Welcome to the Course

Thank you for joining this course!

Before we begin, these are the basic requirements for this course:

Prior knowledge

In this course, we will be creating a 2D platformer game in the Godot Engine. This game will feature a player that can jump around, move between levels, collect coins to increase their score, and avoid obstacles such as enemies and spikes.

Required Knowledge

In order to follow along with this course, you should have a basic understanding of two things:

- You should know how to create nodes, create scenes, and test out your game inside the Godot Editor.
- You should know about variables and functions for use in GD-Script. Don't worry if you don't have a full understanding of GD-Script, as we will still be covering the code in the game line by line to make sure you understand what you are creating!

This is, of course, still an introductory course, so if you are missing a few bits of knowledge here and there, that will be no problem and we will cover what to do here.

Godot Features We Will Cover

Across this course we will cover many different features of Godot, here are 3 important ones:

- We will be covering the use of CharacterBody2D nodes to create the basis of our Player. It will allow us to apply gravity and other forces to our player to create our movement script.
- We will also look into creating user interfaces for our text. In this game, we will use it to display our current score on the screen.
- Finally, we will look at linking our scenes together, to create levels for our players to move through as they progress through the game.

Gameplay Features We Will Create

Across this course we will create multiple different gameplay features for the game, here is a list of what we will cover:

- **Player** – We will cover creating a player character which can move left and right using the arrow keys, and jump when the user presses the spacebar.
- **Enemies** – We will create enemies that will reset the player when they touch them. They will move back and forth between 2 points that can be set in the Editor.
- **Spikes** – Similarly to the enemies, we will create spikes, that when touched, reset the player.
- **Coins** – Coins will bob up and down using a sine wave to set their position. They can be collected by the player and will add to a score that appears on the screen as a UI element.

About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

Now that you know what this course is about, let's get started with the first lesson!

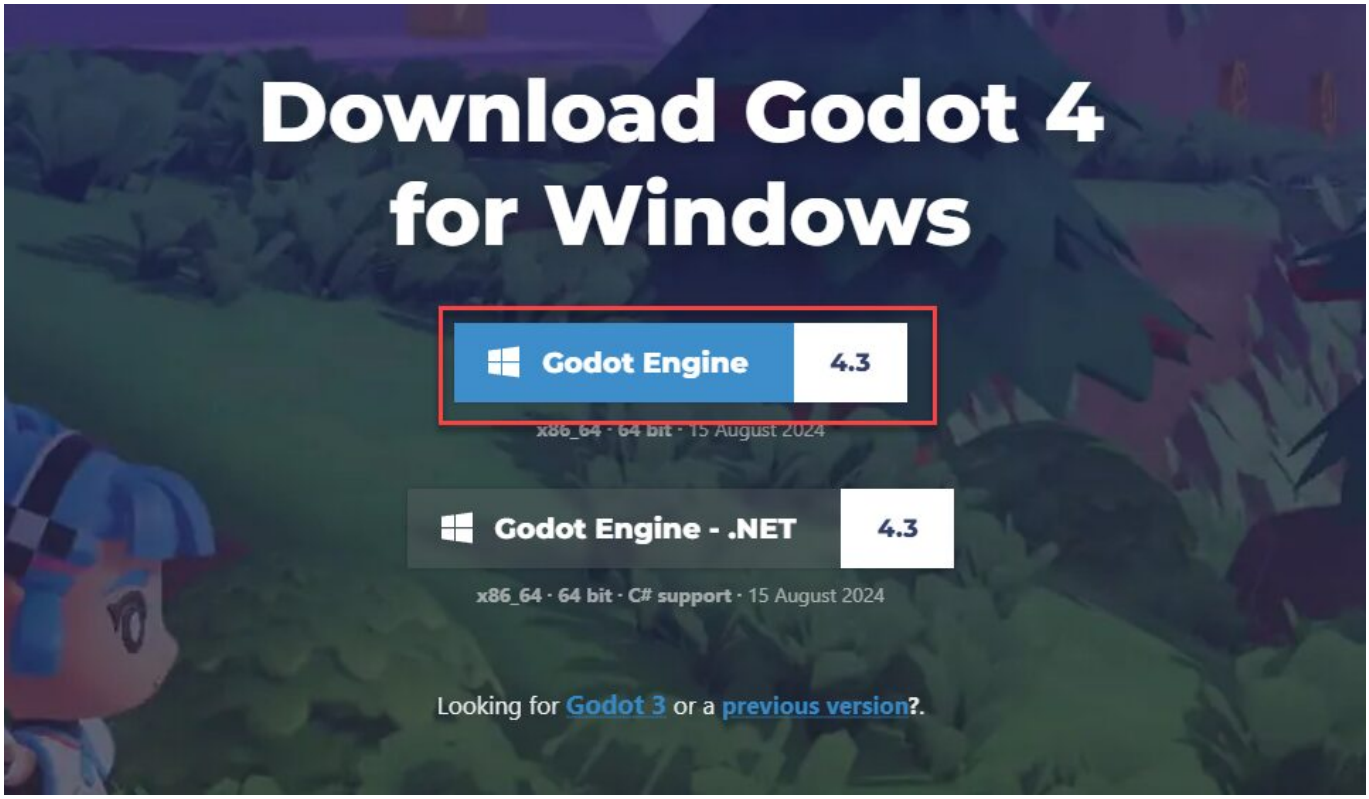
Course Updated to Godot 4.3

We've updated the project files to Godot version 4.3 for this course – the latest stable release.

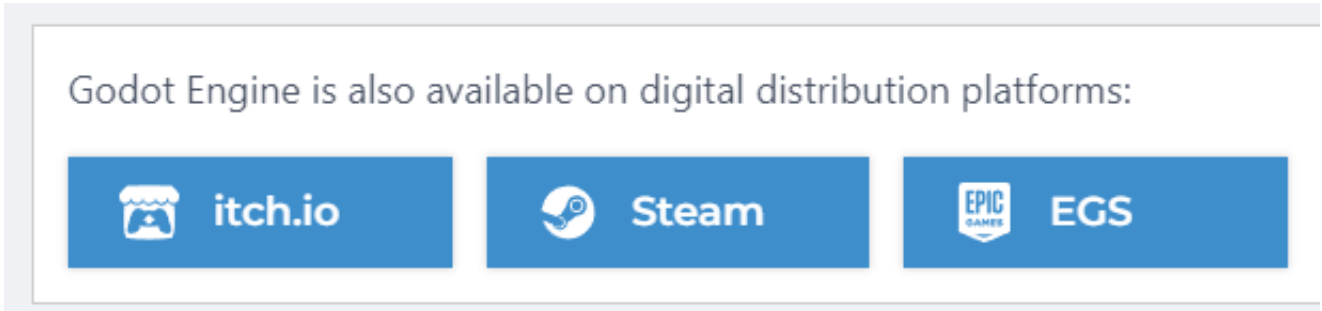
How to Install Version 4.3

You can download the most recent version of Godot by heading to the Godot download page here: https://godotengine.org/download/windows/

Then, just click the option for **Godot 4.3.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application file – no further installation steps required!



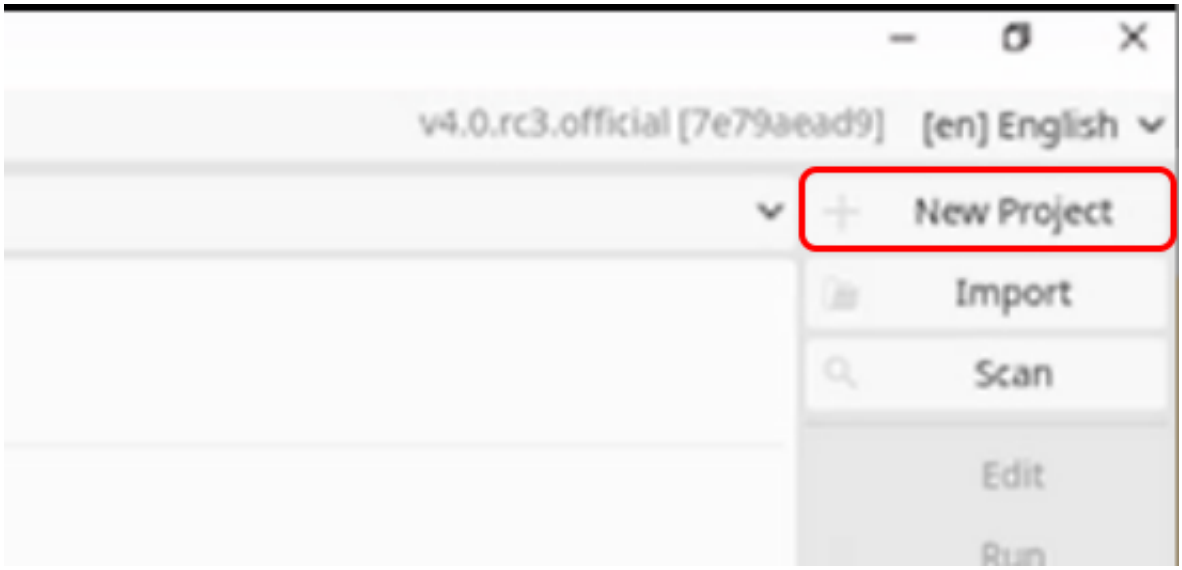
If you are using another non-Windows operating system, such as MacOS or Linux, you can scroll down on the page to change the download link before hitting the button above.



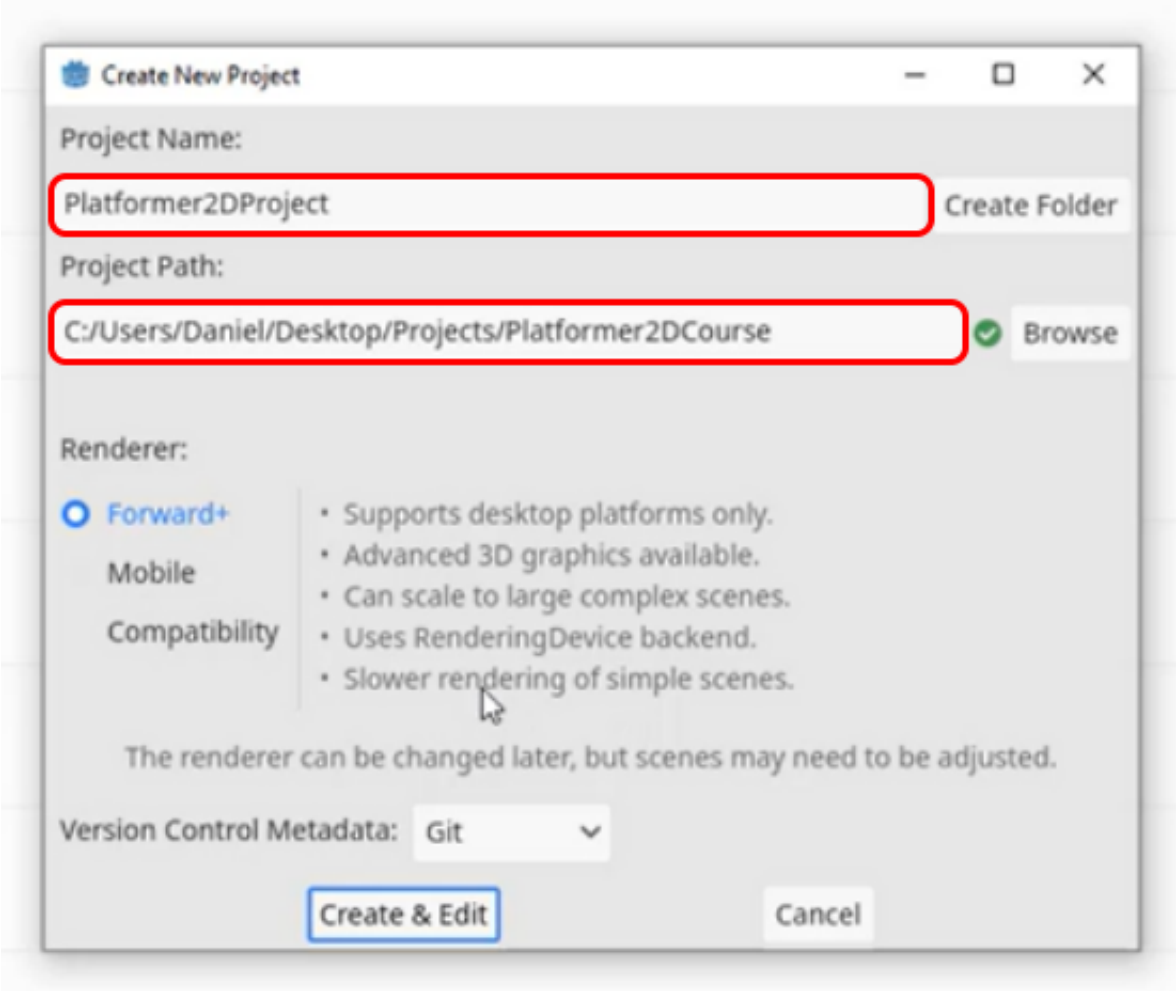
Creating a 2D game with Godot is a great way to get started with game development. In this lesson, we will go through the steps of setting up a project and adding sprites to it.

Opening the Project Manager

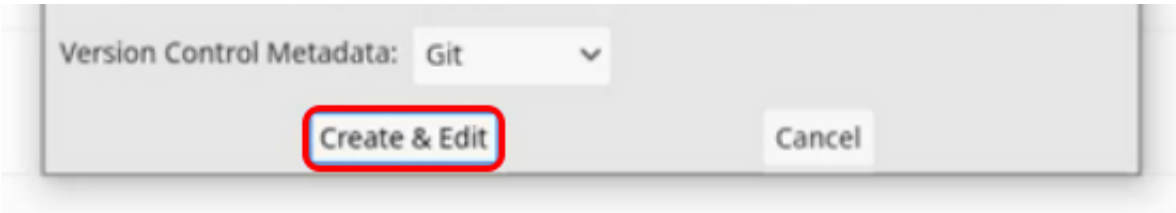
The first step is to open up the Godot project manager by running the .exe, we can then create a **New Project**.



We can then fill out the **Project Name** section, we will be calling our project *Platformer2DProject.* Then, you can choose a *Project Path* to where you want to store the Godot project files. This needs to be an empty folder.

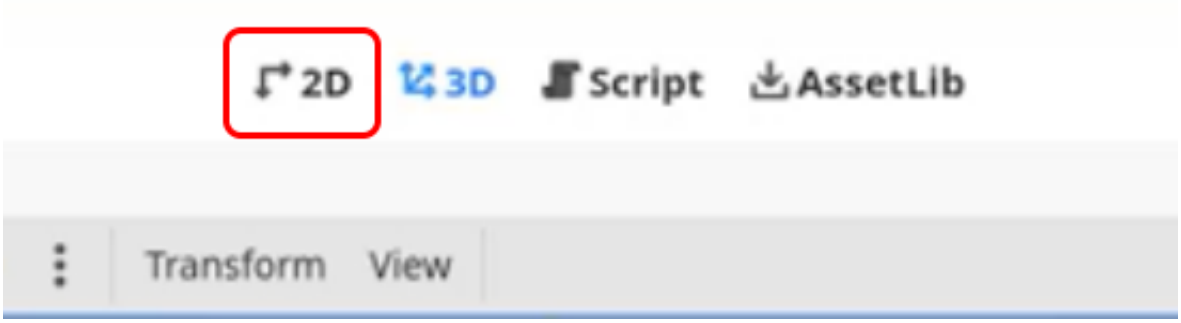


Then press **Create & Edit** to open the new project.

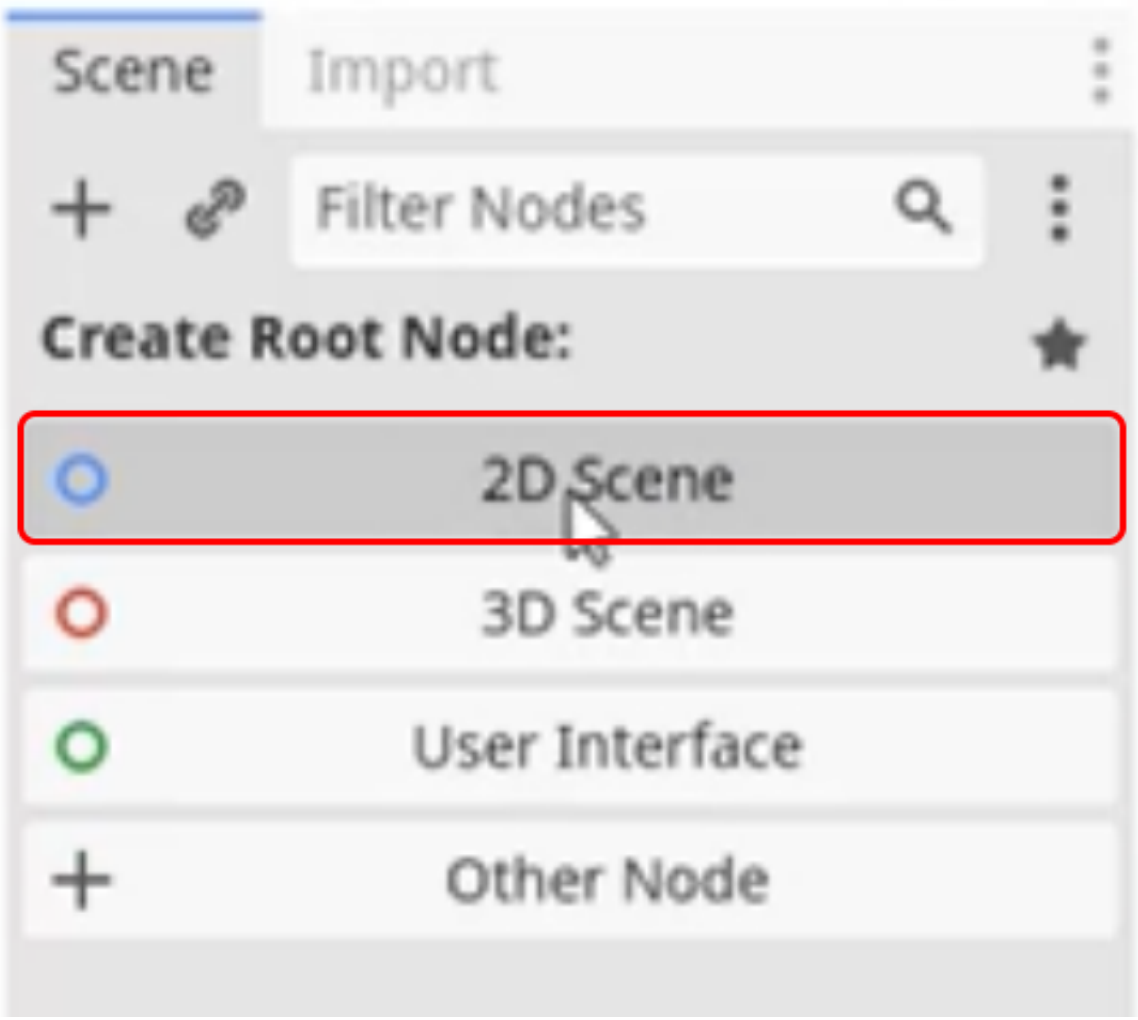


Creating the Root Node

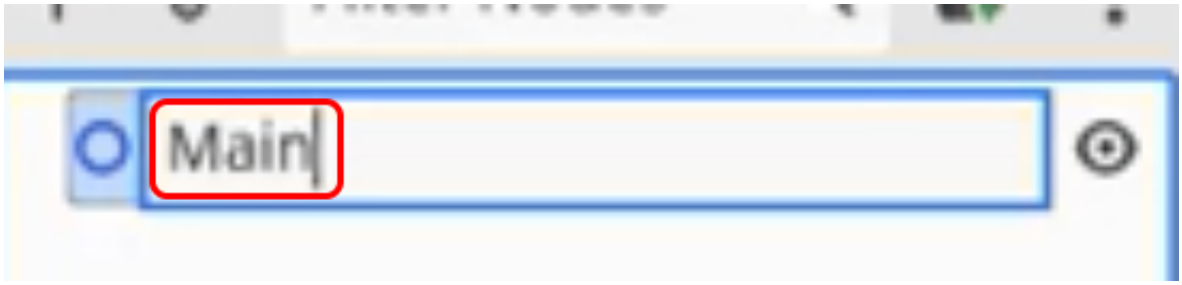
As we will be creating a 2D game, we first need to swap to the **2D** view on the top row.



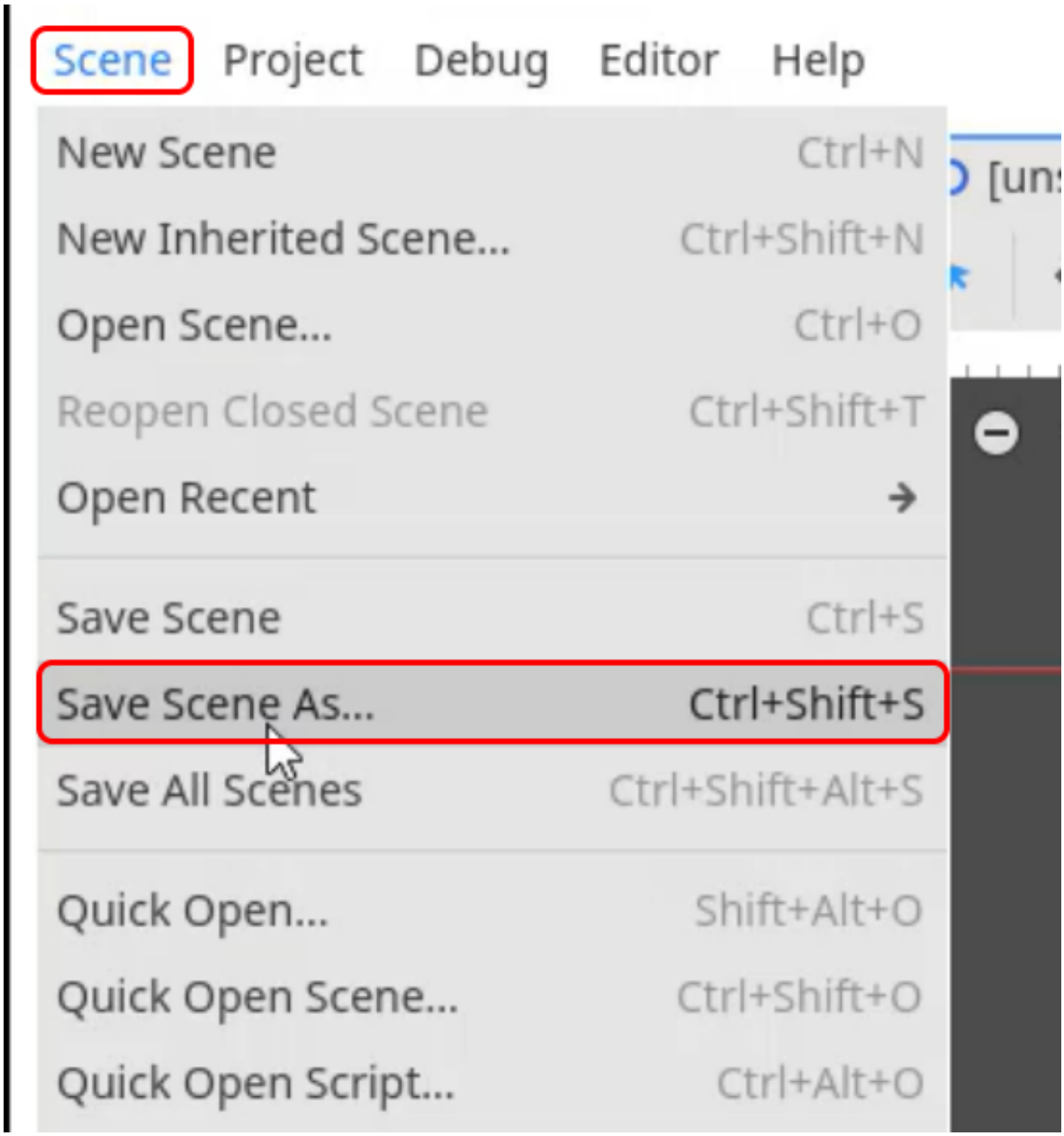
We then need a *root node* for our scene, to do this we will select the **2D Scene** option in the *Scene* tab.



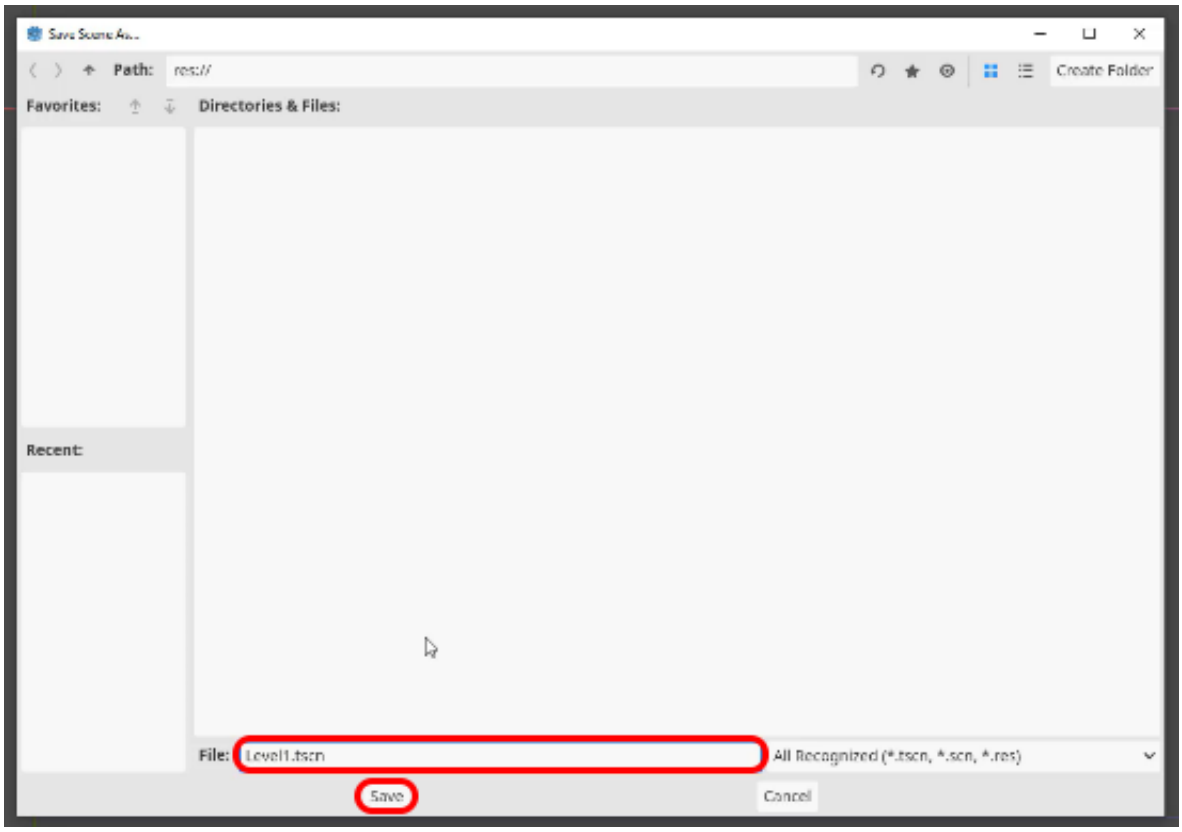
We can then rename this to *Main* so that it is easily identifiable as our scene's root node.



