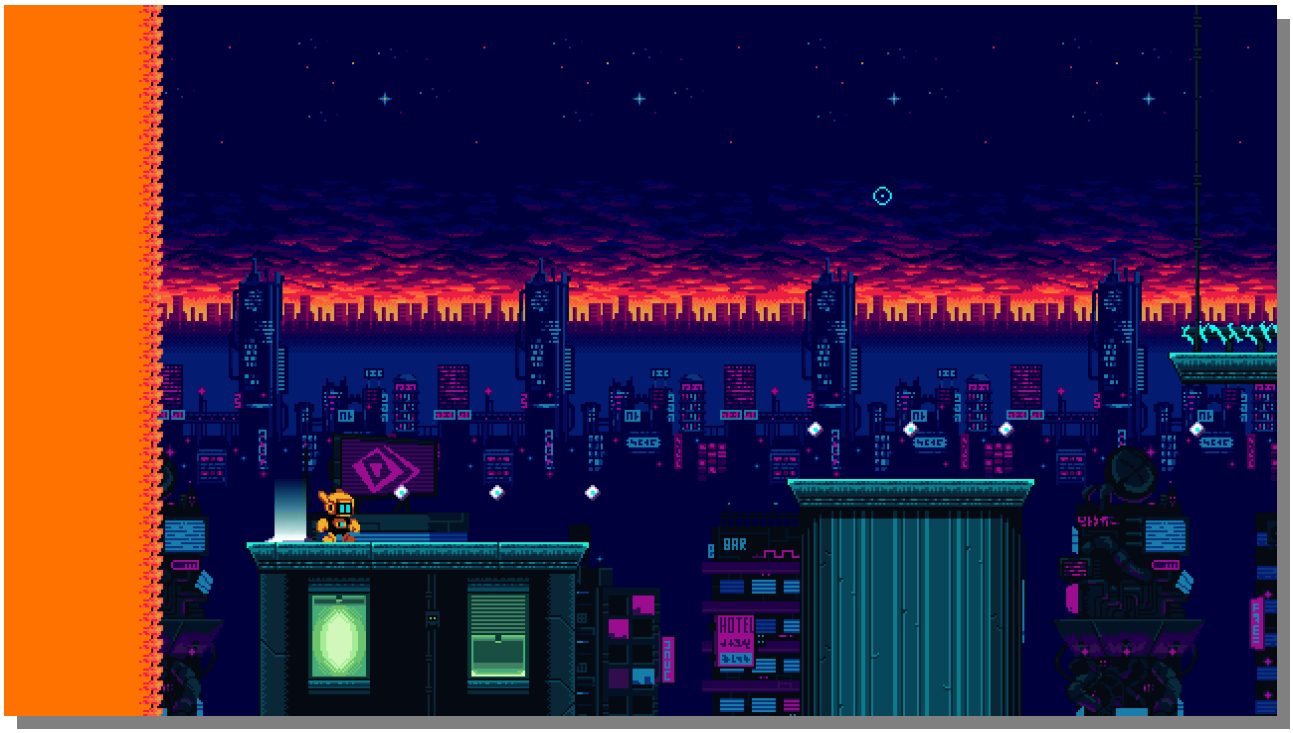
From there, the player has to make it through a small gap, requiring the use of contracting the rope mid-air.

Afterwards, the player has to use the platforming mid-air control to navigate through a small area.



Finally, the player has to slowly navigate through, using the rope extending and contracting ability in a more difficult and subtle way.

Level #4



This level involves running from a wall of fire. Why? Felt like it. There's not much technical things to this level, but it's extremely difficult and fast-paced.

3.4 Evaluation

3.4.1 Testing to Inform Evaluation

Main Menu

The following tests the functionality of the main menu – the first screen the user encounters after the game has loaded.

No.	Requirement	Input	Expected Outcome	Met
1	The highlighted menu option changes	Valid: up arrow key, down arrow key, "W" key, or "S" key Invalid: "X" key	Valid: the menu option is highlighted red Invalid: the highlighted menu option stays the same	✓
2	The highlighted menu option changes to where the mouse cursor is	Valid: mouse movement Invalid: mouse scroll	Valid: the menu option under the mouse cursor is highlighted red Invalid: the highlighted menu option stays the same	✓
3	If the highlighted menu option is "New Game", and is selected, the game loads the first level	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the first level Invalid: the game stays on the menu screen	✓

4	If the highlighted menu option is “Continue”, and is selected, the game loads the saved level	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the saved level Invalid: the game stays on the menu screen	✓
5	If the highlighted menu option is “Options”, and is selected, the game loads the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the menu screen	✓
6	If the highlighted menu option is “Exit”, and is selected, the game exits	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to black and exits Invalid: the game stays on the menu screen	✓

Options Menu

The following tests the functionality of the options (and controls) menu – this can be accessed through the main menu or pause menu.

No.	Requirement	Input	Expected Outcome	Met
7	If the highlighted options menu button is “Back”, and is selected, the game reverts back to the main menu / pause menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the main menu or pause menu Invalid: the game stays on the options menu	✓

8	If the highlighted options menu button is “Fullscreen”, and is selected, the game toggles fullscreen	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game toggles whether it is fullscreen or not Invalid: the game remains in its fullscreen / windowed state	✓
9	If the highlighted options menu button is “Controls”, and is selected, the game loads the controls menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the controls menu Invalid: the game stays on the options menu	✓
10	If the master volume slider is highlighted, and the value has changed, change the master volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the master volume is changed Invalid: the master volume stays the same	✓
11	If the music volume slider is highlighted, and the value has changed, change the music volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the music volume is changed Invalid: the music volume stays the same	✓

12	If the sound effects volume slider is highlighted, and the value has changed, change the sound effects volume	Valid: left arrow key, right arrow key, “A” key, or “D” key Invalid: right mouse button	Valid: the sound effects volume is changed Invalid: the sound effects volume stays the same	✓
13	If the highlighted controls menu button is “Back”, and is selected, the game reverts back to the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the controls menu	✓
14	If the highlighted controls menu button is “Up”, and is selected, the next key pressed becomes the key-bind for moving up.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving up changes Invalid: the key-bind remains the same	✓
15	If the highlighted controls menu button is “Down”, and is selected, the next key pressed becomes the key-bind for moving down.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving down changes Invalid: the key-bind remains the same	✓

16	If the highlighted controls menu button is “Left”, and is selected, the next key pressed becomes the key-bind for moving left.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving left changes Invalid: the key-bind remains the same	✓
17	If the highlighted controls menu button is “Right”, and is selected, the next key pressed becomes the key-bind for moving right.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for moving right changes Invalid: the key-bind remains the same	✓
18	If the highlighted controls menu button is “Jump”, and is selected, the next key pressed becomes the key-bind for jumping.	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the key-bind for jumping changes Invalid: the key-bind remains the same	✓

Pause Menu

The following tests the functionality of the pause menu – this can be accessed by pressing “Escape”.

No.	Requirement	Input	Expected Outcome	Met
19	The player can pause and un-pause the game.	Valid: “Escape” key Invalid: “G” key	Valid: the game’s pause state is toggled Invalid: the game’s pause state remains the same	✓

20	If the highlighted pause menu button is “Resume”, and is selected, the game un-pauses and resumes	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game resumes and the options menu is hidden Invalid: the game stays pauses	✓
21	If the highlighted pause menu button is “Options”, and is selected, the game loads the options menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the options menu Invalid: the game stays on the pause menu	✓
22	If the highlighted pause menu button is “Exit”, and is selected, the game saves and exits back to the main menu	Valid: enter or the left mouse button Invalid: right mouse button	Valid: the game transitions to the main menu and saves the game Invalid: the game stays on the pause menu	✓

Main Game

The following tests the functionality of the actual game – including both the rope and platformer states.

No.	Requirement	Input	Expected Outcome	Met
23	If the mouse is moved, the in-game cursor also moves	Valid: mouse movement Invalid: mouse scroll	Valid: the in-game cursor sprite matches the mouse cursor’s position Invalid: the in-game cursor sprite does not move	✓

24	If the rope can grapple on to a tile, the in-game cursor turns green	Valid: mouse movement (over a tile) Invalid: mouse scroll	Valid: the in-game cursor changes to green Invalid: the in-game cursor stays the same	✓
25	If the rope will collide on a “non-ropeable” tile, the in-game cursor turns red	Valid: mouse movement (over a non-ropeable tile) Invalid: mouse scroll	Valid: the in-game cursor changes to red Invalid: the in-game cursor stays the same	✓
26	If the rope will not collide with anything, the in-game cursor turns blue	Valid: mouse movement (over no collidable tile) Invalid: mouse scroll	Valid: the in-game cursor changes to blue Invalid: the in-game cursor stays the same	✓
27	The rope grapples on to the tile underneath the mouse cursor	Valid: left mouse button Invalid: middle mouse button	Valid: the player enters the “swing” state and a rope appears Invalid: the player’s state remains the same	✓

28	The player can exit the rope state back to the normal platformer state	Valid: spacebar (while in rope state) Invalid: “M” key	Valid: the player exits the rope state and enters the normal platforming state Invalid: the player remains in the rope state	✓
29	While in the rope state, the player can move the rope clockwise	Valid: the left arrow key, or the “A” key Invalid: the “#” key	Valid: the rope turns clockwise Invalid: the rope stays where it is (or falls due to gravity)	✓
30	While in the rope state, the player can move the rope anti-clockwise	Valid: the left arrow key, or the “D” key Invalid: the “#” key	Valid: the rope turns anti-clockwise Invalid: the rope stays where it is (or falls due to gravity)	✓
31	While in the rope state, the player can extend the rope	Valid: the down arrow key, or the “S” key Invalid: the “#” key	Valid: the rope extends (until it reaches the maximum length) Invalid: the rope stays the same length	✓

32	While in the rope state, the player can contract the rope	Valid: the up arrow key, or the “W” key Invalid: the “#” key	Valid: the rope contracts (until it reaches the minimum length) Invalid: the rope stays the same length	✓
33	While in the platformer state, the player can move right	Valid: the right arrow key, or the “D” key Invalid: the “=” key	Valid: the player moves towards the right (unless colliding with a wall) Invalid: the player does not move	✓
34	While in the platformer state, the player can move left	Valid: the left arrow key, or the “A” key Invalid: the “=” key	Valid: the player moves towards the left (unless colliding with a wall) Invalid: the player does not move	✓
35	While in the platformer state, the player can jump	Valid: the spacebar, or “W” key, or up arrow key Invalid: the “=” key	Valid: the player jumps off the ground Invalid: the player stays on the ground	✓
36	The player falls if not in the rope state	N/A	The player falls to the ground, and stops descending once hit the ground	✓

