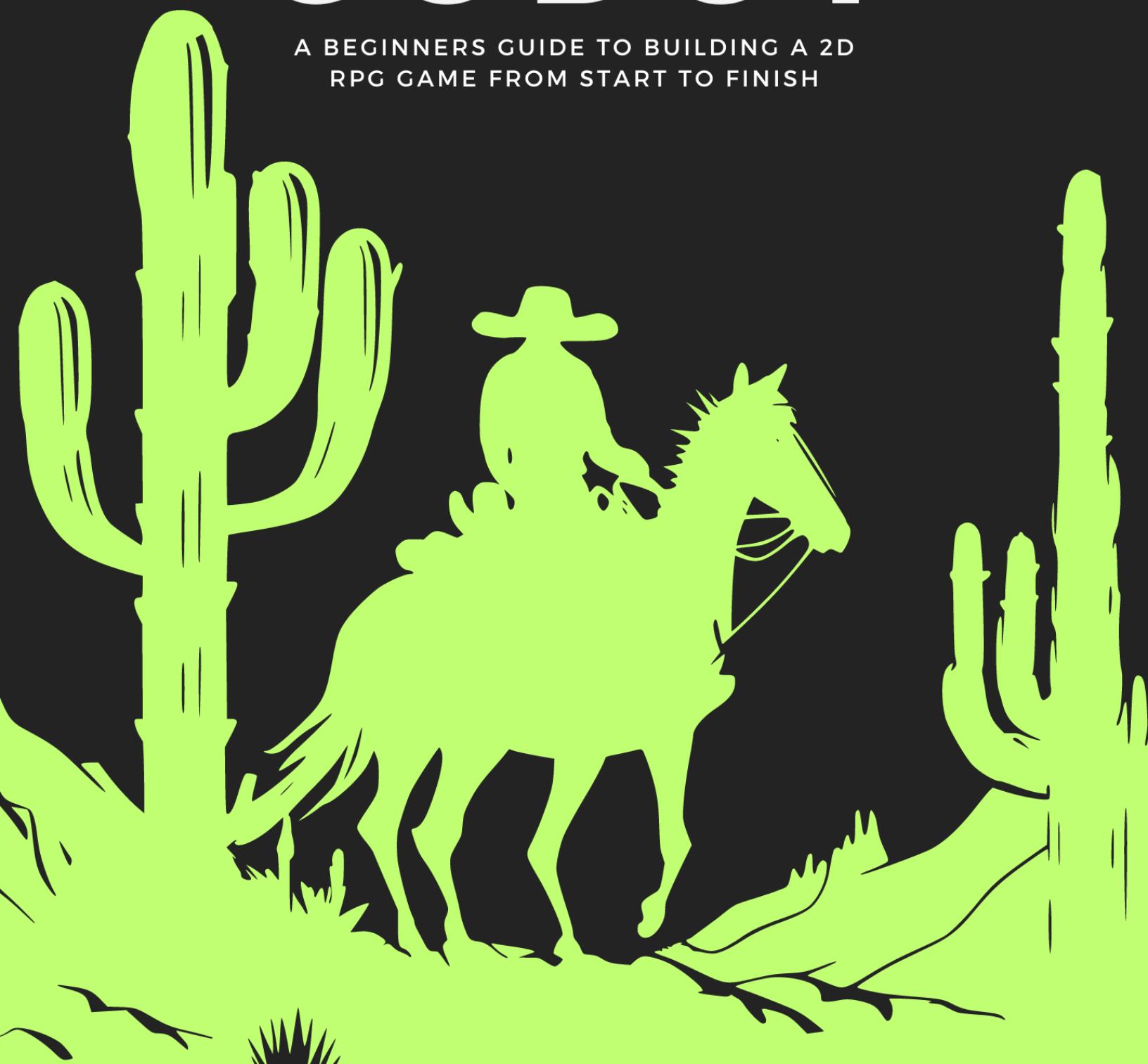


BY CHRISTINE COOMANS

LEARN GODOT

A BEGINNERS GUIDE TO BUILDING A 2D
RPG GAME FROM START TO FINISH



ABOUT ME

Hey there fellow Godot dev, and welcome to my Book of Nodes! My name is Christine Coomans, and I create and publish free tutorials for aspiring game developers in Godot 4!

I love to learn and teach others, and that's why I decided to put my experience, BSc in Computer Science, and my certifications in Creative Writing, Marketing, Graphic Design, and Web Development to good use by creating tutorials on game development for others. I plan to write more in the future covering other aspects of game development apart from programming tutorials, such as asset design and game logic fundamentals.

If you need assistance or want to give me feedback during the course of this book, please do so via the contact details below.

Email Address: christinec.dev@gmail.com

Twitter: @christinec_dev

Discord: <https://discord.gg/4N6b4bkC8h>

Website: <https://christinecdevs.site>

Good luck with your game dev journey, and don't forget to have fun!

TABLE OF CONTENTS

ABOUT ME	2
TABLE OF CONTENTS.....	3
PART 1: PROJECT OVERVIEW & SETUP.....	7
Editor Overview.....	9
Project Setup.....	20
PART 2: PLAYER SETUP & MOVEMENT.....	30
Player Setup.....	30
Movement Inputs.....	40
PART 3: PLAYER ANIMATIONS.....	51
PART 4: GAME TILEMAP & CAMERA SETUP.....	70
Map Creation.....	75
Player Camera	104
PART 5: SETTING UP THE GAME GUI #1.....	107
Health & Stamina Bars	109
PART 6: SETTING UP THE GAME GUI #2.....	124
PART 7: SETTING UP THE GAME GUI #3.....	132
PART 8: ADDING AMMO PICKUPS AND CONSUMABLES.....	138
Pickups Setup	138
Using Pickups	146
Spawning Pickups.....	156
PART 9: ENEMY AI SETUP.....	167
Enemy Scene Setup.....	168
Moving The Enemy	173

PART 10: ANIMATING ENEMY AI MOVEMENT	182
PART 11: SPAWNING ENEMY AI	189
Spawner Setup.....	189
Spawning Enemies	192
Pickup Spawner	199
PART 12: PLAYER SHOOTING & DAMAGE	203
Spawnable Bullet	203
Player Shooting.....	213
Enemy Damage	217
Enemy Death.....	228
PART 13: ENEMY DROPPING LOOT ON DEATH	232
PART 14: ENEMY SHOOTING & DEALING DAMAGE	235
Enemy Shooting.....	235
Spawning Bullets.....	239
Damaging Player.....	243
PART 15: PLAYER DEATH & GAME OVER	248
Game Over Screen	248
PART 16: LEVEL & XP	255
Level Up Popup.....	256
Showing & Hiding Cursor	275
Showing Values On Load.....	278
PART 17: BASIC NPC & QUEST	281
Npc Setup	282
Dialog Popup & Player Setup.....	286
Npc Dialog Tree	302

Quest Item Setup	311
PART 18: SCENE TRANSITIONS & DAY- NIGHT CYCLE	323
World Transitions Option 1	323
World Transitions Option 2.....	341
Realtime Day/Night Cycle.....	355
PART 19: PAUSE MENU & MAIN MENU	368
Pause Menu GUI Setup	368
Main Menu GUI Setup.....	372
Pause Menu Functionality	374
Main Menu Functionality.....	383
PART 20: PERSISTENT SAVING & LOADING SYSTEM.....	387
Saving The Game	390
Loading The Game.....	395
PART 21: SIMPLE SHOPKEEPER	404
PART 22: MUSIC AND SOUND EFFECTS	434
Main Menu Music	436
Pause Menu	438
Background Music.....	442
Game Over Music	445
Dialog Music.....	446
Level Up Music.....	449
Pickups Sound Effect	450
Consuming Sound Effect.....	451
Bullet Impact Sound Effect	453
Shooting Sound Effect.....	455

Enemy Death Sound Effect	458
PART 23: TILEMAPLAYERS CONVERSION	462
Conversion	462
Code Fixes	465
PART 24: TESTING, DEBUGGING, & EXPORTING.....	475
Debugging.....	477
Testing: Enemy Difficulty	482
Testing: Autosave	484
Exporting	486
PART 25: FURTHER RESOURCES.....	494

PART 1: PROJECT OVERVIEW & SETUP

Have you ever wondered what made Arthur Morgan from Red Dead Redemption so iconic? For me, it's the nostalgic feeling that arises when I get to play in an era that I would never be able to experience in real-time. It's the magic of fantasy, and that is what brought me to create the rootinest-tootinest 2D RPG game tutorial series out there, made with no other than Godot 4 and GDScript. This tutorial series aims to teach you how to make a game from start to finish, and ultimately show you all the things that you need to know as a beginner Godot developer!

So what exactly is this game that we will be creating about? Well, Dusty Trails is a Western-esque, Wildwest RPG in which our player can run around a full map, whilst shooting and looting enemies and completing quests. Enemies will spawn randomly, and they will roam and chase the player if they are close to the player. We'll also have NPC's and a Shopkeeper, which will give our player quests and sell pickup items to our player.

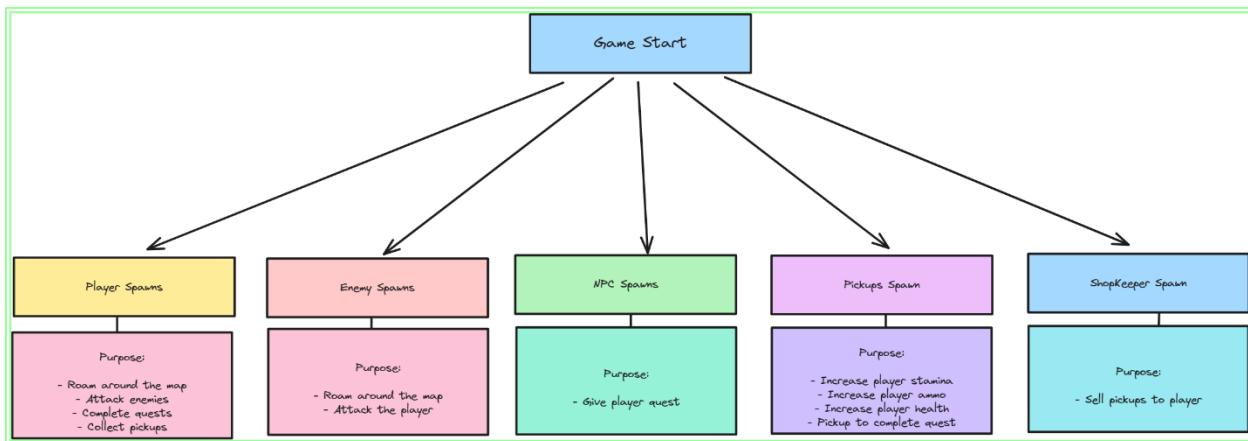


Figure 1: Overview of our game and the entities.

What is an RPG?

An RPG, or Role-Playing Game, is a genre of video game where players assume the roles of characters in a fictional setting. Players take responsibility for acting out these roles within a narrative, either through literal acting or through a process of structured decision-making regarding character development. RPGs often involve story-telling, world-building, and character progression.

This tutorial series will cover the following aspects:

- Player Animations, Movement, and Statistics
- Complete Game UI
- Pickups and Consumables
- Enemy Movement, Animations, and Statistics
- Enemy Random Roaming & Spawning
- Random Loot Drops
- Player & Enemy Death, Damage, and Attacks
- XP and Leveling
- Basic Customizable NPC with a Quest
- Scene Transition (Location Changing)
- Game Music & SFX
- Persistent Saving, Pausing, and Loading System
- Game Map with Collisions & Camera
- Simple Shopkeeper
- Project Exporting

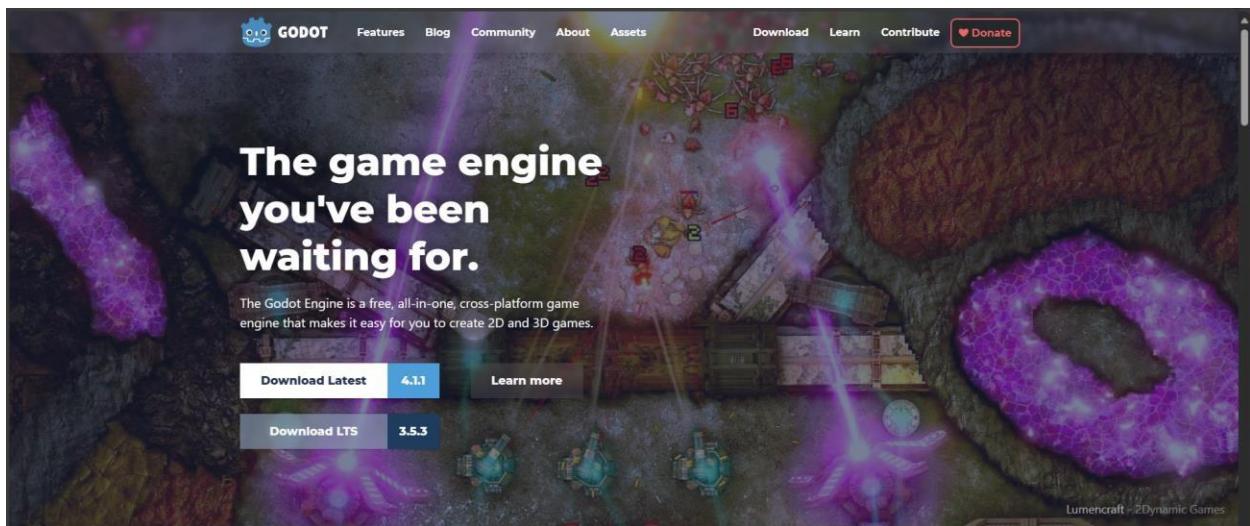
Please note that this tutorial series is meant for beginner GDScript & Godot developers. Also, this tutorial uses [Godot version 4.1.1](#) – so if you run into errors when using a newer version, please don't panic as this might happen. I'll try and update this project as much as possible to work with newer versions of Godot. I'll also link the sectional source code to each part at the end of the individual chapters so that you can easily compare the projects' code to yours.

WHAT YOU WILL LEARN IN THIS PART:

- How to navigate through the Godot Editor.
- How to change your project settings, as well as work with node properties.
- How to create scenes.
- How to add nodes to scenes.

EDITOR OVERVIEW

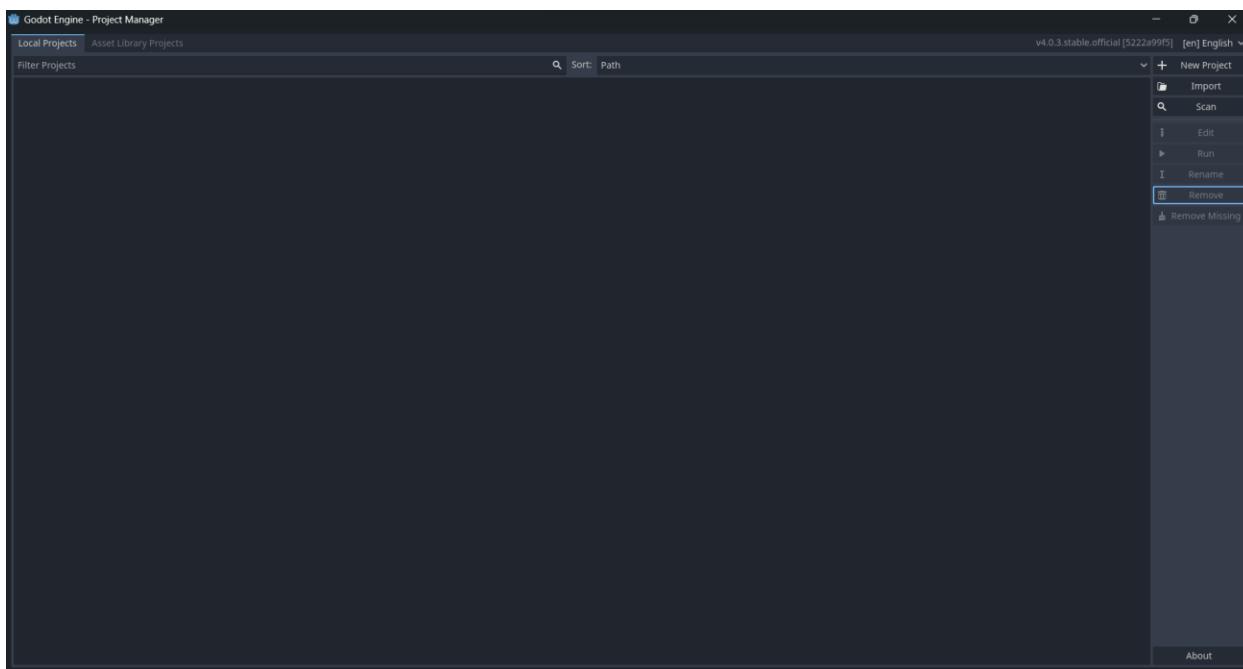
Before we get crazy and begin coding, we first need to understand the layout of the Godot Editor. If you are already acquainted with the editor, you can skip to the [Project Setup](#) section of this part, but if not, I'd advise you to take a tour of the editor with me.



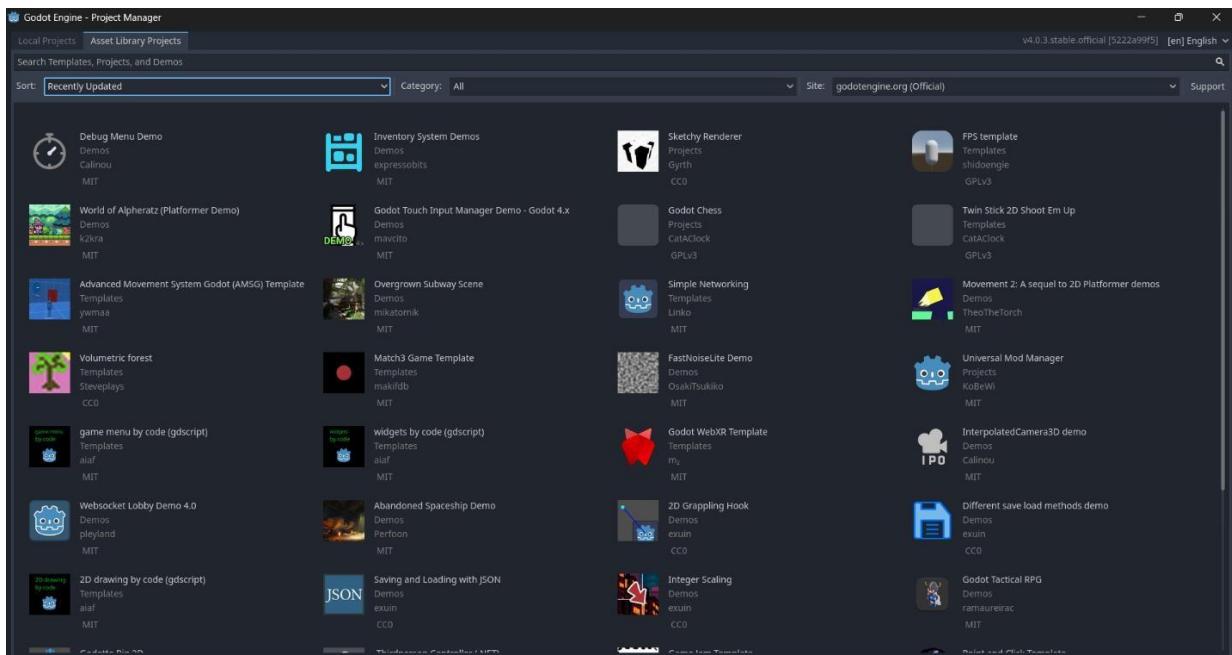
If you haven't installed Godot yet, head on over to their [website](#) and download the latest version that is available to you. Once it's done downloading, extract the file, and since Godot does not require any installation, you can just drag this to where you prefer to have it stored on your computer - or open it directly from your Downloads folder.

Name	Date modified	Type	Size
▼ Today			
Godot_v4.1.1-stable_win64	9/28/2023 9:08 AM	Application	116,662 KB
Godot_v4.1.1-stable_win64_console	9/28/2023 9:08 AM	Application	193 KB

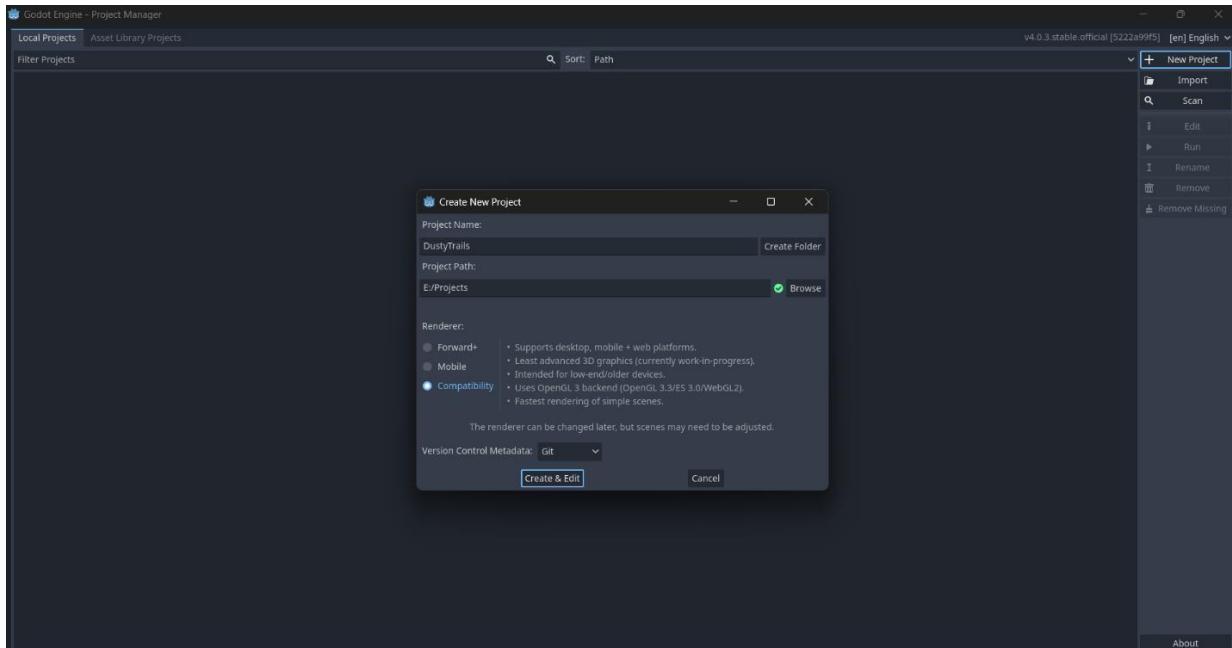
Launch the executable, and you'll be met by the Project Manager window. Here you can create new projects, load existing projects, or scan entire directories to import multiple projects.



On top, you will see a menu called "Asset Library Projects". This is where you can download and find projects, templates, or demos that are made for Godot by community members. It's similar to other online marketplaces such as Itch.io, Unreal Marketplace, or the Unity Asset Store. The only difference is that everything here is free.



To create our project, we'll click on "New Project", and a window will appear where we will choose our project's save location as well as give it a name. I named mine Dusty Trails but go ahead and call it whatever you want.

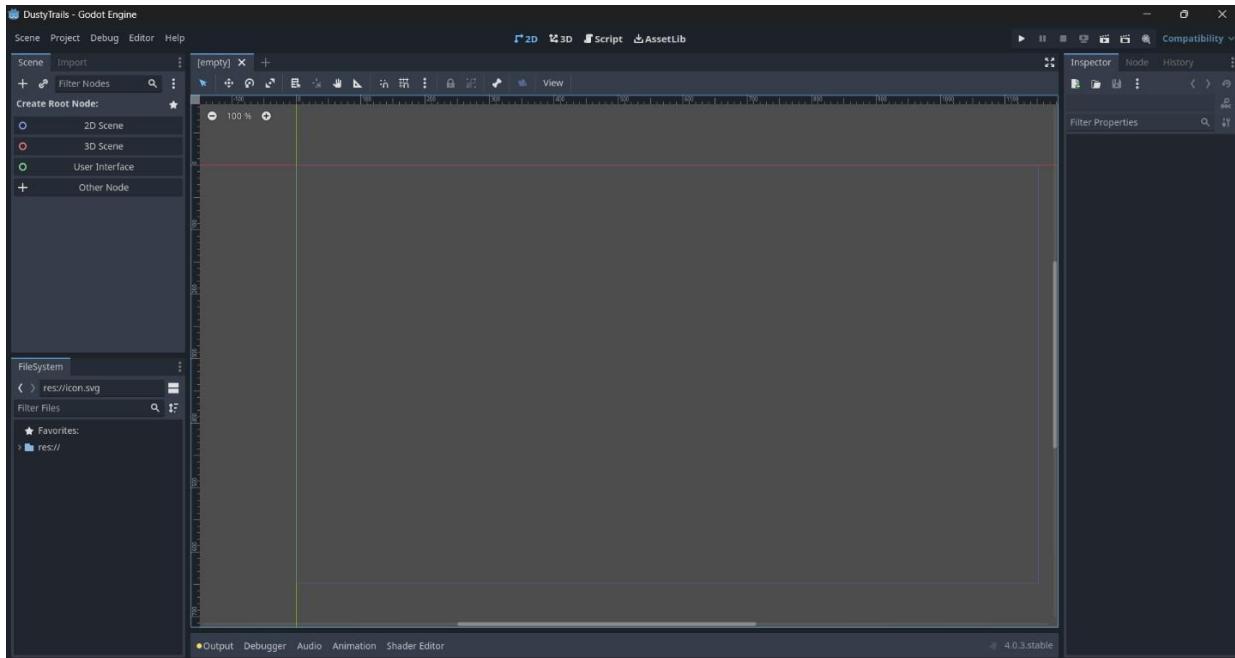


What are the rendering modes?

- **Forward+ mode**, which is for more upscaled game projects that are more graphics and rendering intensive.
- **Mobile mode**, which is generally used for mobile games.
- **Compatibility mode**, which uses the same renderer as Godot 3, meaning the graphics and rendering quality is lower making it more suitable for older or more low-spec devices.

You can start your game either in Compatibility or Forward+ mode – it just depends on if your Graphics Driver supports the Vulkan version provided with Godot. You can read more about the new rendering updates [here](#).

Select "Create & Edit", and you'll be magically transported to the Project Window. On top, we have our workspaces, which is the space in which we complete our tasks. 3D will always be active by default. If you click on the other options, you can switch between the different workspaces.



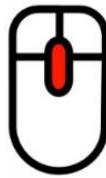
Godot Workspace Navigation



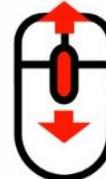
Select



Add Node



Pan/Move



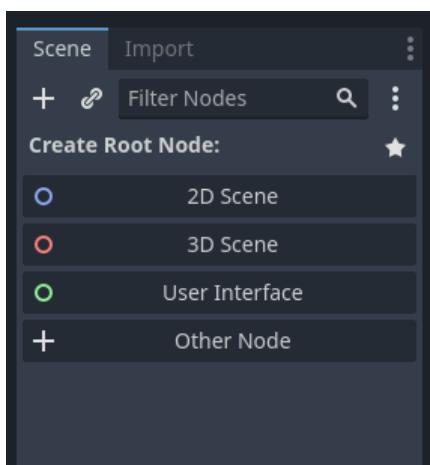
Zoom In/Out

Figure 2: Mouse Inputs for Workplace Navigation

What are the workspaces for?

- The **2D workspace** is used to edit and view 2D scenes for 2D games.
- The **3D workspace** is used to edit and view 3D scenes for 3D games.
- The **Script workspace** is the code editor where we write, edit, and view our scripts.
- The **AssetLib workspace** is the Asset Library Projects that you can use in your projects for free.

The panel on the left side of the window is the Scene Dock. This lists the content of the current scene that you have open, and you can also add new nodes to your scene in this dock. A scene in Godot is the main holder or container for nodes, and nodes are the main building blocks of your game.



Different nodes have different functions, for example, collision shapes and animation sprites each have their purpose, so having the ability to combine them in a singular scene allows us to create components for our game with more complex functions. Scenes can also instantiate other scenes, for example, having enemy scenes in your world scene. I recommend you go and have a look at the Godot Documentation on [Nodes & Scenes](#).

What is a Scene and a Node?

A scene is a container that holds a collection of nodes arranged in a hierarchical structure - such as our Player or Enemy. A node is the basic unit within a scene that makes up the scene - such as a Sprite or Camera within our scene.

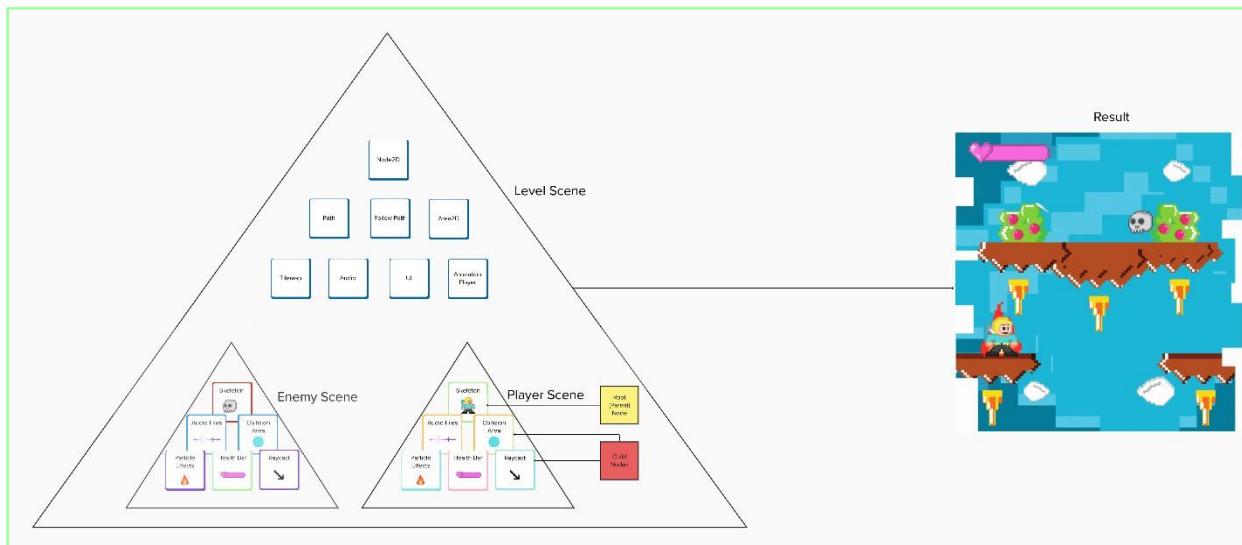
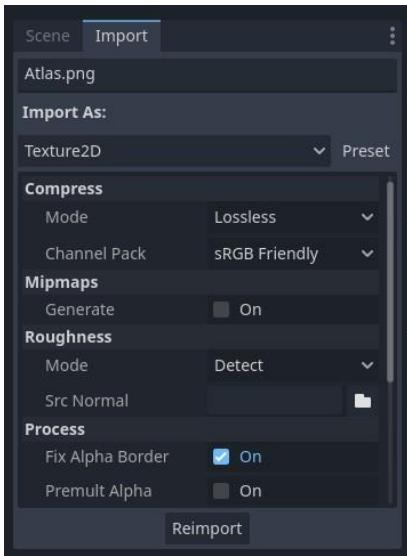
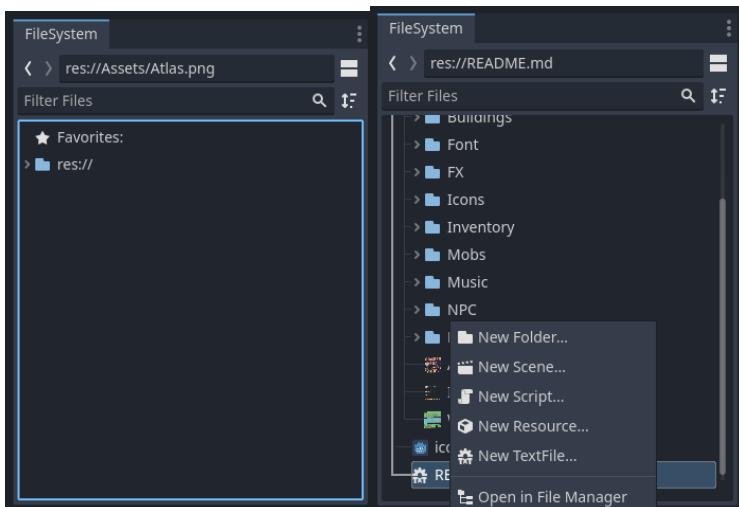


Figure 3: Example of nodes and scenes. View the online version [here](#).

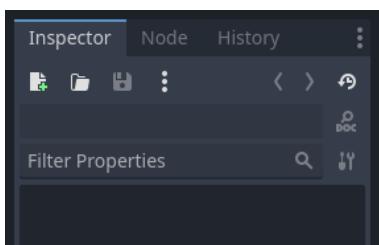
The other panel is called the Import Dock, and this is where you can set or edit the properties of assets such as images, fonts, and audio files that you've added to your project folder, and then you can reimport them with those properties saved.

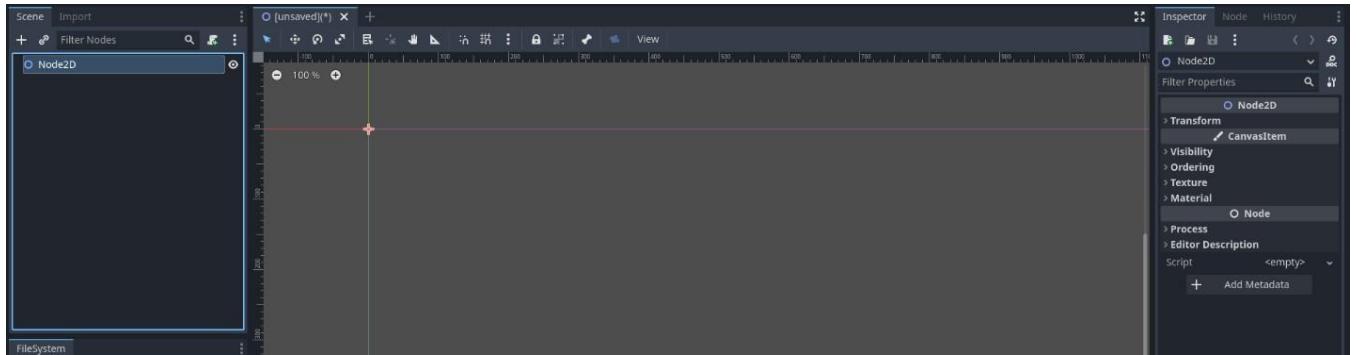


Below the Scene and Import Dock, you will find the FileSystem panel, which is where you'll manage your project files, folders, and assets. You can drag new assets into this panel to import them into your projects, as well as add new folders, scripts, scenes, and more.



The panel on the right side of the window is called the Inspector Panel, which is where you can view and edit the properties of currently selected nodes.





The Node Panel next to it is where you can hook up [Signals](#) and add [Groups](#) to nodes. Groups can be used to group types of scenes or nodes together, for example, our NPCs, so that our player knows how to interact with those nodes. Signals can be used in many cases, for example, to notify the game that a button has been pressed or that an animation has finished playing. This allows us to then tell the game what to do when certain events have occurred. Most nodes have built-in signals, such as buttons and animation players, but we can also create our own custom signals. We'll explain these concepts a bit more in-depth later on.

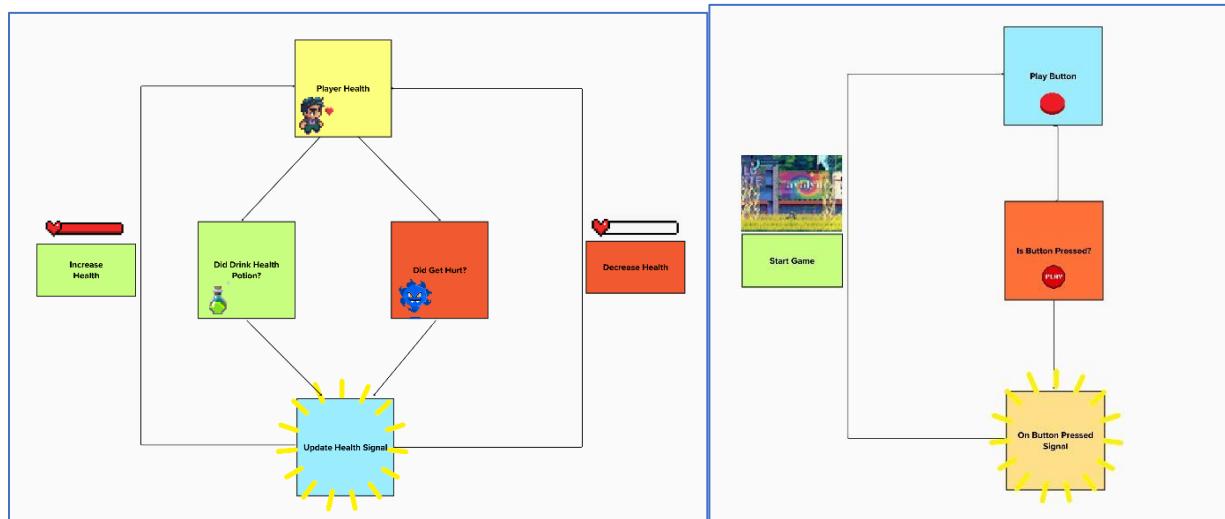
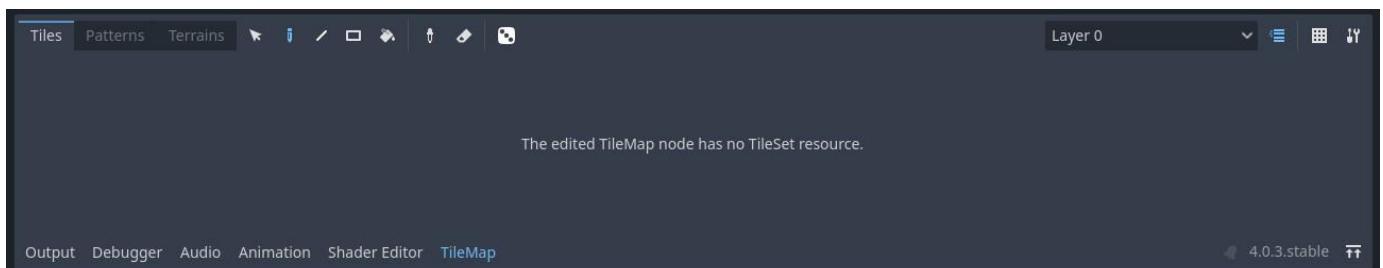
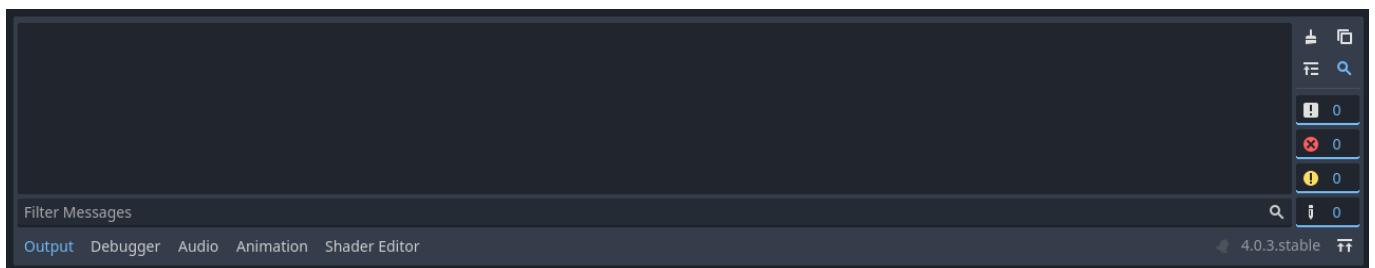


Figure 4: Custom Signal (Left) & Built-in Signal (Right). View the online version [here](#).

Above the Inspector and Node Dock, you will see the options we will use to play, pause, restart, or stop our scenes. This is where we will launch our game to test it.



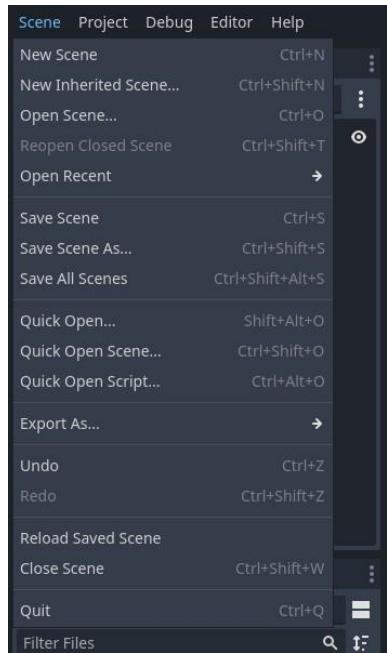
At the bottom of the Editor window, there is the Output Panel, the Debugger Console, the Audio Mixer, the Search Results Panel, and the Animation and Shader Editor. They are collapsed by default to save screen space, but if you click on them, they will maximize. Certain nodes that are customizable, such as the tilemap or animation-player node, will add new options to this window.



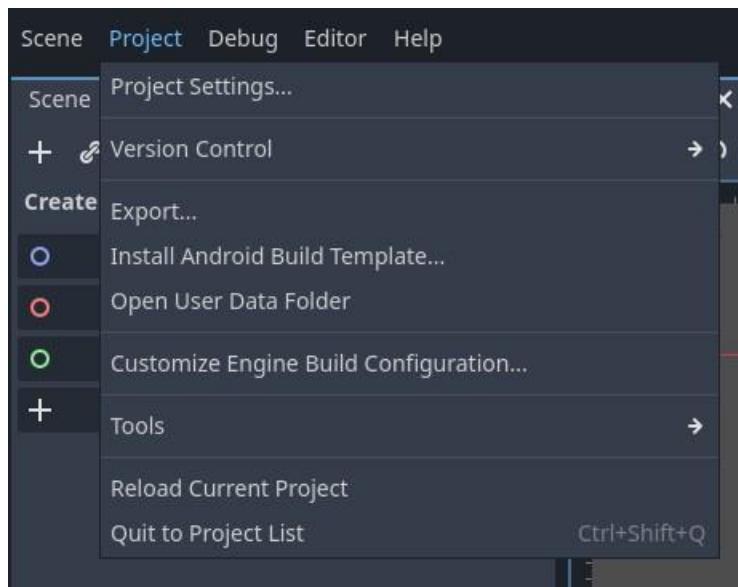
Let's go over each of the pre-existing options:

- The **output console** will return printed messages or notifications that we add to the code for testing.
- The **debugger console** is used for - you guessed it - debugging and error handling.
- The **audio mixer** is used to edit audio files.
- The **animation editor** is used to create animations for the Animation Player node.

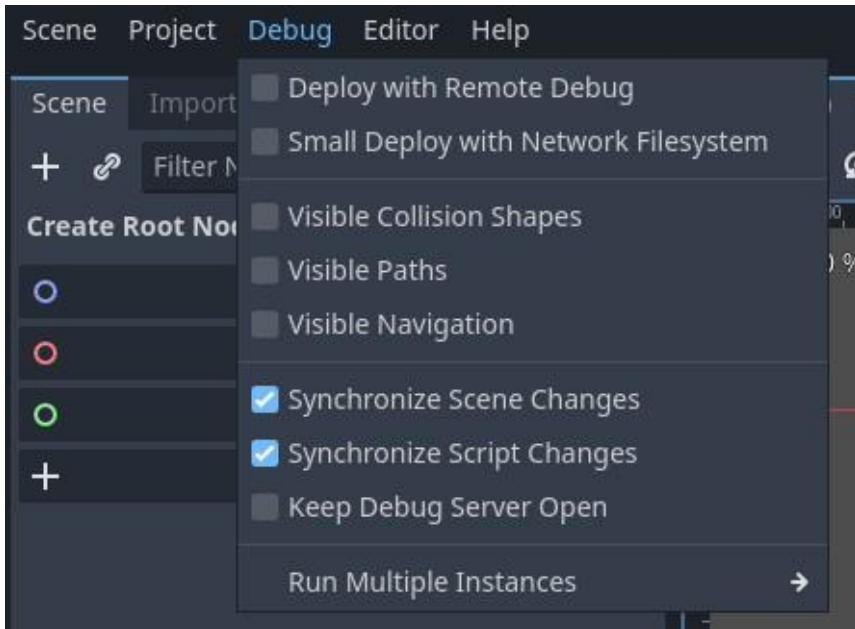
In the Ribbon or Menu bar above, you have a few options. Here you can manage your scenes underneath the Scenes tab. Scenes can be added, saved, opened, or closed.



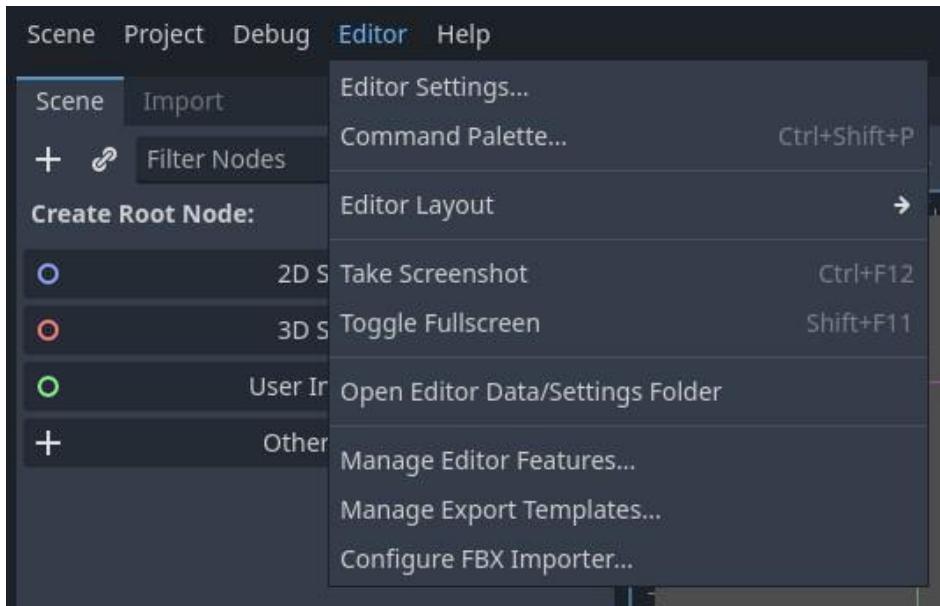
You can manage your project settings and properties (such as input actions, window size, autoload singletons, etc.) underneath the Projects tab. You can also quit your Project Manager window from here. This is also where we will export our project from.



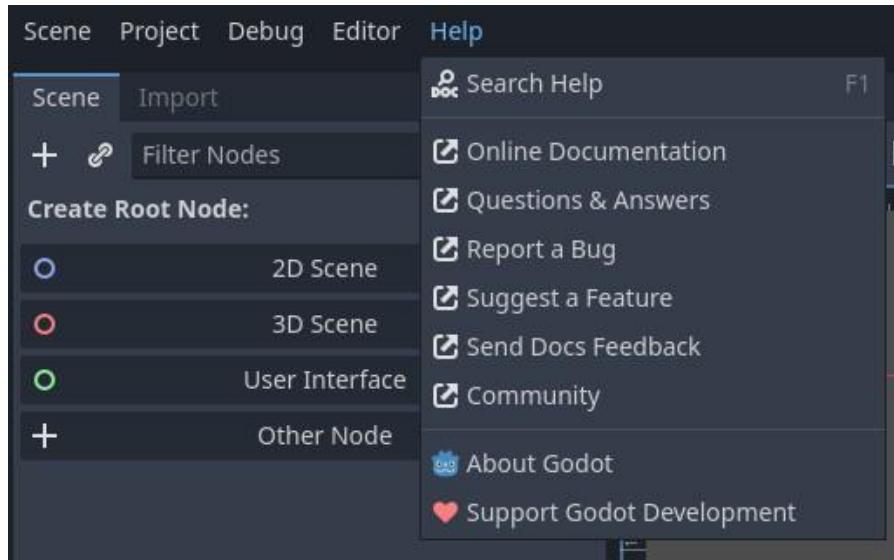
You can enable or disable debugging tools in the Debug tab, for instances where you want to say, make collisions visible so that you can see if collision events are executing during gameplay.



The Editor tab allows you to change settings related to the editor, for example, you can change its current layout or theme. You can even take screenshots of your current window.



And lastly, the Help tab gives you quick access to the Godot 4 documentation and community forum. If you're feeling a bit more comfortable with working with the editor, we can go ahead and set up our project.



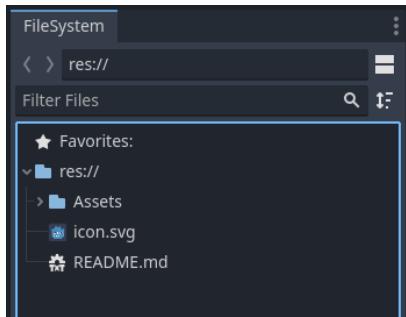
PROJECT SETUP

Before I start a project, I generally make a list of steps (or checklist) that I need to take to make the game. This is generally just an outline, and I make different amendments to these steps as I go about actually making the game.

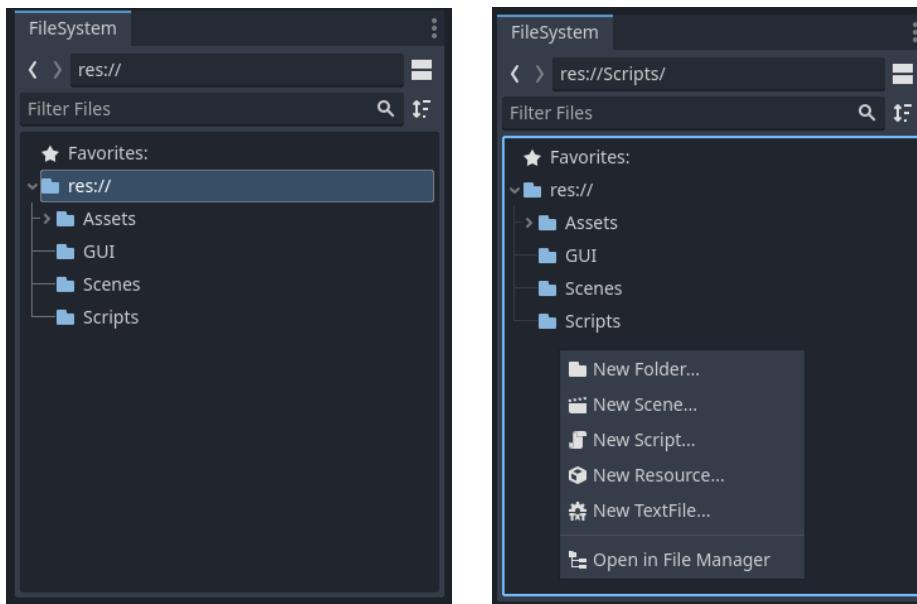
One thing to remember about game development is that this is not supposed to be a linear journey. Everyone goes about this differently, but if you're interested in what my steps usually look like, you can view this example of the Dusty Trails project's steps [here](#) and use it as a boilerplate for your own game (or don't, it's 100% up to you). I'm also using [this color palette](#) that I found online for this game, just in case you were wondering where I got my color scheme values from!

Now, onto the setup. The assets you will need for this section can be downloaded [here](#). The credits for these free assets can be found on my [GitHub repository](#). Once you've downloaded your assets, extract the file, and drag it into your FileSystem panel. We can

also go ahead and delete the default icon.svg and README.MD files that are present in your project.

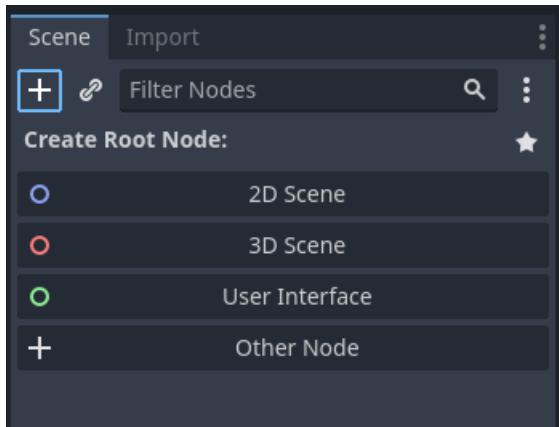


Right-click on the panel and let's create three new folders: Scenes, GUI and Scripts. The Scenes folder will contain all of our .tscn files (player scene, NPC scene, map scene, etc.), the Scripts folder will contain all of our .gd files, and the GUI folder will contain all of our UI element scenes (menus, popups, HUDs, etc).



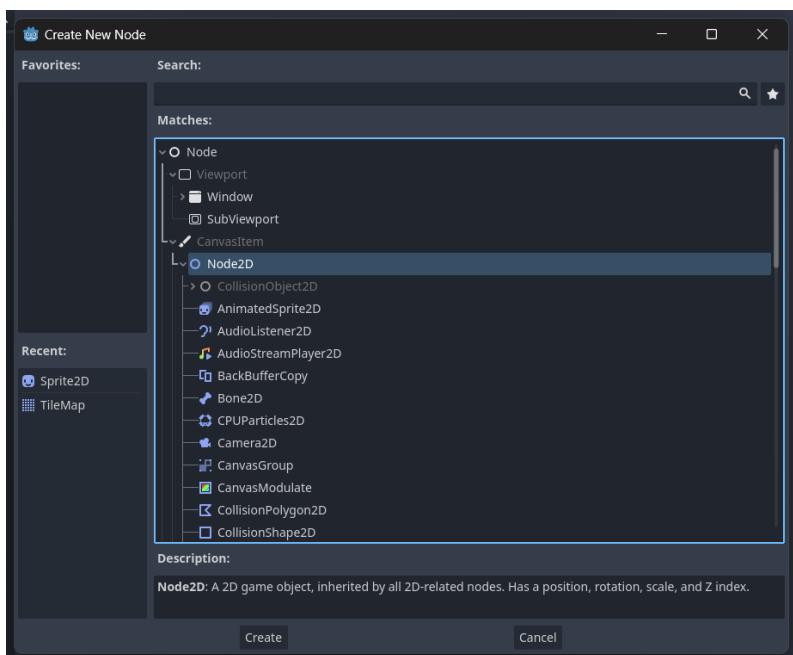
In the Scenes folder, let's create our first scene, which will be our Main Scene. The Main Scene is the scene that will serve as the core container for our game elements. It is here where we will add our map, characters, and items so that we can play the game. The Main Scene is also the one that runs automatically when the game launches.

To add a scene, you can right-click on the FileSystem panel and add it like we added our folders, or you can click on the plus icon in our Scene Dock. This is the quicker way, so I will do it that way from here on out.

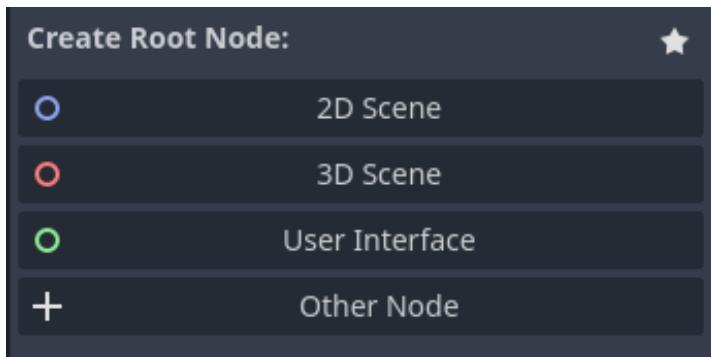


You will see that it will prompt you to select a node. The node that you choose here will be the root or parent node. The root node is the highest node in the tree structure and has no parent.

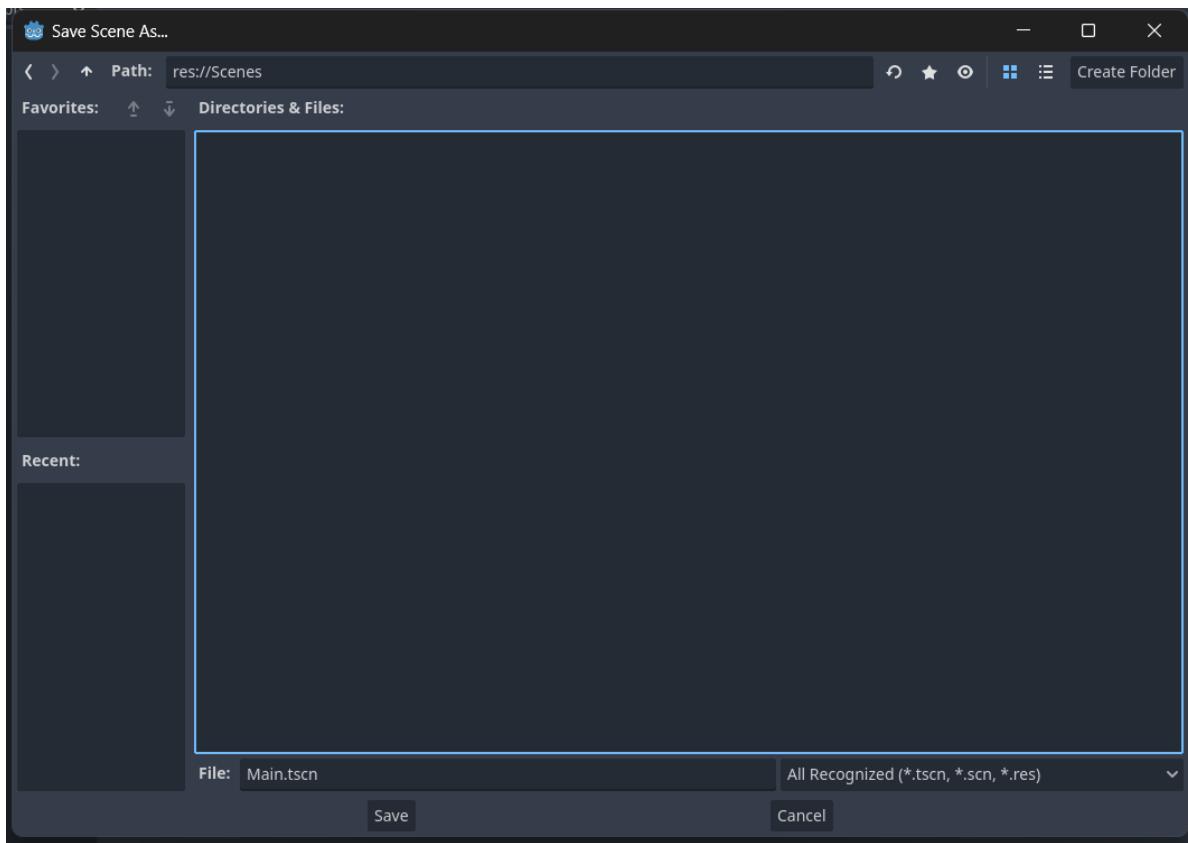
We can always change this node later on, so don't worry if you choose the wrong one! We need to select [Node2D](#) as the root node. The Node2D node is the base node for a 2D game. We usually use this node in our Main scene, or any other scene/hierarchy tree that holds other nodes.

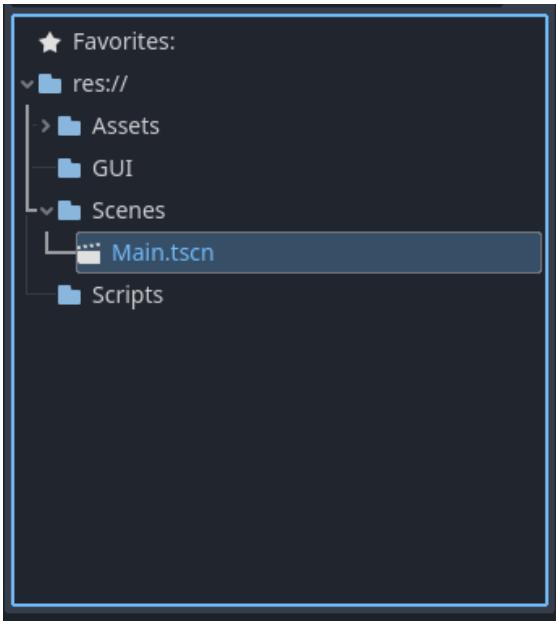


You can also automatically choose this node to be your root node in the Scene Dock "2D Scene" option.

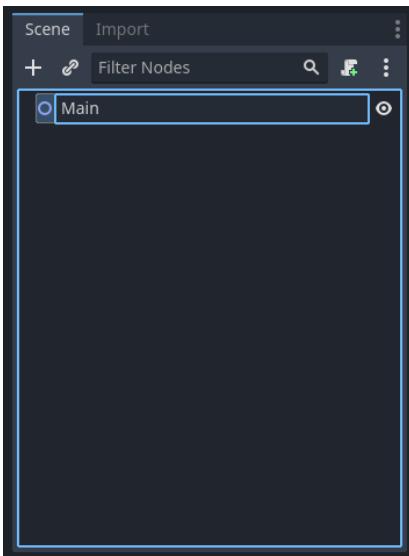


You will now see that a Node2D node has been added as our root, but our Scene hasn't been saved yet. Save it in the Scenes menu above or use the shortcut CTRL + S. Save the Scene as "Main" in your Scenes folder.

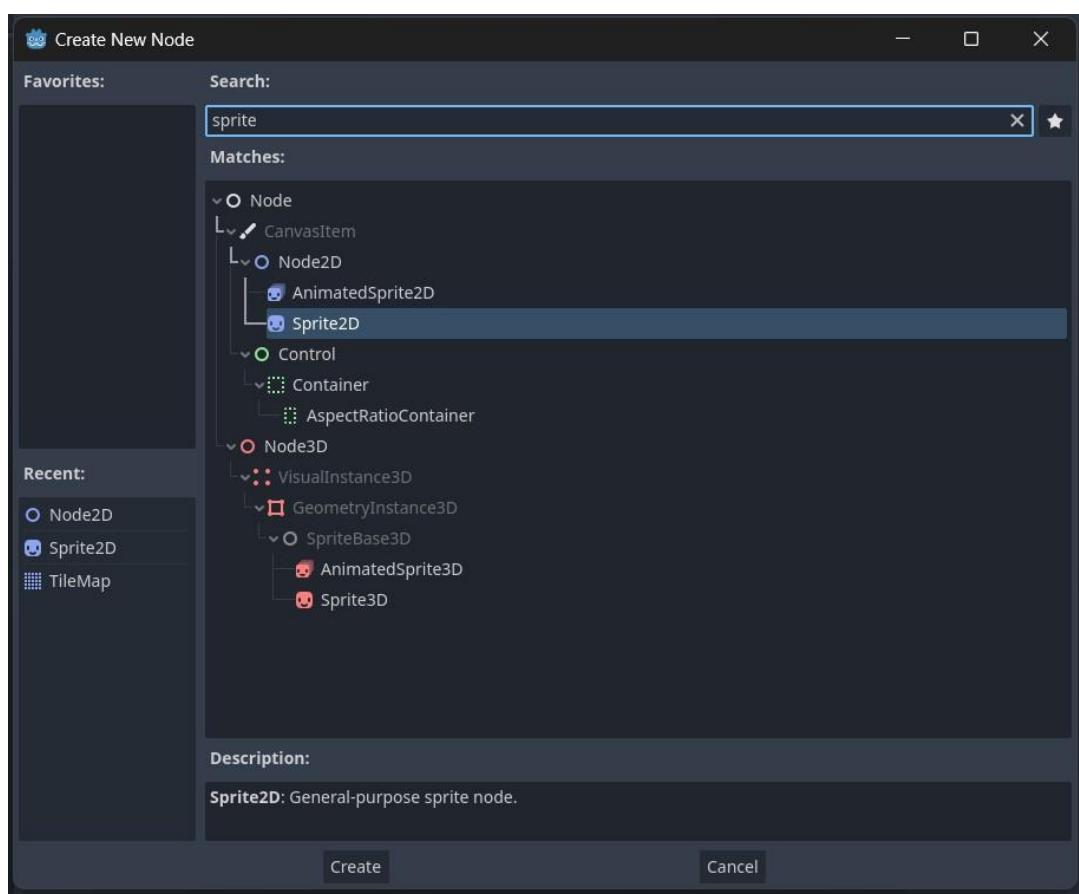
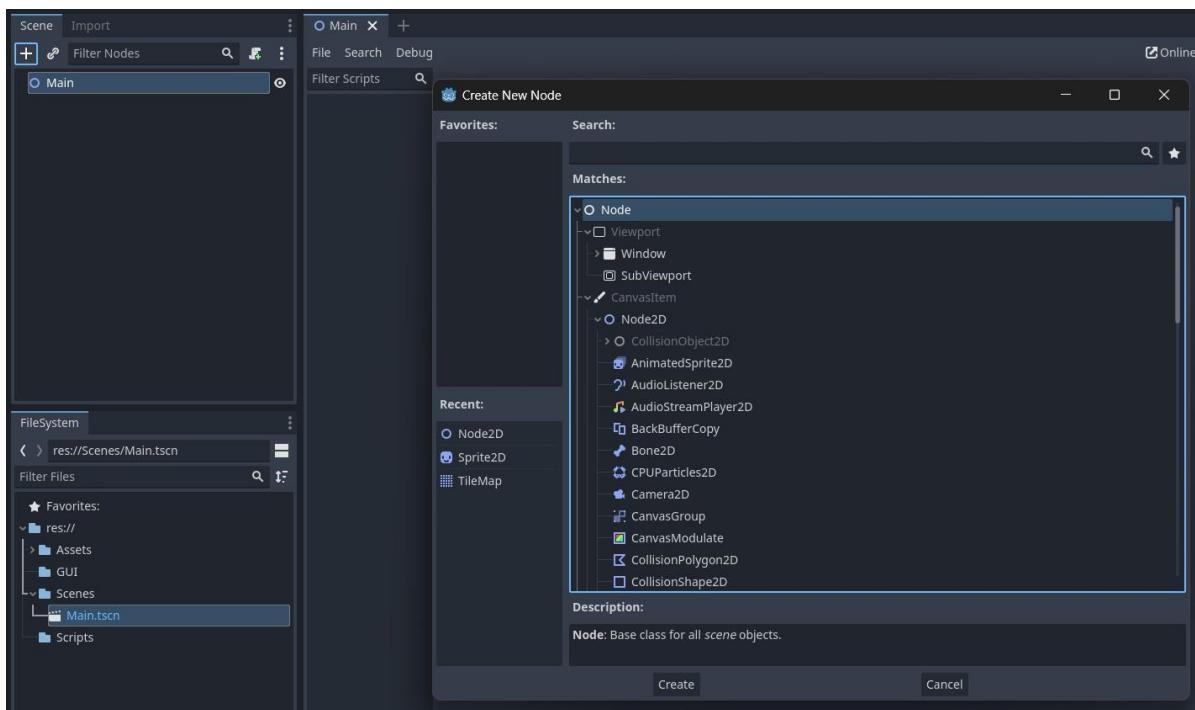


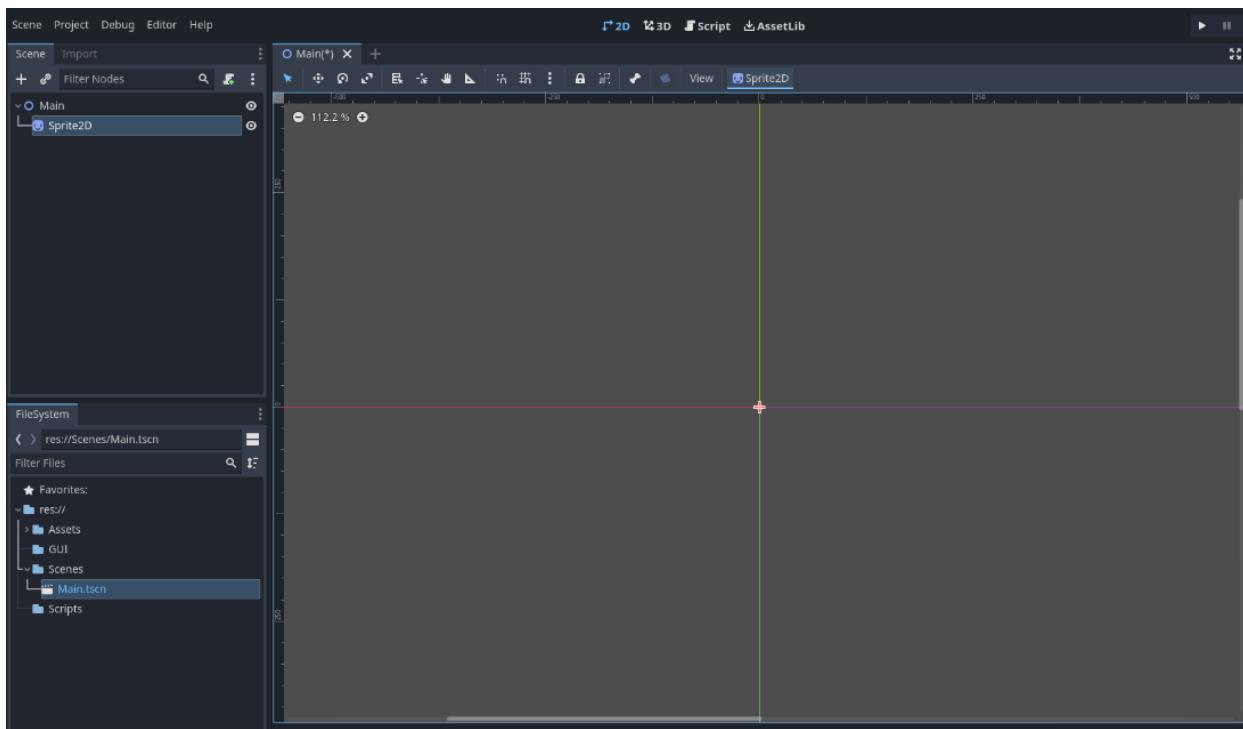


Double-click on the Node2D root node to rename it. Let's rename it to match the name of our scene, just for simplicity.

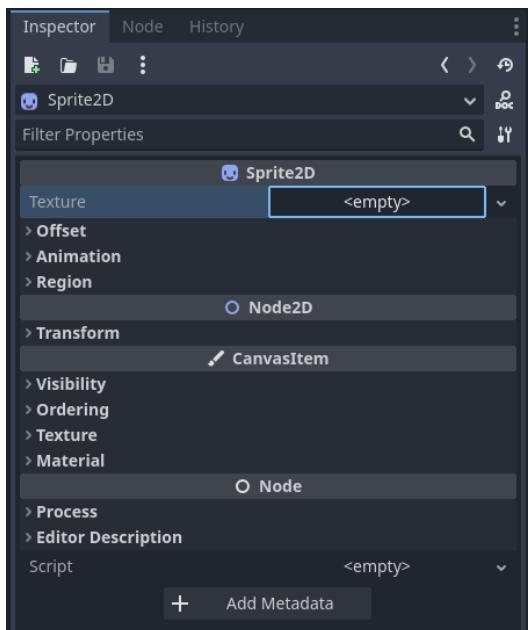


Now we can add children nodes to our root node. You can do this via the plus icon again. Remember, we are making a 2D game, so nodes that end with "3D" will not work with our node! Let's add a Sprite2D node for now because I want to show you something. Take note that any node you add will be added to your recent panel in the node menu, so you can easily select it from there next time without having to search for it!

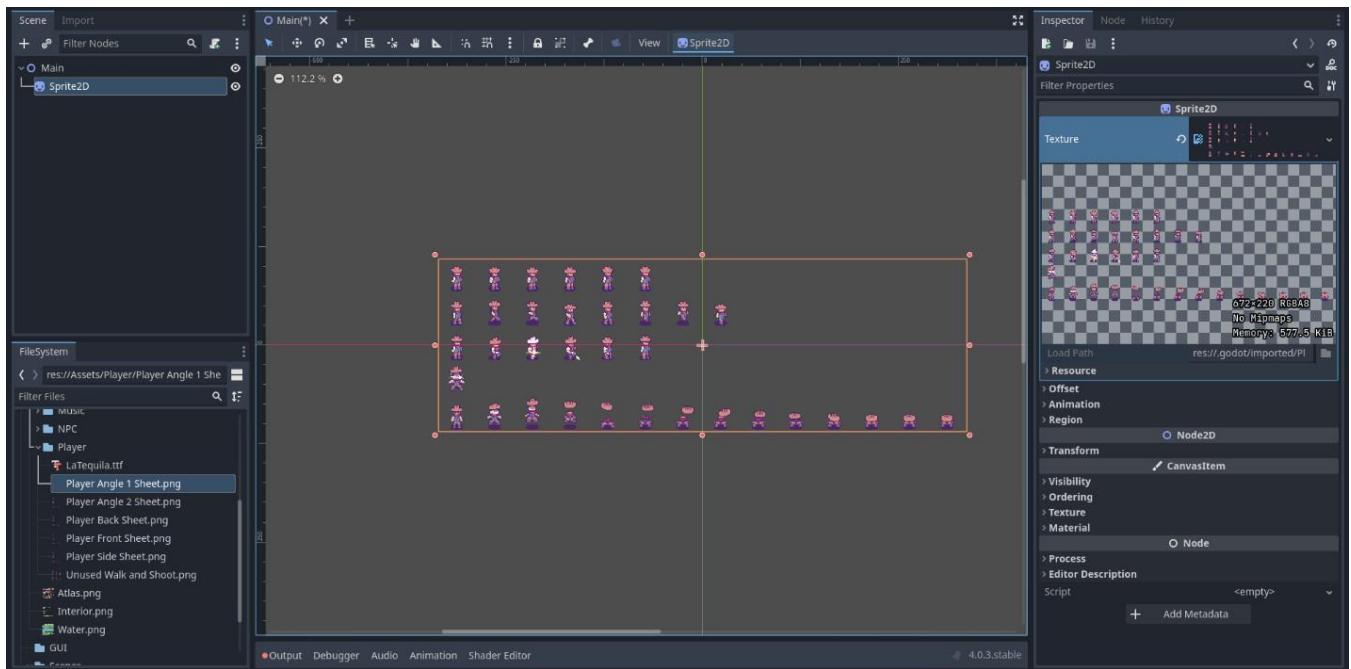




In the 2D workspace (because we are making a 2D game), select your Sprite2D node and press F on your keyboard to focus on your selected node. With your node selected, in the inspector panel, you will see the option for texture. This is our sprite's image.



If you go into your assets folder, you can drag any image you want into that texture property.



If you zoom in on your sprite, you might notice that it is a little bit blurry. This won't do, but luckily, I have a fix for you.

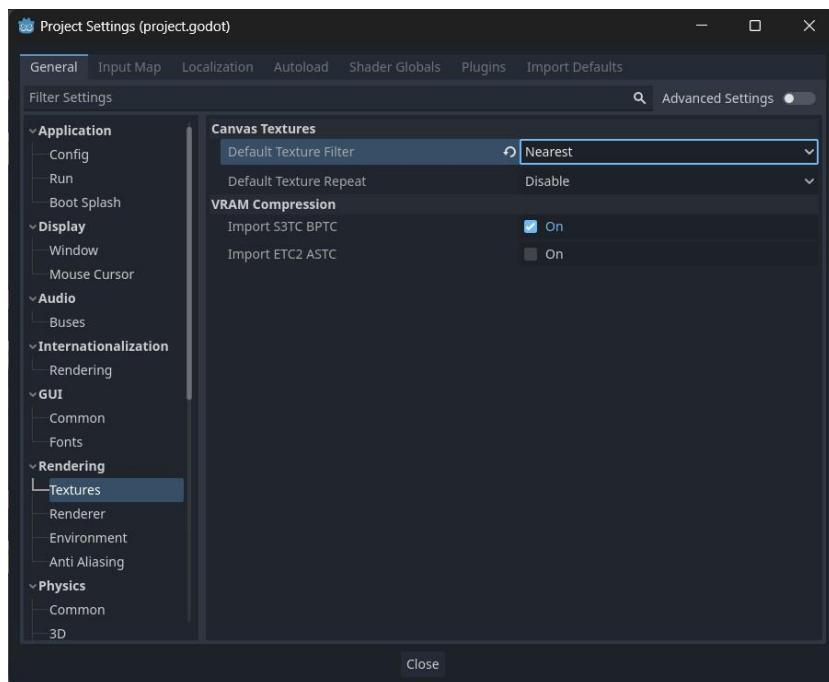


In your projects settings option underneath Project > Project Settings, change your Textures Rendering from Linear to Nearest. This will scale up our texture rendering so that image assets are always high quality.

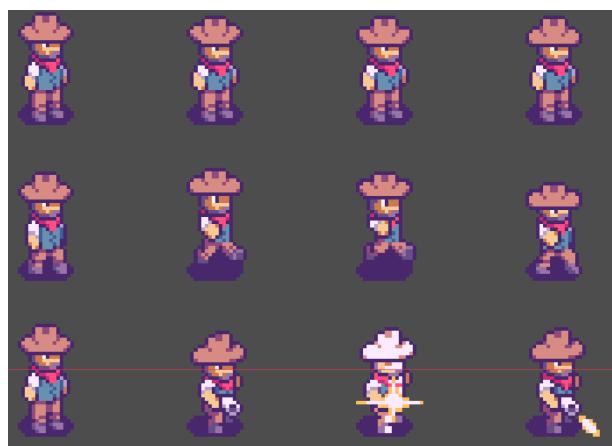
What is texture filtering?

Texture filtering is a technique used in computer graphics to improve the appearance of textures when they are applied to 3D models or 2D images. It becomes particularly important when a texture is scaled, rotated, or otherwise transformed.

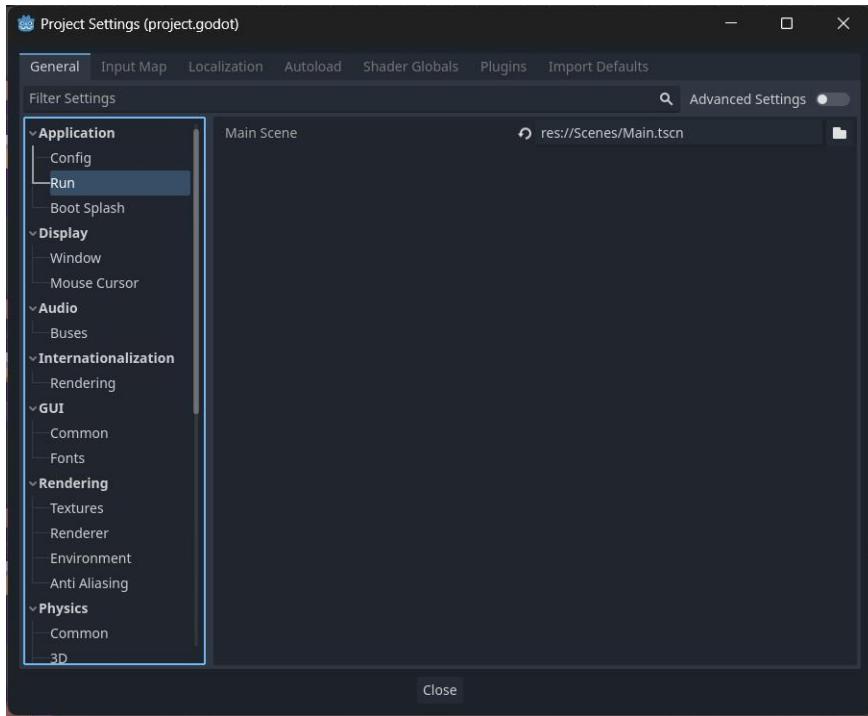
Choosing the correct texture filtering mode is essential for improving image quality, performance, and the overall visual experience in both 2D and 3D graphics.



Your sprites should now be clear.



The last thing we need to do for this part is set our Main scene as well, our main scene. In your Project Settings, underneath Application > Run, you can set your Main Scene here to be our newly created scene. This will then run this scene whenever we run our game. Later on, we will change this to be our menu scene, because you want the player to be greeted by the start menu instead of just throwing them into the game.



Congratulations, you've learnt how to navigate through the Godot Editor, as well as work with nodes and scenes - which are the core fundamentals you'll use throughout your game. It would be good practice for you to create your own [GitHub repository](#) for your project so that you can make regular backups of your project.

In the next part, we will set up our game character, i.e., our player, as well as add the functionality to move them around our game. That means more nodes, and you'll get to work with Scripts for the first time! Remember to save your game project, and I'll see you in the next part.

*You can also use the official [Git Plugin](#) from Godot, but as of July 2023, the Git plugin hasn't been updated to work with Godot 4.1 and later yet.

PART 2: PLAYER SETUP & MOVEMENT

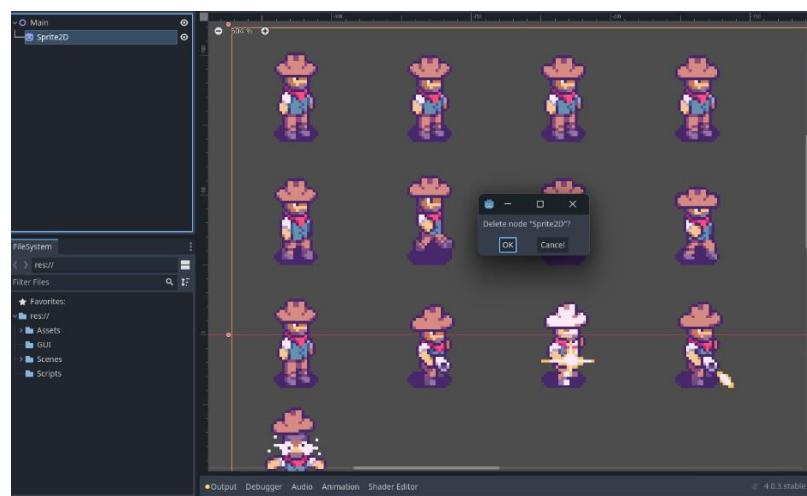
Now that we have our project setup, we can now go on to creating our player scene. Our player scene will contain nodes and scripts that will allow us to see a character and control them so that we can run around and interact with the world.

WHAT YOU WILL LEARN IN THIS PART:

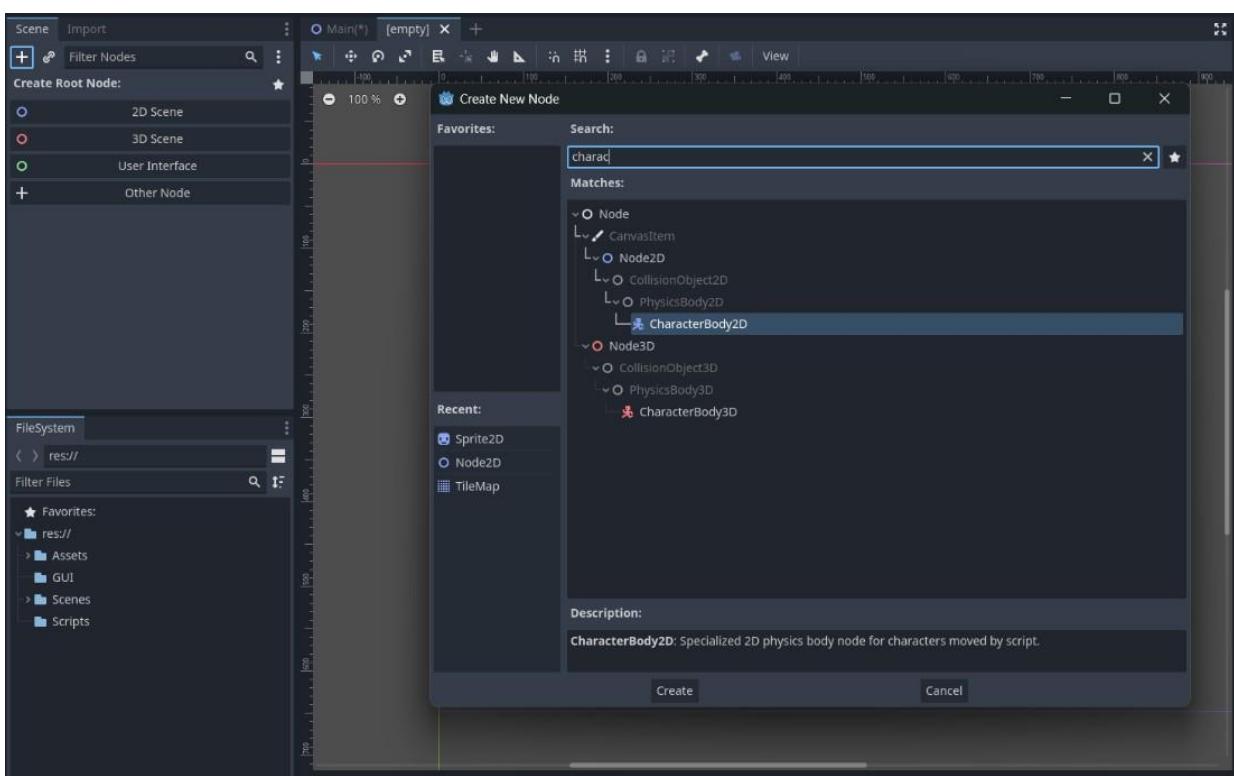
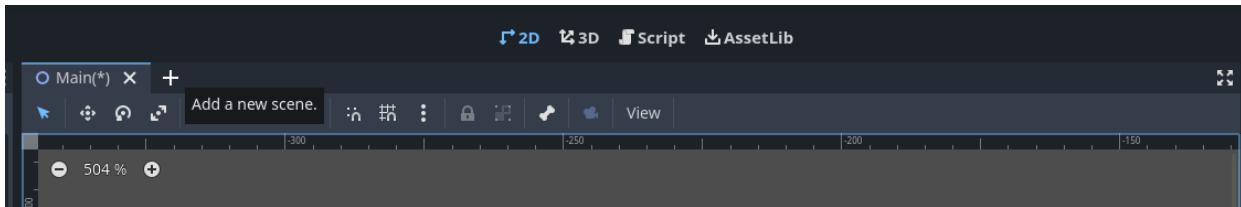
- How to create, run, and instance new scenes.
- How to add input actions, collisions, and animations to a node.
- How to add scripts and code to a scene.
- How to add movement to a node.

PLAYER SETUP

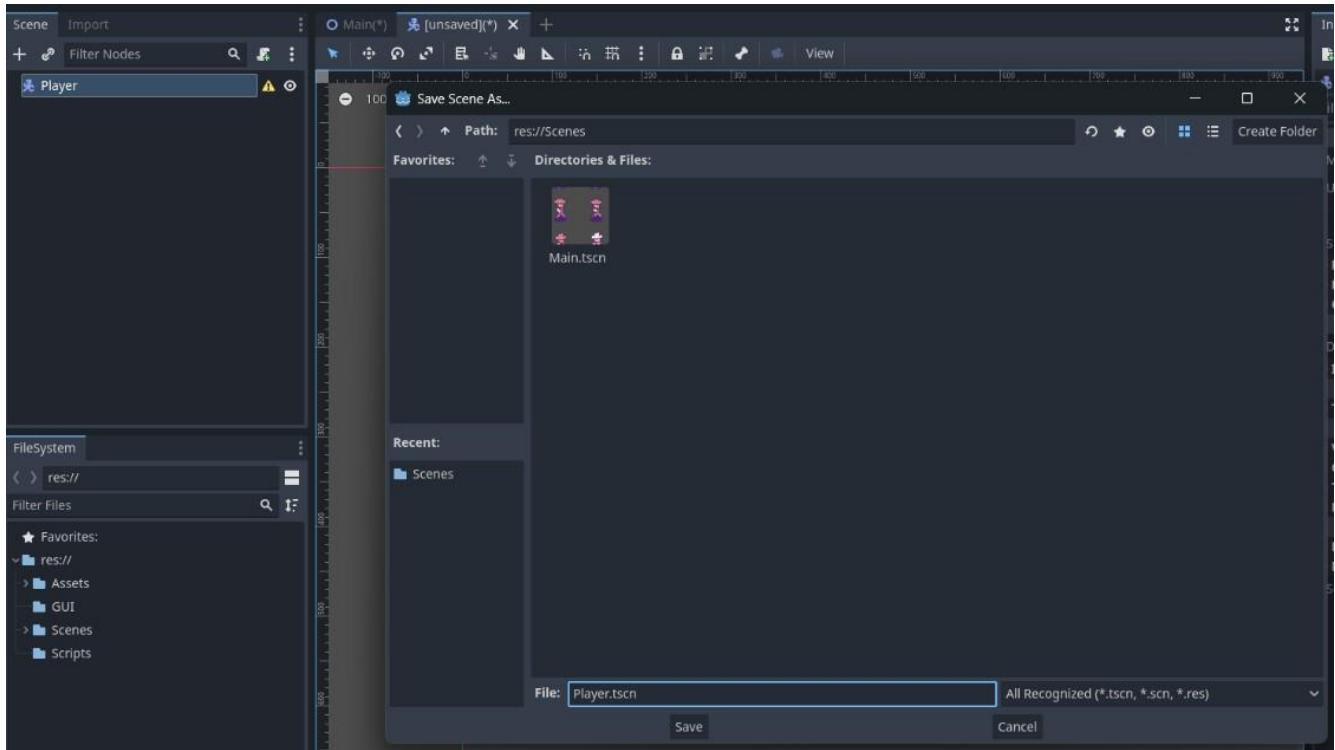
In our Main scene, we can go ahead and delete that Sprite2D node that we added earlier. We will create a whole new scene for our Player, and then [instantiate](#) it in our Main scene. Instancing allows us to replicate an object from a template so that we can modify them separately. In other words, we will have our Player separate from our Main scene, so we can treat it as a separate object. You'll see how this works more clearly in a minute.



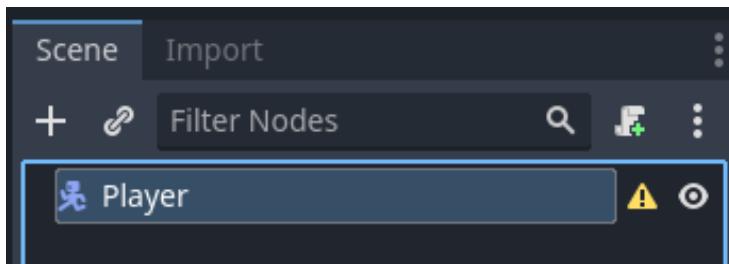
In your workspace, click the plus icon next to Main(*) to create a new scene. In the Scene Dock, add a new root node. We need this root node to be a [CharacterBody2D](#) node, which is a specialized 2D physics body node for characters that can be moved via scripts. We use this node for entities that need physics processing for movement.

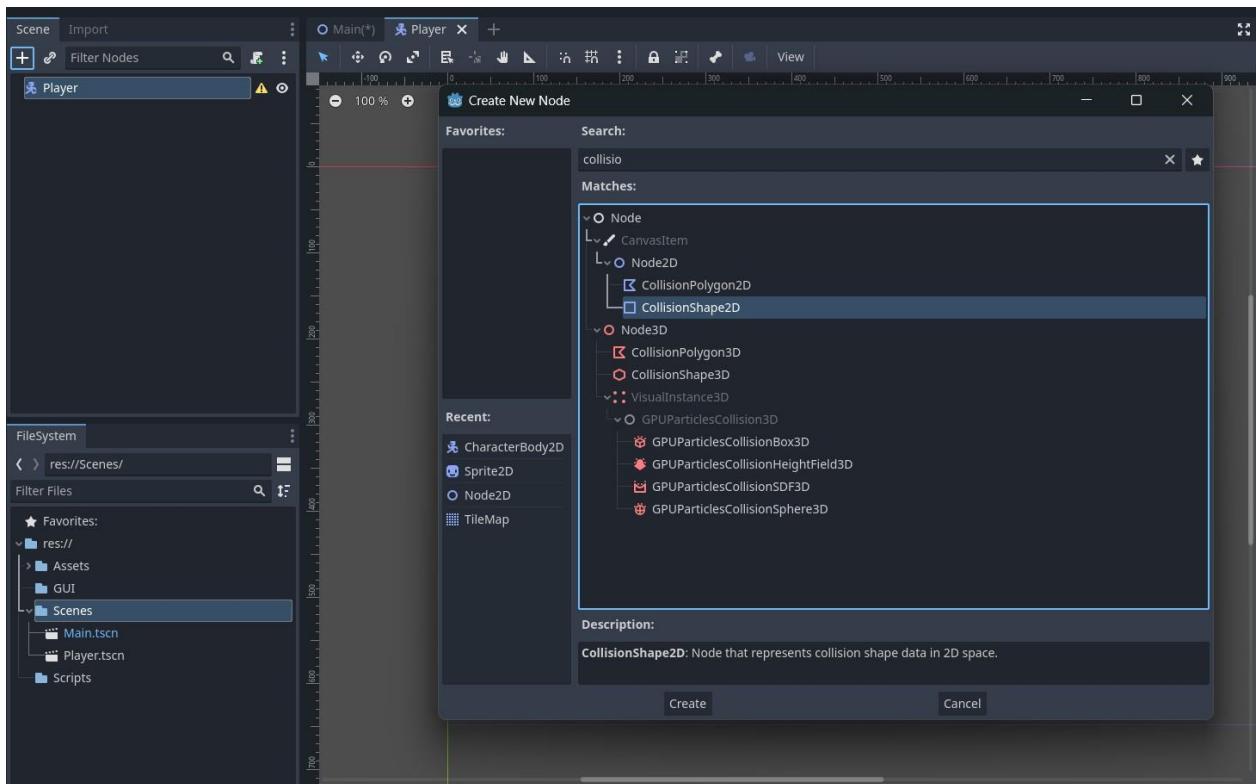
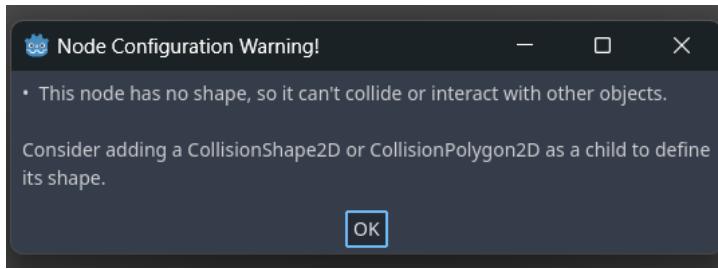


Double-click the CharacterBody2D node to rename it to "Player". We can also go ahead and save our new scene as "Player" in our Scenes folder.

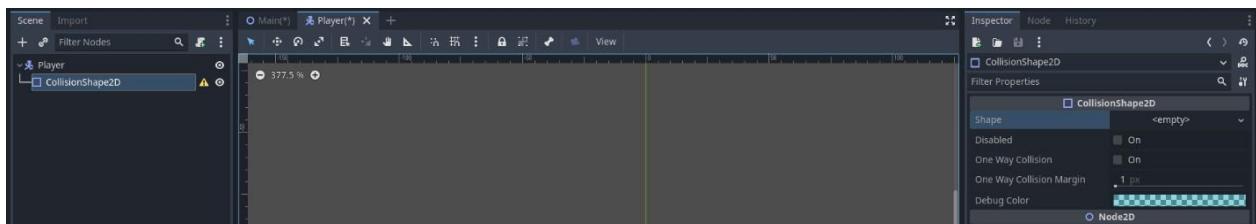


You will notice that there is a warning icon next to your root node. If you hover over it, you will see the warning message. It tells us that our character has no shape, so it cannot collide or interact with others. To fix this we need to add a [CollisionShape2D](#) node. This node will allow our player to collide with other collision bodies which will block, damage, or trigger events in our player.



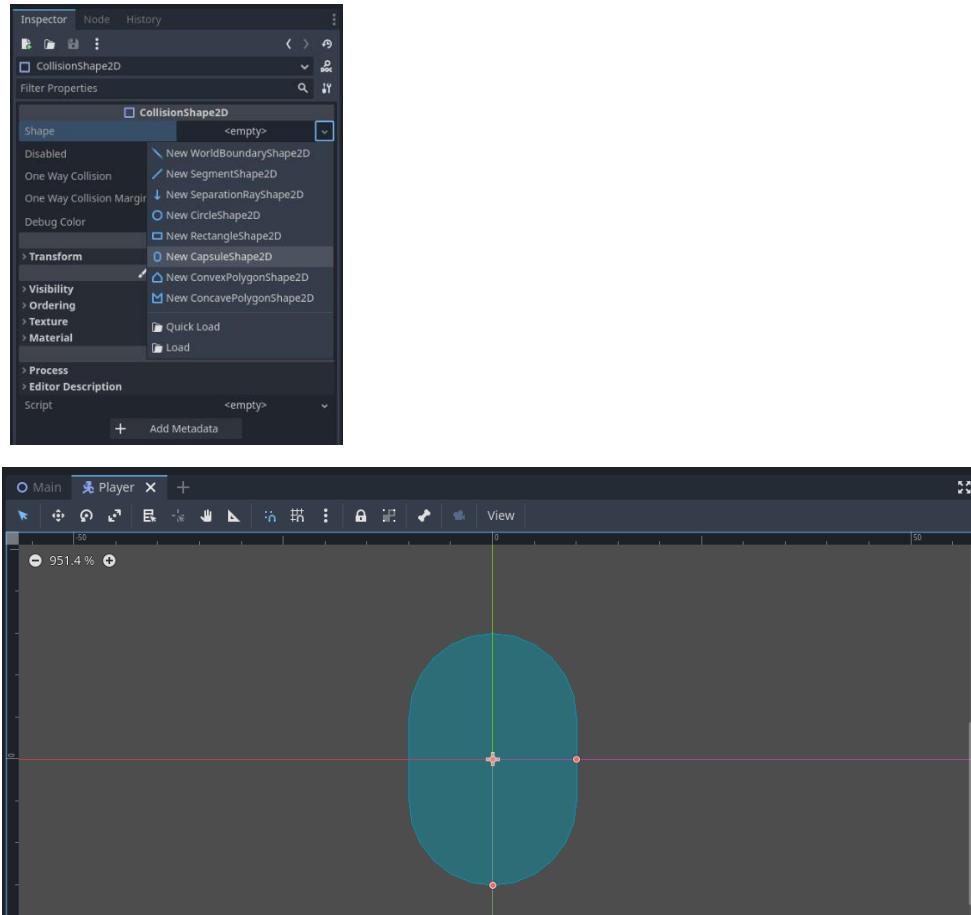


This node will also return a warning message because we haven't assigned it a shape in the Inspector panel yet.



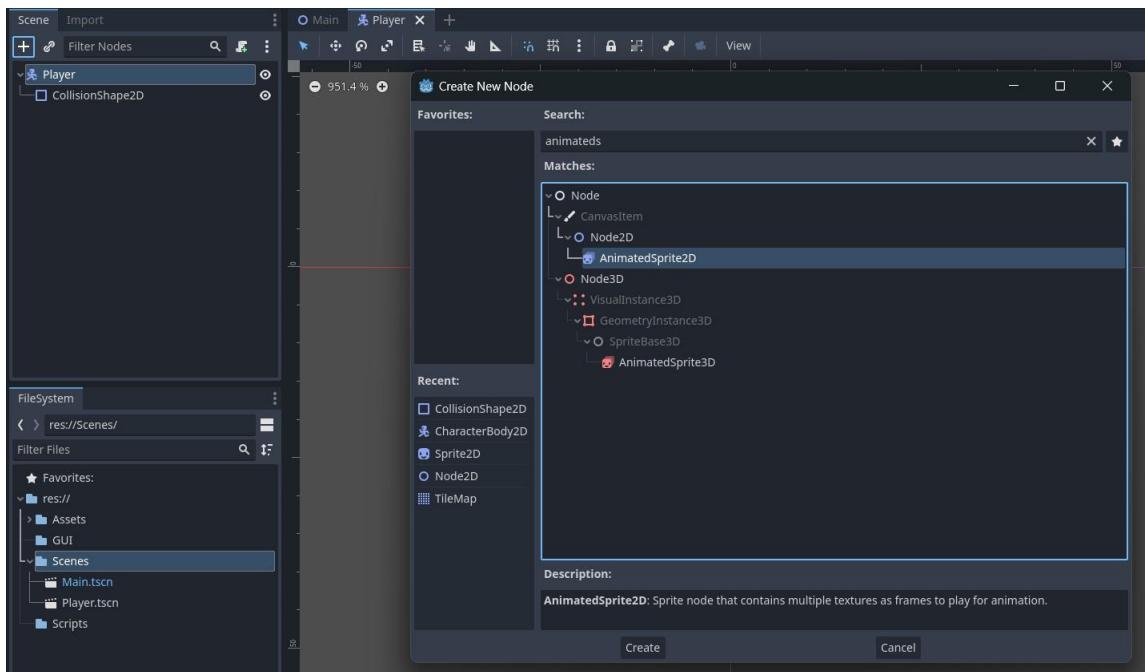
Let's fix this by clicking on the `<empty>` box next to Shape and selecting a shape. The type of shape that you choose for your object will depend on what you want them to accomplish (from where they need to collide), as well as their body shape. So, a circular

character might have a circle collision, and so on. For our sprite, let's choose the CapsuleShape2D shape. You will now see that we have a collision shape, so we can collide with other objects in our game, but we don't even have a body yet!

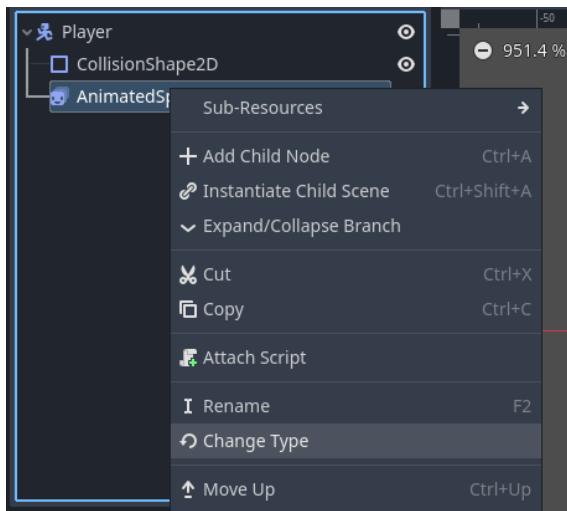


To see our character, we need to assign it to a Sprite. Now, we can go about this two ways, via the [Sprite2D](#) node, and the [AnimatedSprite2D](#) node. The Sprite2D node is perfect if we want to have a static object, i.e., an object that doesn't move or that has no animations. The AnimatedSprite2D object, however, is the exact opposite. It is perfect for when we have an object that requires sprite sheets to be animated.

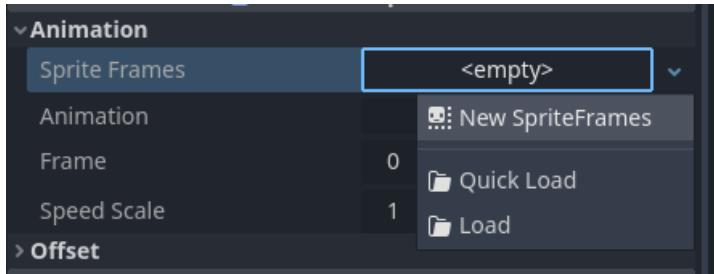
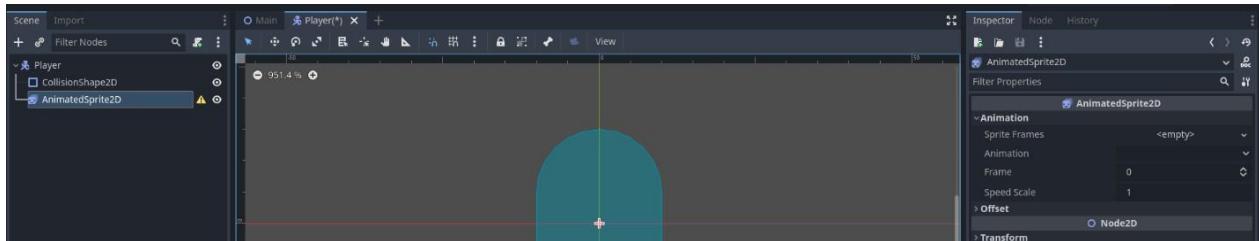
With your Player's root node selected (make sure you select it and not your CollisionShape2D node because otherwise, it will add our node to it as its child), add a new AnimatedSprite2D node.



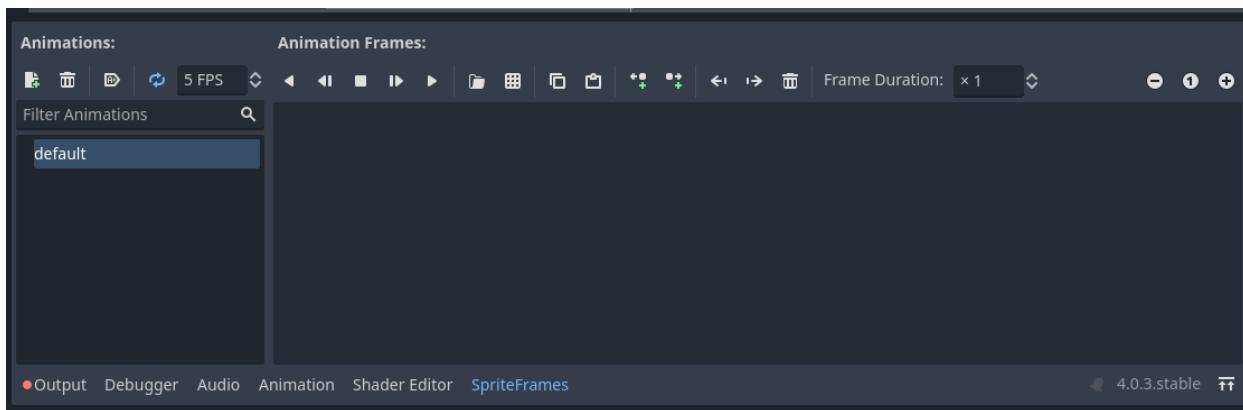
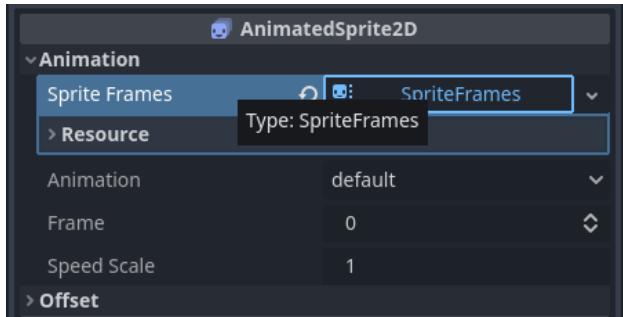
If you ever choose the wrong node, right-click the node you want to replace and say change type.



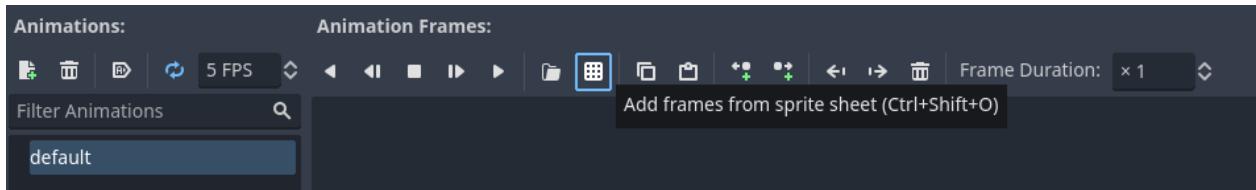
This node will once again return a warning, and that is because it has no sprite sheet assigned to it, so it cannot return animation frames. Let's get rid of this warning by going into our Inspector panel, and underneath <empty> next to Sprite Frames, select New SpriteFrames.



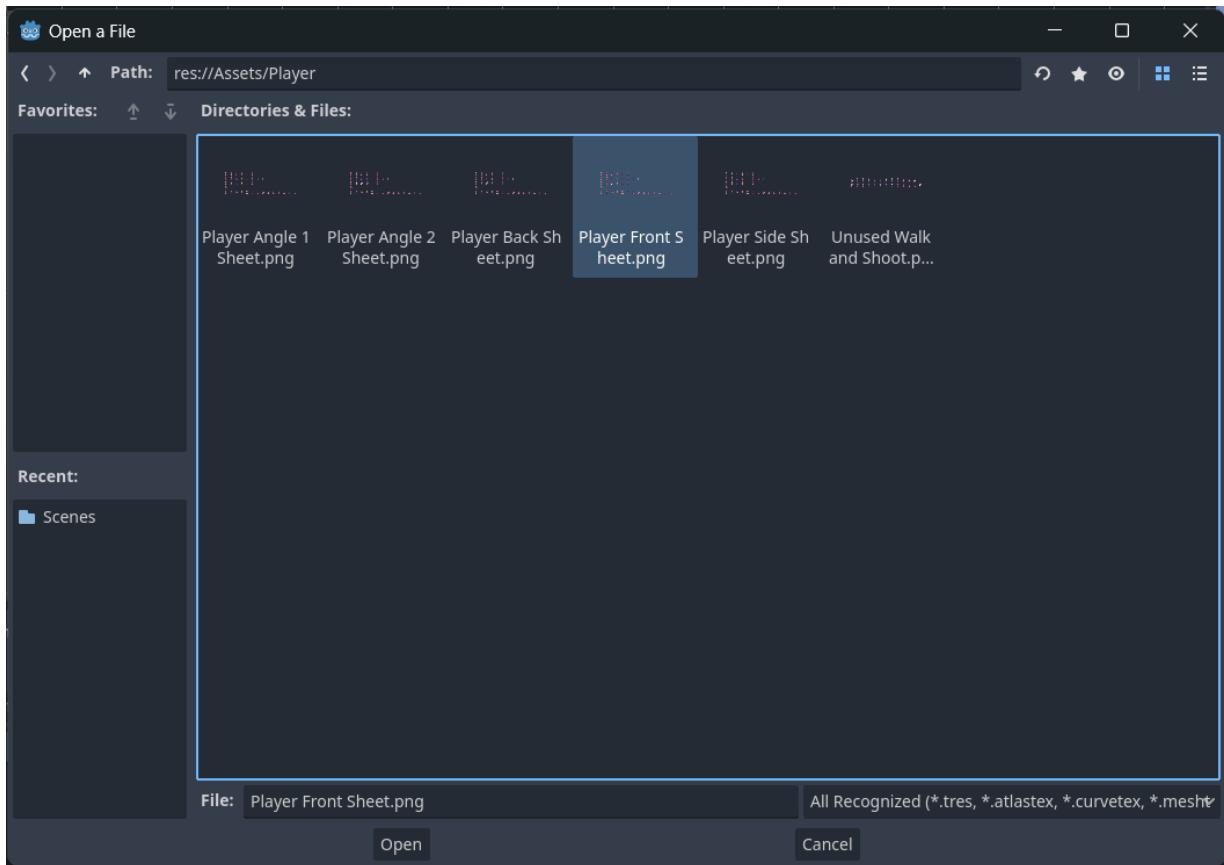
This SpriteFrames resource allows us to import image files (or a folder containing said files) to provide the animation frames for the sprite. To configure our SpriteFrames resource, click on it, and you will see a new option pop up in our panel below.



It is in this panel where we can add new animations, delete animations, play animations, or edit animations. The default animation can be deleted or renamed. We will add all of our player's animations in the next part so that we can explore this node in more detail. For now, let's just assign a sprite to our default animation by clicking the "Add Frames from Sprite Sheet" option.



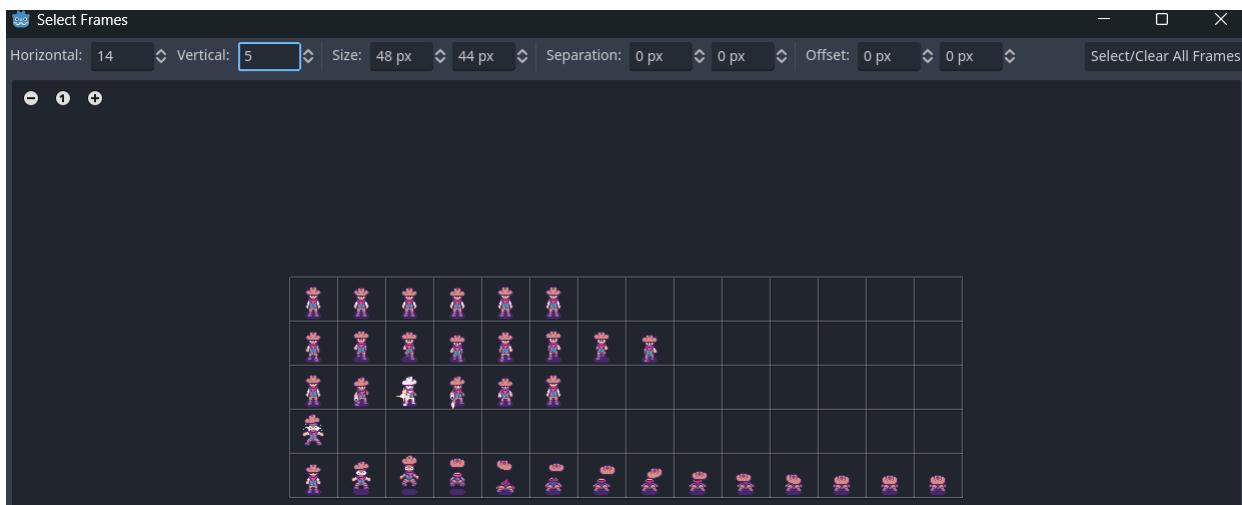
A window will pop up so that we can select the Sprite Sheet we want to add, so head into Assets > Player, and select the Front Sheet.



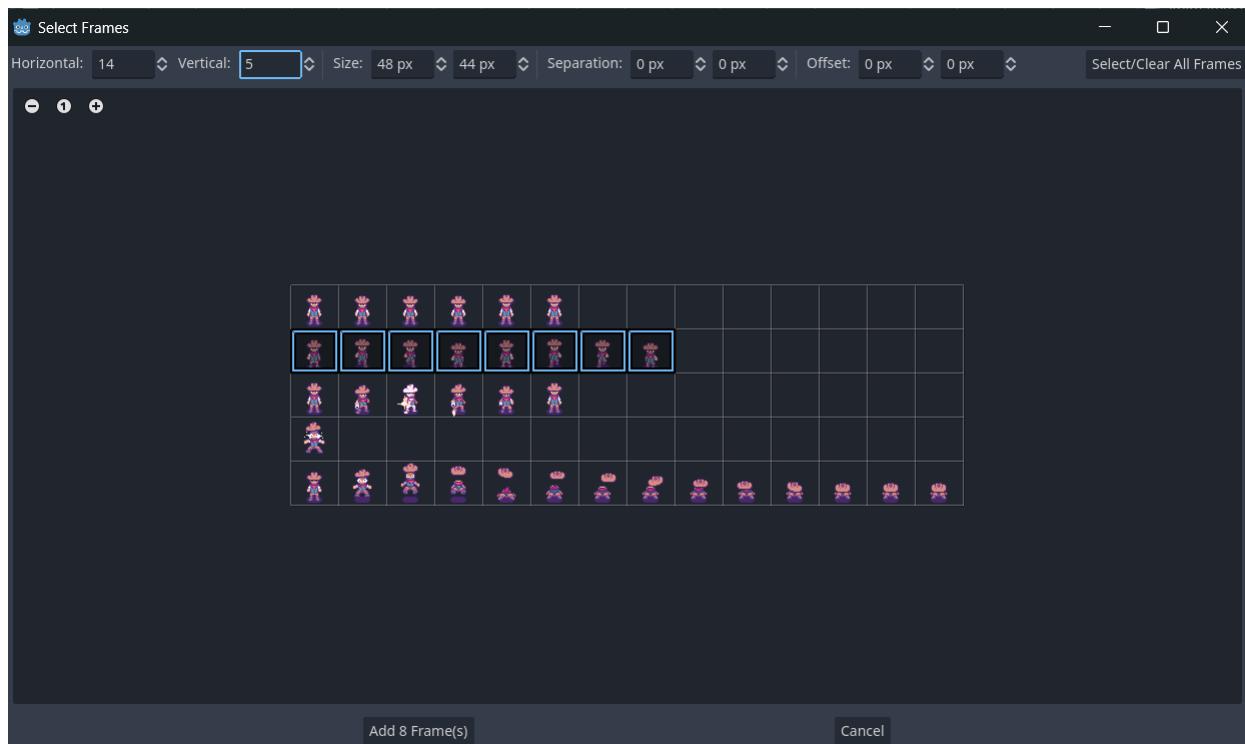
Another window will pop up, and it is here where we need to crop our individual sprites to make an animation. The default horizontal and vertical values are 4. Horizontally, we can count that our player frames go up to 14, so change the horizontal value to 14.



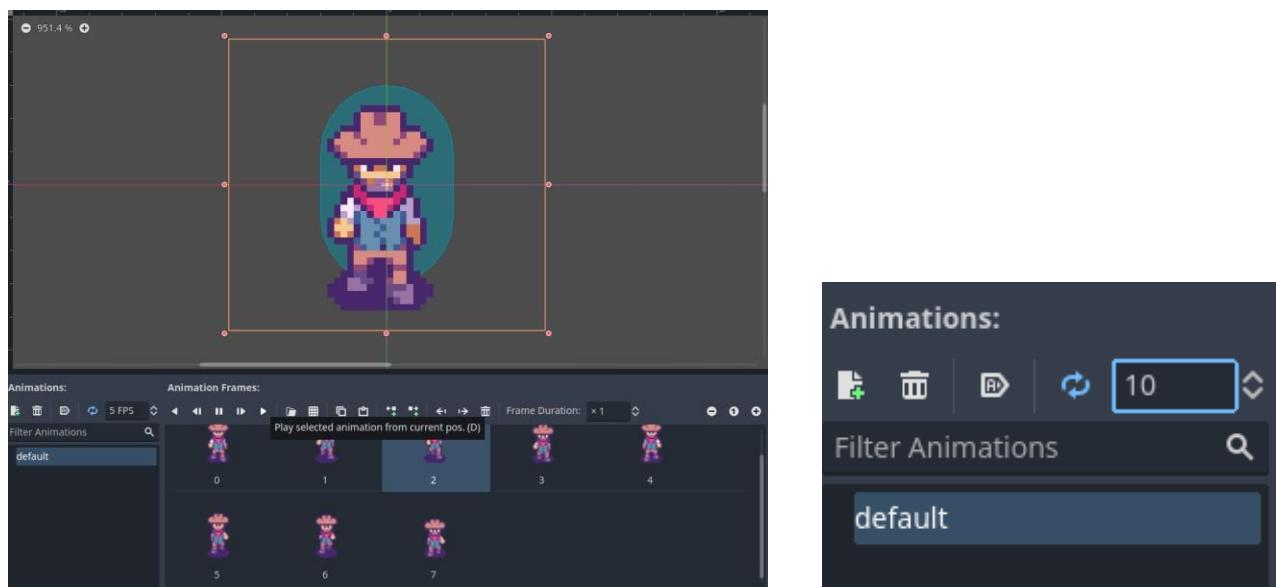
Vertically, we can count that our player frames go up to 5, so change the vertical value to 5.



Now we can select the frames we want (in order) to create one animation. This sheet has five possible animations: idle, walk, attack, damage, and death. For now, let's just select the second row to create a basic walk animation. Make sure you select it in order, otherwise, you will have to rearrange it in the SpriteFrames panel.



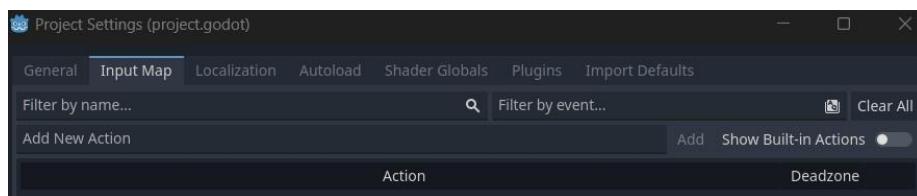
Our default animation now has 8 frames added to it. If you press play, we can see the animation play in our workspace. Our player walks a bit slow, so let's up their FPS value from 5 to 10. Leave the looping value to be enabled.



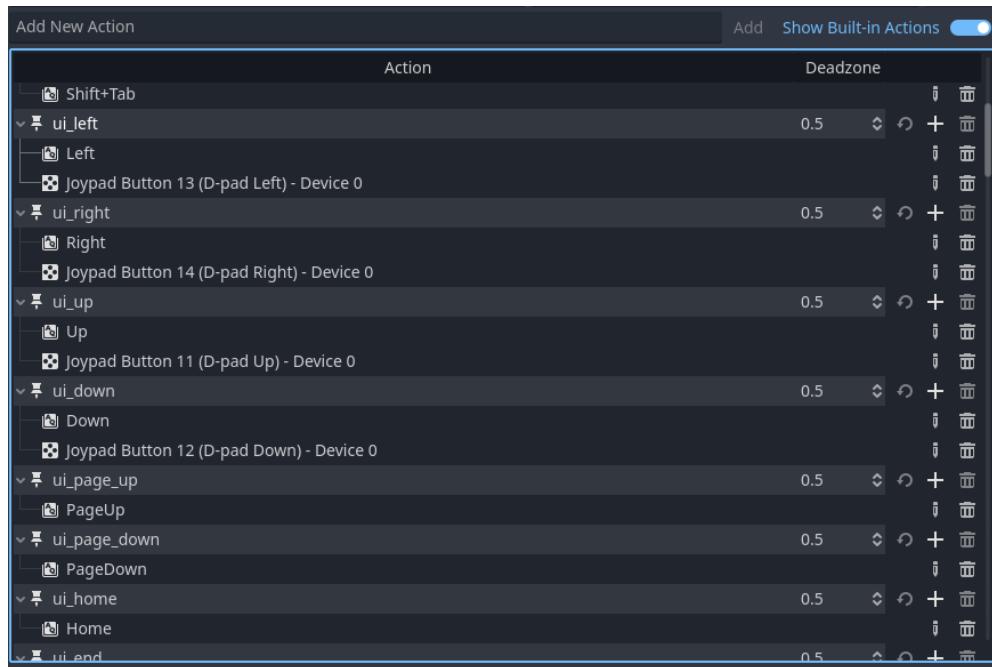
MOVEMENT INPUTS

With our walk animation added, we can now add the inputs and functionality to move them around. Before we can jump the gun and start coding, we need to add an [input action](#). This is the physical controls, such as our keyboard keys, joysticks, buttons, etc. that we will map to our functions to move our character around the game.

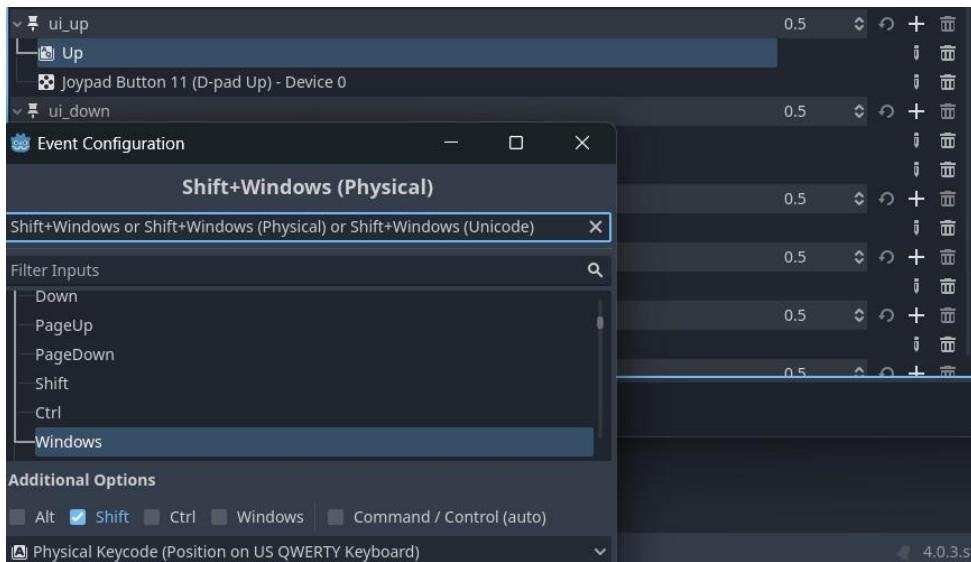
To add input actions, we go to Project Settings > Input Map.



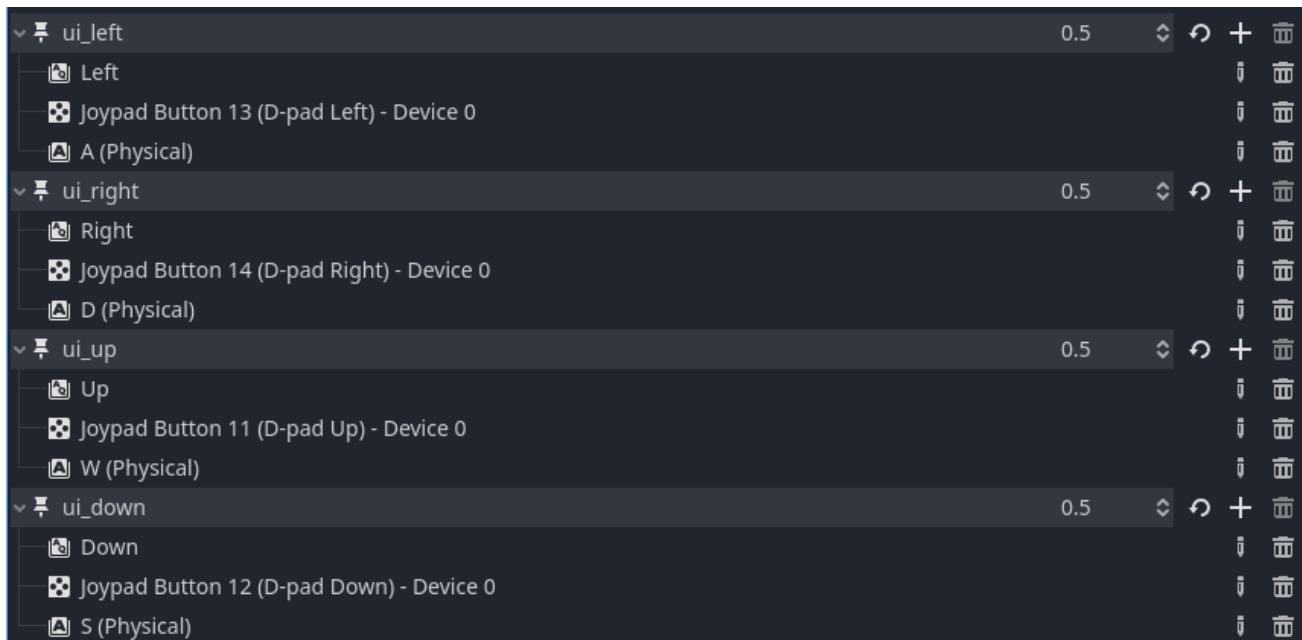
Godot 4 comes with pre-existing input actions. To see them, enable the "Show Built-in Actions" toggle. The `ui_` stands for "user input", and you don't have to stick to this naming convention for your future input actions, but for this project we will just keep our code consistent.



You'll see Godot has already pre-configured the inputs to move a character left, right, up, and down. They assigned the keyboard keys UP, DOWN, LEFT, and DOWN to these actions. You can edit them by pressing on the pencil icon next to them, or even delete them. You can also assign other keys to this input by clicking on the plus icon next to it.



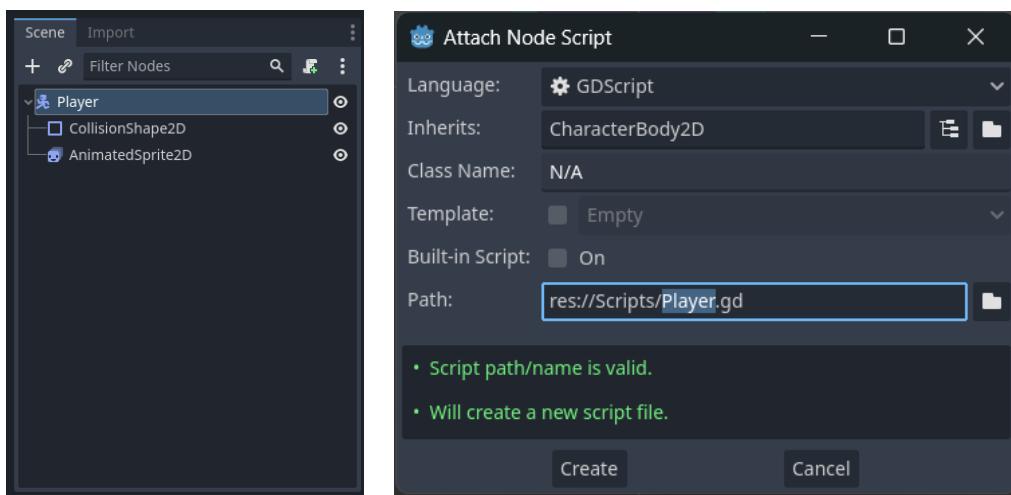
Let's assign our WASD keys to ui_up (W key), ui_down (S key), ui_left (A key), and ui_right (D key).



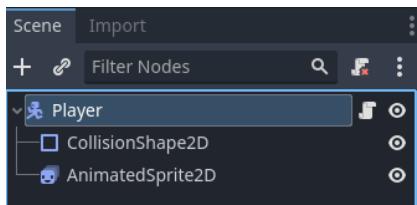
We will come back to add our own input actions for attacking, consuming, interacting, and sprinting later on. For now, there is nothing that we need to do here since most actions have already been added for us, so you can close this menu entirely.

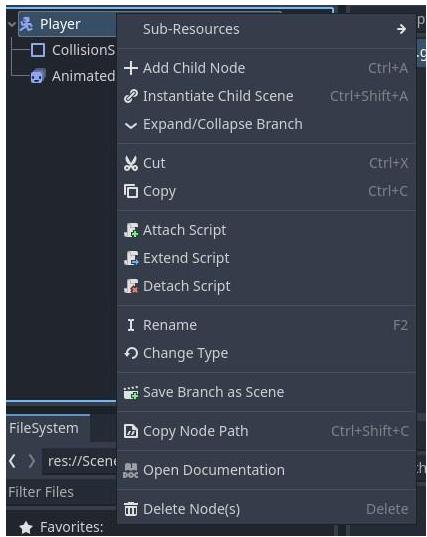
Let's now add a script to the player so that we can move them around with these inputs. You can assign a script to any node but remember that - that script will only impact that node and only that node's class objects will be callable. Except, however, root nodes. If you add a script to a root node, it will be able to impact and call the classes of all its child nodes.

To assign a script to a node, select the node you want to add functionality to (which in this case is our root node), and in your Scene dock, click the little scroll icon next to it. Save this script as Player.gd underneath your Scripts folder.



Once a node has a script attached to it, you will see the scroll image appear next to it. If you right-click on your node, you can detach this script and attach a new one. Click on the scroll to open your script, which will open the Script workspace.





At the top, it says "extends CharacterBody2D", because it is extending from the base class CharacterBody2D. That means all the properties and sub-functions of CharacterBody2D will be callable from this script.



Now to make our character move, we need to first define a movement speed variable.

```
### Player.gd
extends CharacterBody2D

# Player movement speed
@export var speed = 50
```

The `@export` means that we can export the variable to be editable in the Inspector panel. So, with your node selected, you can now change the speed value in the properties without having to assign it a constant value. This comes in helpful for when we create things like enemies and want to instance them multiple times in a scene and want them all to have different movement speeds.



Next up, we will use our `_physics_process(delta)` function, which processes the character's movement physics constantly. In this function, we need to get the player input direction (left, right, up, down). This input needs to be normalized for smoother diagonal movement because our player moves on a cartesian plane level. This means they need to move 1 space up, down, left, or right, and normalization will set the vector to the same length in any direction.

When to use `_process()` and `_physics_process()`?

Use the `_process()` function for things that are graphical or need to respond quickly.

Use the `_physics_process()` function for things that are physics-based or need to happen at a consistent, predictable rate.

What is a Vector?

Vectors are objects that represent quantities like force, velocity, and position. In 2D games, we use Vectors to determine and calculate the position of entities on the X and Y axes.

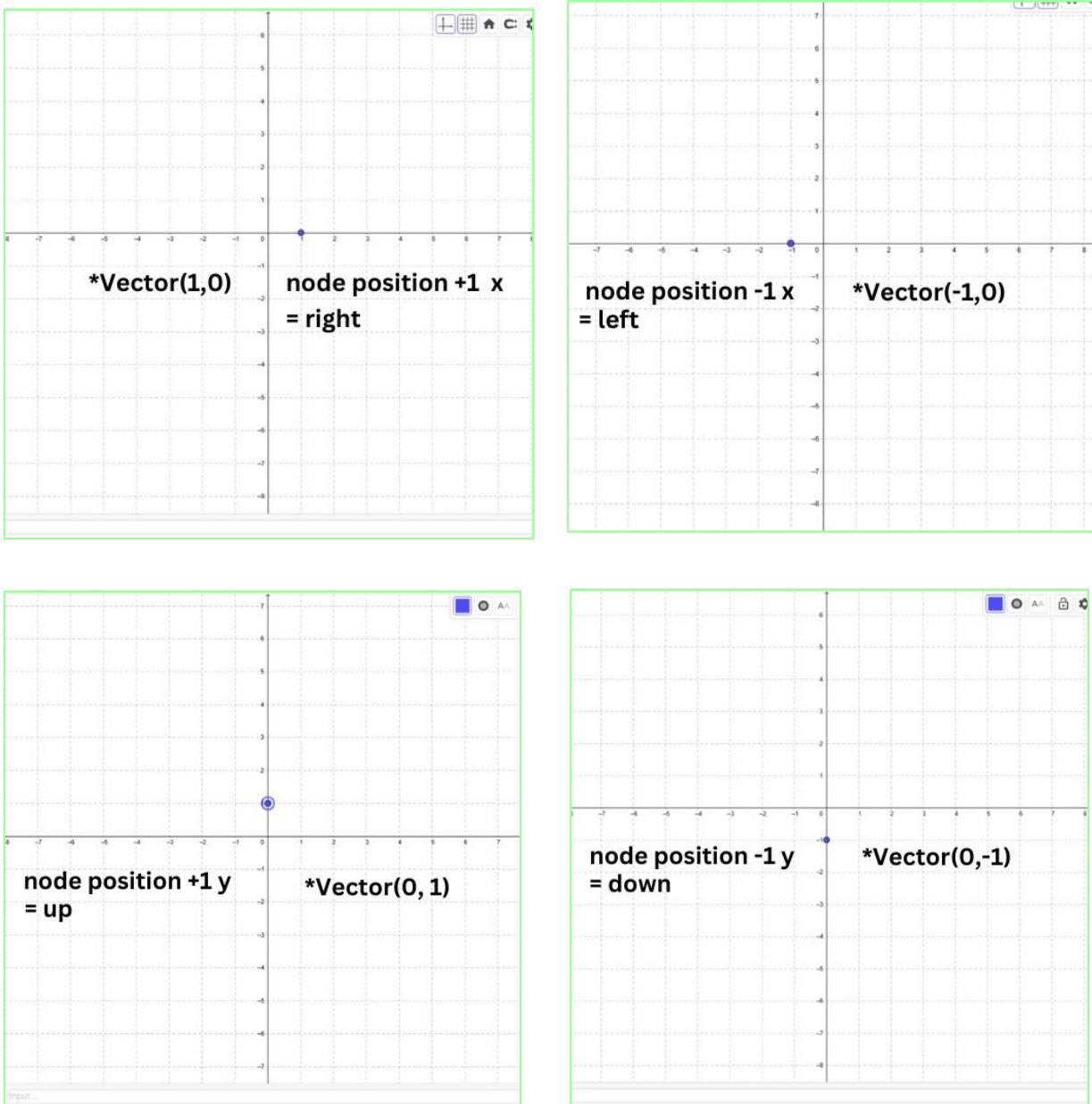


Figure 5: What vector movements look like on a plane scale.

We will then use the built-in [move_and_collide\(\)](#) method from Godot to move and to enable the physics to move our player around, whilst also enforcing collisions so that they come to a stop when bouncing into other objects.

When to use move_and_slide and move_and_collide()?

Use move_and_slide() when you want the character to move in general directions, such as in platformer games.

Use move_and_collide() when you want detailed information on your character's collisions to perform custom actions, such as in puzzle games where characters need to move or dodge obstacles.

```
### Player.gd

extends CharacterBody2D

# Player movement speed
@export var speed = 50

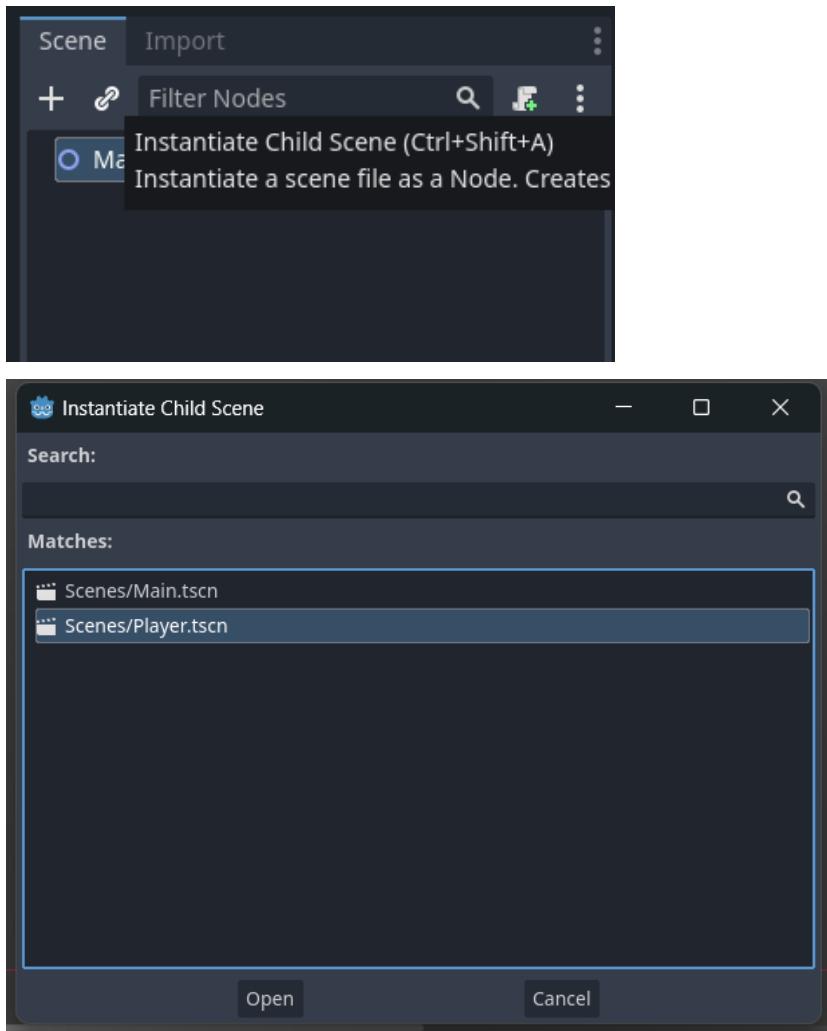
func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
        Input.get_action_strength("ui_left")

    direction.y = Input.get_action_strength("ui_down") -
        Input.get_action_strength("ui_up")

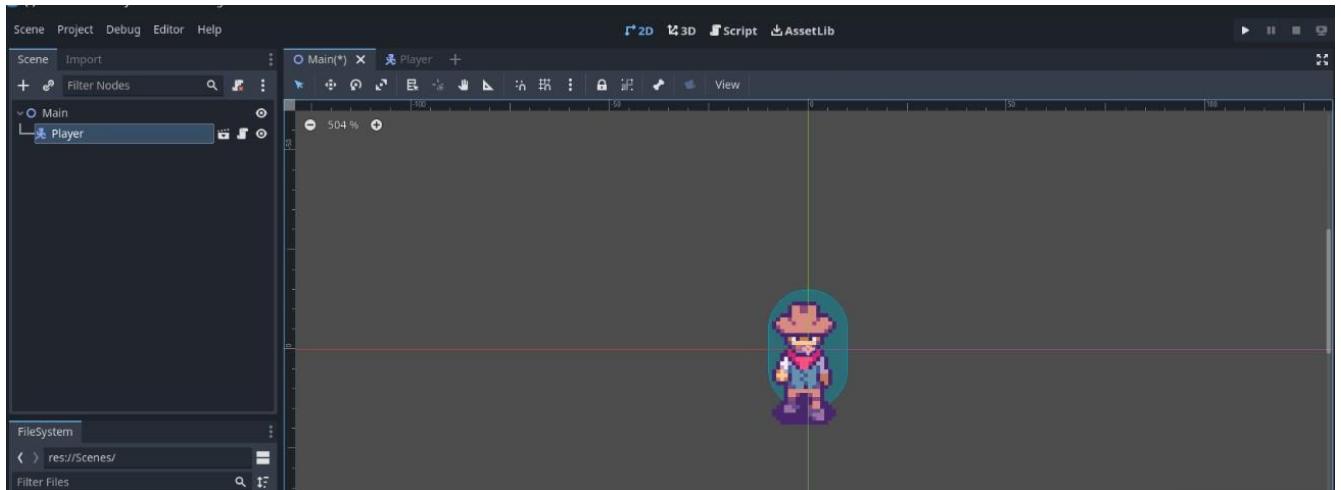
    # If input is digital, normalize it for diagonal movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()

    # Apply movement
    var movement = speed * direction * delta
    # Moves our player around, whilst enforcing collisions so that they come to a
    # stop when colliding with another object.
    move_and_collide(movement)
```

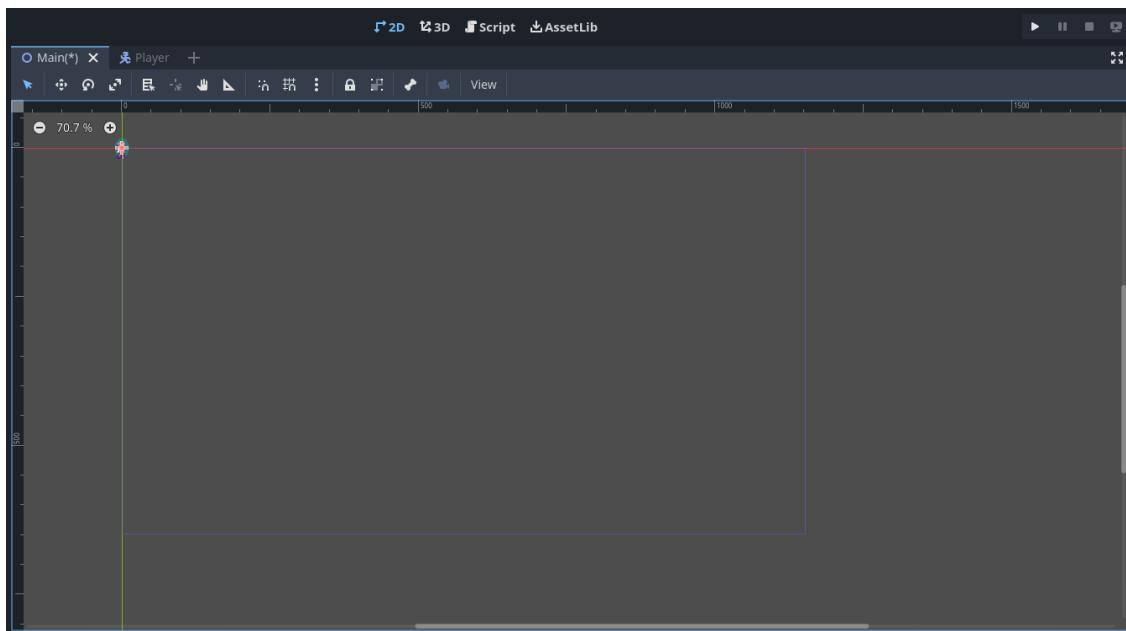
Save your script and let's go back to our Main scene. Click the chain icon next to the plus icon, which will help us instance (add) our Player scene to our Mode scene. Select the Player scene.



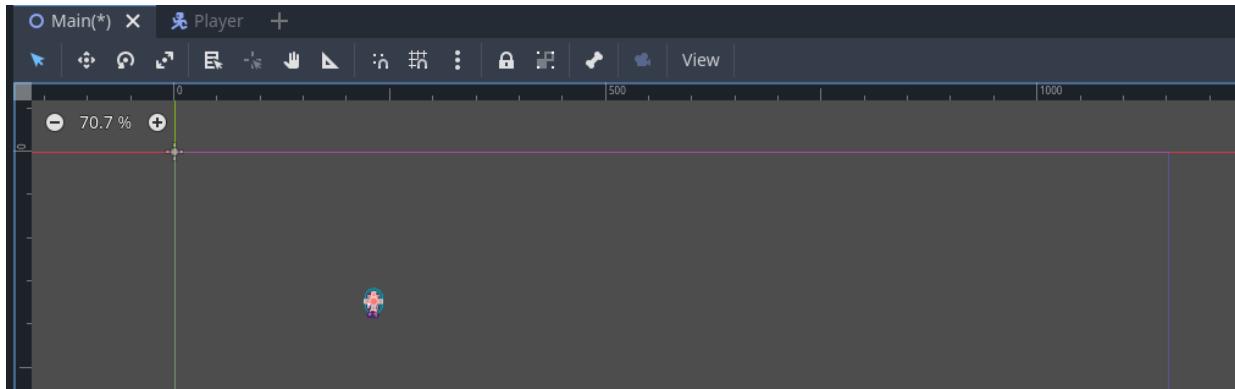
Press F to focus on the newly instanced Player scene in our Main scene, and you will see that it has been added, but we cannot see its children (the collision and sprite nodes). That's because it is treated as a complete object now, and not individual nodes.



Now zoom out until you see a blue border in your Main scene. This blue window is the border of your visible game screen or [viewport](#), which in simple terms, is the screen size you will see when you run the game.

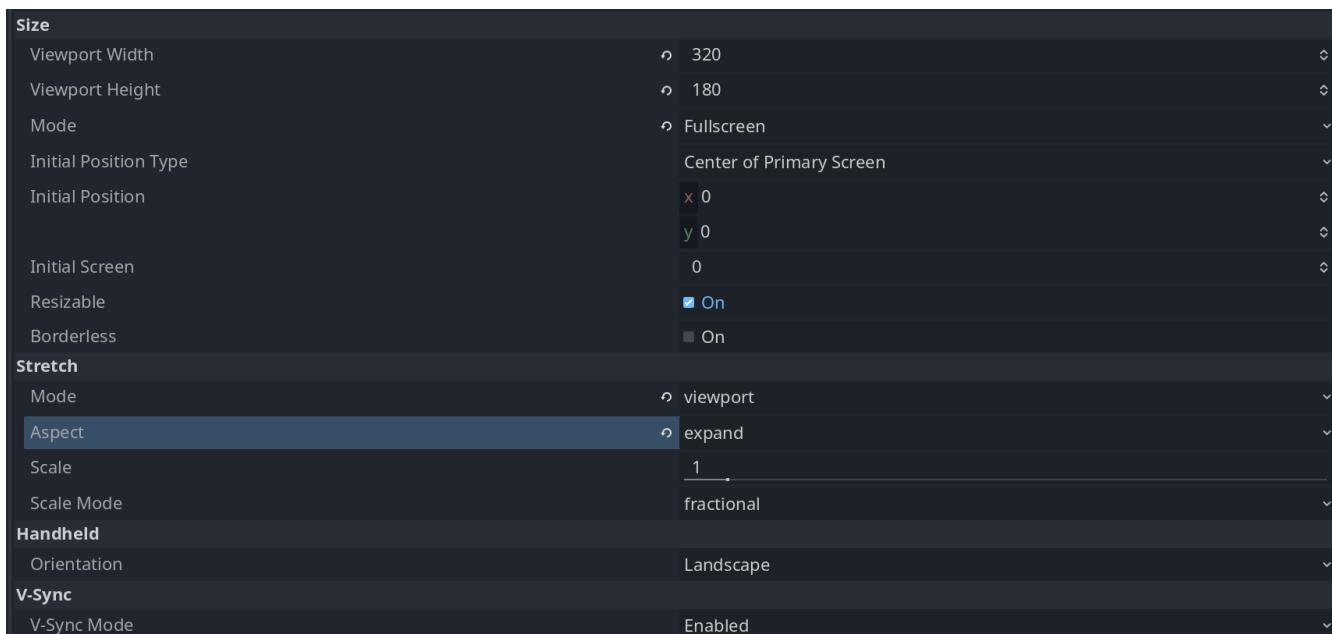


Our players will spawn in the top left corner, so we won't be able to see them properly. To prevent this, let's select them and drag them to a space closer to the middle or inner edges. Ensure your select tool is enabled for this, which is the cursor.

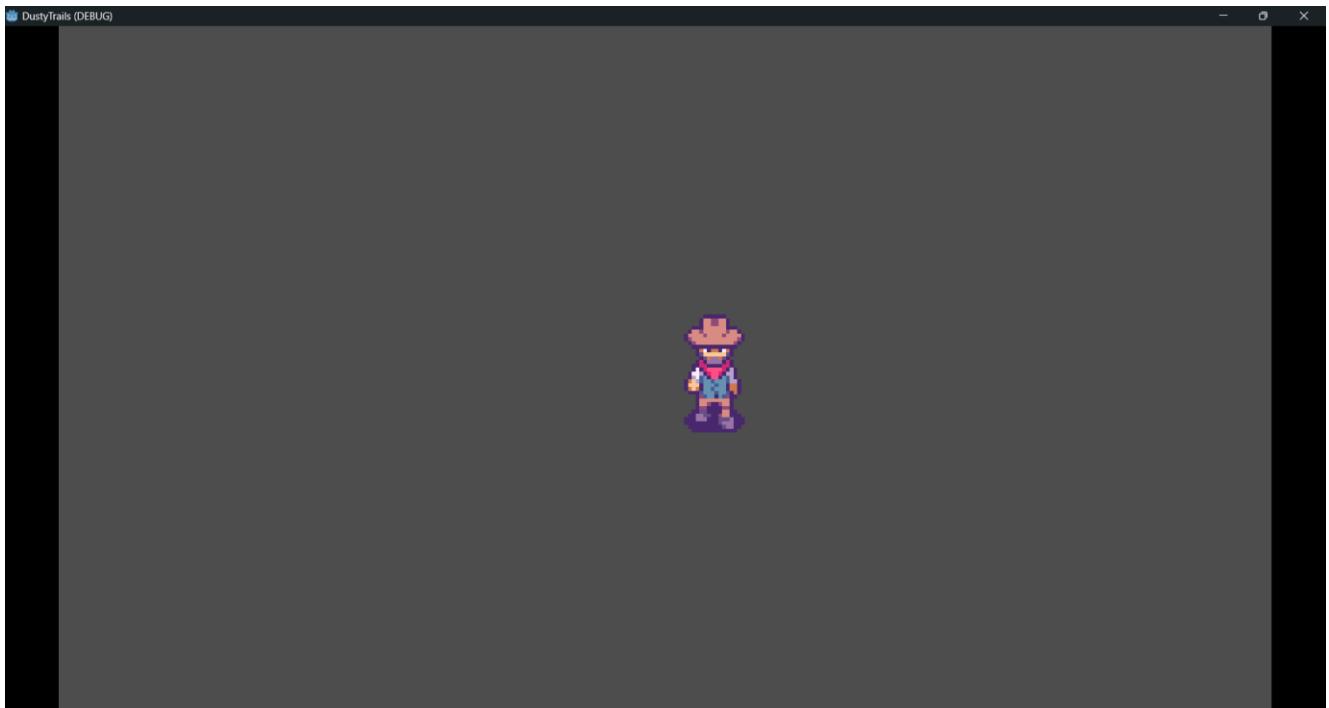


If we were to run our game now, the player will be very small. That is because our game window is zoomed out pretty far from where our player is. This is because our [viewport](#) size is too big, and our stretch mode needs to change.

Go into Project Settings > Display > Window, and make the following changes:



Our character will be zoomed in now, and we can run our game by clicking on the play button or by pressing F5.



The player is moving around the map when we enter WASD or our direction keys but without any animations! That's because we haven't yet assigned any animations for our inputs, but we will do that in the next part when we focus on animations as an isolated topic.

With our player character set up, we can move on to animations. You might be wondering about the world map, as well as the camera setup for our character, but don't worry, we'll get to everything in due time. Remember to save your game project and to make a backup, and I'll see you in the next part.

Your final code for this section should look like [this](#).

PART 3: PLAYER ANIMATIONS

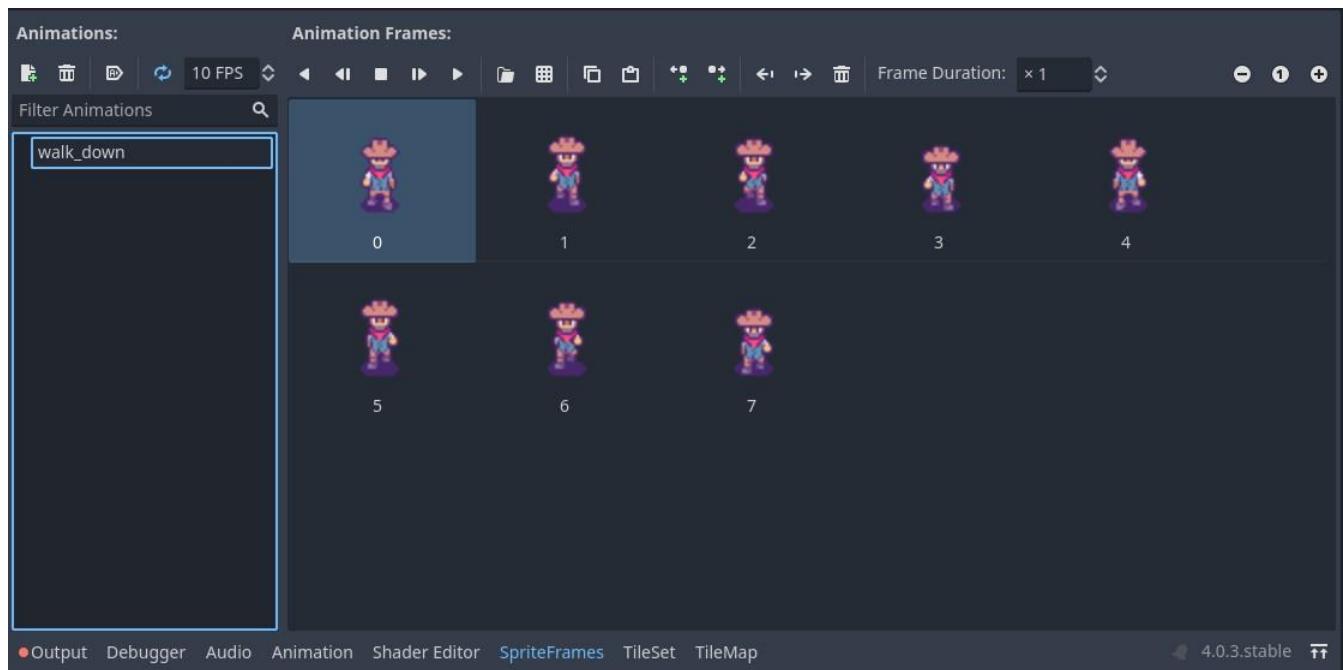
Our player has been set up, and it moves around the map when we run the game. There is one problem though, and that is that it moves around statically. It has no animations configured! In this section, we are going to focus on adding animations to our player, so that it can walk around and start coming to life.

WHAT YOU WILL LEARN IN THIS PART:

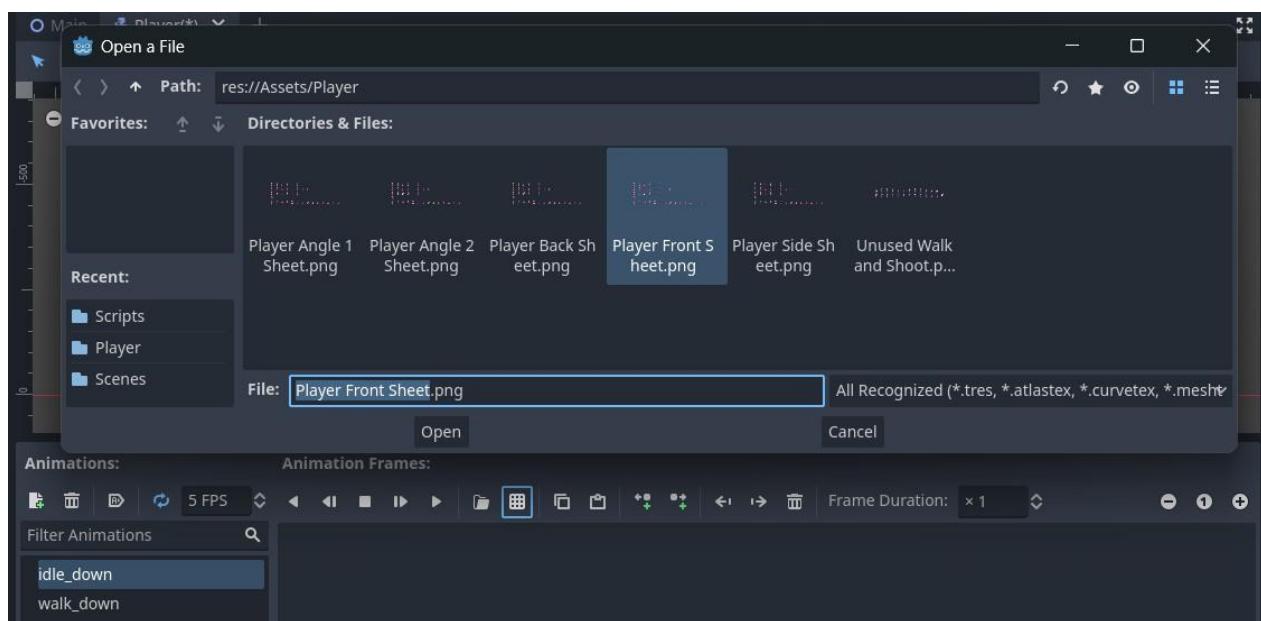
- How to work with sprite sheets for animation.
- How to add animations using the AnimatedSprite2D node.
- How to connect animations to input actions.
- How to add custom input actions.
- How to work with the input() function and Input singleton.
- How to work with built-in signals.

For our player, we need animations to move it up, down, left, and right. For each of those directions, we will set up animations for idling, walking, attacking, damage, and death. Since we have so many animations to tackle, I will take you through the first direction, and then give you a table to complete the rest.