We will then open the **Scene** menu from the top row and choose **Save Scene As...**

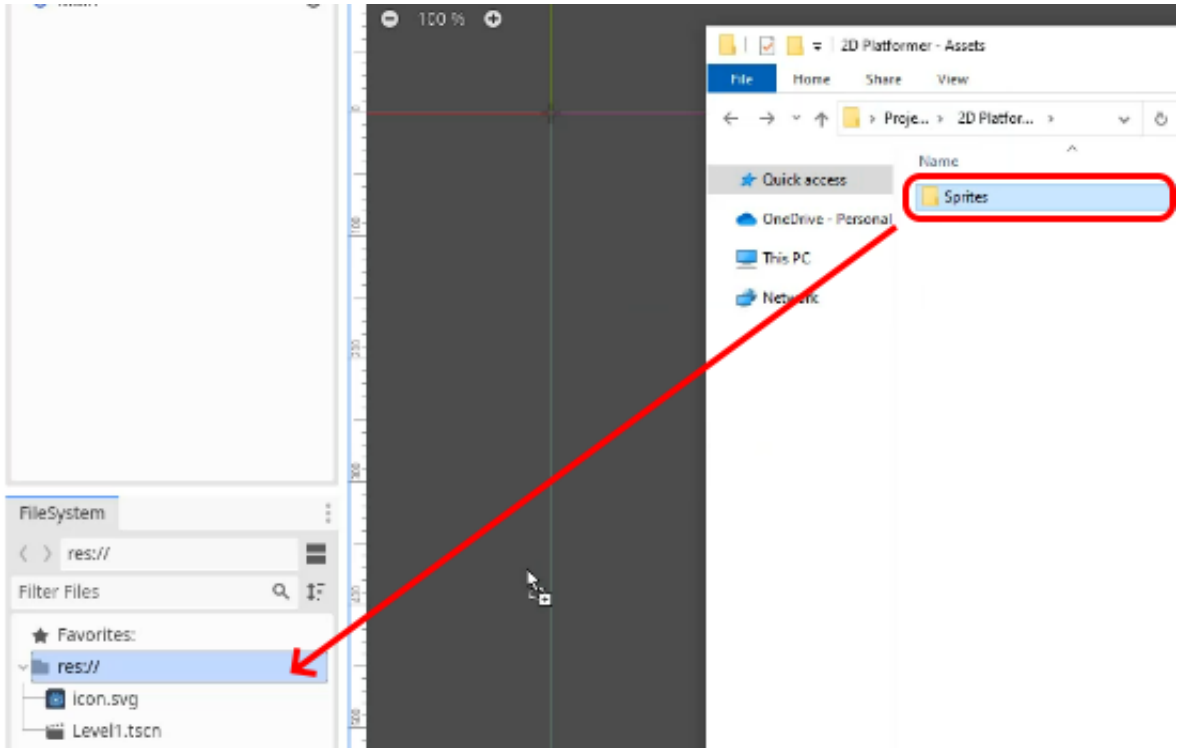


We will then save it as *Level1.tscn* or a similar name and press the **Save** button.

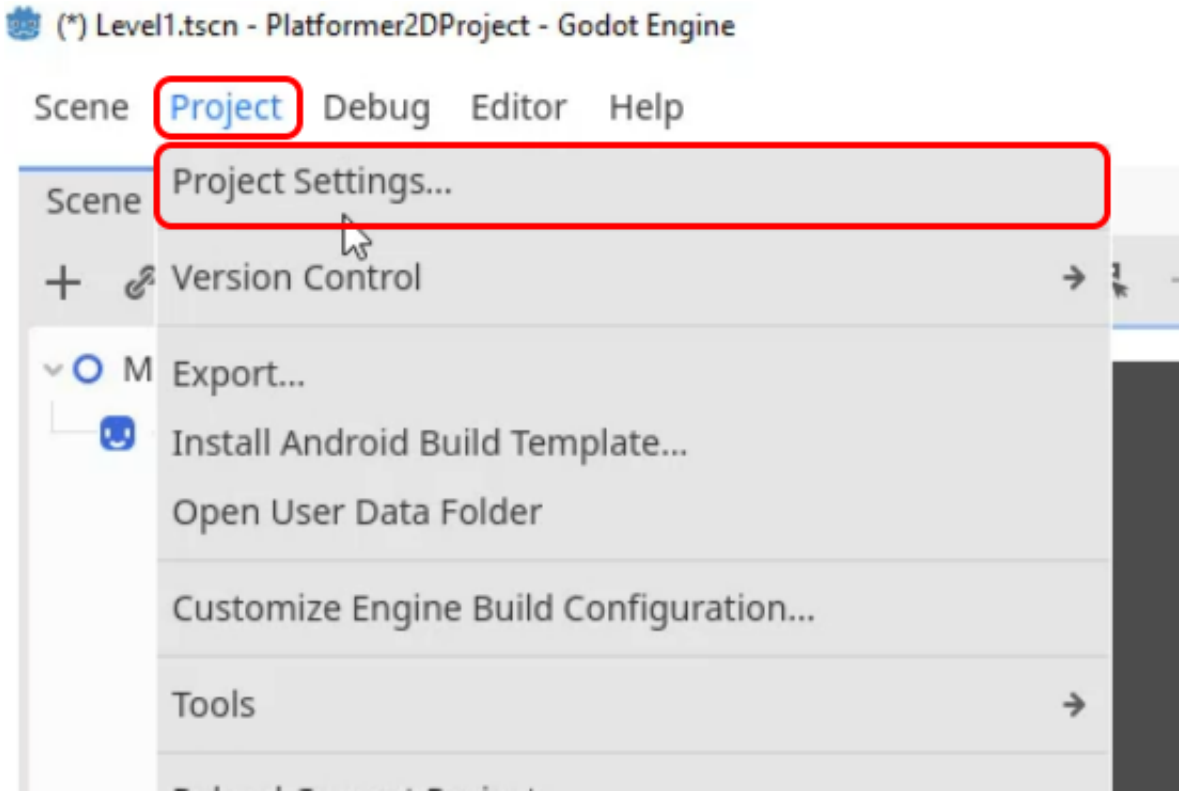


Adding Sprites

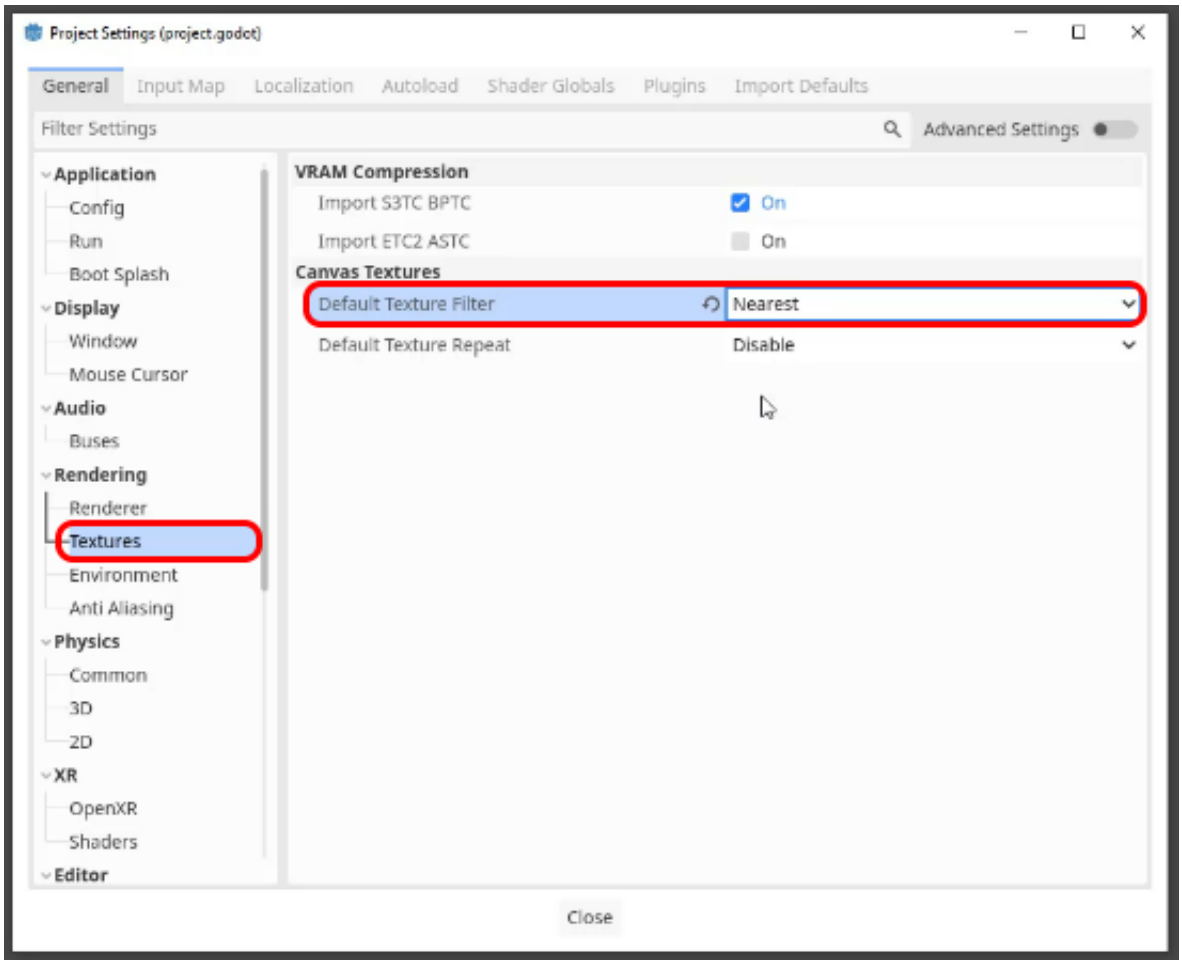
The next step is to add sprites to your project. You can use your own 2D sprites, or you can download the sprites included in the course files tab of this course. If you choose to use our provided sprites, **extract** the contents of the zip file. With the new folder open, *drag-and-drop* the **Sprites** folder into the Godot Editor's *FileSystem* tab.



As we are using pixel art sprites the default import settings make them appear blurry, you can fix this by going to **Project** and selecting **Project Settings**.



Under the *Rendering* section, select the **Textures** tab, and change the **Default Texture Filter** to **Nearest**.

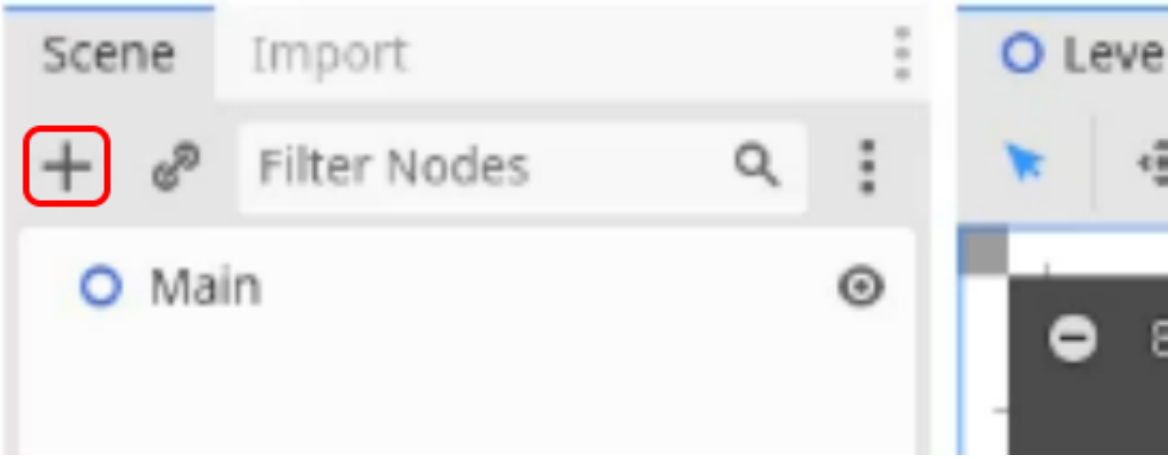


Now that you have set up your project and added sprites to it, we are ready to start creating your game. In the next lesson, we will look at setting up the game environment.

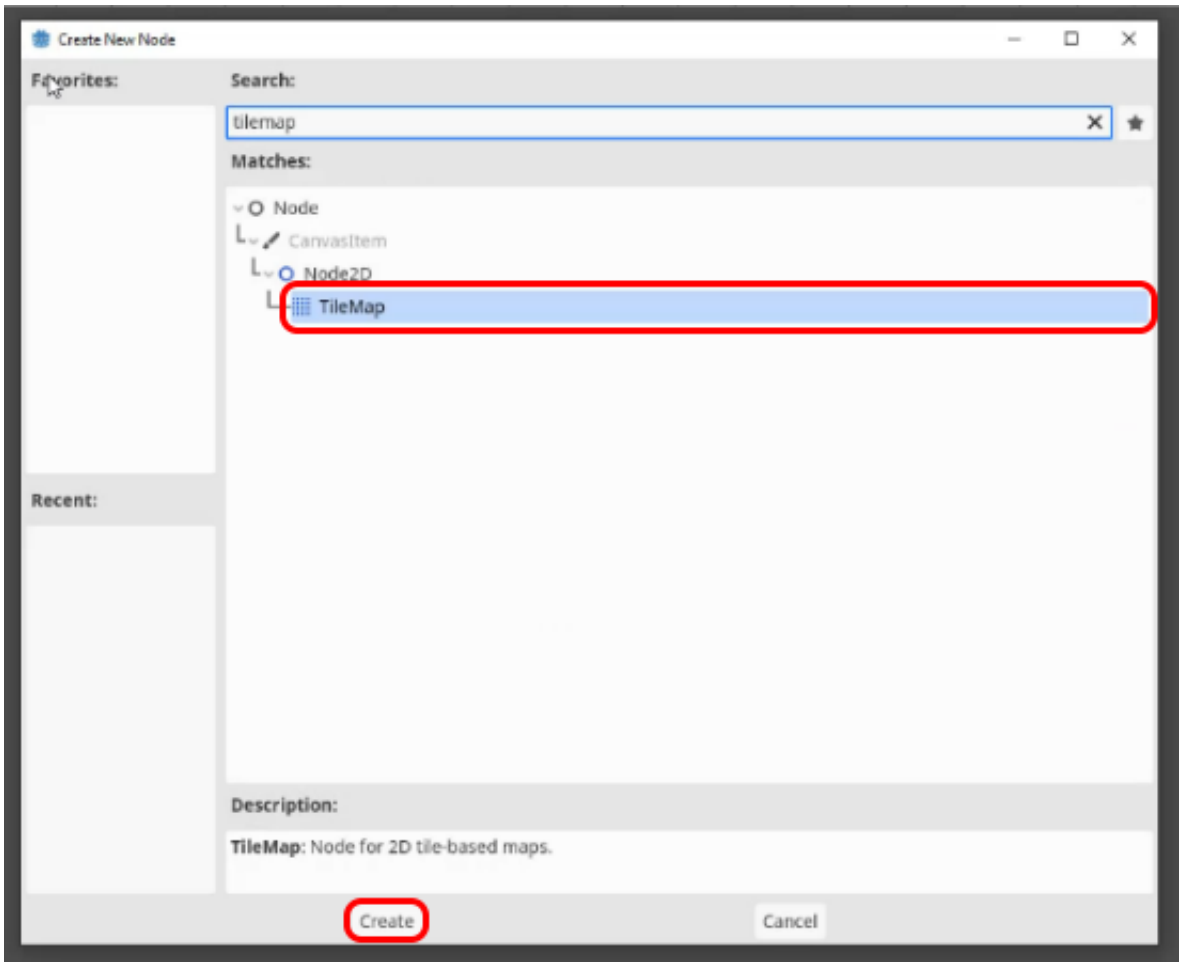
Creating a tile map is an important step in setting up a platform game. It allows you to create a grid that can be used to paint with sprites. In this article, we will go through the steps of creating a tile map and setting up the environment for our platform game.

Creating the Tile Map

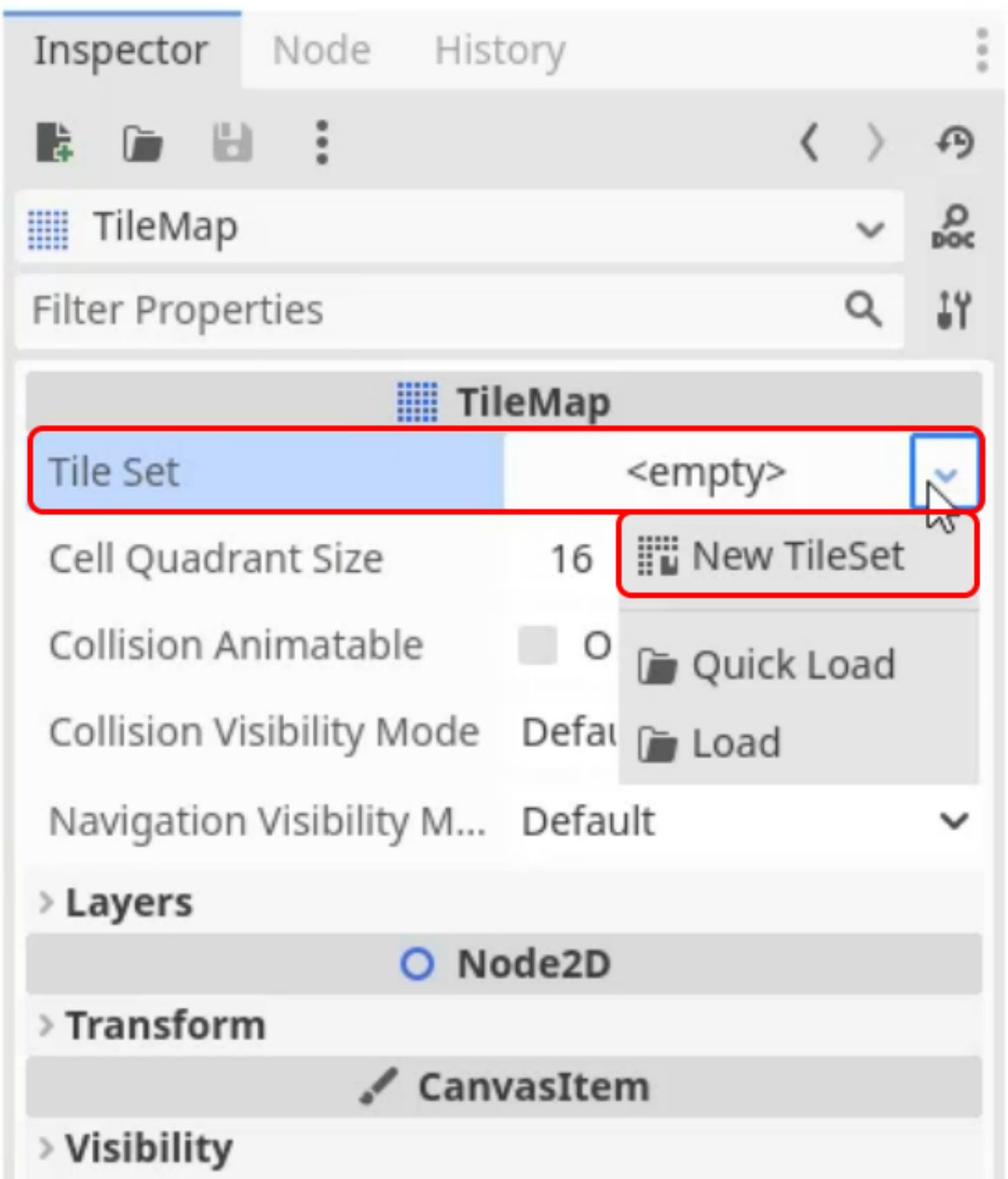
To begin, we need to create a **TileMap** node. To do this, go to the **Scene** tab and click on the **plus** sign.



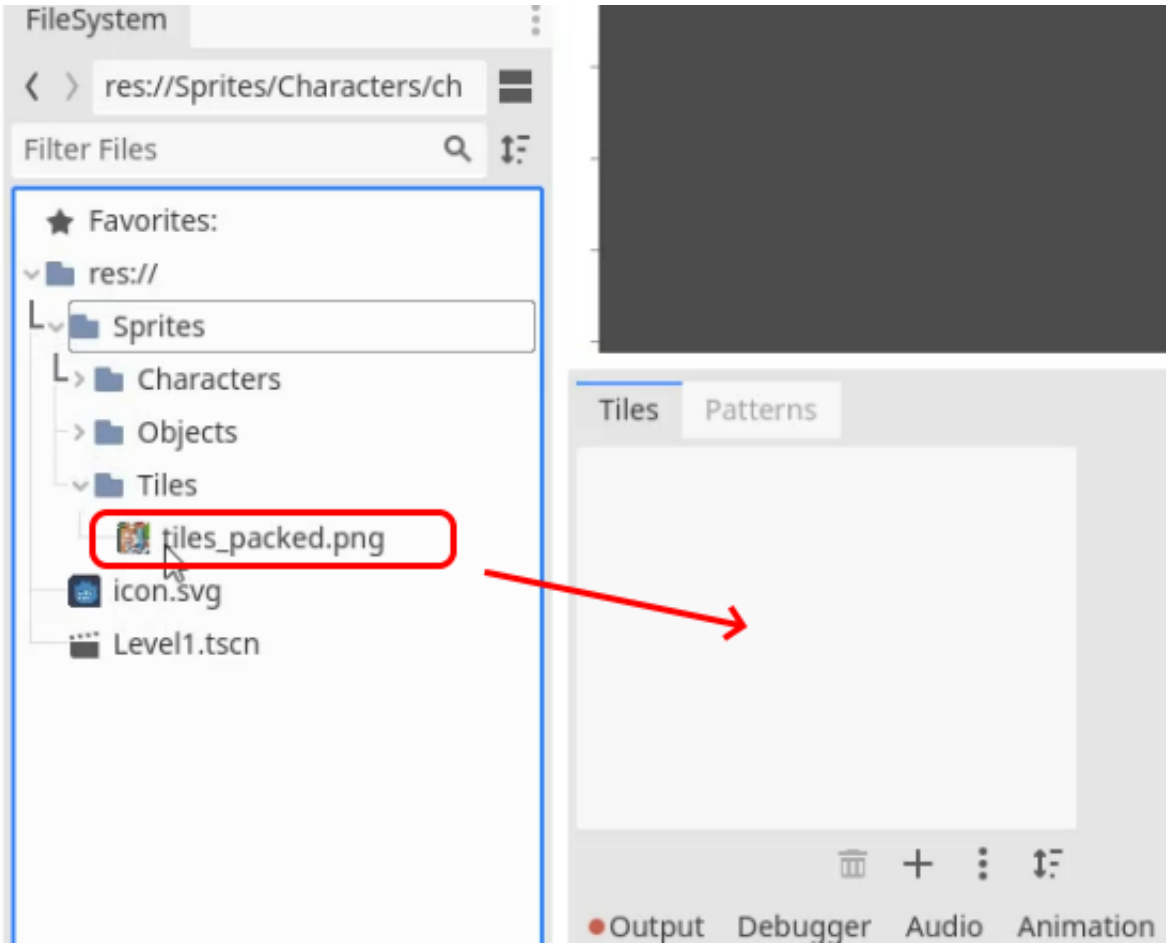
We then search for the **TileMap** node and create it.



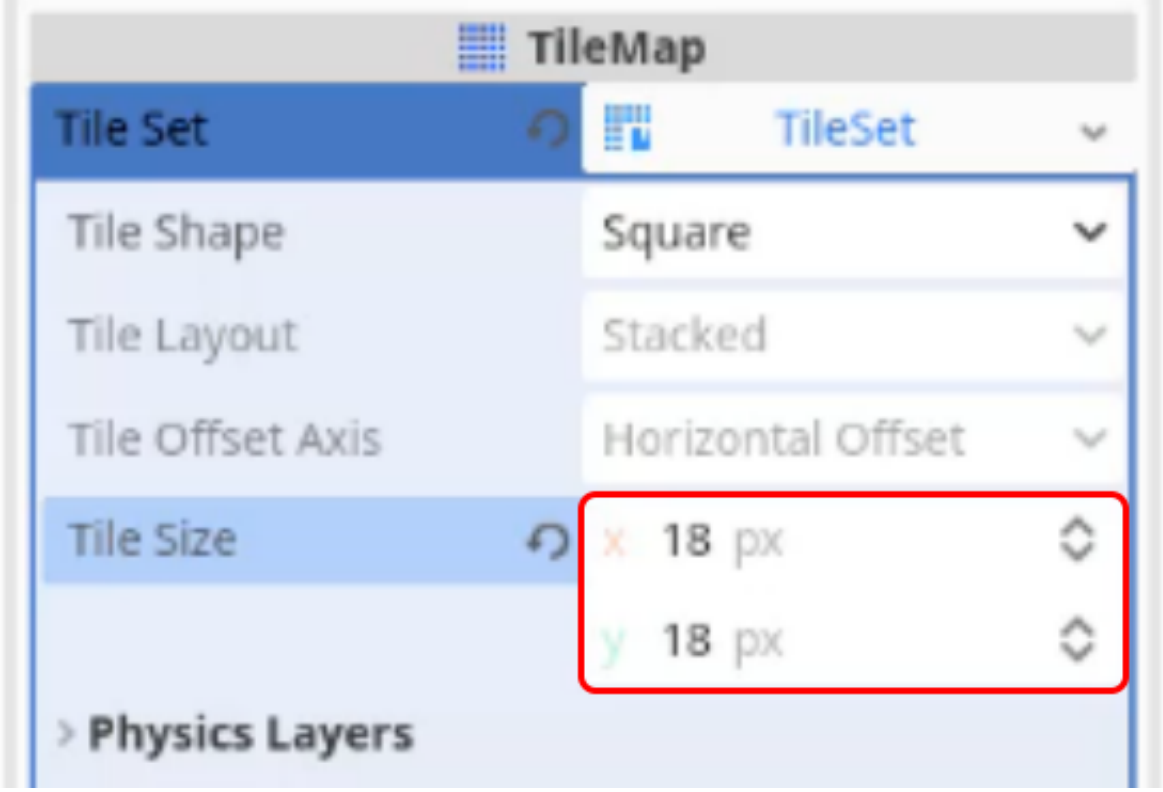
Once the tile map is created, go to the inspector tab and click on the tile set button to create a **New TileSet**.