37	The rope swings downwards and side-to-side (like a pendulum)	Valid: not the "A" key, "D" key, left arrow key, or right arrow key Invalid: the "A" key, "D" key, left arrow key, or right arrow key	Valid: the rope acts like a pendulum according to gravity, and swings towards the ground Invalid: the rope moves according to the previous requirements	✓
38	The player stops moving if colliding with a tile	Valid: the player is colliding with a tile Invalid: the player is not colliding with a tile	Valid: the player stops moving, but can still move in the opposite direction (away from the tile) Invalid: the player can continue moving	✓
39	The player dies if colliding with a spike	Valid: the player is colliding with a spike Invalid: the player collides with a coin	Valid: the player dies, the screen transitions to black, and 1 is added to the death count Invalid: the player collects the coin	✓
40	The player respawns at the activated checkpoint	Valid: the player dies Invalid: the player collides with a coin	Valid: the player is teleported to the checkpoint while the screen is black, then the screen transitions back Invalid: the player collects the coin	✓

41	The checkpoint activates if the player collides with it	Valid: the player collides with a deactivated checkpoint Invalid: the player collides with an activated checkpoint	Valid: the deactivated checkpoint activates, and its animation changes to a waving flag. All other checkpoints are deactivated Invalid: the checkpoint remains activated	✓
42	The player collects a coin if it collides with one	Valid: the player collides with a coin Invalid: the player collides with a spike	Valid: the coin disappears and 1 is added to the coin count Invalid: the player dies	✓

In-Game GUI

The following tests the functionality of the GUI shown whilst playing the game (i.e., excluding pause, main, and options menus)

No.	Requirement	Input	Expected Outcome	Met
43	If a value displayed in the GUI is updated, it briefly shows said value	Valid: the player collides with a spike or with a coin Invalid: the player collides with a checkpoint	Valid: the GUI appears and displays the newly updated value Invalid: the GUI stays hidden	✓

44	The player can display the GUI whenever wanted	Valid: left control key Invalid: the “Y” key	Valid: the GUI briefly shows all values Invalid: the GUI stays hidden	✓

Sounds

The following tests the functionality of sound-effects and music.

No.	Requirement	Input	Expected Outcome	Met
45	A sound effect if played whenever the player jumps	Valid: the spacebar, or “W” key, or up arrow key Invalid: the “=” key	Valid: the sound effect is played Invalid: the sound effect is not played	✓
46	A sound effect is played whenever the player dies	Valid: the player collides with a spike Invalid: the player collides with the ground	Valid: the sound effect is played Invalid: the sound effect is not played	✓

47	A sound effect is played whenever the player collects a coin	Valid: the player collides with a coin Invalid: the player collides with the ground	Valid: the sound effect is played Invalid: the sound effect is not played	✓
48	A sound effect is played whenever the player activates a checkpoint	Valid: the player collides with a deactivated checkpoint Invalid: the player collides with an activated checkpoint	Valid: the sound effect is played Invalid: the sound effect is not played	✓
49	A sound effect is played whenever the player is falling	Valid: the player is falling Invalid: the player is on the ground	Valid: the sound effect is played Invalid: the sound effect is not played	✓
50	A sound effect is played whenever the player lands on the ground	Valid: the player collides in the ground (from previously being in the air) Invalid: the player is in the air	Valid: the sound effect is played Invalid: the sound effect is not played	✓

51	Whenever a new level is started, unique music begins playing.	N/A	Each level plays its own unique music.	✓
52	While in the main menu, unique music begins playing.	N/A	The main menu has its own unique music.	✓

Video Evidence

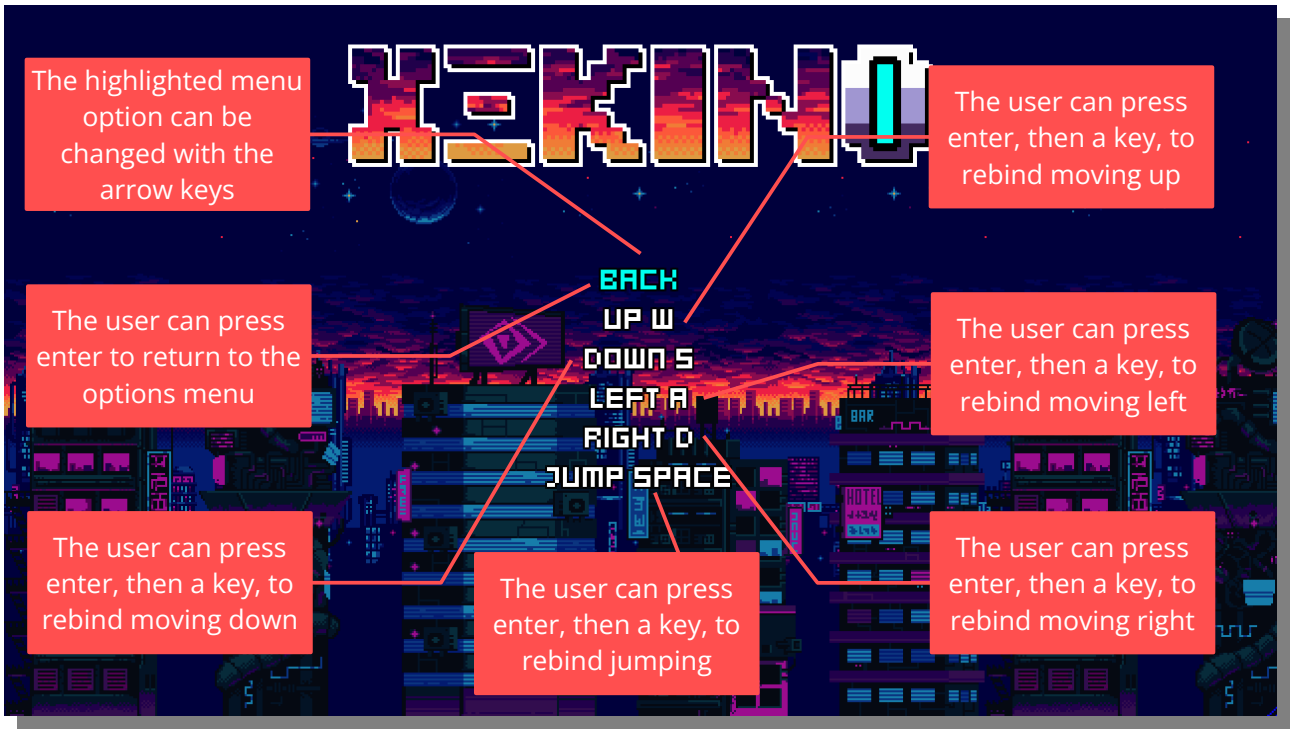
The attached video (also available on YouTube[13] with chapters) contains gameplay footage demonstrating the above criteria. The chapters are:

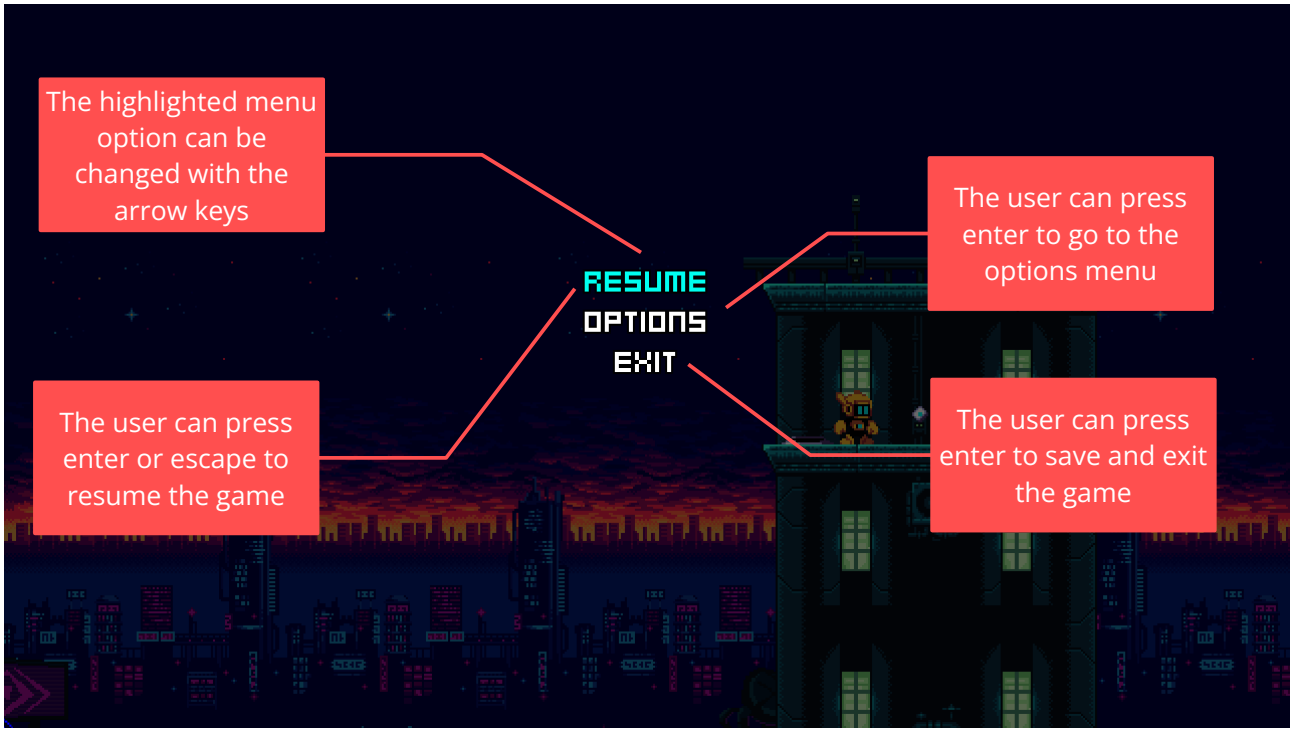
- 0:00 – Main Menu, and Options Menu
- 0:25 – Main Gameplay
  - 0:25 – Level 1
  - 1:18 – Level 2
  - 2:18 – Level 3
  - 3:47 – Level 4
- 6:22 – Ending Level / Credits
- 6:56 – Pause / Exit
- 6:58 – Save File Demonstration