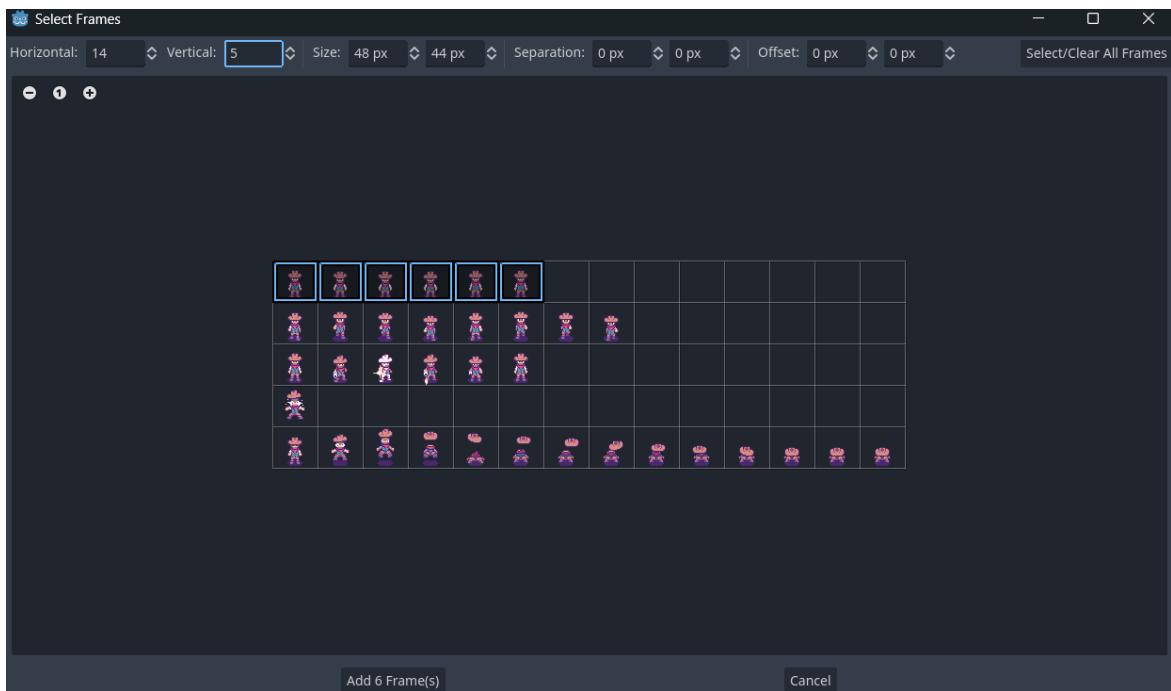
In your Player scene, with your AnimatedSprite2D selected, we can see that we already have an animation added for walking downwards. Let's rename this default animation as **walk_down**.



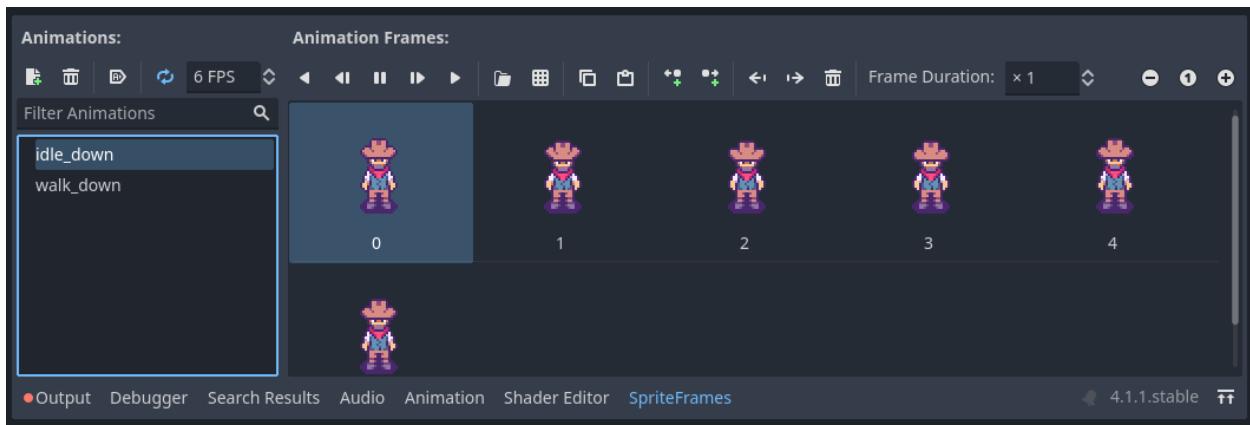
Click on the file icon to add a new animation and call this **idle_down**. Just like before, let's select the "Add Frames from Sprite Sheet" option, and navigate to the player's front-sheet animation.



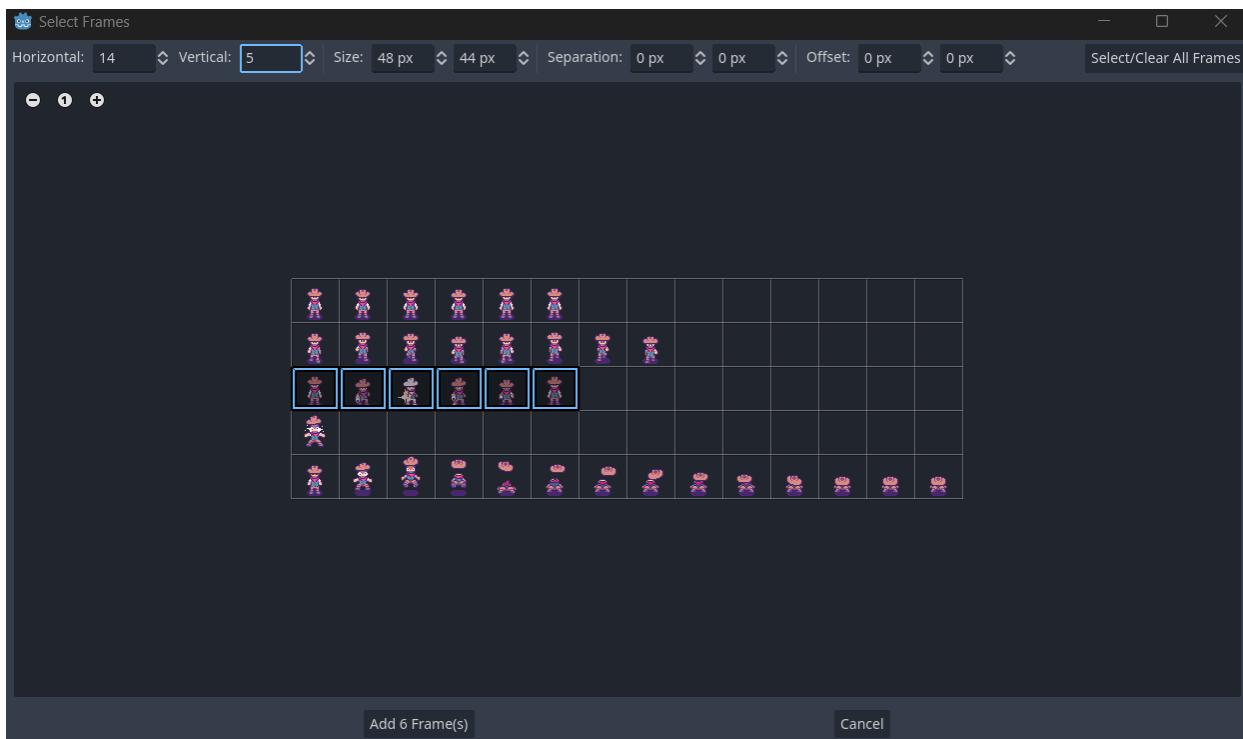
Change the horizontal value to 14, and the vertical value to 5, and select the first row of frames for our idle animation.



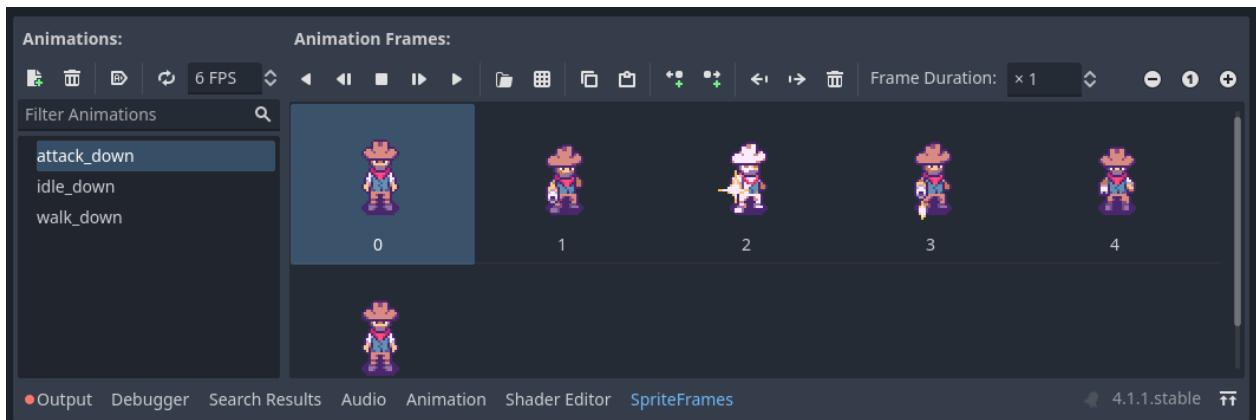
Change its FPS value to 6.



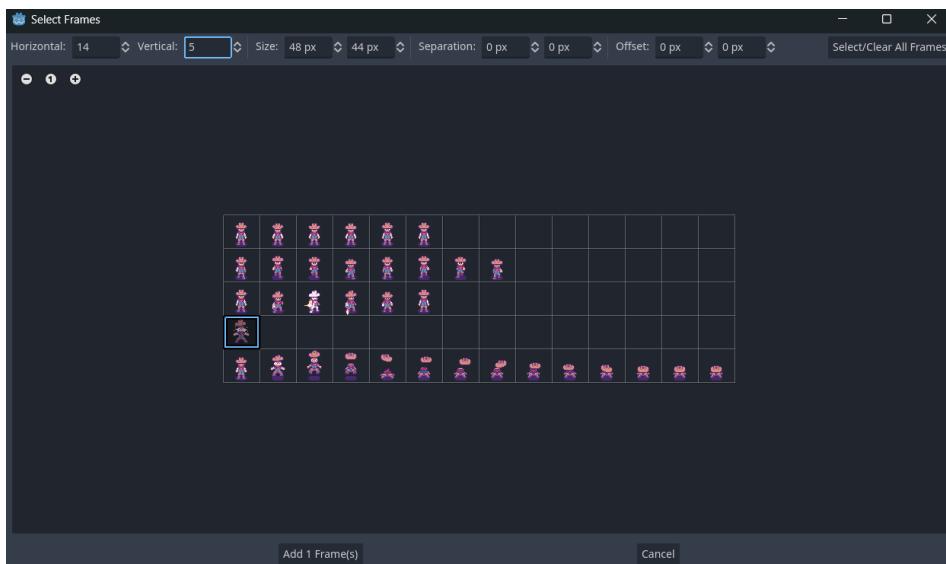
Add a new animation, and call it **attack_down**. For this one, you want to select the third row of animation frames.



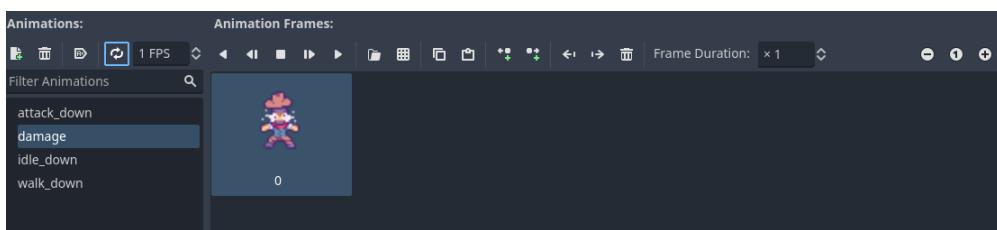
Change its FPS value to 6 and turn off looping because we only want this animation to fire off once, and not continuously.



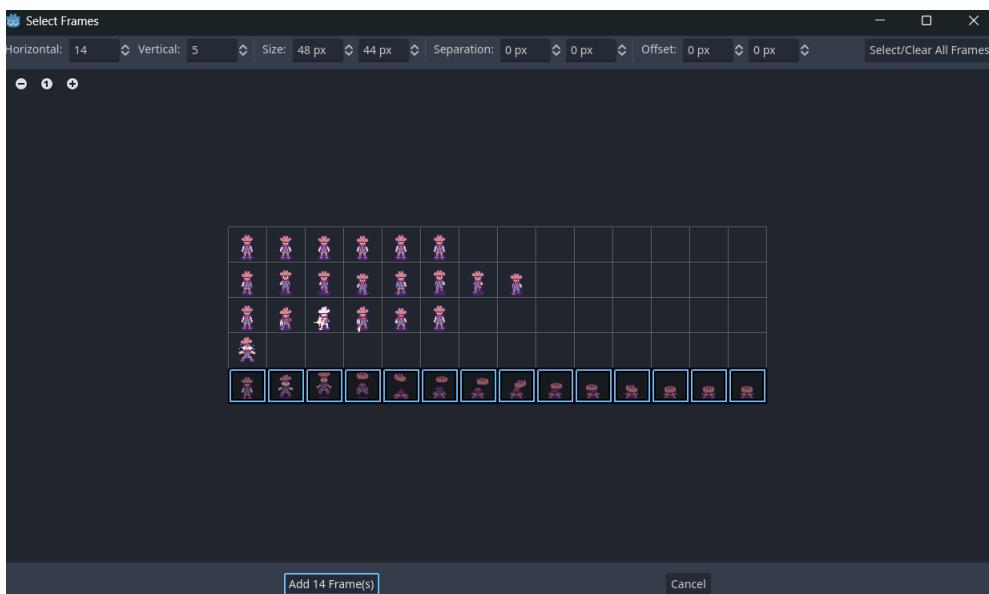
Next, let's add a new animation, and call it **damage**. Select the singular frame from row 4 for this animation.



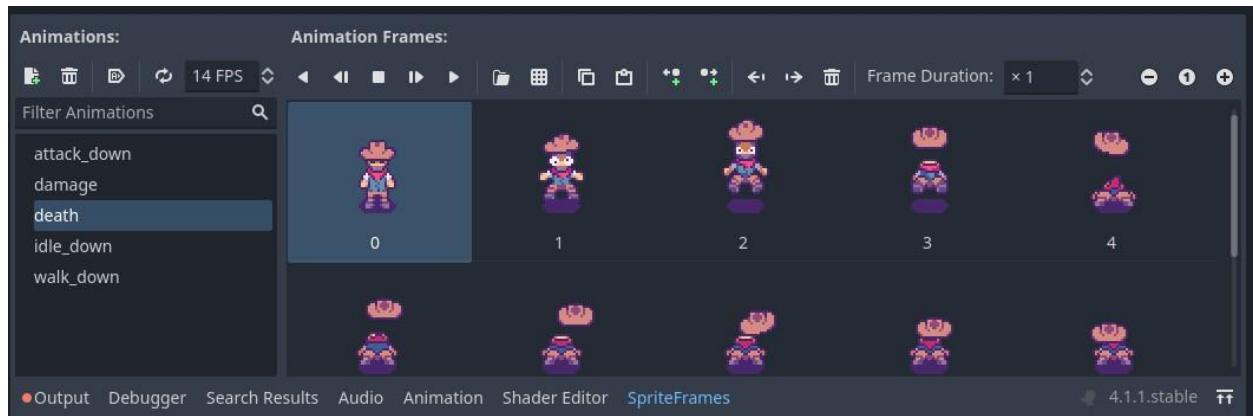
Change its FPS value to 1 and turn off looping.



Finally, create a new animation and call it **death**. Add the final row of frames to this animation.



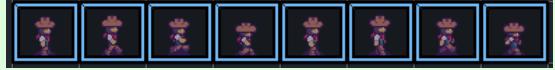
Change its FPS value to 14 and turn off looping.



Now we have our downward animations created, as well as our death and damage animations. Do you think you can handle adding the rest on your own? If not, reach out to me, and I will amend this section with the instructions for those as well.

For the more daring ones, here's a table of animations you need to add:

NAME	SPRITE FILE	FRAMES	FPS	LOOPING
idle up	Player Back Sheet.png	Row 1, frames 1 > 6	6	On
walk_up	Player Back Sheet.png	Row 2, frames 1 > 8	10	On
attack_up	Player Back Sheet.png	Row 3, frames 1 > 6	6	Off
idle_side	Player Side Sheet.png	Row 1, frames 1 > 6	6	On

walk_side	Player Side Sheet.png	 Row 2, frames 1 > 8	10	On
attack_side	Player Side Sheet.png	 Row 3, frames 1 > 6	6	Off

At the end, your complete animations list should look like this:



Now that we've added all of our player's animations, we need to go and update our script so that we can link these animations to our input actions. We want our "up" animations to play if we press W or UP, our "down" animations to play if we press S or DOWN, and "left" and "right" animations to play if we press A and LEFT, or D and RIGHT.

Let's open up our script by clicking on the scroll icon next to your Player node. Because we already have our movement functionality added in our `_physics_process()` function, we can go ahead and create a custom function that will play the animations based on

our player's direction. This function needs to take a [Vector2](#) as a parameter, which is the floating-point coordinates that represent our player's position or direction in a particular space and time (refer to the vector images in the previous sections as a refresher on this).

Underneath your existing code, let's create our *player_animations()* function.

```
### Player.gd

extends CharacterBody2D

# Player movement speed
@export var speed = 50

func _physics_process(delta):
    # older code

# Animations
func player_animations(direction : Vector2):
    pass
```

To determine the direction that the player is facing, we need to create two new variables. The first variable is the variable that we will compare against a zero vector. If the direction is not equal to Vector(0,0), it means the player is moving, which means the direction of the player. The second variable will be to store the value of this direction so that we can play its animation.

Let's create these new variables on top of our script, underneath our speed variable.

```
### Player.gd

extends CharacterBody2D

# Player movement speed
@export var speed = 50

# Direction & Animation variables
var new_direction = Vector2(0,1)
var animation
```

To make our code more organized, we will use the `@onready` annotation to create an instance of a reference to our `AnimatedSprite2D` node. This way we can reuse the variable name instead of saying `$AnimatedSprite2D.play()` each time we want to change the animation. Take note that we also flip our sprite horizontally when playing the `side_` animation. This is so that we can reuse the same animation for both left and right directions.

```
### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
```

Now, in our newly created function, we need to compare our `new_direction` variable to zero, and then assign the animation to it. If the direction is not equal to zero, we are moving, and thus our walk animation should play. If it is equal to zero, we are still, so the idle animation should play.

```
### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D

# older code

# Animations
func player_animations(direction : Vector2):
    #Vector2.ZERO is the shorthand for writing Vector2(0, 0).
    if direction != Vector2.ZERO:
        #update our direction with the new_direction
        new_direction = direction
        #play walk animation because we are moving
        animation = #todo
        animation_sprite.play(animation)
    else:
```

```
#play idle animation because we are still
animation = #todo
animation_sprite.play(animation)
```

But how do we determine the animation? We can do a long conditional check, or we can create a new function that will determine our direction (left, right, up, down) based on our plane directions (x, y) after the player presses an input action. If our player is pressing up, then we should return "up", and the same goes for down, side, and left. If our player presses both up and down, we will return side.

Remember when we had to put `_up`, `_down`, and `_side` after each animation? Well, there was also a reason for that. Let's create a new function underneath our `player_animations()` function to see what I mean by this.

```
### Player.gd

# older code

# Animation Direction
func returned_direction(direction : Vector2):
    #it normalizes the direction vector
    var normalized_direction = direction.normalized()
    var default_return = "side"

    if normalized_direction.y > 0:
        return "down"
    elif normalized_direction.y < 0:
        return "up"
    elif normalized_direction.x > 0:
        #(right)
        $AnimatedSprite2D.flip_h = false
        return "side"
    elif normalized_direction.x < 0:
        #flip the animation for reusability (left)
        $AnimatedSprite2D.flip_h = true
        return "side"

    #default value is empty
    return default_return
```

In this function, we check the values of our player's x and y coordinates, and based on that, we return the animation suffix, which is the end of the word (_up, _down, _side). We will then append this suffix to the animation to play, which is walk or idle. Let's go back to our *player_animations()* function and replace the *#todo* code with this functionality.

```
### Player.gd

# older code

# Animations
func player_animations(direction : Vector2):
    #Vector2.ZERO is the shorthand for writing Vector2(0, 0).
    if direction != Vector2.ZERO:
        #update our direction with the new_direction
        new_direction = direction
        #play walk animation, because we are moving
        animation = "walk_" + returned_direction(new_direction)
        animation_sprite.play(animation)
    else:
        #play idle animation, because we are still
        animation = "idle_" + returned_direction(new_direction)
        animation_sprite.play(animation)
```

Now all we have to do is actually call this function in the function where we added our movement, which is at the end of our *_physics_process()* function.

```
### Player.gd

# older code

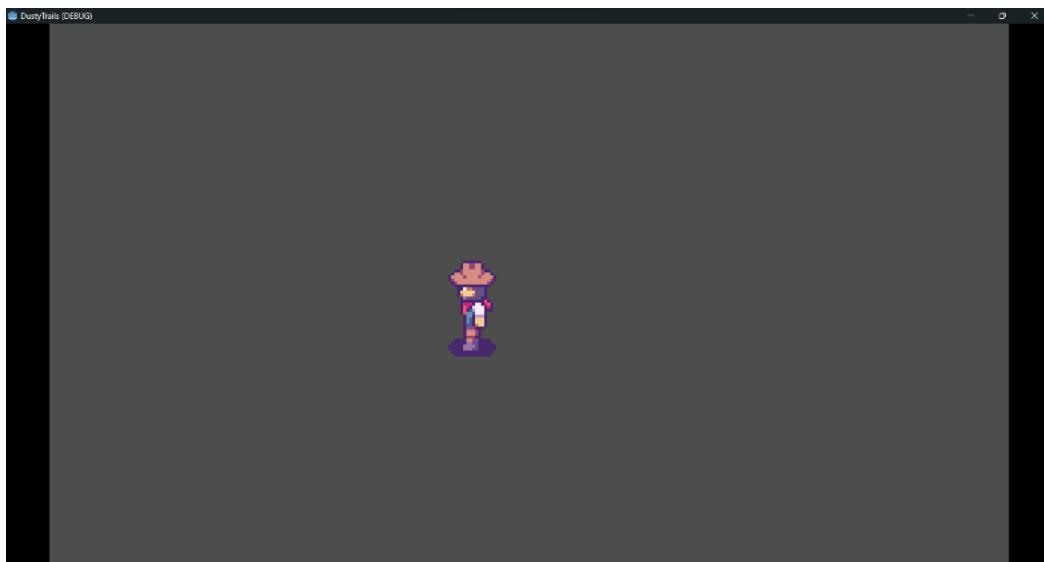
func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
    Input.get_action_strength("ui_left")
    direction.y = Input.get_action_strength("ui_down") -
    Input.get_action_strength("ui_up")
    # If input is digital, normalize it for diagonal movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()
```

```

# Apply movement
var movement = speed * direction * delta
# moves our player around, whilst enforcing collisions so that they come to a
stop when colliding with another object.
move_and_collide(movement)
#plays animations
player_animations(direction)

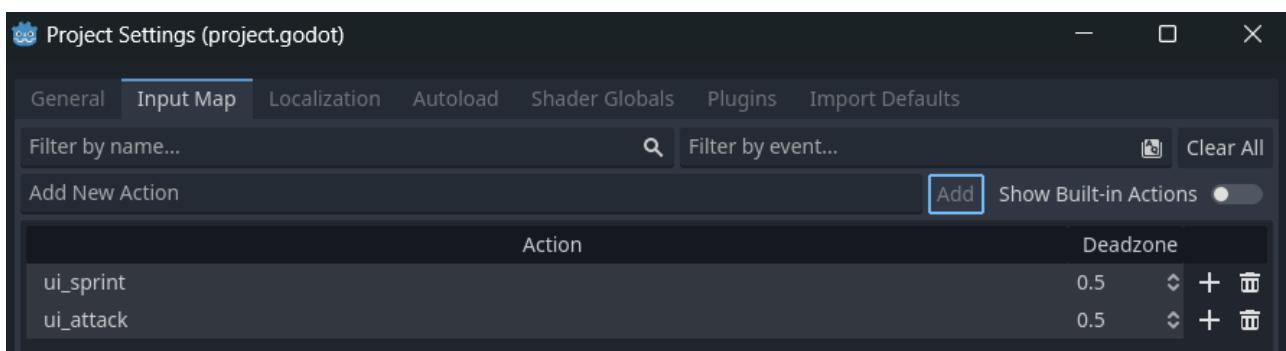
```

If you run your scene now, you will see that your player plays the idle animation if it's still, and the walk animation if it's moving, and he also changes direction!

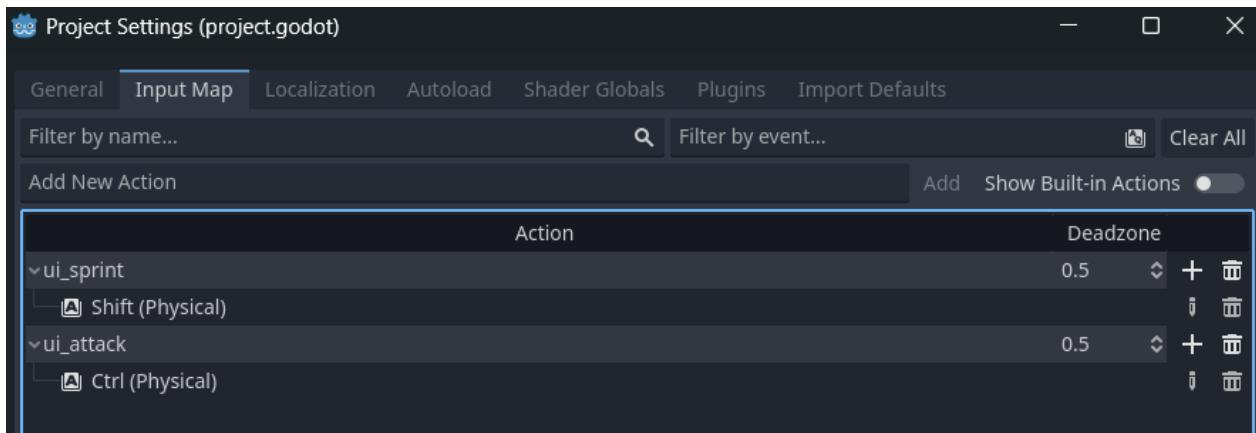


While we are at it, let's also add the animation functionality for our players attacking and sprinting. For these animations, we will require new input actions, so in your project settings underneath Input Map, let's add a new input.

Call the first one **ui_sprint** and the second one **ui_attack**.



Assign the Shift key to the ui_sprint, and the CTRL key to ui_attack. You can also assign joystick keys to this if you want. We will add the sprinting input in our `_physics_process()` function because we will use this later to track our stamina values at a constant rate, but we will add the attacking input in our `_input(event)` function because we want the input to be captured, but not tracked during the game loop.



You can either add inputs in Godot in its built-in `input` function, or via the `Input` Singleton class. You will use the Input singleton if you want the state of the input actions stored all throughout the game loop, because it is independent of any node or scene, for example, you might use it in a scenario where you want to be able to press two buttons at once.

The `_input` function can capture inputs `'is_action_pressed()'` and `'is_action_released()'` functions, whilst the Input singleton captures inputs via the `'is_action_just_pressed()'` / `'is_action_just_released()'` functions. I made some images to simply explain the different types of input functions.

When to use the Input Singleton vs. `input()` function?

In Godot you can cause input events in two ways: the `input()` function, and the `Input` [Singleton](#). The main difference between the two is that inputs that are captured in the `input()` function can only be fired off once, whereas the Input singleton can be called in any other method.

For example, we're using the Input singleton in our `physics_process()` function because we want the input to be captured continuously to trigger an event, whereas if we were to pause the game or shoot our gun, we would capture these inputs in the `input()` function because we only want the input to be captured once - which is when we press the button.

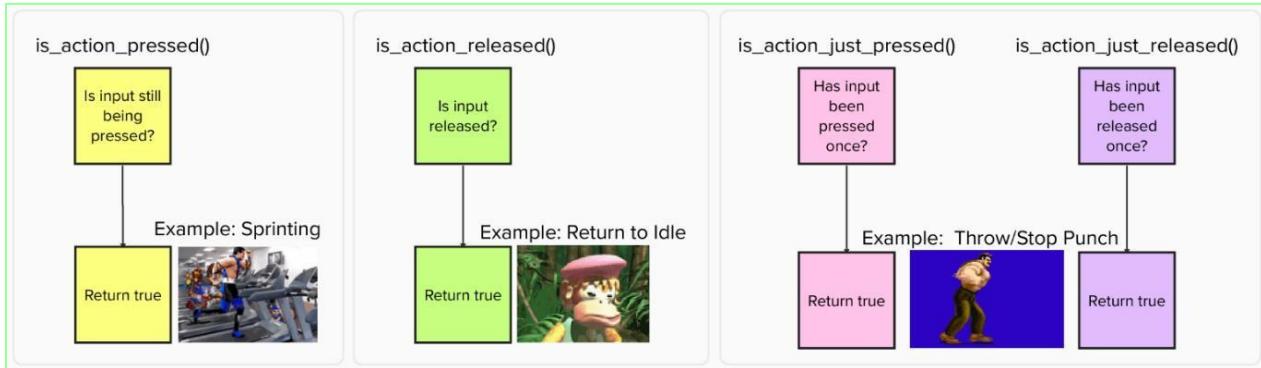


Figure 6: Input methods. View the online version [here](#).

Let's start with the code for our attack input. We only want to play our other animations only if our player isn't attacking, and vice versa. To do this, we need to create a new variable which we will use to check if the player is currently attacking or not.

```

### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D

# Player states
@export var speed = 50
var is_attacking = false

```

Now we can amend our `_physics_process()` function to only play our returned animations and to only process our movement if the player is not attacking.

```

### Player.gd

extends CharacterBody2D

# older code

func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
    Input.get_action_strength("ui_left")
    direction.y = Input.get_action_strength("ui_down") -
    Input.get_action_strength("ui_up")
    # If input is digital, normalize it for diagonal movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()
    # Apply movement if the player is not attacking
    var movement = speed * direction * delta

    if is_attacking == false:
        move_and_collide(movement)
        player_animations(direction)

```

We now need to call on our built-in `func _input(event)` function so that we can play the animation for our attack_ prefix. Let's add this function underneath our `_physics_process()` function.

```

### Player.gd

extends CharacterBody2D

# older code

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #attacking/shooting anim
        is_attacking = true
        var animation  = "attack_" + returned_direction(new_direction)
        animation_sprite.play(animation)

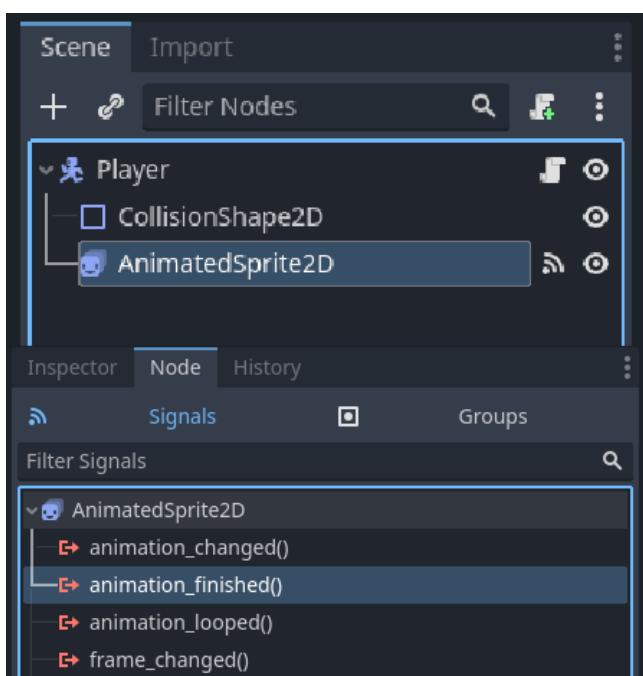
```

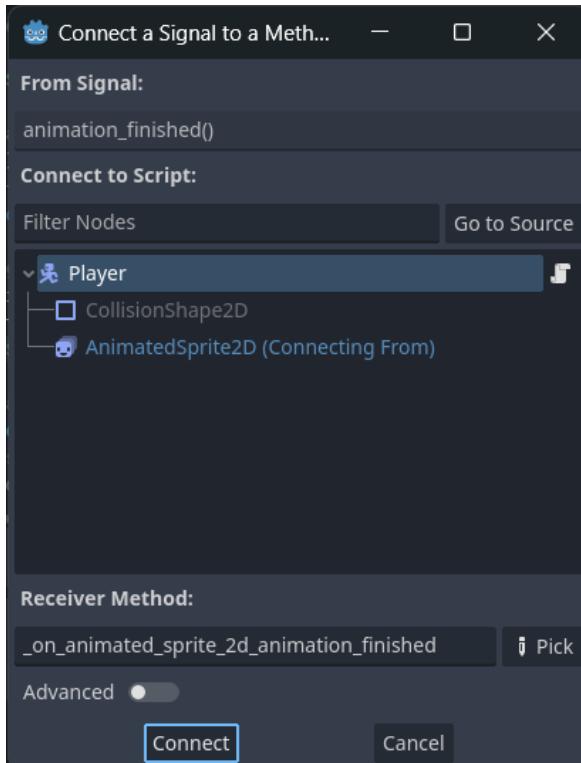
You will notice now that if you run your scene and press CTRL to fire off your attack animation, the animation plays but our character does not return to its previous idle or walk animations. This is because our character is stuck on the last animation frame of our attack function, and to fix this, we need to use a signal to notify our game that the animation has finished playing so that it can set our *is_attacking* variable back to false.

To do this, we need to click on our AnimatedSprite2D node, and in the Node panel we need to hook up the *animation_finished* signal to our player script. This signal will trigger when our animation has reached the end of its frame, and thus if our attack_ animation has finished playing, this signal will trigger the function to reset our *is_attacking* variable back to false. Double-click on the signal and select the script to do so.

What is a signal?

A [signal](#) is an emitter that provides a way for nodes to communicate without needing to have a direct reference to each other. This makes our code more flexible and maintainable. Signals emit after certain events occur, for example, if our enemy count changes after an enemy is killed, we can use signals to notify the UI that it needs to change the value of the enemy count from 10 to 9.





You will now see that a new method has been created at the end of your player script. We can now simply reset our *is_attacking* variable back to false.

```
### Player.gd

extends CharacterBody2D

# older code

# Reset Animation states
func _on_animated_sprite_2d_animation_finished():
    is_attacking = false
```

Next, we can go ahead and add in our sprinting functionality for when the player holds down Shift. Since sprinting is just movement, we can add the code for it in our *_physics_process()* function, which handles our player's movement and physics (not animations).

```
### Player.gd
```

```

extends CharacterBody2D

# Movement & Animations
func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
    Input.get_action_strength("ui_left")
    direction.y = Input.get_action_strength("ui_down") -
    Input.get_action_strength("ui_up")
    # Normalize movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()
    # Sprinting
    if Input.is_action_pressed("ui_sprint"):
        speed = 100
    elif Input.is_action_just_released("ui_sprint"):
        speed = 50
    # Apply movement if the player is not attacking
    var movement = speed * direction * delta
    if is_attacking == false:
        move_and_collide(movement)
        player_animations(direction)
    # If no input is pressed, idle
    if !Input.is_anything_pressed():
        if is_attacking == false:
            animation = "idle_" + returned_direction(new_direction)

```

Finally, if the player is not pressing any input, then the idle animation should play. This will prevent our player from being stuck in a running state even if our inputs are released.

```

### Player.gd

extends CharacterBody2D

# Movement & Animations
func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
    Input.get_action_strength("ui_left")

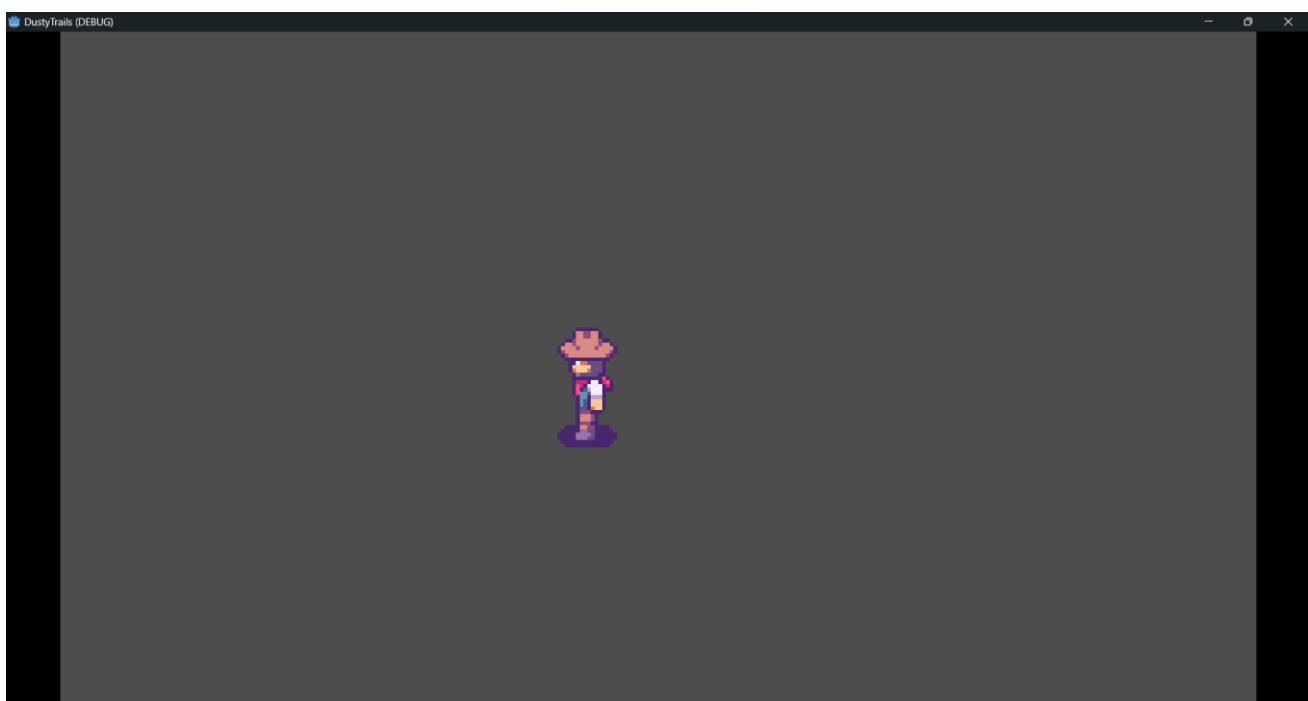
```

```

direction.y = Input.get_action_strength("ui_down") -
Input.get_action_strength("ui_up")
# Normalize movement
if abs(direction.x) == 1 and abs(direction.y) == 1:
    direction = direction.normalized()
# Sprinting
if Input.is_action_pressed("ui_sprint"):
    speed = 100
elif Input.is_action_just_released("ui_sprint"):
    speed = 50
# Apply movement if the player is not attacking
var movement = speed * direction * delta
if is_attacking == false:
    move_and_collide(movement)
    player_animations(direction)
# If no input is pressed, idle
if !Input.is_anything_pressed():
    if is_attacking == false:
        animation = "idle_" + returned_direction(new_direction)

```

If we run our scene, we can see that the player is moving around the map with animations, and they can also sprint and attack!



With our player character set up, we can move on to animations. In the next part we will be creating the map for our game, i.e., the world, as well as set up our player's camera. Remember to save your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 4: GAME TILEMAP & CAMERA SETUP

We now have a functioning player, well, sort of, we still have a lot to add to our little person to make them a real game hero! But for now, our player has to wait, because in this part we are going to add our games [TileMap](#) with collisions, as well as add a [Camera2D](#) to our game so that we can follow our player around. We didn't add the camera node during the player setup section of this tutorial because the camera node does not work without some sort of background, which is our tilemap.

WHAT YOU WILL LEARN IN THIS PART:

- How to work with sprite sheets for the TileMap node.
- How to work with the TileMap and TileSheet panels.
- How to use the TileMap tools to paint tiles onto the map.
- How to create tilemap layers, as well as physics and terrain layers.
- How to add a camera to your player.

First things first, what is a tilemap or tilesheet? Well, tilemaps are the base for building a game world or level map out of small, regular-shaped images called tiles. These tiles are stored in something we call an atlas, tilesheet, or spritesheet, which we've seen before when adding our animations. In other words, the tilesheet is the actual graphics of the tilemap.

What is the difference between a TileMap and a TileSet?

The TileMap is the grid where you place tiles from the Tilesheet to actually build your level. A Tilesheet is a collection of tiles that you can use to build your level.

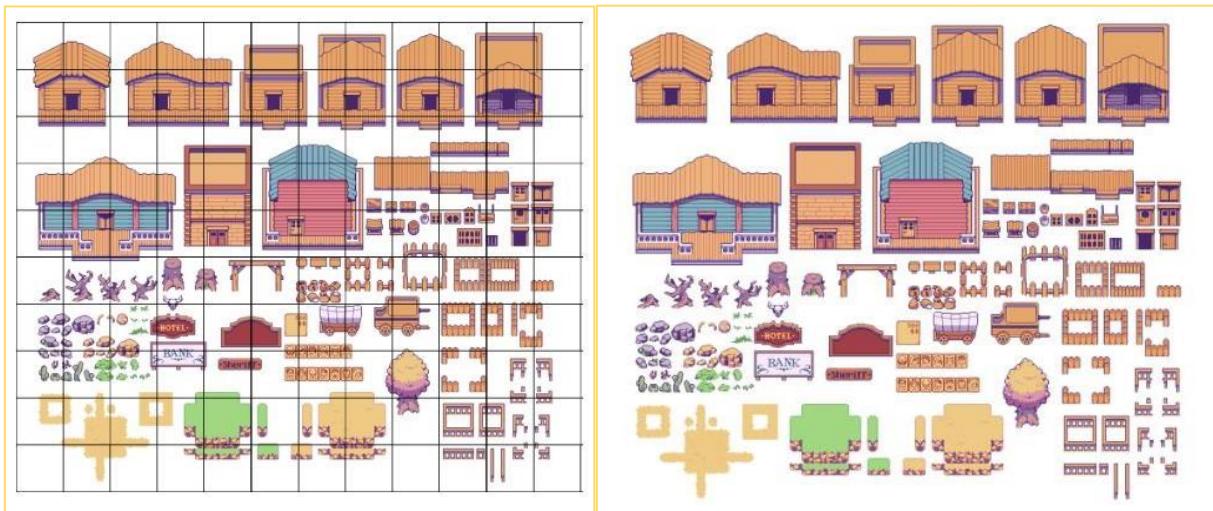
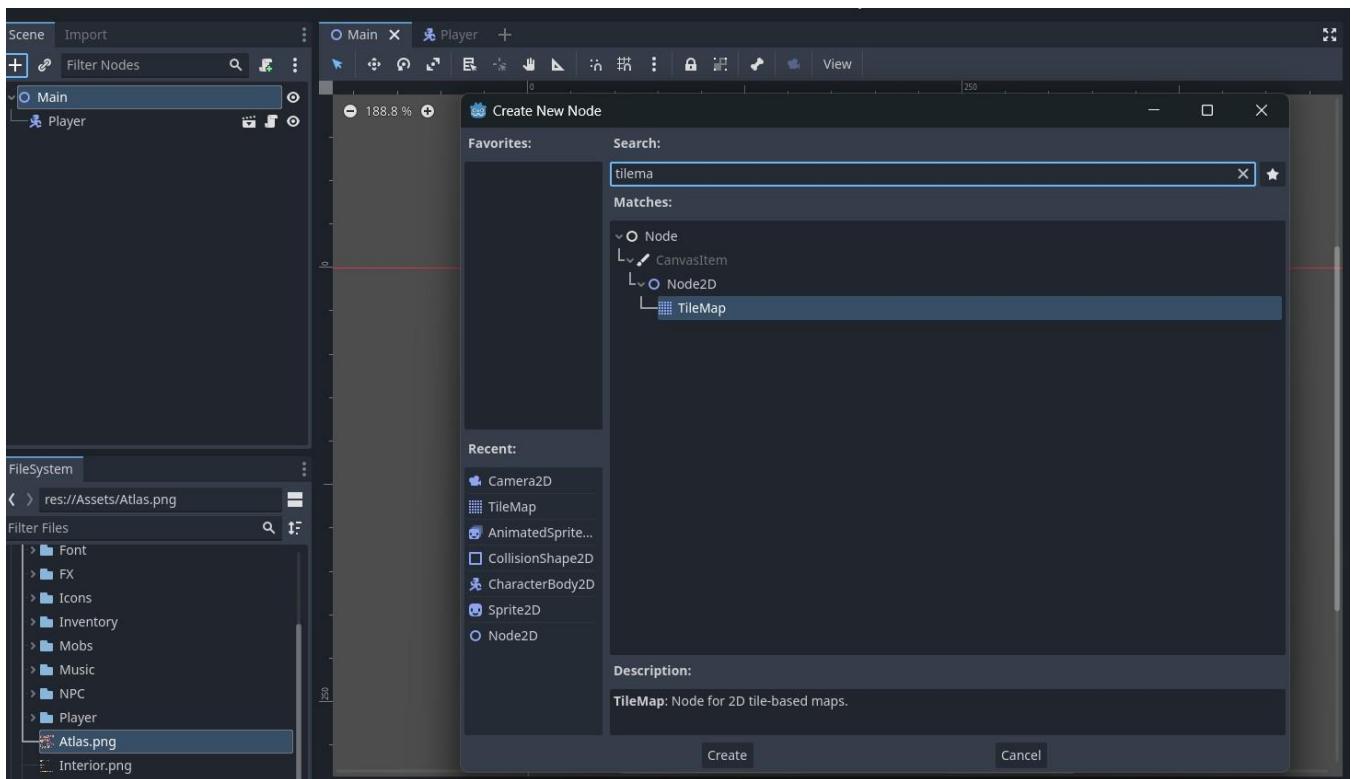
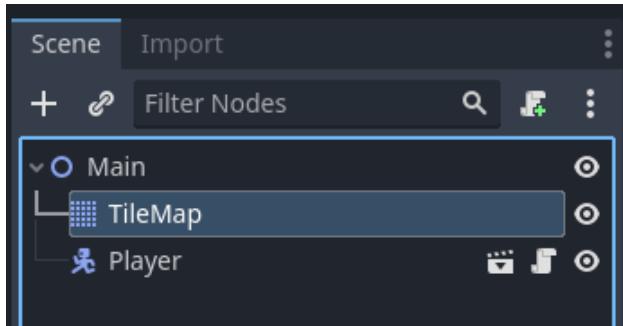


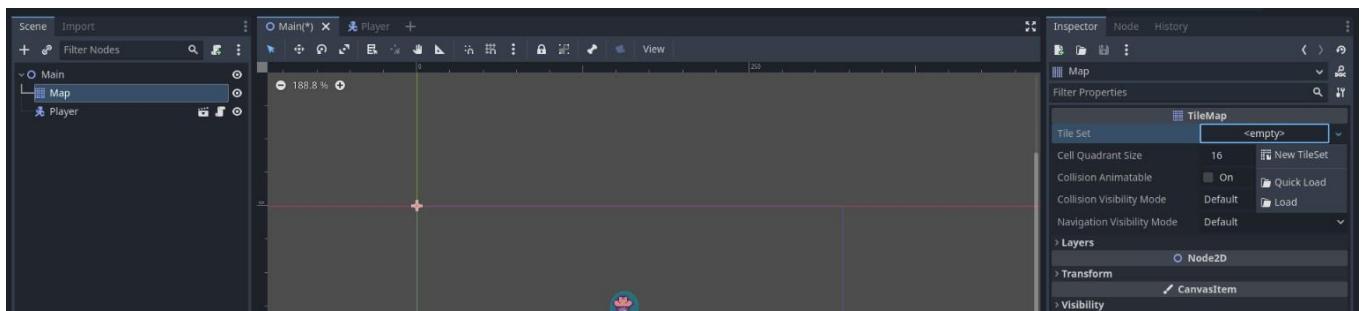
Figure 7: Tileset (left) and a TileMap (right)

In our Main scene, we can add a new node of type TileMap. Click and drag this node so that it is behind our Player scene, otherwise, the player will be hidden by our map!

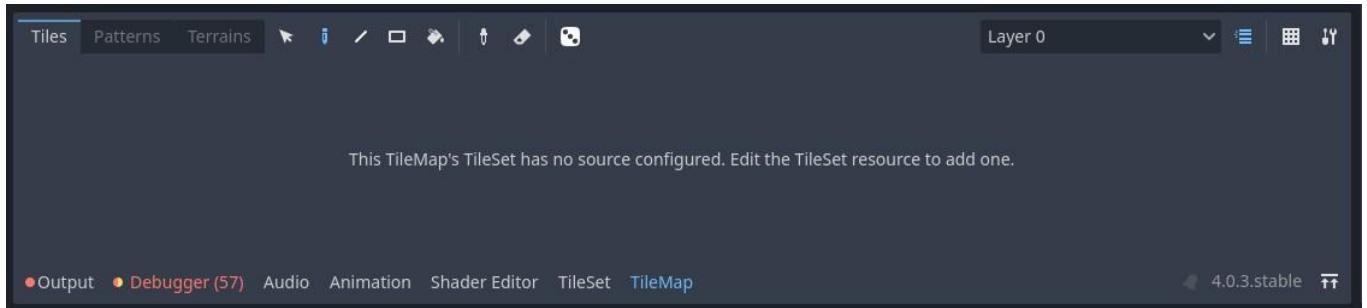




Let's rename this node as "Map", and in the Inspector panel, click on <empty> next to TileSet to create a new tileset.

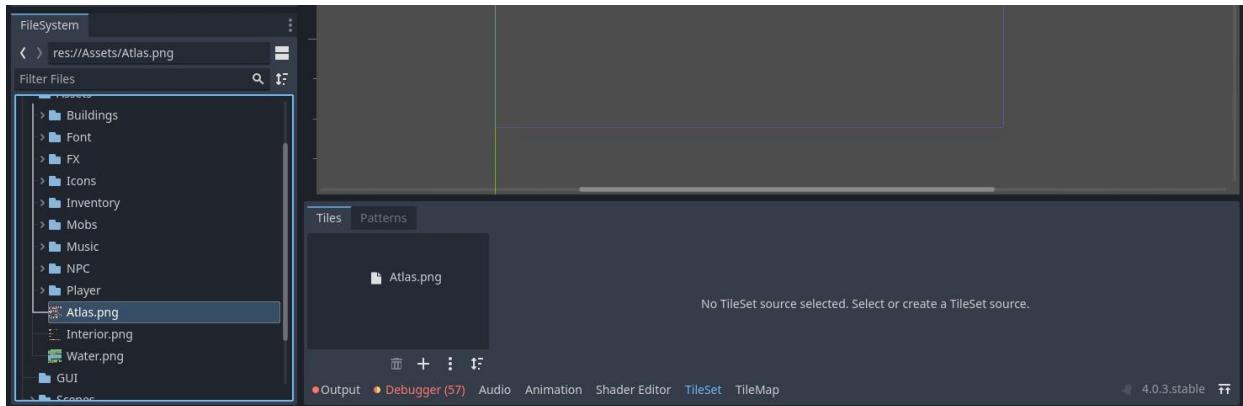


You will see two new options pop up below: [TileMap](#) and [TileSet](#).

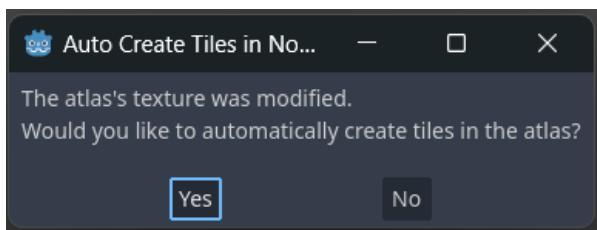


To create our TileMap (which is the map we will draw on our screen), we first need to assign it a TileSet (which is the image we will draw from). To do this, click on the TileSet option, and open your Assets folder in your FileSystem panel. You will see that I gave you three tilesets: Interior, Water, and Atlas. We will use these tilesets to create our world.

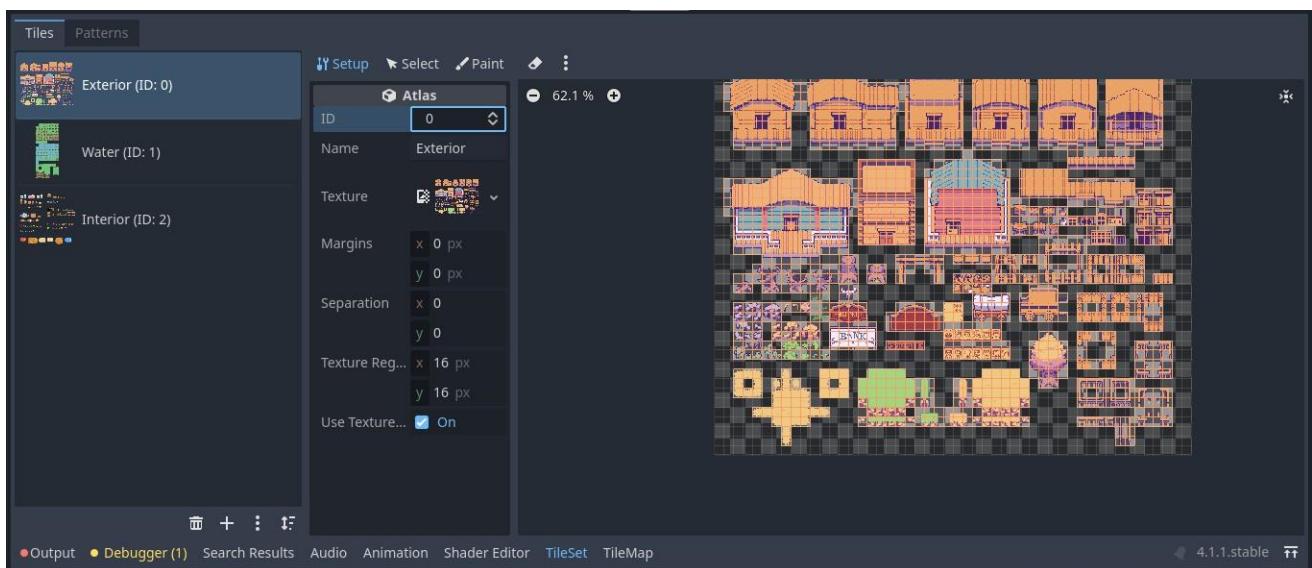
You can drag these tilesheet into the Tiles option in your TileSet panel.



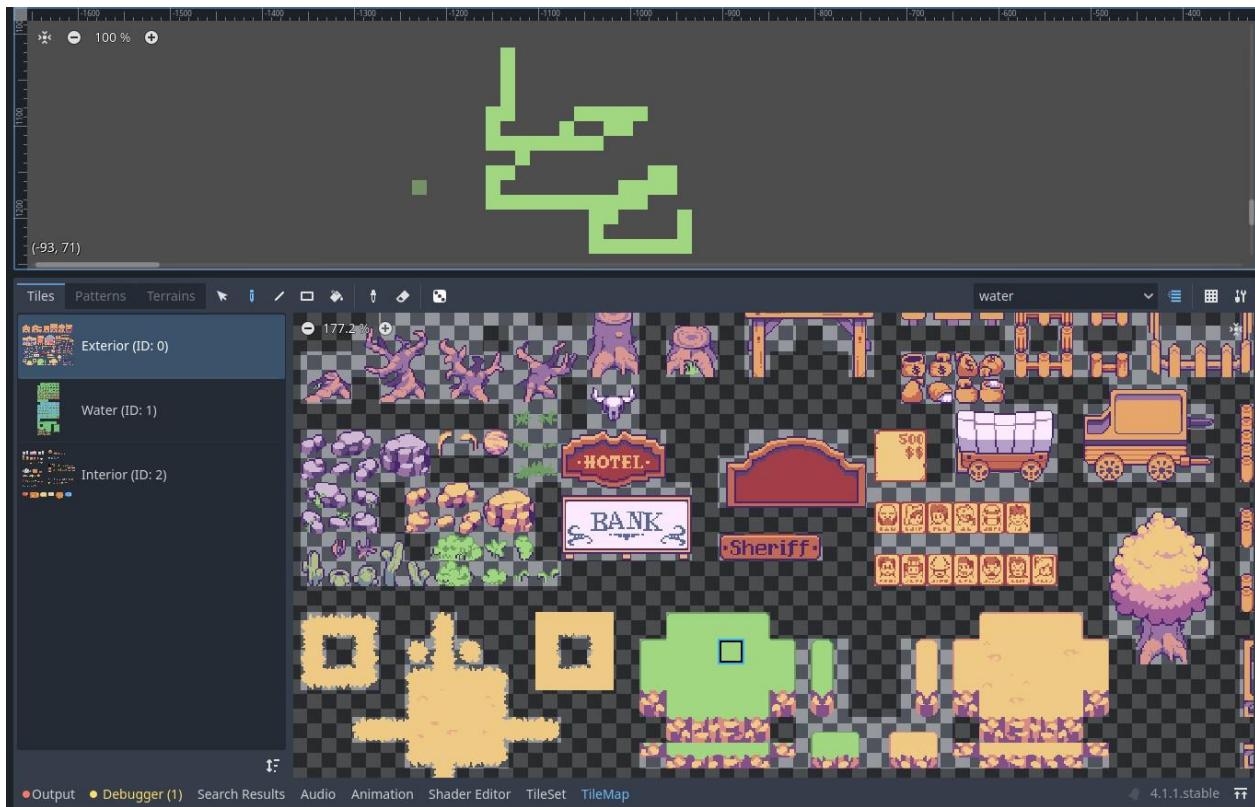
A popup will arise asking you if you want the engine to create automatic tiles. Say yes to this prompt, because this will break our image into nice squares to form a grid.



You can now change the tilesheets name, ID, and grid separation values. Change your names and ID's as follows:



Now if we go back to the TileMap panel, we can see that our tiles have been added. You can now click on the tilesheet and start painting the images onto the screen.



Here is a brief overview of the tools present in the TileMaps panel:

SELECT TOOL	PAINT TOOL	RECTANGLE TOOL	BUCKET TOOL	EYE DROPPER TOOL
Allows you to select and move tiles	Allows you to paint individual tiles	Allows you to paint large selections of tiles	Allows you to fill areas with a tile	Allows you to copy tiles

The next part will be a bit confusing, so I'm going to try and break it down into sections so that you can better understand the concepts. If things get a bit too complex for you, please [download](#) the Map scene that I created and drag in the Map node from the scene into your Main scene.

MAP CREATION

1. Terrain Layers Introduction (Auto-Tiling)

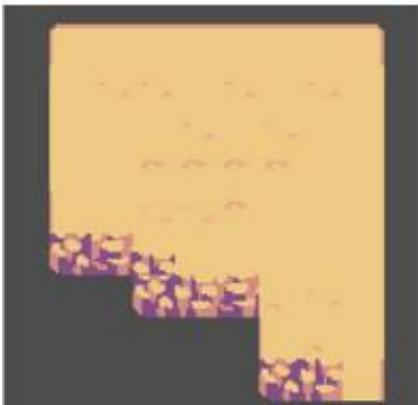
We can paint or create our map with just the paint tools provided, but our environment would look pretty bland, especially for our terrain. To make things easier for us, we can make use of something called [autotiles](#).

Terrains - also referred to as autotiles, create organic terrain shapes that will autonomously draw corners and edges in the area that we tell them to. This removes the tediousness of manually placing down edges and sides of say grass patches or water bodies while trying to create interesting-looking terrains ourselves.



Without autotiles:

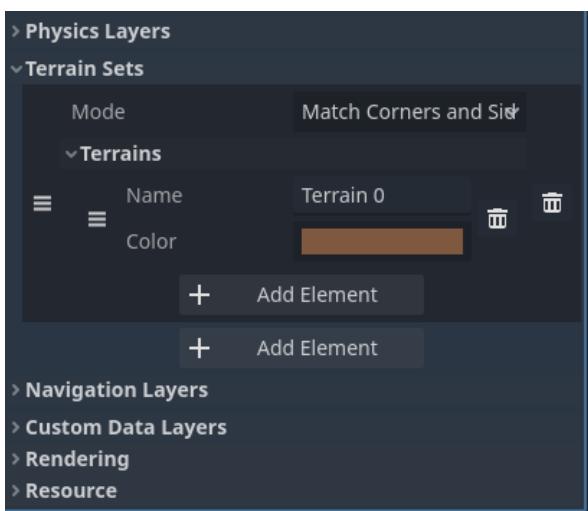
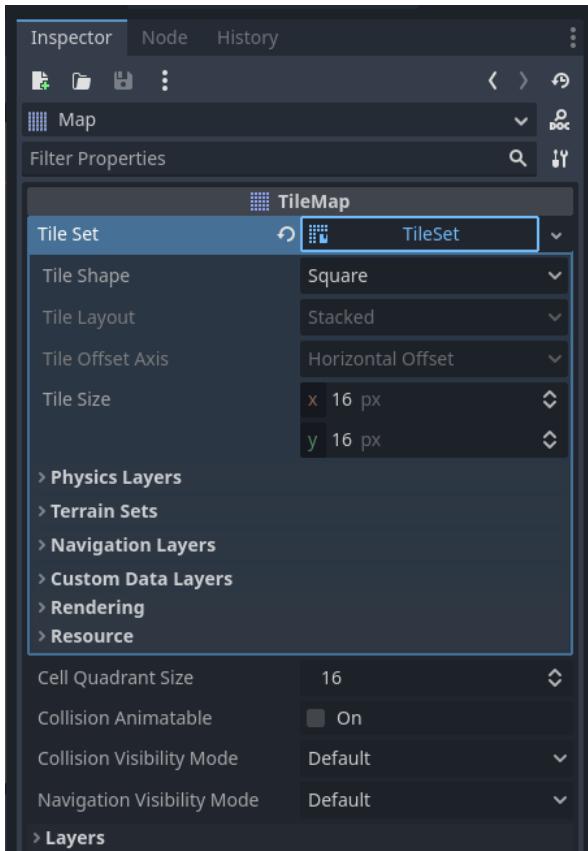
We have to manually draw in terrain tiles one by one.



With autotiles:

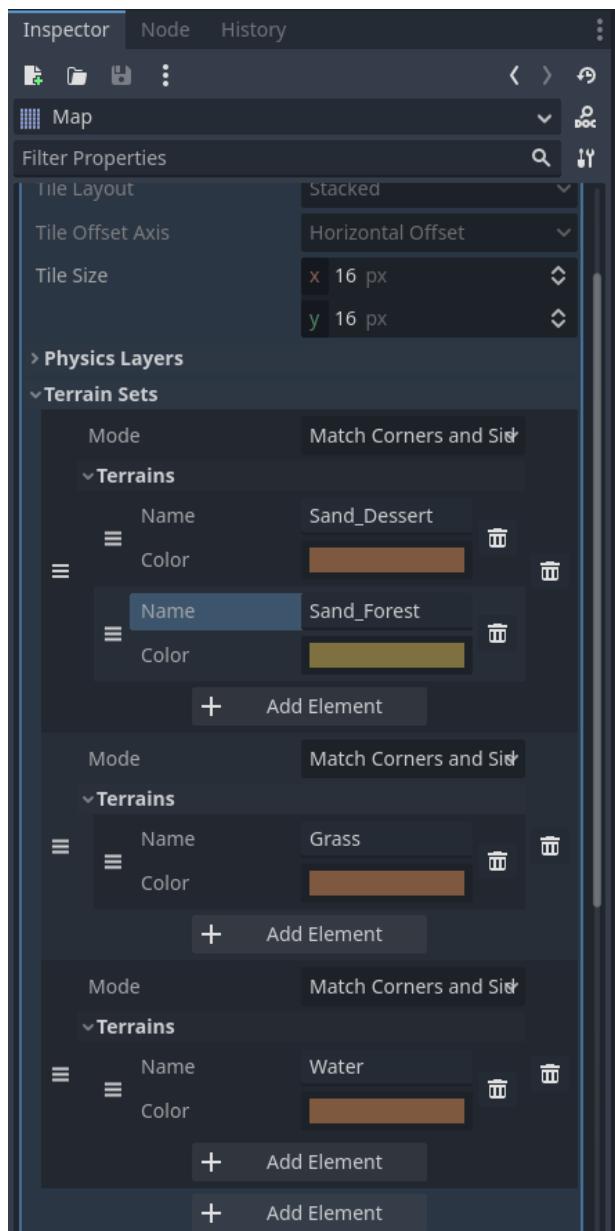
We just drag out terrain and it automatically creates the terrain & shapes for us.

To create autotiles, simply go over to the Inspector panel, click on the TileSet resource (the one with the image), and underneath Terrain Sets add a terrain set element. Remember to add an element to the terrain set element as well.



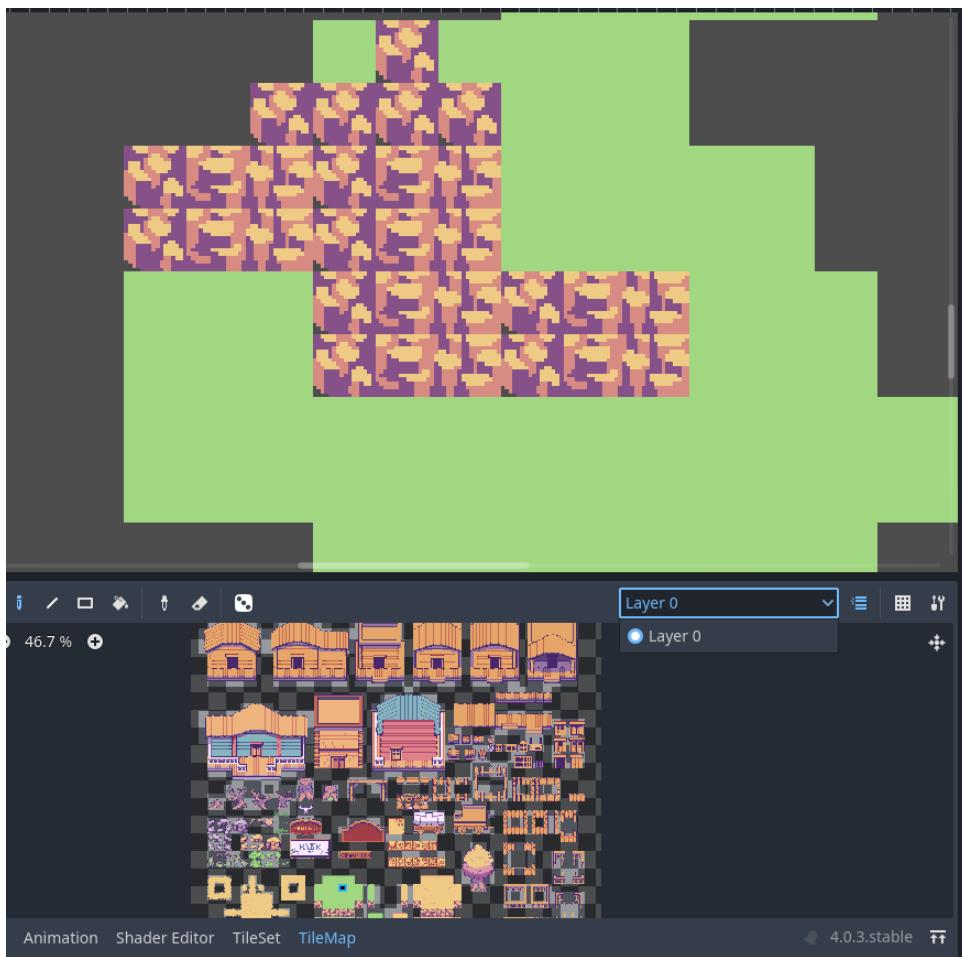
You can create different terrain sets for different terrain elements. For example, you'll put your grass autotile on one set, and your dirt autotile on another set, and so forth.

Let's create terrain elements for grass, sand, and water. You will see from the image below, that I have a Sand_Dessert and a Sand_Forest. That is just to show you that you can have multiple sands in one terrain element set. Grass, sand, and water, in my case, are all in different sets, but the sand elements are in the same set.

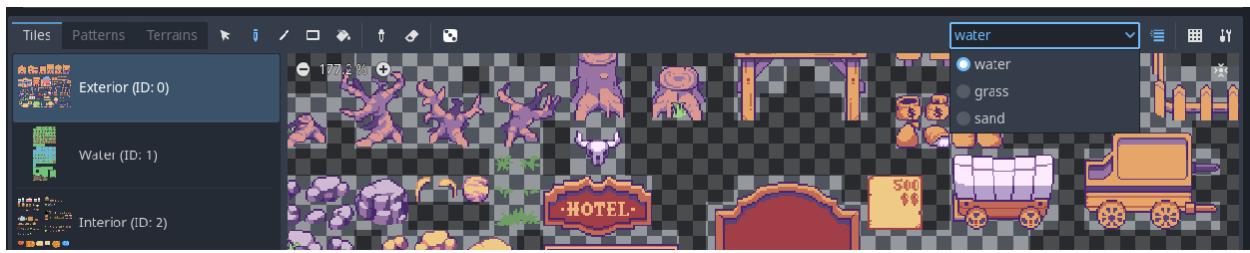
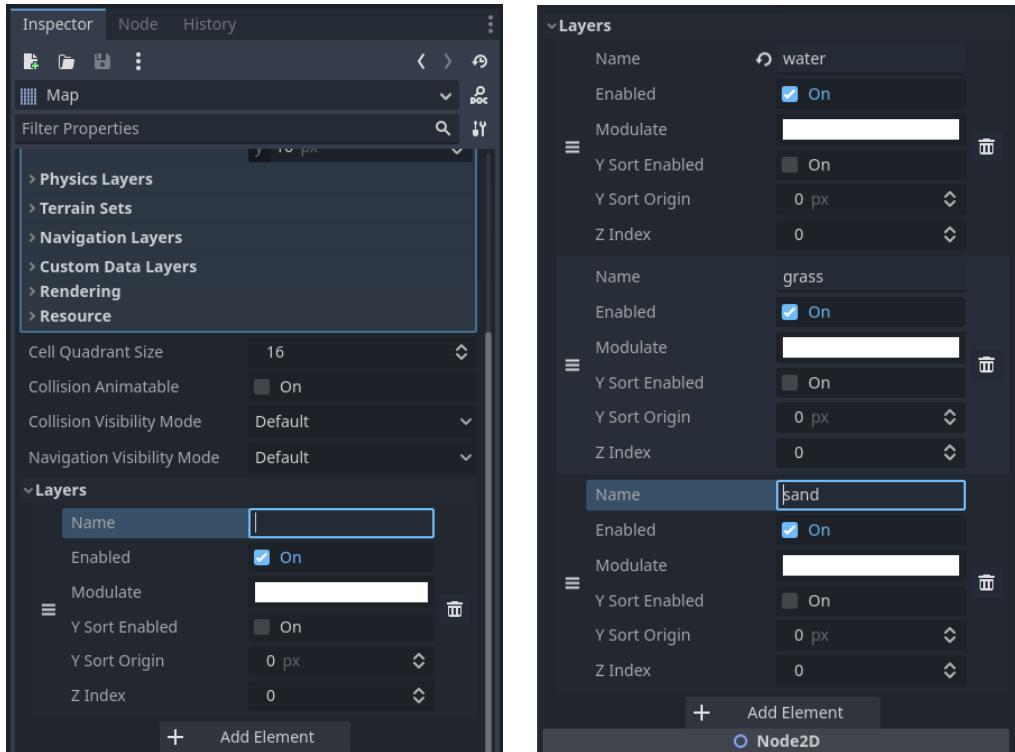


2. TileMap Layers

While we're at it, let's also create [layers](#) so that we can paint these elements on separate layers so that they can overlap. If we paint on the same layer, the tiles will erase each other.

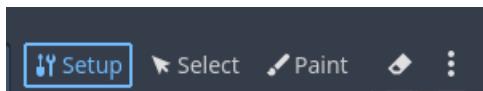


To add a new layer, simply go into the Inspector panel, and underneath the Layers property, add a new layer element. Let's add one for each of our terrain sets: water, grass, and sand. You can add more layers later on for your buildings and decorations.



3. Terrain Layer Creation (Auto-Tiling)

We can now paint on our layers separately. But first, let's create those autotiles. In your Tileset panel, on top, you'll see three options: setup, select, and paint.



In setup, we've already seen that it is here where we can change the name and ID of our tileset. In our select panel, after we've added layers such as our terrain or physics layers from the TileMap properties, we can change the properties of tiles, i.e., add collisions or autotiling to groups of tiles via selecting them and then changing them. In

the paint panel, we can do the same as in the select panel, but this time we can just paint these properties onto our tiles, i.e., painting collisions to multiple tiles at once.

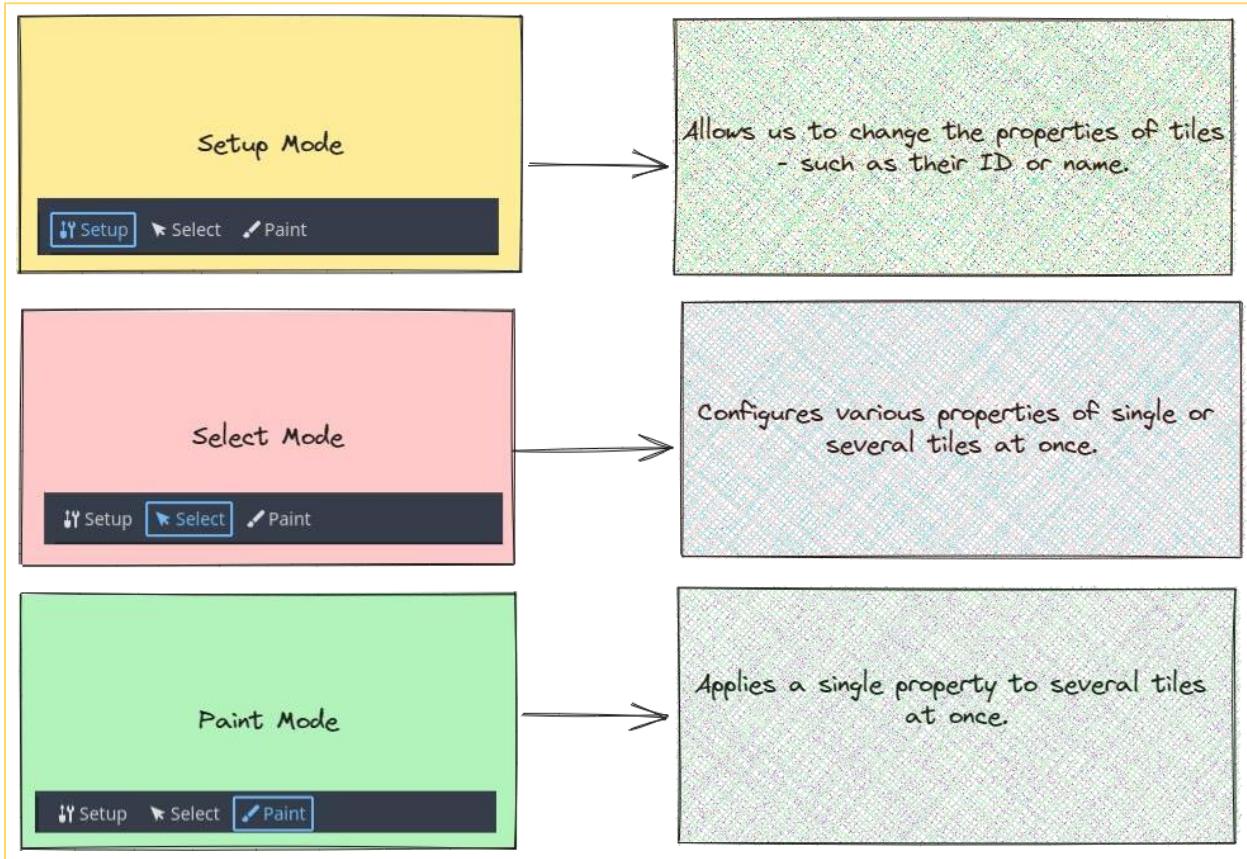
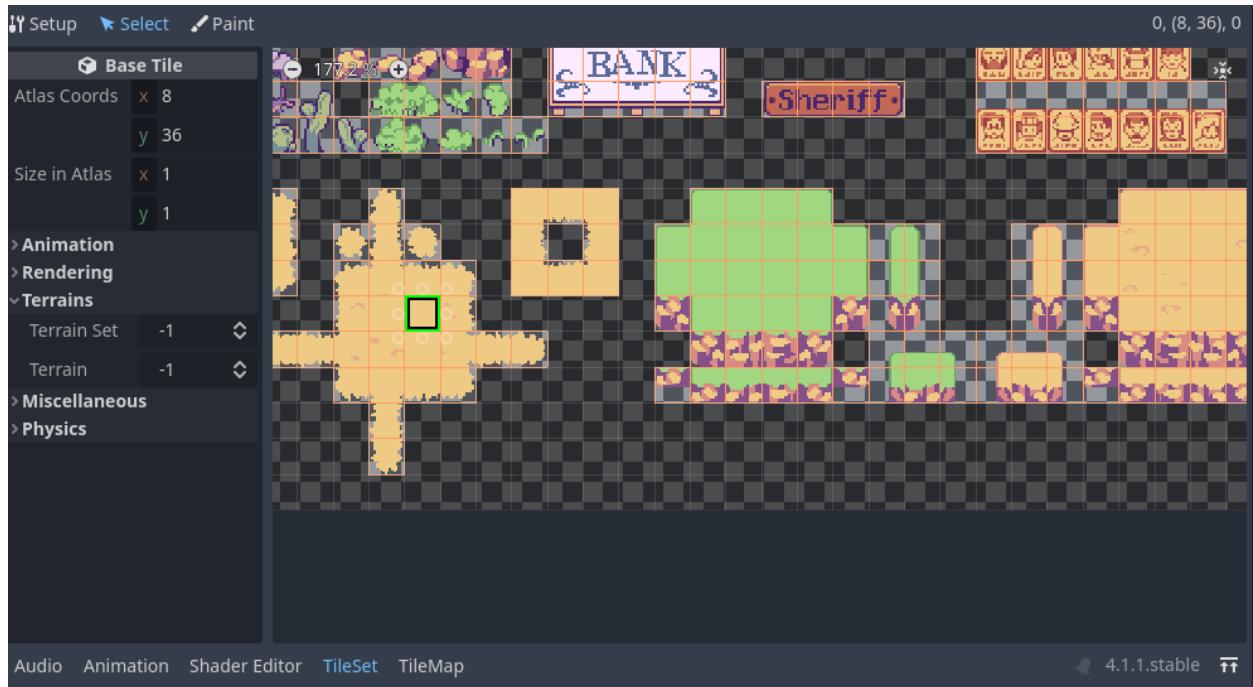


Figure 8: TileSet Modes overview

a. Autotiling in the Select Panel

Remember, the select panel only allows us to select individual or multiple tiles and then change their properties via the properties panel.

You'll see that our Terrains property has two numerical values: -1 for the Terrain set, and -1 for Terrain. This refers to the ID of our terrain sets and terrain elements. -1 means "no terrain set" or "no terrain" (it will erase the terrain), which means you must set Terrain Set to 0 or greater before you can set Terrain to 0 or greater.



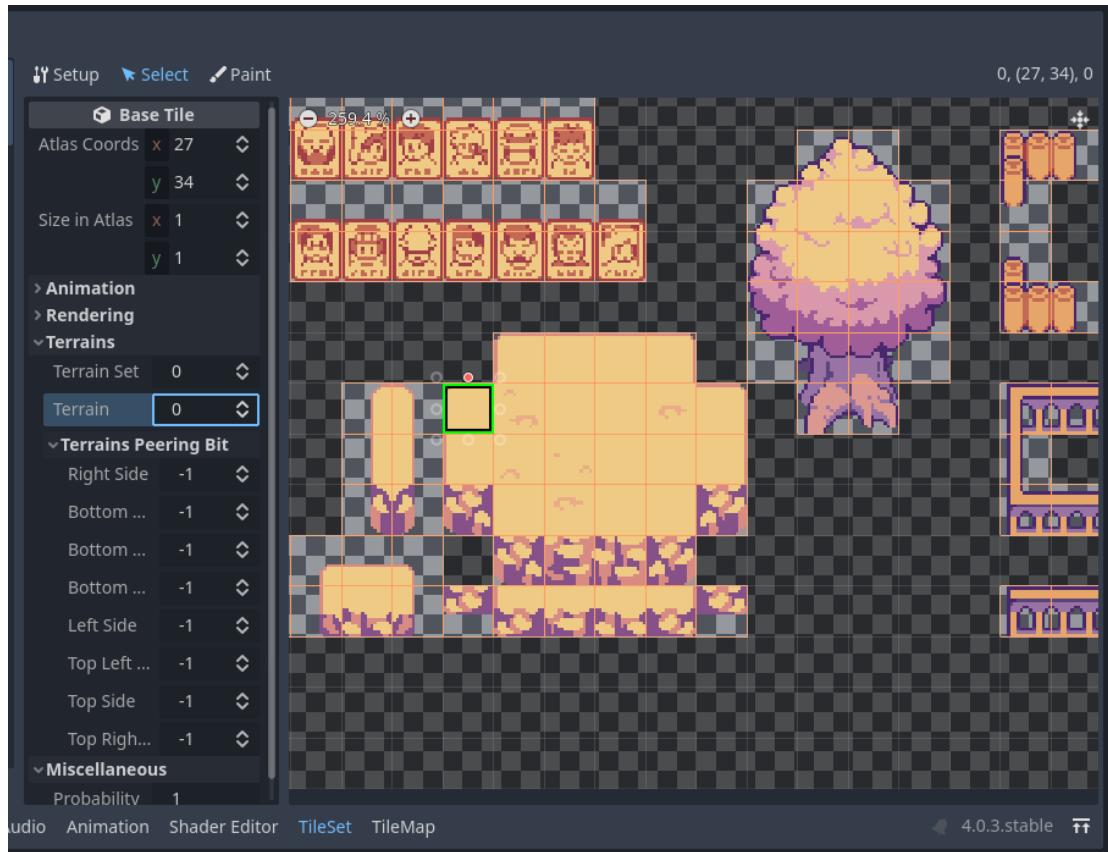
For our terrains, we have three terrain sets, one for sand, water, and grass. In each of those terrain sets, we have terrain elements. For our sand terrain set, we have the `Sand_Dessert` terrain element and the `Sand_Forest` terrain element. For our water terrain set, we only have the water terrain element. And for our grass terrain set, we only have the grass terrain element. I hope this is making sense because it might be a bit confusing or overwhelming.

For our terrain layers, our elements will correlate as follows:

Terrains in Select Panel	Terrains Layers Added
<p><i>Terrain Set 0 = Sand</i></p> <p><i>Terrain -1 = no terrain element</i></p> <p><i>Terrain 0 = Sand_Dessert element</i></p>	<p>Terrain Sets</p> <ul style="list-style-type: none"> Mode: Match Corners and Sides Terrains Name: Sand_Dessert Color: Brown Name: Sand_Forest Color: Tan <p>+ Add Element</p>

<p>Terrains</p> <p>Terrain Set 0</p> <p>Terrain 0</p>	<p><i>Terrain 1 = Sand_Forest element</i></p>
<p>Terrains</p> <p>Terrain Set 0</p> <p>Terrain 1</p>	
<p><i>Terrain Set 1 = Grass</i></p> <p><i>Terrain -1 = no terrain element</i></p> <p><i>Terrain 0 = grass element</i></p>	<p>Mode Match Corners and Sides</p> <p>Terrains</p> <ul style="list-style-type: none"> Name: Grass Color: Brown <p>+ Add Element</p>
<p>Terrains</p> <p>Terrain Set 1</p> <p>Terrain 0</p>	
<p><i>Terrain Set 2= Water</i></p> <p><i>Terrain -1 = no terrain element</i></p> <p><i>Terrain 0 = water element</i></p>	<p>Mode Match Corners and Sides</p> <p>Terrains</p> <ul style="list-style-type: none"> Name: Water Color: Brown <p>+ Add Element</p>
<p>Terrains</p> <p>Terrain Set 2</p> <p>Terrain 0</p>	

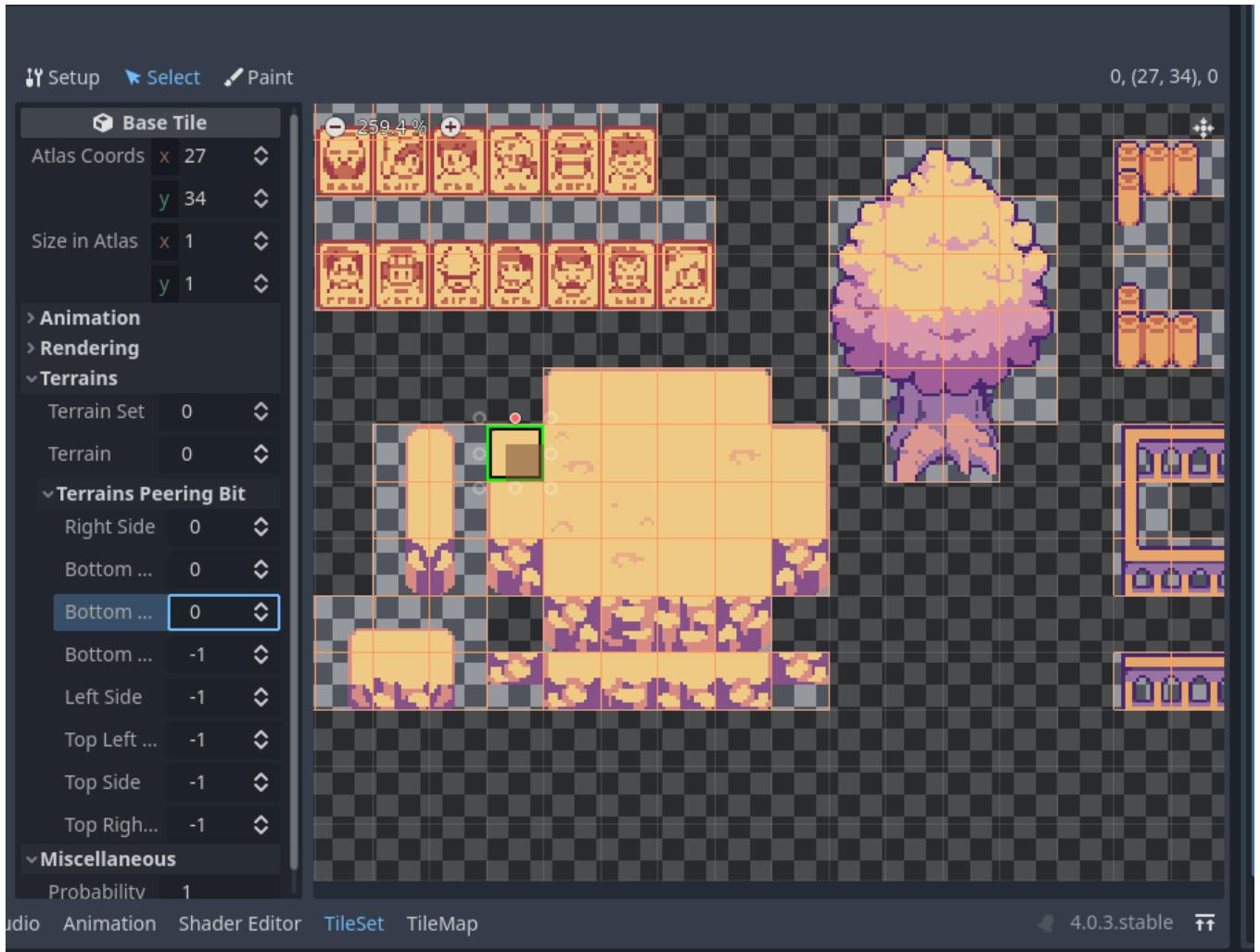
Let's add autotiles to terrain set 0 (sand), terrain element 0 (Sand_Dessert). We're going to use our Atlas.png tileset for this. Now we can go ahead and select our sand tiles. I'm going to select the top-left tile.



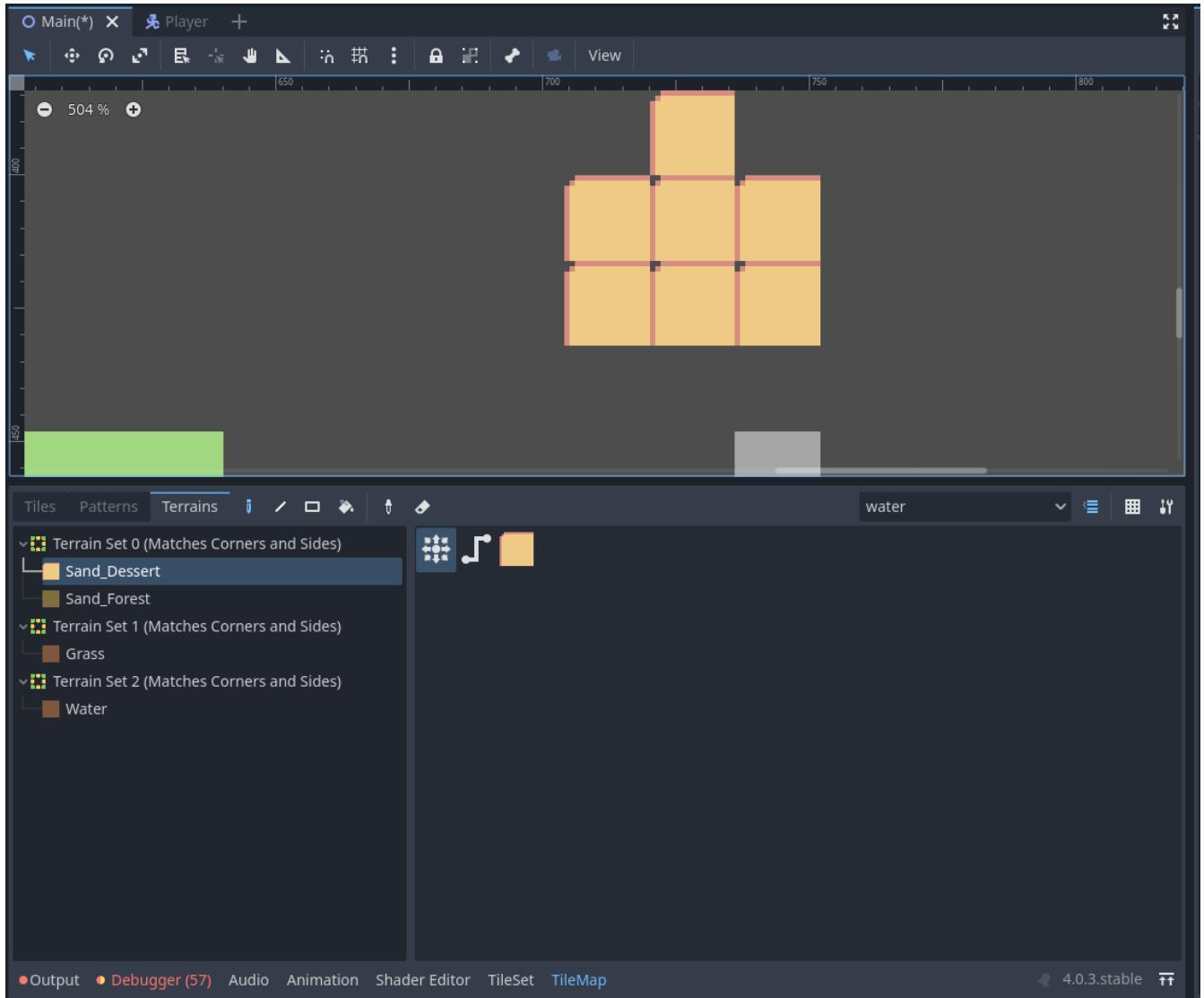
Now, we'll have to assign the bit-peering values for these tiles. The peering bits determine which tile will be placed depending on neighboring tiles. -1 is a special value that refers to empty space.

For example, if a tile has all its bits set to 0 or greater, it will only appear if all its neighboring tiles are using a tile with the same terrain ID. If a tile has its bits set to 0 or greater, but the top-left, top, and top-right bits are set to -1, it will only appear if there is empty space on top of it (including diagonally).

When we assign this value, we want to avoid assigning values to obvious corners or sharp edges.

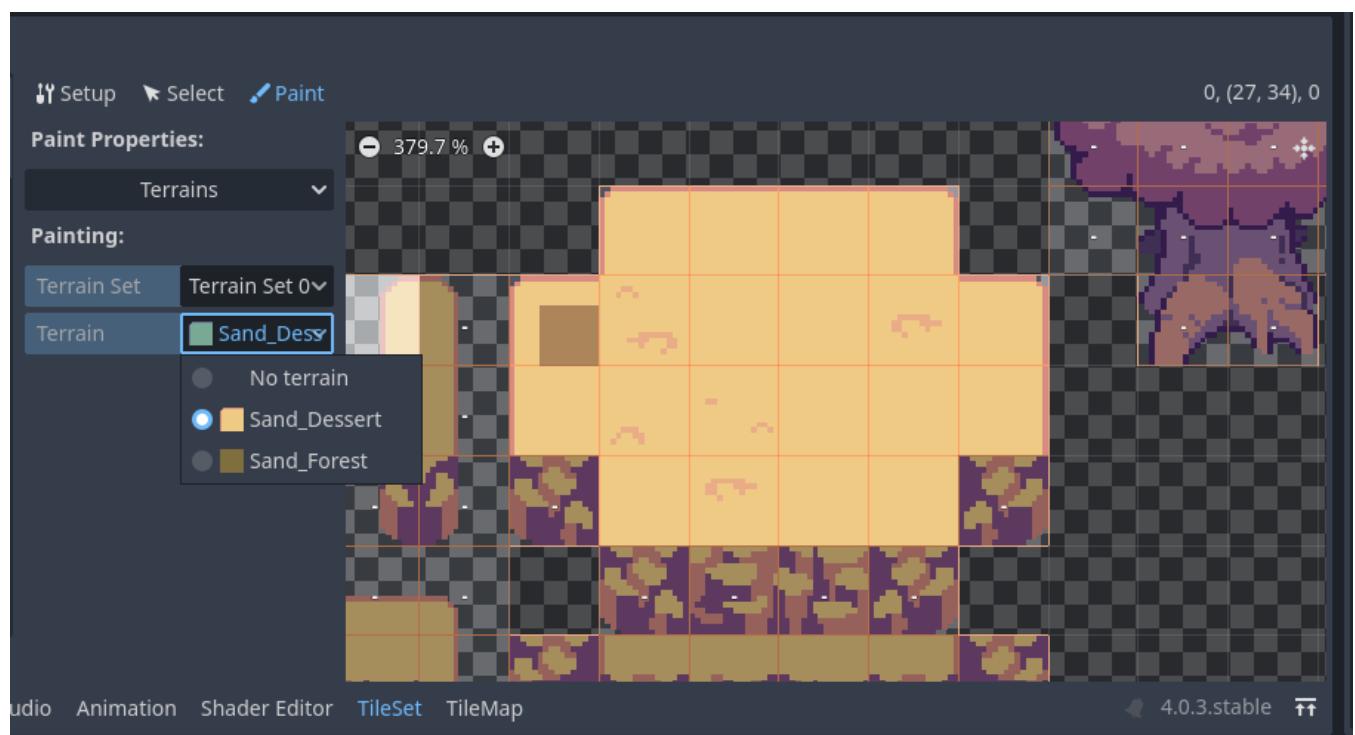
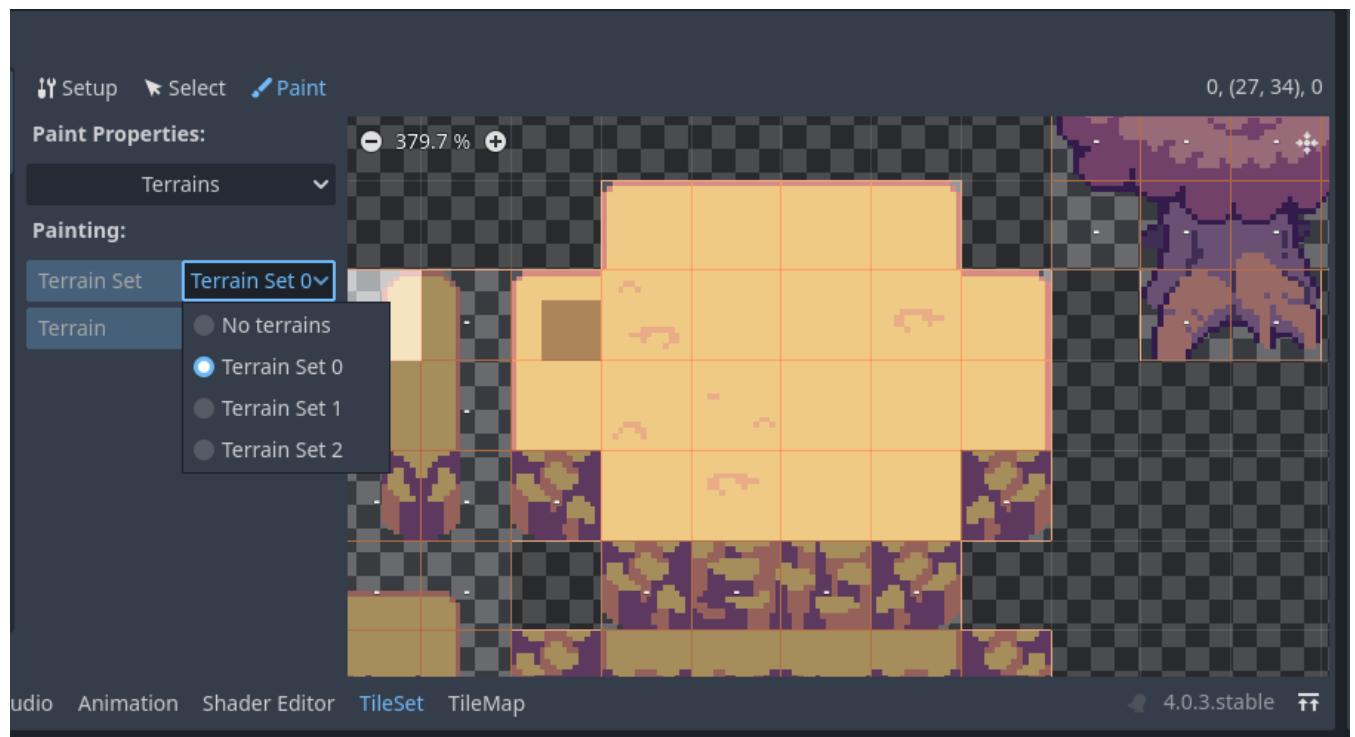


If we go back to our TileMaps panel, underneath our Sand_Dessert terrain element, we can now paint our newly created autotile onto the map. Now this won't do anything because we need to add the other sand tiles. For this, we will speed it up via the paint panel, where we can paint the bit peerings directly onto the terrain.

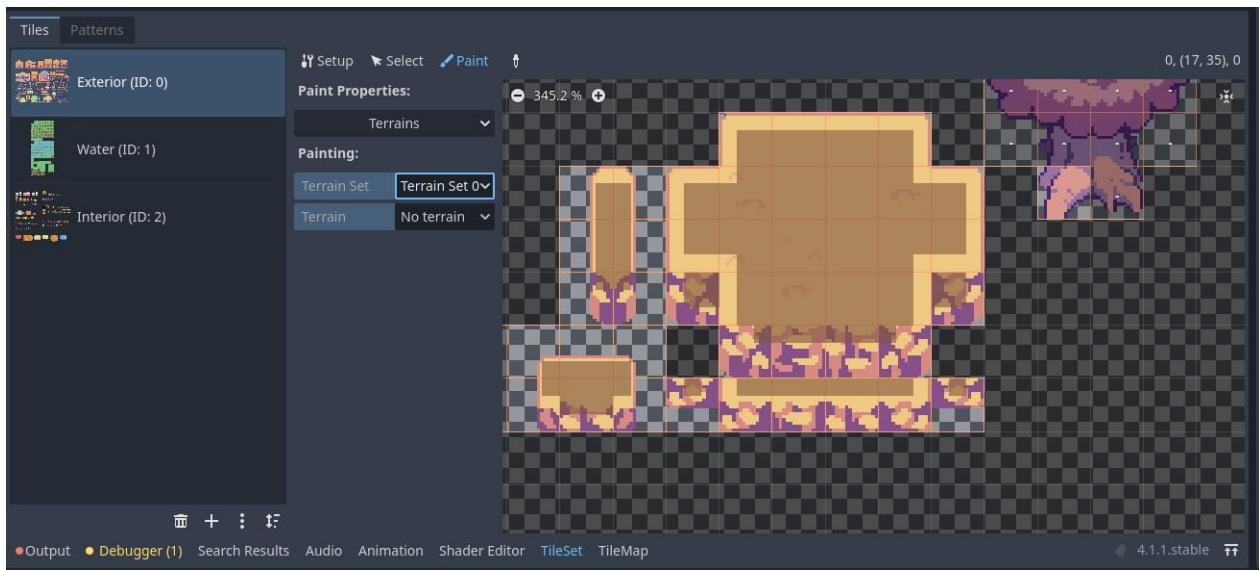


b. Autotiling in the Paint Panel

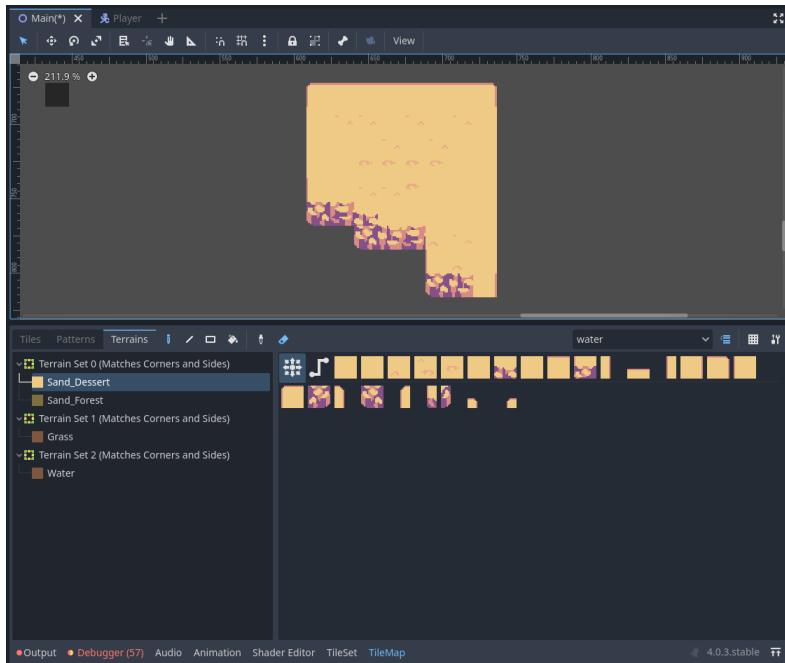
In your TileSet panel, underneath paint, you can see that you can select your terrain sets and elements directly, without providing its ID.



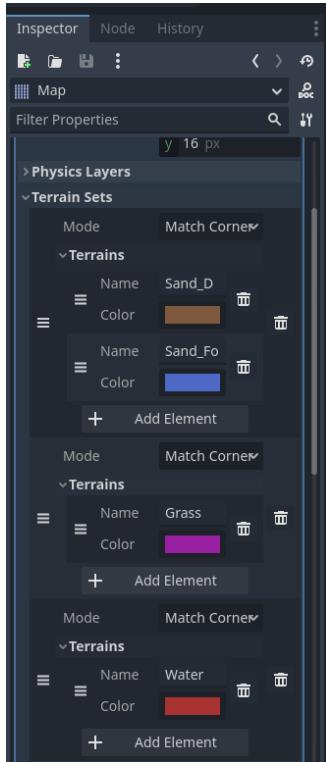
Let's make sure that our terrain set element for Sand_Dessert is selected and, on your tiles, start drawing in the bit peerings directly. Remember to avoid corners, sharp edges, or lines. To add a bit value, left click on your mouse on the tile. If you need to erase a bit value, just right-click on your added bit value to remove it.



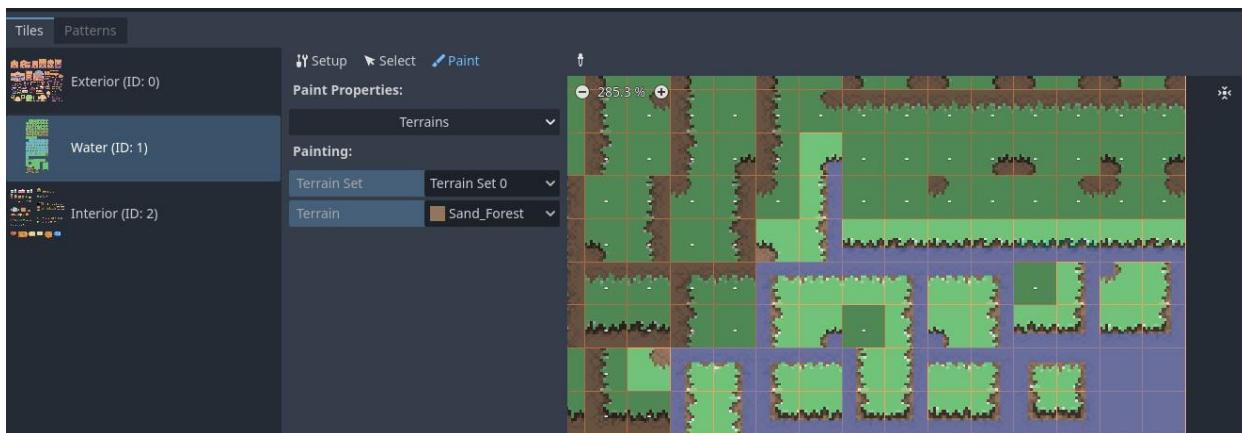
Now if you go back into your TileMap panel, you will see your other tiles added to your terrain set autotile. If you select your Sand_Dessert terrain and start drawing on your map, you'll see that it automatically draws nice, organic terrains.



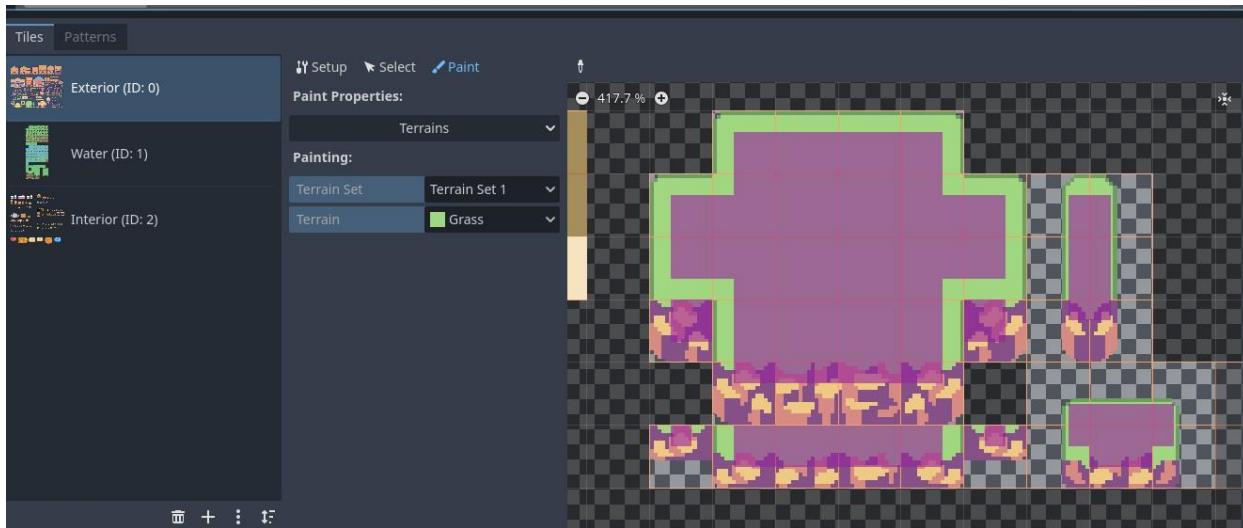
We can now go ahead and create autotiles for the rest of our terrain set elements. If you cannot see your bit blocks, you can always change the terrain element color in the Inspector panel underneath Terrain Sets > Element > Color.



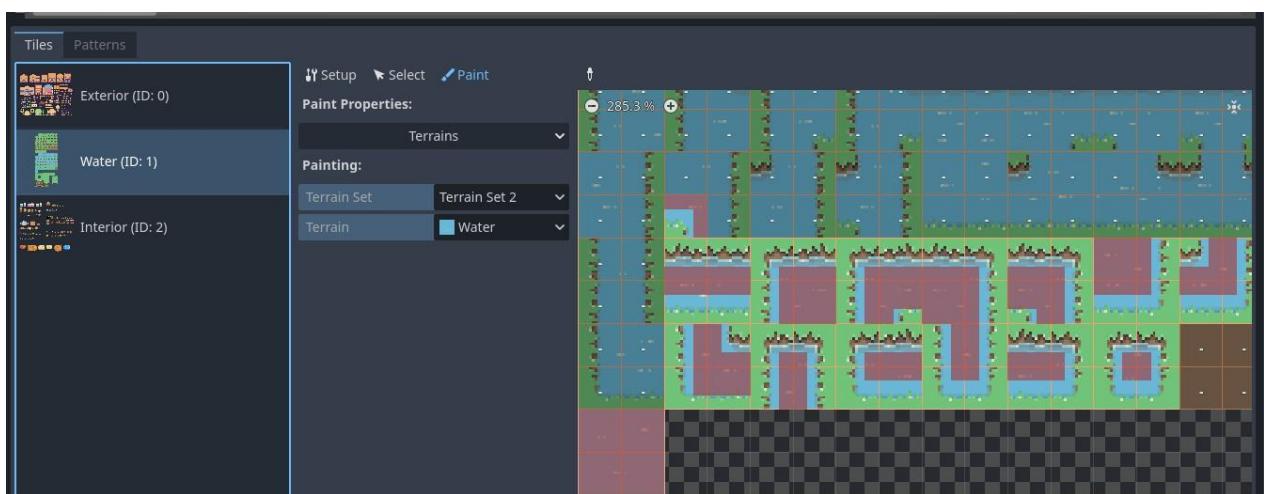
This is the terrain bits for our Sand_Forest autotile (using Water.png tileset):



This is the terrain bits for our Grass autotille (using Atlas.png tileset):

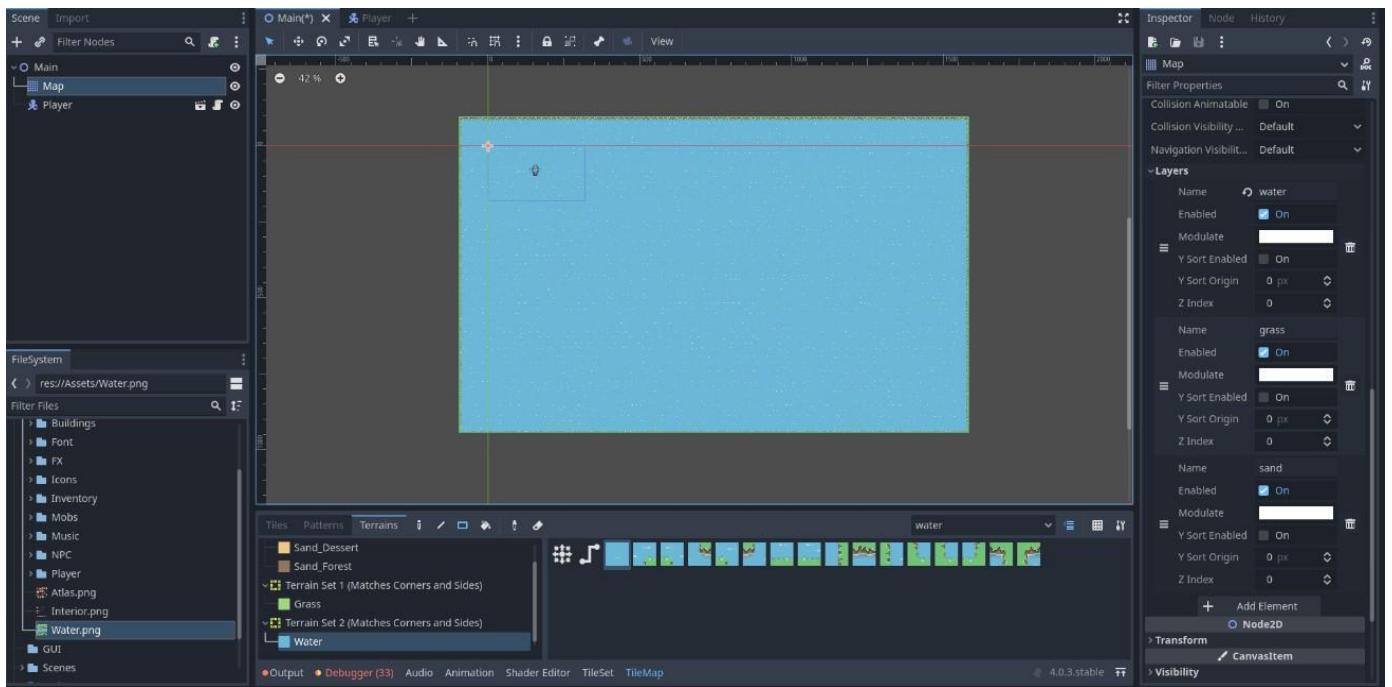


This is the terrain bits for our Water autotille (using Water.png tileset):



4. Painting Our Terrain

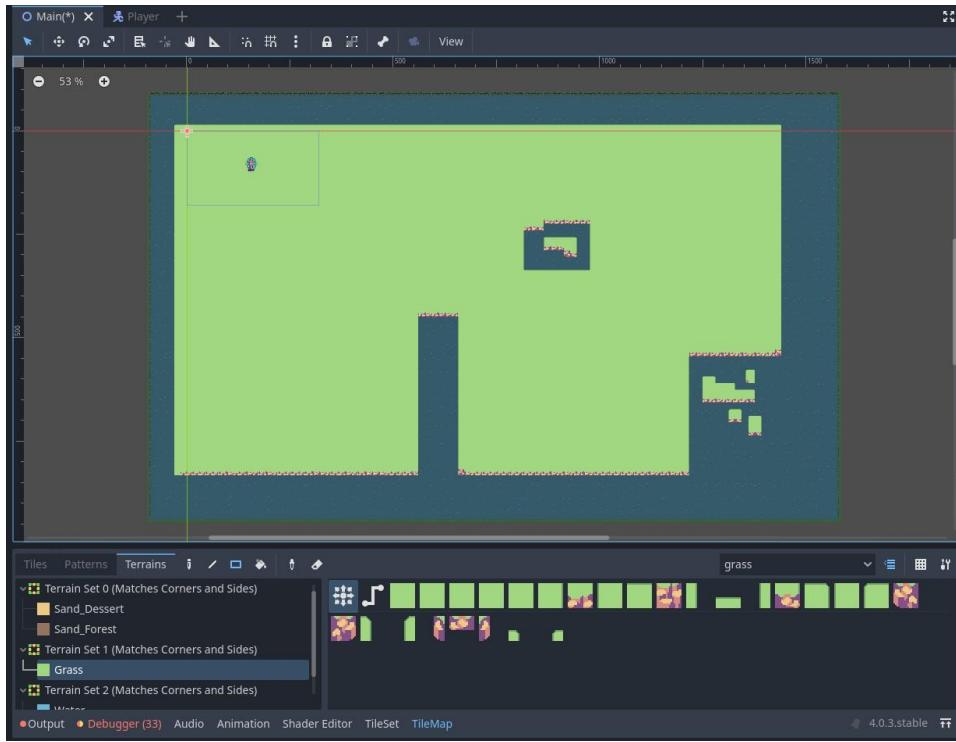
For our game, I will use my rectangle tool to draw a semi-large area of water using the water autotile. How large you draw this rectangle does not matter, what matters is how many objects you add to it. Too many items in a singular scene will take up a lot of memory. So just keep that in mind.



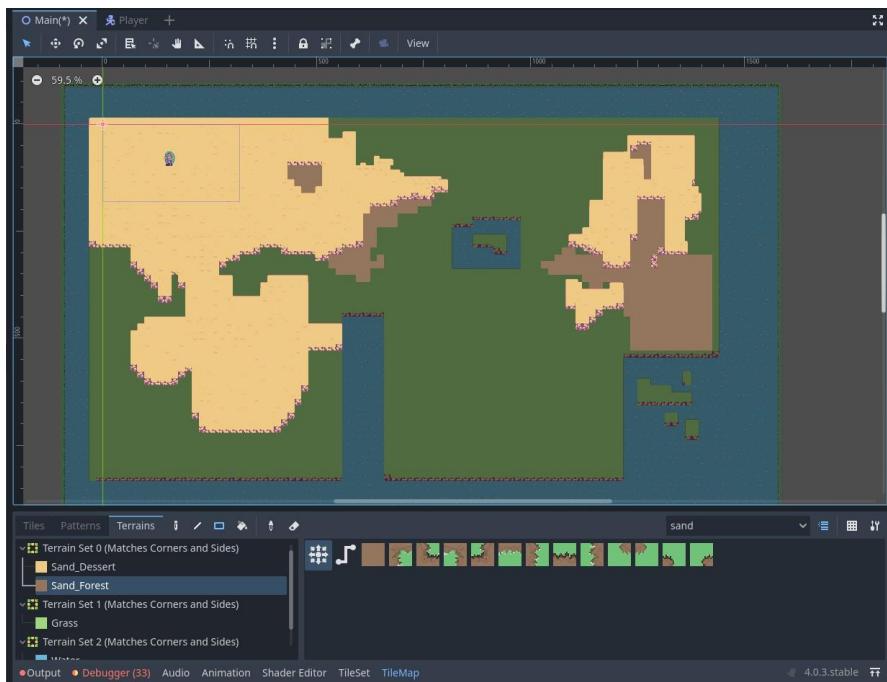
Next, on our grass layer, I'm going to draw the land for my map. Go crazy with this design. Just keep in mind your navigation paths for your map. If you want a better idea of the workflow behind designing maps, go and look at [this article](#). I'm just creating this map for tutorial's sake, so whatever I end up with does not matter since I will not be selling or releasing this game!



Figure 9: Example of a map layout



We also will draw the collisions in for the edges of the grass tiles later on. Let's finally add our sand to our sand layer. I'll only be adding a few pieces of the Sand_Forest tiles since the grass edges don't match our main grass. I just created it to show you how the sets work.



You can add the grass from the Water.png tilesheet if you want to create different environments/sections for your maps.

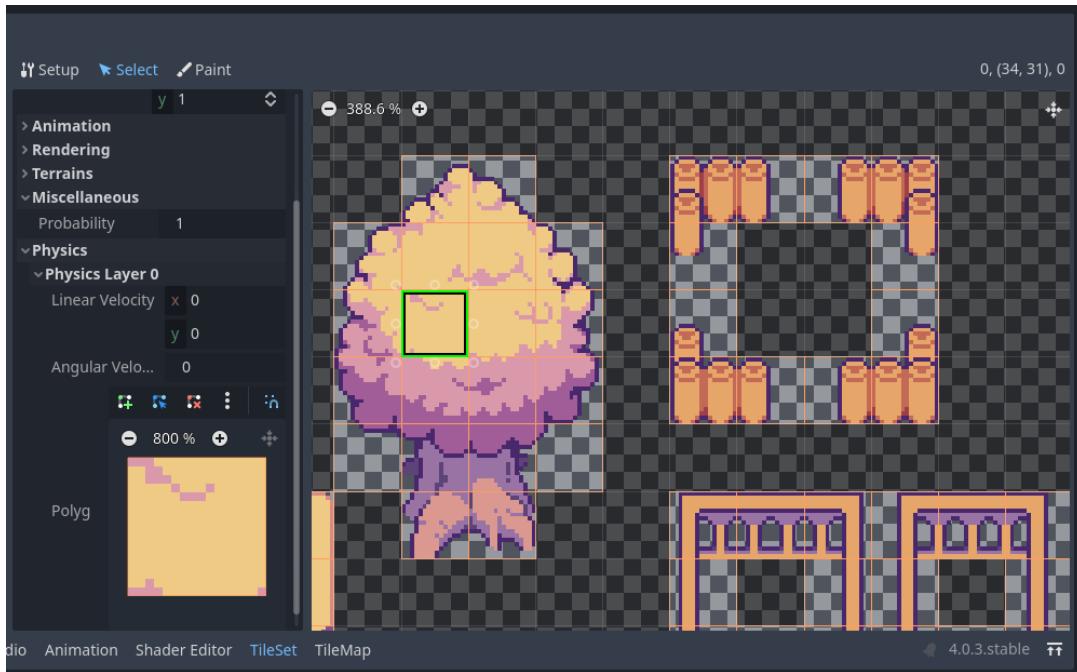
5. Adding World Items

Now that we have our terrain set down (remember, you can always make changes to it later on), we can go ahead and add things like flowers, trees, and even buildings. Before we do this though, let's add collisions to our tiles so that our player can run into items.

For this, we need to add physics layers to our TileMap. In your Inspector Panel, make sure your tileset resource is selected, and underneath Physics Layers add a new element.



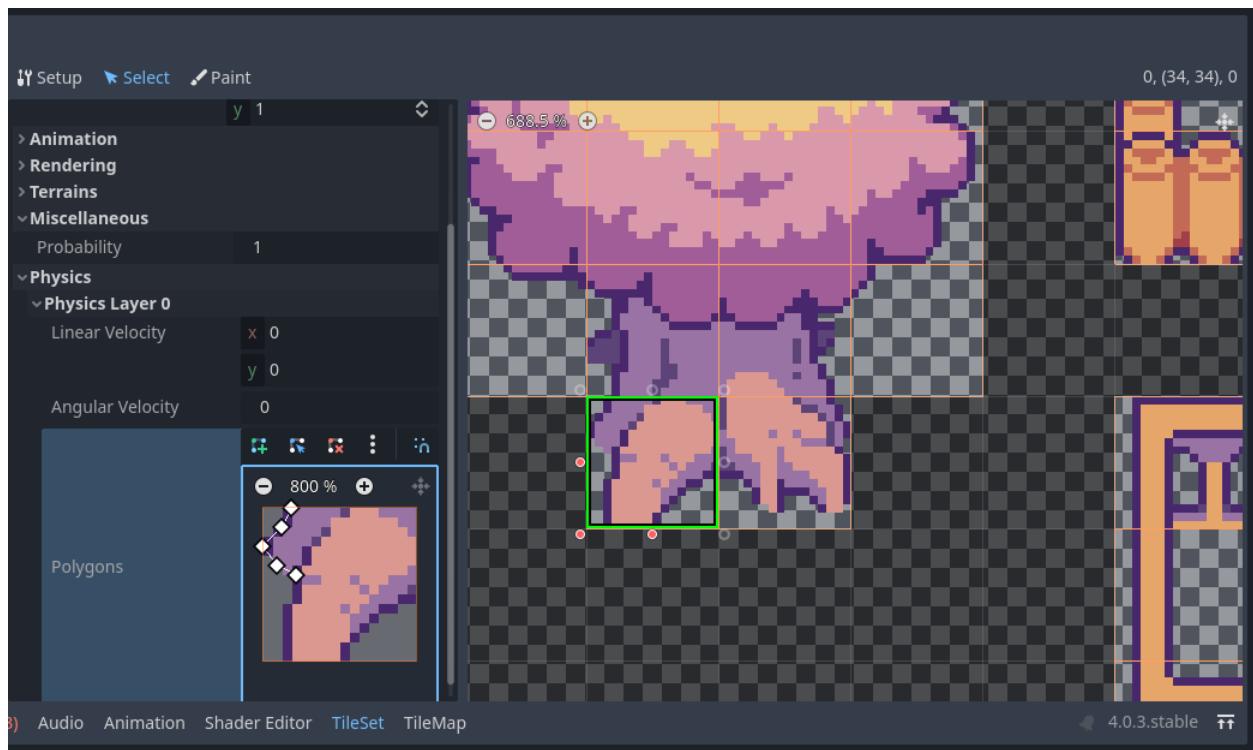
After creating a physics layer, you will now have access to the Physics Layer option in the TileSet panel.



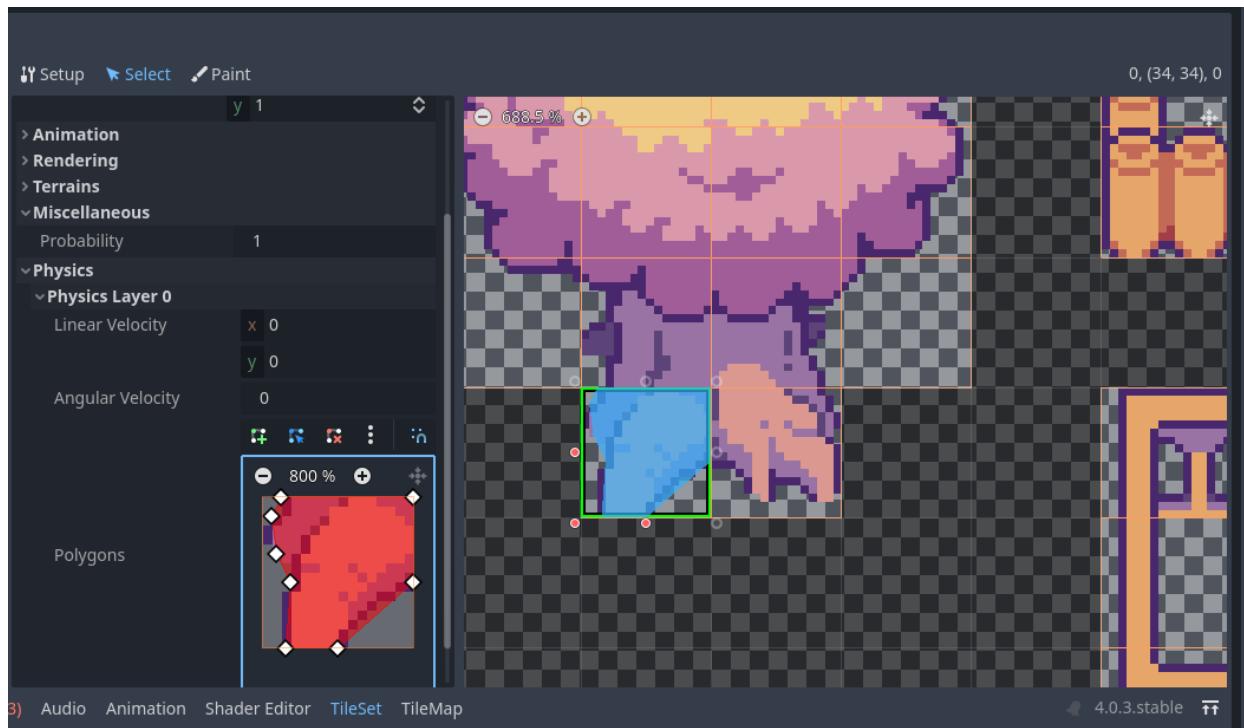
We can now add collisions by either adding them in the select panel or the paint panel. Let's do it again both ways.

a. Collisions in the Select Panel

In our select panel, let's add some collisions to our tree in our `Atlas.png` tilesheet. We don't want our player to be able to run through this tree at all, so select the bottom left root of the tree, and underneath Physics Layer > Polygons, we can start drawing in our collisions in that little box. Let's border the entire image outline. You can edit and delete the points with point tools above the box once you've completed your border.

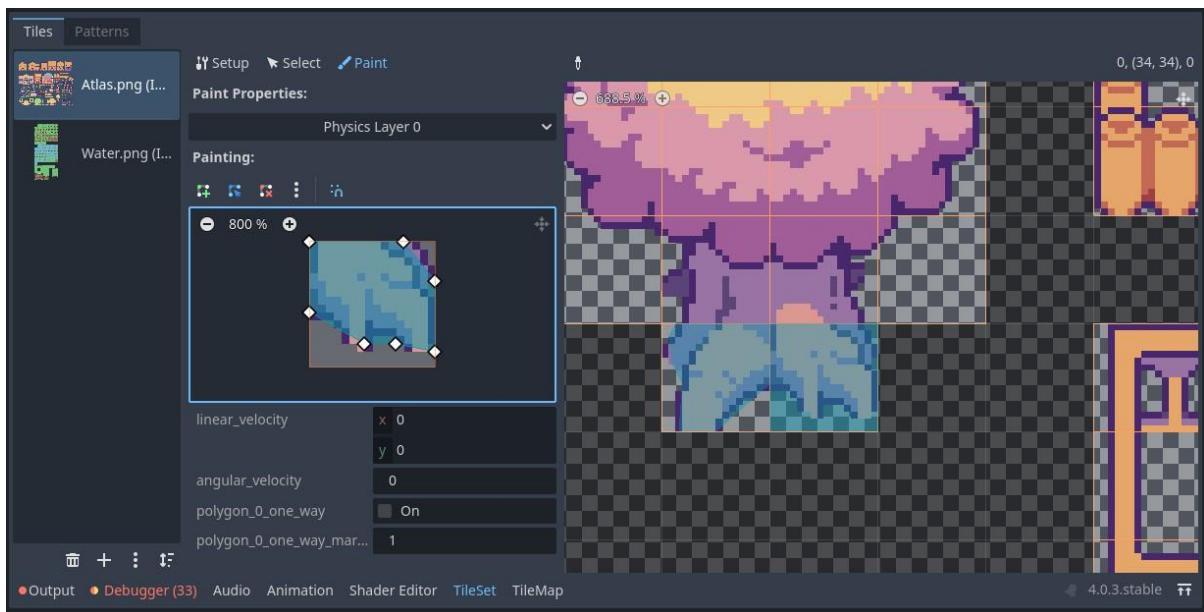
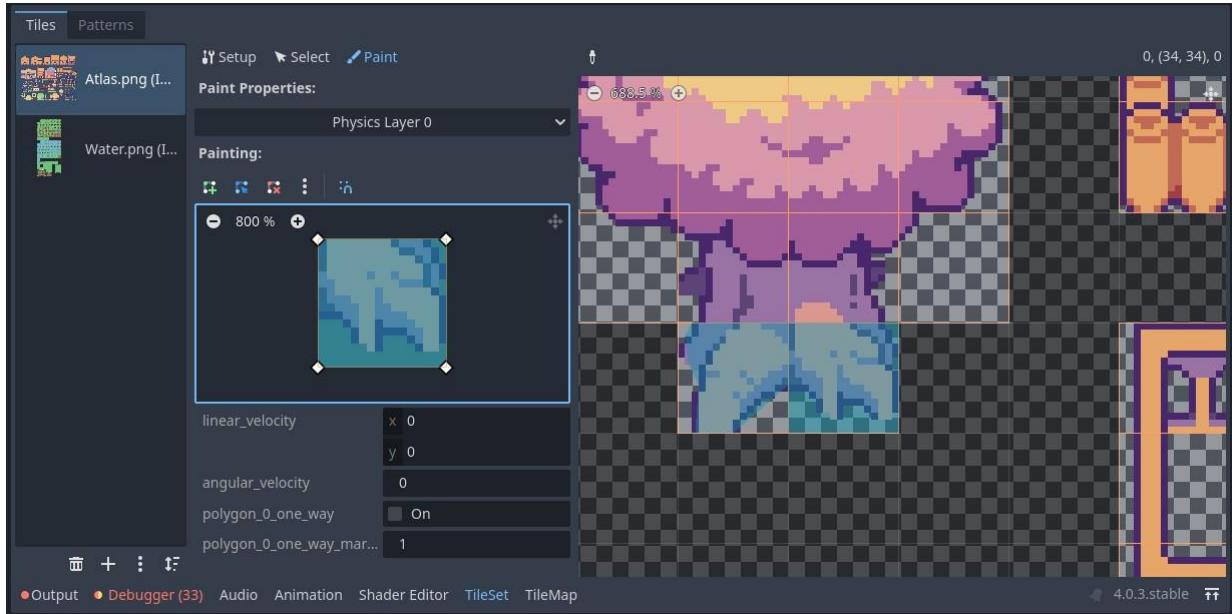


Once you've finished drawing your collisions border or outline, and you've connected your last points, you will see that the shape drawn turns red. This means it now has a collision polygon added to it.

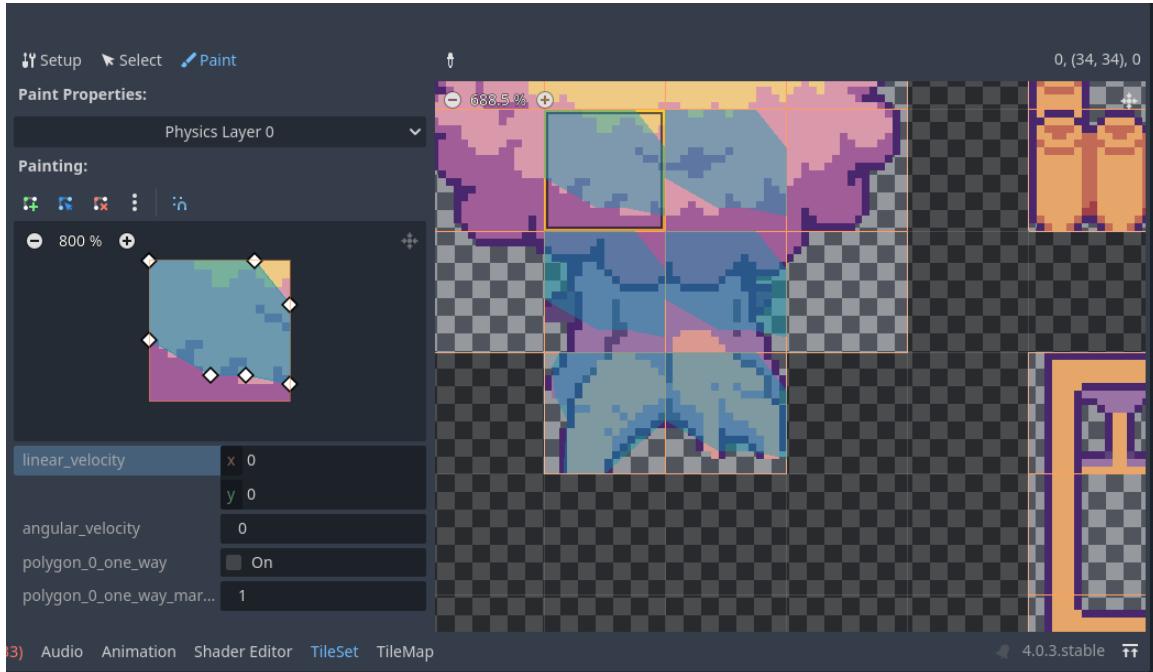


b. Collisions in the Paint Panel

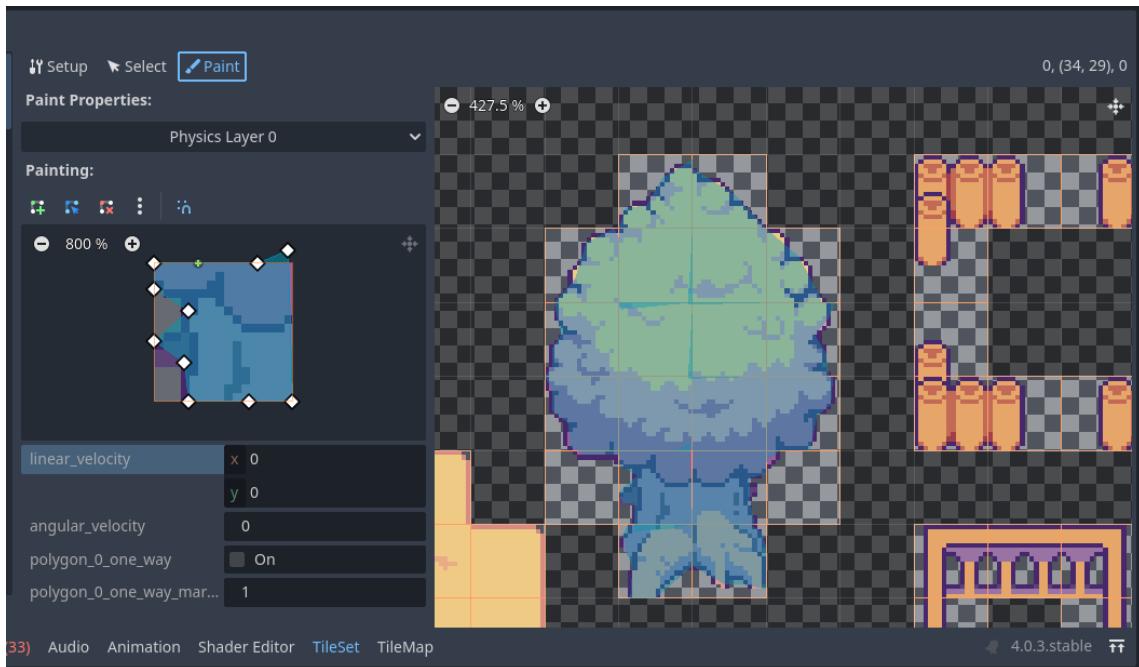
You can also paint your collisions onto your shape via the paint panel. Select your Physics Layer in your paint panel and click on another tree tile. If you click on it, a box will appear. The blue means that a collision shape has been added to the entire tile, but you can drag in new points to outline the shape you want.



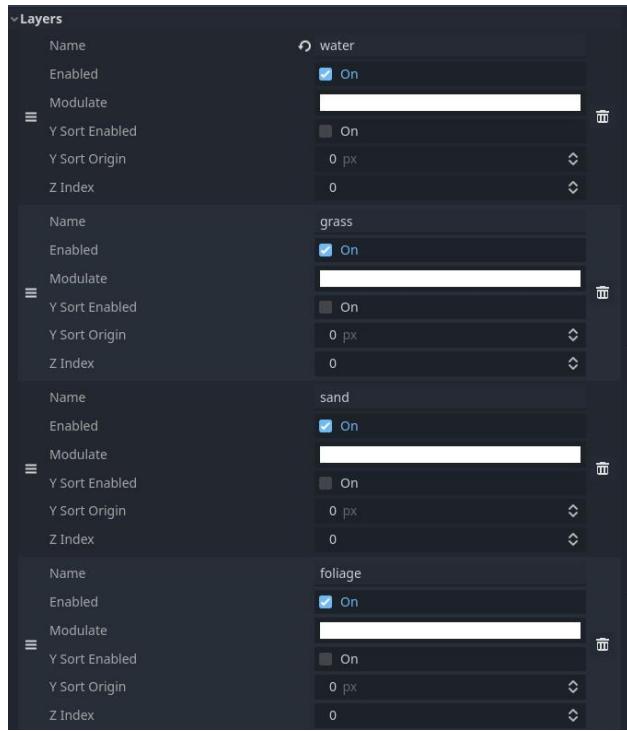
Now, click on the tile you want to assign the collision shape to - to assign it. You can also paint this collision shape onto multiple tiles if you'd like.



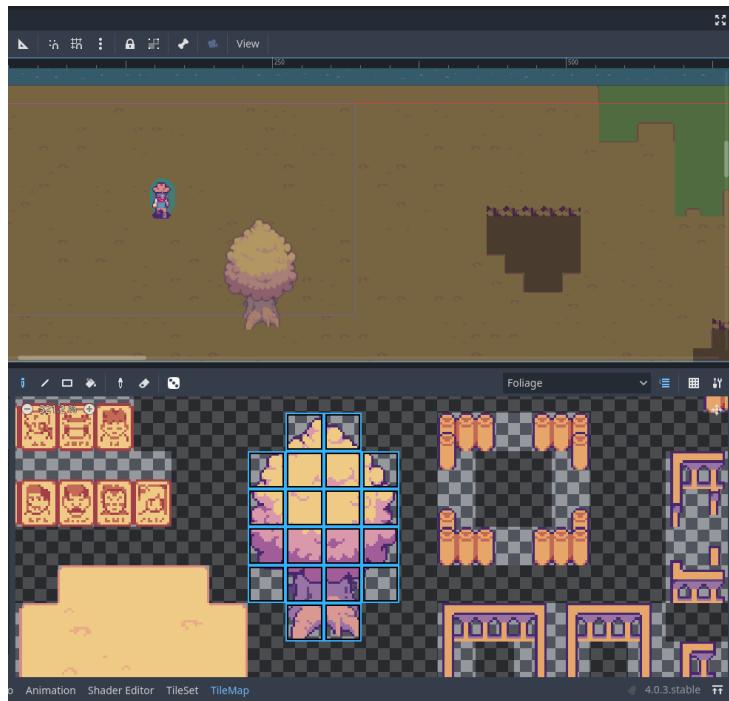
I'm not going to do that, so let's go ahead and draw in the collision shapes for the rest of our tree.



Let's create a new layer and call it foliage.

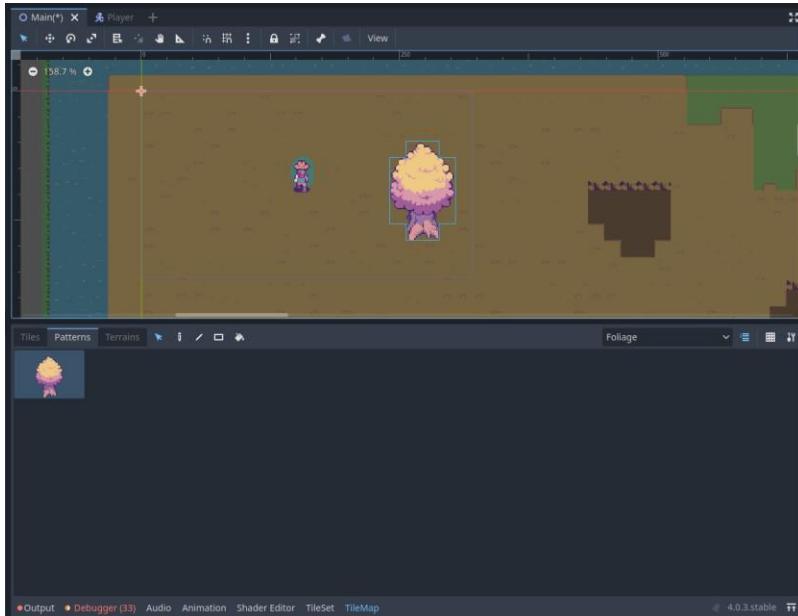


Now on this layer, select the tiles that make up the tree, and paint it onto your map (close to your player).

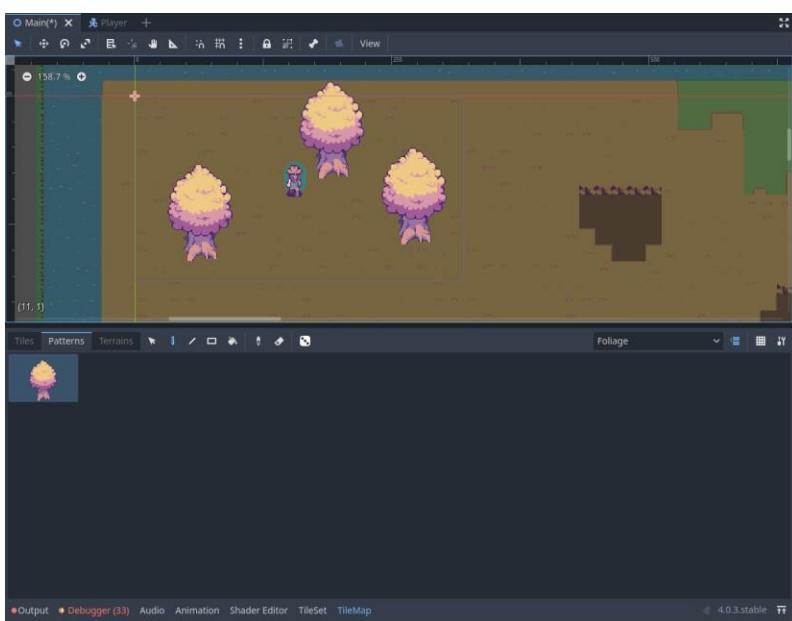


You can also go ahead and use your select tool to select the entire tree to create a pattern of the tree so that we don't have to keep on re-selecting all the tiles just to create new trees.

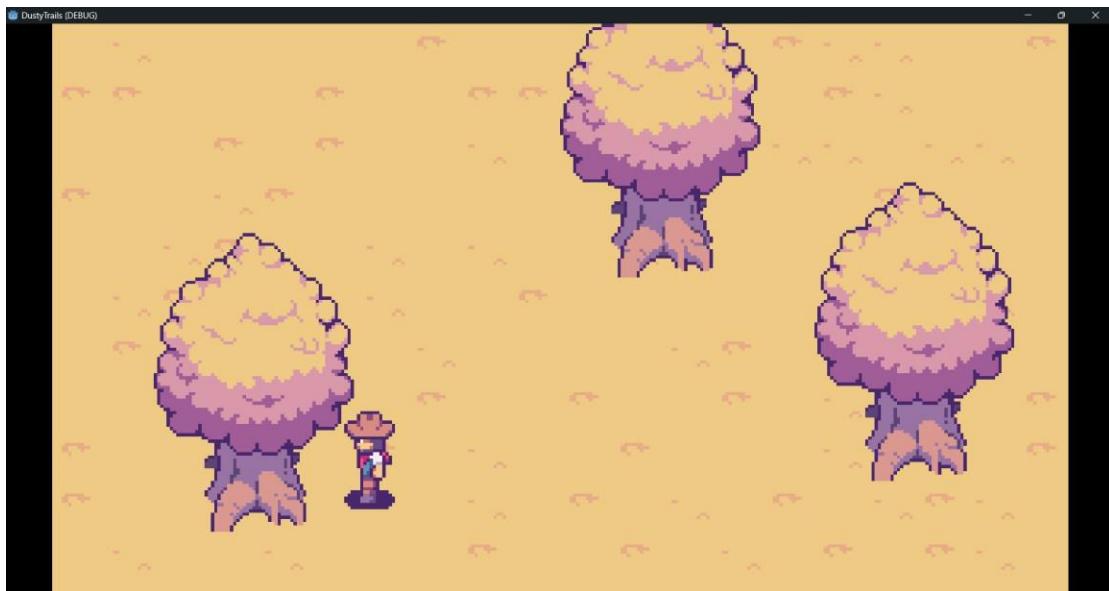
You can just copy and paste or drag this selected item into the Patterns panel in your TileMap option.



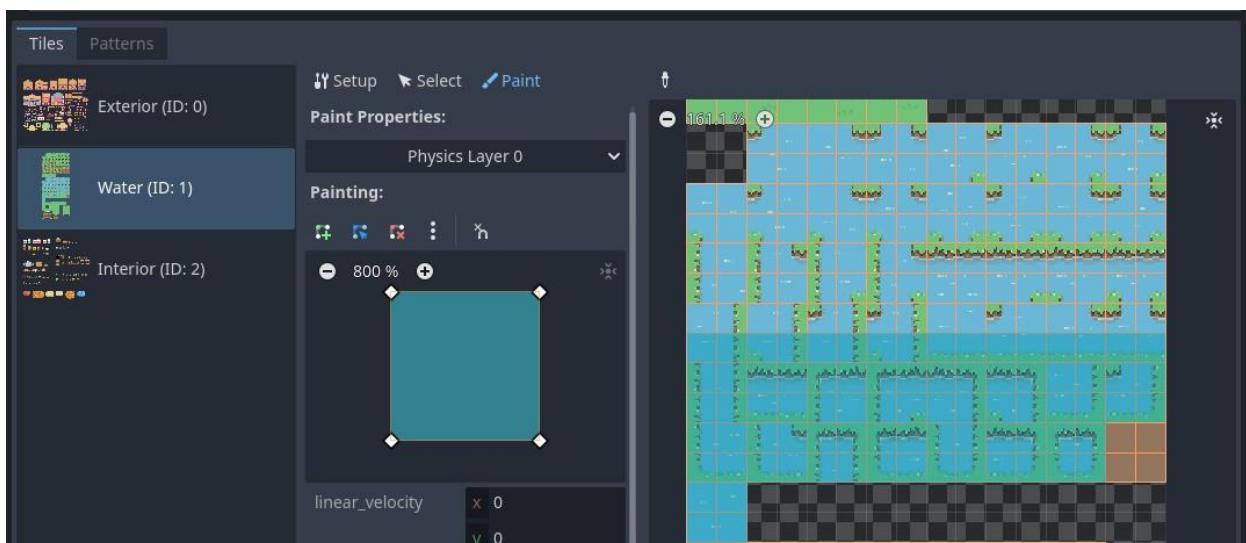
Now you can easily paint your tree as a singular object.

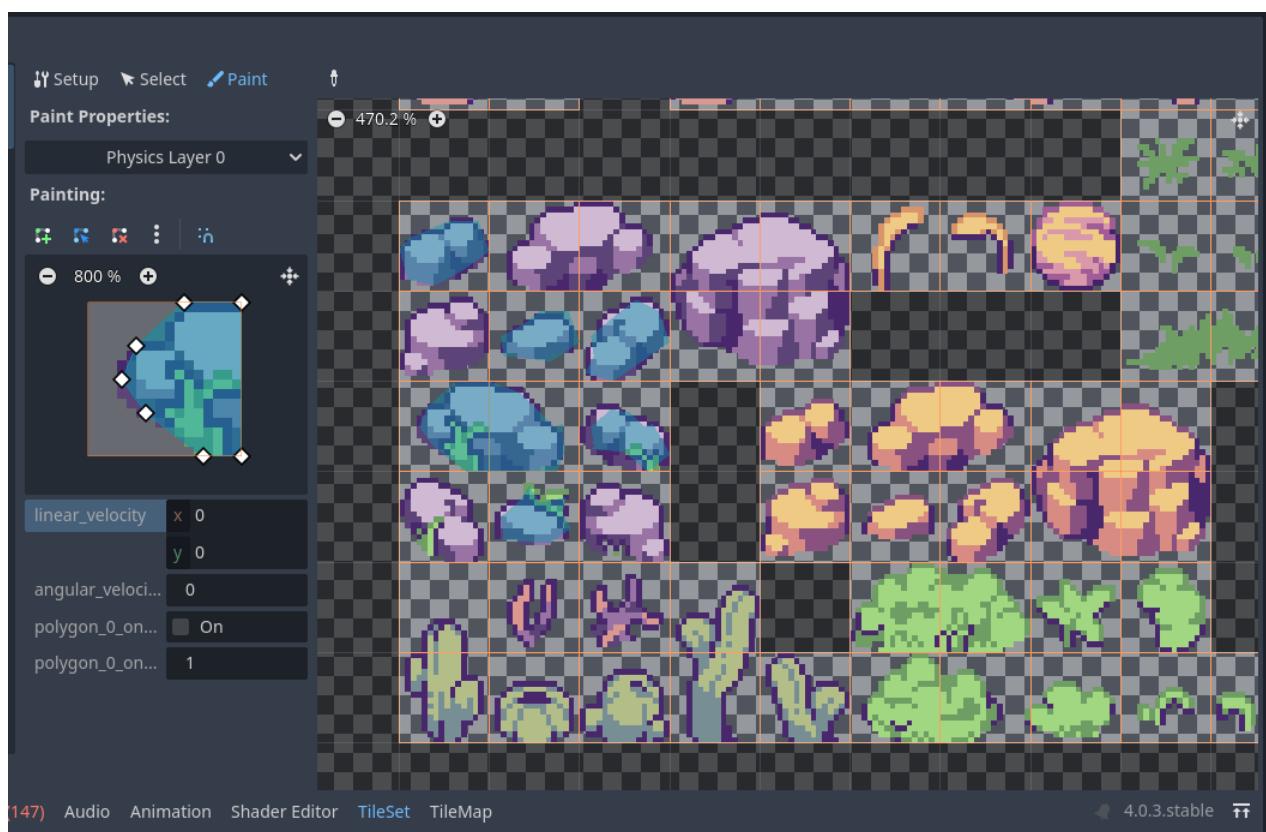
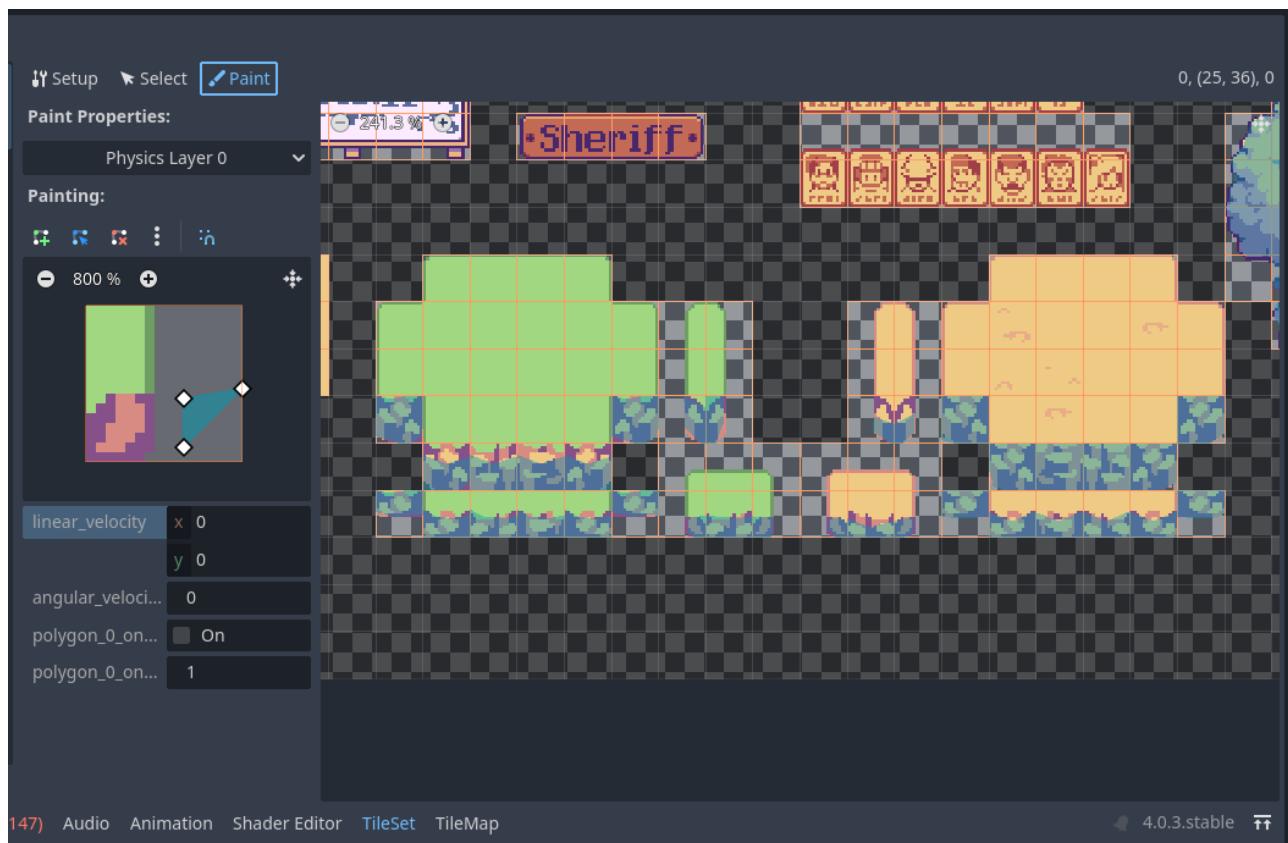


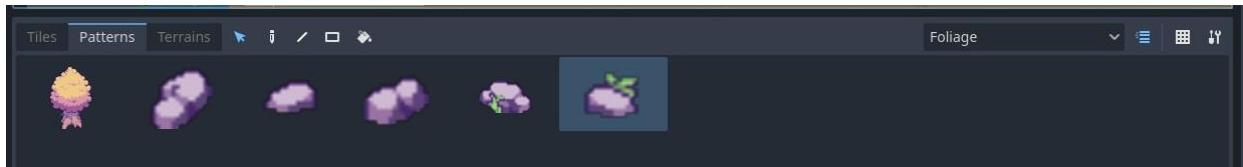
If you run your scene, you will now see that your player runs into the trees, and not through them. Let's add collisions to other objects, such as rocks, and our terrain edges.



You can go ahead and add collisions to your water and anything else you want to add to your world! Also, don't forget to create patterns for your tiles.

