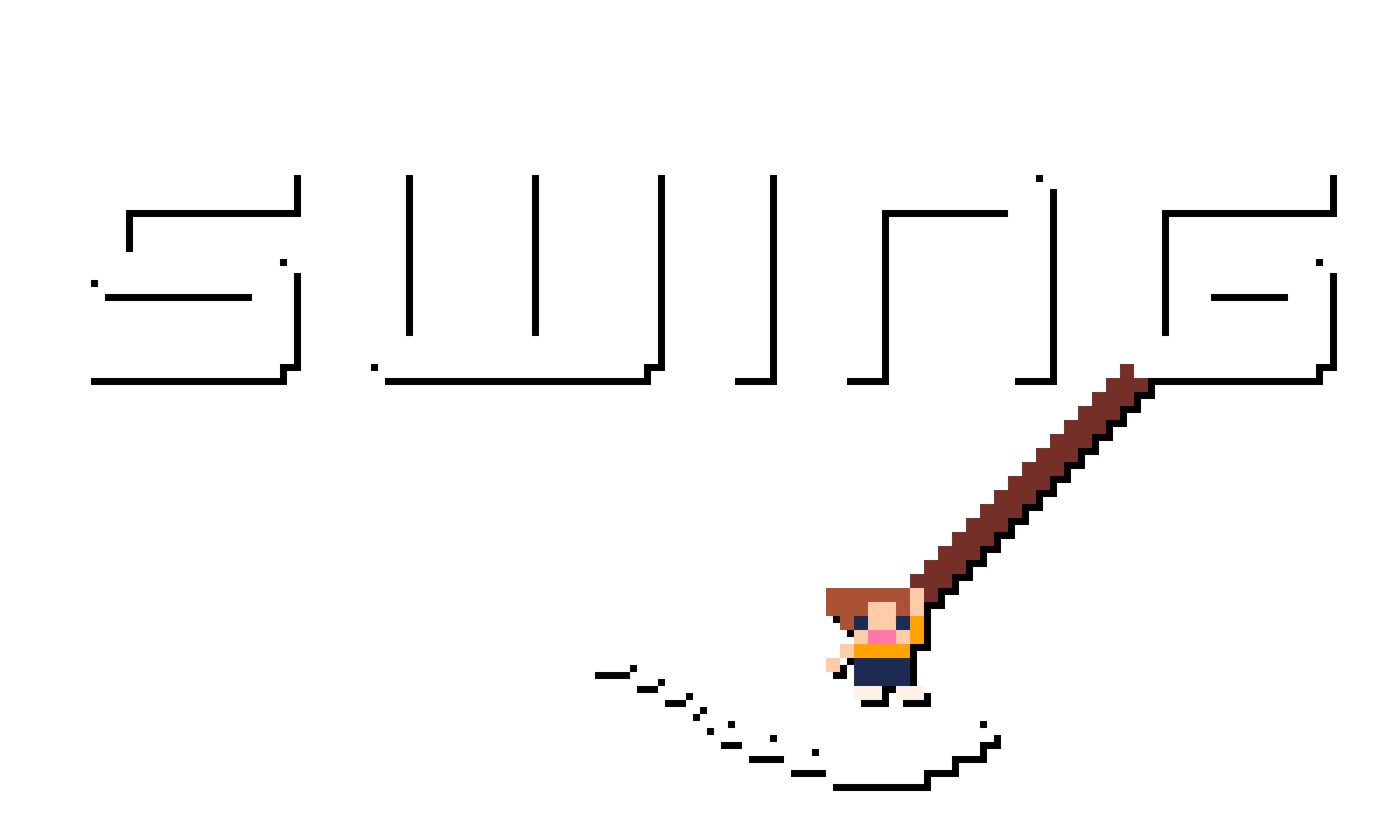
x



OCR, A-Level, Programming Project, 2021 – 2023, write-up, including analysis, design, development, and evaluation. “Rope Swing Game” is a temporary title.

Malachy Moran-Tun Candidate #6178

Contents

[3.1. Analysis of the Problem 4](#_Toc1)

[3.1.1 Problem Identification 4](#_Toc2)

[3.1.2 Stakeholders 5](#_Toc3)

[3.1.2 Solving the Problem Using Computational Methods 5](#_Toc4)

[Thinking Ahead 5](#_Toc5)

[Thinking Abstractly 6](#_Toc6)

[Thinking Concurrently 6](#_Toc7)

[Thinking Procedurally & Decomposition 7](#_Toc8)

[Thinking Logically 7](#_Toc9)

[3.1.3 Research the Problem 8](#_Toc10)

[Comparison of Retro-Styled Platformers 8](#_Toc11)

[Interview based on the Comparisons 8](#_Toc12)

[Review of the Interview 11](#_Toc13)

[3.1.5 Specify the Proposed Solution 12](#_Toc14)

[Interview based on the Review 12](#_Toc15)

[Review of the Interview – establish the main features of the game 13](#_Toc16)

[Proposed Solution 14](#_Toc17)

[Hardware Requirements 15](#_Toc18)

[Software Requirements 16](#_Toc19)

[Success Criteria 17](#_Toc20)

[3.2 Design of the Solution 18](#_Toc21)

[3.2.1 Decompose the Problem 18](#_Toc22)

[Systems Diagram 18](#_Toc23)

[3.2.2 Describe the Solution 19](#_Toc24)

[Explanation of the Systems Diagram 19](#_Toc25)

[Design of the Menu 20](#_Toc26)

[Controls 22](#_Toc27)

[Movement 23](#_Toc28)

[Level Design 26](#_Toc29)

[Scoring 28](#_Toc30)

[Pausing 28](#_Toc31)

[3.2.3 Describe the Approach to Testing 28](#_Toc32)

[3.3 Developing the Solution 28](#_Toc33)

[3.3.1 Iterative Development Process 28](#_Toc34)

[Setting Up the Player Node and Environment 28](#_Toc35)

[Adding Movement Code to the Player 29](#_Toc36)

[Adding Jumping and Gravity 30](#_Toc37)

[Interview of Features 1 32](#_Toc38)

[Raycasting from the Player 33](#_Toc39)

[Moving the Player with the Rope 34](#_Toc40)

[Allowing the Player to Control the Rope 38](#_Toc41)

[Interview of Features 2 40](#_Toc42)

[Moving the Camera with the Mouse 41](#_Toc43)

[Adding “Gravity” to the Rope 42](#_Toc44)

[Adding Animations 44](#_Toc45)

[Fixing the Raycast 44](#_Toc46)

[Interview of Features 3 46](#_Toc47)

[Adding a Background 47](#_Toc48)

[Adding Level Hazards 49](#_Toc49)

[Adding a Checkpoint System 50](#_Toc50)

[Adding Collectibles 52](#_Toc51)

[Adding a GUI 53](#_Toc52)

[Adding a Menu 53](#_Toc53)

[Adding Sounds 53](#_Toc54)

[Adding Music 53](#_Toc55)

[3.4 Evaluation 54](#_Toc56)

[3.4.1 Testing to Inform Evaluation 54](#_Toc57)

[3.4.2 Success of the Solution 54](#_Toc58)

[3.4.3 Describe the Final Product 54](#_Toc59)

[3.4.4 Maintenance and Development 54](#_Toc60)

[Appendices 54](#_Toc61)

[Code Listings 54](#_Toc62)

# 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]](#footnote-3). 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 graphics and platformers. However, thankfully, games like Celeste, Spelunky[[2]](#footnote-4), Hollow Knight[[3]](#footnote-5), Shovel Knight[[4]](#footnote-6), VVVVVV[[5]](#footnote-7), and more have brought precise, tight jumps back into the market’s interest, with more coming on the horizon, like Pokey Poke[[6]](#footnote-8) and Hollow Knight: Silksong.

Taking Celeste, for an example, it’s 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.

## 3.1.2 Stakeholders

The retro-inspired mechanics, music, sound design, and graphics create a broad target audience that, at first, may seem over-saturated in the market. 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 “stick out” 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 to develop for, 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.

## 3.1.2 Solving the Problem Using Computational Methods

Computational methods can aid in creating this game, especially in time constraints, as well as 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 below points 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 easier. 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 ( 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 & 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 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, 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

## 3.1.3 Research the Problem

### Comparison of Retro-Styled Platformers

As very often aforementioned 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, as well as their 8-bit counter parts). There is a significant difference between the 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 is 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, however, 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”.



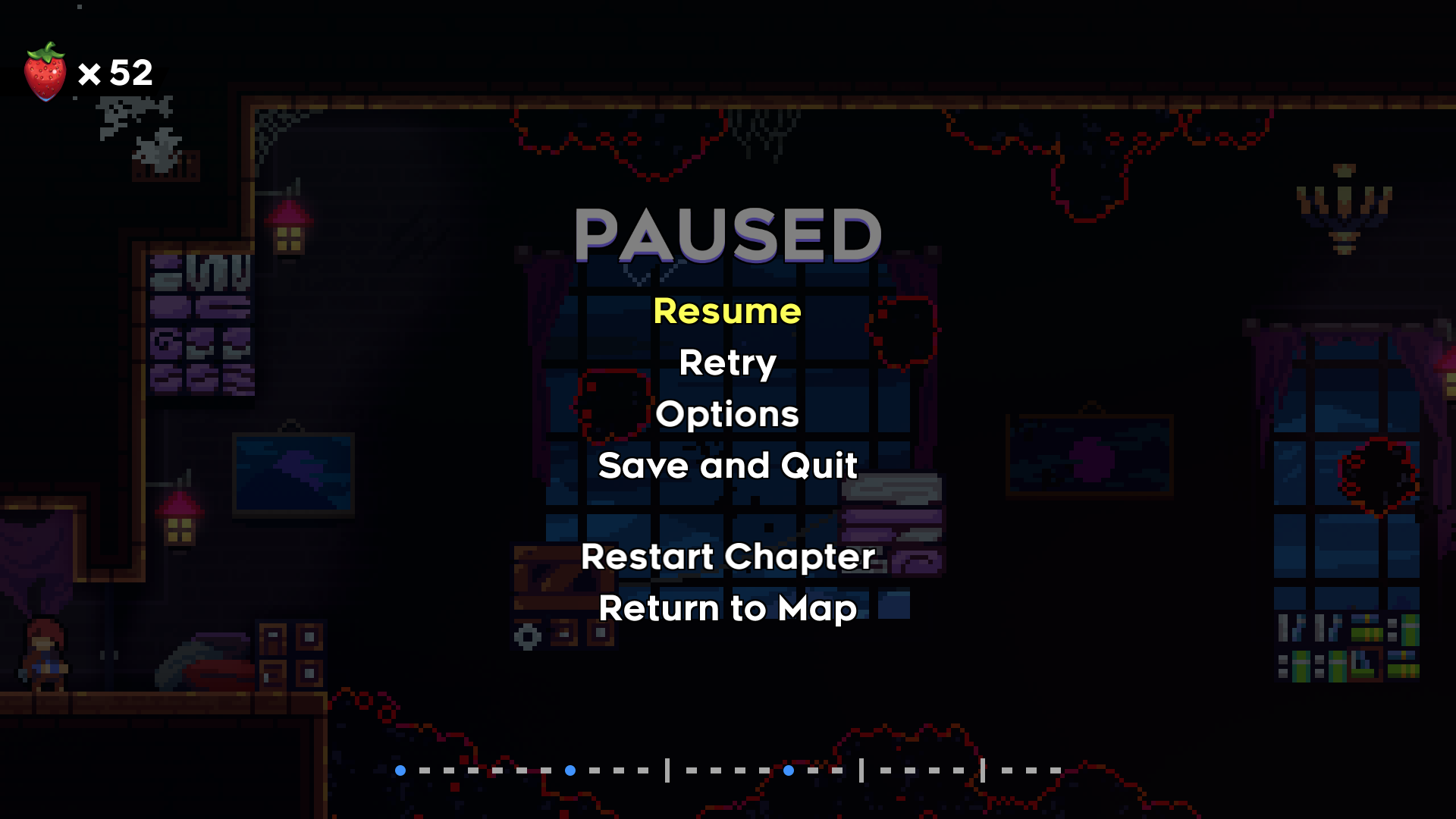
Above shows the minimum and maximum height the protagonist, Madeline, can jump. This is all controlled by the player depending on how long the jump button is pressed. The immense amount of control this gives to the player cannot be understated. Many more of these important jump mechanics can be found in a wide variety of platformer games, such as coyote time, a jump buffer, air brake, a higher down gravity etc., however 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 collectible 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 3 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 your 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, XΞKINO[[7]](#footnote-9), 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 longer how long it takes. I wouldn’t call it unfair. XΞKINO, while the unique idea behind it is just something I’ve never seen before. It really stood out to me.”