13 Available here: https://youtu.be/yEeLctoIZig

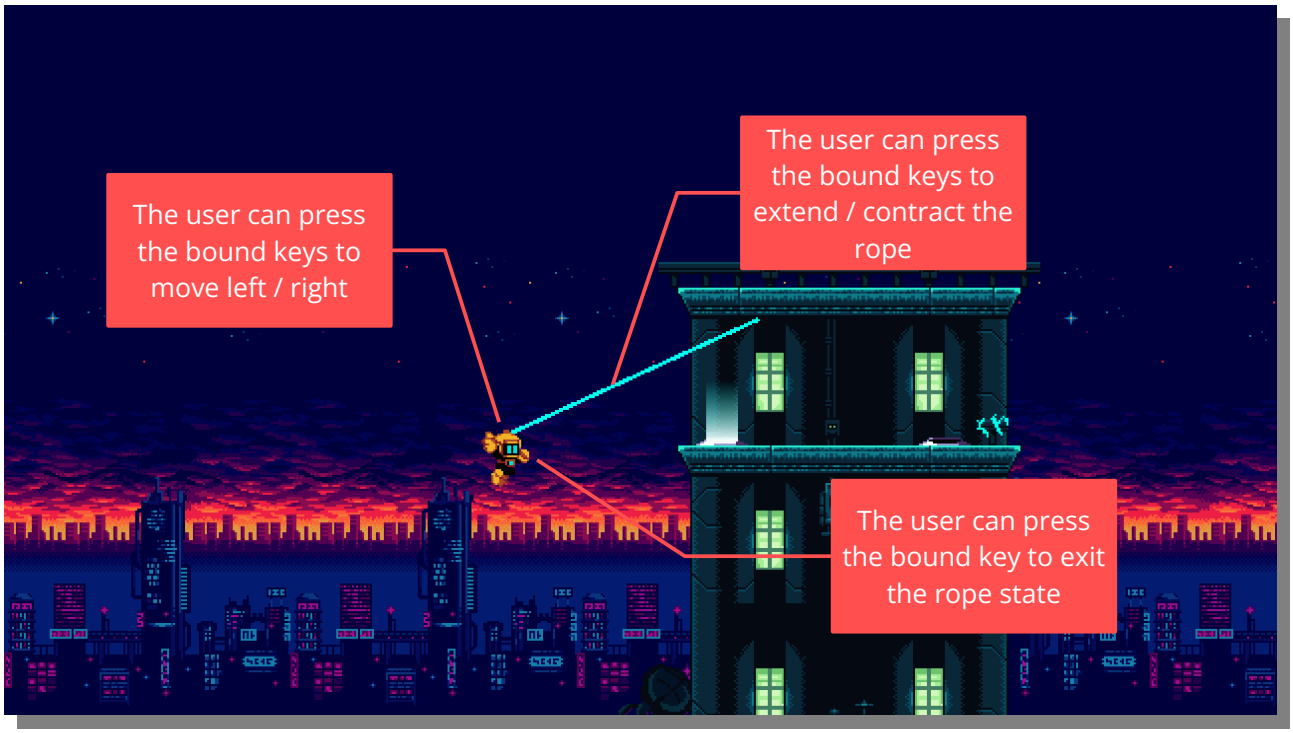
3.4.2 Usability Features

Navigation and Controls


The highlighted menu option can be changed with the arrow keys
The user can press enter, then a key, to rebind moving up
BACK
UP W
DOWN S
LEFT A
RIGHT D
JUMP SPACE
The user can press enter to return to the options menu
The user can press enter, then a key, to rebind moving left
The user can press enter, then a key, to rebind moving down
The user can press enter, then a key, to rebind jumping
The user can press enter, then a key, to rebind moving right


The highlighted menu option can be changed with the arrow keys
The user can press enter to go to the options menu
RESUME
OPTIONS
EXIT
The user can press enter or escape to resume the game
The user can press enter to save and exit the game


The user can press the bound key to jump
The user can press the bound keys to move left / right
The user can press the left mouse button to swing


The user can press the bound keys to move left / right
The user can press the bound keys to extend / contract the rope
The user can press the bound key to exit the rope state

Navigation and Controls Interview

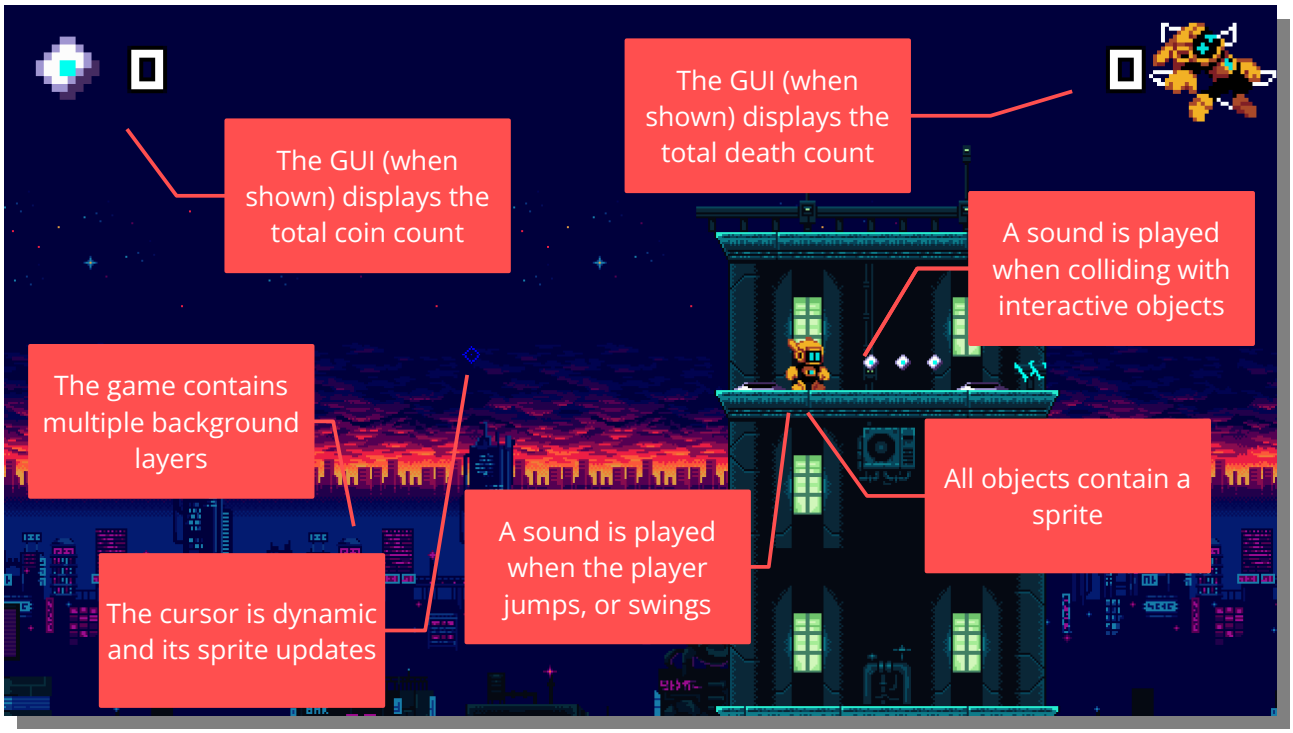
My statement:

“The stakeholder has specified that the game must be easy-to-control, allowing the player to move swiftly and precisely. This has been accomplished through the use of standard controls, as well as the ability to rebind keys to suit the user’s needs (for example if they use a different keyboard layout than QWERTY. The menus can be controlled with the keyboard or mouse, which allows the user to use whichever they prefer, as the game requires a mouse, so the user will definitely have both options.”

Onyx B.:

“The game is certainly easy-to-control, and easy-to-understand, as it uses default keys that are already commonplace. In addition, it is easy to change the keys if the user desires, which is important for the target audience, who may only own a keyboard where certain layouts feel uncomfortable. Overall, the only improvement would be if the menu controls could be rebound, as currently, it can only be controlled using the arrow keys and enter.”

GUI and Sounds



GUI and Sounds Interview

My statement:

"The GUI clearly shows the two main scores: the coin count and the death count. This only shows when the player presses the left-control button, or collect a coin / die, in order to keep the clean aesthetic. Moreover, the game contains a dynamic cursor for increased visibility and usability, as it allows the player to know if they can or cannot swing. Finally, sounds and music play to provide a more engaging and vibrant gaming experience."

Onyx B.:

"The aesthetic choices are really nice, particularly how the GUI hides when it is not necessary. This also makes it less cluttered and therefore easier to see where to go. This is important as the target audience is a variety of players, including those who have lots of or little platforming knowledge. In addition, the parallax background increases the player's immersion in-game. Finally, the sounds and music are all unique and give the game some personality. Overall, the GUI and sounds match the game's style and provide useful functionality."

3.4.3 Success of the Solution

Platforming State

No.	Requirement	Justification	Reference
1	Player can run and jump using left, right, and jump controls	Basis of the platformer genre, allowing the player to traverse the various levels	Celeste analysis
2	Jumping triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
3	Player can click on a tile to switch to the rope state	Allows the player to use the primary rope-swing gimmick	N/A
4	Switching state triggers a sound effect	Provides the player with some feedback other than visual	Interview #2

All requirements for the platforming state have been met. The player can move around using standard platforming controls, as well as change into the rope state when the mouse is over a tile that can be grappled on to.

Rope State

No.	Requirement	Justification	Reference
5	Player can swing the rope using left and right controls	Allows the player to control the main rope-swing gimmick	N/A
6	Rope is affected by gravity in a “pendulum” effect	Provides some challenge to the rope-swing gimmick, as well as adding a small sense of realism	N/A

7	Player can switch back, from the rope state, to the platformer state using the jump button	Allows the player to switch back into the normal state to make simple movements	N/A
8	Switching back to the platforming state maintains momentum	Allows the player to swing across gaps further than the rope can go	N/A
9	Switching back to the platforming state triggers a sound effect	Provides the player with some feedback other than visual	Interview #2

All requirements have been met for the rope-swing state; the player can control the rope, switch back to the platformer state, and maintain the momentum built-up during the rope swing. In addition, the pendulum effect was successfully implemented, adding a small sense of realism into the game.

Levels

No.	Requirement	Justification	Reference
10	3 unique levels	Allows the player to experience all the mechanics the game has to offer	Interview #1
11	First level should introduce mechanics to the player	Provides a safe level to allow the player to experiment with all the mechanics of the game	Celeste analysis
12	Coin collectibles can be found throughout each level	Allows for optional competition between players	Interview #2
13	Coin-count increases when the coin is collected	Allows players to compare coins collected for said competition	Interview #2

14	Coin-count is saved when the game is exited	Allows the player to continue game at a later point with their progress	Interview #2
15	Collecting a coin triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
16	Checkpoints throughout the level	Allows for larger levels, rather than many small levels, without unfair difficulty	Interview #1
17	Colliding with the checkpoint activates it and deactivates the other(s)	Allows for levels to have multiple checkpoints, rather than just one	Interview #1
18	Colliding with the checkpoint triggers a sound effect	Provides the player with some feedback other than visual	Interview #2
19	Player returns to the checkpoint if they die	This is just how checkpoints work.	Interview #1
20	Death-count increases if the player dies	Allows for optional competition between players	Interview #2
21	Death-count is saved when the game is exited	Allows the player to continue game at a later point with their progress	Interview #2
22	Dying triggers a sound effect	Provides the player with some feedback other than visual	Interview #2

All requirements for a diverse set of levels and level-obstacles have been met. In addition requirement 10 has been exceeded with 4 levels, instead of three. Once the player reaches the end of each level, they can proceed to the next one, until the end.