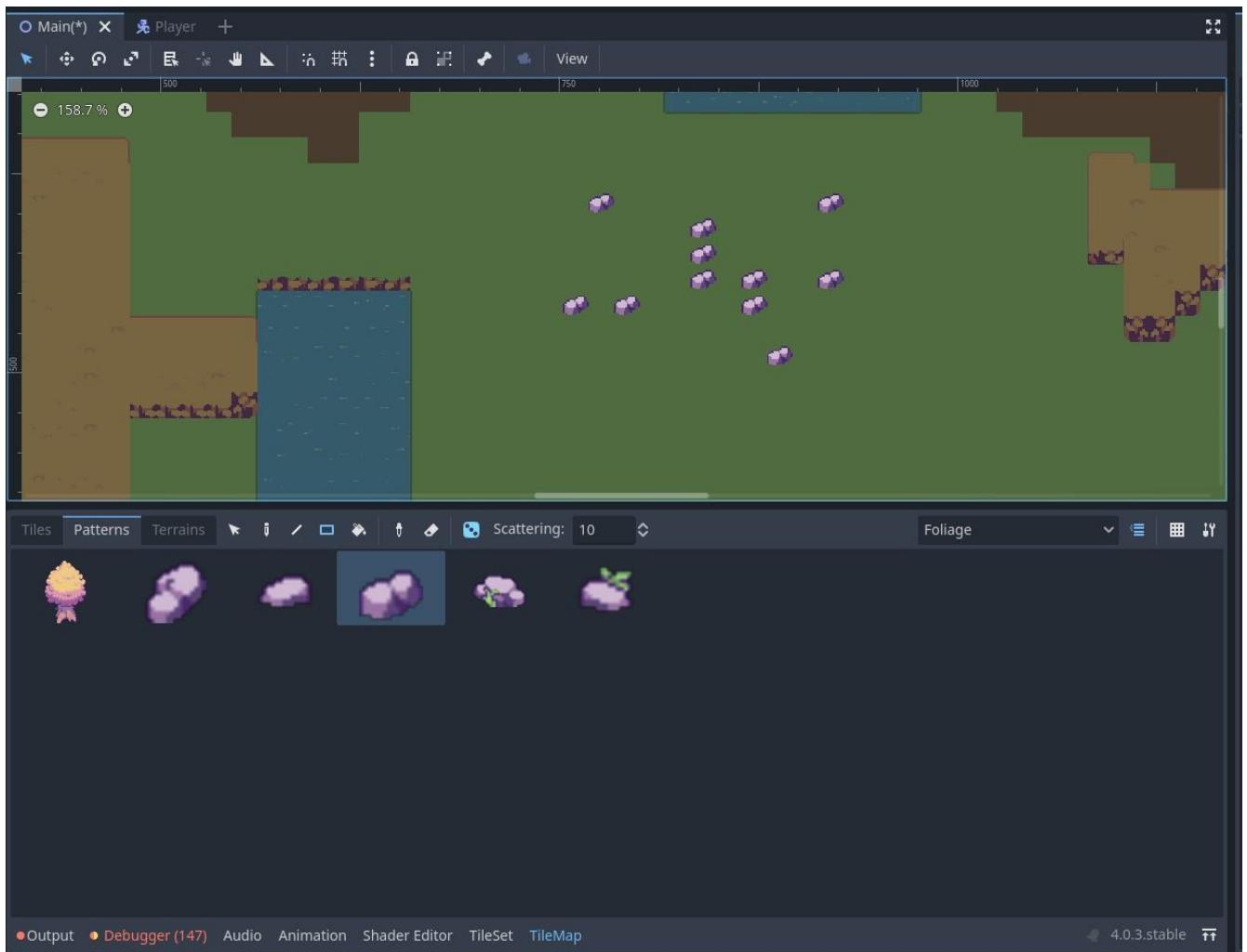






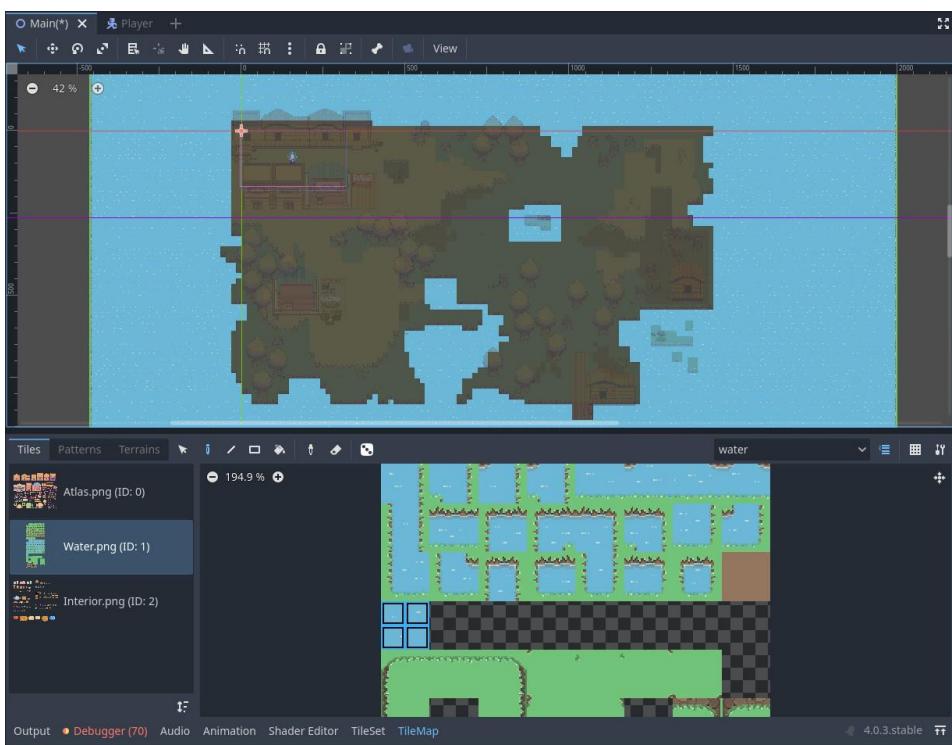
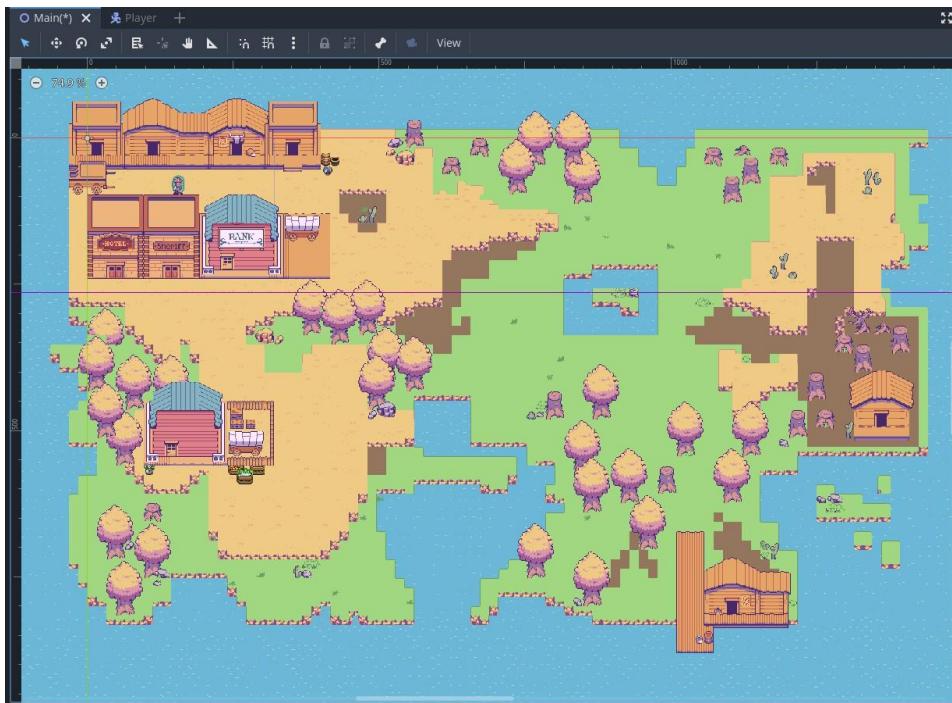
6. Create Your Map

You can also use the scattering tool (the dice icon) to place these tiles randomly on the map. The higher the scattering value, the fewer tiles will be placed in a close radius of each other.

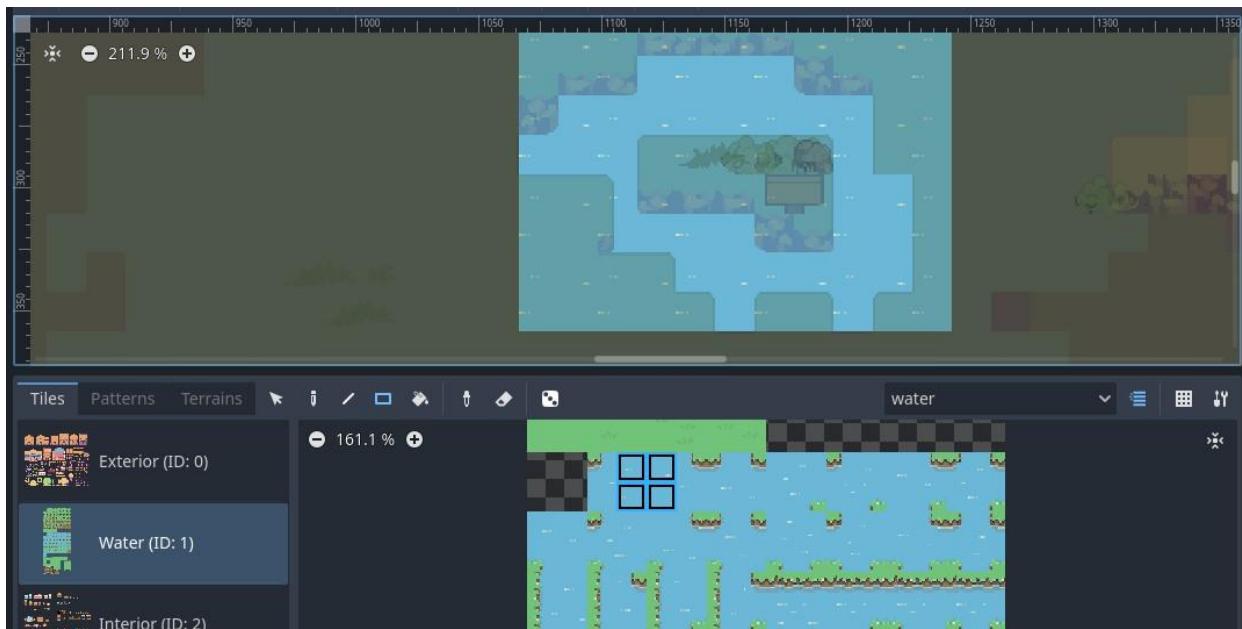


I'm going to go ahead and create a basic map with placeholder buildings and such. Really let your creativity flow here.

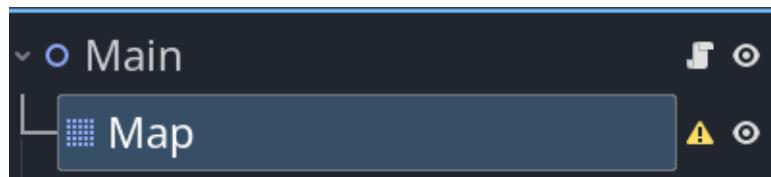
If you notice that your player isn't moving, it's because of the water collisions. You can fix this problem by just removing the water tiles from underneath your land/ground.



You can also fill the area where the player should walk with water tiles that do not have collisions added to them.



If you are following this tutorial in Godot 4.3 and above, you will get a deprecation warning next to your TileMap node. This is because the TileMap node has been replaced with the [TileMapLayer](#) node, and it works exactly like the TileMap node, but unlike the TileMap node, the TileMapLayer has only one layer of tiles.



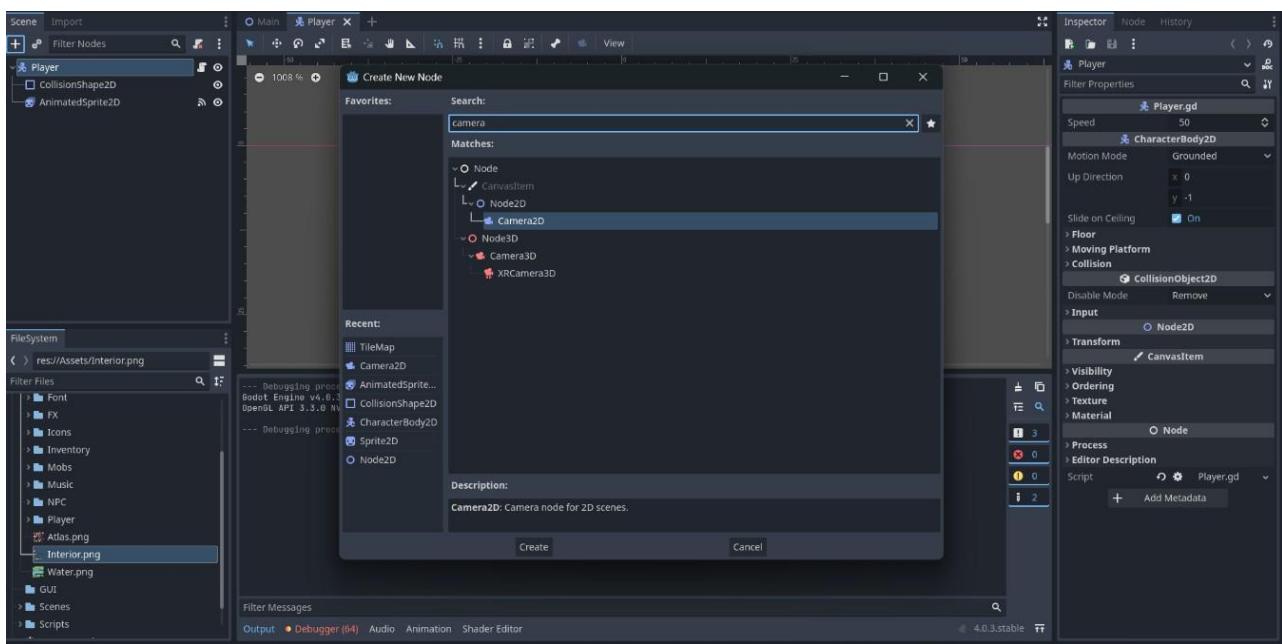
Now, because I wrote this tutorial using Godot version 1.2.1, I will still be using the deprecated node throughout this booklet. Remember, just because something is deprecated, doesn't always mean it's unusable. Godot still allows us to use the TileMap node, and the project will still work 100% with it - it's just that in the future, the Godot team will only provide support for the TileMapLayer node.

But don't worry about missing out, because I want you to be able to use Godot to its full extent, so I have added a new part at the end of this booklet where we will convert our TileMap into TileMapLayers, and we will update our code.

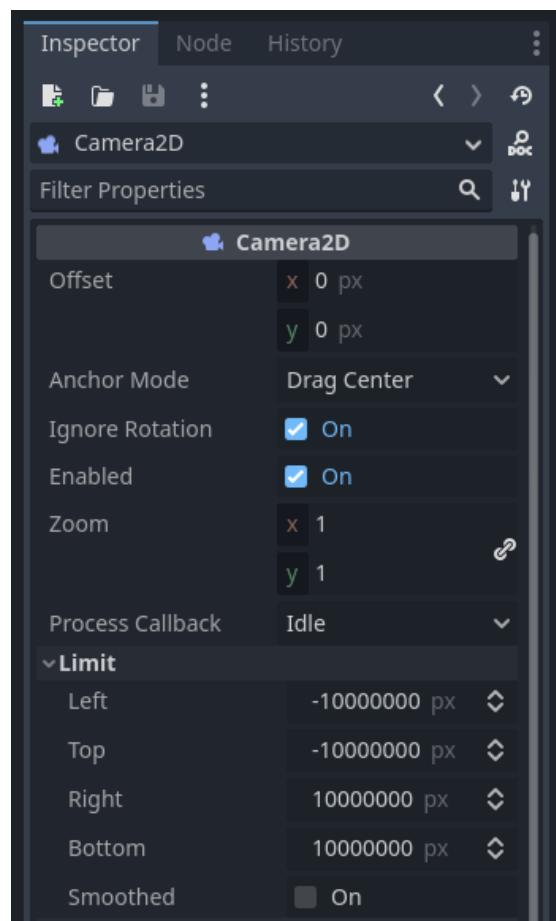
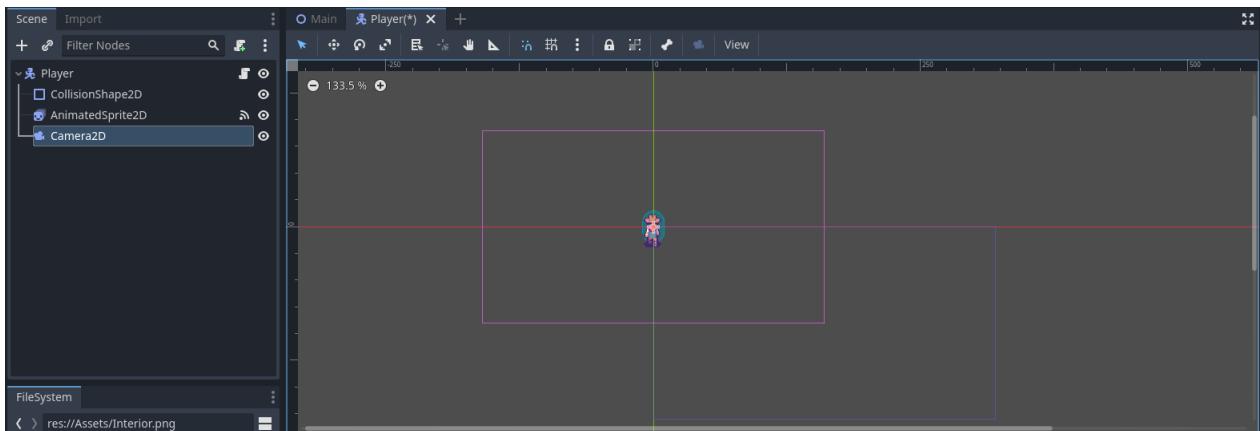
PLAYER CAMERA

Okay, now that we have our map, we will notice that if we run our scene, our player will run off of the map and we can't follow them! To fix this issue, we need to add a camera to our player. If you want to capture your entire world, you will attach the camera to your main scene, but if you want to follow a specific character around, you will attach the camera to their scene.

Let's open up our Player scene, and in the Scene Dock, let's add a [Camera2D](#) node.



You will notice that it creates a purple border around your player. This is the camera frame, and we can add [limits](#) to these edges so that our player cannot move or see beyond a certain point in our map. Since we only have water around our map, we do not need to worry about this limit for now.



If you were to run your scene now, your camera will follow your player around your newly created map!



And that's it for this part. I know this was a very long section, and it might have even overwhelmed you a bit, but don't worry because in the next part of this tutorial series, we will be doing something new and simple when we start creating the game's GUI. Remember to save your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 5: SETTING UP THE GAME GUI #1

Before we get to the fun parts, such as setting up enemies and quests, I first want us to get the creation of the game's GUI out of the way. The GUI will allow us to visually show the changes in our player's stats, such as their ammo and pickup amounts, health and stamina values, and current XP and level. By creating the basic user interface first, we can eliminate the problem of having to constantly check if these values are being influenced or changing via the console output.

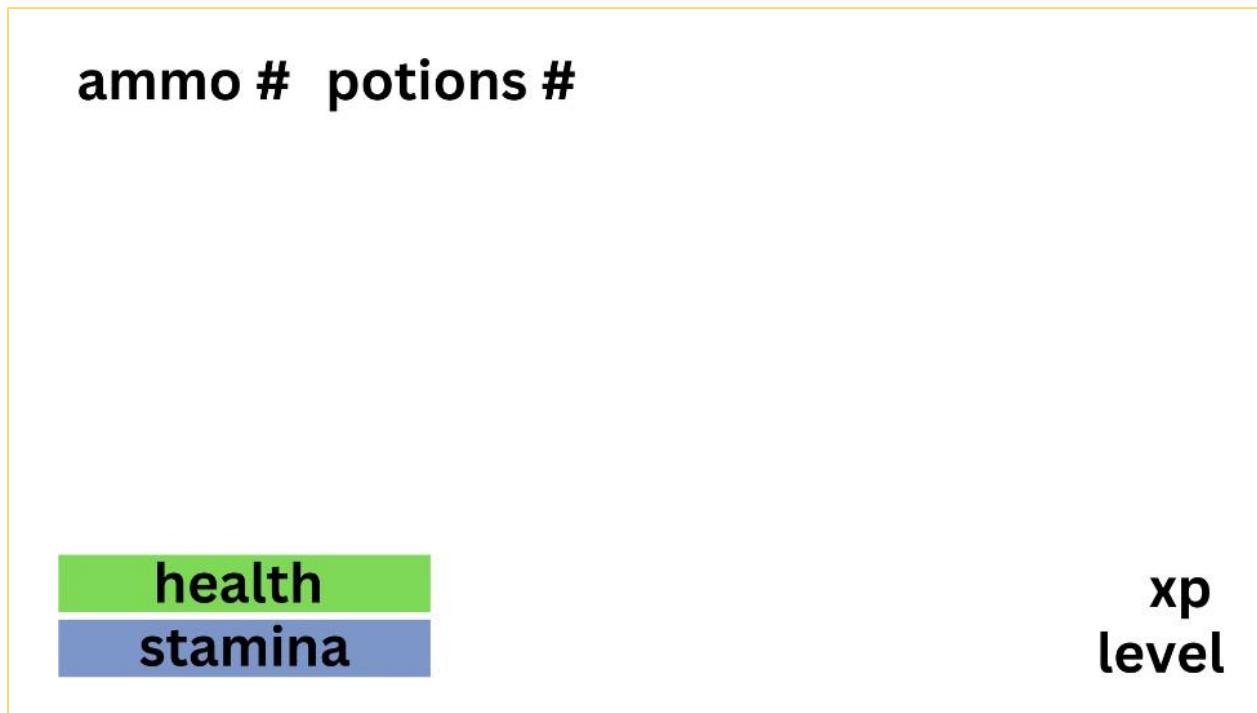


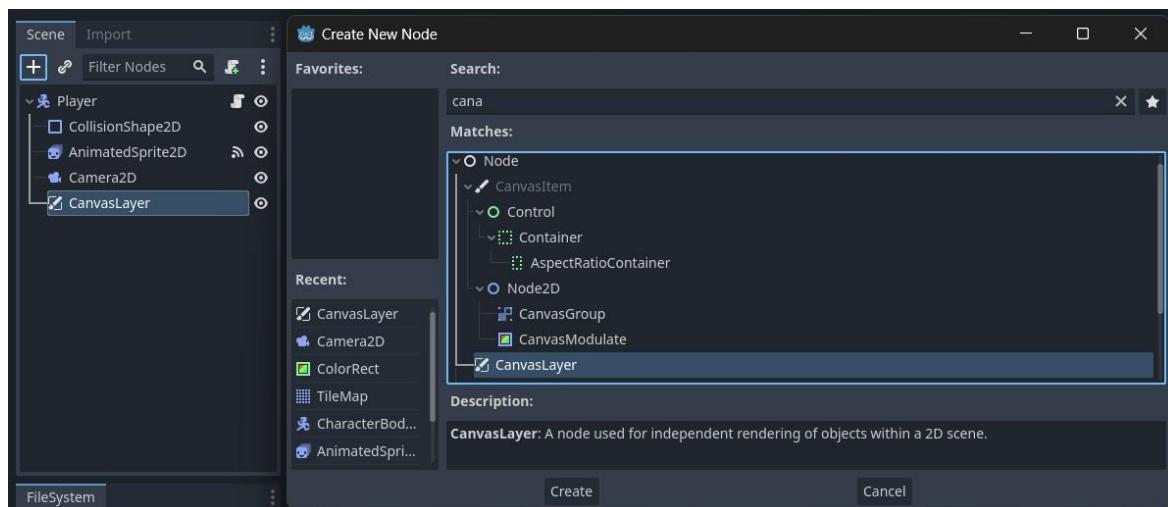
Figure 10: GUI Layout plan.

This part, like the TileMap's part before, might be a bit long, so I'm going to break our GUI creation into three manageable parts: Health & Stamina; Pickups; and XP and Leveling.

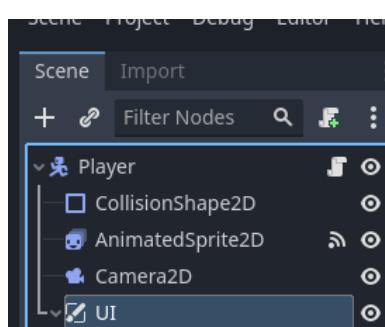
WHAT YOU WILL LEARN IN THIS PART:

- How to add UI elements to your scene.
- How to duplicate nodes.
- How to update UI elements via custom-signals.
- How to change the anchoring of nodes.
- How to create, initialize, and connect custom-signals.

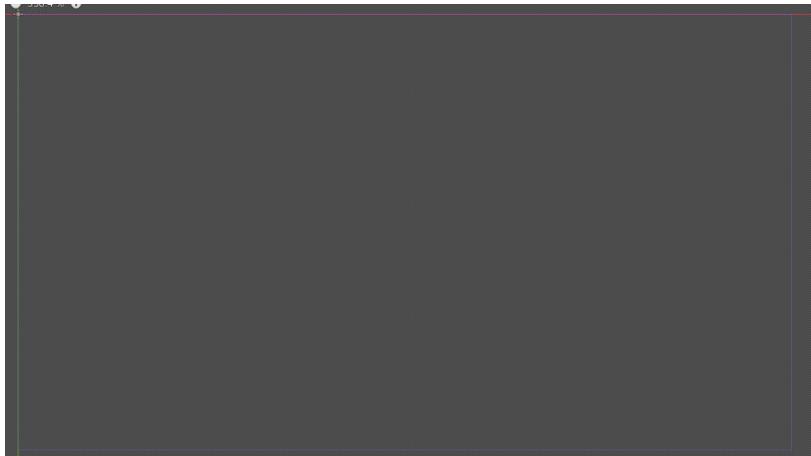
In your Player Scene, add a new [CanvasLayer](#) node. This node will serve as the container for the graphics that we will draw onto the screen, such as our progress bars or labels. We are adding the UI elements to the Main scene because we want to connect these elements from our instantiated Player script via [signals](#) to update their values.



Let's also go ahead and rename it as UI, just so that it stays nice and organized.

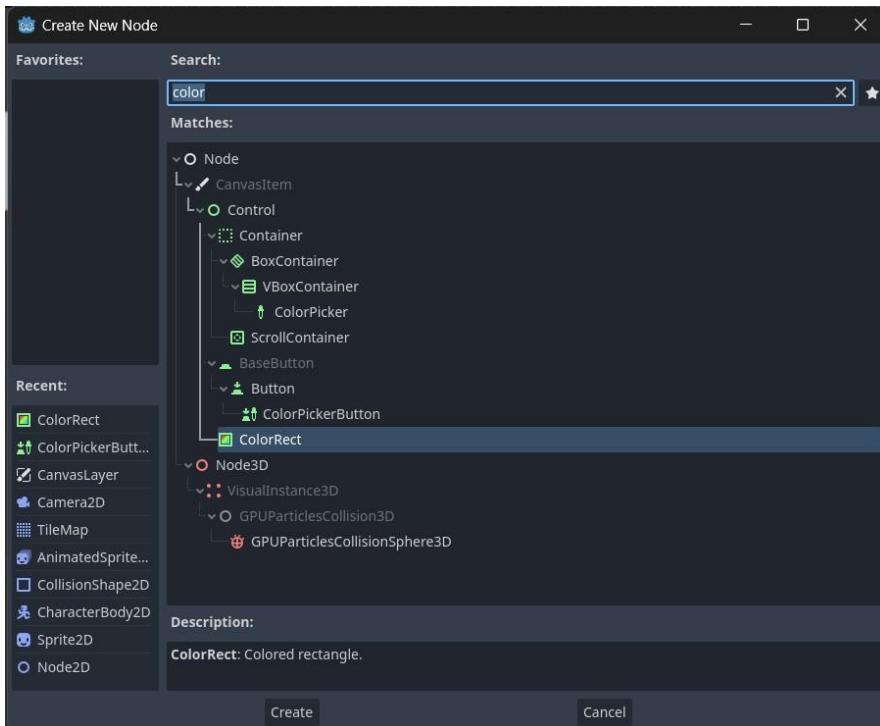


We want the UI elements to be contained within the borders of the blue frame on your screen. I hid my existing nodes so that we can solely focus on the UI for now, so don't panic if you see that I don't have a map or player anymore!

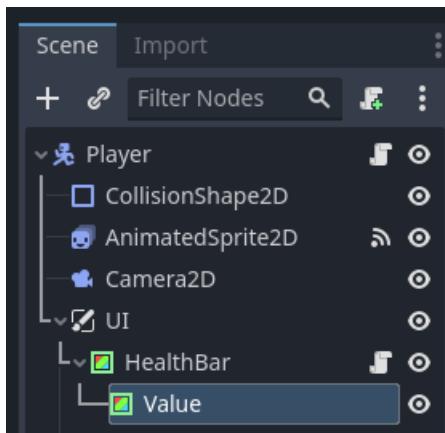


HEALTH & STAMINA BARS

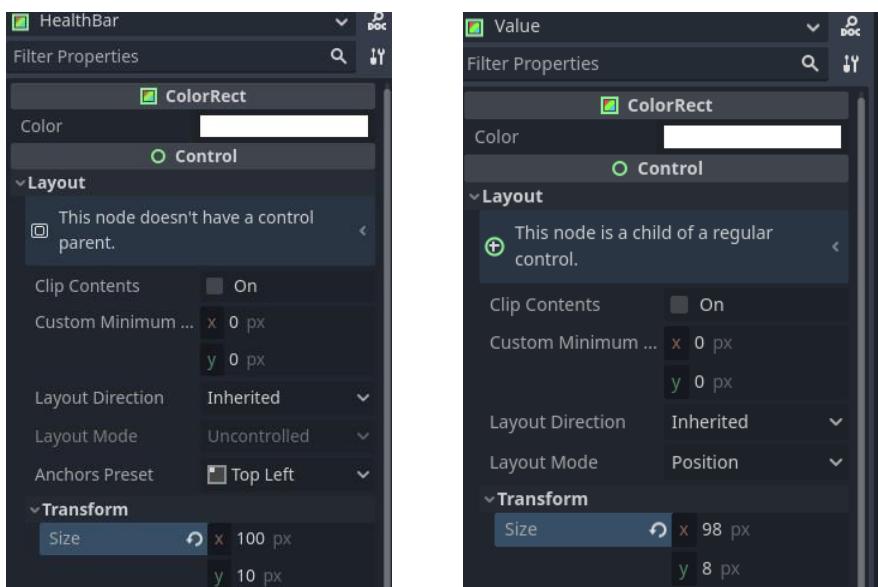
Let's add a [ColorRect](#) node as a child to the UI node. This draws a rectangle that we can fill with color. This node will serve as our progress bar for our health values.



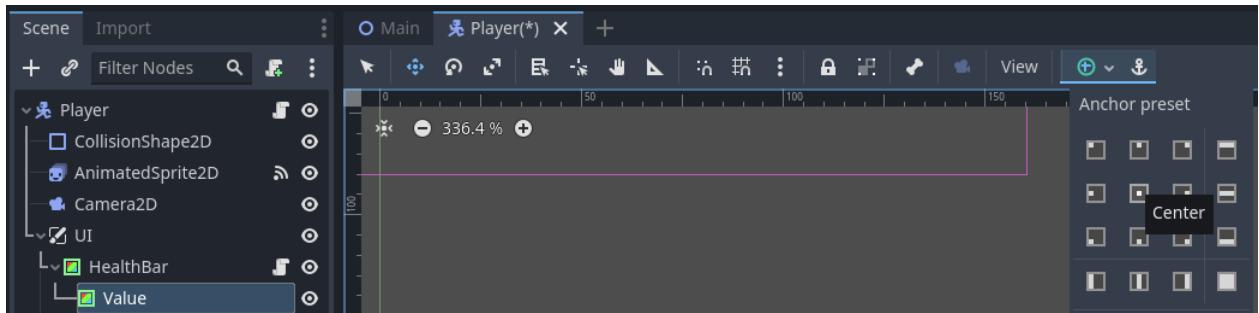
Add another ColorRect node to your newly added node. Rename the first ColorRect as HealthBar, and the second ColorRect as Value. The outer rectangle (HealthBar) will serve as the box or border for our progress bar, and the inner rectangle (Value) will be the actual color that changes to show the progress value.



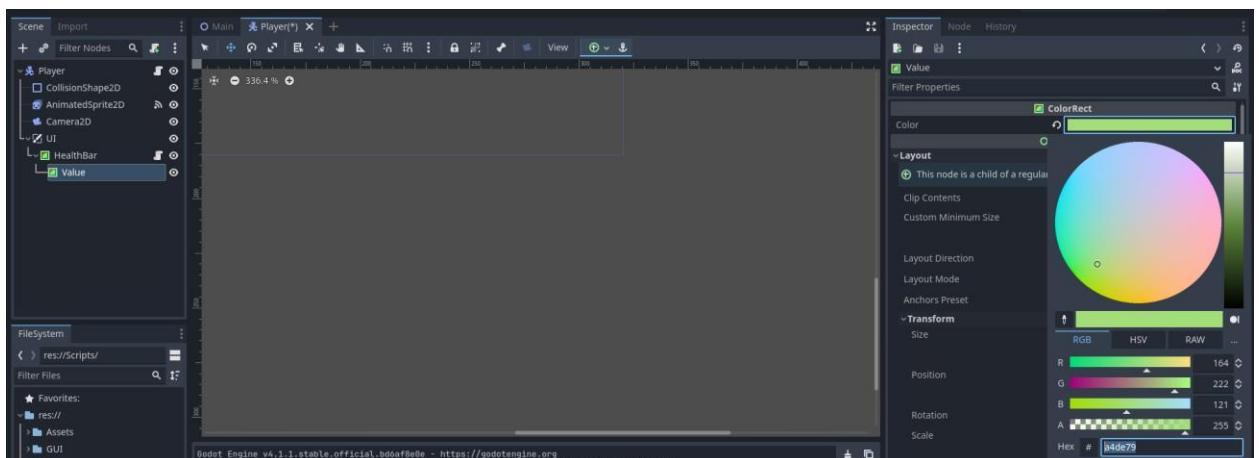
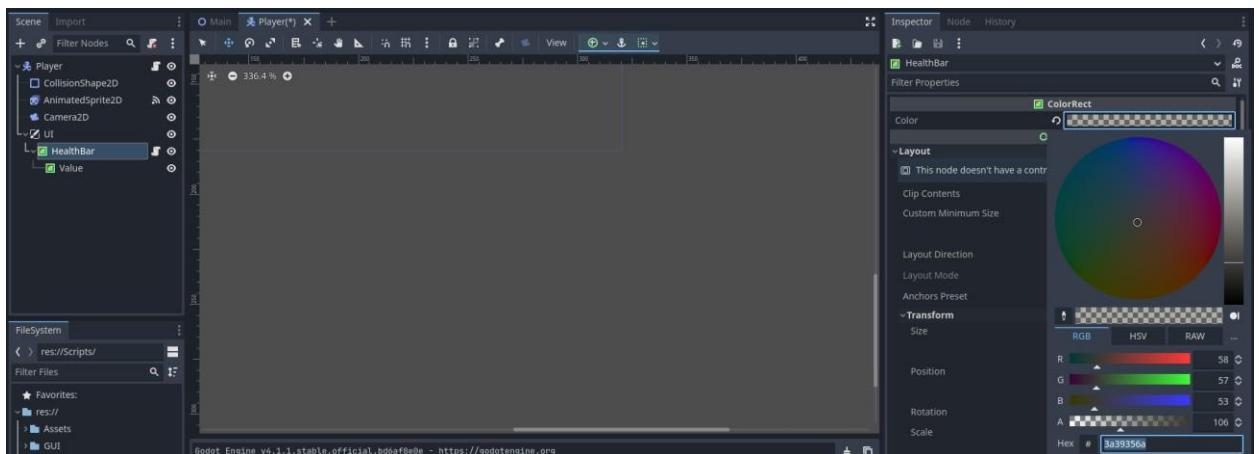
Now, in the Inspector panel, underneath Layout > Transform, change the HealthBar node size to x: 100, and y: 10. Do the same for the Value node, but change its x: 98, and y: 8.



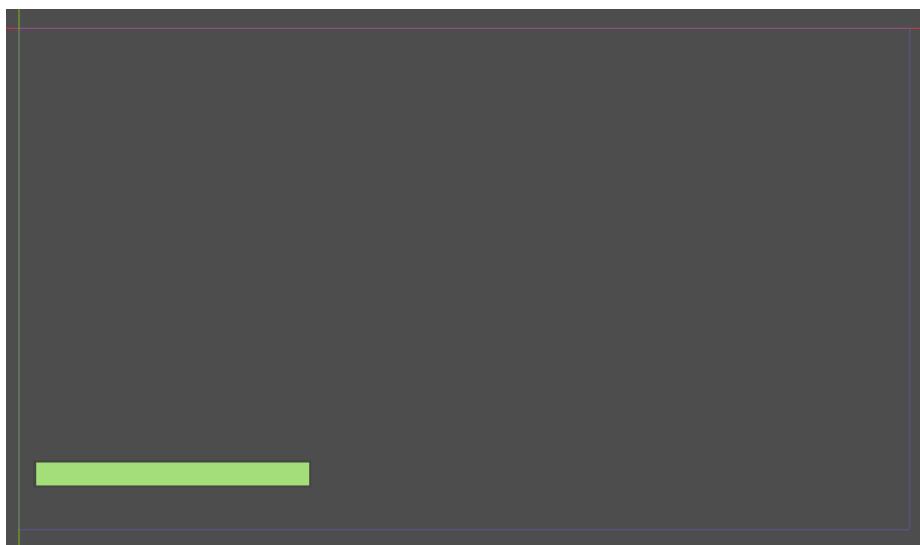
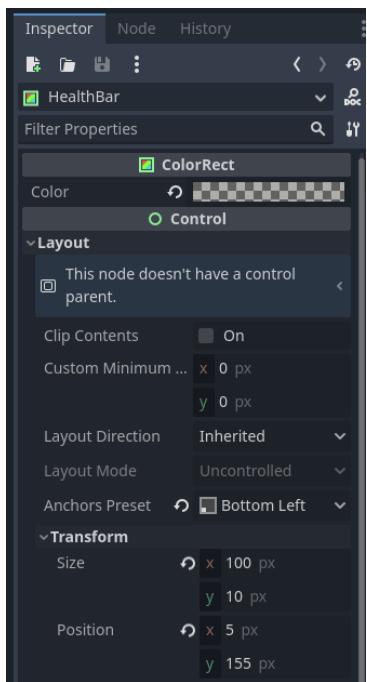
We also want to center our Value rectangle inside of the HealthBar rectangle. To do this, change its anchor preset to center.



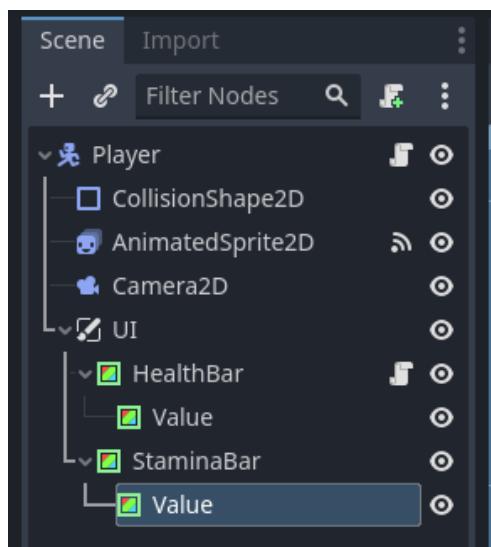
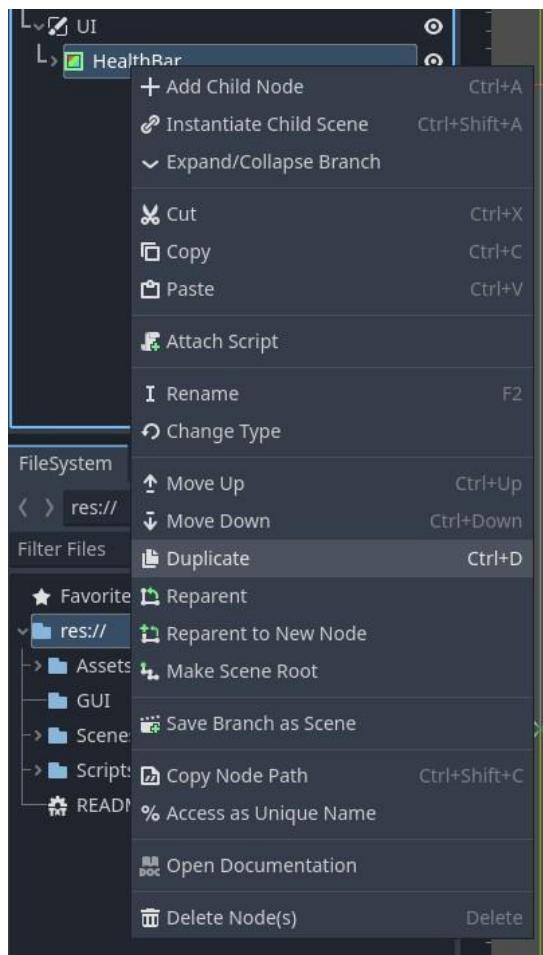
Change the color of the HealthBar to #3a39356a, and the color of Value to #a4de79. In my game, the healthbar will be green, but you can change this color to whatever your heart desires.



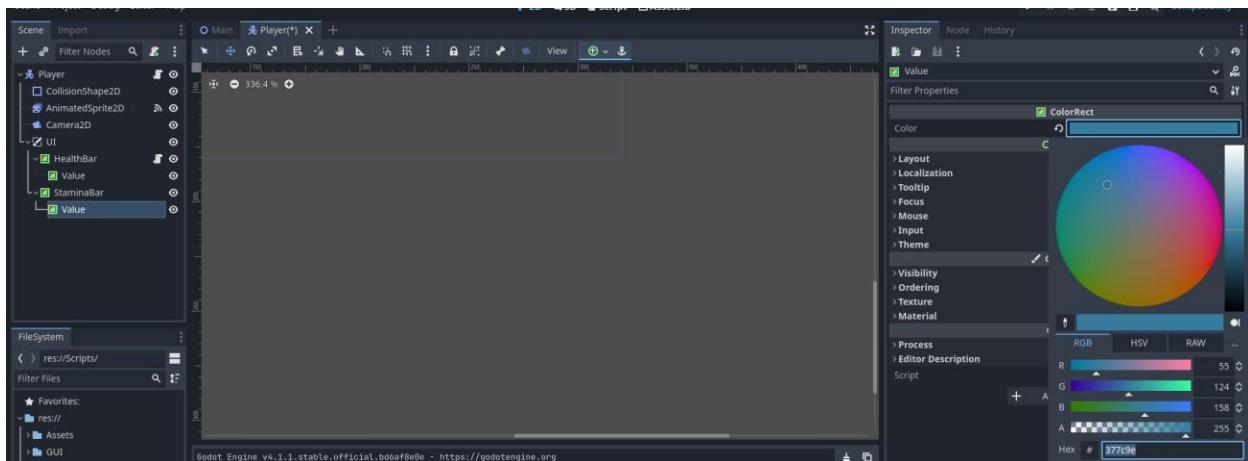
Let's anchor our healthbar to the bottom-left of our screen. In your Inspector panel, underneath Layout, change the anchor preset from center to bottom-left, and its position values to x: 5, y: 155.



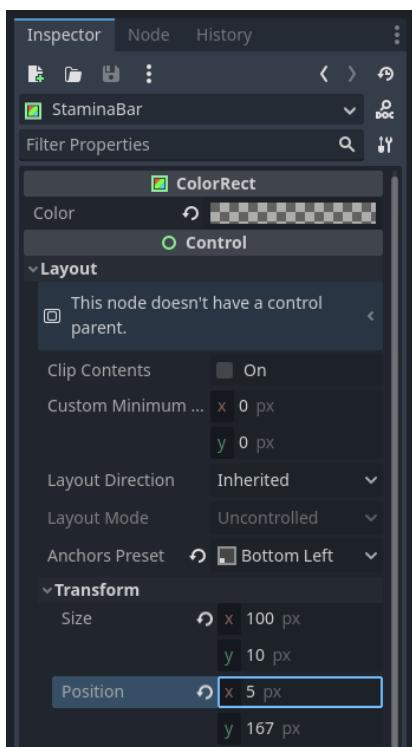
Duplicate the entire HealthBar node (with its child-node Value) and rename it to StaminaBar.



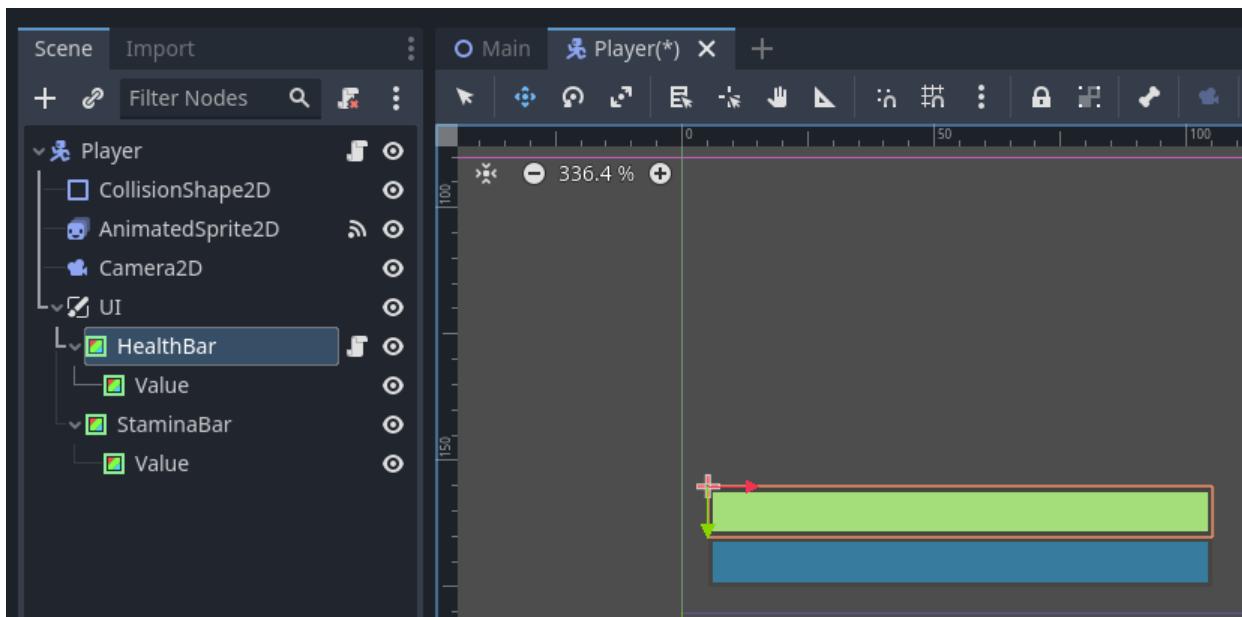
Change the color of the Value node within StaminaBar to #377c9e. This will make it a blue color, but once again, this is your game so make it any color you'd like.



Let's also anchor our stamina bar to the bottom-left of our screen. In your Inspector panel, underneath Layout, change the anchor preset from center to bottom-left, and its position values to x: 5, y: 165.



Now we have our health and stamina progress bars set up! We can now go ahead and implement the functionality for these elements in our code.



In our Player script, we need to add variables to capture our health value, max health, and health regeneration value, as well as the same for our stamina. On top of your Player.gd script, underneath the *is_attacking* code, let's add these variables.

```
### Player.gd

# older code

# UI variables
var health = 100
var max_health = 100
var regen_health = 1
```

With our variables created, we need to create custom signals for both health and stamina so that other objects in our game can listen and react to change events, without directly referencing one another. In other words, these signals will notify our game that a

change in our health or stamina values has occurred, and then trigger other events to occur because of this.

If our health and stamina variable changes, we want to fire off the signal to it so that visually these changes are shown in our progress bar and our health value is updated. We worked with signals before when we connected the built-in `on_animation_finished()` signal to our player script, but this time, we will create our own [custom signal](#).

Why use custom signals?

While built-in signals cover a lot of common use cases (like button clicks or mouse events), they may not handle all the specific interactions or events that are unique to your game or application. Custom signals provide a way to define your own set of events that are specific to your game's logic, making your codebase more organized, reusable, and maintainable.

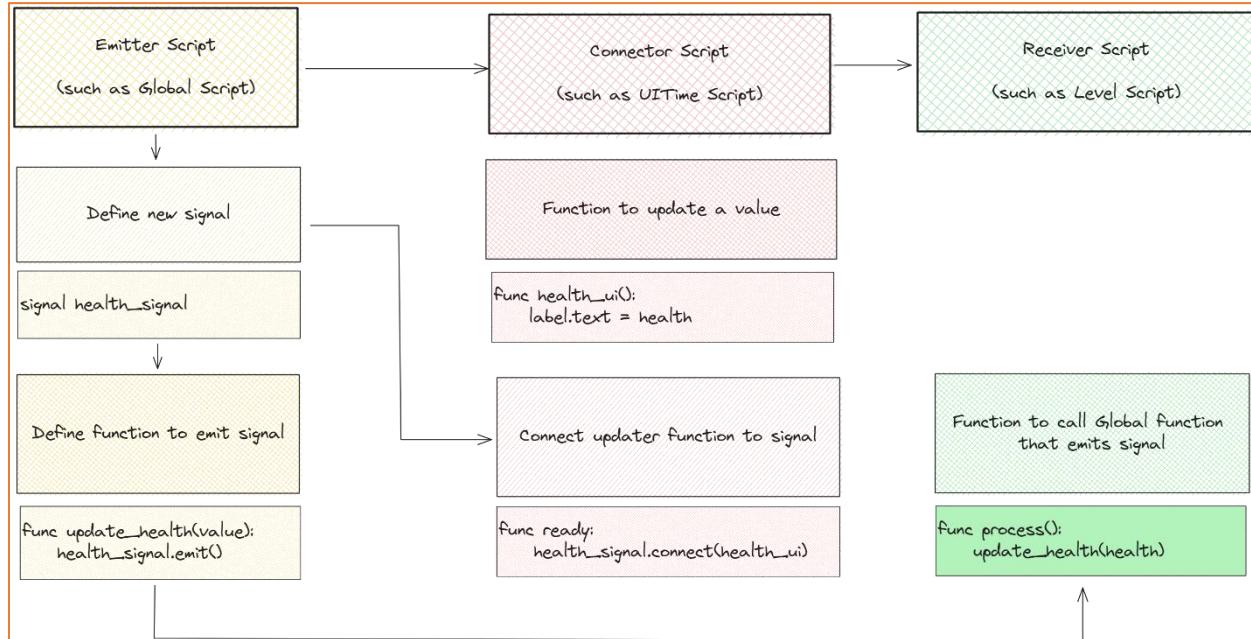


Figure 11: Overview of custom signal flow

Now, before we create our custom signal, let's try and understand when we want these signals to emit and notify our game that our health and stamina values have changed. We want the signals to emit when we press the sprint input, and when we get damaged from a bullet or drink a health potion later on.

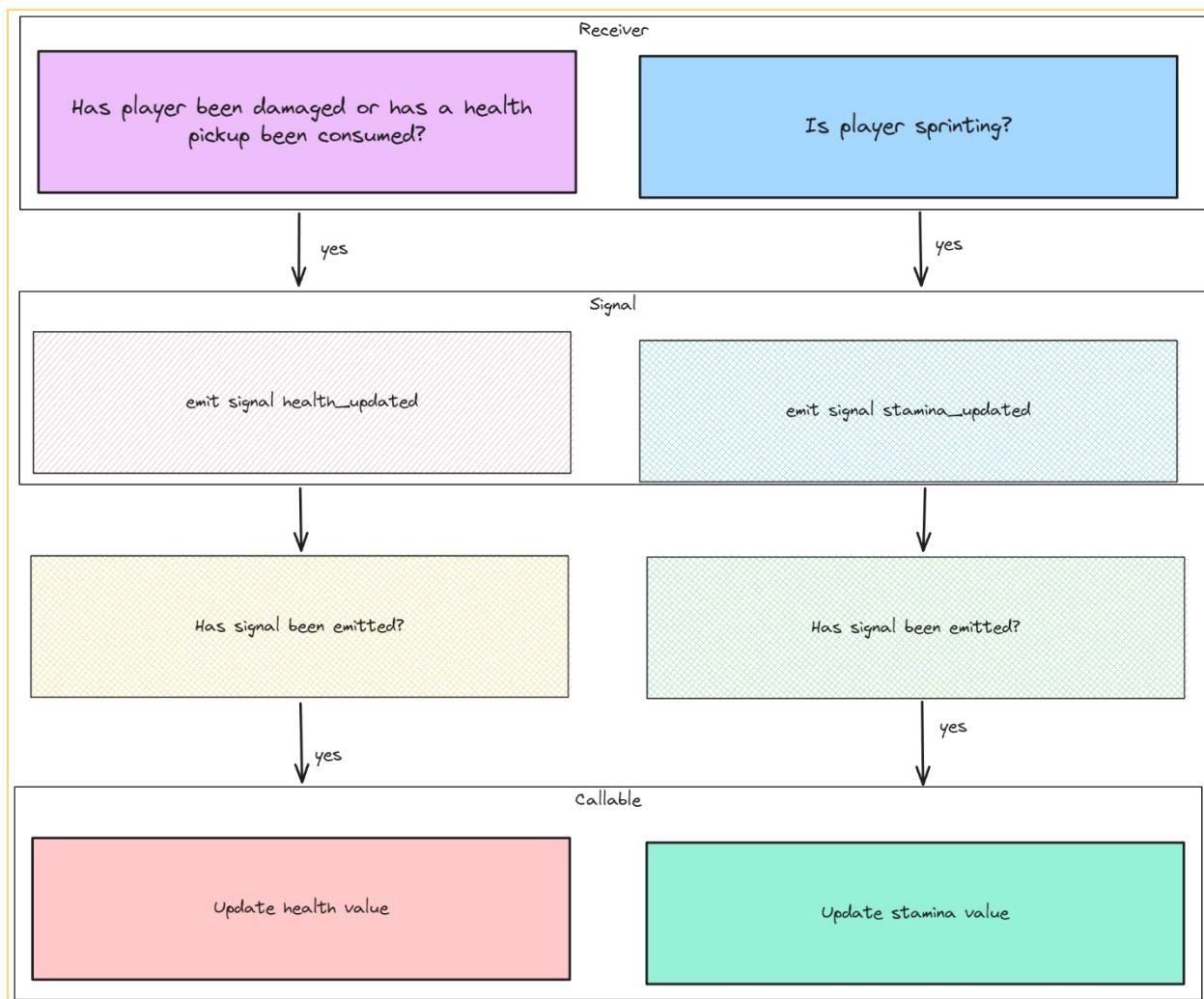


Figure 12: Custom signal overview

We will add our custom signals to the top of our code because we will add more signals later on. This will keep all of our signals neat and organized. To define custom signals, we use the keyword `signal` followed by the name of the signal. Our signals will update our health and stamina values, so we will call them `health_updated` and `stamina_updated`.

```
### Player.gd

# older code

# Custom signals
signal health_updated
signal stamina_updated
```

Now that we have our signals and variables created, we also need to constantly check them to see if our health or stamina values need to be updated or regenerated. We can do this check in the [_process\(\)](#) function, which will be called every time a frame is drawn (60 times a second). The function has a parameter called *delta*, which represents the elapsed time that has passed since the previous frame.

For each health and stamina, we need to calculate the updated values for each. We will do this by using the [min\(\)](#) method, which will ensure that it will never exceed the maximum values of our `max_health` and `max_stamina` variables. Inside this `min()` function, we will calculate the updated value of our health and stamina variables and add it to the value of these variables that are captured in the current frame.

```
### Player.gd

# older code

# ----- UI -----
func _process(delta):
    #calculate health
    var updated_health = min(health + regen_health * delta, max_health)
    #calculate stamina
    var updated_stamina = min(stamina + regen_stamina * delta, max_stamina)
```

If these values are different than our original values (`var health` and `stamina`), we will then update our health to the new value and emit our signal to notify the game of this change.

```
### Player.gd
```

```
# older code

# ----- UI -----
func _process(delta):
    #regenerates health
    var updated_health = min(health + regen_health * delta, max_health)
    if updated_health != health:
        health = updated_health
        health_updated.emit(health, max_health)
    #regenerates stamina
    var updated_stamina = min(stamina + regen_stamina * delta, max_stamina)
    if updated_stamina != stamina:
        stamina = updated_stamina
        stamina_updated.emit(stamina, max_stamina)
```

While we're at it, let's update our sprinting input action to use up some of our stamina when pressed. You'll notice that we emit the signal via the `.emit()` method. This emits this signal, and all the `callables` connected to this signal will be triggered. We'll create functions in our UI script which will be our callables, and so if the signal is emitted, this callable will be notified to update our UI components.

```
### Player.gd

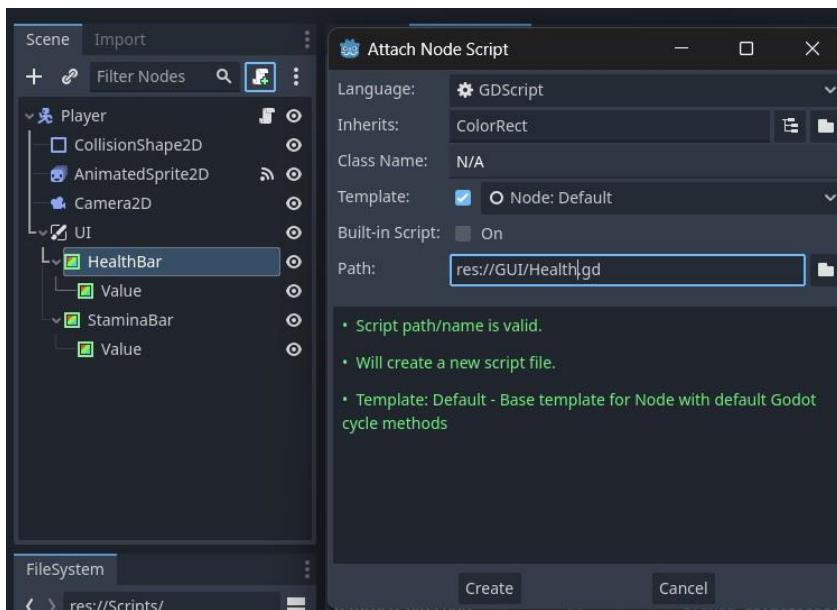
# older code

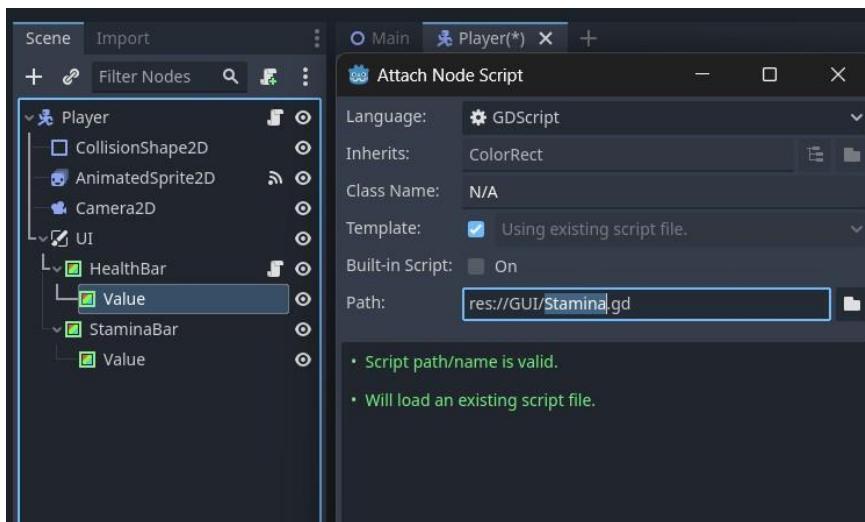
# ----- Movement & Animations -----
func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
    Input.get_action_strength("ui_left")
    direction.y = Input.get_action_strength("ui_down") -
    Input.get_action_strength("ui_up")
    # Normalize movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()
    # Sprinting
    if Input.is_action_pressed("ui_sprint"):
        if stamina >= 25:
            speed = 100
            stamina = stamina - 5
            stamina_updated.emit(stamina, max_stamina)
```

```
elif Input.is_action_just_released("ui_sprint"):
    speed = 50
# older code
```

You'll notice that we passed our stamina variables as parameters into our signal. Since we'll be using these signals to update our progress bars in our UI, passing both health/stamina and max_health/max_stamina in the signals provides the necessary information for UI components to accurately and flexibly display the player's stamina status. It also ensures consistency and efficiency in the game's code.

Now we need to connect our signal to our UI components (callables), so in your Player scene add a new script to both HealthBar and StaminaBar and save these scripts under your GUI folder. Call the one Health and the other one Stamina.





In your newly created Health.gd script, let's create an `@onready` variable for the value of our HealthBar/Value node.

```
### Health.gd

extends ColorRect

# Node refs
@onready var value = $Value
```

We then need to create a function that will update the color value of the Value node. We can do this by multiplying its width (98) by the value of the player's health divided by max_health. This will return a percentage value that will reflect our Value node's new width.

```
### Health.gd

extends ColorRect

# Node refs
@onready var value = $Value

# Updates UI
func update_health_ui(health, max_health):
    value.size.x = 98 * health / max_health
```

Do the same for your stamina value in Stamina.gd.

```

### Stamina.gd

extends ColorRect

# Node refs
@onready var value = $Value

# Updates UI
func update_stamina_ui(stamina, max_stamina):
    value.size.x = 98 * stamina / max_stamina

```

Now we can connect the functions from our UI components to our signals in our built-in [_ready\(\)](#) function. This function will connect our callables to our signal when the Player node enters the Main scene – thus the UI will be able to update the progress bars upon game load.

When to use _ready()?

We use the `_ready()` function whenever we need to set or initialize code that needs to run right after a node and its children are fully added to the scene. This function will only execute once before any `_process()` or `_physics_process()` functions.

We will connect our Player's `health_updated` signal via the [connect](#) keyword, and the callable that it will connect to is our `HealthBar`'s `update_health_ui` function. This means each time there is a change in our health value, the player script will emit the signal, and our `healthbar` will update its value. We'll create a node reference to our `HealthBar` and `StaminaBar` nodes so that we can access the functions from their attached scripts.

```

### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar

# older code

```

```
func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
```

If you run your scene now and you sprint, you will see that the stamina bar decreases, as well as regenerates!



Let's move on to our next GUI part for our Pickups UI. Remember to save and to make a backup of your project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 6: SETTING UP THE GAME GUI #2

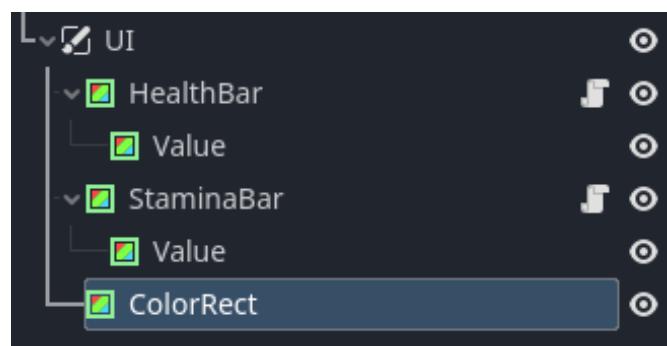
Previously, we created the UI for our player's health and stamina, as well as implemented the functionality to update these values both in the code and visually. In this part, we will create the user interface to show and update our Pickups. Pickups are the things that our player will be able to, well, pick up! This includes health and stamina regeneration drinks, and ammo. We don't have these items yet, and we'll create them later on, but for now we will create the UI for them.

WHAT YOU WILL LEARN IN THIS PART:

- How to add UI elements to your scene.
- How to add labels and icons.
- How to change a node's Color properties.
- How to change the fonts and anchoring of nodes.

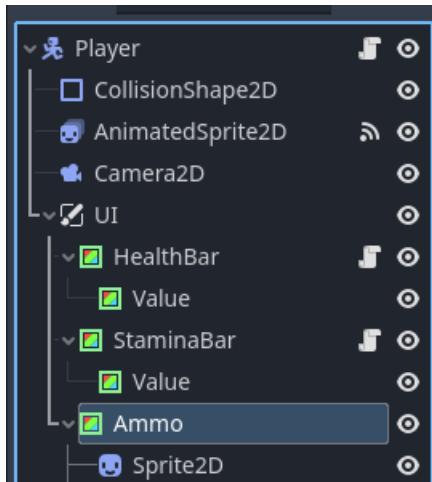
PICKUP AMOUNTS UI

In your Player scene, underneath your UI node, add a new ColorRect node.

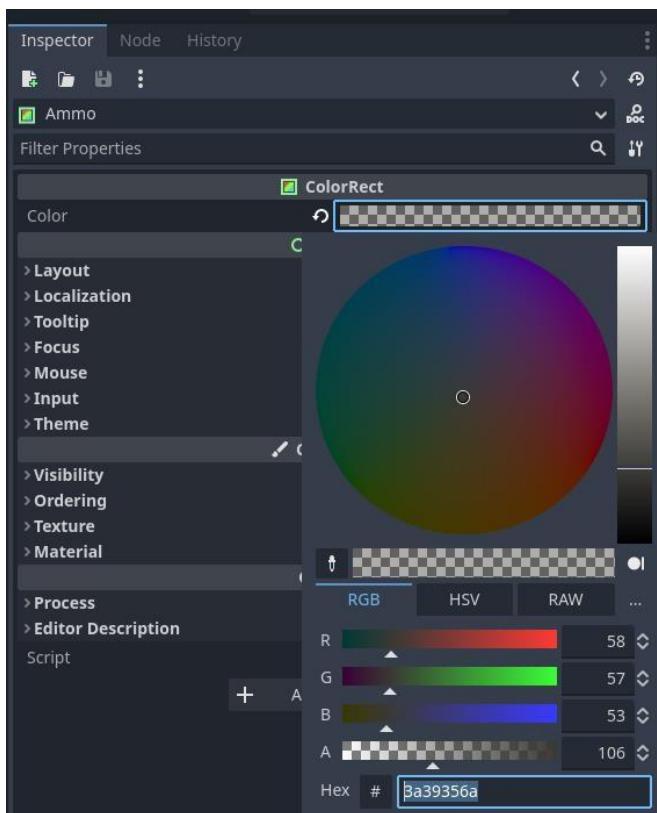


Rename this node to `AmmoAmount` and add a [Sprite2D](#) node to it as its child. This `Sprite2D` node will be our ammo's icon. An `AnimatedSprite2D` node can contain multiple sprite frames and thus we can organize these frames to play an animation, whereas

a [Sprite2D](#) node is a static object that contains one singular sprite frame. Add another child to it as type [Label](#). This will show the ammo amount that our player has.



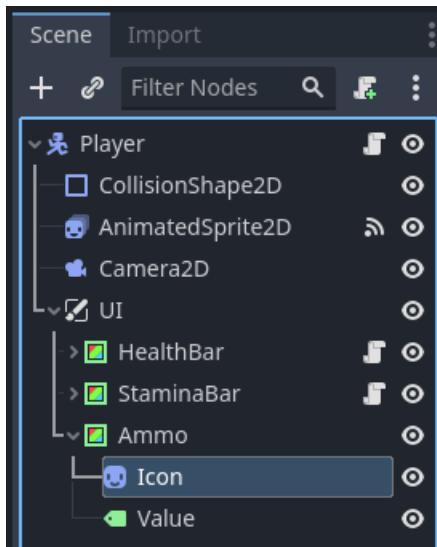
Select the Ammo node, and underneath Layout, change its color value to #3a39356a.



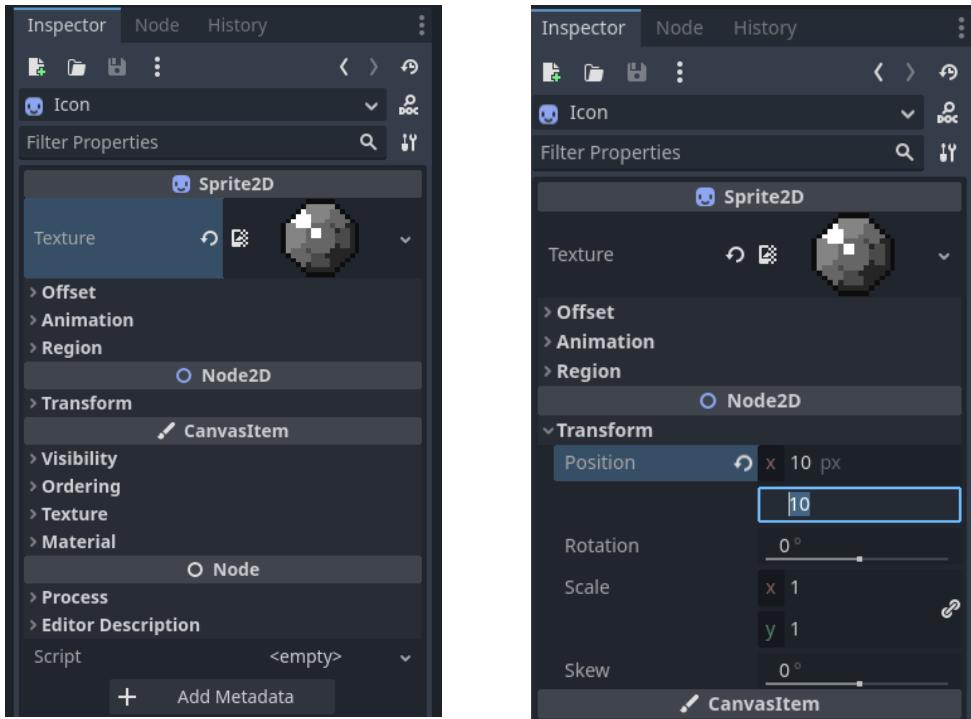
Change its size to x: 40, y: 20 - and its position to x: 5, y: 8.



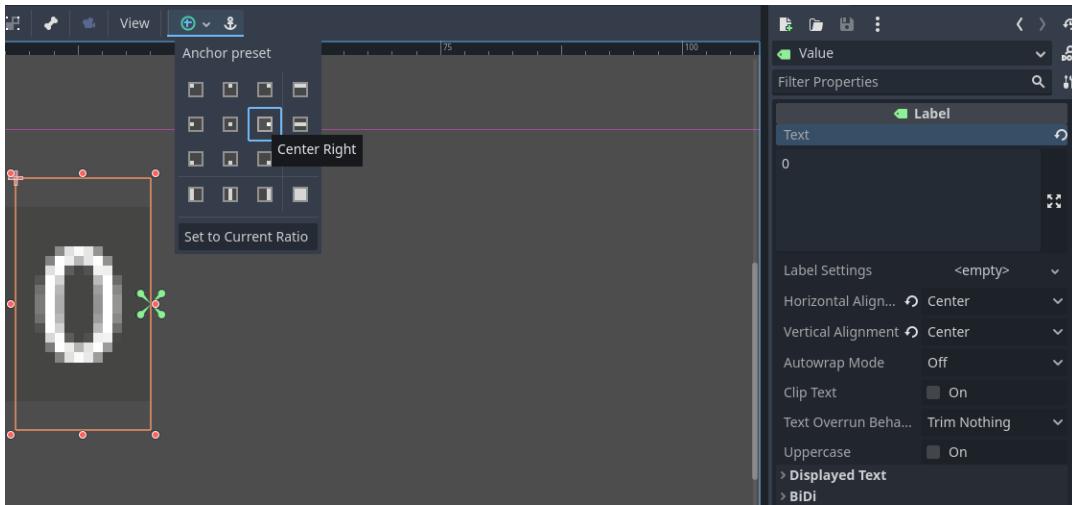
Now select the Sprite2D node, and let's rename it to Icon. Whilst we're at it, rename the label to Value.

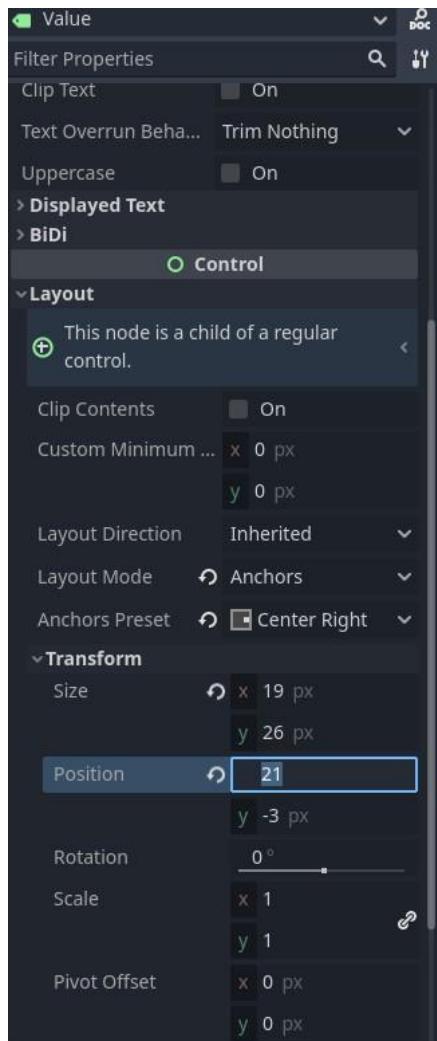


Back to the Icon node, let's assign it a texture. I don't have an icon to represent bullets, so I'm going to make do with an icon of a shard! In our Assets > Icons folder, drag the image called "shard_01i.png" into the texture property. Then go ahead and change its position to x: 10, y: 10.

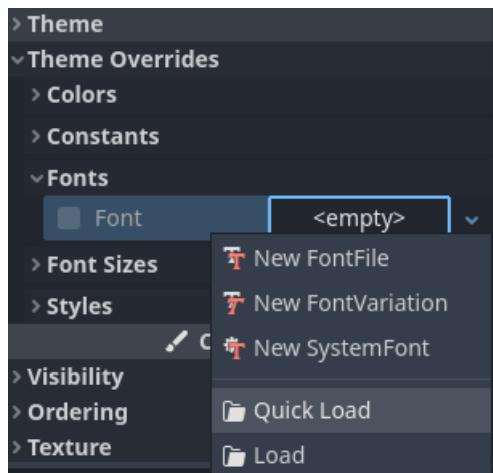


With your Value node, change its text to "0". Also, change its horizontal and vertical alignment to be centered, and anchor it to the center-right side of the box. Its size should be x: 19, y: 26 - and its position should be x: 21, y: -3.

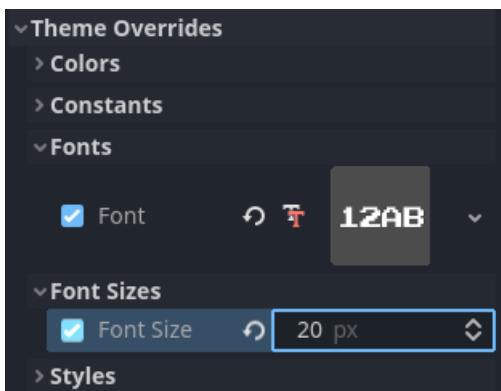
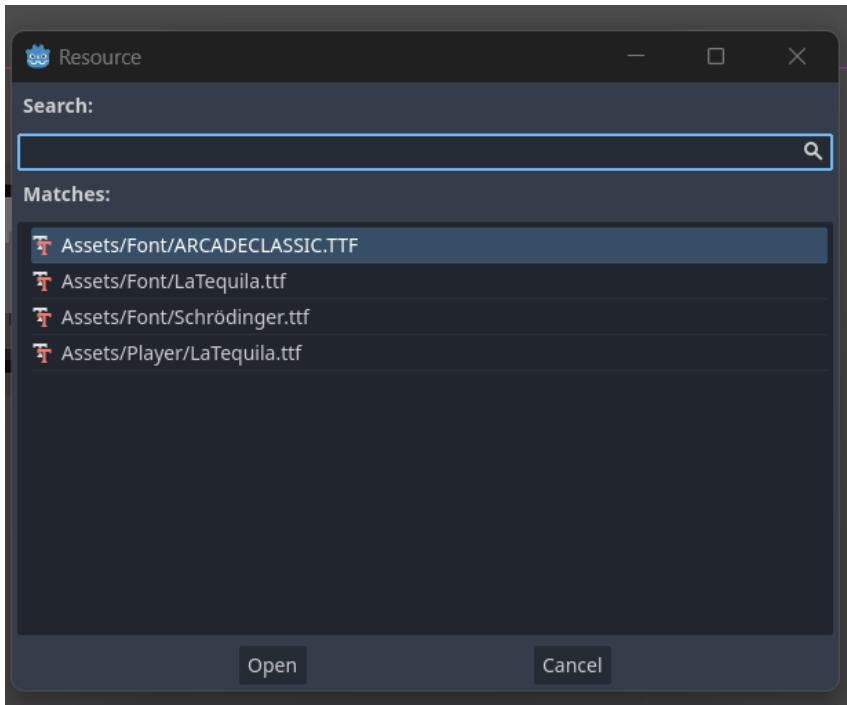




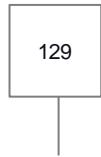
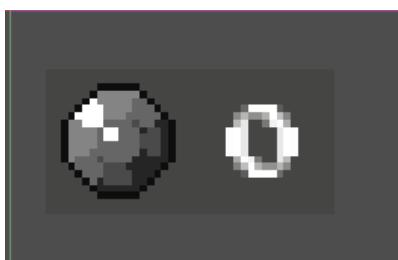
We also want to use a custom font for this text. Underneath Theme > Theme Overrides> Fonts, select the option "Quick Load".



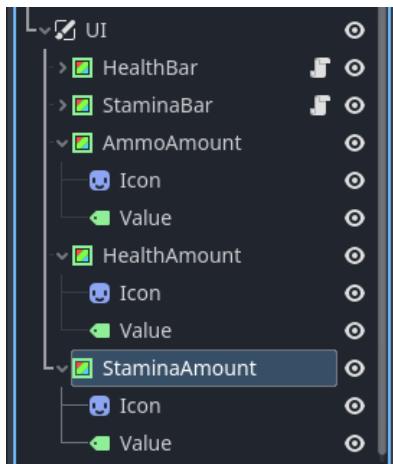
I have a few fonts added to this asset pack, but we're going to use ARCADECLASSIC. Also, change its font size to 20 (or 15).



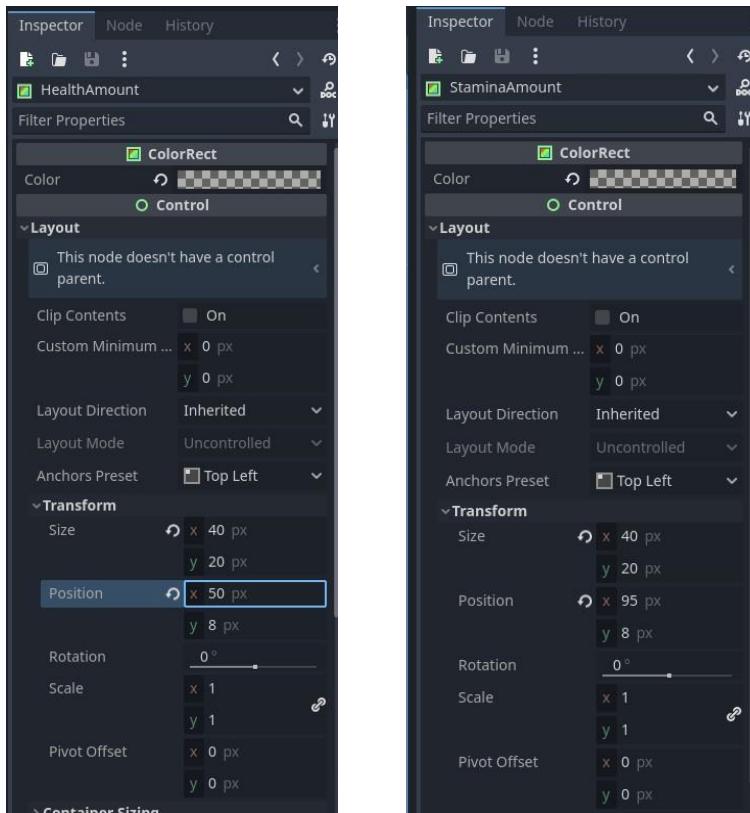
Your AmmoAmount element should now look like this:



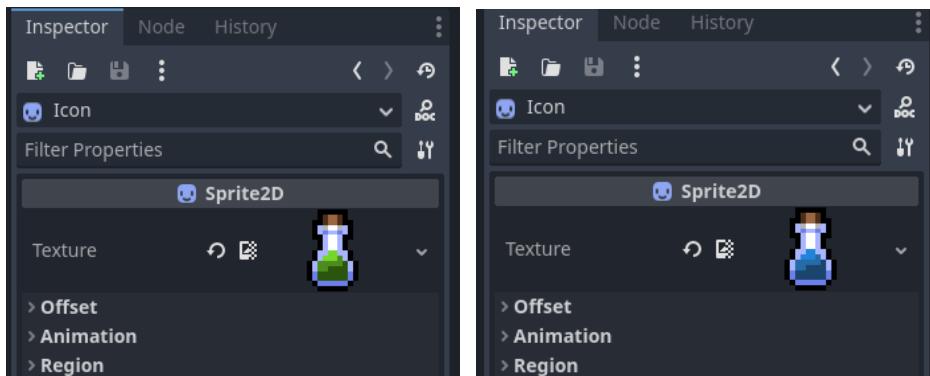
Duplicate this element twice. Rename the first duplicate to "HealthAmount", and the second one to "StaminaAmount". These new elements will show our player's health and stamina drinks (regeneration potions) amount. Please rename the Ammo node to AmmoAmount.



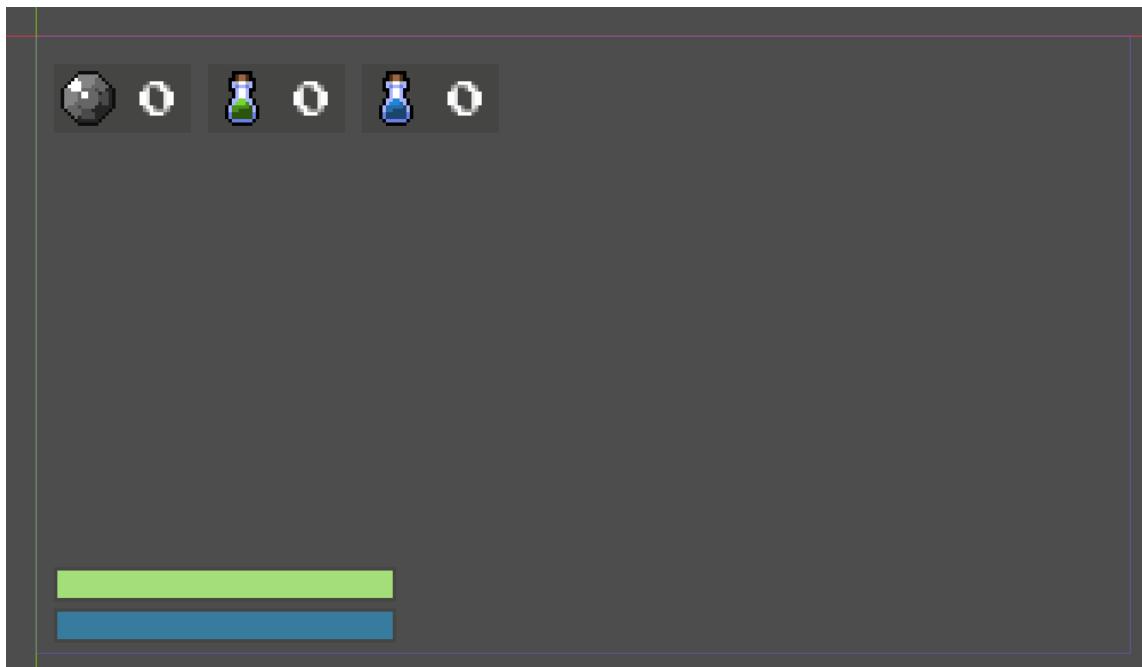
Your HealthAmount node's position property should be x: 50, y: 8. The StaminaAmount node's position property should be x: 96, y: 8.



Change the HealthAmount node's icon to "potion_02c.png", which you can find under Assets > Icons. Then change the StaminaAmount node's icon to "potion_02b.png".



Your UI should now look something like this:



We cannot change our newly created pickup values yet because we do not have pickups. We do have our UI however, so things are starting to come together. Let's move on to our final GUI: Level & XP. Remember to save your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 7: SETTING UP THE GAME GUI #3

In our two previous parts, we created the UI elements for our health and stamina values, as well as our pickups. In the final part of our GUI section, we are going to create the elements needed to display our player's Level and XP values.

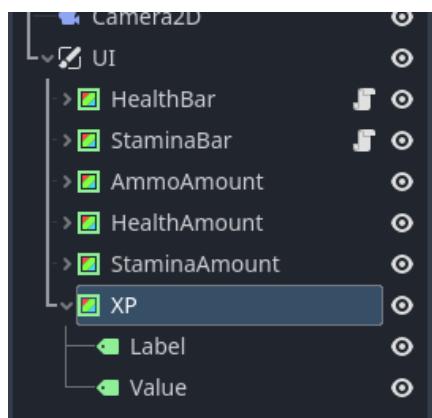
WHAT YOU WILL LEARN IN THIS PART:

- How to add UI elements to your scene more hands-off.
- How to position labels.
- How to change the fonts and anchoring of nodes.

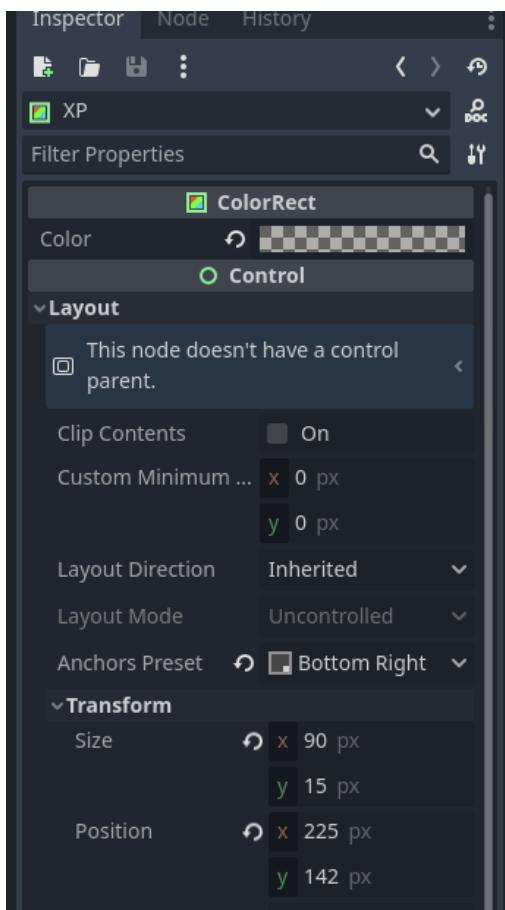
LEVEL & XP UI

We aren't going to add any code to update this yet because we first need to create an enemy for that first. Let's go ahead and create these elements now.

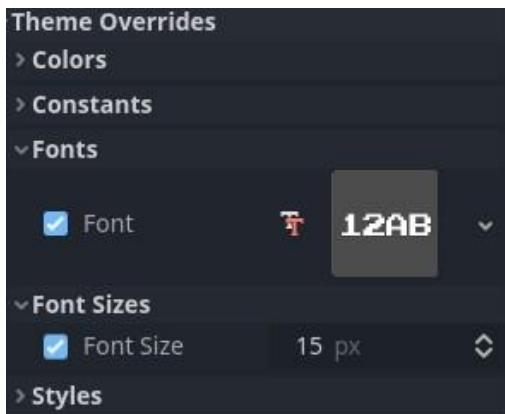
In your Player scene, let's add a new ColorRect to our UI node. Rename this node to XP and add two Label nodes to it. Rename the second Label node to "Value".



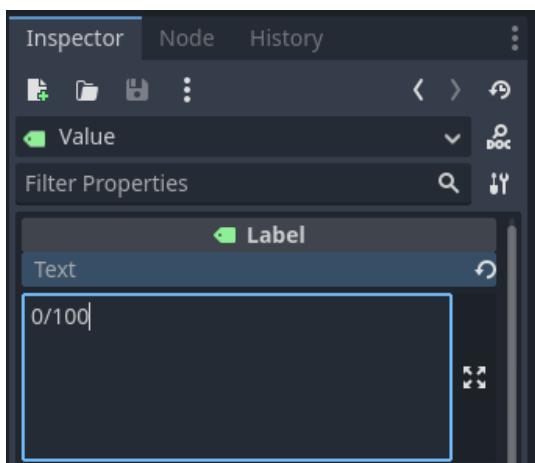
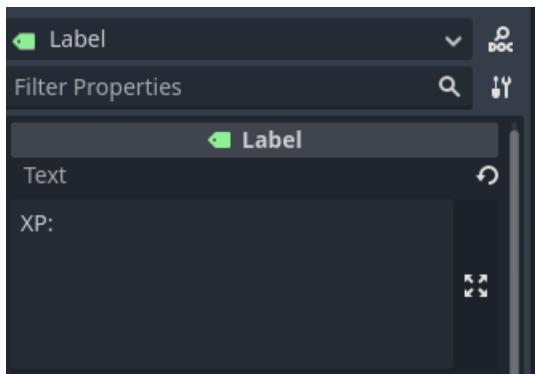
With your XP node selected, change its Color property to #3a39356a, and its transform properties to the following:



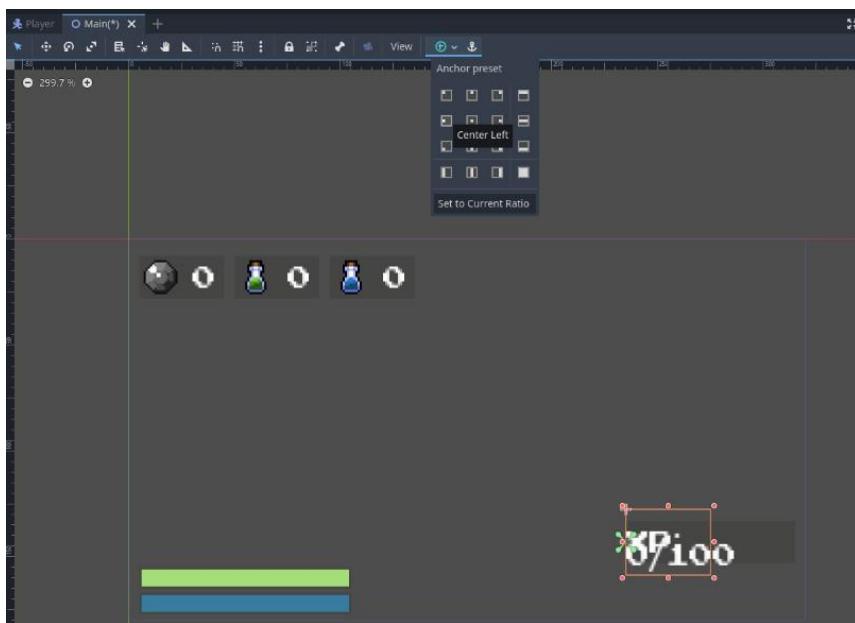
Change the font of your two label nodes to ARCADECLASSIC, and the font size to 15.



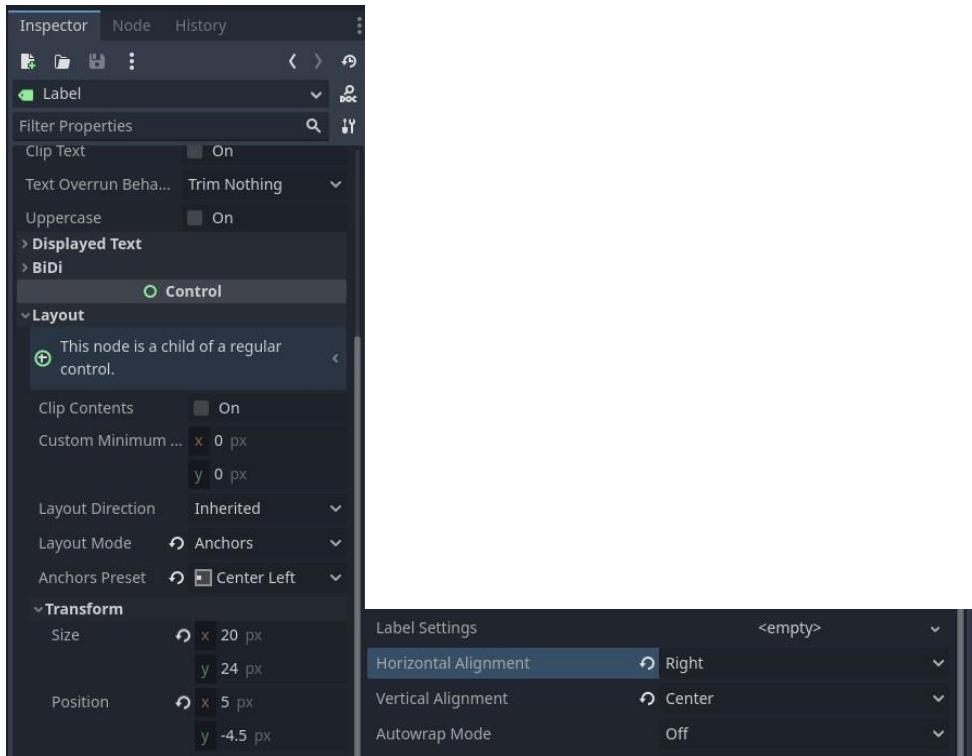
Change the text of your Label to "XP:" and the text of your Value to "0/100".



Then, change the anchor preset of the Label node to center-left.



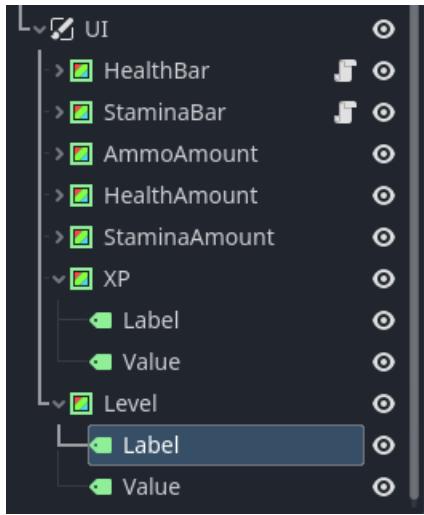
Change its properties to the following:



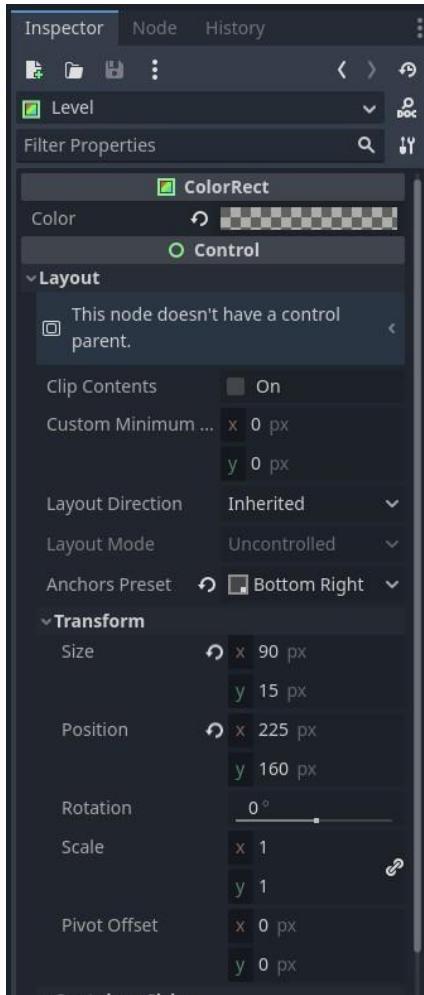
Do the same for the Value node but change it to center-right. Change its properties to the following:



Duplicate your XP node with all of its children and rename it to "Level".



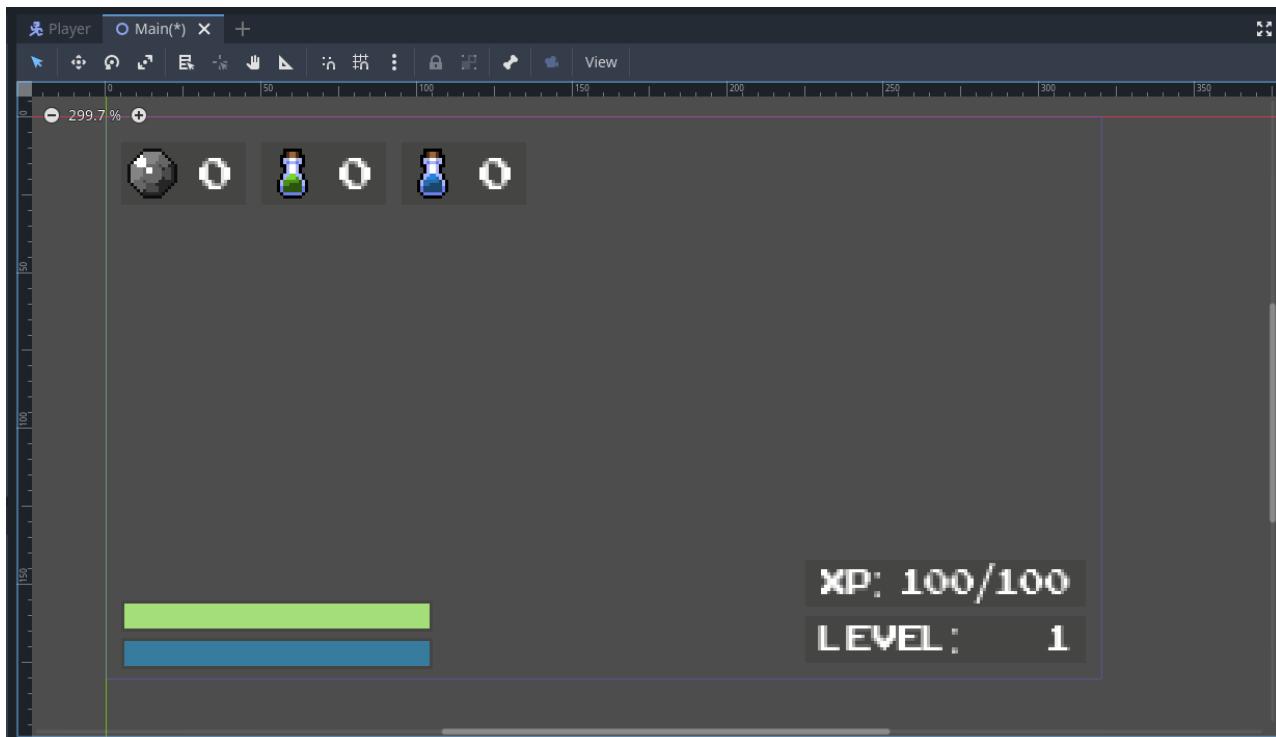
Change the Level node's transform properties to the following:



Finally, change the text of your Label to "Level:" and the text of your Value to "1".



Your result should look like this:



And that's it for this game GUI. I know this was a very long three-parter, but in the next part, we at least will be able to do something a bit more fun when we create our Ammo and Consumable Pickups! Remember to save and backup your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 8: ADDING AMMO PICKUPS AND CONSUMABLES

If you've successfully made it to this part of our tutorial series, then you should have a basic map, movable character, and game GUI set up. In this section, we are going to focus on adding Pickups to our game. These pickups will consist of health and stamina consumables, as well as ammo. We will also be updating our GUI values when our player picks these items up.

WHAT YOU WILL LEARN IN THIS PART:

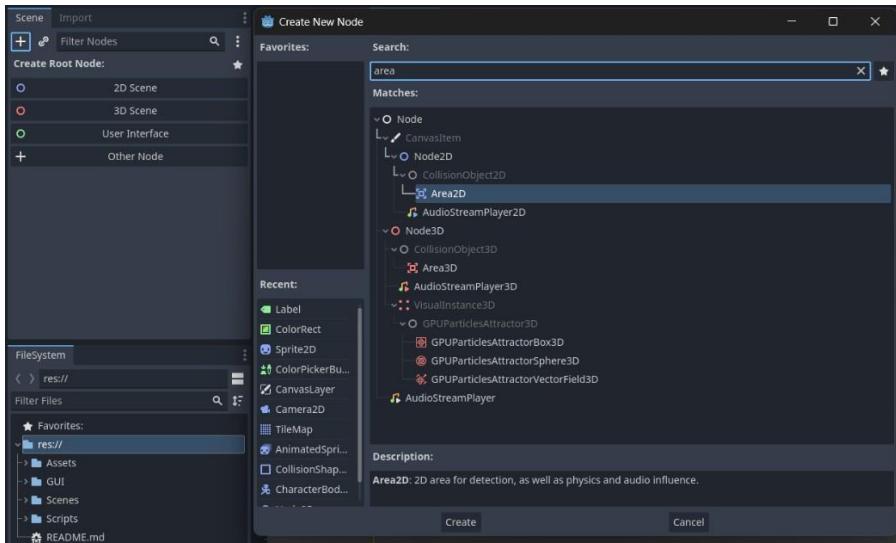
- How to work with Enums.
- How to work with the @tool object.
- How to execute code in the editor.
- How to delete/remove nodes from a scene via the memory queue.
- How to work with Area2D detection bodies.
- How to reference TileMap properties.

PICKUPS SETUP

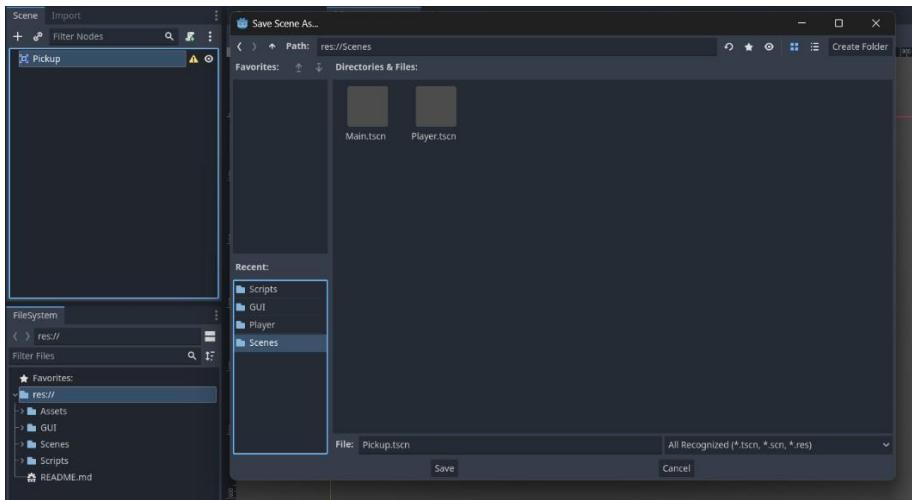
Instead of creating separate pickup items for each of our consumables, i.e., a scene for AmmoPickup, StaminaPickup, and HealthPickup - we are going to create a singular base scene for our Pickups that we can reuse throughout our game. We will do this via [enums](#) and our [export](#) resource.

In your project, create a new scene with an [Area2D](#) node as its root. An Area2D node serves as an area that will detect collisions, so in other words, we will use this area to detect if our Player scene has entered its body via collisions. If the player has run over

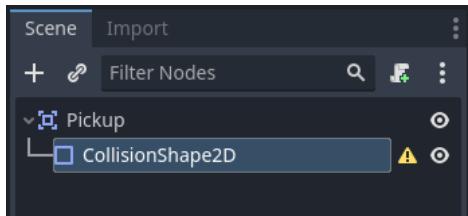
the Pickup scene's collision body, the pickup should be removed and we should update our UI values.



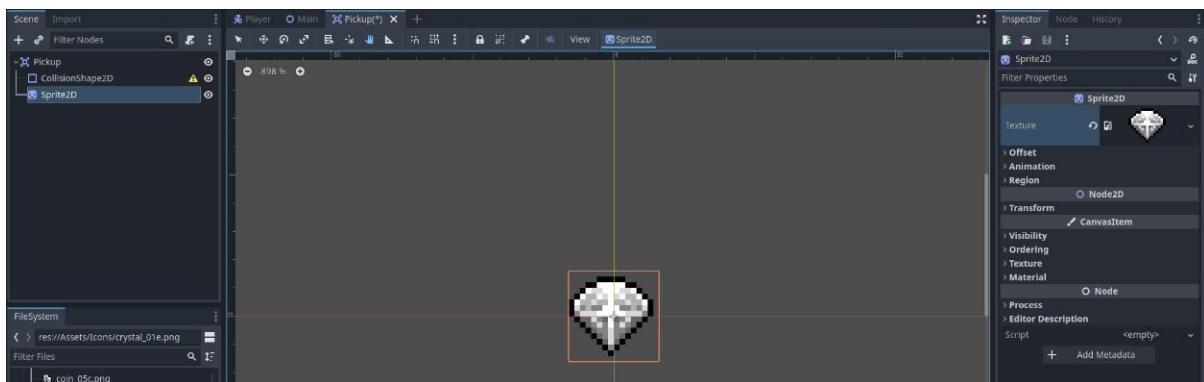
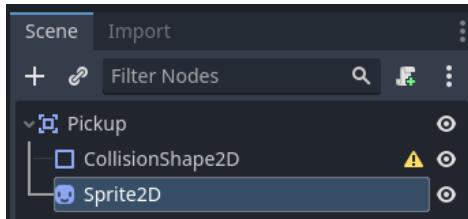
Rename this Area2D node to "Pickup" and save the scene as the same underneath the Scenes folder.



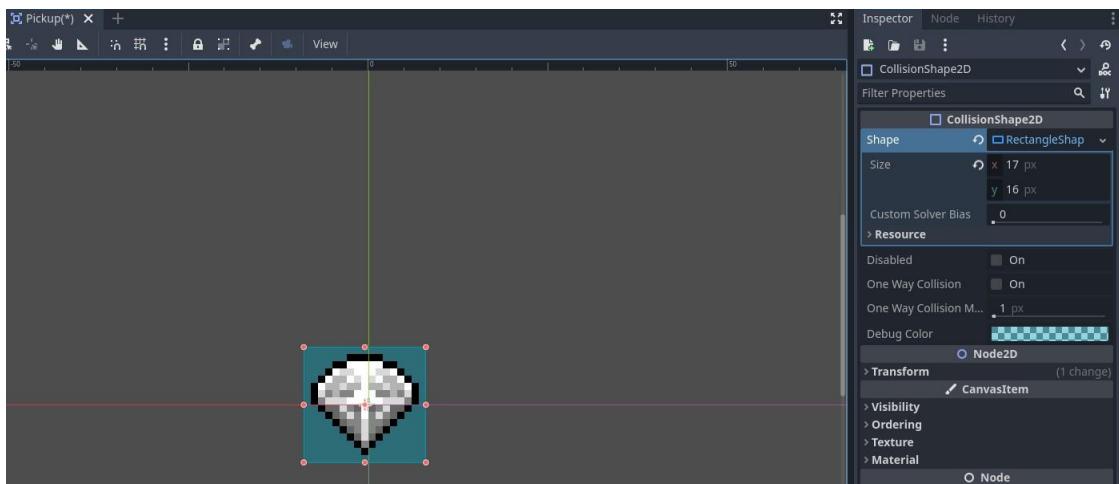
You will notice that the node has a warning next to it because just like our CharacterBody2D node it requires a shape or collision. Let's go ahead and add a CollisionShape2D node to it to resolve this warning.



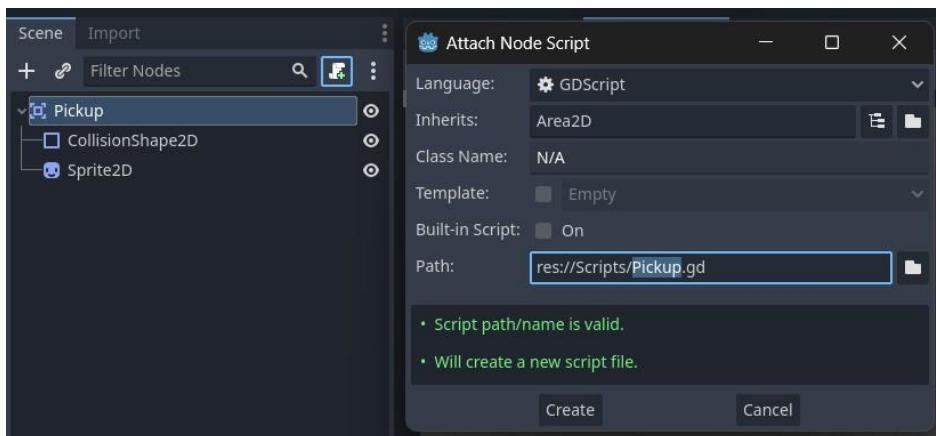
Before we select the shape for the collision, let's give it a Sprite so that we can see what image it is before we give it a collision's bounds. Add a Sprite2D node and assign any icon to it from your Assets/Icons folder.



Now, let's assign a shape to our CollisionShape2D node. Make it a new RectangleShape2D, and then drag the bounds of the collision shape to frame your icon.



This Pickup scene will serve as the base scene for all of our pickups. From here we want to be able to dynamically choose the pickup type and add it to our player's inventory whenever they run over it to pick it up. To do this, we need to attach a script to the root node of our Pickup scene. Save it underneath your Scripts folder.



At the top of our script, let's create an [enum](#) of all of our pickup items (which are our ammo, health, and stamina drinks). We will export this enum so that we can dynamically choose the type of pickup we want to place down from the Inspector panel/editor. Enums are a shorthand for constants and are pretty useful if you want to assign consecutive integers to some constant. To create enums, we use the keyword `enum`, followed by the constants in brackets.

```
### Pickup.gd
```

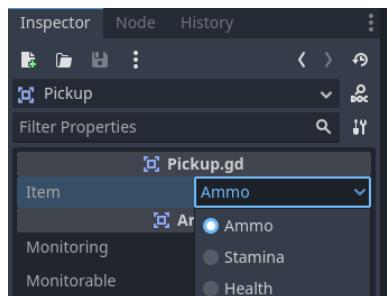
```

extends Area2D

# Pickups to choose from
enum Pickups { AMMO, STAMINA, HEALTH }
@export var item : Pickups

```

Now if we look in our Inspector panel, we can choose the Pickup item from the dropdown.



Next, we want to also choose its icon or texture, because not all of our pickup items can look like diamonds! To do this we will make use of something new so that we can see real-time changes to these textures in our editor. We will use [@tool](#), which is a powerful line of code that, when added at the top of your script, makes it execute in the editor. You can also decide which parts of the script execute in the editor, which in-game, and which in both.

You need to add [@tool](#) to the top of your script before your node class extends.

```

### Pickup.gd
@tool

extends Area2D

# Pickups to choose from
enum Pickups { AMMO, STAMINA, HEALTH }
@export var item : Pickups

```

Then we need to assign our textures, so for each of our Pickups, we will preload our textures. Preloading is when resources are loaded before a scene is playing/running. We preload resources to speed up loading and reduce freezing since resources are

already loaded. Let's assign the textures that we used for our GUI icons to each of our pickup items.

```
### Pickup.gd
@tool

extends Area2D

# Pickups to choose from
enum Pickups { AMMO, STAMINA, HEALTH }
@export var item : Pickups

# Texture assets/resources
var ammo_texture = preload("res://Assets/Icons/shard_01i.png")
var stamina_texture = preload("res://Assets/Icons/potion_02b.png")
var health_texture = preload("res://Assets/Icons/potion_02c.png")
```

Since we want to constantly check whether or not these sprite frames are changing when we assign new Pickups, we want to add the conditional check to change these textures in our `process()` function. We used this function before when we implemented our custom signals.

We want to execute our conditional in the editor to see our texture changes in the editor without having to run the game. To do this we will use our `@tool` functionality, which requires the [`Engine.is_editor_hint\(\)`](#) function to run the code in the editor. We'll also create a node reference to our Node2D node.

```
### Pickup.gd

# Node refs
@onready var sprite = $Sprite2D

#older code

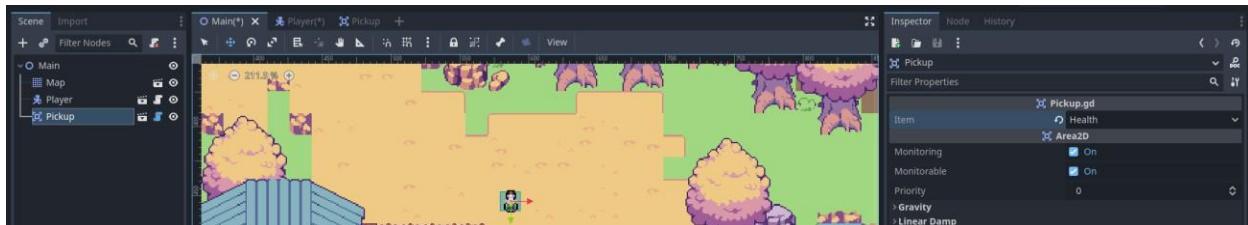
# ----- Icon -----
#allow us to change the icon in the editor
func _process(_delta):
    #executes the code in the editor without running the game
    if Engine.is_editor_hint():
        #if we choose x item from Inspector dropdown, change the texture
        if item == Pickups.AMMO:
```

```

        sprite.set_texture(ammo_texture)
    elif item == Pickups.HEALTH:
        sprite.set_texture(health_texture)
    elif item == Pickups.STAMINA:
        sprite.set_texture(stamina_texture)

```

Now if you instance your Pickup scene in your main scene, and you change your Pickup item in your Inspector panel, your texture should change!



We have to also reflect these settings in the game, so for that, we will set our textures in the *ready()* function.

```

### Pickup.gd

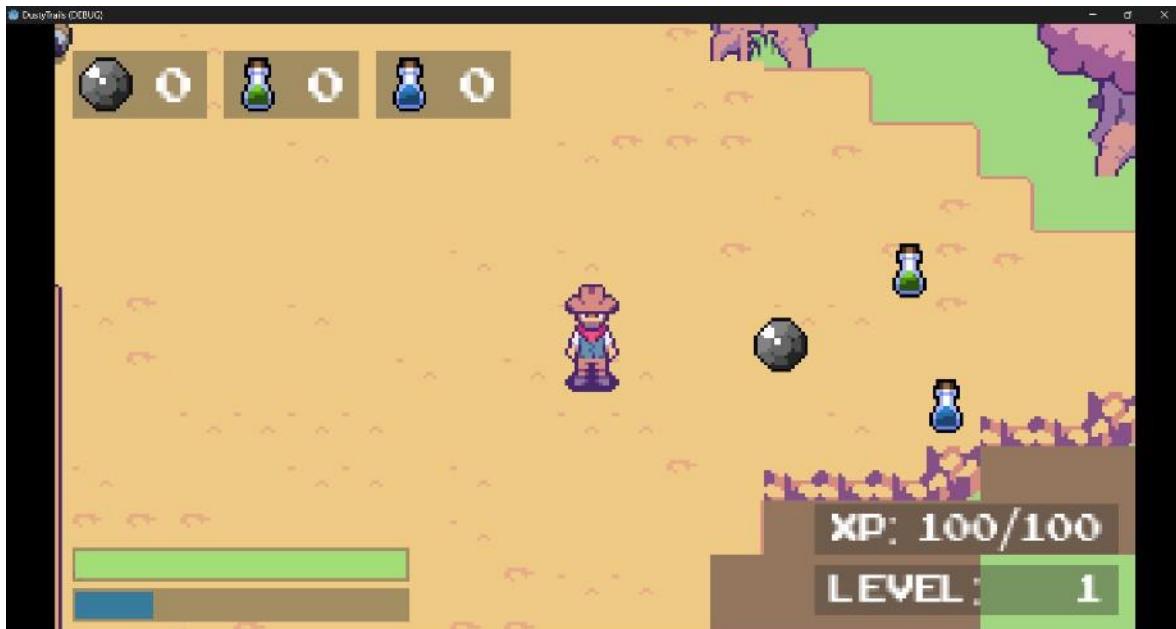
# Node refs
@onready var sprite = $Sprite2D

#older code

# ----- Icon -----
func _ready():
    #executes the code in the game
    if not Engine.is_editor_hint():
        #if we choose x item from Inspector dropdown, change the texture
        if item == Pickups.AMMO:
            sprite.set_texture(ammo_texture)
        elif item == Pickups.HEALTH:
            sprite.set_texture(health_texture)
        elif item == Pickups.STAMINA:
            sprite.set_texture(stamina_texture)

```

If you instance multiple Pickup scenes in your Main scene and change their item value, and you run your scene, you will now see the textures changed in-game as well.

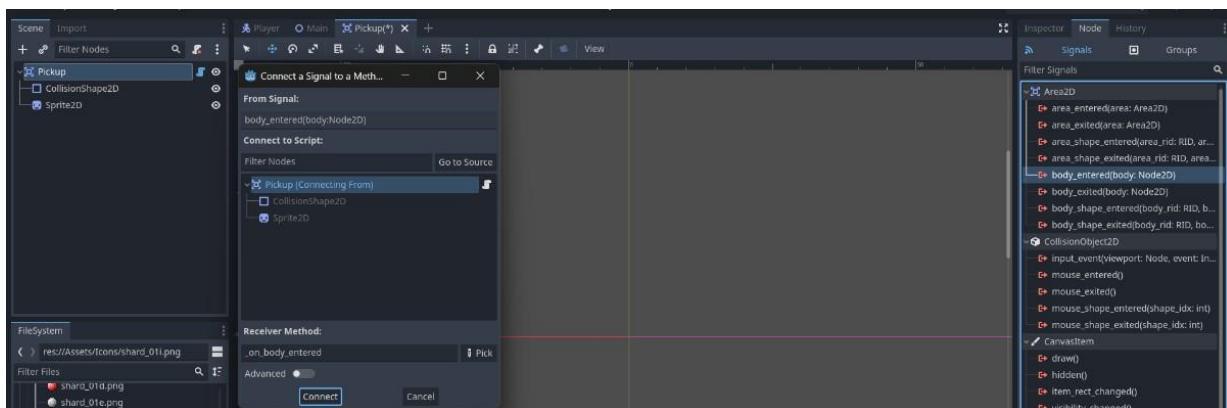


USING PICKUPS

If our player runs through these pickups, we want to remove them from our scene and add them to our player's inventory. We will have to make use of some custom signals and functions again to make this work in our Player script, but first, let's focus on removing the items from the scene when our player collides with it.

The Area2D node comes with a built-in signal called [body_entered\(\)](#), which emits when a defined body enters this area. This defined body is any body that has a collision shape added to it, such as our CharacterBody2D from our Player scene.

Connect this signal to your Pickup script. You will see that it creates a new '`func _on_body_entered(body):`' function at the end of your script.



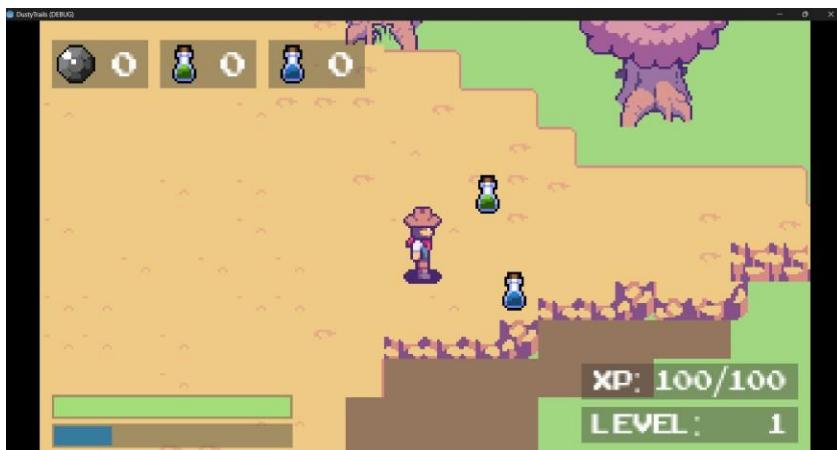
In this new signal function, we want to check if the body that entered our area is called "Player", and if so, our Pickup scene should delete itself from our Main [scene tree](#). When a node is part of a scene tree, the tree can be obtained by calling the [get_tree\(\)](#) method. To remove a node from a scene tree, we can use either [queue_free\(\)](#) or [queue_delete\(\)](#).

```
### Pickup.gd
#older code

# ----- Using Pickups -----
func _on_body_entered(body):
    if body.name == "Player":
```

```
#todo: adding function will come here  
#delete from scene tree  
get_tree().queue_delete(self)
```

If you run your scene now and you have your player run through your pickup items, it should be removed from the scene completely.



You'll see in my code sample that I added a line for `"#todo: adding function will come here"`. We will first need to create the function that will add these items to our player's inventory from the Player script, and then reference it in our Pickup script. Our player's inventory won't be a traditional one that we can add many items and objects to, but more of an invisible one that will only be shown through our pre-existing GUI. In other words, our inventory will only be able to contain Pickup items.

In your Player script, let's define some more signals to update our pickup item's values. These signals will emit whenever we remove or add a pickup item.

```
### Player.gd  
  
# older code  
  
# Custom signals  
signal health_updated  
signal stamina_updated  
signal ammo_pickups_updated  
signal health_pickups_updated  
signal stamina_pickups_updated
```

We also need to define our `Pickups` enum again so that we can use the same constants.

```
### Player.gd

# older code

# Custom signals
signal health_updated
signal stamina_updated
signal ammo_pickups_updated
signal health_pickups_updated
signal stamina_pickups_updated

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }
```

Next, we need to define a new variable for each pickup item so that we can store their amount.

```
### Player.gd

# older code

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }
var ammo_pickup = 0
var health_pickup = 0
var stamina_pickup = 0
```

Finally, we can go ahead and create our custom function that will add the pickups to our player's inventory (the GUI in the top-left of the screen). We will call this function in our `Pickups` script, so we can pass the `item` variable into our parameter as that is what we will choose from in our Inspector panel.

We will do a conditional to check which item has been picked up, and then add x amount of that item to our player. We also need to emit our signals to notify the game that the value of our variables has changed.

```

### Player.gd

# older code

# ----- Consumables -----
# Add the pickup to our GUI-based inventory
func add_pickup(item):
    if item == Pickups.AMMO:
        ammo_pickup = ammo_pickup + 3 # + 3 bullets
        ammo_pickups_updated.emit(ammo_pickup)
        print("ammo val:" + str(ammo_pickup))
    if item == Pickups.HEALTH:
        health_pickup = health_pickup + 1 # + 1 health drink
        health_pickups_updated.emit(health_pickup)
        print("health val:" + str(health_pickup))
    if item == Pickups.STAMINA:
        stamina_pickup = stamina_pickup + 1 # + 1 stamina drink
        stamina_pickups_updated.emit(stamina_pickup)
        print("stamina val:" + str(stamina_pickup))

```

Now we can go back to our Pickup script and reference this function via the body method to add that item.

```

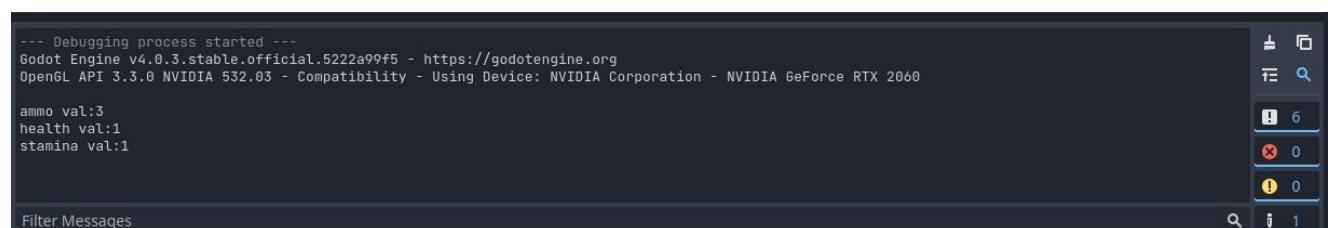
### Pickup.gd

# older code

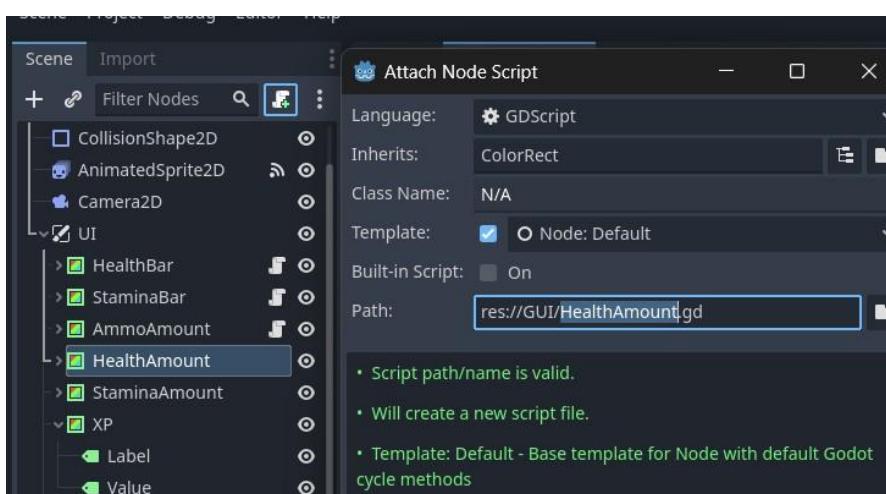
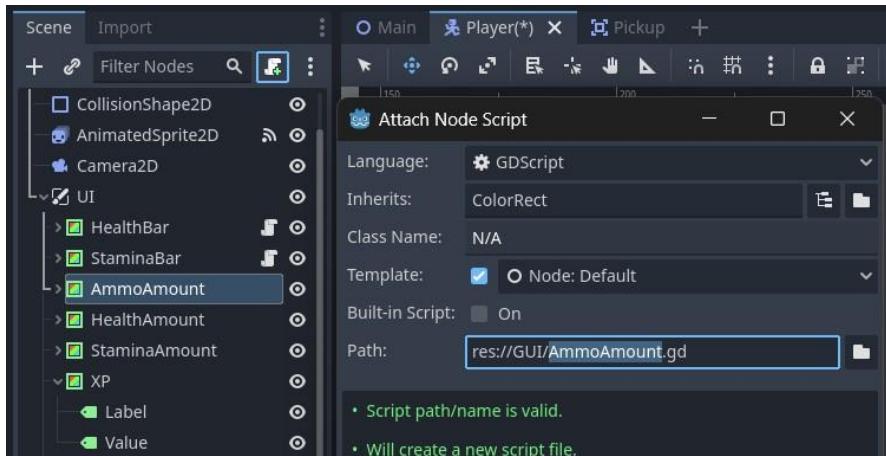
# ----- Using Pickups -----
func _on_body_entered(body):
    if body.name == "Player":
        body.add_pickup(item)
        #delete from scene tree
        get_tree().queue_delete(self)

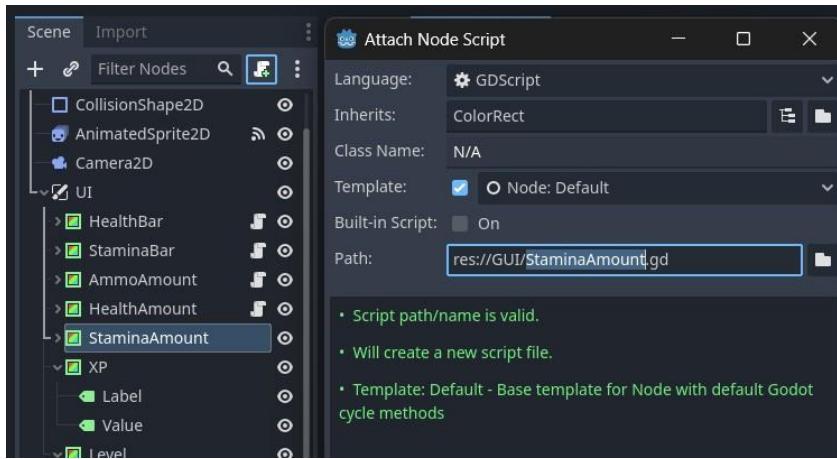
```

We can now run our scene to see if this works when we have our player run through our Pickup items.



Our pickup items now get added to our player's inventory, but we do not see a change in our GUI yet. We will add that now so that our changes can be reflected in the game and not in the console. In your Player scene, attach a script to each of your UI nodes called AmmoAmount, HealthAmount, and StaminaAmount. Save them underneath your GUI folder.





Just like with the scripts for our HealthBar and StaminaBar elements, we will create a function that will update its value via the connected signal in the Player script.

```
### AmmoAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value

# Update ui
func update_ammo_pickup_ui(ammo_pickup):
    value.text = str(ammo_pickup)
```

```
### HealthAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value

# Update ui
func update_health_pickup_ui(health_pickup):
    value.text = str(health_pickup)
```

```
### StaminaAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value
```

```
# Update ui
func update_stamina_pickup_ui(stamina_pickup):
    value.text = str(stamina_pickup)
```

```
### Player.gd

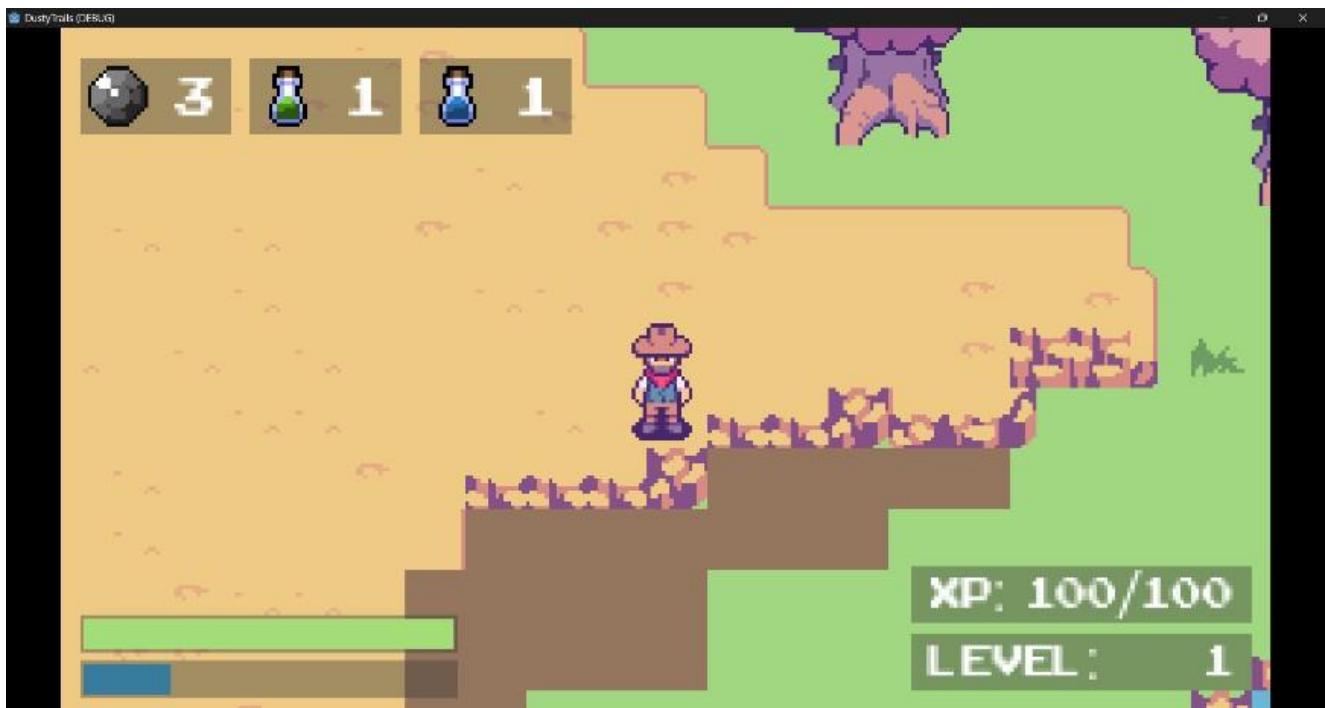
extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount

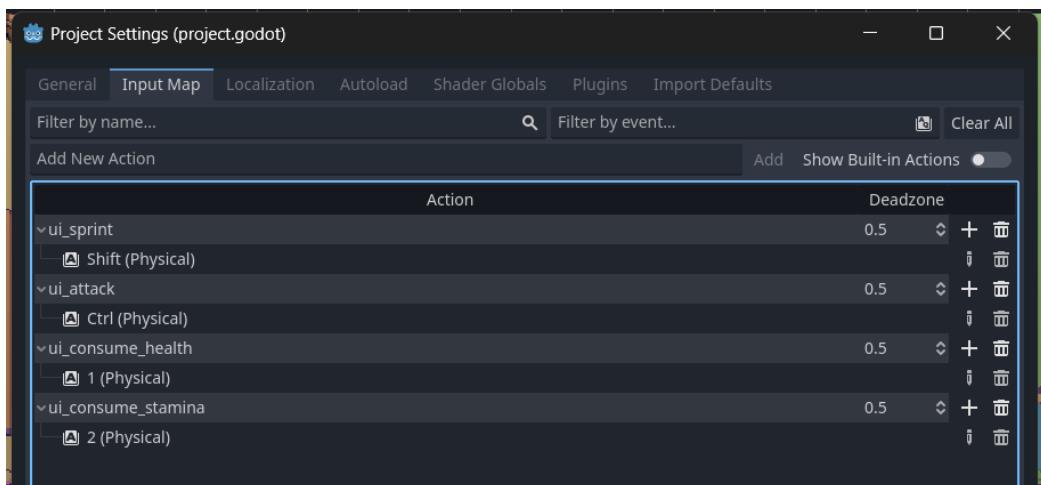
# older code

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
    health_pickups_updated.connect(health_amount.update_health_pickup_ui)
    stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)
```

Now if you run your scene and you have your player character run through your pickups, your GUI should update. Whilst we're adding the functionality to add pickups to our player, let's also go ahead and add some inputs so that our player can consume our health and stamina pickups. We'll update our code later on, to also use ammo pickups, or bullets, to attack.



Create two new input actions called "ui_consume_health" and "ui_consume_stamina" and assign input keys to them. I used the “1” and “2” keys on my keyboard but assign whatever keys you feel comfortable with.



In our Player script, in our *input(event)* function, let's create new conditionals for these input actions. If we have pickups, and our health or stamina value on our progress bar is low, we want to deduct 1 from our pickups amount when our player presses “1” or “2” to consume either a health or stamina drink.

Drinking these consumables will restore 50 points of the total 100 points assigned (remember our health and stamina variables are equal to 100). After our health or stamina has been added to and a pickup has been removed, we need to then emit our signals accordingly.

```
### Player.gd

extends CharacterBody2D

# older code

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #attacking/shooting anim
        is_attacking = true
        var animation = "attack_" + returned_direction(new_direction)
        animation_sprite.play(animation)
    #using health consumables
    elif event.is_action_pressed("ui_consume_health"):
        if health > 0 && health_pickup > 0:
            health_pickup = health_pickup - 1
            health = min(health + 50, max_health)
            health_updated.emit(health, max_health)
            health_pickups_updated.emit(health_pickup)
    #using stamina consumables
    elif event.is_action_pressed("ui_consume_stamina"):
        if stamina > 0 && stamina_pickup > 0:
            stamina_pickup = stamina_pickup - 1
            stamina = min(stamina + 50, max_stamina)
            stamina_updated.emit(stamina, max_stamina)
            stamina_pickups_updated.emit(stamina_pickup)
```

To also have our UI nodes show the correct values when we spawn (for example, say we want to give our player 5 ammo on spawn instead of 0), we will have to update our value nodes on load in our UI scripts.

```
### AmmoAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value
```

```

@onready var player = $".../..."

# Show correct value on load
func _ready():
    value.text = str(player.ammo_pickup)

# Update ui
func update_ammo_pickup_ui(ammo_pickup):
    value.text = str(ammo_pickup)

```

```

### HealthAmount.gd
extends ColorRect

# Node ref
@onready var value = $value
@onready var player = $".../..."

# Show correct value on load
func _ready():
    value.text = str(player.health_pickup)

# Update ui
func update_health_pickup_ui(health_pickup):
    value.text = str(health_pickup)

```

```

### StaminaAmount.gd
extends ColorRect

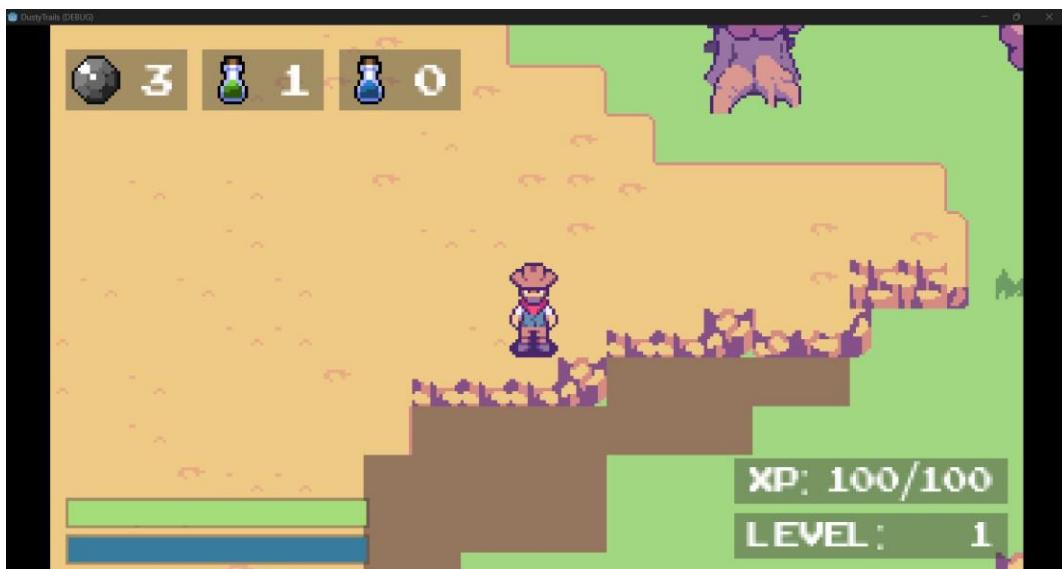
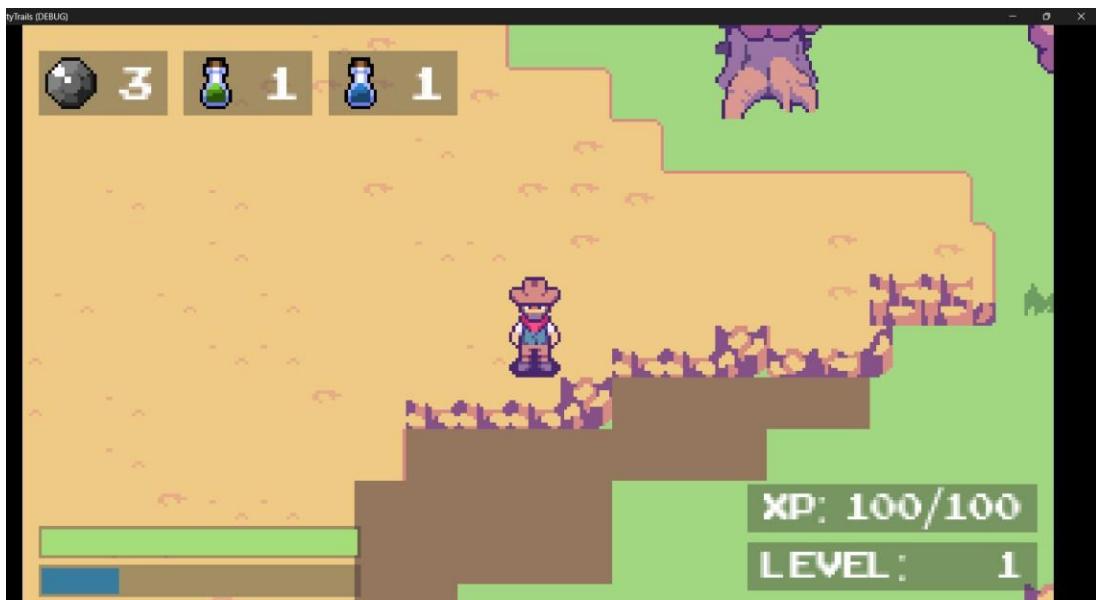
# Node ref
@onready var value = $value
@onready var player = $".../..."

# Show correct value on load
func _ready():
    value.text = str(player.stamina_pickup)

# Update ui
func update_stamina_pickup_ui(stamina_pickup):
    value.text = str(stamina_pickup)

```

Now if you change your variables to have different values (such as `ammo_pickup = 2`) and you run your scene, the correct values should show. If you pick up your stamina pickup, sprint, and then press "2" to consume your stamina pickup, your GUI value for your stamina should update as well as your stamina progress bar!



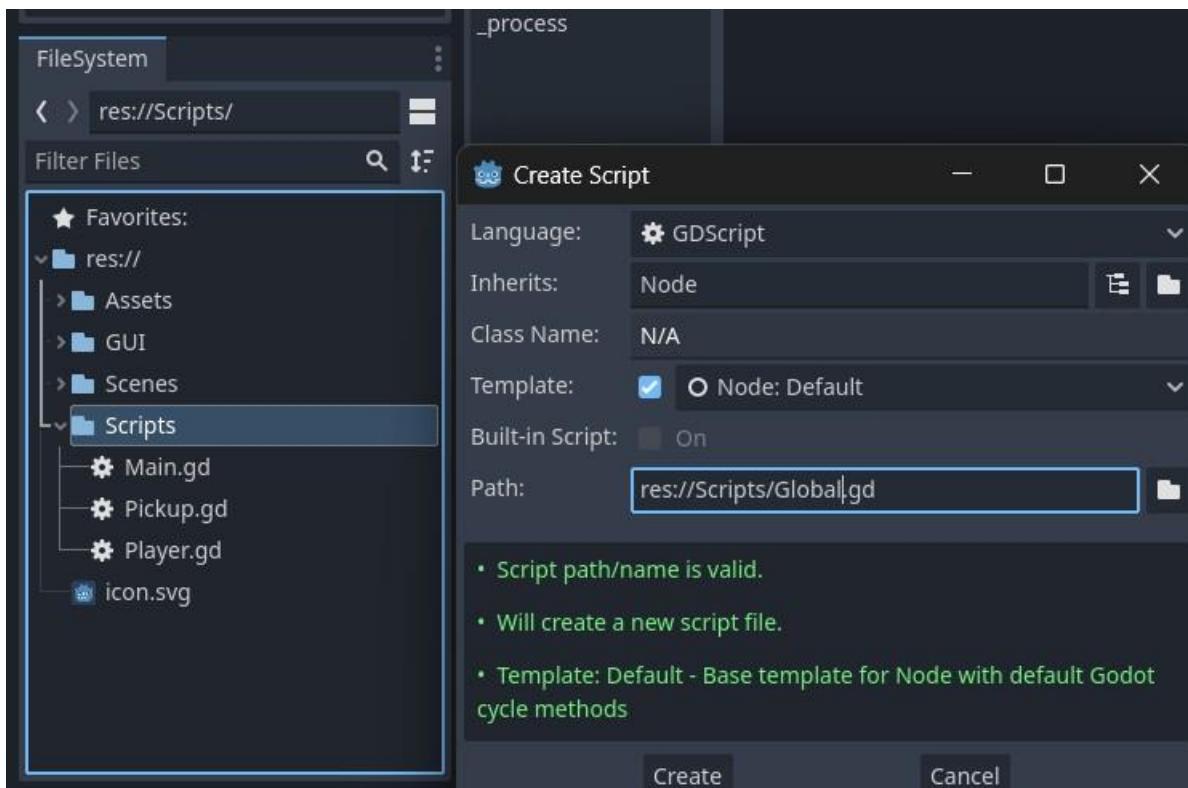
SPawning PICKUPS

Instead of manually placing down our pickups, let's create a randomizer which will randomly generate x amount of pickups on our map when we run the game. For this we will create a new Global [Autoload Singleton](#) script. This script will contain all the variables, scene resources, and methods that will be used throughout our game in multiple scenes.

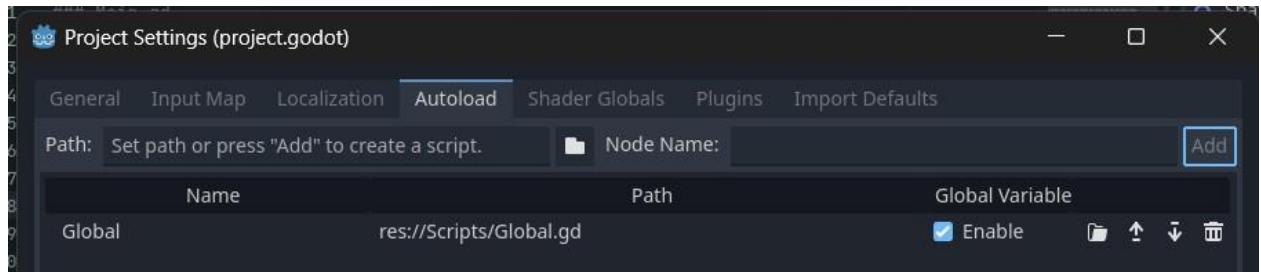
What is a Singleton Script?

A singleton script is a script that is automatically loaded when the game starts and remains accessible throughout the entire lifetime of the game. This makes it ideal for managing game-wide data, functions, and systems that need to be accessed from multiple scenes or scripts.

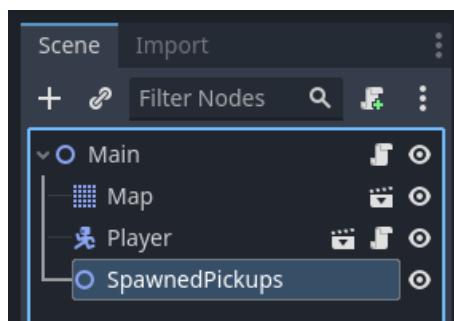
Underneath your Scripts folder, create a new script called "Global.gd".



Then, in your Project Settings > Autoload (Global in Godot 4.3) menu, assign the Global script as a new singleton. This will make the variables and functions stored in this script accessible in every script.



Now, in your Main scene, delete any Pickups scene that you instanced. Then add a Node2D node renamed to “SpawnedPickups”. This node will act as an organizer for our pickups, i.e. it will hold all of our spawned pickups instances.



Now, in your newly created Global script, let's [preload](#) the Pickups scene so that it can be instantiated later. By preloading scenes in our Global script, we can instantiate these scenes in our other scenes without loading them in each script. Preloading scenes reduces resource load times and potential lagging.

When to use load vs preload?

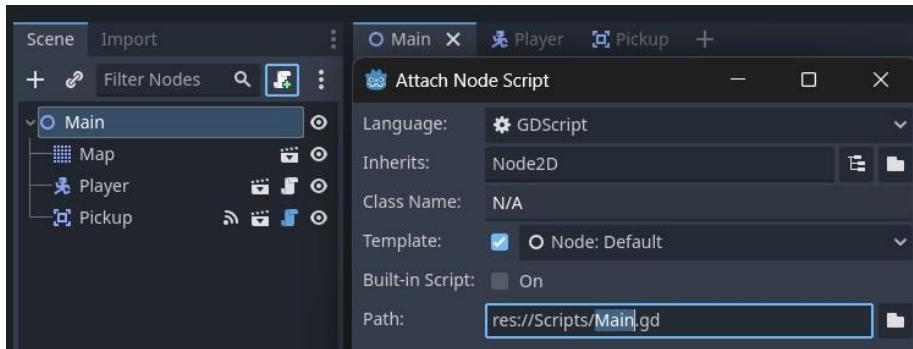
Both load and preload can be used to load resources in GDScript. Use preload when you know that a resource will be used on load, and you want it to be loaded during the script's compilation time. Use load when you want to only load the resource when a certain condition is met, such as when a function is called.

```
### Global.gd

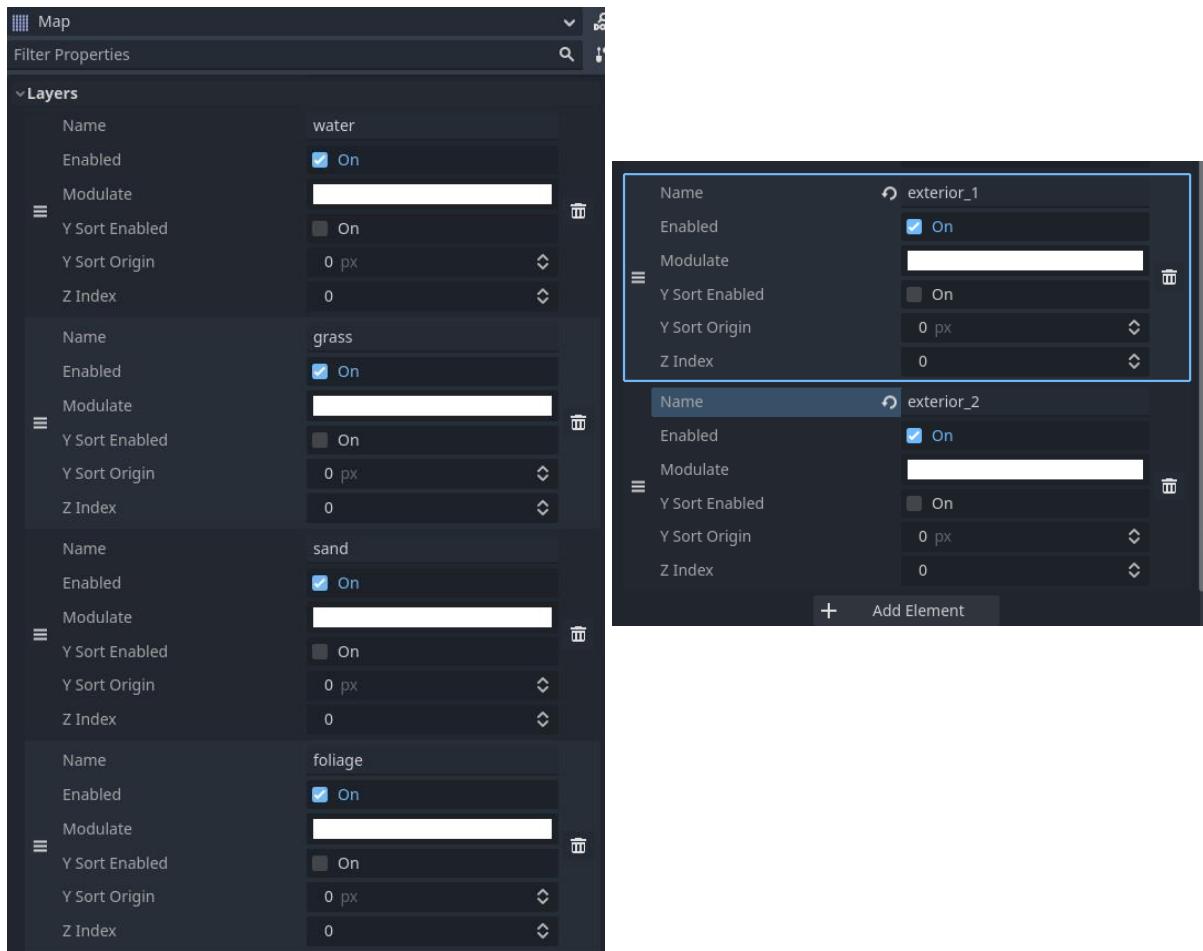
extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
```

Next, attach a new script to your Main scene. Save this script underneath your Scripts folder.



Now, in our Main script, let's create a node reference to our TileMap node (Map) as well as the layers on our Map node. Remember, the ID of the layers start from zero, hence water = 0, grass = 1, sand = 2, foliage = n. Only add references to the layers that you have. For example, I have exterior_1 and exterior_2, but you might not.



```
### Main.gd

extends Node2D

# Node refs
@onready var map = $Map

# TileMap layers
const WATER_LAYER = 0
const GRASS_LAYER = 1
const SAND_LAYER = 2
const FOLIAGE_LAYER = 3
const EXTERIOR_1_LAYER = 4
const EXTERIOR_2_LAYER = 5
```

Now, let's create a function that checks if a given position on the TileMap is a valid spawn location. This function will check the cell type in the specified layer (either the first or second layer) to determine if it's a valid location for spawning a pickup. We will check the cell type on the specified layer via the [get_cell_source_id\(\)](#) method, which retrieves information about a tile at a specific cell position in a particular layer of a TileMap.

What's the difference between get_cell_source_id and get_cell_tile_data?

Both `get_cell_source_id` and `get_cell_tile_data` are methods used to query information about a cell in a tilemap, but they return different types of information. The method [get_cell_source_id](#) returns only the ID of a specific cell, and thus we use this method for simple checks, like whether a cell is empty or contains a specific type of tile. The [get_cell_tile_data](#) method returns more detailed information about the tile, not just its ID, and thus we use this method to get custom data, transform properties, or state values of the cell or tile.

If the cell is on layer SAND or GRASS, then we will return it as a valid spawn location. If the cell is on any other layer, then it will not be a valid spawn location. This will prevent the pickups from spawning on the buildings and water.

```

### Main.gd

# older code

# Valid pickup spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)

    # Check if there's a tile on the water, foliage, or exterior layers
    if map.get_cell_source_id(WATER_LAYER, cell_coords) != -1 ||
    map.get_cell_source_id(FOLIAGE_LAYER, cell_coords) != -1 ||
    map.get_cell_source_id(EXTERIOR_1_LAYER, cell_coords) != -1 ||
    map.get_cell_source_id(EXTERIOR_2_LAYER, cell_coords) != -1:
        return false

    # Check if there's a tile on the grass or sand layers
    if map.get_cell_source_id(GRASS_LAYER, cell_coords) != -1 ||
    map.get_cell_source_id(SAND_LAYER, cell_coords) != -1:
        return true

    return false

```

Now, to spawn the pickups we'll need to create a new function. This function will randomly choose a position on the TileMap and check if it's a valid spawn location. If it is, it will instantiate a pickup at that location. We'll instance the scene that we loaded in our Global script. In Godot 4, we instance a scene reference via the [instantiate](#) method.

```

### Main.gd

# older code

# Spawn pickup
func spawn_pickups(amount):
    var spawned = 0
    while spawned < amount:
        # Randomly choose a location on the first or second layer
        var random_position = Vector2(randi() % map.get_used_rect().size.x,
                                       randi() % map.get_used_rect().size.y)
        var layer = randi() % 2
        # Spawn it underneath SpawnedPickups node
        if is_valid_spawn_location(layer, random_position):
            var pickup_instance = Global.pickups_scene.instantiate()
            pickup_instance.position = map.map_to_local(random_position)

```

```
    spawned_pickups.add_child(pickup_instance)
    spawned += 1
```

What is map_to_local?

The TileMap node's [map_to_local\(Vector2i map_position\)](#) method converts a given map position (such as our player's coordinates) to the TileMap's local coordinate space. This is useful when you want to find the pixel position of a specific cell within the TileMap in its local coordinate system.

Finally, in our ready function, we can call our function to spawn a random number of pickups on our map on load. We'll need to use the [RandomNumberGenerator](#) to randomize the amount of pickups so that it is different each time the function is called.

```
### Main.gd

extends Node2D

# Node refs
@onready var map = $Map
@onready var spawned_pickups = $SpawnedPickups

# TileMap layers
const WATER_LAYER = 0
const GRASS_LAYER = 1
const SAND_LAYER = 2
const FOLIAGE_LAYER = 3
const EXTERIOR_1_LAYER = 4
const EXTERIOR_2_LAYER = 5
var rng = RandomNumberGenerator.new()

func _ready():
    # Spawn between 5 and 10 pickups
    var spawn_pickup_amount = rng.randf_range(5, 10)
    spawn_pickups(spawn_pickup_amount)
```

If you run your scene now, it will only spawn one type of pickup - ammo, health, or stamina. We want it to be randomized. Let's remove our Pickups enum from our Player and Pickup scripts, and re-define it in our Global script. We'll then need to update our references to our enum.

```
### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }
```

```
### Player.gd

# older code

# ----- Consumables -----
# Add the pickup to our GUI-based inventory
func add_pickup(item):
    if item == Global.Pickups.AMMO:
        ammo_pickup = ammo_pickup + 3 # + 3 bullets
        ammo_pickups_updated.emit(ammo_pickup)
        print("ammo val:" + str(ammo_pickup))
    if item == Global.Pickups.HEALTH:
        health_pickup = health_pickup + 1 # + 1 health drink
        health_pickups_updated.emit(health_pickup)
        print("health val:" + str(health_pickup))
    if item == Global.Pickups.STAMINA:
        stamina_pickup = stamina_pickup + 1 # + 1 stamina drink
        stamina_pickups_updated.emit(stamina_pickup)
        print("stamina val:" + str(stamina_pickup))
```

```
### Pickup.gd

# Pickups to choose from
@export var item : Global.Pickups

# ----- Icon -----
func _ready():
    #executes the code in the game
    if not Engine.is_editor_hint():
        #if we choose x item from Inspector dropdown, change the texture
        if item == Global.Pickups.AMMO:
```

```

        sprite.set_texture(ammo_texture)
    elif item == Global.Pickups.HEALTH:
        sprite.set_texture(health_texture)
    elif item == Global.Pickups.STAMINA:
        sprite.set_texture(stamina_texture)

#allow us to change the icon in the editor
func _process(_delta):
    #executes the code in the editor without running the game
    if Engine.is_editor_hint():
        #if we choose x item from Inspector dropdown, change the texture
        if item == Global.Pickups.AMMO:
            sprite.set_texture(ammo_texture)
        elif item == Global.Pickups.HEALTH:
            sprite.set_texture(health_texture)
        elif item == Global.Pickups.STAMINA:
            sprite.set_texture(stamina_texture)

```

Now, in our Main scene, let's randomize the pickup type before we add it to our scene.

```

### Main.gd

# older code

# Spawn pickup
func spawn_pickups(amount):
    var spawned = 0
    while spawned < amount:
        # Randomly choose a location on the first or second layer
        var random_position = Vector2(randi() % map.get_used_rect().size.x,
                                       randi() % map.get_used_rect().size.y)
        var layer = randi() % 2
        # Spawn it underneath SpawnedPickups node
        if is_valid_spawn_location(layer, random_position):
            var pickup_instance = Global.pickups_scene.instantiate()
            # Randomly select a pickup type
            pickup_instance.item = Global.Pickups.values()[randi() % 3]
            # Add pickup to scene
            pickup_instance.position = map.map_to_local(random_position)
            spawned_pickups.add_child(pickup_instance)
            spawned += 1

```

Additionally, the loop in `spawn_pickups` might be infinite if there aren't enough valid spawn locations. To prevent this, you can add a maximum number of attempts:

```
### Main.gd

# older code

# Spawn pickup
func spawn_pickups(amount):
    var spawned = 0
    var attempts = 0
    var max_attempts = 1000 # Arbitrary number, adjust as needed

    while spawned < amount and attempts < max_attempts:
        attempts += 1
        var random_position = Vector2(randi() % map.get_used_rect().size.x,
                                       randi() % map.get_used_rect().size.y)
        var layer = randi() % 2
        if is_valid_spawn_location(layer, random_position):
            var pickup_instance = Global.pickups_scene.instantiate()
            pickup_instance.item = Global.Pickups.values()[randi() % 3]
            pickup_instance.position = map.map_to_local(random_position)
            spawned_pickups.add_child(pickup_instance)
            spawned += 1
```

This will ensure that the loop doesn't run indefinitely if there aren't enough valid spawn locations. If you run your scene now, your pickups should spawn randomly on the map!



In the next part, we will get to the fun part of the game. We will set up our first enemy that will challenge our player in the game. It's going to be a long ride, so remember to save your game project, and I'll see you in the next part.