#### 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 […] 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.:

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 too.”

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 (#2)

After giving a brief summary of the main idea of the game, I asked the following questions to 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 – establish the main features of the game

##### What, not why.

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 for progression to hear, 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 as 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 grapple will “hit”, and the player’s state will be changed into the grapple state | This is a simple solution that allows for an extreme amount of precision and control for the player, since the direction of the grapple can be easily changed using the mouse, instead of being locked to 8 directions using the arrow keys (or similar) |
| 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, it is still a useful feature to have as it allows the levels to become much more 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 provides a community within the game. Crucially, these are not going to be challenges implemented *within* the game, due to time limitations, as well as demonstrating that they are just optional |
| 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 with. 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 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, and feel 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 from any place at any time. The GUI hides in order to not distract the player whilst making more difficult movements, as well as creating a much cleaner look to the game |

### 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 any more advanced hardware.

|  |  |  |
| --- | --- | --- |
| Component | Requirement | Justification |
| CPU | 2.5 GHz Dual core | 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 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 any more advanced software.

|  |  |  |
| --- | --- | --- |
| Component | Requirement | Justification |
| Operating System | Windows 7  MacOS 10.10  Linux with X11 or Wayland Desktop environment  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 | 3.3 | 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

|  |  |  |
| --- | --- | --- |
| Requirement | Justification | Reference |
| 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 |
| Jumping triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| Player can click on a tile to switch to the rope state | Allows the player to switch into the primary rope-swing gimmick |  |
| Switching state triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| Player can swing the rope using left and right controls | Allows the player to control the main rope-swing gimmick |  |
| 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 |  |
| Player can switch back from the rope state using the jump button | Allows the player to switch back into the normal state to make simple movements |  |
| Switching back to the platforming state maintains momentum | Allows the player to swing across gaps further than the rope can go |  |
| Switching back to the platforming state triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| 3 unique levels | Allows the player to experience all the mechanics the game has to offer | Interview #1 ? |
| Coin collectibles can be found throughout each level | Allows for optional competition between players | Interview #2 |
| Coin-count increases when the coin is collected | Allows players to compare coins collected for said competition | Interview #2 |
| Coin-count is saved when the game is exited | Allows the player to return to the game at a later point | Interview #2 |
| Collecting a coin triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| Checkpoints throughout the level | Allows for larger levels, rather than many small levels, without unfair difficulty | Interview #1 ? |
| Colliding with the checkpoint activates it and deactivates the other(s) | Allows for levels to have multiple checkpoints, rather than just one | Interview #1 ? |
| Colliding with the checkpoint triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| Player returns to the checkpoint if they die | This is just how checkpoints work. | Interview #1 ? |
| Death-count increases if the player dies | Allows for optional competition between players | Interview #2 |
| Death-count is saved when the game is exited | Allows the player to return to the game at a later point | Interview #2 |
| Dying triggers a sound effect | Provides the player with some feedback other than visual | Interview #2 |
| GUI showing coin-count in the top left | Allows the player to actually view the coin count | Celeste analysis |
| GUI showing death-count in the top right | Allows the player to actually view the death count | Celeste analysis |
| 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 |
| 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 ? |
| 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 ? |
| 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 ? |
| Player can rebind the up, down, left, right, and jump keys | Allows the player to use any preferred keys or keyboard layouts | Celeste analysis ? |
| Sprites have a pixel art æsthetic | Keeps consistency throughout the game, as well as matching the retro inspired mechanics | Interview #2 |
| 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 |

# 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, letting me work systematically through each separate component to create a 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 playing 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 pause the game whilst playing. A new pause menu will present options 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. In order to control the player, inputs will be taken and processed, which will then be used elsewhere in the code for running and jumping. |
| 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. |
| 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 user first launches the program, 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 of the menu design I intend to implement.

#### Main Menu



Rope Swing Game

> New Game

- Continue

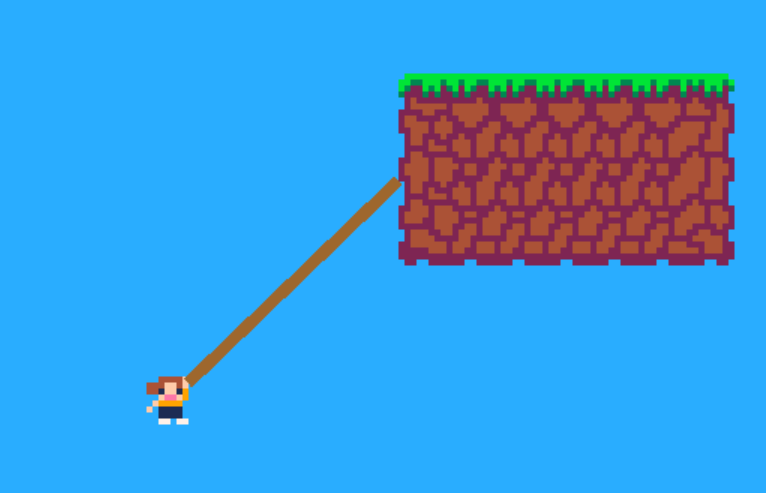
- Options

- Exit

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. 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.

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 option 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.

#### Options Menu



Options

> Mute music

- Mute sounds

- Toggle fullscreen

- Change Controls

- Back to Main 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.

The menu has a different image to demonstrate the game’s two main states: normal platforming, and rope swinging. This subtly shows the player how the levels of the game will be structured, even before even starting the game.

#### Controls Menu

CONTROLS

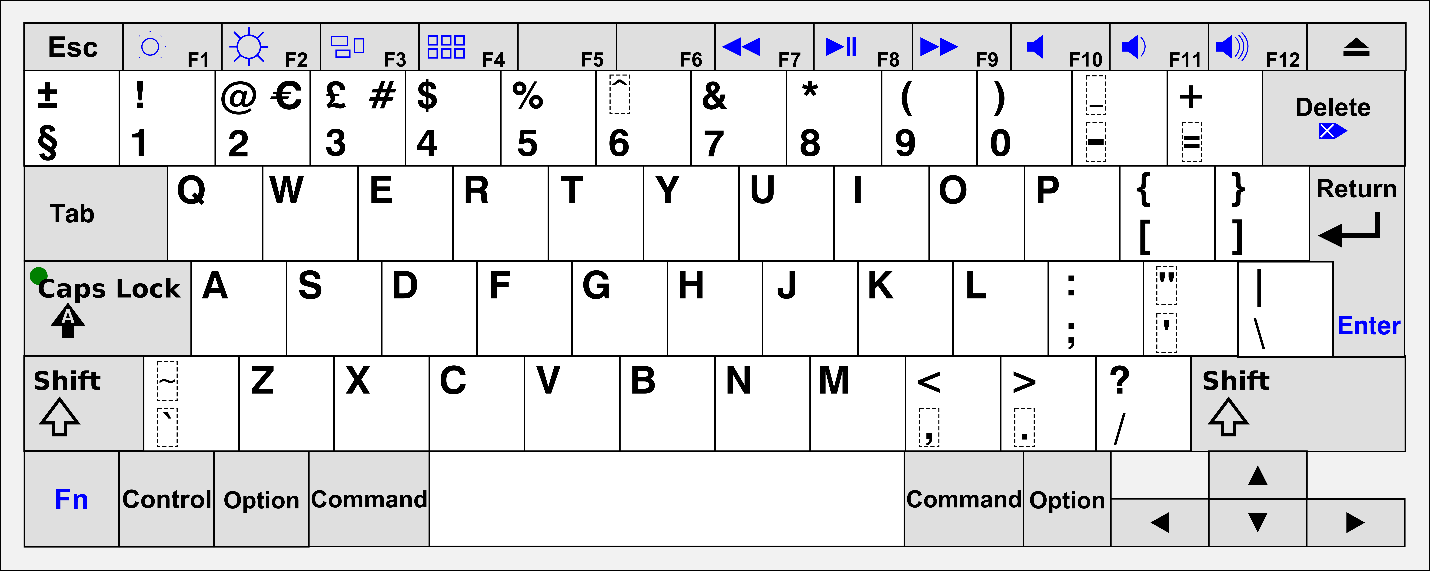
> Back to Main Menu

The design above is another mock-up of the controls menu. By pressing the enter key, the user will be able to rebind all controls, including using the left-mouse button to begin the rope swing. However, since a mouse is required, there is no additional setting to change the use of the mouse pointer, only the mouse button. The spritework is, once again, different from the two other menus in order to allow easier distinguishing between the multiple screens.

### 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 majority of players will not feel the need to modify it. If this default is poor, it will result in a poor user-experience, meaning it is essential to provide simple to understand and easy to modify controls.

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 integral 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 max speed for the player 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

##### To do: some example level designs, and the process of level design. Talk about interfacing the tutorial into the level for the user (or perhaps do that as a separate heading). Mention the level difficulty progression being a key point in the interview

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”, such as many levels seen in Celeste, or older, 1980s platformer games, such as Donkey Kong, a checkpoint system will be a suitable solution. 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 | Also to tie in with the scoring system, I plan on implementing coins, as specified in the interview stage of the project, since it will provide another layer of scoring that is not essential is players do not want to hunt for collectibles. This means that those who find trying to collect as many as possible 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, meaning there is no one way the game has to be played, broadening the target audience. |

#### 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. |
| Introduce Simplistically; Expand on Afterwards |  |

##### To do: continue this section after levels have been designed

#### 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 playtesting 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 important 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

##### To do: detailed explanation of the scoring, particularly the reasoning with reference to the interviews. Talk about the reasoning why death counts were favoured over lives, with reference to games such as Celeste.

### Pausing

##### To do: explanation of pausing and integrating it into game mechanics

Pausing is necessary to allow the player to take a break 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 unpaused, 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.

##### To do: pause menu mock-up

### \_delta\_deltaAlgorithms

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. All algorithms below are to be run at the games framerate (i.e., 60 times a second) unless otherwise stated.