GUI and Visuals

No.	Requirement	Justification	Reference
23	GUI showing coin-count in the top left	Allows the player to actually view the coin count	Celeste analysis
24	GUI showing death-count in the top right	Allows the player to actually view the death count	Celeste analysis
25	Respective part of the GUI is shown when the player collects a coin / dies	Keeps the general game clean, whilst still providing important information when necessary	Celeste analysis
26	Transitions when the player enters and exits the level, dies, or the screen changes	Clearly establishes the end of the level, as well as making screen changes look nice	Celeste analysis
27	Main menu allowing the player to start the game or change options	Allows the player to launch the game without immediately being put into the game	Celeste analysis
28	Options menu allowing the player to go fullscreen and change the volume of the game	Allows the player to change any important options before the game begins	Celeste analysis
29	Pause menu, allowing the game to be temporarily paused, and allowing the player to exit back to the main menu	Allows the player to take a short break, and then resume the game from the previous place	Celeste analysis

30	Player can rebind the up, down, left, right, and jump keys	Allows the player to use any preferred keys or keyboard layouts	Celeste analysis
31	Sprites have a pixel art aesthetic	Keeps consistency throughout the game, as well as matching the retro inspired mechanics	Interview #2

All requirements for the user-interface, as well as the pixel art aesthetic, have been met. The user can use the menu system to navigate the various options, controls settings, and start / end the game. Moreover, the player can bring up a pause menu whilst playing the game, allowing them to take a short break.

Additional Successes

No.	Criterion	Justification	Reference
1	Options can be changed whilst the game is running via a menu that appears after pressing pause.	Allows the player to change any settings, most likely key-binds, after they have gathered how the functions of the game work.	Celeste analysis
2	Two tilesets: one which the rope can grapple onto, and one which cannot be grappled onto	Provides a broader set of tools to design levels with, as well as more complex level-design.	N/A
3	Controller support	Additional control methods are always a positive thing to add, particularly as it demonstrates if the game could be suitable for platforms other than PC / Mac.	Celeste analysis

4	Online multiplayer	Whilst co-operative multiplayer is unnecessary and not part of the game's scope, and local split-screen multiplayer would be too difficult to control due to the small screen size, online multiplayer would allow for multiple players on one stage with their own controls and full-sized view.	Celeste modding
5	Pixel-perfect viewport scaling	Creates a more succinct style to the game, as it will scale the game according to a pixel art style, meaning that even rotations will still line up with a pixelated grid.	N/A
6	Unique style	A non-generic style goes a long way for a game's branding and overall identity, so having something that isn't like the first level of a Mario game allows the game to stand out.	Problem identification
7	Social media integration	This allows players to share scores easily	N/A

Requirements 1, 2, 5, and 6 have all been met. This has given the game a clean and unique aesthetic that goes a long way into creating unique branding.

Unfortunately, requirements 3, 4, and 7 were unable to be met, for several reasons.

For requirement 3: using a controller is an objectively worse and harder way to play the game, and (since I do not plan on creating a console version) was not necessary to include, as there was not a significant enough reason to.

For requirement 4: online multiplayer is something that requires significant knowledge with modern networking. Moreover, I am unfamiliar with creating peer-to-peer networks and client-server networks, therefore my current skill-set is too lacking to create this feature.

For requirement 7: social media integration was not necessary to include as the game works best as a singleplayer game. It does not benefit from others publicly sharing scores and those who want to share scores can do so themselves. Ultimately, there is no need for integration, thus it was not necessary to include.

3.4.4 Maintenance and Development

The following section details any information about maintaining and continuing development of further versions of the game.

Currently Implemented Maintenance Features

Currently, there are little to no maintenance features added to the game.

For my own development and testing, the code has been designed to be organised and easy-to-navigate, thanks mostly due to the object-oriented nature of Godot, and the decomposition steps taken in the design section of this document. This means that any additional bugs to be fixed and features to be added should not be more difficult than anything previously documented.

Moreover, the game utilises source control through Git, meaning I can access the source code from multiple computers, allowing me to quickly and efficiently edit the most recent version of the game. This means that if a game-breaking bug is found, it can be swiftly fixed and an update put out, even if I was using a computer that did not have the source code on it originally. Furthermore, since Godot can run via a portable executable file, even if the computer did not have the Godot IDE installed, I could still run it without any administrator privileges. Overall, this means that maintaining the game can be done from essentially anywhere.

Current Limitations

If updates are pushed out to the game, there is currently no way to notify the user, meaning unless they check the download page before launching the game, there is a high likelihood that they will end up playing an outdated version of the game before updating. This could lead to them experiencing game-breaking bugs that have already been resolved, and requesting that they would be fixed, even though they already are in a later version.

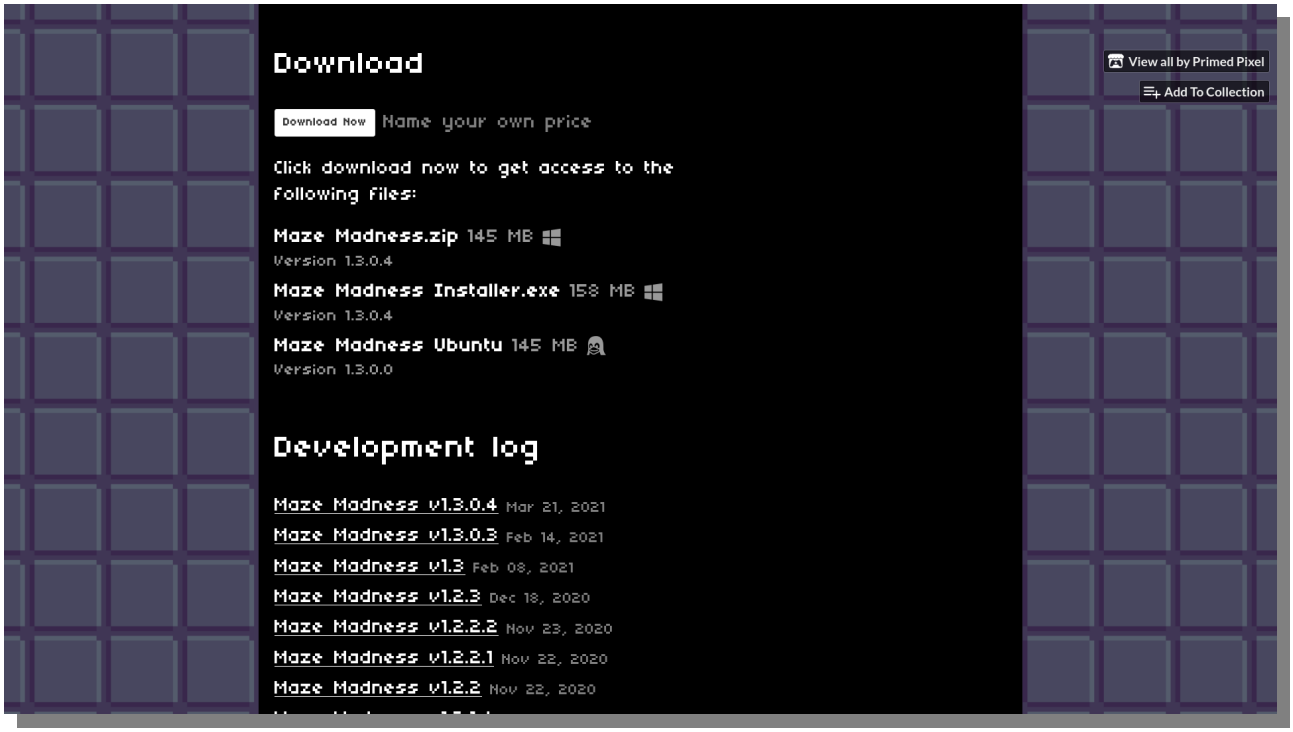
Additionally, there is no official process of contacting me about the game and its development. This means that, if a user were to run into a bug, they could not report it easily from in-game, but would have to find my contact details first, then report the bug there. This means that maintaining the game could be slow and inefficient, as not many (if any) bugs would be reported.

Furthermore, additional features, such as controller support and online multiplayer, were not thought of during development. This means that adding these (and other features) to the game would, in fact, be more difficult, as they usually require some sort of planning beforehand – particularly for any online measures. This essentially means that the majority of the current code would have to be treated instead as a prototype, and a large majority of the code would have to be re-written to accommodate such a drastic change. This, however, is not important as it is not a main goal to add such features, as discussed in the previous section.

Future Maintenance Features

Some of the limitations discussed above can be addressed, if and where possible, in order to fully maximise the potential of the game, and allow the future development of the game to be easy and efficient.

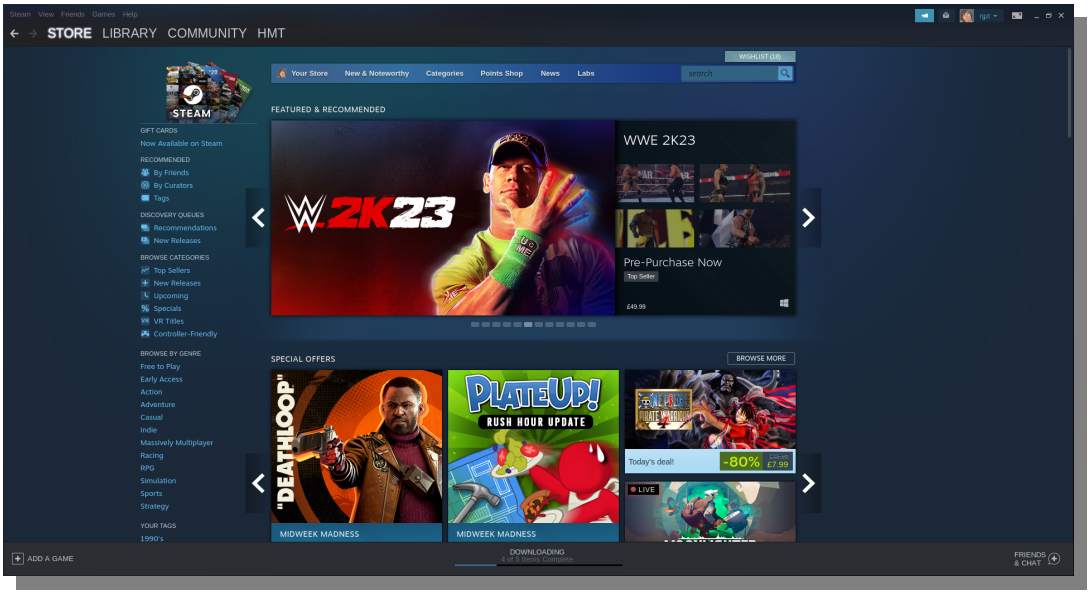
With updating, the game could utilise the itch.io API[14]. This means that, when the game opens, it could request the current game version uploaded to itch.io, and if it does not match it, the game could request that the user downloads the latest version. This is a simple way of adding an update checker.