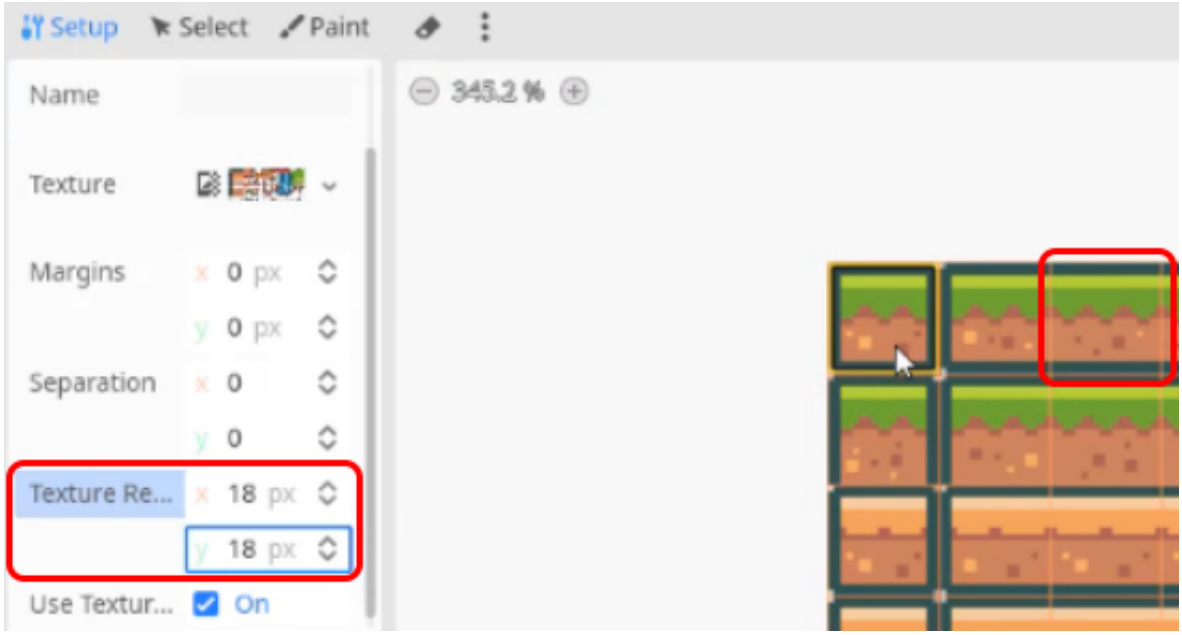
This will open up the tile set window along the bottom of your editor. Here, we need to tell the tile map what sprites we want to use to construct our level. To do this, *drag-and-drop* the **tiles-packed.png** file into the white box.



If you get a popup, click yes. If the tile size appears wrong inside the *Tiles* editor, go to the *Inspector* window and change the **Tile Size** to **18 by 18**. This will ensure that the sprites are correctly aligned, if you are using your own textures, you may want to use different values.

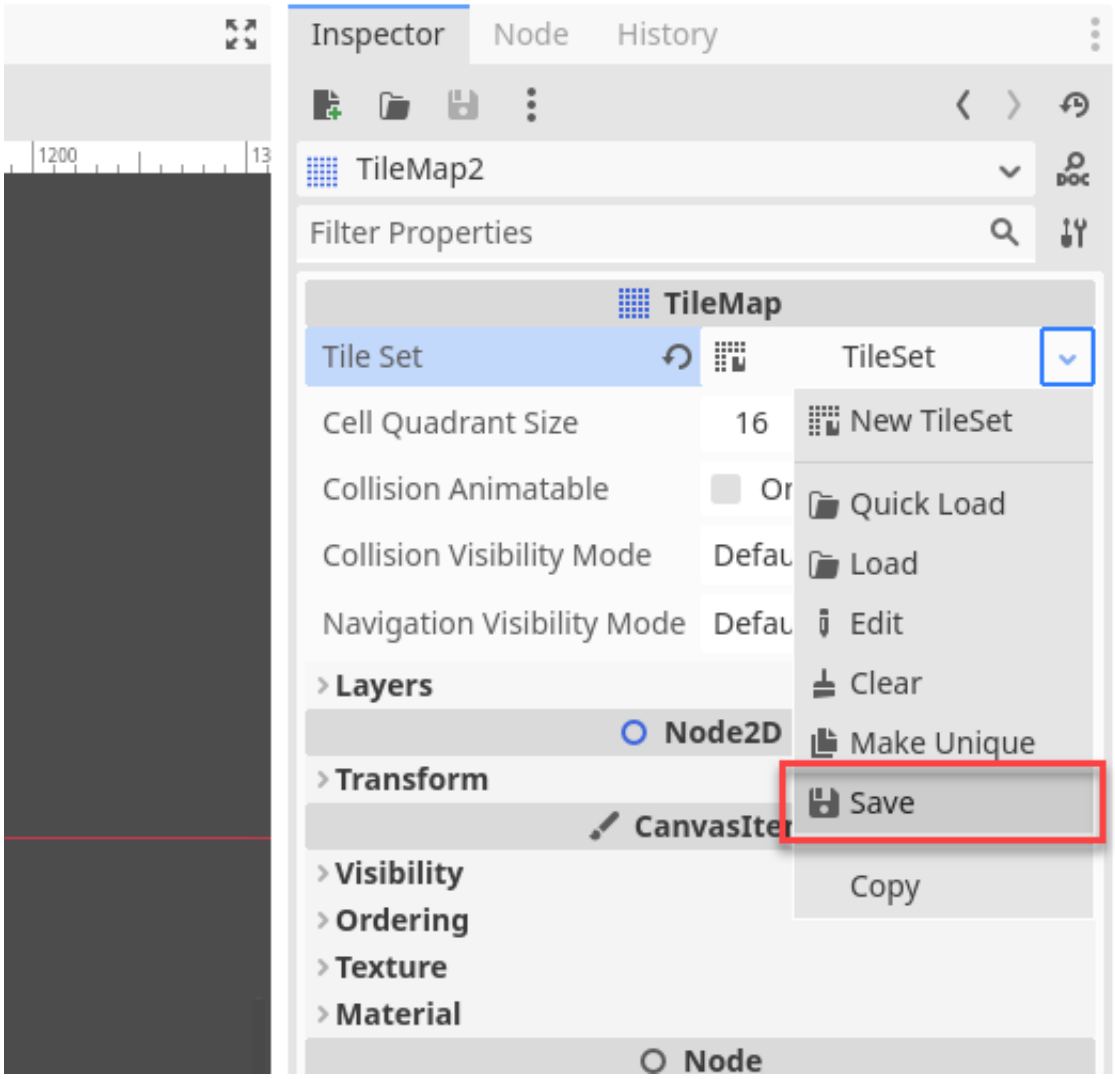


Then, in the *Properties* panel of the *Tiles* editor, change the **Texture Resolution** to be the same **18 by 18**. This will make the grid match up with your tiles.



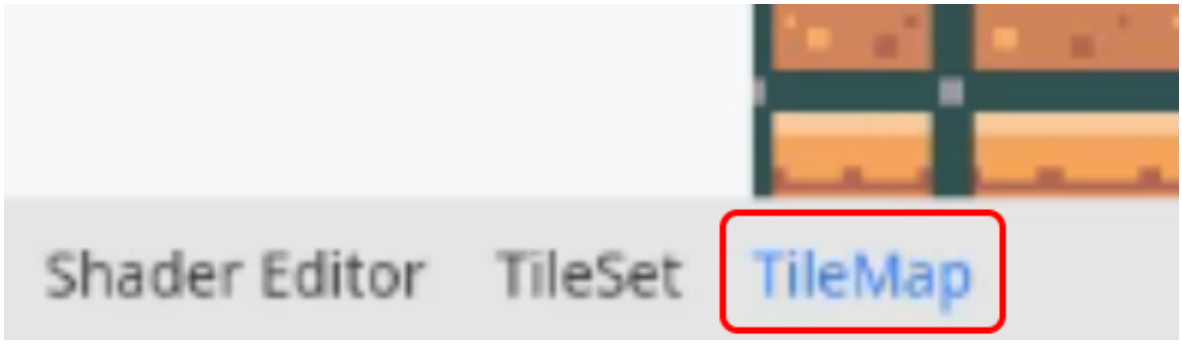
If you are using a custom tileset, it may have gaps in between the tiles. You can add this using the *Separation* and *Margins* options.

Now, if you want to use this tile set across multiple tile maps, in other scenes for example, then you can save it to your File System. Just click on the down arrow next to the **Tile Set** and select *Save*. This will then create a new resource in your File System which you can drag and drop into a new tile map.

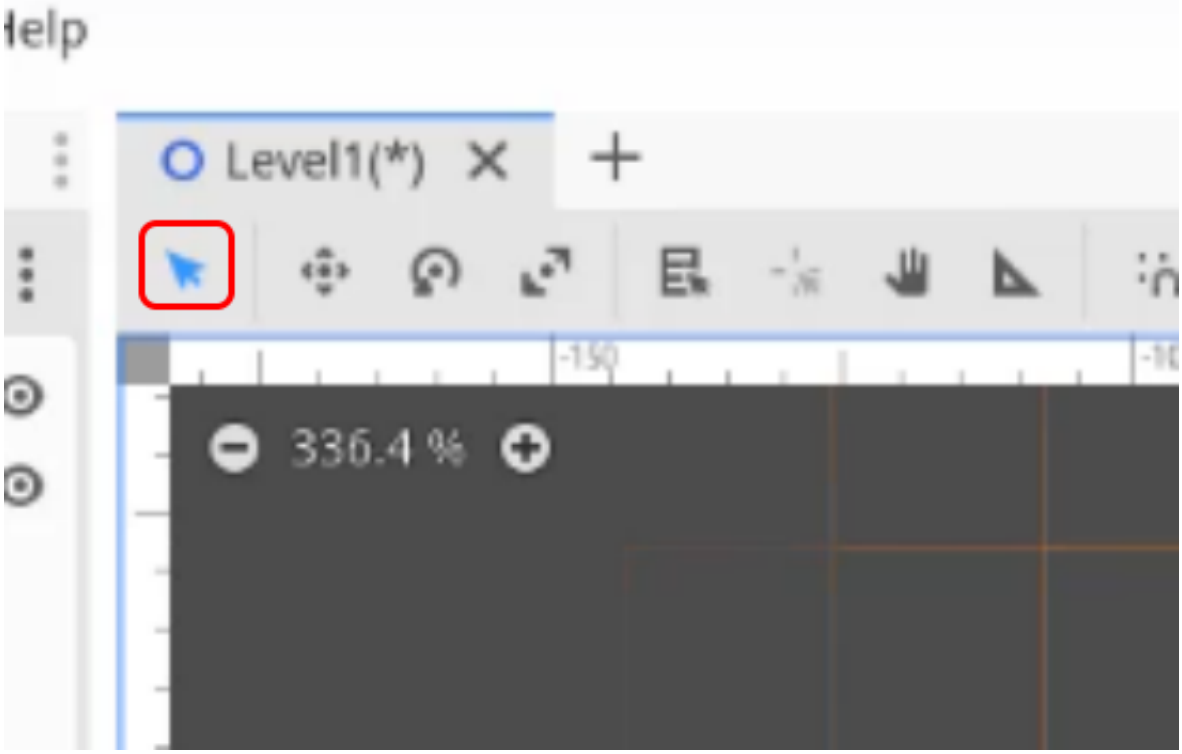


Painting the Level

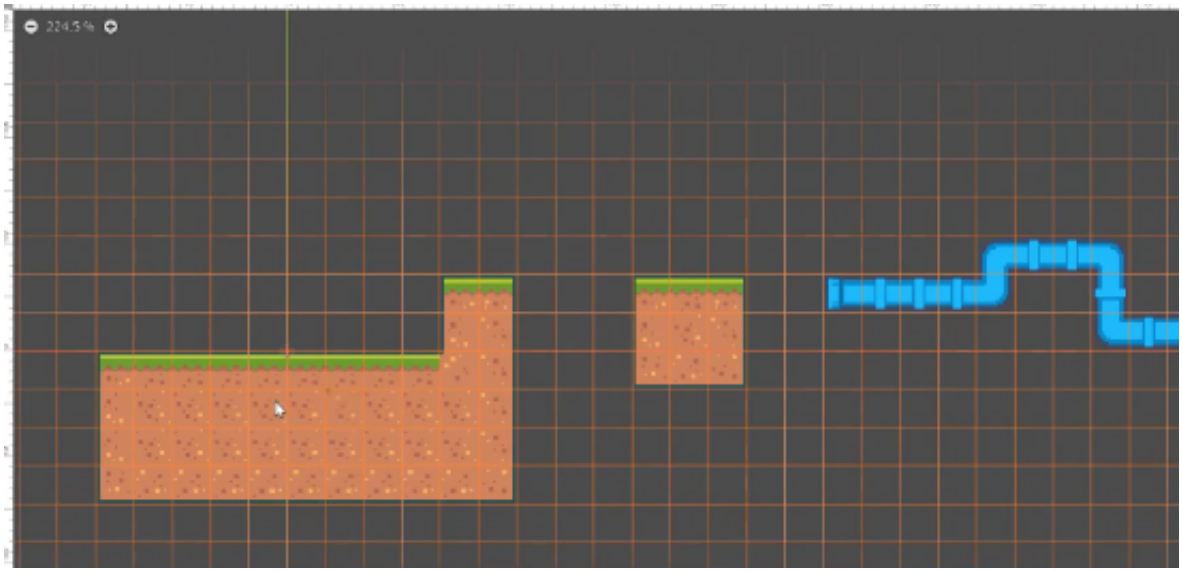
Now that the tile map is set up, we can begin painting the level. To do this, open the **tile map** window.



Then chose the **Select** tool in the editor (or use the shortcut *Q*), as this is the tool needed to draw our tiles on the *Tilemap* node.



Now if you *Left-Click* on the grid in the editor you can place tiles, or *Right-Click* to remove tiles. You can then select the tile you want to draw with from the *Tiles* panel and use them to construct the level.

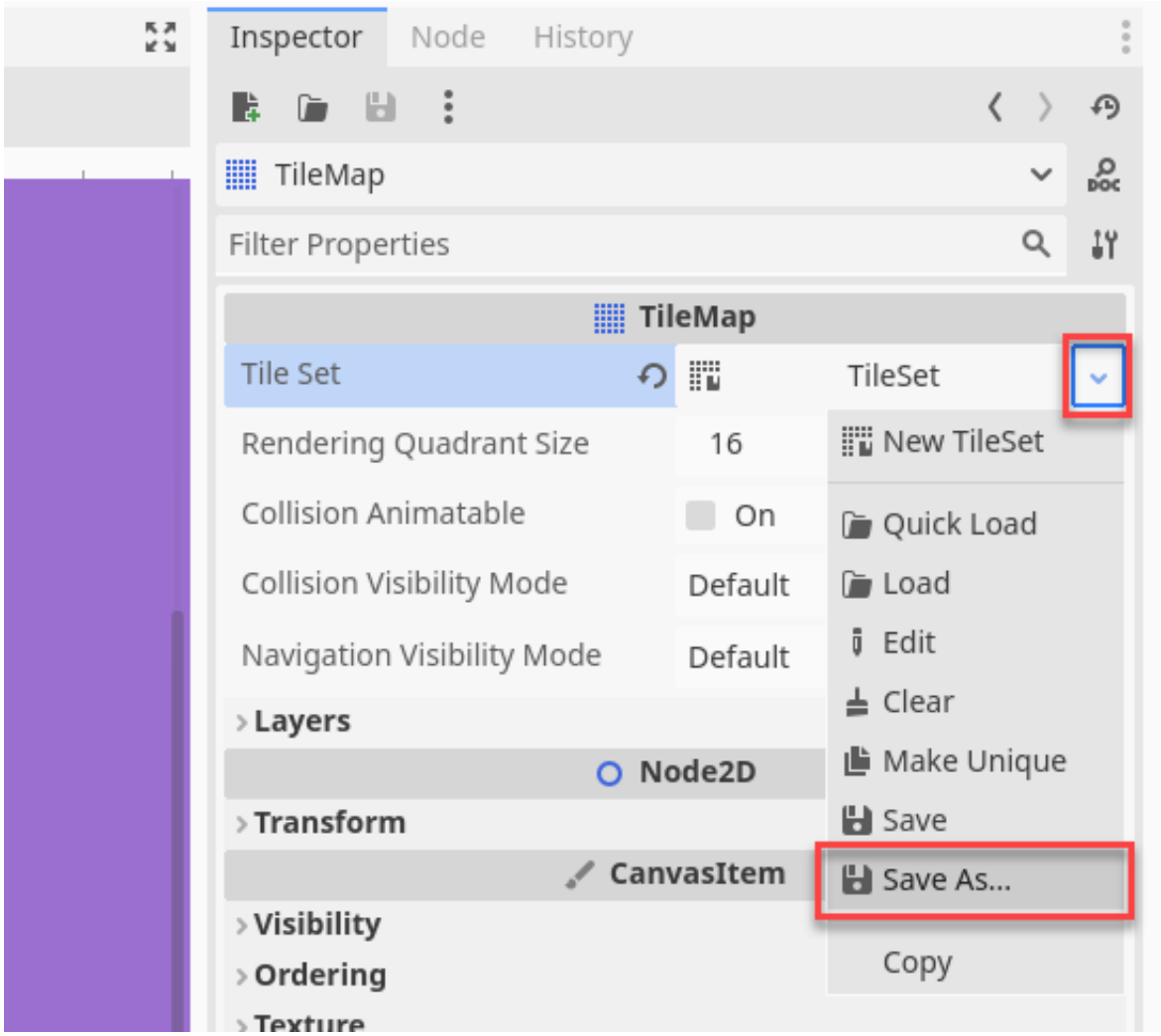


In the next lesson, we will look at setting up our *Player Controller* so that we have a player that can walk around our level.

Multiple Level TileSet Fix

In Godot, there is a bug where copy and pasting a tile map causes the TileSet of the original to break. To fix this, we can save our current TileSet as a resource and use that for each new tile map we create.

1. Select the TileMap
2. In the Inspector, click on the dropdown arrow next to the **Tile Set** property.
3. Select **Save As...** and save it to the FileSystem.

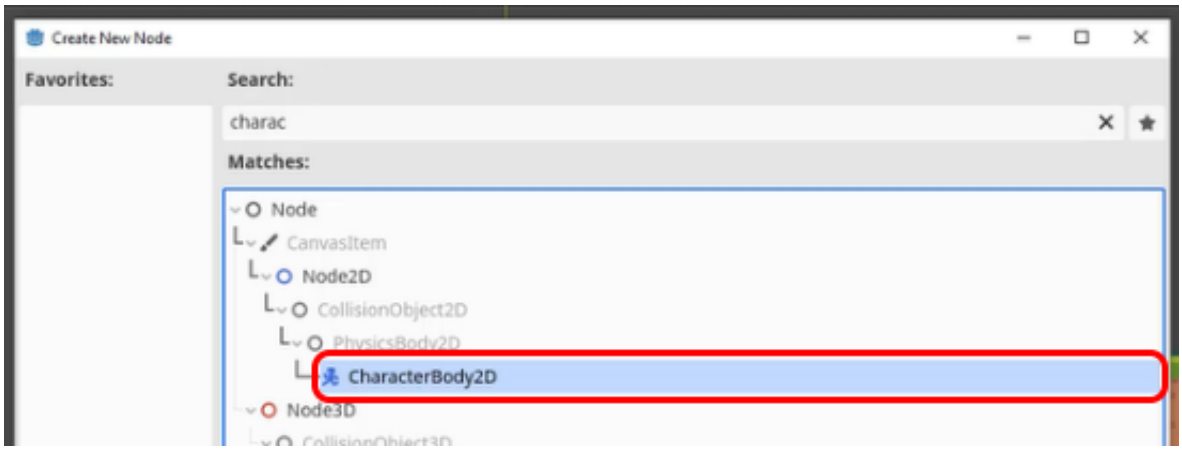


Now when you create a new tile map, you can use that saved TileSet and when you duplicate a tile map, no error should occur.

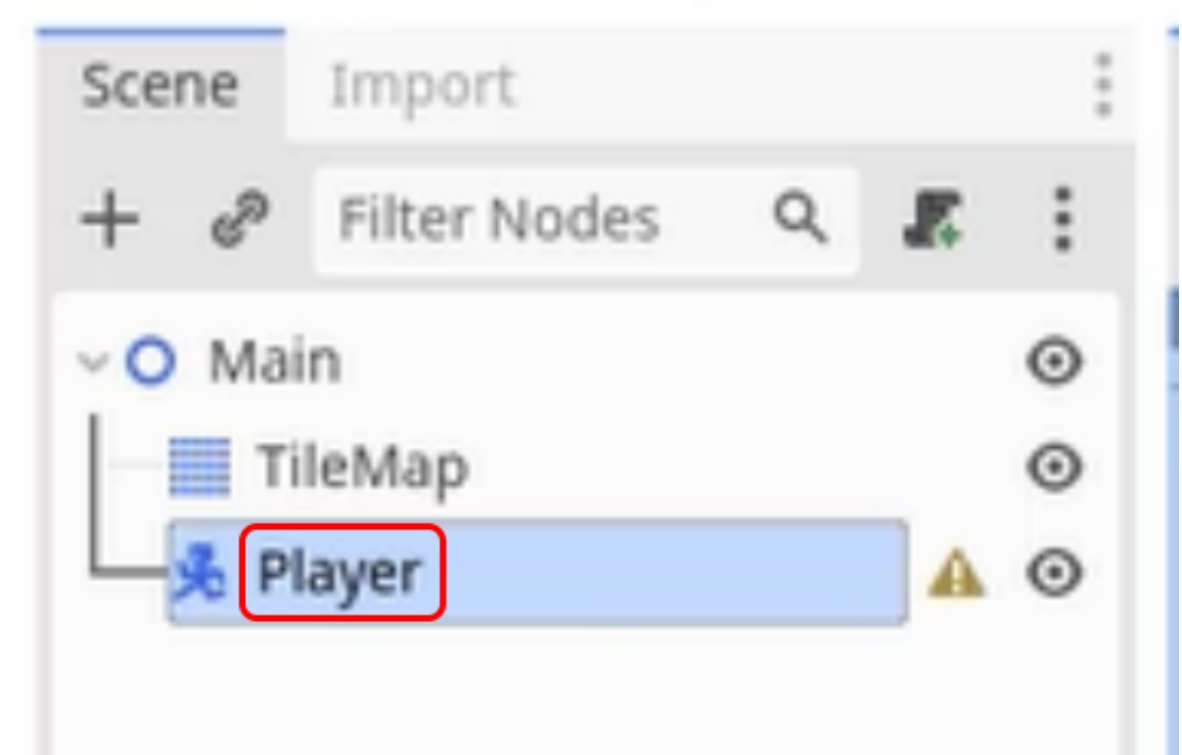
In this lesson, we are going to be setting up our player controller. It will allow us to give it a velocity input that will automatically move our player throughout the world. It will also take into account grounding the player, making sure they won't fall for any obstacles.

Creating a Node

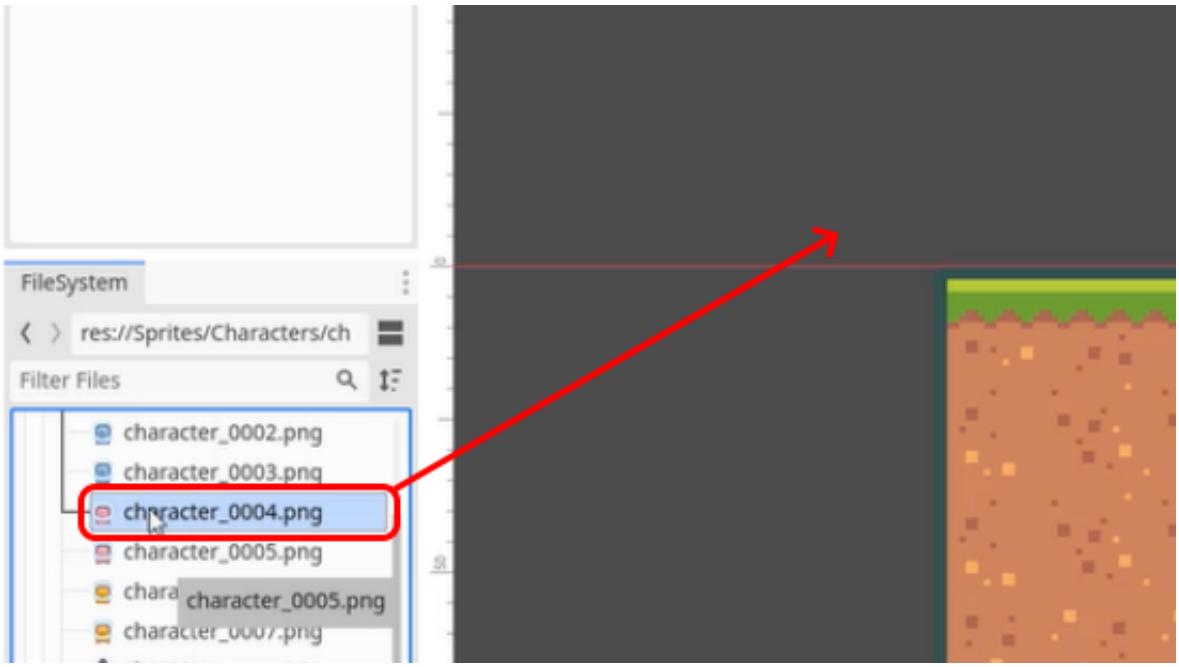
To begin, we are going to be creating a **CharacterBody2D** node by clicking the **plus** button in the *Scene* tab.