#### Basic Platforming Movement

|  |  |
| --- | --- |
| Horizontal Movement | Vertical Movement |
|  |  |

#### Rope Algorithms

|  |  |
| --- | --- |
| Raycast | Rope Control |
|  |  |

|  |  |
| --- | --- |
| Pendulum Effect | Rope Line |
|  |  |

#### Camera

|  |
| --- |
| Camera Algorithm |
|  |

#### Collidables

|  |  |  |
| --- | --- | --- |
| Spikes | Checkpoint | Coins |
|  |  |  |

## 3.2.3 Describe the Approach to Testing

# 3.3 Developing the Solution

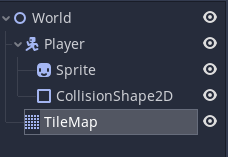
## 3.3.1 Iterative Development Process

### 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.



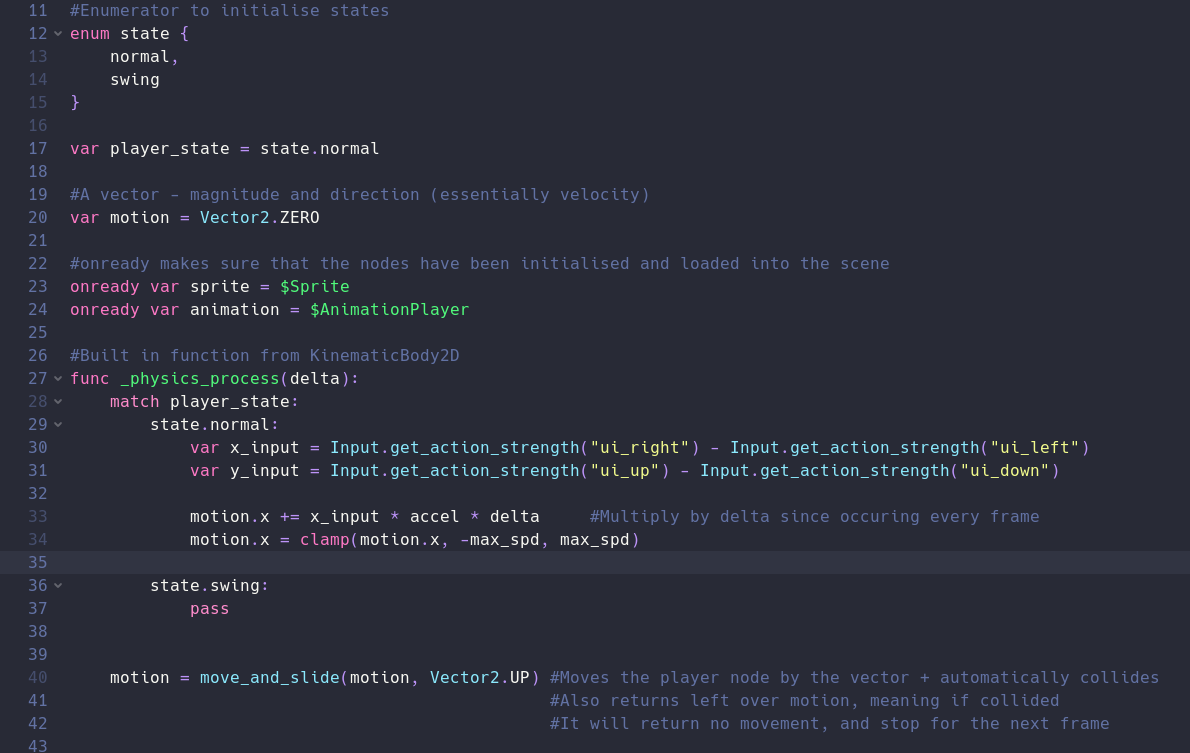
After all the initial setup, the scene has the nodes as seen to the left, as well as the player and tiles.

### Adding Movement Code to the Player

Now that the player node is setup, 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.

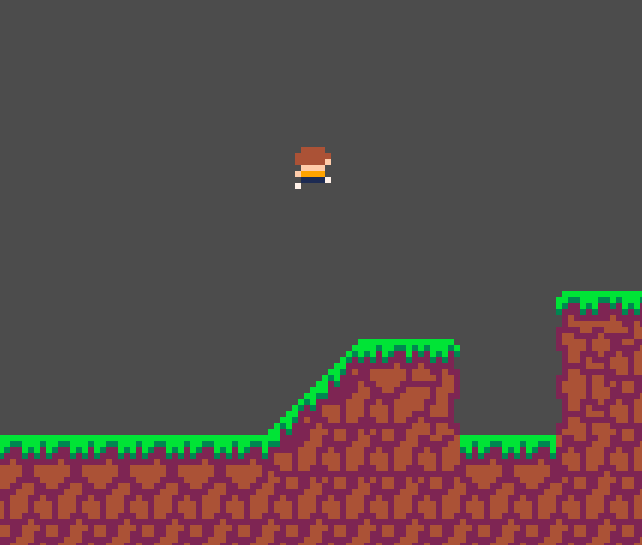
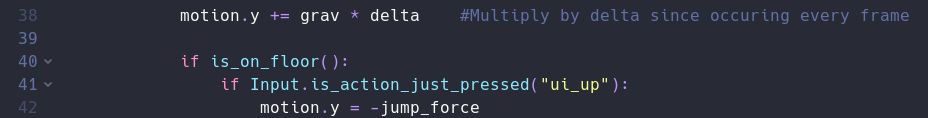
When I went to test this functionality, the player did not move. After re-examining the code, I realised that I forgot to set the player state, meaning no code was being executed. After correcting this, the behaviour works as expected, and the player can now move left and right.



### Adding Jumping and Gravity

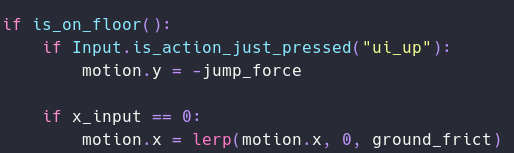
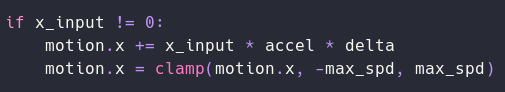
Now that my inputs and player states are working, I need to fully flesh out the platformer mechanics. To do this, I first went about adding gravity to the game, by applying a downwards vertical force to the y-value of the motion vector. Now the player does not “fly” away, depending on whatever direction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



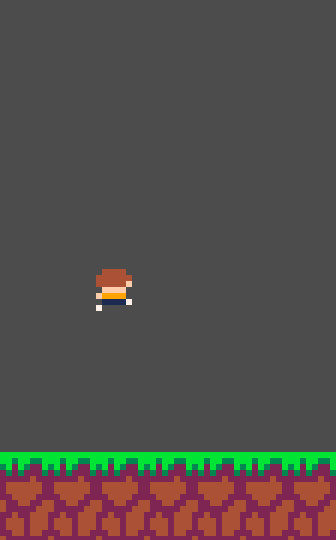
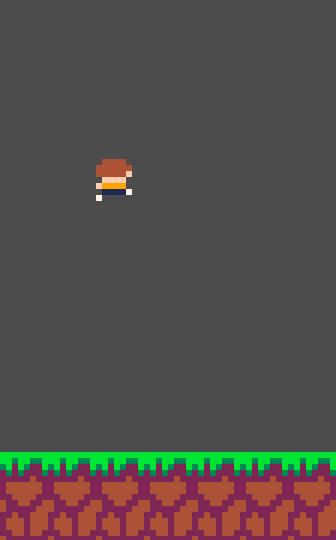
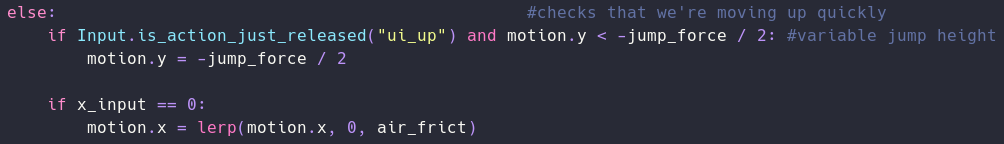
In order to test this, I used Godot’s auto-tiling feature to create a tileset that will collide with the player’s “CollisionShape2D” node and created a basic “demo” screen to allow me to test gravity and jumping.

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.



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 over 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.



Variable Jump Height

### 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?

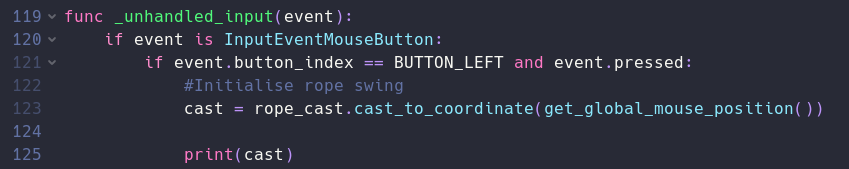
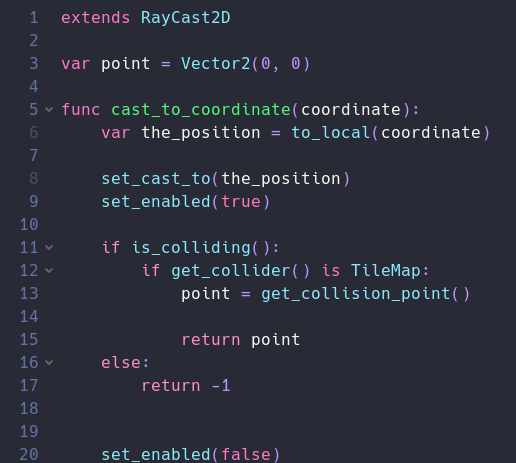
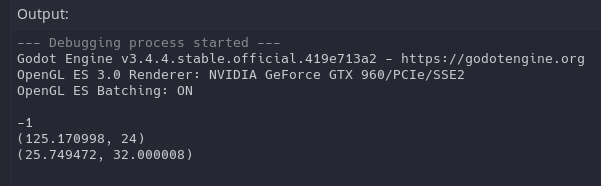
“The game now needs the rope mechanics, as well as some small animation for the player, as currently, the player sprite does not change, which would provide more feedback to the user.”

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 use the location of the mouse as an argument in the function. It would then print this location to the output.

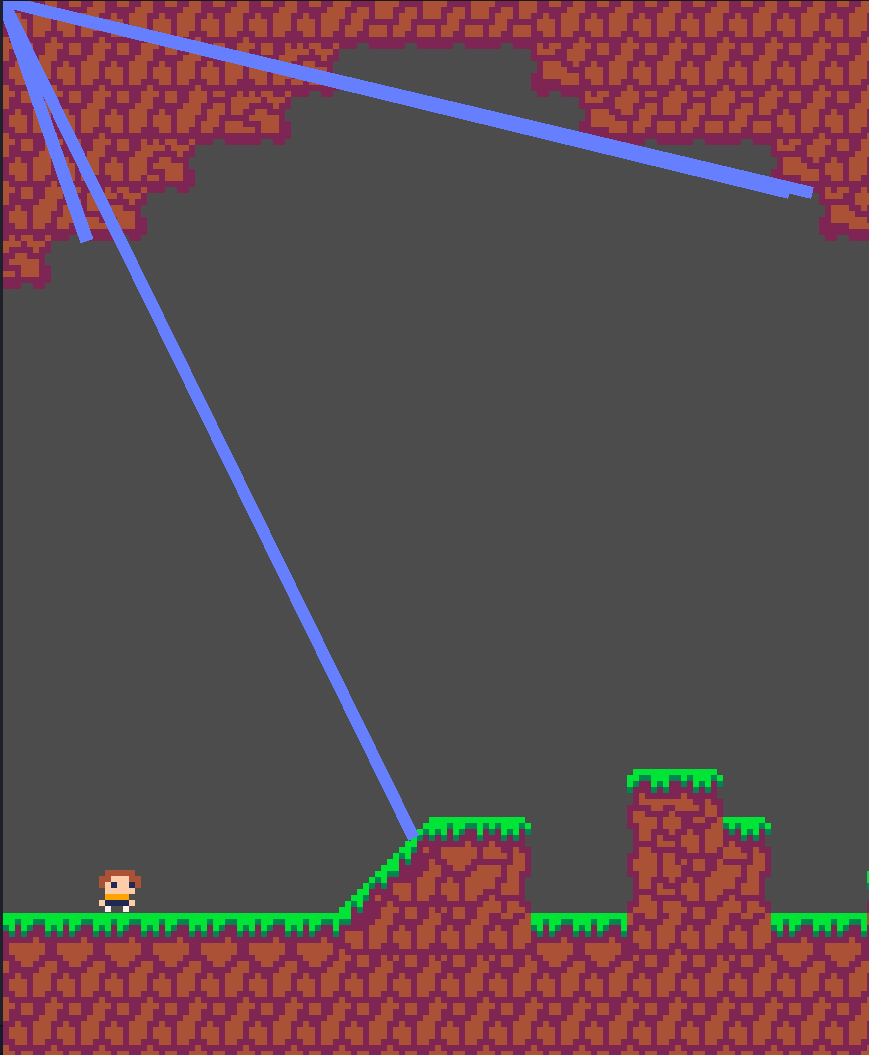


Thanks to the output, I now knew that the raycast was:

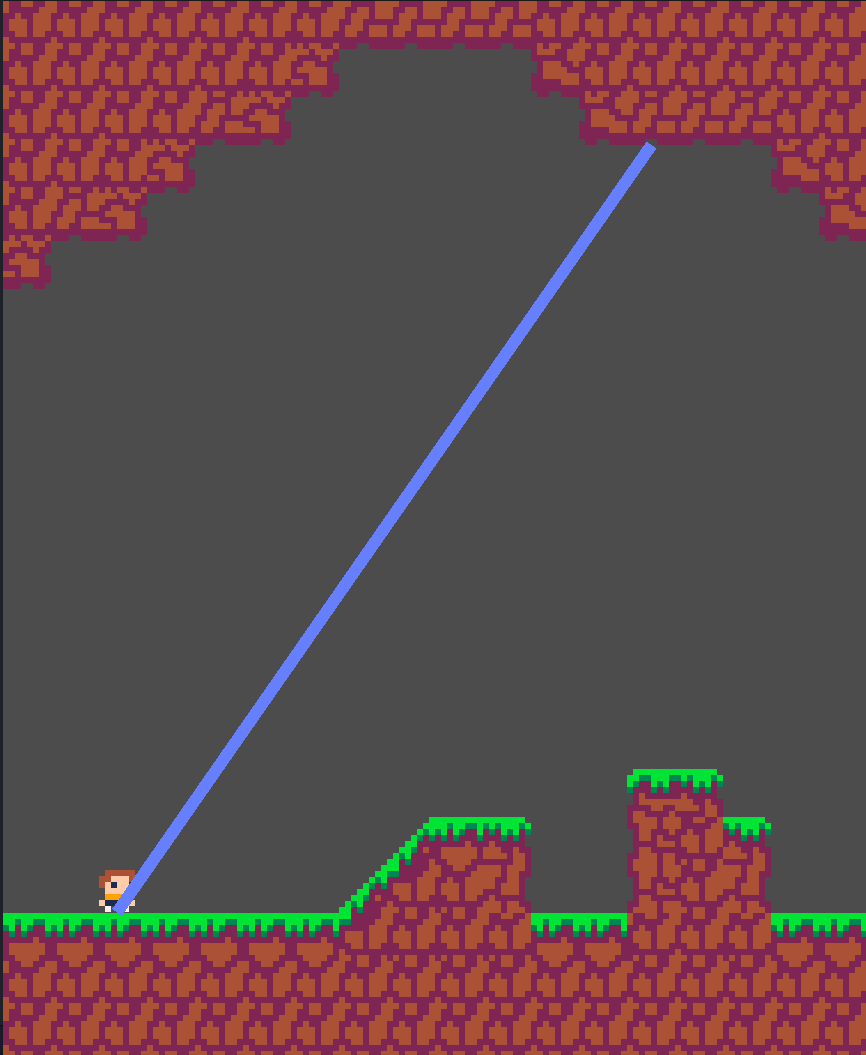
* Colliding with a tile
* Returning -1 if the mouse was not over a tile
* Colliding with a different tile if the mouse moves

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.

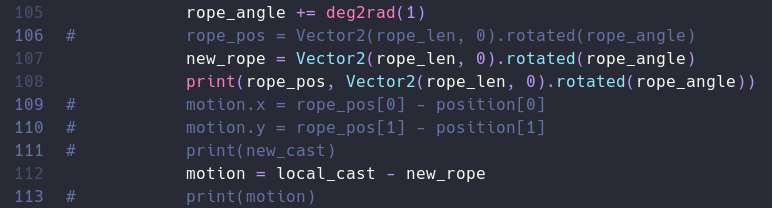
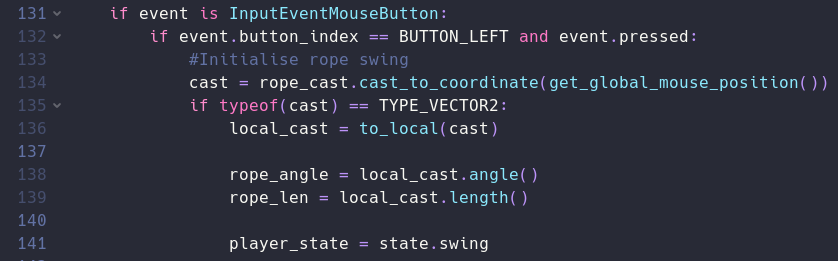


I was initially unsure about what was causing the other issue, so I researched online into how Godot deals with positions. It quickly became apparent that the “Line2D” node requires the use of global positions, whereas the position of the player was local. Global positions are stored as vectors from the top-left corner of the screen, whereas local positions are stored as vectors from the position of the node. Finally, to resolve the issue, I converted the position to a global one, and the line now draws the raycast correctly.

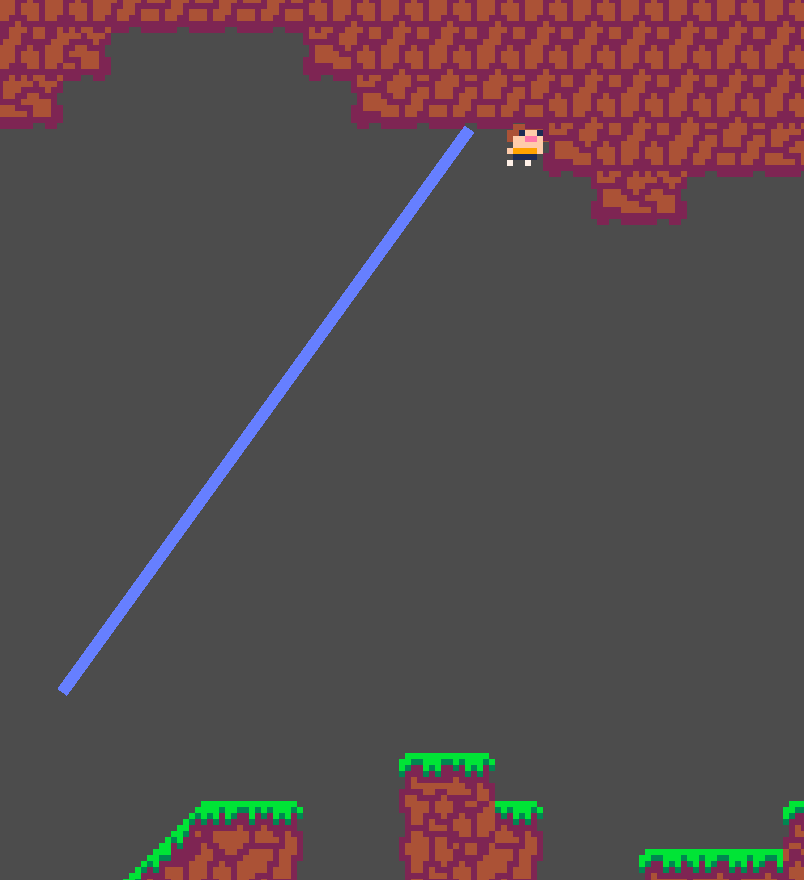
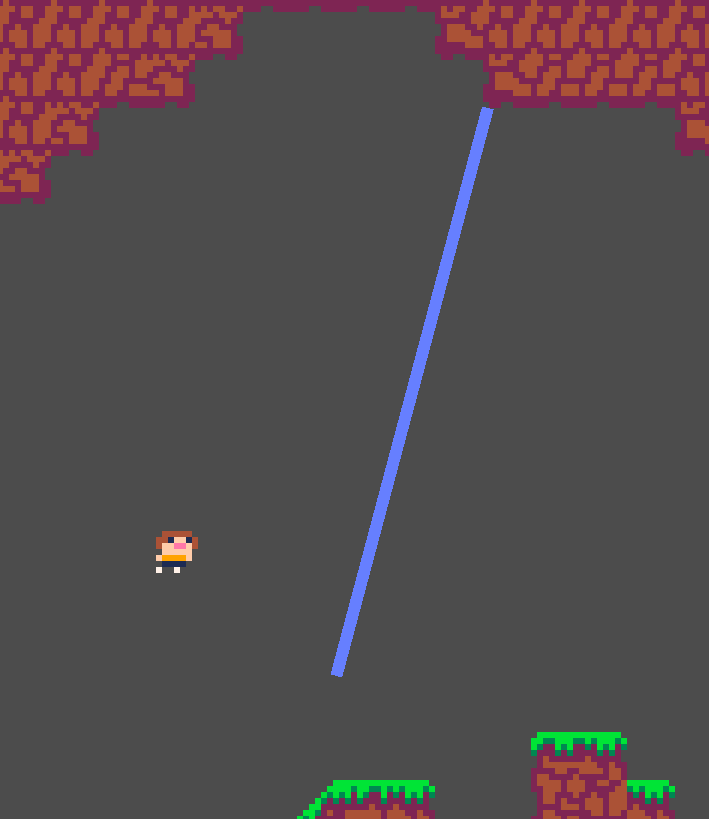
### Moving the Player with the Rope

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 letter. 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.

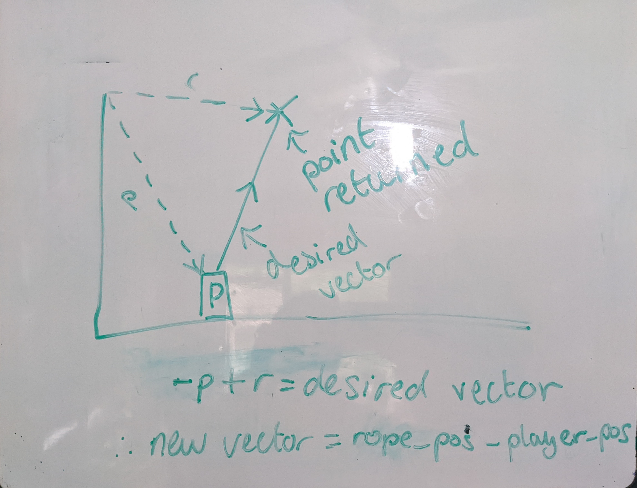
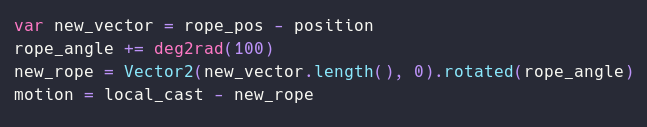


Instead, the player seemed to move in a sporadic, and seemingly random way. If the increment angle was increased, it would “gravitate” towards the point of rotation, but just shake around it. Moreover, the line position did not update, however I was not concerned about this, as it should be simply to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



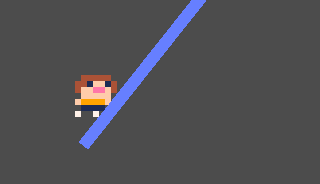
My new solution was to find the vector between the previous rope position and the next one, and then apply this vector to the player’s motion:

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 around in a circle around the base of the vector.