14 The itch.io API for updating can be found here: https://itch.io/docs/itch/integrating/updates.html.

One of my previous games, “Maze Madness[15]”, uses this method. The itch.io page above contains the versions and development log for each update, and the game notifies the user if the current version is not the same as theirs.

A more thorough way of updating would be to use some third-party games service, such as Steam or Epic Games, where users must update the game before playing, if an update is available. Currently, this is not in the scope of the project, however, as submitting the game to such services usually requires a small bit of funding, which this project currently does not have.



The steam client has an automatic update service where users can automatically download the latest versions of installed games.

15 Maze Madness can be found here: https://primedpixel.itch.io/mazemadness

In addition, the itch.io page for the game could include a contact email address, which ensures that the player knows how to contact me for any bugs and issues.

A more thorough way of the user contacting me would be to include a hyperlink in the game beginning with “mailto”. This means that if the user clicked on the hyperlink, their mail application of choice would open, with the recipient field already being filled with my email address. This ensures that the user has an easy way to report bugs.

An even more thorough way of implementing this would be a built-in form inside the game, which the user could fill in and get automatically sent to me, without the user having to use any external applications, such as email. This ensures that all bugs, that need to be reported, would be, as many users would fill in a small bug report if it is easy to do so, and does not inconvenience them.

Appendices

Code Listings

Code screenshots can be found below. The project files are available on GitHub: https://github.com/PrimedPixel/ProgrammingProject.

Screenshots are of all code files (.gd files), and listed in alphabetical order.

Cam.gd

```
extends Camera2D

onready var player = get_parent().get_node("Player")
onready var viewpoint_container = get_parent().get_parent().get_parent()
onready var viewport = get_parent().get_parent()
onready var level_bottom = viewport.get_child(0).bottom

var mouse_pos = Vector2.ZERO

const min_interpolate_val = 2
var interpolate_val = min_interpolate_val

var game_size = Vector2(640, 360)
onready var window_scale = (OS.window_size / game_size).x
onready var actual_cam_pos = global_position

func _process(delta):
    #Use player's velocity as the lerp value to stop player from going off screen
    #But keep the minimum at 2 (any lower velocity will still allow camera to move)
    var player_vel = player.motion.length() * 0.03

    interpolate_val = max(min_interpolate_val, player_vel)

#   var target = player.get_global_position()
#
#   var mouse_pos = viewport.get_mouse_position() / window_scale
#
#   var mid_x = (target.x + mouse_pos.x) / 2
#   var mid_y = (target.y + mouse_pos.y) / 2

    var pos = viewport.get_mouse_position() / window_scale - (game_size / 2) + player.global_position

    actual_cam_pos = lerp(actual_cam_pos, pos, interpolate_val * delta)

    actual_cam_pos.x = max(actual_cam_pos.x, (game_size / 2).x)
    actual_cam_pos.y = clamp(actual_cam_pos.y, (game_size / 2).y, level_bottom - (game_size / 2).y)

    var subpixel_pos = actual_cam_pos.round() - actual_cam_pos

    viewpoint_container.material.set_shader_param("camera_offset", subpixel_pos)

    global_position = actual_cam_pos.round()

```

Checkpoint.gd

```
extends Area2D

export var default = false

var checked = 0
onready var animation = $CheckpointAnimation

func _process(_delta):
    if checked < 10 && default && GlobalVariables.checkpoint_pos == Vector2.ZERO:
        GlobalVariables.checkpoint_pos = position
        animation.play("Active")

        checked += 1

    if GlobalVariables.checkpoint_pos != position:
        animation.play("Inactive")

func _on_Checkpoint_body_entered(_body):
    if animation.get_assigned_animation() == "Inactive":
        SoundPlayer.play_sound(SoundPlayer.Checkpoint)
        animation.play("Active")
        GlobalVariables.checkpoint_pos = position

```

Coin.gd

```
extends Area2D

onready var sprite = $Sprite

func _process(_delta):
    sprite.offset = Vector2(0, sin(float(Time.get_ticks_msec()) / 100) * 2)

func _on_Coin_body_entered(_body):
    GlobalVariables.coin_count += 1
    SoundPlayer.play_sound(SoundPlayer.Coin)
    queue_free()    #deletes the instance

```

Cursor.gd

```
1  extends Sprite
2
3  func _process(_delta):
4      global_position = get_global_mouse_position()
5
```

DebugMenu.gd

```
1  extends Control
2
3  var can_process = false
4
5  var label = null
6
7  var level = null
8  var player = null
9  var cam = null
10
11 var spikes = null
12 var checkpoints = null
13 var coins = null
14
15 var fire = null
16
17 onready var master_bus = AudioServer.get_bus_index("Master")
18 onready var music_bus = AudioServer.get_bus_index("Music")
19 onready var sfx_bus = AudioServer.get_bus_index("Sound Effects")
20
21 func enable():
22     label = $Label
23
24     level = get_parent().get_node("ViewportContainer/Viewport").get_child(0)
25     player = level.get_node("Player")
26     cam = level.get_node("Cam")
27
28     spikes = level.get_node("Spikes")
29     checkpoints = level.get_node("Checkpoints")
30     coins = level.get_node("Coins")
31
32     fire = level.get_node_or_null("Fire")
33
34     can_process = !can_process
```

```
36 func _process(_delta):
37     visible = can_process
38 
39     if can_process:
40         var master_vol = AudioServer.get_bus_volume_db(master_bus)
41         var music_vol = AudioServer.get_bus_volume_db(music_bus)
42         var sfx_vol = AudioServer.get_bus_volume_db(sfx_bus)
43 
44         var rope_angle = fmod(player.angle_to, (2 * PI))
45 
46         label.text = "FPS: " + str(Engine.get_frames_per_second()) +
47             "\nFullscreen: " + str(OS.window_fullscreen) +
48             "\nMaster Vol: " + str(master_vol) + "db, " + str(db2linear(master_vol)) +
49             "\nMusic Vol: " + str(music_vol) + "db, " + str(db2linear(music_vol)) +
50             "\nSFX Vol: " + str(sfx_vol) + "db, " + str(db2linear(sfx_vol)) +
51 
52             "\n\nPlayer Velocity: " + str(player.motion) +
53             "\nPlayer Speed: " + str(player.motion.length()) +
54             "\nPlayer Position: " + str(player.position) +
55             "\nPlayer Global Position: " + str(player.global_position) +
56             "\nPlayer State: " + str(player.player_state) +
57             "\nPlayer Collision: " + str(player.colliding) +
58             "\nPlayer is_on_floor: " + str(player.is_on_floor()) +
59             "\nPlayer is_on_wall: " + str(player.is_on_wall()) +
60             "\nPlayer is_on_ceil: " + str(player.is_on_ceiling()) +
61 
62             "\n\nRope Pos Player: " + str(player.rope_pos) +
63             "\nRope Pos Caset: " + str(player.cast) +
64             "\nRope Length: " + str(player.rope_len) +
65             "\nRope Angle: " + str(rope_angle) + ", " + str(rad2deg(rope_angle)) + "°" +
66 
67             "\n\nCamera Position: " + str(cam.position) +
68             "\nCamera Global Position: " + str(cam.global_position) +
69             "\nCamera Interp: " + str(cam.interpolate_val) +
70 
71             "\nSpike Node Count: " + str(spikes.get_child_count()) +
72             "\nDeath Count: " + str(GlobalVariables.death_count) +
73             "\nCheckpoint Node Count: " + str(checkpoints.get_child_count()) +
74             "\nCheckpoint Position: " + str(GlobalVariables.checkpoint_pos) +
75             "\nCoin Node Count: " + str(coins.get_child_count()) +
76             "\nCoin Count: " + str(GlobalVariables.coin_count)
77 
78         if fire != null:
79             label.text += "\nFire Distance: " + str(fire.fire_distance) +
80                           "\nFire Speed: " + str(fire.fire_speed)
81 
82 
83 func _unhandled_input(event):
84     if event is InputEventKey:
85         if event.pressed:
86             match event.scancode:
87                 KEY_F3:
88                     enable()
```

DownButton.gd

```gdscript
extends Button


# Called when the node enters the scene tree for the first time.
func _ready():
    update_text()


func update_text():
    set_text(InputMap.get_action_list("button_down")[0].as_text())

```

Fullscreen.gd

```gdscript
extends Button

func _ready():
    pressed = OS.window_fullscreen


func _process(_delta):
    text = "Fullscreen: " + str(OS.window_fullscreen)

```

Game.gd

```
extends Node2D

onready var viewport = $ViewportContainer/Viewport

func _ready():
    if GlobalVariables.level_to is String && GlobalVariables.level_to != "":
        # Changes the default level in the "Game" scene to the new one
        var viewport_child = viewport.get_child(0)
        viewport_child.queue_free()

        var new_scene = load(GlobalVariables.level_to)

        # Waits until the scene has been deleted (1 frame)
        yield(get_tree(), "idle_frame")

        # Creates the new scene
        var new_level = new_scene.instance()
        viewport.add_child(new_level)

```

GlobalKeys.gd

```
extends Node

func _unhandled_input(event):
    if event is InputEventKey:
        if event.pressed:
            match event.scancode:
                KEY_R:
                    var error = get_tree().reload_current_scene()

                    if error != OK:
                        printerr("Cannot reload scene!")
                KEY_T:
                    Transition.exit_level_transition()
                KEY_Y:
                    Transition.enter_level_transition()
                KEY_F:
                    GlobalVariables.write_savegame()
                    print("Saved game!")
                KEY_F11:
                    OS.window_fullscreen = !OS.window_fullscreen