This can then be named *Player*.

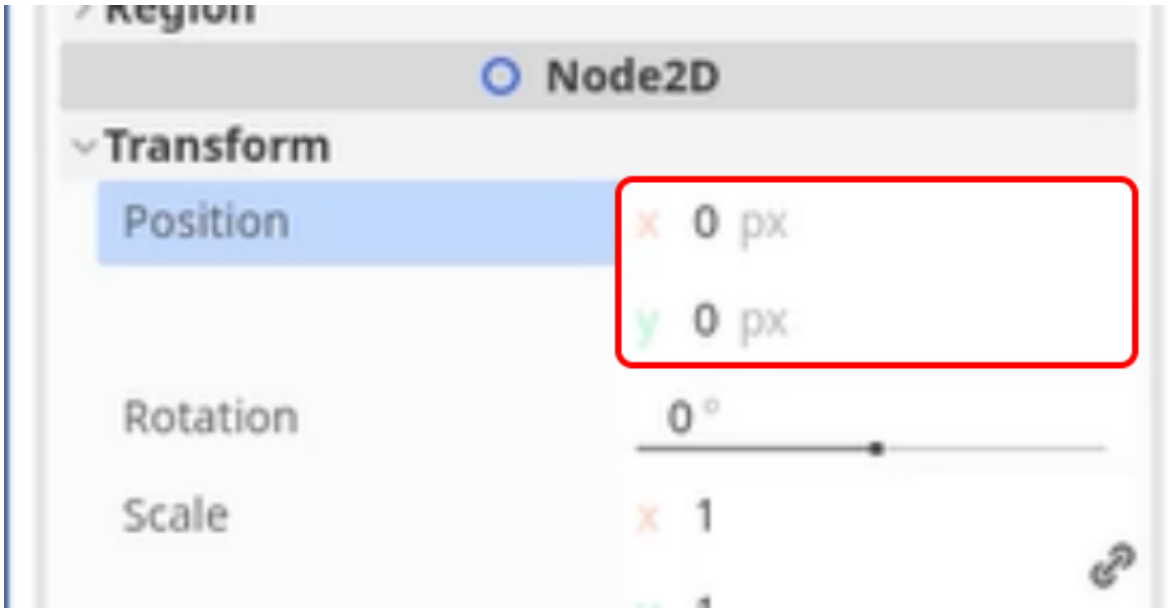


We need to give our player a *Sprite*. We can do this by selecting one of our character Sprites and dragging it into the scene to create a *Sprite* node.

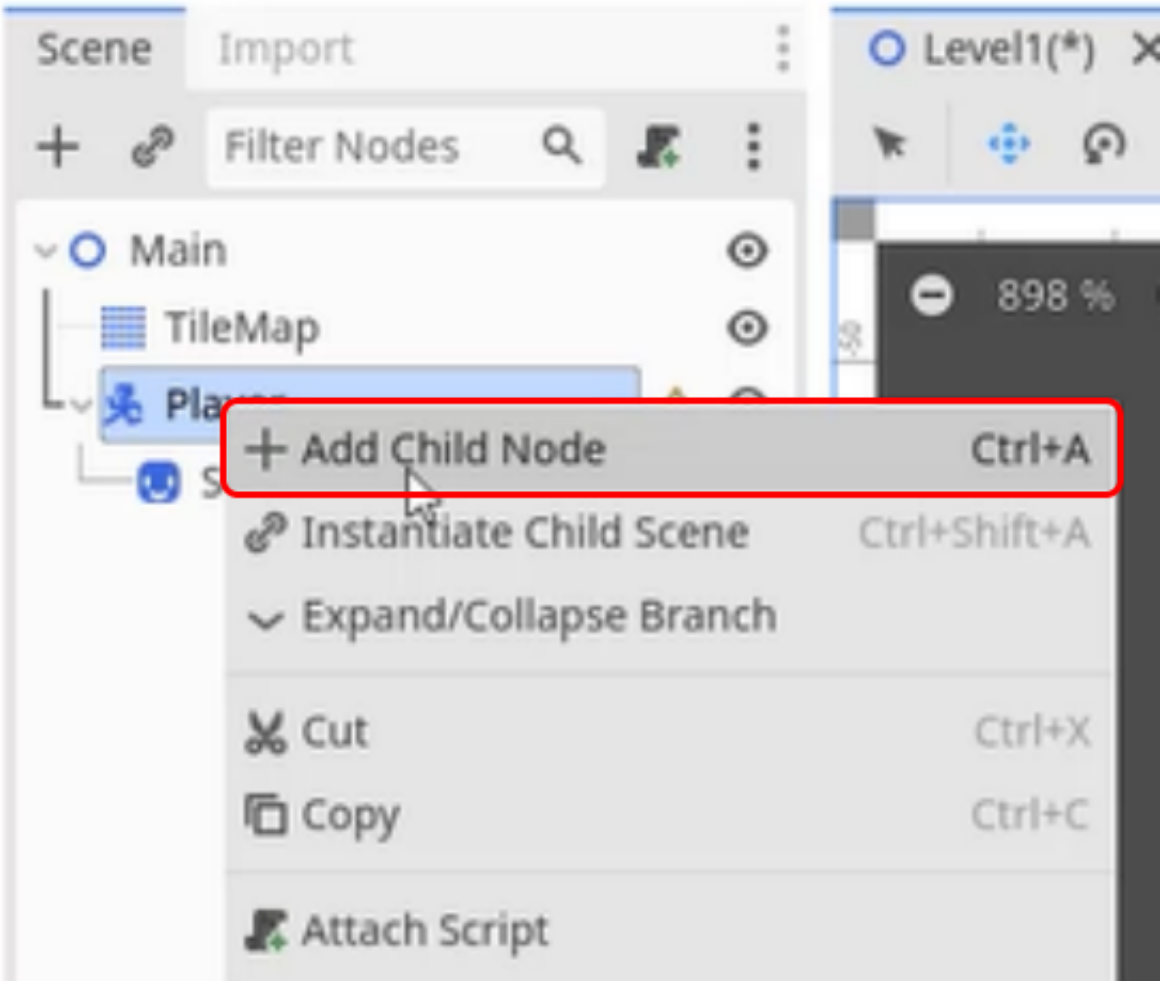


We then need to make this sprite a **child** of the *Player* node (by dragging and dropping the node on top of *Player* in the Scene view) and rename it to *Sprite*.

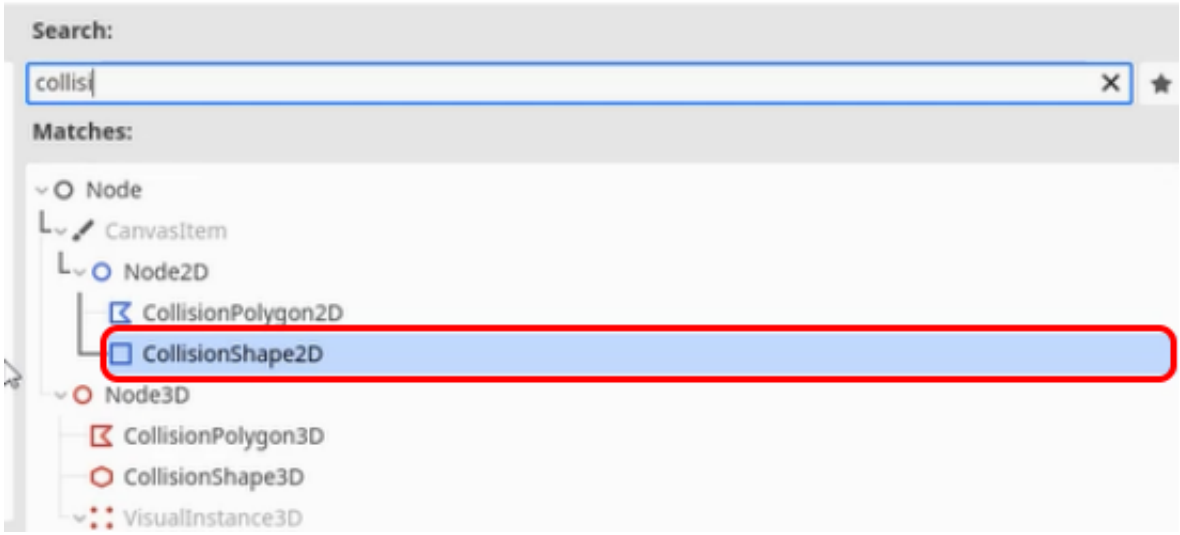
Then to center the sprite, set the sprite's **Position** value to **(0, 0)** in the *Inspector* tab.



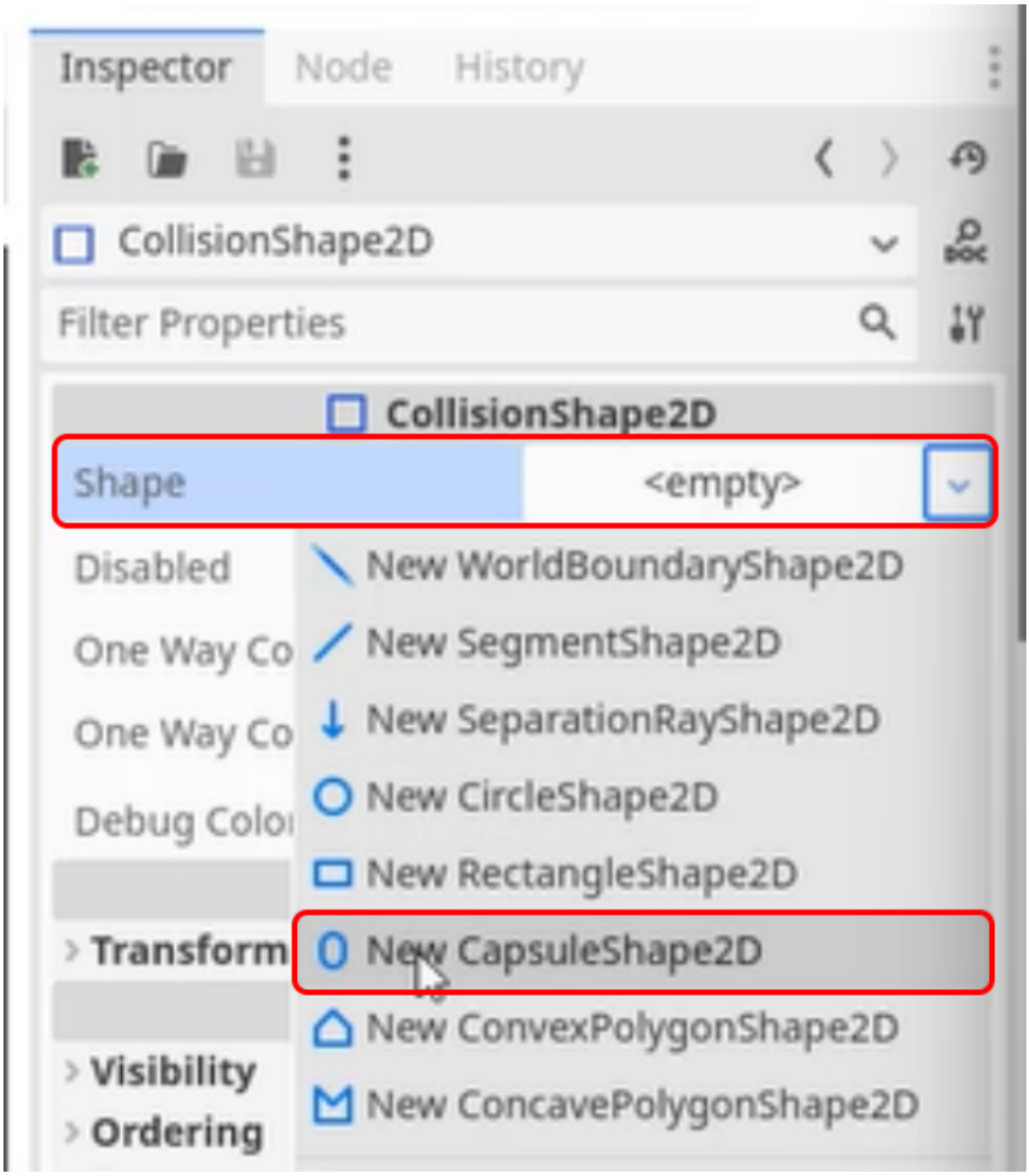
We then need to add in a collider. To do this, *right-click* on the *Player* node in the *Scene* tab and select **Add Child Node.**



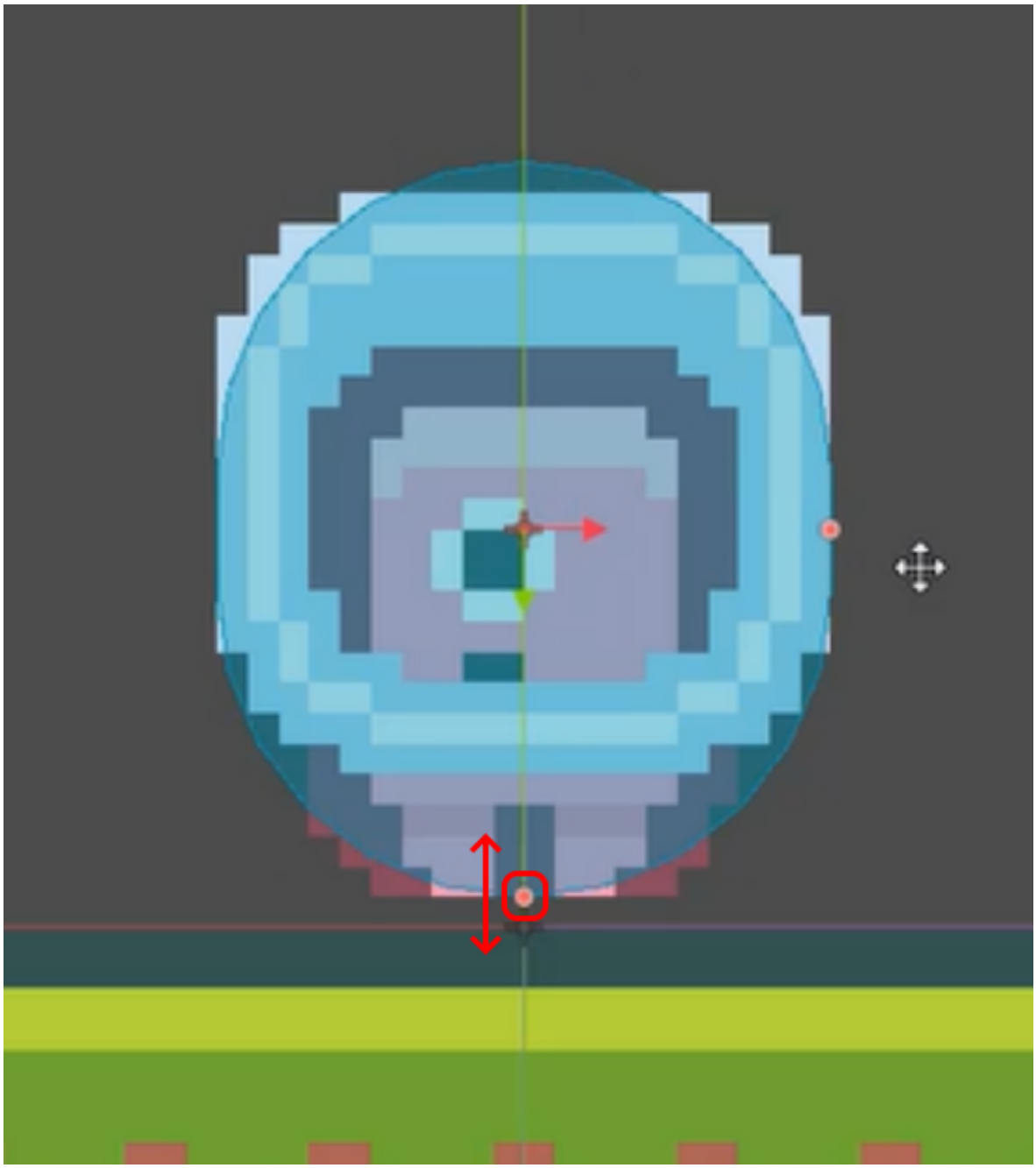
We will then add a **CollisionShape2D** node.



We can then go to the **Shape** value in the *Inspector and* create a capsule.

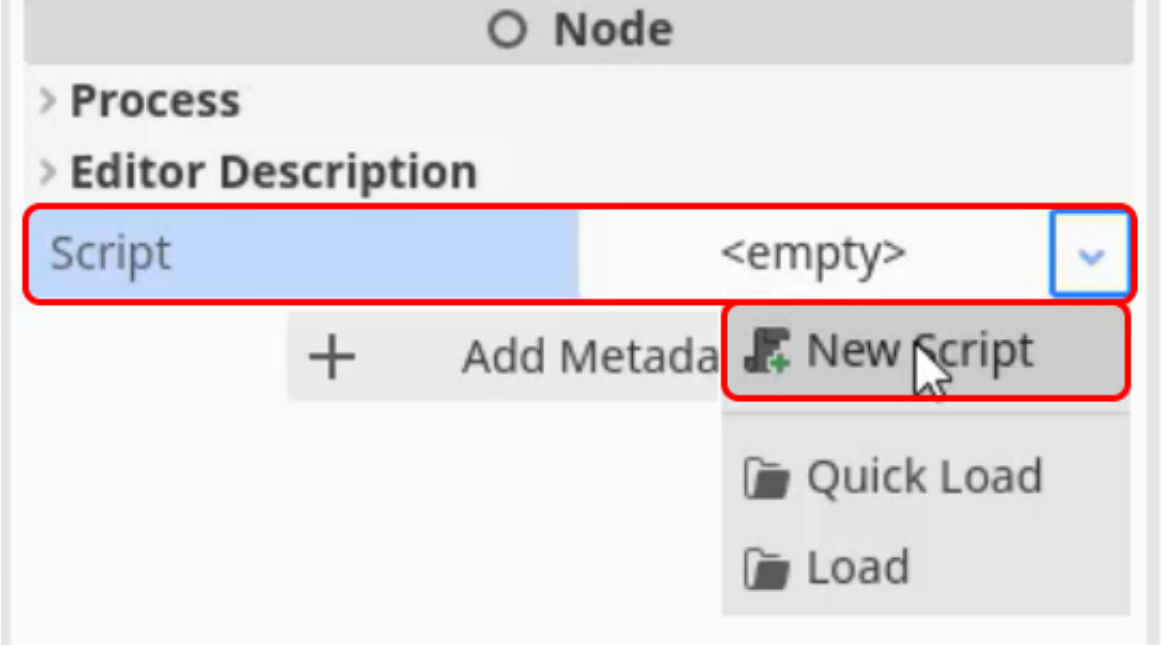


Then use the orange circles on the new blue capsule in the scene view, to scale the capsule to the size of the *Sprite*.

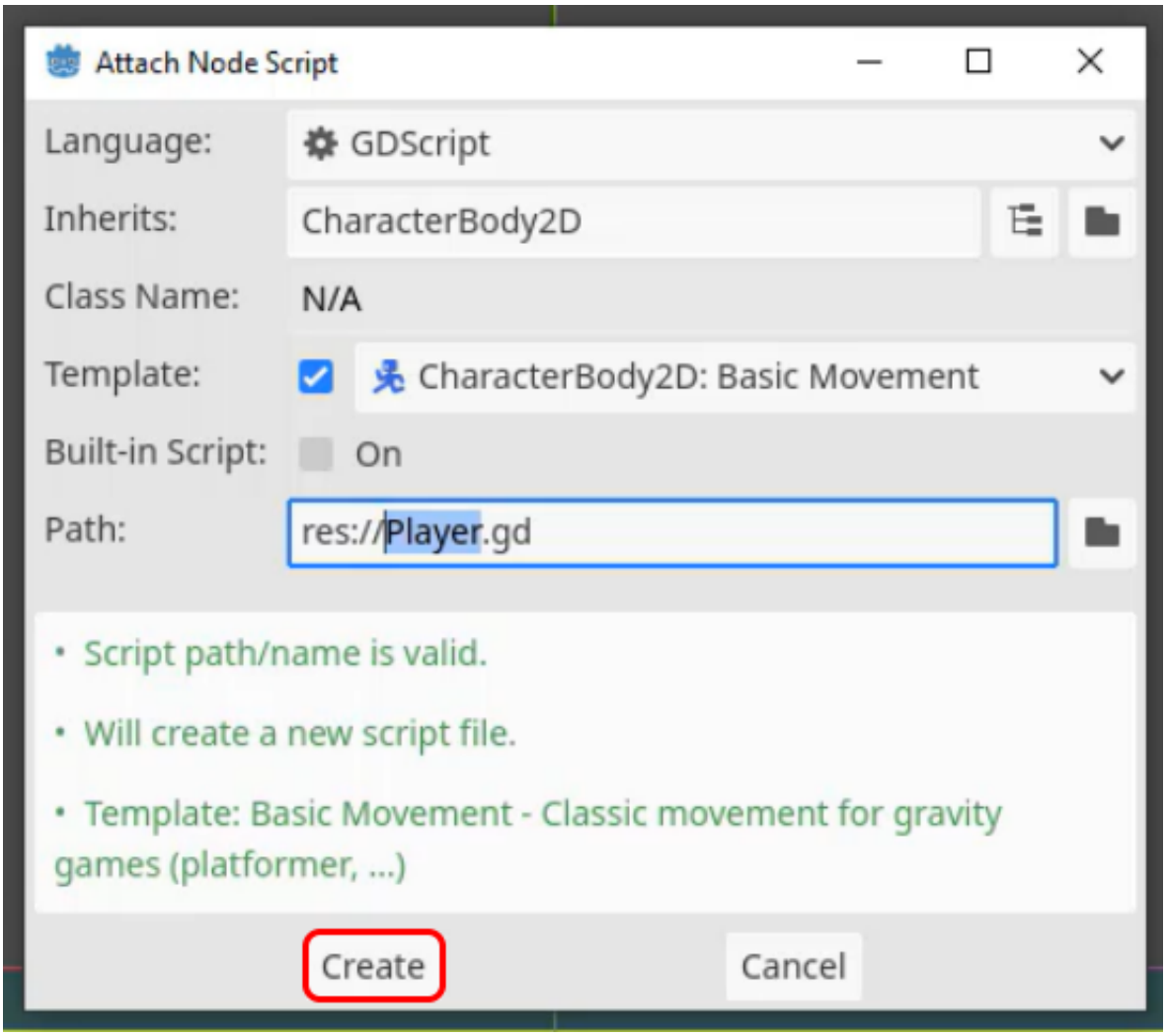


Setting up our first script

We then need to create a script that is going to give our player the ability to move around and jump. To do this, in the *Inspector* tab, choose **Create Script** under the **Script** property.



The default values should be fine, just ensure the script is called *Player.gd* and that it inherits from *CharacterBody2D*. Then press the **Create** button.



There may be some default code in the editor, but for this lesson, we will select all of it (apart from the first line) and delete it. This should leave you with the following code:

```
extends CharacterBody2D
```

This line will give us the ability to communicate with our **CharacterBody2D** node. Firstly, we will need to create some variables. The first will be our *move_speed* variable with the line:

```
var move_speed : float = 100.0
```

We gave the script a value of 100.0 which means our character will move at 100 pixels per second.

For the next line, we will also add a variable called *jump_force,* which will also be of type *float,* using the line:

```
var jump_force : float = 200.0
```

This value will be the upwards velocity we add when the user presses the jump button.

We will also create another variable called *gravity,* with the type of *float,* using the line:

```
var gravity : float = 500.0
```

This value will be the downward force we use as our gravity, to combat the *jump_force* value.

Physics Process Function

We are going to create a function called *_physics_process*. The physics process function is generally where you want to run the physics code that you want to check every frame. This function gets run at a consistent rate per second. To do this we will add the function to our code:

```
func _physics__process(delta):
```

In this function, we are going to be applying gravity. To do that, we first want to check to see if the *Player* is not standing on the floor by using the *is_on_floor* function. Then, if we are not standing on the floor, we want to modify our vertical velocity by our gravity value.

```
func _physics_process(delta):
    if not is_on_floor():
        velocity.y += gravity * delta
```

Multiplying by the *delta* value above will convert a per-frame action, into a per-section action.

We then want to have the ability to move left and right. We will being by setting our X velocity to zero, and if we are pressing the left key, we want to set our velocity to be negative move speed. If we are pressing the right key, we then want to set our velocity to be positive move speed. This can be done using the following code:

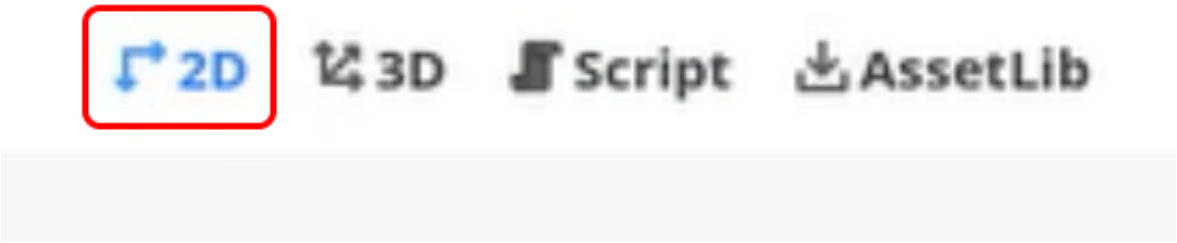
```
func _physics_process(delta):
    ...
    velocity.x = 0

    if Input.is_key_pressed(KEY_LEFT):
        velocity.x -= move_speed
    if Input.is_key_pressed(KEY_RIGHT):
        velocity.x += move_speed
```

Finally, we need to apply the velocity to the *CharacterBody2D* node so that Godot can process our movement. This will be done using the *move_and_slide* function that will move our player based on the velocity and slide across objects like ramps.