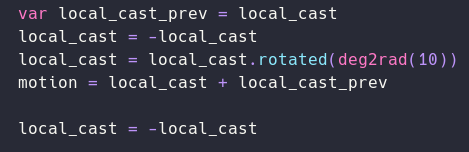
Previous Rope

Next Rope

Motion Vector



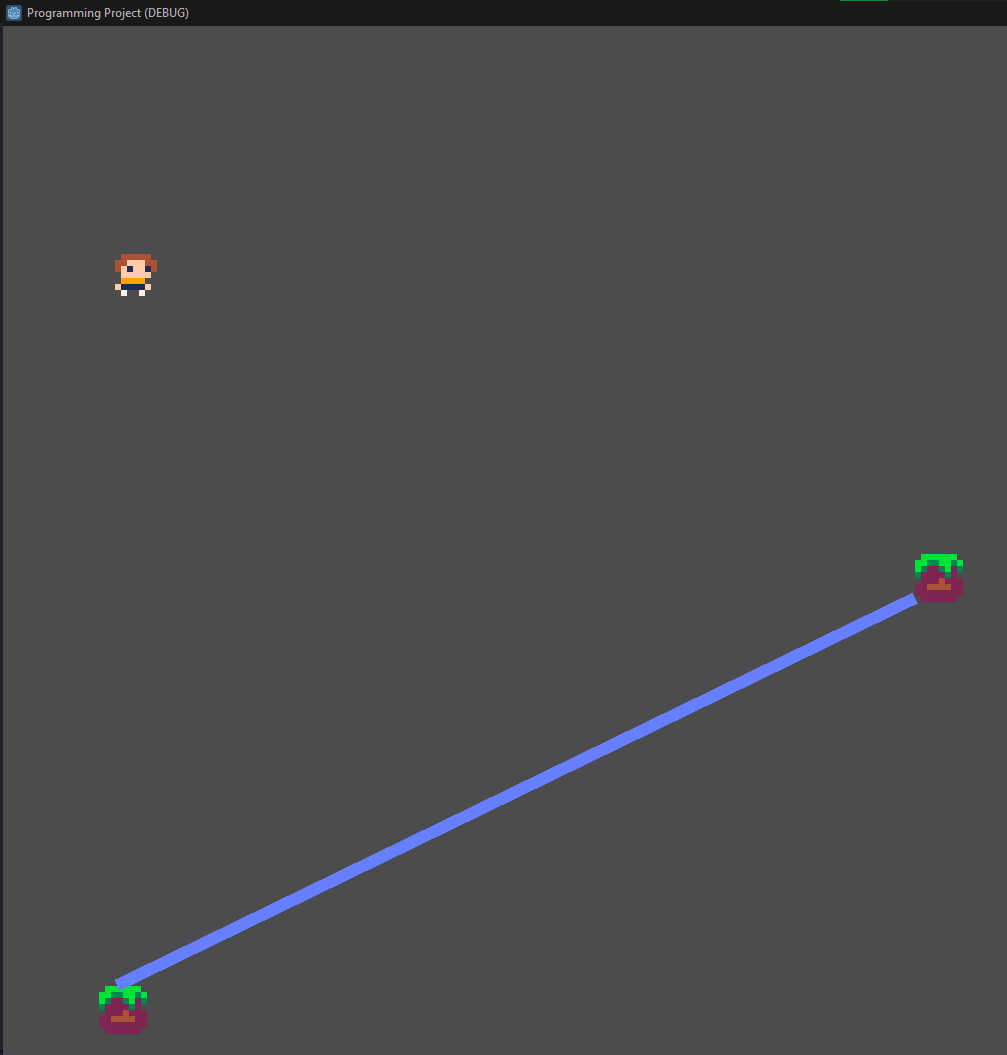
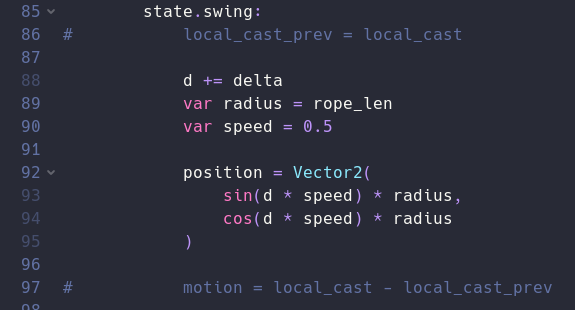
My initial thought was that the “rotated()” method was rotating the vector from the centre, rather than from one of the ends. My new solution was to manually create a new position vector (rather than a motion vector) for the player, and manually assign both the x-values and y-values to a new rotation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

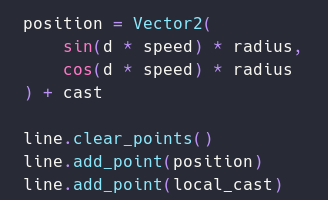
Rope   
Length =

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.

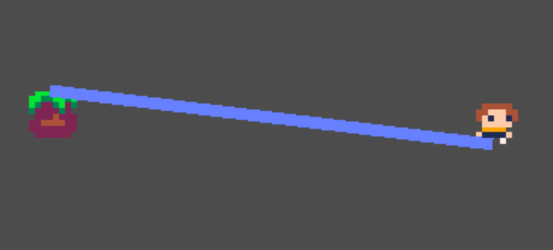


### Allowing the Player to Control the Rope

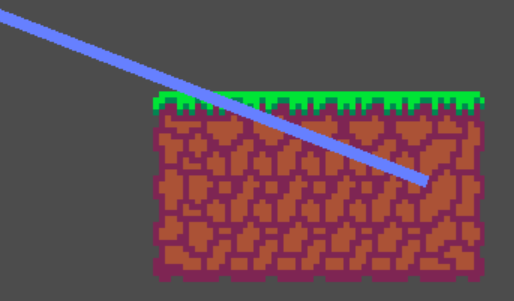
Now that the player was correctly “attached” to the rope, the final step was to allow the user to control the player whilst on the rope. To do this, I need to modify the previous code to instead take a new angle based on the inputs for the platformer movement.



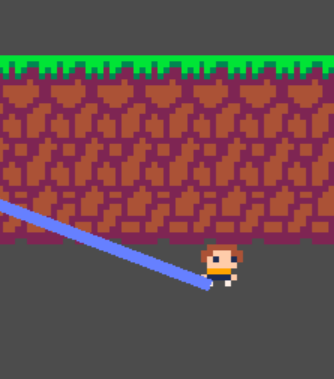
In order to change the length of the rope, only one variable needed to be modified: the radius (now renamed to “rope\_len”). In order to make sure that the length of the rope was never too short or too long, I “clamped” this value to stop it from going over the maximum, or under the minimum value.



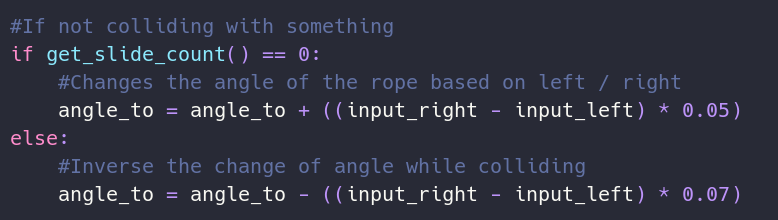
Now the player could control the length of the rope, at a fixed angle, I needed to apply the same logic to the “angle\_to” variable, which would change the angle the new position vector is to be at. Unfortunately, at this point I realised that there were no longer any collisions with the tiles.



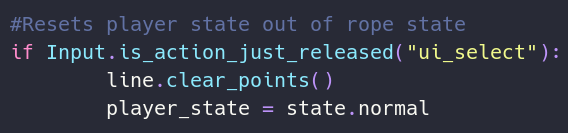
This was because I was manually updating the player’s position, instead of passing a motion vector through Godot’s built-in “move\_and\_slide()” method. To rectify this, I used the same logic that I tried previously, and created a new motion vector using vector maths. I then passed this into the player’s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



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.



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.



### 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.”

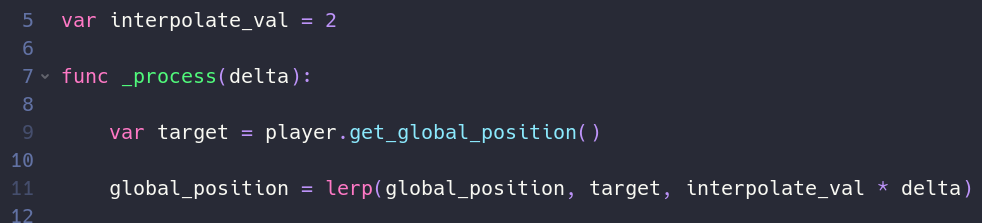
Thanks to this interview, I knew that I now needed to implement pseudo-gravity mechanics into the rope state, as well as allow the mouse to follow the camera. This is what I began to implement in the next stages of development.

### Moving the Camera with the Mouse

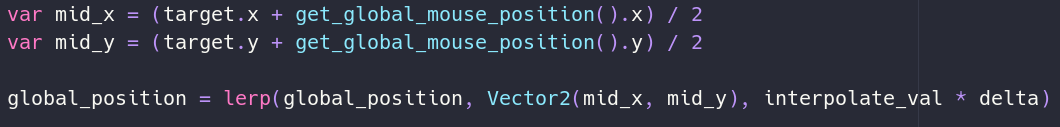
At this point, I noticed that it was very difficult to “aim” the rope at anything, since the camera always followed the player, and did not account for where the mouse was. If the camera followed the player and the mouse, always keeping both on screen, it would allow for much easier aiming.

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, which allowed it to move smoothly.



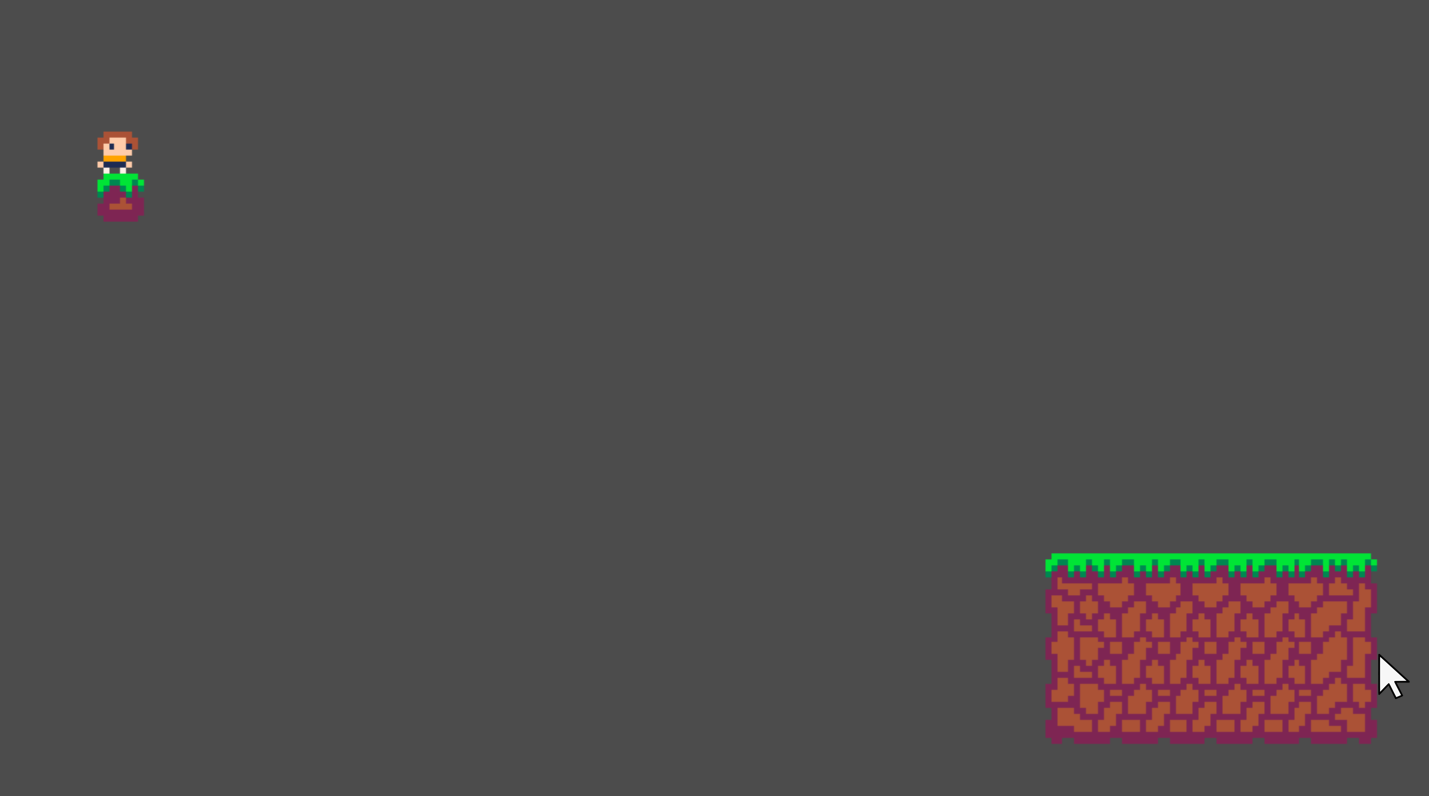
Next, I wanted to incorporate the centre position between the player and the mouse. This was as simple as taking the mean of the x and y component of their vectors, and then using that in the linear interpolation. This meant that the camera now kept both the player and the mouse on screen, while making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