```

GlobalVariables.gd

```gdscript
extends Node

export var checkpoint_pos = Vector2.ZERO
export var coin_count = 0
export var death_count = 0

export var level_to = -1

onready var master_bus = AudioServer.get_bus_index("Master")
onready var music_bus = AudioServer.get_bus_index("Music")
onready var sfx_bus = AudioServer.get_bus_index("Sound Effects")

var save_file = File.new()
var save_file_path = "user://save.json"

func save_exists():
    return save_file.file_exists(save_file_path)

func write_savegame():
    # Checks that the file exists
    var error = save_file.open(save_file_path, File.WRITE)

    if error != OK:
        printerr("Could not open save file for writing!")
        return

    # Stores the data in a dictionary
    var data = {
        "checkpoint_position":
        {
            "x": checkpoint_pos.x,
            "y": checkpoint_pos.y,
        },

        "stats":
        {
            "coin_count": coin_count,
            "death_count": death_count,
        },

        "continue_level": level_to,

        "options":
        {
            "fullscreen": OS.window_fullscreen,
            "master_vol": AudioServer.get_bus_volume_db(master_bus),
            "music_vol": AudioServer.get_bus_volume_db(music_bus),
            "sfx_vol": AudioServer.get_bus_volume_db(sfx_bus),

            "controls":
            {
```

```
                "up": InputMap.get_action_list("button_up")[0].scancode,
                "down": InputMap.get_action_list("button_down")[0].scancode,
                "left": InputMap.get_action_list("button_left")[0].scancode,
                "right": InputMap.get_action_list("button_right")[0].scancode,
                "jump": InputMap.get_action_list("button_jump")[0].scancode,
            },
        },
    }

    # Saves the data into a JSON formatted string, then writes said string
    var json_str = JSON.print(data)
    save_file.store_string(json_str)
    save_file.close()

func load_savegame():
    # Opens the file
    var error = save_file.open(save_file_path, File.READ)

    # Checks the file is readable
    if error != OK:
        printerr("Could not open save file for reading!")
        return

    # Loads the data into a varialbe and closes the file
    var file_data = save_file.get_as_text()
    save_file.close()

    # Parses (decodes) the JSON data
    var data = JSON.parse(file_data).result

    # Applies the data to the variables
    checkpoint_pos = Vector2(data.checkpoint_position.x, data.checkpoint_position.y)
    coin_count = data.stats.coin_count
    death_count = data.stats.death_count

    level_to = data.continue_level

    OS.window_fullscreen = data.options.fullscreen
    AudioServer.set_bus_volume_db(master_bus, data.options.master_vol)
    AudioServer.set_bus_volume_db(music_bus, data.options.music_vol)
    AudioServer.set_bus_volume_db(sfx_bus, data.options.sfx_vol)

    change_key("button_up", data.options.controls.up)
    change_key("button_down", data.options.controls.down)
    change_key("button_left", data.options.controls.left)
    change_key("button_right", data.options.controls.right)
    change_key("button_jump", data.options.controls.jump)

```

```
100 func delete_savegame():
101     if save_exists():
102         var dir = Directory.new()
103         dir.remove(save_file_path)
104 
105     # Perhaps there's a way to do this without repeating?
106     checkpoint_pos = Vector2.ZERO
107     coin_count = 0
108     death_count = 0
109     level_to = -1
110 
111 
112 func change_key(map_key, new_key):
113     # Delete key of pressed button
114     if !InputMap.get_action_list(map_key).empty():
115         InputMap.action_erase_event(map_key, InputMap.get_action_list(map_key)[0])
116 
117     # Add new Key
118     var key = InputEventKey.new()
119     key.scancode = new_key
120     InputMap.action_add_event(map_key, key)
121 
122 
123 # Runs when the game first starts
124 func _ready():
125     Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)
126 
127     load_savegame()
128 
```

GUI.gd

```gdscript
extends Control

onready var coin_count = $CanvasUI/CoinUI/CoinCount
onready var death_count = $CanvasUI/DeathUI/DeathCount

onready var coin_ui = $CanvasUI/CoinUI
onready var death_ui = $CanvasUI/DeathUI

onready var coin_timer = $CanvasUI/CoinUI/CoinTimer
onready var death_timer = $CanvasUI/DeathUI/DeathTimer

#onready var player = get_parent().get_node("ViewportContainer/Viewport/Level/Player")

var previous_coin = 0
var previous_death = 0

var ui_onscreen = 0
var ui_offscreen = 0

func _ready():
    # Multiply by 6 since the UI is 6 times the scale of the in game resolution
    # (320, 180) * 6 = (1920, 1080)
    ui_onscreen = death_ui.rect_position.y * 3
    ui_offscreen = -(ui_onscreen + (64 * 3))

    coin_ui.rect_position.y = ui_offscreen
    death_ui.rect_position.y = ui_offscreen

func update_count(count_node, count_var, ui_node, timer_node, previous):
    count_node.text = str(count_var)

    if previous != count_var:
        timer_node.start()

    if !timer_node.is_stopped():
        previous = count_var

        ui_node.rect_position.y = lerp(ui_node.rect_position.y, ui_onscreen, 0.1)
    else:
        ui_node.rect_position.y = lerp(ui_node.rect_position.y, ui_offscreen, 0.1)

    return previous

func _process(_delta):

    previous_coin = update_count(coin_count, GlobalVariables.coin_count, coin_ui, coin_timer, previous_coin)

    previous_death = update_count(death_count, GlobalVariables.death_count, death_ui, death_timer, previous_death)

    if Input.is_action_pressed("button_lctrl"): # && player.is_on_floor():
        coin_timer.start()
        death_timer.start()

```

JumpButton.gd

```gdscript
extends Button


# Called when the node enters the scene tree for the first time.
func _ready():
    update_text()


func update_text():
    set_text(InputMap.get_action_list("button_jump")[0].as_text())

```

LeftButton.gd

```gdscript
extends Button


# Called when the node enters the scene tree for the first time.
func _ready():
    update_text()


func update_text():
    set_text(InputMap.get_action_list("button_left")[0].as_text())

```

LogoMenu.gd

```gdscript
extends Sprite

func _process(_delta):
    offset = Vector2(0, sin(float(Time.get_ticks_msec()) / 1000) * 2)

```

MasterVolS.gd

```gdscript
extends HSlider

onready var master_bus = AudioServer.get_bus_index("Master")

# Called when the node enters the scene tree for the first time.
func _ready():
    min_value = 0.0001
    value = db2linear(AudioServer.get_bus_volume_db(master_bus))

```

Menu.gd

```gdscript
extends Control

onready var sound_player = $SoundPlayer
onready var music = $Music
onready var fade_out = $Music/FadeOut

onready var main_menu_container = $FirstMenu
onready var main_menu_container_new_game = $FirstMenu/NewGame

onready var options_container = $OptionsMenu/OptionsContainer
onready var options_container_back = $OptionsMenu/OptionsContainer/OptionsBack

func _ready():
    main_menu_container_new_game.grab_focus()


func _on_NewGame_pressed():
    Transition.exit_level_transition()
    yield(Transition, "transition_completed")

    music.fade_out()
    yield(fade_out, "tween_completed")

    GlobalVariables.delete_savegame()
    get_tree().change_scene("res://Scenes and Scripts/Game/Game.tscn")

    Transition.enter_level_transition()

func _on_Continue_pressed():
    Transition.exit_level_transition()
    yield(Transition, "transition_completed")

    music.fade_out()
    yield(fade_out, "tween_completed")

    get_tree().change_scene("res://Scenes and Scripts/Game/Game.tscn")

    Transition.enter_level_transition()

func _on_Options_pressed():
    main_menu_container.visible = false
    options_container.visible = true

    options_container_back.grab_focus()

func _on_Exit_pressed():
    Transition.exit_level_transition()

    GlobalVariables.write_savegame()

    yield(Transition, "transition_completed")

    get_tree().quit()

func _on_focus_entered():
    sound_player.play()

```

MouseCursor.gd

```gdscript
extends Sprite

func _process(_delta):
	position = get_global_mouse_position()

```

Music.gd

```gdscript
extends AudioStreamPlayer

onready var tween_out = $FadeOut

export var transition_duration = 1.00
export var transition_type = 1 # TRANS_SINE

func fade_out():
	# tween music volume down to 0
	tween_out.interpolate_property(self, "volume_db", 0, -80, transition_duration, transition_type, Tween.EASE_IN, 0)
	tween_out.start()
	# when the tween ends, the music will be stopped

func _on_FadeOut_tween_all_completed(object, _key):
	object.stop()

```

MusicVolS.gd

```gdscript
extends HSlider

onready var music_bus = AudioServer.get_bus_index("Music")

# Called when the node enters the scene tree for the first time.
func _ready():
	min_value = 0.0001
	value = db2linear(AudioServer.get_bus_volume_db(music_bus))


```

NextLevelCollider.gd

```gdscript
extends Area2D

onready var animation = $AnimationPlayer
onready var music = get_parent().get_node("Music")

# Sets up, in the Godot IDE, a space to select the levels
#export(PackedScene, FILE, "*.tscn") var target_level_path
export(PackedScene) var target_level_path

func _on_NextLevelCollider_body_entered(body):
    if body is Player:
        body.sprite.visible = false

        animation.play("End Level")

        music.fade_out()

        yield(animation, "animation_finished")

        GlobalVariables.checkpoint_pos = Vector2.ZERO

        Transition.exit_level_transition()
        yield(Transition, "transition_completed")

        var viewport = get_viewport()
        var viewport_child = viewport.get_child(viewport.get_child_count() - 1)
        viewport_child.queue_free()

        GlobalVariables.level_to = target_level_path
        var new_scene = target_level_path.instance()
        viewport.add_child(new_scene)

        Transition.enter_level_transition()