Your final source code for this part should look like [this](#).

PART 9: ENEMY AI SETUP

No game can be complete without some sort of threat or enemy to defeat. In our game, we will have a cactus enemy that spawns on the map at a constant value, meaning that there will never be more or less than x amount of enemies on our map during the game loop. This enemy will damage our player if our player gets shot by it, and it will also shoot at our player. Let's get started with our enemy AI.

WHAT YOU WILL LEARN IN THIS PART:

- How to add movement to non-controllable nodes.
- How to work with the Timer node.
- How to work with the RandomNumberGenerator class.
- How to make nodes move around randomly, as well as move towards other nodes.

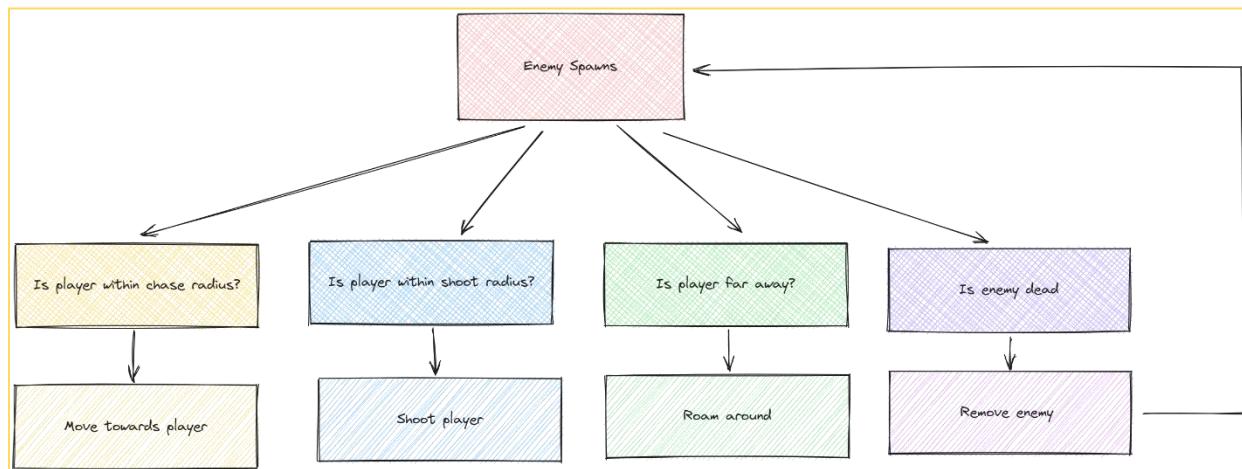
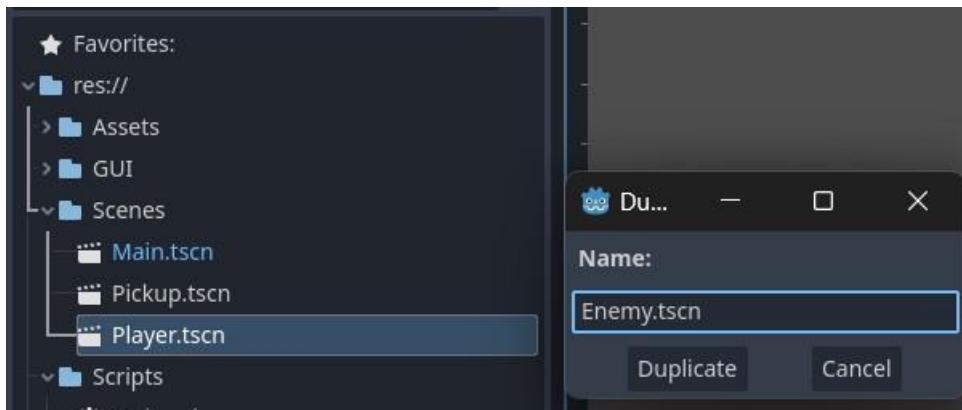
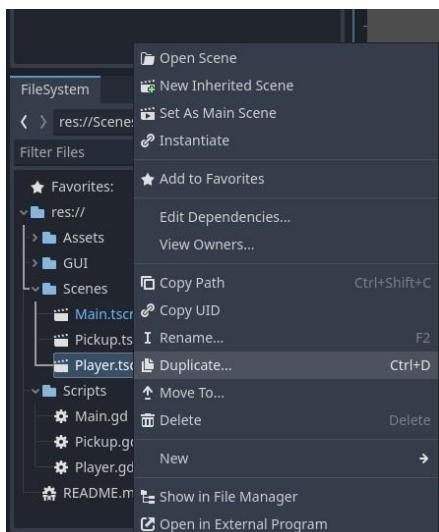


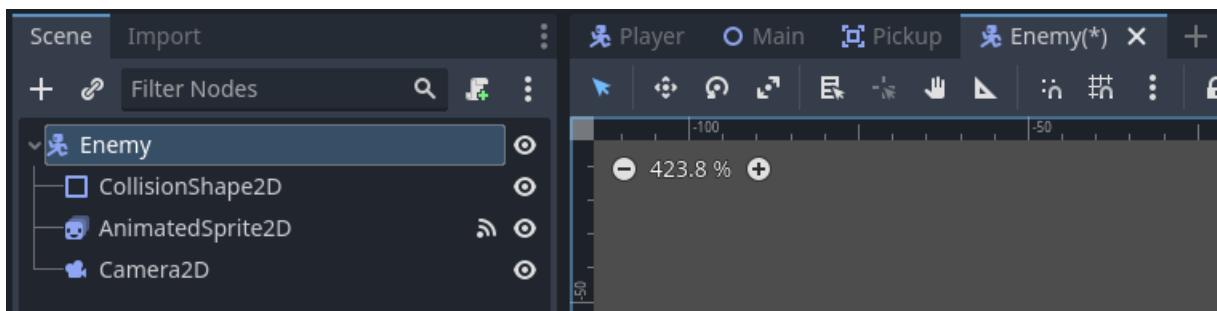
Figure 13: Enemy Overview

ENEMY SCENE SETUP

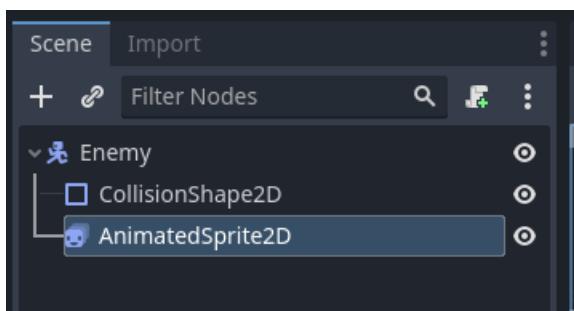
Our Enemy scene will have the same structure as our Player scene, with a CharacterBody2D as its root node followed by an AnimatedSprite2D node and a CollisionShape2D node. Therefore, we can just go ahead and duplicate our Player scene and rename it to "Enemy" or "Cactus". I'm only going to have a Cactus as my enemy, so I'll call mine the general term "Enemy", but if you're going to have multiple different enemies, you can create a scene for "Cactus", "Bandit", "Tumbleweed", etc.



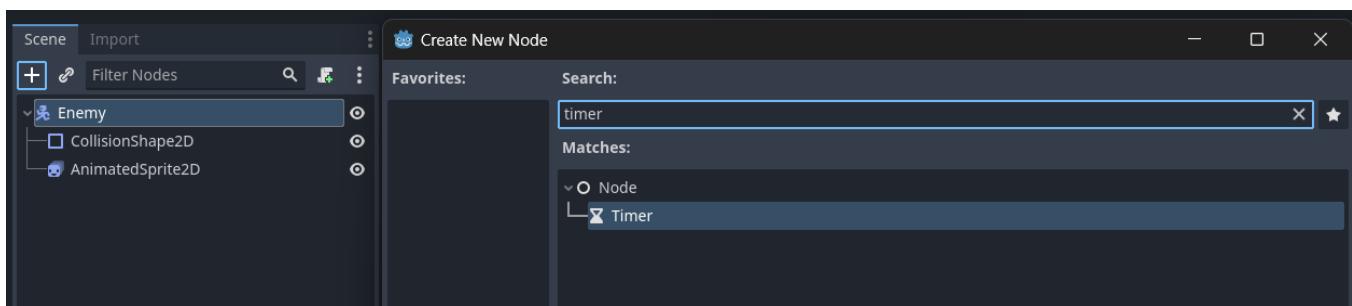
Rename your scene Root to whatever you called it and detach the Player script from your scene. Delete the entire UI tree from this scene.



You should also delete the Camera2D node since we won't follow this node around and also disconnect your `on_animation_finished()` signal from your AnimatedSprite2D node. Your final scene should look like the image below.

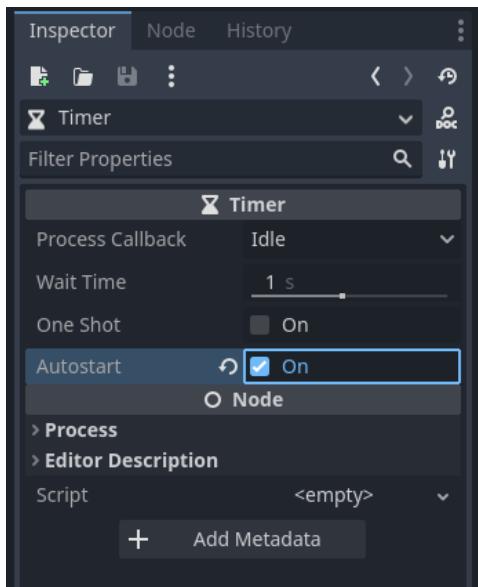


This Enemy scene will constantly have to be monitored to update its behavior. For example, let's say we want it to roam around for 1 minute, and after that 1 minute, we want it to stop for 30 seconds before redirecting and roaming again. We can use a [Timer](#) node that will emit its built-in `timeout()` signal after it counts down a specified interval until it reaches 0. So each time the timer times out, the enemy should redirect and roam again.



Your timer has two options: **one-shot** (if true, the timer will stop when reaching 0. If false, it will restart) and **autostart** (if true, the timer will automatically start when entering

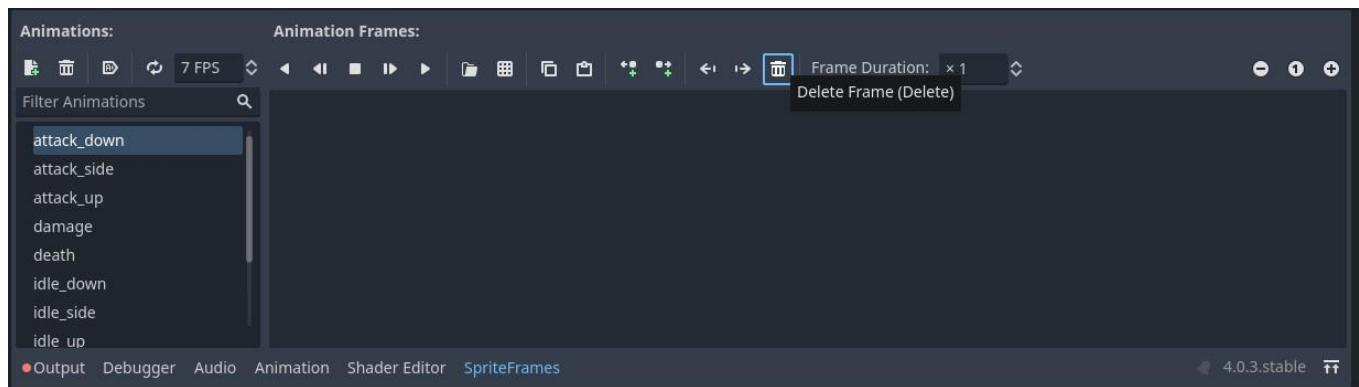
the scene tree). We want this timer to start as soon as our game starts because we want to use it to update our enemy's movements after a certain amount of time. Therefore, you need to enable the Autostart property in the Timer node's Inspector panel.



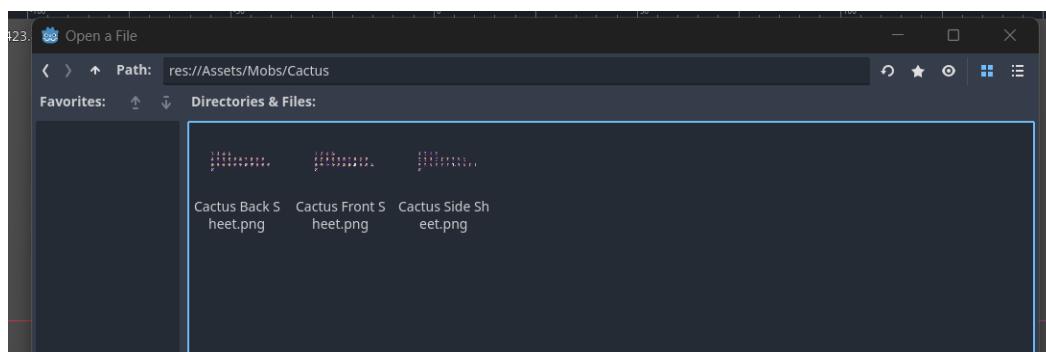
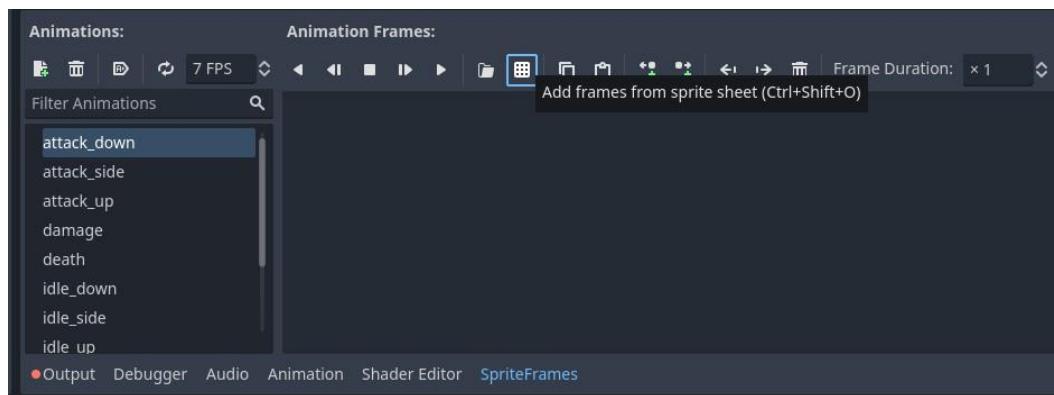
We will come back to this timer node later on when we add the functionality for our enemy's roaming. For now, let's set up the enemy's animations just like we did for our Player. We already have all of our animation names set up for our Enemy since they will be able to do exactly what our Player does, we just need to switch out the animation frames.



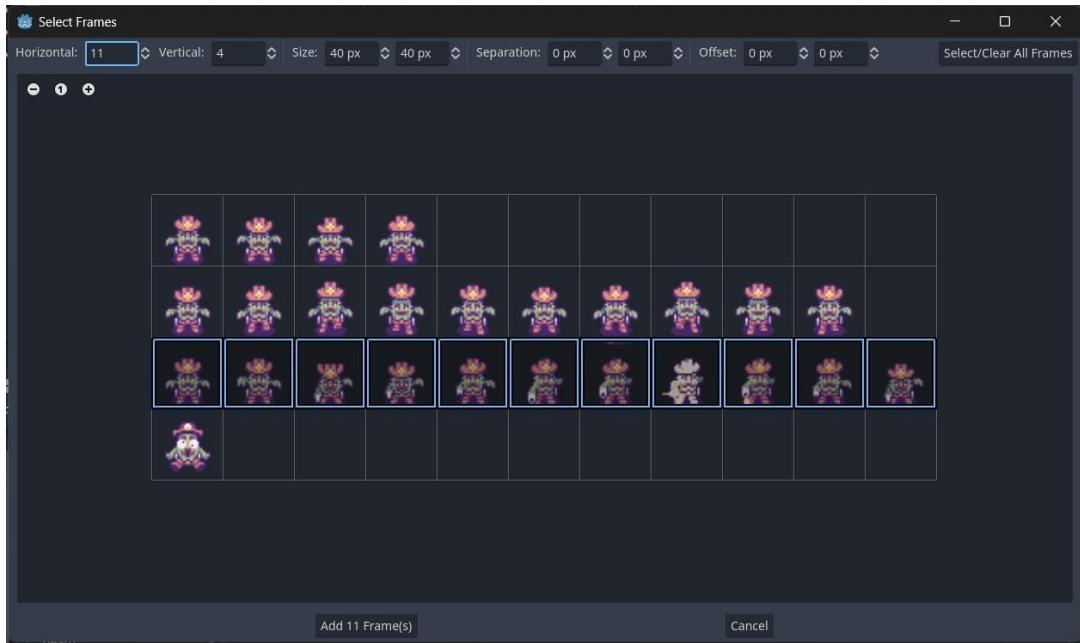
To do that you need to delete the existing sprite frames in your animations. Do this for all the animations, but don't delete any animations.



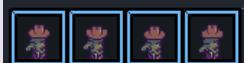
Let's start with our `attack_down` animation. Select the option to "add new sprite frames from sprite sheet" option, and in your Assets > Mobs > Cactus directory, you will find all of your sprite sheets for your enemy. For our `attack_down` animation, we will use the "`Cactus Front Sheet.png`" sheet to create our animation.

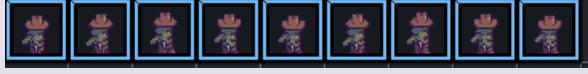
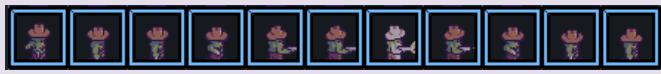
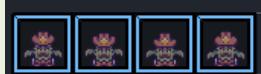
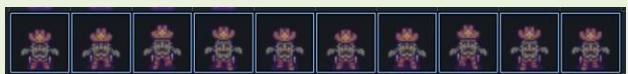
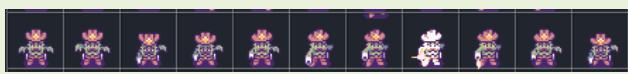


We count 11 frames horizontally and 4 frames vertically, so change your numbers accordingly to crop out your frames correctly. For attack down, we will use the frames in the third row.



I challenge you to now add the rest of the animations on your own using the table below as a guide.

NAME	SPRITE FILE	FRAMES
idle up	Cactus Back Sheet.png	 Row 1, frames 1 > 4 – FPS 4; Looping on
walk_up	Cactus Back Sheet.png	 Row 2, frames 1 > 10, FPS 10; Looping on
attack_up	Cactus Back Sheet.png	 Row 3, frames 1 > 11, FPS 11; Looping off
idle_side	Cactus Side Sheet.png	 Row 1, frames 1 > 4 – FPS 4; Looping on

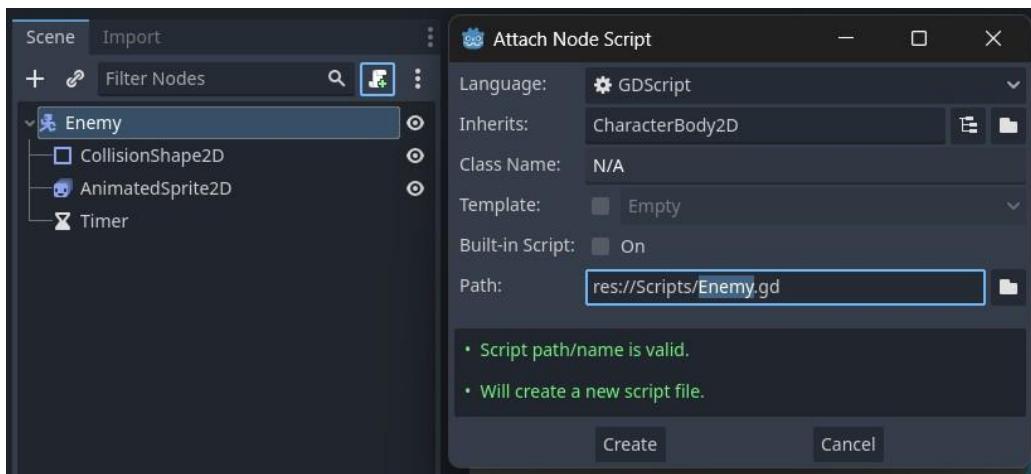
walk_side	Cactus Side Sheet.png	 Row 2, frames 1 > 9, FPS 10; Looping on
attack_side	Cactus Side Sheet.png	 Row 3, frames 1 > 11, FPS 11; Looping off
idle down	Cactus Front Sheet.png	 Row 1, frames 1 > 4 – FPS 4; Looping on
walk_down	Cactus Front Sheet.png	 Row 2, frames 1 > 10, FPS 10; Looping on
attack_down	Cactus Front Sheet.png	 Row 3, frames 1 > 11, FPS 11; Looping off
death	Cactus Front Sheet.png	 Row 4, frames 1 – FPS 1; Looping off
damage	Cactus Front Sheet.png	 Row 4, frames 1 – FPS 1; Looping off

MOVING THE ENEMY

Now we want to be able to move the enemy around autonomously. For that, we need to create a few variables that will store our enemy's direction, and new direction after a random amount of time has passed. Instead of making it a set amount that they will wait

1 minute before redirecting, we will randomize this value. We also want our enemy to redirect after colliding with objects.

Let's attach a new script to the Enemy scene and save it under your Scripts folder.



Our enemy's movement will work similarly to our player's movement, so let's add some familiar variables from our `Player.gd` script to capture its movement speed and new direction. We also need to create a variable that will store its current direction.

```
### Enemy.gd

extends CharacterBody2D

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction = Vector2(0,1) # next direction
```

Our direction will change when our timer runs out after a randomized countdown. We will generate this random countdown value using the [RandomNumberGenerator](#) class. As the name says, it's a class for generating pseudo-random numbers. The `new()` method is used to create an object from a class.

```
### Enemy.gd

extends CharacterBody2D
```

```

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction = Vector2(0,1) # next direction

# Direction timer
var rng = RandomNumberGenerator.new()
var timer = 0

```

We'll also need to move the enemy towards the player if they spot our player in a certain radius, so let's add a reference to our player scene.

```

### Enemy.gd

extends CharacterBody2D

# Node refs
var player

```

Now that we have defined our variables, we can go ahead and initialize our random number and player reference in our built-in `ready()` function, because we want these objects to be initialized as soon as our Enemy scene enters our Main scene.

We will connect our player reference to the Player node in our Main scene. Since the Enemy scene will also be instanced in the Main scene - hence sharing a scene tree with the Player scene, we can get our player by the `get_tree().root.get_node` method. *Main* is our Main scene, and */Player* is the Player instance in our Main scene.

```

### Enemy.gd

extends CharacterBody2D

# Node refs
@onready var player = get_tree().root.get_node("Main/Player")

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction = Vector2(0,1) # next direction

# Direction timer

```

```
var rng = RandomNumberGenerator.new()
var timer = 0

func _ready():
    rng.randomize()
```

Next, let's add our enemy's movement code. The coding process for our enemy's movement will be similar to that of our player. First, we'll add the code to move them. Then, we simply need to add our timer to redirect our enemy and move them toward the player if they "see" us. After we've done those things, we'll build on that to change their animations according to their movement direction. In this part, we will add the redirection and movement, but not the animations yet.

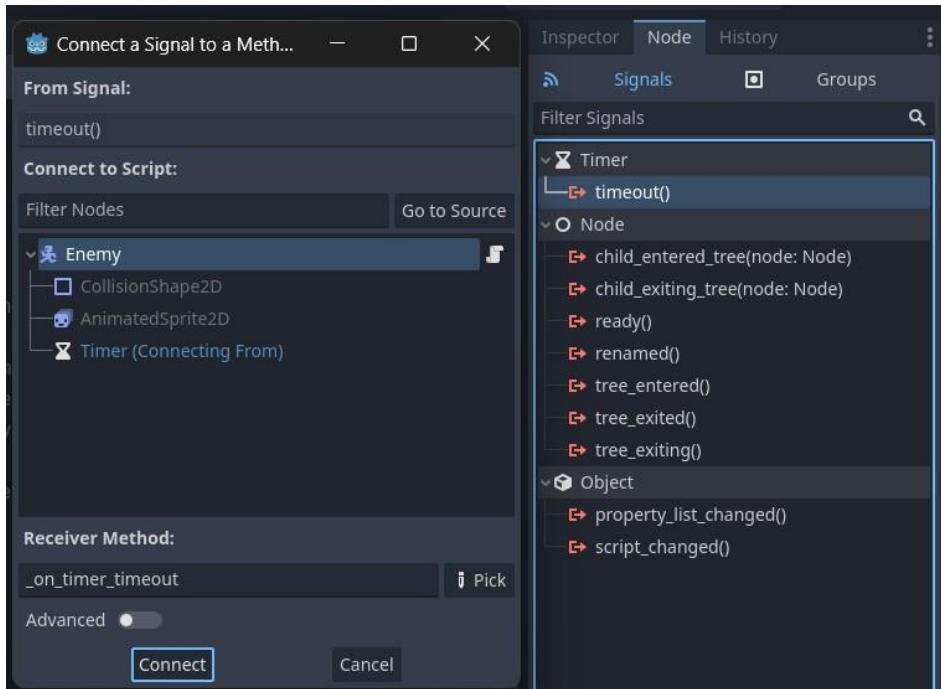
We'll also add our enemy's movement code in our *physics_process()* function since we put all things related to our node's movement and physics in this code. Let's start by adding the functionality for them to move via our [*move_and_collide*](#) method, just like we did for our player.

```
### Enemy.gd

# older code

# ----- Movement & Direction -----
# Apply movement to the enemy
func _physics_process(delta):
    var movement = speed * direction * delta
    var collision = move_and_collide(movement)
```

Now, we need to connect our Timer node's *timeout()* signal to our script. This signal will emit when our timer reaches 0. You'll see that it creates a *func _on_timer_timeout():* function at the end of our script.



In this timeout function, we need to do a few things. First, we need to calculate the player's position relative to our enemy. We can find our player's position returned as a `Vector(0,0)` value by accessing our node's transform values (position, rotation, and scale). By knowing this, we can access our player's position by simply saying: `player.position` - and if we wanted their rotation we could say `player.rotation.x`, and so forth. We can also access our current node's position (which is our enemy node) by simply saying `position` or `self.position`.

After we get our player position, we need to minus it from our enemy's position to get the player's distance from our enemy. Let's go ahead and get this value.

```
### Enemy.gd

# older code

# ----- Movement & Direction -----
# Apply movement to the enemy
func _physics_process(delta):
    var movement = speed * direction * delta
    var collision = move_and_collide(movement)

func _on_timer_timeout():
```

```
# Calculate the distance of the player relative position to the enemy's
position
var player_distance = player.position - position
```

Now, if that distance is within 20 pixels of the enemy, it means that the enemy is close enough to the player that it doesn't have to chase them, but it can go ahead and "attack" or "engage" with the player. You can make this sight value any number, but I'm going to go with 20 pixels.

```
### Enemy.gd

# older code

func _on_timer_timeout():
    # Calculate the distance of the player relative position to the enemy's
    position
    var player_distance = player.position - position
    #turn towards player so that it can attack if within radius
    if player_distance.length() <= 20:
        new_direction = player_distance.normalized()
```

If they are within 100 pixels of the enemy and the timer has run out, it means that the enemy is not close enough to attack the player, so they'll have to move towards and start chasing the player. You can make this chase value any number, but I'm going to go with 100 pixels.

```
### Enemy.gd

# older code

func _on_timer_timeout():
    # Calculate the distance of the player relative position to the enemy's
    position
    var player_distance = player.position - position
    #turn towards player so that it can attack if within radius
    if player_distance.length() <= 20:
        new_direction = player_distance.normalized()
    #chase/move towards player to attack them
    elif player_distance.length() <= 100 and timer == 0:
        direction = player_distance.normalized()
```

Otherwise, if the player is not close to the enemy, or not in our chase radius, then our enemy can go about its day and roam randomly. The enemy's direction will be calculated randomly via the [Vector.DOWN.rotate](#) method, which will calculate a random angle between 0 to 360°. This direction will change each time the timer times out.

```
### Enemy.gd

# older code

func _on_timer_timeout():
    # Calculate the distance of the player relative position to the enemy's
    # position
    var player_distance = player.position - position
    #turn towards player so that it can attack if within radius
    if player_distance.length() <= 20:
        new_direction = player_distance.normalized()
        #chase/move towards player to attack them
    elif player_distance.length() <= 100 and timer == 0:
        direction = player_distance.normalized()
        #random roam
    elif timer == 0:
        #this will generate a random direction value
        var random_direction = rng.randf()
        #This direction is obtained by rotating Vector2.DOWN by a random angle.
        if random_direction < 0.05:
            #enemy stops
            direction = Vector2.ZERO
        elif random_direction < 0.1:
            #enemy moves
            direction = Vector2.DOWN.rotated(rng.randf() * 2 * PI)
```

Finally, we need to use our collider variable from our *physics_process()* function to see if our enemy is colliding with our Player, as well as add our timer range to randomize from. If they are colliding with our player, the timer needs to be set to 0 to trigger our *timeout()* function so that the enemy will chase us.

If they aren't colliding with our player, we need to set our timer randomizer value as well as randomize their direction rotation value so that they can turn around if they collide with other objects. This rotation angle is obtained using the [randf_range\(\)](#) function. This

angle has a value between 45° to 90° . You can change these values to make it smoother or sharper if you'd like.

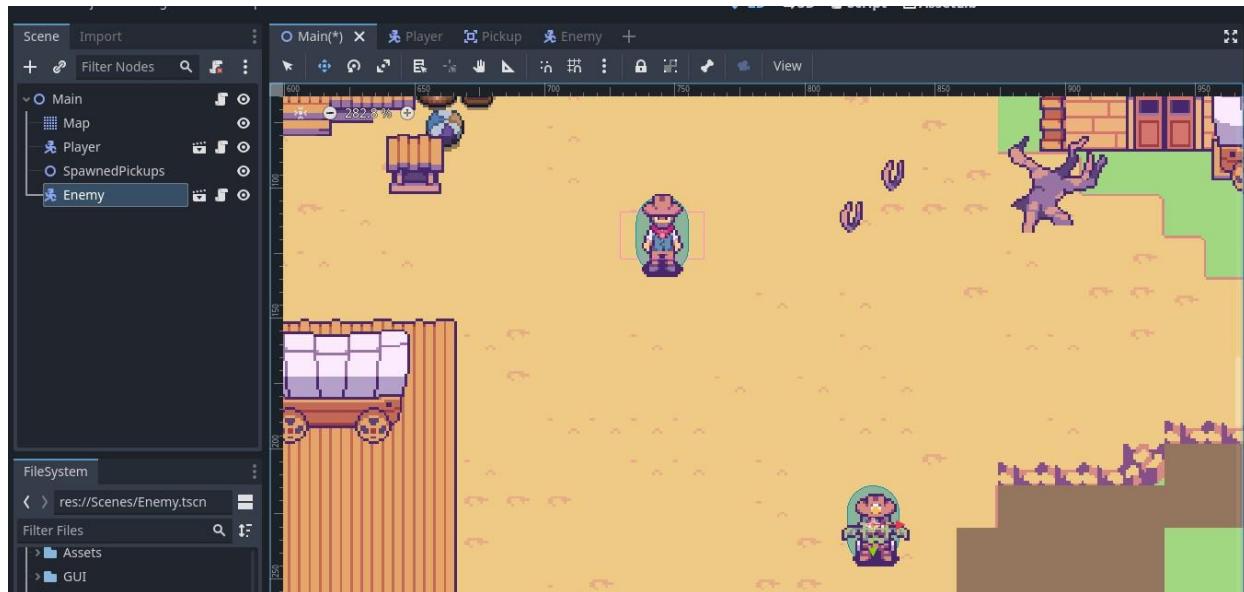
```
### Enemy.gd

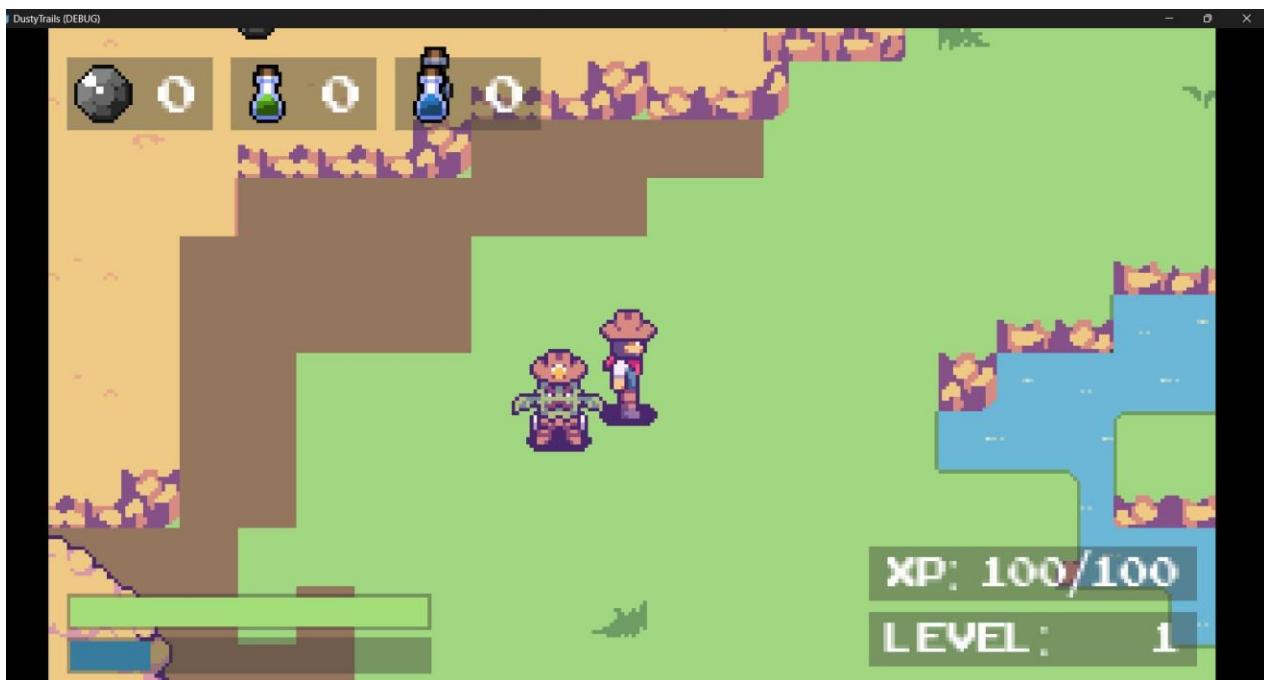
# older code

# Apply movement to the enemy
func _physics_process(delta):
    var movement = speed * direction * delta
    var collision = move_and_collide(movement)

    #if the enemy collides with other objects, turn them around and re-randomize
    #the timer countdown
    if collision != null and collision.get_collider().name != "Player":
        #direction rotation
        direction = direction.rotated(rng.randf_range(PI/4, PI/2))
        #timer countdown random range
        timer = rng.randf_range(2, 5)
        #if they collide with the player
        #trigger the timer's timeout() so that they can chase/move towards our player
    else:
        timer = 0
```

If you instance your Enemy scene in your Main scene, and you run it, then they should chase you or roam around.





And so, we have added an enemy that isn't any threat to us. Our enemy has no animations or any value to it yet, but that will come in the next few parts. In the next section, we will add the functionality to add animations to our enemy's movement. Luckily, we've already added our animations, so it will be a quick setup to display these animations in our enemy's movement! Remember to save your game project and make a backup, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 10: ANIMATING ENEMY AI MOVEMENT

What good is an enemy if it just floats around our map as a static image? That won't do, so let's get to work on implementing our animations so that our enemy can come to life. I also have a surprise for you: we already wrote most of our enemy's animation code. Well, not really, I mean it's still in the Player code, but that means we can go ahead and copy and paste over some code which speeds up our development time tremendously.

WHAT YOU WILL LEARN IN THIS PART:

- How to add animations to non-controllable nodes.
- Further practice with Vectors.

In your Player script, we want to copy two entire functions over to our Enemy script. The first one you should copy is your **func player_animations(direction: Vector2)** function, and the second one is your **func returned_direction(direction: Vector2)** function. Rename your *player_animations()* function in your Enemy script to *enemy_animations()*.

```
90    #animations to play
91 > func player_animations(direction : Vector2): ⇥
103 > ⇥
127   >
128   #returns the animation direction
129 > func returned_direction(direction : Vector2): ⇥
```

```
### Enemy.gd

extends CharacterBody2D

# Node refs
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D
```

```

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction = Vector2(0,1) # next direction
var animation

# older code

# Animation Direction
func returned_direction(direction : Vector2):
    #it normalizes the direction vector to make sure it has length 1 (1, or -1
    # up, down, left, and right)
    var normalized_direction = direction.normalized()
    var default_return = "side"

    if normalized_direction.y > 0:
        return "down"
    elif normalized_direction.y < 0:
        return "up"
    elif normalized_direction.x > 0:
        #(right)
        $AnimatedSprite2D.flip_h = false
        return "side"
    elif normalized_direction.x < 0:
        #flip the animation for reusability (left)
        $AnimatedSprite2D.flip_h = true
        return "side"

    #default value is empty
    return default_return

# Animations
func enemy_animations(direction : Vector2):
    #Vector2.ZERO is the shorthand for writing Vector2(0, 0).
    if direction != Vector2.ZERO:
        #update our direction with the new_direction
        new_direction = direction
        #play walk animation, because we are moving
        animation = "walk_" + returned_direction(new_direction)
        animation_sprite.play(animation)
    else:
        #play idle animation, because we are still
        animation = "idle_" + returned_direction(new_direction)
        animation_sprite.play(animation)

```

To activate these animations for our enemy's movement, we'll have to first check if any other animations are playing (such as attack or death animations), and if not, we play them in our `physics_process()` function. Once again we did the same for our Player character, so this should not be too confusing for you to understand.

Copy in the `is_attacking` variable from your Player's code.

```
### Enemy.gd

extends CharacterBody2D

# Node refs
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction = Vector2(0,1) # next direction
var animation
var is_attacking = false
```

Then in your `physics_process()` function, let's call our `enemy_animations()` function to play our enemy's animations if they are not attacking.

```
### Enemy.gd

# older code

# ----- Movement & Direction -----
# Apply movement to the enemy
func _physics_process(delta):
    var movement = speed * direction * delta
    var collision = move_and_collide(movement)

    #if the enemy collides with other objects, turn them around and re-randomize
    #the timer countdown
    if collision != null and collision.get_collider().name != "Player":
        #direction rotation
        direction = direction.rotated(rng.randrange(PI/4, PI/2))
        #timer countdown random range
        timer = rng.randrange(2, 5)
```

```

#if they collide with the player
#trigger the timer's timeout() so that they can chase/move towards our player
else:
    timer = 0
#plays animations only if the enemy is not attacking
if !is_attacking:
    enemy_animations(direction)

```

If you were to run your enemy scene from here, you might notice that the enemy will try and attack your player - but the problem is that they will try and attack your player even when they are turned away from our player.

In our *timeout()* function, we are setting the new_direction for animation, but this is not updating our new_direction accurately. To fix this, we need to make sure that our new_direction is set accurately during an attack and that it is in sync with the direction the enemy is facing.

For that, we will create a new function that syncs our new_direction with the actual movement direction of our enemy. We will then call this function whenever the enemy moves or rotates. This will make sure that our new_direction is accurately representing the direction the enemy is facing when it's attacking.

Let's create a new function that will sync our new_direction. You can do this above your *timeout()* function.

```

#### Enemy.gd

# older code

# ----- Movement & Direction -----
#syncs new_direction with the actual movement direction and is called whenever
the enemy moves or rotates
func sync_new_direction():
    if direction != Vector2.ZERO:
        new_direction = direction.normalized()

```

Then, we'll call this function in the *func_on_timer_timeout()*: function, which is the place where the enemy decides its behavior (whether it should attack, chase, or roam

randomly). This ensures that whenever the enemy updates its behavior, it also updates its direction for animations accordingly.

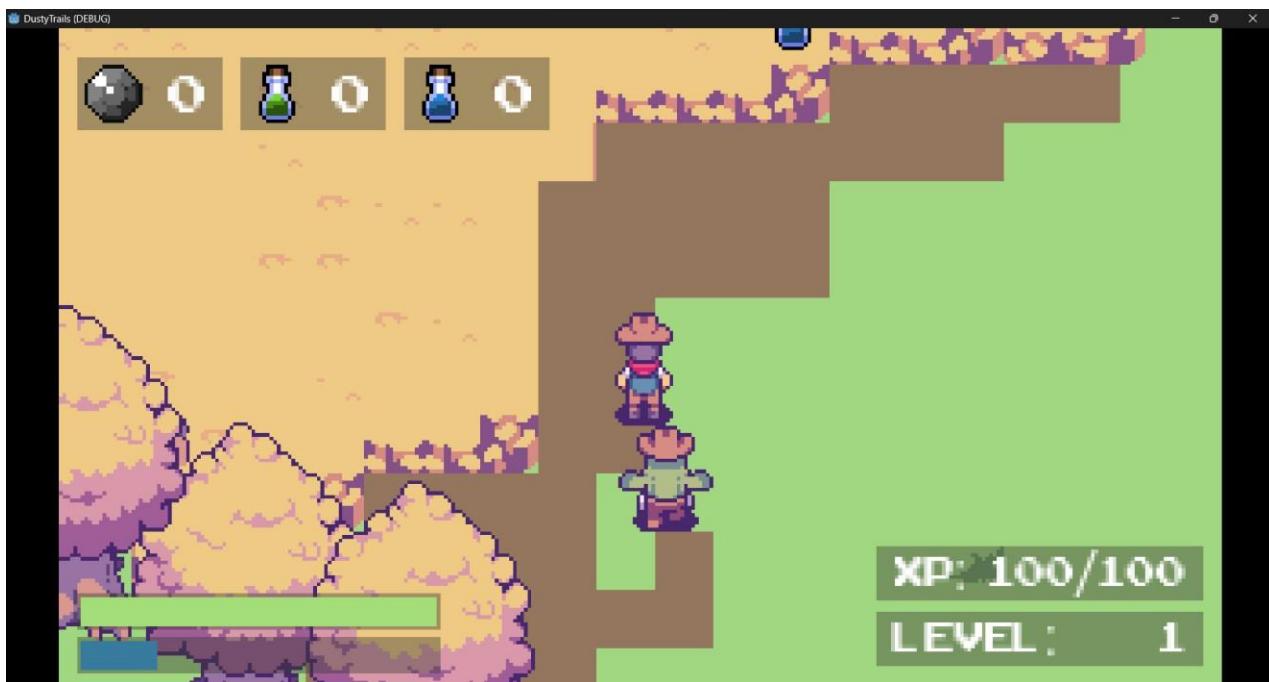
```
### Enemy.gd

# older code

# ----- Movement & Direction -----
func _on_timer_timeout():
    # Calculate the distance of the player's relative position to the enemy's
    # position
    var player_distance = player.position - position
    #turn towards player so that it can attack
    if player_distance.length() <= 20:
        new_direction = player_distance.normalized()
        sync_new_direction()
        direction = Vector2.ZERO
    #chase/move towards player to attack them
    elif player_distance.length() <= 100 and timer == 0:
        direction = player_distance.normalized()
        sync_new_direction()
    #random roam radius
    elif timer == 0:
        #this will generate a random direction value
        var random_direction = rng.randf()
        #This direction is obtained by rotating Vector2.DOWN by a random
        #angle between 0 and 2π radians (0 to 360°).
        if random_direction < 0.05:
            #enemy stops
            direction = Vector2.ZERO
        elif random_direction < 0.1:
            #enemy moves
            direction = Vector2.DOWN.rotated(rng.randf() * 2 * PI)
        sync_new_direction()
```

Remember, your enemy will attack in the direction of your player's last known location when they start attacking, so it is natural if there is a delay in them turning towards your player's new location when attacking. For example, if you were on their left when they started attacking and then you suddenly ran to the right, they will finish their attack animation towards the left first before redirecting to the right.

If you run your scene now you will see that your enemy character idles and animates according to its current direction.



You might notice another issue here: our side animation never plays. This is because our existing `returned_direction()` function checks the y-direction first and then the x-direction. This means that if there's any y-movement, it will always prioritize the up/down animations over the side animations. To fix this, we should prioritize the x-direction when it's dominant:

```
### Enemy.gd

# older code

# ----- Movement & Direction -----
# Animation Direction
func returned_direction(direction : Vector2):
    var normalized_direction = direction.normalized()
    var default_return = "side"
    if abs(normalized_direction.x) > abs(normalized_direction.y):
        if normalized_direction.x > 0:
            #(right)
            $AnimatedSprite2D.flip_h = false
        return "side"
```

```

else:
    #flip the animation for reusability (left)
    $AnimatedSprite2D.flip_h = true
    return "side"
elif normalized_direction.y > 0:
    return "down"
elif normalized_direction.y < 0:
    return "up"
#default value is empty
return default_return

```

Now our enemy will play their side animations.



This is it for now, as we will implement the attacking animations in the next few parts. In Part 11, we will add our Enemy Spawner scene so that we don't have to manually instance x amount of enemies in our Scene. Remember to save your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 11: SPAWNING ENEMY AI

Imagine how much work it would be to manually add 50+ enemies to our scene. This not only looks cluttered, but it also means that we will only have 50 enemy's in our scene, and if we kill them all, they'd be gone until we reload the game. To fix this little conundrum, we can create an Enemy Spawner that will spawn our enemies at random locations throughout the map and at a constant value, meaning we will never have more or fewer enemies than we defined. Each time an enemy is removed by us killing them, the spawner will spawn a new one, and so forth.

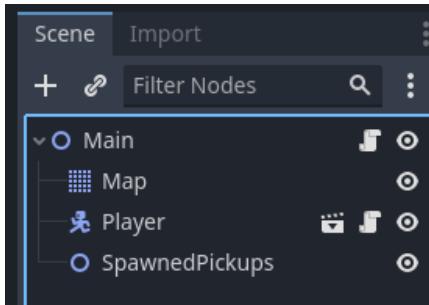
We'll also make our spawner flexible enough so that we can change the spawn area and max enemies in the Inspector panel. This means we can instance multiple spawners in our Main scene to have x amount of enemies spawn in different areas. Does this sound interesting? Well, what are you waiting for? Let's spawn some enemies!

WHAT YOU WILL LEARN IN THIS PART:

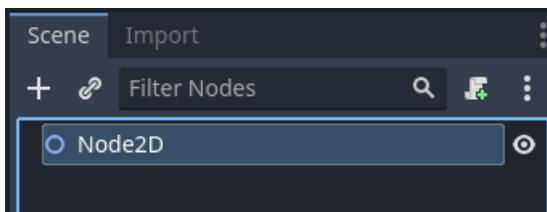
- How to work with the Timer node.
- How to create scene references.
- How to reference nodes in other scenes.
- How to reference TileMap properties.
- How to work with rectangle properties.

SPAWNER SETUP

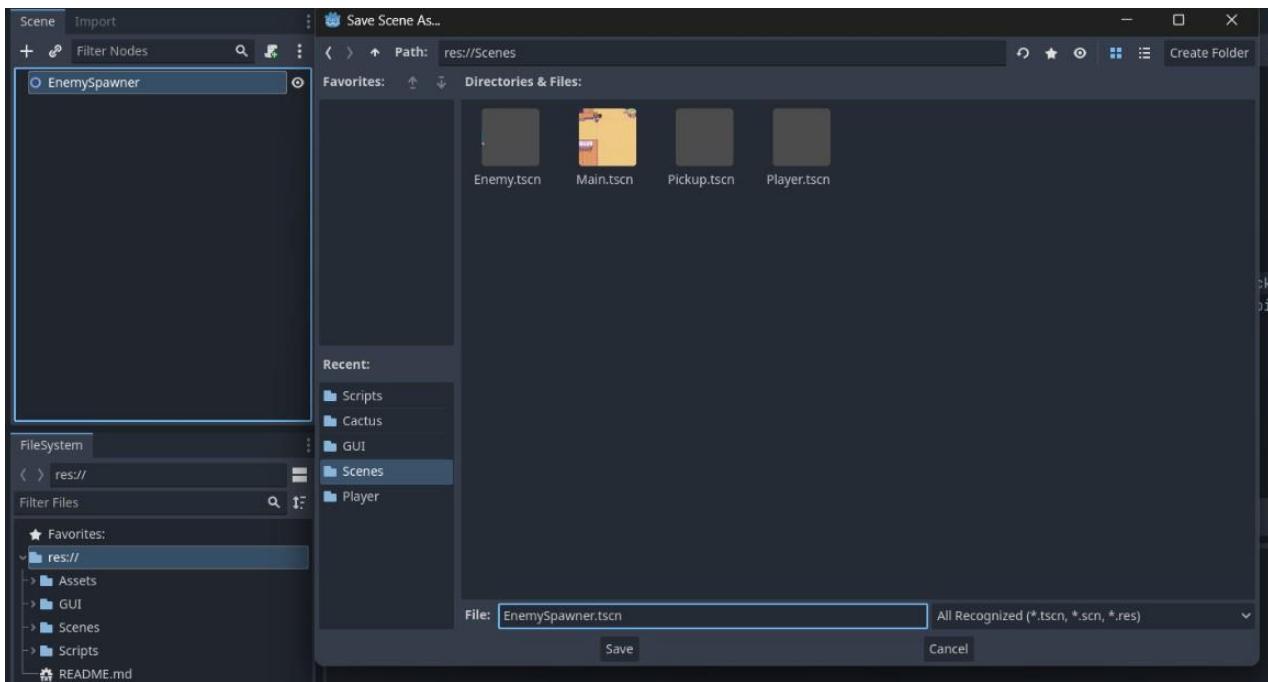
First, let's remove all the instances of the Enemy scene from our Main scene. We won't instance our enemy like this again since the spawner will take care of that.



Let's create a new Scene with a [Node2D](#) as its root. This will allow us to draw our enemy nodes onto our scene.

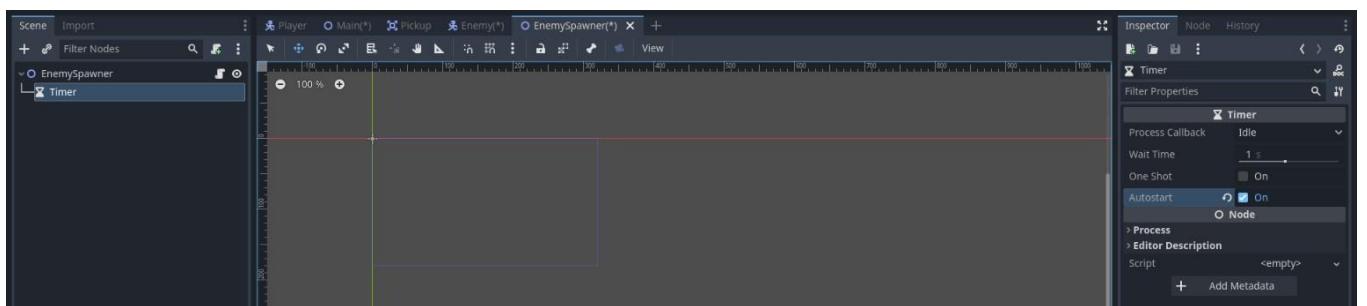
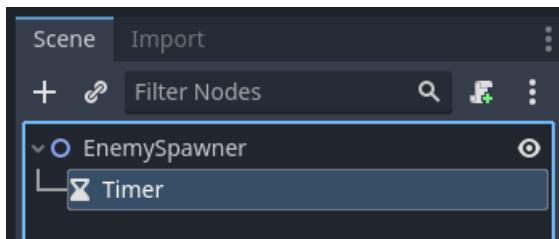


Rename this Node2D as “EnemySpawner” and save it under your Scenes folder.

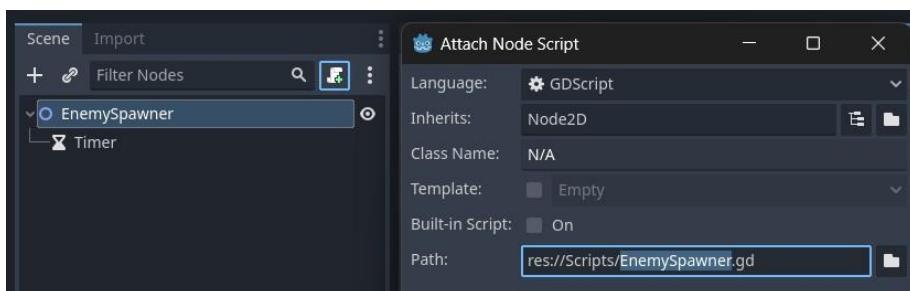


We need to add a Timer node to this scene so that we can spawn an enemy every second - unless we've already reached our max enemy amount. Enable AutoStart,

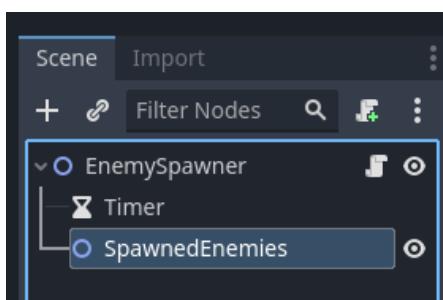
because we want this timer to start as soon as our game starts so that we can start spawning enemies.



Attach a script to your root node and save it underneath your Scripts folder.



Finally, let's add a new Node2D node that will we'll organize the spawned enemies underneath.



SPawning ENEMIES

In our script, we need to define a few variables. These variables will:

- Randomize our enemy's spawn position.
- Set and export the enemy max values.
- Set and export the enemy's current spawn count.
- Set a reference to our Map's tilemap property so that our enemy doesn't spawn on certain layers (for example, our water and foliage).

```
###EnemySpawner.gd

extends Node2D

# Node refs
@onready var spawned_enemies = $SpawnedEnemies
@onready var tilemap = get_tree().root.get_node("Main/Map")

# Enemy stats
@export var max_enemies = 20
var enemy_count = 0
var rng = RandomNumberGenerator.new()
```

In our Global script, we'll also load a reference to our Enemy scene.

```
### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
```

To spawn our enemies, we need to create a function that will spawn the existing enemies at the start of the game, and the rest of the enemies when the timer times out. We can call our function `spawn_enemy()`. In this function, we will create an instance of the reference to our enemy scene. When we create an [instance](#) of our scene it saves

this as a resource without loading it from the disk each time we call it. This saves space and time.

```
###EnemySpawner.gd

# older code

# ----- Spawning -----
func spawn_enemy():
    var enemy = Global.enemy_scene.instantiate()
```

Now that we have an instance of our Enemy scene, we can add our enemy as a child of our SpawneEnemies node hierarchy via the [add_child\(\)](#) method. In simple terms, we're adding enemies from our Enemy scene to our spawner's scene tree.

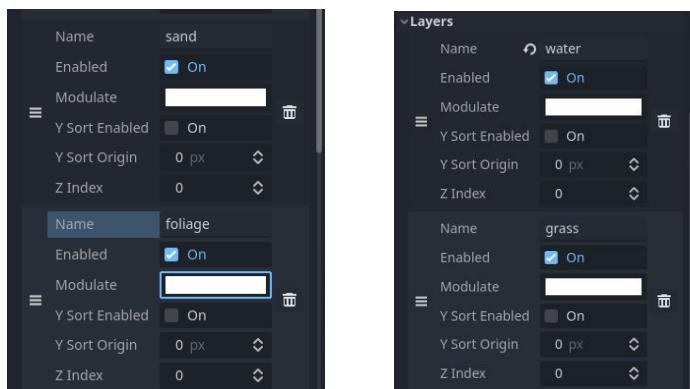
```
###EnemySpawner.gd

# older code

# ----- Spawning -----
func spawn_enemy():
    var enemy = Global.enemy_scene.instantiate()
    spawned_enemies.add_child(enemy)
```

If you think about it logically, you know that you cannot just spawn an enemy in any position. What if our game spawns it under a building? Or in the water? We won't be able to get to the enemy, so to prevent this issue from happening, we have to create a function that will define the valid spawn positions for our enemy. We've done this before in our Pickups spawner. Now depending on the layers that you added to your Tilemap, the next part might be different on your side than it is on mine.

In the part where we added our Map, we created the following layers:



As you already know, our layers each have an ID assigned to each of them. We start counting at 0, so that means that water == 0, grass == 1, sand == 2, and foliage ==3. We only want our enemy to spawn on our grass(1) or sand(2) layers. Anywhere where our other layers (0, 3) are present, we don't want them to spawn.

Since we'll be referencing these layers in both our Pickup spawning functionality and Enemy spawning, let's redefine the layer constants from our Main script in our Global script. We'll then also need to reference these constants correctly in our Main script.

```
### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }

# TileMap layers
const WATER_LAYER = 0
const GRASS_LAYER = 1
const SAND_LAYER = 2
const FOLIAGE_LAYER = 3
const EXTERIOR_1_LAYER = 4
const EXTERIOR_2_LAYER = 5
```

```
### Main.gd

# older code

# ----- Pickup spawning -----
# Valid pickup spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)

    # Check if there's a tile on the water, foliage, or exterior layers
    if map.get_cell_source_id(Global.WATER_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.FOLIAGE_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.EXTERIOR_1_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.EXTERIOR_2_LAYER, cell_coords) != -1
```

```

map.get_cell_source_id(Global.EXTERIOR_1_LAYER, cell_coords) != -1 ||
map.get_cell_source_id(Global.EXTERIOR_2_LAYER, cell_coords) != -1:
    return false

# Check if there's a tile on the grass or sand layers
if map.get_cell_source_id(Global.GRASS_LAYER, cell_coords) != -1 ||
map.get_cell_source_id(Global.SAND_LAYER, cell_coords) != -1:
    return true

return false

```

Now in our `EnemySpawner` script, let's copy and paste the entire `is_valid_spawn_location()` function from our `Main` script. Replace the variable `map` with `tilemap`.

```

###EnemySpawner.gd

# ----- Spawning -----
# older code

# Valid spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)
    # Check if there's a tile on the water, foliage, or exterior layers
    if tilemap.get_cell_source_id(Global.WATER_LAYER, cell_coords) != -1 ||
    tilemap.get_cell_source_id(Global.FOLIAGE_LAYER, cell_coords) != -1 ||
    tilemap.get_cell_source_id(Global.EXTERIOR_1_LAYER, cell_coords) != -1 ||
    tilemap.get_cell_source_id(Global.EXTERIOR_2_LAYER, cell_coords) != -1:
        return false
    # Check if there's a tile on the grass or sand layers
    if tilemap.get_cell_source_id(Global.GRASS_LAYER, cell_coords) != -1 ||
    tilemap.get_cell_source_id(Global.SAND_LAYER, cell_coords) != -1:
        return true
    return false

```

Now in our `spawn_enemy` function, we need to randomly select a position on the map. We'll then need to check if that position is a valid spawn location using the `is_valid_spawn_location` function. If it's valid, we'll spawn the enemy at that position. If not, we'll try another random position. This once again works similarly to what we did when we spawned our pickups.

```

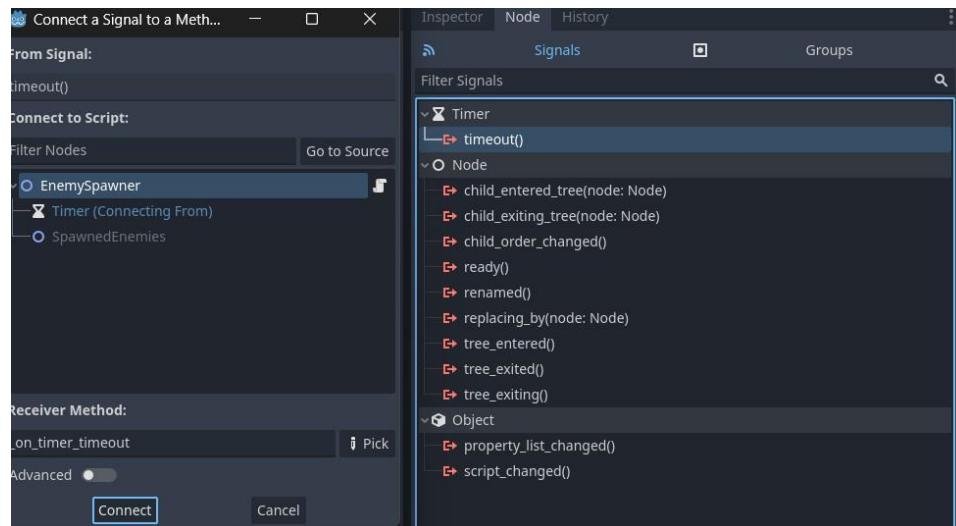
####EnemySpawner.gd

# ----- Spawning -----
func spawn_enemy():
    var attempts = 0
    var max_attempts = 100 # Maximum number of attempts to find a valid location
    var spawned = false

    while not spawned and attempts < max_attempts:
        # Randomly select a position on the map
        var random_position = Vector2(
            rng.randi() % tilemap.get_used_rect().size.x,
            rng.randi() % tilemap.get_used_rect().size.y
        )
        # Check if the position is a valid spawn location
        if is_valid_spawn_location(Global.GRASS_LAYER, random_position) ||
           is_valid_spawn_location(Global.SAND_LAYER, random_position):
            # Spawn enemy
            var enemy = Global.enemy_scene.instantiate()
            enemy.position = tilemap.map_to_local(random_position) + Vector2(16,
                16) / 2
            spawned_enemies.add_child(enemy)
            spawned = true
        else:
            attempts += 1
    if attempts == max_attempts:
        print("Warning: Could not find a valid spawn location after",
              max_attempts, "attempts.")

```

Finally, let's connect our `timeout()` signal from our Timer to our EnemySpawner script.

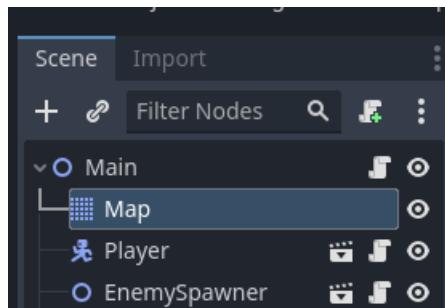


If our enemy count is not bigger than our max enemies, we will call our `spawn_enemies` function. This will spawn an enemy every 1 second until the max amount of enemies allowed have been spawned.

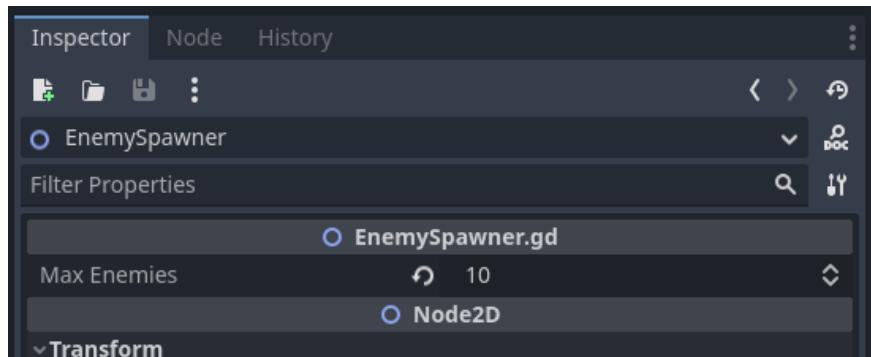
```
###EnemySpawner.gd
```

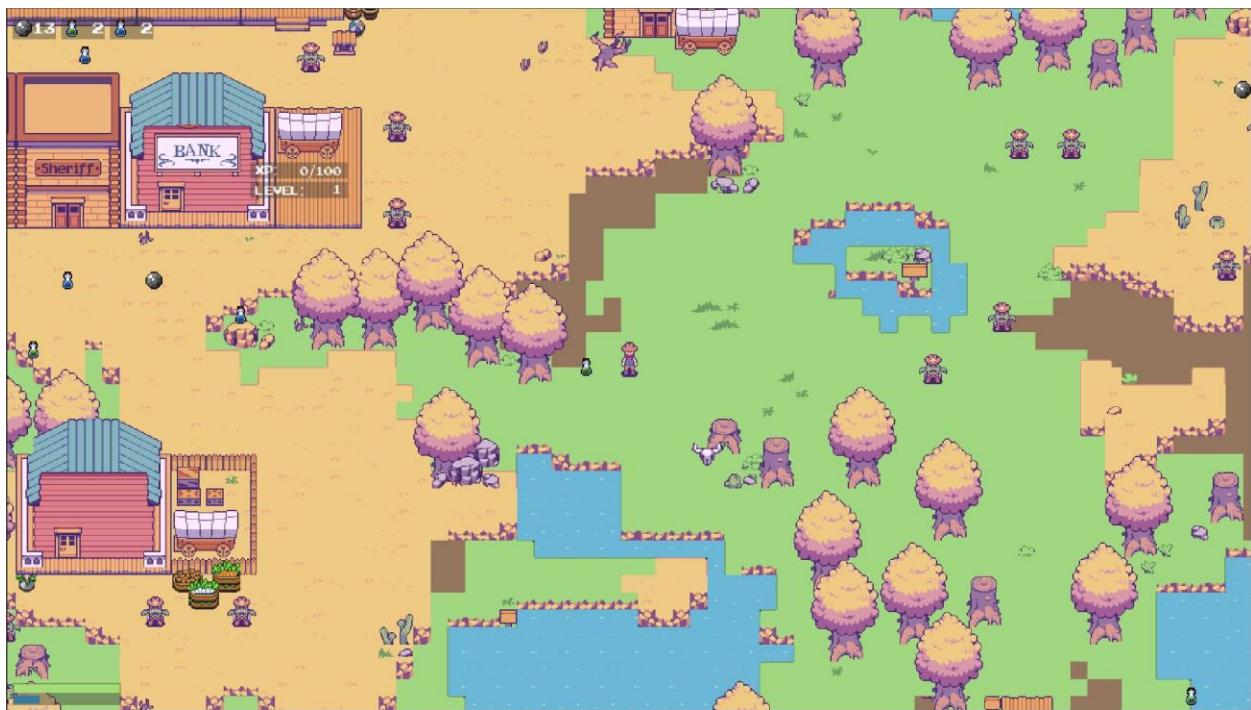
```
# ----- Spawning -----
# Spawn enemy
func _on_timer_timeout():
    if enemy_count < max_enemies:
        spawn_enemy()
        enemy_count = enemy_count + 1
```

Now if you create an instance of your EnemySpawner scene in your Main scene (make sure you have no Enemy scene in your Main scene), and you run your game, you will notice that the enemies spawn.



*If you don't want enemies in your scene, just set your spawner values to 0 in the Inspector panel.



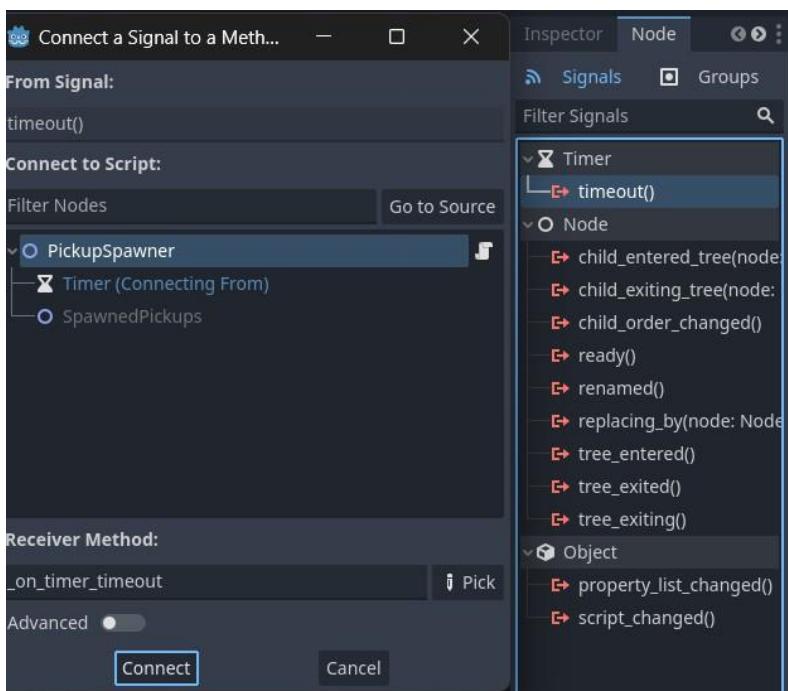
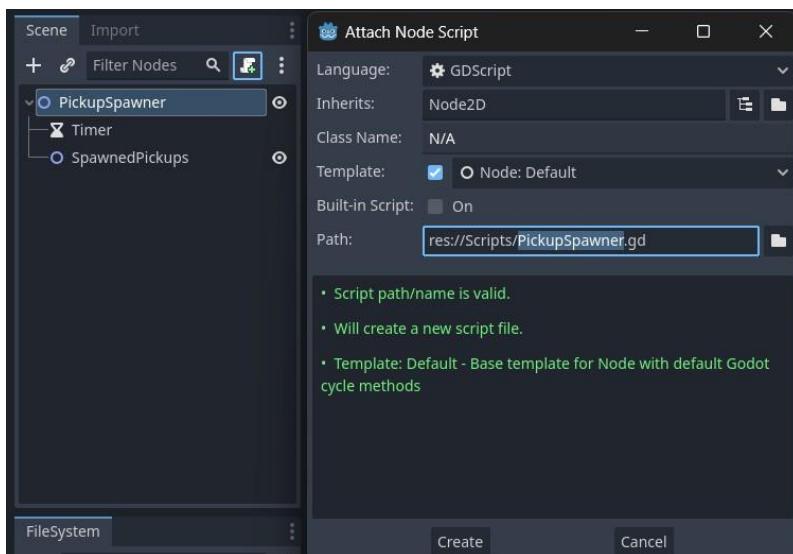


Ensure that you TileMap node's transform properties are set to (0,0) – otherwise your enemies will spawn with an offset!



PICKUP SPAWNER

Whilst we're at it, let's create a new scene that will contain our PickupSpawner. This will make our project more dynamic and reusable. Recreate the steps that you took for your EnemySpawner (Create a new scene, add nodes, connect script, connect timeout signal). Please don't add a Timer node to this scene



Now, in your PickupSpawner.gd script, copy over the code that you added in your Main script into your newly created script.

```
### PickupSpawner.gd

extends Node2D

# Node refs
@onready var map = get_tree().root.get_node("Main/Map")
@onready var spawned_pickups = $SpawnedPickups

var rng = RandomNumberGenerator.new()

func _ready():
    # Spawn between 5 and 10 pickups
    var spawn_pickup_amount = rng.randf_range(5, 10)
    spawn_pickups(spawn_pickup_amount)

# ----- Pickup spawning -----
# Valid pickup spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)
    # Check if there's a tile on the water, foliage, or exterior layers
    if map.get_cell_source_id(Global.WATER_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.FOLIAGE_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.EXTERIOR_1_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.EXTERIOR_2_LAYER, cell_coords) != -1:
        return false
    # Check if there's a tile on the grass or sand layers
    if map.get_cell_source_id(Global.GRASS_LAYER, cell_coords) != -1 ||
       map.get_cell_source_id(Global.SAND_LAYER, cell_coords) != -1:
        return true
    return false

# Spawn pickup
func spawn_pickups(amount):
    var spawned = 0
    var attempts = 0
    var max_attempts = 1000 # Arbitrary number, adjust as needed

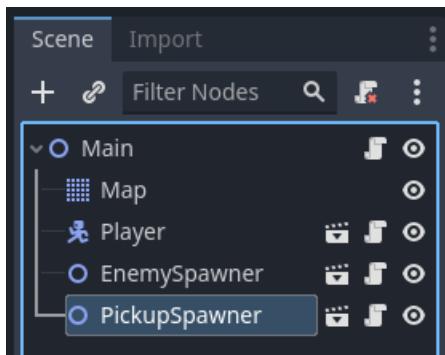
    while spawned < amount and attempts < max_attempts:
        attempts += 1
        var random_position = Vector2(randi() % map.get_used_rect().size.x,
                                       randi() % map.get_used_rect().size.y)
```

```

var layer = randi() % 2
if is_valid_spawn_location(layer, random_position):
    var pickup_instance = Global.pickups_scene.instantiate()
    pickup_instance.item = Global.Pickups.values()[randi() % 3]
    pickup_instance.position = map.map_to_local(random_position) +
        Vector2(16, 16) / 2
    spawned_pickups.add_child(pickup_instance)
    spawned += 1

```

Then, in your Main scene, delete the SpawnedPickups node, and instance the PickupSpawner instead. Your Main script should look like the one below.



```

### Main.gd

extends Node2D

```

Now your pickups and enemies should spawn when you run the scene! The enemy should still roam around and chase the player if you come close to them.



Next up, we're going to add the functionality for our player to shoot and deal damage to our enemy. Remember to save your game project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 12: PLAYER SHOOTING & DAMAGE

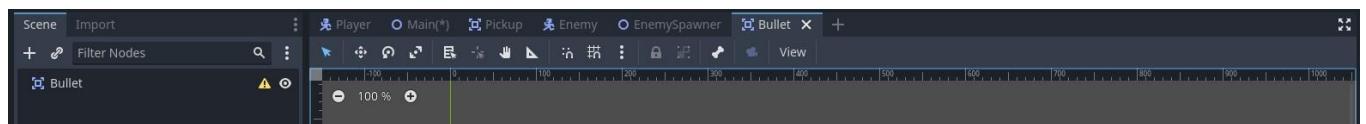
What good is an enemy if you can't shoot them? In this part, we're going to create bullets that our player can shoot in the direction that they're facing. Then we're going to add an animation that will show that our enemy is being damaged if our bullet hits them. If the bullets hit them enough for their health value to run out, they will also die. Later on, we will level up our player's XP and add loot to the enemy on their death. But for now, let's buckle up because this part is going to be a long one!

WHAT YOU WILL LEARN IN THIS PART:

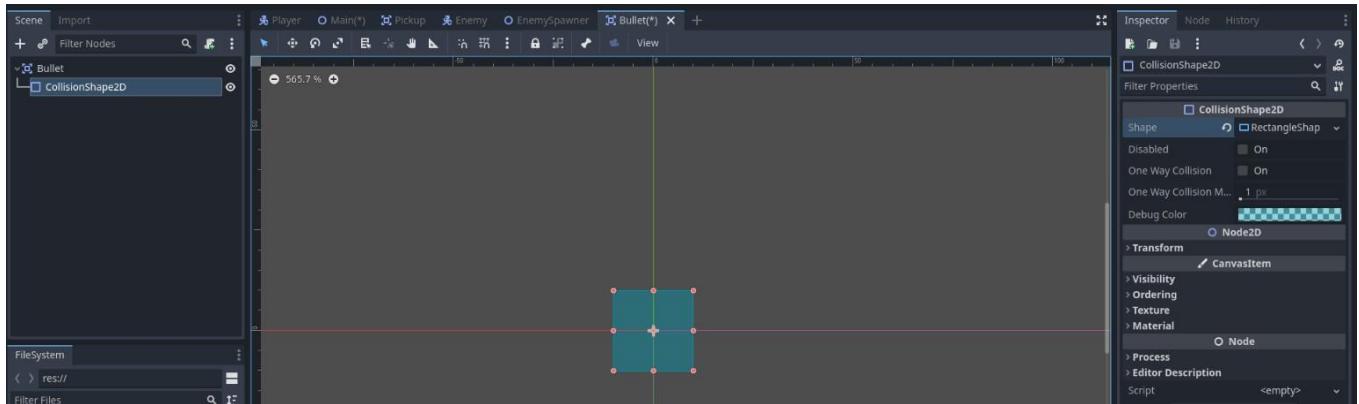
- How to use the AnimationPlayer node.
- How to use the RayCast2D node.
- How to work with modulate values.
- How to work with the Time class.

SPAWNABLE BULLET

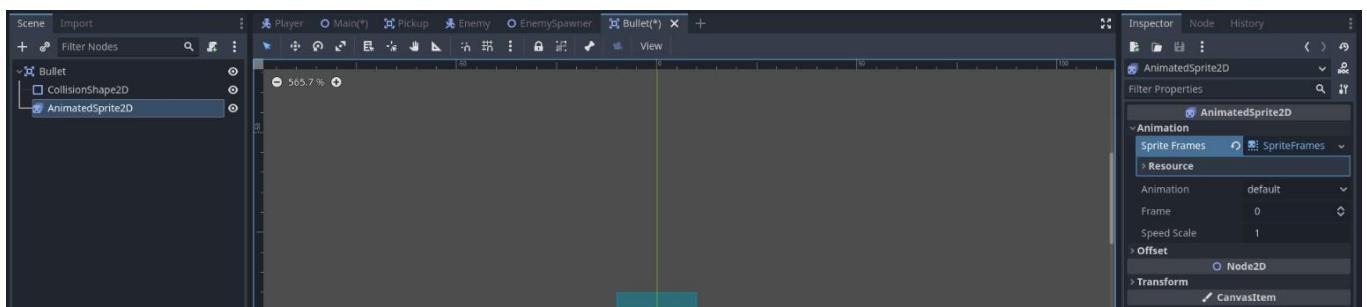
Let's start by creating the bullets that will spawn when our player shoots or attacks. This scene will be similar to our Pickups scene's structure. Create a new scene with an Area2D node as its root. Rename this node as Bullet and save it under your Scenes folder.



It has a warning message because it has no shape. Let's fix this by adding a CollisionShape2D node to it with a RectangleShape2D as its shape.

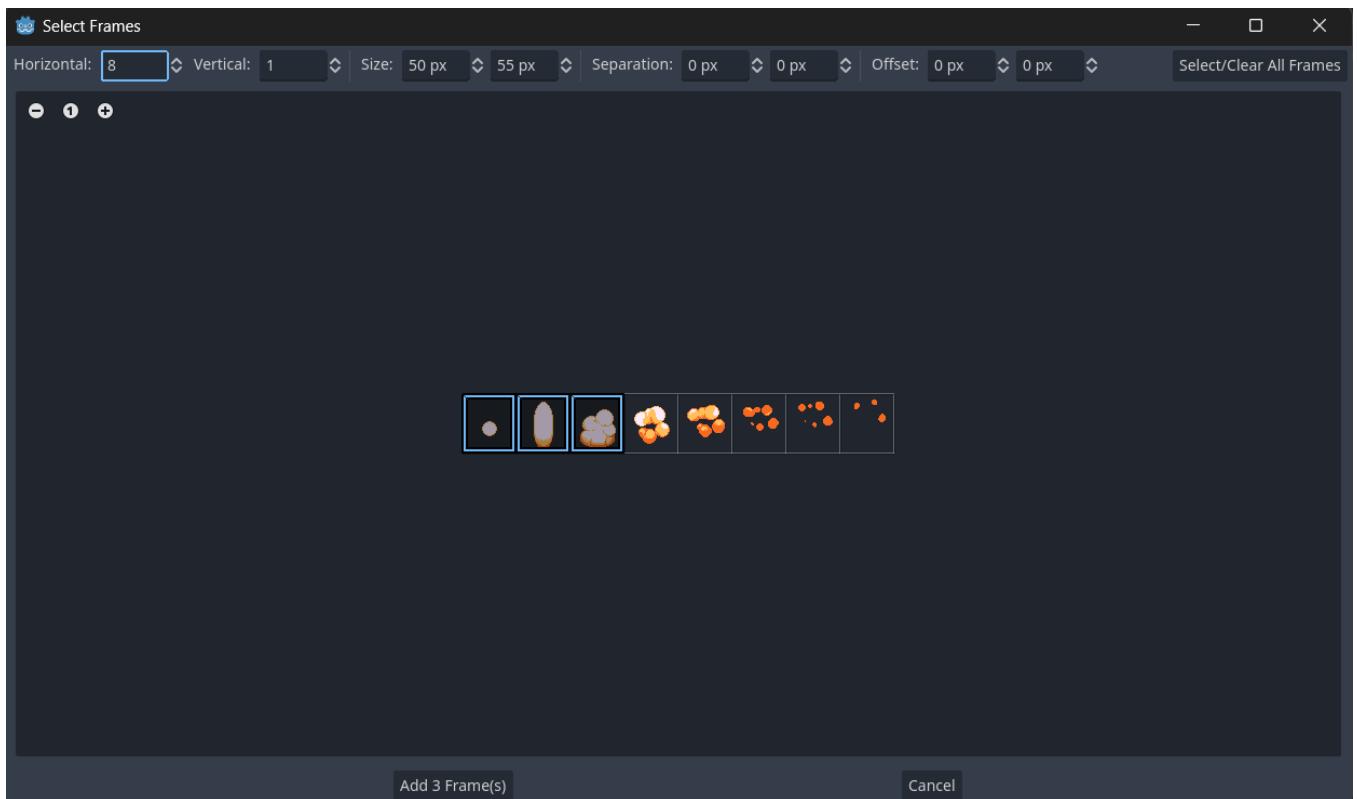


We also need to see our node/bullet. Our bullet will have an impact animation, so we need to add an AnimatedSprite2D node.

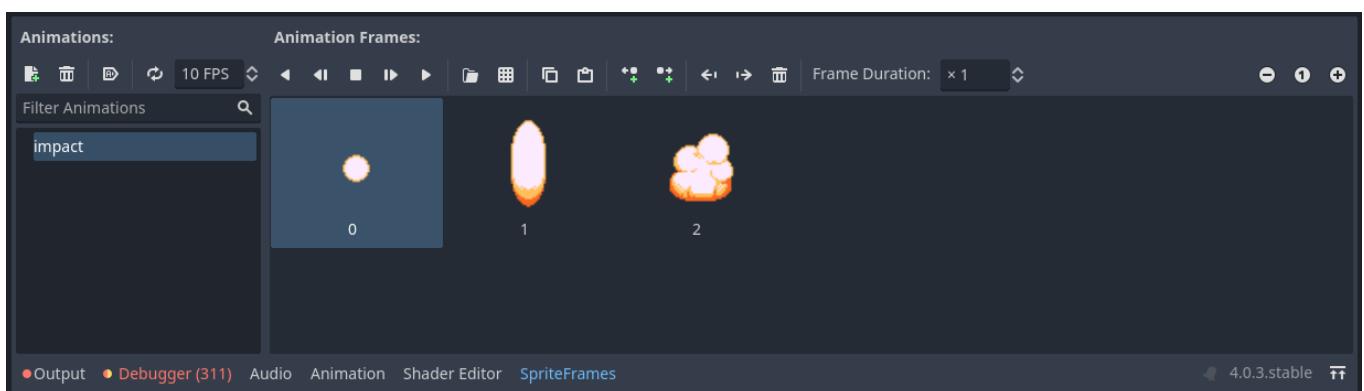


Add a new SpriteFrames resource to it in the Inspector panel, and in your SpriteFrames pane below, let's add a new animation called "impact". The spritesheet we will use for our bullet can be found under Assets > FX > Death Explosion.png.

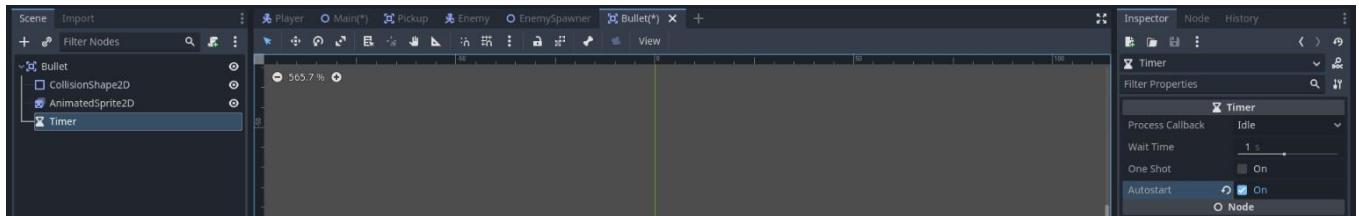
Horizontally, we count 8 frames, and vertically we count 1 frame, so let's change our values accordingly to crop out our animation frames. Also, select only the first three frames for your animation.



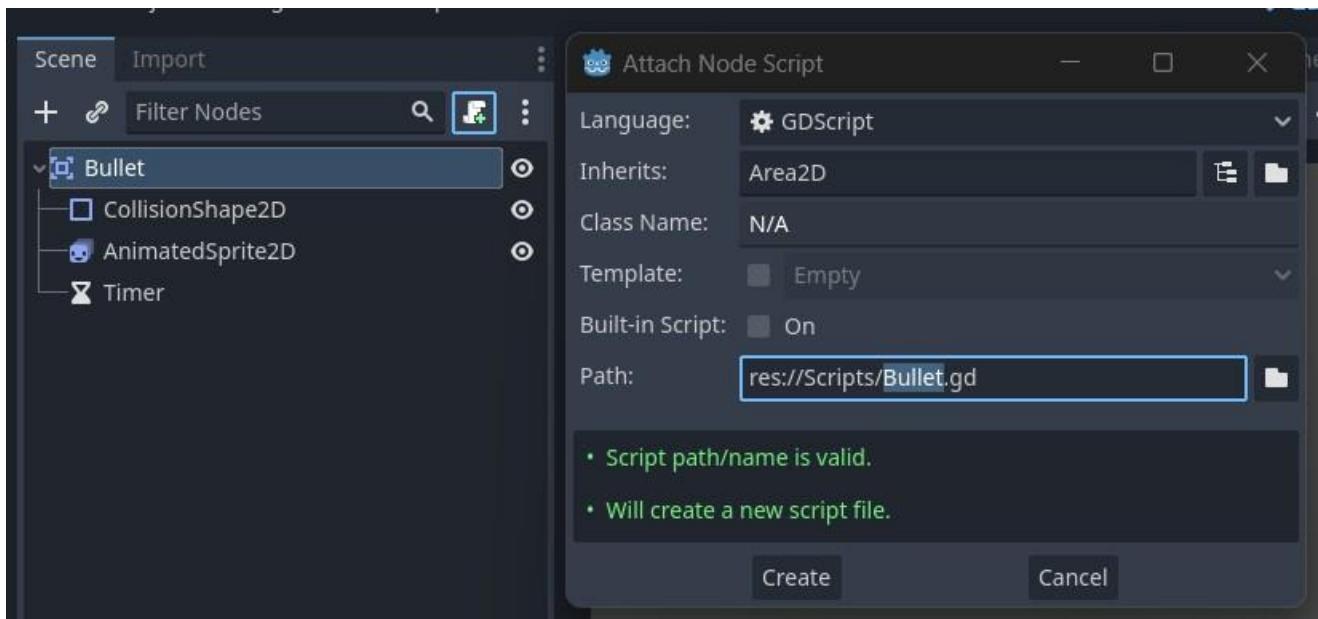
Change the FPS value to 8 and turn its looping value to off.



We also need to add a Timer node to our scene with autoplay enabled. This timer will emit its `timeout()` signal when the bullet needs to "self-destruct" after it's not hit or impacted anything.



Finally, add a script to your scene and save it under your Scripts folder.



Okay, now we can start talking about what our Bullet scene needs to do. We first need to define a few variables which will set the bullet's speed, direction, and damage. We also need to reference our Tilemap node again so that we can ignore the collision with certain layers, such as our water, which has collisions added to it, but it shouldn't stop the bullet.

```
### Bullet.gd
extends Area2D

# Bullet variables
@onready var tilemap = get_tree().root.get_node("Main/Map")
var speed = 80
var direction : Vector2
var damage
```

Then we need to calculate the position of our bullet after it's been shot. This position should be re-calculated for each frame in case our player moves. Therefore we can calculate it in our built-in *process()* function so that it updates its direction during each frame movement. We can calculate this position by adding its current position by its direction times its speed at the current frame captured as delta.

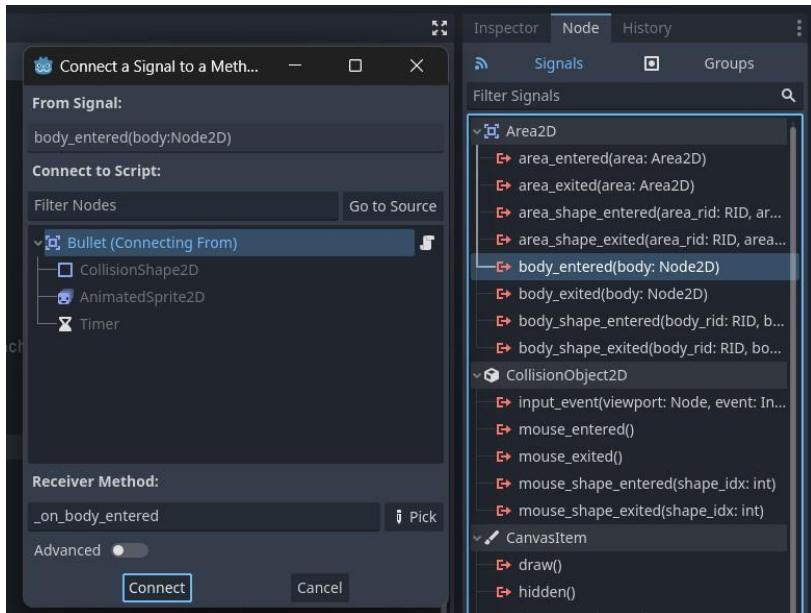
```
### Bullet.gd
extends Area2D

# Bullet variables
@onready var tilemap = get_tree().root.get_node("Main/Map")
var speed = 80
var direction : Vector2
var damage

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction
```

When we did our Pickups, we connected the Area2D's *body_entered()* signal to our scene to add the pickups to our player's inventory. We want to do the same for our Bullet scene, but this time we will check the body entered is our enemy so that we can injure them. We also want to check if the body entering our Bullets collision is the Player or "Water" layer from our TileMap because we want to ignore these collisions.

Connect the *body_entered()* signal to your Bullet script. You will see that it creates a new '*func _on_body_entered(body):*' function at the end of your script. Inside this new function, let's accomplish the objectives we set above.



```
### Bullet.gd
extends Area2D

# Bullet variables
@onready var tilemap = get_tree().root.get_node("Main/Map")
var speed = 80
var direction : Vector2
var damage

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Player
    if body.name == "Player":
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
        if tilemap.get_layer_name(Global.WATER_LAYER):
            return
    # If the bullets hit an enemy, damage them
    if body.name.find("Enemy") >= 0:
        #todo: add damage/hit function to enemy scene
        pass
```

We cannot damage the enemy yet because they do not have any health variables or functions set up. We will do that in a minute, but for now, let's stop the bullet from moving and change its animation to "impact" if it does not hit anything. We're doing this because we don't want a random bullet just floating around in our scene.

```
### Bullet.gd
extends Area2D

# Node refs
@onready var tilemap = get_tree().root.get_node("Main/Map")
@onready var animated_sprite = $AnimatedSprite2D

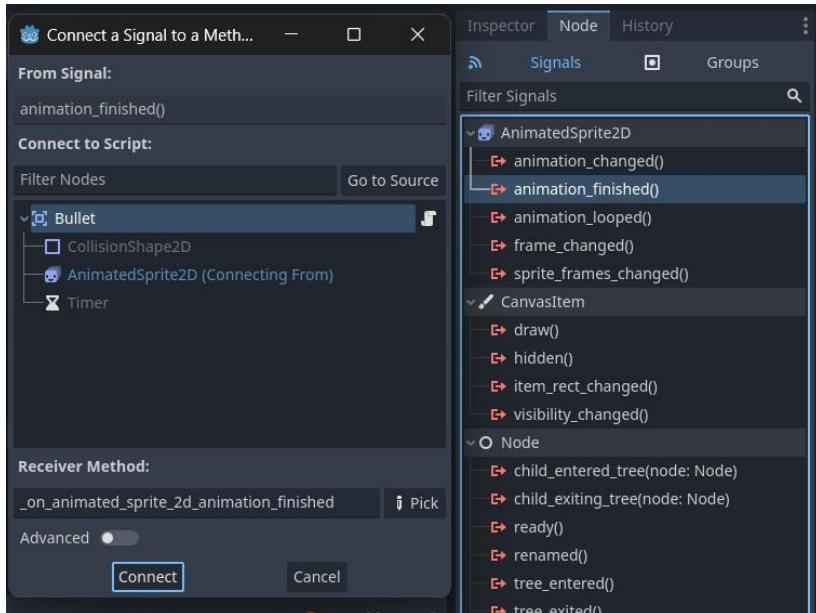
# older code

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Player
    if body.name == "Player":
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
        if tilemap.get_layer_name(Global.WATER_LAYER):
            return
    # If the bullets hit an enemy, damage them
    #todo: add damage/hit function to enemy scene

    # Stop the movement and explode
    direction = Vector2.ZERO
    animated_sprite.play("impact")
```

We also need to delete the bullet from the scene after it stopped moving and stopped playing the "impact" animation. We've done this before in our Pickups scene, so let's go ahead and connect our AnimationSprite2D *animation_finished* signal to our Bullet script.



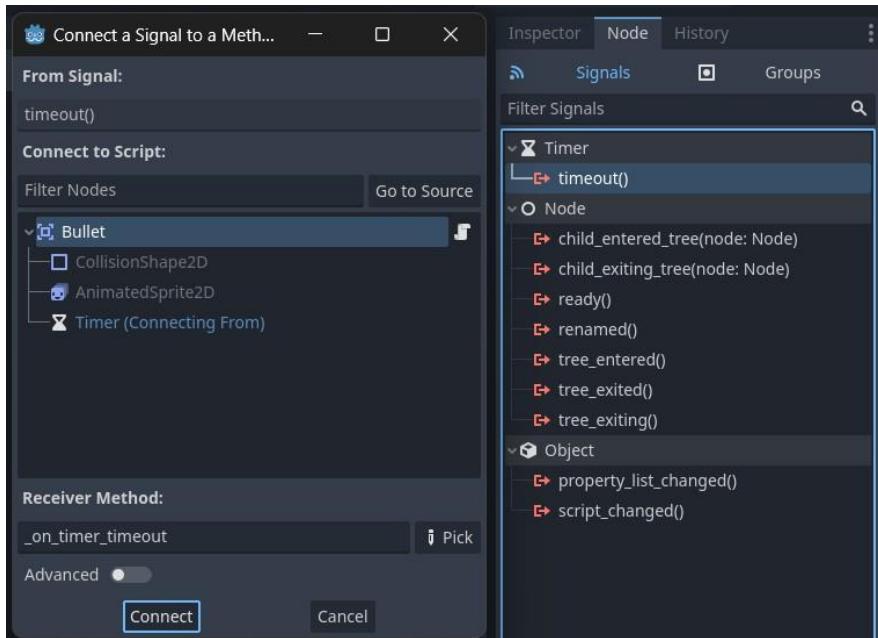
```
### Bullet.gd

# older code

# ----- Bullet -----
# older code

# Remove
func _on_animated_sprite_2d_animation_finished():
    if animated_sprite.animation == "impact":
        get_tree().queue_delete(self)
```

Finally, in your Bullet scene, connect your Timer node's `timeout()` signal to your Bullet script. We want this function to also play the "impact" animation after it's not hit anything after two seconds because playing this animation will trigger the `animation_finished` signal, which will delete the bullet from our scene.



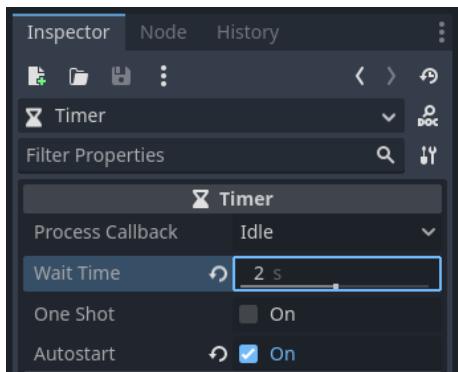
```
### Bullet.gd

# older code

# ----- Bullet -----
# older code

# Self-destruct
func _on_timer_timeout():
    animated_sprite.play("impact")
```

Don't forget to change your Timer node's wait time to 2 so that it plays the animation after 2 seconds and not 1.



We also want our Bullet to “explode” when it hits buildings, trees, and other objects. So, on the layers on your TileMap where you added buildings, trees, or objects such as crates or décor, allow your bullets to explode on impact.

```
#### Bullet.gd
extends Area2D

# Node refs
@onready var tilemap = get_tree().root.get_node("%s/Map" % Global.current_scene_name)
@onready var animated_sprite = $AnimatedSprite2D

# Bullet variables
var speed = 80
var direction : Vector2
var damage

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Player
    if body.is_in_group("player"):
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
        if tilemap.get_layer_name(Global.WATER_LAYER):
            return
        # trees, buildings and objects
        if tilemap.get_layer_name(Global.FOLIAGE_LAYER) or
        tilemap.get_layer_name(Global.EXTERIOR_1_LAYER) or
        tilemap.get_layer_name(Global.EXTERIOR_2_LAYER):
            animated_sprite.play("impact")

    # If the bullets hit an enemy, damage them
    #todo: add damage/hit function to enemy scene

    # Stop the movement and explode
    direction = Vector2.ZERO
    animated_sprite.play("impact")
```

```

# Remove
func _on_animated_sprite_2d_animation_finished():
    if animated_sprite.animation == "impact":
        get_tree().queue_delete(self)

# Self-destruct
func _on_timer_timeout():
    animated_sprite.play("impact")

```

PLAYER SHOOTING

For now, we are done with our Bullet scene since we have our bullet movement and animation setup, plus the ability to remove the bullet from a scene. We need to now go back to our Player scene so that we can fire off bullets via our *ui_attack* input.

Let's preload our Bullet script in our Global script.

```

### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
@onready var bullet_scene = preload("res://Scenes/Bullet.tscn")

```

In your Player script, let's define some variables that would set the bullet damage, reload time, and the bullet fired time.

```

### Player.gd

# older code

# Bullet & attack variables
var bullet_damage = 30
var bullet_reload_time = 1000
var bullet_fired_time = 0.5

```

Now we can change our *ui_attack* input to spawn bullets, calculate the reload, and remove ammo. When we spawn these bullets, we need to take into consideration the time the bullet was fired vs. the reload time. We want our player to take a 1000-ms break before being able to fire off the next round. To get the time in Godot, we can use the [Time object](#). The Time singleton allows converting time between various formats

and also getting time information from the system. We will use the method `.get_ticks_msec()` for precise time calculation.

First, we will check if our current time is bigger or equal to our `bullet_fired_time` AND if we have ammo to shoot (make sure you assign some ammo to your player first).

If it is bigger, that means we can fire off a round. We will then return our `is_attacking` boolean as true and play our shooting animation. We will update our `bullet_fired_time` by adding our `reload_time` to our current time. This means our player will have a 1000-ms pause before they fire off the next round. Finally, we will update our ammo pickup amount.

```
### Player.gd

# older code

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #checks the current time as the amount of time passed in milliseconds
        #since the engine started
        var now = Time.get_ticks_msec()
        #check if player can shoot if the reload time has passed and we have ammo
        if now >= bullet_fired_time and ammo_pickup > 0:
            #shooting anim
            is_attacking = true
            var animation  = "attack_" + returned_direction(new_direction)
            animation_sprite.play(animation)
            #bullet fired time to current time
            bullet_fired_time = now + bullet_reload_time
            #reduce and signal ammo change
            ammo_pickup = ammo_pickup - 1
            ammo_pickups_updated.emit(ammo_pickup)

    # older code
```

We will spawn our bullet in our `_on_animated_sprite_2d_animation_finished` function, because only after the shooting animation has played do we want our bullet to be added to our Main scene.

We'll have to create another instance of our Bullet scene, where we will update its

damage, direction, and position as the direction the player was facing when they fired off the round, and the position 4-5 pixels in front of the player (you don't want the bullet to come from inside of your player, but instead from the gun's "barrel").

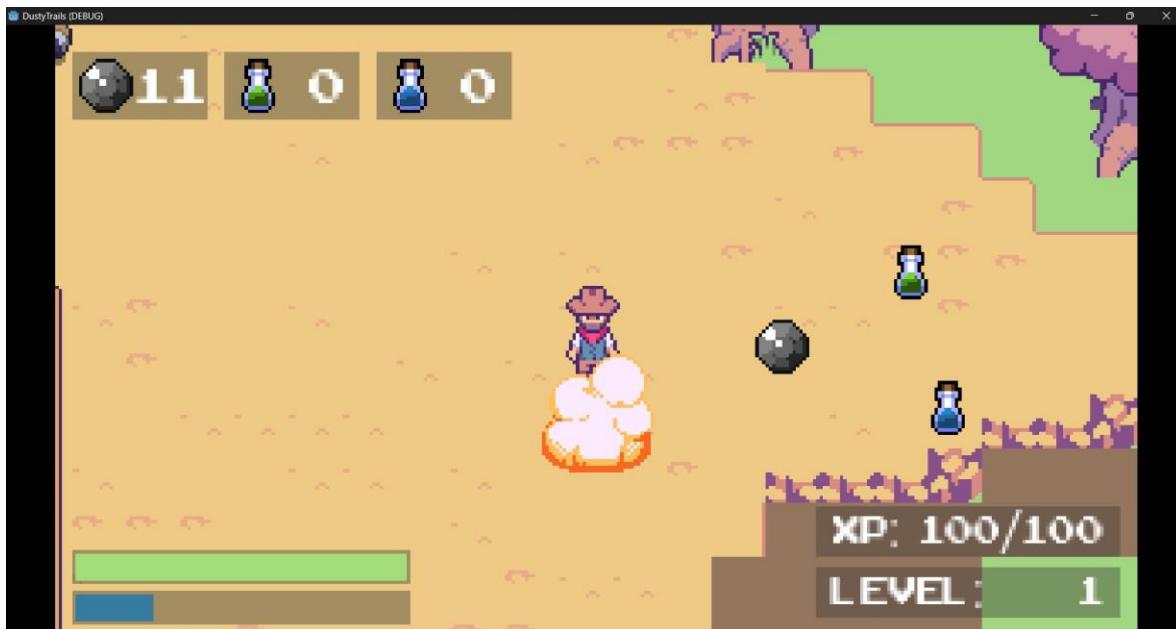
```
### Player.gd

# older code

# Reset Animation states
func _on_animated_sprite_2d_animation_finished():
    is_attacking = false

    # Instantiate Bullet
    if animation_sprite.animation.begins_with("attack_"):
        var bullet = Global.bullet_scene.instantiate()
        bullet.damage = bullet_damage
        bullet.direction = new_direction.normalized()
        # Place it 4-5 pixels away in front of the player to simulate it coming
        # from the guns barrel
        bullet.position = position + new_direction.normalized() * 4
        get_tree().root.get_node("Main").add_child(bullet)
```

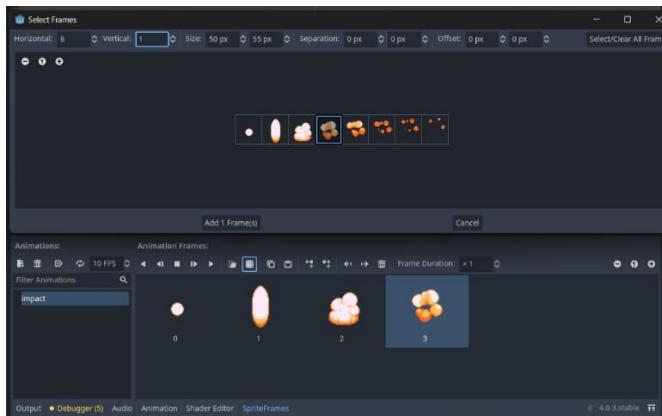
Now if you run your scene, a bullet should spawn after you press CTRL to fire off a bullet (ui_attack input action). It's a bit big for a bullet, so let's change its size!



In your Bullet scene, with your AnimatedSprite2D node selected, change its scale value from 1 to 0.4 underneath its Transform property.



Also, add that fourth frame of the bullet FX to your "impact" animation. Sorry for the inconvenience of adding it now, I just thought it looked better at the last minute!



Now if you run your scene, your bullet should be much smaller. It should also self-destruct after 2 seconds, or when it hits another collision body. Now we need to add the functionality for our enemy to be damaged by a bullet impact.



ENEMY DAMAGE

Back in our Bullet script, I added a `#todo` since we still do not have a damage function or health variables set up in our Enemy scene. We will do that now.

In your Enemy script, let's set up its health variables. Just like the player, we need variables to store its health, max health, and health regeneration, as well as a signal to fire off when it dies.

```
### Enemy.gd

extends CharacterBody2D

# Node refs
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D
```

```

# Enemy stats
@export var speed = 50
var direction : Vector2 # current direction
var new_direction : Vector2(0,1) # next direction
var animation
var is_attacking = false
var health = 100
var max_health = 100
var health_regen = 1

# Direction timer
var rng = RandomNumberGenerator.new()
var timer = 0

# Custom signals
signal death

```

We will calculate its health regeneration in its `_process(delta)` function since we want to calculate it at each frame. We've already calculated this in the Player script when we did its `updated_health` value.

```

### Enemy.gd

extends CharacterBody2D

# older code

----- Damage & Health -----
func _process(delta):
    #regenerates our enemy's health
    health = min(health + health_regen * delta, max_health)

```

Let's also go ahead and start creating our `damage` function. This will be the function that we call in our Player and Bullet scenes to damage the enemy upon bullet/attack impact. Before we do that, we need to also give the enemy some attack variables. We can just copy over the attack variables from our Player script.

```

### Enemy.gd

extends CharacterBody2D

```

```
# older code

# Bullet & attack variables
var bullet_damage = 30
var bullet_reload_time = 1000
var bullet_fired_time = 0.5
```

The damage function will decrease our Enemy's health by the bullet damage passed in by the Player or attacker. If their health is more than zero, the enemy will be damaged. If it's less than or equal to zero, the enemy will die.

```
### Enemy.gd

# older code

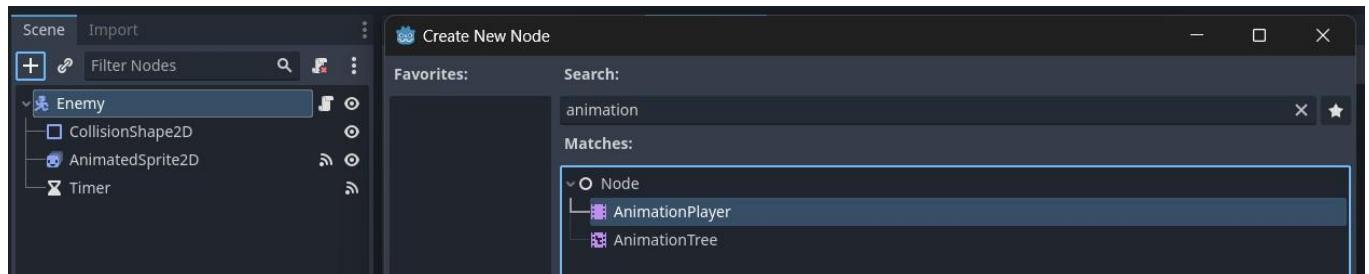
----- Damage & Health -----
func _process(delta):
    #regenerates our enemy's health
    health = min(health + health_regen * delta, max_health)

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        pass
    else:
        #death
        pass
```

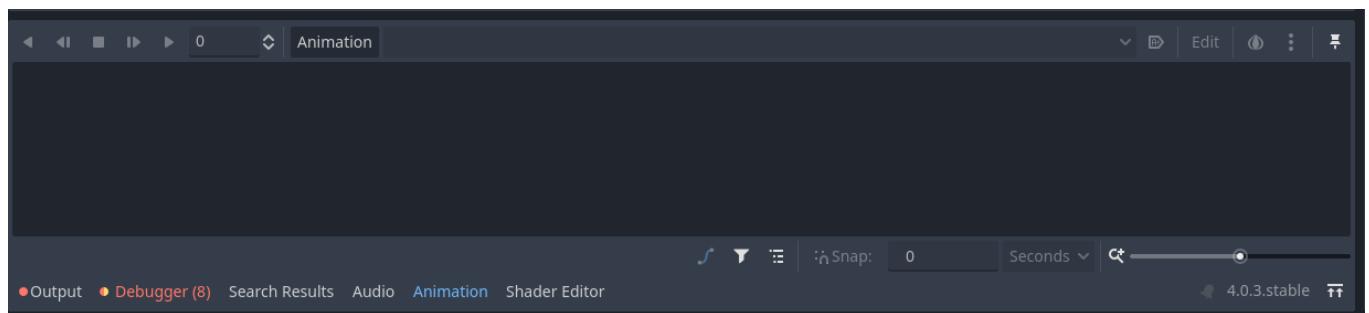
We will slowly make our way to completing our damage function, so let's get started by adding an animation that would indicate that our Enemy has been hit. For this, we can change the enemy's [modulate](#) value via the [AnimationPlayer](#) node. The modulate value refers to the node's color, so in simple terms, we will use the AnimationPlayer to briefly turn our enemy's color red so that we can see that they are damaged. You will use this AnimationPlayer node whenever you need to animate non-sprite items, such as labels or colors. Let's add this node to our Enemy scene.

When to use AnimatedSprite2D node vs. AnimationPlayer node?

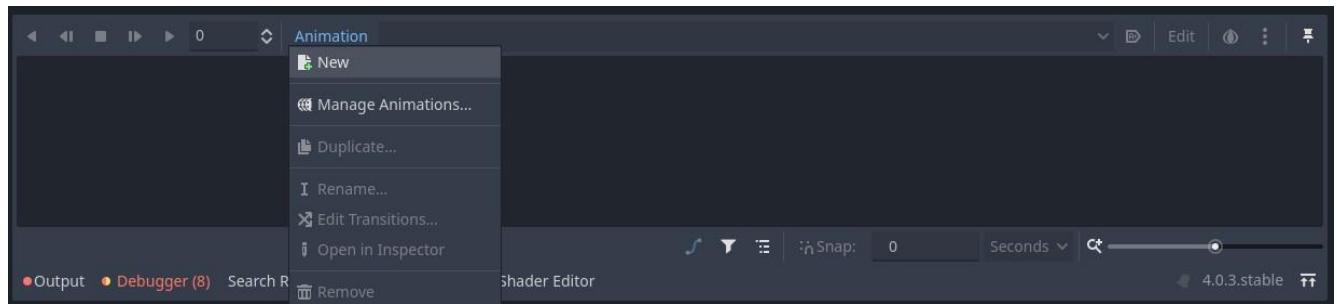
Use the AnimatedSprite2D node for simple, frame-by-frame 2D sprite animations. Use the AnimationPlayer for complex animations involving multiple properties, nodes, or additional features like audio and function calls.

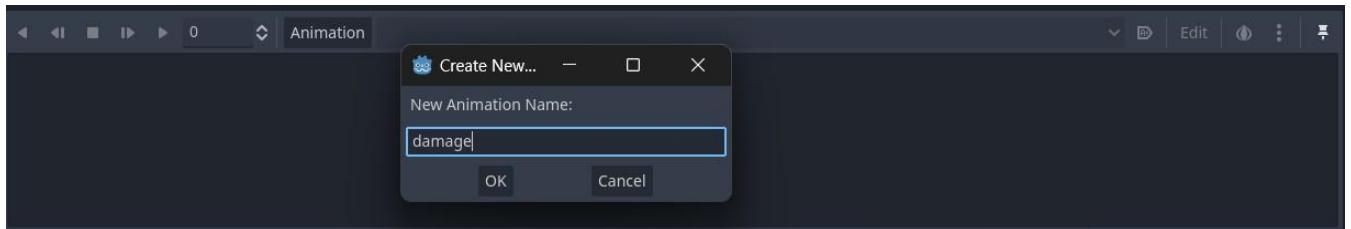


We add animations to the AnimationPlayer node in the Animations panel at the bottom of the editor.



To add an animation, click on the "Animation" label and say new. Let's call this new animation "damage".





Your new animation will have a length of "1" assigned to it as you can see the values run from 0 to 1. Let's change this length to 0.2 because we want this damage indicator to be very short. You can zoom in/out on your track by holding down CTRL and zooming with your mouse wheel.

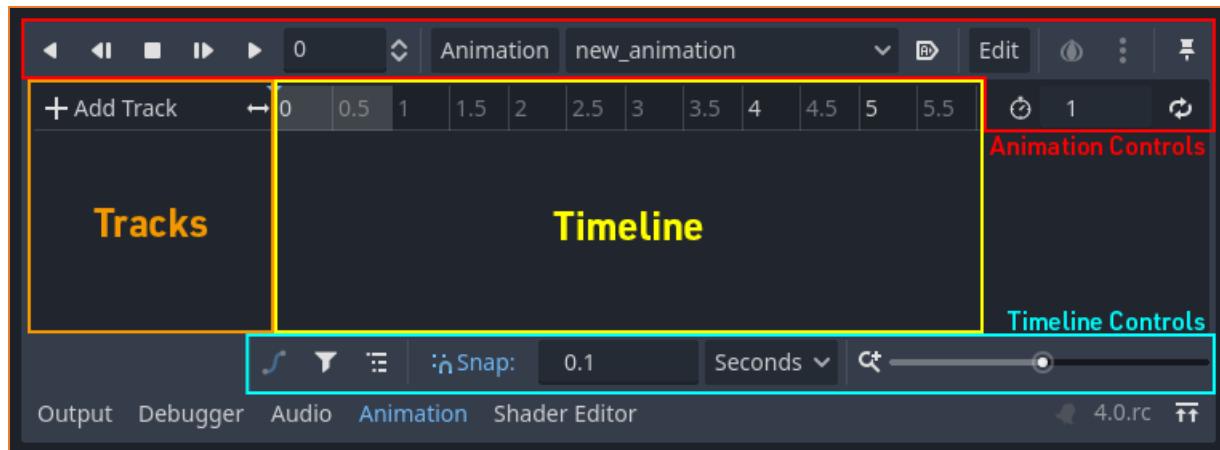
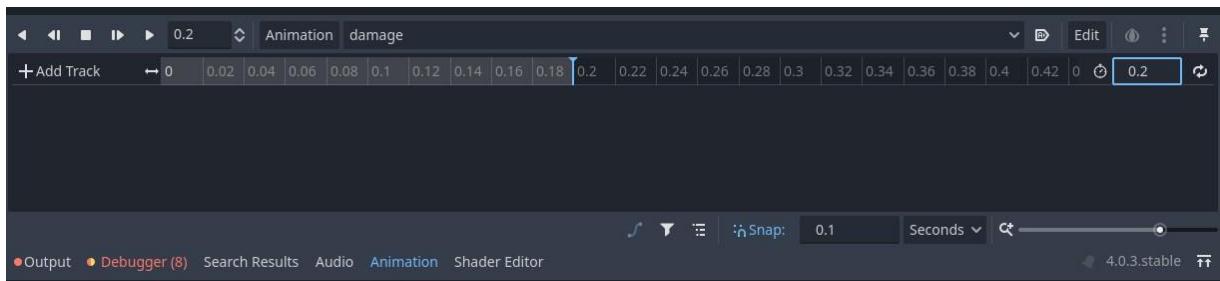
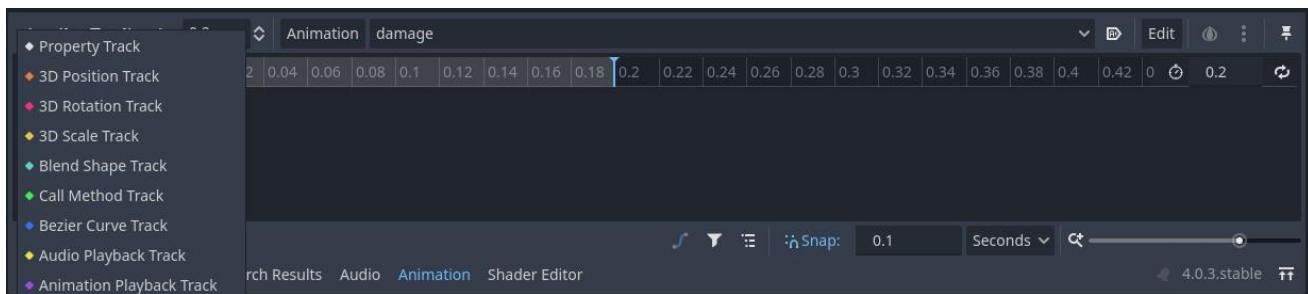


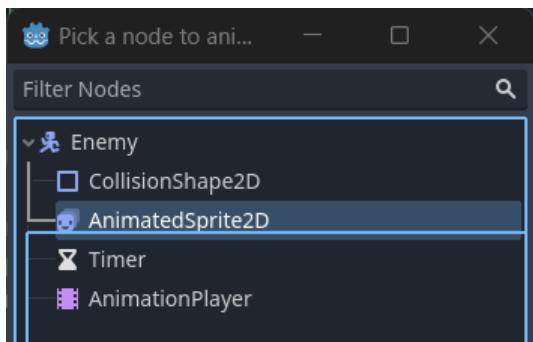
Figure 14: Animations Panel Overview



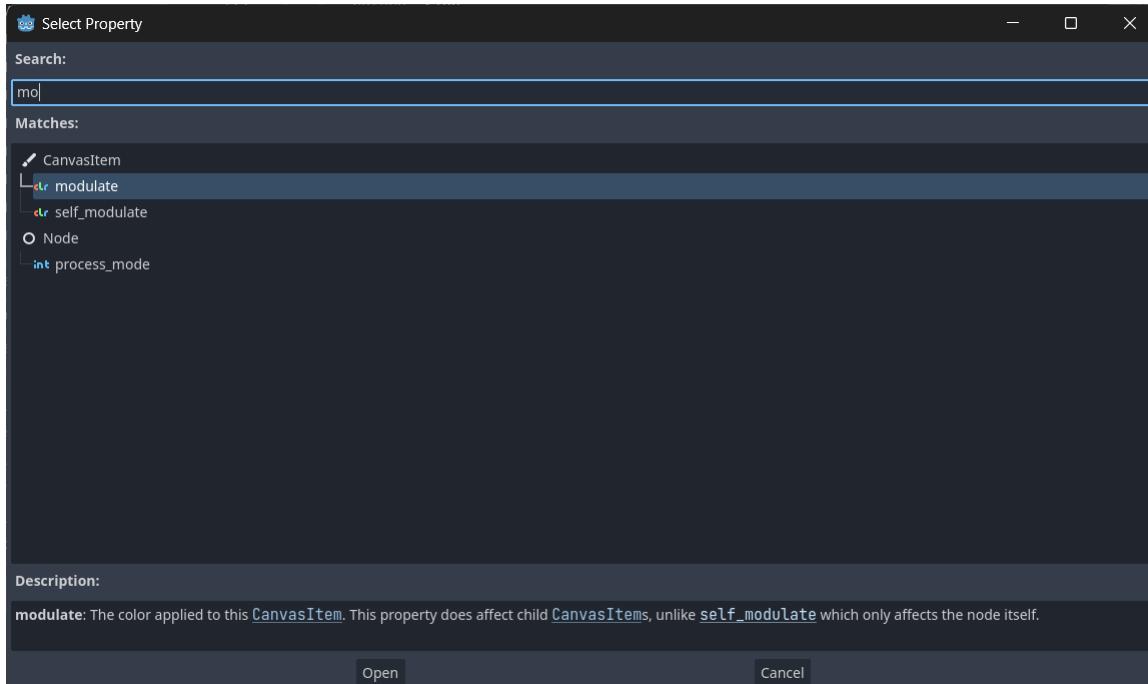
We want this animation to briefly change our `AnimatedSprite2D` node's `modulate` (color) value, which is a property that we can change in the node's Inspector panel. Therefore, we need to add a new Property Track. Click on "+ Add Track" and select Property Track.



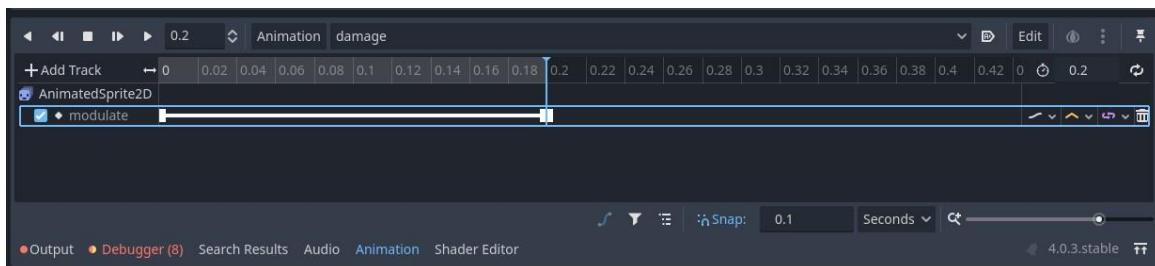
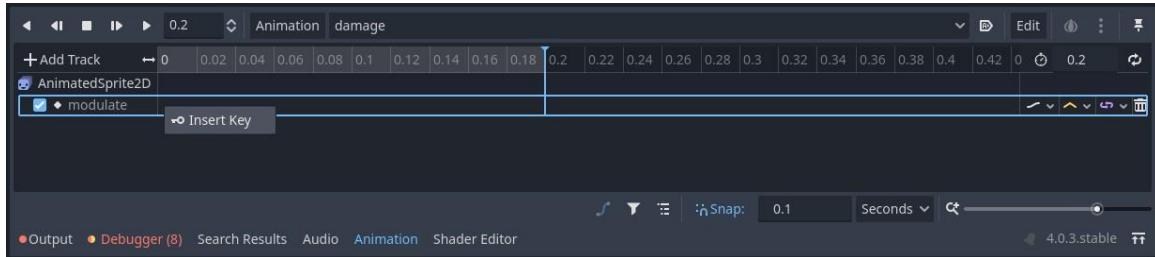
Connect this track to your AnimatedSprite2D node, since this is the node you want to animate.



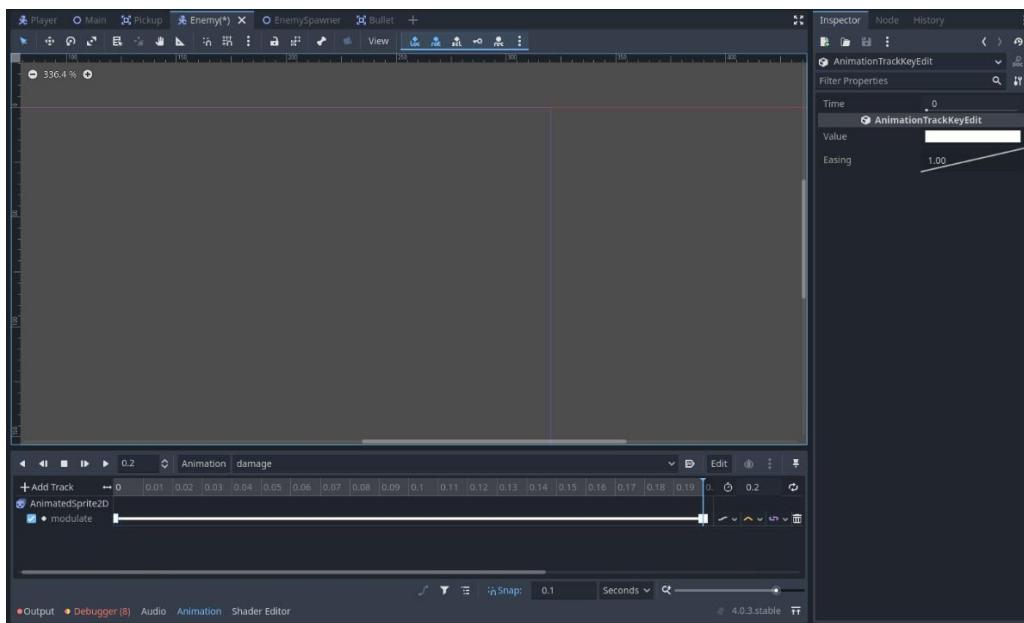
Next, we need to choose the property of the node that we want to change. We want the modulate value, so choose it from the list.



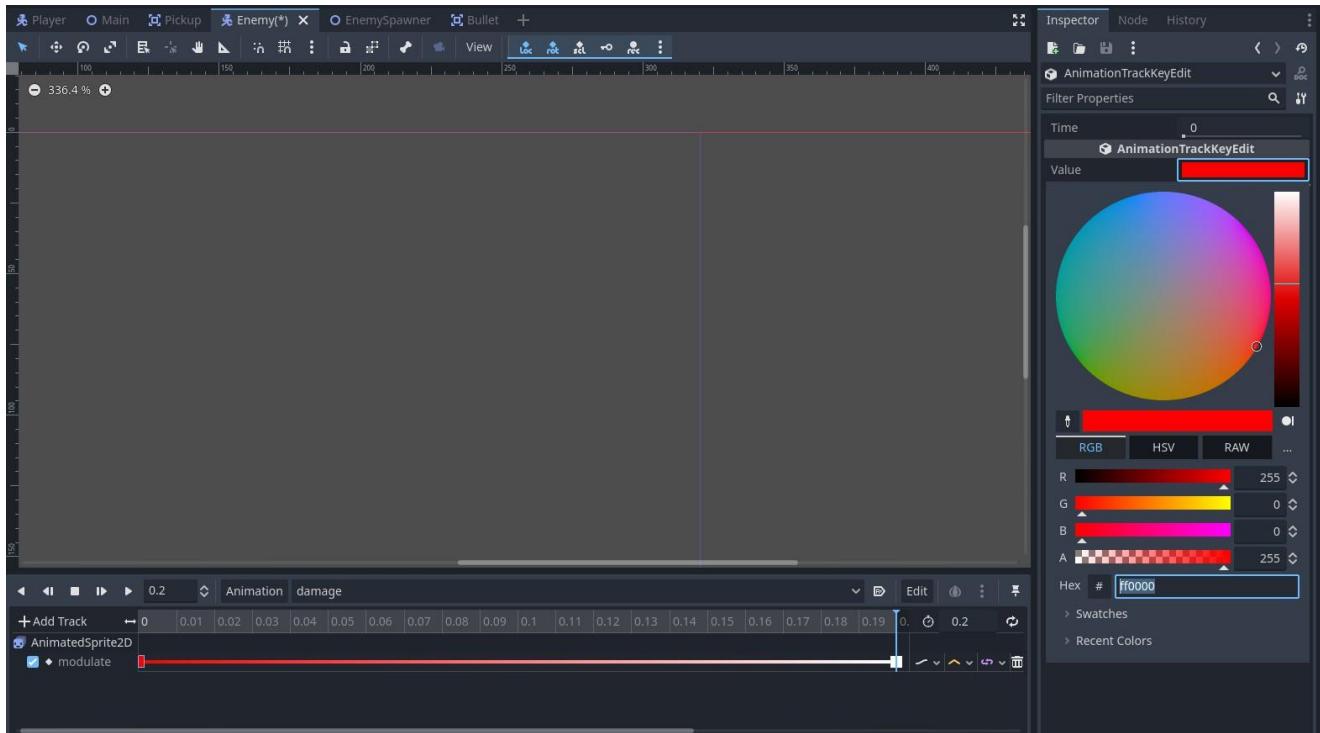
Now that we have the property that we want to change, we need to insert keys to the track. These will be the animation keyframes, which define the starting and/or ending point of our animation. Let's insert two keys: one on our 0 value (starting point), and one on our 0.2 value (ending point). To insert a key, just right-click on your track and select "Insert Key".



If you click on the cubes or keys, the Inspector panel will open, and you will see that you can change the modulate value of our node at that frame underneath "Value".



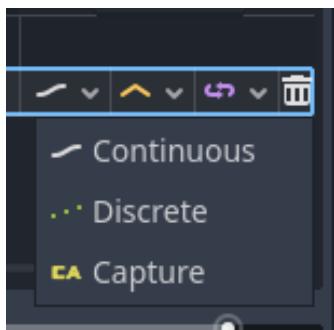
We want our modulate value to go from red at keyframe 0 to white at keyframe 1. To make our modulate value red, we can just change its [RGB](#) values to (255, 0, 0).



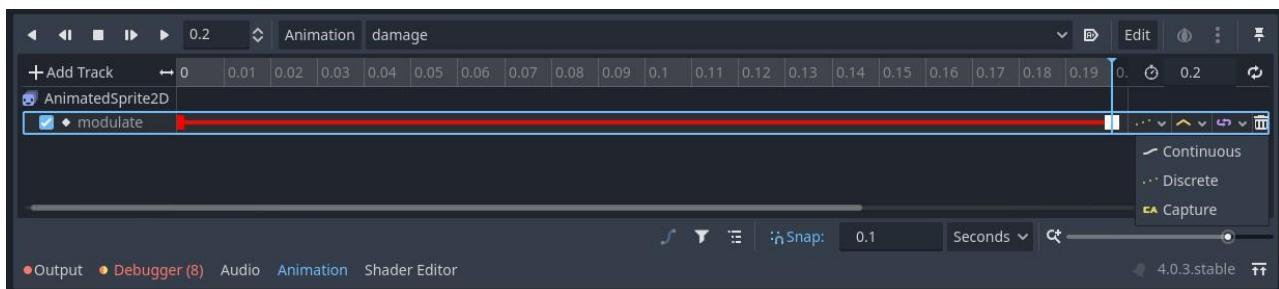
We also want to change the [track rate](#) of our animation. If you were to run the animation now, the *modulate* color would gradually change from red to white. Instead, we want it to change instantly from red to white when the 0.2 keyframe value is reached. To do this we can change its track rate, which is found next to our track under the curved white line.

Godot has three options for our track rate:

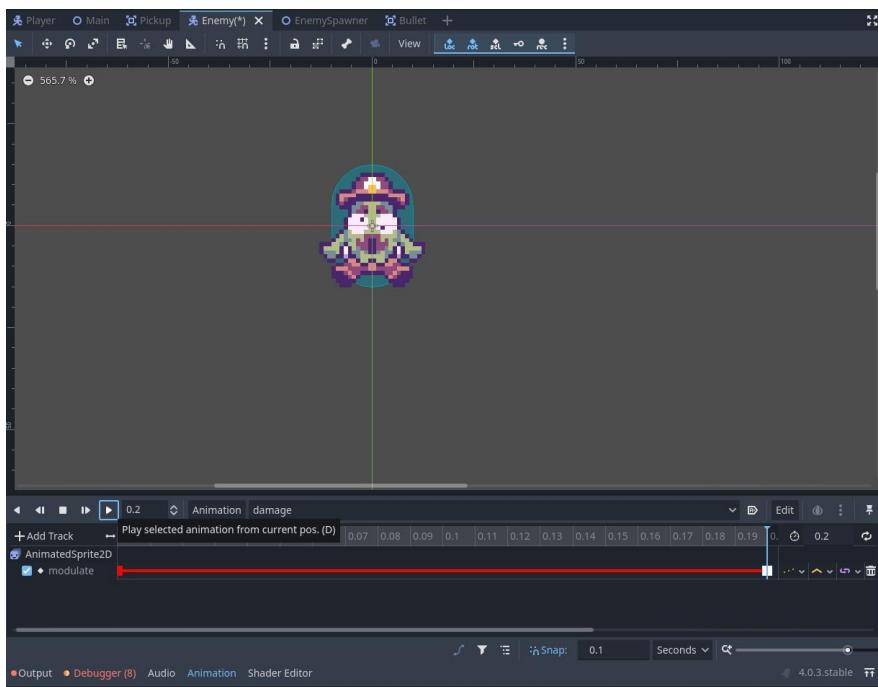
- **Continuous:** Update the property on each frame.
- **Discrete:** Only update the property on keyframes.
- **Trigger:** Only update the property on keyframes or triggers.



We need to change our track rate from continuous to discrete. It should look like this:



Now if you run this animation, it should briefly change your enemy sprite's color to red and then back to its default color.



We can go back to our damage function and now play this animation if the enemy gets damaged.

```
### Enemy.gd

extends CharacterBody2D

# Node refs
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D
@onready var animation_player = $AnimationPlayer

# older code

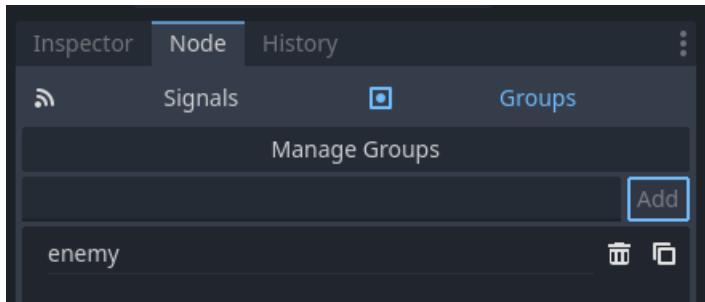
----- Damage & Health -----
func _process(delta):
    #regenerates our enemy's health
    health = min(health + health_regen * delta, max_health)

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        pass
```

Now, to damage the enemy, we will have to add our enemy to a [group](#). This will allow us to use our Area2D node in our Bullet scene to only damage our body if they belong to our "enemy" group. Click on your root node in your Enemy scene, and underneath the Groups property assign the enemy to the group "enemy".

What are Groups?

Groups are a way to organize and manage nodes in a scene tree. They provide a convenient way for applying operations or logic to a set of nodes that share a common characteristic or purpose.



Let's go back to our #todo in our Bullet script and replace it with the damage function that we just created.

```
### Bullets.gd
extends Area2D

# older code
# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Player
    if body.name == "Player":
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
        if tilemap.get_layer_name(Global.WATER_LAYER):
            return
    # If the bullets hit an enemy, damage them
    if body.is_in_group("enemy"):
        body.hit(damage)
    # Stop the movement and explode
    direction = Vector2.ZERO
    animated_sprite.play("impact")
```

ENEMY DEATH

Lastly for this section, we need to add the functionality for our enemy to die. We already created our signal for this, now we just need to update our existing code in our Enemy script to play our death animation and remove the enemy from the scene tree, as well as update our Enemy Spawner code to connect to this signal and update our enemy count.

Let's start in our Enemy script underneath our death conditional. When our enemy dies, the first thing we do is stop the timer that handles its movement and direction. We also need to stop our *process()* function from regenerating our enemy's health value, so we will set the [set_process](#) value to false.

```
### Enemy.gd

# older code

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        #stop movement
        timer_node.stop()
        direction = Vector2.ZERO
        #stop health regeneration
        set_process(false)
        #trigger animation finished signal
        is_attacking = true
        #Finally, we play the death animation and emit the signal
        animation_sprite.play("death")
        death.emit()
```

We also need to set our `is_attacking` variable equal to true so that we can trigger our animation finished signal so that we can remove our node from our scene after our death animation plays. We simply want our enemy to stop, play its death animation, and then signal that it has died so that a new enemy can spawn.

```
### Enemy.gd

# ----- Damage & Health -----
# remove
func _on_animated_sprite_2d_animation_finished():
    if animation_sprite.animation == "death":
        get_tree().queue_delete(self)
    is_attacking = false
```

Now we need to go to our `EnemySpawner` script to create a function that will remove a point from our enemy count after the death signal from our `Enemy` script has been emitted.

```
###EnemySpawner.gd

# ----- Spawning -----
# older code

# Remove enemy
func _on_enemy_death():
    enemy_count = enemy_count - 1
```

We want this function to be connected to our signal in our `spawn` function so that our spawner knows that an enemy has been removed and it should spawn a new one.

```
###EnemySpawner.gd

# ----- Spawning -----
func spawn_enemy():
    var attempts = 0
    var max_attempts = 100 # Maximum number of attempts to find a valid spawn
    location
    var spawned = false

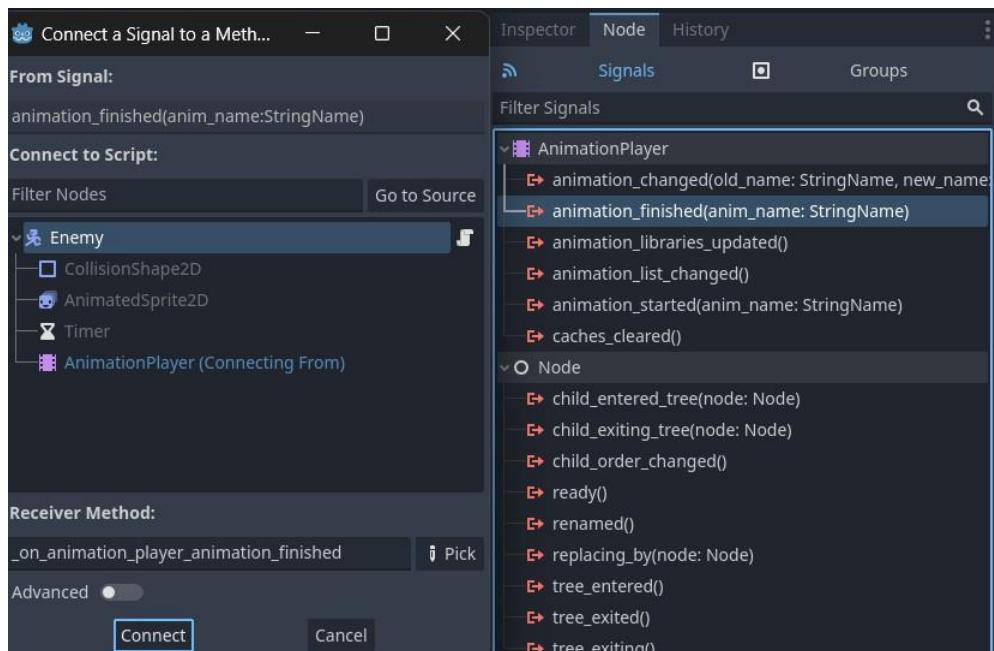
    while not spawned and attempts < max_attempts:
        # Randomly select a position on the map
```

```

var random_position = Vector2(
    rng.randi() % tilemap.get_used_rect().size.x,
    rng.randi() % tilemap.get_used_rect().size.y
)
# Check if the position is a valid spawn location
if is_valid_spawn_location(Global.GRASS_LAYER, random_position) ||
    is_valid_spawn_location(Global.SAND_LAYER, random_position):
    var enemy = Global.enemy_scene.instantiate()
    enemy.death.connect(_on_enemy_death) # add this
    enemy.position = tilemap.map_to_local(random_position) + Vector2(16,
        16) / 2
    spawned_enemies.add_child(enemy)
    spawned = true
else:
    attempts += 1
if attempts == max_attempts:
    print("Warning: Could not find a valid spawn location after",
        max_attempts, "attempts.")

```

Finally, back in our Enemy script, we'll need to reset our Enemy's modulate value after the damage animation has played as well as when they spawn. This will prevent our enemy from spawning or staying red after they've been damaged. Connect your AnimationPlayer node's *animation_finished* signal to your Enemy script.



```
### Enemy.gd

# older code

func _ready():
    rng.randomize()
    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)

# Reset color
func _on_animation_player_animation_finished	anim_name):
    animation_sprite.modulate = Color(1,1,1,1)
```

And so, our Player can now shoot a bullet and if it hits an enemy it damages them. After three hits, it kills the enemy and removes them from the scene!



And that's it for this part. It was quite a lot of work, and if you made it this far, congratulations on being persistent! Next up we're going to spawn loot upon our enemy's death, and then we'll add the functionality for them to attack our player character and deal damage to us! Remember to save your game project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 13: ENEMY DROPPING LOOT ON DEATH

We can finally shoot and kill our enemy, but we can do one better. I want us to upgrade our enemy's death function so that our enemy drops loot that we can pick up. This loot drop will spawn at a 90% chance, and the loot will be randomized from our Pickups enum. This means our enemy might drop either ammo, a health drink, or a stamina drink.

WHAT YOU WILL LEARN IN THIS PART:

- How to randomize percentage and Enum return values.

In our death conditional in our damage() function, we need to use our randomized method to randomize the loot change percentage. If the value is less than 0.9 (90%), we will randomize a pickup. This pickup will also be randomized according to the number of enum values we have, which is 3. This pickup will then be added to our scene and spawned at our killed enemy's location.

```
### Enemy.gd
extends Area2D

# older code

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        #stop movement
        timer_node.stop()
        direction = Vector2.ZERO
```

```

#stop health regeneration
set_process(false)
#trigger animation finished signal
is_attacking = true
#Finally, we play the death animation and emit the signal for the
spawner.
animation_sprite.play("death")
death.emit()
#drop loot randomly at a 90% chance
if rng.randf() < 0.9:
    var pickup = Global.pickups_scene.instantiate()
    pickup.item = rng.randi() % 3 #we have three pickups in our enum

    get_tree().root.get_node("Main/PickupSpawner/SpawnedPickups").call_d
eferred("add_child", pickup)
    pickup.position = position

```

What is `call_deferred`?

[call_deferred](#) is a method that allows you to defer the call of another method to the end of the current frame or cycle. This can be particularly useful in situations where you want to make changes to the scene tree or other objects but don't want those changes to take effect immediately during the current function or signal.

If you now run your scene and you kill your enemy, they should drop loot at a 90% chance. You can lower this value to 50% or 60% if you want - set it according to your preference of how probable you want your enemy to drop loot on death.



Now that our enemy drops loot, we don't have to continuously find pickups. In the next part we will add the functionality to our enemy to be able to shoot at our player and damage us. Remember to save your project, and I'll see you in the next part.

The final source code for this part can be found [here](#).

PART 14: ENEMY SHOOTING & DEALING DAMAGE

It won't be fair if the player is the only entity in our game that can deal damage. That would make them an overpowered bully with no threat. That's why in this part we're going to give our enemies the ability to fight back, and they're going to be able to do some real damage to our player! This process will be similar to what we did when giving our player the ability to shoot and deal damage. This time it would just be the other way around.

This part might take a while, so get comfortable, and let's make our enemy worthy of being our enemy!

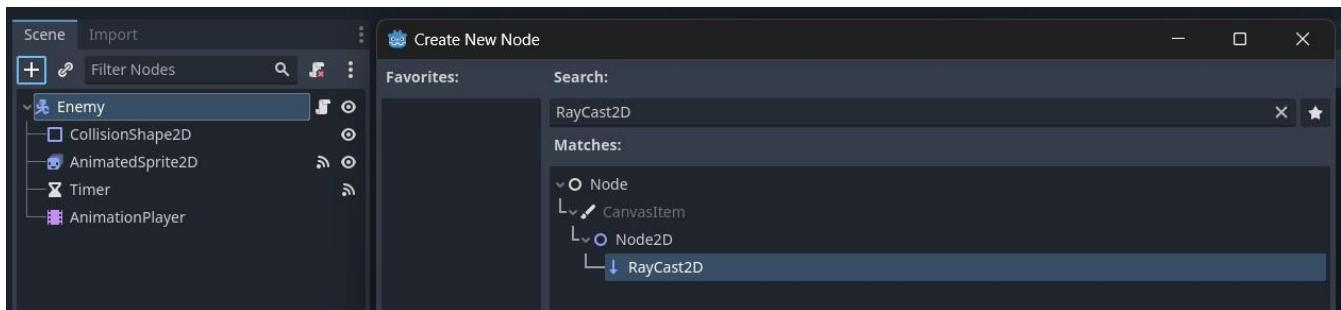
WHAT YOU WILL LEARN IN THIS PART:

- How to use the AnimationPlayer node.
- How to use the RayCast2D node.
- How to work with modulate values.
- How to copy/paste nodes, and duplicate objects.

ENEMY SHOOTING

Previously, in our enemy script, we added some bullet and attack variables that we have not used yet. We aren't controlling our enemies, so we need some way to determine whether or not they are facing us. We can make use of a [RayCast2D](#) node which will create a ray or line which will hit the nodes with collisions around the enemy. This ray cast will then be used to see if they are hitting the Player's collision which is called "Player". If they are, we will trigger the enemy to shoot at us since this means that they are facing us. This will spawn a bullet which, if it hits our player, will damage us.

Let's add this node to our Enemy scene tree.



You will see that a ray or arrow is now coming from your enemy. You can change the length of this ray in the Inspector panel. I'll leave mine at 50 for now.



We want to move this ray in the direction that our Enemy is facing. Since we do all of our movement code in our `_physics_process()` function, we can just do this there. We will turn the ray cast in the direction of our enemy, times the value of their ray cast arrow length (which for me is 50). This is the extent that they'll be able to hit other collisions.

```
### Enemy.gd

extends CharacterBody2D

# Node refs
```

```

@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D
@onready var animation_player = $AnimationPlayer
@onready var timer_node = $Timer
@onready var ray_cast = $RayCast2D

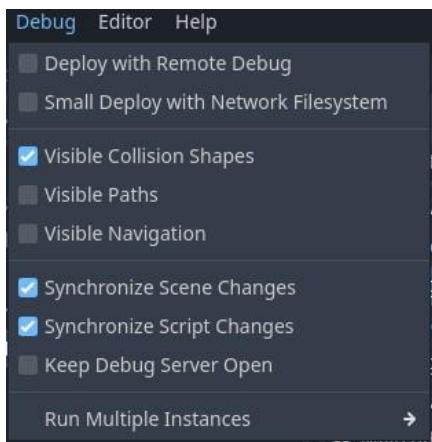
# older code

# ----- Movement & Direction -----
# Apply movement to the enemy
func _physics_process(delta):
    var movement = speed * direction * delta
    var collision = move_and_collide(movement)

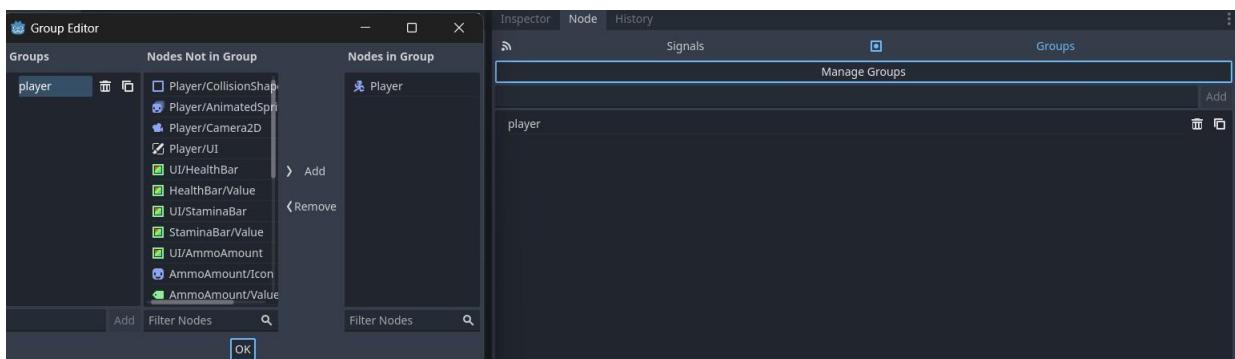
    #if the enemy collides with other objects, turn them around and re-randomize
    #the timer countdown
    if collision != null and collision.get_collider().name != "Player":
        #direction rotation
        direction = direction.rotated(rng.randf_range(PI/4, PI/2))
        #timer countdown random range
        timer = rng.randf_range(2, 5)
        #if they collide with the player
        #trigger the timer's timeout() so that they can chase/move towards our player
    else:
        timer = 0
        #plays animations only if the enemy is not attacking
        if !is_attacking:
            enemy_animations(direction)
        # Turn RayCast2D toward movement direction
        if direction != Vector2.ZERO:
            ray_cast.target_position = direction.normalized() * 50

```

If you enable your collision visibility in your Debug menu, and you run your game, you will see your enemies run around with a ray cast that hits any collision that is in the direction that they're facing.



Let's organize our Player underneath a new group called "player".



Now we can change our `process()` function to spawn bullets and play our enemy's shooting animation if they are colliding with nodes in the "player" group. This whole process is similar to our Player code under `ui_attack` - without the time calculation.

```
### Enemy.gd

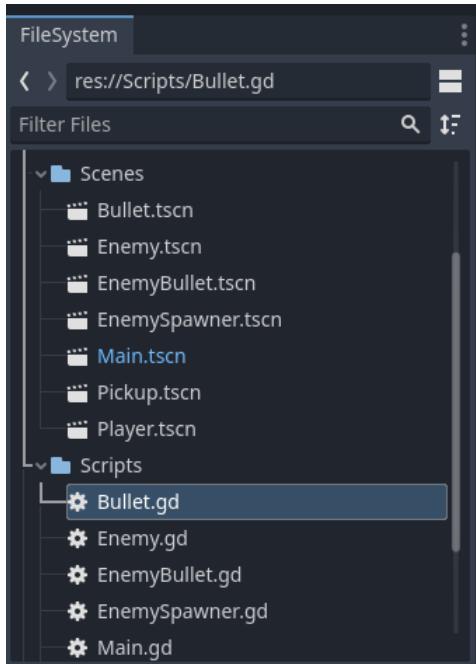
# older code

----- Damage & Health -----
func _process(delta):
    #regenerates our enemy's health
    health = min(health + health_regen * delta, max_health)
    #get the collider of the raycast ray
    var target = ray_cast.get.collider()
    if target != null:
        #if we are colliding with the player and the player isn't dead
        if target.is_in_group("player"):
            #shooting anim
            is_attacking = true
            var animation  = "attack_" + returned_direction(new_direction)
            animation_sprite.play(animation)
```

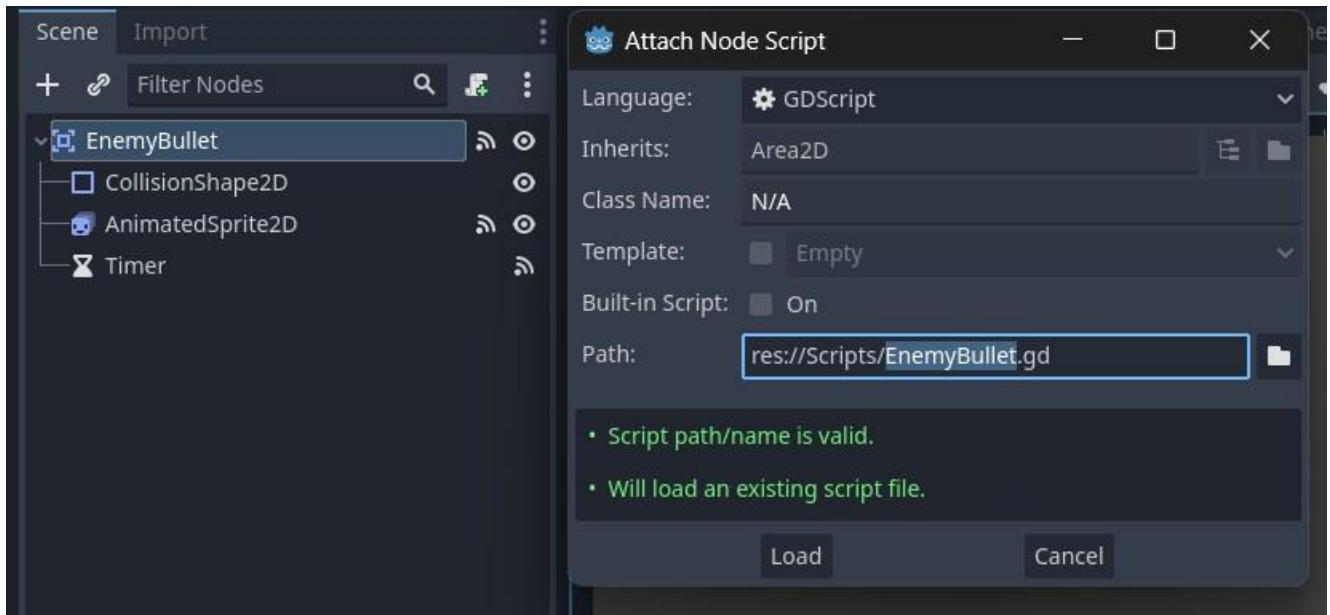
SPAWNING BULLETS

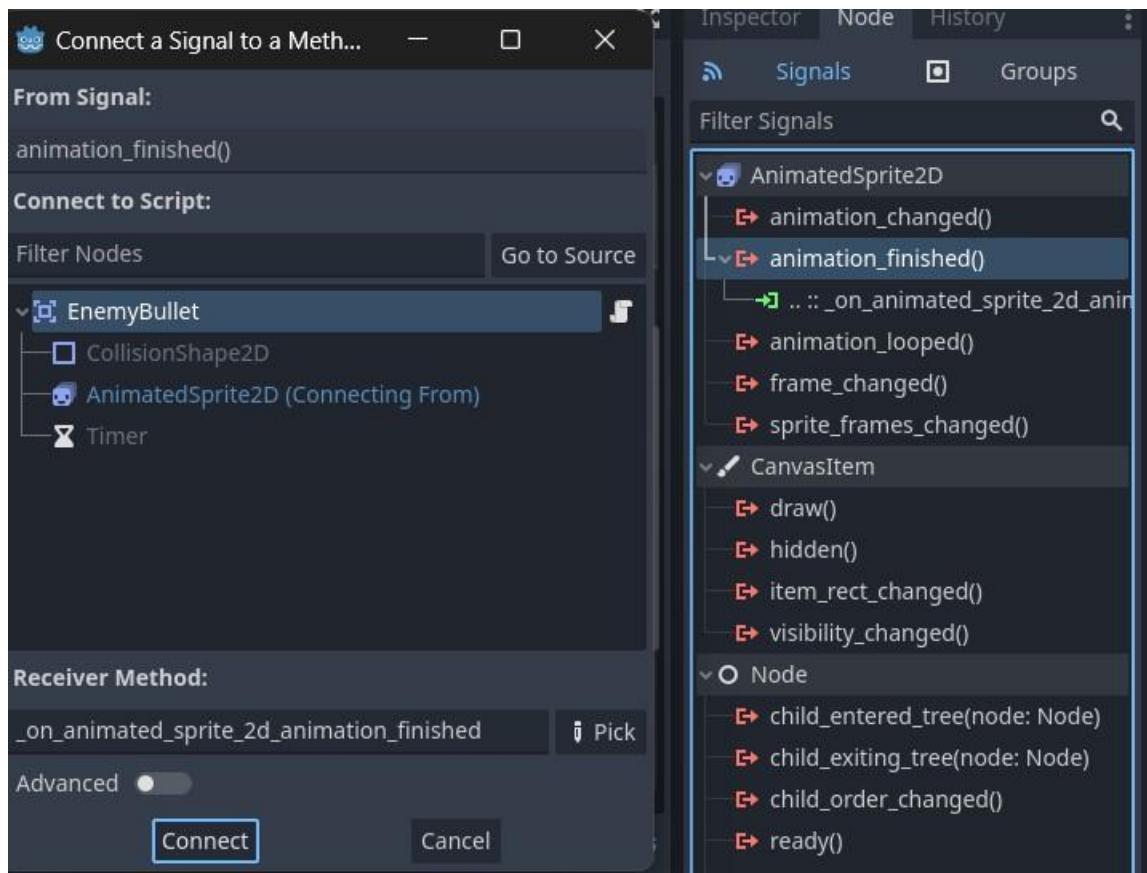
We will also spawn our bullet in our `func _on_animated_sprite_finished()`: function, because only after the shooting animation has played do we want our bullet to be added to our Main scene. Before we can go about instantiating our scene, we need to create a Bullet scene for our enemy. This is because our existing Bullet scene is being told to ignore collisions with the player, so it would just be easier to duplicate our existing scene and swap out the code to ignore the enemy.

Go ahead and duplicate both the Bullet scene and script and rename these to `EnemyBullet.tscn` and `EnemyBullet.gd`.



Rename your new duplicated scene root to `EnemyBullet` and attach the `EnemyBullet` script to it. Also reconnect the `Timer` and `AnimationPlayer`'s signals to the `EnemyBullet` script instead of the `Bullet` script.





In the `EnemyBullet` script, swap around the "Player" and "Enemy" groups in your `on_body_entered()` function.

```
### EnemyBullet.gd

# older code

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Enemy
    if body.is_in_group("enemy"):
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
```

```

if tilemap.get_layer_name(Global.WATER_LAYER):
    return
# trees, buildings and objects
if tilemap.get_layer_name(Global.FOLIAGE_LAYER) or
tilemap.get_layer_name(Global.EXTERIOR_1_LAYER) or
tilemap.get_layer_name(Global.EXTERIOR_2_LAYER):
    animated_sprite.play("impact")

# If the bullets hit player, damage them
if body.is_in_group("player"):
    body.hit(damage)
# Stop the movement and explode
direction = Vector2.ZERO
animated_sprite.play("impact")

```

Now in your Global script, preload the EnemyBullet scene.

```

### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
@onready var bullet_scene = preload("res://Scenes/Bullet.tscn")
@onready var enemy_bullet_scene = preload("res://Scenes/EnemyBullet.tscn")

```

We can now go back to our Enemy scene and spawn our bullet in our *func _on_animated_sprite_2d_animation_finished()* function. We'll have to create another instance of our EnemyBullet scene, where we will update its damage, direction, and position as the direction the enemy was facing when they fired off the round, and the position 8 pixels in front of the enemy - since we want them to have a further shooting range than our player.

```

### Enemy.gd

# older code

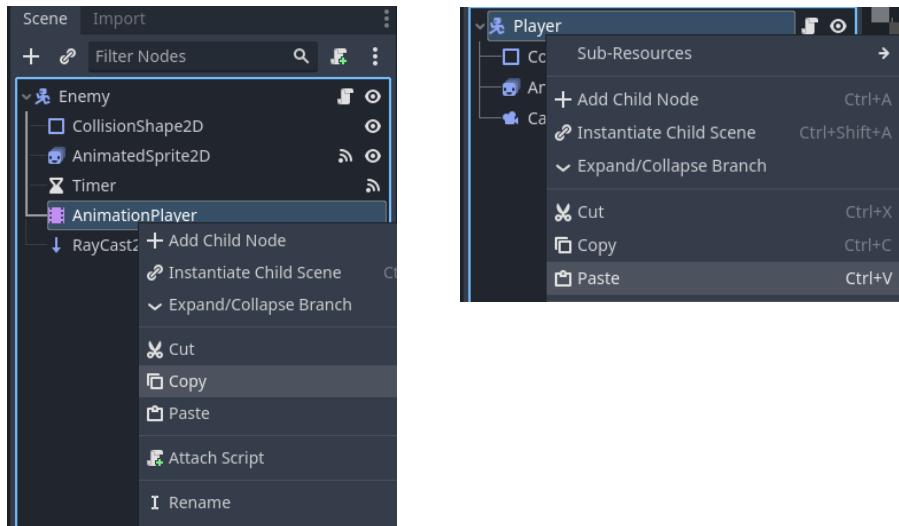
# Bullet & removal
func _on_animated_sprite_2d_animation_finished():
    if animation_sprite.animation == "death":
        get_tree().queue_delete(self)
    is_attacking = false
    # Instantiate Bullet
    if animation_sprite.animation.begins_with("attack_"):
        var bullet = Global.enemy_bullet_scene.instantiate()
        bullet.damage = bullet_damage
        bullet.direction = new_direction.normalized()

        # Place it 8 pixels away in front of the enemy to simulate it coming from
        # the guns barrel
        bullet.position = player.position + new_direction.normalized() * 8
        get_tree().root.get_node("Main").add_child(bullet)

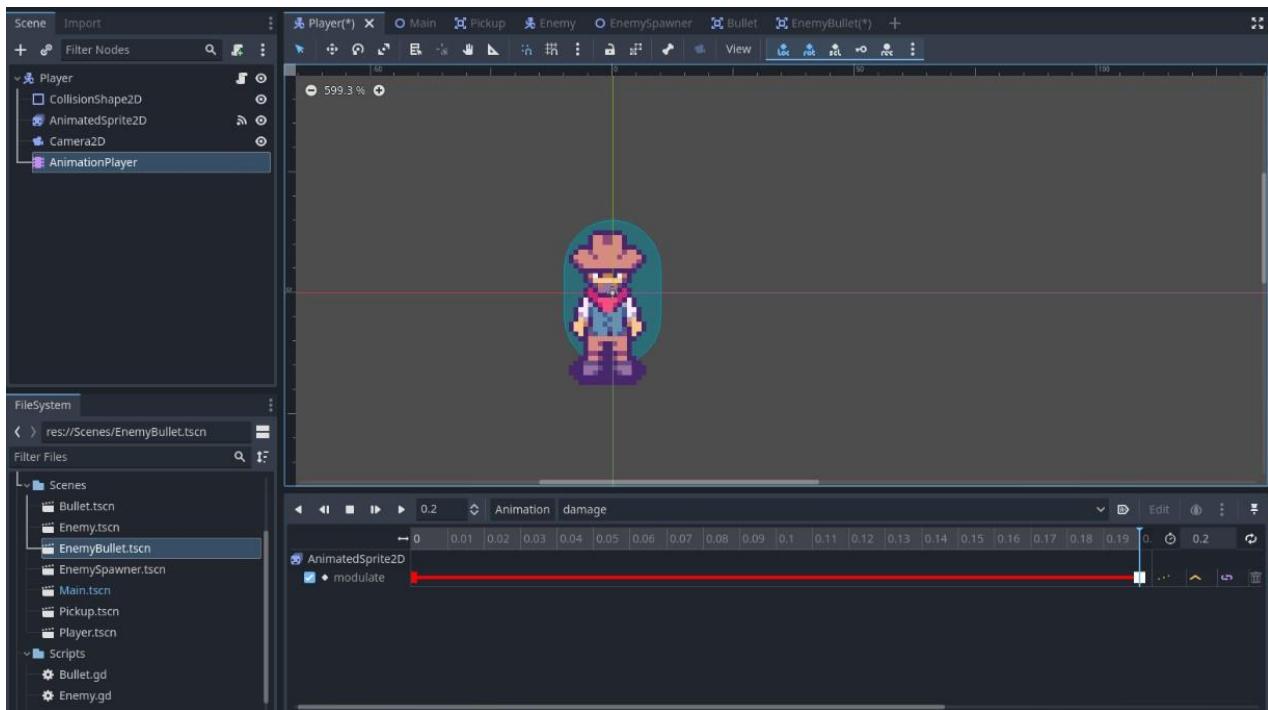
```

DAMAGING PLAYER

Now we need to go ahead and add a damage function in our Player script so that our `body.hit(damage)` code in our EnemyBullet script can work. Before we do this, we'll also go ahead and add that damage animation that we added to our Enemy to our Player. You can recreate it if you want to practice working with the AnimationPlayer, but I'm just going to copy the node from my Enemy scene and paste it into my Player scene.