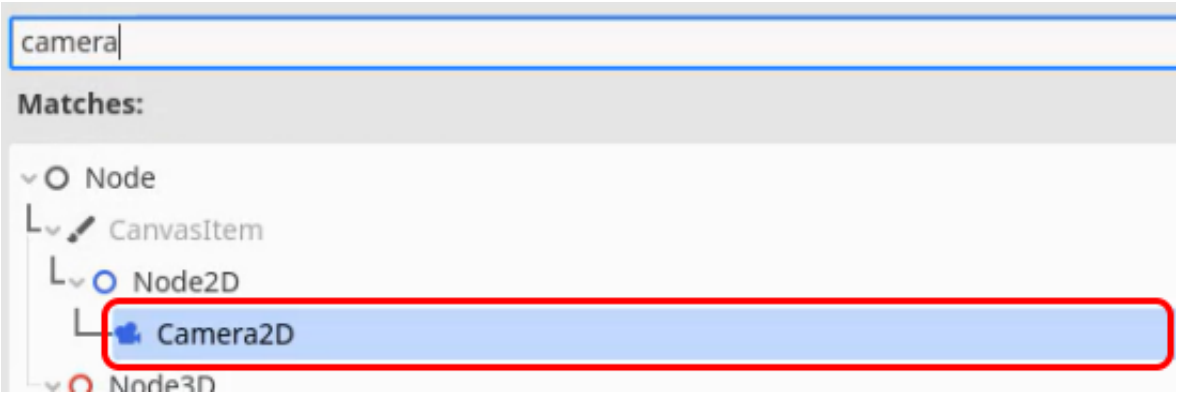
```
func _physics_process(delta):
    ...
    move_and_slide()
```

Testing the Movement

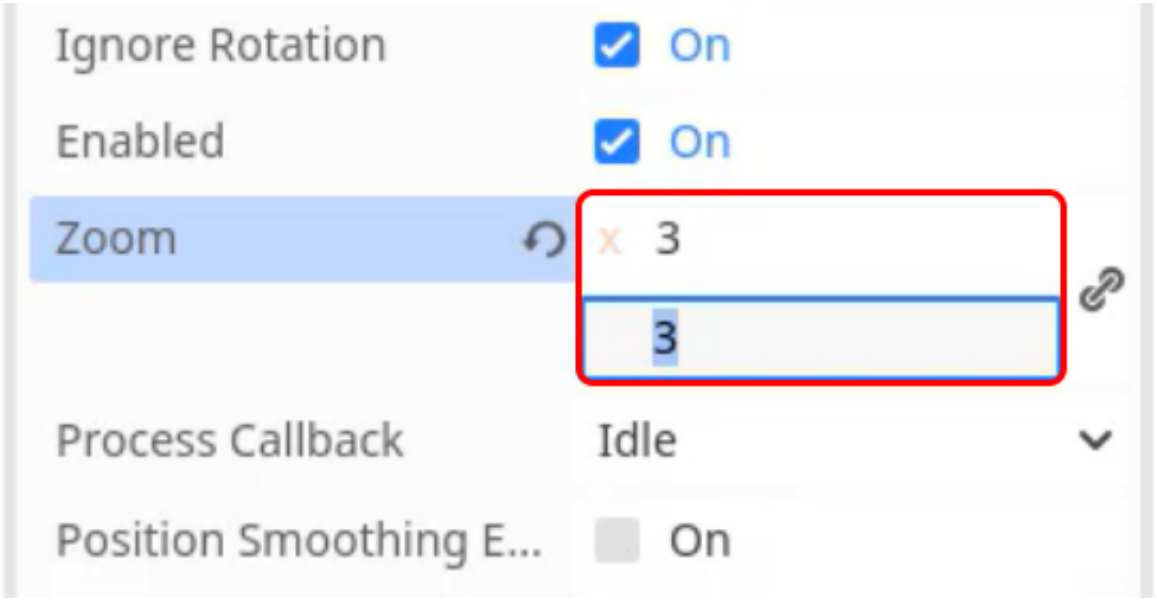
We can then save this script (**CTRL + S**) and return back to our 2D scene view.



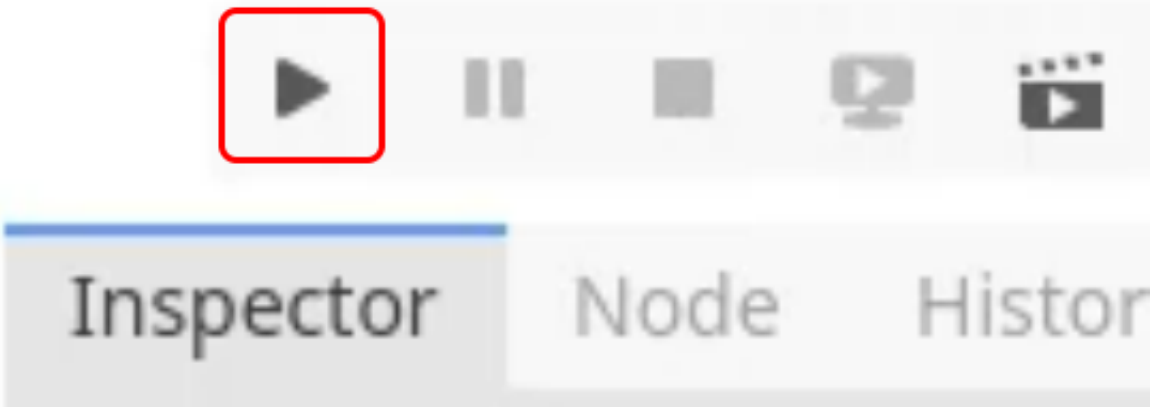
If you test the game you will notice we can't see anything, this is due to our camera isn't showing what we want to render. To fix this, we are going to add a new node of **Camera2D**.



We can then zoom in a bit by changing the **zoom** property to **(3, 3)**.



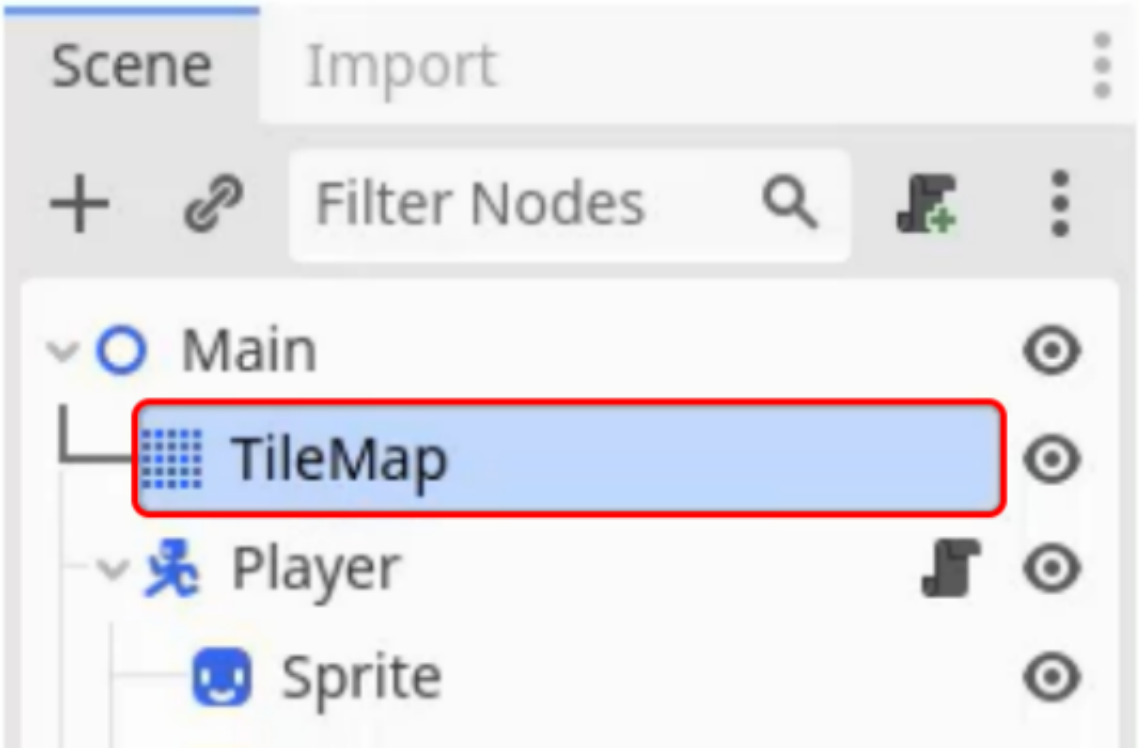
Now, we can press the **Play** button, and we should be able to see our character. If you get a pop-up when pressing play, choose the *Select Current* option.



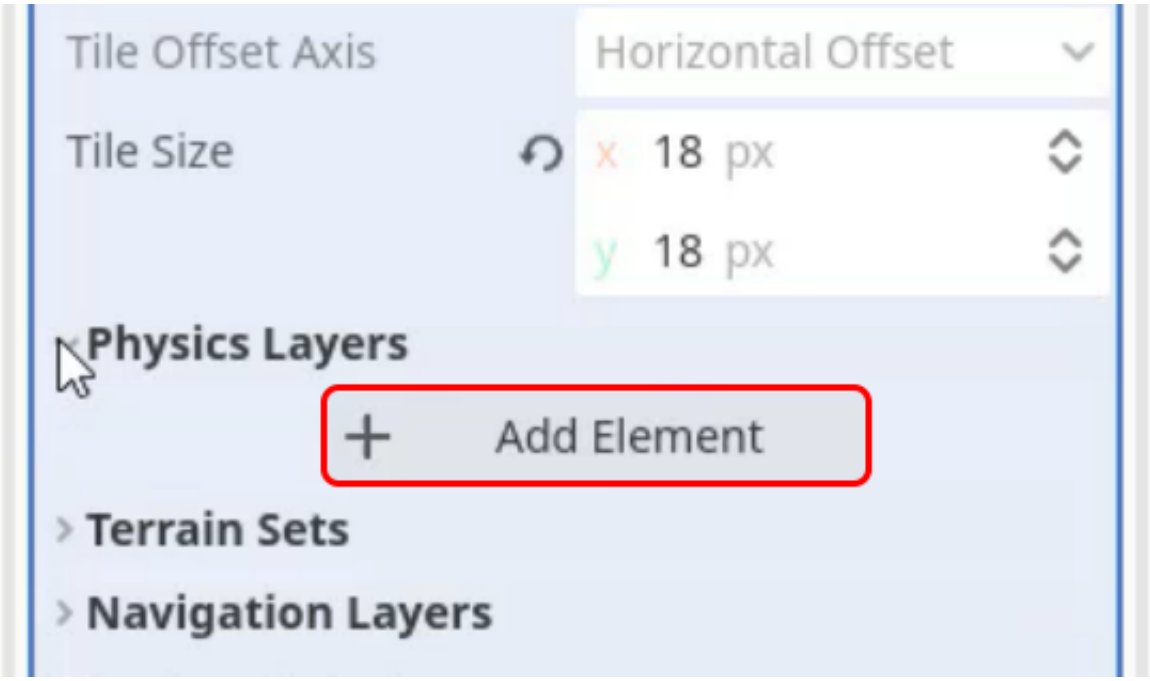
When you test the game you will notice our player falls through the ground. Don’t worry, this is an easy fix.

Adding Ground Collisions.

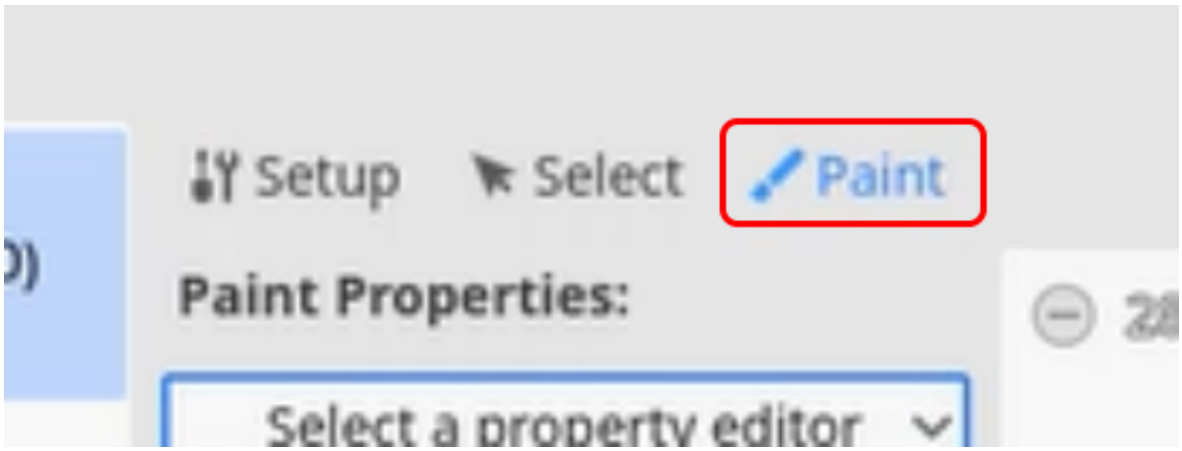
Firstly, we want to select our **TileMap** node in the *Scene* tab.



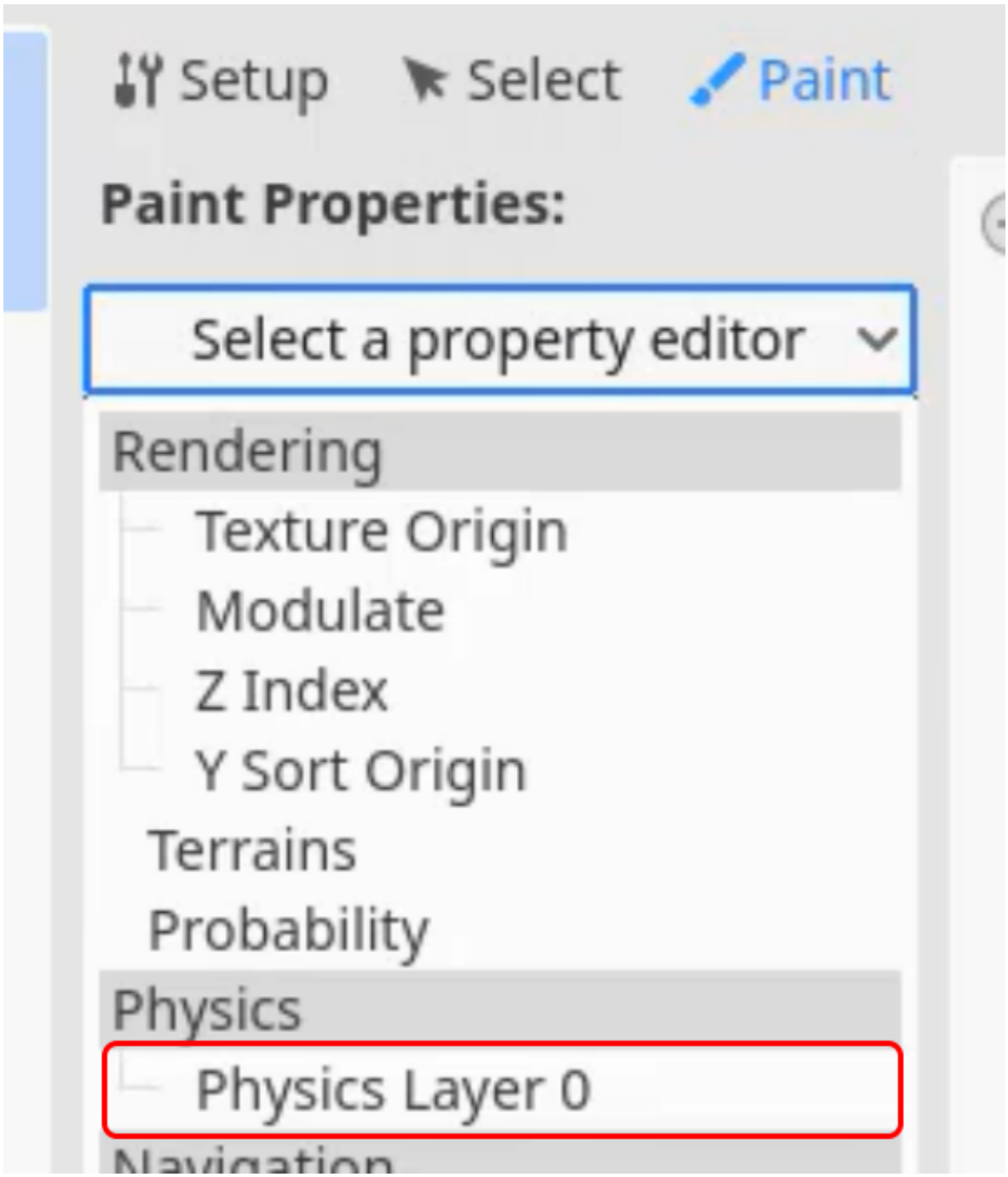
We then want to open the **Physics Layers** tab in the *Inspector* and press **Add Element**.



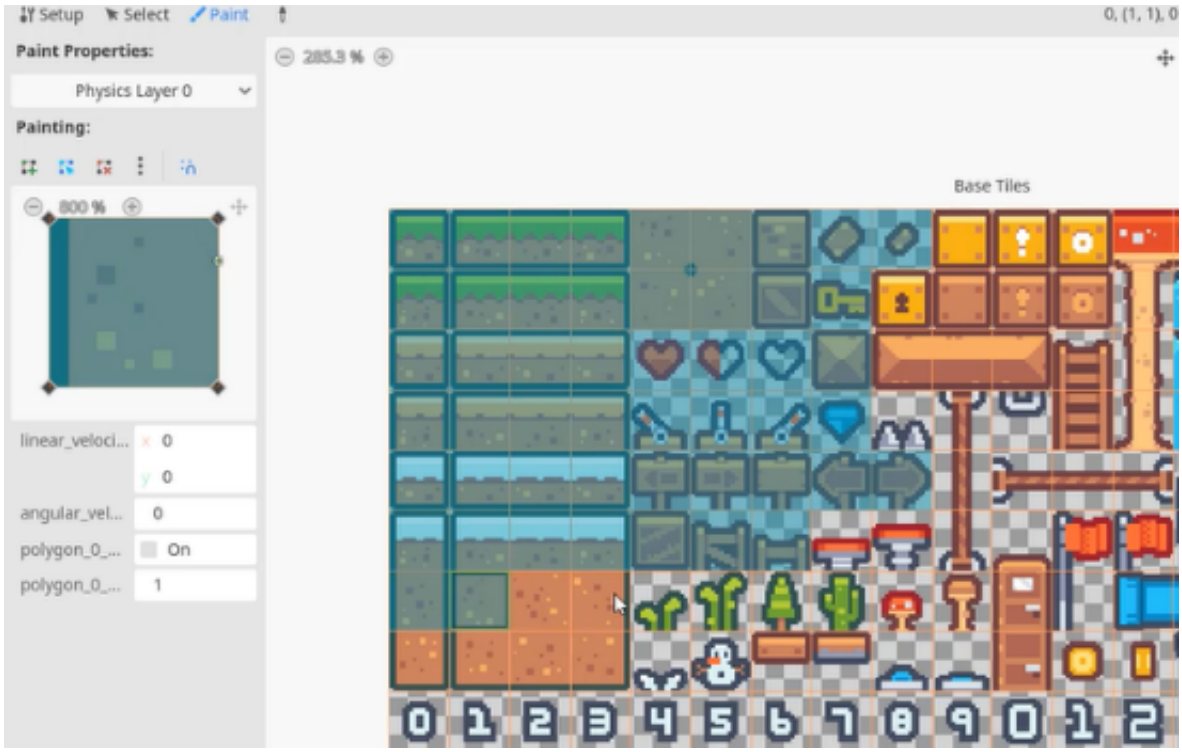
We then need to add this physics layer to our tiles. In the *Tiles* tab, select the **Paint** option.



Then choose the **Physics Layer 0** option in the *Select a property editor* option.

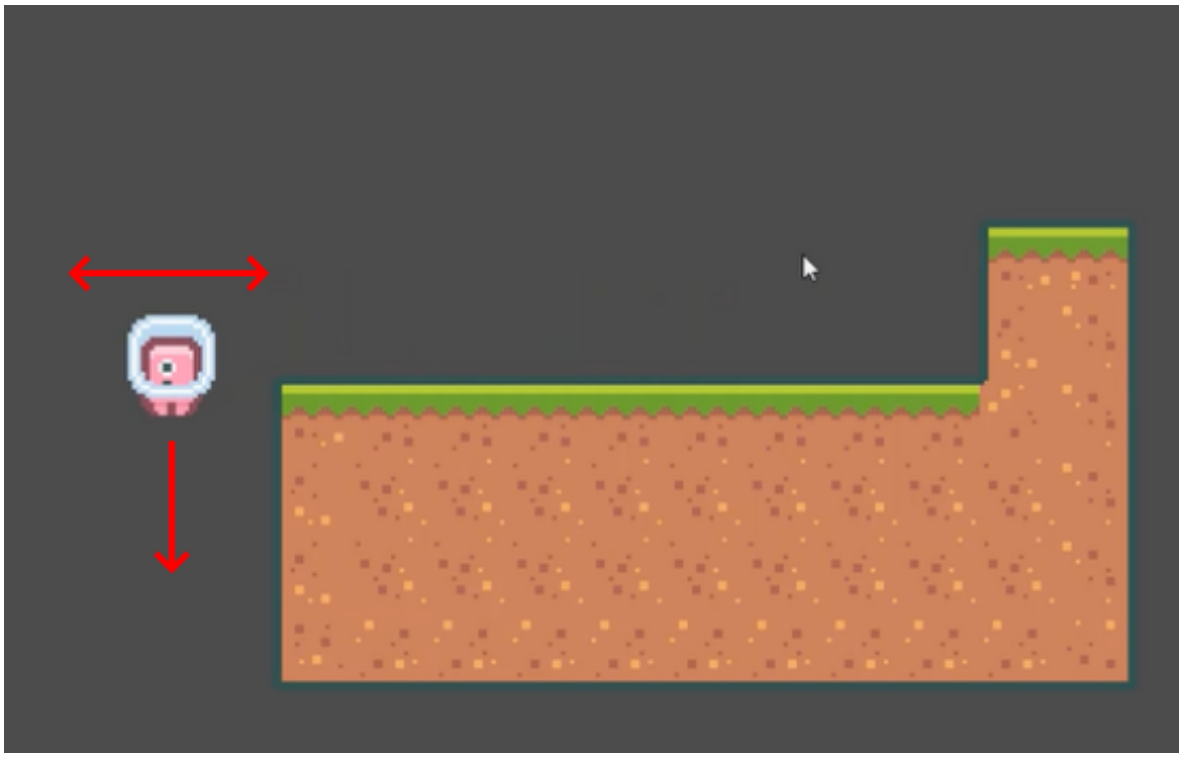


Then we can go through and “paint” all of the tiles that we want to have collisions with, using the left mouse button.



Testing the Game

Now when you press **Play** you should be able to move left and right using the keys we set up earlier, and gravity will affect the *Player* if it falls off the edge of the terrain we made.



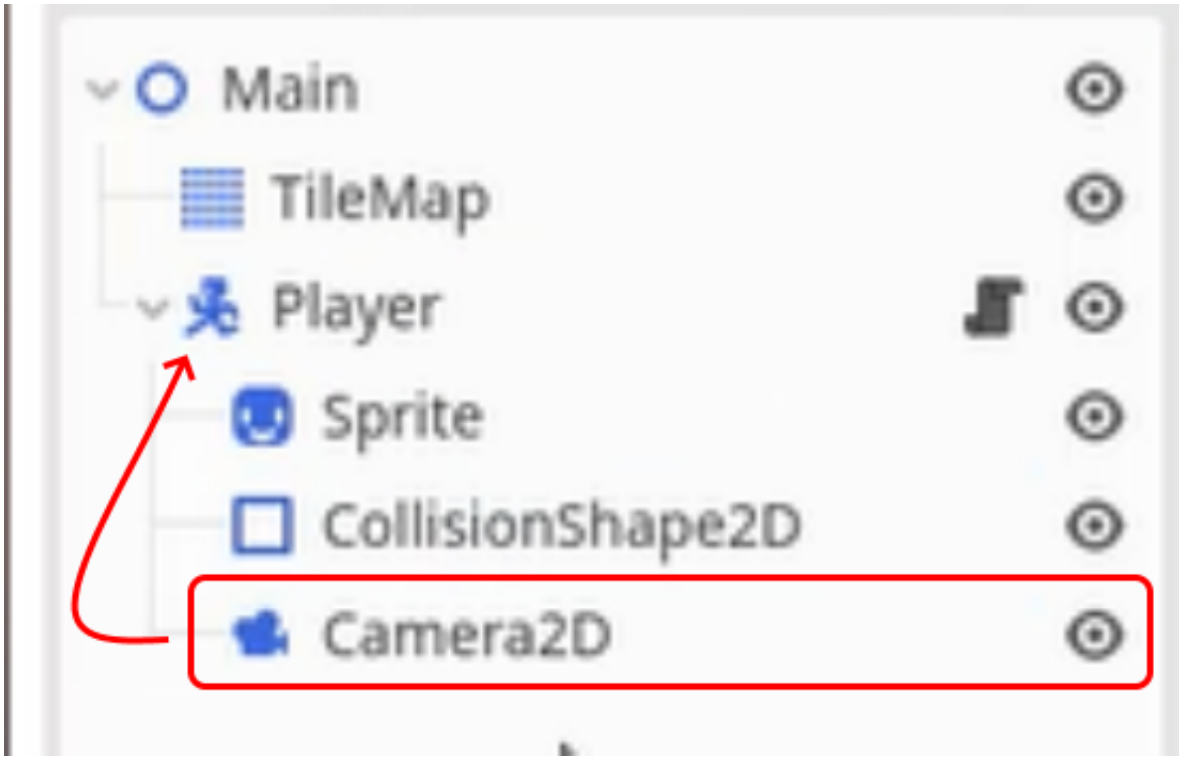
Adding Jumping to our Player Script

As well as moving and falling, another important section of a platformer game is jumping. To do this,

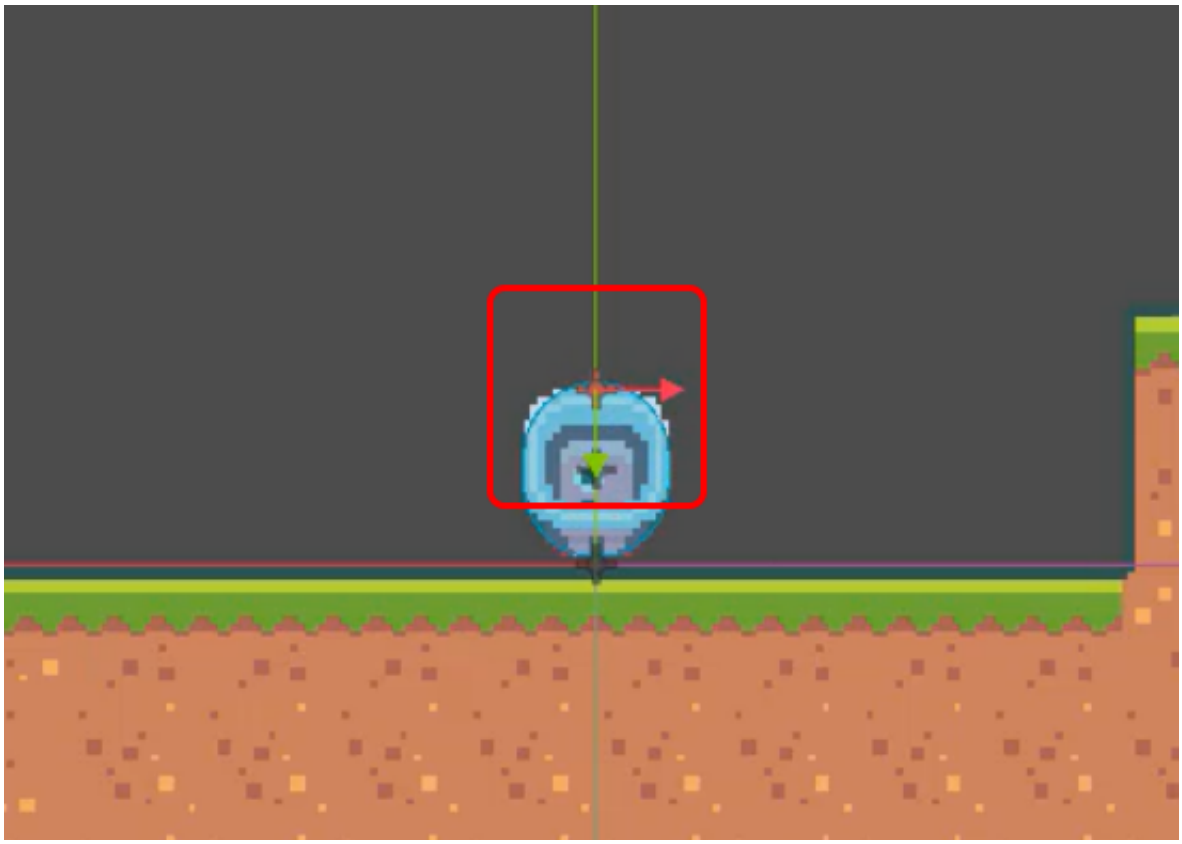
we will reopen our *Player* script and add the following lines above the *move_and_slide* function call.

```
func _physics_process(delta):
    ...
    if Input.is_key_pressed(KEY_SPACE) and is_on_floor():
        velocity.y = -jump_force
        move_and_slide()
```

This will add our jump force to the player (negative to say going upward) if the player presses the space bar and is on the ground. We also want our *Camera* to follow the *Player*, to do this, just drag and drop the **Camera** node as a child of the **Player** node.