```

OptionsMenu.gd

```gdscript
extends Control

onready var options_container = $OptionsContainer
onready var options_container_back = $OptionsContainer/OptionsBack

onready var controls_container = $ControlsContainer
onready var controls_container_back = $ControlsContainer/ControlsBack

onready var first_menu = get_parent().get_node("FirstMenu")
onready var first_menu_node = first_menu.get_child(0)

var changing = false
var changing_key = ""

onready var up_button = $ControlsContainer/UpContainer/UpButton
onready var down_button = $ControlsContainer/DownContainer/DownButton
onready var left_button = $ControlsContainer/LeftContainer/LeftButton
onready var right_button = $ControlsContainer/RightContainer/RightButton
onready var jump_button = $ControlsContainer/JumpContainer/JumpButton

onready var master_bus = AudioServer.get_bus_index("Master")
onready var music_bus = AudioServer.get_bus_index("Music")
onready var sfx_bus = AudioServer.get_bus_index("Sound Effects")

# Options Menu
func _on_OptionsBack_pressed():
    first_menu.visible = true
    options_container.visible = false

    first_menu_node.grab_focus()


func _on_Controls_pressed():
    controls_container.visible = true
    options_container.visible = false

    controls_container_back.grab_focus()


func _on_Fullscreen_pressed():
    OS.window_fullscreen = !OS.window_fullscreen


func _on_MasterVolS_value_changed(value):
    if master_bus != null:
        AudioServer.set_bus_volume_db(master_bus, linear2db(value))


func _on_MusicVolS_value_changed(value):
    if music_bus != null:
        AudioServer.set_bus_volume_db(music_bus, linear2db(value))

func _on_SFXVolS_value_changed(value):
    if sfx_bus != null:
        AudioServer.set_bus_volume_db(sfx_bus, linear2db(value))
```

```gdscript
# Controls
func _on_ControlsBack_pressed():
    options_container.visible = true
    controls_container.visible = false

    options_container_back.grab_focus()


func change_key(map_key, new_key):
    # Delete key of pressed button
    if !InputMap.get_action_list(map_key).empty():
        InputMap.action_erase_event(map_key, InputMap.get_action_list(map_key)[0])

    # Add new Key
    InputMap.action_add_event(map_key, new_key)


func _on_UpButton_button_down():
    changing = true
    changing_key = "button_up"

    disable_controls_focus()

    up_button.set_text("Press Key")


func _on_DownButton_button_down():
    changing = true
    changing_key = "button_down"

    disable_controls_focus()

    down_button.set_text("Press Key")


func _on_LeftButton_button_down():
    changing = true
    changing_key = "button_left"

    disable_controls_focus()

    left_button.set_text("Press Key")


func _on_RightButton_button_down():
    changing = true
    changing_key = "button_right"

    disable_controls_focus()

    right_button.set_text("Press Key")
```

```gdscript
func _on_JumpButton_button_down():
    changing = true
    changing_key = "button_jump"

    disable_controls_focus()

    jump_button.set_text("Press Key")


func disable_controls_focus():
    up_button.focus_mode = Control.FOCUS_NONE
    down_button.focus_mode = Control.FOCUS_NONE
    left_button.focus_mode = Control.FOCUS_NONE
    right_button.focus_mode = Control.FOCUS_NONE
    jump_button.focus_mode = Control.FOCUS_NONE


func _input(event):
    if event is InputEventKey:
        if InputMap.action_has_event("ui_accept", event):
                return

        if changing:
            change_key(changing_key, event)
            changing = false

            yield(get_tree(), "idle_frame")

            up_button.focus_mode = Control.FOCUS_ALL
            down_button.focus_mode = Control.FOCUS_ALL
            left_button.focus_mode = Control.FOCUS_ALL
            right_button.focus_mode = Control.FOCUS_ALL
            jump_button.focus_mode = Control.FOCUS_ALL

            match changing_key:
                "button_up":
                    up_button.update_text()
                    up_button.pressed = false
                    up_button.grab_focus()
                "button_down":
                    down_button.update_text()
                    down_button.pressed = false
                    down_button.grab_focus()
                "button_left":
                    left_button.update_text()
                    left_button.pressed = false
                    left_button.grab_focus()
                "button_right":
                    right_button.update_text()
                    right_button.pressed = false
                    right_button.grab_focus()
                "button_jump":
                    jump_button.update_text()
                    jump_button.pressed = false
                    jump_button.grab_focus()

```

PauseMenu.gd

```gdscript
extends Control

onready var pause_container = $FirstMenu
onready var pause_container_resume = $FirstMenu/Resume

onready var options_container = $OptionsMenu/OptionsContainer
onready var options_container_back = $OptionsMenu/OptionsContainer/OptionsBack

onready var viewport = get_viewport()

func _process(_delta):
    var key_pause = Input.is_action_just_pressed("button_pause")

    # Inverts the pause state (i.e., true -> false, and false -> true)
    if key_pause:
        visible = !visible
        get_tree().paused = visible
        pause_container_resume.grab_focus()

# Pause Menu
func _on_Resume_pressed():
    visible = false
    get_tree().paused = false


func _on_Options_pressed():
    pause_container.visible = false
    options_container.visible = true

    options_container_back.grab_focus()


func _on_Exit_pressed():
    Transition.exit_level_transition()

    GlobalVariables.level_to = viewport.get_child(0).get_filename()
    GlobalVariables.write_savegame()

    yield(Transition, "transition_completed")

#   music.fade_out()
#   yield(fade_out, "tween_completed")
    get_tree().paused = false

    var error = get_tree().change_scene("res://Scenes and Scripts/Menus/Menu.tscn")

    if error != OK:
        printerr("Cannot change scene!")

    Transition.enter_level_transition()

```

Player.gd

```gdscript
extends KinematicBody2D
class_name Player

# Equivalent to macros in gamemaker
const accel = 512
const max_air_spd = 128
const max_ground_spd = 64
const term_vel = 192
const ground_frict = 0.5
const air_frict = 0.04
const grav = 200
const jump_force = 128

const max_rope_len = 250
const min_rope_len = 20

const max_rope_spd = 250

var key_jump = "button_jump"    # Spacebar is mapped to UI Select (to begin with)
var key_up = "button_up"
var key_down = "button_s"
var key_left = "button_a"
var key_right = "button_d"

# Enumerator to store player states (represetns 0, 1, or 2 respectively)
enum state {
    normal,
    swing,
    debug,
}

var player_state = state.normal

# Variables for the rope that cannot be re-initialised
var cast = -1
var rope_len = 0
var angle_to = 0
var rope_pos = Vector2.ZERO

var rope_angle_vel = 0

var offset = Vector2(9, -9)

# A vector - magnitude and direction (essentially velocity)
var motion = Vector2.ZERO

var colliding = false

var global_mouse_pos = Vector2.ZERO

var death_animation = "Die"

# onready makes sure that the nodes have been initialised and loaded into the scene
onready var sprite = $Sprite
onready var animation = $AnimationPlayer
onready var coyote_timer = $CoyoteTimer
onready var jump_buffer_timer = $JumpBufferTimer

onready var rope_cast = get_parent().get_node("Player/RopeCast")
onready var line = $RopeLine

onready var level = get_viewport().get_child(0)
onready var level_bottom = level.bottom
```

```
func horizontal_movement(x_input, delta):
    motion.x += x_input * accel * delta
    sprite.flip_h = x_input < 0

func gravity(delta):
    motion.y += grav * delta
    motion.y = clamp(motion.y, -term_vel, term_vel)

func normal_animation():
    if floor(abs(motion.x)) == 0:
        animation.play("Idle")
    else:
        animation.play("Run")

func ground_speed_modifiers(x_input):
    # Maximum horizontal speed on ground
    motion.x = clamp(motion.x, -max_ground_spd, max_ground_spd)

    # Ground friction - slowly decelerating the player (if no input)
    if x_input == 0:
        motion.x = lerp(motion.x, 0, ground_frict)

func air_speed_modifiers():
    # Maximum horizontal speed in air
    motion.x = clamp(motion.x, -max_air_spd, max_air_spd)

    # Air friction - slowly decelerating the player
    motion.x = lerp(motion.x, 0, air_frict)

func rope_angle_changes(x_input):
    # Pendulum
    if !colliding:
        var rope_angle_accel = 0.03 * cos((rope_pos - (position + offset)).angle())
        rope_angle_vel += rope_angle_accel
        rope_angle_vel *= 0.5

#   # Limits the velocity of swinging on the rope
#   if colliding:
#       rope_angle_vel = clamp(rope_angle_vel, -0.5, 0.5)
#   else:
#       rope_angle_vel = clamp(rope_angle_vel, -1.5, 1.5)

        angle_to += rope_angle_vel
    else:
        angle_to -= rope_angle_vel
        rope_angle_vel = 0

#   angle_to = angle_to + (x_input * 0.04)

    if !colliding:
        # Changes the angle of the rope based on left / right
        angle_to += x_input * 0.04
    else:
        # Inverse the change of angle while colliding
        angle_to -= x_input * 0.2

func reset_rope():
    line.clear_points()
    animation.playback_speed = 1    # Reset animation speed
    player_state = state.normal
```

```gdscript
func rope_animation(x_input):
    # Change speed of players animation depending on speed of movement in x axis
#       if abs(motion.x) > 1.5:         # Minimum animation speed
#           animation.playback_speed = abs(motion.x) * 0.005

        # Updates the rope's offset if the player's sprite has flipped
        # This makes it line up with the player's hand
        if x_input != 0 && !colliding:
            sprite.flip_h = x_input < 0

        if !sprite.flip_h:
            offset = Vector2(-9, -9)
        else:
            offset = Vector2(9, -9)

        # Adds rope points to the line
        # This has to be after the move_and_slide so the position has been updated
        line.clear_points()
        line.add_point(offset)
        line.add_point(to_local(rope_pos))

func initialise_rope():
        # Initialise rope swing
        cast = rope_cast.point

        if typeof(cast) == TYPE_VECTOR2:
            # Take properties of rope len
            rope_pos = cast
            rope_len = (position - rope_pos).length()

            if rope_len < max_rope_len:
                angle_to = deg2rad(90) - (position - rope_pos).angle()

                SoundPlayer.play_sound(SoundPlayer.Grapple)
                player_state = state.swing

func die():
    if animation.get_current_animation() != death_animation:
        animation.play(death_animation)

        SoundPlayer.stop_sound(SoundPlayer.Wind)
        SoundPlayer.play_sound(SoundPlayer.Damage)

        Transition.exit_level_transition()
        yield(Transition, "transition_completed")

        level = get_viewport().get_child(0)

        if level.get_filename() == "res://Levels/Level4.tscn":
            level.get_node("Fire").fire_distance = 0
            level.get_node("Fire").fire_speed = 0.3

            level.get_node("Music").seek(0)

        animation.play("Idle")
        death_animation = "Die"
        position = GlobalVariables.checkpoint_pos
        GlobalVariables.death_count += 1

        reset_rope()

        Transition.enter_level_transition()
```

```gdscript
func wind_noise():
    # Gets the channel in which the wind sound is playing
    var sound_channel = SoundPlayer.is_playing(SoundPlayer.Wind)
    var vel = motion.length()
    if sound_channel && SoundPlayer.get_channel_pitch_scale(sound_channel) && vel > 40:
        SoundPlayer.set_channel_pitch_scale(sound_channel, 0.6 * pow(vel / 100, 1.01))
        SoundPlayer.set_channel_vol(sound_channel, -100 / pow(4, vel / 100))

func _ready():
    if GlobalVariables.checkpoint_pos != Vector2.ZERO:
        global_position = GlobalVariables.checkpoint_pos

# Built in function from KinematicBody2D
# _physics_process causes jitter issues on !=60Hz monitors
# _process seems to eliminate this issue without any caveats?
func _process(delta):

    if Transition.rect_animation.is_playing():
        return

    var input_up = Input.get_action_strength("button_up")
    var input_down = Input.get_action_strength("button_down")
    var input_left = Input.get_action_strength("button_left")
    var input_right = Input.get_action_strength("button_right")
    var input_jump = Input.get_action_strength("button_jump")

    var input_mouse = Input.is_action_just_pressed("mouse_left")

    var x_input = input_right - input_left

    colliding = get_slide_count() != 0

    match player_state:
        state.normal:

            if x_input != 0:
                horizontal_movement(x_input, delta)

            # Add gravity to the player & stops the player from infinitely accel
            gravity(delta)

            # Animate any ground animations now that the motion has been calculated
            normal_animation()

            # True if the player is on the floor, or was in the last 0.15 seconds
            var coyote_on_floor = is_on_floor() || !coyote_timer.is_stopped()

            if input_jump:
                jump_buffer_timer.start()

            if coyote_on_floor:

                ground_speed_modifiers(x_input)

                # Allows the player to jump if the jump buffer timer has not stopped
                if !jump_buffer_timer.is_stopped():
                    SoundPlayer.play_sound(SoundPlayer.Jump)
                    motion.y = -jump_force
                    jump_buffer_timer.stop()
                    coyote_timer.stop()
```

```
            if !is_on_floor():
                air_speed_modifiers()

                # Variable jump height
                # Checks that jump button isn't pressed and moving up quickly
                if Input.is_action_just_released(key_jump) and motion.y < (-jump_force * 0.5):
                    # Halves the jump force, making the player land quickly
                    motion.y = -jump_force * 0.5

                # Sets the player's animation to jump / in air
                animation.play("Jump")
            elif SoundPlayer.stop_sound(SoundPlayer.Wind) && !SoundPlayer.is_playing(SoundPlayer.Land):
                SoundPlayer.play_sound(SoundPlayer.Land)

            if position.y > level_bottom + 32:
                die()

        state.swing:
            animation.play("Swing")

            rope_angle_changes(x_input)

            if !colliding:
                # Changes the length of the rope based on up / down
                rope_len = clamp(rope_len + ((input_down - input_up) * 2), min_rope_len, max_rope_len)
            else:
                rope_len = clamp(rope_len - (abs(input_down - input_up) * 2), min_rope_len, max_rope_len)

            # Calculates new rope position
            var position_to = Vector2(
                sin(angle_to),
                cos(angle_to)
            ) * rope_len + rope_pos

            # Calculates the movement from the current position to the new one
            motion = position_to - position

            if motion.length() > max_rope_spd:
                motion = motion.normalized() * max_rope_spd

            # Will move quicker to the target position if not colliding
            # (since KinematicBody2D slowly approaches, like a bungee chord)
            if !colliding:
                motion *= 5

            # Resets player state out of rope state
            if Input.is_action_just_released(key_jump) || is_on_floor():
                reset_rope()
        state.debug:
            motion.x = x_input * 500
            motion.y = (input_down - input_up) * 500

            if input_mouse:
                global_position = global_mouse_pos
```

```
305     # Checks to see if the player is on the floor before the move_and_slide
306     # function updates the player's position
307     var floor_before_move = is_on_floor()
308 
309     motion = move_and_slide(motion, Vector2.UP) # Moves the player node by the vector + automatically collides
310                                                 # Also returns left over motion, meaning if collided
311                                                 # It will return no movement, and stop for the next frame
312 
313     # If the player's new position is not on the floor, but it was the previous frame
314     if floor_before_move && !is_on_floor():
315         var channel = SoundPlayer.play_sound(SoundPlayer.Wind)
316         SoundPlayer.set_channel_vol(channel, -100)
317         coyote_timer.start()
318 
319     # Rope animation
320     if player_state == state.swing:
321         rope_animation(x_input)
322 
323     # Wind noise
324     wind_noise()
325 
326     if input_mouse && !is_on_floor():
327         initialise_rope()
328 
329 func _unhandled_input(event):
330     if event is InputEventKey:
331         if event.pressed:
332             match event.scancode:
333                 KEY_1:
334                     player_state = state.normal
335                 KEY_2:
336                     player_state = state.swing
337                 KEY_3:
338                     player_state = state.debug
339 
340     if event is InputEventMouseMotion:
341             global_mouse_pos =  get_canvas_transform().affine_inverse() * event.position
342 
```