Because we already have an AnimatedSprite2D node in our Player scene, it would automatically connect the animation to our modulate value.



In our code, we can go ahead and create our damage function. This function is similar to the one we created in our enemy scene, where we get damaged after being hit by a bullet. The red "damage" indicator animation also plays upon the damage, and our health value gets updated. We are not going to add our death functionality in this part, because we want to implement that in the next part along with our game-over screen.

```
### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount
@onready var animation_player = $AnimationPlayer
```

```

# older code

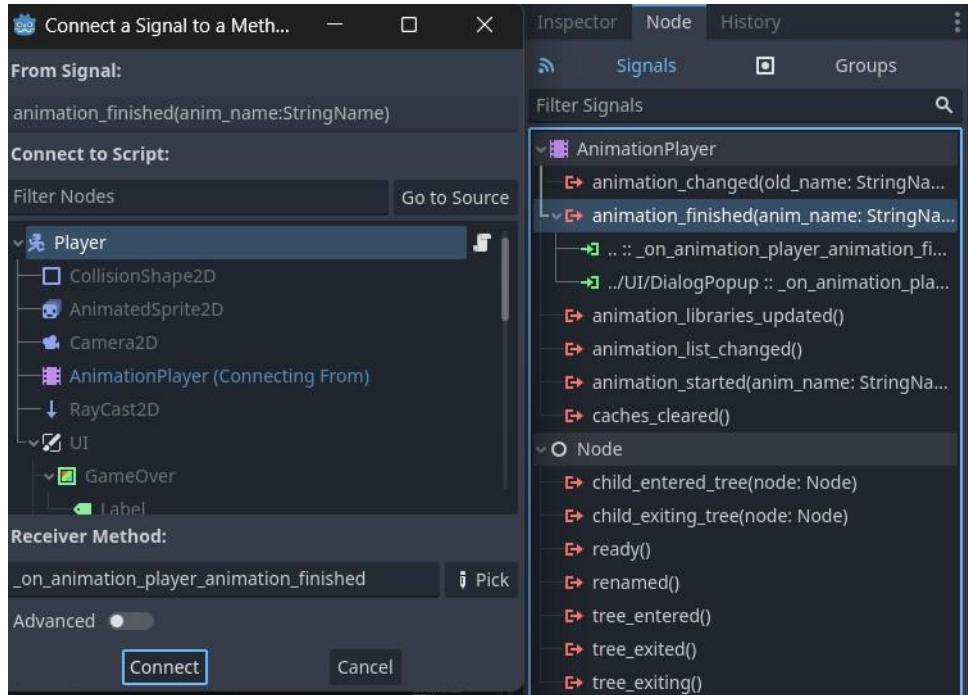
# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
    health_updated.emit(health, max_health)
    if health > 0:
        #damage
        animation_player.play("damage")
        health_updated.emit(health)
    else:
        #death
        set_process(false)
        #todo: game over

```

Now if you run your scene, your enemy should chase you and shoot at you, and when the bullet impacts your player node, it should decrease the player's health value.



We also need to connect our Animation Player's `animation_finished()` signal to our main script and reset our value there so that it will reset the modulate even when the animation gets stuck halfway through completion.

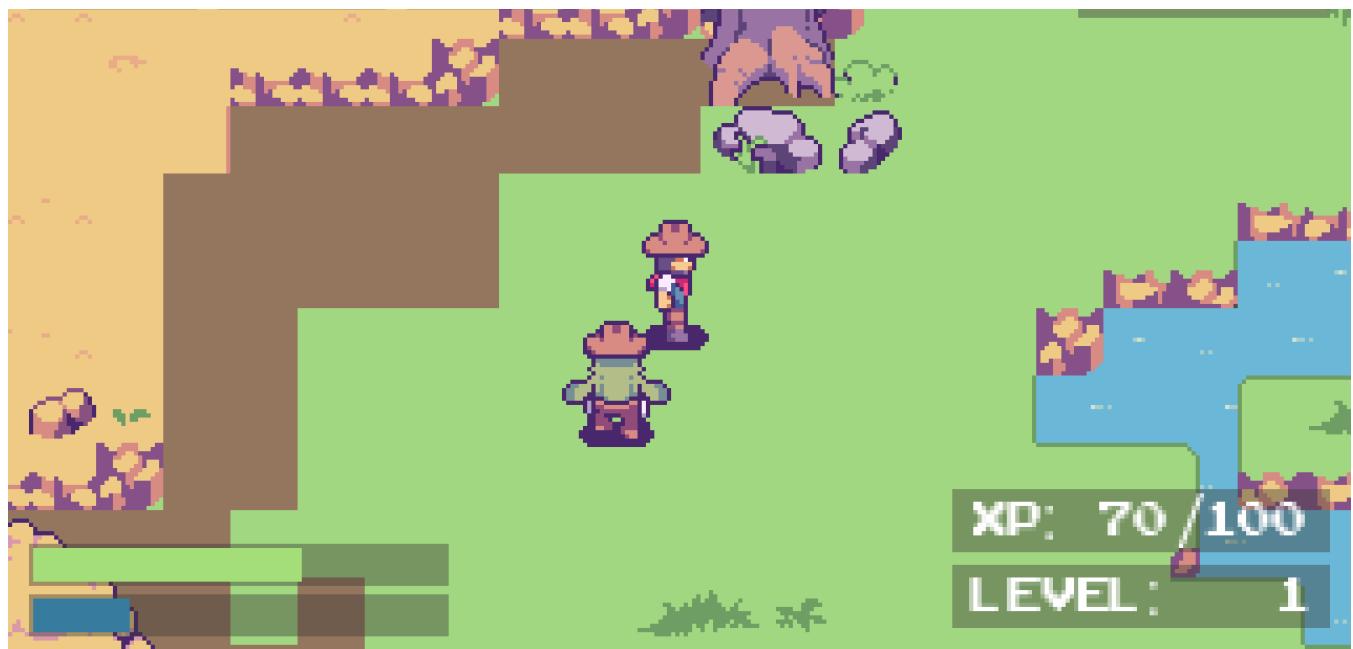


```
### Player.gd

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
    health_pickups_updated.connect(health_amount.update_health_pickup_ui)
    stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)

    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)

func _on_animation_player_animation_finished(anim_name):
    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)
```



Congratulations, you now have an enemy that can shoot your player! Next up, we're going to be giving our player the ability to die, and we'll be implementing our game over system. Remember to save your project, and I'll see you in the next part.

Your final code for this part should look like [this](#).

PART 15: PLAYER DEATH & GAME OVER

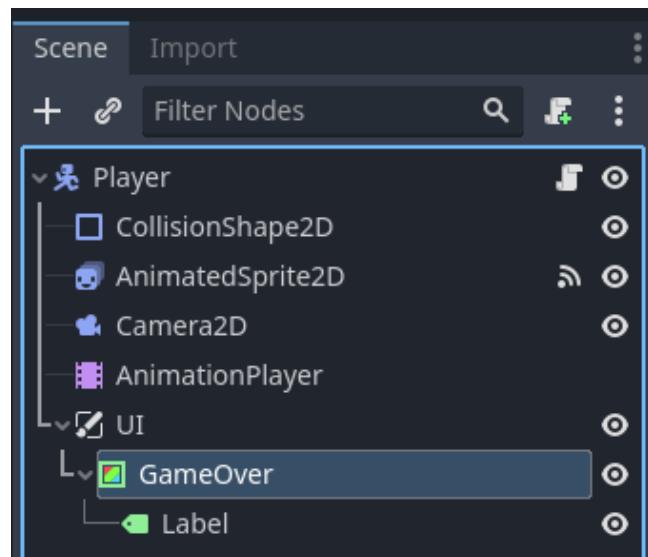
Now that our player can be damaged by our enemies, we can go ahead and implement the functionality so that they can die, and if they have died, the game is over, and a game-over screen is displayed. Later on, we'll build on this game-over system so that we get redirected toward the Main Menu screen.

WHAT YOU WILL LEARN IN THIS PART:

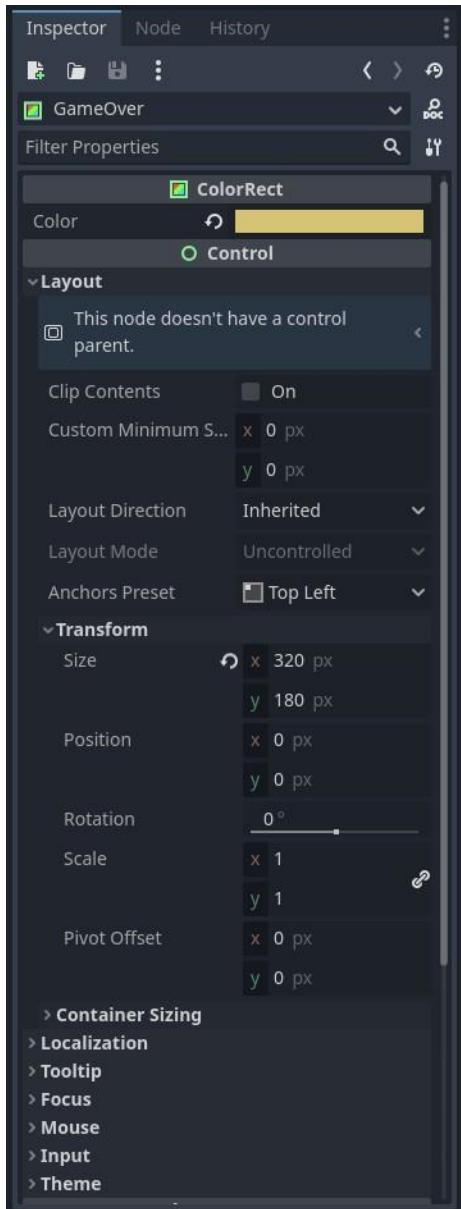
- Further practice with the AnimationPlayer node.
- Further practice with the CanvasLayer nodes.

GAME OVER SCREEN

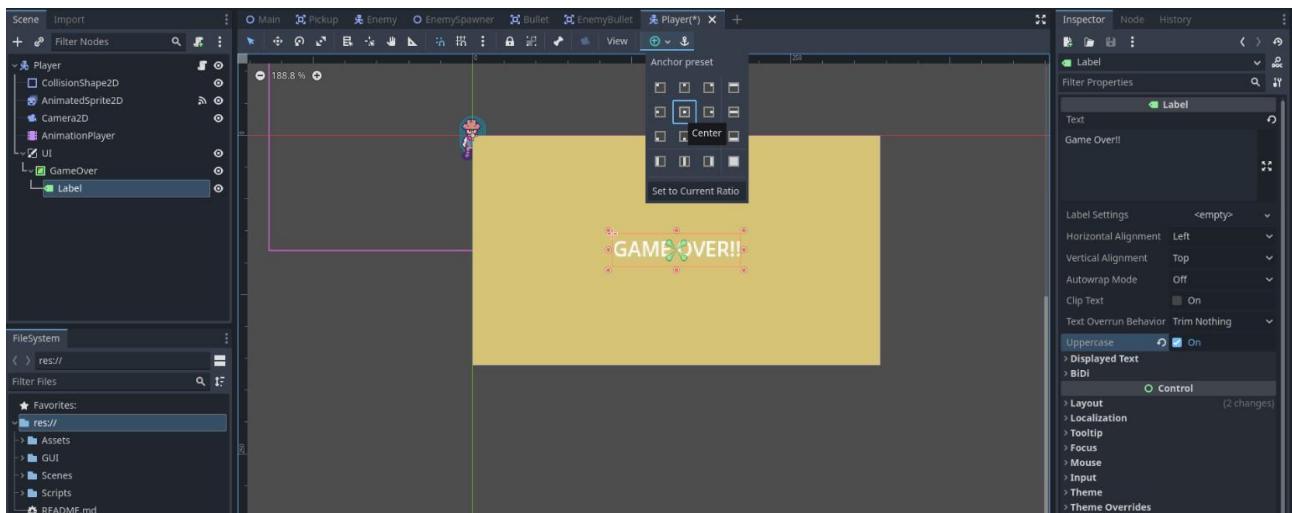
In your Player scene, add a new Label and ColorRect to your UI node. Rename the ColorRect to “GameOver”. It should look like this:



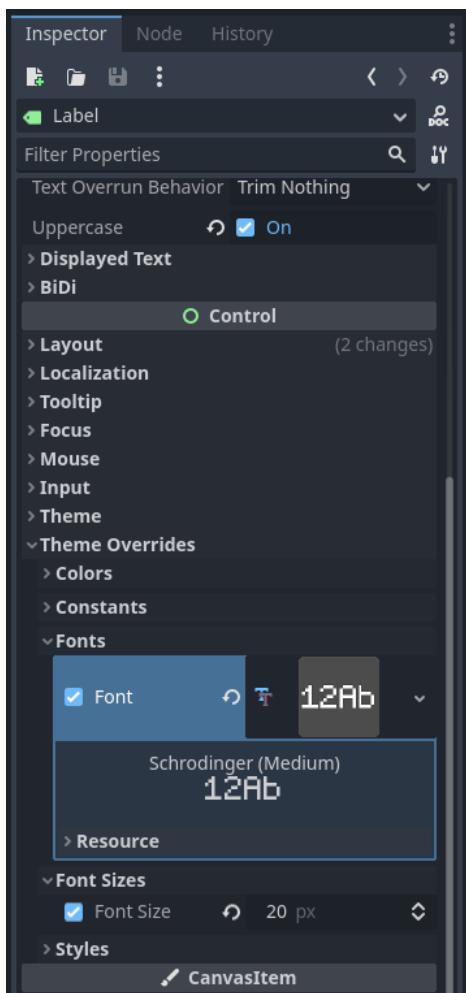
Change the GameOver node's size to the size of your game window (x: 320, y: 180). Also, change its Color property to #d6c376.



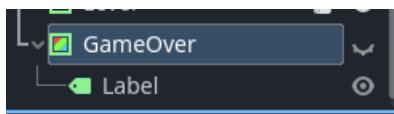
Select your Label node and change its text to "GAME OVER!!!". Make sure that it is in uppercase, and change the anchor preset to be centered.



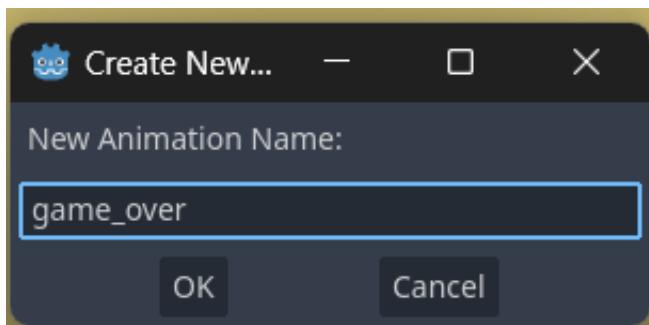
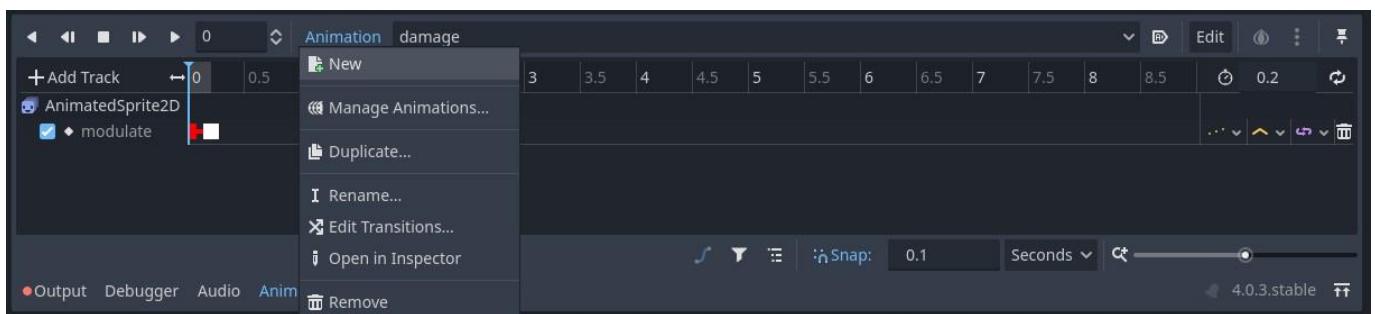
Let's assign a new Font to this label. Underneath Control > Theme Overrides > Fonts, Quick Load a new font and select the font to be "Schrödinger.ttf". Set the font size to 20.



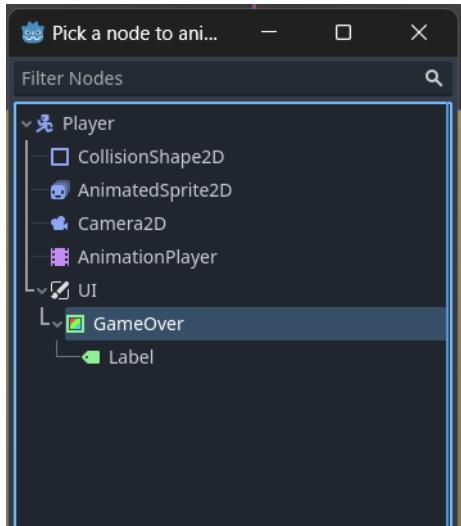
Now hide your GameOver node by clicking on the eye icon next to it.



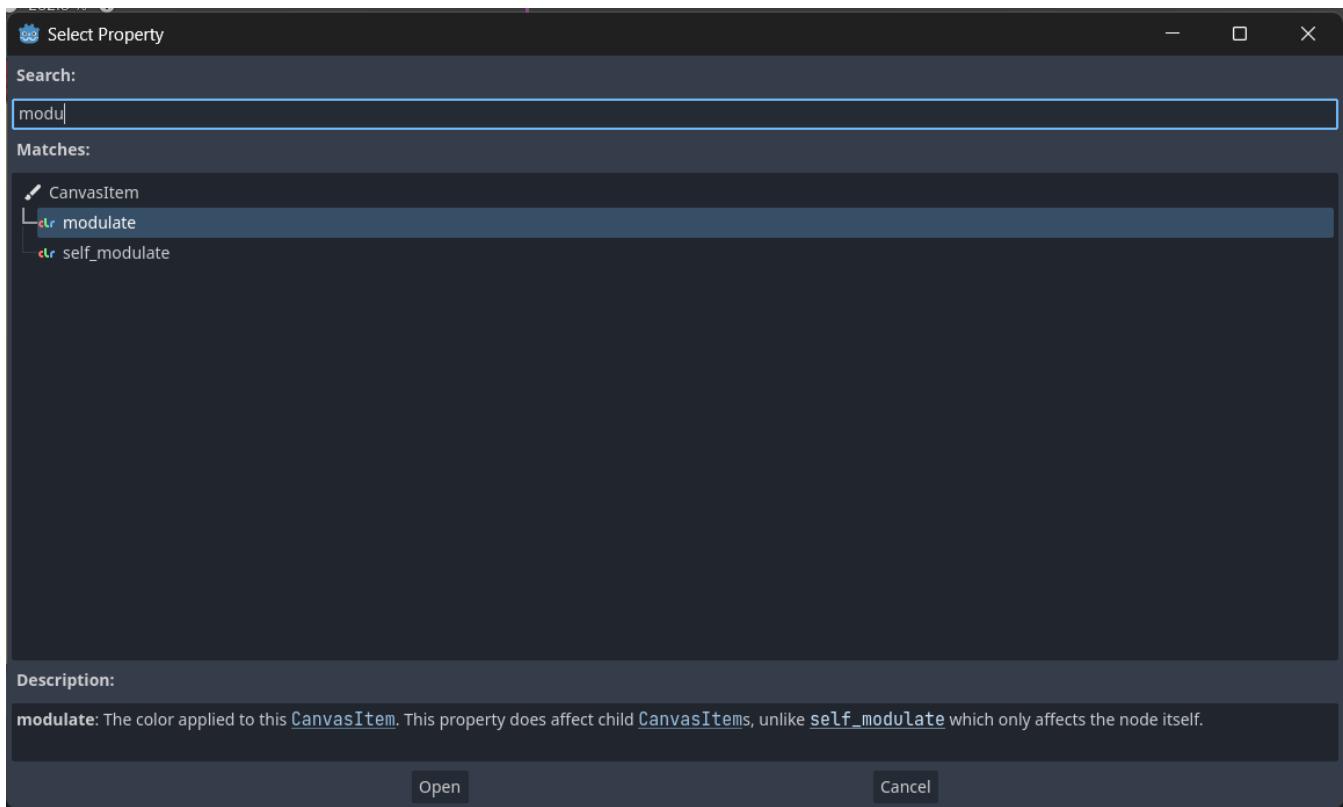
In your Animations panel at the bottom window, create a new animation by clicking on the Animation label. Call this animation "game_over".



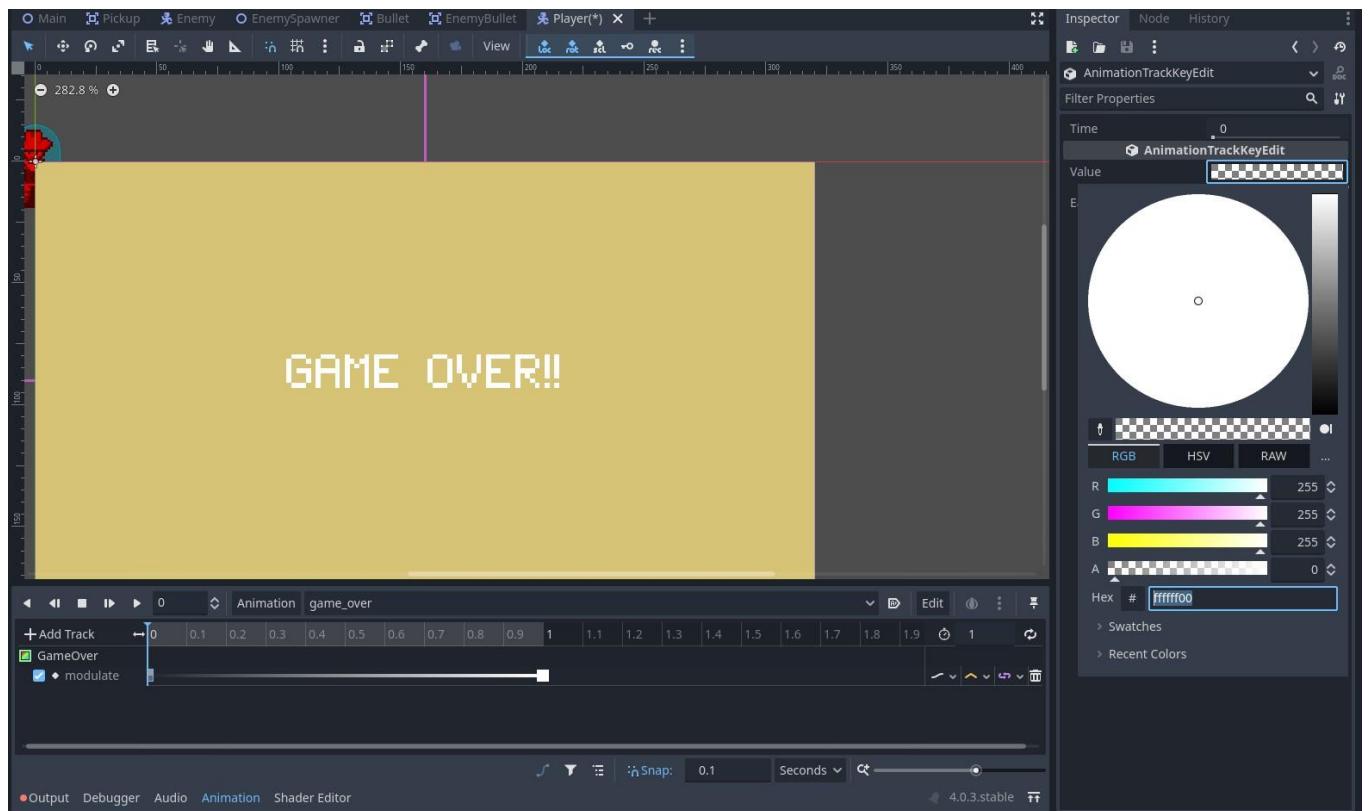
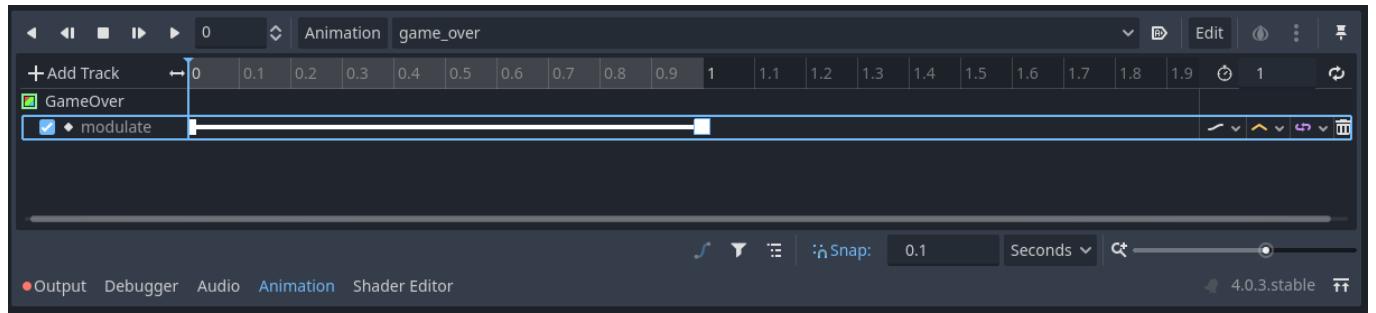
For this animation, we want our game over screen to slowly be revealed if our player dies. Therefore, we will change the ColorRect node's (GameOver) modulate property. Add a new Property Track and connect it to your GameOver node.



Select the modulate value property and add two keys - one at keyframe 0 and the other one at keyframe 1.



Select the first keyframe (0) and change the Alpha value in your Value property to 0. This is because we want the window to go from invisible to visible. You can play the animation to see the effect we're going for.



Now in your Player's damage code underneath our death conditional, let's play this animation.

```
### Player.gd

# older code

# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
```

```
health_updated.emit(health, max_health)
if health > 0:
    #damage
    animation_player.play("damage")
    health_updated.emit(health, max_health)
else:
    #death
    set_process(false)
    animation_player.play("game_over")
```

If you run your scene and your player gets shot enough times for its health value to fall below 0, your game over screen should show. We'll make this screen pausable later on when we return it to our main screen.



Now our player can "die" and the game notifies us that our game is over. We'll build on this in the next few parts to improve on this system, but in the next part, we're going to add our Level and XP functionalities which will increase our XP and Levels after completing quests or killing enemies. Remember to save your project, and I'll see you in the next part.

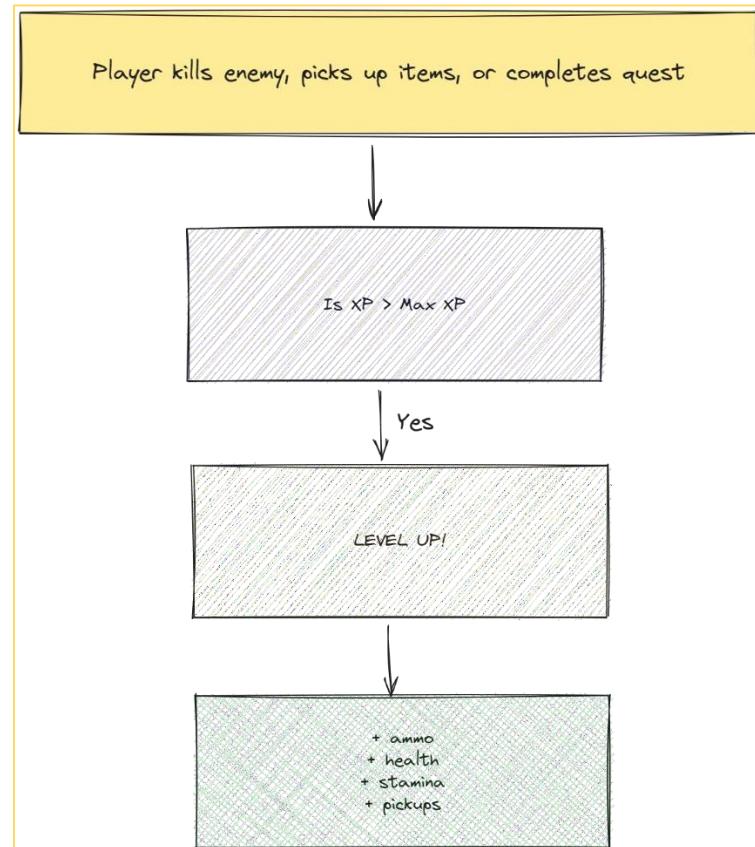
Your final source code for this part should look like [this](#).

PART 16: LEVEL & XP

Now that we have a basic enemy and player, we need to give ourselves a motive to kill the enemies. We can do this via a leveling system that increases our player's level after a certain amount of XP is gained. If the player levels up, they get rewarded with pickups, and a stats (health, stamina) refill. We will also increase their max health and max stamina values upon a level-up.

WHAT YOU WILL LEARN IN THIS PART:

- How to pause the scene tree.
- How to allow/disallow input processing.
- How to change a node's Processing Mode.
- How to (optionally) change a mouse cursors image and visibility.



LEVEL UP POPUP

Once you're ready, open up your game project, and in your Player script by your signals, let's define three new signals that will update our xp, xp requirements, and level values.

```
### Player.gd

# Custom signals
signal health_updated
signal stamina_updated
signal ammo_pickups_updated
signal health_pickups_updated
signal stamina_pickups_updated
signal xp_updated
signal level_updated
signal xp_requirements_updated
```

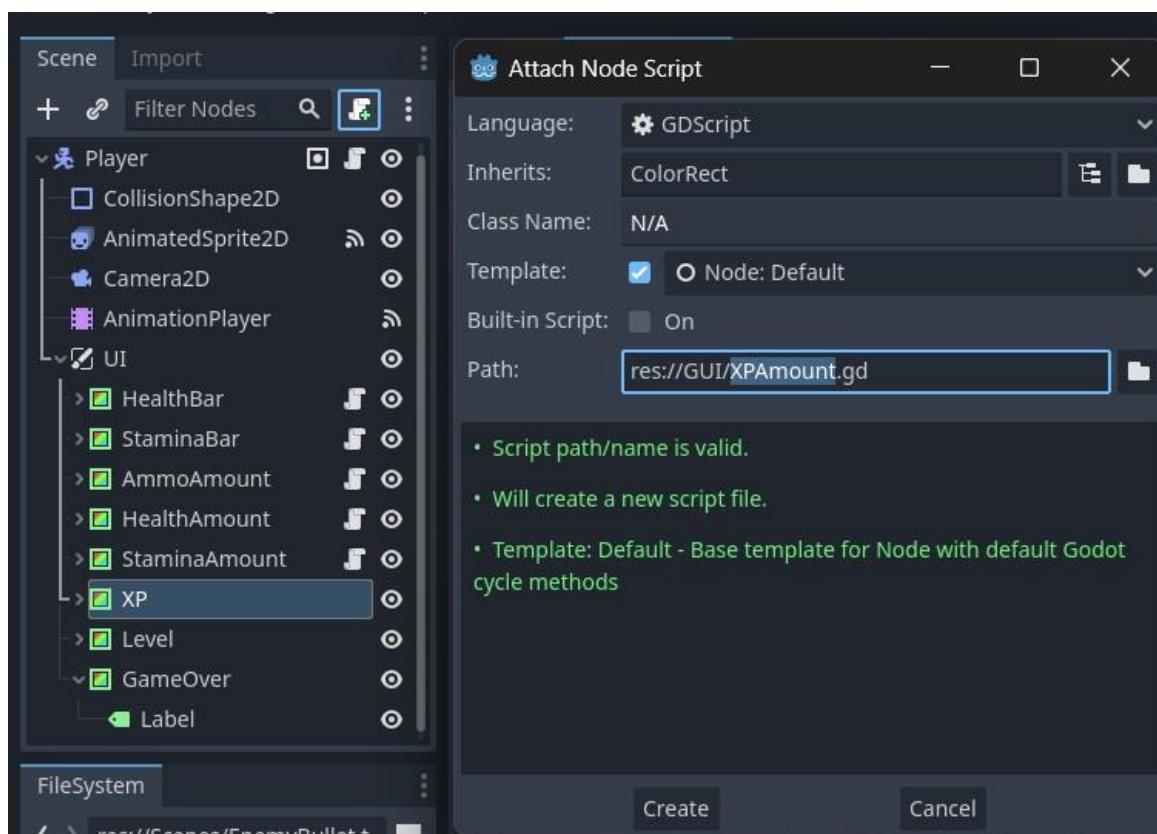
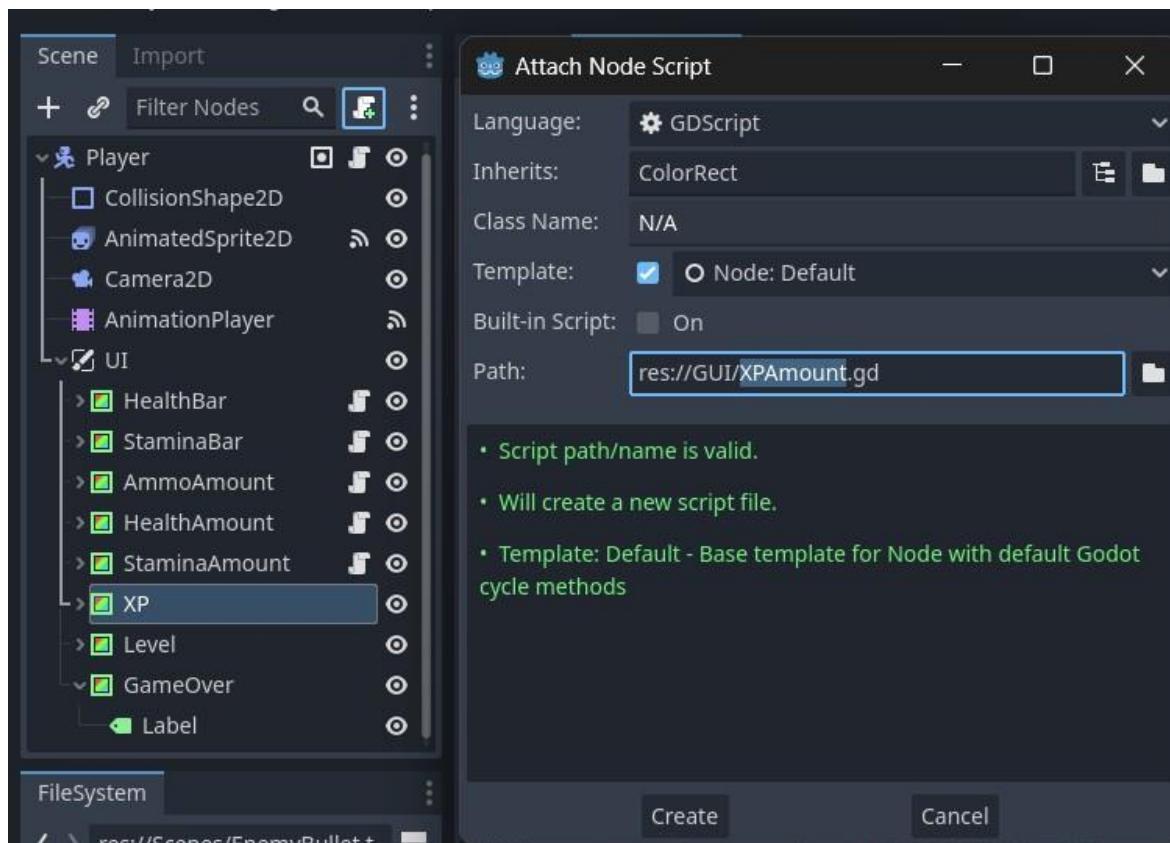
Next, we will need to create the variables that these signals will update when they are emitted. You can change the xp, level, and required xp values that the player will start with to be any value you want.

```
### Player.gd

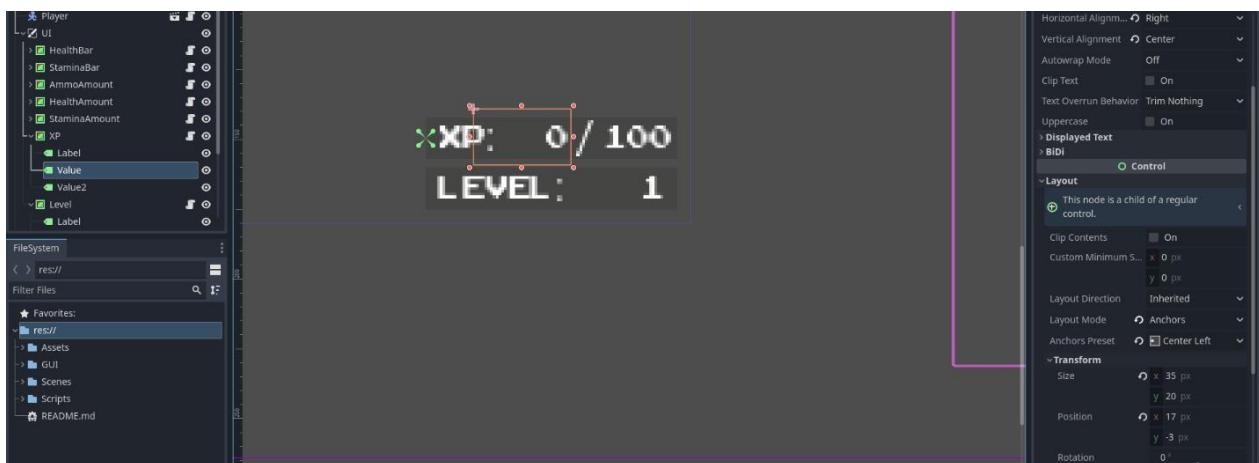
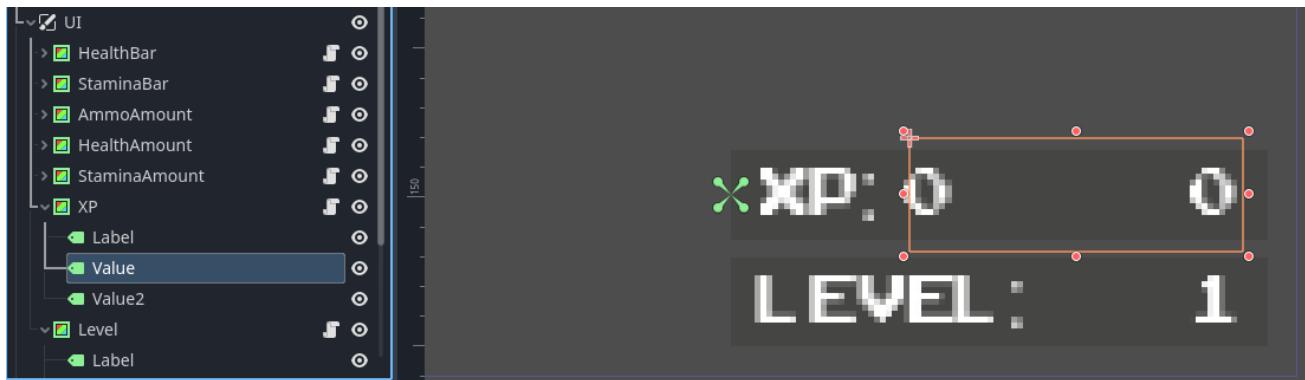
# XP and levelling
var xp = 0
var level = 1
var xp_requirements = 100
```

If you remembered how we updated our health and stamina GUI elements in the previous parts, you will know that we will have to create functions in our XP and Level elements in our Player scene, and then connect them to our signals in our Player scene.

In your Player Scene, underneath your UI CanvasLayer, attach a new script to your XP and Level ColorRect. Make sure to save this underneath your GUI folder.



Our XP ColorRect should also have two values, one for our XP and one for our XP Requirements, so go ahead and duplicate your Value node. It's transform values can be seen in the images below.



We want to update the Value child nodes from these ColorRects (not the Label), so let's go ahead and create a function for each new script to update our XP and Level values.

```
### XPAmount.gd

extends ColorRect

# Node refs
@onready var value = $Value
@onready var value2 = $Value2

#return xp
func update_xp_ui(xp):
    #return something like 0
    value.text = str(xp)

#return xp_requirements
func update_xp_requirements_ui(xp_requirements):
    #return something like / 100
    value2.text = "/" + str(xp_requirements)
```

```
### LevelAmount.gd

extends ColorRect

# Node refs
@onready var value = $Value

# Return level
func update_level_ui(level):
    #return something like 0
    value.text = str(level)
```

Now we just have to connect these UI element functions to each of our newly created signals in our Player script.

```
### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
```

```

@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount
@onready var xp_amount = $UI/XP
@onready var level_amount = $UI/Level
@onready var animation_player = $AnimationPlayer

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
    health_pickups_updated.connect(health_amount.update_health_pickup_ui)
    stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)
    xp_updated.connect(xp_amount.update_xp_ui)
    xp_requirements_updated.connect(xp_amount.update_xp_requirements_ui)
    level_updated.connect(level_amount.update_level_ui)

```

We want to update our xp amount when we've killed an enemy, so to do that we need to create a new function in our Player script that will update our xp value and that emits our *xp_updated* signal.

```

### Player.gd

# older code

# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value

    #emit signals
    xp_requirements_updated.emit(xp_requirements)
    xp_updated.emit(xp)
    level_updated.emit(level)

```

This function can then be called anywhere we want to update our xp, such as in our *add_pickups()* function or our Enemy's death conditional in their *hit()* function.

```
### Player.gd

# older code

# ----- Consumables -----
# Add the pickup to our GUI-based inventory
func add_pickup(item):
    if item == Global.Pickups.AMMO:
        ammo_pickup = ammo_pickup + 3 # + 3 bullets
        ammo_pickups_updated.emit(ammo_pickup)
        print("ammo val:" + str(ammo_pickup))
    if item == Global.Pickups.HEALTH:
        health_pickup = health_pickup + 1 # + 1 health drink
        health_pickups_updated.emit(health_pickup)
        print("health val:" + str(health_pickup))
    if item == Global.Pickups.STAMINA:
        stamina_pickup = stamina_pickup + 1 # + 1 stamina drink
        stamina_pickups_updated.emit(stamina_pickup)
        print("stamina val:" + str(stamina_pickup))
    update_xp(5)
```

```
### Enemy.gd

# older code

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        #stop movement
        timer_node.stop()
        direction = Vector2.ZERO
        #stop health regeneration
        set_process(false)
        #trigger animation finished signal
        is_attacking = true
        #Finally, we play the death animation and emit the signal for the
        spawner.
        animation_sprite.play("death")
```

```

#add xp values
player.update_xp(70)
death.emit()
#drop loot randomly at a 90% chance
if rng.randf() < 0.9:
    var pickup = Global.pickups_scene.instantiate()
    pickup.item = rng.randi() % 3 #we have three pickups in our enum

    get_tree().root.get_node("Main/PickupSpawner/SpawnedPickups").call_deferred("add_child", pickup)
    pickup.position = position

```

We'll build on this function more throughout this part, as we still need to update our *xp_requirements* and run the check to see if our player has gained enough xp to level up. If they've gained enough xp, we need to reset our current xp amount back to zero and increase our player's level and required xp values. Then we need to emit our signals to notify our game of the changes in these values.

```

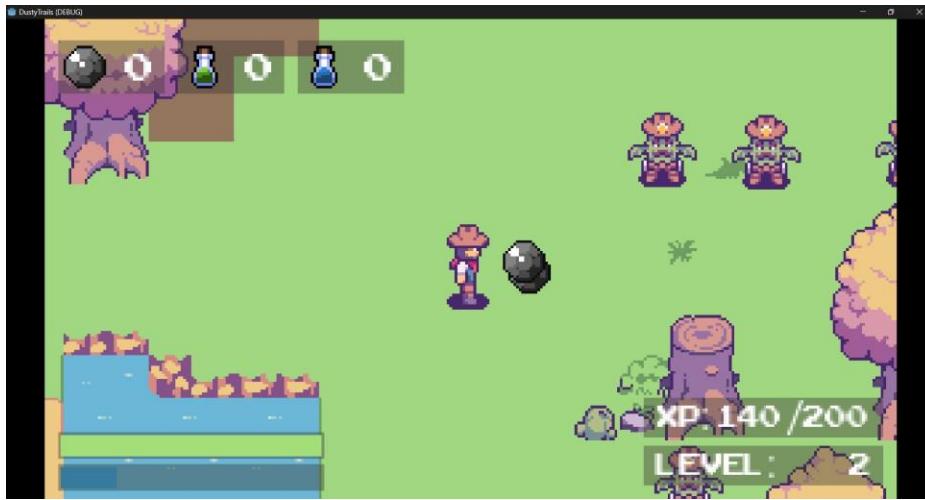
### Player.gd

# older code

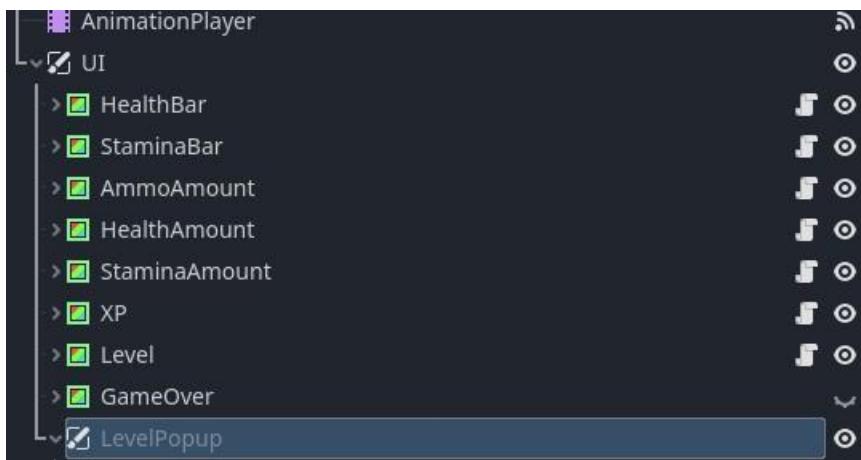
# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        #reset xp to 0
        xp = 0
        #increase the level and xp requirements
        level += 1
        xp_requirements *= 2
    #emit signals
    xp_requirements_updated.emit(xp_requirements)
    xp_updated.emit(xp)
    level_updated.emit(level)

```

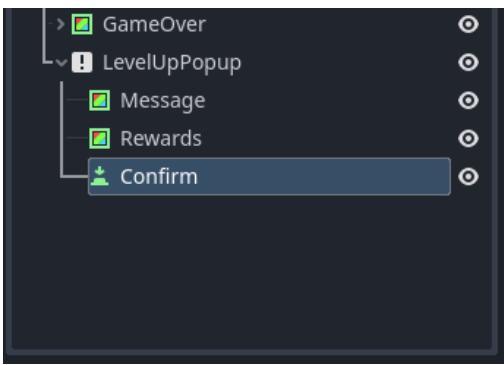
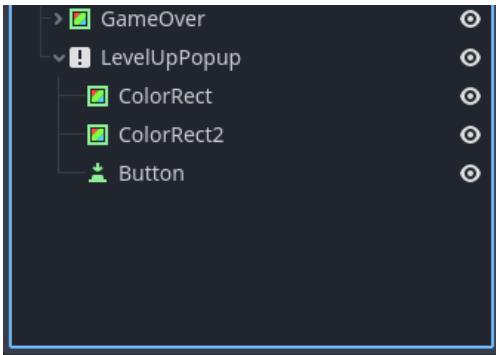
If you run your scene now and you kill some enemies (make sure you change their damage value to zero so that they can't kill you during this test), you will see that your xp and level values update.



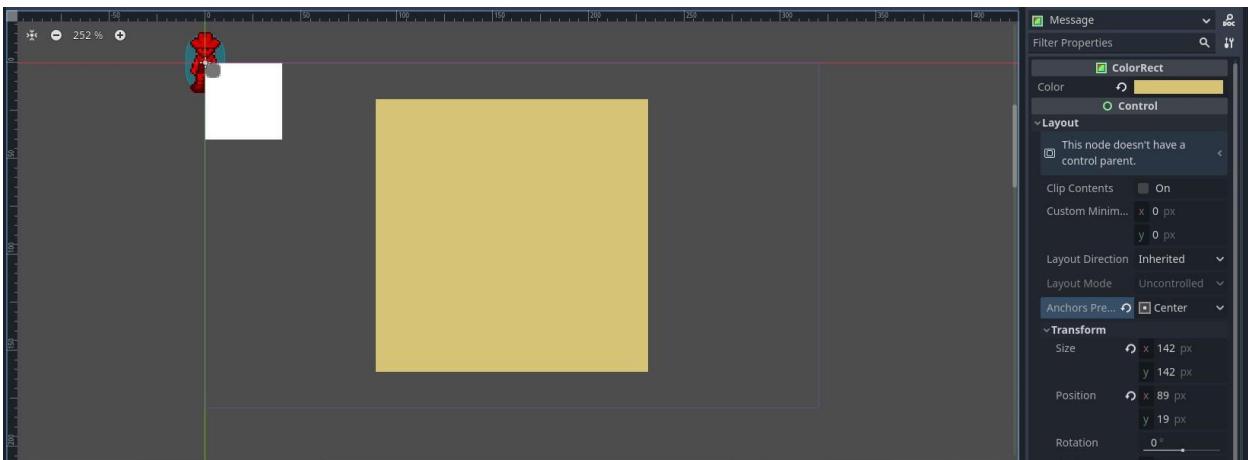
If our player levels up, we want a screen to show with a notification that our player has leveled up, plus a summary of the rewards they've gained from doing so. We will accomplish this by adding a `CanvasLayer` node to our Player's `UI` node. In your Player scene, add a new `CanvasLayer` node to your `UI` layer. Rename it to `LevelUpPopup`.



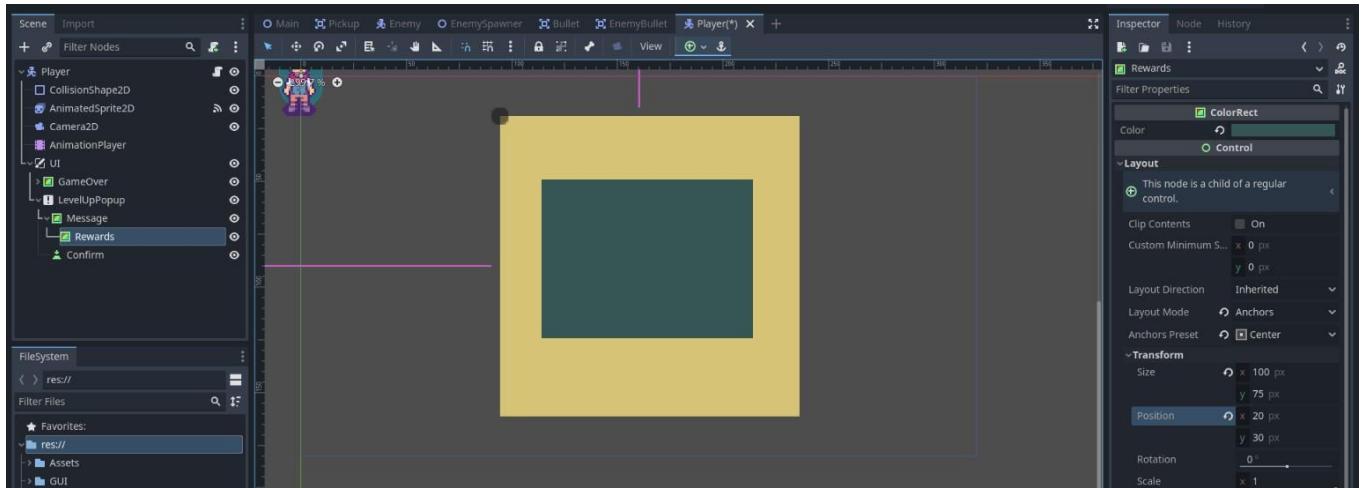
In this `CanvasLayer` node, add two `ColorRects` and a [Button](#) node. The first `ColorRect` will contain our level-up label, and the second `ColorRect` will contain the summary of our rewards. The button will allow the player to confirm the notification and continue with their game. You can rename them as follows:



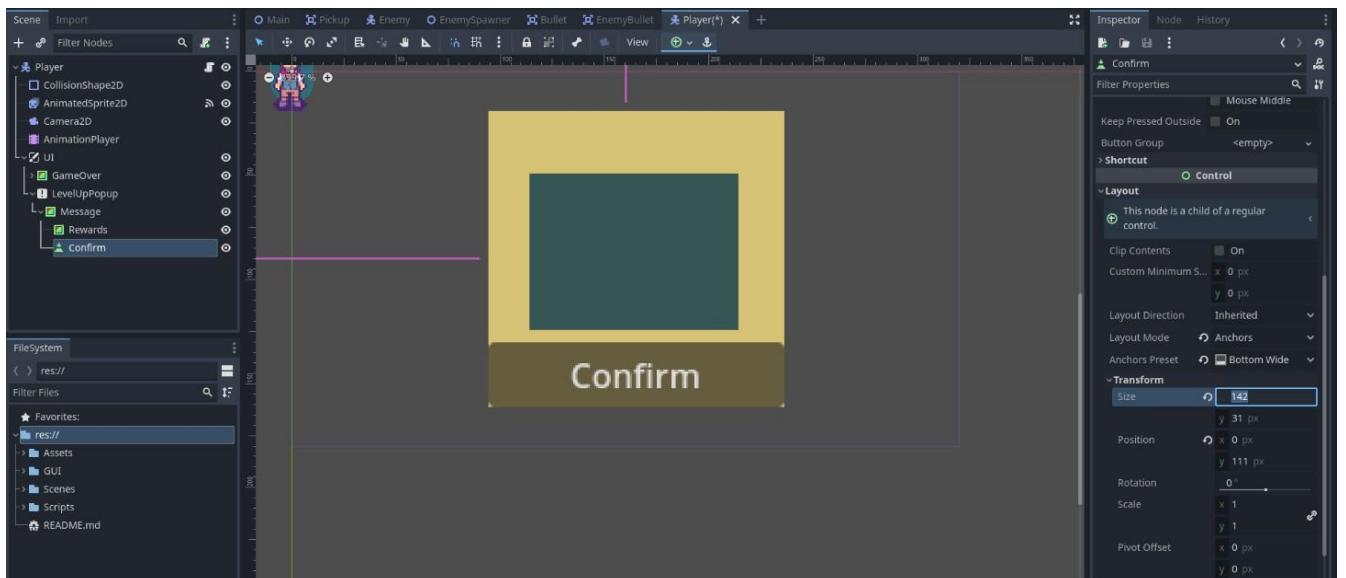
Change your Message node's color to #d6c376, and its size (x: 142, y: 142); position (x: 89, y: 19); anchor_preset (center).

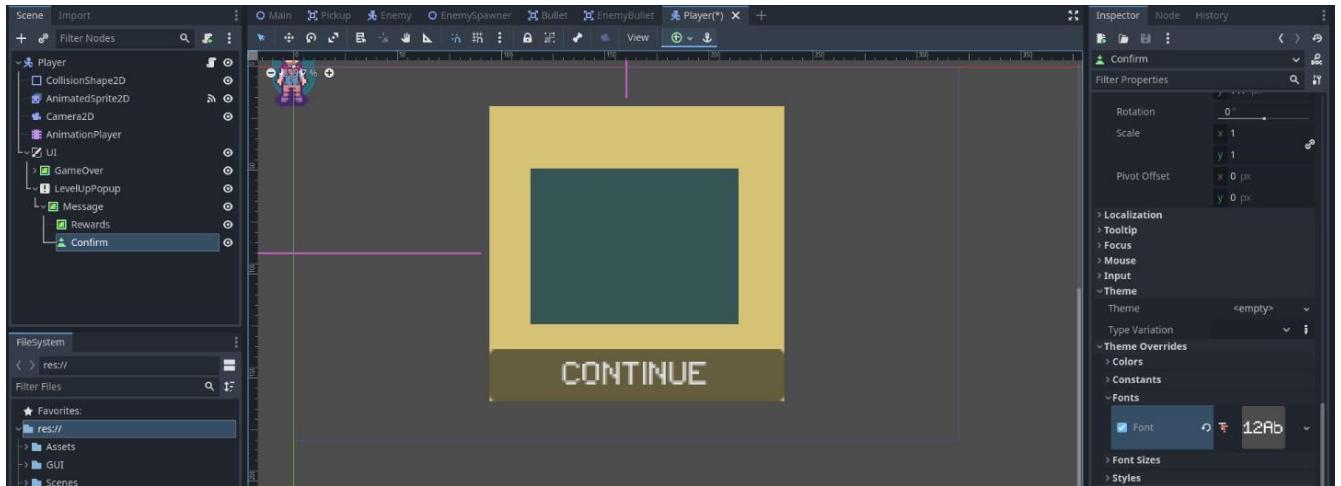


Drag your Rewards node into your Message node to turn it into a child of that node. Change its color to #365655, and its size (x: 100, y: 75); position (x: 20, y: 30); anchor_preset (center).

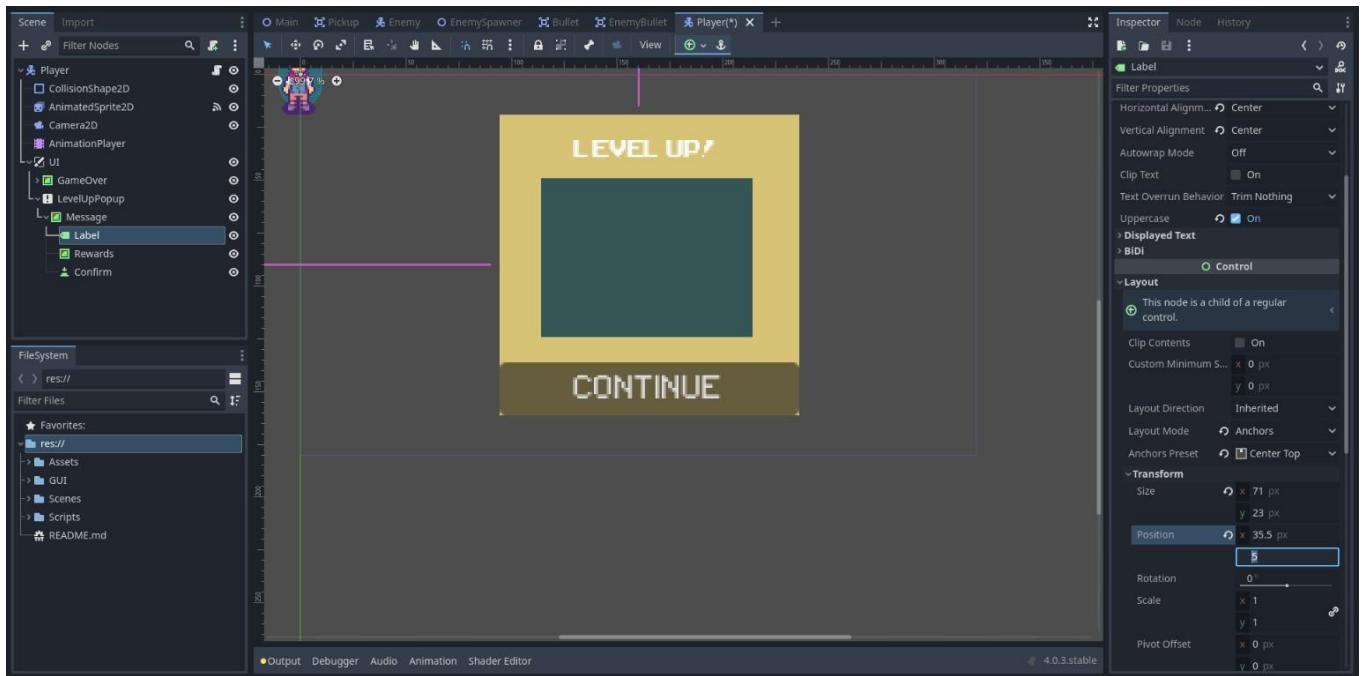


Also, drag your Confirm node into your Message node to turn it into a child of that node. Change its color to #365655, and its size (x: 142, y: 31); position (x: 0, y: 111); anchor_preset (bottom-wide). Change its font to "Schrodinger" and its text to "CONTINUE".

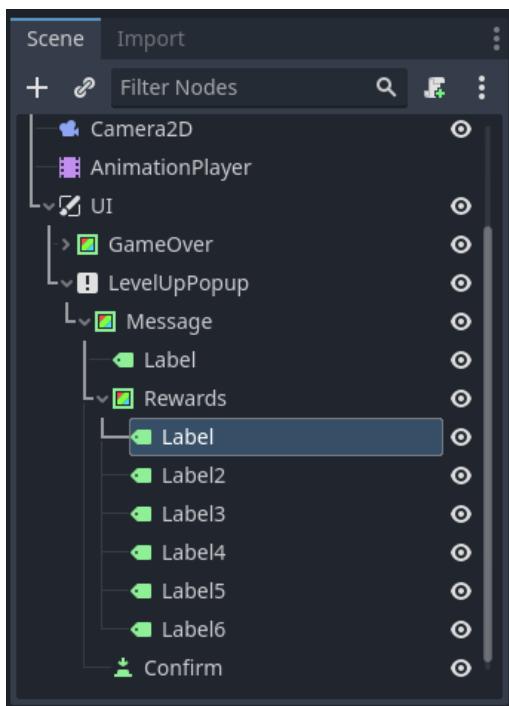




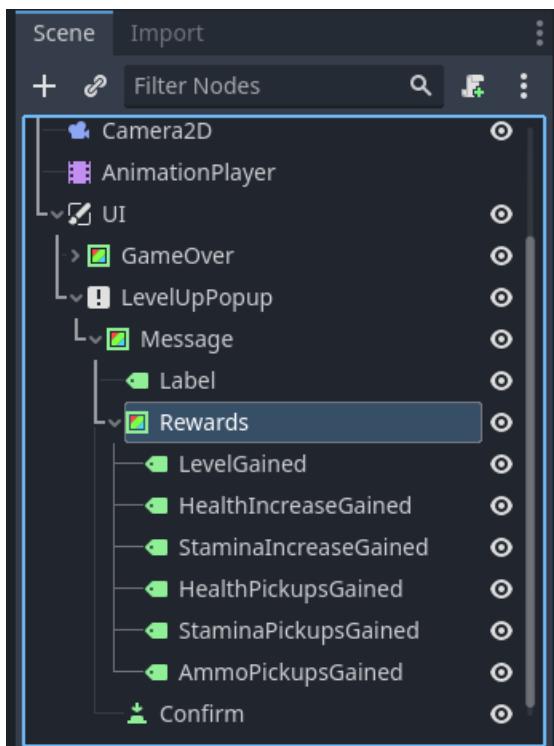
Now, in your Message node, at the top, let's add a new Label node to display our welcome text "Level Up!". Change its font to "Arcade Classic" and its font size to 15. Then change its anchor_preset (center-top); horizontal alignment (center); vertical alignment (center); and position (y: 5).



In your Rewards node, add six new Label nodes.



Rename them as follows:



Change their properties as follows:

- **All of them:**

- Text = "1"
- Font = "Schrödinger"
- Font-size = 10
- Anchor Preset = center-top
- Horizontal Alignment = center
- Vertical Alignment = center

- **LevelGained:**

- Position= y: 0

- **HealthIncreaseGained:**

- Position= y: 10

- **StaminaIncreaseGained:**

- Position= y: 20

- **HealthPickupsGained:**

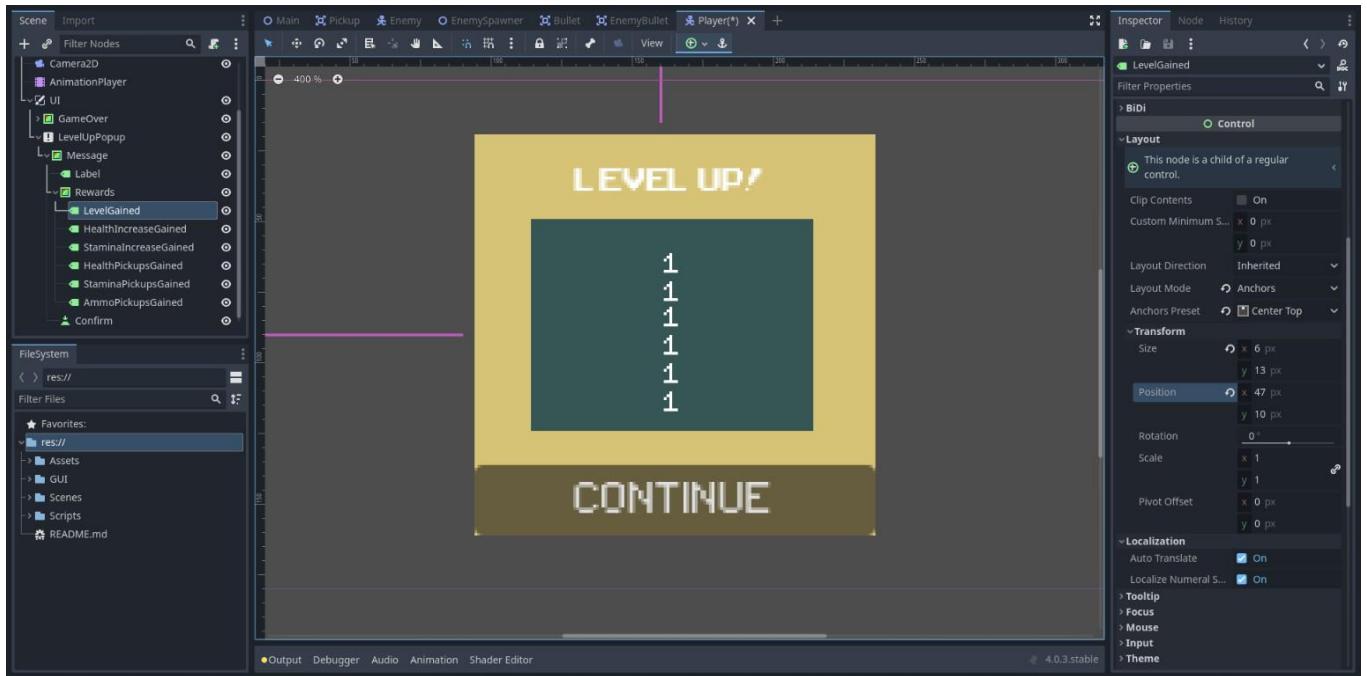
- Position= y: 30

- **StaminaPickupsGained:**

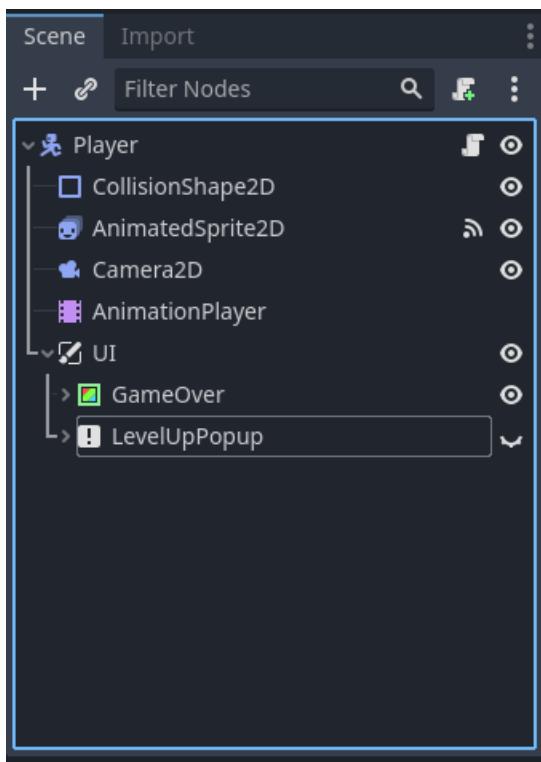
- Position= y: 40

- **AmmoPickupsGained:**

- Position= y: 50

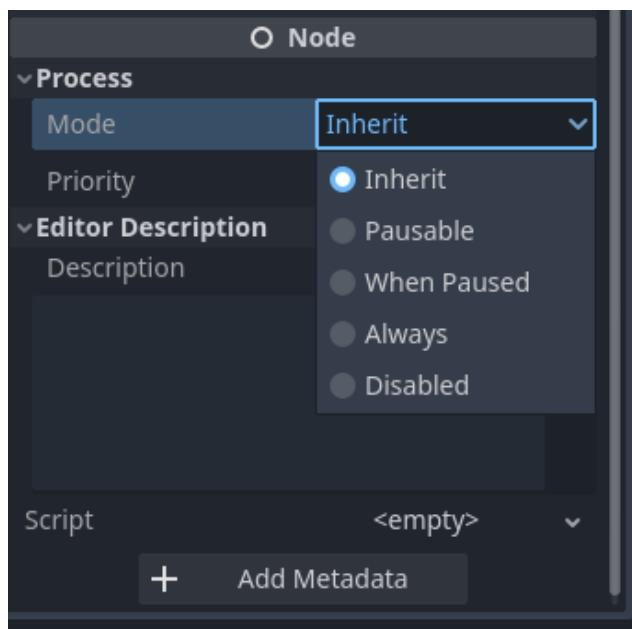


With our Popup created, we can go ahead and hide our popup for now. You can do this in the Inspector panel underneath Canvas Item > Visibility, or just click the eye icon next to the node to hide it.



We need to go back to our `update_xp()` function in our Player scene to update our conditional that checks if the player levels up. If they are leveling up, we need to pause the game, display the popup with all of the reward values and only hide the popup if the player clicks the confirm button. In this function we will have to update the player's max health and stamina, as well as give them some ammo, and health and stamina drinks. After we've done this, we'll need to reflect these changes in our UI elements.

If we want to pause the game, we simply use the [SceneTree.paused](#) method. If the game is paused, no input from the player or us will be accepted, because everything is paused. That is unless we change our node's process mode. Each node in Godot has a "[Process Mode](#)" that defines when it processes. It can be found and changed under a node's [Node](#) properties in the inspector.

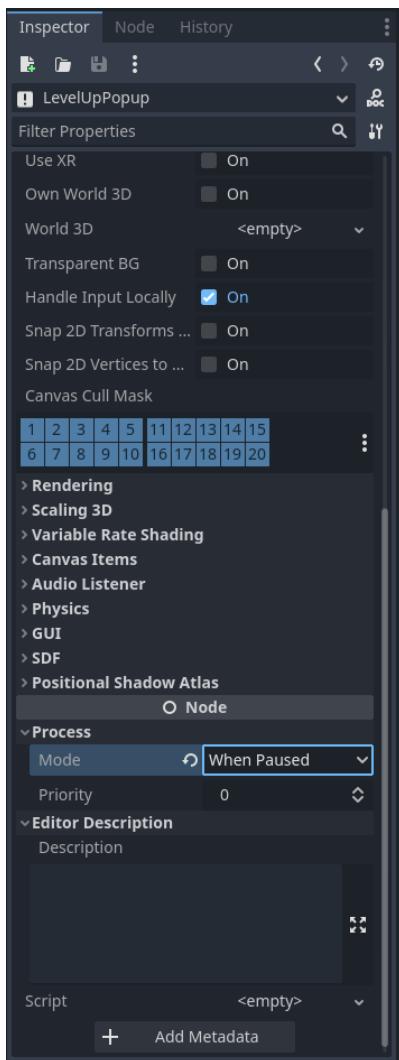


What are the available Processing Modes?

- **Inherit:** Process depending on the state of the parent, grandparent, etc. The first parent has a non-inherit state.
- **Pausable:** Process the node (and its children in Inherit mode) only when the game is not paused.

- **WhenPaused**: Process the node (and its children in Inherit mode) *only* when the game is paused.
- **Always**: Process the node (and its children in Inherit mode) no matter what. Paused or not, this node will process.
- **Disabled**: The node (and its children in Inherit mode) will not process at all.

We need our LevelUpPopup node to only work when the game is paused. This will allow us to click the Confirm button to unpause the game, thus allowing the other nodes to continue processing since they all only work or process input if the game is in an unpause state. Let's change our LevelUpPopup's Process Mode to **WhenPaused**. You can find this option under Node > Process > Mode.



Because we changed the LevelUpPopup node's process mode, all of its children will also inherit that processing mode, so all of them will work when the game is paused. Before we pause our game in our code, we'll also need to first allow input to be processed via the [set_process_input](#) method. This method enables or disables input processing. Then we will increase our health, stamina, xp, level and pickup values and reflect these changes on our UI! Let's make these changes in our code.

```
### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount
@onready var xp_amount = $UI/XP
@onready var level_amount = $UI/Level
@onready var animation_player = $AnimationPlayer
@onready var level_popup = $UI/LevelPopup

# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        #allows input
        set_process_input(true)
        #pause the game
        get_tree().paused = true
        #make popup visible
        level_popup.visible = true
        #reset xp to 0
        xp = 0
        #increase the level and xp requirements
        level += 1
        xp_requirements *= 2

        #update their max health and stamina
```

```

max_health += 10
max_stamina += 10

#give the player some ammo and pickups
ammo_pickup += 10
health_pickup += 5
stamina_pickup += 3

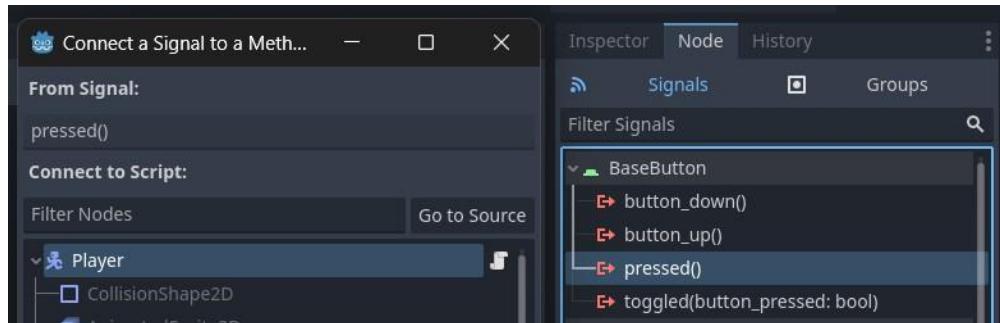
#update signals for Label values
health_updated.emit(health, max_health)
stamina_updated.emit(stamina, max_stamina)
ammo_pickups_updated.emit(ammo_pickup)
health_pickups_updated.emit(health_pickup)
stamina_pickups_updated.emit(stamina_pickup)
xp_updated.emit(xp)
level_updated.emit(level)

#reflect changes in Label
$UI/LevelPopup/Message/Rewards/LevelGained.text = "LVL: " + str(level)
$UI/LevelPopup/Message/Rewards/HealthIncreaseGained.text = "+ MAX HP: " +
str(max_health)
$UI/LevelPopup/Message/Rewards/StaminaIncreaseGained.text = "+ MAX SP: " +
str(max_stamina)
$UI/LevelPopup/Message/Rewards/HealthPickupsGained.text = "+ HEALTH: 5"
$UI/LevelPopup/Message/Rewards/StaminaPickupsGained.text = "+ STAMINA: 3"
$UI/LevelPopup/Message/Rewards/AmmoPickupsGained.text = "+ AMMO: 10"

#emit signals
xp_requirements_updated.emit(xp_requirements)
xp_updated.emit(xp)
level_updated.emit(level)

```

Finally, we need to give our Confirm button the ability to close the popup and unpause our game. We can do this by connecting its *pressed()* signal to our Player script.



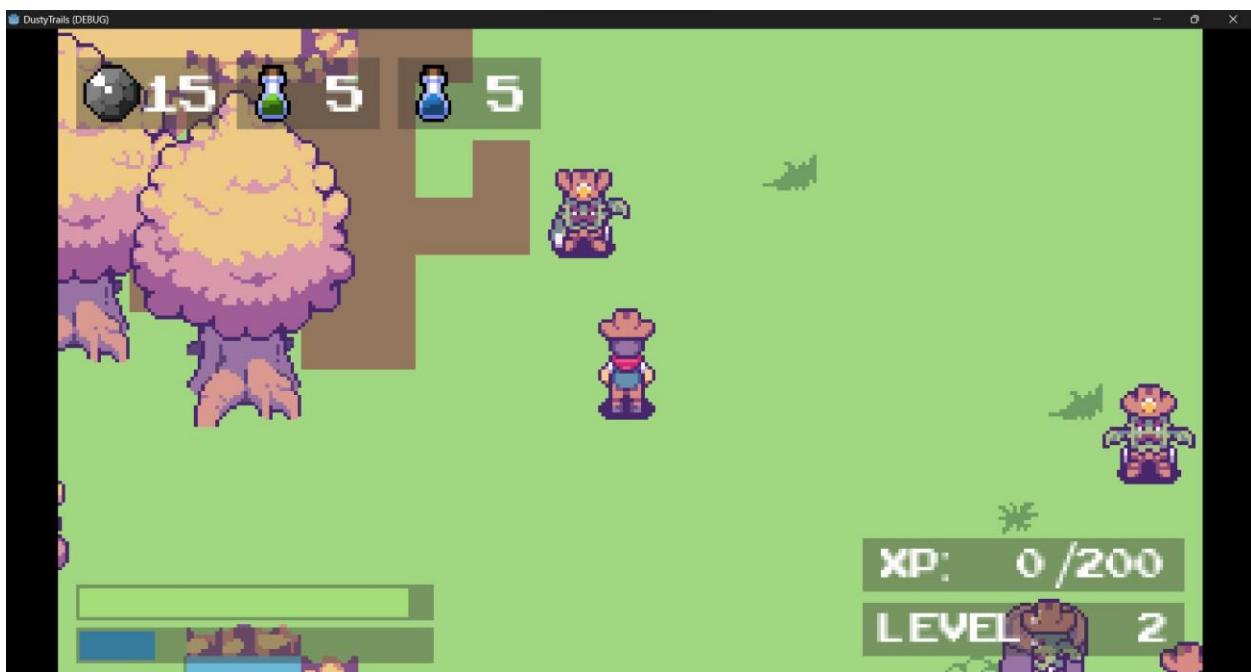
In this newly created `_on_confirm_pressed()`: function, we will simply hide the popup again and unpause the game.

```
### Player.gd

# older code

# close popup
func _on_confirm_pressed():
    level_popup.visible = false
    get_tree().paused = false
```

Now if we run our scene and we shoot enough enemies, we will see the popup appear with our rewards values, and if we click on the confirm button, our popup closes, and we can continue the game with our new values!





SHOWING & HIDING CURSOR

I don't like the way our cursor always shows. Whether or not the game is paused, our cursor can always be found lingering on our screen. Since this is not a point-and-click game, there is no reason for our cursor to show when our game is not paused, since we'll spend our time running around and shooting enemies. Our cursor should hence only show if we are in a menu screen such as a pause or main menu, or even during our dialog screen - i.e. when the game is paused and we need the cursor for input.

Luckily, this is a quick fix. We can change the visibility of our mouse's cursor via our Input singleton's [MouseMode](#) method. In our Player's script we will show the cursor whenever game is paused, and if it is not paused then we will hide the cursor.

```
### Player.gd

# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
```

```

if xp >= xp_requirements:
    #allows input
    set_process_input(true)
    Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
    #pause the game

#emit signals
xp_requirements_updated.emit(xp_requirements)
xp_updated.emit(xp)
level_updated.emit(level)

# close popup
func _on_confirm_pressed():
    level_popup.visible = false
    get_tree().paused = false
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

```

We also need to hide our cursor on load.

```

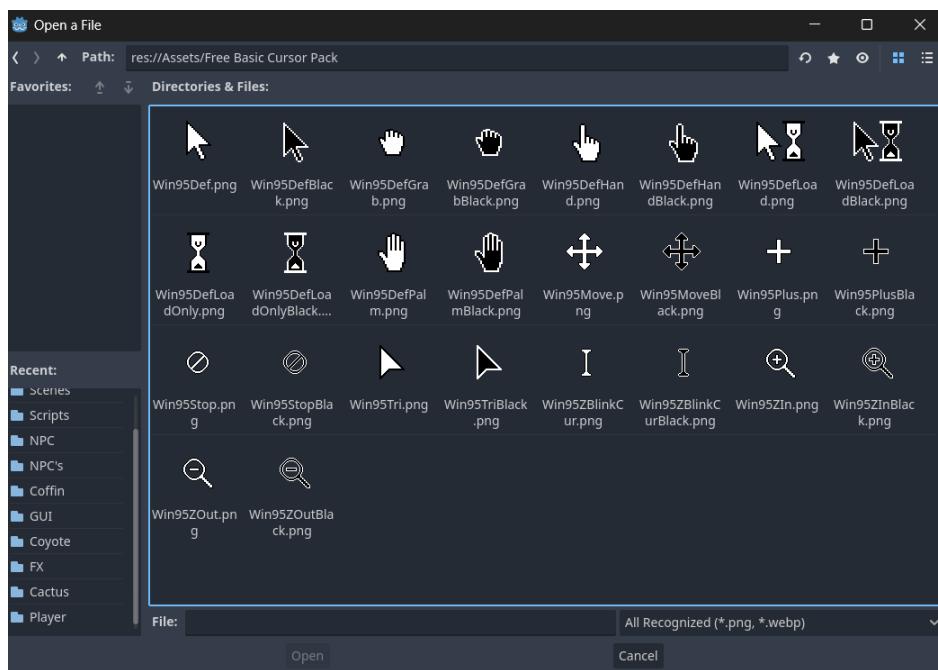
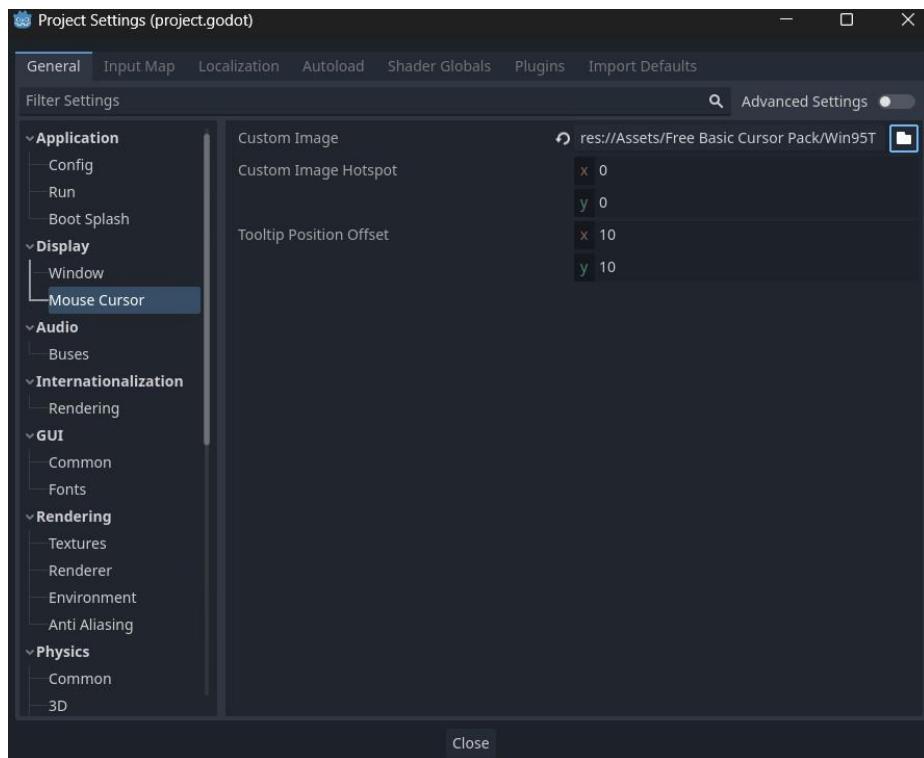
### Player.gd

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
    health_pickups_updated.connect(health_amount.update_health_pickup_ui)
    stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)
    xp_updated.connect(xp_amount.update_xp_ui)
    xp_requirements_updated.connect(xp_amount.update_xp_requirements_ui)
    level_updated.connect(level_amount.update_level_ui)

    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

```

We can also change our cursor's image. If you go into your Project Settings > Display > Mouse Cursor, you can change your mouse cursor's image.



Now if you run your game, your cursor should be hidden/shown when your pause state changes.

SHOWING VALUES ON LOAD

If we change the values of our variables and we run our game, you might've noticed that your Level, XP, and Pickup values aren't updating. We need to fix this by going into our UI scripts and calling our values from our Player script on load.

```
### HealthAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value
@onready var player = $".../.."

# Show correct value on load
func _ready():
    value.text = str(player.health_pickup)

# Update ui
func update_health_pickup_ui(health_pickup):
    value.text = str(health_pickup)
```

```
### StaminaAmount.gd
extends ColorRect

# Node ref
@onready var value = $Value
@onready var player = $".../.."

# Show correct value on load
func _ready():
    value.text = str(player.stamina_pickup)

# Update ui
func update_stamina_pickup_ui(stamina_pickup):
    value.text = str(stamina_pickup)
```

```
### LevelAmount.gd
extends ColorRect

# Node refs
@onready var value = $Value
```

```

@onready var player = $".../..."

# On load
func _ready():
    value.text = str(player.level)

# Return level
func update_level_ui(level):
    #return something like 0
    value.text = str(level)

```

```

### XPAmount.gd

extends ColorRect

# Node refs
@onready var value = $value
@onready var value2 = $value2
@onready var player = $".../..."

# On load
func _ready():
    value.text = str(player.xp)
    value2.text = "/" + str(player.xp_requirements)

#return xp
func update_xp_ui(xp):
    #return something like 0
    value.text = str(xp)

#return xp_requirements
func update_xp_requirements_ui(xp_requirements):
    #return something like / 100
    value2.text = "/" + str(xp_requirements)

```

```

### AmmoAmount.gd
extends ColorRect

# Node ref
@onready var value = $value
@onready var player = $".../..."

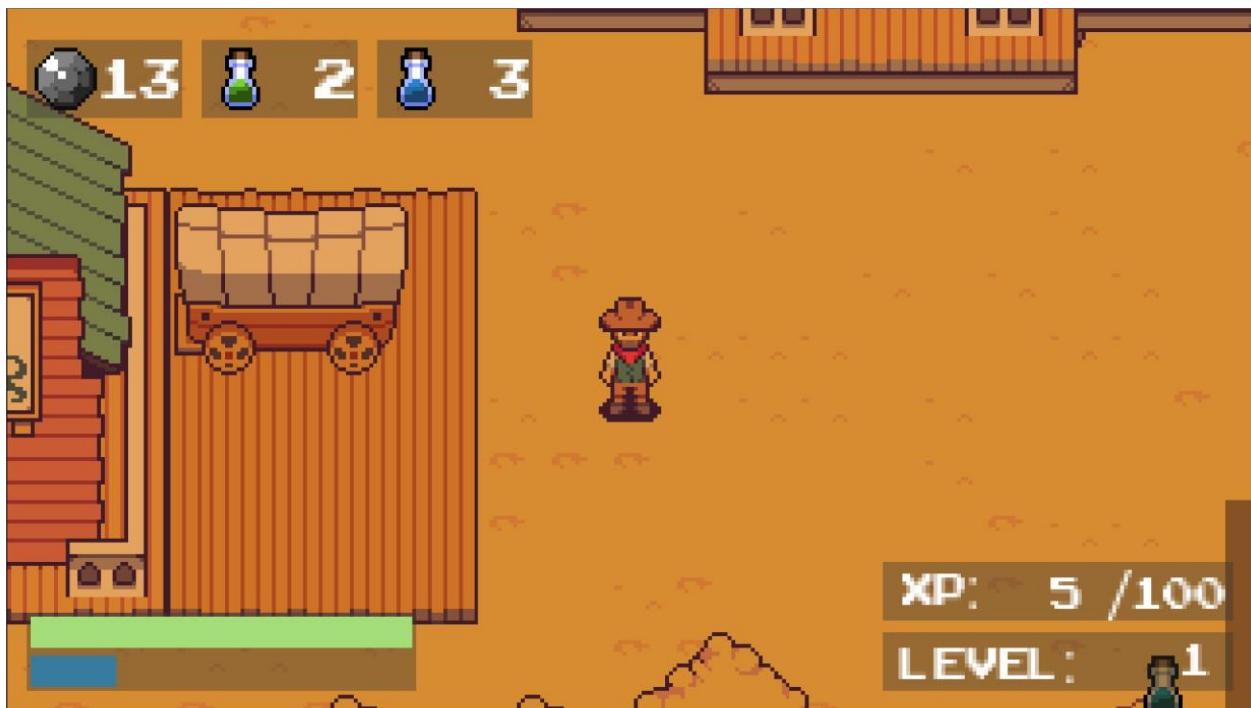
# Show correct value on load
func _ready():

```

```
value.text = str(player.ammo_pickup)

# Update ui
func update_ammo_pickup_ui(ammo_pickup):
    value.text = str(ammo_pickup)
```

Now if you run your scene, your values should show correctly, which will be useful when we load our game later on!



Congratulations, your player can now get rewarded for killing bad guys. We're still not 100% done with this, for in the next part we will add a basic NPC and quest that will also reward our player with XP upon completing this quest. Remember to save your project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 17: BASIC NPC & QUEST

Now that we have most of our game set up, we can go ahead and create an NPC with a Quest. Please note that this quest system will not be a dynamic system with quest chains. No, unfortunately, this quest system only contains a simple collection quest with a dialog and state tree. I left it to be this simple system because a dialog, quest, and even inventory system is quite complex, and therefore I will make a separate tutorial series that will focus on those advanced concepts in isolation!

WHAT YOU WILL LEARN IN THIS PART:

- How to work with game states.
- How to add dialog trees.
- Further practice with popups and animations.
- How to work with the InputEventKey object.
- How to add Groups to nodes.
- How to work with the RayCast2D node.

If you are curious about how these systems might work, below are some written resources that could give you an idea of how to implement this.

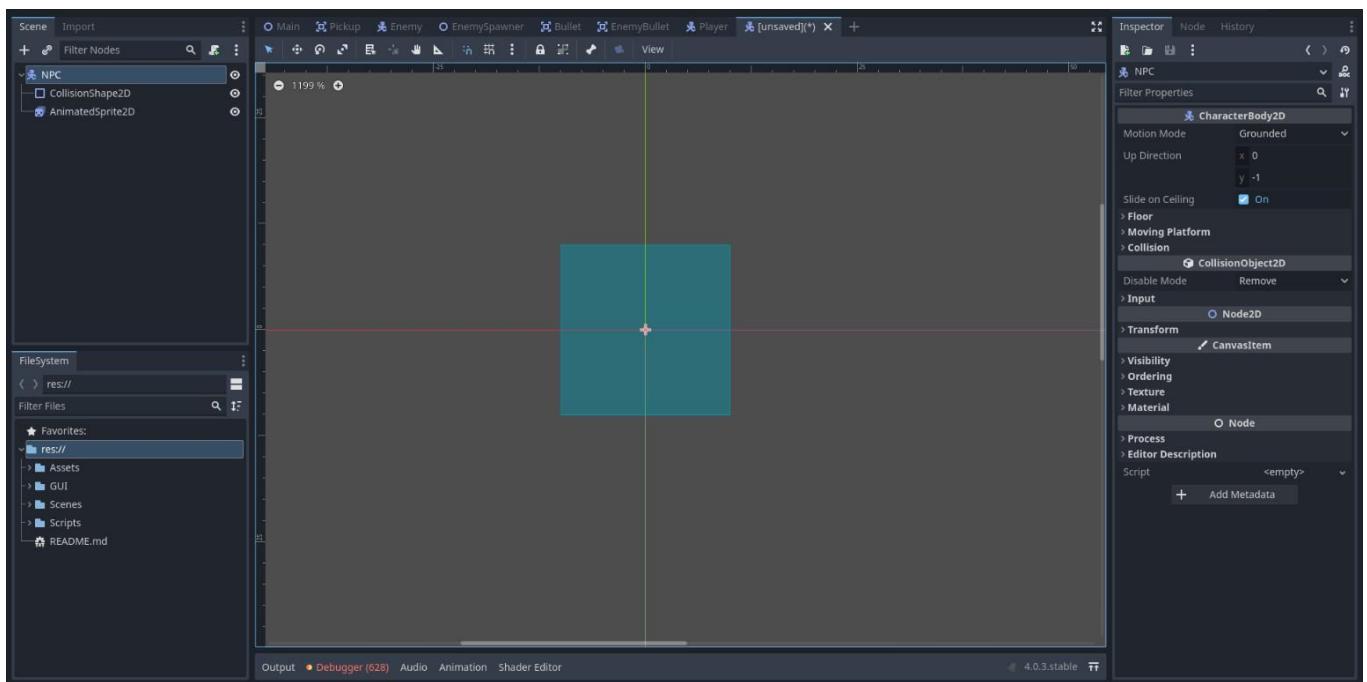
Further Resources:

- <https://github.com/christinec-dev/DustyTrailsParts/tree/main/Notes>
- https://www.reddit.com/r/godot/comments/m3ghec/how_to_build_a_dialog_system_without_selling_your/
- <https://wordeater-dev.itch.io/bittersweet-birthday/devlog/224241/howto-a-simple-dialogue-system-in-godot>

- <https://www.linkedin.com/advice/1/what-best-ways-design-dynamic-immersive-games-open-world>
- <https://medium.com/@thrivevolt/making-a-grid-inventory-system-with-godot-727efedb71f7>

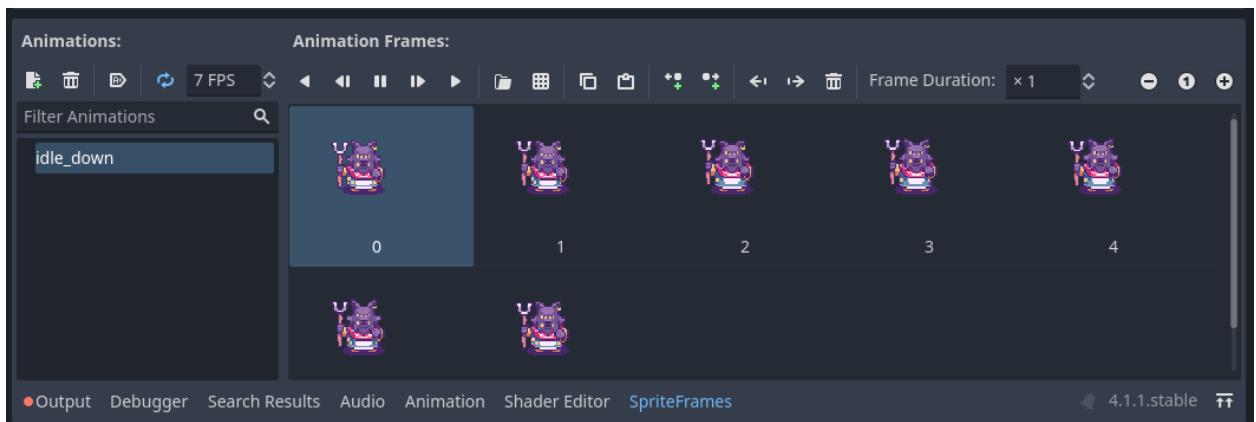
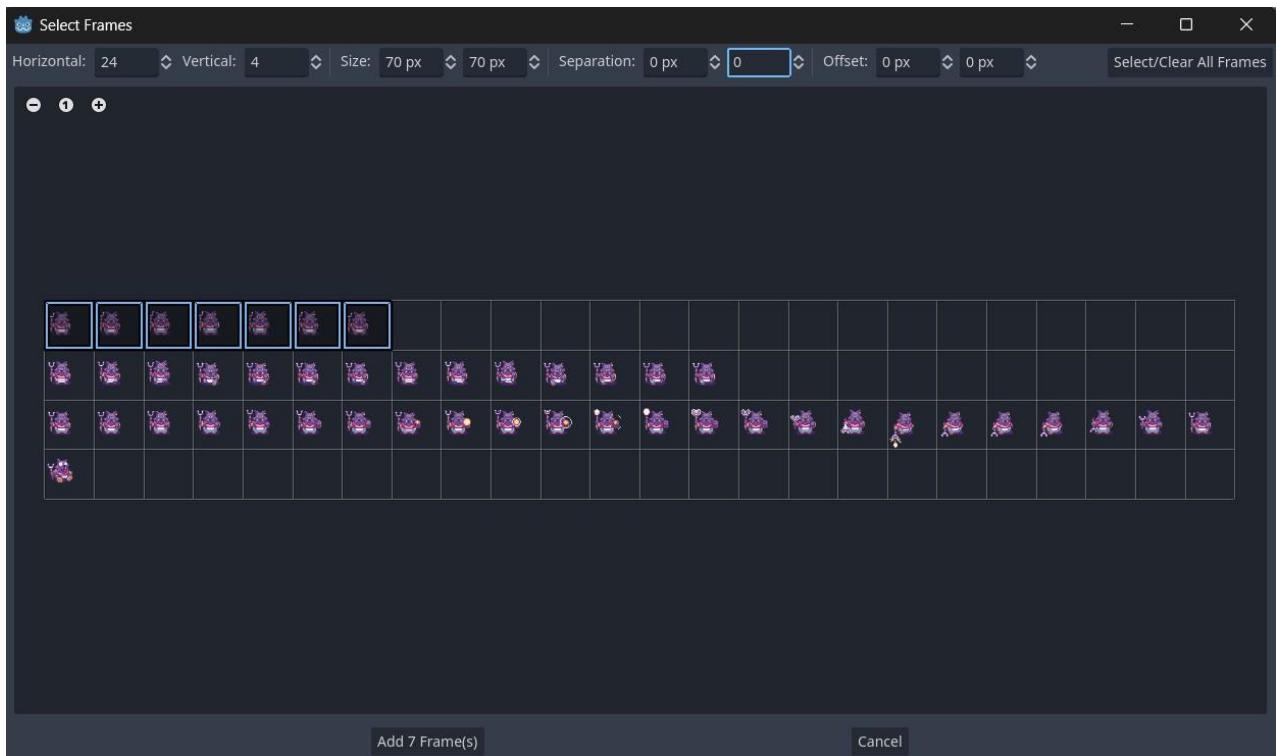
NPC SETUP

Let's create our NPC scene. In your project, create a new scene with a CharacterBody2D node as the scene root, and an AnimatedSprite2D and CollisionBody2D node as its children. Make the collision shape for your collision node to be a RectangleShape. Save this scene underneath your Scenes folder.

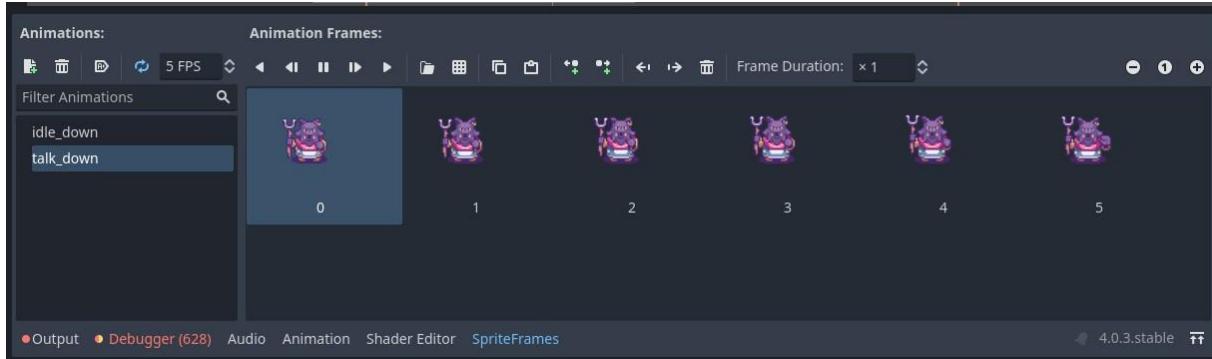
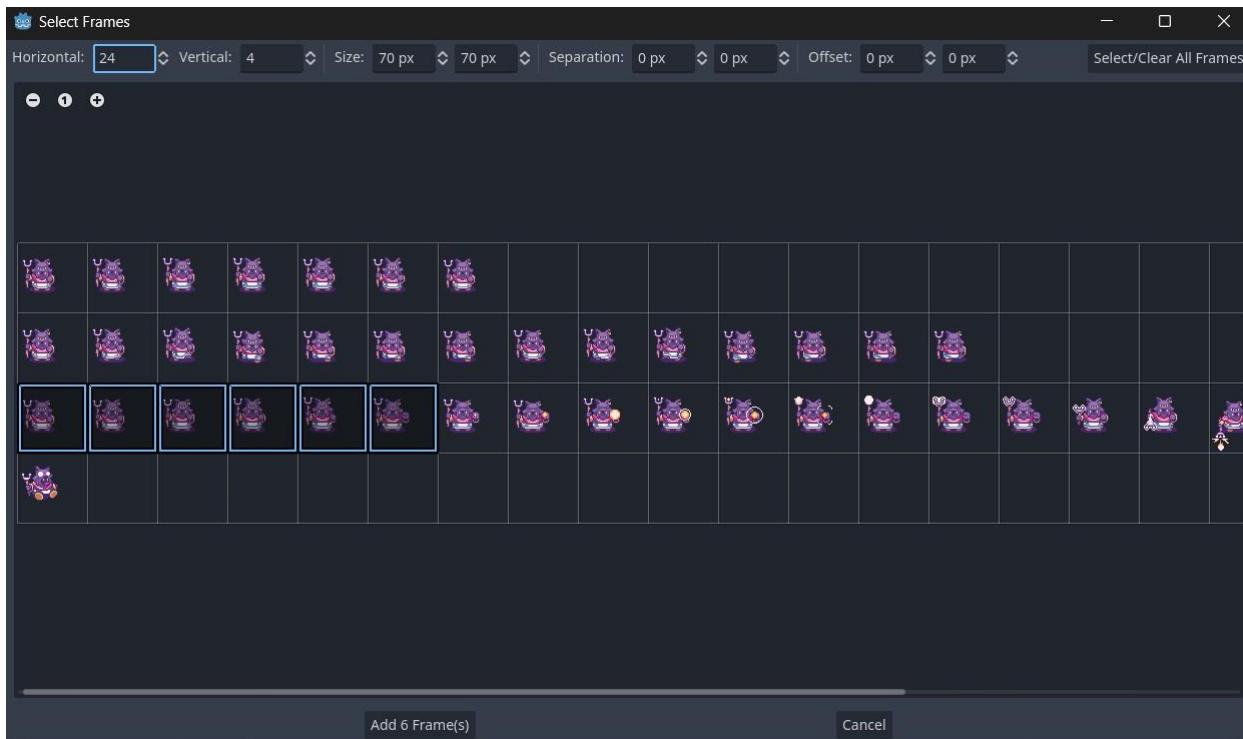


Let's add two new animations to the AnimatedSprite2D node: idle_down and talk_down. You can find the animations spritesheet that I used for this node underneath Assets > Mobs > Coyote.

idle_down:

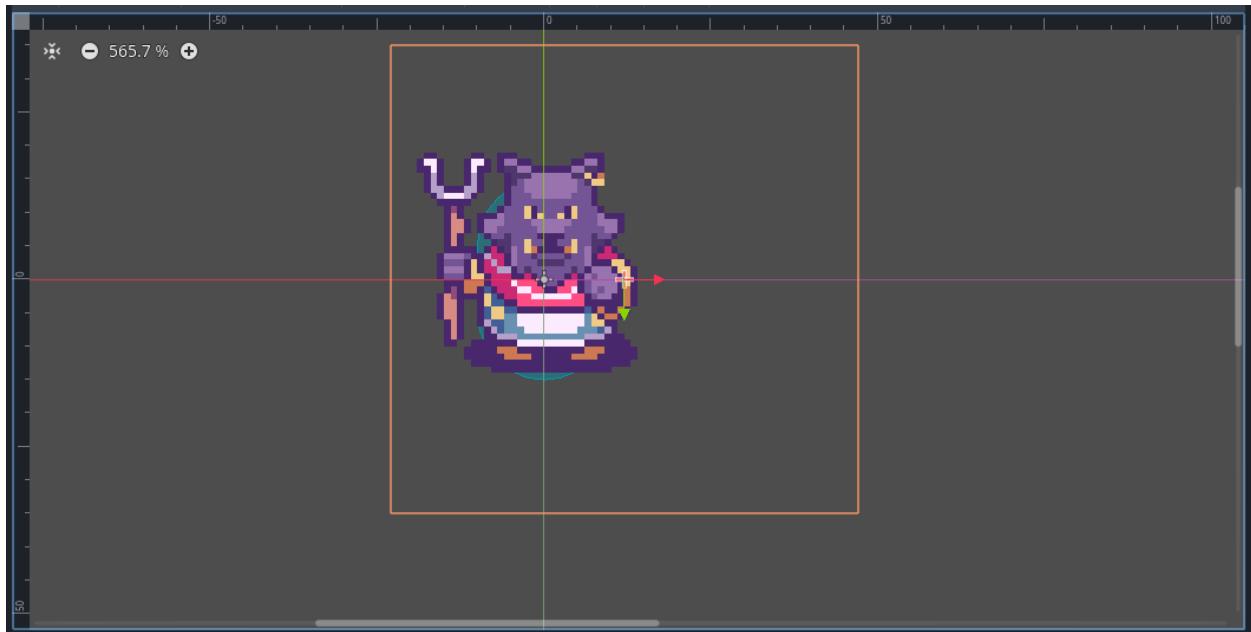


talk_down:

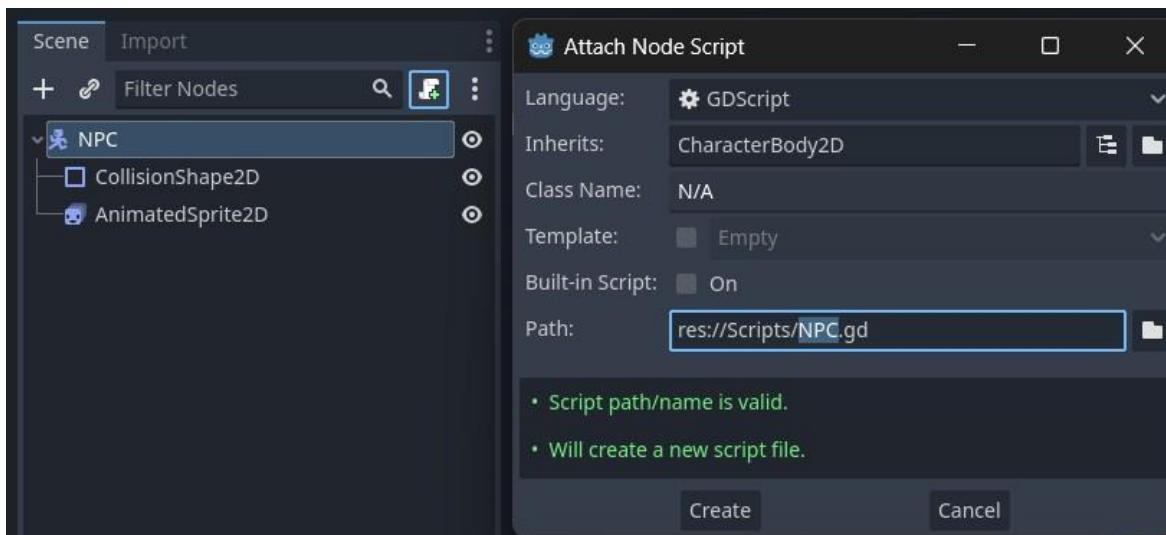


For now, our NPC will be anchored in one place, and they won't attack our enemies, so therefore they will only have *down* animations. Leave the FPS and Looping as their default values.

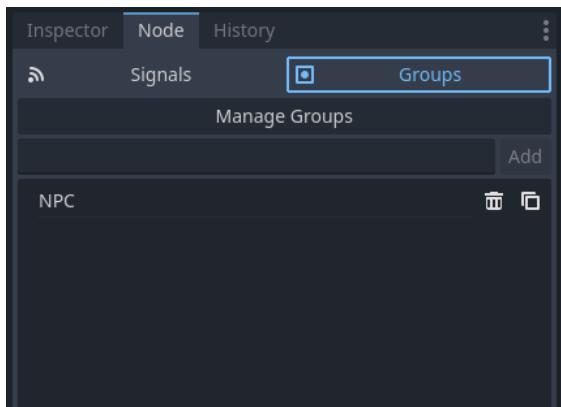
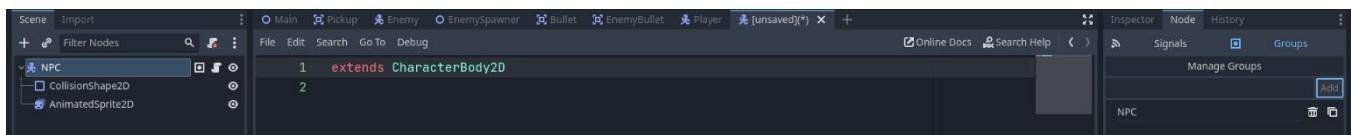
Please make sure that your sprite is in the middle of your CollisionShape.



Attach a script to your NPC scene and save it underneath your Scripts folder.

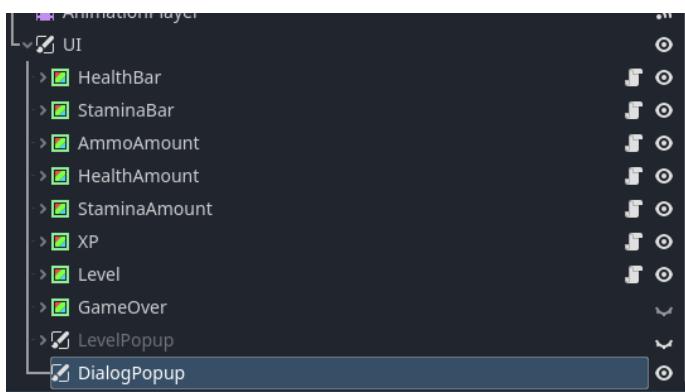


We also want to add a [Group](#) to this node so that our players can check for the special "NPC" group when they interact with it. We will have to also add a RayCast2D node to our player so that we can check its "target", and if that target is part of the NPC group, we'll launch the function to talk to the NPC.

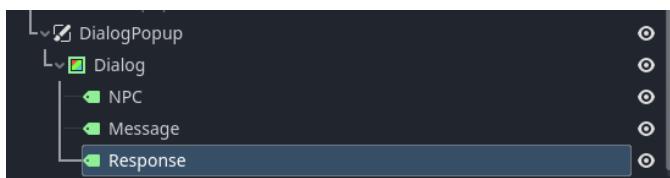


DIALOG POPUP & PLAYER SETUP

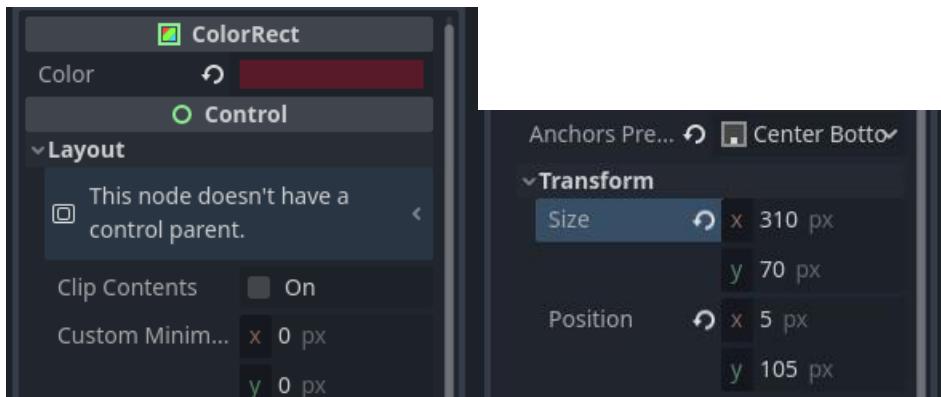
Now, in your Player scene, we need to create another popup node that will be made visible when the player interacts with the NPC. Add a new CanvasLayer node and rename it to "DialogPopup".



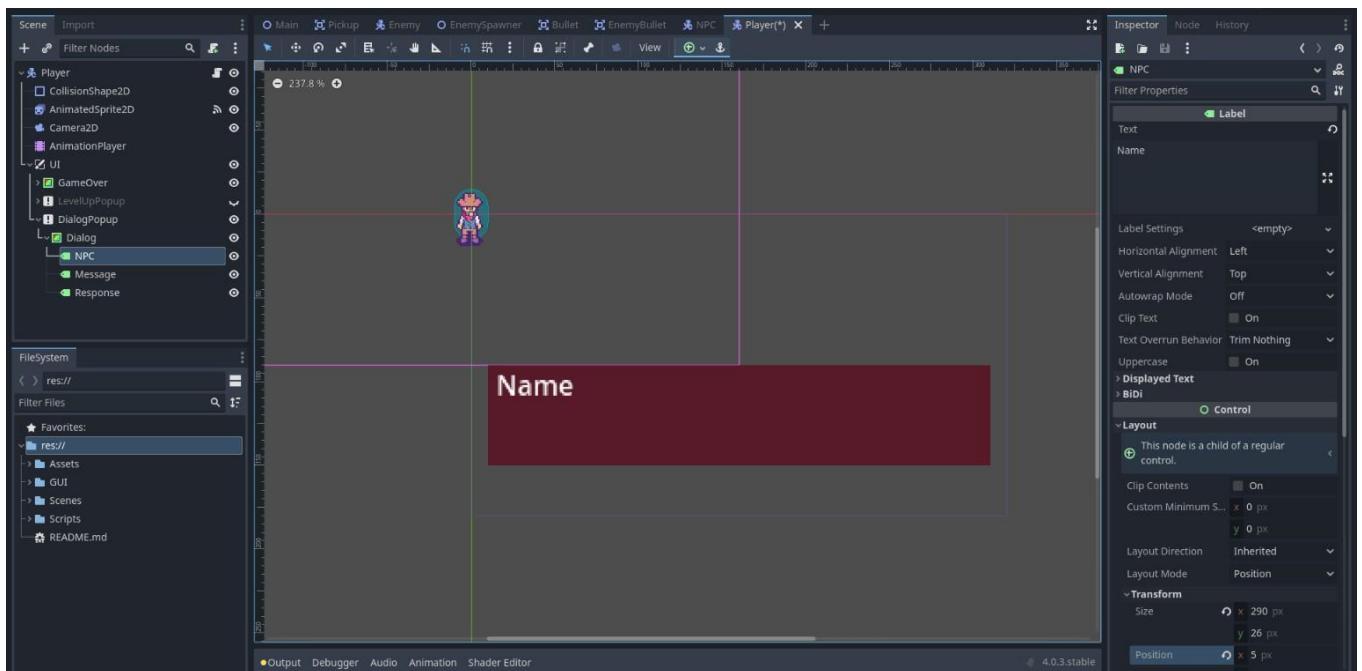
In this DialogPopup node, let's add a ColorRect with three Label nodes as its children. Rename it as follows:



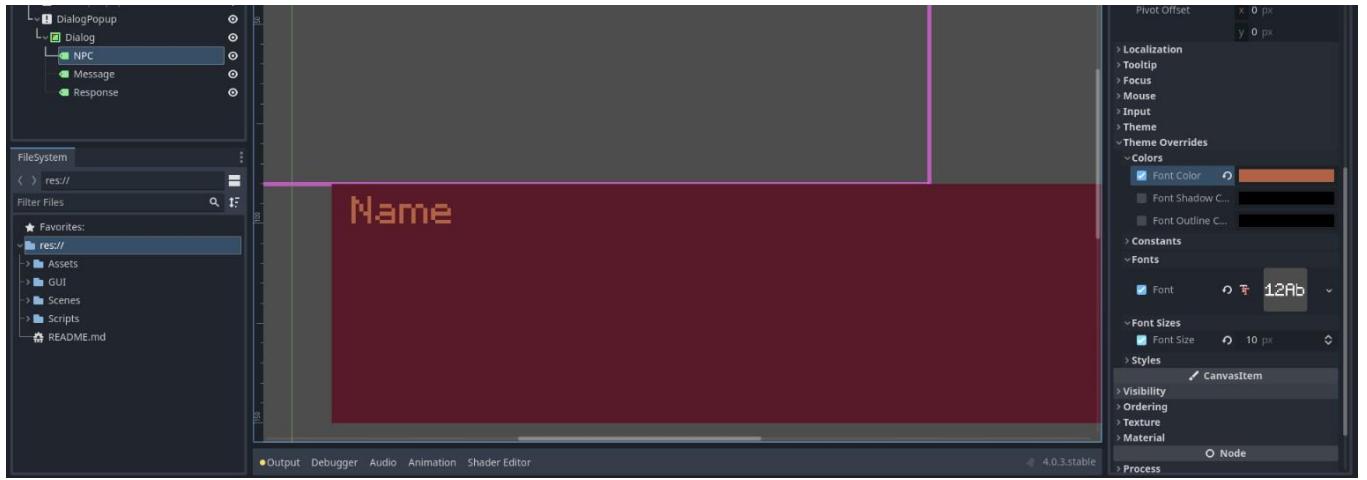
Select your Dialog node and change its Color property to #581929. Change its size (x: 310, y: 70), and its anchor_preset (center_bottom).



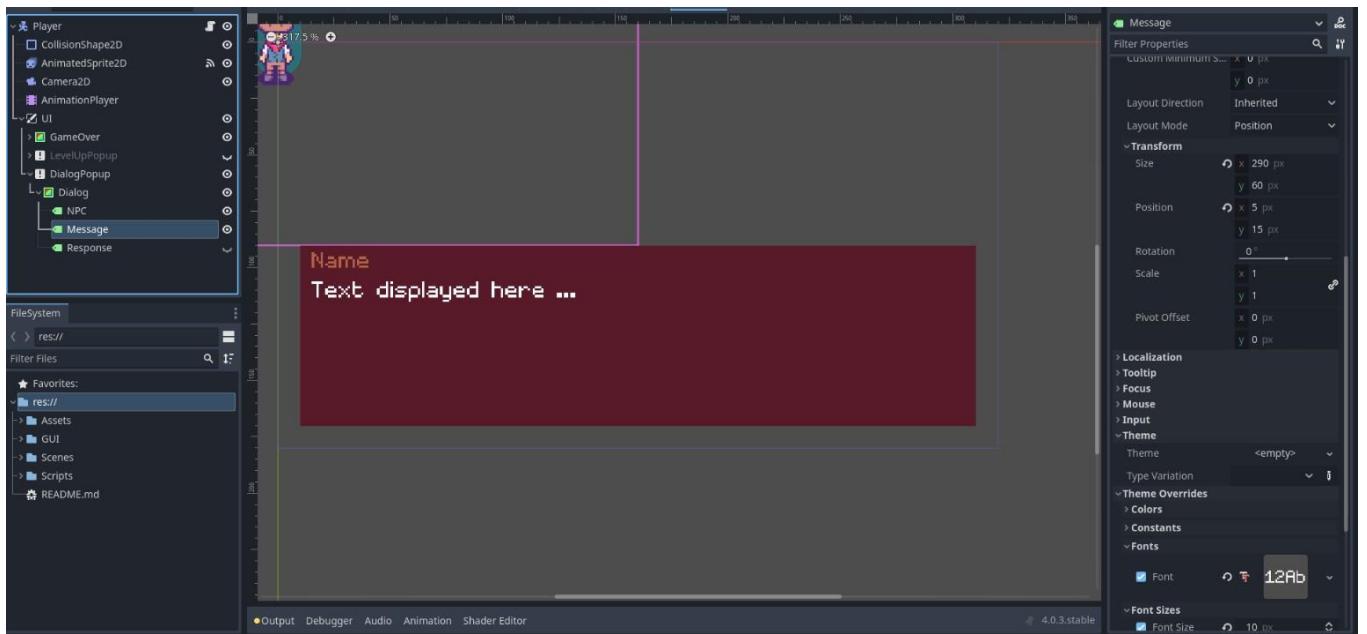
Select your NPC Label node and change its text to "Name". Change its size (x: 290, y: 26); position (x: 5, y: 2).



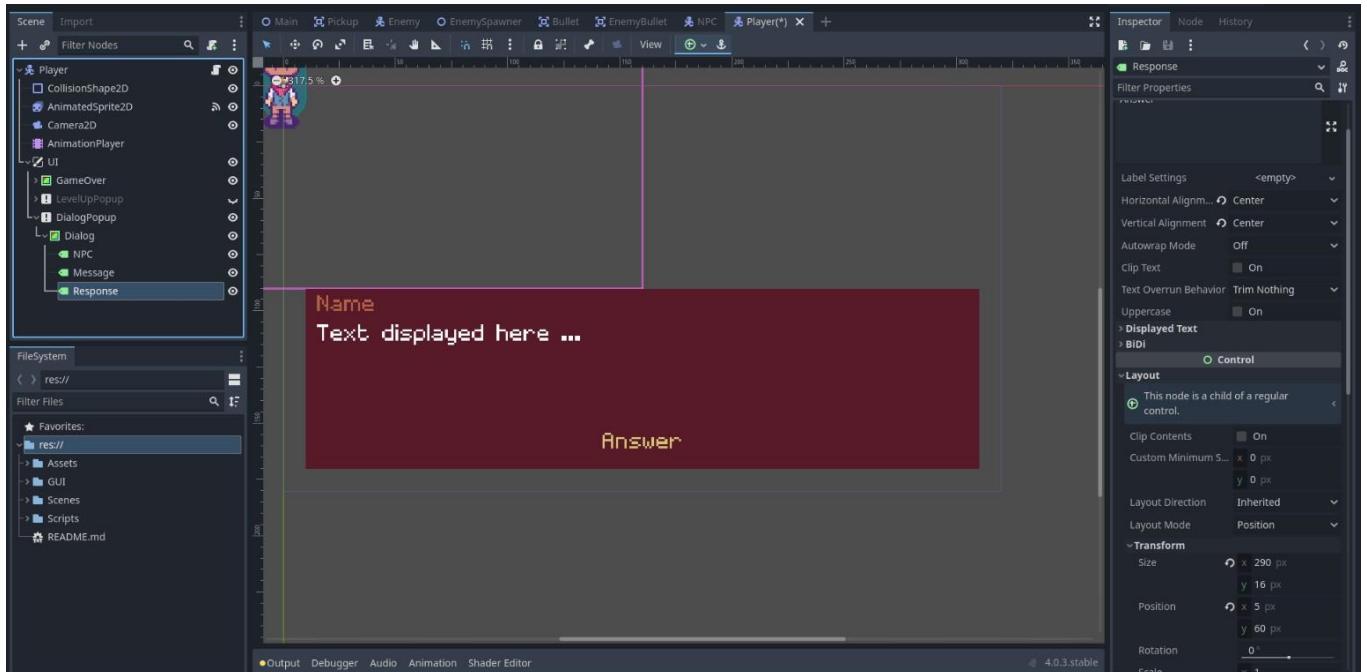
Now change its font to "Scroundinger", and its font size to 10. We also want to change its font color, so underneath Color > Font Color, change the color to #B26245.



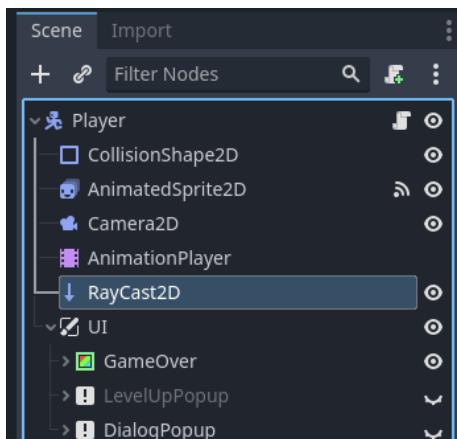
Select your Message Label node and change its text to "Text here...". Change its size (x: 290, y: 30); position (x: 5, y: 15). Also change its font to "Scrondinger", and its font size to 10. The AutoWrap property should also be set to "Word".



Select your Response Label node and change its text to "Answer". Change its size (x: 290, y: 16); position (x: 5, y: 60); horizontal and vertical alignment (center). Also, change its font to "Scrondinger", and its font size to 10. Underneath Color > Font Color, change the color to #D6c376.



Change the DialogPopup's visibility to be hidden and add a RayCast2D node before your UI layer.



In your Player script, we have to do the same as we did in our Enemy script when we set the direction of our raycast node to be the same as the direction of our character. In your `_physics_process()` function, let's turn the RayCast2D node to follow our player's movement direction.

```

### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount
@onready var xp_amount = $UI/XP
@onready var level_amount = $UI/Level
@onready var animation_player = $AnimationPlayer
@onready var level_popup = $UI/LevelPopup
@onready var ray_cast = $RayCast2D

# ----- Movement & Animations -----

func _physics_process(delta):
    # Get player input (left, right, up/down)
    var direction: Vector2
    direction.x = Input.get_action_strength("ui_right") -
        Input.get_action_strength("ui_left")
    direction.y = Input.get_action_strength("ui_down") -
        Input.get_action_strength("ui_up")
    # Normalize movement
    if abs(direction.x) == 1 and abs(direction.y) == 1:
        direction = direction.normalized()
    # Sprinting
    if Input.is_action_pressed("ui_sprint"):
        if stamina >= 25:
            speed = 100
            stamina = stamina - 5
            stamina_updated.emit(stamina, max_stamina)
    elif Input.is_action_just_released("ui_sprint"):
        speed = 50
    # Apply movement if the player is not attacking
    var movement = speed * direction * delta
    if is_attacking == false:
        move_and_collide(movement)
        player_animations(direction)
    # If no input is pressed, idle
    if !Input.is_anything_pressed():

```

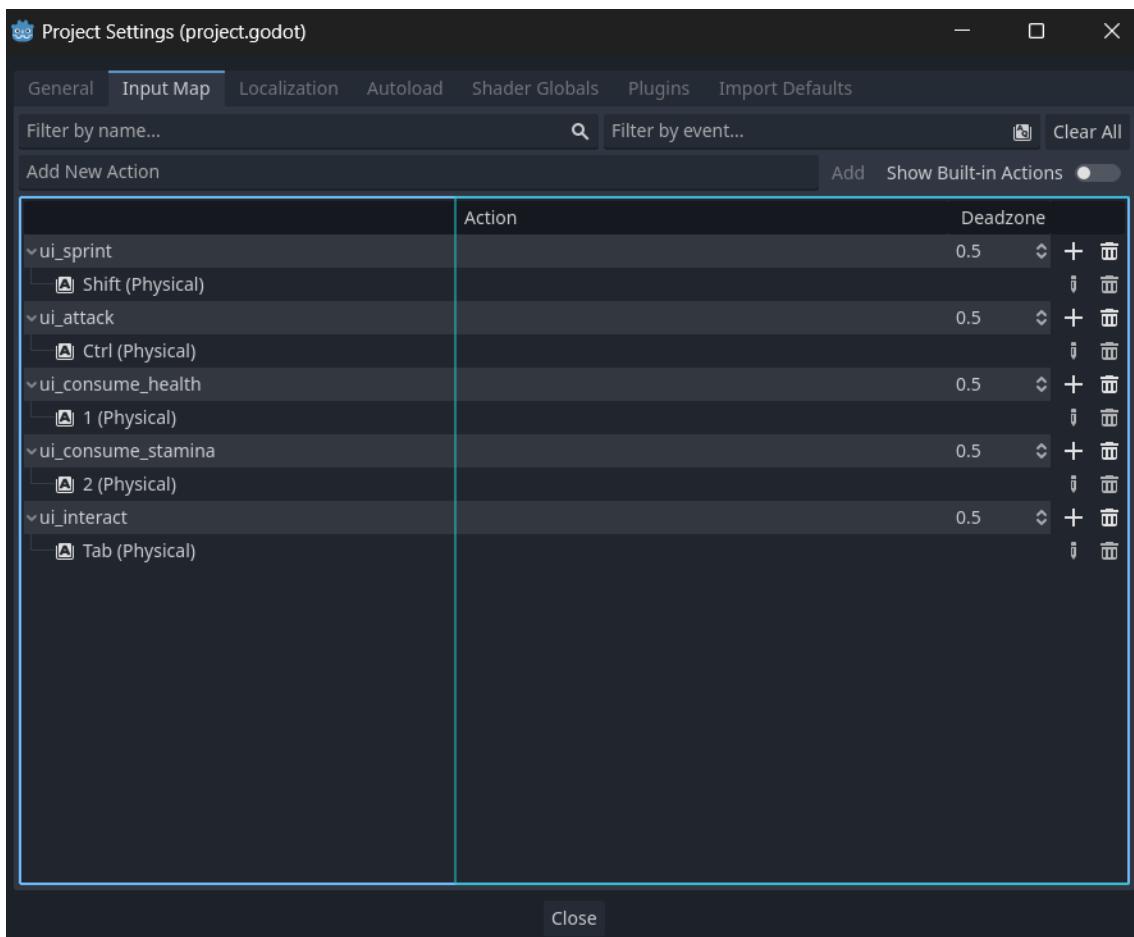
```

if is_attacking == false:
    animation = "idle_" + returned_direction(new_direction)
# Turn RayCast2D toward movement direction
if direction != Vector2.ZERO:
    ray_cast.target_position = direction.normalized() * 50

```

This raycast will hit any node that has a collision body assigned to it. We want it to hit our NPC, and if it does and we press our interaction button, it will launch the dialog popup and run the dialog tree. For this, we need to first add a new input action that we can press to interact with NPCs.

In your Input Actions menu in your Project Settings, add a new input called "ui_interact" and assign a key to it. I assigned the TAB key on my keyboard to this action.



Now, in our input() function, we will expand on it to add an input event for our ui_interact action. If our player presses this button, our raycast will capture the colliders it's hitting,

and if one of those colliders is part of the "NPC" group, it will launch the NPC dialog. We've done something similar to this in our Enemy script.

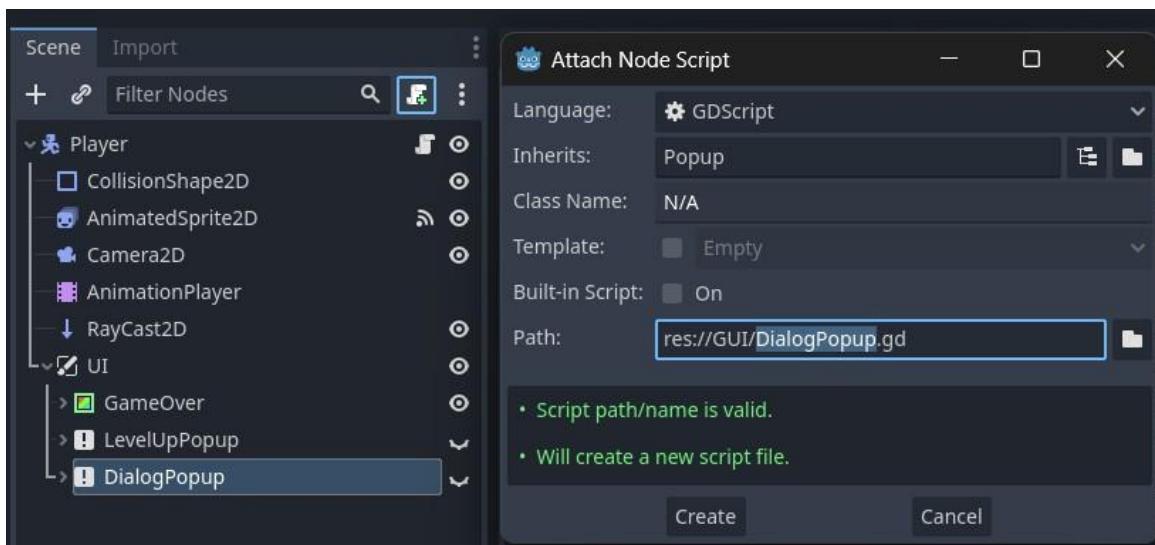
```
### Player.gd

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #checks the current time as the amount of time passed in milliseconds
        #since the engine started
        var now = Time.get_ticks_msec()
        #check if player can shoot if the reload time has passed and we have ammo
        if now >= bullet_fired_time and ammo_pickup > 0:
            #shooting anim
            is_attacking = true
            var animation  = "attack_" + returned_direction(new_direction)
            animation_sprite.play(animation)
            #bullet fired time to current time
            bullet_fired_time = now + bullet_reload_time
            #reduce and signal ammo change
            ammo_pickup = ammo_pickup - 1
            ammo_pickups_updated.emit(ammo_pickup)
    #using health consumables
    elif event.is_action_pressed("ui_consume_health"):
        if health > 0 && health_pickup > 0:
            health_pickup = health_pickup - 1
            health = min(health + 50, max_health)
            health_updated.emit(health, max_health)
            health_pickups_updated.emit(health_pickup)
    #using stamina consumables
    elif event.is_action_pressed("ui_consume_stamina"):
        if stamina > 0 && stamina_pickup > 0:
            stamina_pickup = stamina_pickup - 1
            stamina = min(stamina + 50, max_stamina)
            stamina_updated.emit(stamina, max_stamina)
            stamina_pickups_updated.emit(stamina_pickup)
    #interact with world
    elif event.is_action_pressed("ui_interact"):
        var target = ray_cast.get.collider()
        if target != null:
            if target.is_in_group("NPC"):
                # Talk to NPC
                #todo: add dialog function to npc
```

```
return
```

Next up we want our NPC script to update the DialogPopup node in our Player scene's labels (npc, message, and response). We also want it to handle the player's input for the dialog's response, so if the player says "A" or "B" for yes/no the dialog popup will update the message values according to the dialog tree. We'll get more into this when we set up our dialog tree later on.

Attach a new script to your DialogPopup node and save it underneath your GUI folder.



Sometimes, you want a class's member variable to do more than just hold data and perform some validation or computation whenever its value changes. It may also be desired to encapsulate its access in some way. For this, GDScript provides a special syntax to define properties using the [set and get](#) keywords after a variable declaration. Then you can define a code block that will be executed when the variable is accessed or assigned.

At the top of this script, we need to declare a few variables. We will define these variables using our set/get properties, which will allow us to **set** our Label values from the data that we **get** from our NPC script. We also need to define a variable for our NPC reference.

```

### DialogPopup.gd

extends CanvasLayer

#gets the values of our npc from our NPC scene and sets it in the label values
var npc_name : set = npc_name_set
var message: set = message_set
var response: set = response_set

#reference to NPC
var npc

```

Next, we need to create three functions to set the values of our set/get variables. This will capture the value passed from our NPC and assign the value to our Labels.

```

### DialogPopup.gd

extends CanvasLayer

#gets the values of our npc from our NPC scene and sets it in the label values
var npc_name : set = npc_name_set
var message: set = message_set
var response: set = response_set

#reference to NPC
var npc

#sets the npc name with the value received from NPC
func npc_name_set(new_value):
    npc_name = new_value
    $Dialog/NPC.text = new_value

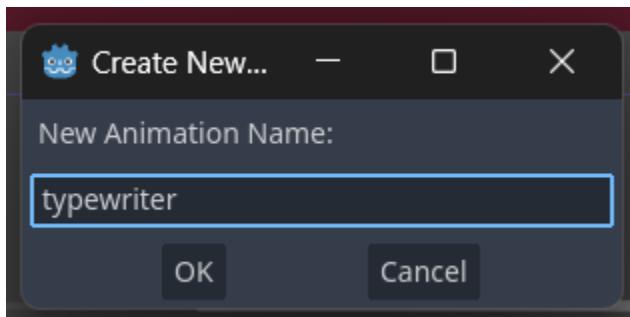
#sets the message with the value received from NPC
func message_set(new_value):
    message = new_value
    $Dialog/Message.text = new_value

#sets the response with the value received from NPC
func response_set(new_value):
    response = new_value
    $Dialog/Response.text = new_value

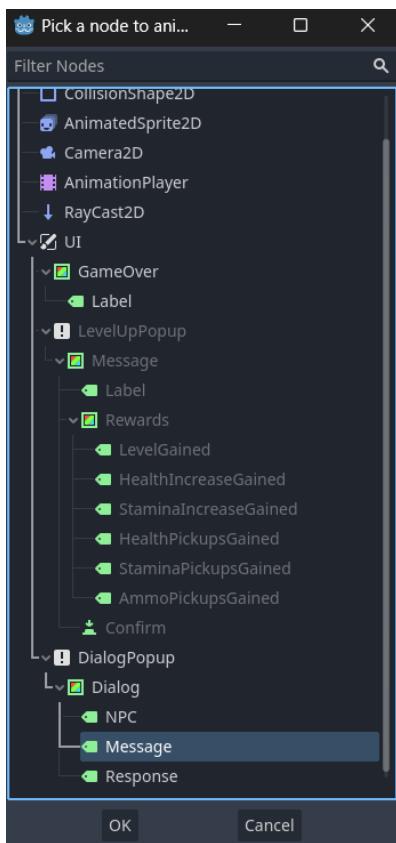
```

Before we continue with this script, let's add an animation to our Message node so that it has a typewriter effect. This is a classic feature in RPG-type games, so we will also implement this. In Godot 4, we can do this by changing our text's visible ratio in the AnimationPlayer. This will transition our text visibility slowly from invisible to visible.

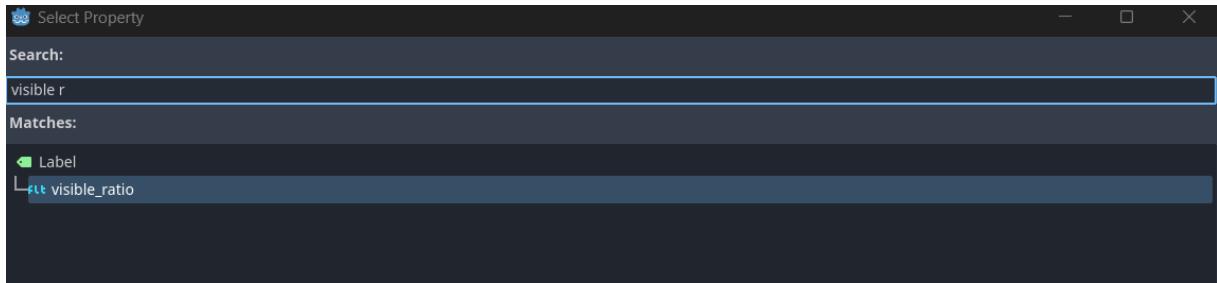
In your AnimationPlayer node in your Player scene, add a new animation called "typewriter".



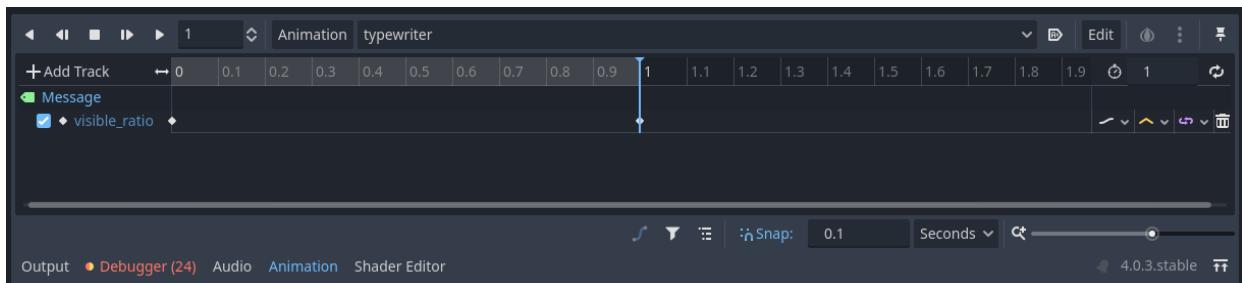
Add a new Property Track to your animation and assign it to your Message node.



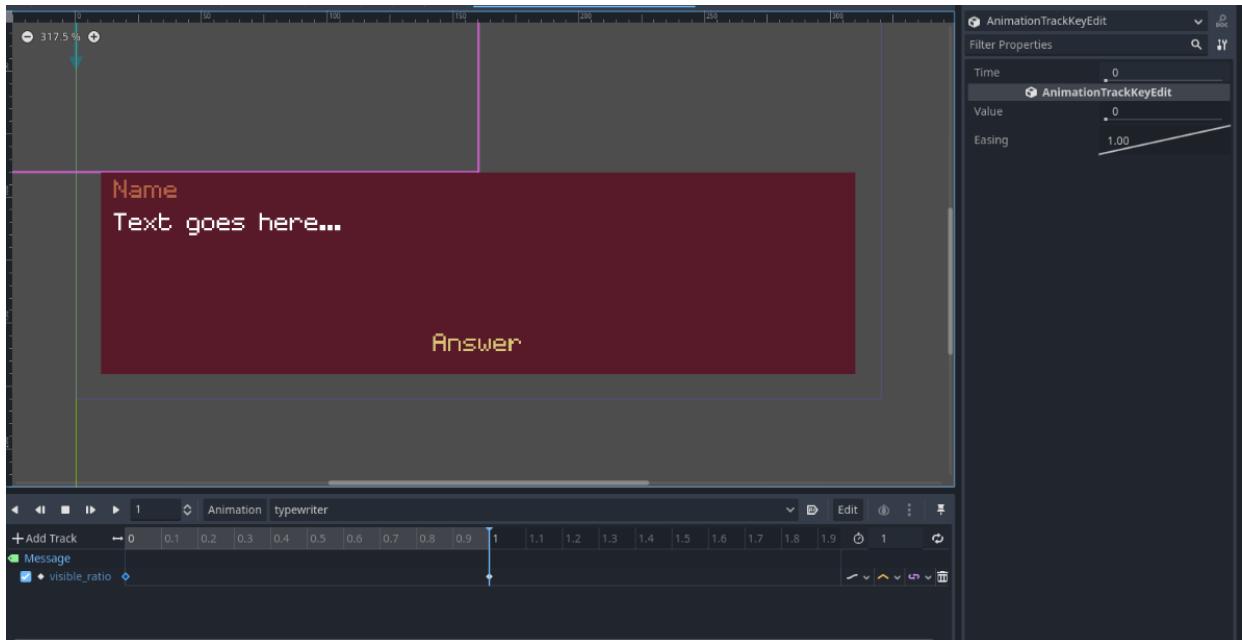
The property we want to change is the [visible ratio](#) of our dialog text.

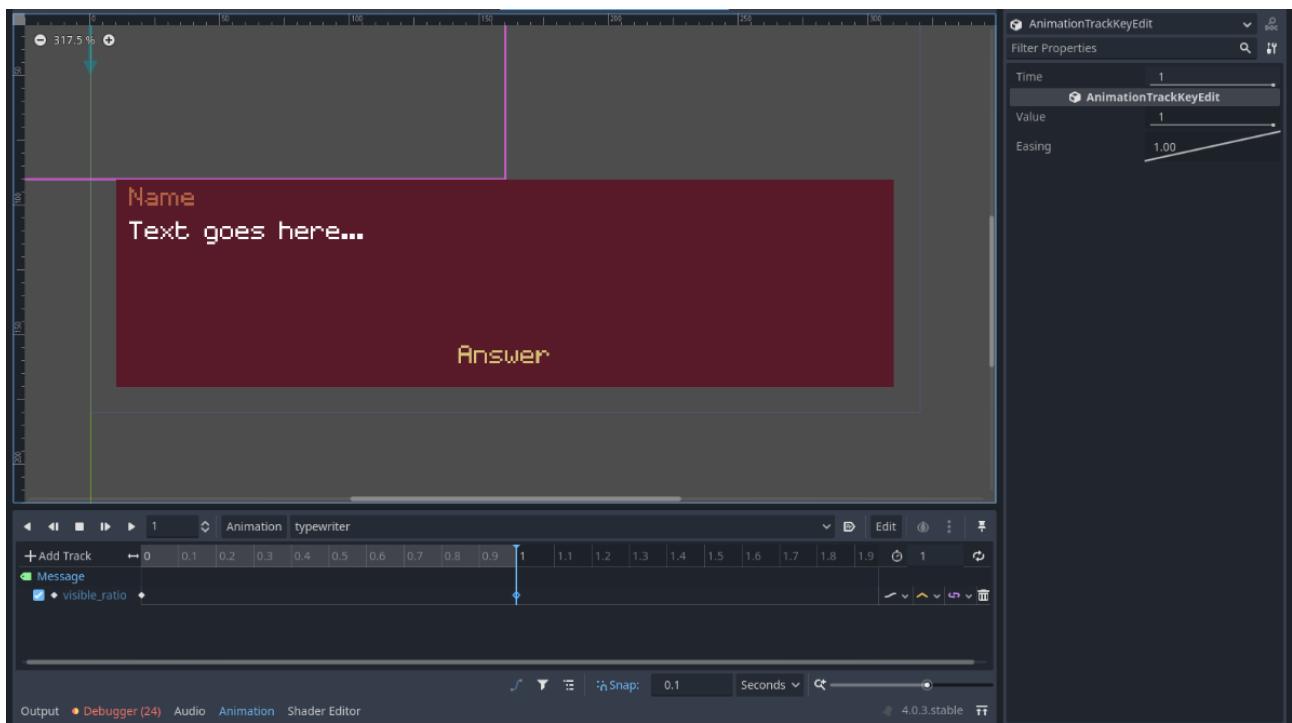


This visible ratio will make the dialog ratio visible from 0 to 1. In your `visible_ratio` track, add two new keys, one at keyframe 0 and the other one at keyframe 1.

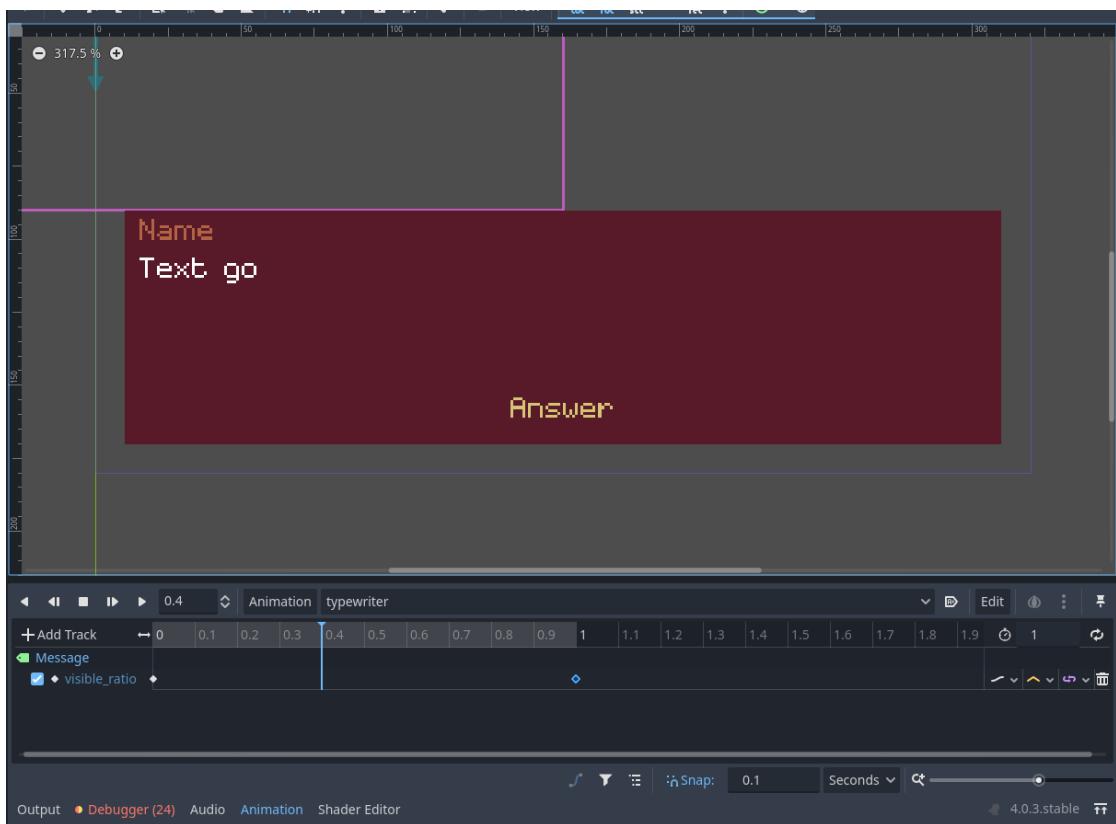


Change the Value of your keyframe 0 to 0, and the Value of your keyframe 1 to 1.





Now if you play your animation, your typewriter effect should work!



Back in your DialogPopup code, let's create two functions that will be called by our NPC script. The first function should pause the game and show the dialog popup, and play the typewriter animation. The other function should hide the dialog and unpause the game.

```
### DialogPopup.gd

extends CanvasLayer

# Node refs
@onready var animation_player = $"../../AnimationPlayer"

#gets the values of our npc from our NPC scene and sets it in the label values
var npc_name : set = npc_name_set
var message: set = message_set
var response: set = response_set

#reference to NPC
var npc

#sets the npc name with the value received from NPC
func npc_name_set(new_value):
    npc_name = new_value
    $Dialog/NPC.text = new_value

#sets the message with the value received from NPC
func message_set(new_value):
    message = new_value
    $Dialog/Message.text = new_value

#sets the response with the value received from NPC
func response_set(new_value):
    response = new_value
    $Dialog/Response.text = new_value

#opens the dialog
func open():
    get_tree().paused = true
    self.visible = true
    animation_player.play("typewriter")

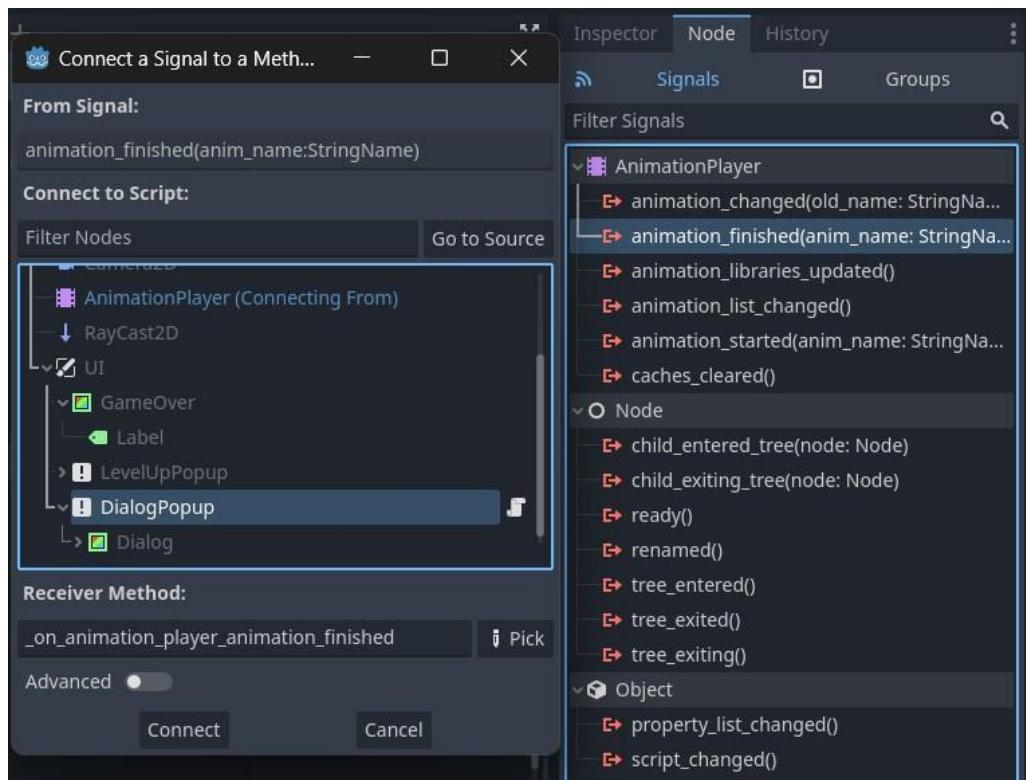
#closes the dialog
func close():
```

```
get_tree().paused = false  
self.visible = false
```

If this node is hidden and if the message text has not been completed yet, this node should not receive input. So, in the `_ready()` function, we must call the `set_process_input()` function to disable input handling. This will disable the input function and Input singleton in our Player script from processing any input.

```
### DialogPopup.gd  
  
# older code  
  
# ----- Processing -----  
#no input on hidden  
func _ready():  
    set_process_input(false)  
  
#opens the dialog  
func open():  
    get_tree().paused = true  
    self.visible = true  
    animation_player.play("typewriter")  
  
#closes the dialog  
func close():  
    get_tree().paused = false  
    self.visible = false
```

We only want the player to be able to insert inputs if our "typewriter" animation has finished animating our message text. Therefore, we can connect the AnimationPlayer node's `animation_finished()` signal to our DialogPopup script.



```
### DialogPopup.gd

# older code

# ----- Processing -----
#no input on hidden
func _ready():
    set_process_input(false)

#opens the dialog
func open():
    get_tree().paused = true
    self.visible = true
    animation_player.play("typewriter")

#closes the dialog
func close():
    get_tree().paused = false
    self.visible = false

#input after animation plays
func _on_animation_player_animation_finished(anim_name):
    set_process_input(true)
```

Finally, we can write our code that will accept the player's input in response to our dialog options. We don't have to create unique input actions for this, as we can just use the [InputEventKey](#) object. This object stores key presses on the keyboard. So if we press "A" or "B", the object will capture those keys as input and trigger the dialog tree to respond to these inputs.

Here's a visual representation to help you understand what we want to achieve:



```
### DialogPopup.gd

#older code

# ----- Dialog -----
func _input(event):
    if event is InputEventKey:
        if event.is_pressed():
            if event.keycode == KEY_A:
                #todo: add dialog function to npc
                return
            elif event.keycode == KEY_B:
                #todo: add dialog function to npc
                return
```

NPC DIALOG TREE

What is a Dialog Tree?

A dialog tree, often referred to as a conversation tree or branching dialog, is a form of interactive narrative. It represents a series of branching choices in character dialogues or interactions, allowing players or users to navigate through various conversation paths based on their choices.

We'll come back to this `_input` function once we have the `dialog` function in our NPC script. We're going to start working on that now, so let's open up our NPC script and define some variables that will store our quest and dialog states. We'll create an enum that will hold the states of our Quest - which is when it's not started, started, and completed. Then we will instance that enum to set its initial state to be `NOT_STARTED`. We then need to store the state of our dialog as an integer.

We'll define our dialog state as an integer so that we can increment it throughout our dialog tree, and then we can select our dialog based on our number value in a `match` statement. A `match` statement is used to branch the execution of a program. It's the equivalent of the `switch` statement found in many other languages, and it works in the way that an expression is compared to a pattern, and if that pattern matches, the corresponding block will be executed. After that, the execution continues below the `match` statement.