You can also position the **Camera** to be just above the **Player Sprite** using the arrow tools in the editor window.



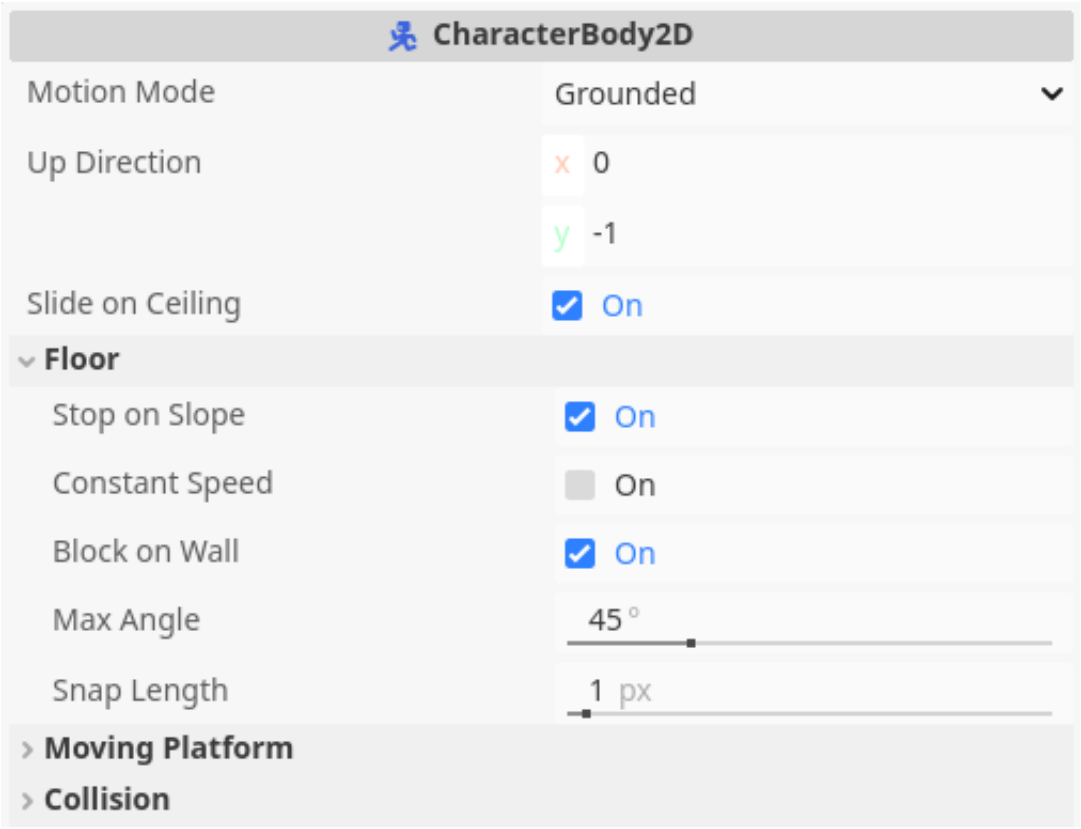
Now, you can press space to jump around your level, use the arrow keys to move left and right, and you will fall due to gravity, all from this one simple *Player* script.

In the next lesson, we will set up the ability to have enemies and falling set off our game over screen.

The **CharacterBody2D** node is what we use as the foundation for our 2D characters. This can be used for both the player and any NPC's you wish to add in the future. This node takes care of the more complex features when it comes to creating a character.

In the Inspector, there are many properties we can modify. Here are some that you might want to modify in the future:

- **Motion Mode** – This determined if the character is to be used in a platformer style game (Grounded), or in a top-down style game (Floating).
- **Max Angle** – This is the maximum angle grade that the character can walk along before sliding. So by default, any slope greater than 45 degrees will cause the character to slide down it.



Scripting

When we create a script and attach it to the CharacterBody2D node, you will notice that it extends from that class type.

```
extends CharacterBody2D
```

This means that we have the functionality to access variables and call functions associated with the CharacterBody2D, which is how we can move our player around, check to see if we are grounded, etc.

Variables

In the previous lesson, we utilized a number of variables associated with the CharacterBody2D:

- **velocity** – This is a 2D vector that determines the direction and distance that we wish to travel every second. E.g. a velocity of (50, 0) means we want to travel to the right at 50 pixels per second. We assign the velocity variable and then the CharacterBody2D node takes care of the rest.

Functions

We also called two different functions associated with the node:

- **is_on_floor()** – This returns true or false for if the character is standing on the ground. We check this in two places: first, when we apply gravity, we only want to do so if we are in the air. Secondly, when we jump. The player can only jump if they are standing on the ground.
- **move_and_slide()** – This function is what we call to apply the *velocity* to our character. It will move us, check for collisions, etc. That is why we call it at the end of the _physics_process function.

Additional Resources

If you wish to learn more, you can refer to the Godot documentation:

- CharacterBody2D

The CharacterBody2D node utilizes Godot's physics system. There are two main aspects we are going to explore in this lesson that will hopefully help you in understanding it better.

Physics Process Function

When writing scripts in Godot, there is the **_process** function which gets called once *every* frame. This is used to run processes over time, such as counters, movement, transitions, condition checks, etc. But as well as this, you will have also noticed another function called **_physics_process**. It uses the same parameter and acts very much in a similar way, but these two functions vary in one important way.

Where **_process** runs every frame (30 times at 30 fps, 200 times at 200 fps), **_physics_process** will run a consistent amount of times per second (60 times at 30 fps, 60 times at 200 fps).

Why? Because when calculating physics, consistency is key. So that is why we are running our player controller code inside of **_physics_process**. Because we are modifying velocities that are being applied to a node that interacts with the physics engine.

Collision Detection

Detecting collisions can be a complicated topic, but thanks to Godot, that problem has already been solved for us.

On our player, we use the **CollisionShape2D** node which allows us to define a collision shape for our player. This basically makes our player a physical and solid being in the world, where it can bump into other colliders to prevent it from falling into the void. In Godot, one of the more head scratching aspects of collision, would be the layers and masks section. What are these numbered boxes?

- Each number represents a collision **layer**. In a more complex game, you might want some colliders to interact with some other colliders, but not all. For our game though, it's not a problem.
- **Layer Section** – This describes the collision layer that the object appears in. By default, all bodies are on layer 1.
- **Mask Section** – This describes what layers the body will check for collisions. If you want this object to ignore certain layers, you can disable them here.

Additional Resources

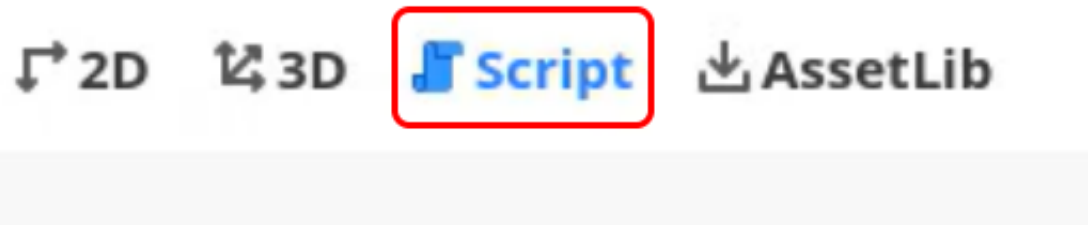
If you wish to learn more, you can refer to the Godot documentation:

- Physics Process Function
- Collision Detection

In this article, we will go over how to set up a game over state in a 2D game. This state will occur when our player hits spikes, enemies, or anything else that hurts them.

Setting up the Game-Over Function

The first step is to create a game-over function in the player script. To do this, return to the **Script** tab, and make sure the *Player.gd* script from the last lesson is still open.



At the bottom of the script, add the new function using the following line:

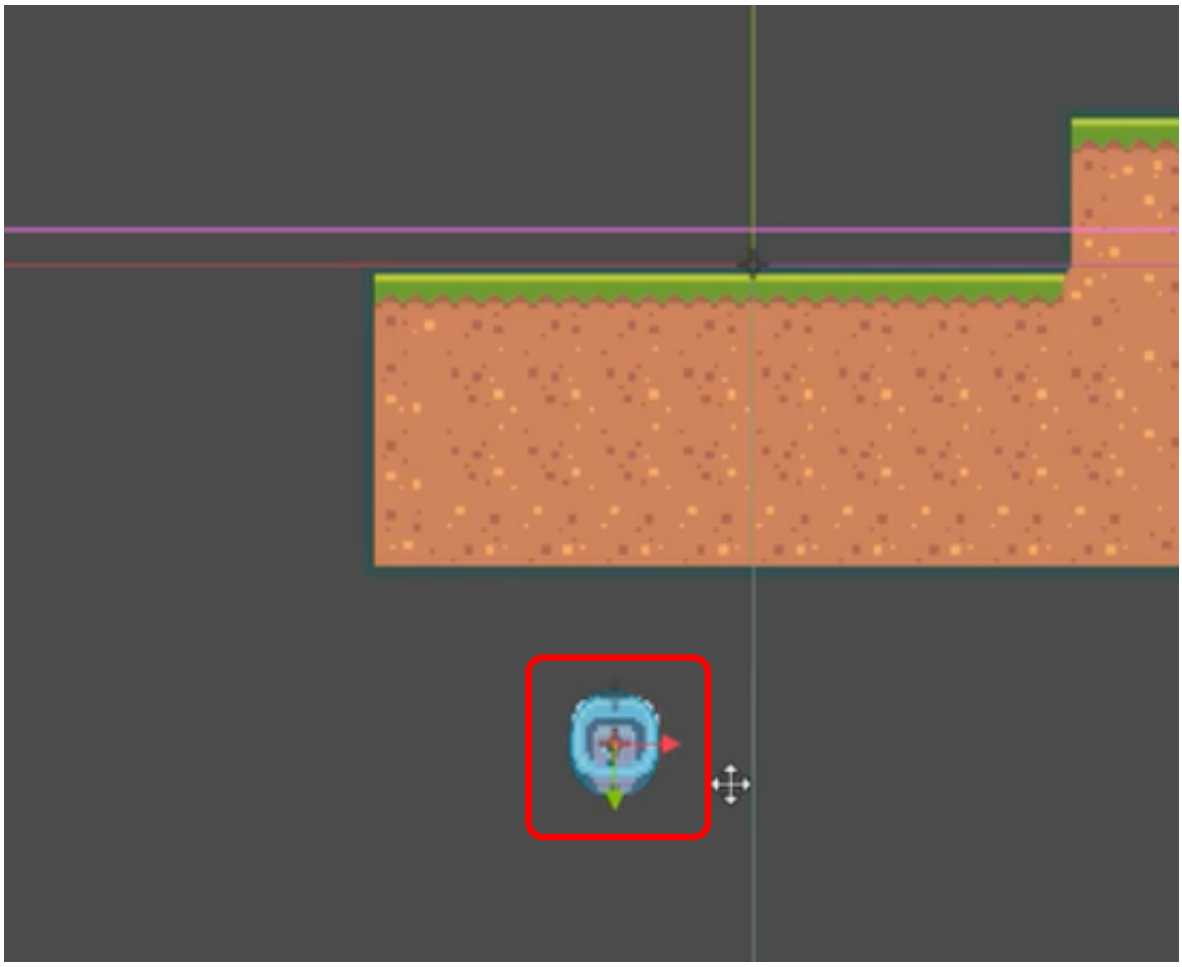
```
func game_over():
```

This function will simply reload the current scene. To do this, we need to get the node tree and then call the *reload_current_scene()* function on the tree.

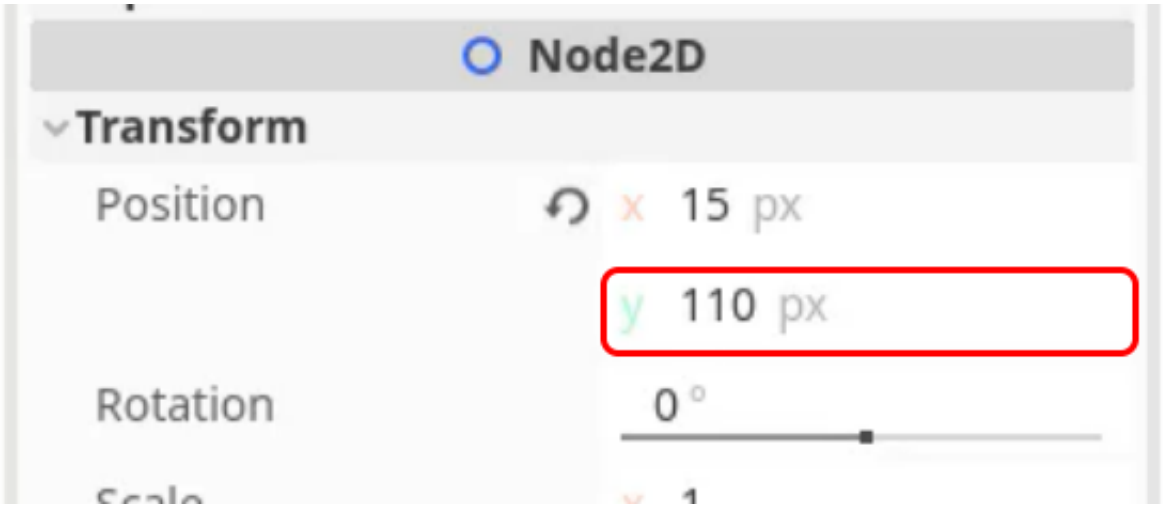
```
func game_over():
    get_tree().reload_current_scene()
```

Checking for Fall Deaths

The next step is to check the player's position, if they are below the terrain, we want to call the *game_over* function. The first step is to find a position under your world by dragging your *Player* to the lowest position they should be able to reach.



And check the **Y** position value to find the lowest position you want the player to reach. For us, we will be using **100** pixels or lower as our range for calling the *game_over* function.



We can then add an if statement to the **_physics_process** function. This statement will check the player's Y position and if it is **greater than 100** (or the value you found earlier), the game-over function will be called.

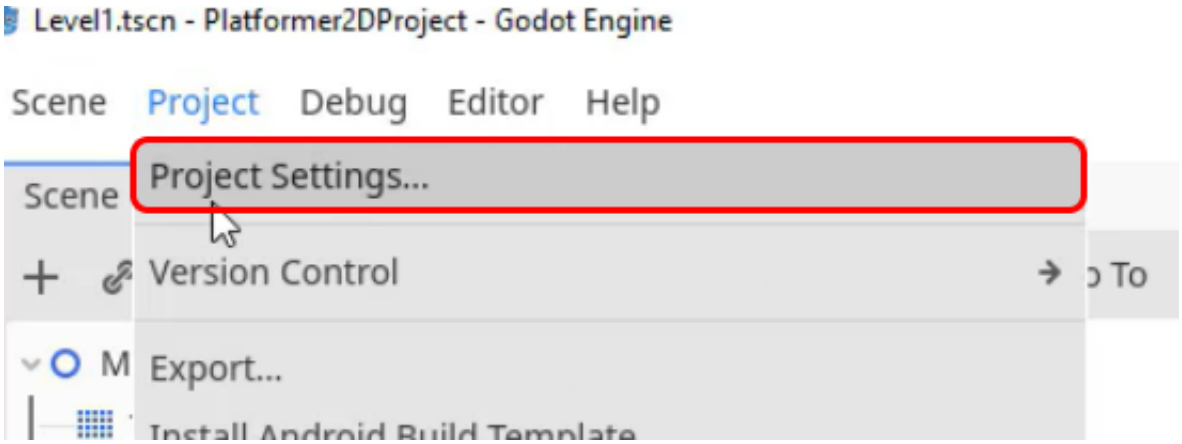
```
func _physics_process(delta):
    ...
```

```
if global_position.y > 100:
    game_over()
```

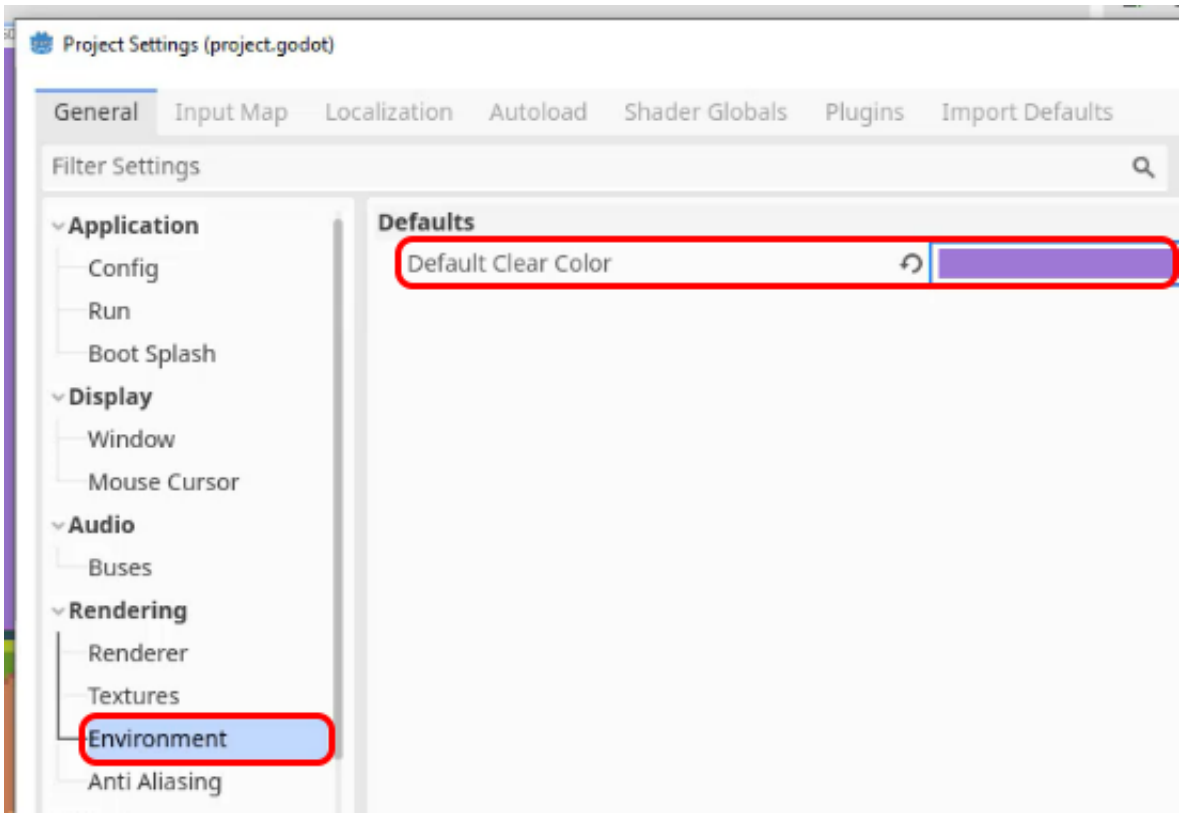
You can now press **Play** to test the game, and you will notice if you fall off the platform, the scene will be reloaded.

Changing the background

Finally, we can change the background of the game. To do this, open up the **project settings** window.



Then open the **Environment** tab (under *Rendering*), and we can then change the **Default Clear Color**. This can be set to any color you want.



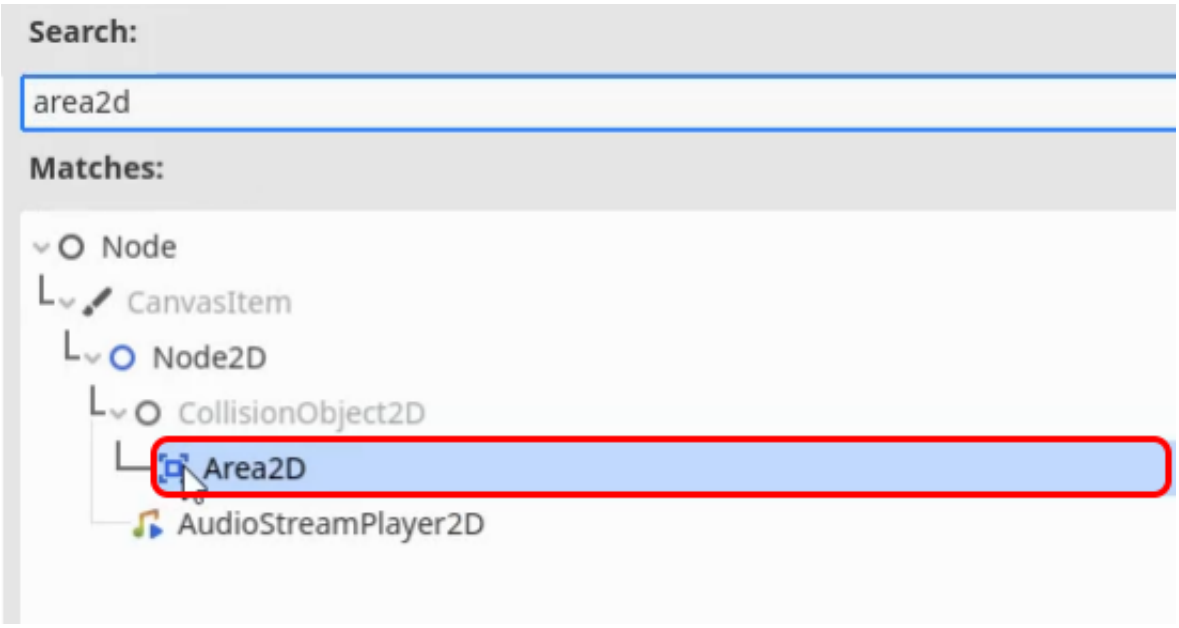
You will then see the background color in both your scene view and the game if you press *Play*.

In the next lesson, we will be making enemies that if the player hits, will produce a game_over state.

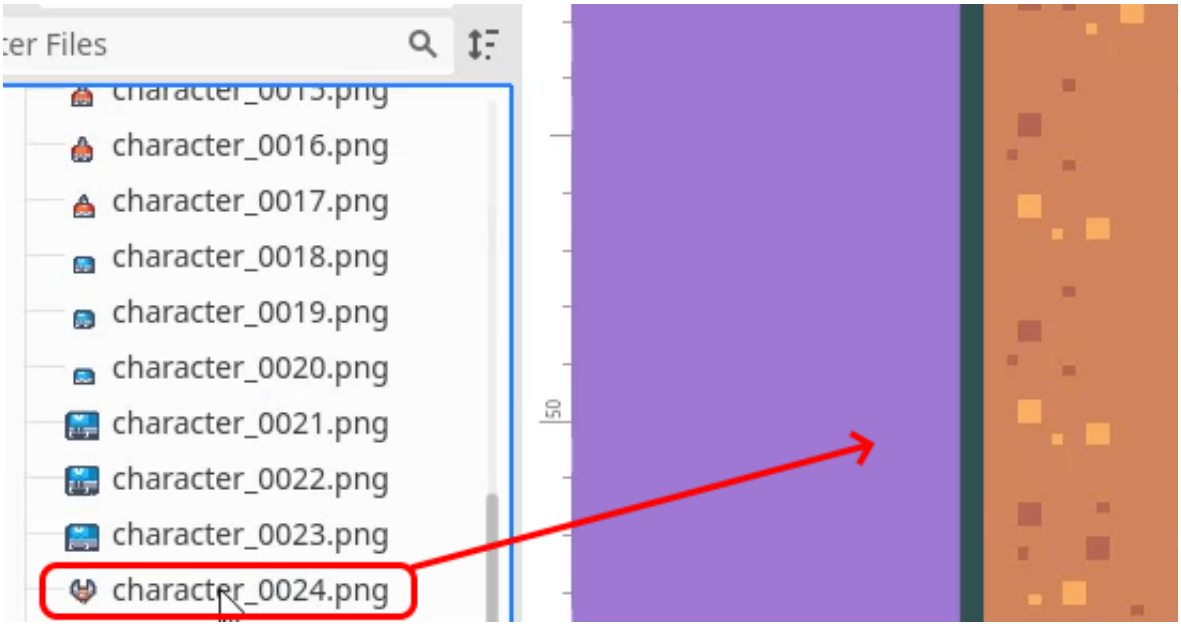
Creating an enemy scene in a game is an important part of the development process. In this lesson, we will be setting up an enemy scene in a 2D game.