Raycast.gd

```gdscript
extends RayCast2D

onready var rope_able = get_parent().get_parent().get_node("Tiles/RopeAble")
onready var non_rope_able = get_parent().get_parent().get_node("Tiles/NonRopeAble")
onready var non_rope_able_through = get_parent().get_parent().get_node("Tiles/NonRopeAbleThrough")

onready var mouse_cursor = get_tree().get_current_scene().get_node("MouseCursor")

onready var player = get_parent()

const rope_able_texture = preload("res://Dynamic Assets/Ropeable Cursor.png")
const non_rope_able_texture = preload("res://Dynamic Assets/Non-Ropeable Cursor.png")
const no_block_texture = preload("res://Dynamic Assets/No Tile Cursor.png")

var point = Vector2(0, 0)

var mouse_pos = Vector2.ZERO

# This raycast will occur every frame
# The player will read the "point" variable
# Once the left mouse button has been pressed
func _physics_process(_delta):
    point = -1

    if is_colliding():
        var tile = get_collider()
        if tile is TileMap:
            # switch statement depending on the tile type
            match tile:
                rope_able:
                    point = get_collision_point()
                    mouse_cursor.set_texture(rope_able_texture)

                non_rope_able, non_rope_able_through:
                    mouse_cursor.set_texture(non_rope_able_texture)

            if (player.global_position - get_collision_point()).length() > player.max_rope_len:
                mouse_cursor.set_texture(non_rope_able_texture)
    else:
        mouse_cursor.set_texture(no_block_texture)

# Sets the raycast to the mouse_position
# The viewport mode in Godot transforms
# The mouse position twice, so it's necessary
# To use an inverse matrix first
func _unhandled_input(event):
    if event is InputEventMouseMotion:
        mouse_pos =  get_canvas_transform().affine_inverse() * event.position

        set_cast_to(to_local(mouse_pos))

```

RightButton.gd

```gdscript
extends Button


# Called when the node enters the scene tree for the first time.
func _ready():
    update_text()


func update_text():
    set_text(InputMap.get_action_list("button_right")[0].as_text())

```

SFXVolS.gd

```gdscript
extends HSlider

onready var sfx_bus = AudioServer.get_bus_index("Sound Effects")

# Called when the node enters the scene tree for the first time.
func _ready():
    min_value = 0.0001
    value = db2linear(AudioServer.get_bus_volume_db(sfx_bus))

```

Spikes.gd

```gdscript
extends Area2D

onready var animation_player = $AnimationPlayer

func choose(list):
    return list[randi() % list.size()]

func _ready():
    randomize()
    animation_player.play("Electric")

    # Randomises the starting frame
    animation_player.advance(choose([0, 0.1]))

func _on_Spikes_body_entered(body):
    if body is Player:
        body.die()

```

Transition.gd

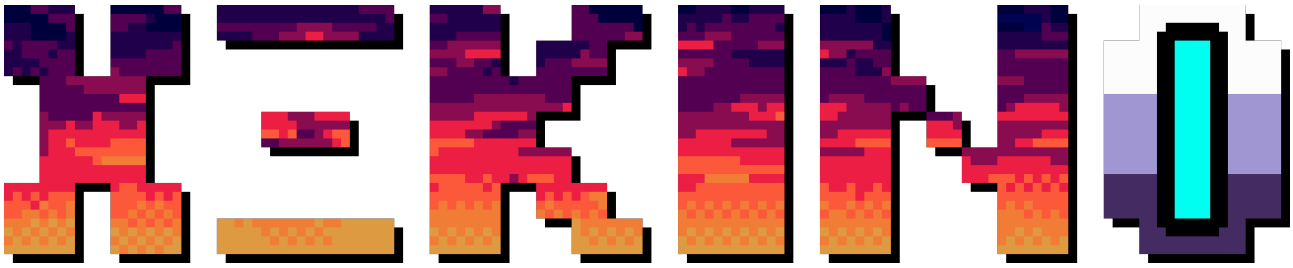
```gdscript
extends CanvasLayer

onready var rect_animation = $RectAnimation

signal transition_completed

func exit_level_transition():
    rect_animation.play("ExitLevel")

func enter_level_transition():
    rect_animation.play("EnterLevel")


func _on_RectAnimation_animation_finished(_anim_name):
    emit_signal("transition_completed")

```

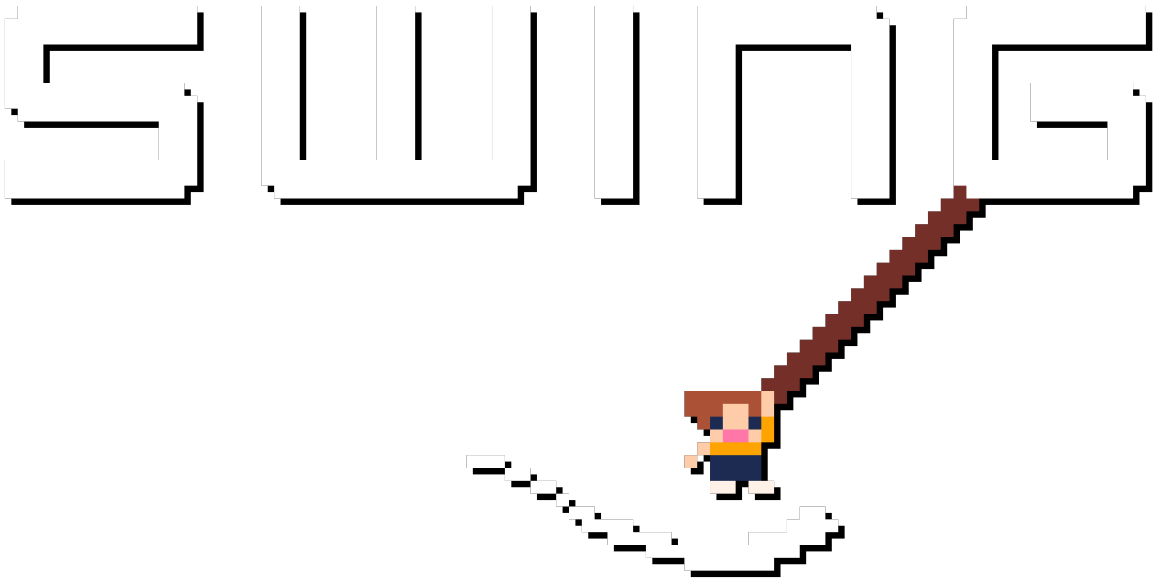
UpButton.gd

```gdscript
extends Button


# Called when the node enters the scene tree for the first time.
func _ready():
    update_text()


func update_text():
    set_text(InputMap.get_action_list("button_up")[0].as_text())

```

Brand Assets

Brand assets used for the game, including the various logos (old and new) can be found below. The currently used files are available on the previous GitHub link, under “Logos”.





Primed Pixel Games