Our dialog and quest states will be executed in this match-case pattern:

```
match (quest_status):
    NOT_STARTED:
        match (dialog_state):
            0:
                //dialog message
```

```
//increase dialog state

match (answer):
    //dialog message
    //increase dialog state

1:
    //dialog message
    //increase dialog state

    match (answer):
        //dialog message
        //increase dialog state

STARTED:

match (dialog_state):
    0:
        //dialog message
        //increase dialog state

        match (answer):
            //dialog message
            //increase dialog state

    1:
        //dialog message
        //increase dialog state

        match (answer):
            //dialog message
            //increase dialog state
```

COMPLETED:

```
match (dialog_state):  
    0:  
        //dialog message  
        //increase dialog state  
    match (answer):  
        //dialog message
```

Let's define our variables.

```
### NPC  
  
extends CharacterBody2D  
  
#quest and dialog states  
enum QuestStatus { NOT_STARTED, STARTED, COMPLETED }  
var quest_status = QuestStatus.NOT_STARTED  
var dialog_state = 0  
var quest_complete = false
```

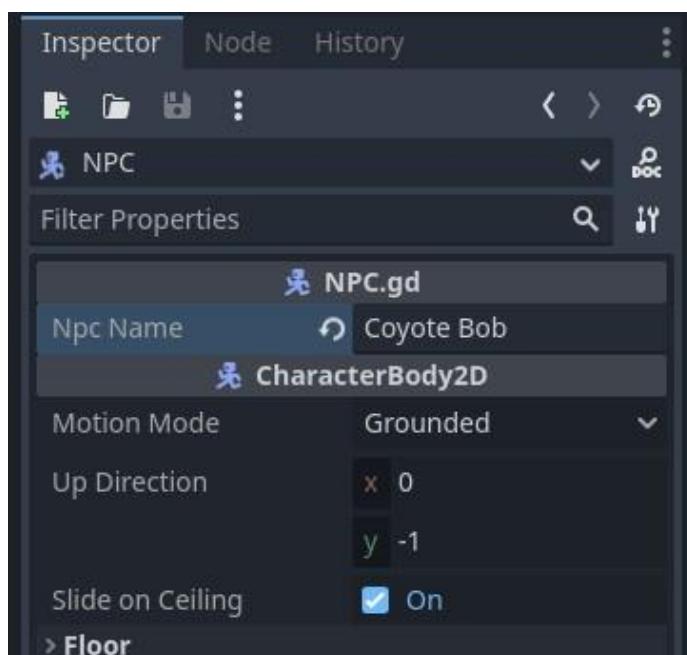
We also need to define some variables that will reference our DialogPopup node in our Player scene, as well as our Player itself - because our player is the one that will initiate the interaction with our NPC.

```
### NPC  
  
extends CharacterBody2D  
  
# Node refs  
@onready var dialog_popup =  
get_tree().root.get_node("Main/Player/UI/DialogPopup")  
@onready var player = get_tree().root.get_node("Main/Player")  
@onready var animation_sprite = $AnimatedSprite2D  
  
#quest and dialog states  
enum QuestStatus { NOT_STARTED, STARTED, COMPLETED }  
var quest_status = QuestStatus.NOT_STARTED
```

```
var dialog_state = 0  
var quest_complete = false
```

I also want us to type in the NPC's name in the Inspector panel, instead of giving them a constant value like "Joe". To do this, we can export our variable.

```
### NPC  
  
extends CharacterBody2D  
  
# Node refs  
@onready var dialog_popup =  
get_tree().root.get_node("Main/Player/UI/DialogPopup")  
@onready var player = get_tree().root.get_node("Main/Player")  
@onready var animation_sprite = $AnimatedSprite2D  
  
#quest and dialog states  
enum QuestStatus { NOT_STARTED, STARTED, COMPLETED }  
var quest_status = QuestStatus.NOT_STARTED  
var dialog_state = 0  
var quest_complete = false  
  
#npc name  
@export var npc_name = ""
```



In our `ready()` function we need to set the default animation of our NPC to be "idle_down".

```
### NPC
extends CharacterBody2D

# Node refs
@onready var dialog_popup =
get_tree().root.get_node("Main/Player/UI/DialogPopup")
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D

#quest and dialog states
enum QuestStatus { NOT_STARTED, STARTED, COMPLETED }
var quest_status = QuestStatus.NOT_STARTED
var dialog_state = 0
var quest_complete = false

#npc name
@export var npc_name = ""

#initialize variables
func _ready():
    animation_sprite.play("idle_down")
```

Now we can create our `dialog()` function. This dialog tree is quite long, so I recommend you go ahead and just copy and paste it from below. A dialog tree, or conversation tree, is a gameplay mechanic that runs when a player character interacts with an NPC. In this tree, the player is given a choice of what to say and makes subsequent choices until the conversation ends.

In our dialog tree, our NPC runs our Player through a quest to go and find a recipe book. We have not yet created this recipe book, or quest item, which will call our NPC to notify them that the quest has been completed. If we haven't gotten this quest item yet, the NPC will remind us to get it. If we've gotten this quest item, the NPC will thank us and reward us, and complete the quest.

```
### NPC
extends CharacterBody2D
```

```

# Node refs
@onready var dialog_popup =
get_tree().root.get_node("Main/Player/UI/DialogPopup")
@onready var player = get_tree().root.get_node("Main/Player")
@onready var animation_sprite = $AnimatedSprite2D

#quest and dialog states
enum QuestStatus { NOT_STARTED, STARTED, COMPLETED }
var quest_status = QuestStatus.NOT_STARTED
var dialog_state = 0
var quest_complete = false

#npc name
@export var npc_name = ""

#initialize variables
func _ready():
    animation_sprite.play("idle_down")

#dialog tree
func dialog(response = ""):
    # Set our NPC's animation to "talk"
    animation_sprite.play("talk_down")
    # Set dialog_popup npc to be referencing this npc
    dialog_popup.npc = self
    dialog_popup.npc_name = str(npc_name)
    # dialog tree
    match quest_status:
        QuestStatus.NOT_STARTED:
            match dialog_state:
                0:
                    # Update dialog tree state
                    dialog_state = 1
                    # Show dialog popup
                    dialog_popup.message = "Howdy Partner. I haven't seen anybody round these parts in quite a while. That reminds me, I recently lost my momma's secret recipe book, can you help me find it?"
                    dialog_popup.response = "[A] Yes [B] No"
                    dialog_popup.open() #re-open to show next dialog
                1:
                    match response:
                        "A":
                            # Update dialog tree state
                            dialog_state = 2

```

```

# Show dialog popup
dialog_popup.message = "That's mighty kind of you,
thanks."
dialog_popup.response = "[A] Bye"
dialog_popup.open() #re-open to show next dialog

"B":
# Update dialog tree state
dialog_state = 3
# Show dialog popup
dialog_popup.message = "Well, I'll be waiting like a
tumbleweed 'till you come back."
dialog_popup.response = "[A] Bye"
dialog_popup.open() #re-open to show next dialog

2:
# Update dialog tree state
dialog_state = 0
quest_status = QuestStatus.STARTED
# Close dialog popup
dialog_popup.close()
# Set NPC's animation back to "idle"
animation_sprite.play("idle_down")

3:
# Update dialog tree state
dialog_state = 0
# Close dialog popup
dialog_popup.close()
# Set NPC's animation back to "idle"
animation_sprite.play("idle_down")

QuestStatus.STARTED:
match dialog_state:
    0:
        # Update dialog tree state
        dialog_state = 1
        # Show dialog popup
        dialog_popup.message = "Found that book yet?"
        if quest_complete:
            dialog_popup.response = "[A] Yes [B] No"
        else:
            dialog_popup.response = "[A] No"
        dialog_popup.open()

    1:
        if quest_complete and response == "A":
            # Update dialog tree state
            dialog_state = 2
            # Show dialog popup

```

```

        dialog_popup.message = "Yeehaw! Now I can make cat-eye
soup. Here, take this."
        dialog_popup.response = "[A] Bye"
        dialog_popup.open()
    else:
        # Update dialog tree state
        dialog_state = 3
        # Show dialog popup
        dialog_popup.message = "I'm so hungry, please hurry..."
        dialog_popup.response = "[A] Bye"
        dialog_popup.open()
2:
    # Update dialog tree state
    dialog_state = 0
    quest_status = QuestStatus.COMPLETED
    # Close dialog popup
    dialog_popup.close()
    # Set NPC's animation back to "idle"
    animation_sprite.play("idle_down")
    # Add pickups and XP to the player.
    player.add_pickup(Global.Pickups.AMMO)
    player.update_xp(50)
3:
    # Update dialog tree state
    dialog_state = 0
    # Close dialog popup
    dialog_popup.close()
    # Set NPC's animation back to "idle"
    animation_sprite.play("idle_down")
QuestStatus.COMPLETED:
    match dialog_state:
        0:
            # Update dialog tree state
            dialog_state = 1
            # Show dialog popup
            dialog_popup.message = "Nice seeing you again partner!"
            dialog_popup.response = "[A] Bye"
            dialog_popup.open()
        1:
            # Update dialog tree state
            dialog_state = 0
            # Close dialog popup
            dialog_popup.close()
            # Set NPC's animation back to "idle"
            animation_sprite.play("idle_down")

```

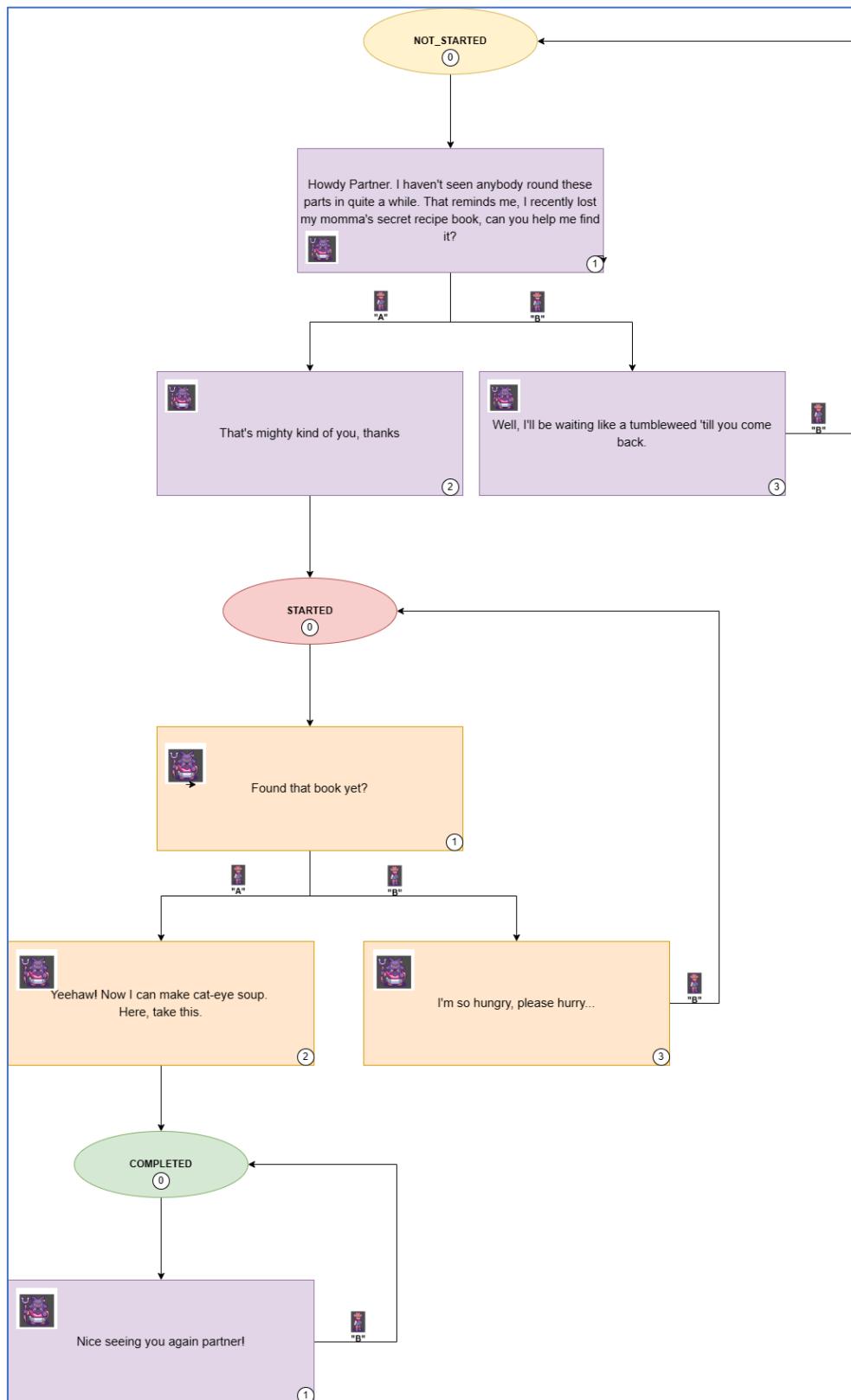
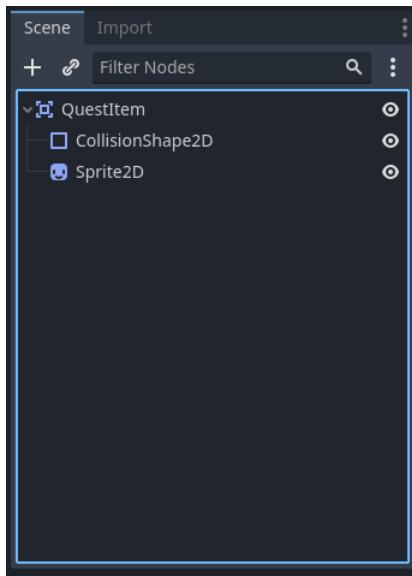


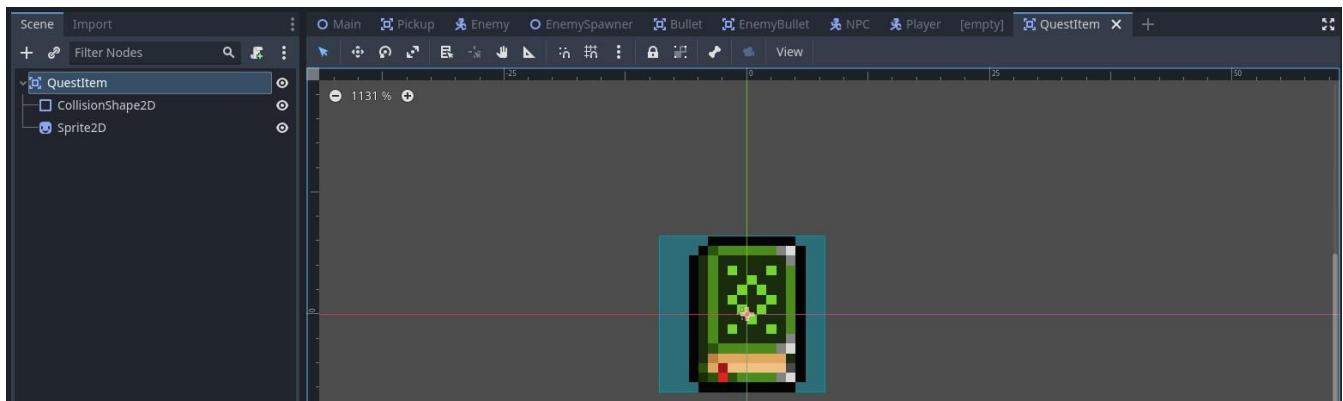
Figure 15: Visual Representation of our Dialog Tree.

QUEST ITEM SETUP

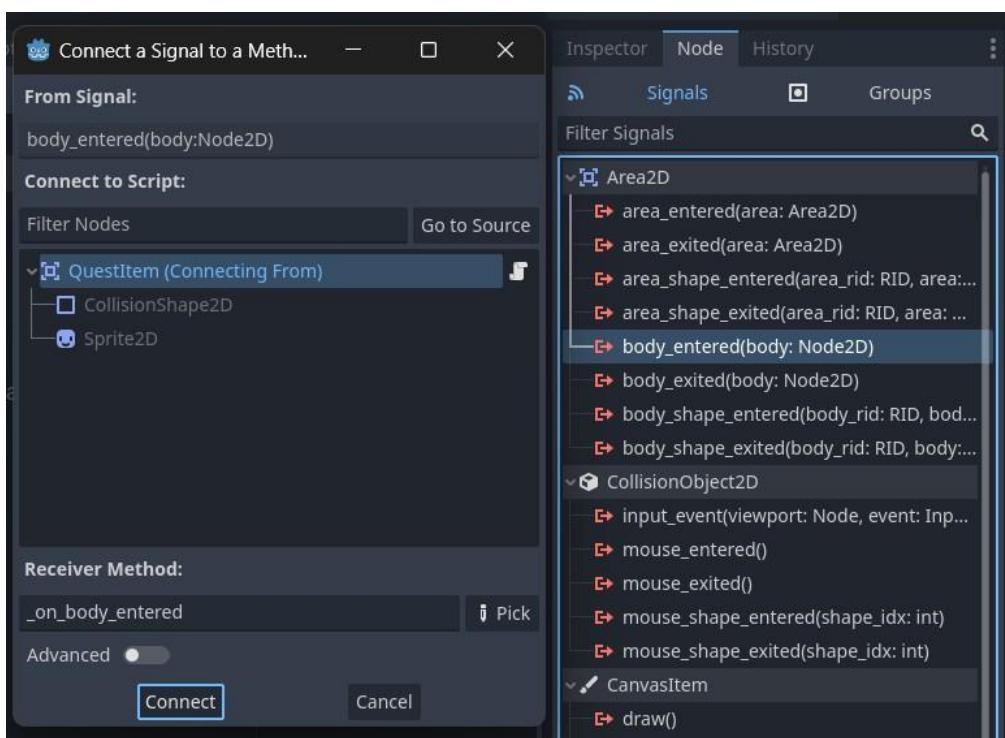
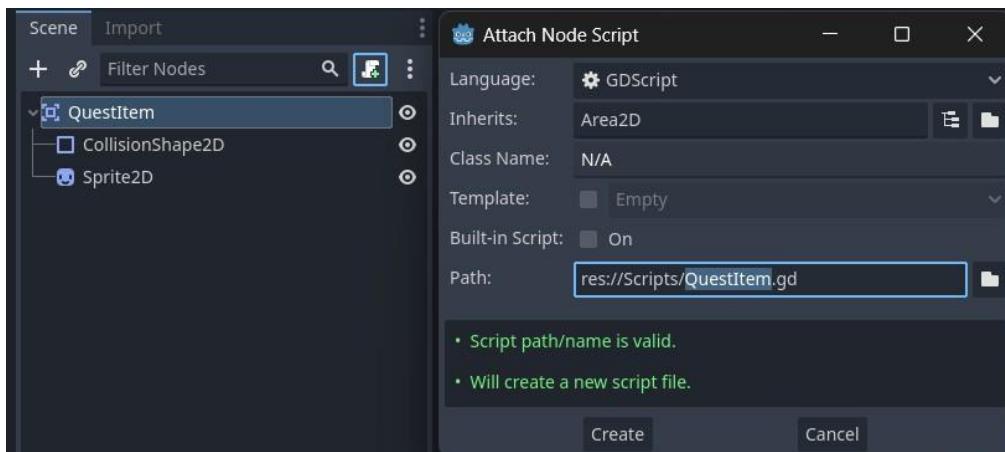
For this quest item, we can duplicate our Pickups scene and rename it to "QuestItem". Detach the Pickups script and signal from the newly created scene and rename its root to "QuestItem". Delete the script and signal from the root node.



Change its sprite to be anything you want. Since our NPC is looking for a recipe book, I'm going to change my sprite to "book_02d.png", which can be found under the Assets > Icons directory.



Attach a new script to the root of this scene and save it underneath your Scripts folder. Also, connect its `body_entered()` signal to this new script.



In this script, we need to get a reference to our NPC scene which we will instance in our Main scene. From this, we'll need to see if the Player has entered the body of this scene, and if true, we can call our to our NPC to change its quest status to complete - since we've found the requirements to complete the quest.

```
### QuestItem.gd

extends Area2D

#npc node reference
```

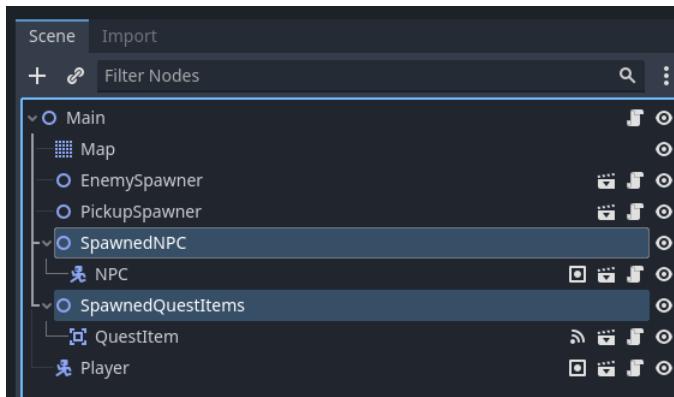
```

@onready var npc = get_tree().root.get_node("Main/SpawnedNPC/NPC")

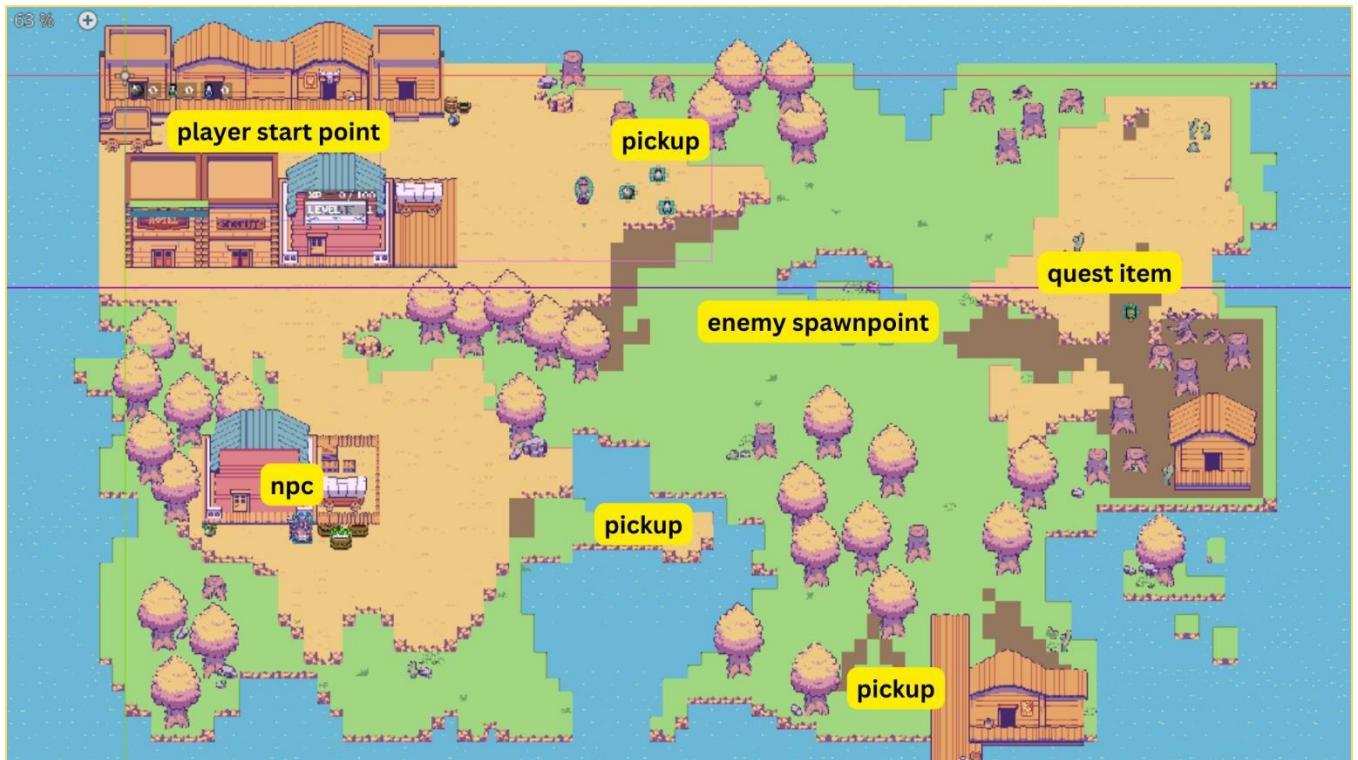
#if the player enters the collision body, destroy item and update quest
func _on_body_entered(body):
    if body.name == "Player":
        print("Quest item obtained!")
        get_tree().queue_delete(self)
        npc.quest_complete = true

```

Let's instance our NPC and our Quest Item in our Main scene underneath two new nodes called SpawneQuestItems and SpawneNPC (both should be Node2D nodes).



Here is an example of my map's layout for my quest:



Now we need to go back to our Player scene so that we can update our interact input to call the dialog function in our NPC script.

```
### Player.gd

# older code

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #checks the current time as the amount of time passed
        var now = Time.get_ticks_msec()
        #check if player can shoot if the reload time has passed and we have ammo
        if now >= bullet_fired_time and ammo_pickup > 0:
            #shooting anim
            is_attacking = true
            var animation  = "attack_" + returned_direction(new_direction)
            animation_sprite.play(animation)
            #bullet fired time to current time
            bullet_fired_time = now + bullet_reload_time
```

```

#reduce and signal ammo change
ammo_pickup = ammo_pickup - 1
ammo_pickups_updated.emit(ammo_pickup)

#using health consumables
elif event.is_action_pressed("ui_consume_health"):
    if health > 0 && health_pickup > 0:
        health_pickup = health_pickup - 1
        health = min(health + 50, max_health)
        health_updated.emit(health, max_health)
        health_pickups_updated.emit(health_pickup)

#using stamina consumables
elif event.is_action_pressed("ui_consume_stamina"):
    if stamina > 0 && stamina_pickup > 0:
        stamina_pickup = stamina_pickup - 1
        stamina = min(stamina + 50, max_stamina)
        stamina_updated.emit(stamina, max_stamina)
        stamina_pickups_updated.emit(stamina_pickup)

#interact with world
#interact with world
elif event.is_action_pressed("ui_interact"):
    var target = ray_cast.get.collider()
    if target != null:
        if target.is_in_group("NPC"):
            # Talk to NPC
            target.dialog()
            return

```

We also need to add our dialog functions to our DialogPopup's input code to capture our A and B responses.

```

### DialogPopup.gd

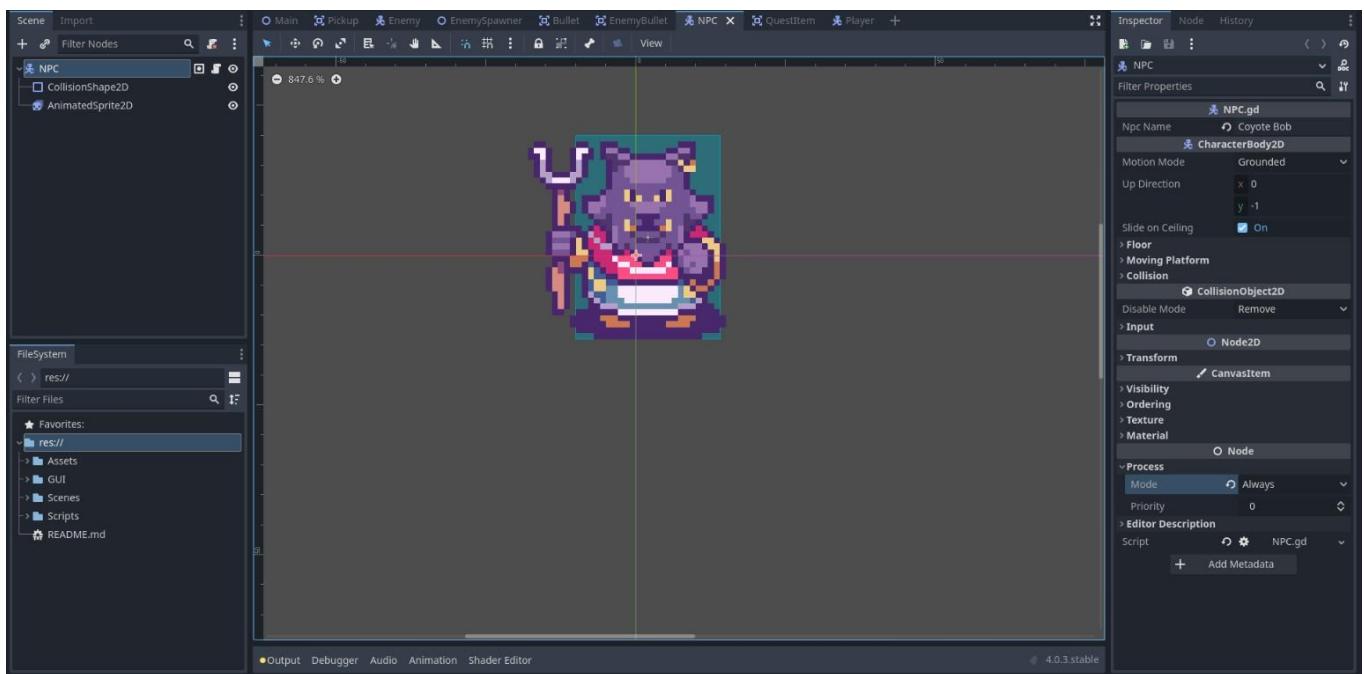
#older code

# ----- Dialog -----
func _input(event):
    if event is InputEventKey:
        if event.is_pressed():
            if event.keycode == KEY_A:
                npc.dialog("A")
            elif event.keycode == KEY_B:
                npc.dialog("B")

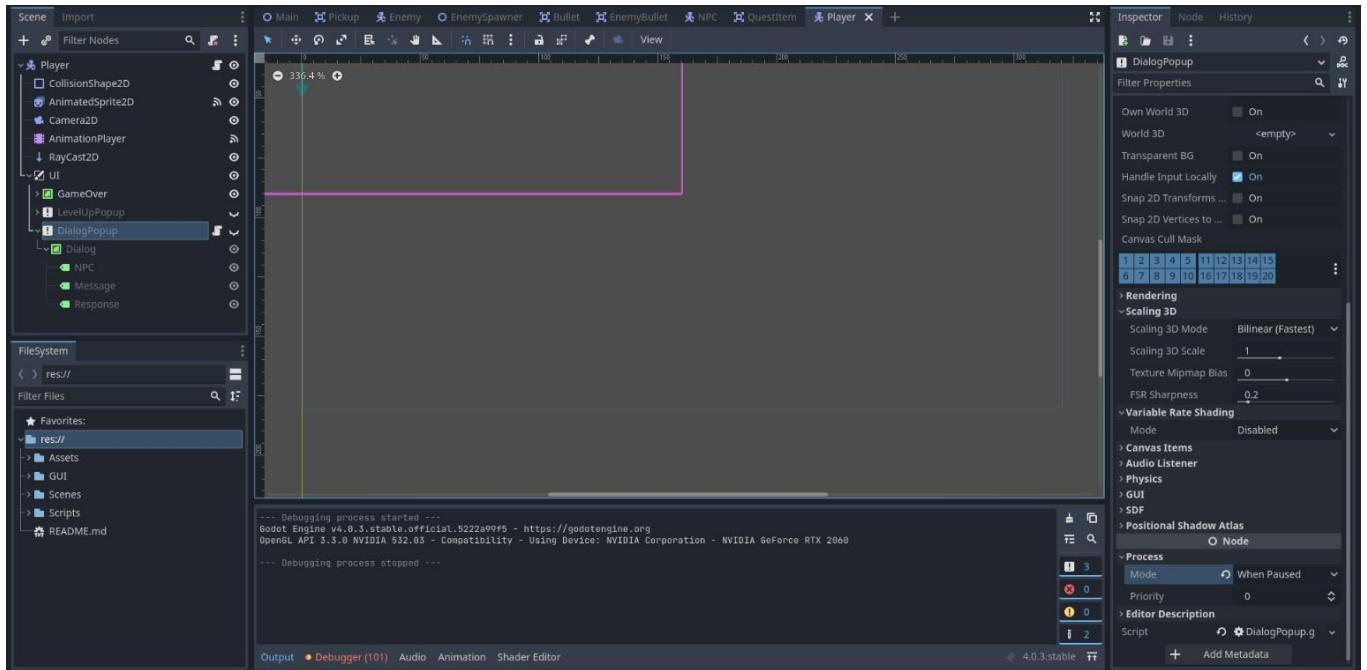
```

So now if our player were to run into our NPC and press TAB, the dialog popup should be made visible. We can then navigate through the conversation with our NPC by responding with our "A" for yes and "B" for no keys. If we get the quest item and we go back to the NPC, the dialog options should update, and we should receive our Pickups as a reward. We've then reached the end of our dialog tree, so if we return to our NPC, the last line will play.

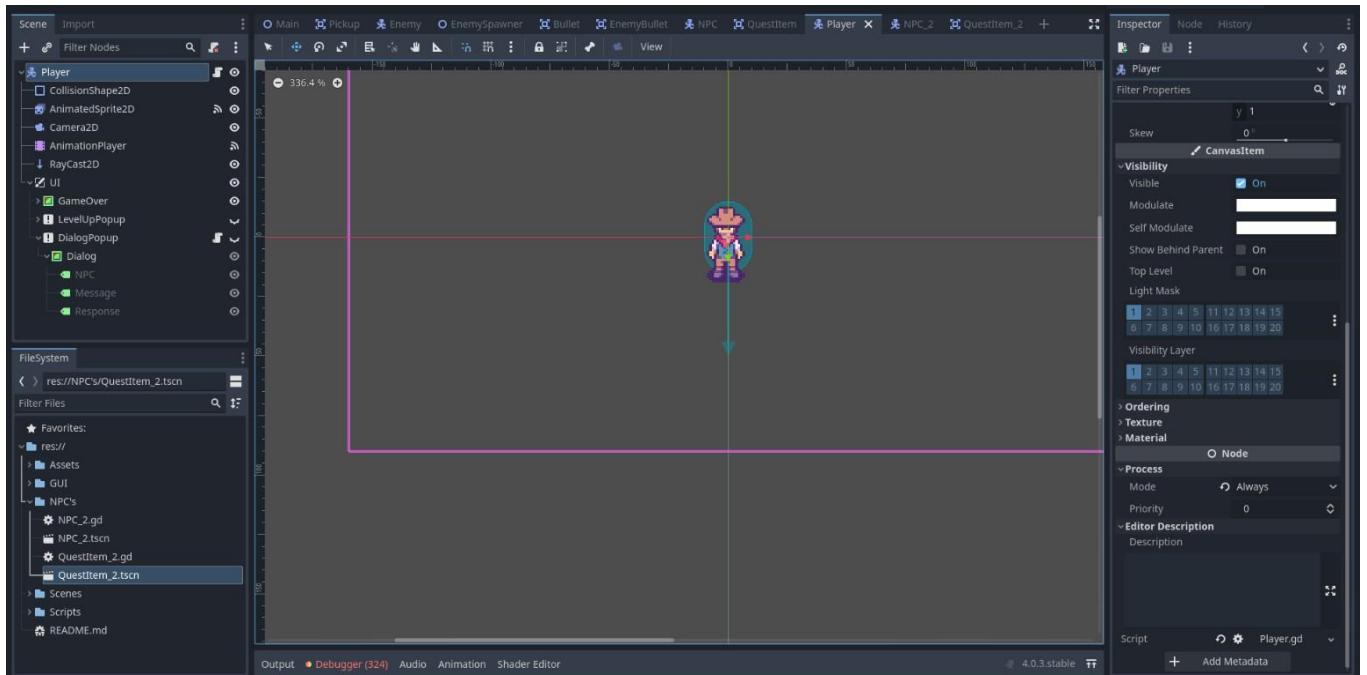
Before we can test this out, we need to change our node's process mode, because the DialogPopup will pause the game. Change the NPC's process mode to "Always" because we need their animations to play even if the game is paused - which is when the dialog is playing.



Change the DialogPopup's process mode to "When Paused", because we need to be able to run our input when the game is paused.



Finally, we'll need to change our Player's process mode to "Always". This means our player will be able to add input to the game even when the game is paused.



A problem will arise if we now play our game because our player will be able to walk away from the NPC when the dialog runs, so to fix this, we need to disable our player's movement that is executed in their `physics_process()` function. To do this, we can

simply call it at the end of our open() function and set its processing to false! Then in our close function, we just need to set it back to true because if the dialog popup is hidden, we want our player to move again. We'll also show/hide our cursor.

```
### DialogPopup.gd

extends CanvasLayer

# Node refs
@onready var animation_player = $"../../AnimationPlayer"
@onready var player = $"../../"

#gets the values of our npc from our NPC scene and sets it in the label values
var npc_name : set = npc_name_set
var message: set = message_set
var response: set = response_set

#reference to NPC
var npc

# ----- Text values -----
#sets the npc name with the value received from NPC
func npc_name_set(new_value):
    npc_name = new_value
    $Dialog/NPC.text = new_value

#sets the message with the value received from NPC
func message_set(new_value):
    message = new_value
    $Dialog/Message.text = new_value

#sets the response with the value received from NPC
func response_set(new_value):
    response = new_value
    $Dialog/Response.text = new_value

# ----- Processing -----
#no input on hidden
func _ready():
    set_process_input(false)

#opens the dialog
func open():
    get_tree().paused = true
```

```

        self.visible = true
        animation_player.play("typewriter")
        player.set_physics_process(false)
        Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)

#closes the dialog
func close():
    get_tree().paused = false
    self.visible = false
    player.set_physics_process(true)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

#input after animation plays
func _on_animation_player_animation_finished(anim_name):
    set_process_input(true)

# ----- Dialog -----
func _input(event):
    if event is InputEventKey:
        if event.is_pressed():
            if event.keycode == KEY_A:
                npc.dialog("A")
            elif event.keycode == KEY_B:
                npc.dialog("B")

```

If you run your game, and you run over to your NPC and press "TAB", your NPC dialog tree should run, and you should be able to accept the quest. If you then run over the quest item and return to your NPC, your NPC will notice that you've gotten your quest item and completed the quest. Congratulations, you now have an NPC with a simple quest!



```
--- Debugging process started ---
Godot Engine v4.0.3.stable.official.5222a99f5 - https://godotengine.org
OpenGL API 3.3.0 NVIDIA 532.03 - Compatibility - Using Device: NVIDIA Corporation - NVIDIA GeForce RTX 2060

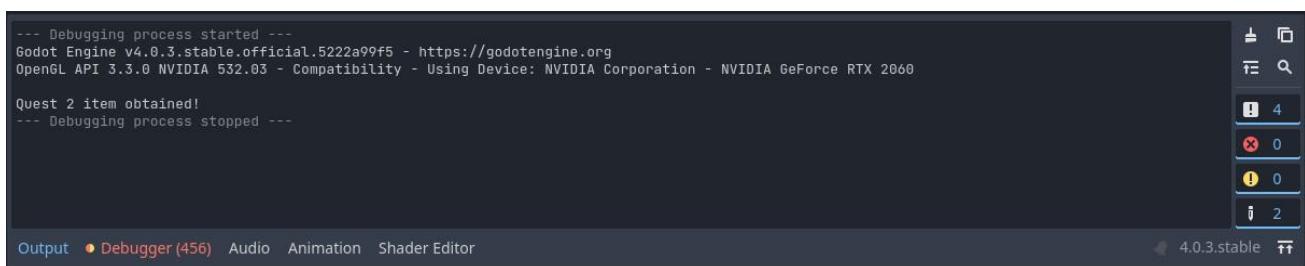
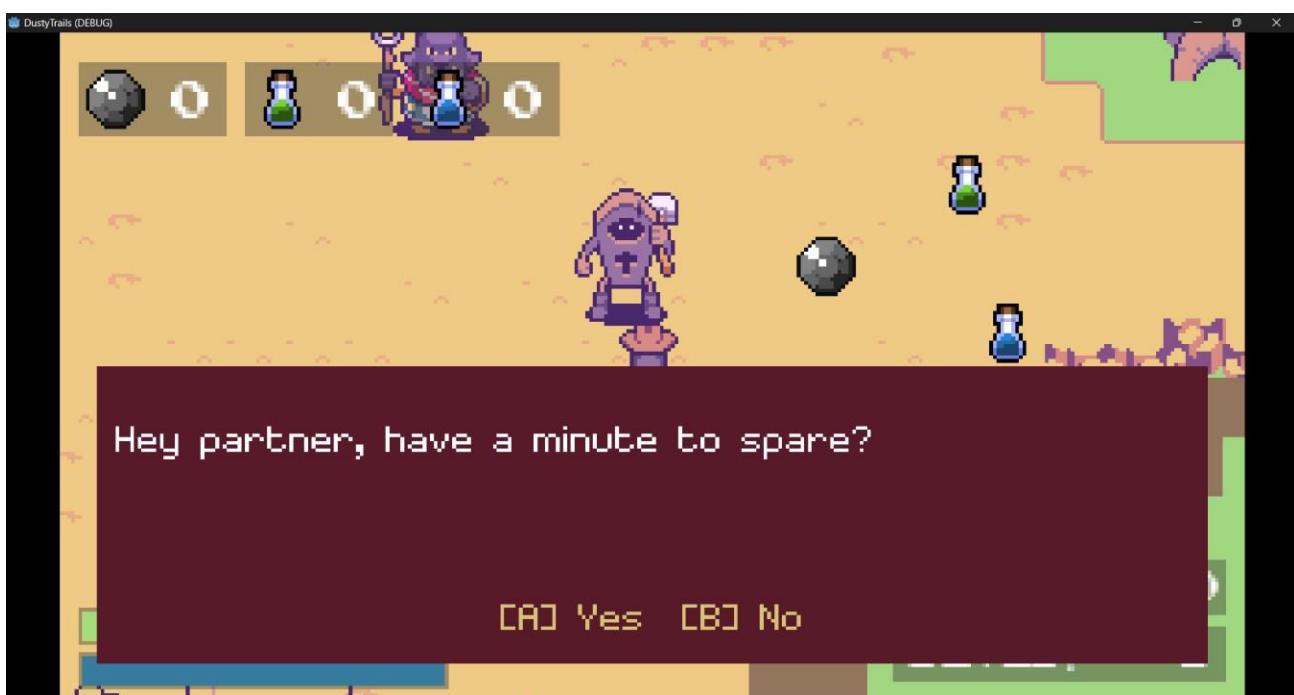
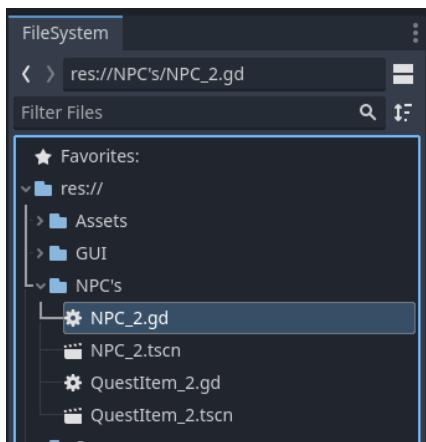
Quest item obtained!
--- Debugging process stopped ---

Output • Debugger (452) Audio Animation Shader Editor 4.0.3.stable
```



Unfortunately, our quest and dialog are directly tied to our NPC - and in a real game, you would have the NPC, Quest, and Dialog system in their scripts. I'm going to make a separate tutorial series on this, but for now, I just wanted to show you the basics of dialog trees and states.

If you want multiple NPCs, you'll have to duplicate the NPC scene and script, as well as the Quest Item scene and script - and then just update the values to have unique dialogs and references to your second NPC. I'll include an example of another NPC in the code reference below.



And there you have it! Now your game has an NPC with a Quest. The rest of the features that we'll add from here on will be quick to implement. Most of the hard work is done, so now we just need to add a scene-transition ability to transport our player

between worlds, and we'll also give our player the ability to sleep to restore their health and stamina values in the next part! Remember to save your project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 18: SCENE TRANSITIONS & DAY- NIGHT CYCLE

After a long day of shooting bad guys and completing quests, our player deserves to go home and take a nice, long nap. Before they can do this, however, we need to give them a house with a bed. In this part we're going to be doing that, as well as adding the ability for our Player to transition between scenes. We're also going to be adding a "saloon" that the player can enter and exit as well as a day and night cycle that will change the color of our game depending on the time of the day!

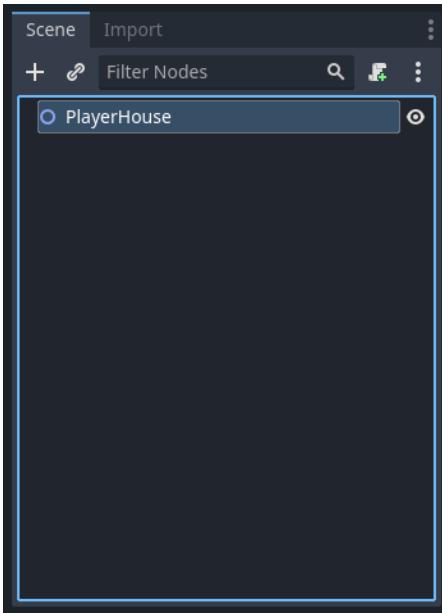
WHAT YOU WILL LEARN IN THIS PART:

- How to trigger node visibilities via the Area2D node.
- How to change game scenes in the code.
- How to work with the CanvasModulate node.
- How to access the systems time.

This section will show you two ways of having your Player explore more of the world. One will show/hide nodes to simulate us entering existing buildings, and the other one will change the scene that your player is in - like what you'd get in Stardew Valley.

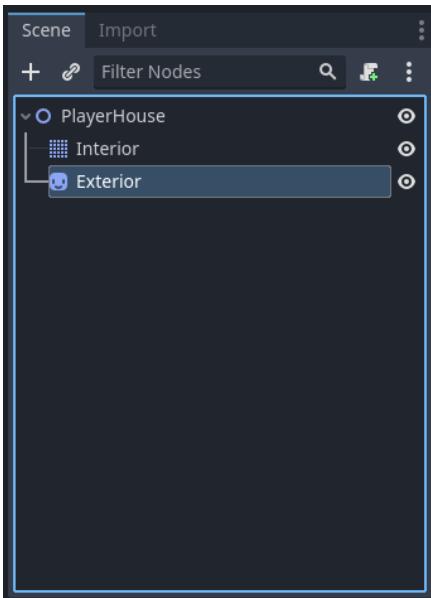
WORLD TRANSITIONS OPTION 1

Since our Player will be sleeping, we need to create a new scene for our player's house. Create a new scene with a Node2D node as its root node. We used this node for our EnemySpawner and Main scene, so you should already be familiar with it. It's the base node for Godot nodes, so it can contain any nodes. Rename the root node as "PlayerHouse" and save this scene under your Scenes folder.

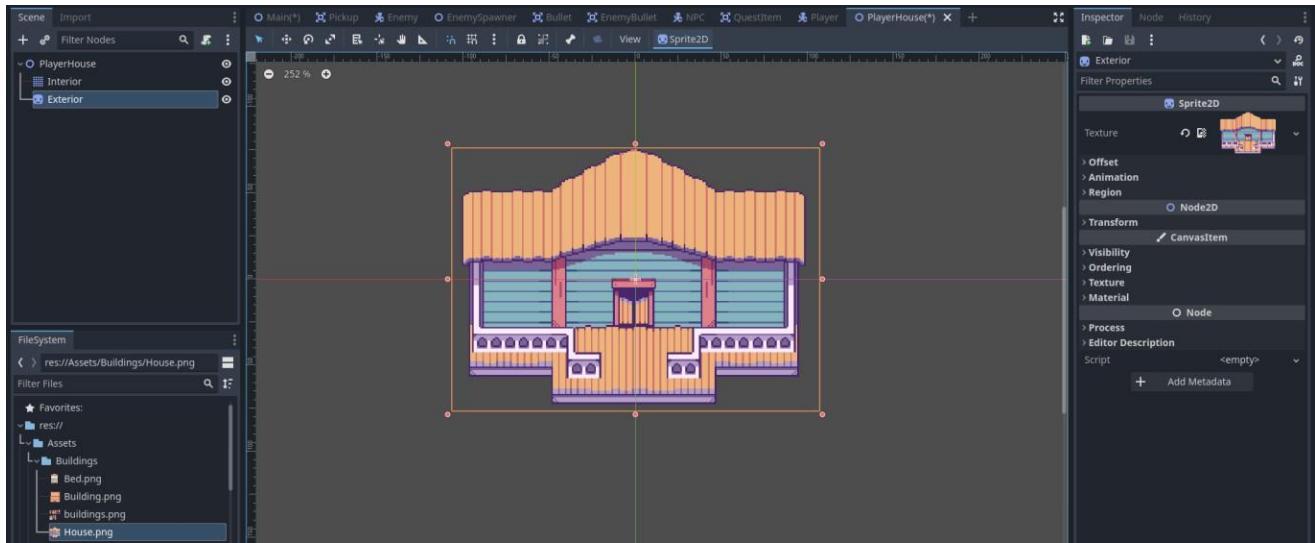


This scene will have a TileMap node for the floor, and a Sprite2D node for the image of the roof. When our player enters the house, the roof will be hidden and the floor and furniture inside the house will be shown. When the player exits the house, the roof will be made visible again.

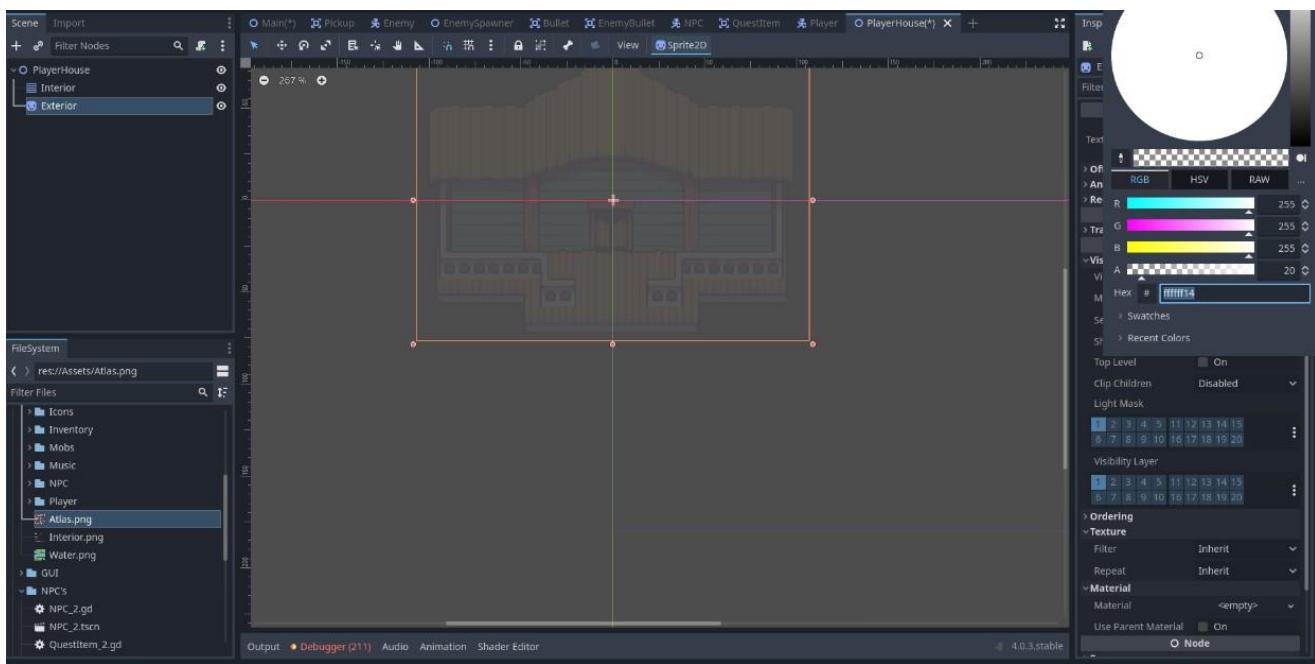
Let's add a Tilemap node and rename it "Interior". Then add a Sprite2D node and rename it "Exterior".

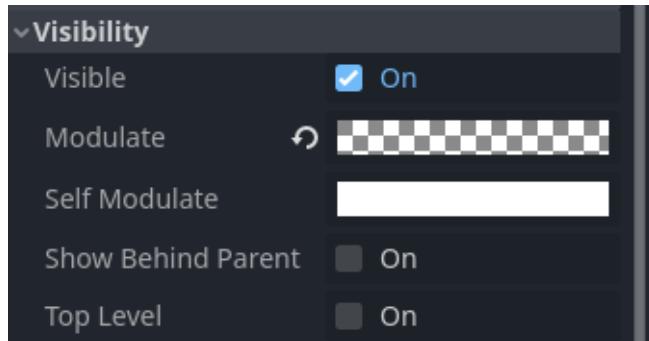


You can assign the Exterior sprite node an image of the exterior of our house. This image can be found in your Assets > Buildings > House.png directory.



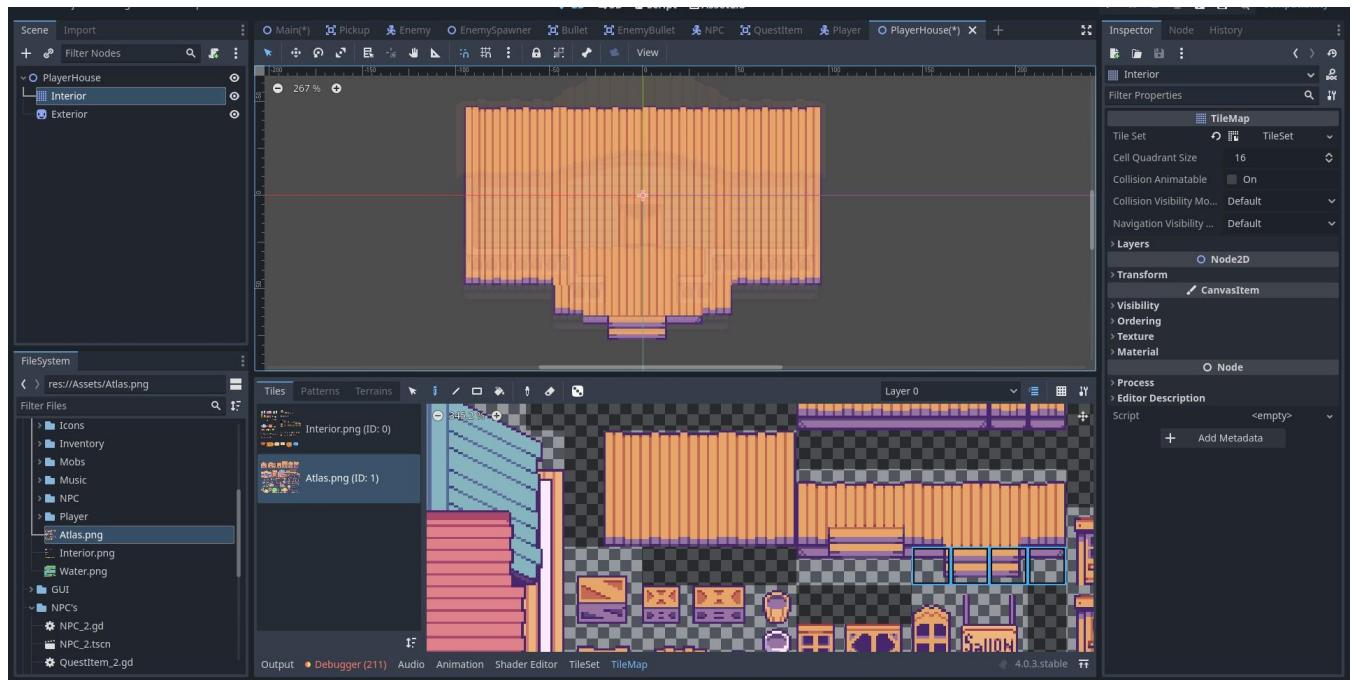
Temporarily change your Exterior node's Alpha (A) modulate value to "20" so that we can see through our roof to draw in our floor. You can change the modulate value underneath Visibility > Modulate in your Inspector Panel.



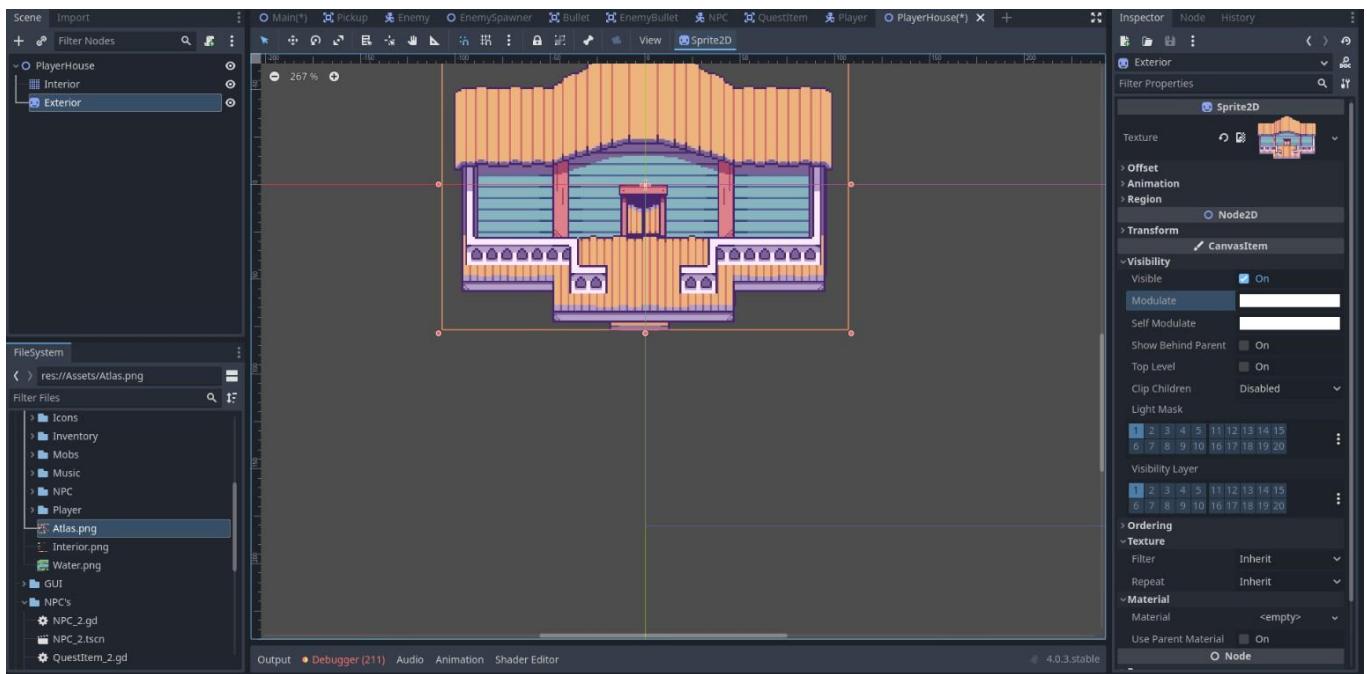


Now we have the size of our house, we can assign a Tilesheet resource to our TileMap node so that we can draw in our floor and add our furniture. In your Tilemap's Inspector panel, add a new tileset resource. In your Tileset panel below, add two new tilesheets: Interior and Atlas. These timesheets can be found in the root of your Assets folder.

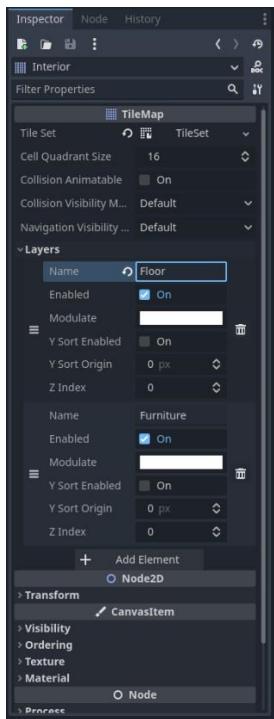
Now draw in your floor using the wood tiles in your Atlas tilesheet. You don't have to draw in terrains for this if you don't want to - you can just draw them freely.



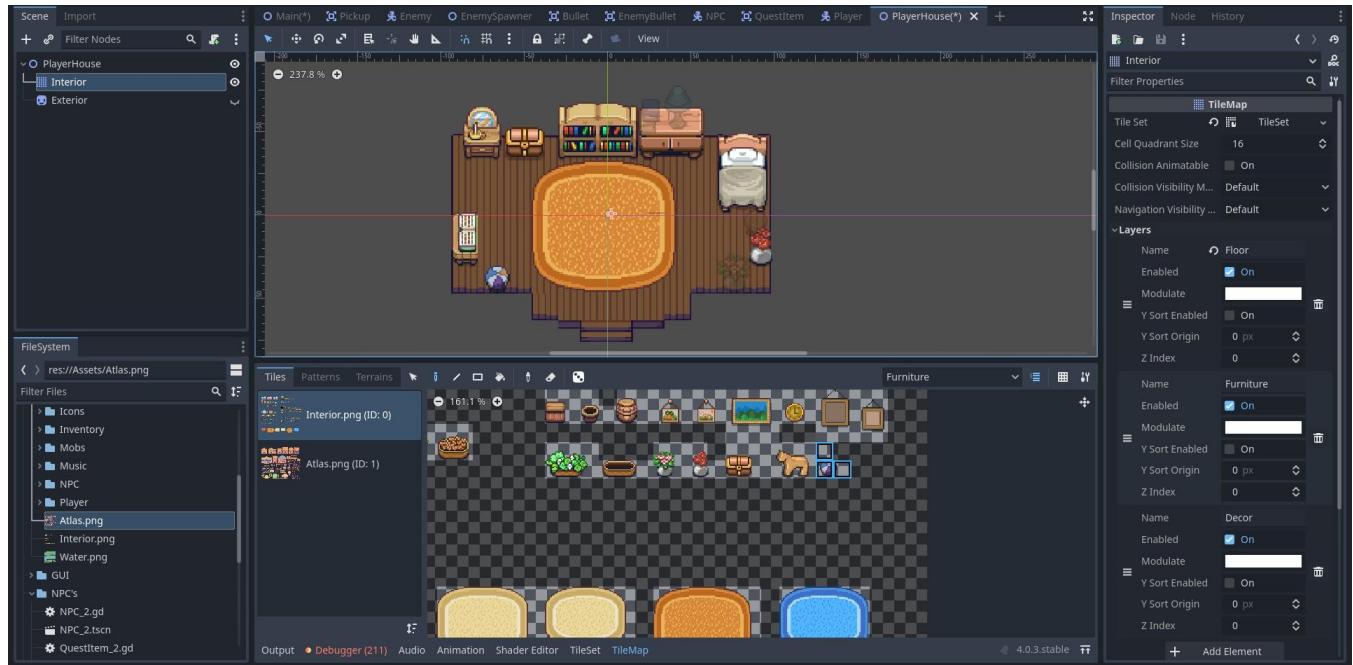
Then, reset your Exterior's modulate value to display your roof with full visibility again. It should look like this:



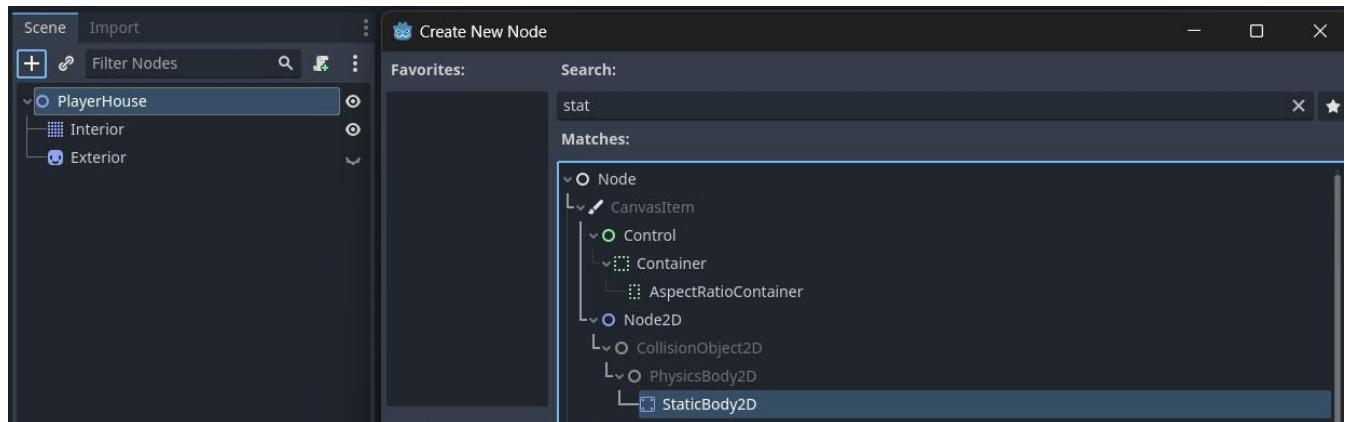
Now, let's add some furniture. You can add collisions to these tiles as we did in our Terrains part, but I'm going to do it another way today using pure collision shapes - just to speed things up! You should however add layers for your tiles so that we can draw our furniture on top of our floor.



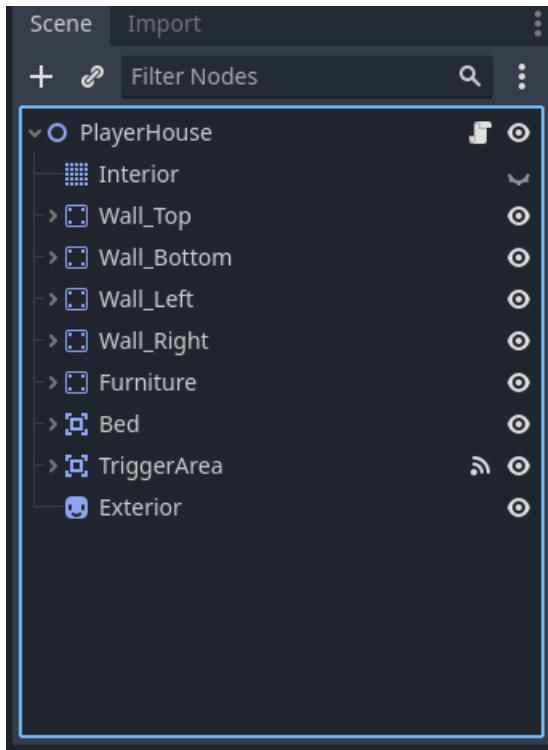
Now, paint some furniture using the Furniture tilesheet. You can add any objects you want here - just make sure there's enough space for your player to get to their bed.



Next, we need to add those collisions I talked about. We want our player to collide into each of our walls - as well as our furniture. To do this, we'll add [StaticBody2D](#) nodes that will hold the collisions for our walls and furniture. A static body is a simple body that doesn't move under physics simulation, i.e. it can't be moved by external forces or contacts, but its transformation can still be updated manually by the user. It is ideal for implementing objects in the environment, such as walls or platforms. We'll use this node as the container for our sky.

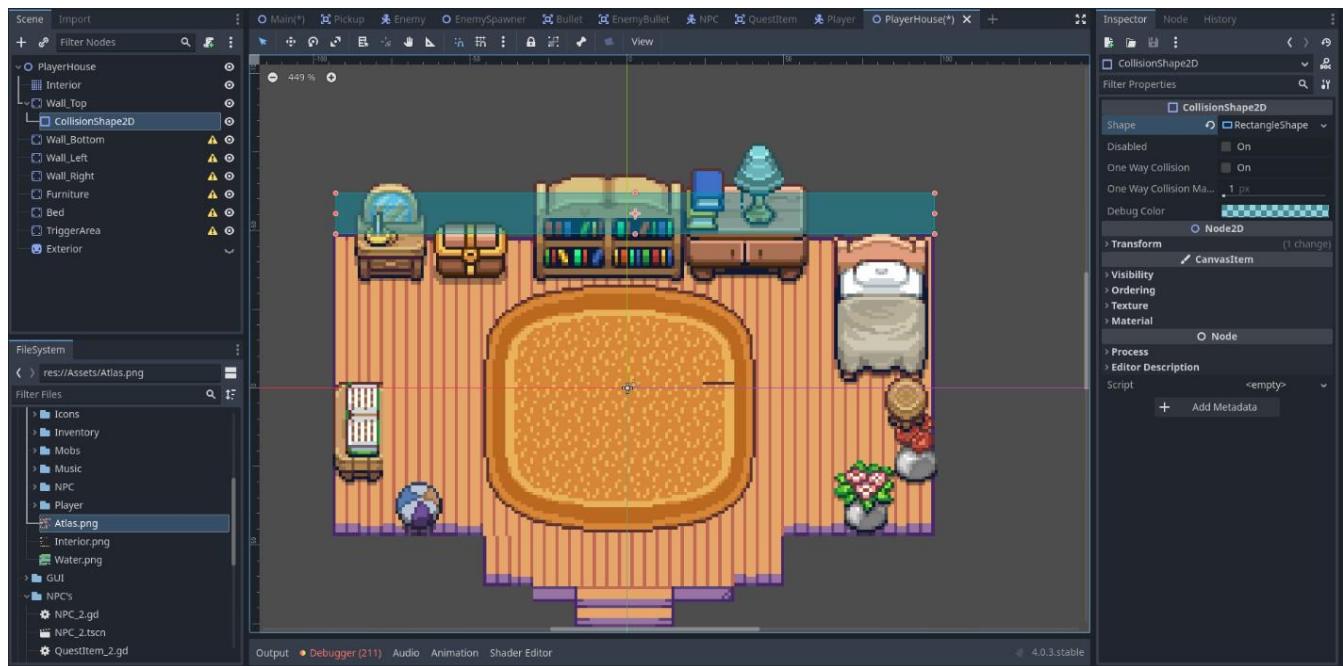


Add five StaticBody2D nodes (▣) and two Area2D (▢) nodes. Rename them as indicated in the image below.

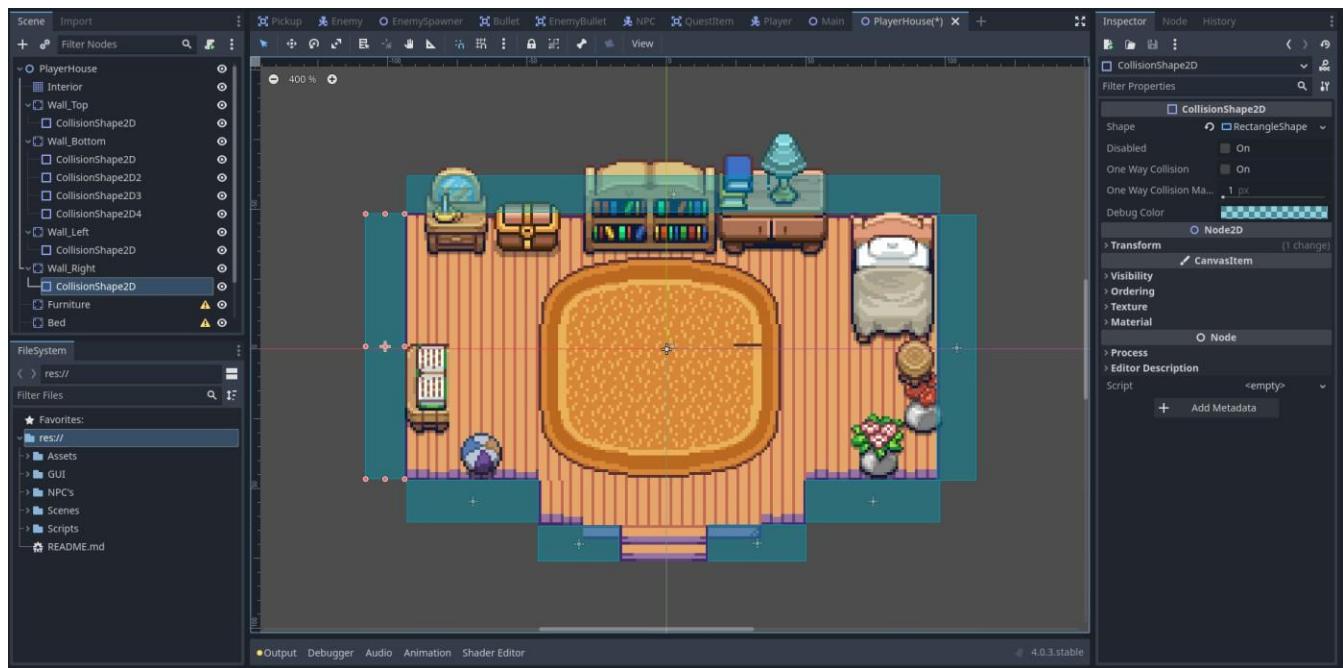


You will see a warning appear next to each of these new StaticBody2D and Area2D nodes. That is because they need a collision area.

Let's start with our Wall_Top node. Add a CollisionShape2D node to it. Make it a rectangle shape and draw it to be above the top of your TileMap node. This will ensure that our player does not go beyond this "wall".



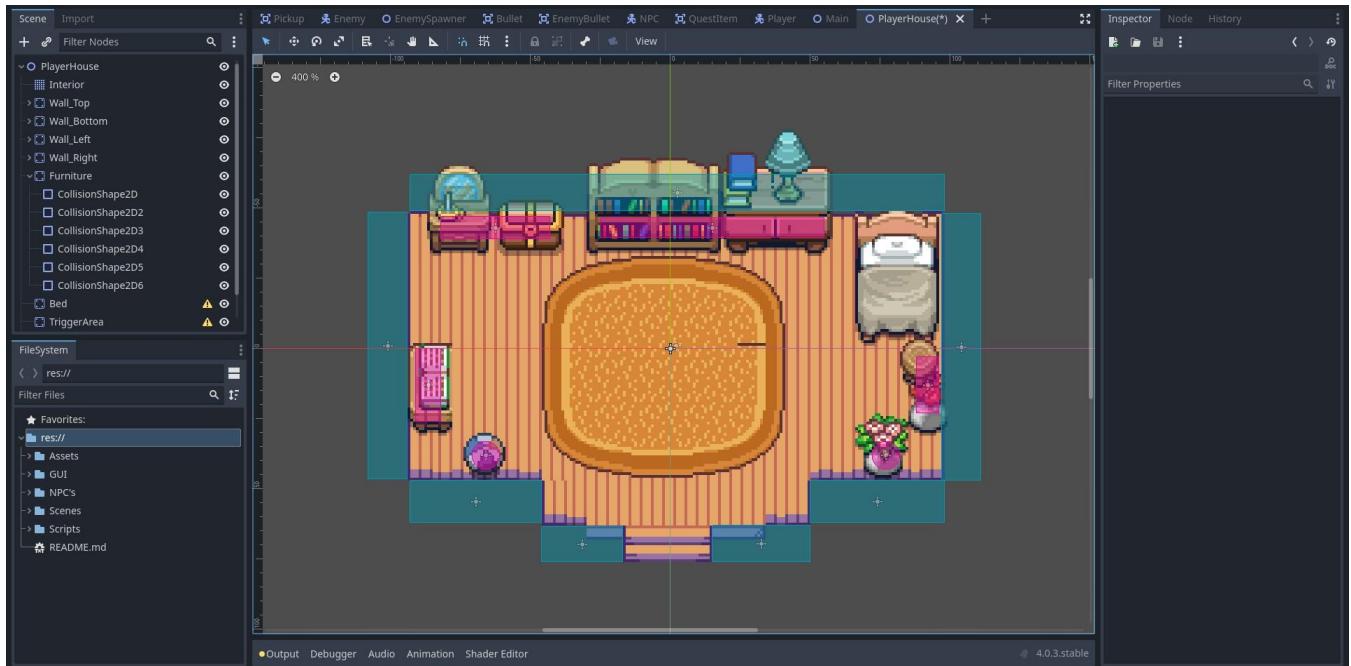
Do the same for the rest of your walls.



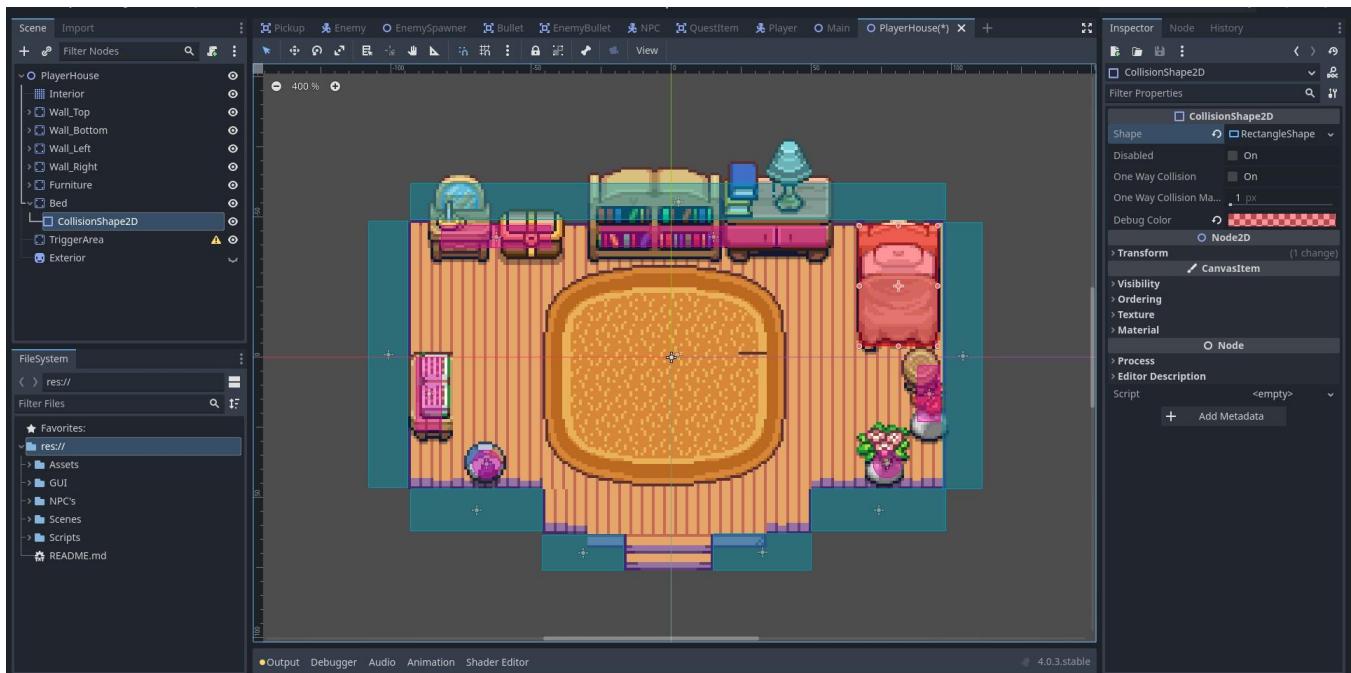
Now, for your furniture, add CollisionShapes2D nodes to them too (except your bed).

This would've been quicker via physics layers, but I'm giving you creation options here!

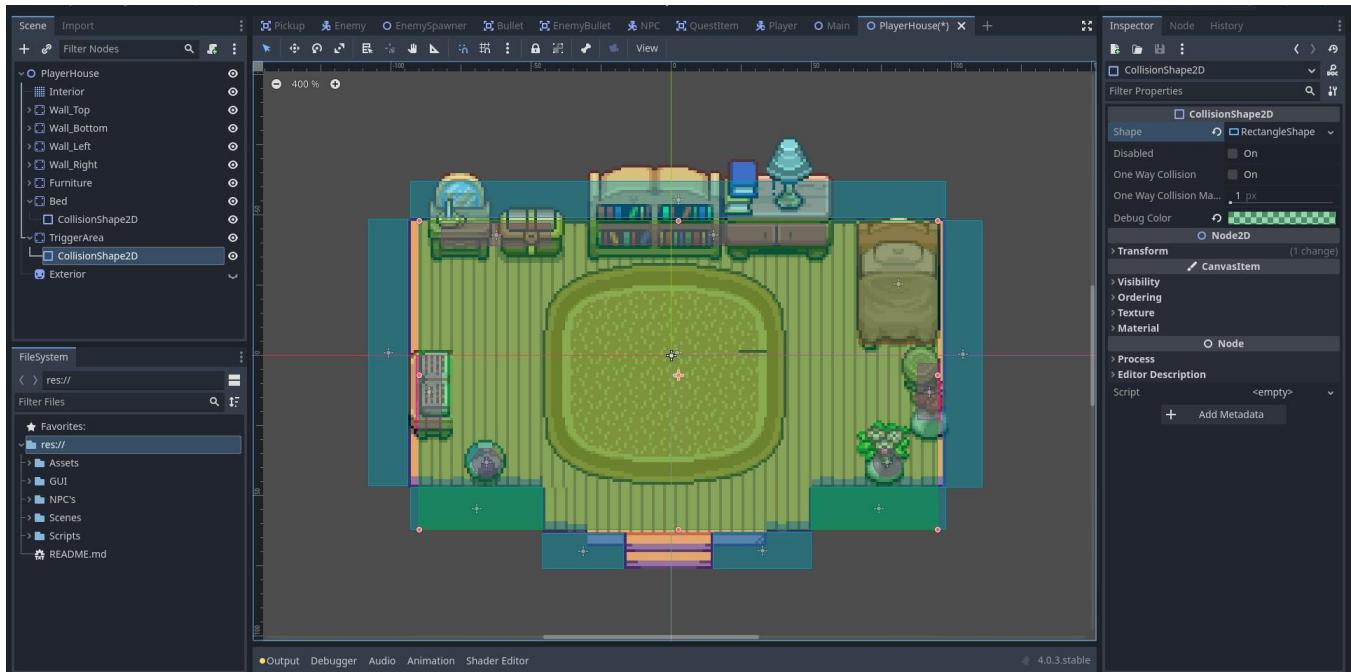
My collisions are highlighted in pink below.



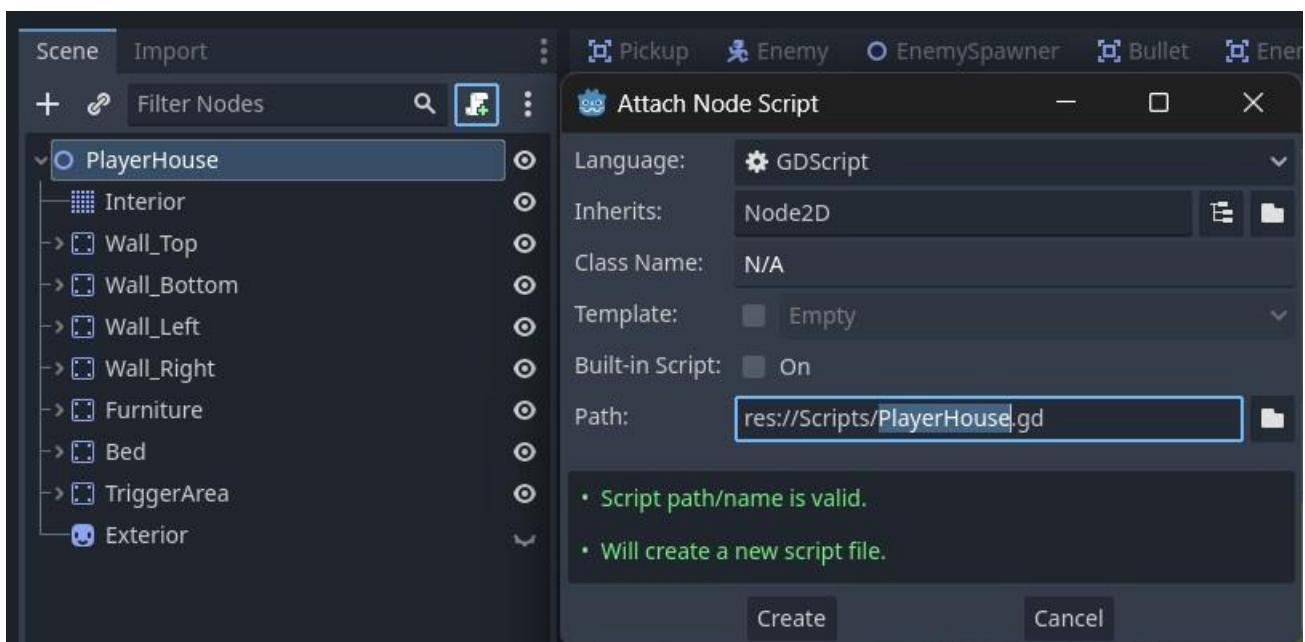
Add a collision to your bed. Mine is highlighted in red below.



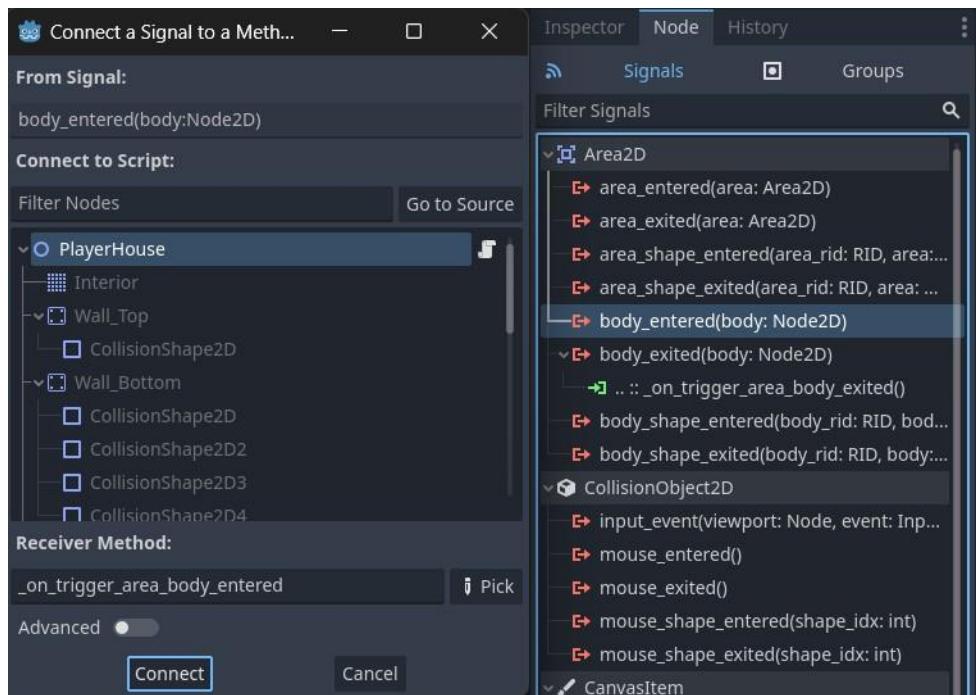
Now, let's add a collision to our TriggerArea. The TriggerArea will trigger the exterior and interior to be shown or hidden. Make this area big enough to cover your entire floor. Mine is highlighted in green below.



Let's add a script to our scene. Save it underneath your Scripts folder.



In this script, we want to show/hide our interior and exterior. We'll have to connect our TriggerArea node's body entered/ exited signals to our script. If the player enters the trigger area, the interior will be made visible & exterior hidden - and vice versa for when they exit the trigger area.



```
###PlayerHouse.gd

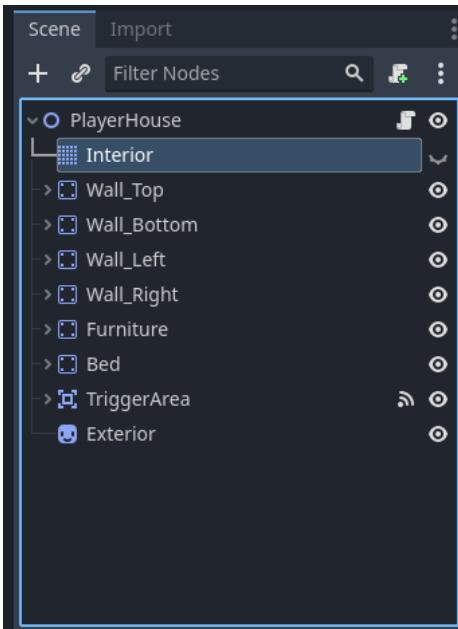
extends Node2D

# Node refs
@onready var exterior = $Exterior
@onready var interior = $Interior

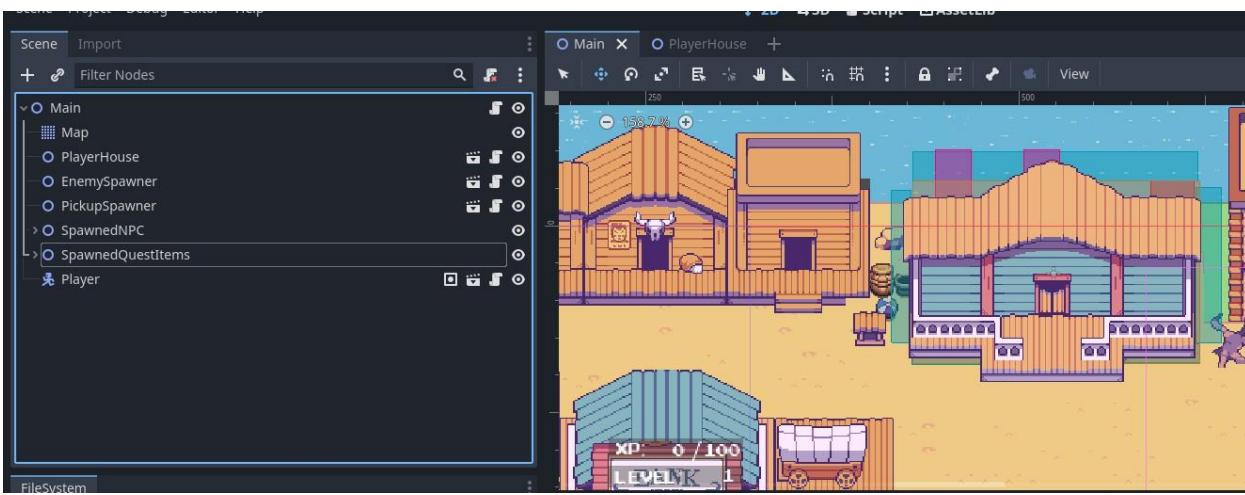
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        interior.show()
        exterior.hide()

func _on_trigger_area_body_exited(body):
    if body.is_in_group("player"):
        interior.hide()
        exterior.show()
```

Hide your Interior node by default so that our furniture does not stick out on game load.



Instance your PlayerHouse node in your Main scene, and make sure that you put the node above your Player node in your scene tree so that your Player node shows in front of it. Please don't place your house over any collision bodies - such as your water.



Now if you run your scene, and you run through the front door the roof should collapse and the interior should show. If you run out by the stairs, the roof should show, and the interior should be hidden. You should also not be able to run through your walls or furniture.



We have a problem now - anybody can enter our player's house, especially enemies! To fix this, we'll need to update our code to block our Enemy bodies. If they happen to stumble into our house, our code needs to redirect their movement back to the outside. We do this by redirecting their rotation every 4 seconds until they are out of our house.

```
###PlayerHouse.gd  
  
extends Node2D  
  
# Node refs
```

```

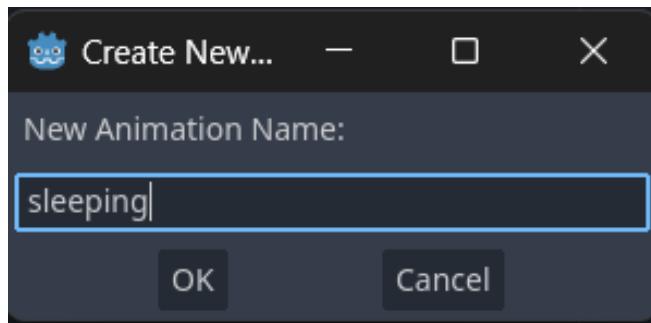
@onready var exterior = $Exterior
@onready var interior = $Interior

func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        interior.show()
        exterior.hide()
    #prevent enemy from entering our direction
    elif body.is_in_group("enemy"):
        body.direction = -body.direction
        body.timer = 16

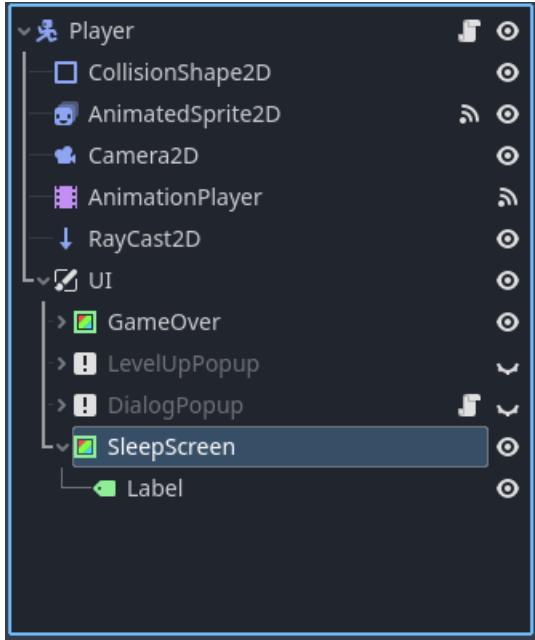
```

With our house set up, we can go ahead and tell our player to sleep if they interact with our bed. We can use the RayCast node to see if it hits any bodies named "Bed" (which will be our Bed StaticBody2D node). If it does, we'll play an animation and restore our player's stats. We already have the input for this setup underneath *ui_interact* - meaning we will press TAB close to our bed to go to sleep.

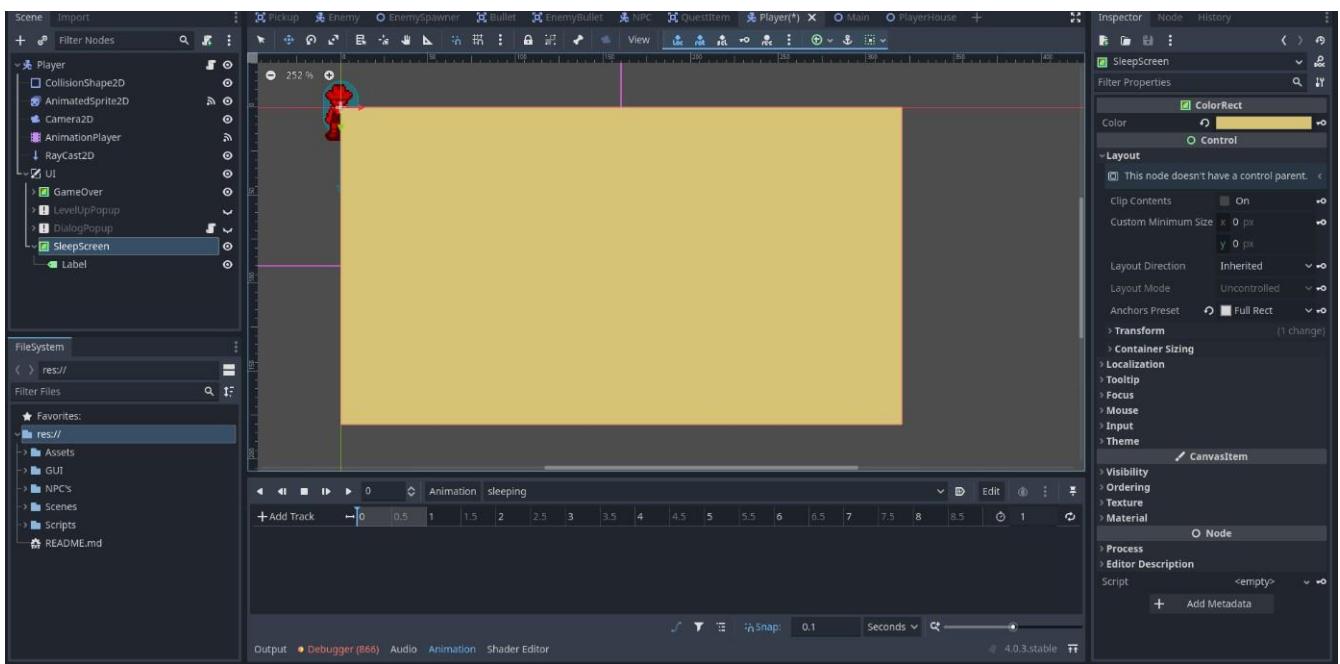
Let's create our sleeping animation before we update our Player's code. In your Player scenes AnimationPlayer, add a new animation called "sleeping".



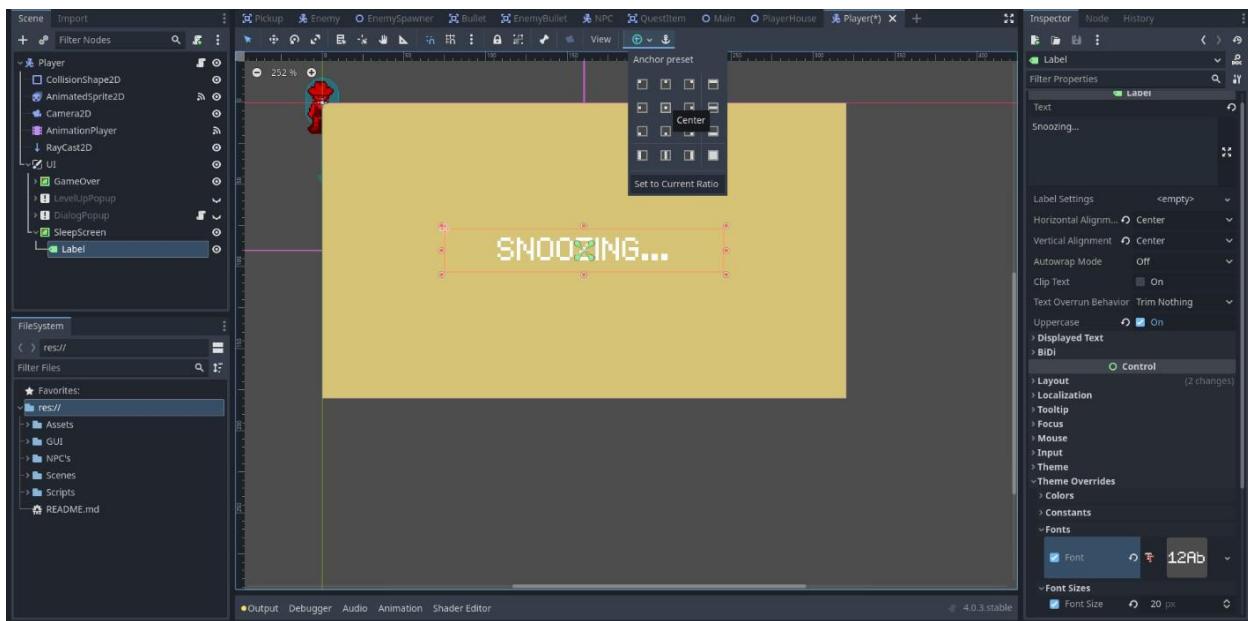
Before we add our animation track, we need to create the UI for it. We want our screen to show a "Sleeping" screen when we sleep. Underneath your UI layer in your Player scene, add a new ColorRect with a Label as its child. Rename the ColorRect to "SleepScreen".



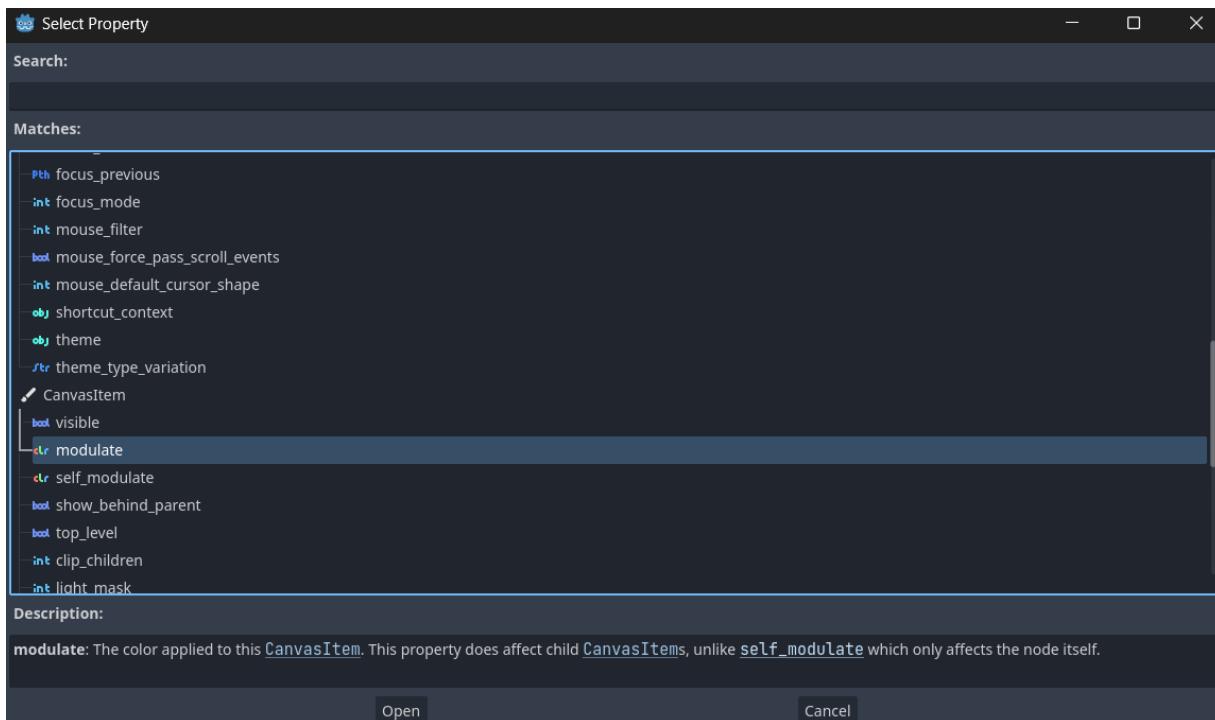
Change the SleepScreen node's anchor preset to "Full Rect", and its color to #d6c376.



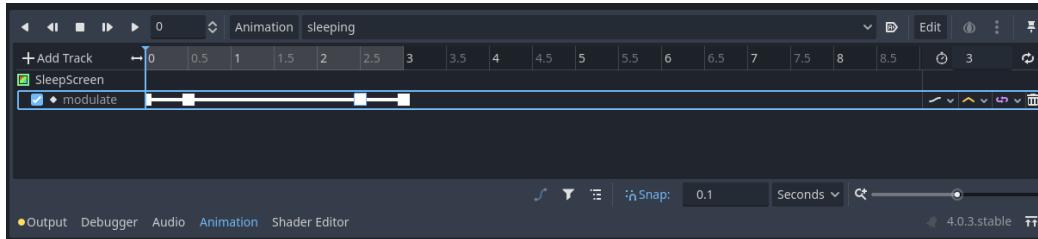
Change the label's text to "Snoozing...". Change its font to "Schrödinger"; font size (20), and vertical and horizontal alignment (center). Its anchor-preset should be center.



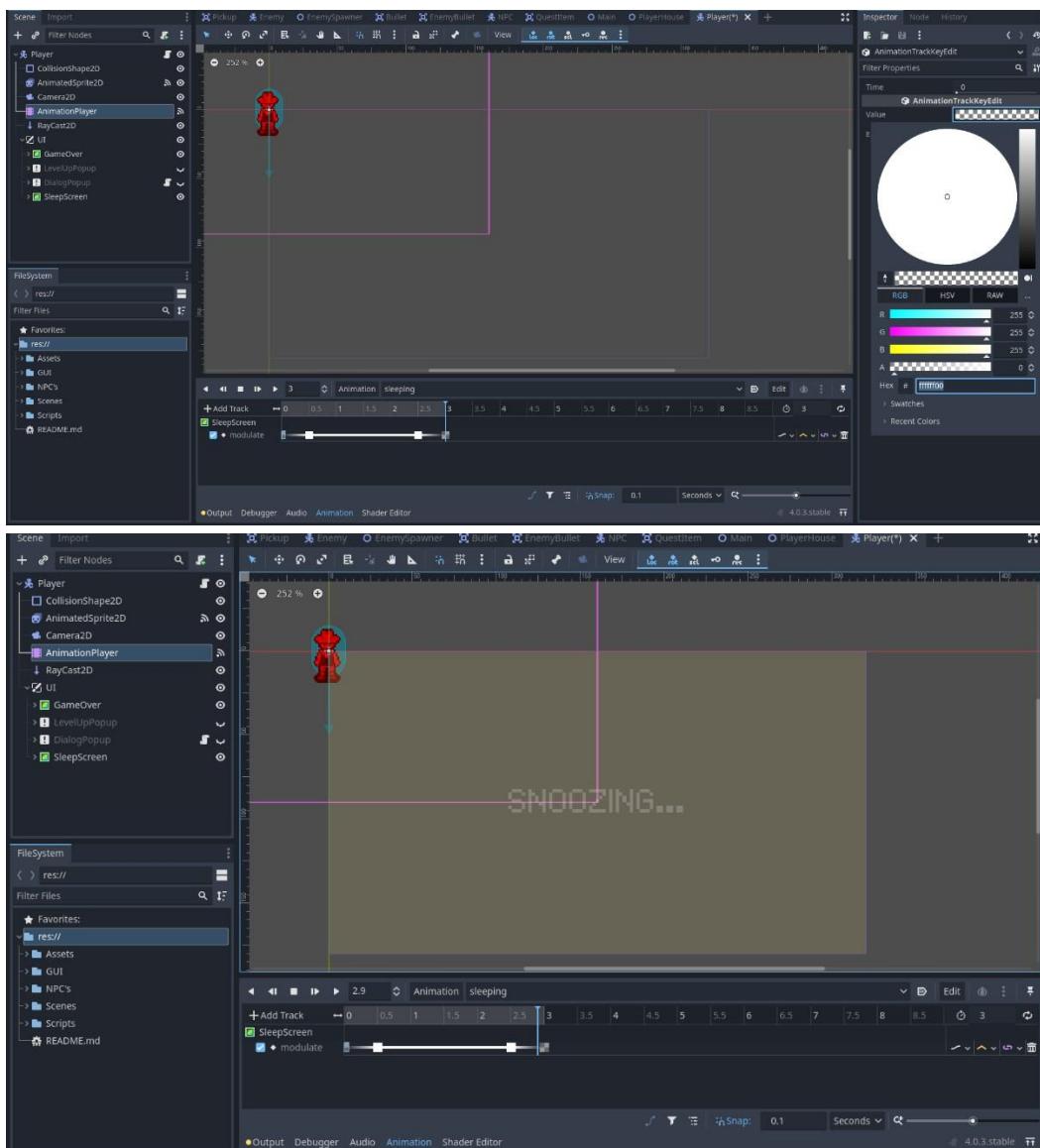
Now back in our AnimationPlayer's "sleeping" animation, we want to add an animation that makes our sleeping screen visible. We've done something like this before with our Game Over screen. Add a property track, connect it to your SleepScreen node, and select the property to be our modulate property.



We want our sleep screen to show for 3 seconds, so change the time to 3. Also add three keys at the following timestamps (0, 0.5, 2.5, 3).



Change the Alpha(A) value of your keyframe 0 and 3 to 0 (invisible). We want it to flow as invisible -> visible -> invisible. If you play your animation, it should flow as desired.

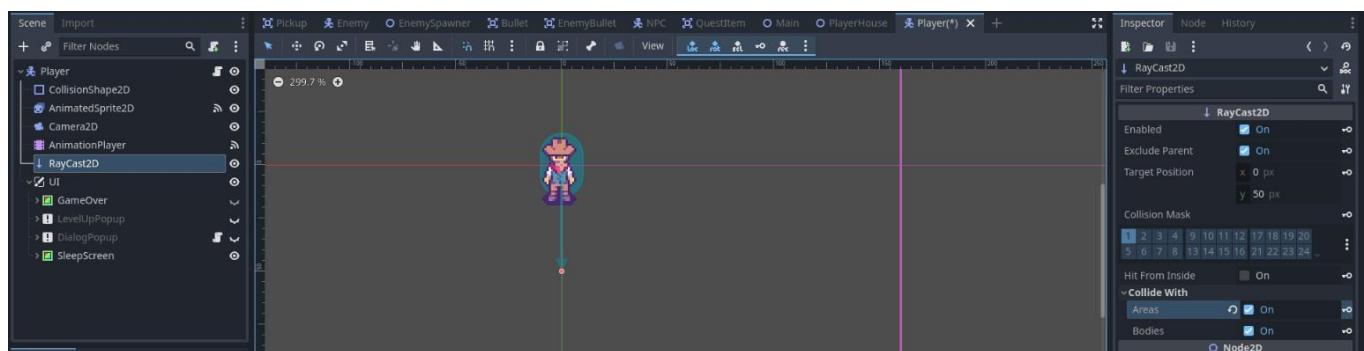


Now in your Player script, underneath your `ui_interact` input, let's check if the Raycast is hitting our Bed. If it is, we will play our sleepscreen animation, and restore our player's health and stamina values.

```
### Player.gd

func _input(event):
    #older code
    #interact with world
    elif event.is_action_pressed("ui_interact"):
        var target = ray_cast.get_collider()
        if target != null:
            if target.is_in_group("NPC"):
                # Talk to NPC
                target.dialog()
                return
            #go to sleep
            if target.name == "Bed":
                # play sleep screen
                animation_player.play("sleeping")
                health = max_health
                stamina = max_stamina
                health_updated.emit(health, max_health)
                stamina_updated.emit(stamina, max_stamina)
                return
```

For this to work, we need to enable our RayCast2D node to be able to collide with bodies (such as our Area2D body).



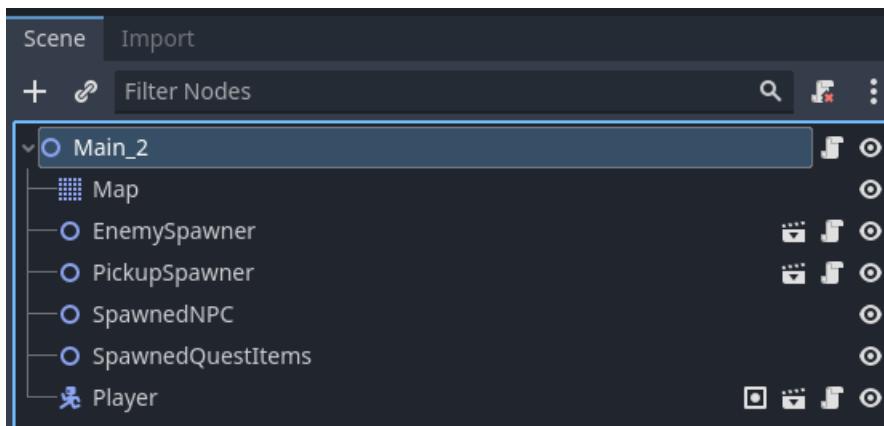
If you were to run your scene now and press TAB near your bed, your animation will play and your stats will refill!



WORLD TRANSITIONS OPTION 2

Now, let's start on our second world transition. In this one, we'll create a new scene for our saloon - you can make it an entirely different map if you want. This scene can be anything. It could be a beach area, a cave, a mansion - anything, but we're going to create a simple saloon that will show us how to change complete areas via scene transitions.

We can duplicate our Main scene for this. Rename this scene as "Main_2" and detach the signals and the script. Also, remove all the painted tiles in your tilemap. We want this scene to be a blank sheet.

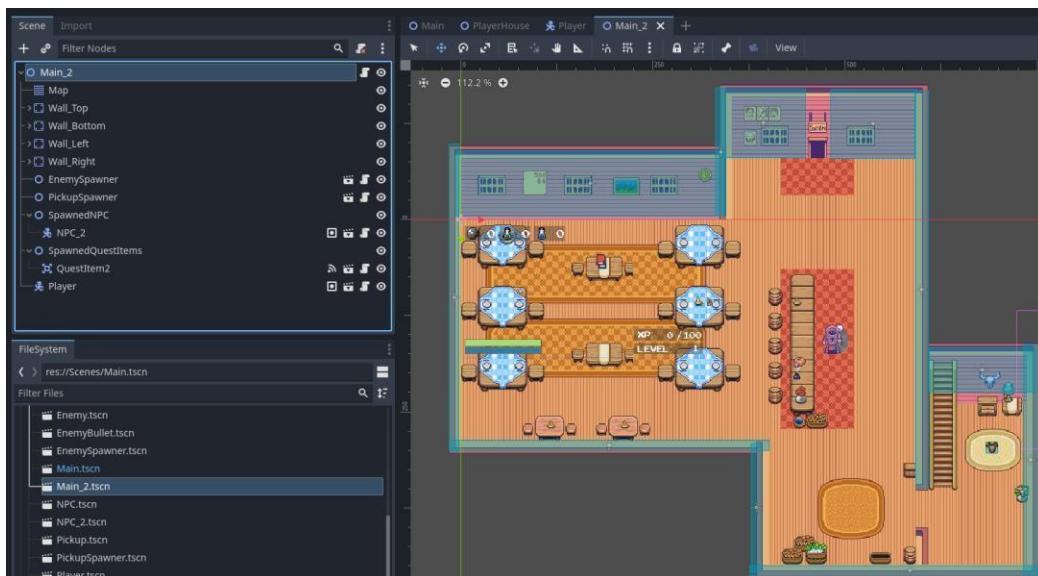


Now, it is here where you can go crazy and create the Saloon of your dreams - or if you don't want to create a Saloon and you instead want your Player to travel to a new area such as a forest, go ahead and create that. Just remember to add Collisions (Physics Layers) to your tilemap if you are creating a new world map, just like we did in our Map creation tutorial.

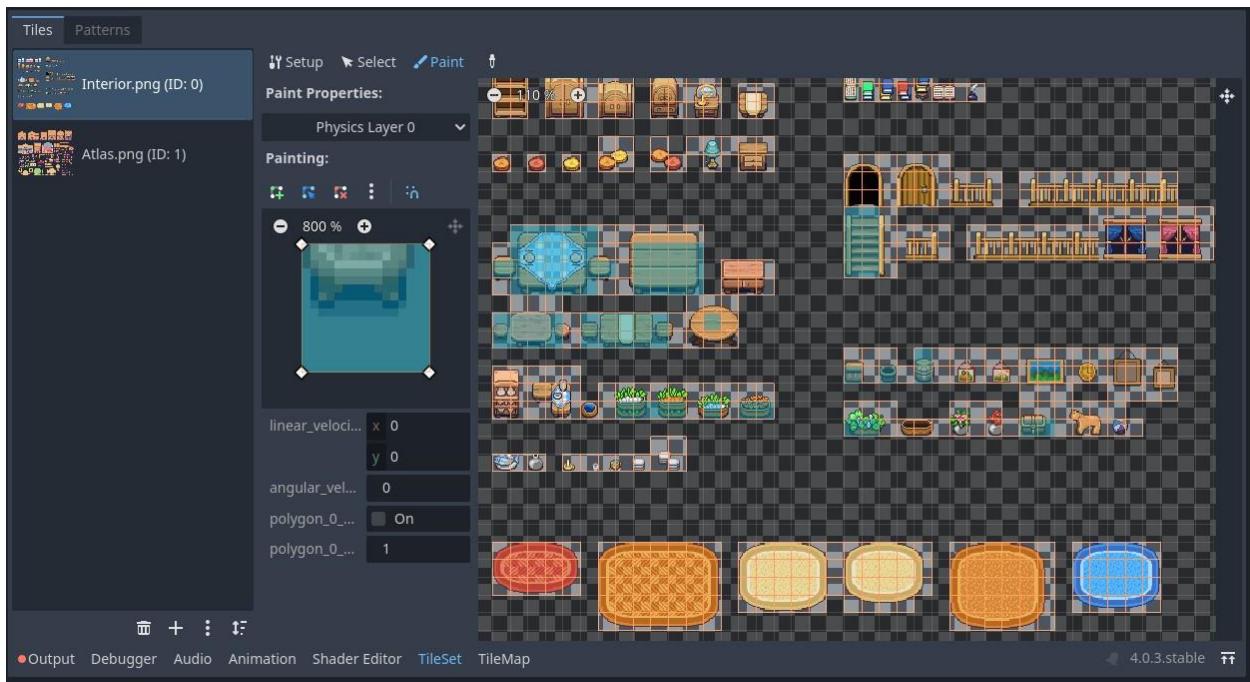
I'm going to go ahead and make a Saloon.



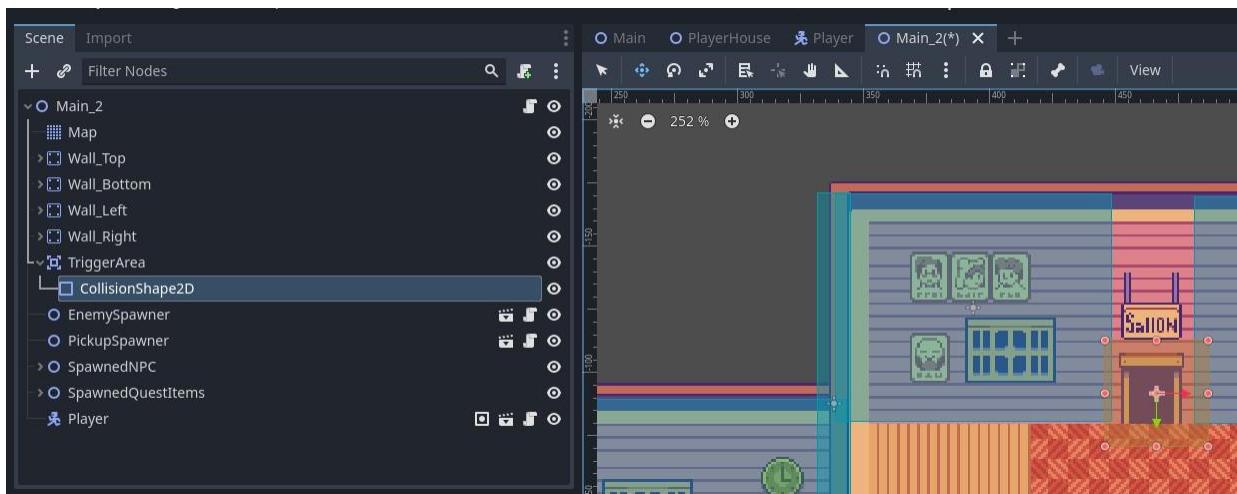
Now that we have our new area, we need to add our collisions to our walls.



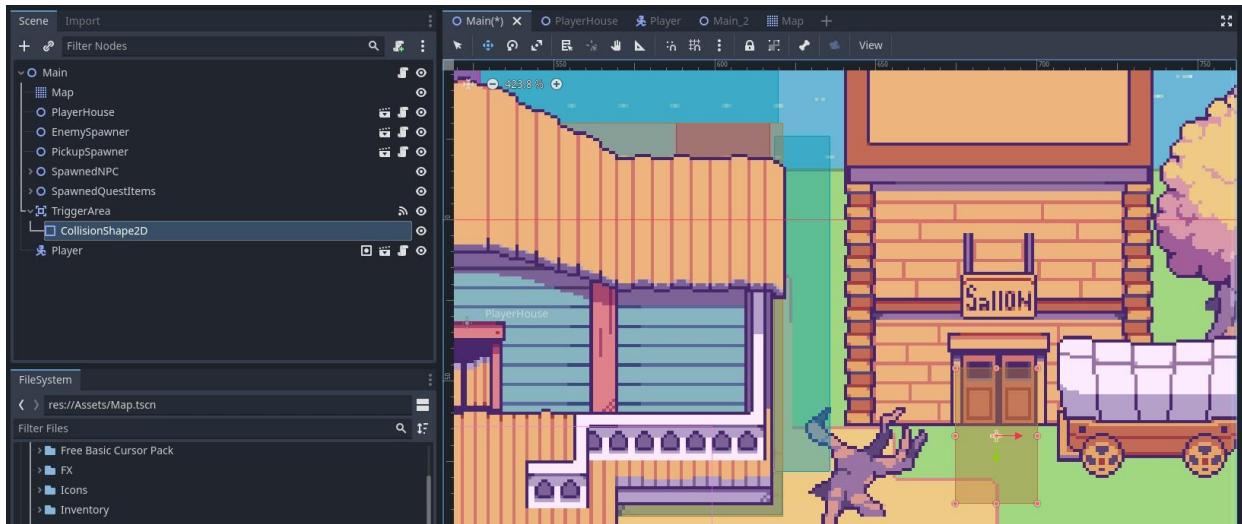
Next, we'll need to add collisions to our furniture. To speed up this process, I'm going to add collisions to my furniture via the Physics Layers from the TileMap resource. You can do the same. If you forgot how, please refer to the [documentation](#) or our previous tutorial (part 4) on this.



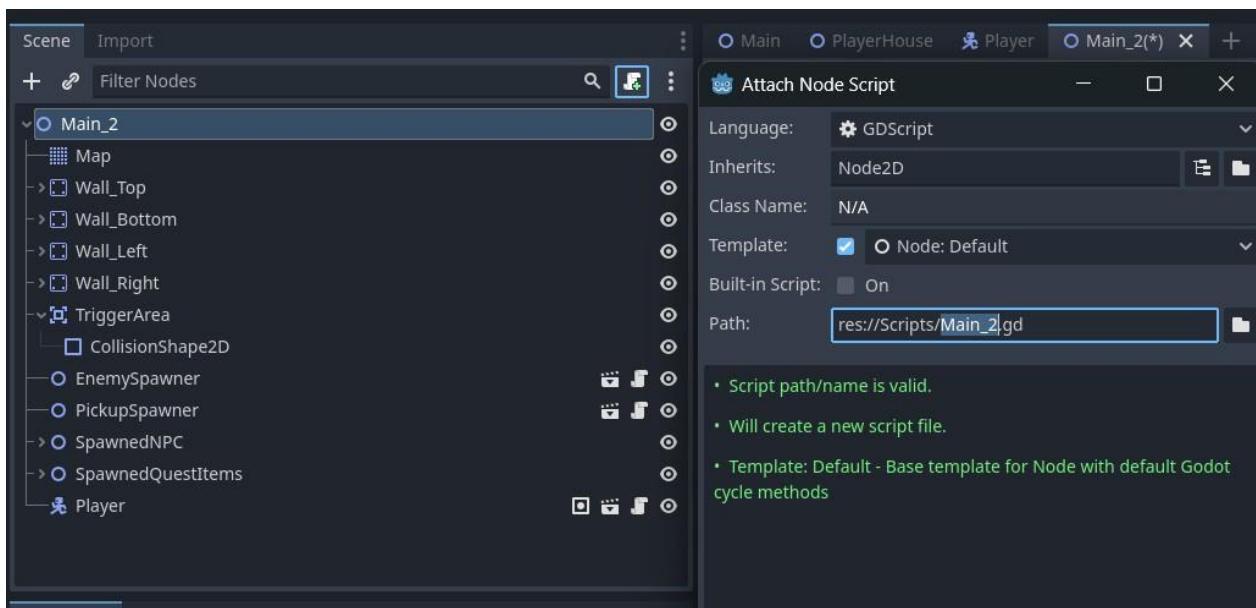
Then add the TriggerArea collision by the exit or door to your scene. If our player runs through this, they'll be transported back to our Main scene's map.



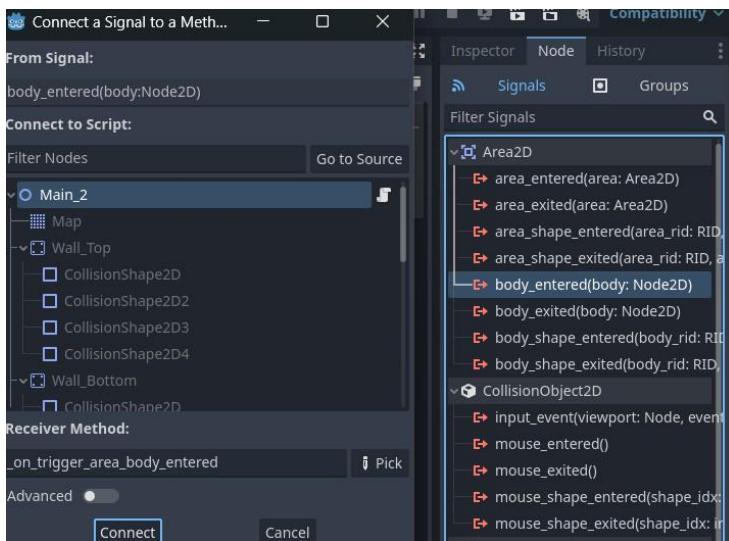
Do the same in your Main scene so that we can go to the Main_2 scene.



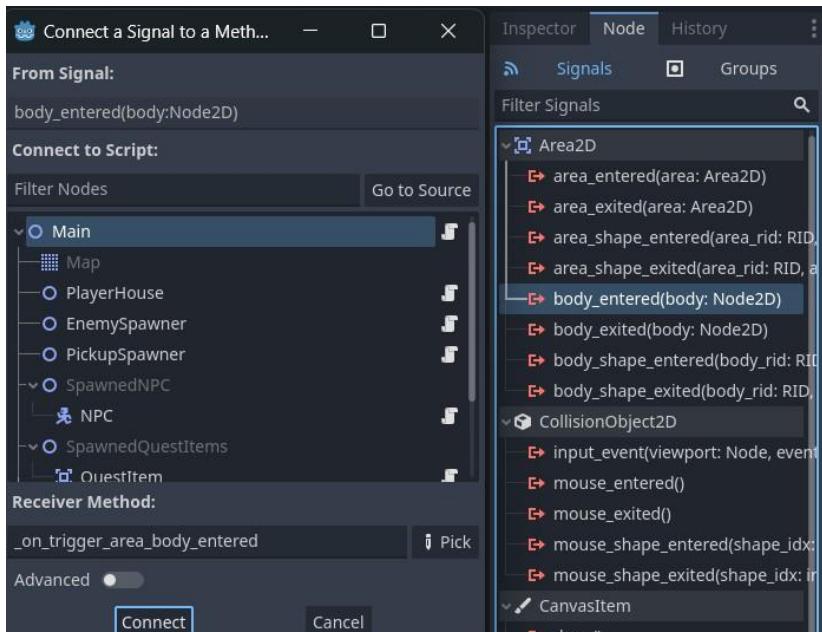
Next, let's attach a new script to the root node of our Main_2 scene.



In this script, we want the player to be able to go back to the Main scene if they run through our Trigger Area. You might already have guessed that we will have to connect our *body_entered()* signal from our TriggerArea node to our Main_2 script to do this.



Also connect your Main scene's TriggerArea body_entered signal to your Main script.



Okay now to change our scenes, we'll need to keep track of the current scene that our player is in so that we can dynamically change our scene references. Currently, we are referencing our Main scene everywhere, so we will get errors when we go to our Main_2 scene.

For example, previously we had this path referencing in our scenes:

```
player = get_tree().root.get_node("/Main/Player")
```

What if we wanted to reuse this node in our Main_2 scene? Our game would crash because Main/Player does not exist in that tree! But with the help of our Global script, we can dynamically change the name of the scene based on the current scene that our player is in.

We will end up using something like this:

```
player = get_tree().root.get_node("%s/Player" % Global.current_scene_name)
```

Now, in our Global script, let's set the name of our current loaded scene in our ready function. This could be "MainMenu", "Menu", or "Main_2". We'll change this later on in its own custom function.

```
### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
@onready var bullet_scene = preload("res://Scenes/Bullet.tscn")
@onready var enemy_bullet_scene = preload("res://Scenes/EnemyBullet.tscn")

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }

# TileMap layers
const WATER_LAYER = 0
const GRASS_LAYER = 1
const SAND_LAYER = 2
const FOLIAGE_LAYER = 3
const EXTERIOR_1_LAYER = 4
const EXTERIOR_2_LAYER = 5

# current scene
var current_scene_name

#set current scene on load
func _ready():
    current_scene_name = get_tree().get_current_scene().name
```

Now in our other scripts, we can switch out all of our static "/Main/" paths to reference our `current_scene_name` variable instead. We'll do this by formatting our path string using the % operator to insert the value of `Global.current_scene_name`. The %s is a placeholder for a string value, such as "Main".

```
### Bullet.gd
extends Area2D

# Node refs
@onready var tilemap = get_tree().root.get_node("%s/Map" %
Global.current_scene_name)
@onready var animated_sprite = $AnimatedSprite2D
```

```
### Enemy.gd

# Node refs
@onready var player = get_tree().root.get_node("%s/Player" %
Global.current_scene_name)

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        #stop movement
        timer_node.stop()
        direction = Vector2.ZERO
        #stop health regeneration
        set_process(false)
        #trigger animation finished signal
        is_attacking = true
        #Finally, we play the death animation
        animation_sprite.play("death")
        #add xp values
        player.update_xp(70)
        death.emit()
        #drop loot randomly at a 90% chance
        if rng.randf() < 0.9:
            var pickup = Global.pickups_scene.instantiate()
```

```

pickup.item = rng.randi() % 3 #we have three pickups in our enum
get_tree().root.get_node("%s/PickupSpawner/SpawnedPickups" %
Global.current_scene_name).call_deferred("add_child", pickup)
pickup.position = position

# Bullet & removal
func _on_animated_sprite_2d_animation_finished():
    if animation_sprite.animation == "death":
        get_tree().queue_delete(self)
    is_attacking = false
    # Instantiate Bullet
    if animation_sprite.animation.begins_with("attack_"):
        var bullet = Global.enemy_bullet_scene.instantiate()
        bullet.damage = bullet_damage
        bullet.direction = new_direction.normalized()
        # Place it 8 pixels away in front of the enemy
        bullet.position = player.position + new_direction.normalized() * 8
        get_tree().root.get_node("%s" %
Global.current_scene_name).add_child(bullet)

```

```

### EnemyBullet.gd
extends Area2D

# Node refs
@onready var tilemap = get_tree().root.get_node("%s/Map" %
Global.current_scene_name)
@onready var animated_sprite = $AnimatedSprite2D

```

```

###EnemySpawner.gd

extends Node2D

# Node refs
@onready var spawned_enemies = $SpawnedEnemies
@onready var tilemap = get_tree().root.get_node("%s/Map" %
Global.current_scene_name)

```

```

### NPC
extends CharacterBody2D

# Node refs

```

```

@onready var dialog_popup = get_tree().root.get_node("%s/Player/UI/DialogPopup" %
Global.current_scene_name)
@onready var player = get_tree().root.get_node("%s/Player" %
Global.current_scene_name)
@onready var animation_sprite = $AnimatedSprite2D

```

```

### PickupSpawner.gd

extends Node2D

# Node refs
@onready var map = get_tree().root.get_node("%s/Map" % Global.current_scene_name)
@onready var spawned_pickups = $SpawnedPickups

```

```

### Player.gd

# Reset Animation states
func _on_animated_sprite_2d_animation_finished():
    is_attacking = false

    # Instantiate Bullet
    if animation_sprite.animation.begins_with("attack_"):
        var bullet = Global.bullet_scene.instantiate()
        bullet.damage = bullet_damage
        bullet.direction = new_direction.normalized()
        # Place it 4-5 pixels away in front of the player
        bullet.position = position + new_direction.normalized() * 4
        get_tree().root.get_node("%s" %
Global.current_scene_name).add_child(bullet)

```

```

### QuestItem.gd

extends Area2D

#npc node reference
@onready var npc = get_tree().root.get_node("%s/SpawnedNPC/NPC" %
Global.current_scene_name)

```

Let's create a function that will allow us to change scenes in our Global script. This will update our scene name to be the current scene that our player is in. In this function, we will get the current scene and then free it. So if our current scene is our Main scene, and

we are moving to the Main_2 scene, it will get our Main and free it from our scene tree. Then it will load and instantiate our new scene and add our new scene as a child of the root node. Finally, it will set the new scene as the current scene.

This sequence of operations effectively replaces the current scene with the new one. We'll also later on load the player's data (such as their health, coins, pickup amounts, etc.) when the player enters their new scene after we've added our saving & loading system. This will fix the issue of the data not persisting when our player changes areas. Then, when the player leaves an area it will save their data so that it can be loaded when they enter the new area.

```
### Global.gd

extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
@onready var bullet_scene = preload("res://Scenes/Bullet.tscn")
@onready var enemy_bullet_scene = preload("res://Scenes/EnemyBullet.tscn")

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }

# TileMap layers
const WATER_LAYER = 0
const GRASS_LAYER = 1
const SAND_LAYER = 2
const FOLIAGE_LAYER = 3
const EXTERIOR_1_LAYER = 4
const EXTERIOR_2_LAYER = 5

# current scene
var current_scene_name

# ----- Scene handling -----
#set current scene on load
func _ready():
    current_scene_name = get_tree().get_current_scene().name

# Change scene
```

```

func change_scene(scene_path):
    # Get the current scene
    current_scene_name = scene_path.get_file().get_basename()
    var current_scene =
        get_tree().get_root().get_child(get_tree().get_root().get_child_count() - 1)
    # Free it for the new scene
    current_scene.queue_free()
    # Change the scene
    var new_scene = load(scene_path).instantiate()
    get_tree().get_root().call_deferred("add_child", new_scene)
    get_tree().call_deferred("set_current_scene", new_scene)

```

Now in our Main and Main_2 scripts we can change our scene via this Global function by calling it and passing the scene path that we want to change to as a parameter. We'll then queue_free the scene so that Main or Main_2 is effectively removed from our scene tree.

```

### Main.gd

extends Node2D

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main_2.tscn")
        queue_free()

```

```

### Main_2.gd

extends Node2D

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main.tscn")
        queue_free()

```

In our Main scenes, we now free our scene resources directly after changing our scene. When you call `queue_free()`, it doesn't immediately delete the node. It schedules the node to be deleted at the end of the current frame or later when it's safe to do so.

Hence, there's a chance that our old scene will still be present before the new scene is properly set. The queue_free() function is asynchronous for good reasons, to prevent situations where nodes get deleted while they are still in use. To avoid this potential race condition between scene switching and auto-saving, we can use a signal in our Global script that gets emitted once a scene has fully transitioned.

```
### Global.gd

# older code

#notifies scene change
signal scene_changed()
```

Then, we emit this signal in our change_scene function after we've loaded the new scene.

```
### Global.gd

# older code

# Change scene
func change_scene(scene_path):
    # Get the current scene
    current_scene_name = scene_path.get_file().get_basename()
    var current_scene =
        get_tree().get_root().get_child(get_tree().get_root().get_child_count() - 1)
    # Free it for the new scene
    current_scene.queue_free()
    # Change the scene
    var new_scene = load(scene_path).instantiate()
    get_tree().get_root().call_deferred("add_child", new_scene)
    get_tree().call_deferred("set_current_scene", new_scene)
    call_deferred("post_scene_change_initialization")

func post_scene_change_initialization():
    scene_changed.emit()
```

Then, in Main.gd and Main_2.gd, you connect to this signal and queue_free when it's emitted:

```
### Main.gd
```

```

extends Node2D

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main_2.tscn")
        Global.scene_changed.connect(_on_scene_changed)

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()

```

```

### Main_2.gd

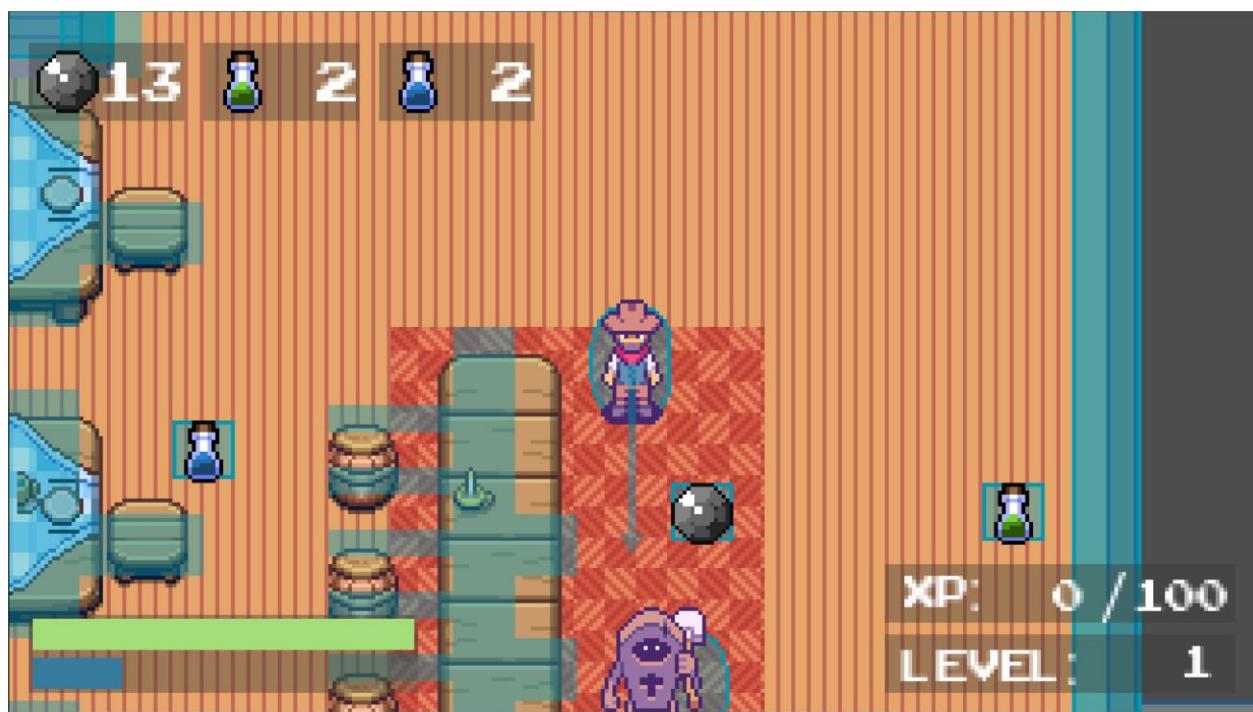
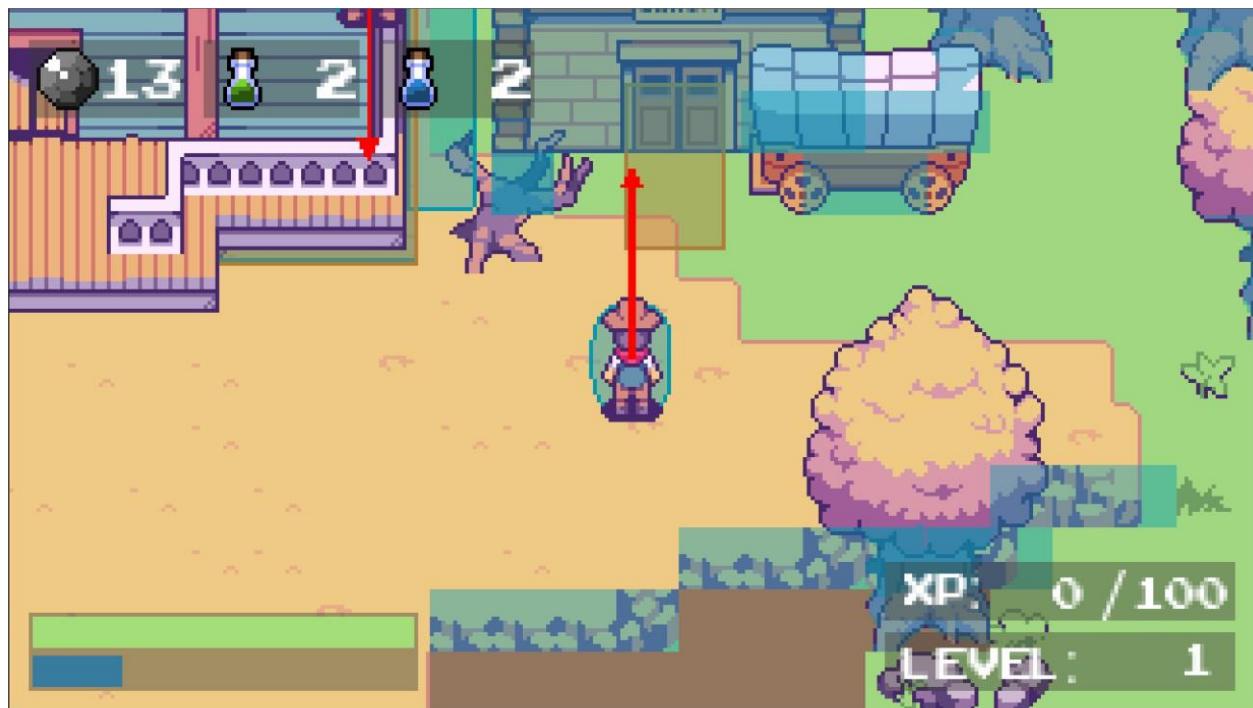
extends Node2D

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main.tscn")
        Global.scene_changed.connect(_on_scene_changed)

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()

```

Now if you run your scene and you run into your trigger areas, your scenes should change and your player and other entities should spawn! Remember to place your Player node in an area where they can spawn in both scenes.

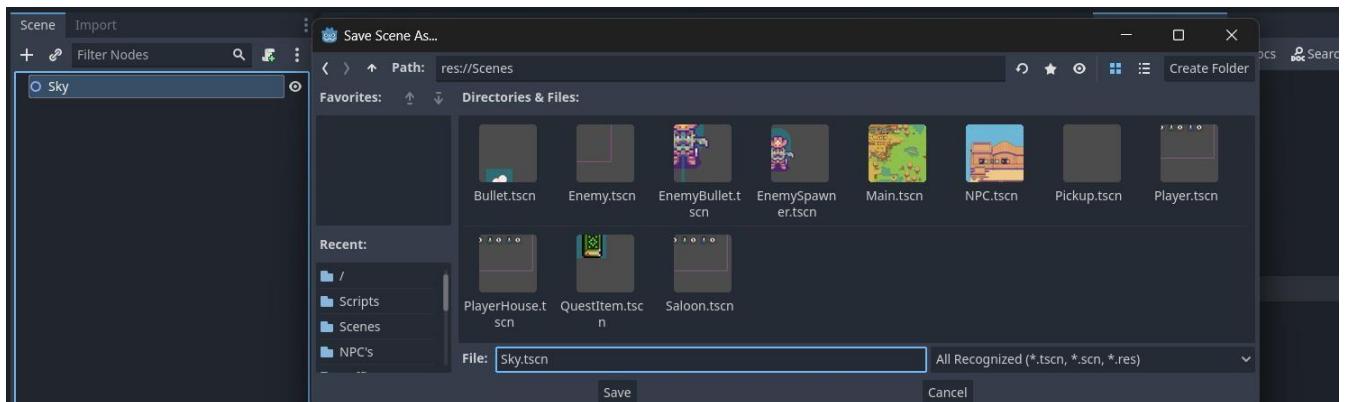


REALTIME DAY/NIGHT CYCLE

Now that our Player can travel to other areas, and they aren't homeless anymore, let's add some realism to these new areas by giving our game some "time". We'll do this via a real-time day/night cycle. This means that the color of our scenes will change to match the current time that the Player is playing. So, if you're playing the game in the afternoon, it will be afternoon time in the game, and so forth!

You can skip this if you don't want such a system in your game, but if you want to stick around and see how this works, let's get started.

In our project, let's create a new scene with a Node2D node as its root. This is the base node for our 2D games, and we're using it because it can contain and call any other node. Rename this node as "Sky" and save the scene under your Scenes folder.



For our Sky, we'll tint the sky according to the colors of our sky when we transition from day to night. I'll be using [this](#) color palette as my reference for this. We want to tint our entire screen. Now you can do this with a normal ColorRect node that you drag to be the size of your screen, or you can take the easier route and add a [CanvasModulate](#) node. This node tints the entire canvas element using its assigned [color](#).

Pastel Day to Night Color Palette



1kindjune

★ 6 Favorites □ 0 Comments

Login to add palette to your favorites.

For a simple background transition colors.

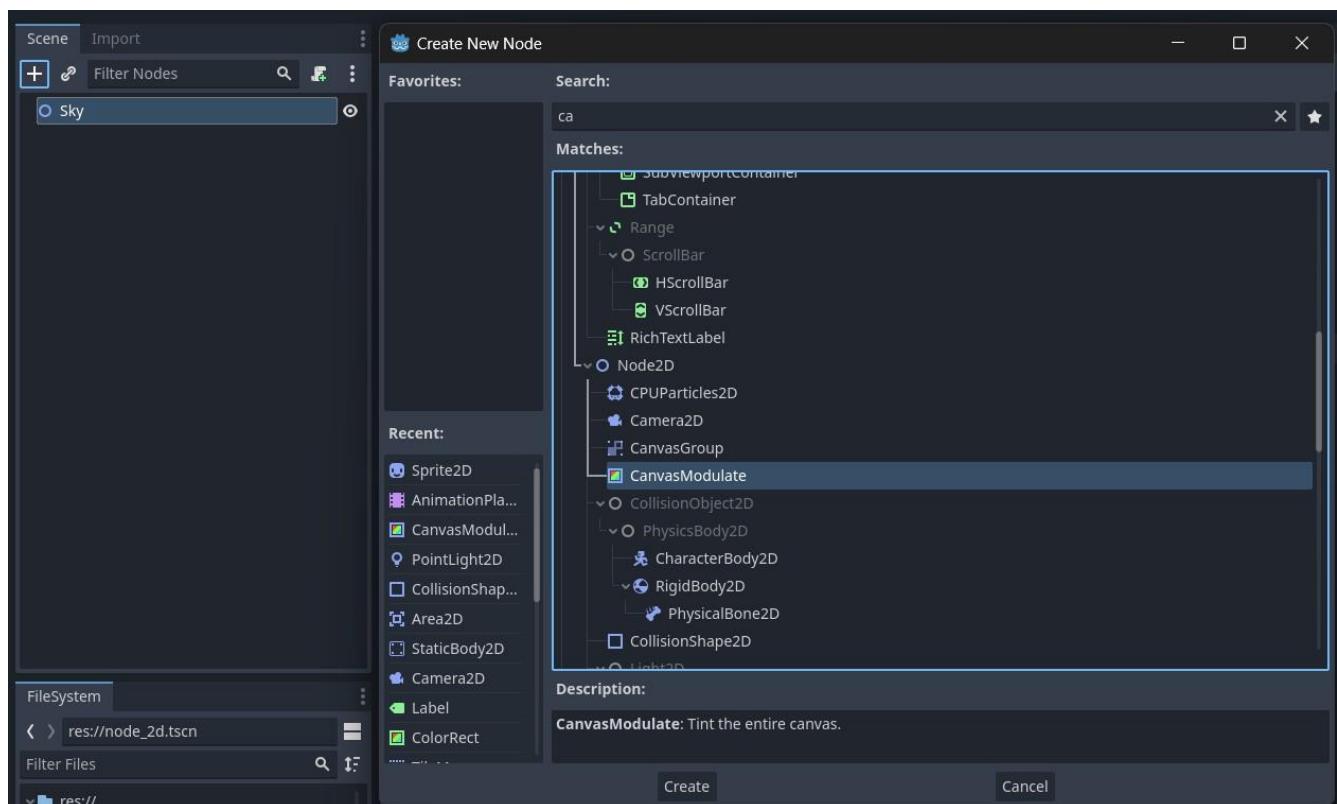
Colors in Palette

Color	Hex	RGB
	#6696ba	(102,150,186)
	#e2e38b	(226,227,139)
	#e7a553	(231,165,83)
	#7e4b68	(126,75,104)
	#292965	(41,41,101)

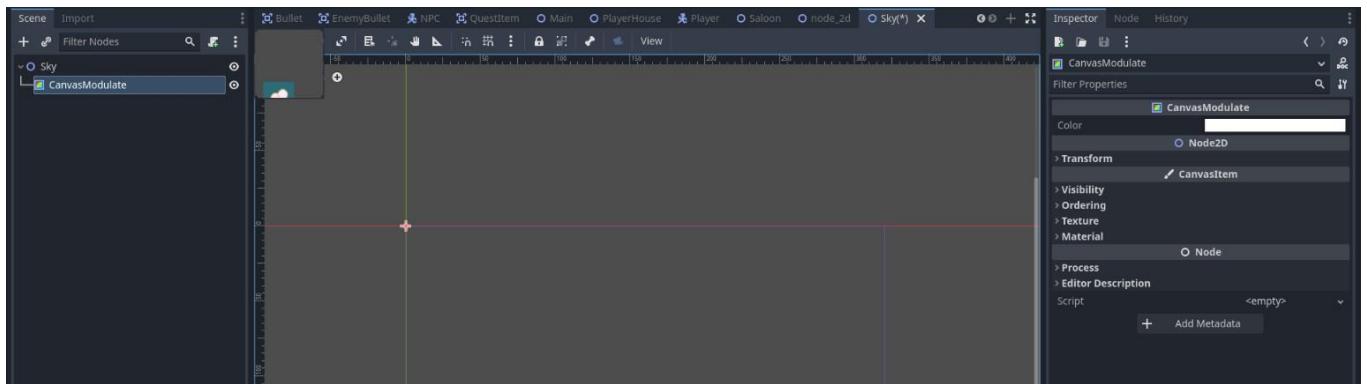
Facebook

Twitter

Add a CanvasModulate node to your Sky scene.

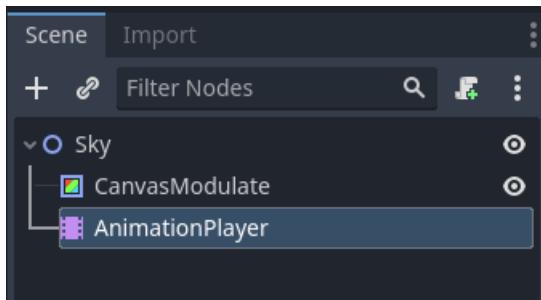


You'll see that it only has a Color property that you can change in the Inspector panel.

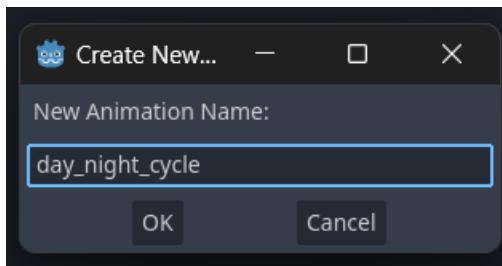


Now, we want this canvas to change color according to the time of the day. You might be thinking of using a Timer node with a few conditionals for this, but for a smooth transition between our colors, we need to use an AnimationPlayer node instead. We'll add a bunch of color tracks to the CanvasModule node in an animation, and according to the time of the day, the color would slowly transition on the animation timeline.

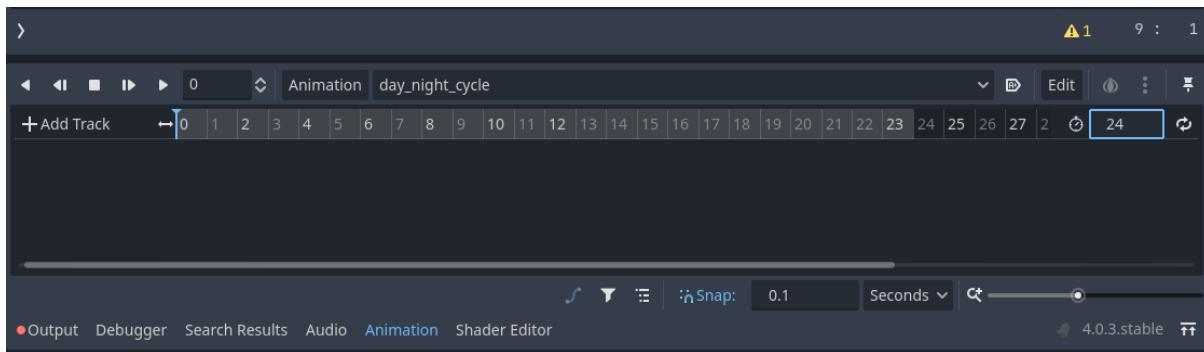
This might sound complex, but don't worry - you'll get it in a minute! Add an AnimationPlayer node to your Sky scene.



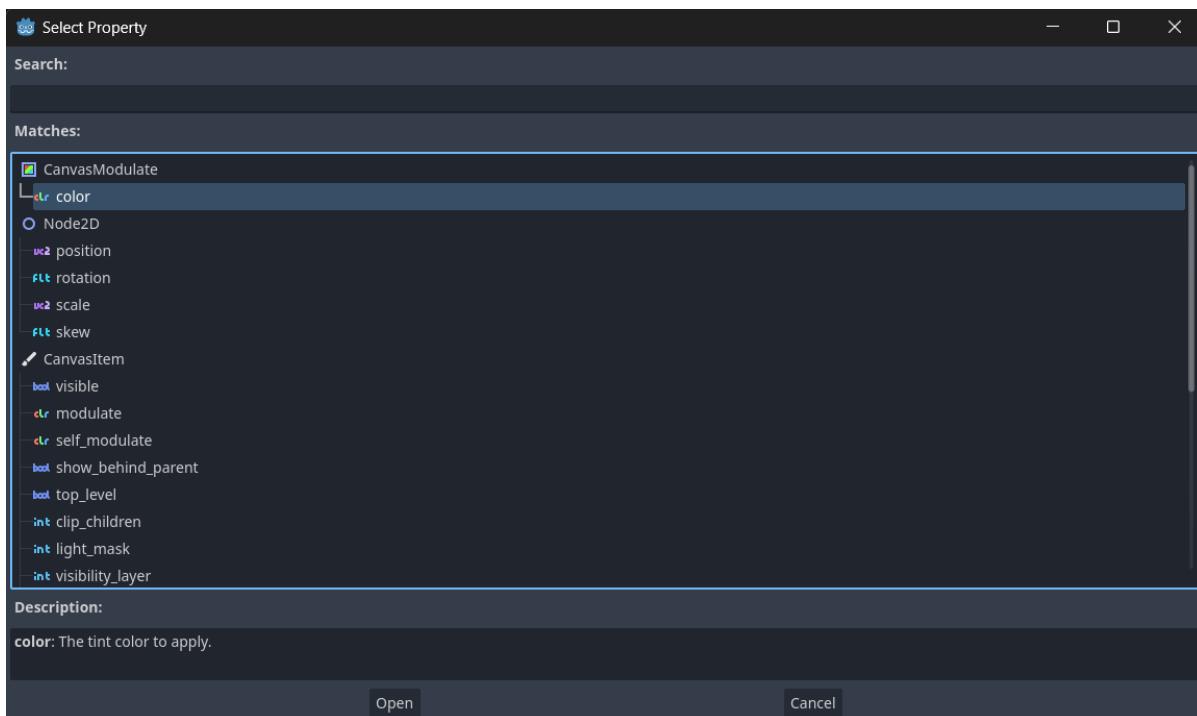
In your AnimationPlayer node, add a new animation called "day_night_cycle".



In real life, our day-to-night cycle lasts 24 hours, so set the animation length to also be 24. We'll add code later on that will seek the second on the timeline and assign a color to that in relation to our time in real life. So if it is 5 AM in real life, then the animation player will return the color that is set on the 5-second mark in our timeline.



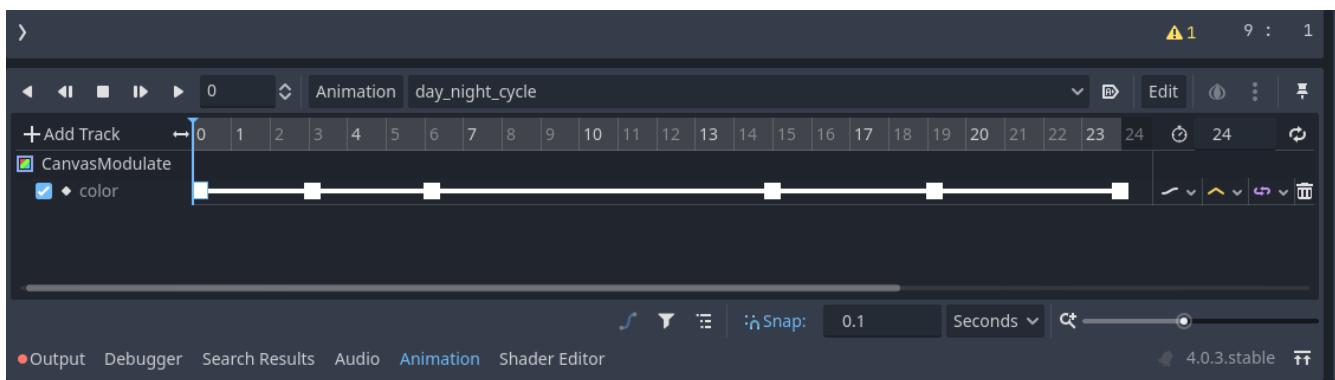
We want to change the color property of our CanvasModulate node, so let's add a Property Track that's connected to our canvas node and select "Color" as the property to change.



So for our `day_night_cycle` we have six colors that we need to assign to six timeslots:

- early morning -> between 12 AM - 3 AM
- morning -> between 3 AM - 5 AM
- day -> between 6 AM - 3 PM
- afternoon -> between 3 PM - 5 PM
- late afternoon -> between 5 PM - 7 PM
- night -> between 7 PM - 12 AM

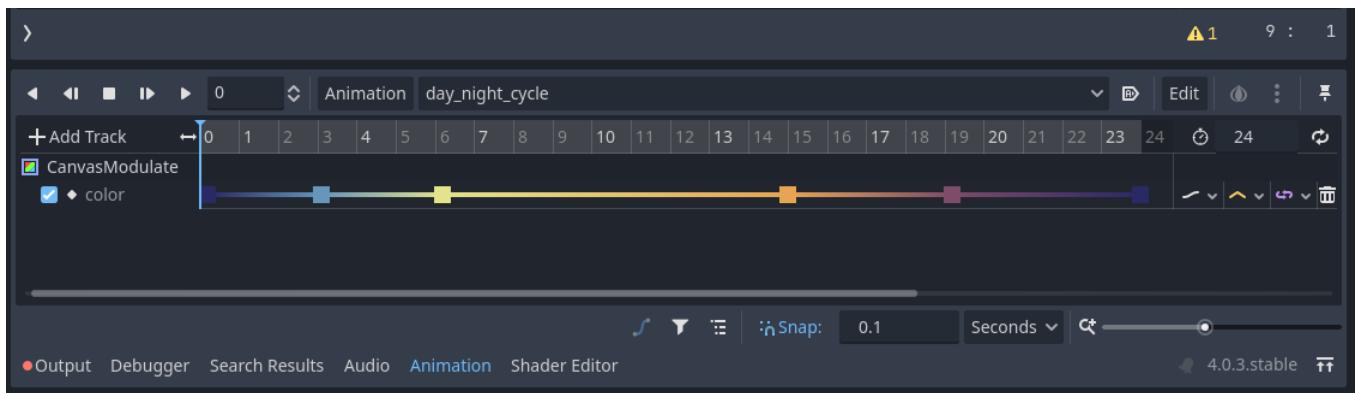
Change these times to suit the area that you live in, but in my country, that is the approximate time that the color of the sky would change. Let's assign these timeslots to our animation timeline. Assign six keys to where you want the time to change.