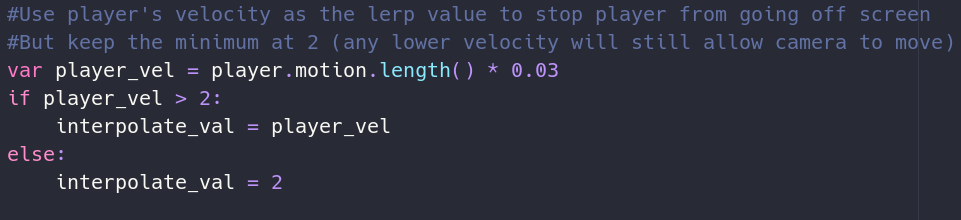
Camera Focus Point

Mouse Position

Player Position

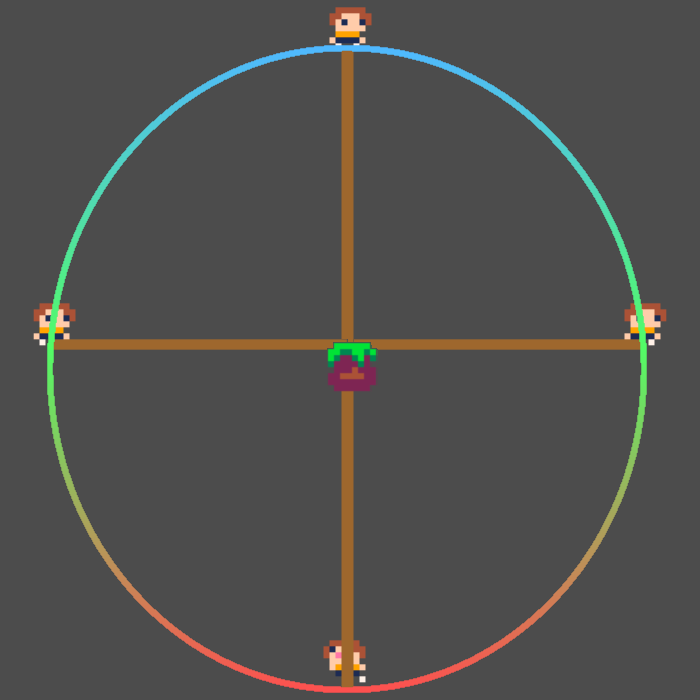


Finally, I wanted to make the camera move quicker if the player was moving quicker, allowing for more precision, as well as making sure the player is never off screen. Thanks to the linear interpolation, this is as simple as modifying the interpolation value, making sure this only changes if the player’s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



### Adding “Gravity” to the Rope

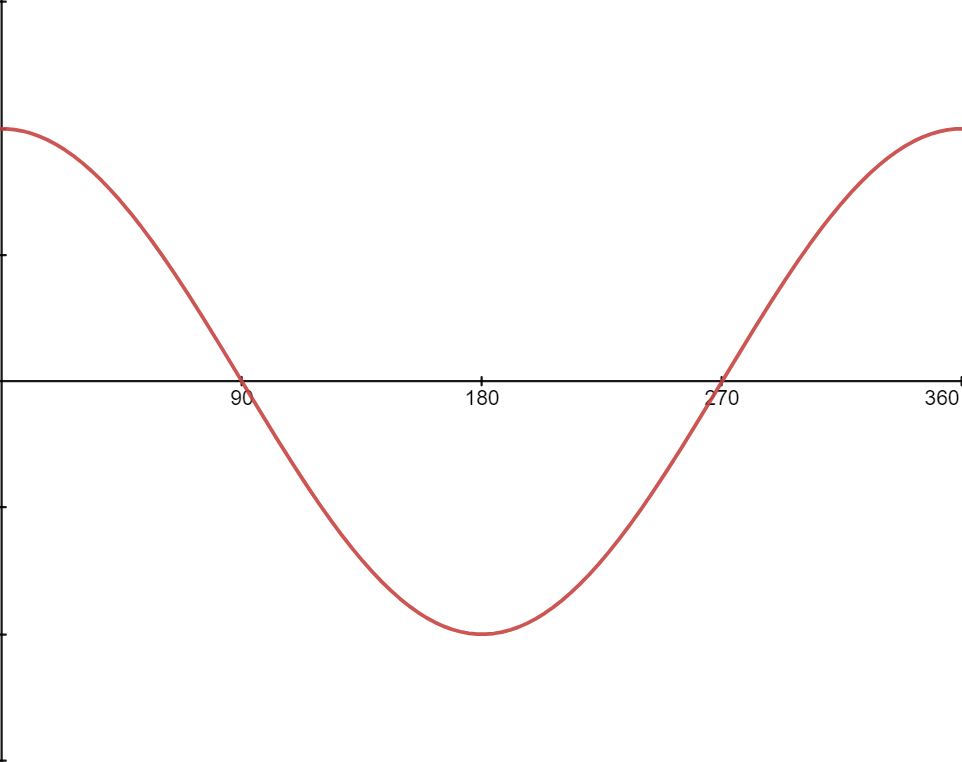
Now everything was working, although the rope physics felt a bit off. I realised that this was because there was no “gravity” on the rope, i.e., it was just as easy to move the rope at the bottom of the “circle” that the rope goes around, as it was at the top. To rectify this, I needed to apply an additional “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.



As the gradient changes to green, the “resistance” increases

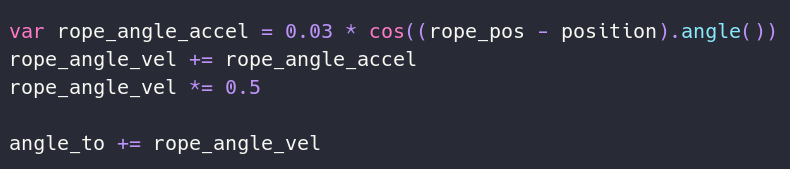
As the gradient changes to green, the “resistance” decreases

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 is perfect for what I need.



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.

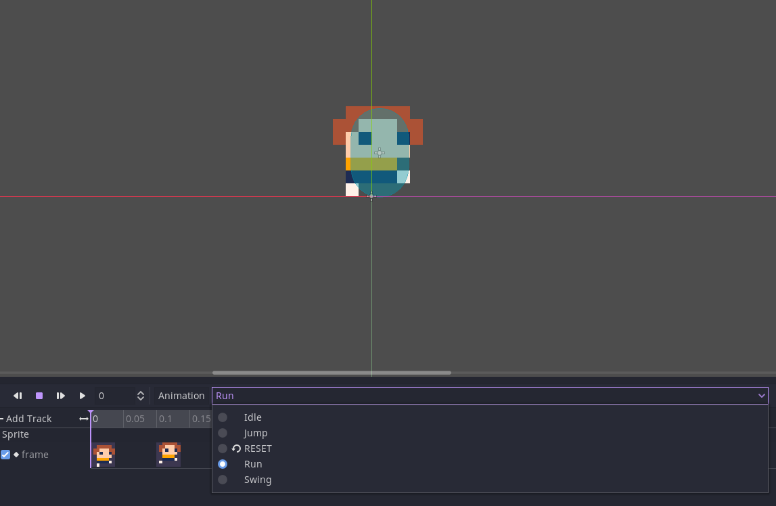
Now that the mathematics were figured out, finally implementing this system was an easy task.



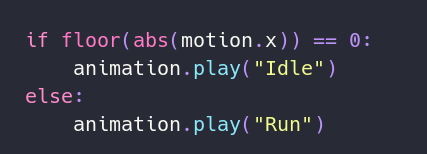
### Adding Animations

Now for an easy part: adding animations. To do this, I used Godot’s built-in animation editor by adding another 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.



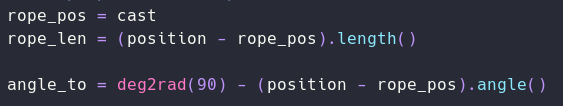
### Fixing the Raycast

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.

Rope Vector Angle,

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.

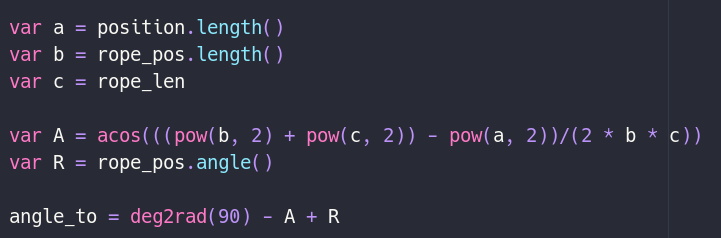
Implementing the fix to the code is extremely simple and resolves any problems with the rope’s angle.



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.



### 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”

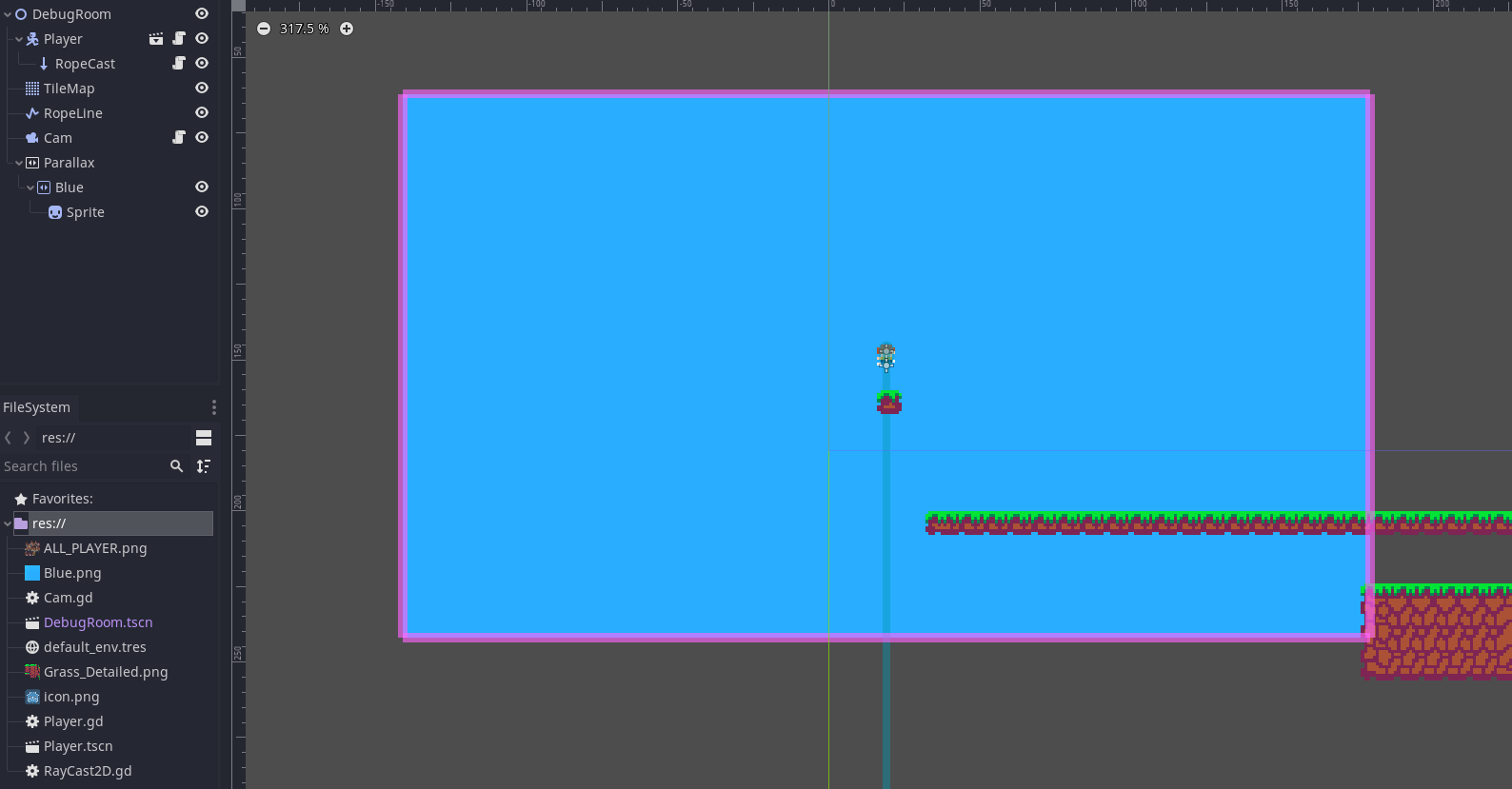
#### Question 3: What additional features do you feel should be implemented?

“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.