Creating the Enemy Scene

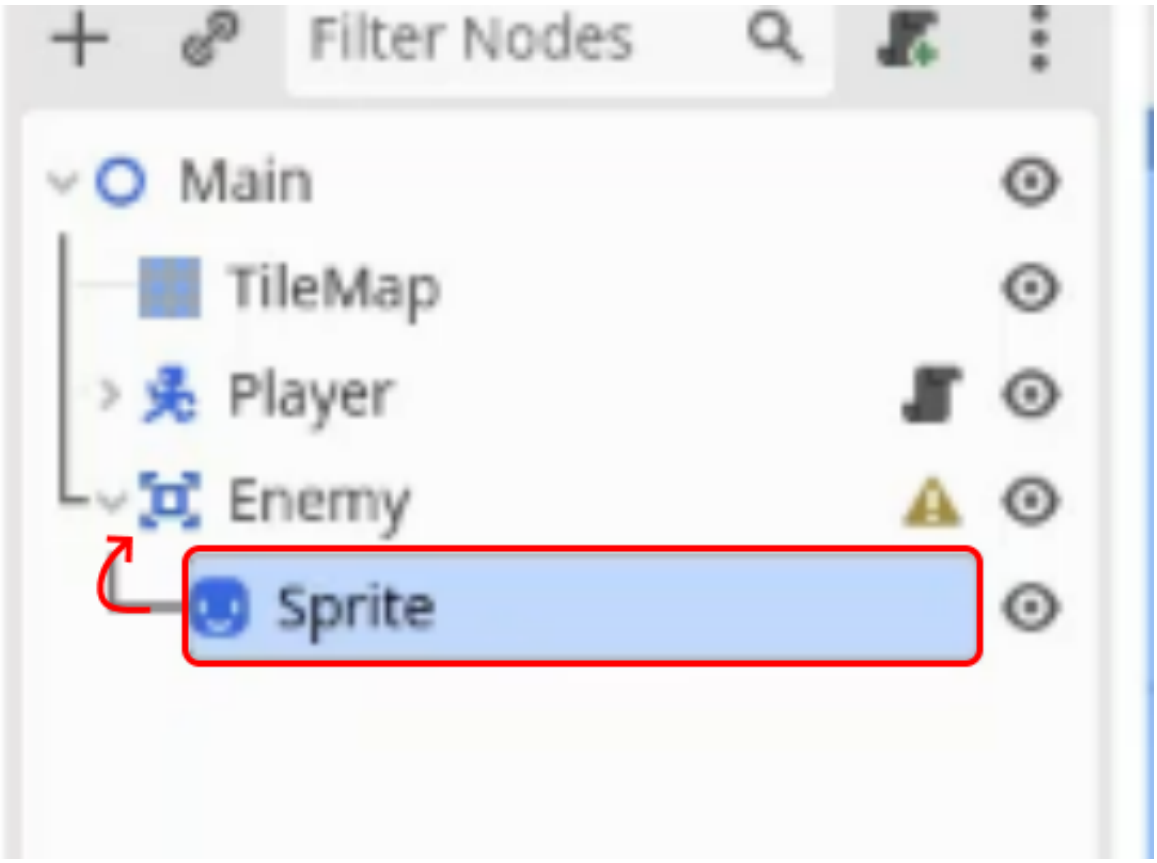
The first step is to create a new node of type **Area2D**. This node will detect collisions and run an event in the code when another collider passes through it. You can rename this to *Enemy*.



We will then add a **Sprite** node as a child of the Area2D node. We will be using the *character_0024.png* sprite from our pack.

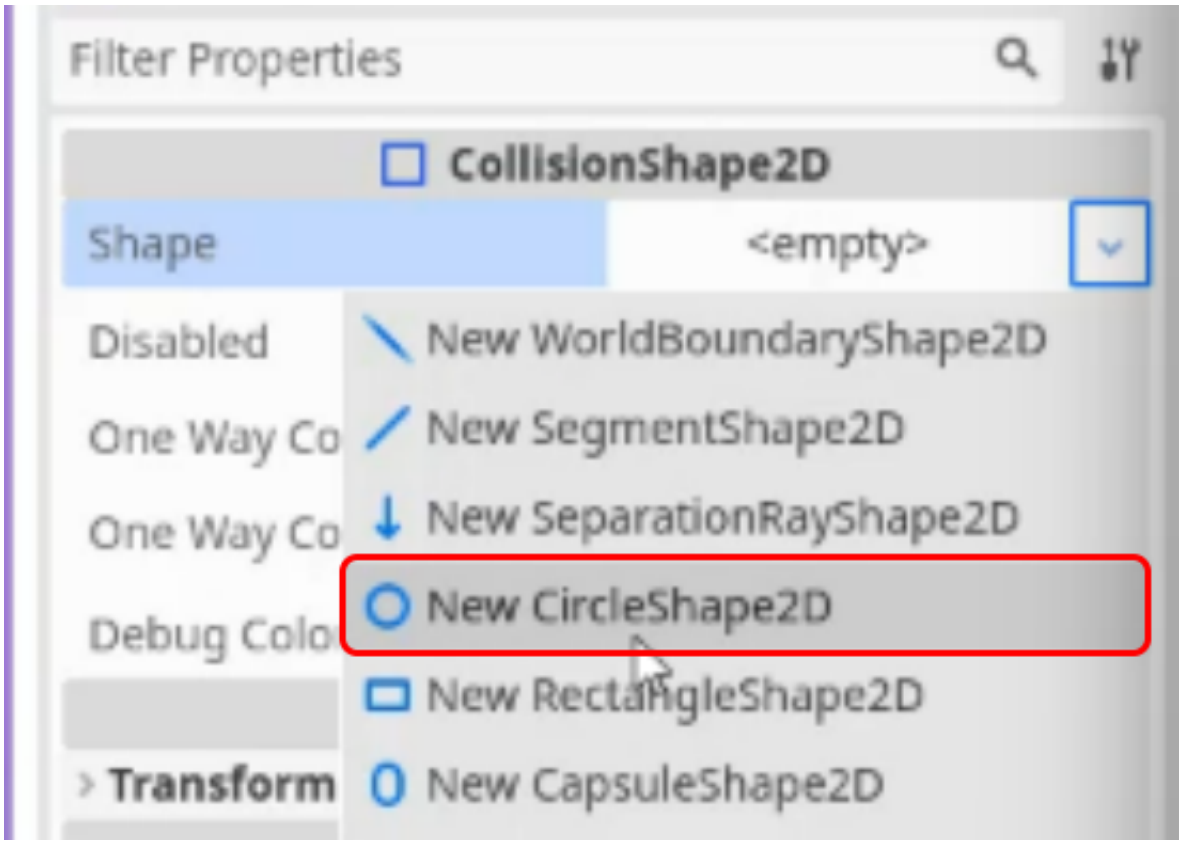


This will give the enemy a visual representation in the game. We can then rename the node to *Sprite.*



Next, we need to add a **CollisionShape2D** node to the enemy to give our *Area2D* a shape to check.

Then set the **Shape** property to a new **CircleShape2D**.

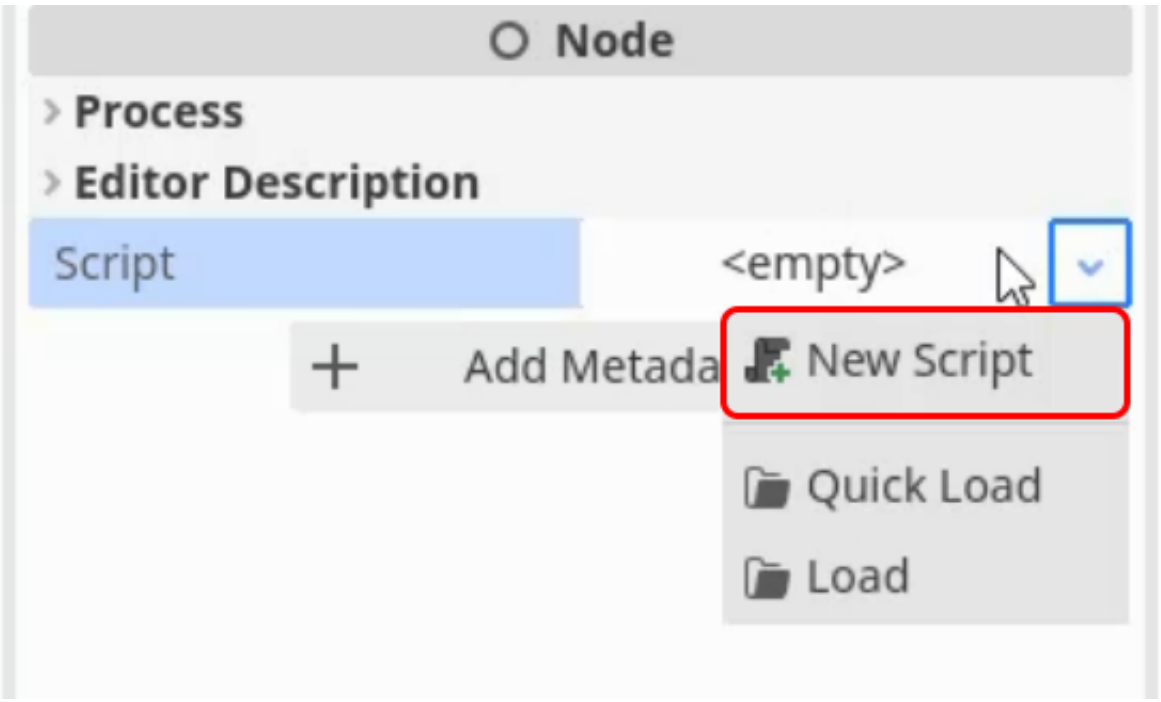


Next, adjust its size to be slightly smaller than the Sprite.

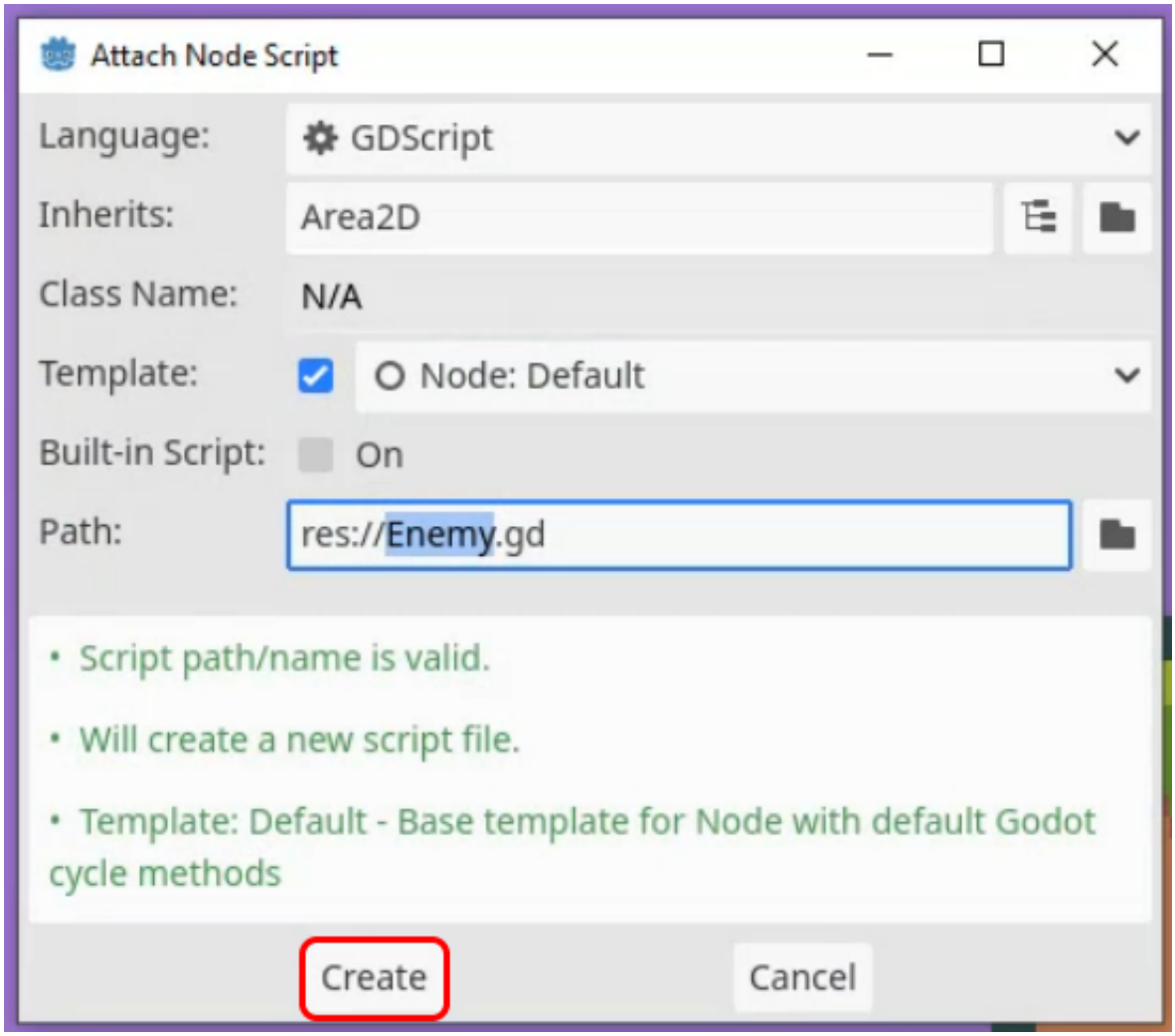
This will allow the player to pass by the enemy without colliding with it but still set off a collision event.

Creating the Enemy Script

With the *Enemy* node selected, create a **new script.**



This script will be named *Enemy.gd* and inherit from *Area2D*, as this is the *Enemy* node's type.

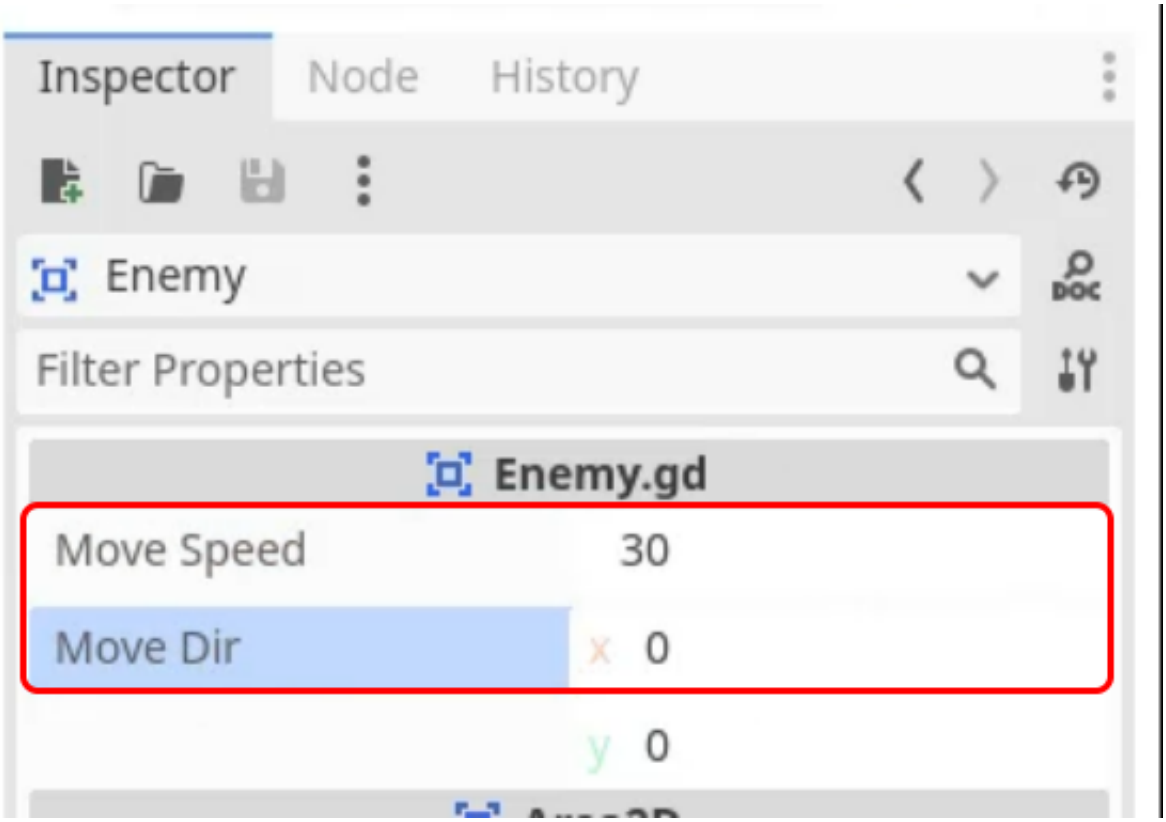


This time we will keep the default functions, but we will add variables for the enemy's move_speed, move_direction, start_position, and target_position.

```
extends Area2D

@export var move_speed : float = 30.0
@export var move_dir : Vector2

var start_pos : Vector2
var target_pos : Vector2
```

- *move_speed* – The speed at which our enemy will move.
- *move_direction* – The maximum point our enemy will move to before moving back.
- *start_pos* – The position the enemy starts in and will move back to.
- *target_pos* – The position the enemy will move to and away from.

We have used the **@export** tag to make our variables visible to edit in the inspector.



With these variables, we can then set the *start_pos* and the *target_pos* in the *_ready* function.

```
func _ready():
    start_pos = global_position
    target_pos = start_pos + move_dir
```

We can now create the logic to move toward our target position. This will be done in the *_process* function, as it runs once every frame.

```
func _process(delta):
```



```
    global_position = global_position.move_toward(target_pos, move_speed * delta)
```

We will also need to check to see if we have reached our target position, and if we have, change the target position to the other way.

```
func _process(delta):
    ...

    if global_position == target_pos:
        if global_position == start_pos:
            target_pos = start_pos + move_dir
        else:
            target_pos = start_pos
```

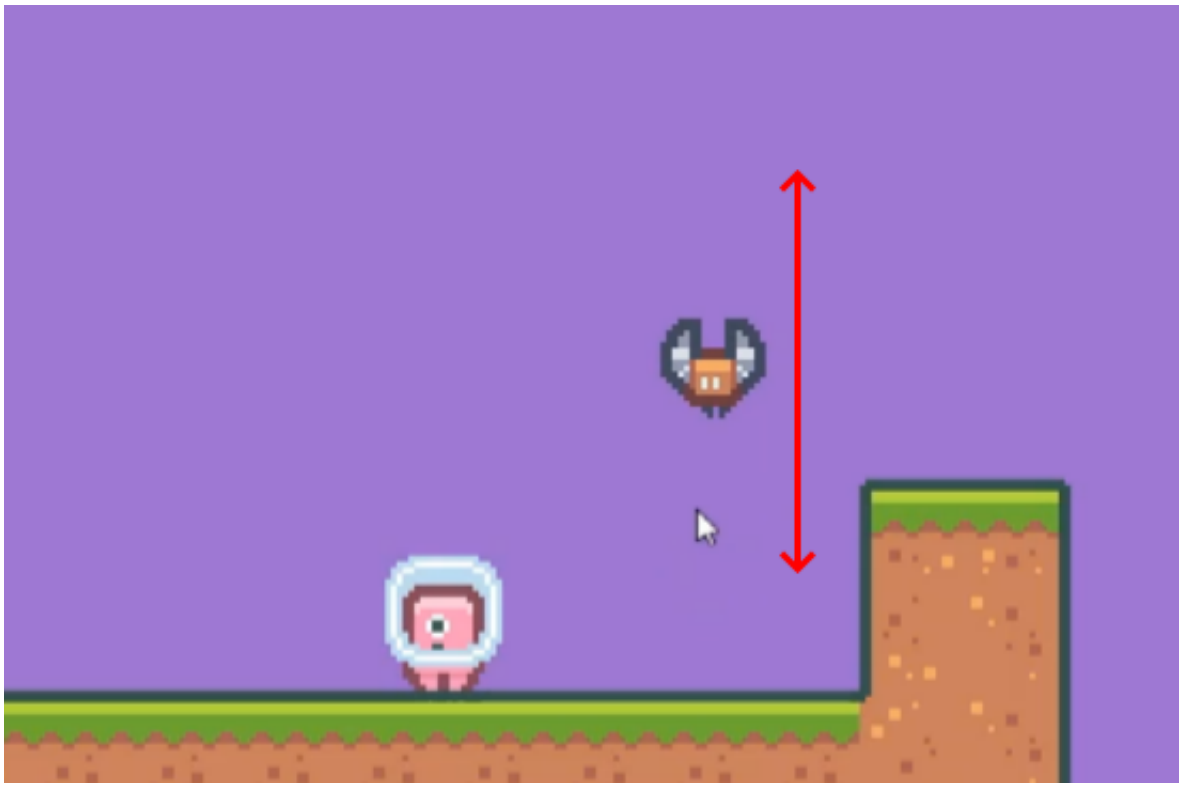
This code will check to see if we have reached our *target_pos*, and then proceed to check if we are at the *start_pos* or not. If we are, we will change the *target_pos* to our *move_dir* value, if not, we want to move back to the *start_pos*.

We can now save this script (**CTRL+S**) and return to the 2D scene view.

Setting up our Enemy

We now need to set the **Move Dir** value we exported earlier, we will use the value **(0, -50)** to state move 50 pixels upward, you can of course change this value however you want.

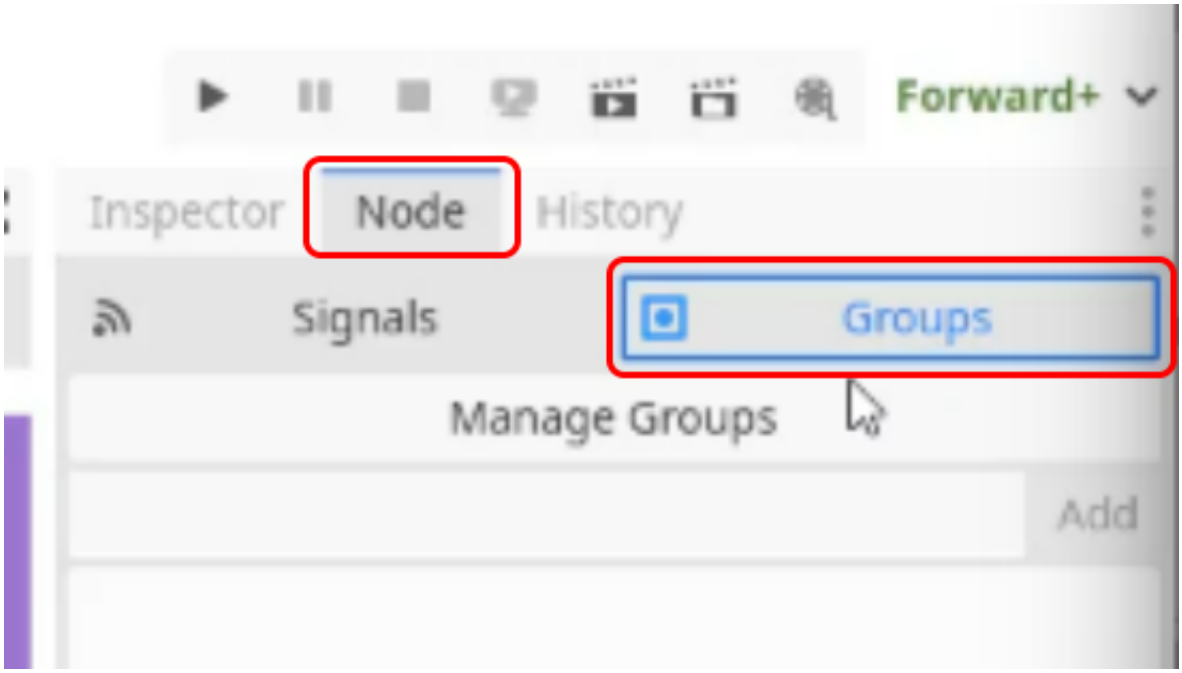
If you now press **Play**, you will see the *Enemy* we created go up and down between its start position and target position.



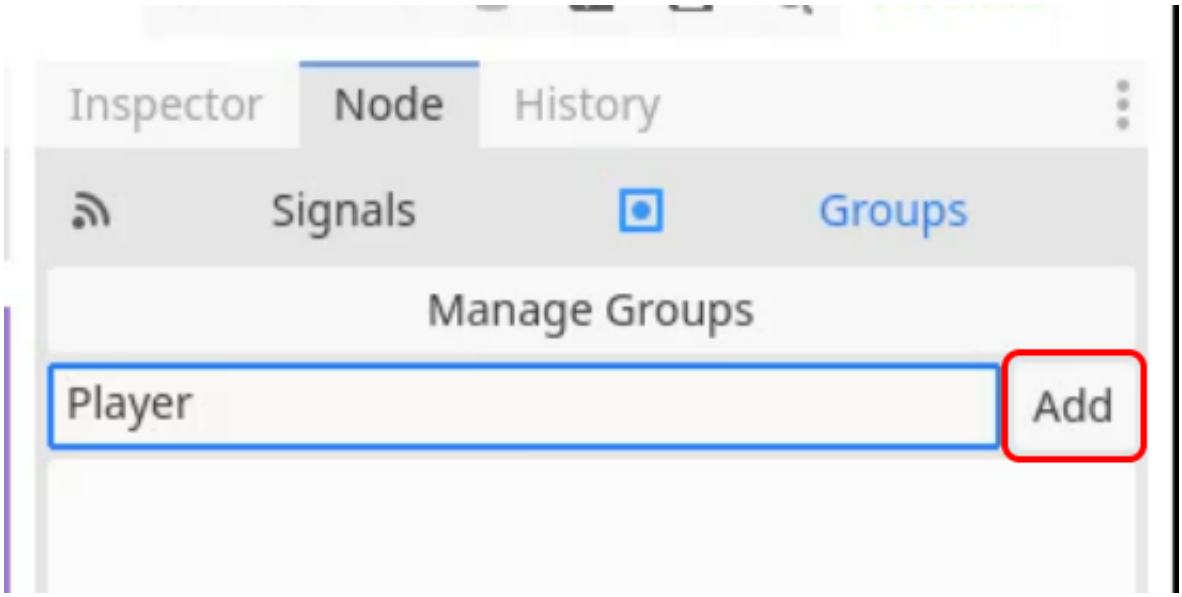
You will notice, however, if you touch the enemy, it doesn't yet call our game over function.