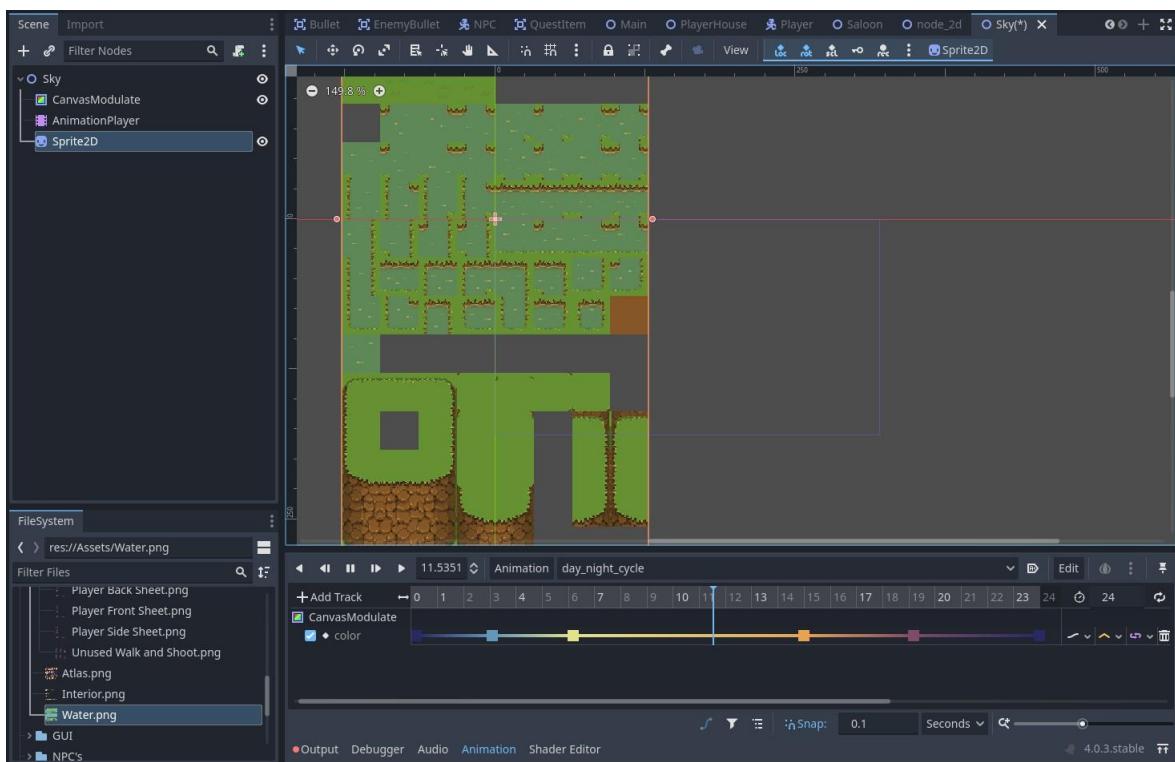
Now in the Inspector panel, assign the values from our color palette to each of these times. Remember to start with the night color and end with the night color.

- early morning -> between 12 AM - 3 AM -> [#292965](#)
- morning -> between 3 AM - 5 AM -> [#6696ba](#)
- day -> between 6 AM - 3 PM -> [#e2e38b](#)

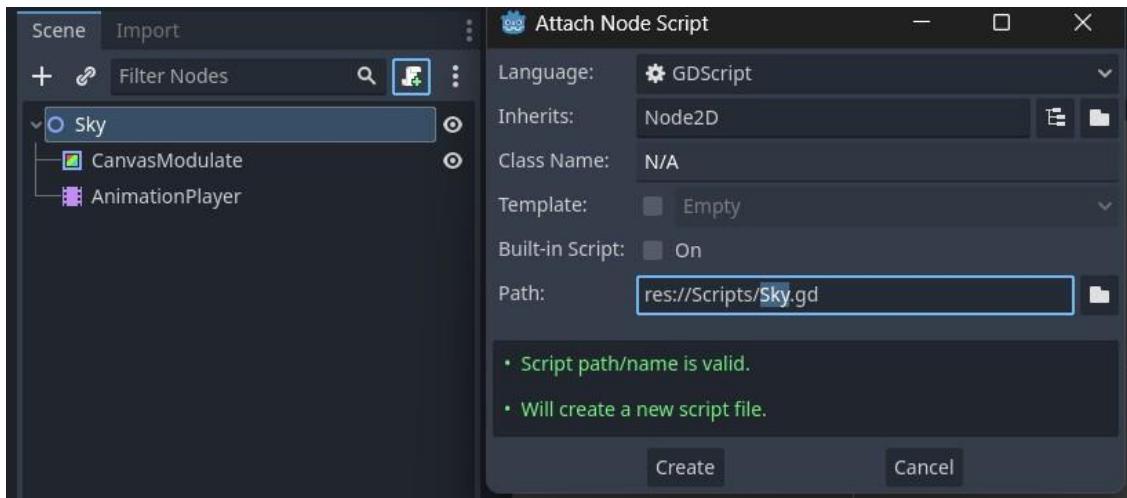
- afternoon -> between 3 PM - 5 PM -> [#e7a553](#)
- late afternoon-> between 5 PM - 7 PM -> [#7e4b68](#)
- night -> between 7 PM - 12 AM -> [#292965](#)



Now if you add a `Sprite2D` node into your scene and you run your animation, the color should change according to where the timeline's keyframe is currently at. Remember to delete this `Sprite2D` node afterward because this Sky scene will be instanced in our Main scene!



Let's add a script to our Scene. Save it under your Scripts folder.



We want our current time in hours, minutes, and seconds to be calculated constantly throughout the game so that we can assign the current time to the *day_night_cycle* animation's timeline. Let's define a few variables to store our current time, our current time in seconds, and then the value of our seconds mapped to our value on our animation timeline.

```
### Sky.gd

extends Node2D

#time variables
var current_time
var time_to_seconds
var seconds_to_timeline
```

This will make more sense in a minute. We simply want to do a calculation that gets our current time. Then with that current time, we convert it into seconds. And then with those seconds, we play the animation at that keyframe range on our animation timeline (so if we returned 9.864 seconds, the animation at 9.864 on the timeline will play). We want this animation to be calculated constantly, so we'll do this in our *_process()* function.

To get the current time, we will use our [Time](#) object. We used this before in our Player scene to get the reload time for our bullets.

```
### Sky.gd

extends Node2D

#time variables
var current_time
var time_to_seconds
var seconds_to_timeline

#calculate the time
func _process(delta):
    #gets the current time
    current_time = Time.get_time_dict_from_system()
```

Then we'll call the *current_time* value's hours, minutes, and seconds, and we'll convert the total *current_time* value into seconds.

```
### Sky.gd

extends Node2D

#time variables
var current_time
var time_to_seconds
var seconds_to_timeline

#calculate the time
func _process(delta):
    #gets the current time
    current_time = Time.get_time_dict_from_system()
    #converts the current time into seconds
    time_to_seconds = current_time.hour * 3600 + current_time.minute * 60 +
        current_time.second
```

With our *time_to_seconds* value, we need to use our [remap\(\)](#) method to calculate the linear interpolation within a range. In programming, lerp (linear interpolation) is a

common function used in various fields, such as game development, graphics, and animation. It's used to find a value that is a specific blend between two other values.

We will use our remap() method to convert our *time_to_seconds* into a value that can be used in our animation timeline. We'll do this by scaling it from a range of [0, 86400] to [0, 24]. 86400 is the number of seconds in a day, and 24 could be representing the total frames or units of an animation timeline representing 24 hours (remember we set it to 24 seconds).

```
### Sky.gd

extends Node2D

#time variables
var current_time
var time_to_seconds
var seconds_to_timeline

#calculate the time
func _process(delta):
    #gets the current time
    current_time = Time.get_time_dict_from_system()
    #converts the current time into seconds
    time_to_seconds = current_time.hour * 3600 + current_time.minute * 60 +
    current_time.second
    #converts the seconds into a remap value for our animation timeline
    seconds_to_timeline = remap(time_to_seconds, 0, 86400, 0, 24)
```

Now we can use our *seconds_to_timeline* value to play the animation on our timeline.

```
### Sky.gd

extends Node2D
# Node refs
@onready var animation_player = $AnimationPlayer

#time variables
var current_time
var time_to_seconds
var seconds_to_timeline

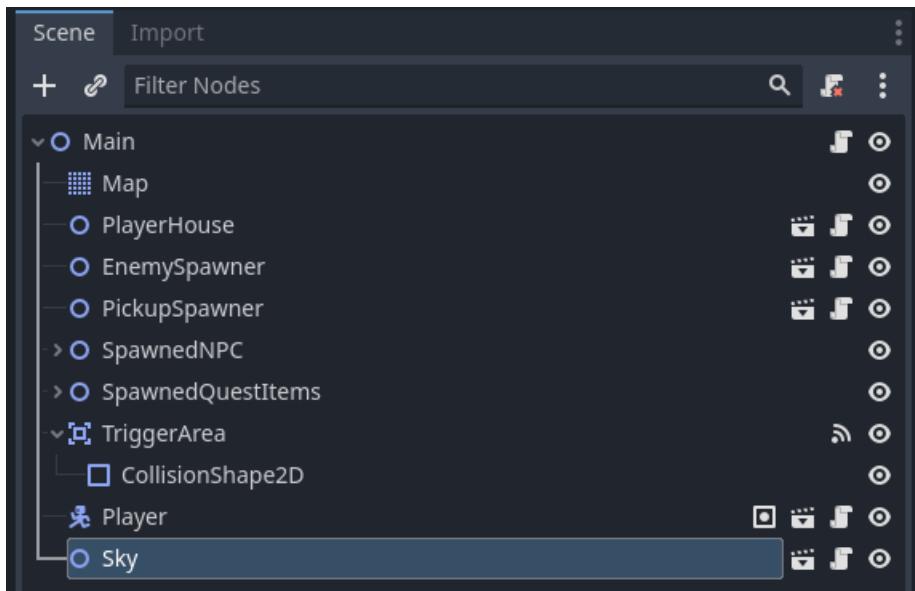
#calculate the time
```

```

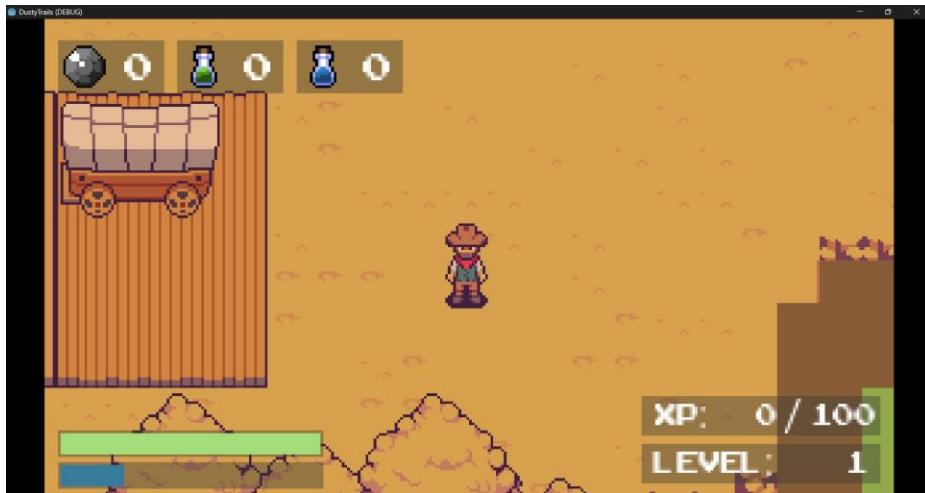
func _process(delta):
    #gets the current time
    current_time = Time.get_time_dict_from_system()
    #converts the current time into seconds
    time_to_seconds = current_time.hour * 3600 + current_time.minute * 60 +
    current_time.second
    #converts the seconds into a remap value for our animation timeline
    seconds_to_timeline = remap(time_to_seconds, 0, 86400, 0, 24)
    #plays the animation at that second value on the timeline
    animation_player.seek(seconds_to_timeline)
    animation_player.play("day_night_cycle")

```

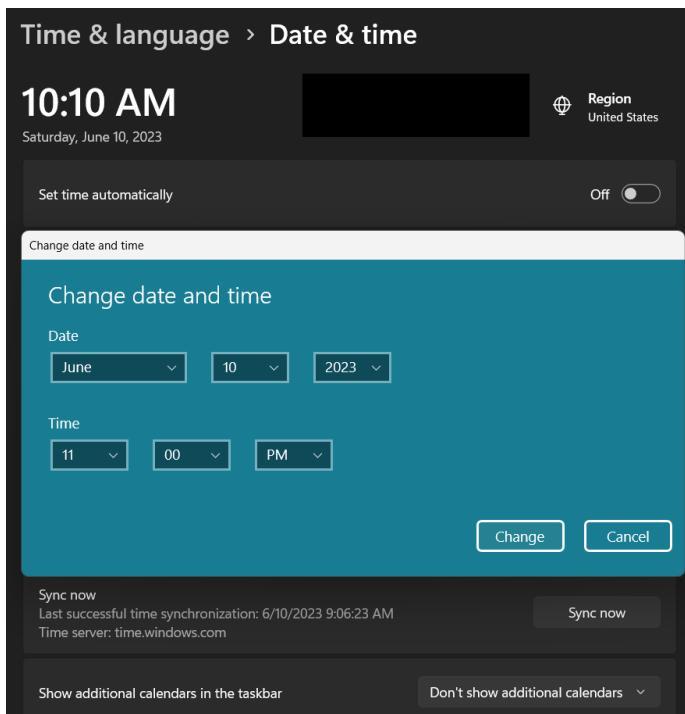
And so, our day-night cycle is complete. All we need to do is go back to our Main scene and instance our Sky scene!



By the time I was writing this, it was the afternoon in my area - so my color should be in my "afternoon" range when I now run my scene:

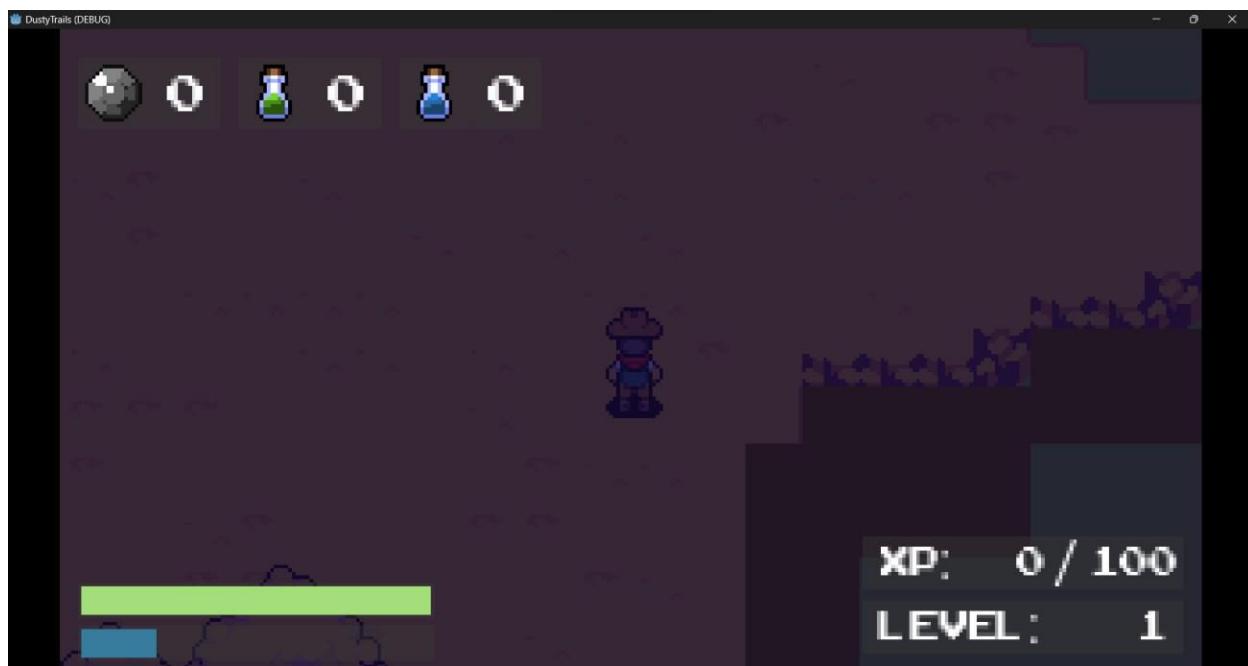


You can always tweak your colors if you want. Let's test the other times to see if it would work. Make sure your game scene is still running to test this. In your computer settings, under "Date and Time", disable the feature to "Set Time Automatically" and change the time to a different time, such as 11 PM.

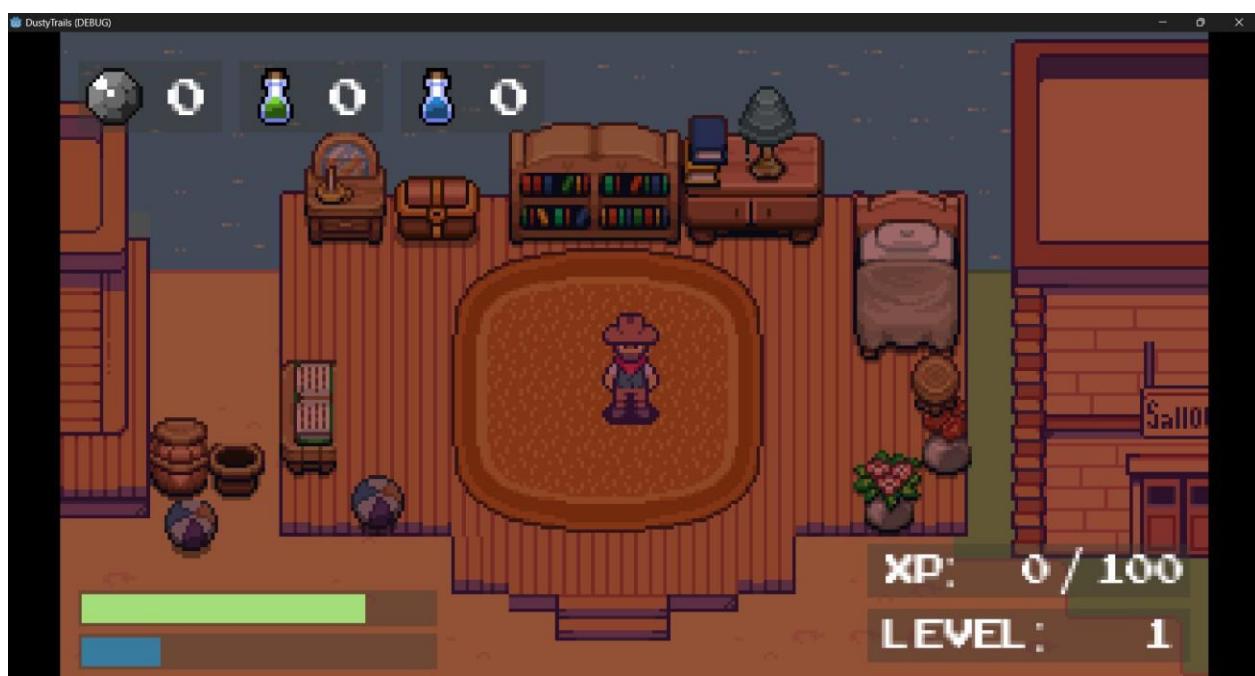


If you go back to the running instance of your game, your scene should now be the color of your system's new time!

11 PM:



6 PM:



2 AM:



These colors are a bit dark for my taste, so I recommend you tweak them. There you have it, two new ways to transition into new areas in your game plus a day-and-night cycle! We're so close to the end of this tutorial series, with the only thing left to do is to add a pause and main menu, saving and loading, and music & sound effects to our game! Remember to save your project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 19: PAUSE MENU & MAIN MENU

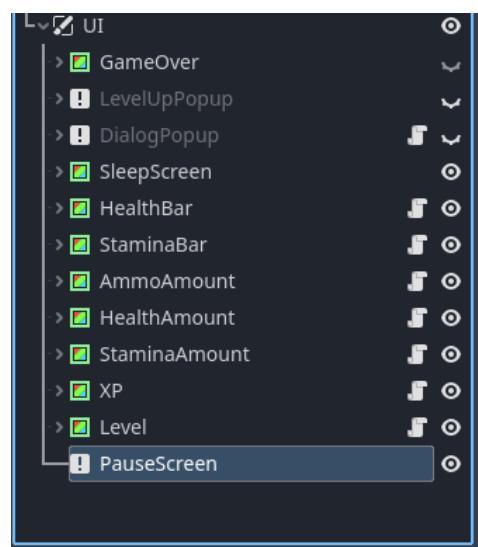
A pause and main menu are a default feature that comes with every game. We don't want our player to die each time the gamer has to go and get a snack, and we also don't want to spawn directly into the game! After this part, our player needs to be able to quit and start the game via the main menu, as well as pause the game. The pause screen will be upgraded in the next part to allow our players to save their game. The main screen will also be upgraded in the next part to allow our players to load their saved game.

WHAT YOU WILL LEARN IN THIS PART:

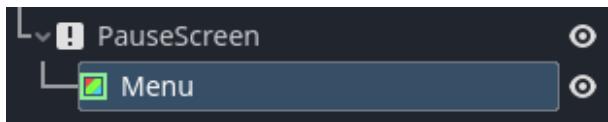
- How to pause and unpause the game state.
- How to quit out of the game project.

PAUSE MENU GUI SETUP

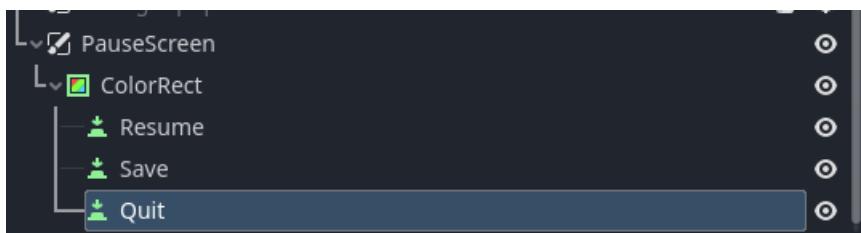
In your Player scene, add a new CanvasLayer node to your UI Layer and call it "PauseScreen".



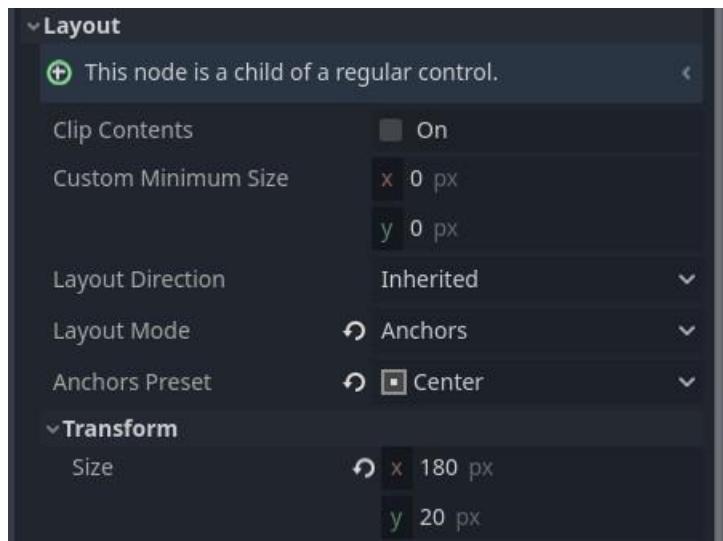
To this node, add a ColorRect node, and call it "Menu". Set the anchor-preset to "Full Rect" and change its color to #365655.



To the Menu node, we will add three buttons renamed to resume, save, and quit.



Now, set each of your button's sizes to be (x: 180, y: 20) and their anchor presets to be centered.



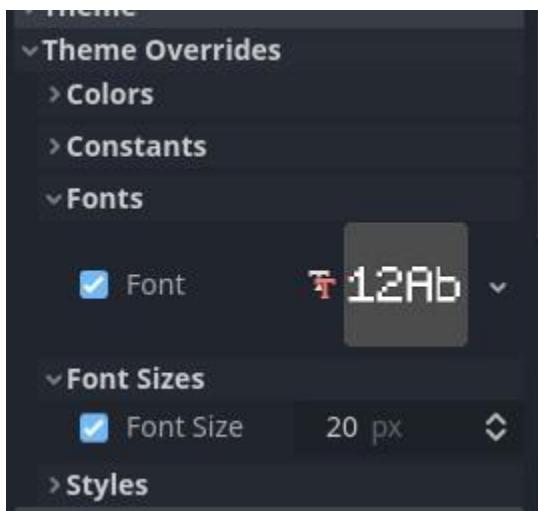
Then change the positions as follows:

The screenshot shows two panels of the Inspector panel. The left panel shows the properties for the 'Resume' node, and the right panel shows the properties for the 'Save' node. Both nodes have identical configurations except for their positions:

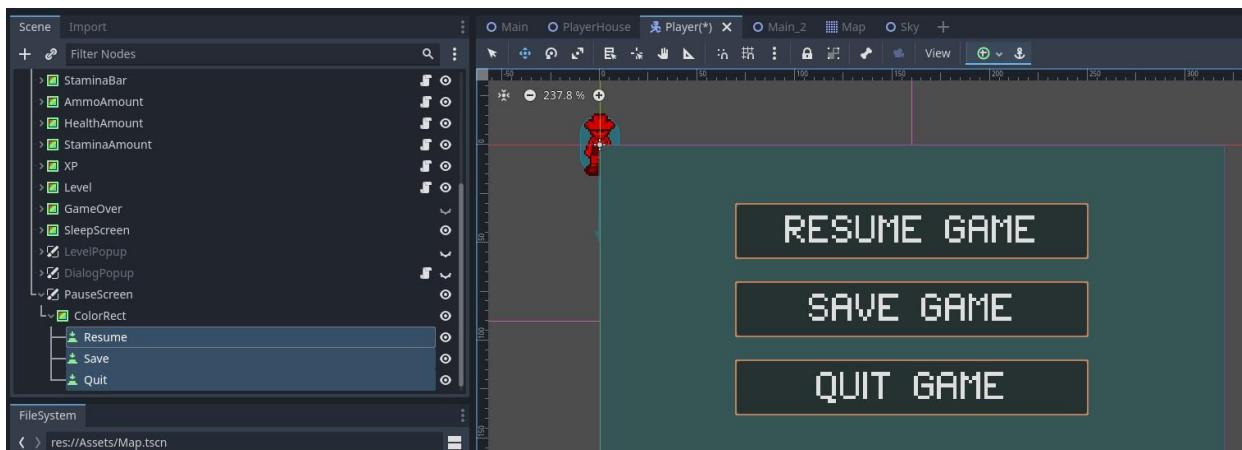
- Action Mode: Button Release
- Button Mask: Mouse Left (checked)
- Keep Pressed Outside: On
- Button Group: <empty>
- Shortcut**: Control
- Layout**:
 - Clip Contents: On
 - Custom Minimum Size: x 0 px, y 0 px
 - Layout Direction: Inherited
 - Layout Mode: Anchors
 - Anchors Preset: Center
- Transform**:
 - Size: x 180 px, y 20 px
 - Position: x 70 px, y 80 px
 - Rotation: 0°
 - Scale: x 1, y 1
 - Pivot Offset: x 0 px, y 0 px



Change each of their fonts to be "Schrödinger" size 20.



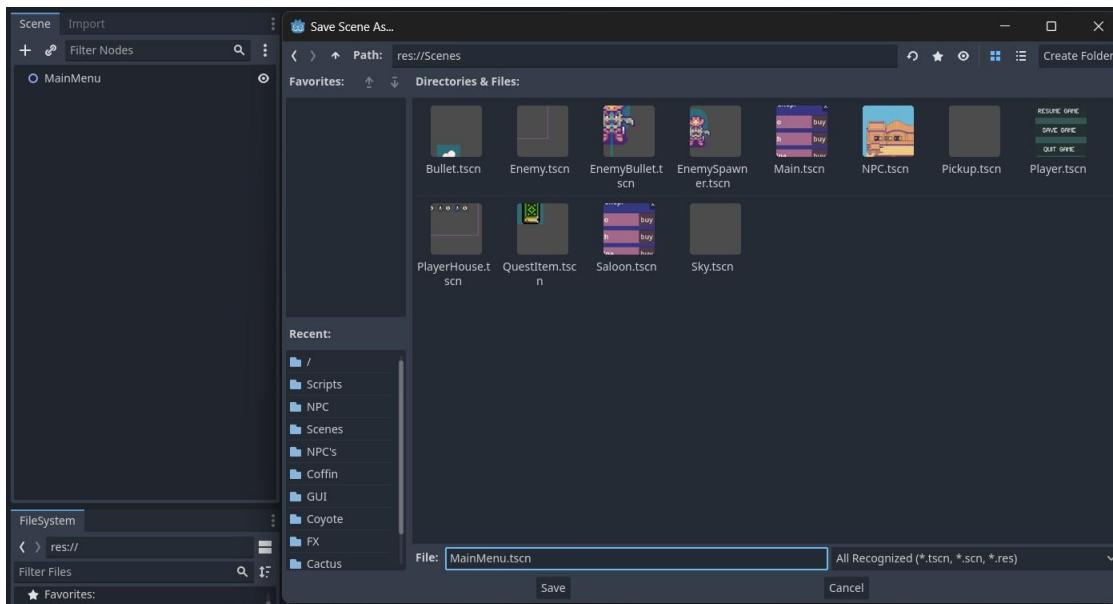
Then update label texts to be like the image below:



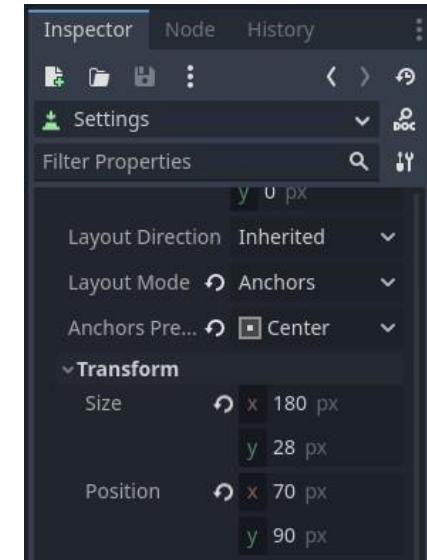
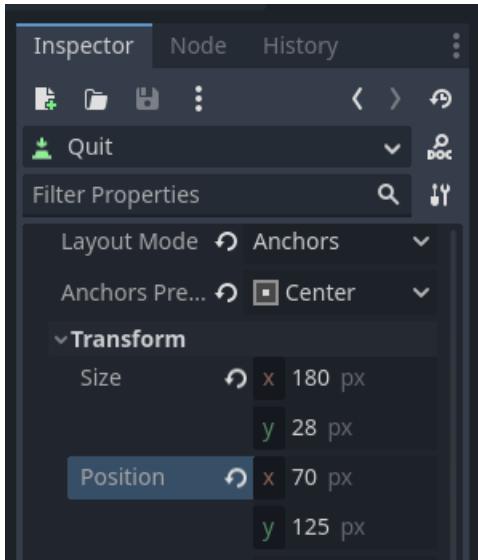
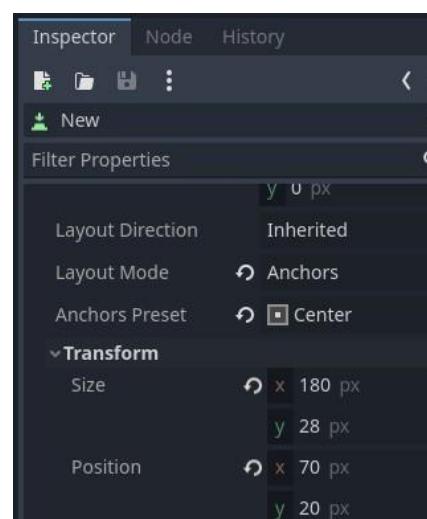
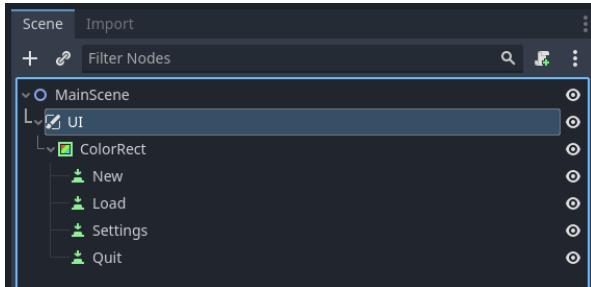
MAIN MENU GUI SETUP

Now that our pause menu is created, we can go ahead and create our Main Menu. Our Main Menu will be in its own scene because it is this scene that we will load up when we run the game. From here on, we can choose to start a new game, load a saved game, change some settings, or quit the game.

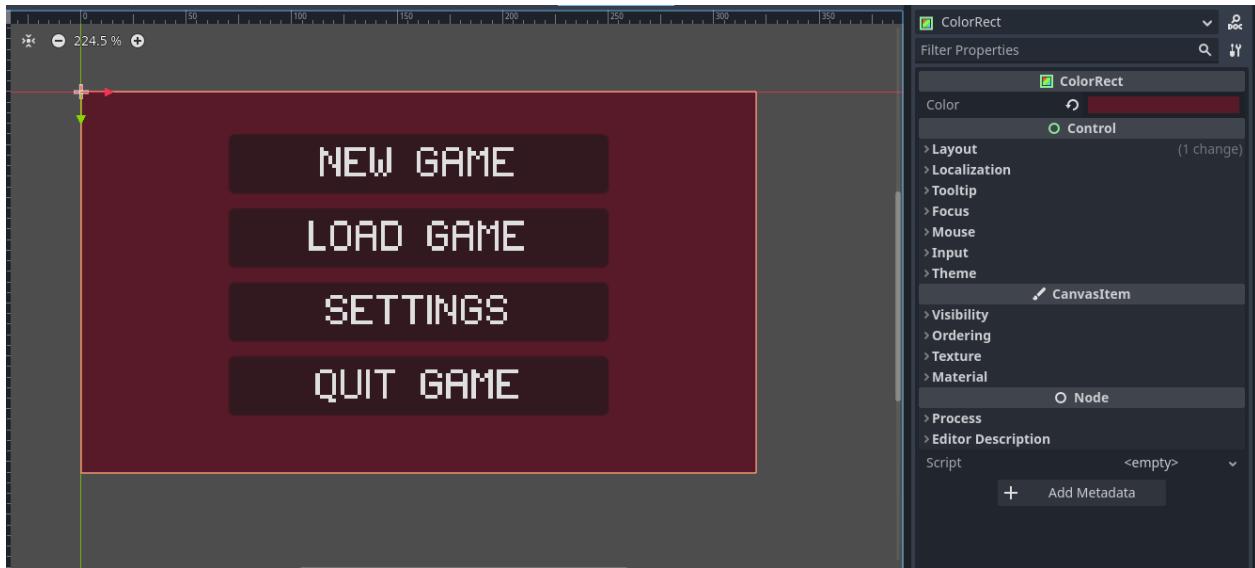
Create a new scene with a Node2D node as its root. We used the same node for our Main scene. Rename it to MainScene and save this scene underneath your Scenes folder.



The rest of this scene is similar to our PauseScreen. Let's take the same steps as we did in our Pause Screen to create this menu screen (the properties for the button positions can be found below):

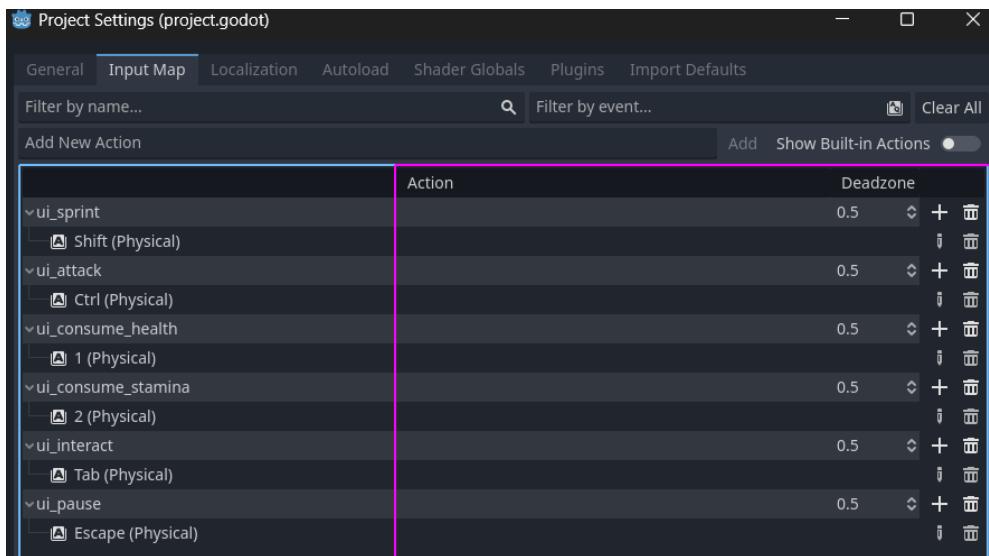


Change your ColorRect's color to #581929 so that we can differentiate it from our Pause Menu.



PAUSE MENU FUNCTIONALITY

For our pause menu, we want the game to pause if the player presses the ESC (escape) key on their keyboard. Let's start with adding the input for this. Call the new input *ui_pause*.



In our code, we want to capture the paused state of our game. If we press the ui_pause input, this paused state should be set to true, and the game should be paused. If the game is paused, our pause screen should be shown - so make sure you change its visibility to be hidden. Since we changed our player's processing mode to "always", we also need to disable the player's movement processing.

In our Player script, let's define a variable that will hold our game's pause state.

```
### Player.gd

# older code

#paused state
var paused
```

Then, in our input() function, let's set the game to pause and the PauseScreen to be shown only if the PauseScreen node is not already visible.

```
### Player.gd

# UI nodes
@onready var pause_screen = $UI/PauseScreen

func _input(event):
    #input event for our attacking, i.e. our shooting
    if event.is_action_pressed("ui_attack"):
        #checks the current time as the amount of time passed
        var now = Time.get_ticks_msec()
        #check if player can shoot if the reload time has passed and we have ammo
        if now >= bullet_fired_time and ammo_pickup > 0:
            #shooting anim
            is_attacking = true
            var animation  = "attack_" + returned_direction(new_direction)
            animation_sprite.play(animation)
            #bullet fired time to current time
            bullet_fired_time = now + bullet_reload_time
            #reduce and signal ammo change
            ammo_pickup = ammo_pickup - 1
            ammo_pickups_updated.emit(ammo_pickup)
```

```

#using health consumables
elif event.is_action_pressed("ui_consume_health"):
    if health > 0 && health_pickup > 0:
        health_pickup = health_pickup - 1
        health = min(health + 50, max_health)
        health_updated.emit(health, max_health)
        health_pickups_updated.emit(health_pickup)

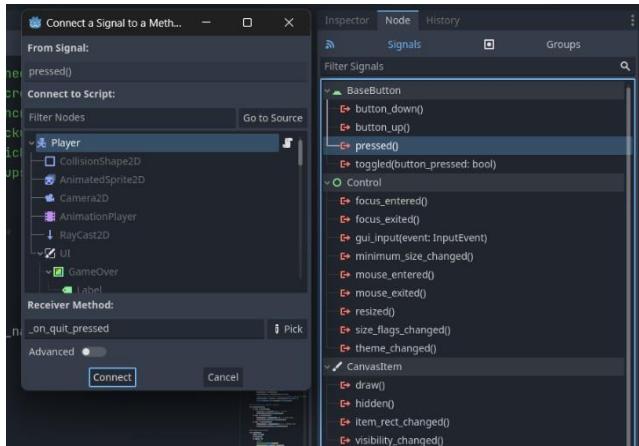
#using stamina consumables
elif event.is_action_pressed("ui_consume_stamina"):
    if stamina > 0 && stamina_pickup > 0:
        stamina_pickup = stamina_pickup - 1
        stamina = min(stamina + 50, max_stamina)
        stamina_updated.emit(stamina, max_stamina)
        stamina_pickups_updated.emit(stamina_pickup)

#interact with world
elif event.is_action_pressed("ui_interact"):
    var target = ray_cast.get.collider()
    if target != null:
        if target.is_in_group("NPC"):
            # Talk to NPC
            target.dialog()
            return
        #go to sleep
        if target.name == "Bed":
            # play sleep screen
            animation_player.play("sleeping")
            health = max_health
            stamina = max_stamina
            health_updated.emit(health, max_health)
            stamina_updated.emit(stamina, max_stamina)
            return

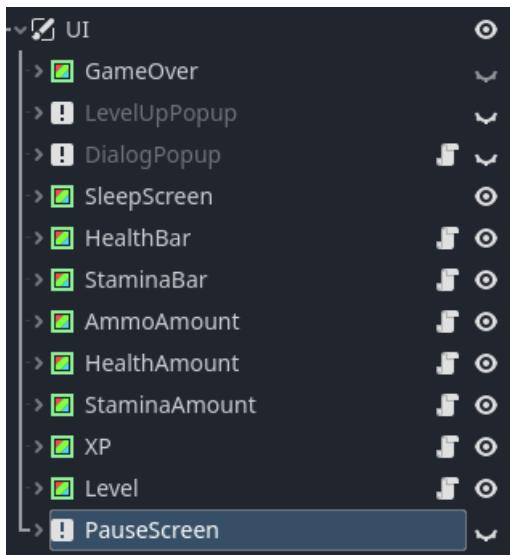
    #show pause menu
    if !pause_screen.visible:
        if event.is_action_pressed("ui_pause"):
            #pause game
            get_tree().paused = true
            #show pause screen popup
            pause_screen.visible = true
            #stops movement processing
            set_physics_process(false)
            #set pauses state to be true
            paused = true

```

Now, let's connect each of our buttons - resume, save, quit - *pressed()* signals to our script.



If the game's paused state is not true, then we need to unpause our game in our resume button's `on_pressed()` function. We also need to allow our player to process its movement again, and we need to hide our `PauseScreen` node.



```
### Player.gd

# ----- Pause Menu -----
#resume game
func _on_resume_pressed():
    #hide pause menu
    pause_screen.visible = false
    #set pauses state to be false
    get_tree().paused = false
    paused = false
```

```
#accept movement and input  
set_process_input(true)  
set_physics_process(true)
```

Then in our quit button's *on_pressed()* function, we'll redirect the player back to the main screen in the MainScreen scene.

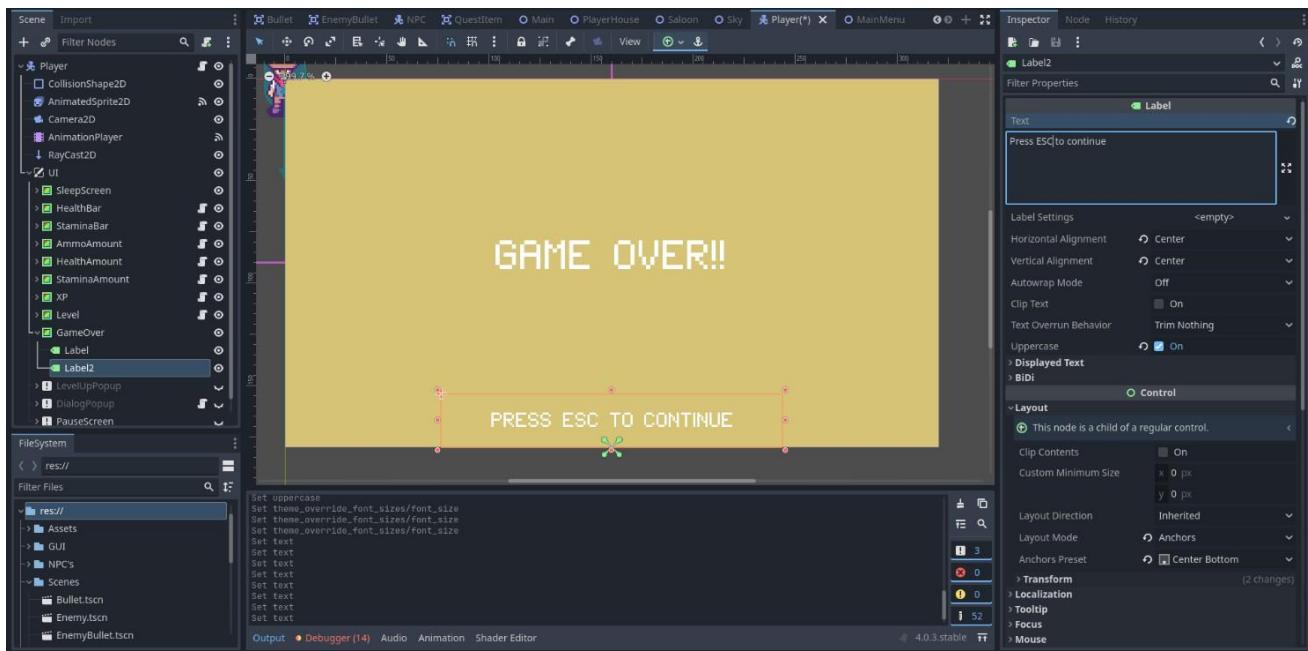
```
### Player.gd  
  
# ----- Pause Menu -----  
func _on_quit_pressed():  
    Global.change_scene("res://Scenes/MainScene.tscn")  
    get_tree().paused = false
```

We also want to make the cursor show again.

```
### Player.gd  
  
# ----- Pause Menu -----  
func _on_quit_pressed():  
    Global.change_scene("res://Scenes/MainScene.tscn")  
    get_tree().paused = false  
    Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
```

We will come back to our save button's *on_pressed()* function in the next part when we add saving and loading functions to our game. Whilst we're at it, let's take the time to also fix our Game Over screen to redirect our player to the MainMenu scene when they die and press ESC.

Let's add a new Label node to our GameOver node. This Label will tell our player to press ESC to go back to the Main Menu. Change the font to "Schrodinger" with the size set to 10. Also set its anchor-preset to be "Center Bottom", and its horizontal and vertical alignment to be centered as well.



In our `ui_pause` input, let's update our code to redirect the player to the `MainMenu` scene if the player's health is 0.

```
### Player.gd

func _input(event):
    # older code
    #show pause menu
    if !pause_screen.visible:
        if event.is_action_pressed("ui_pause"):
            #pause game      S
            get_tree().paused = true
            #show pause screen popup
            pause_screen.visible = true
            #stops movement processing
            set_physics_process(false)
            #set pauses state to be true
            paused = true

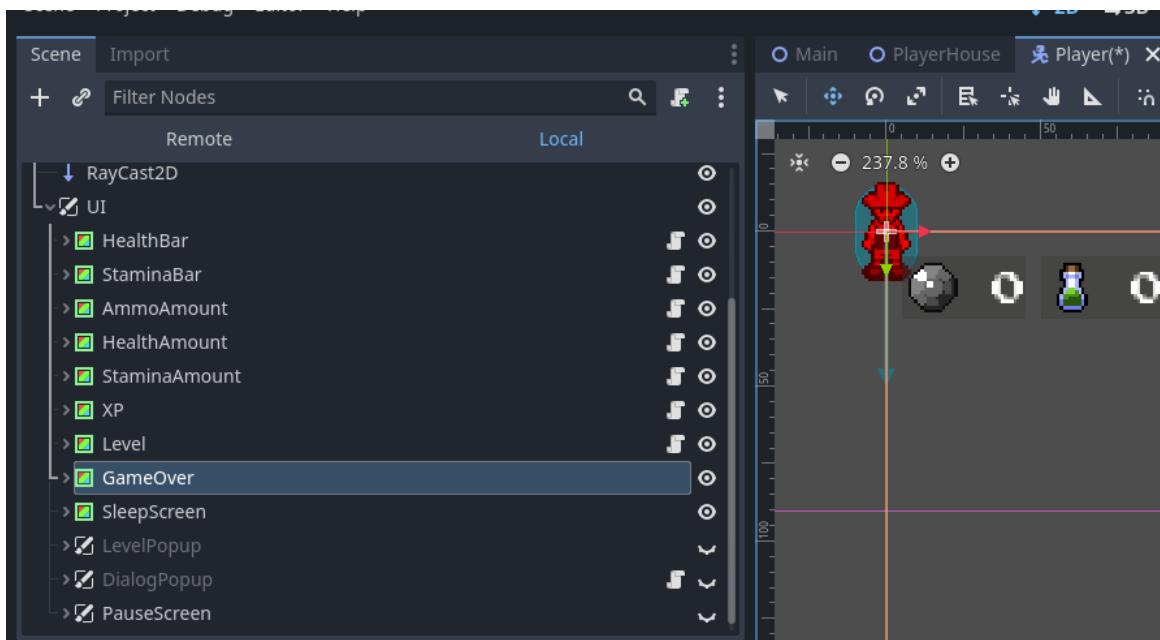
            # if the player is dead, go to back to main menu screen
            if health <= 0:
                get_node("/root/%s" % Global.current_scene_name).queue_free()
                Global.change_scene("res://Scenes/MainScene.tscn")
                get_tree().paused = false
                return
```

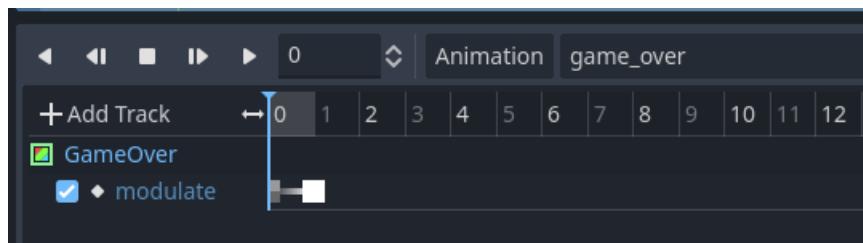
We also need to update our player's `hit()` function to pause the game and allow for input if they do die.

```
### Player.gd

# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
    health_updated.emit(health, max_health)
    if health > 0:
        #damage
        animation_player.play("damage")
        health_updated.emit(health, max_health)
    else:
        #death
        set_process(false)
        get_tree().paused = true
        paused = true
        animation_player.play("game_over")
```

Make sure your GameOver screen is set to visible. The animation for our game_over screen should be set on keyframe 0.





If you now run your scene, you should be able to pause/unpause your game - and if you die or quit, you should be redirected to your MainScene screen.



NEW GAME

LOAD GAME

SETTINGS

QUIT GAME

RESUME GAME

SAVE GAME

QUIT GAME

We also need to show our cursor whenever the pause screen is visible.

```
### Player.gd

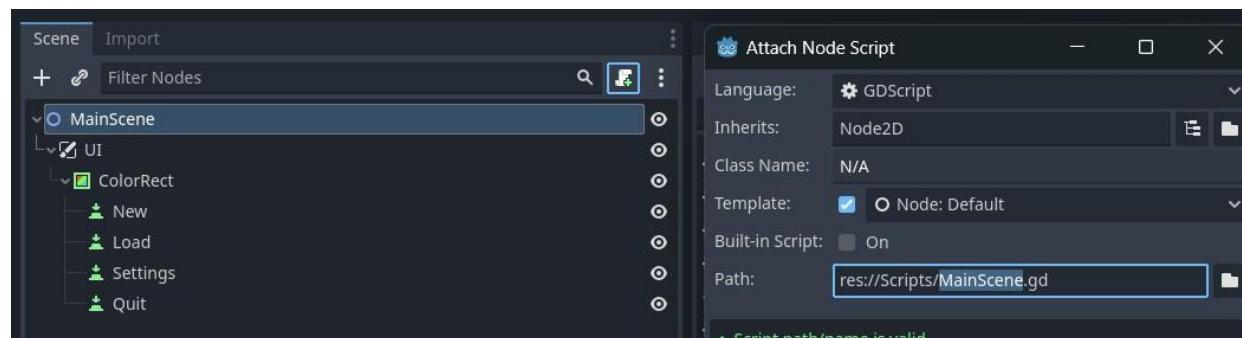
func _input(event):
    # older code

    #show pause menu
    if !pause_screen.visible:
        if event.is_action_pressed("ui_pause"):
            #pause game
            Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
            get_tree().paused = true
            #show pause screen popup
            pause_screen.visible = true
            #stops movement processing
            set_physics_process(false)
            #set pauses state to be true
            paused = true

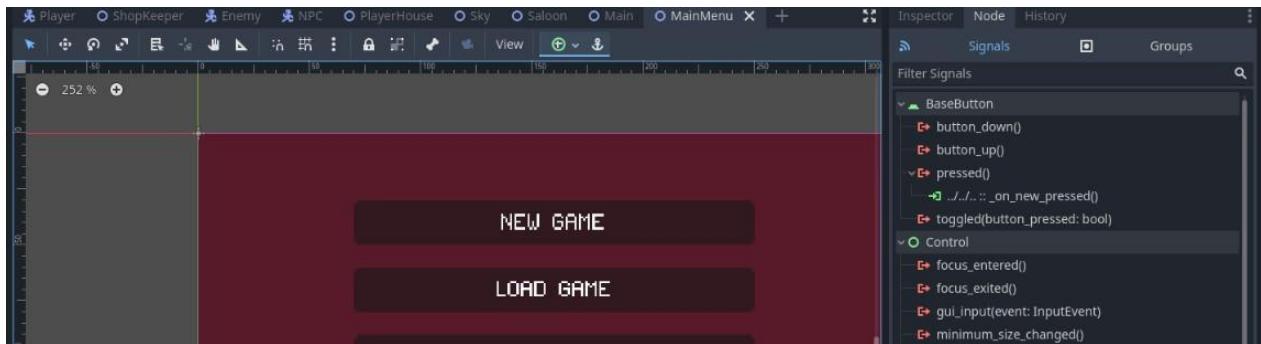
    # if the player is dead, go to back to main menu screen
    if health <= 0:
        get_node("/root/%s" % Global.current_scene_name).queue_free()
        Global.change_scene("res://Scenes/MainScene.tscn")
        get_tree().paused = false
        return
```

MAIN MENU FUNCTIONALITY

In your MainScene scene, attach a new script to its root node and save it under your Scripts folder.



Connect the `pressed()` signal from each of your buttons - new, load, and quit - to your new script. We'll not be adding settings to our game - so the settings button is just there for show!



Let's add our code to change the scene to our Main scene when we start a new game via the new button, as well as close the game when we press the quit button. We can quit our game via the `get_tree().quit()` method, which shuts down our entire game window. We'll add the functionality to load the game in the next part. We also need to show our cursor whenever this scene is active.

```
### MainScene.gd

extends Node2D

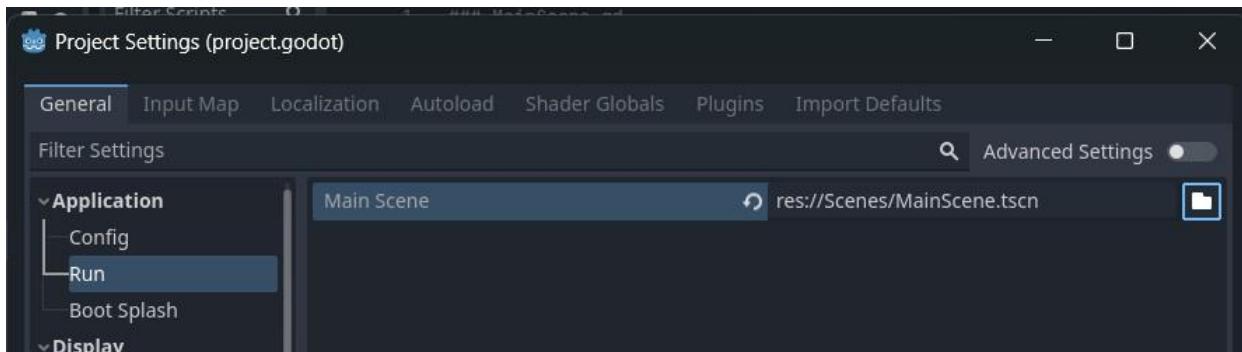
func _ready():
    Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)

# New game
func _on_new_pressed():
    Global.change_scene("res://Scenes/Main.tscn")
    Global.scene_changed.connect(_on_scene_changed)

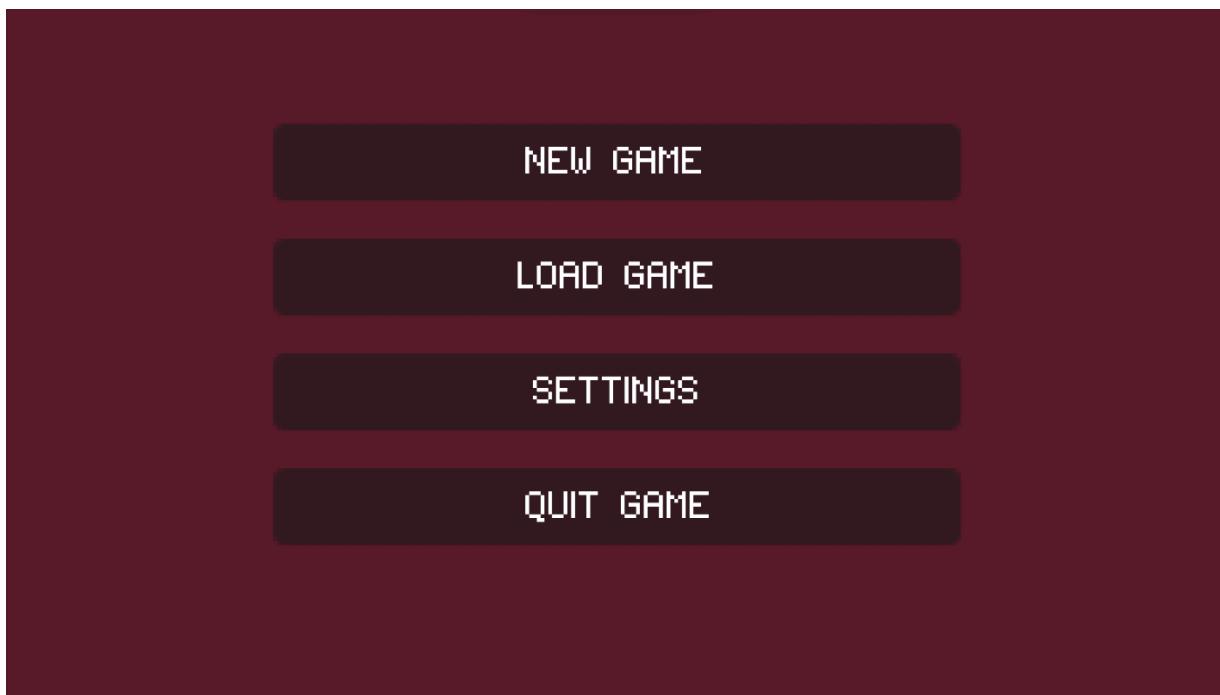
# Quit Game
func _on_quit_pressed():
    get_tree().quit()

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()
```

The last thing that we have to do for now is to change our run scene to our MainMenu scene. In your Project Settings > General > Application > Run, change your main scene from "Main" to "MainScene".



Now if you run your scene, you should open up on the MainMenu scene, and from here you can quit your game or start a new game.





Now our game has a Main Menu and a Pause Menu! Next up, we're going to be adding the functionality to save and load our game. Remember to save your project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 20: PERSISTENT SAVING & LOADING SYSTEM

In this part, we'll be adding the ability for our players to save and load their game. When we save the game, we need to store all the necessary variables from all of our different scripts to save the current state of those variables.

WHAT YOU WILL LEARN IN THIS PART:

- How to create persistent saving and loading systems
- How to parse JSON files.
- How to read and write files using the FileAccess object.
- How to save/load game variables.

We'll save these variables in a [dictionary](#), which will store our values as keys. The syntax of [Dictionaries](#) is similar to JSON, which is beneficial to us since we will be saving our save_file into a JSON format. JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays. This is useful for serializing data to save to a file or send over the network.

Our save file will follow the following format:

```
save_dictionary = {  
    "variable name": variable reference,  
    "player_health": health  
}
```

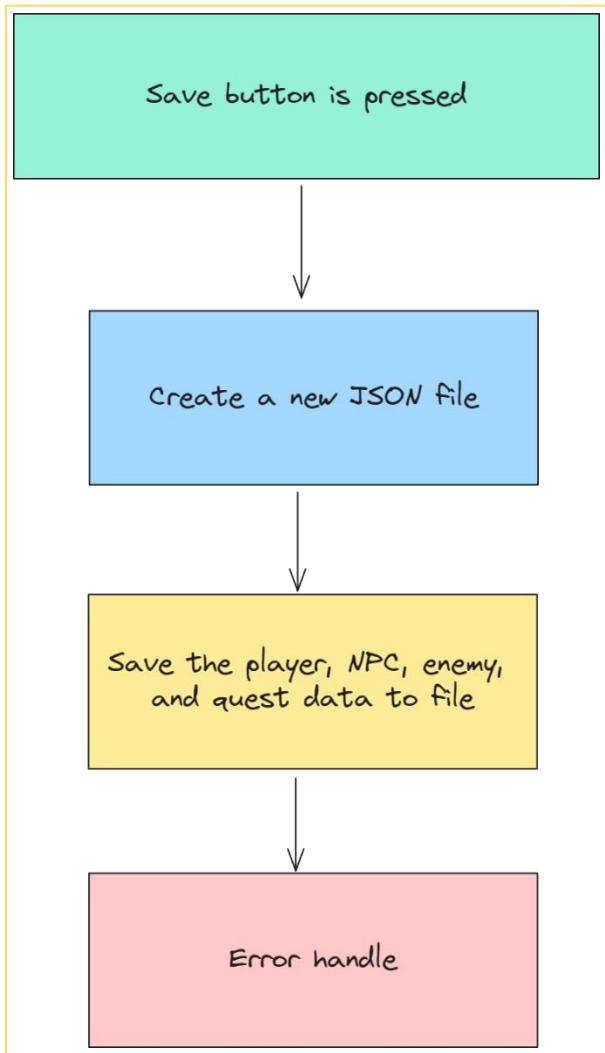


Figure 16: Save Game Overview

To load the game from our save file that we'll create, we'll have to convert the JSON file back into dictionary format. We'll do this via a function that will load the data from the JSON file.

Our load function will follow the following format:

```

func load_save_file(data):

    variable name = data.variable reference

    player_health = data.health

```

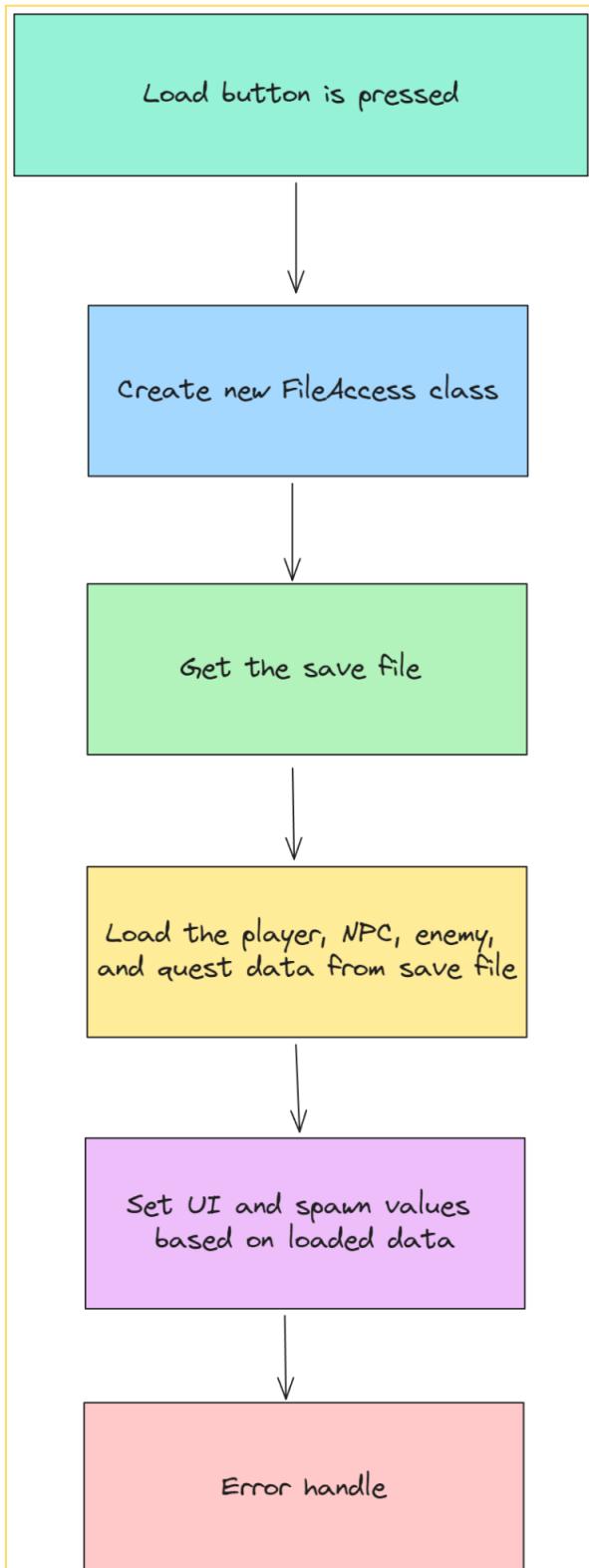


Figure 17: Load Game Overview

Let's get started with our Saving functionality!

SAVING THE GAME

In our Global script, let's create a new variable that will hold the save path of our JSON save file. We will set our path to be under "user://dusty_trails_save.json".

On a Windows machine, this save file will be found
under **%APPDATA%\Roaming\Godot\app_userdata\Dusty-Trails**.

```
### Global.gd

# older code

# Saving & Loading
var save_path = "user://dusty_trails_save.json"
```

To save our game, we need to go and create a dictionary in each script to store our values. We'll then store all these dictionaries in another dictionary in our Main scene which will compile all the separate dictionaries into a singular structure which will then be stored in our JSON file.

We want to save the following values from the following scripts:

- Player -> position, health, pickups, stamina, xp, and level
- Enemy -> position, health
- EnemySpawner -> spawned enemies
- NPC -> position, quest status, quest completion state

At the end of each of these scripts, let's create these dictionaries to store these values. We'll store the enemies in our EnemySpawner as an array so that when we load our enemies our spawner knows how many to continue counting from to add/remove enemy counts. We'll also append the enemy data from the Enemy scripts `data_to_save()`

function in the EnemySpawner's save function. Therefore, our spawner saves our enemy count plus each enemy's health and position values.

```
### Player.gd

#older code

# ----- Saving & Loading -----
#data to save
func data_to_save():
    return {
        "position" : [position.x, position.y],
        "health" : health,
        "max_health" : max_health,
        "stamina" : stamina,
        "max_stamina" : max_stamina,
        "xp" : xp,
        "xp_requirements" : xp_requirements,
        "level" : level,
        "ammo_pickup" : ammo_pickup,
        "health_pickup" : health_pickup,
        "stamina_pickup" : stamina_pickup
    }
```

```
### NPC.gd

#older code

# ----- Saving & Loading -----
#data to save
func data_to_save():
    return {
        "position" : [position.x, position.y],
        "quest_status": quest_status,
        "quest_complete": quest_complete
    }
```

```
### Enemy.gd

#older code

# ----- Saving & Loading -----
```

```
#data to save
func data_to_save():
    return {
        "position" : [position.x, position.y],
        "health" : health,
        "max_health" : max_health
    }
```

```
### EnemySpawner.gd

#older code

# ----- Saving & Loading -----
#data to save
func data_to_save():
    var enemies = []
    for enemy in spawned_enemies.get_children():
        #saves enemy amount, plus their stored health & position values
        if enemy.name.find("Enemy") >= 0:
            enemies.append(enemy.data_to_save())
    return enemies
```

Now in our Global script, we need to create a new function that will save our game. In this function, we'll need to create a dictionary of items to save, and that will include the dictionaries we added in our Player, EnemySpawner, and NPC scripts. We also only save the data if the nodes are present in the current scene. This means that it will only save NPC data if we have a npc in our scene. If we don't, it will skip the data for that NPC and save the other valid data fields. In summary, this function saves the game by retrieving the current scene, collecting data from specific nodes within the scene, converting the data to JSON format, and storing it in a file. The data saved includes the scene name, player data, NPC data, and enemy spawner data if they exist in the current scene.

To save a file to this path, we will have to use the [FileAccess](#) object. We'll first need to convert our "data" dictionary to a JSON string. We can do this by using the [stringify](#) method, which converts the data to a JSON-formatted string.

Then we will use our [.open](#) method from our FileAccess object to open our save_path file. Opening this file allows us to read or write to it. In this instance, we will use the [FileAccess.WRITE](#) method to specify that the file should be opened in write mode.

After it has been opened, we will write to this file using the [store_line](#) function that writes a string to the file and adds a new line character at the end.

After all that's been done, we need to [close](#) the file. This is important to ensure that all data is written and resources are released.

```
# ----- Saving & Loading -----
# save game
func save():
    var current_scene = get_tree().get_current_scene()
    if current_scene != null:
        current_scene_name = current_scene.name
        # data to save
        var data = {
            "scene_name" : current_scene_name,
        }
        #check if nodes exist before saving
        if current_scene.has_node("Player"):
            var player = get_tree().get_root().get_node("%s/Player" %
                current_scene_name)
            print("Player exists: ", player != null)
            data["player"] = player.data_to_save()
        if current_scene.has_node("SpawnedNPC/NPC"):
            var npc = get_tree().get_root().get_node("%s/SpawnedNPC/NPC" %
                current_scene_name)
            print("NPC exists: ", npc != null)
            data["npc"] = npc.data_to_save()
        if current_scene.has_node("EnemySpawner"):
            var enemy_spawner = get_tree().get_root().get_node("%s/EnemySpawner" %
                current_scene_name)
            print("EnemySpawner exists: ", enemy_spawner != null)
            data["enemies"] = enemy_spawner.data_to_save()
        # converts dictionary (data) into json
        var json = JSON.new()
        var to_json = json.stringify(data)
        # opens save file for writing
        var file = FileAccess.open(save_path, FileAccess.WRITE)
        # writes to save file
```

```

        file.store_line(to_json)
        # close the file
        file.close()
    else:
        print("No active scene. Cannot save.")

```

Now we will save our game when we change scenes. This will prevent our game from returning `<null>` values for our scene paths. It also helps us carry over the last captured player data from the previous scene into the new scene.

```

### Global.gd

# older code

# Change scene
func change_scene(scene_path):
    save()
    # Get the current scene
    current_scene_name = scene_path.get_file().get_basename()
    var current_scene =
        get_tree().get_root().get_child(get_tree().get_root().get_child_count() - 1)
    # Free it for the new scene
    current_scene.queue_free()
    # Change the scene
    var new_scene = load(scene_path).instantiate()
    get_tree().get_root().call_deferred("add_child", new_scene)
    get_tree().call_deferred("set_current_scene", new_scene)
    call_deferred("post_scene_change_initialization")

func post_scene_change_initialization():
    scene_changed.emit()

```

Now we need to go back to our Player script and call our save function from our Global script if we press our save button.

```

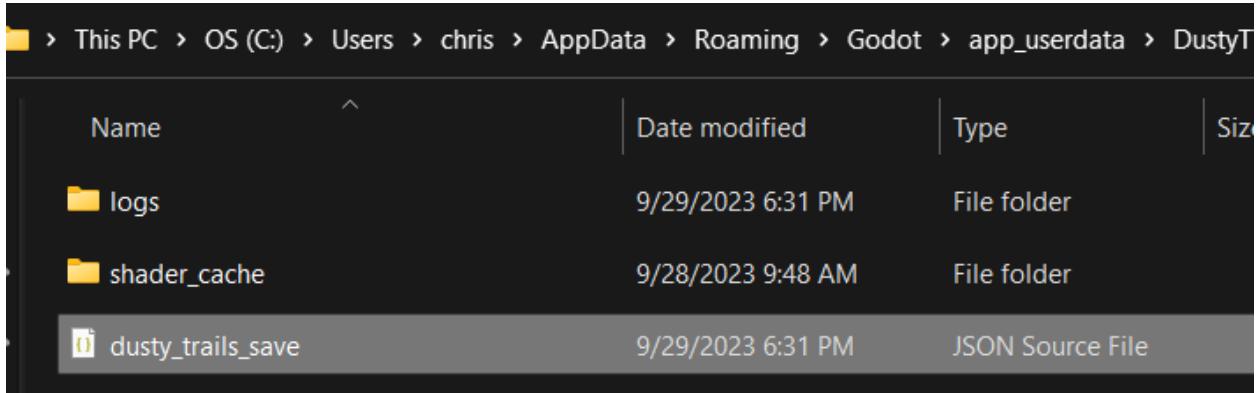
### Player.gd

# older code

# save game
func _on_save_pressed():
    Global.save()

```

If you run your scene now, and in your Pause menu you save your game, your data should be saved.



A screenshot of a Windows File Explorer window. The path is displayed at the top: This PC > OS (C:) > Users > chris > AppData > Roaming > Godot > app_userdata > DustyT. Below the path is a table with three columns: Name, Date modified, and Type. The table contains three items: 'logs' (File folder, 9/29/2023 6:31 PM), 'shader_cache' (File folder, 9/28/2023 9:48 AM), and 'dusty_trails_save' (JSON Source File, 9/29/2023 6:31 PM). The 'dusty_trails_save' file is highlighted with a gray background.

Name	Date modified	Type
logs	9/29/2023 6:31 PM	File folder
shader_cache	9/28/2023 9:48 AM	File folder
dusty_trails_save	9/29/2023 6:31 PM	JSON Source File

JSON SAVE FILE DATA:

```
{"enemies":[], "player": {"ammo_pickup":13, "health":100, "health_pickup":2, "level":1, "max_health":100, "max_stamina":100, "position": [439,154], "stamina":100, "stamina_pickup":2, "xp":0, "xp_requirements":100}, "scene_name": "Main"}
```

LOADING THE GAME

To load our game, we will have to go to each of our scripts once again and define the values that we want to load from them.

We want to load the following values from the following scripts:

- Player -> position, health, pickups, stamina, xp, and level
- Enemy -> position, health
- EnemySpawner -> spawned enemies
- NPC -> position, quest status, quest completion state

We'll do this by creating a function in each of our Player, NPC, Enemy, and EnemySpawner scripts. This function will pass the "data" parameter, which will

reference the saved values from our "data" dictionary that we created in our Main scene.

```
### Player.gd

#older code

#loads data from saved data
func data_to_load(data):
    position = Vector2(data.position[0], data.position[1])
    health = data.health
    max_health = data.max_health
    stamina = data.stamina
    max_stamina = data.max_stamina
    xp = data.xp
    xp_requirements = data.xp_requirements
    level = data.level
    ammo_pickup = data.ammo_pickup
    health_pickup = data.health_pickup
    stamina_pickup = data.stamina_pickup
```

```
### NPC.gd

#older code

#loads data from save
func data_to_load(data):
    position = Vector2(data.position[0], data.position[1])
    quest_status = int(data.quest_status)
    quest_complete = data.quest_complete
```

```
### Enemy.gd

#older code

#data to load from save file
func data_to_load(data):
    position = Vector2(data.position[0], data.position[1])
    health = data.health
    max_health = data.max_health
```

```

### EnemySpawner.gd

#older code

#load data from save file
func data_to_load(data):
    enemy_count = data.size()
    for enemy_data in data:
        var enemy = Global.enemy_scene.instantiate()
        enemy.data_to_load(enemy_data)
        add_child(enemy)

```

Now, we want to load our entire game data. In our load function, we will load a saved game state by reading our JSON-formatted save file, loading the corresponding scene, adding it to the scene tree, setting it as the current scene, and loading the saved data into specific nodes within the scene. We'll only load the data for our respective nodes (such as npc, player, and enemy) if the data is present in our save file.

We can do this via the [file_exists](#) method. If the file does exist, we need to open our file and read its content. We can do this by specifying that we want to read the file via the FileAccess.READ method. After we open it, we need to read the entire contents of the file as text using [file.get_as_text\(\)](#) and parse it as JSON using [JSON.parse_string\(\)](#).

We then need to close our file. After we've read the file, we need to load our data from our player, enemy spawner, and npc from the parsed JSON. If the quest state is set to complete, we also need to remove the quest item from the scene. You can also do the same for your pickups, but I want our pickups to respawn on load.

```

### Global.gd

#older code

# Saving & Loading
var save_path = "user://dusty_trails_save.json"
var loading = false

# older code

```

```

func load_game():
    if loading and FileAccess.file_exists(save_path):
        print("Save file found!")
        var file = FileAccess.open(save_path, FileAccess.READ)
        var data = JSON.parse_string(file.get_as_text())
        file.close()
        # Load the saved scene
        var scene_path = "res://Scenes/%s.tscn" % data["scene_name"]
        print(scene_path)
        var game_resource = load(scene_path)
        var game = game_resource.instantiate()
        # Change to the loaded scene
        get_tree().root.call_deferred("add_child", game)
        get_tree().call_deferred("set_current_scene", game)
        current_scene_name = game.name
        # Now you can load data into the nodes
        var player = game.get_node("Player")
        var npc = game.get_node("SpawnedNPC/NPC")
        var enemy_spawner = game.get_node("EnemySpawner")
        #checks if they are valid before loading their data
        if player:
            player.data_to_load(data["player"])
        if npc:
            npc.data_to_load(data["npc"])
        if enemy_spawner:
            enemy_spawner.data_to_load(data["enemies"])
        if(npc and npc.quest_complete):
            game.get_node("SpawnedQuestItems/QuestItem").queue_free()
    else:
        print("Save file not found!")

```

We also need to create a function that will load our player's data when they enter a new scene. This function will load our player data (such as their health, ammo, and coin count from the previous scene) when entering a new scene by checking if a save file exists, reading and parsing the file to obtain the player data, and loading the data into the "Player" node of the current scene if it exists. This will allow our UI components to show the correct values on game load.

```

## Global.gd

#player data to load when changing scenes
func load_data():

```

```

var current_scene = get_tree().get_current_scene()
if current_scene and FileAccess.file_exists(save_path):
    print("Save file found!")
    var file = FileAccess.open(save_path, FileAccess.READ)
    var data = JSON.parse_string(file.get_as_text())
    file.close()

    # Now you can load data into the nodes
    var player = current_scene.get_node("Player")
    if player and data.has("player"):
        player.values_to_load(data["player"])
    else:
        print("Save file not found!")

```

When this function loads our player data, it looks for a function inside of our Player script called *values_to_load()*. This is a new function that loads all of our Player's data except its position since we don't want our position to load in some random area on the map (which could be an area that is out of bounds)! In your Player script, let's create this function.

```

## Player.gd

# older code

func values_to_load(data):
    health = data.health
    max_health = data.max_health
    stamina = data.stamina
    max_stamina = data.max_stamina
    xp = data.xp
    xp_requirements = data.xp_requirements
    level = data.level
    ammo_pickup = data.ammo_pickup
    health_pickup = data.health_pickup
    stamina_pickup = data.stamina_pickup
    coins = data.coins

    # Emit signals to update UI
    health_updated.emit(health, max_health)
    stamina_updated.emit(stamina, max_stamina)
    ammo_pickups_updated.emit(ammo_pickup)
    health_pickups_updated.emit(health_pickup)

```

```

stamina_pickups_updated.emit(stamina_pickup)
xp_updated.emit(xp)
level_updated.emit(level)
coins_updated.emit(coins)

# Update UI components directly
$UI/AmmoAmount/Value.text = str(data.ammo_pickup)
$UI/StaminaAmount/Value.text = str(data.stamina_pickup)
$UI/HealthAmount/Value.text = str(data.health_pickup)
$UI/XP/Value.text = str(data.xp)
$UI/XP/Value2.text = "/ " + str(data.xp_requirements)
$UI/Level/Value.text = str(data.level)
$UI/CoinAmount/Value.text = str(data.coins)

```

We now also need to update our MainMenu script's code to call our newly created functions. Our load button will call our *load_game()* function from our Global script.

```

### MainScene.gd

extends Node2D

func _ready():
    Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)

# New game
func _on_new_pressed():
    Global.change_scene("res://Scenes/Main.tscn")
    Global.scene_changed.connect(_on_scene_changed)

# Load game
func _on_load_pressed():
    Global.loading = true
    Global.load_game()
    queue_free()

# Quit Game
func _on_quit_pressed():
    get_tree().quit()

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()

```

Finally, we need to also load our data when we change the scene to keep our data persistent.

```
### Global.gd

# older code

# ----- Scene handling -----
#set current scene on load
func _ready():
    current_scene_name = get_tree().get_current_scene().name

# Change scene
func change_scene(scene_path):
    save()
    # Get the current scene
    current_scene_name = scene_path.get_file().get_basename()
    var current_scene =
        get_tree().get_root().get_child(get_tree().get_root().get_child_count() - 1)
    # Free it for the new scene
    current_scene.queue_free()
    # Change the scene
    var new_scene = load(scene_path).instantiate()
    get_tree().get_root().call_deferred("add_child", new_scene)
    get_tree().call_deferred("set_current_scene", new_scene)
    call_deferred("post_scene_change_initialization")

func post_scene_change_initialization():
    load_data()
    scene_changed.emit()
```

We also need to set our player's UI values in their ready function to update the values when they enter a new scene.

```
### Player.gd

# older code

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
```

```

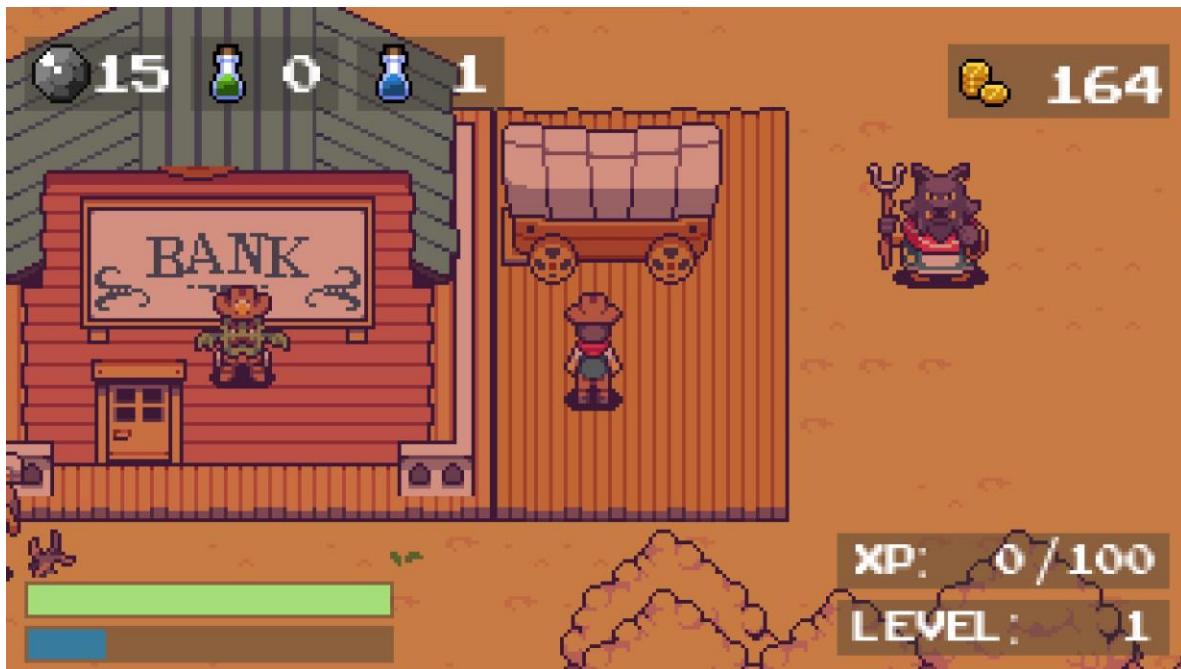
health_pickups_updated.connect(health_amount.update_health_pickup_ui)
stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)
xp_updated.connect(xp_amount.update_xp_ui)
xp_requirements_updated.connect(xp_amount.update_xp_requirements_ui)
level_updated.connect(level_amount.update_level_ui)
coins_updated.connect(coin_amount.update_coin_amount_ui)

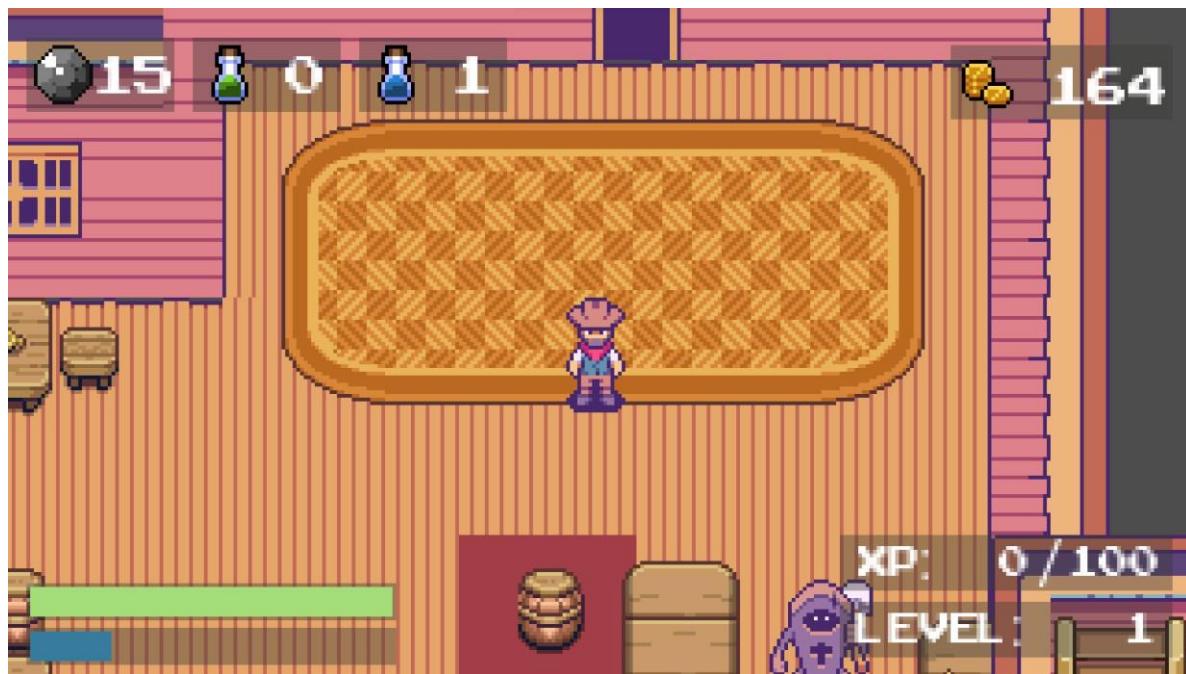
# Reset color
animation_sprite.modulate = Color(1,1,1,1)
Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

#update ui components to show correct loaded data
$UI/AmmoAmount/Value.text = str(ammo_pickup)
$UI/StaminaAmount/Value.text = str(stamina_pickup)
$UI/HealthAmount/Value.text = str(health_pickup)
$UI/XP/Value.text = str(xp)
$UI/XP/Value2.text = "/ " + str(xp_requirements)
$UI/Level/Value.text = str(level)

```

Now if you run your scene, you should be able to save/load as per usual. You should also be able to change scenes, and your data from the previous scene should carry over!





Congratulations, you now have a persistent saving and loading system! Remember to save your project, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 21: SIMPLE SHOPKEEPER

We can't finish off our RPG-Series without adding a shopkeeper to our game. We want our player to be able to buy ammo, health, and stamina pickups from our shopkeeper. This means our player does not constantly have to risk their lives to find ammo and consumables! Without any further diddle-daddling, let's add a simple shopkeeper to our game!

WHAT YOU WILL LEARN IN THIS PART:

- How to crop animation frames in Sprite2D nodes.

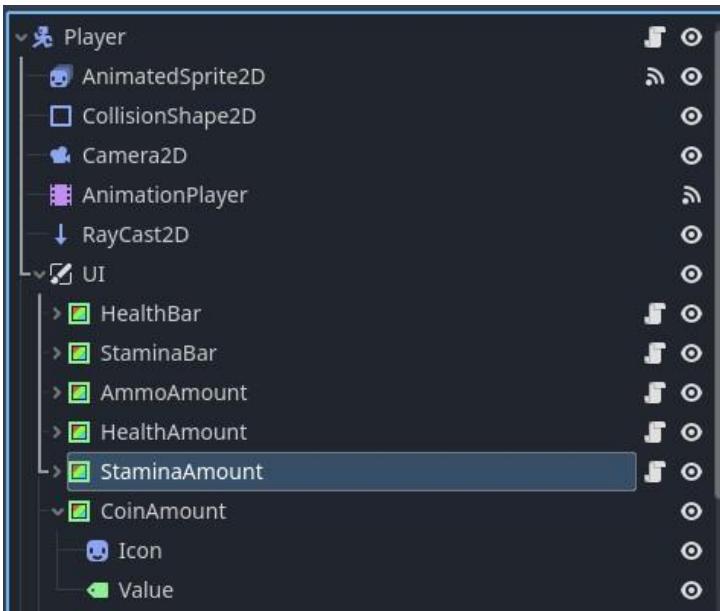
Before we create our Shop-keeper scene, we need to first give our player some coins - plus update our NPC and Enemy scripts to give our player coins when they complete a quest or kill an enemy. In your Player script, define a new variable named "coins" and give it a value. I'm going to give my player 200 coins to start with.

```
### Player.gd

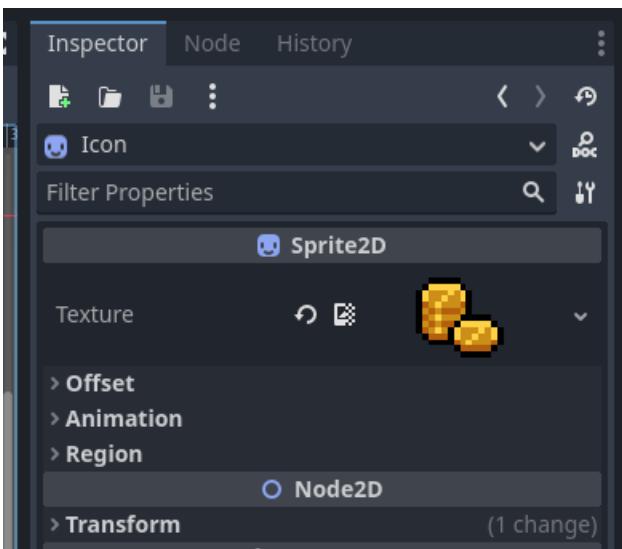
# older code

# Pickups
var ammo_pickup = 13
var health_pickup = 2
var stamina_pickup = 2
var coins = 200
```

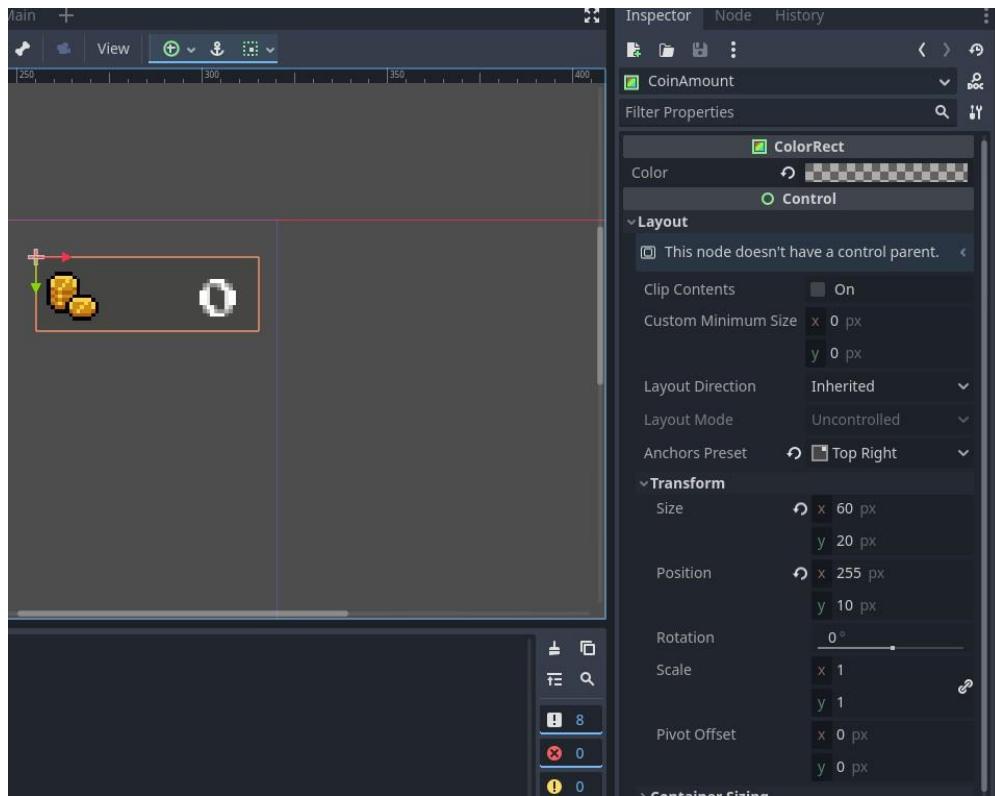
We want this coin amount to be displayed in our UI, so let's add a new UI element just for our coin amount. You can copy and paste your StaminaAmount element and rename it to "CoinAmount".



Change the CoinAmount icon to "coin_04d.png".



Then, change your CoinAmount ColorRect's transform and anchor-preset properties to be as indicated in the image below. I'm showing you the properties via images to speed things up since you should know how to change these properties by now.

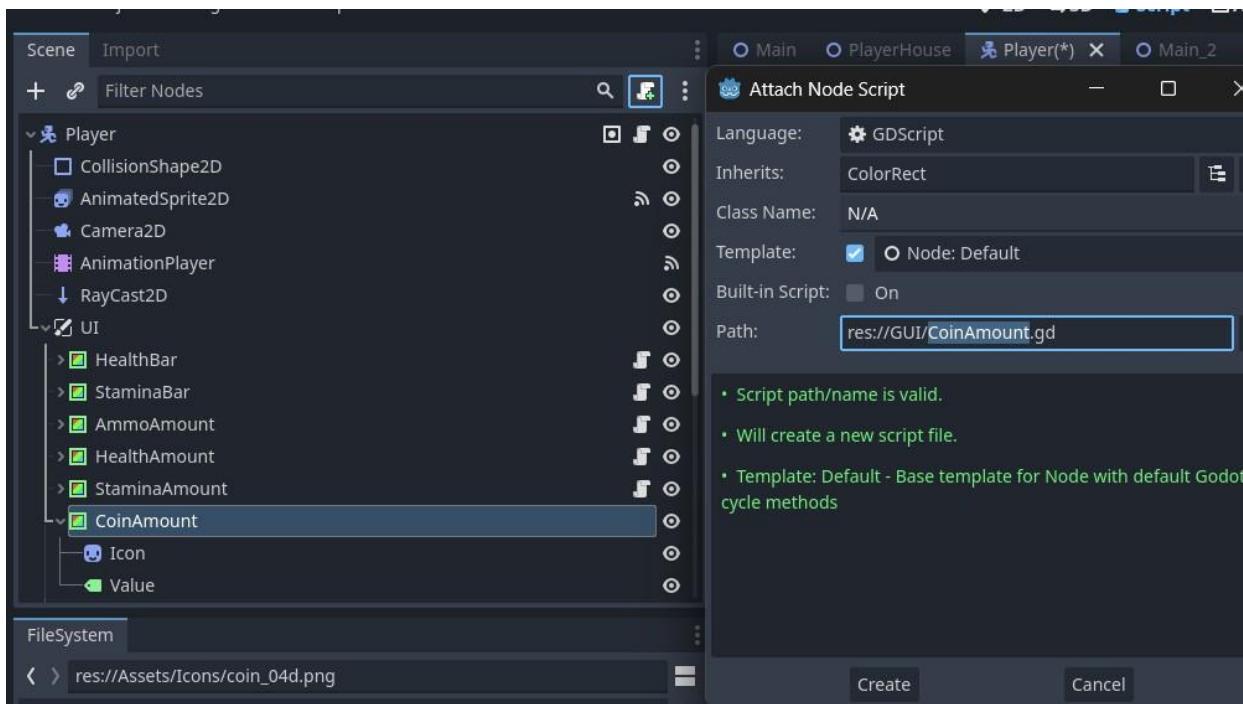


Just like with our other UI components, let's define a new signal, and attach a script to our CoinAmount node.

```
### Player.gd

# older code

# Custom signals
signal health_updated
signal stamina_updated
signal ammo_pickups_updated
signal health_pickups_updated
signal stamina_pickups_updated
signal xp_updated
signal level_updated
signal xp_requirements_updated
signal coins_updated
```



In our CoinAmount script, let's create a function to update our UI value based on our coin amount. Then, in our Player script, we will connect this function to our signal.

```
### CoinAmount.gd

extends ColorRect

# Node ref
@onready var value = $Value
@onready var player = $".../..."

# Show correct value on load
func _ready():
    value.text = str(player.coins)

# Update ui
func update_coin_amount_ui(coin_amount):
    value.text = str(coin_amount)
```

```

### Player.gd

extends CharacterBody2D

# Node references
@onready var animation_sprite = $AnimatedSprite2D
@onready var health_bar = $UI/HealthBar
@onready var stamina_bar = $UI/StaminaBar
@onready var ammo_amount = $UI/AmmoAmount
@onready var stamina_amount = $UI/StaminaAmount
@onready var health_amount = $UI/HealthAmount
@onready var coin_amount = $UI/CoinAmount

# older code

func _ready():
    # Connect the signals to the UI components' functions
    health_updated.connect(health_bar.update_health_ui)
    stamina_updated.connect(stamina_bar.update_stamina_ui)
    ammo_pickups_updated.connect(ammo_amount.update_ammo_pickup_ui)
    health_pickups_updated.connect(health_amount.update_health_pickup_ui)
    stamina_pickups_updated.connect(stamina_amount.update_stamina_pickup_ui)
    xp_updated.connect(xp_amount.update_xp_ui)
    xp_requirements_updated.connect(xp_amount.update_xp_requirements_ui)
    level_updated.connect(level_amount.update_level_ui)
    coins_updated.connect(coin_amount.update_coin_amount_ui)
    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

```

Next, we'll create a new function in our Player script that will emit the signal whenever our coin amount changes.

```

### Player

# ----- Consumables -----
# older code

# Add coins to inventory
func add_coins(coins_amount):
    coins += coins_amount
    coins_updated.emit(coins)

```

Then in our NPC and Enemy scripts, we will call this function whenever we complete a quest or kill the enemy. We'll pass in the amount of coins that we want to reward the player with as a parameter.

```
### Enemy.gd

#will damage the enemy when they get hit
func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
    else:
        #death
        #stop movement
        timer_node.stop()
        direction = Vector2.ZERO
        #stop health regeneration
        set_process(false)
        #trigger animation finished signal
        is_attacking = true
        #Finally, we play the death animation
        animation_sprite.play("death")
        #add xp values
        player.update_xp(70)
        player.add_coins(10)
        death.emit()
        #drop loot randomly at a 90% chance
        if rng.randf() < 0.9:
            var pickup = Global.pickups_scene.instantiate()
            pickup.item = rng.randi() % 3 #we have three pickups in our enum
            get_tree().root.get_node("%s/PickupSpawner/SpawnedPickups" %
                Global.current_scene_name).call_deferred("add_child", pickup)
            pickup.position = position
```

```
### NPC.gd

#dialog tree
func dialog(response = ""):
    # Set our NPC's animation to "talk"
    animation_sprite.play("talk_down")

    # Set dialog_popup npc to be referencing this npc
    dialog_popup.npc = self
```

```

dialog_popup.npc_name = str(npc_name)

# dialog tree
match quest_status:
    QuestStatus.NOT_STARTED:
        match dialog_state:
            # older code
    QuestStatus.STARTED:
        match dialog_state:
            0:
                # Update dialog tree state
                dialog_state = 1
                # Show dialog popup
                dialog_popup.message = "Found that book yet?"
                if quest_complete:
                    dialog_popup.response = "[A] Yes [B] No"
                else:
                    dialog_popup.response = "[A] No"
                dialog_popup.open()
            1:
                if quest_complete and response == "A":
                    # Update dialog tree state
                    dialog_state = 2
                    # Show dialog popup
                    dialog_popup.message = "Yeehaw! Now I can make
cat-eye soup. Here, take this."
                    dialog_popup.response = "[A] Bye"
                    dialog_popup.open()
                else:
                    # Update dialog tree state
                    dialog_state = 3
                    # Show dialog popup
                    dialog_popup.message = "I'm so hungry, please
hurry..."
                    dialog_popup.response = "[A] Bye"
                    dialog_popup.open()
            2:
                # Update dialog tree state
                dialog_state = 0
                quest_status = QuestStatus.COMPLETED
                # Close dialog popup
                dialog_popup.close()
                # Set NPC's animation back to "idle"
                animation_sprite.play("idle_down")
                # Add pickups and XP to the player.

```

```
        player.add_pickup(Global.Pickups.AMMO)
        player.update_xp(50)
        player.add_coins(20)
```

Don't forget to also save and load your coin data in your Player script.

```
### Player.gd

----- Saving & Loading -----
#data to save
func data_to_save():
    return {
        "position" : [position.x, position.y],
        "health" : health,
        "max_health" : max_health,
        "stamina" : stamina,
        "max_stamina" : max_stamina,
        "xp" : xp,
        "xp_requirements" : xp_requirements,
        "level" : level,
        "ammo_pickup" : ammo_pickup,
        "health_pickup" : health_pickup,
        "stamina_pickup" : stamina_pickup,
        "coins" : coins
    }

#loads data from saved data
func data_to_load(data):
    position = Vector2(data.position[0], data.position[1])
    health = data.health
    max_health = data.max_health
    stamina = data.stamina
    max_stamina = data.max_stamina
    xp = data.xp
    xp_requirements = data.xp_requirements
    level = data.level
    ammo_pickup = data.ammo_pickup
    health_pickup = data.health_pickup
    stamina_pickup = data.stamina_pickup
    coins = data.coins

#loads data from saved data
func values_to_load(data):
    health = data.health
    max_health = data.max_health
```

```

stamina = data.stamina
max_stamina = data.max_stamina
xp = data.xp
xp_requirements = data.xp_requirements
level = data.level
ammo_pickup = data.ammo_pickup
health_pickup = data.health_pickup
stamina_pickup = data.stamina_pickup
coins = data.coins

#update ui components to show correct loaded data
$UI/AmmoAmount/Value.text = str(data.ammo_pickup)
$UI/StaminaAmount/Value.text = str(data.stamina_pickup)
$UI/HealthAmount/Value.text = str(data.health_pickup)
$UI/XP/Value.text = str(data.xp)
$UI/XP/Value2.text = "/" + str(data.xp_requirements)
$UI/Level/Value.text = str(data.level)
$UI/CoinAmount/Value.text = str(data.coins)

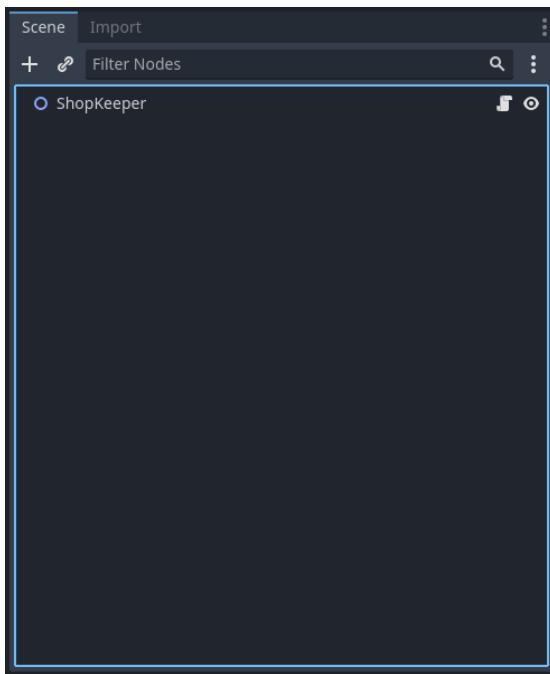
```

If you run your scene now and you kill an enemy or complete a quest, your coin amount should update!

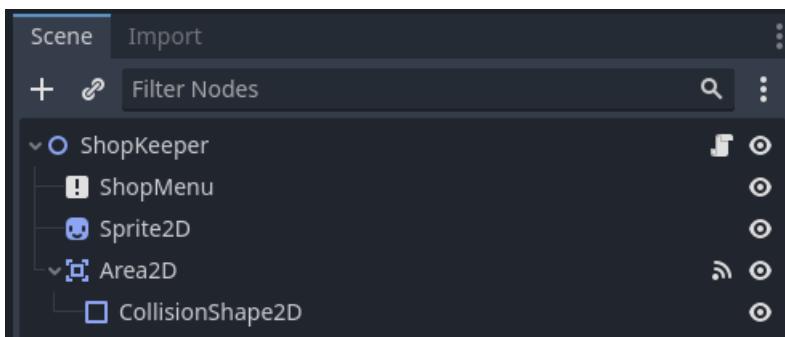


With our player's coins set up, we can go ahead and create our shopkeeper. Let's create a new scene with a Node2D node as its root. We're using this node because we won't move this character around, so a CharacterBody2D node would be redundant.

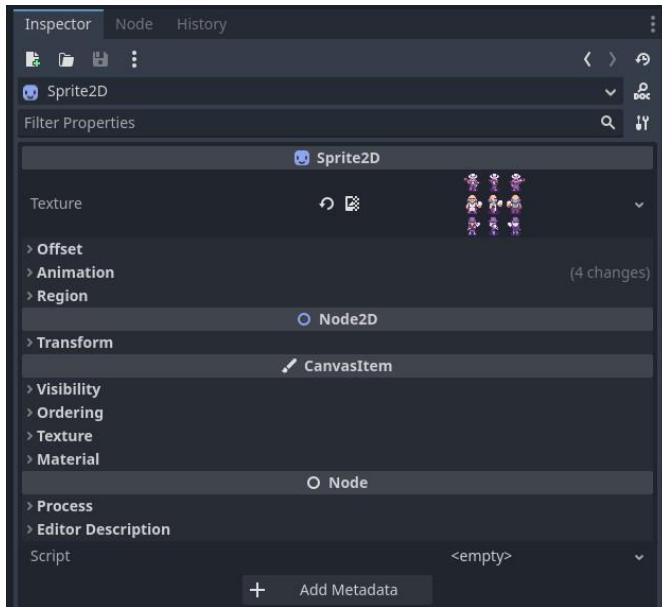
Rename this root as “ShopKeeper” and save the scene in your Scenes folder. Also, attach a script to it and save it in your Scripts folder.



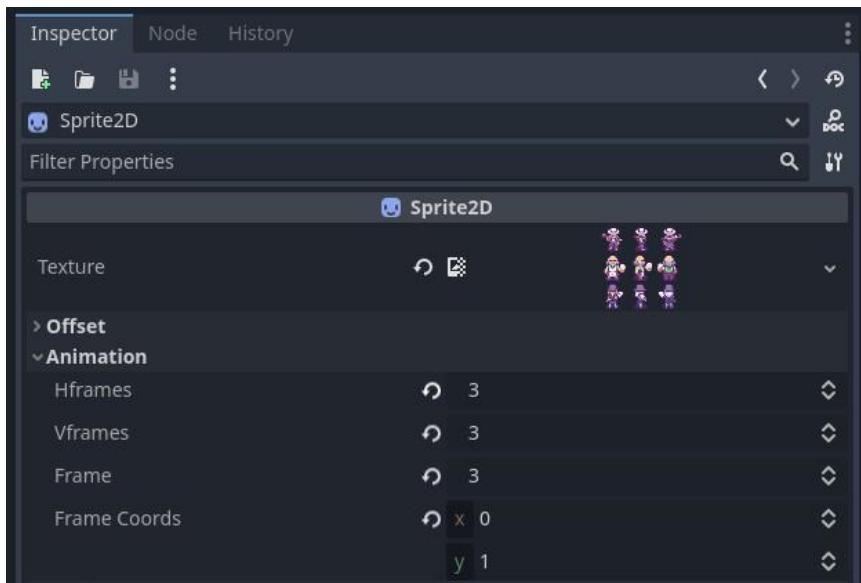
For this node, we want to have a simple **Sprite2D** that will show our shopkeeper's body. In front of this body we want to have an **Area2D** node that if the player enters its body, the ShopMenu **CanvasLayer** will be displayed. The ShopMenu popup will contain a list of our pickups items that the player can buy for certain prices. Let's add the following nodes:



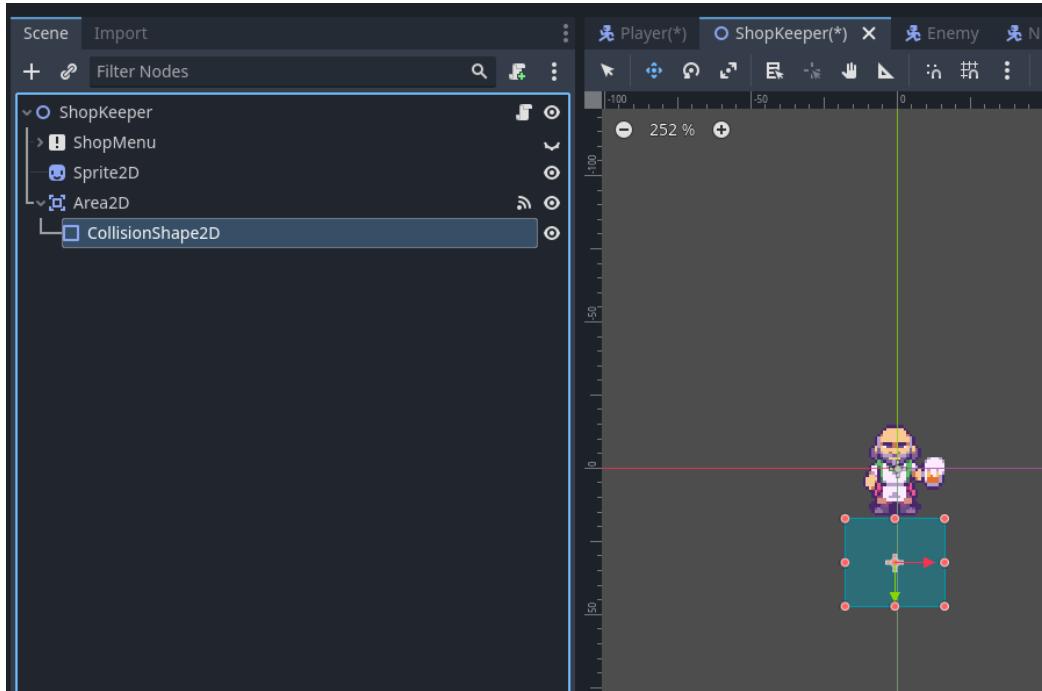
In your Assets directory, there is a folder called "NPC". Assign the "NPC's.png" image to your Sprite2D node.



We want to crop out the first person in the second row (the man holding the beer). To do this, we need to change the [HFrames](#), [VFrames](#), and [Frames](#) values in our Animations property in the Inspector panel. The HFrames refer to horizontal frames. We can count 3 frames because there are 3 people per row, so its value should be three. The same should go for our VFrames. Then we just change our Frames value until we get to our beer-guy!

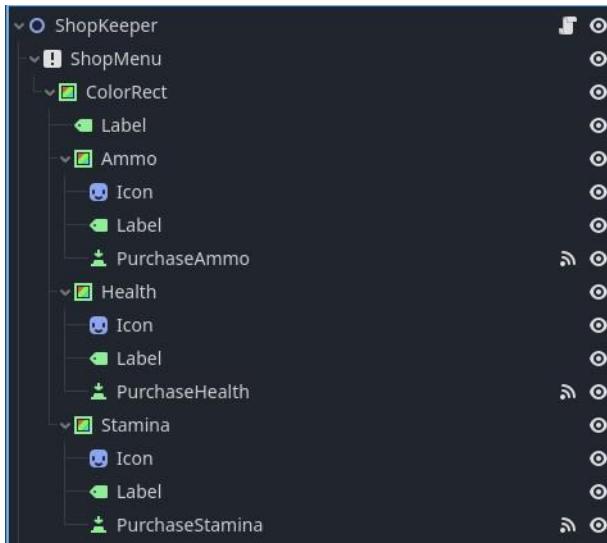


Then, let's add a rectangular collision shape to our Area2D node and move it in front of our shopkeeper.

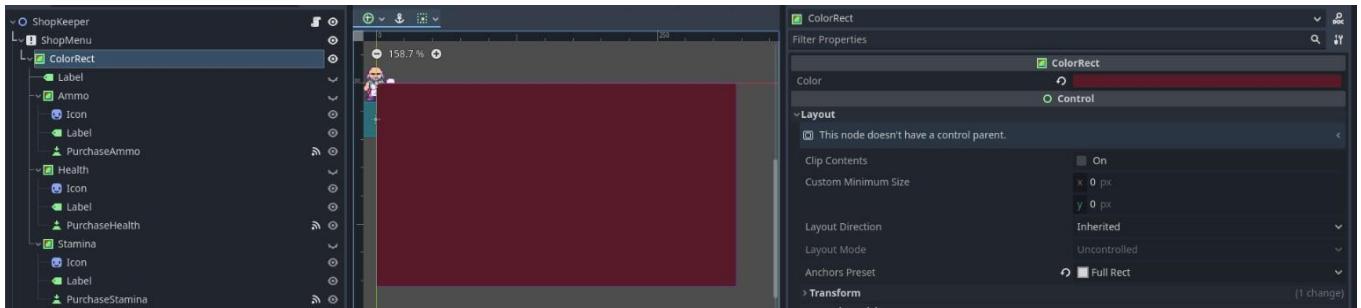


Now, here comes the work! UI creation is always the most tedious part of game development - well, for me at least. For our ShopMenu, we want three ColorRects to show the icon, label, and purchasing button for our Ammo, Health, and Stamina pickups. If we had a dynamic inventory (an inventory that changes item types), we'd be doing this via [Lists and Boxes](#), but because we have a static inventory (an inventory that doesn't change) that is composed of just 3 items, we'll just go ahead and add a ColorRect, Label, Sprite2D, and Button node for each.

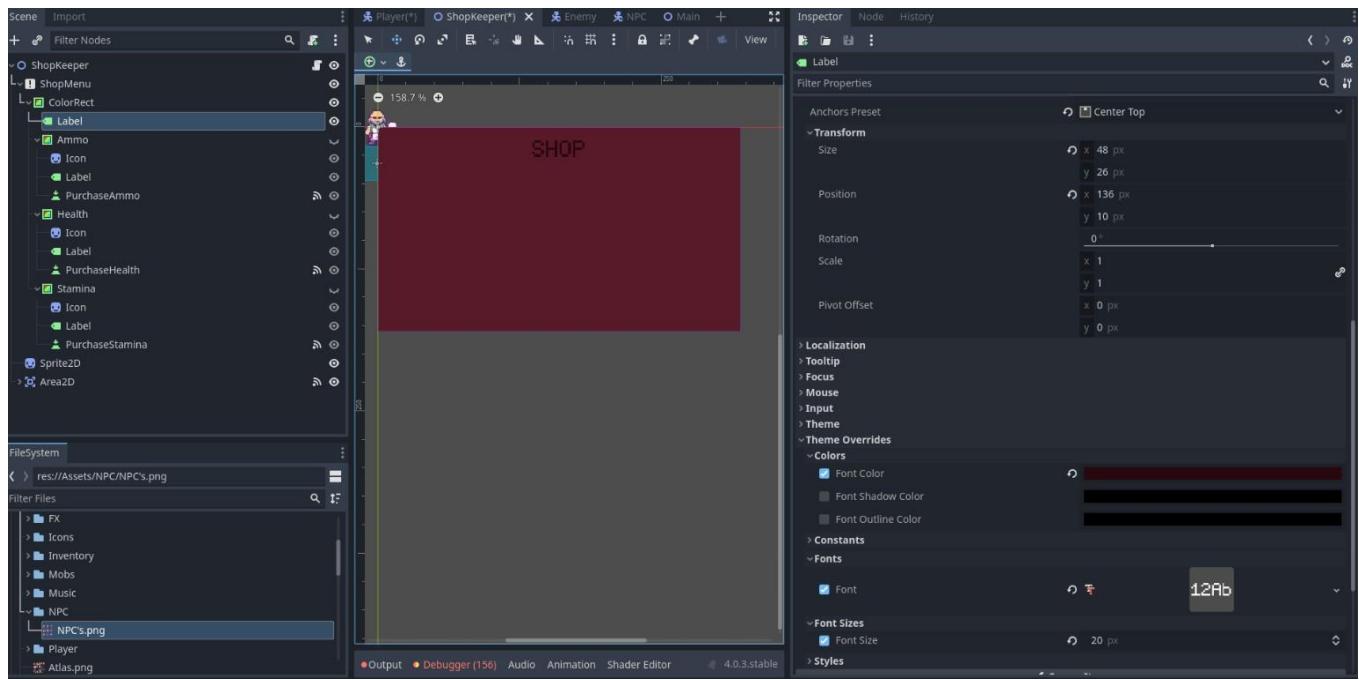
Add the following nodes (ColorRect > Label and 3 x ColorRect > Sprite2D > Label > Button) and rename them as indicated in the picture below.



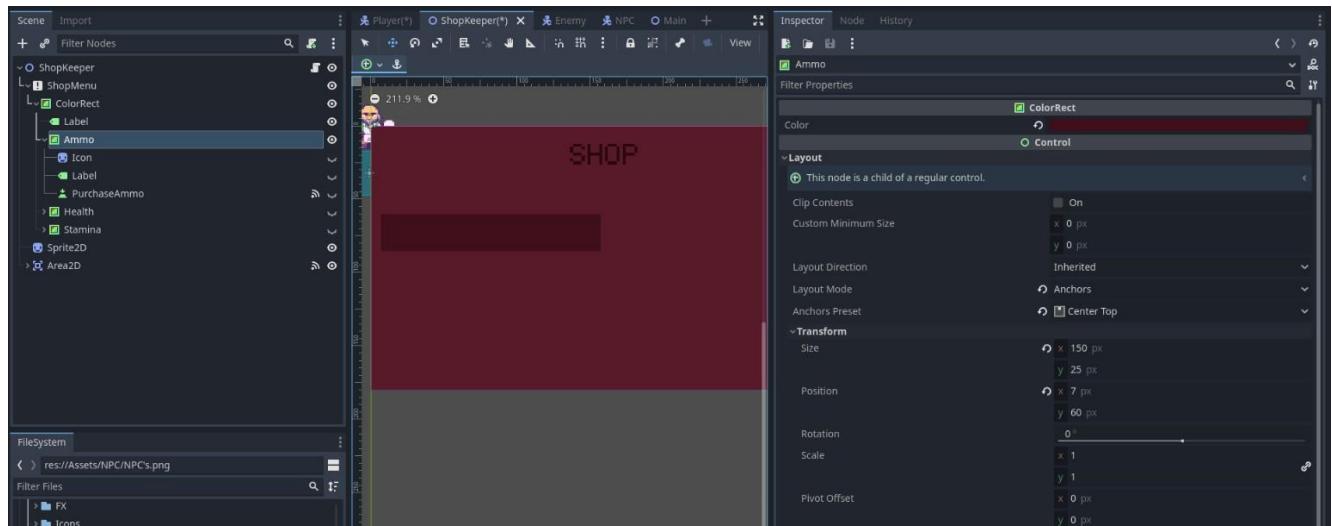
Change your first ColorRect's color to #581929, and change its anchor preset to "Full-Rect".

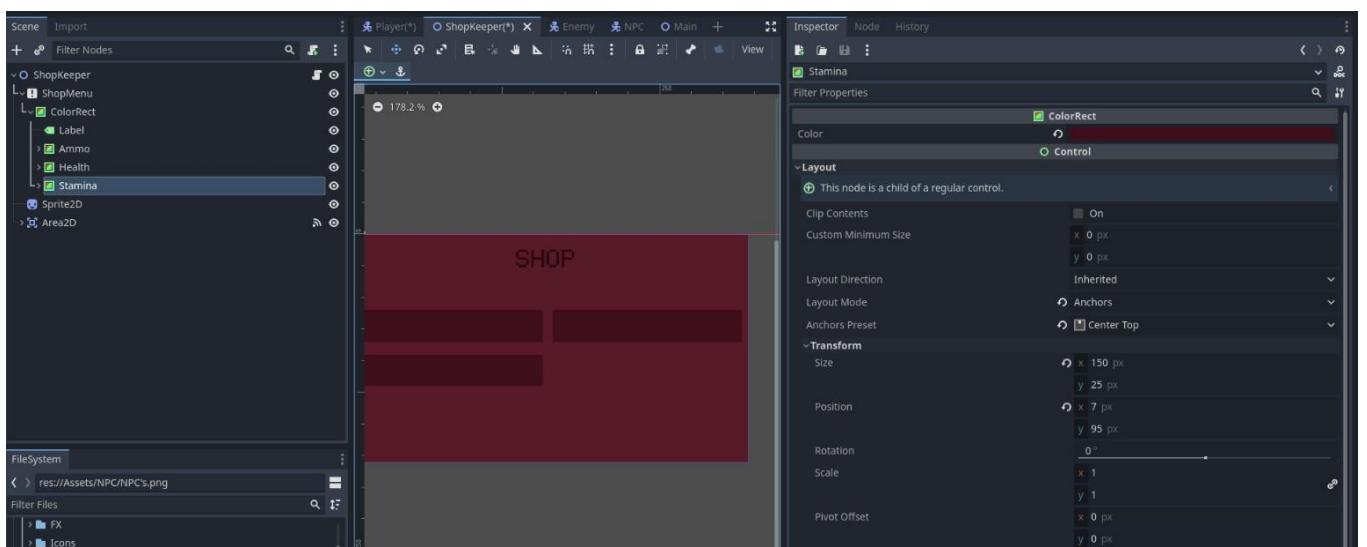
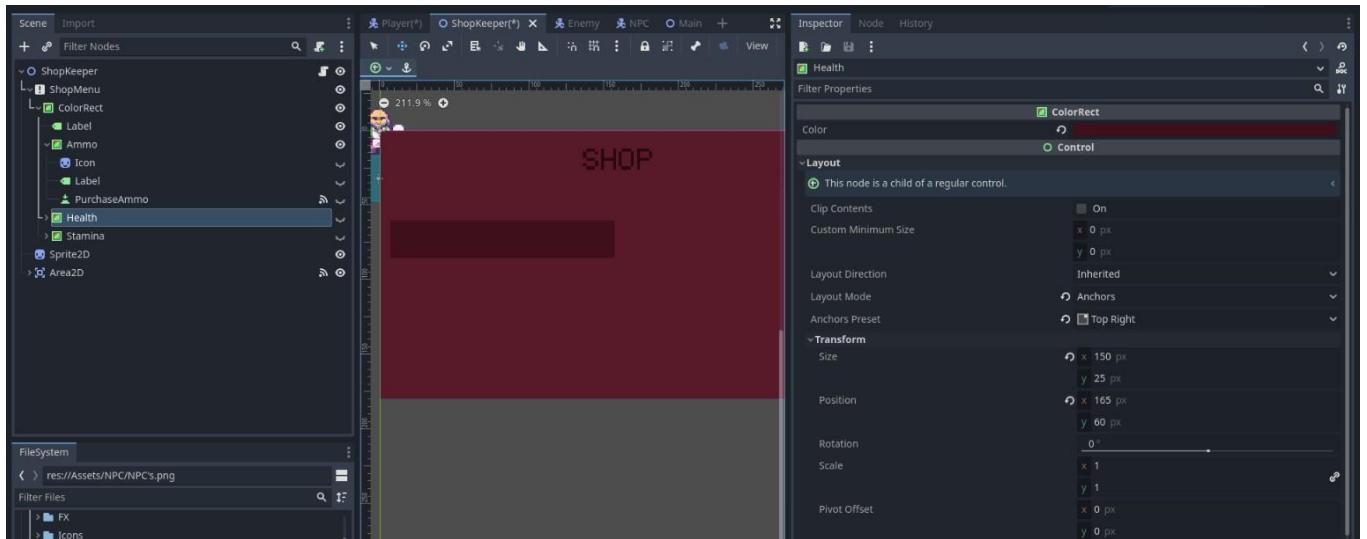


Then, change your Label nodes text to "SHOP". Change its font size to 20, font to "Schrödinger", and its font color to #2a0810. Change its transform and preset values to match that of the image below.

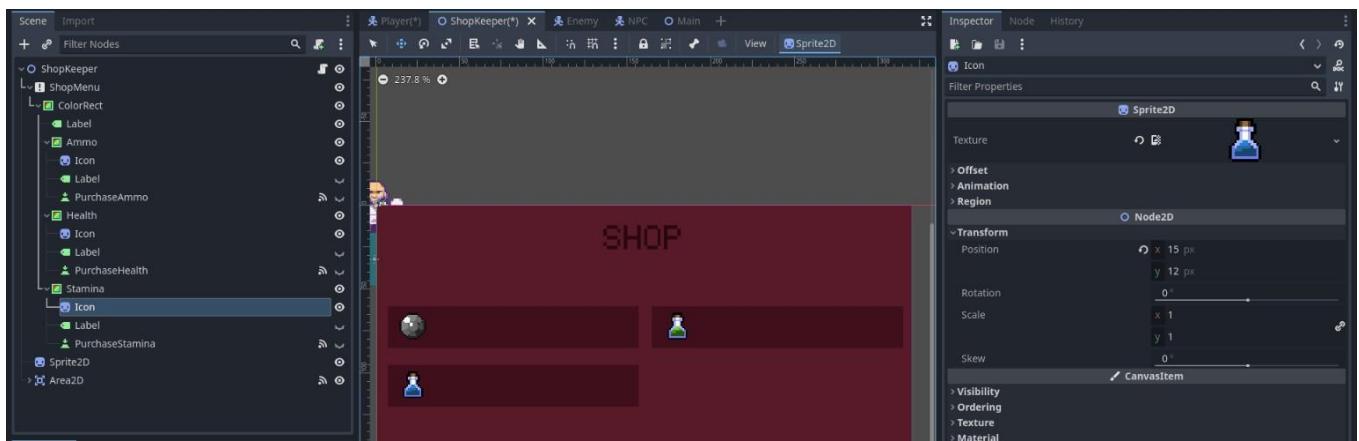
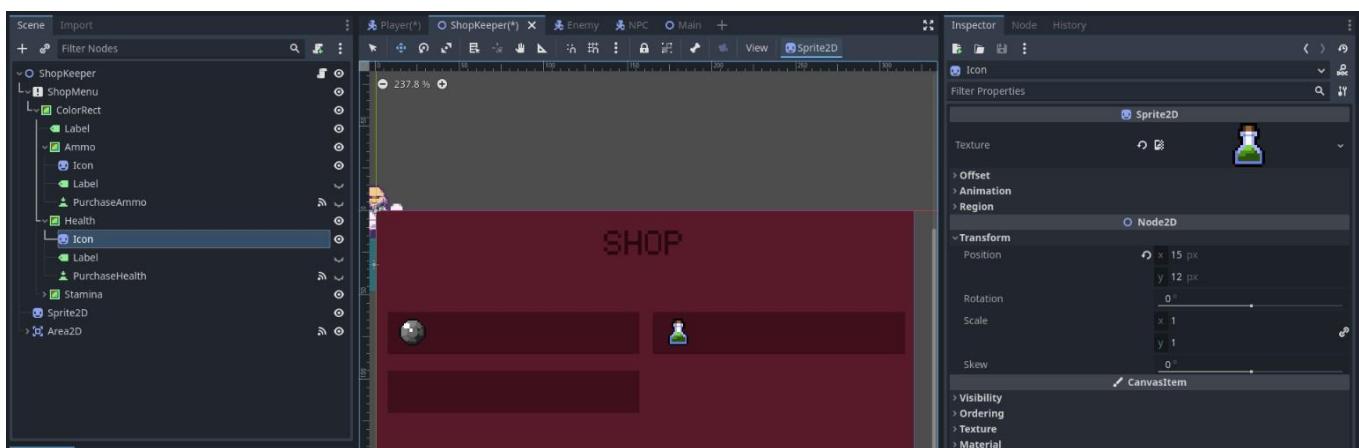
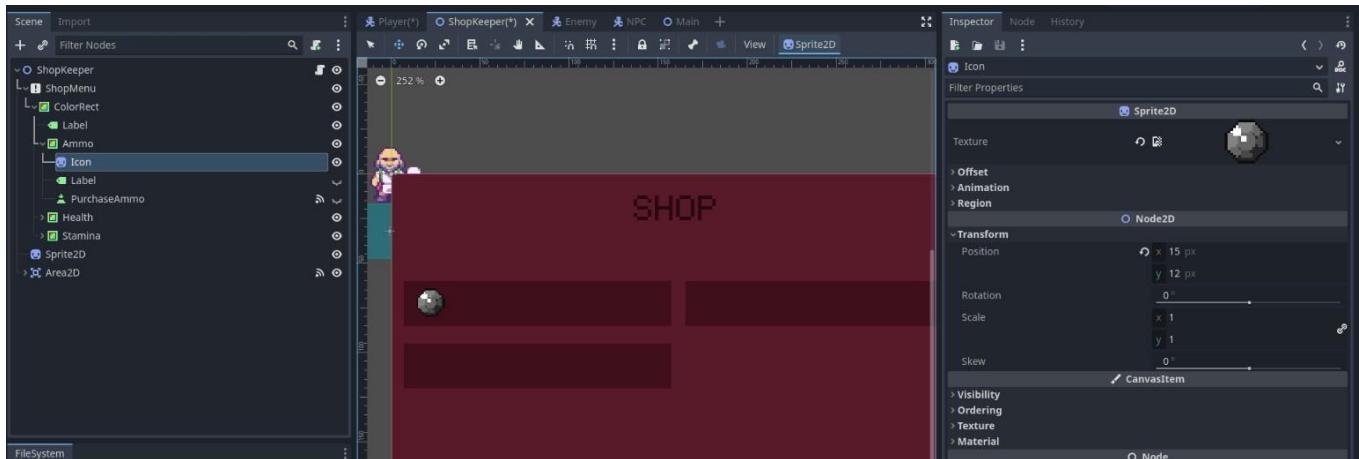


Change your Ammo ColorRect's color to #3f0f1b. Change its transform and preset values to match that of the image below. You can do the same for your Health and Stamina ColorRects.

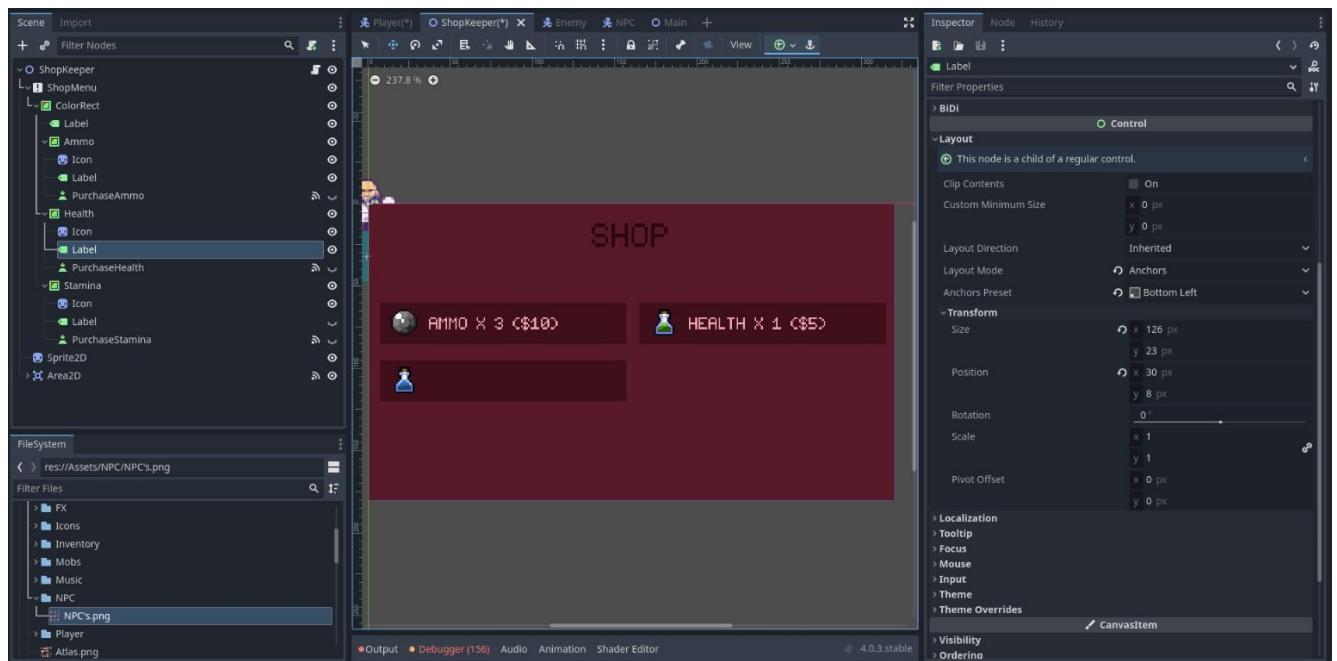
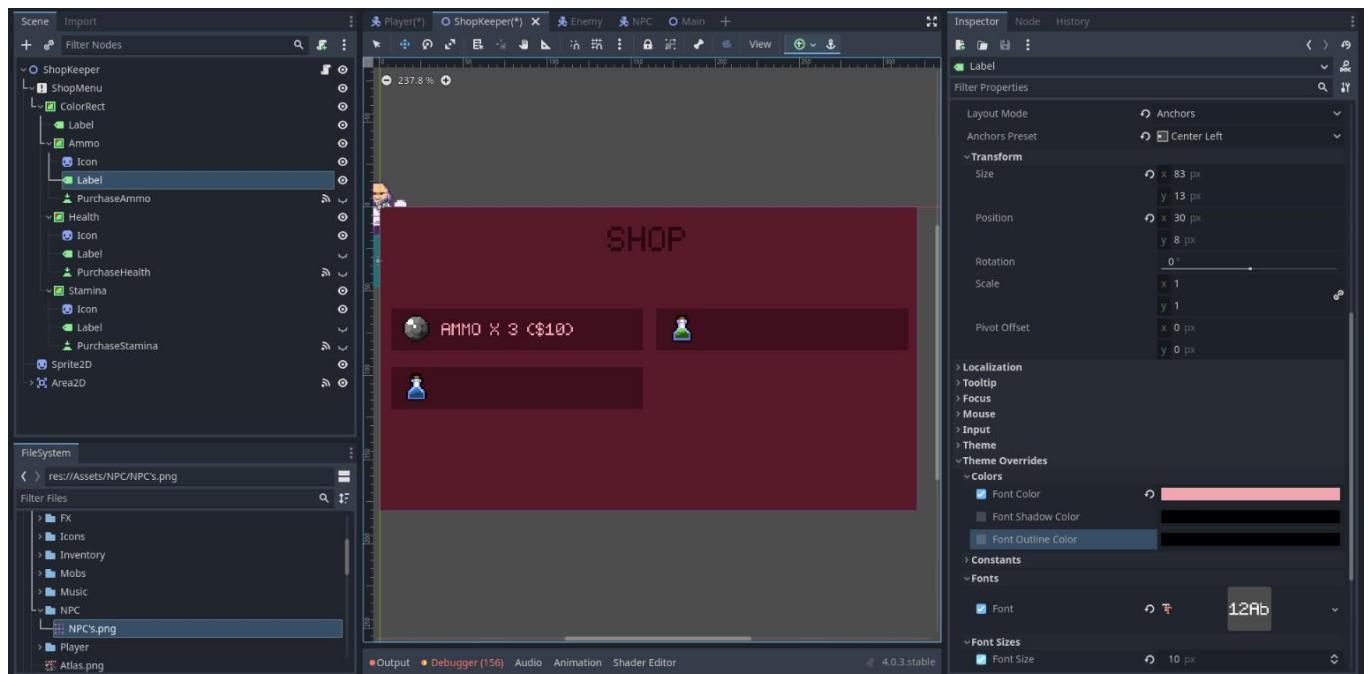


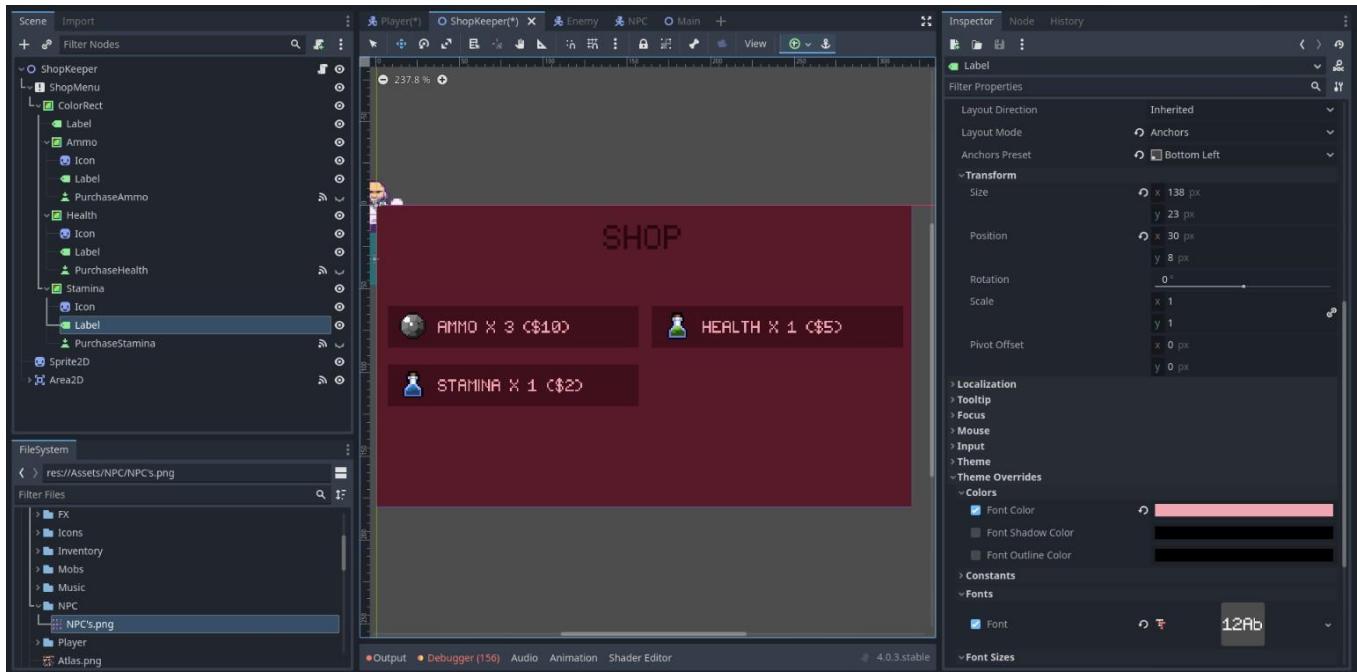


Then, change your Icon to the icon you chose for your Ammo on your Player's UI. Change its transform and preset values to match that of the image below. You can do the same for your Health and Stamina Icons.

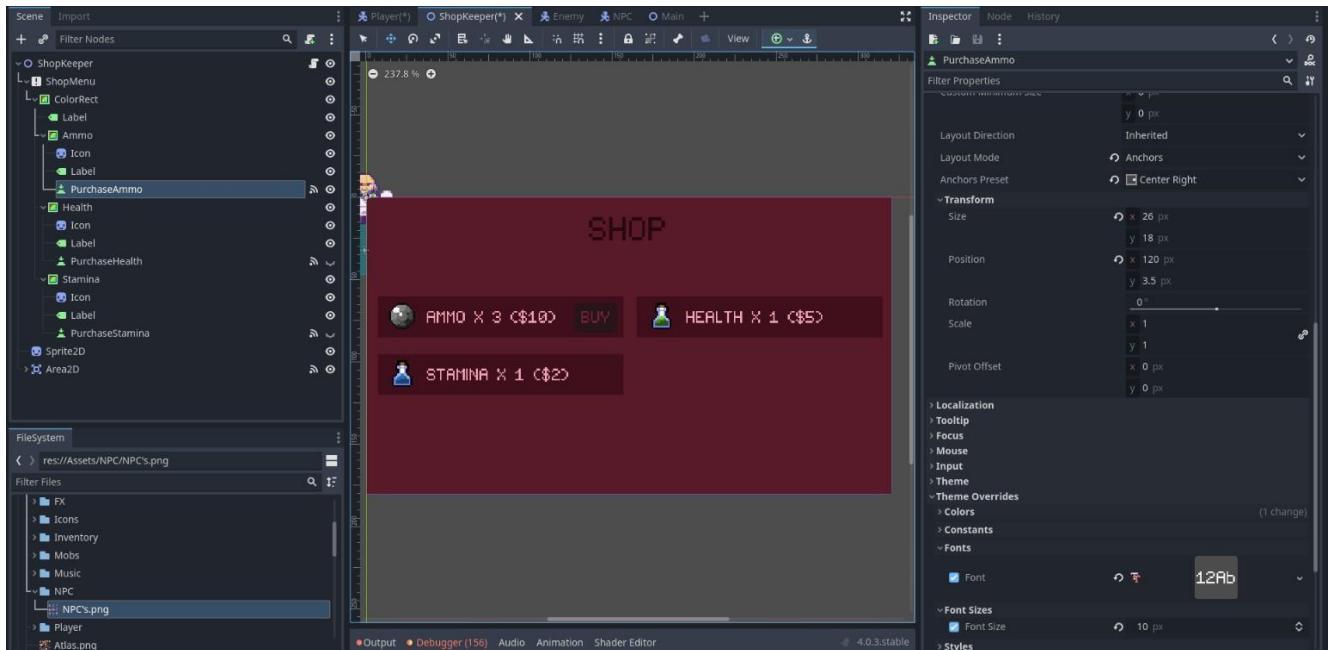


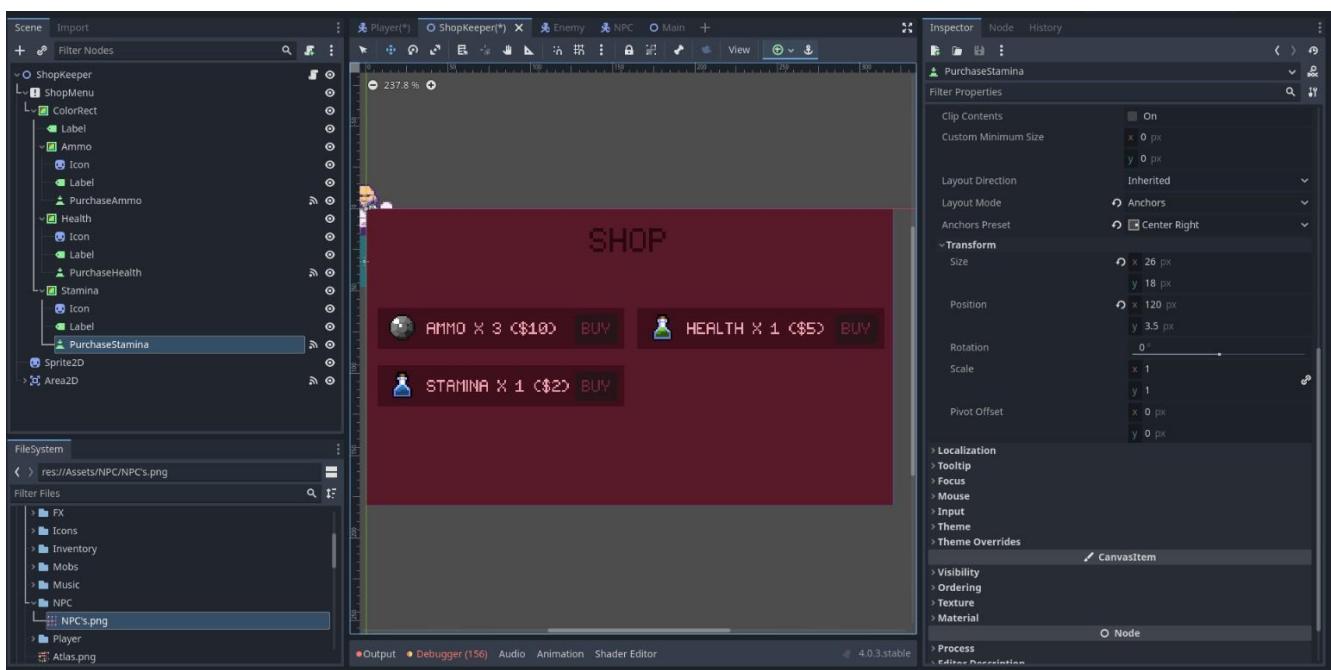
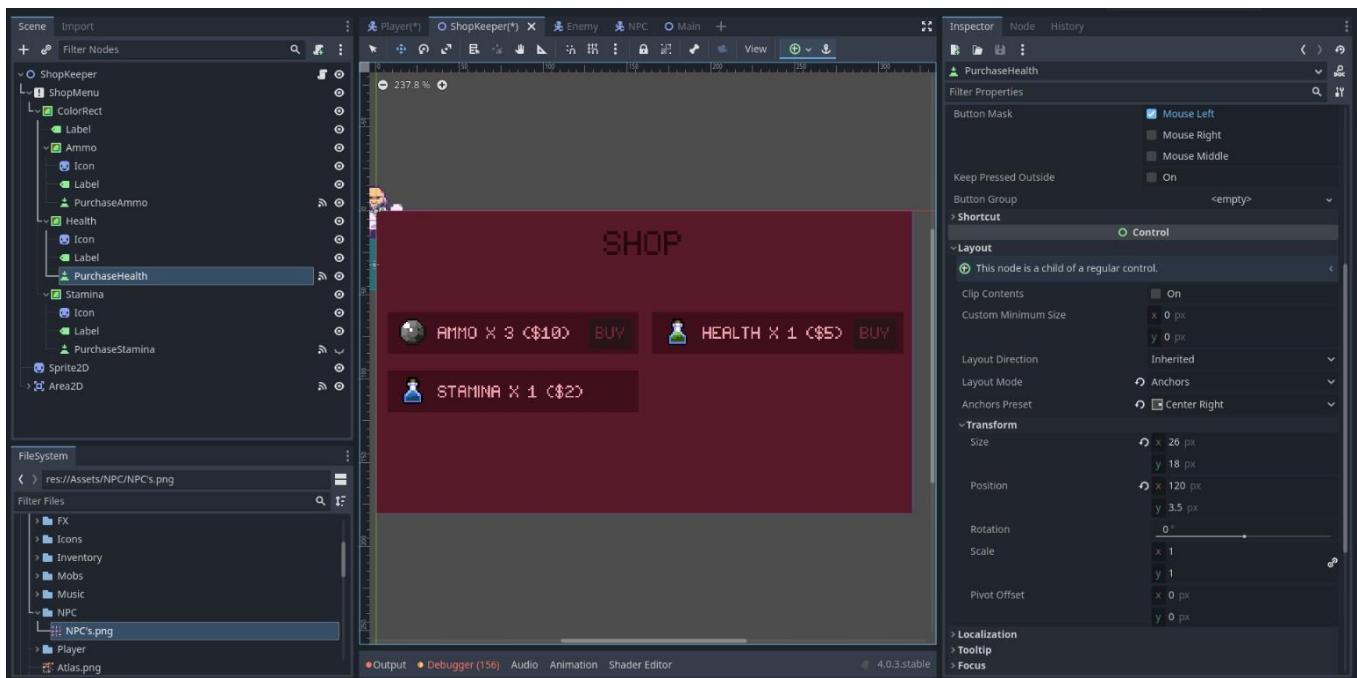
Change your Ammo's Label to be with the font "Schrödinger", size 10, and font color #f2a6b2. Change its transform, text, and preset values to match that of the image below. You can do the same for your Health and Stamina Labels.





Next, change your PurchaseAmmo button's font to "Schrödinger", size 10, and font color #77253a. Change its transform, text, and preset values to match that of the image below. You can do the same for your HealthPurchase and StaminaPurchase buttons.

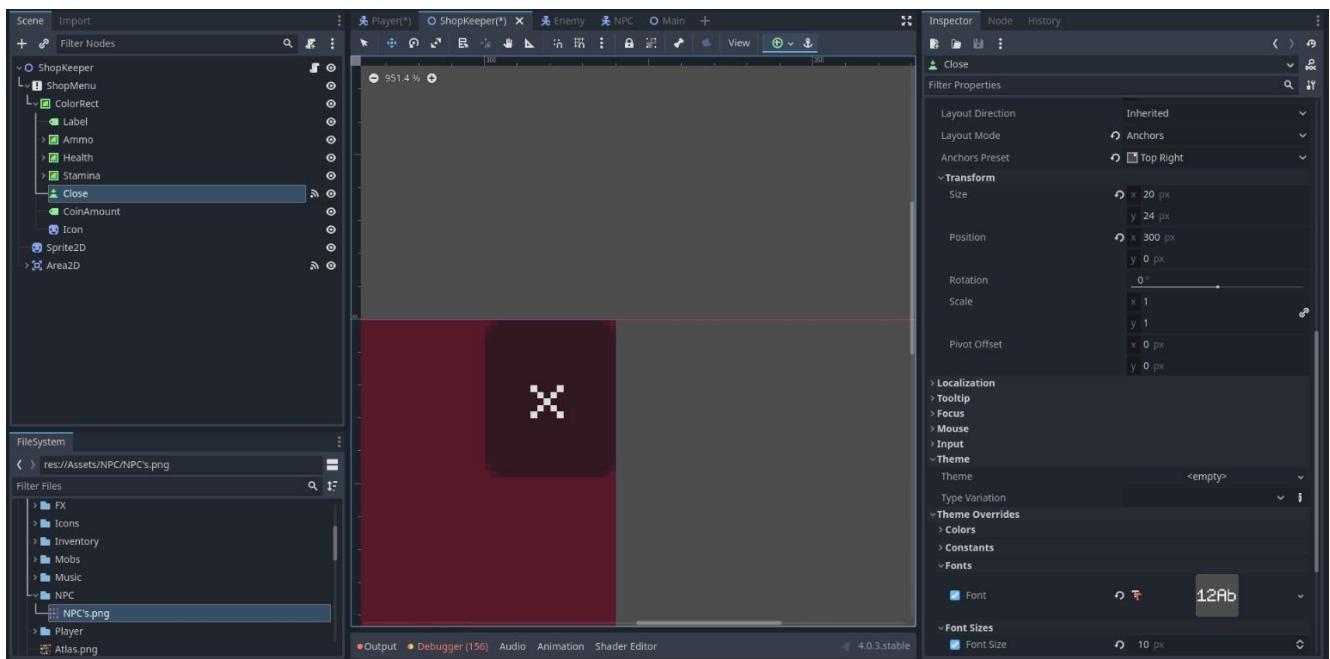


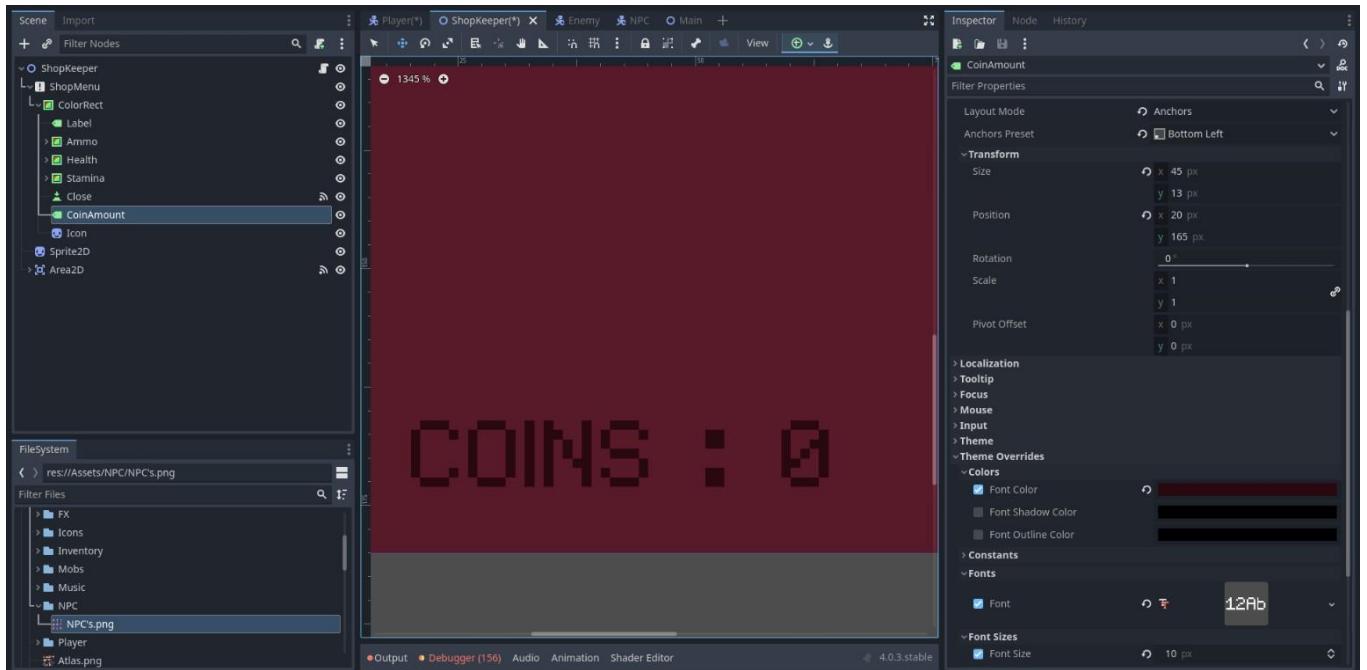


Now, to our main ColorRect, we need to add three new nodes: Sprite2D, Label, and Button. The Sprite2D and Label will show us our remaining coin value and the Button will allow us to close the menu.

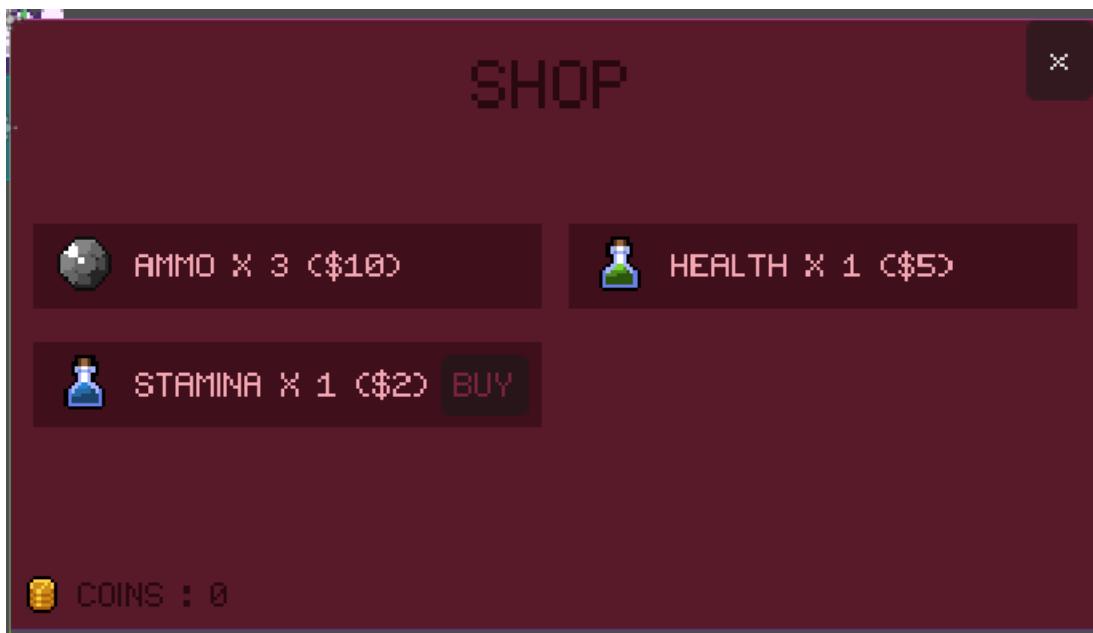


Change their values to match that of the images below. The Close node is the Button, the CoinAmount node is the Label (change its font color to #2a0810, with the font "Schrödinger"), and the Icon is our coin Sprite2D (choose "coin_03d.png" to be its texture).

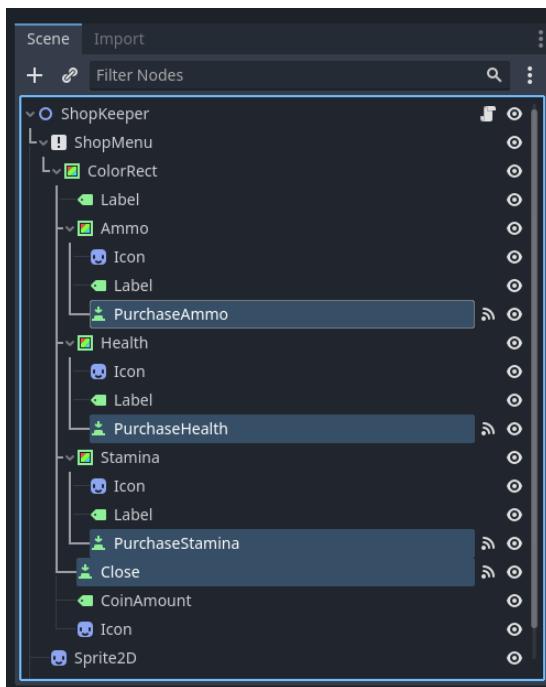


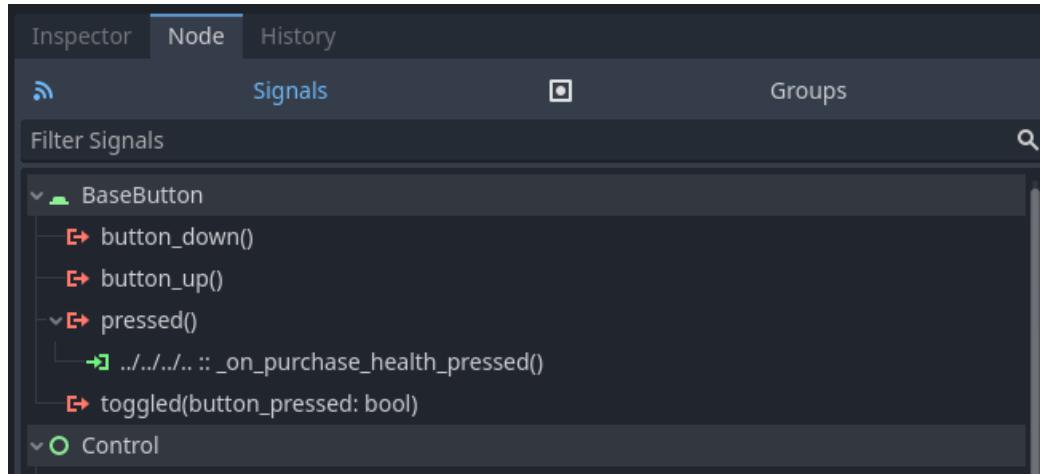


Your final UI for your ShopMenu should look like this:

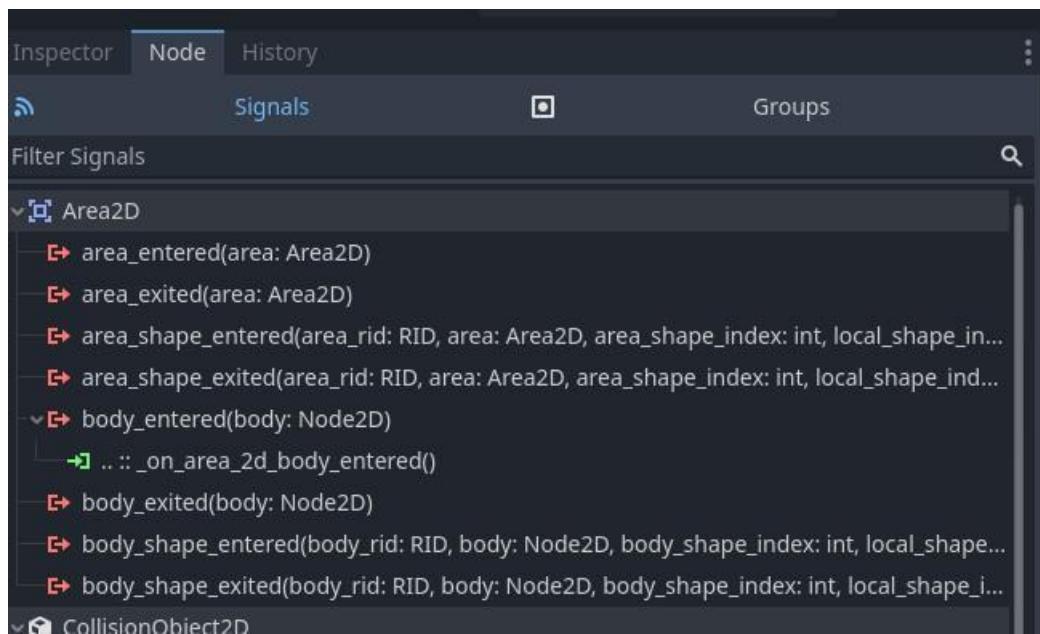


Now, connect each of your Button's *pressed()* signal to your script. If we press these buttons, we will purchase our Pickup for each, and our coin amount should be updated. Our close button should hide the screen and unpause our game.