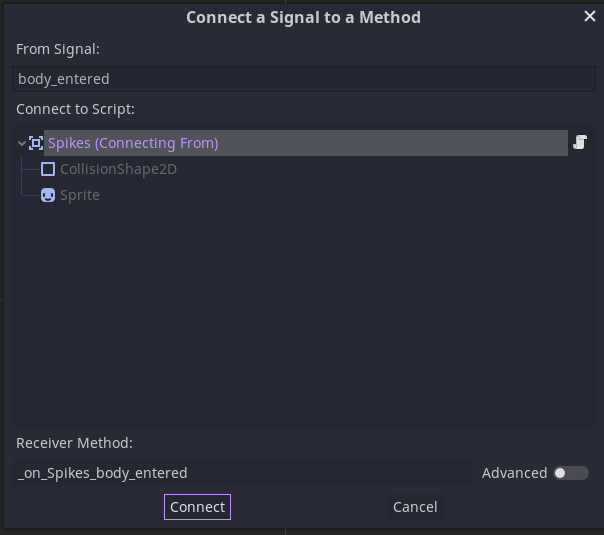
Now that the nodes were setup, I simply needed to make the background repeat itself indefinitely, rather than just in a small rectangle. To do this, I needed to add a region that was the same width and height as the camera. This makes the background “tile” itself, which is perfect for what I need. Doing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



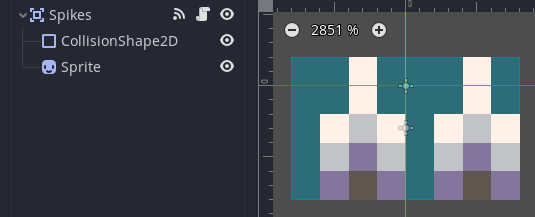
### Adding Level Hazards

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.

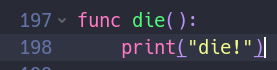
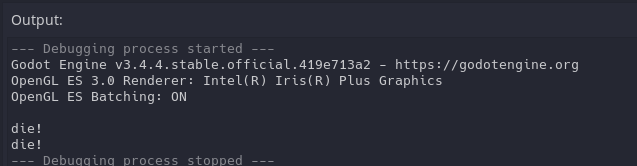
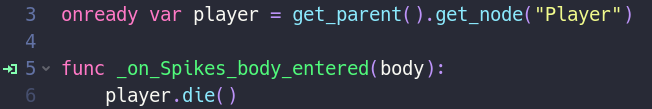


Once my collision was setup correctly, I needed to add a signal. Signals are events that attach to functions, which will run once the event happens. In this case, I setup an event that triggers whenever a body (i.e., the player) enters the spikes’ collision rectangle, and attach this to a function named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

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.

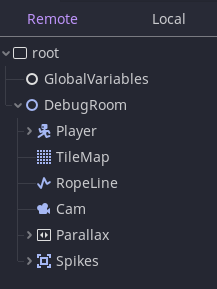


### Adding a Checkpoint System

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, which are always loaded.

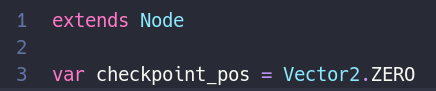
Firstly, I created a new script called “GlobalVariables”. Unlike all other scripts so far, this was not attached to any node to begin with. I then created a new singleton called “GlobalVariables”, and assigned it said script. This can be seen when the game is loaded.

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 room is loaded, or the current room is reloaded, the global variables node is not reloaded, which is perfect for this use-case.

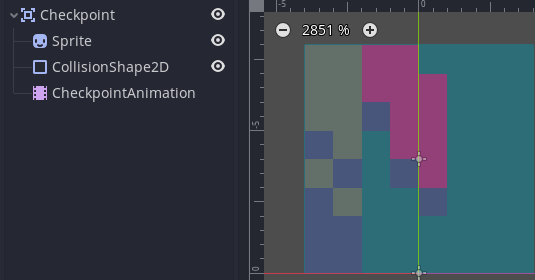
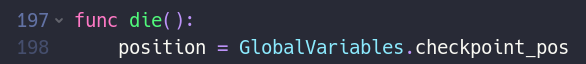


Inside the attached script, I added a variable called “checkpoint\_pos” and set it to a 2D vector at (0, 0) as a default.

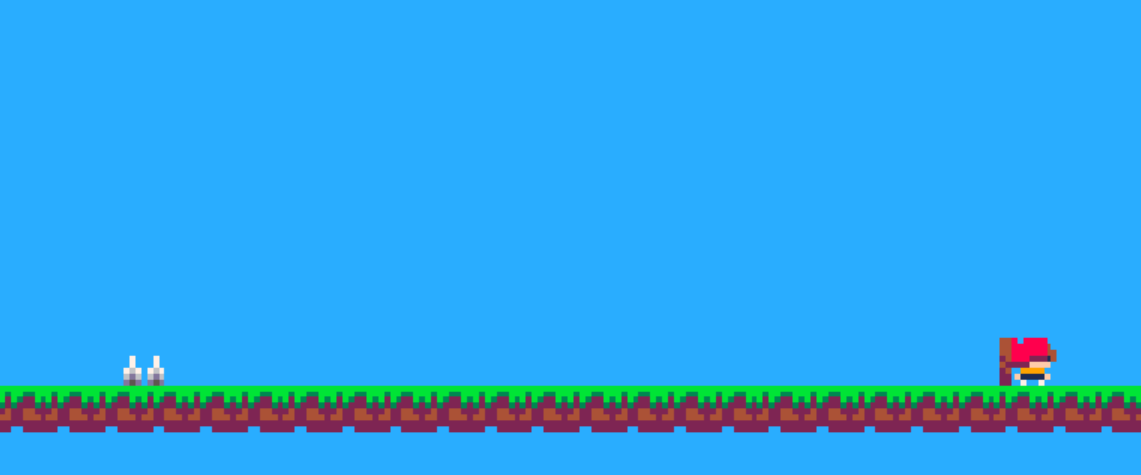
Finally, I modified the “die()” function to update the player’s position to this new global position, and tested the functionality by running into the spikes. This teleported the player to the (0, 0) position.



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.

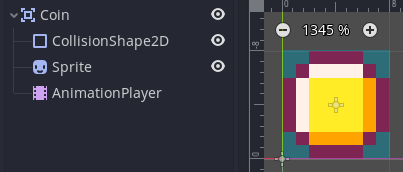
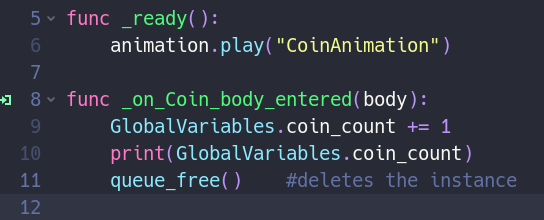


Once this was done, I added another signal to the checkpoint flag, which would update the animation and the global variable if collided with. This meant that if the player collided with the spikes, they would be teleported to the checkpoint. Implementing this and testing this out demonstrated that it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

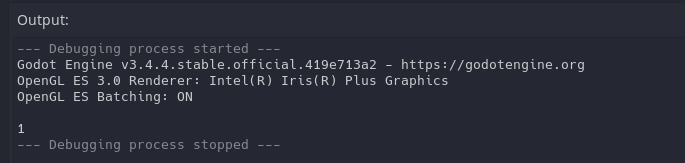


### Adding Collectibles

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.



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

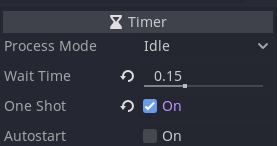


### Adding Coyote Time and Jump Buffer

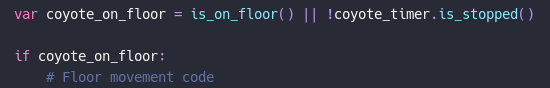
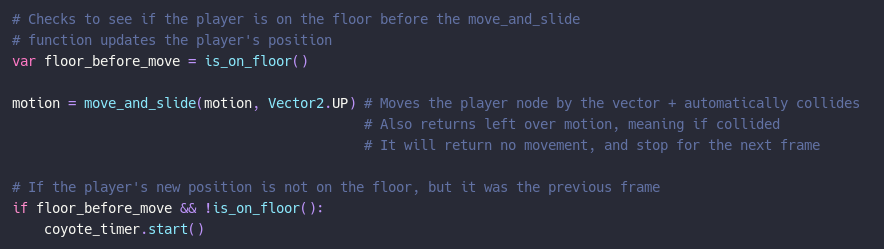
Now that all other in-game mechanics were sorted, I decided to tackle the lesser-needed, but still extremely nice to have mechanics, such as coyote time and a jump buffer. As mentioned in the design section, coyote time allows the player to jump even if they are not on the floor, as long as they only 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 was updated, I check to see if the player is no longer on the floor, if so, the timer starts.

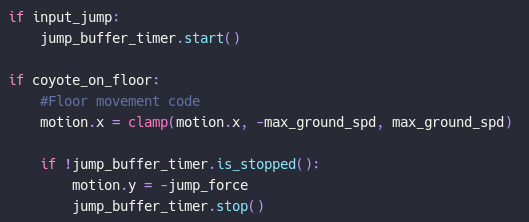


Finally, I modified the existing jumping code to check if either the player was on the floor, or the timer was still running.



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.



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.

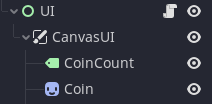
|  |  |
| --- | --- |
| Without Coyote Time | With Coyote Time |
|  |  |

### Adding a GUI

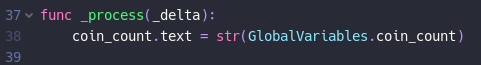
Now that all the game mechanics were added, it was time to add a GUI (graphical user interface). A GUI will allow me to display various statistics that the player needs to know in game, such as the coin and death count. To give the game a clean aesthetic, I want these counts to only show when the coin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

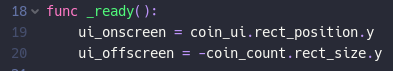
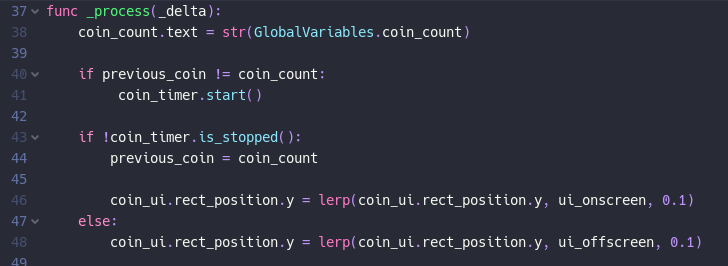
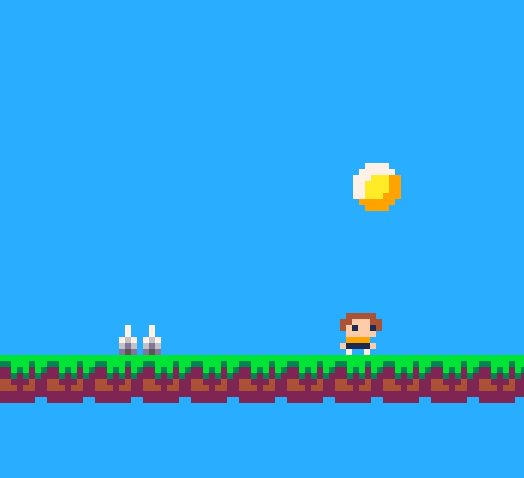


Now that the coin count was updating, I decided to add the showing functionality. This can be done by modifying the y value of the label and sprite. To make this easier, I created another control node, and made both the label and the sprite a child of.