Also, connect your Area2D node's body_entered() signal to your script. We will use this to show our screen and pause the game.



We first need to get a reference to our player node since we want to update and check their coin amount, as well as call their `add_pickup()` function. We'll also update the coin value returned in our popup in our `process()` function. In our `ready()` function, we will initialize our player reference and hide our screen to ensure that it is hidden when the shopkeeper enters the Main scene on game load.

```
###ShopKeeper.gd

extends Node2D

@onready var player = get_tree().root.get_node("Main/Player")
@onready var shop_menu = $ShopMenu

#player reference
func _ready():
    shop_menu.visible = false

#updates coin amount
func _process(delta):
    $ShopMenu/ColorRect/CoinAmount.text = "Coins: " + str(player.coins)
```

Then, we'll open and close our “popup”. Remember to set the nodes visibility to hidden by default.

```
###ShopKeeper.gd

extends Node2D

@onready var player = get_tree().root.get_node("Main/Player")
@onready var shop_menu = $ShopMenu

#player reference
func _ready():
    shop_menu.visible = false

#updates coin amount
func _process(delta):
    $ShopMenu/ColorRect/CoinAmount.text = "Coins: " + str(player.coins)

func _on_close_pressed():
    shop_menu.visible = false
    get_tree().paused = false
    set_process_input(false)
    player.set_physics_process(true)

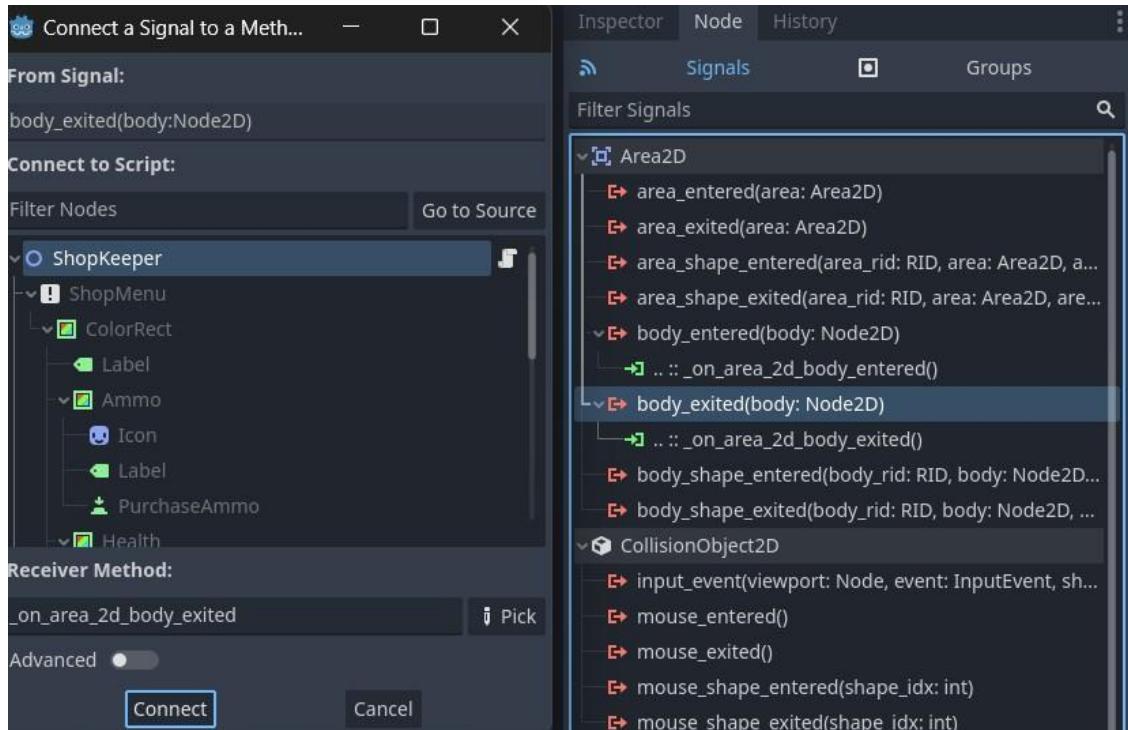
func _on_area_2d_body_entered(body):
    if body.is_in_group("player"):
        shop_menu.visible = true
        get_tree().paused = true
```

```

set_process_input(true)
player.set_physics_process(false)

```

We can also hide our menu in our Area2D node's *body_exited* signal, which will ensure that the menu is disabled if we aren't in the Area2D body. Also show/hide your cursor.



```

###ShopKeeper.gd

extends Node2D

@onready var player = get_tree().root.get_node("Main/Player")
@onready var shop_menu = $ShopMenu

func _ready():
    shop_menu.visible = false

#updates coin amount
func _process(delta):
    $ShopMenu/ColorRect/CoinAmount.text = "Coins: " + str(player.coins)

# Show Menu
func _on_area_2d_body_entered(body):
    if body.is_in_group("player"):

        set_process_input(true)
        player.set_physics_process(false)

    else:
        set_process_input(false)
        player.set_physics_process(true)

```

```

shop_menu.visible = true
get_tree().paused = true
set_process_input(true)
player.set_physics_process(false)
Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)

# Close Menu
func _on_close_pressed():
    shop_menu.visible = false
    get_tree().paused = false
    set_process_input(false)
    player.set_physics_process(true)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

func _on_area_2d_body_exited(body):
    if body.is_in_group("player"):
        shop_menu.visible = false
        get_tree().paused = false
        set_process_input(false)
        player.set_physics_process(true)
        Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)

```

And then finally, we need to purchase our pickups only if our player has enough coins. You can set this value to be anything, or you could define a variable for each instead of making it a constant value as I did.

```

###ShopKeeper.gd

extends Node2D

@onready var player = get_tree().root.get_node("Main/Player")
@onready var shop_menu = $ShopMenu

func _ready():
    shop_menu.visible = false

#updates coin amount
func _process(delta):
    $ShopMenu/ColorRect/CoinAmount.text = "Coins: " + str(player.coins)

#purchases ammo at the cost of $10
func _on_purchase_ammo_pressed():
    if player.coins >= 10:
        player.add_pickup(Global.Pickups.AMMO)

```

```

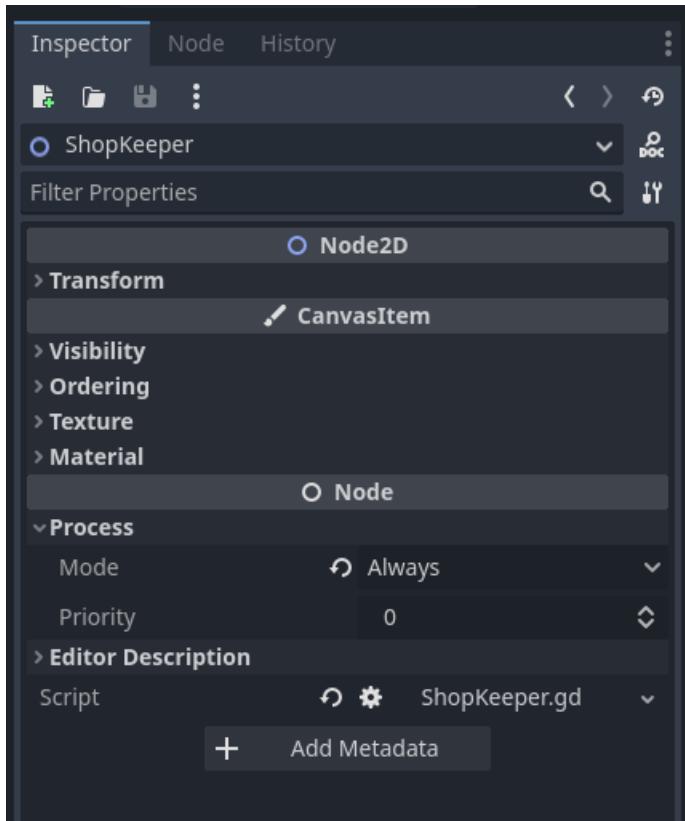
player.coins -= 10
player.add_coins(player.coins)

#purchases health at the cost of $5
func _on_purchase_health_pressed():
    if player.coins >= 5:
        player.add_pickup(Global.Pickups.HEALTH)
        player.coins -= 5
        player.add_coins(player.coins)

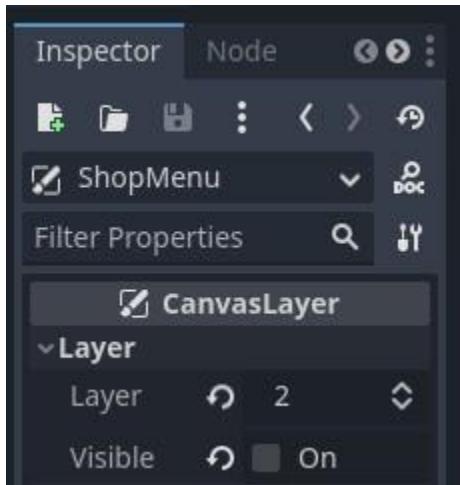
#purchases stamina at the cost of $2
func _on_purchase_stamina_pressed():
    if player.coins >= 2:
        player.add_pickup(Global.Pickups.STAMINA)
        player.coins -= 2
        player.add_coins(player.coins)

```

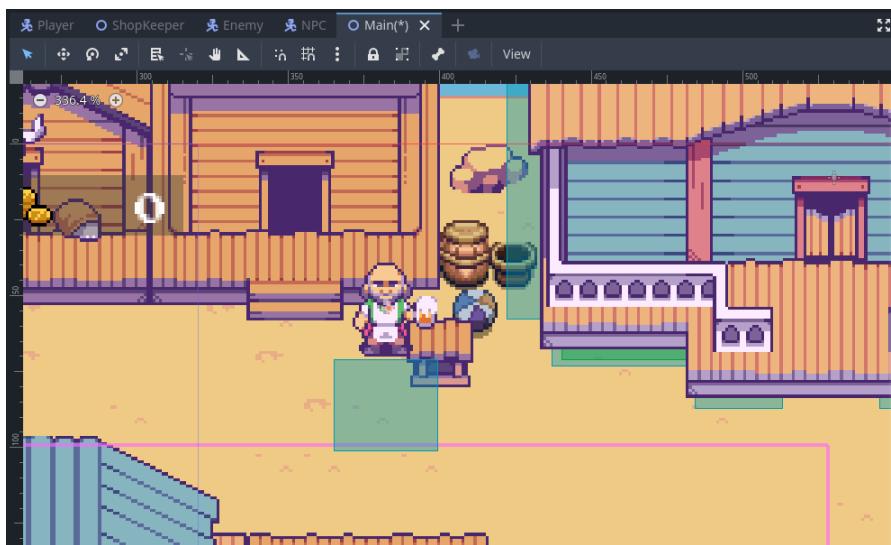
The last thing that we need to do is to change our ShopKeeper's processing mode to Always because their popup must show when the game is paused, but the Area2D node must trigger the signal if the player runs into it when the game is not paused.



We'll also need to change our ShopMenu's layer property to be 2 or higher. This will show the menu over our Player's UI, as it is on a higher z-index. The z-index determines which element appears "on top" when multiple elements occupy the same space. Elements with a higher Z-index value are rendered on top of elements with a lower Z-index value.



Instance your ShopKeeper in the Main scene. Now if you run your scene, and you run into your ShopKeeper, your menu should show, and you should be able to purchase some pickups. If you close your menu, the values should carry over into your Player's HUD. Killing Enemies and completing quests should also increase your coin amount!



SHOP

X



AMMO X 3 (\$100)

BUY



HEALTH X 1 (\$5)

BUY

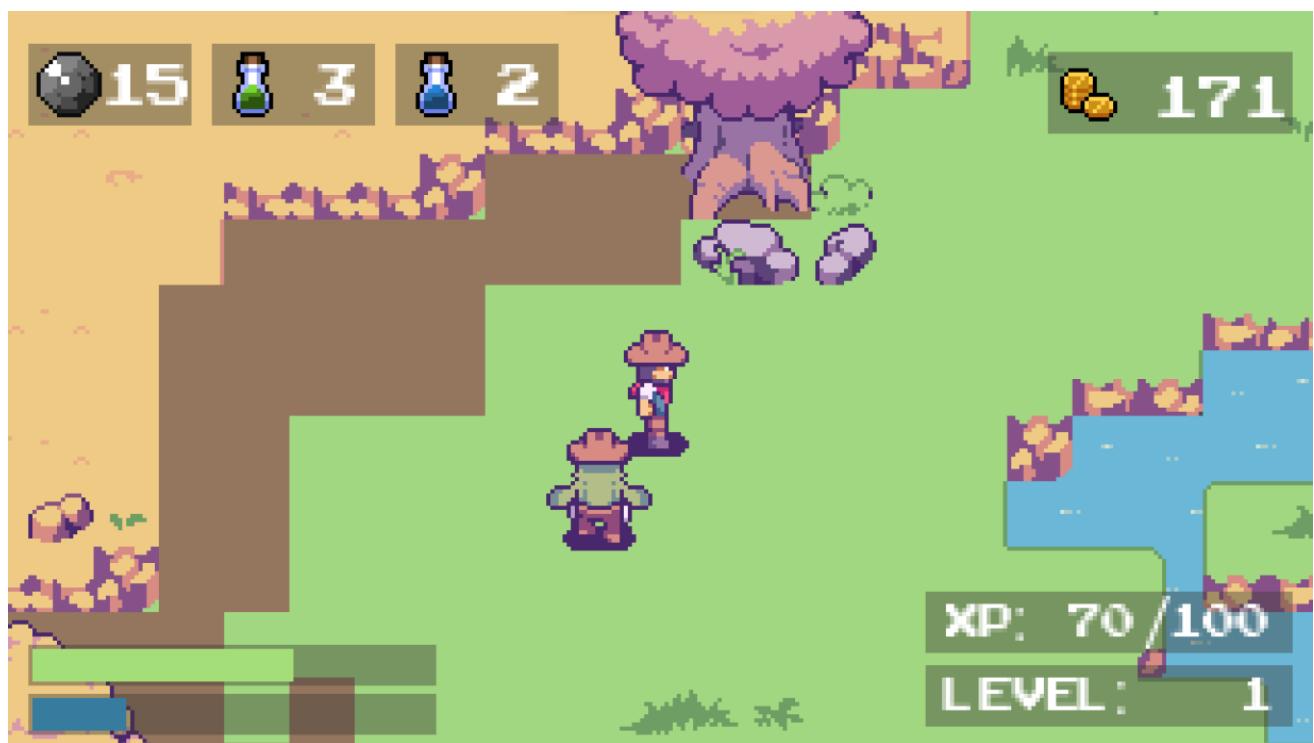


STAMINA X 1 (\$2)

BUY



COINS: 176



There are many ways to implement a shopkeeper system, and many of them are a lot better than this, but this was the simplest way that worked for our game. In the next part, we will add music and SFX to our game. Remember to save your project, and I'll see you in the next part!

The final source code for this part should look like [this](#).

PART 22: MUSIC AND SOUND EFFECTS

In this part we are going to add that final touch of life to our game by adding some music and SFX (sound effects) to our game. We want there to be background music when we are in our pause, death, and main menu screens, as well as in our Main scene. We'll also add shooting and damage sound effects to our Player and Enemy, as well as dialog music and pickup effects.

WHAT YOU WILL LEARN IN THIS PART:

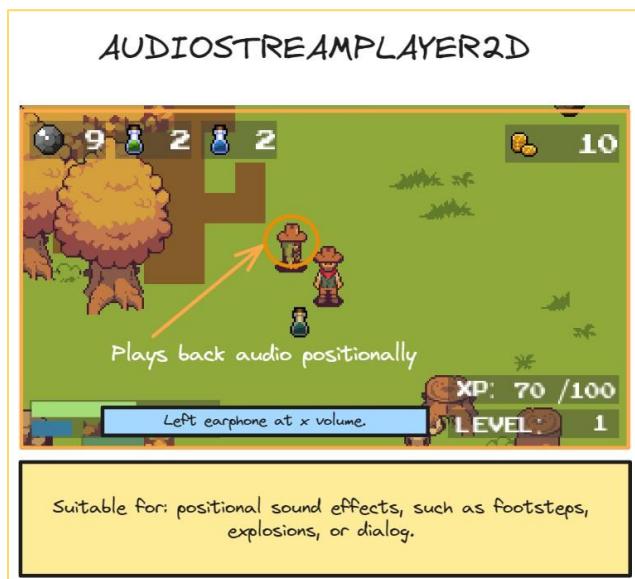
- How to work with the AudioStreamPlayer node.
- How to work with the AudioStreamPlayer2D node.
- How to play, set, and stop audio streams within your code.
- How to loop audio files in the Import Dock.

In Godot, you have three primary options for playing audio:

1. **AudioStreamPlayer:** This is a general audio player suitable for playing music or any non-positional audio. Use this when you don't need the audio to have a position in the world. For example, if you have background music or dialog music that should play irrespective of the position of the characters or objects in the game, use AudioStreamPlayer.
2. **AudioStreamPlayer2D:** This is designed for 2D games where you need positional audio. It stimulates the position of the sound in a 2D space and will make sounds quieter the further away they are from the listener (camera, usually). Use this when you are making a 2D game and you want the audio to have a position in your 2D world (e.g., a sound effect that occurs at a certain location on the screen).
3. **AudioStreamPlayer3D:** Similar to AudioStreamPlayer2D, but for 3D games. It takes into account the position of the sound in three-dimensional space. Use this in 3D

games when you want the sound to emanate from a specific location in the 3D world.

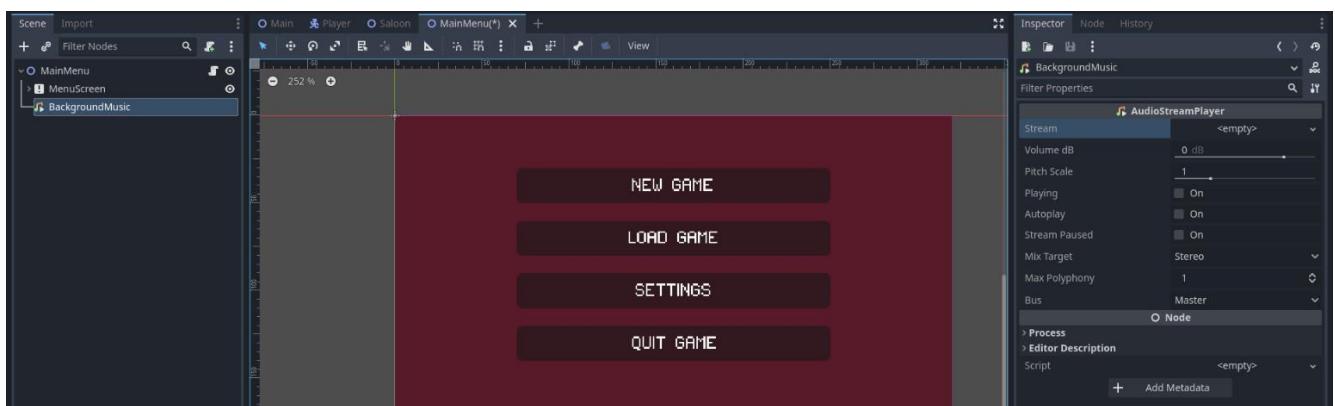
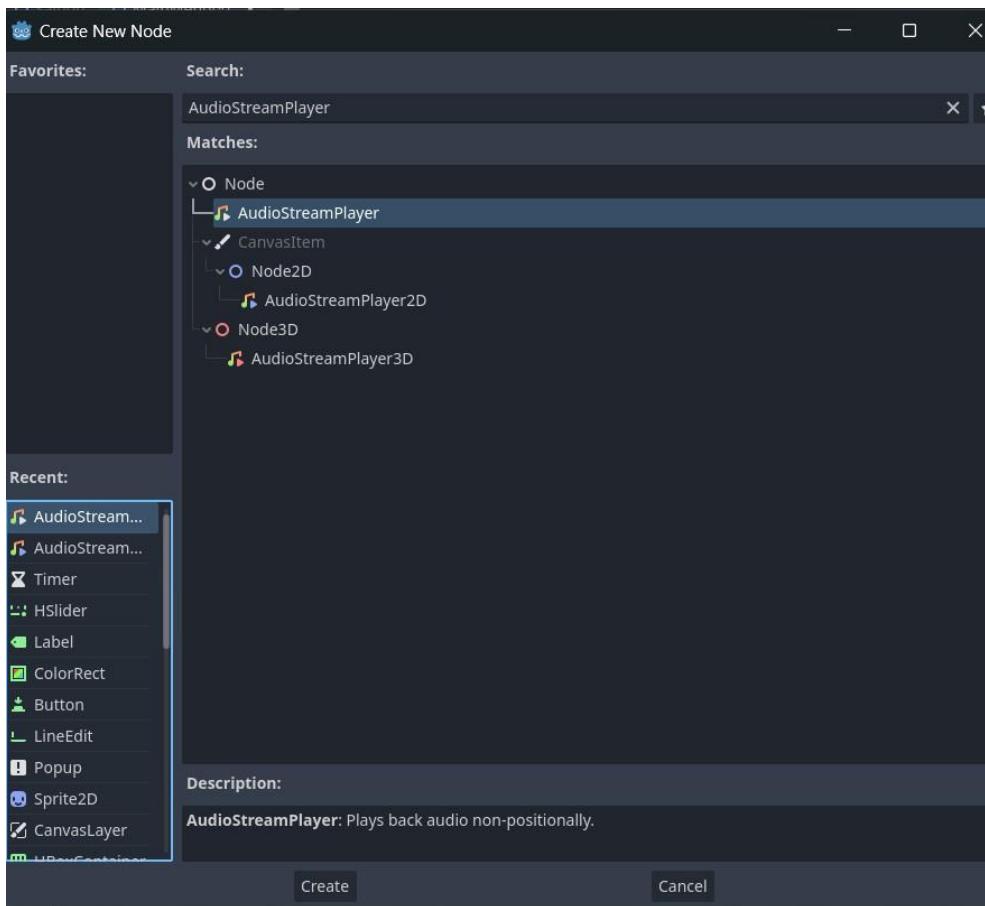
Throughout this tutorial, we will use our [AudioStreamPlayer](#) node for constant stable sounds such as our Background music or Dialog music. We will use our [AudioStreamPlayer2D](#) node however for our sound effects - because we want some panning in the sound and we also want the sound loudness to depend on the location of the sound.



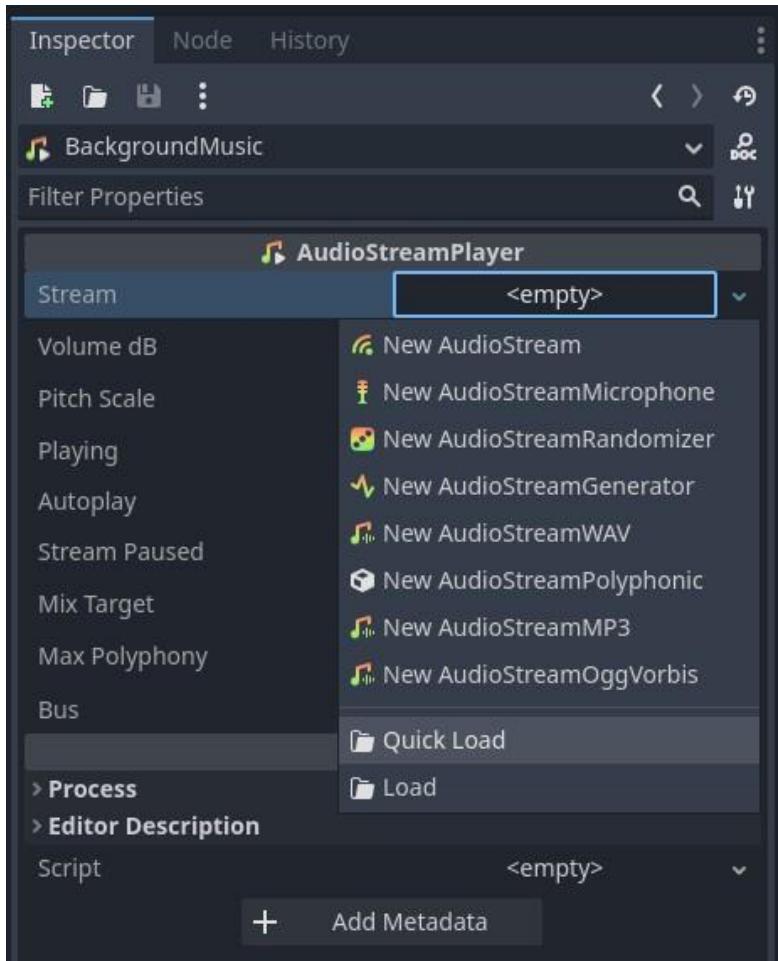
MAIN MENU MUSIC

Open up your MainScene scene, and add a new node called [AudioStreamPlayer](#).

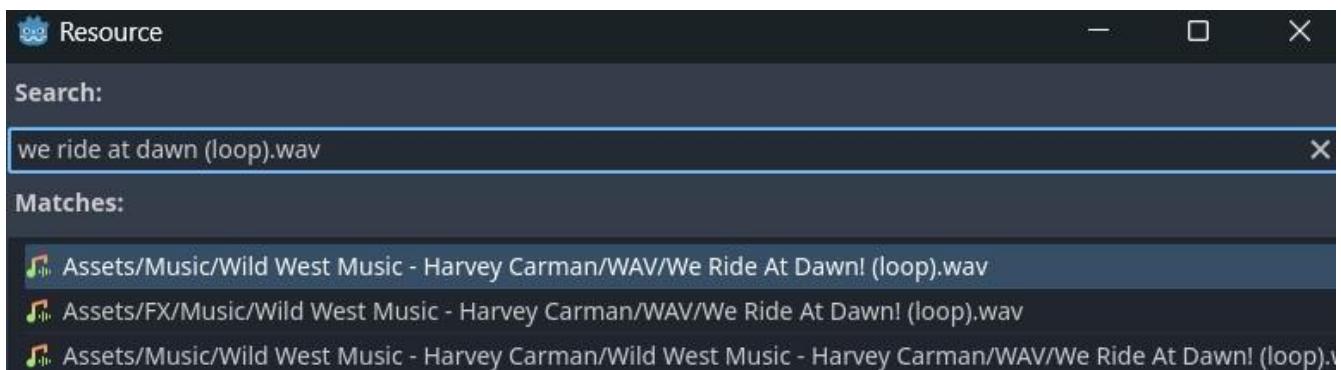
Rename this node to "BackgroundMusic".



In the Inspector panel, we can assign the audio file that should play for this audio player node. Click on <empty> next to your Stream property in your Inspector panel and select "Quick Load". This will set the [AudioStream](#) object to be played.

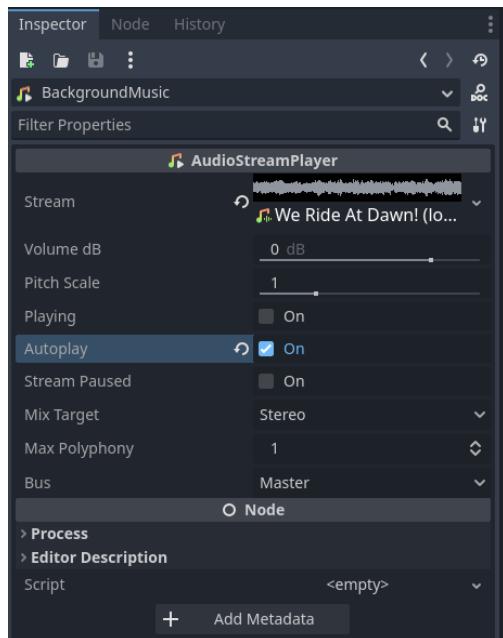


For our background music, we want the audio file "We Ride at Dawn.wav" to play. You can find all the music files underneath your Assets > Music directory.



In the Inspector panel, you can set its playing and auto-playing values. If the playing value is true, the audio is playing or is queued to be played (see [play](#)). If autoplay is true, the audio plays when the scene is loaded. I recommend you read the [documentation](#) to see what the rest of the properties do, such as bus or mix target.

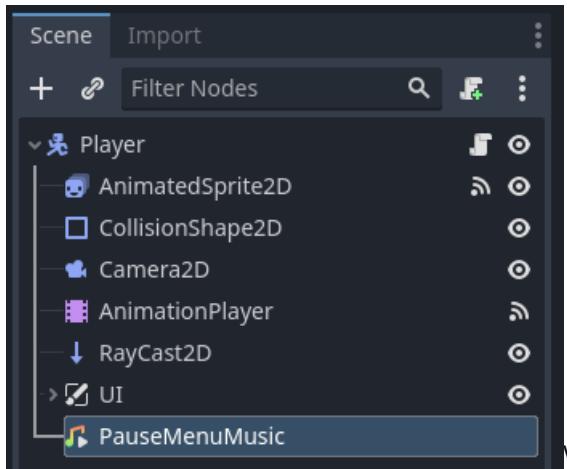
We want this music to play as soon as the game loads, so we need to enable "autoplay". If you want to listen to the music, you can enable "playing", but make sure you disable it afterward!



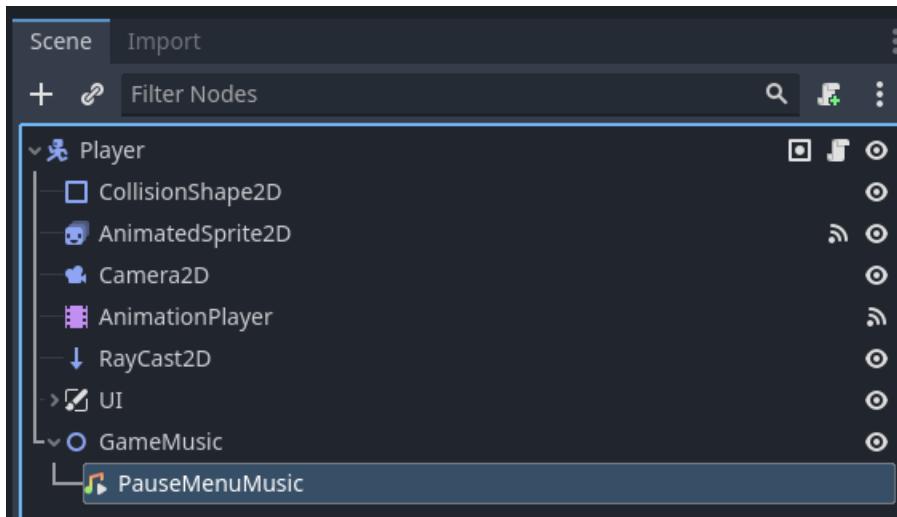
If you now run your scene, your music should play by default but only when you are in your Main Menu scene.

PAUSE MENU

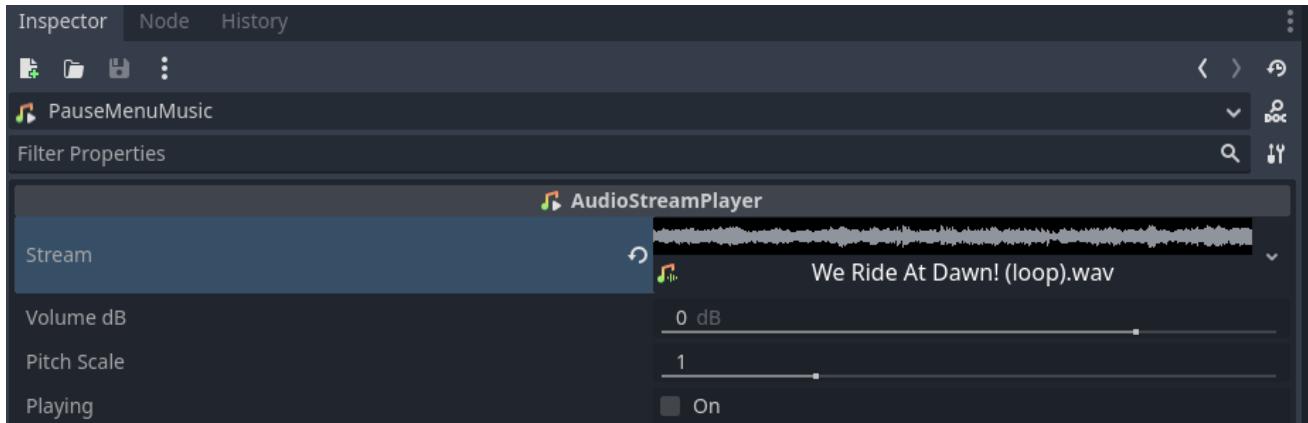
In our Player scene, let's add an AudioStreamPlayer node. Rename it to "PauseMenuMusic".



Let's organize all of our music underneath a Node2D node renamed to "GameMusic".



We also want the audio to be "We Ride at Dawn.wav" when our pause menu is open. Do not enable autoplay or playing, as we will set this node to play in our code only when our pause menu is open.



To play our audio, we simply need to reference our node and then call its [.play\(\)](#) method. This method plays the audio in seconds. We will play this when our game is paused after we've called our *ui_pause* input.

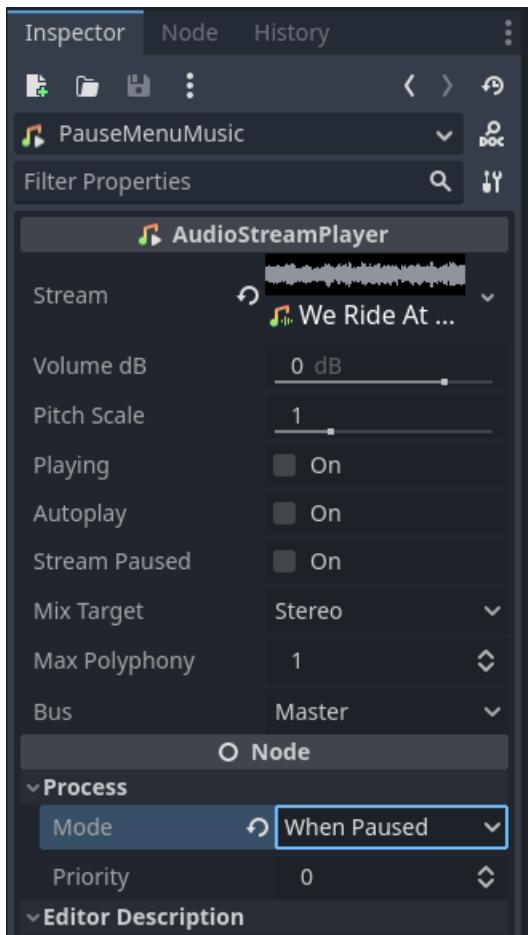
```
### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic

func _input(event):
    # older code
    #show pause menu
    if !pause_screen.visible:
        if event.is_action_pressed("ui_pause"):
            #play music
            pause_menu_music.play()
            #pause game
            Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
            get_tree().paused = true
            #show pause screen popup
            pause_screen.visible = true
            #stops movement processing
            set_physics_process(false)
            #set pauses state to be true
            paused = true
            # if the player is dead, go to back to main menu screen
            if health <= 0:
                get_node("/root/%s" % Global.current_scene_name).queue_free()
                Global.change_scene("res://Scenes/MainScene.tscn")
                get_tree().paused = false
```

```
return
```

We also need to set our PauseMenuMusic node's process mode to "When Paused", because we only want this to process when our game is in the paused state.



We also need to stop our music from playing when we quit our scene or resume our game - otherwise, it will play over the other audio. We can do this via the `stop()` method.

```
# ----- Pause Menu -----
#resume game
func _on_resume_pressed():
    #hide pause menu
    pause_screen.visible = false
    #set pauses state to be false
    get_tree().paused = false
    paused = false
    #accept movement and input
```

```

set_process_input(true)
set_physics_process(true)
Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)
#stop music
pause_menu_music.stop()

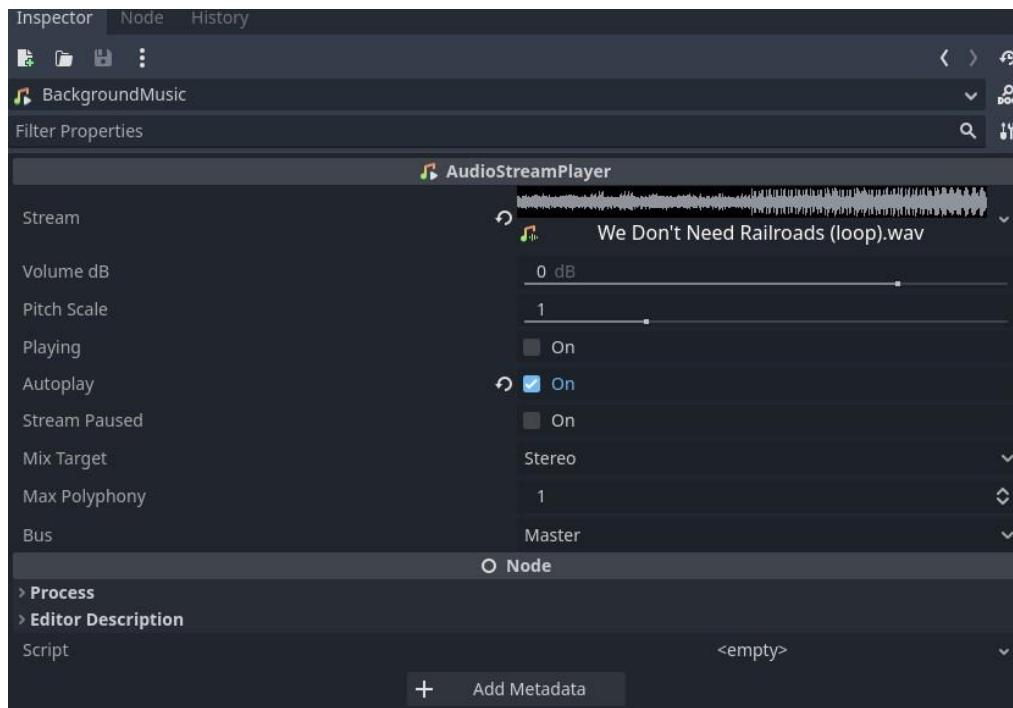
```

Now if you run your scene, and you pause/unpause, your pause music should play correctly.

BACKGROUND MUSIC

We also need music to play during the game loop. We will attach this node to our Player scene because we want this sound to follow them around. Let's add another AudioStreamPlayer node to our Player scene and call it "BackgroundMusic".

We want this audio track to be "We Don't Need Railroads.wav". Also, enable autoplay because we want this audio track to play by default.



We need to stop this music from playing when our pause menu is open, so update your `ui_pause` input to stop the background music audio track.

```

### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic

func _input(event):
    # older code
    #show pause menu
    if !pause_screen.visible:
        if event.is_action_pressed("ui_pause"):
            #pause game
            Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
            get_tree().paused = true
            #show pause screen popup
            pause_screen.visible = true
            #stops movement processing
            set_physics_process(false)
            #set pauses state to be true
            paused = true
            #play music
            background_music.stop()
            pause_menu_music.play()
            # if the player is dead, go to back to main menu screen
            if health <= 0:
                get_node("/root/%s" % Global.current_scene_name).queue_free()
                Global.change_scene("res://Scenes/MainScene.tscn")
                get_tree().paused = false
            return

```

Our background music should also play after we've pressed our confirm and resume buttons.

```

### Player.gd

# close popup
func _on_confirm_pressed():
    level_popup.visible = false
    get_tree().paused = false
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)
    background_music.play()

```

```

# ----- Pause Menu -----
#resume game
func _on_resume_pressed():
    #hide pause menu
    pause_screen.visible = false
    #set pauses state to be false
    get_tree().paused = false
    paused = false
    #accept movement and input
    set_process_input(true)
    set_physics_process(true)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)
    #stop music
    pause_menu_music.stop()
    background_music.play()

```

If our game is over, we will play another sound - so we also need to stop our background music to play in our death conditional.

```

### Player.gd

# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
    health_updated.emit(health, max_health)
    if health > 0:
        #damage
        animation_player.play("damage")
        health_updated.emit(health, max_health)
    else:
        #stop background music
        background_music.stop()
        #death
        set_process(false)
        get_tree().paused = true
        paused = true
        animation_player.play("game_over")

```

The same goes for our *update_xp* function.

```
### Player.gd

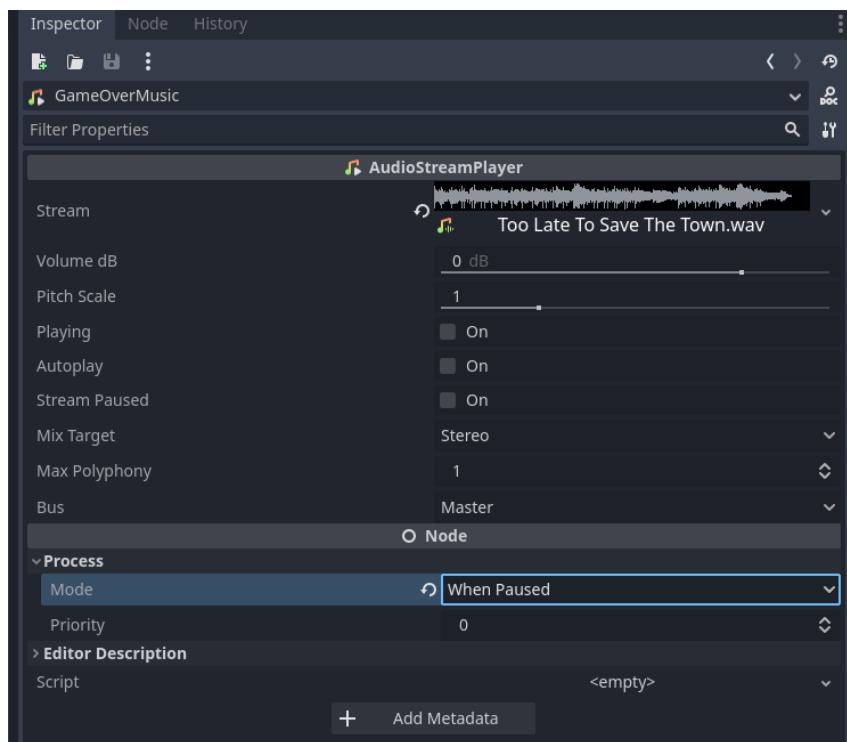
# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        #stop background music
        background_music.stop()
```

If you run your scene, your music should play when you're in the game.

GAME OVER MUSIC

When our player dies, we also want to play our GameOverMusic. For this, we need to add another AudioStreamPlayer node to our Player scene.

We want this audio track to be "Too Late To Save The Town.wav". Also, change its processing mode to "When Paused".



Update your `hit()` function to play the audio after the background music has been stopped.

```
### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic
@onready var game_over_music = $GameMusic/GameOverMusic

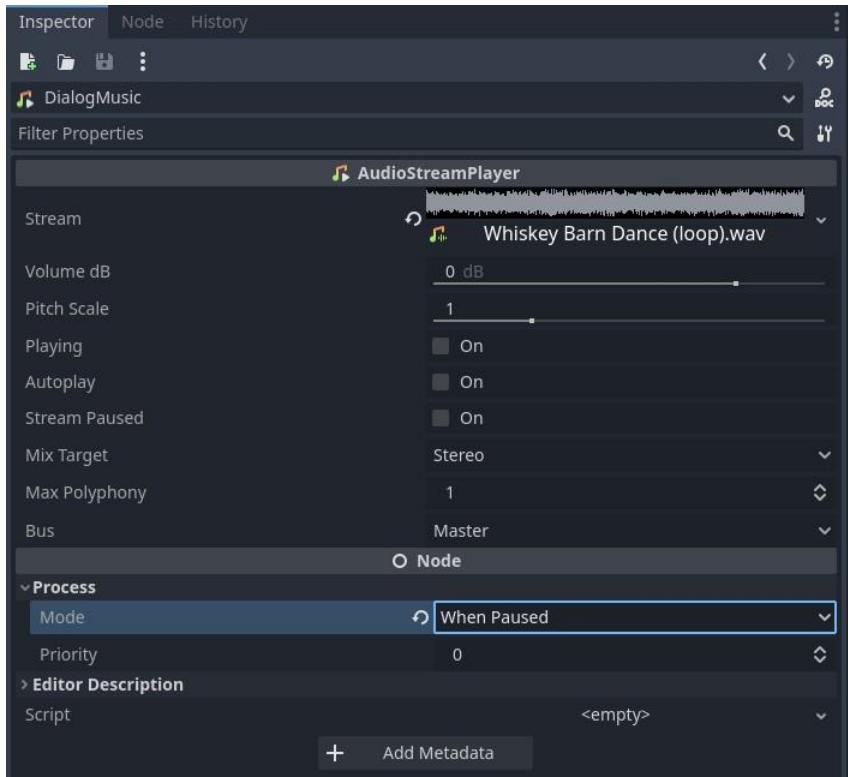
# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
    health_updated.emit(health, max_health)
    if health > 0:
        #damage
        animation_player.play("damage")
        health_updated.emit(health, max_health)
    else:
        #death
        set_process(false)
        get_tree().paused = true
        paused = true
        animation_player.play("game_over")
        #stop background music
        background_music.stop()
        game_over_music.play()
```

If you run your scene, your music should play when your player dies.

DIALOG MUSIC

We'll have to set our dialog music in two places, which are in our Player and DialogPopup scripts. If our player interacts with the NPC, the dialog music should play - and if the player is done interacting and our popup closes, our background music should play.

Add a new AnimationPlayer node called "DialogMusic" with the audio track "Whiskey Barn Dance (loop).wav". Set its processing mode to "When Paused". You can also use the "Imposter Syndrome (wav)" audio for this.



We're playing the audio in the Player script because if we had to do it in our DialogPopup's `open()` function the music would restart each time the popup changes! Play and stop the Dialog Music as follows:

```
### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic
@onready var game_over_music = $GameMusic/GameOverMusic
@onready var dialog_music = $GameMusic/DialogMusic

# ----- Damage & Death -----
#does damage to our player
func _input(event):
```

```

# older code
#interact with world
elif event.is_action_pressed("ui_interact"):
    var target = ray_cast.get.collider()
    if target != null:
        if target.is_in_group("NPC"):
            # Talk to NPC
            target.dialog()
            # Music
            background_music.stop()
            dialog_music.play()
            return

```

```

### DialogPopup.gd

extends CanvasLayer

# Node refs
@onready var animation_player = $"../../AnimationPlayer"
@onready var player = $"../../"
@onready var background_music = $"../../GameMusic/BackgroundMusic"
@onready var dialog_music = $"../../GameMusic/DialogMusic"

#closes the dialog
func close():
    get_tree().paused = false
    self.visible = false
    player.set_physics_process(true)
    Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)
    # Music
    dialog_music.stop()
    background_music.play()

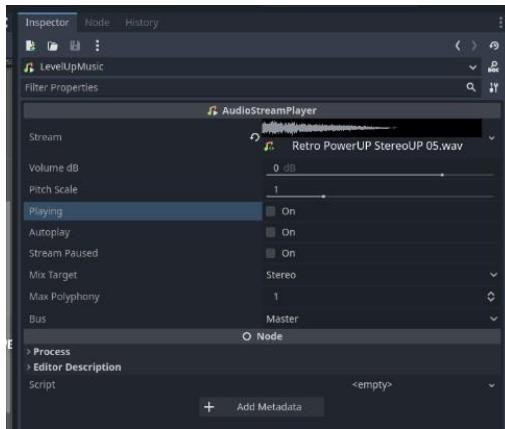
```

If you now run your scene, your dialog music should play when your player interacts with the NPC.

LEVEL UP MUSIC

When our player level's up, we want to play a little victory tune. We will still use the AnimationPlayer node for this because we want the audio noise to come from a central point - we don't want it to pan between our left and right ear.

Add a new node and call it "LevelUpMusic". We want the audio to be "Retro PowerUP StereoUP 05.wav".



Now, in our `update_xp()` function, we want to play our LevelUpMusic.

```
### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic
@onready var game_over_music = $GameMusic/GameOverMusic
@onready var dialog_music = $GameMusic/DialogMusic
@onready var level_up_music = $GameMusic/LevelUpMusic

# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        #stop background music
        background_music.stop()
```

level_up_music.play()

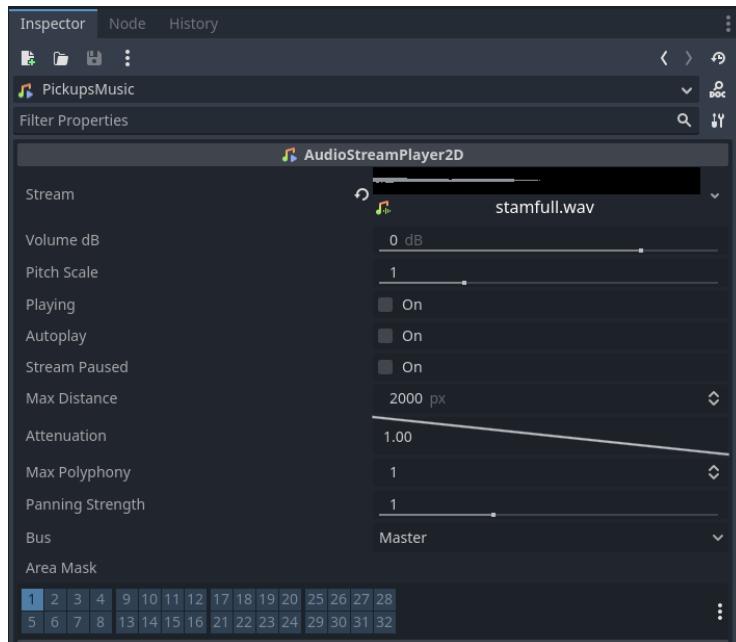
If you run your scene, your LevelUpMusic music should play when your player levels up after completing quests and shooting enemies.

PICKUPS SOUND EFFECT

If our player runs over some ammo or a drink, or even our quest items, we want to play a pickup sound effect. For this, we will use an [AudioStreamPlayer2D](#) node, since we want some audio attenuation for our sound effects. It gives the sound a bit more of a realistic feel, as it plays more like environmental background noise.

You can attach this node to something like a fire scene, which, depending on how far/close you are from the fire, the sound volume will differ. Since we'll be adding this sound to our Player, the sound won't attenuate much since our camera always focuses on our player.

Add a new AudioStreamPlayer2D node and call it "PickupsMusic". Set its audio file to "stamfull.wav".



In our code, we need to play this audio when our player runs over a pickup. We can do this in our `add_pickup()` function.

```
### Player.gd

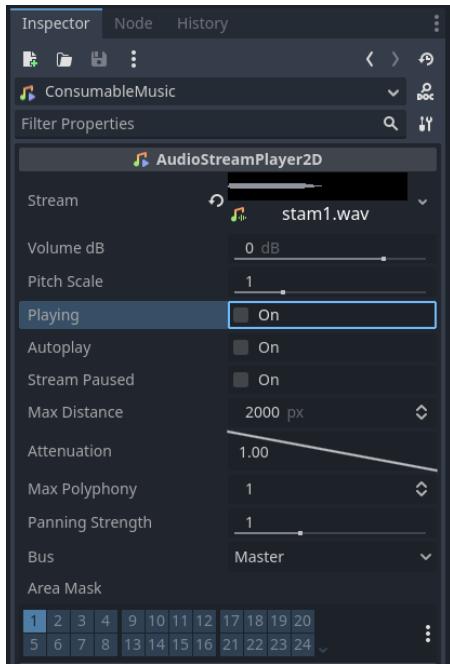
# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic
@onready var game_over_music = $GameMusic/GameOverMusic
@onready var dialog_music = $GameMusic/DialogMusic
@onready var level_up_music = $GameMusic/LevelUpMusic
@onready var pickups_sfx = $GameMusic/PickupsMusic

# ----- Consumables -----
# Add the pickup to our GUI-based inventory
func add_pickup(item):
    if item == Global.Pickups.AMMO:
        ammo_pickup = ammo_pickup + 3 # + 3 bullets
        ammo_pickups_updated.emit(ammo_pickup)
        print("ammo val:" + str(ammo_pickup))
    if item == Global.Pickups.HEALTH:
        health_pickup = health_pickup + 1 # + 1 health drink
        health_pickups_updated.emit(health_pickup)
        print("health val:" + str(health_pickup))
    if item == Global.Pickups.STAMINA:
        stamina_pickup = stamina_pickup + 1 # + 1 stamina drink
        stamina_pickups_updated.emit(stamina_pickup)
        print("stamina val:" + str(stamina_pickup))
    # SFX
    pickups_sfx.play()
    update_xp(5)
```

Now if you run over your pickups items, the audio should play!

CONSUMING SOUND EFFECT

We want a sound effect to play each time our player consumes a health or stamina drink by pressing "1" or "2". Add another AudioStreamPlayer2D node and call it "ConsumableMusic". Set its audio file to "stam1.wav".



In our input code, let's update our `ui_consume_health` and `ui_consume_stamina` inputs to play our consumable sound effects. You can also make both of these different audio streams by loading a new stream resource into your `ConsumableMusic` node before playing it.

```
### Player.gd

# Audio nodes
@onready var pause_menu_music = $GameMusic/PauseMenuMusic
@onready var background_music = $GameMusic/BackgroundMusic
@onready var game_over_music = $GameMusic/GameOverMusic
@onready var dialog_music = $GameMusic/DialogMusic
@onready var level_up_music = $GameMusic/LevelUpMusic
@onready var pickups_sfx = $GameMusic/PickupsMusic
@onready var consume_sfx = $GameMusic/ConsumableMusic

func _input(event):
    # older code
    #using health consumables
    elif event.is_action_pressed("ui_consume_health"):
        if health > 0 && health_pickup > 0:
            health_pickup = health_pickup - 1
            health = min(health + 50, max_health)
```

```

    health_updated.emit(health, max_health)
    health_pickups_updated.emit(health_pickup)
    # SFX
    consume_sfx.play()
#using stamina consumables
elif event.is_action_pressed("ui_consume_stamina"):
    if stamina > 0 && stamina_pickup > 0:
        stamina_pickup = stamina_pickup - 1
        stamina = min(stamina + 50, max_stamina)
        stamina_updated.emit(stamina, max_stamina)
        stamina_pickups_updated.emit(stamina_pickup)
    # SFX
    consume_sfx.stream = load("res://Assets/FX/Music/Free Retro SFX by
@inertsongs/SFX/stam0.wav")
    consume_sfx.play()

```

Now if you run over your pickups items, and consume them, the audio should play!

BULLET IMPACT SOUND EFFECT

When a bullet hits our Player or Enemy, we want the bullet impact sound to play. We will add this audio in our Enemy and Player scenes. Let's play the "Retro Impact LoFi 09.wav" sound using the `AudioStreamPlayer2D` node which we'll rename as "`BulletImpactMusic`". Add this node to both your Enemy and Player scenes.

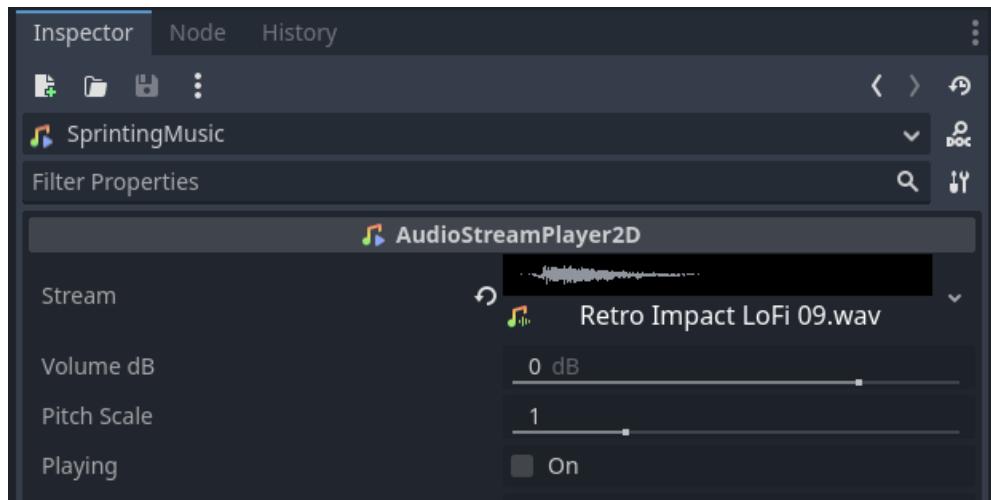
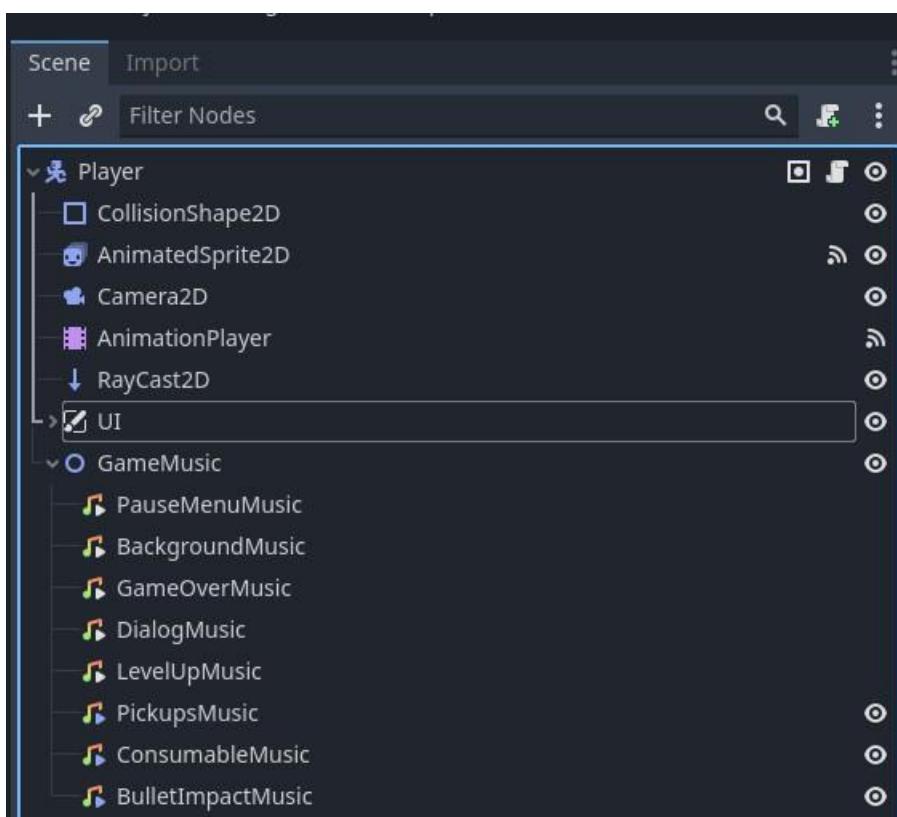
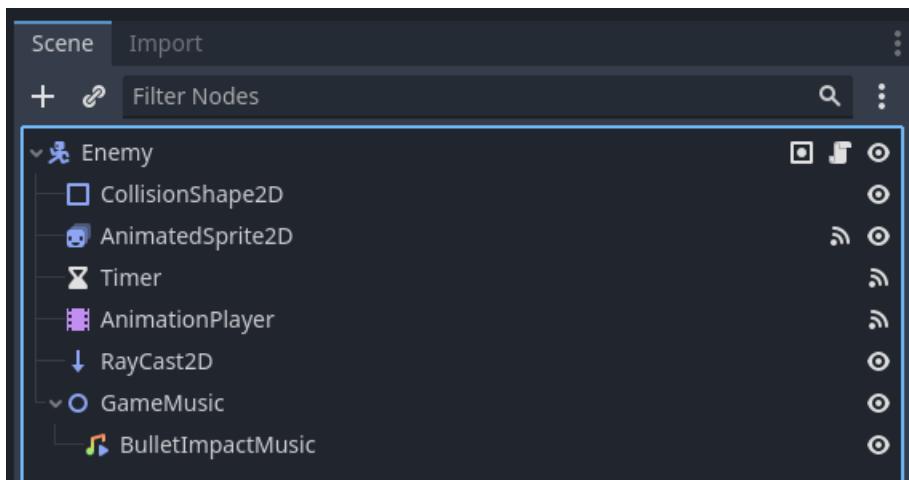


Figure 18: *BulletImpactMusic not SprintingMusic



After we impact with our node, let's play this sound.

```
### Enemy.gd

# Audio nodes
@onready var bullet_sfx = $GameMusic/BulletImpactMusic

#will damage the enemy when they get hit
```

```

func hit(damage):
    health -= damage
    if health > 0:
        #damage
        animation_player.play("damage")
        # SFX
        bullet_sfx.play()

### Player.gd

# Audio nodes
@ready var pause_menu_music = $GameMusic/PauseMenuMusic
@ready var background_music = $GameMusic/BackgroundMusic
@ready var game_over_music = $GameMusic/GameOverMusic
@ready var dialog_music = $GameMusic/DialogMusic
@ready var level_up_music = $GameMusic/LevelUpMusic
@ready var pickups_sfx = $GameMusic/PickupsMusic
@ready var consume_sfx = $GameMusic/ConsumableMusic
@ready var bullet_sfx = $GameMusic/BulletImpactMusic

# ----- Damage & Death -----
#does damage to our player
func hit(damage):
    health -= damage
    health_updated.emit(health, max_health)
    if health > 0:
        #damage
        animation_player.play("damage")
        health_updated.emit(health, max_health)
        # SFX
        bullet_sfx.play()

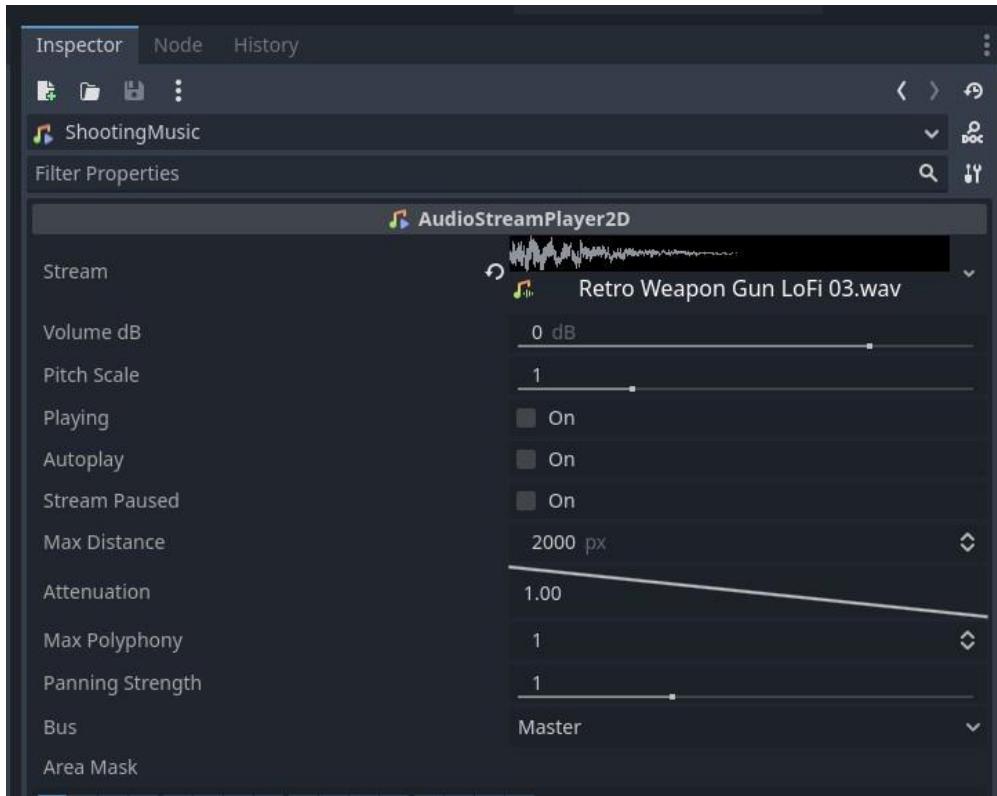
```

Now if we run our scene and we hit the enemy or the enemy hits us with a bullet, our sound effect should play.

SHOOTING SOUND EFFECT

When we shoot our weapons, we also want our guns to play a shooting sound effect. We'll do this for both our Enemy and our Player.

Let's play the "Retro Weapon Gun LoFi 03.wav" sound using the `AudioStreamPlayer2D` node which we'll rename as "ShootingMusic". Add this node to both your Enemy and Player scenes.



Let's play this sound effect in our `ui_attack` input.

```
### Player.gd

# Audio nodes
@ready var pause_menu_music = $GameMusic/PauseMenuMusic
@ready var background_music = $GameMusic/BackgroundMusic
@ready var game_over_music = $GameMusic/GameOverMusic
@ready var dialog_music = $GameMusic/DialogMusic
@ready var level_up_music = $GameMusic/LevelUpMusic
@ready var pickups_sfx = $GameMusic/PickupsMusic
@ready var consume_sfx = $GameMusic/ConsumableMusic
@ready var bullet_sfx = $GameMusic/BulletImpactMusic
@ready var shooting_sfx = $GameMusic/ShootingMusic

func _input(event):
    #input event for our attacking, i.e. our shooting
```

```

if event.is_action_pressed("ui_attack"):
    #checks the current time as the amount of time
    var now = Time.get_ticks_msec()
    #check if player can shoot if the reload time has passed and we have ammo
    if now >= bullet_fired_time and ammo_pickup > 0:
        #SFX
        shooting_sfx.play()
        #shooting anim
        is_attacking = true
        var animation = "attack_" + returned_direction(new_direction)
        animation_sprite.play(animation)
        #bullet fired time to current time
        bullet_fired_time = now + bullet_reload_time
        #reduce and signal ammo change
        ammo_pickup = ammo_pickup - 1
        ammo_pickups_updated.emit(ammo_pickup)

```

For our enemy, we'll need to update its *process()* function to also have a reload time so that it doesn't loop the audio excessively together. We did this in the Player script.

```

### Enemy.gd

# Audio nodes
@onready var bullet_sfx = $GameMusic/BulletImpactMusic
@onready var shooting_sfx = $GameMusic/ShootingMusic


func _process(delta):
    #regenerates our enemy's health
    health = min(health + health_regen * delta, max_health)
    #checks the current time as the amount of time passed
    var now = Time.get_ticks_msec()
    #check if enemy can shoot
    if now >= bullet_fired_time:
        # What's the target?
        var target = $RayCast2D.get.collider()
        if target != null:
            if target.name == "Player" and player.health > 0:
                # SFX
                shooting_sfx.play()
                #shooting anim
                is_attacking = true
                var animation = "attack_" + returned_direction(new_direction)

```

```

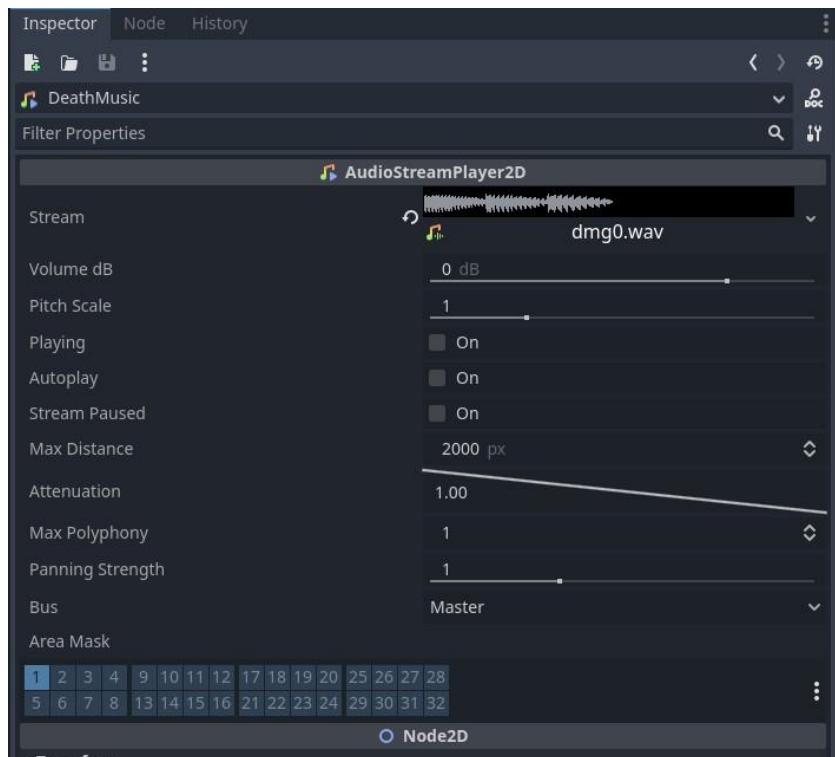
        animation_sprite.play(animation)
        #reload time to bullet fired time
        bullet_fired_time = now + bullet_reload_time
    
```

Now if you run your scene, your player's shooting sound should play when you press CTRL, and the enemy's shooting sound should play when they attack you.

ENEMY DEATH SOUND EFFECT

Finally, we also want to play a sound effect when our enemy dies.

Add a new AudioStreamPlayer2D node to your EnemySpawner scene and call it "Death Music". The audio file should be "dmg0.wav".



We will play this sound effect when our spawner decreases our enemy count by 1.

```

###EnemySpawner.gd

# Audio nodes
@onready var death_sfx = $GameMusic/DeathMusic
    
```

```
# Remove enemy
func _on_enemy_death():
    enemy_count = enemy_count - 1
    death_sfx.play()
```

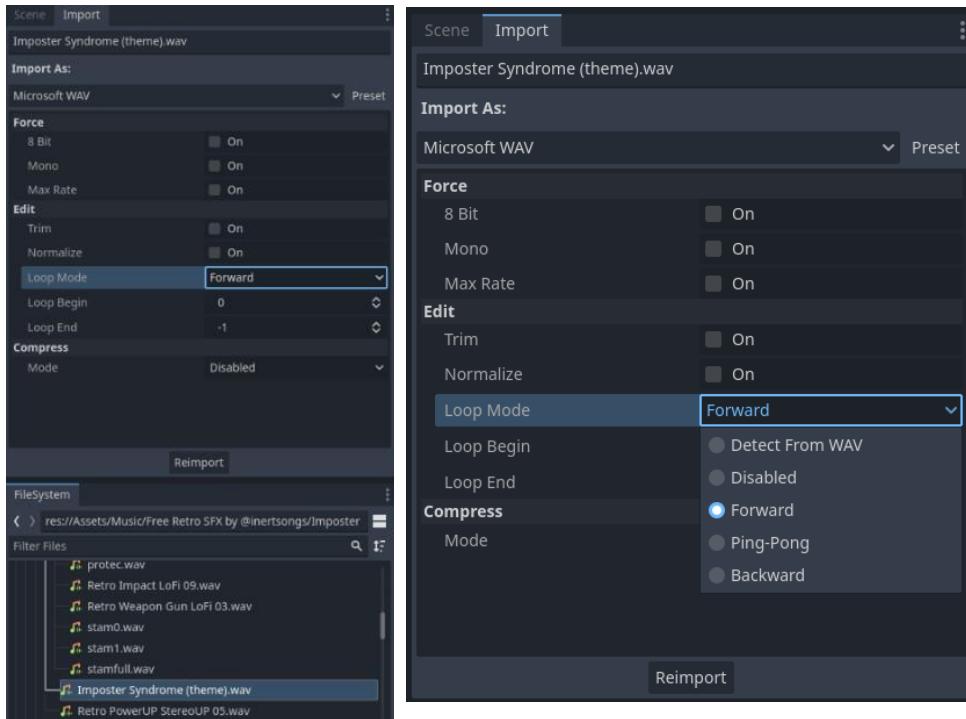
If you run your scene and kill the enemies, the sound effect should play.

PLAYING DIFFERENT MUSIC IN OUR MAIN SCENES

In our Main_2 scene, we want to play our "Imposter Syndrome (wav)" audio track instead of the Background music we assigned to our Player. To do this, we can simply re-assign our stream resource and then play the node again.

Before we do this, we have to re-import our "Imposter Syndrome (wav)" file to be a looped audio file. If we don't do this, it plays and then stops when it finishes. To re-import it, click on it and open your Import Dock. Set its import mode to "Forward", and then re-import it.

You'll notice that we haven't imported any of our other audio files with looping enabled, and that is because the default import settings for WAV files in Godot 4 are set to recognize and utilize looping, while MP3 settings are not. Thus, if we were using the audio files with a .mp3 extension, we would've imported them to enable looping, but since we were using .wav audio files, the Godot engine automatically recognized which audios should loop.



```
### Main_2.gd

extends Node2D

@onready var background_music = $Player/GameMusic/BackgroundMusic

#connect signal to function
func _ready():
    background_music.stream = load("res://Assets/FX/Music/Free Retro SFX by
    @inertsongs/Imposter Syndrome (theme).wav")
    background_music.play()

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main.tscn")
        Global.scene_changed.connect(_on_scene_changed)

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()
```

Now if you run back and forth between your different scenes, your music should change.

In the next part, we will convert our TileMap node into a TileMapLayer node. Remember to save and make a backup, and I'll see you in the next part.

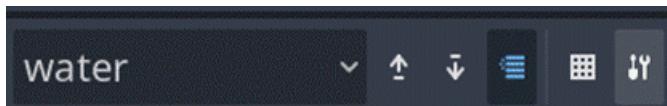
The final source code for this part should look like [this](#).

PART 23: TILEMAPLAYERS CONVERSION

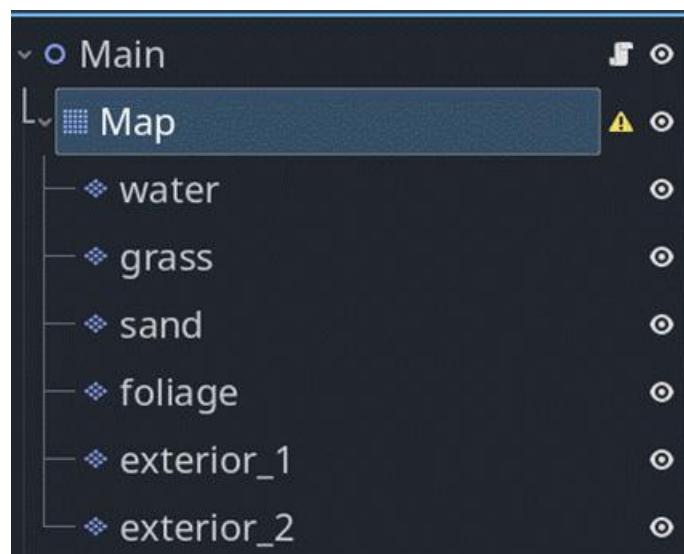
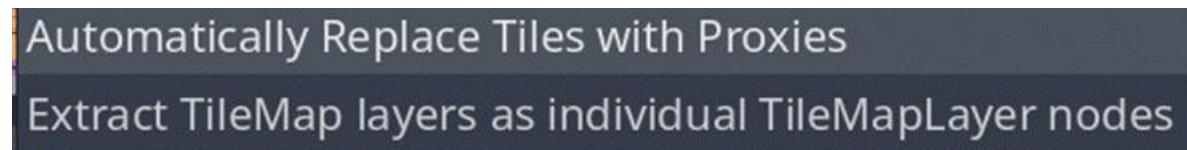
Originally when I wrote this tutorial booklet, the TileMap node was the main way of creating 2D maps in Godot. Recently, as of Godot 4.3, the TileMap node has been replaced by the TileMapLayer node, which is exactly the same as the TileMap node – but this node can only handle one layer at a time.

CONVERSION

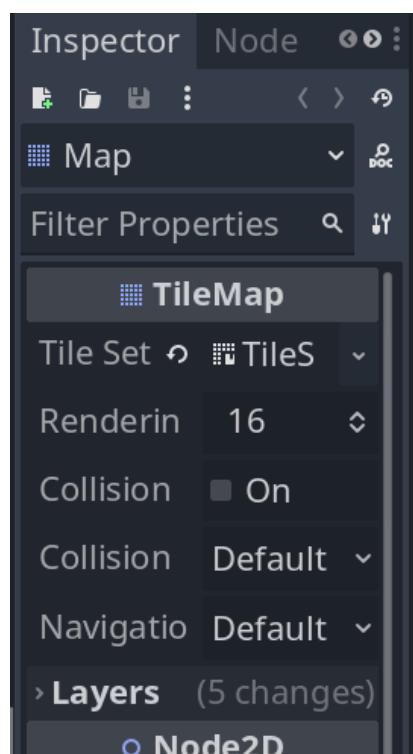
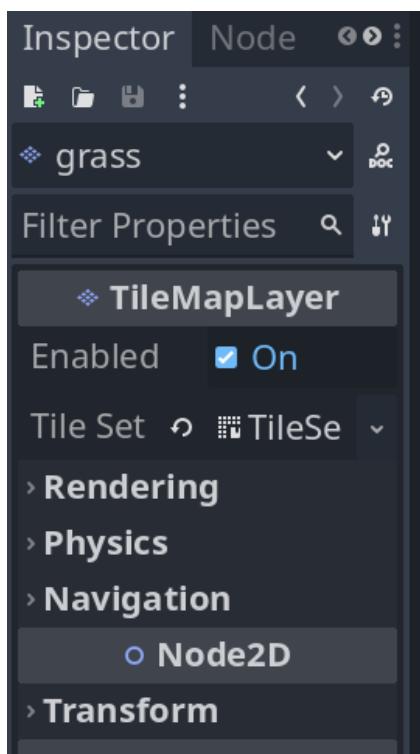
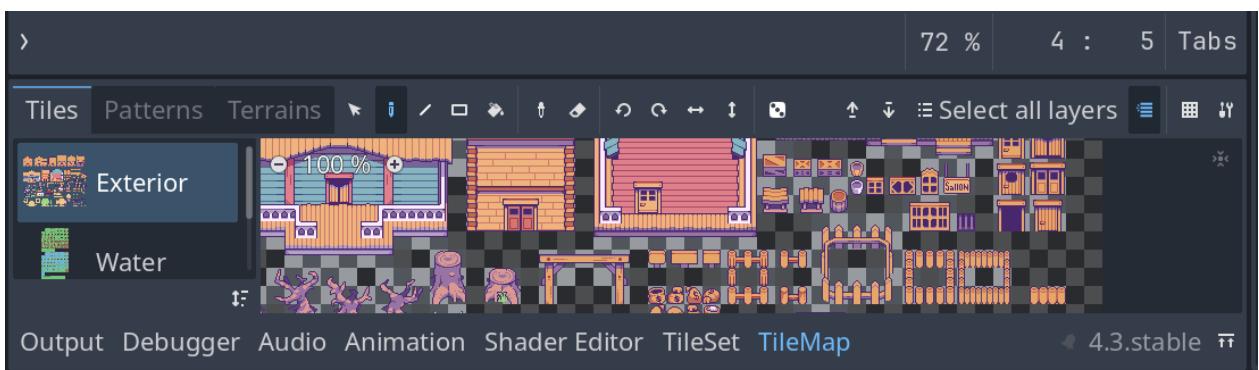
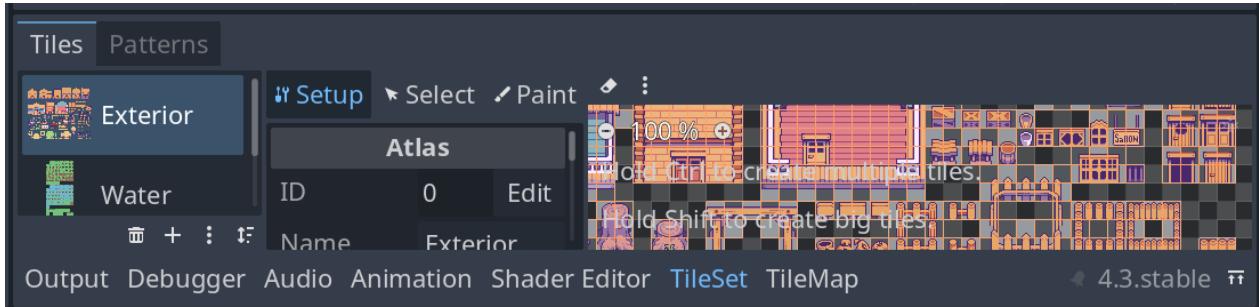
To convert our TileMap node into a TileMapLayer node, we need to navigate to the TileMap panel and click on the tools icon.



Then select the option to “Extract TileMap layers as individual TileMapLayer nodes”. This will convert your TileMap node into TileMapLayer nodes, which are like individual TileMap nodes with separate layers.



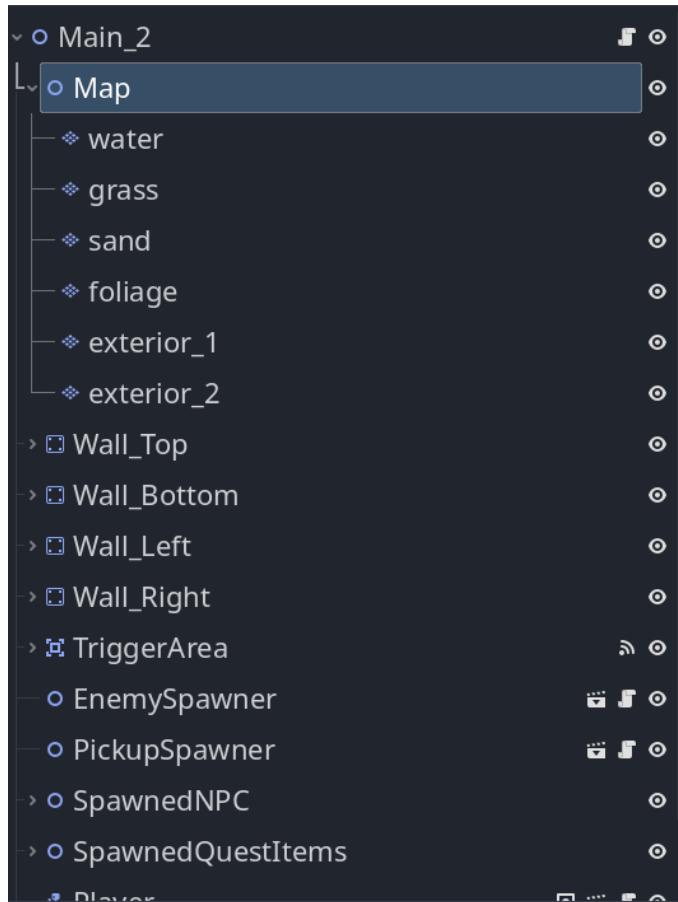
You'll see your TileMap node has been populated with a bunch of TileMapLayer nodes. If you select one of these nodes, you will see that they are exactly the same as the TileMap node. The only difference is that they don't have a "Layers" option in the Inspector Panel.



You can then replace your “Map” node with a Node2D node to hold all of your TileMapLayers.

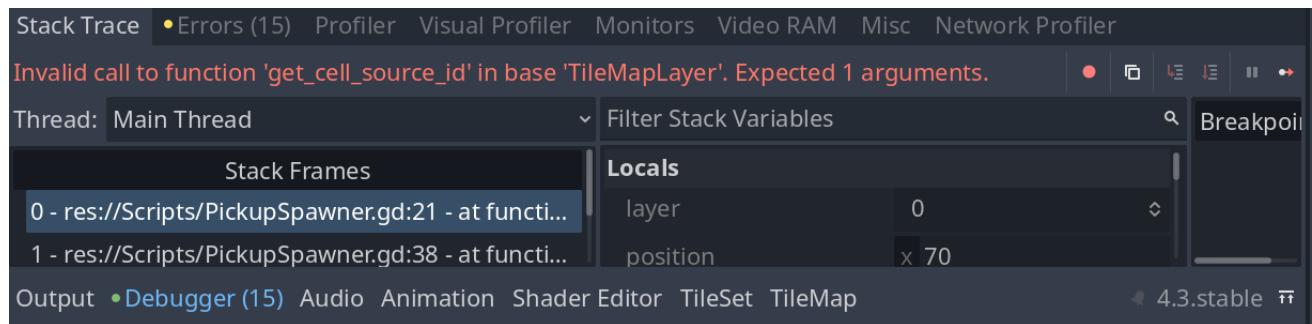


We need to do the same for our second scene.



CODE FIXES

If you tried to run your game now, your entire project might seem broken.



This is because our “Map” node is now a Node2D node, and no longer a TileMap node. To fix this, we will need to replace our Map references with the layers that we want to access. So, let’s start from the top – our Global script. We will need to remove the layer constants that we created previously because we can no longer access these.

```
### Global.gd
extends Node

# Scene resources
@onready var pickups_scene = preload("res://Scenes/Pickup.tscn")
@onready var enemy_scene = preload("res://Scenes/Enemy.tscn")
@onready var bullet_scene = preload("res://Scenes/Bullet.tscn")
@onready var enemy_bullet_scene = preload("res://Scenes/EnemyBullet.tscn")

# Pickups
enum Pickups { AMMO, STAMINA, HEALTH }

# current scene
var current_scene_name

#notifies scene change
signal scene_changed()

var save_path = "user://dusty_trails_save.json"
var loading = false

func _ready():
    current_scene_name = get_tree().get_current_scene().name

# older code
```

Now in all of our other scripts where we referenced these layers, we need to call them directly. We will update our spawners to spawn our enemies and pickups on the grass layers, and then wherever we referenced the layers, we will replace these references with the actual layer node instead.

So, in our Bullet scripts, let's get references to our TileMapLayer nodes, and then we will call these layers whenever we want to block/allow our bullets on the map.

```
### Bullets.gd
extends Area2D

# Node refs
@onready var animated_sprite = $AnimatedSprite2D

@onready var water = get_tree().get_root().get_node("%s/Map/water" %
    Global.current_scene_name)
@onready var grass = get_tree().get_root().get_node("%s/Map/grass" %
    Global.current_scene_name)
@onready var sand = get_tree().get_root().get_node("%s/Map/sand" %
    Global.current_scene_name)
@onready var foliage = get_tree().get_root().get_node("%s/Map/foliage" %
    Global.current_scene_name)
@onready var exterior_1 = get_tree().get_root().get_node("%s/Map/exterior_1" %
    Global.current_scene_name)
@onready var exterior_2 = get_tree().get_root().get_node("%s/Map/exterior_2" %
    Global.current_scene_name)

# Bullet variables
var speed = 80
var direction : Vector2
var damage

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Player
    if body.is_in_group("player"):
        return
    # Ignore collision with Water
    if body.name == "Map":
```

```

#water == Layer 0
if water:
    return
# trees, buildings and objects
if foliage or exterior_1 or exterior_2:
    animated_sprite.play("impact")
# If the bullets hit an enemy, damage them
if body.is_in_group("enemy"):
    body.hit(damage)
# Stop the movement and explode
direction = Vector2.ZERO
animated_sprite.play("impact")

# Remove
func _on_animated_sprite_2d_animation_finished():
    if animated_sprite.animation == "impact":
        get_tree().queue_delete(self)

# Self-destruct
func _on_timer_timeout():
    animated_sprite.play("impact")

```

```

### EnemyBullet.gd
extends Area2D

# Node refs
@onready var tilemap = get_tree().root.get_node("%s/Map" %
Global.current_scene_name)
@onready var animated_sprite = $AnimatedSprite2D

@onready var water = get_tree().get_root().get_node("%s/Map/water" %
Global.current_scene_name)
@onready var grass = get_tree().get_root().get_node("%s/Map/grass" %
Global.current_scene_name)
@onready var sand = get_tree().get_root().get_node("%s/Map/sand" %
Global.current_scene_name)
@onready var foliage = get_tree().get_root().get_node("%s/Map/foliage" %
Global.current_scene_name)
@onready var exterior_1 = get_tree().get_root().get_node("%s/Map/exterior_1" %
Global.current_scene_name)
@onready var exterior_2 = get_tree().get_root().get_node("%s/Map/exterior_2" %
Global.current_scene_name)

```

```

# Bullet variables
var speed = 80
var direction : Vector2
var damage

# ----- Bullet -----
# Position
func _process(delta):
    position = position + speed * delta * direction

# Collision
func _on_body_entered(body):
    # Ignore collision with Enemy
    if body.is_in_group("enemy"):
        return
    # Ignore collision with Water
    if body.name == "Map":
        #water == Layer 0
        if water:
            return
        # trees, buildings and objects
        if foliage or exterior_1 or exterior_2:
            animated_sprite.play("impact")
    # If the bullets hit player, damage them
    if body.is_in_group("player"):
        body.hit(damage)
    # Stop the movement and explode
    direction = Vector2.ZERO
    animated_sprite.play("impact")

# Remove
func _on_animated_sprite_2d_animation_finished():
    if animated_sprite.animation == "impact":
        get_tree().queue_delete(self)

# Self-destruct
func _on_timer_timeout():
    animated_sprite.play("impact")

```

Then, in our spawners (enemy and pickups), let's get references to our TileMapLayer nodes, and then we will spawn these entities on our Grass layer. You can also create a dedicated layer for these entities – such as SpawnLayer, and then just draw the tiles that you want these entities to spawn on-on that layer.

```

#### EnemySpawner.gd

extends Node2D

# Node refs
@onready var player = get_tree().root.get_node("%s/Player" %
Global.current_scene_name)
@onready var spawned_enemies = $SpawnedEnemies

@onready var water = get_tree().get_root().get_node("%s/Map/water" %
Global.current_scene_name)
@onready var grass = get_tree().get_root().get_node("%s/Map/grass" %
Global.current_scene_name)
@onready var sand = get_tree().get_root().get_node("%s/Map/sand" %
Global.current_scene_name)
@onready var foliage = get_tree().get_root().get_node("%s/Map/foliage" %
Global.current_scene_name)
@onready var exterior_1 = get_tree().get_root().get_node("%s/Map/exterior_1" %
Global.current_scene_name)
@onready var exterior_2 = get_tree().get_root().get_node("%s/Map/exterior_2" %
Global.current_scene_name)

# Audio nodes
@onready var death_sfx = $GameMusic/DeathMusic

# Enemy stats
@export var max_enemies = 9
var enemy_count = 0
var rng = RandomNumberGenerator.new()

# Inside the _ready() function
func _ready():
    player.leveled_up.connect(_on_player_leveled_up)

# The function that adjusts max_enemies based on player's level
func _on_player_leveled_up():
    max_enemies += player.level * 1

# ----- Spawning -----
func spawn_enemy():
    var attempts = 0
    var max_attempts = 100 # Maximum number of attempts to find a valid spawn
location
    var spawned = false
    while not spawned and attempts < max_attempts:
        # Randomly select a position on the map

```

```

var random_position = Vector2(
    rng.randi() % grass.get_used_rect().size.x,
    rng.randi() % grass.get_used_rect().size.y
)
# Check if the position is a valid spawn location
if is_valid_spawn_location(grass, random_position) ||
is_valid_spawn_location(sand, random_position):
    var enemy = Global.enemy_scene.instantiate()
    enemy.position = grass.map_to_local(random_position) + Vector2(16, 16)
    / 2
    spawned_enemies.add_child(enemy)
    enemy.death.connect(_on_enemy_death)
    spawned = true
else:
    attempts += 1
if attempts == max_attempts:
    print("Warning: Could not find a valid spawn location after", max_attempts,
        "attempts.")

# Valid spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)
    # Check if there's a tile on the water, foliage, or exterior layers
    if water.get_cell_source_id(cell_coords) != -1 ||
    foliage.get_cell_source_id(cell_coords) != -1 ||
    exterior_1.get_cell_source_id(cell_coords) != -1 ||
    exterior_2.get_cell_source_id(cell_coords) != -1:
        return false
    # Check if there's a tile on the grass or sand layers
    if grass.get_cell_source_id(cell_coords) != -1
    || sand.get_cell_source_id(cell_coords) != -1:
        return true
    return false

# Spawn enemy
func _on_timer_timeout():
    if enemy_count < max_enemies:
        spawn_enemy()
        enemy_count = enemy_count + 1

# Remove enemy
func _on_enemy_death():
    enemy_count = enemy_count - 1
    death_sfx.play()

# ----- Saving & Loading -----

```

```

#data to save
func data_to_save():
    var enemies = []
    for enemy in spawned_enemies.get_children():
        #saves enemy amount, plus their stored health & position values
        if enemy.name.find("Enemy") >= 0:
            enemies.append(enemy.data_to_save())
    return enemies

#load data from save file
func data_to_load(data):
    enemy_count = data.size()
    for enemy_data in data:
        var enemy = Global.enemy_scene.instantiate()
        enemy.data_to_load(enemy_data)
        add_child(enemy)

```

```

### PickupSpawner.gd

extends Node2D

# Node refs
@onready var spawned_pickups = $SpawnedPickups

@onready var water = get_tree().get_root().get_node("%s/Map/water" %
Global.current_scene_name)
@onready var grass = get_tree().get_root().get_node("%s/Map/grass" %
Global.current_scene_name)
@onready var sand = get_tree().get_root().get_node("%s/Map/sand" %
Global.current_scene_name)
@onready var foliage = get_tree().get_root().get_node("%s/Map/foliage" %
Global.current_scene_name)
@onready var exterior_1 = get_tree().get_root().get_node("%s/Map/exterior_1" %
Global.current_scene_name)
@onready var exterior_2 = get_tree().get_root().get_node("%s/Map/exterior_2" %
Global.current_scene_name)

var rng = RandomNumberGenerator.new()

func _ready():
    # Spawn between 5 and 10 pickups
    var spawn_pickup_amount = rng.randf_range(5, 10)
    await spawn_pickups(spawn_pickup_amount)

```

```

# ----- Pickup spawning -----
-----

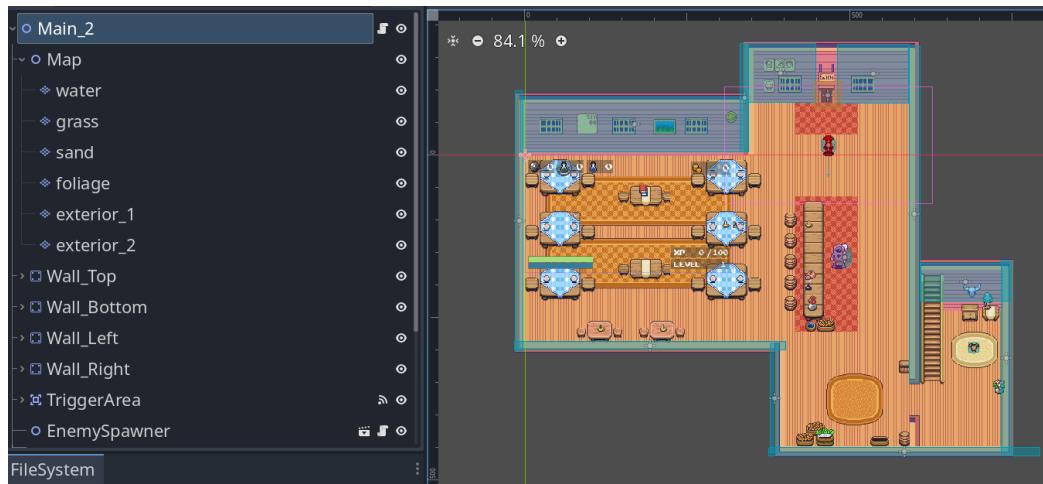
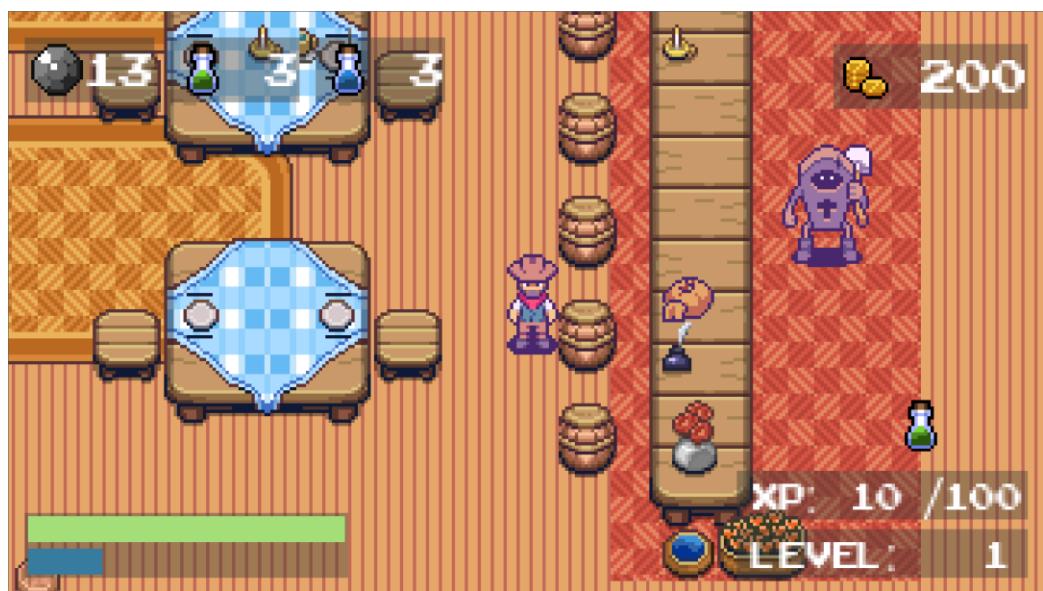
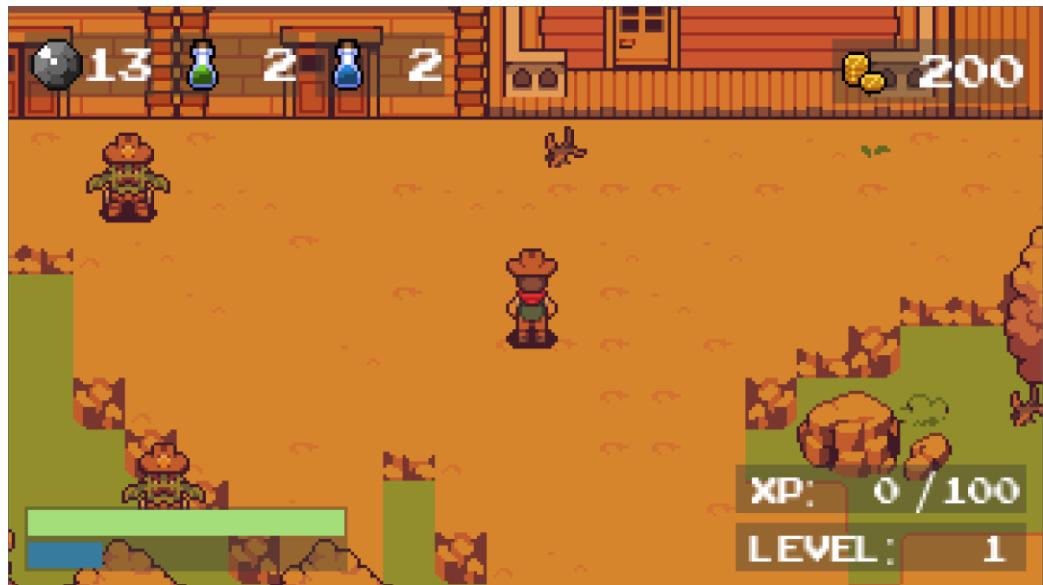
# Valid pickup spawn location
func is_valid_spawn_location(layer, position):
    var cell_coords = Vector2(position.x, position.y)
    # Check if there's a tile on the water, foliage, or exterior layers
    if water.get_cell_source_id(cell_coords) != -1 ||
    foliage.get_cell_source_id(cell_coords) != -1 ||
    exterior_1.get_cell_source_id(cell_coords) != -1 ||
    exterior_2.get_cell_source_id(cell_coords) != -1:
        return false
    # Check if there's a tile on the grass or sand layers
    if grass.get_cell_source_id(cell_coords) != -1
    || sand.get_cell_source_id(cell_coords) != -1:
        return true
    return false

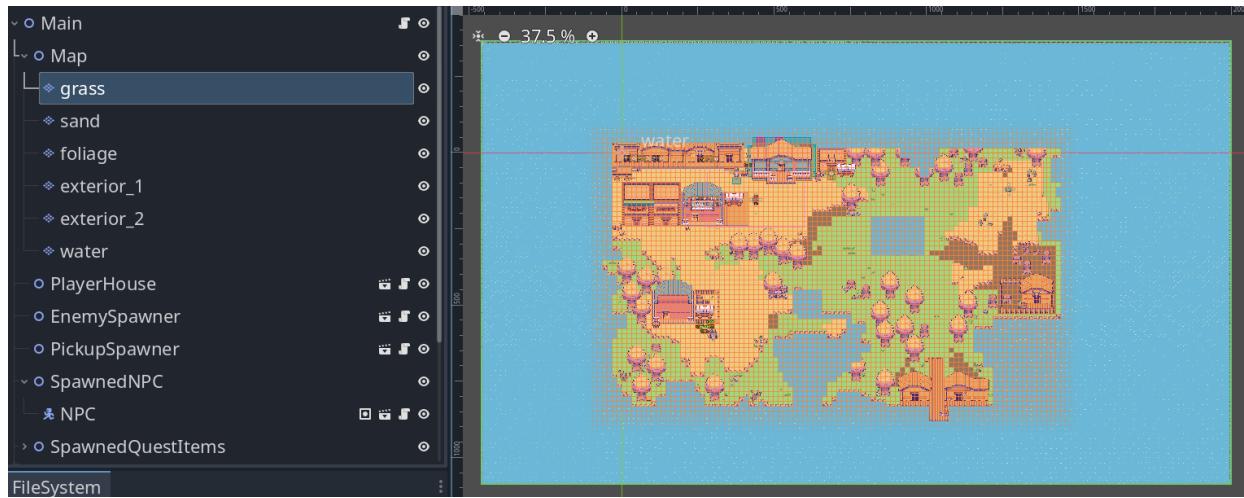
# Spawn pickup
func spawn_pickups(amount):
    var spawned = 0
    var attempts = 0
    var max_attempts = 1000 # Arbitrary number, adjust as needed

    while spawned < amount and attempts < max_attempts:
        attempts += 1
        var random_position = Vector2(randi() % grass.get_used_rect().size.x,
        randi() % grass.get_used_rect().size.y)
        var layer = randi() % 2
        if is_valid_spawn_location(layer, random_position):
            var pickup_instance = Global.pickups_scene.instantiate()
            pickup_instance.item = Global.Pickups.values()[randi() % 3]
            pickup_instance.position = grass.map_to_local(random_position) +
            Vector2(16, 16) / 2
            spawned_pickups.add_child(pickup_instance)
            spawned += 1

```

Now if you run your scene, your Pickups and Enemies should spawn, and your entities should react the same way they did when we had the TileMap node. In the editor, your maps should still look the same as they did when they were compressed in the TileMap node.





And there you have it. You now have a full game with music, a GUI, enemies, an NPC, and a basic quest! If this is the end of the road for you, and you are ready to move on to the next project, then I'll see you in the next part when I show you how test, debug, and export your project. Remember to save and make a backup, and I'll see you in the next part.

The final source code for this part should look like [this](#).

PART 24: TESTING, DEBUGGING, & EXPORTING

Congratulations on making it to the end of our 2D RPG series! It might have taken you a long to get here, but you persisted and hopefully by now you have a working game and you understand all of the concepts that we went over throughout this tutorial. With our game created, you need to go back and test it to make sure that it is as bug-free as possible.

WHAT YOU WILL LEARN IN THIS PART:

- How to install Export Templates.
- How to export projects as Windows Executables.
- How to test and debug your game

For this, you need to delve into the world of gameplay testing. Since this is a small-scale game, we'll focus on the aspects of manual testing, which includes testing the game's mechanics by playing it and trying to break it. Please note, that everybody's testing approach is different, and the following section just contains basic guidelines.

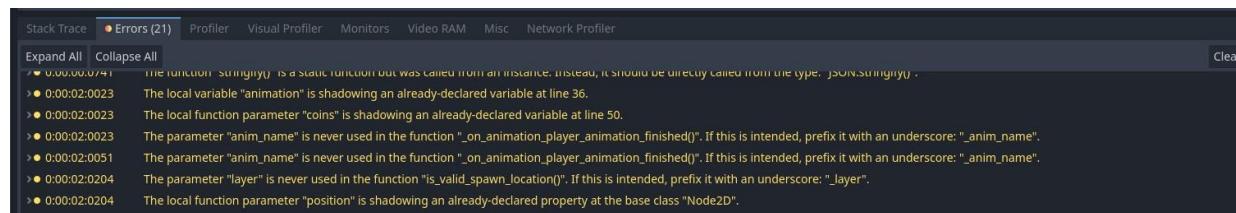
Here's a general guideline for manual testing:

- **Gameplay Mechanics:** This involves testing all of the game's mechanics to ensure they work correctly. In our game, the mechanics to test would include moving, shooting, interacting, damaging entities, consuming pickups, and the ability to progress and end the game. The interaction of the player's character with enemies should also be tested. Are they colliding properly? Do the animations play correctly? Make sure to separate each factor of the game into a checklist which you can then test individually.
- **Levels:** Each level should be tested thoroughly. This includes testing the map generation to make sure that the tiles are generating correctly. Spawn locations of

entities and other obstacles should also be tested to see if they are not out of bounds/unreachable or spawning beyond the game map.

- **User Interface (UI) and Controls:** You should test that the game's controls work correctly and that the UI displays the right information. For example, you might check that the stamina and health progress bar regenerates correctly.
- **Difficulty and Progression:** Check that the game gets harder as you progress through the levels. For example, the player's health increase each time the level progresses - but the enemies health remains the same. Make sure that this progression feels fair and balanced.
- **Audio and Visuals:** Test the game's sound effects, music, and graphics. This would involve testing things like the sounds of picking up items, shooting, and taking damage, as well as the animations for these actions.
- **Performance:** Check the game's performance. It should run smoothly without lagging or stuttering, even when there are lots of entities on the screen.
- **Bugs and Glitches:** Play the game while actively trying to cause bugs and glitches. This might involve things like trying to move into walls, pausing and unpauseing the game, or trying to interact with objects in unexpected ways.
- **Edge Cases:** Test unusual or extreme situations. For example, what happens if the player doesn't move at all? What if they try to run in place whilst shooting?
- **Player Experience:** Lastly, test the overall player experience. Is the game fun to play? Are there any frustrating parts? This will be subjective, so it can be useful to get multiple people to playtest the game.

You can use the debugger to see which functions or methods are returning errors. My debug console returned a lot of warnings (yellow errors) which won't affect my game in any way but instead are giving us suggestions for code optimizations. It also returned a lot of red errors, which might affect our game. Let's go ahead and fix these.



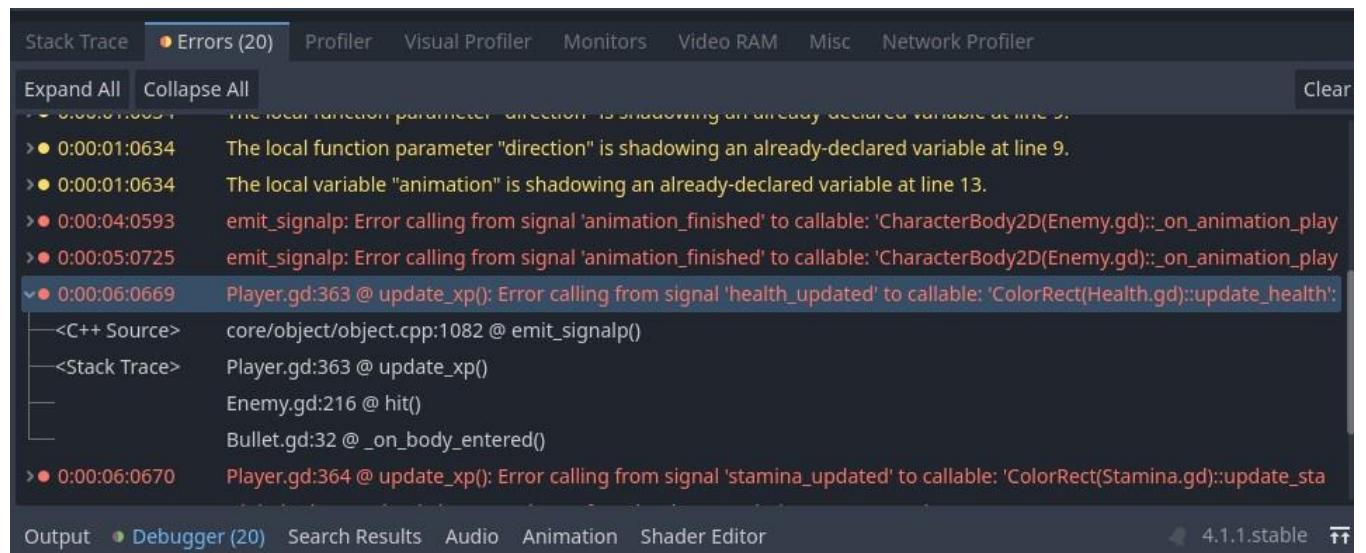
The screenshot shows the Godot Engine's Debug Console with the 'Errors' tab selected. There are 21 errors listed, all of which are warnings (yellow). The errors are as follows:

- 0:00:00:0741 The function "Stringify()" is a static function but was called from an instance. Instead, it should be directly called from the type: `JSON.Stringify()`.
- 0:00:02:0023 The local variable "animation" is shadowing an already-declared variable at line 36.
- 0:00:02:0023 The local function parameter "coins" is shadowing an already-declared variable at line 50.
- 0:00:02:0023 The parameter "anim_name" is never used in the function ".on_animation_player_animation_finished()". If this is intended, prefix it with an underscore: `_anim_name`.
- 0:00:02:0051 The parameter "anim_name" is never used in the function ".on_animation_player_animation_finished()". If this is intended, prefix it with an underscore: `_anim_name`.
- 0:00:02:0204 The parameter "layer" is never used in the function "is_valid_spawn_location()". If this is intended, prefix it with an underscore: `_layer`.
- 0:00:02:0204 The local function parameter "position" is shadowing an already-declared property at the base class "Node2D".

DEBUGGING

Error calling from signal 'health_updated' and 'stamina_updated' to callable.

This error will occur whenever our game tries to update our health and stamina via the signals. Just like the error above, this is due to a mismatch in argument numbers. When we emit the 'health_updated' signal in the Player.gd script, we are passing in an argument. Then, in the Health.gd script, the 'update_health' method is connected to this signal, and is expecting to receive this argument. However, our method 'update_health' is defined without any parameters, hence the mismatch and the error.



To fix this, we just need to emit both our x and max_x variables in our health and stamina signals.

```
### Player.gd
# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        #stop background music
        background_music.stop()
        level_up_music.play()
```

```

#allows input
set_process_input(true)
Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
#pause the game
get_tree().paused = true
#make popup visible
level_popup.visible = true
#reset xp to 0
xp = 0
#increase the level and xp requirements
level += 1
xp_requirements *= 2

#update their max health and stamina
max_health += 10
max_stamina += 10

#give the player some ammo and pickups
ammo_pickup += 10
health_pickup += 5
stamina_pickup += 3

#update signals for Label values
health_updated.emit(health, max_health)
stamina_updated.emit(stamina, max_stamina)
ammo_pickups_updated.emit(ammo_pickup)
health_pickups_updated.emit(health_pickup)
stamina_pickups_updated.emit(stamina_pickup)
xp_updated.emit(xp)
level_updated.emit(level)

```

This issue should now be resolved.

Error Condition "p_scene && p_scene->get_parent() != root" is true.

If you load your game, this error might occur. This error typically occurs when we're trying to set a current scene in Godot, and the scene we're trying to set already has a parent node, which is not the scene tree root. When changing scenes, we usually want to add the new scene as a child of the root node and then set it as the current scene. However, if the scene already has a parent that isn't the root, Godot won't allow us to set it as the current scene, causing this error.

```
>● 0:00:01:0498 The local function parameter "direction" is shadowing an already-declared variable at line 9.  
>● 0:00:01:0498 The local variable "animation" is shadowing an already-declared variable at line 13.  
>● 0:00:01:0505 Global.gd:90 @ load_game(): Condition "p_scene && p_scene->get_parent() != root" is true.  
>● 0:00:02:0419 Global.gd:120 @ load_data(): Node not found: "Player" (relative to "/root/MainMenu").
```

Output Debugger (15) Search Results Audio Animation Shader Editor

4.1.1.stable

We can fix this via the [call_deferred](#) object, which will allow us to set our current scene and change our scene when we load our game both in the same frame, avoiding the error we've encountered.

```
### Global.gd

func load_game():
    if loading and FileAccess.file_exists(save_path):
        #older code
        # Change to the loaded scene
        get_tree().root.call_deferred("add_child", game)
        get_tree().call_deferred("set_current_scene", game)
        current_scene_name = game.name
        #older code
```

Error Node not found: "Player" (relative to "/root/MainMenu").

If you load your game, this error might occur. This error is happening because we're trying to access the "Player" node from the scene "MainScene". It seems like the "MainScene" scene does not have a "Player" node.

```
✖● 0:00:05:0252 Global.gd:90 @ load_game(): Condition "p_scene && p_scene->get_parent() != root" is true.
--<C++ Source> scene/main/scene_tree.cpp:1373 @ set_current_scene()
--<Stack Trace> Global.gd:90 @ load_game()
-- MainMenu.gd:22 @ _on_load_pressed()
✖● 0:00:06:0090 Global.gd:120 @ load_data(): Node not found: "Player" (relative to "/root/MainMenu").
```

Output Debugger (16) Search Results Audio Animation Shader Editor

4.1.1.stable

In our `load_data()` function, we're trying to find the "Player" node from the `current_scene`, which in this case is "MainMenu". Here is the problematic line:

```
var player = current_scene.get_node("Player")
```

If there's no "Player" node in the "MainMenu" scene, we'll get a "Node not found" error. To avoid this error, we can first check if the "Player" node exists in the current_scene before trying to access it:

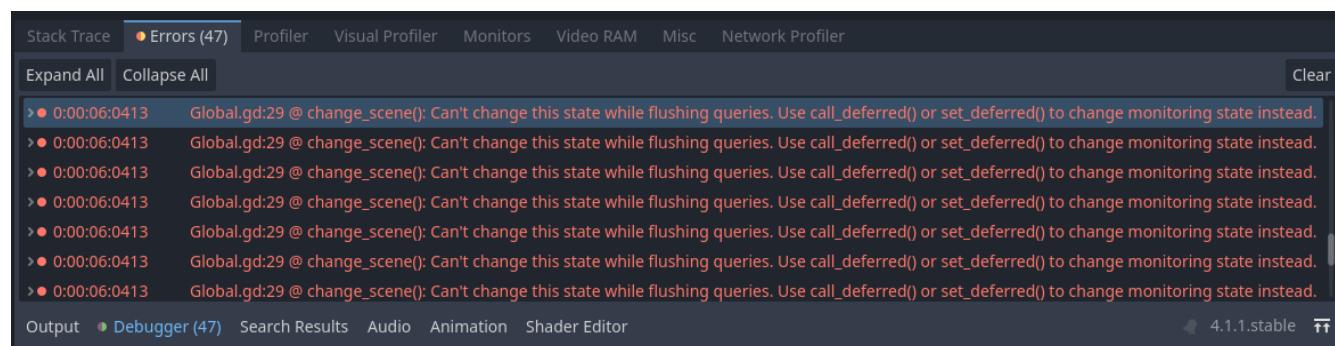
```
### Global.gd

#player data to load when changing scenes
func load_data():
    var current_scene = get_tree().get_current_scene()
    if current_scene and FileAccess.file_exists(save_path):
        print("Save file found!")
        var file = FileAccess.open(save_path, FileAccess.READ)
        var data = JSON.parse_string(file.get_as_text())
        file.close()

        # Now you can load data into the nodes
        if current_scene.has_node("Player"):
            var player = current_scene.get_node("Player")
            if player and data.has("player"):
                player.values_to_load(data["player"])
        else:
            print("Save file not found!")
```

Error change_scene(): Can't change this state while flushing queries.

This error happens when we're trying to change a scene while some physics queries are still being resolved. Godot doesn't allow direct change in some physics properties or change scenes during a physics process as it could lead to crashes or unexpected behavior.

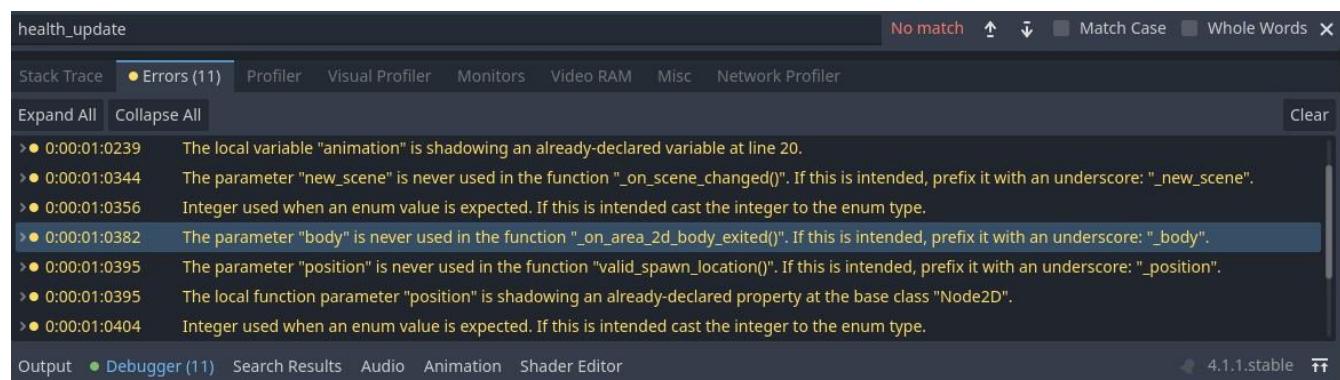


One solution is to use the `call_deferred()` function which schedules the method to be called during the idle time of the next frame. This can prevent race conditions that may arise if physics queries are being flushed while we're trying to change the scene.

```
# Change scene
func change_scene(scene_path):
    save()
    # Get the current scene
    current_scene_name = scene_path.get_file().get_basename()
    var current_scene =
        get_tree().get_root().get_child(get_tree().get_root().get_child_count() - 1)
    # Free it for the new scene
    current_scene.queue_free()
    # Change the scene
    var new_scene = load(scene_path).instantiate()
    get_tree().get_root().call_deferred("add_child", new_scene)
    get_tree().call_deferred("set_current_scene", new_scene)
    call_deferred("post_scene_change_initialization")

func post_scene_change_initialization():
    load_data()
    scene_changed.emit()
```

With these issues, the red errors from our log should be removed. The remaining yellow warnings are of no concern, as removing some of the "shadowed" variables will result in us receiving actual errors. You can however go ahead and fix the warnings that say variable x is never used by adding the indentations to the unused parameters.



TESTING: ENEMY DIFFICULTY

After testing, I realized that I want our level progression to be more fair. Each time we level up, the enemies' health should also increase - making them harder to kill.

We can do this by increasing our enemy's health in their ready function. We can then use the player's level to adjust the enemy's health. For example, for each level the player gains, the enemy's health could increase by a certain percentage or a fixed amount.

```
### Enemy.gd

func _ready():
    rng.randomize()
    # Reset color
    animation_sprite.modulate = Color(1,1,1,1)
    # Adjust enemy health based on player's level
    health += player.level * 10 # Increase health by 10 for each player level
    max_health = health
    print("Enemy health:" , health)
```

Now if we level up, our enemy's health should increase by 10.

```
Godot Engine v4.1.1.stable.official.bd6af8e0e - https://godotengine.org
OpenGL API 3.3.0 NVIDIA 536.23 - Compatibility - Using Device: NVIDIA - NVIDIA GeForce RTX 2060

Save file found!
Enemy health:110
Enemy health:110
Enemy health:110
Enemy health:110
Enemy health:110
Enemy health:110
```

If the player levels up, we can also increase the max amount of enemies that can spawn. To do this, we need to define a new signal that we will emit when the player levels up.

```
### Player.gd

# Custom signals
```

```

signal health_updated
signal stamina_updated
signal ammo_pickups_updated
signal health_pickups_updated
signal stamina_pickups_updated
signal xp_updated
signal level_updated
signal xp_requirements_updated
signal coins_updated
signal leveled_up

# ----- Level & XP -----
#updates player xp
func update_xp(value):
    xp += value
    #check if player leveled up after reaching xp requirements
    if xp >= xp_requirements:
        # older code

        #update signals for Label values
        health_updated.emit(health, max_health)
        stamina_updated.emit(stamina, max_stamina)
        ammo_pickups_updated.emit(ammo_pickup)
        health_pickups_updated.emit(health_pickup)
        stamina_pickups_updated.emit(stamina_pickup)
        xp_updated.emit(xp)
        level_updated.emit(level)
        leveled_up.emit()

```

Then, in our EnemySpawner, we will create a new function that will update our max_enemies + the current level if the signal is emitted.

```

###EnemySpawner.gd

extends Node2D

# Node refs
@onready var player = get_tree().root.get_node("%s/Player" %
Global.current_scene_name)
@onready var spawned_enemies = $SpawnedEnemies
@onready var tilemap = get_tree().root.get_node("%s/Map" %
Global.current_scene_name)

# Audio nodes

```

```

@onready var death_sfx = $GameMusic/DeathMusic

# Enemy stats
@export var max_enemies = 9
var enemy_count = 0
var rng = RandomNumberGenerator.new()

# Inside the _ready() function
func _ready():
    player.leveled_up.connect(_on_player_leveled_up)

# The function that adjusts max_enemies based on player's level
func _on_player_leveled_up():
    max_enemies += player.level * 1
    print("Max enemies adjusted to:", max_enemies)

```

With these changes, the EnemySpawner will spawn more enemies and at a faster rate as the player levels up and the enemies' health will increase, making the game progressively more challenging!

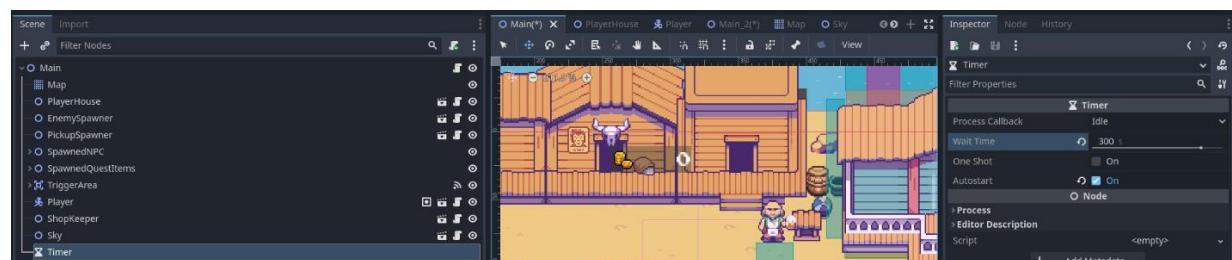
```

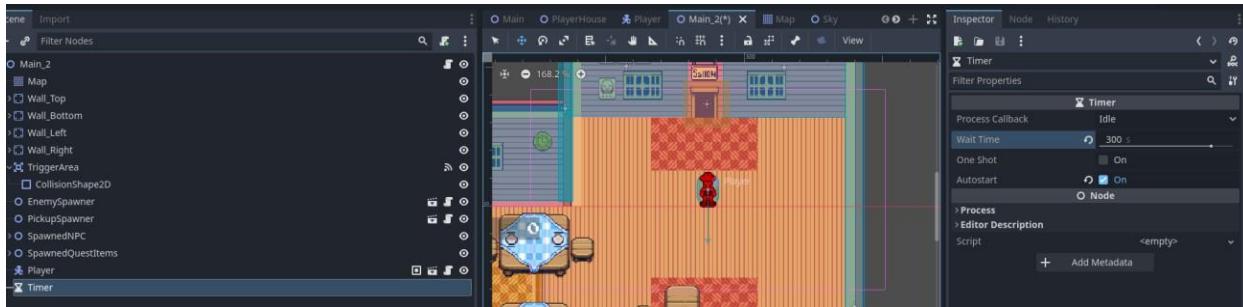
Max enemies adjusted to:16
Enemy health:130
Enemy count:16

```

TESTING: AUTOSAVE

I also want our game to have the functionality to autosave. We can do this with the help of a Timer node. We want our game to autosave every 5 minutes. In your Main and Main_2 scenes, add a new Timer node and set its wait-time to 300 (300 seconds = 5 minutes). Also, ensure that you enable autostart on load.





Connect your timer node's `timeout()` signal to your Main script. In this function, we will simply just call our `Global.save()` function. Every five minutes, the timer will timeout, and save our game.

```
### Main_2.gd

extends Node2D

@onready var background_music = $Player/GameMusic/BackgroundMusic

#connect signal to function
func _ready():
    background_music.stream = load("res://Assets/FX/Music/Free Retro SFX by
    @inertsongs/Imposter Syndrome (theme).wav")
    background_music.play()

# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main.tscn")
        Global.scene_changed.connect(_on_scene_changed)

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()

# Autosave
func _on_timer_timeout():
    Global.save()
    print("Game saved.")
```

```
### Main.gd

extends Node2D
```

```

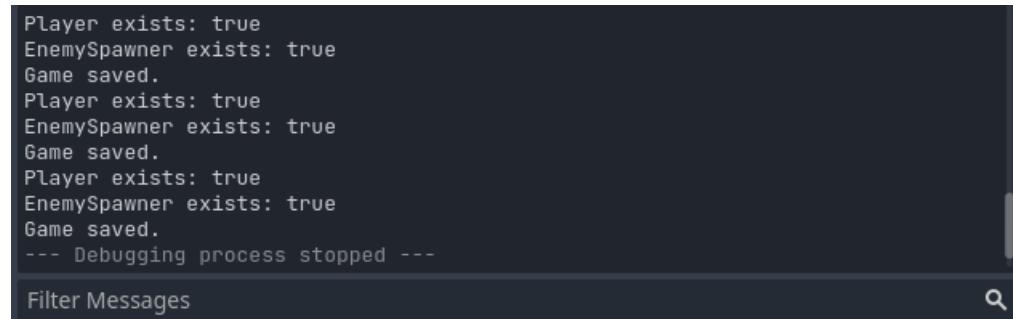
# Change scene
func _on_trigger_area_body_entered(body):
    if body.is_in_group("player"):
        Global.change_scene("res://Scenes/Main_2.tscn")
        Global.scene_changed.connect(_on_scene_changed)

#only after scene has been changed, do we free our resource
func _on_scene_changed():
    queue_free()

# Autosave
func _on_timer_timeout():
    Global.save()
    print("Game saved.")


```

Now after five minutes have passed, your game should autosave!



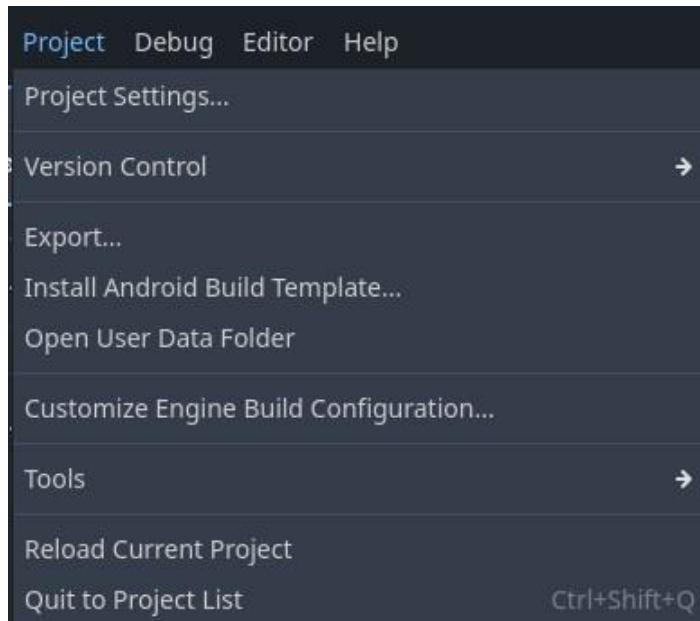
The final source code for this part should look like [this](#).

EXPORTING

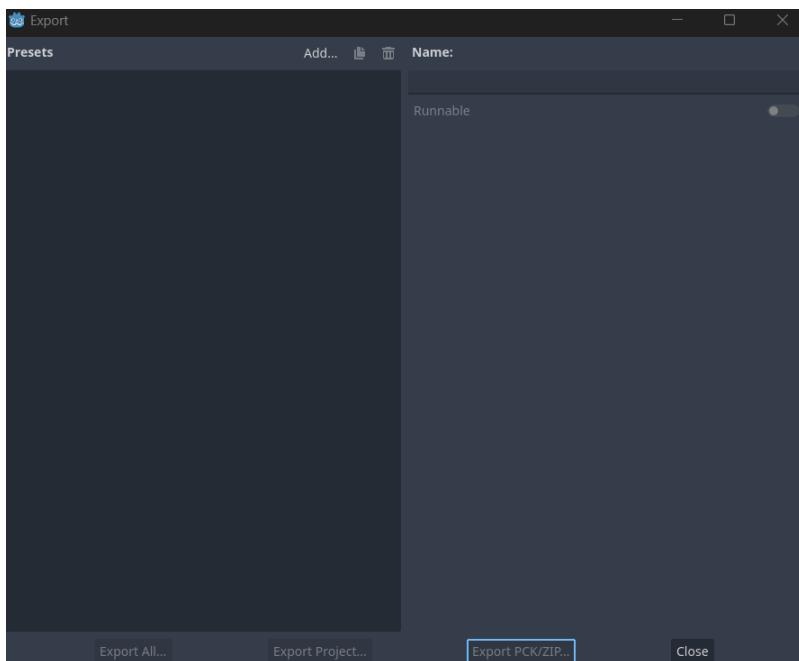
If you've gotten to the point in your game where you have fixed all of your major bugs and you have smoothed out your gameplay, then it might be time for you to [export](#) your game so that others can enjoy it as well! We're going to be exporting our project for PC today. When we export our project, it will be compiled into an executable (.exe) program that we can launch at the click of a button!

Before we can export our project, we need to choose an export template. These templates will compile our binary files into a program file for the platform that we choose.

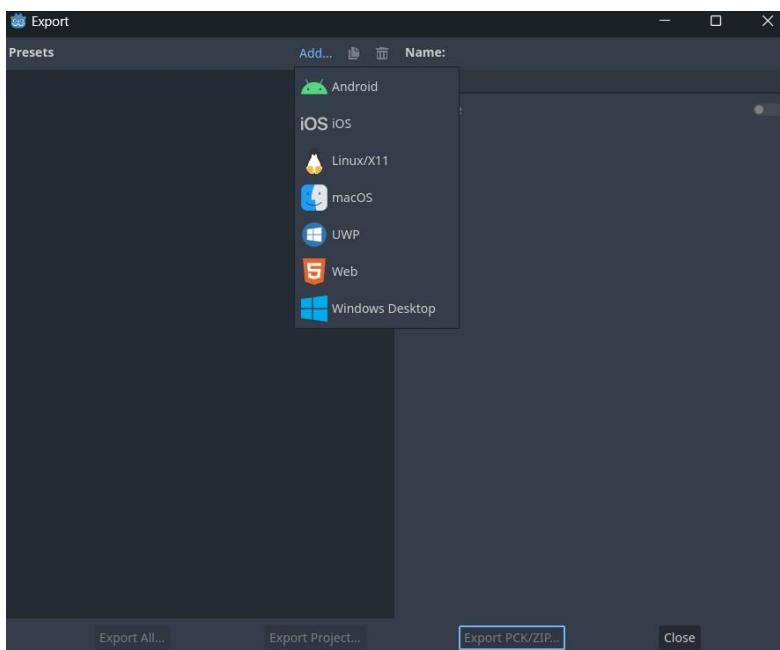
To choose a project template, we need to open our Export menu via our Project > Export property.



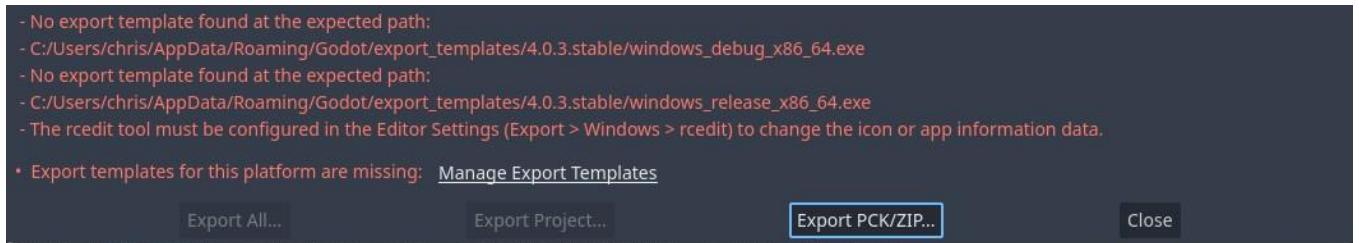
This will open the Export window, and by default, you should have no export preset available to you because you do not have an export template installed.



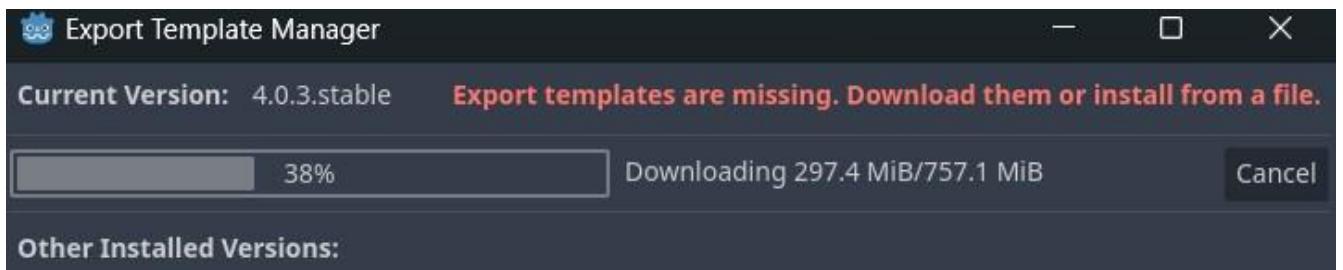
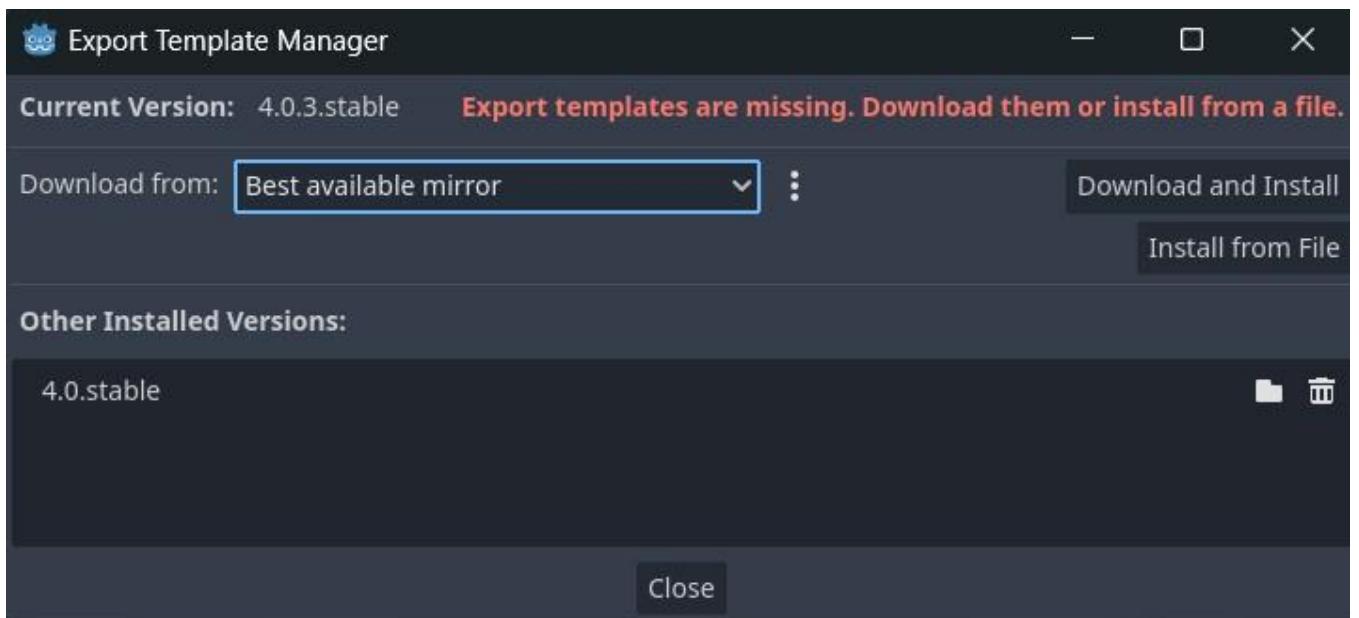
We can add new presets by clicking on the "Add" option next to our Presets menu. This will open a dropdown to which we have to choose the platform that we want to export our project to. We want to export our project as a Windows Desktop application, so select that preset option.



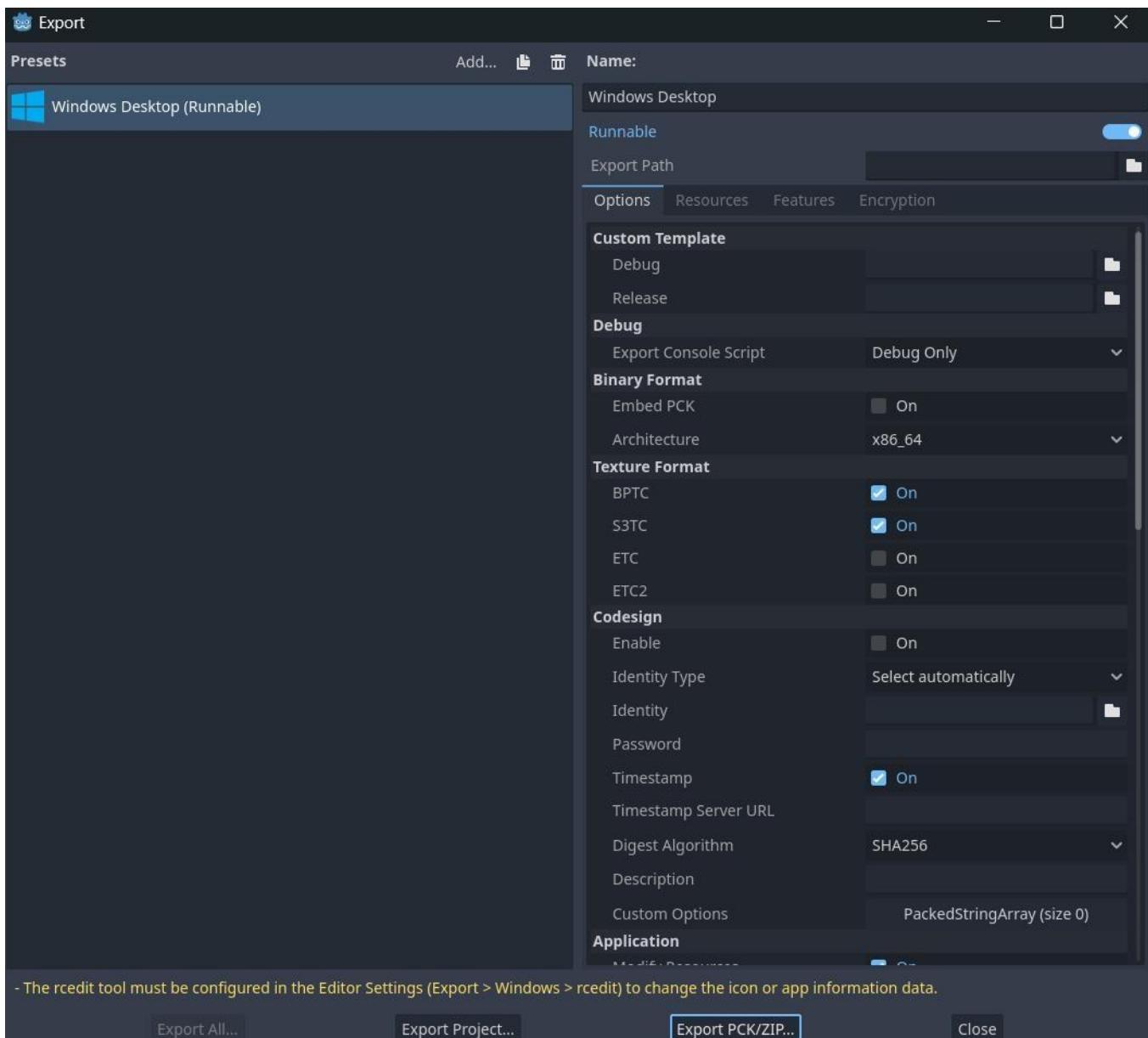
This will display a bunch of text that is written in red. This is because we still do not have our project template installed. Let's install one by clicking on "Manage Export Templates".



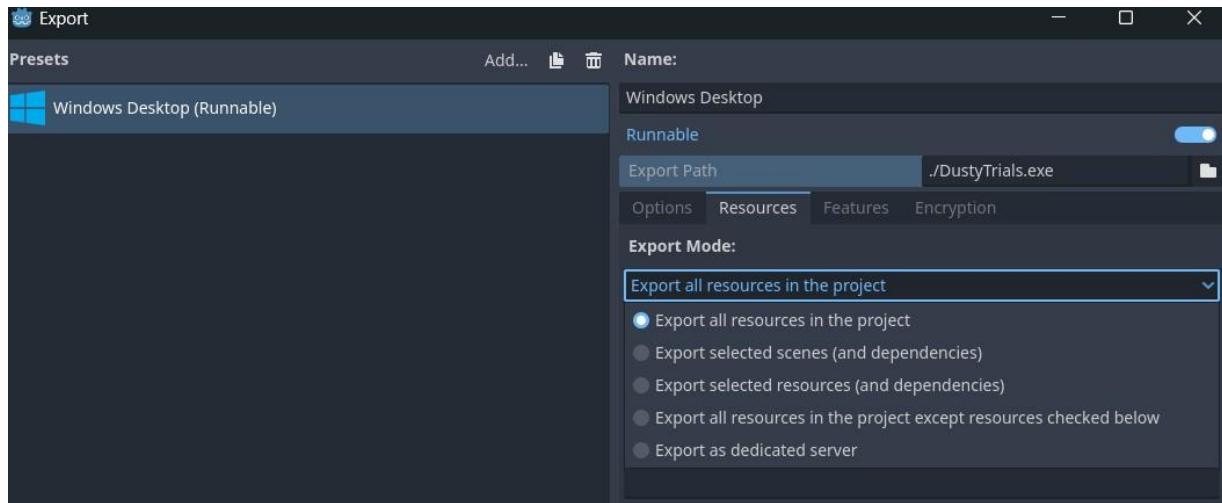
From here you can download the most recent export template available to you. Click "Download and Install" to download the most recent one for your Godot version, or you can download one from the [Godot website](#) and install it from the file.



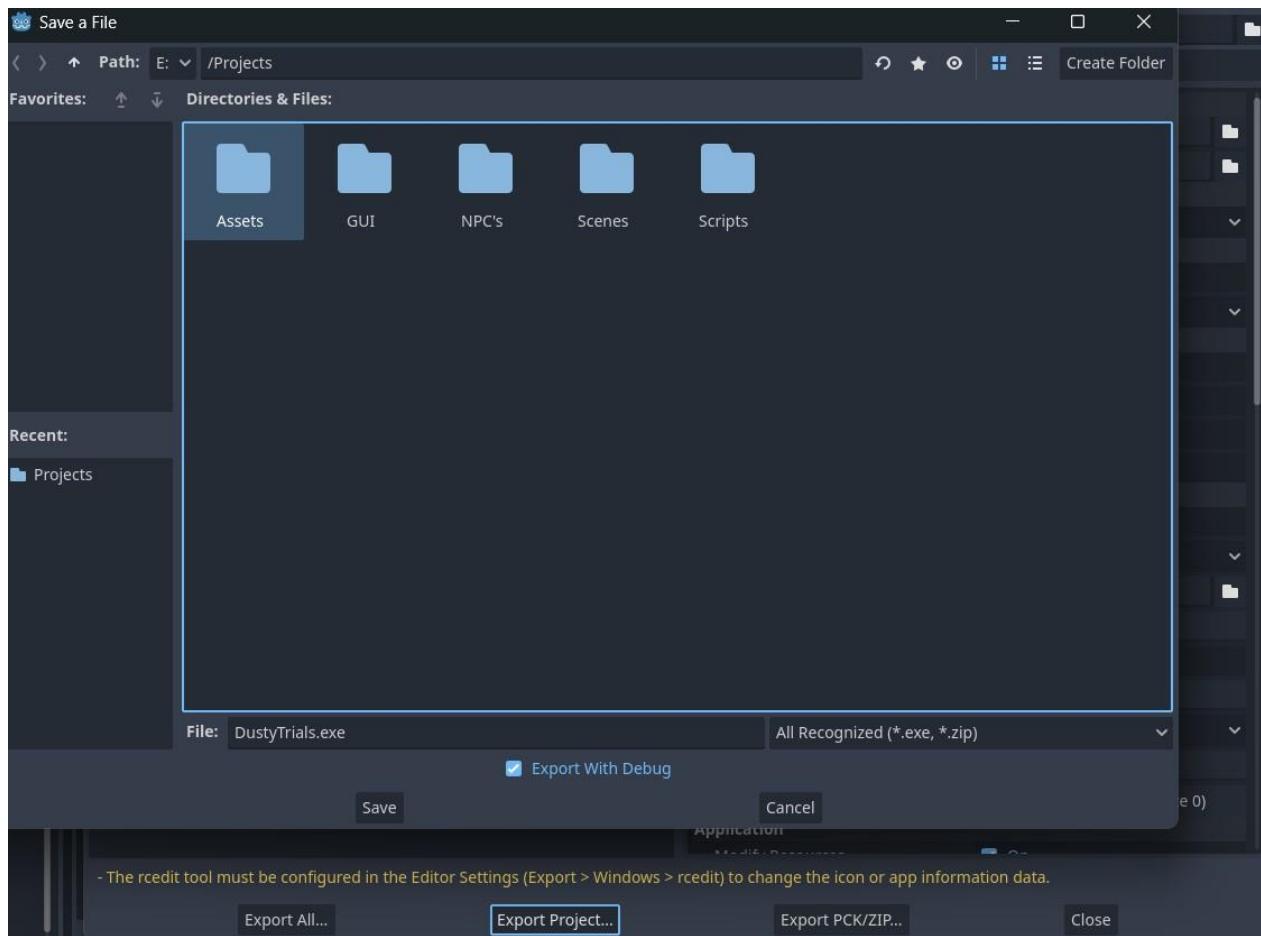
Once it's finished installing, you can close the window and go back to your Exports window. The red warning messages will now be gone. In this window, you can change your game's export name, save the path, and you can even set a password to it - amongst other things.



In your resources menu, you can even set which assets/resources you want to export with the project. For this project, we will export all of our resources. You can read more about the properties in the Export window [here](#).

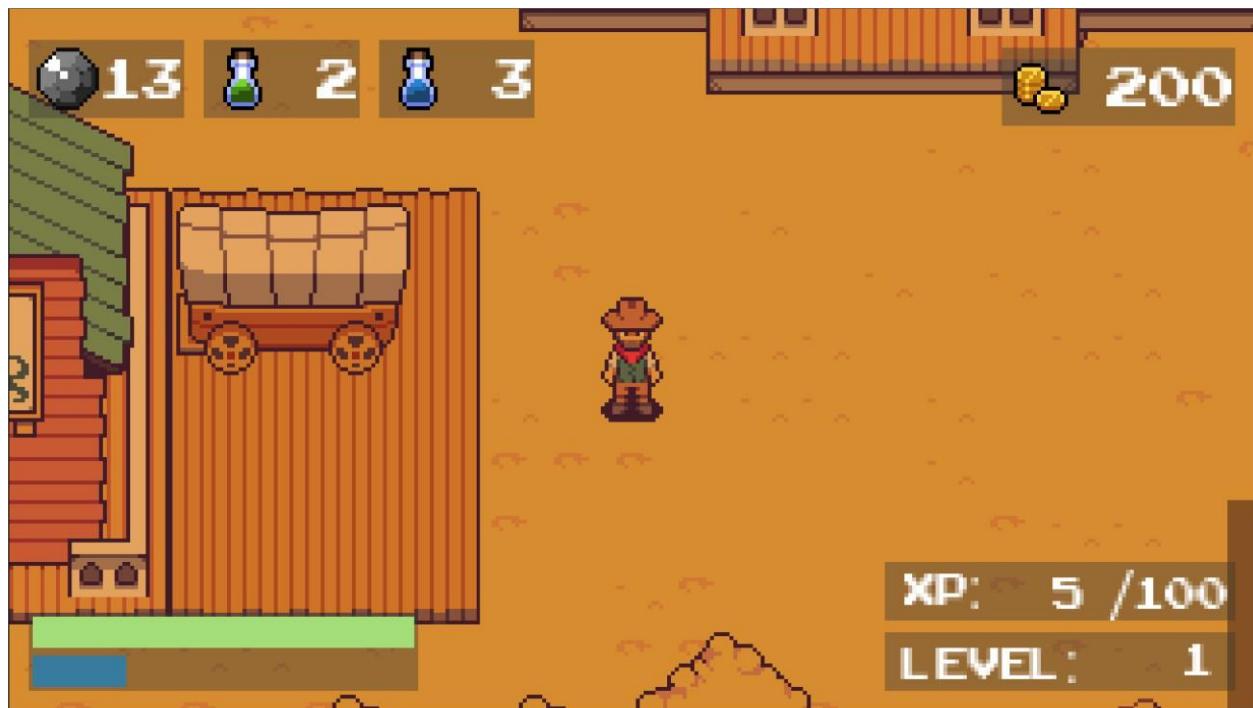


The default properties should be fine for our project — just change its name and save location. When you've added all your properties, you can click on "Export All" to export your executable program to your designated save location.



Now you can navigate to where you exported your project, and voila, your game runs! When you've created your own game and you are confident that you've created a smooth, engaging, and bug-free game, you'll probably export your project so that it can be hosted on online marketplaces such as [Steam](#), or [itch.io](#).

This PC > GAMES (E:) > DustyTrialsApplication			
Name	Date modified	Type	Size
DustyTrials.console	6/11/2023 6:21 PM	Application	182 KB
DustyTrials	6/11/2023 6:21 PM	Application	67,181 KB
DustyTrials.pck	6/11/2023 6:21 PM	PCK File	814,001 KB



XP: 5 /100

LEVEL: 1

PART 25: FURTHER RESOURCES

Congratulations, you've made a game in Godot 4 from start to finish. Hopefully, you've learned a lot on this journey, but you'll only learn as much as you allow yourself to. A good thing to remember is that you'll only really get good at Godot and game development in general if you practice.

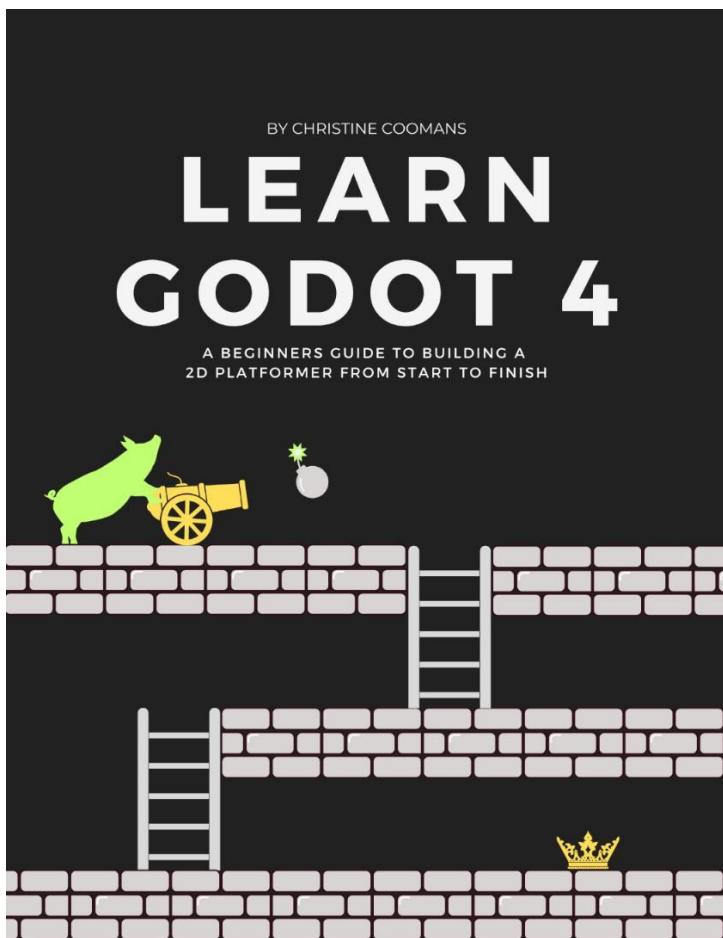
If you're wondering where to go from here, I will recommend the following guidelines:

1. Do as many **tutorials** as possible on game development in Godot for beginners until you get bored. What I like to do, or what works for me when I'm stuck in this phase is that instead of following an entire course, I consume articles or videos that focus on singular topics. I'll list some links to other tutorials at the end of this part.
2. Practicing implementing **simple mechanics** that you've already learned to make. Let's take our player's movement code as an example. Try and implement it by yourself, and if you struggle, look back at how you did it and try again. This will help you to memorize simple things without getting overwhelmed by more complex problems.
3. Move on to more **complex mechanics**. After you've done a few tutorials and practiced what you've learned, you can then challenge yourself to try out more advanced concepts, meaning try to implement things that scare or intimidate you. For example, try and implement a simple slot-based inventory system. If you struggle with it, try something else and come back to it later.
4. Try making a **basic game**. Don't shoot for an AAA game yet. Your first game will never meet the expectations that you've set for it in your head, so don't shoot for the stars with this one. Instead, do something basic that is within your skill range.

5. Focus on **one thing** at a time. When you start out with game dev, don't jump too quickly into other aspects of game dev such as character creation, art, music, etc. You'll get lost if you focus on too many things at once. So, make sure you spend your time where it matters first - learning how to make games before you learn how to make games look and sound good.

If you're looking for something to do next, I'd recommend my "Learn Godot by Making a 2D Platformer" series. This project has been updated to work with Godot 4.1, and it features a King who needs to dodge bombs and boxes from its Pig enemies, as it makes its way to the top of the castle to escape to the next level. At the end of this series, you will know how to make a game from start to finish, and ultimately all the things that you need to know as a beginner Godot developer!

Buy the full offline PDF for \$4 on [KoFi](#).



If that doesn't interest you, maybe try a different tutorial from one of the resources below, or attempt the challenge project!

CHALLENGE PROJECT LINK:

- <https://docs.google.com/document/d/19xAlZQbCgoYLb-knECg0hLFM3DQBPUbI87wkw7HKFel/edit?usp=sharing>

GODOT TUTORIAL LINKS:

- *Oops I Dev'd YouTube (Free):* https://www.youtube.com/@christinec_devs
- *My Website (Free):* <https://christinecdevs.site>
- *Mega List of Godot Courses (Free):*
https://www.reddit.com/r/godot/comments/an0iq5/godot_tutorials_list_of_video_and_written
- *GDQuest Tutorials (Free):* <https://www.gdquest.com/tutorial/godot/>
- *Documentation:* <https://docs.godotengine.org/en/stable/index.html>

UNITY TUTORIAL LINKS:

- *Unity Learning from Unity (Free):* <https://learn.unity.com/>
- *GameDevTV Courses (Paid):* <https://www.gamedev.tv/courses/category/unity>
- *Brackeys (Free):* <https://www.youtube.com/@Brackeys/playlists>
- *Documentation:* <https://docs.unity.com/>

UNREAL ENGINE TUTORIAL LINKS:

- *UE Learning from UE (Free)*: <https://www.unrealengine.com/en-US/students>
- *GameDevTV Courses (Paid)*: <https://www.gamedev.tv/courses/category/Unreal>
- *Ryan Laley (Free)*: <https://www.youtube.com/@RyanLaley>
- *Reids Chanel (Free)*: <https://www.youtube.com/@ReidsChannel/videos>
- *Code Like Me (Free)*: <https://www.youtube.com/@CodeLikeMe>
- *Documentation*: <https://docs.unrealengine.com/5.0/en-US/>

Good luck with the rest of your journey. Remember, this is not supposed to be a linear journey, so feel free to reach out to me if you need some help or advice. I have another tutorial series planned that I will be releasing soon, so once that is out you will be able to see it in my [KoFi](#) updates. Thank you for supporting this series and following along this journey, and good luck with whatever awaits you from here on out!