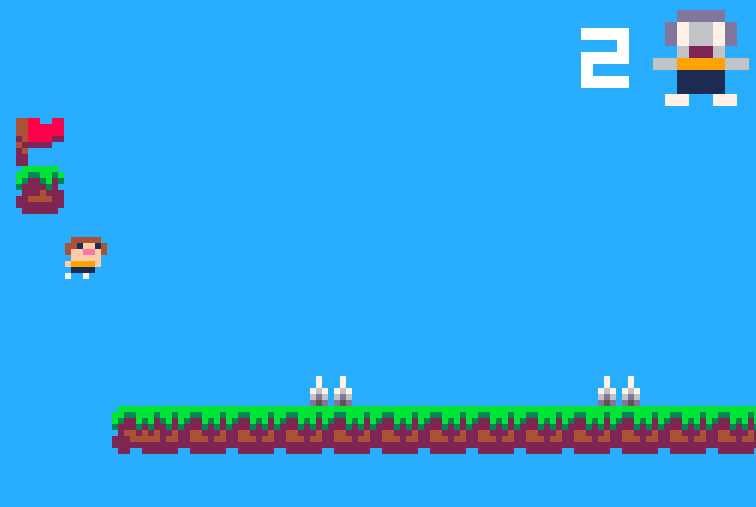
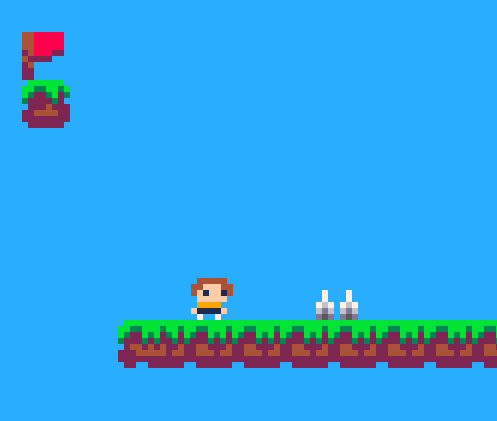
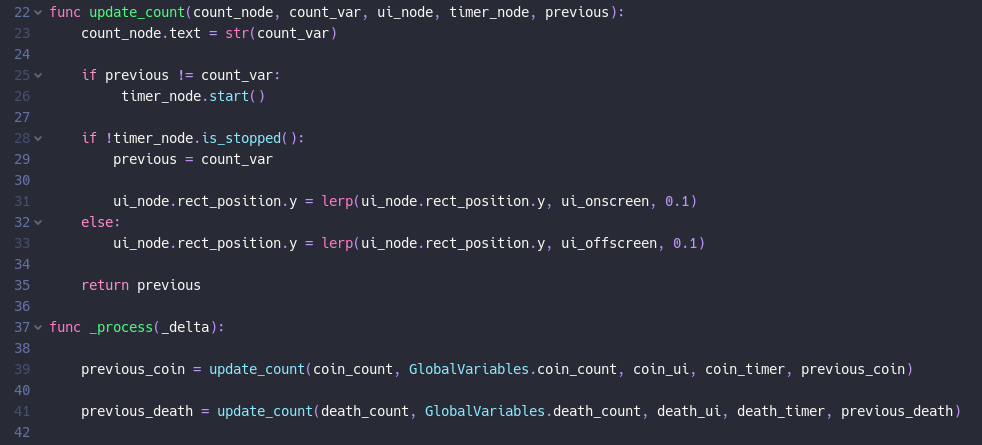


I then added a timer node, as a child of the CoinUI node, which would keep track of how long the UI should be on screen. I set this up to be 2.5 seconds, meaning every time a coin was collected, the UI would show for 2.5, and then hide again.

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



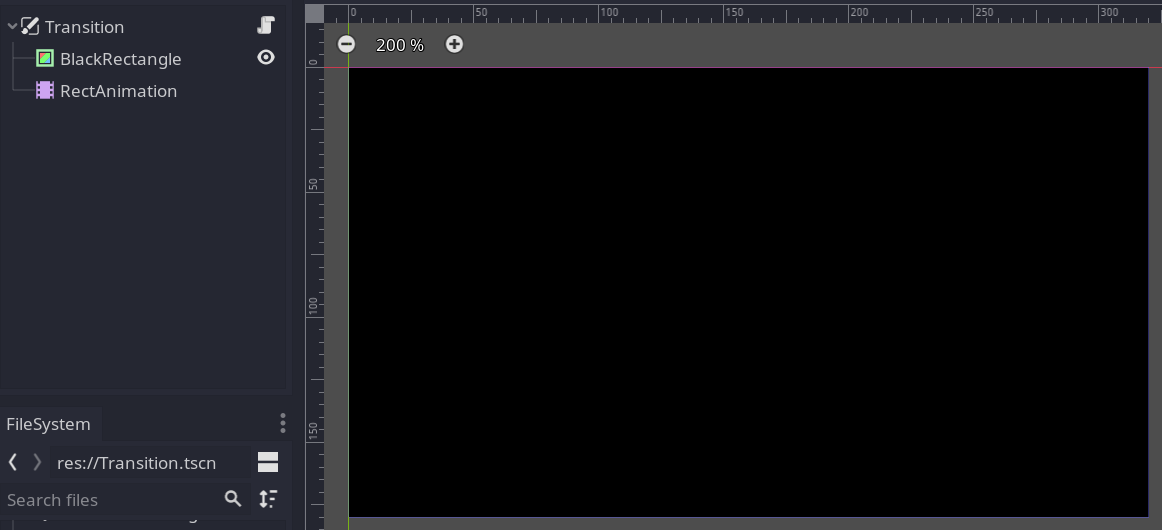
To replicate this functionality for the death count, all I had to do was duplicate the CoinUI node, rename appropriately, change the sprite, and copy the code. This worked, but it made the code hard to read, so I modified it to use a function instead. Initially, this was not working – the UI would stay 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.



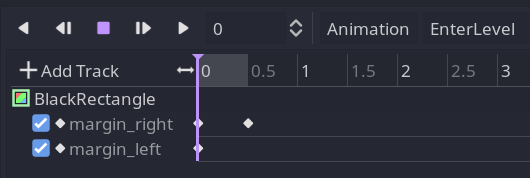
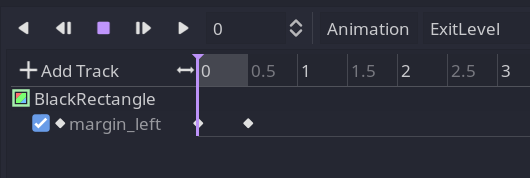
### Adding Level and Scene Transitions

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 created 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 setup 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:

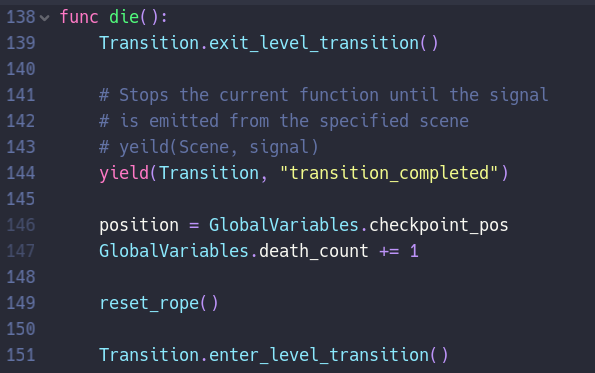
1. Start the transition
2. Wait until the rectangle covers the screen
3. Run unique code for each transition
4. End the transition

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).

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 in covered. The player’s death procedure is an example.



### Adding a Menu

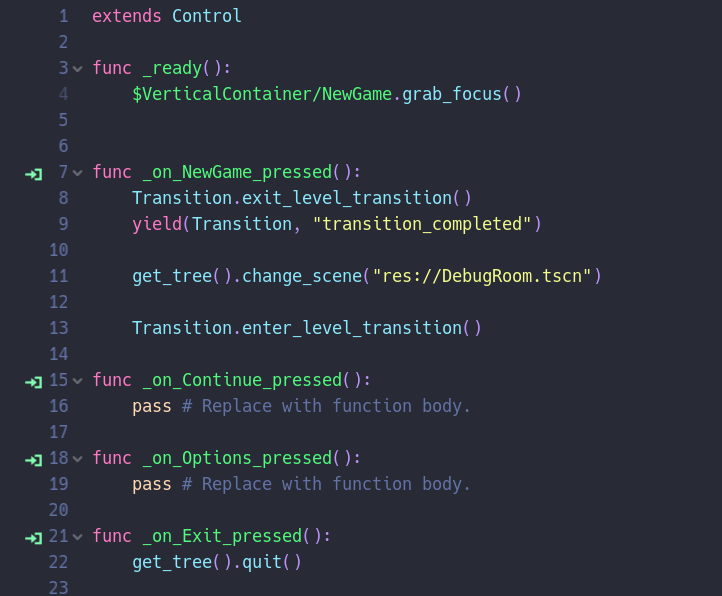
Now that transitions were added, I decided to add a menu, in which the transitions could be used between it and the levels of the game. I planned on only having 4 options for the menu: “New Game”, “Continue”, “Options”, and “Exit”. Implementing a continue and options function would require saving and 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 player left-clicked the button.

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.



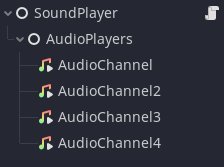
### Adding Sounds

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.

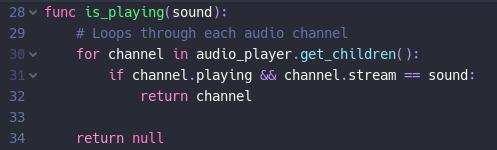
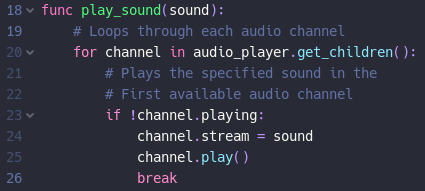
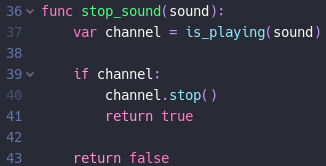


Now that my sounds were generated, I attached a script to the root node. The first thing that I needed to do was preload all the sounds. This allows them to be played in the “AudioStreamPlayer” nodes whenever necessary. Without preloading, the sounds would not be played at the correct time.



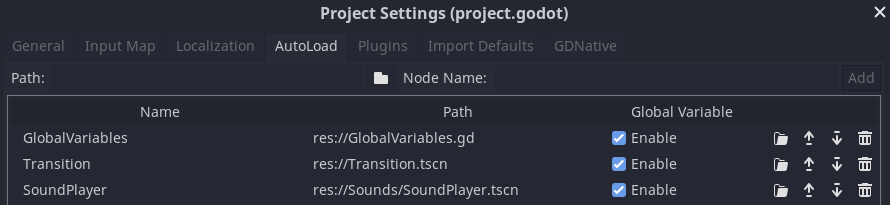
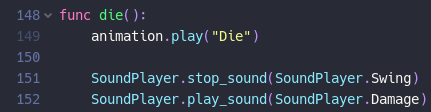
So, how was the system going to work? Whenever a function was called, which takes in an argument for what sound to play. It then loops through all the “AudioStreamPlayer” nodes, and finds if one is available (i.e., not playing a sound). If there is one, then it changes the available channel’s audio to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

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.



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.



### Adding “Non-Ropeable” Tiles

### Designing a Logo

### Setting Up Level Inheritance

### Pixel-Perfect Viewports

### Mouse Issues with Viewport Scaling

### Changing Levels

### Setup for “Continue Game”

### Adding Music

# 3.4 Evaluation

## 3.4.1 Testing to Inform Evaluation

## 3.4.2 Success of the Solution

## 3.4.3 Describe the Final Product

## 3.4.4 Maintenance and Development

# Appendices

## Code Listings

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

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% on steam, and 10/10 on IGN. [↑](#footnote-ref-3)
2. Spelunky is a 2008 2D platformer made by Derek Yu, mixing roguelike elements into platformers [↑](#footnote-ref-4)
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 [↑](#footnote-ref-5)
4. Shovel Knight is a 2D platformer developed by Yacht Club Games, which focuses more on the action side of platformers [↑](#footnote-ref-6)
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) [↑](#footnote-ref-7)
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 [↑](#footnote-ref-8)
7. XΞKINO is a fast paced, puzzle platformer game, created by Primed Pixel Games, aka. Malachy Moran-Tun, for the GMTK 2021 Game Jam. It uses a central mechanic of two players being “tethered” together, in which that tether can destroy enemies, as well as interact with other parts of the game. [↑](#footnote-ref-9)