Detecting Collisions

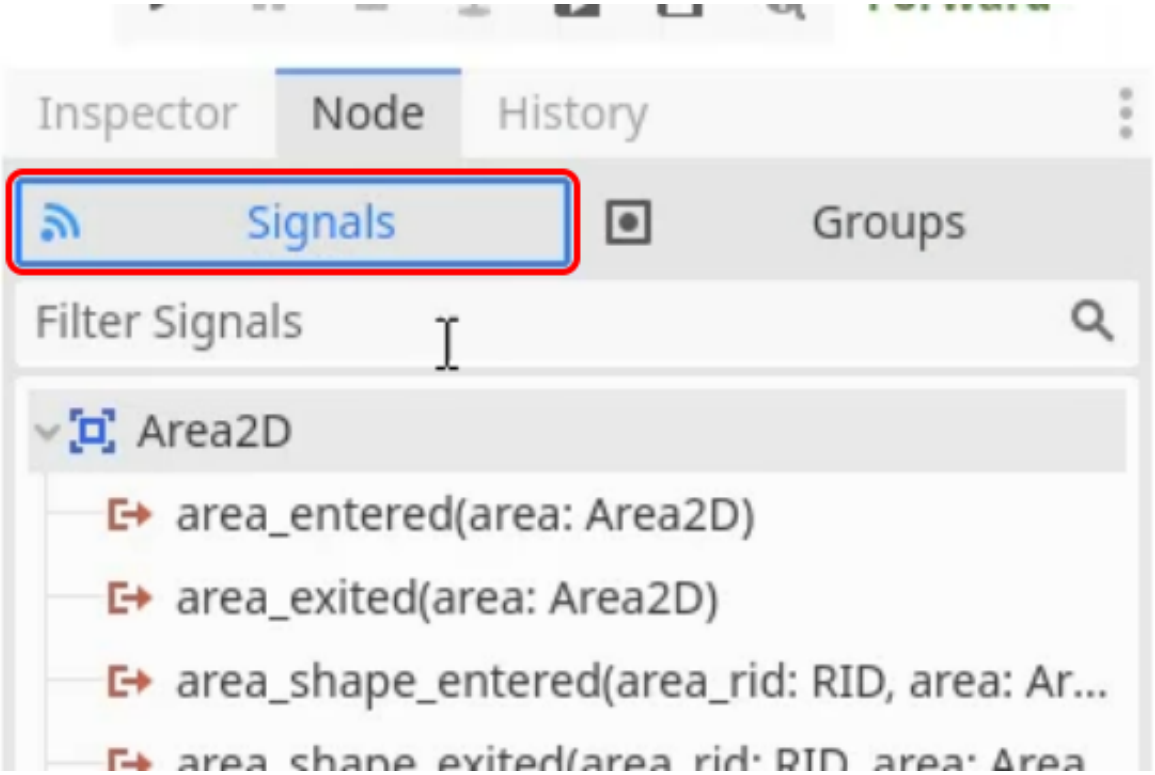
To do this, we will select the **Player** node and open the *Node* tab and select the *Groups* section.



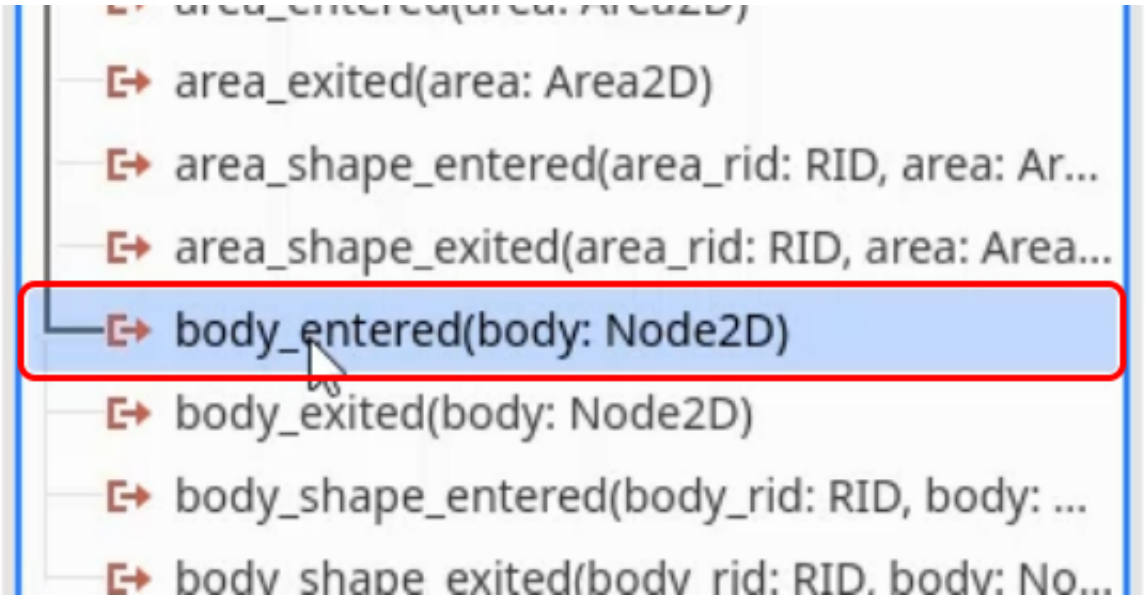
We want to add our *Player* node to a group, this is like tagging our node and will let us know what type of node a collider is when one is detected by our *Enemy*. We will add a group to our *Player* named **Player**.



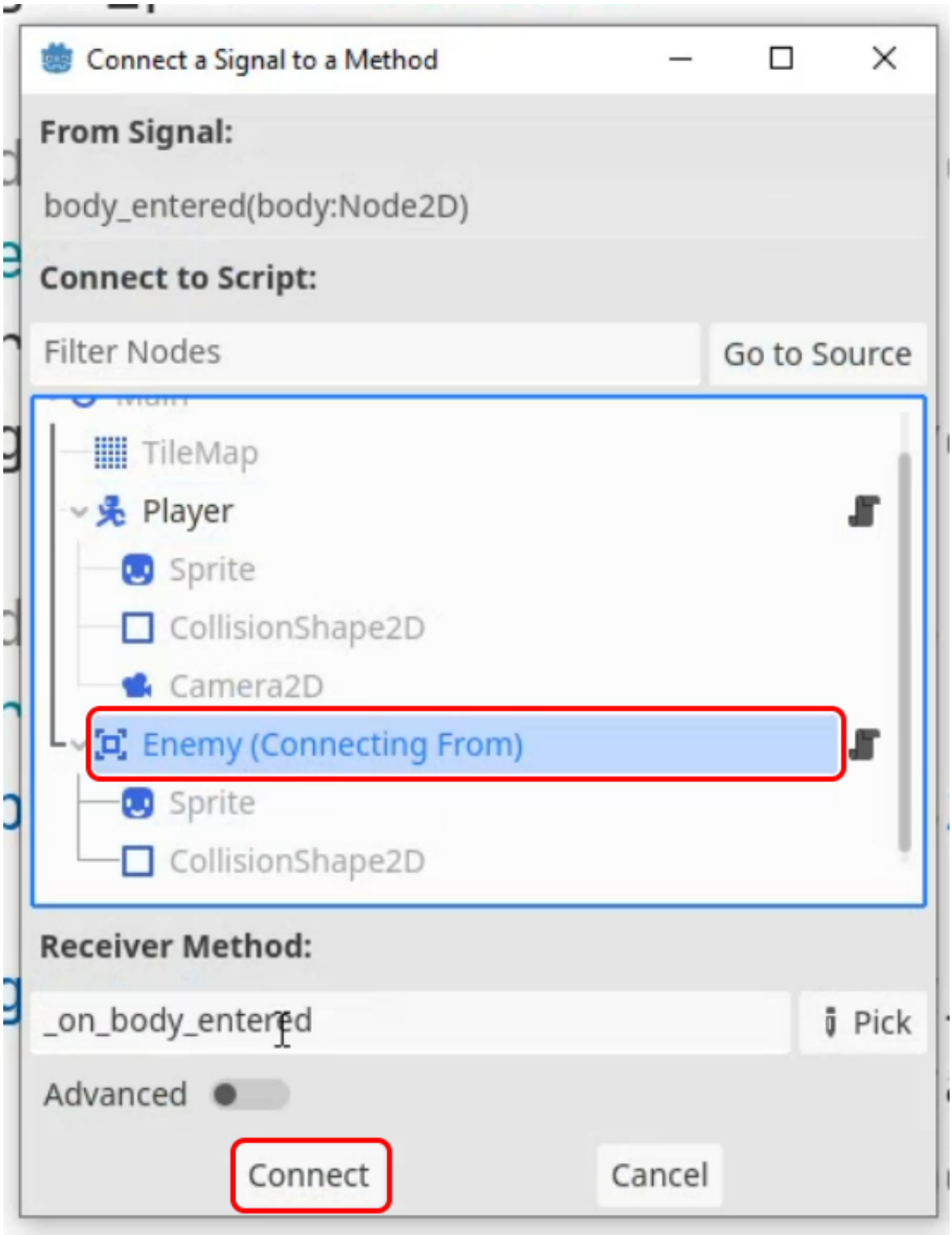
Next, with the **Enemy** node selected, choose the *Signals* section this time.



To add a signal to our *Enemy,* choose the **body_entered** signal and **double-click** it.



Then select our **Enemy** node in the list that appears and press **Connect**.



This will create an *_on_body_entered* function that will call when the signal is set off. The *body* parameter passed through the function is the object we have interacted with, so we want to check if this is the *Player* and call the *game_over* function if it is. We will do this with the following code at the bottom of the *Enemy.gd* script:

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        body.game_over()
```


Now if you save the script, and press **Play** you will see the player is reset when they touch the enemy, as intended.

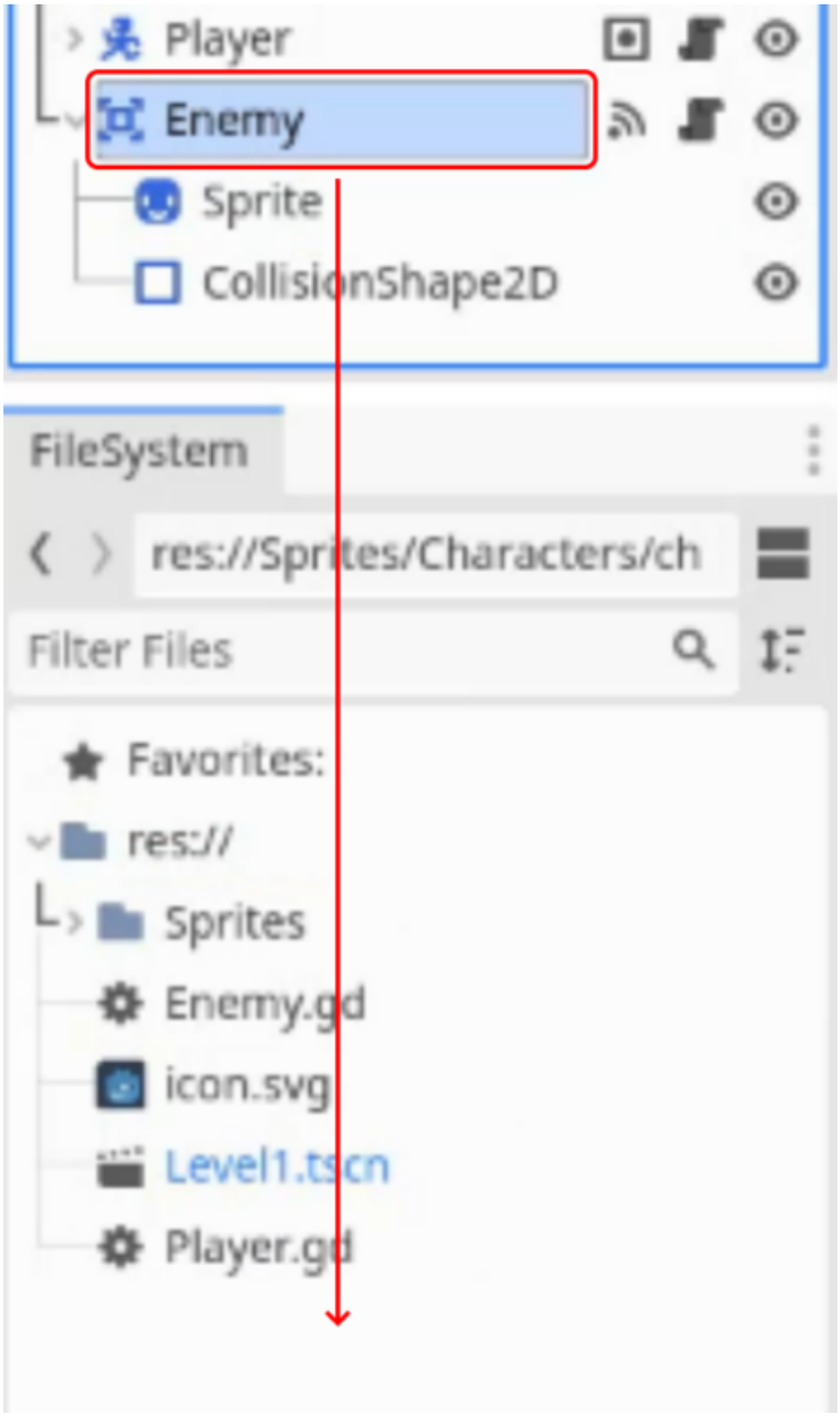
Are you getting an error in the console even though the code is running? This may be due to how we are calling the game over function. To fix this, we can change the function as so:

```
func _on_body_entered(body: Node2D):
    if body.is_in_group("Player"):
        body.call_deferred("game_over")
```

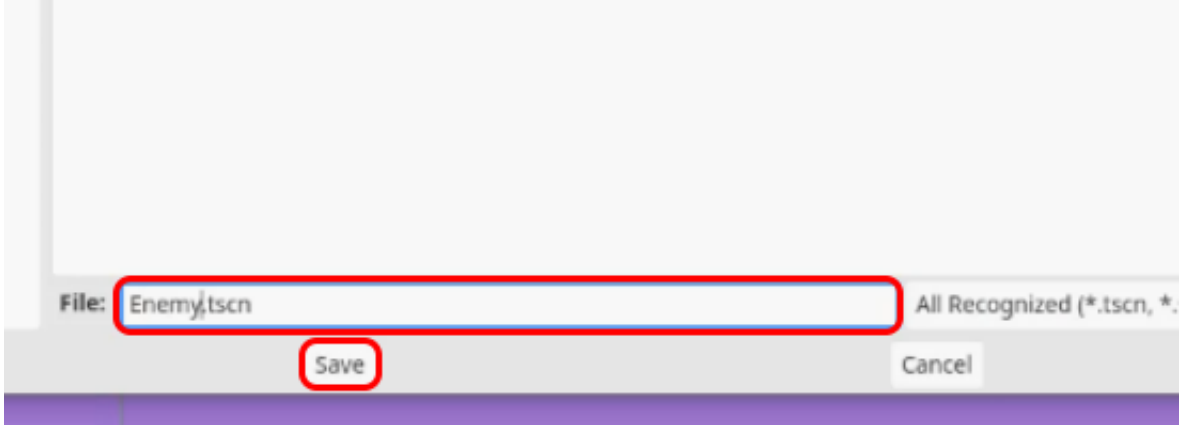
Using *call_deferred* waits until there is time in the main loop before executing the function call.

Turning the Enemy into a Scene

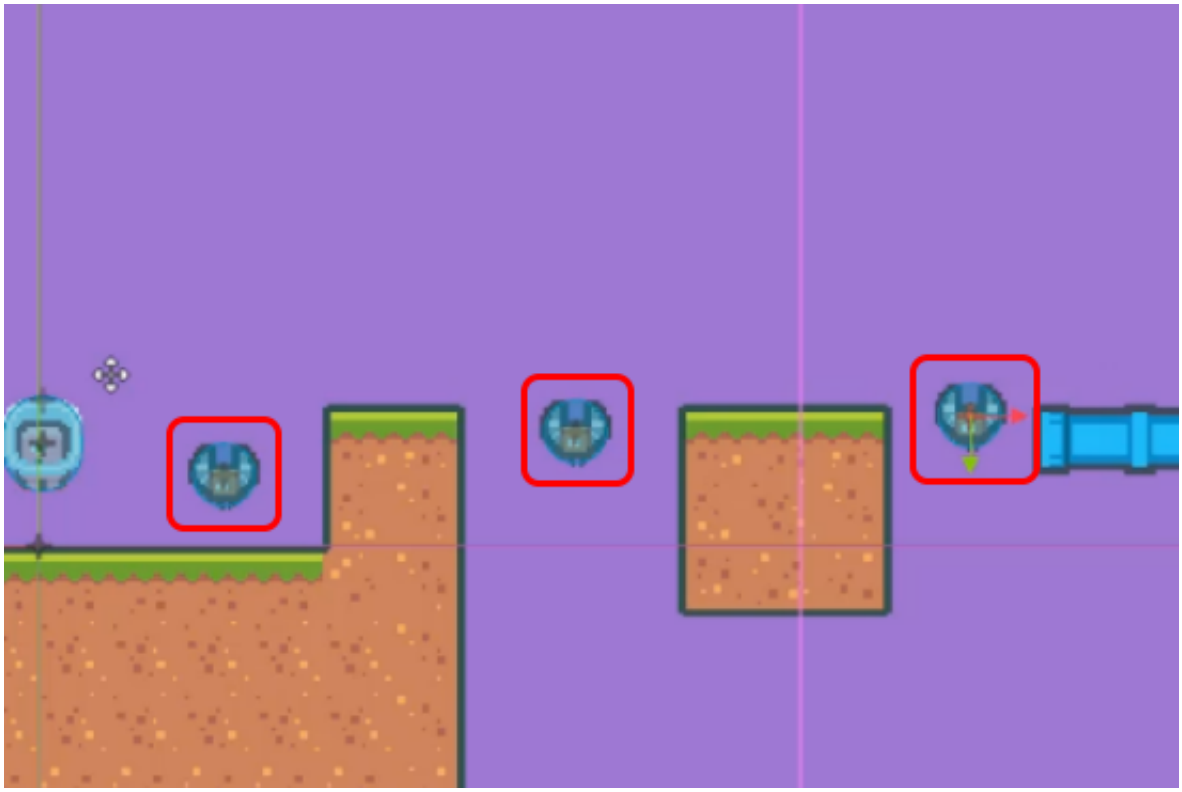
We finally need to save the *Enemy* node tree as a scene. This will allow us to drag in multiple instances of it into the game, and still be able to edit it globally. Do this by dragging the top *Enemy* node from the *Scene* tab to the *FileSystem* tab.



Then, save the scene as *Enemy.tscn*



You can now copy and paste the *Enemy* scene around the level and change the *Move Dir* and *Move Speed* to make differing enemies.



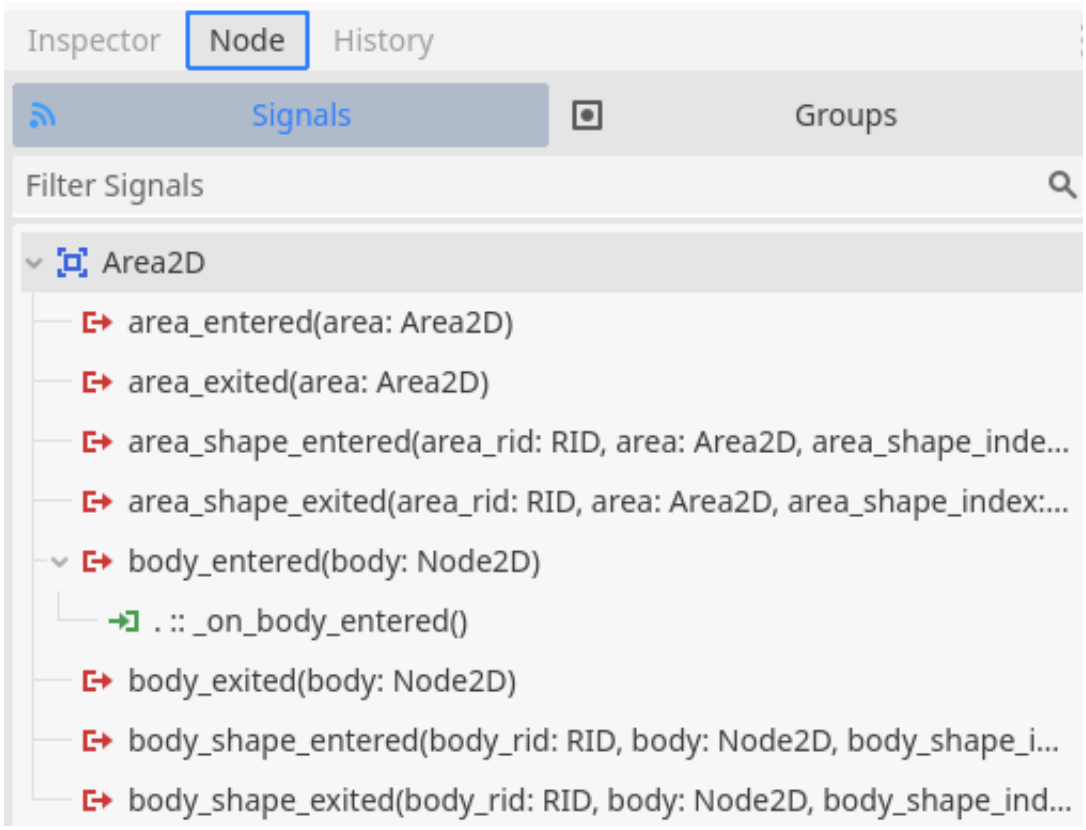
In the next lesson, we will be creating spikes, which will be another trap that the player must avoid.

The the previous lesson, we looked at connecting a **signal** to our enemy. Let's now have a deeper look at what signals are and how they work in Godot.

What is a Signal?

Signals are delegation mechanisms which allow one object to react to a change in another object without directly referencing one another. If you have experience in other game engines, think of signals as events.

For our enemy node, the **body_entered** signal is being emitted when another physics body has entered its collider. This sends over data, such as who this body is. Now on its own, this body_entered signal receives the call, but by default, does nothing with it. What we can then do is connect a function to this signal. Once connected, that function will call when the signal has been emitted.



Think of it as a radio tower, emitting radio signals for people to hear music in the area. The tower does not know where or how many radios want to listen to the signal. The individual radios must tune into the correct frequency (connect to the signal). Then, when the tower emits, those radios will be able to play the audio.

Connecting to a Signal

There are two way to connect to a signal in Godot. The first, is how we have done it so far in this course: navigating to the *Node* panel and connecting the desired signal to a script. This will create a function and notify it as a signal connection via the green icon to the left of the name. This can be a fast and easy way of doing this, but what if we want to connect to a signal in the middle of the game? Or disconnect from a signal? Or start the game not connected, but then after a certain thing happens, connect?

The second way of connecting to a signal, is doing so via scripting.

Inside of our **Enemy** script, we can first create the function that the **body_entered** signal will connect to. The function can be called whatever you want, but make sure it contains the correct parameters. Body_entered requires 1 parameter of type Node2D.

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        body.game_over()
```

Then, up in the **_ready** function, we want to connect that function we just created to the signal. Since the signal is of the Area2D node which our script is attached to, we can simply write this:

```
body_entered.connect(_on_body_entered)
```

It follows this format: [signal].connect([function name])

Now when we press play, the function should call when the body_entered signal has been emitted!

Additional Resources

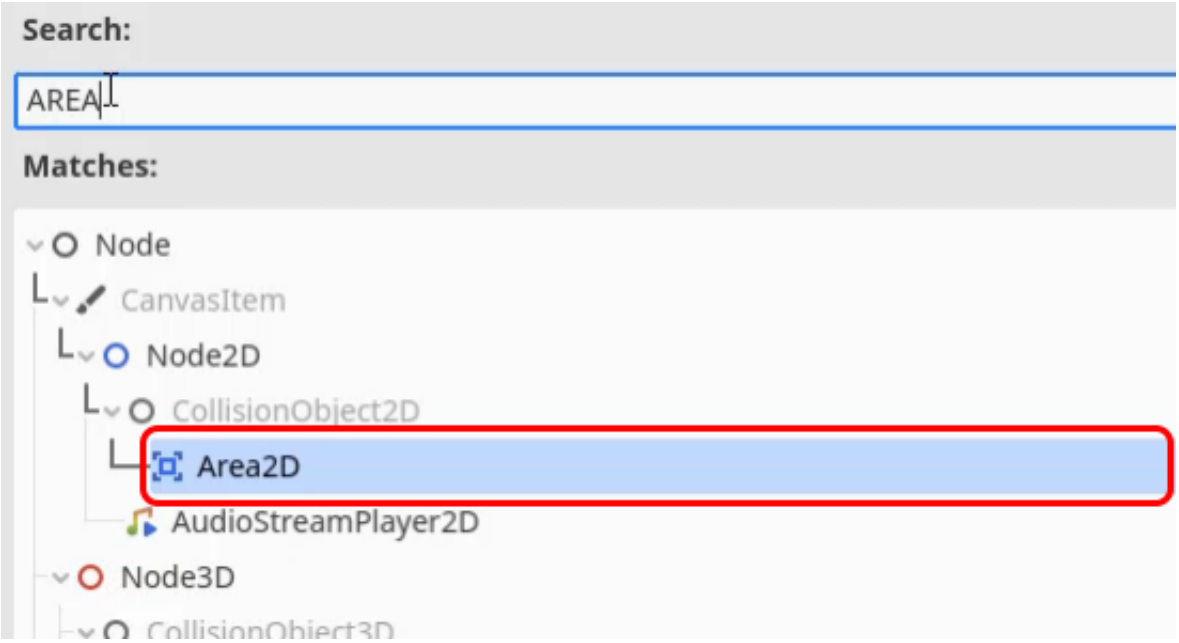
If you wish to explore this topic further, the Godot documentation will certainly help you out.

- Signals
- Connecting a Signal via Code

In this lesson, we are going to be setting up our spikes. Our spikes are nodes that, when the player collides with them, will call the player's game over function.

Creating the Spike Node

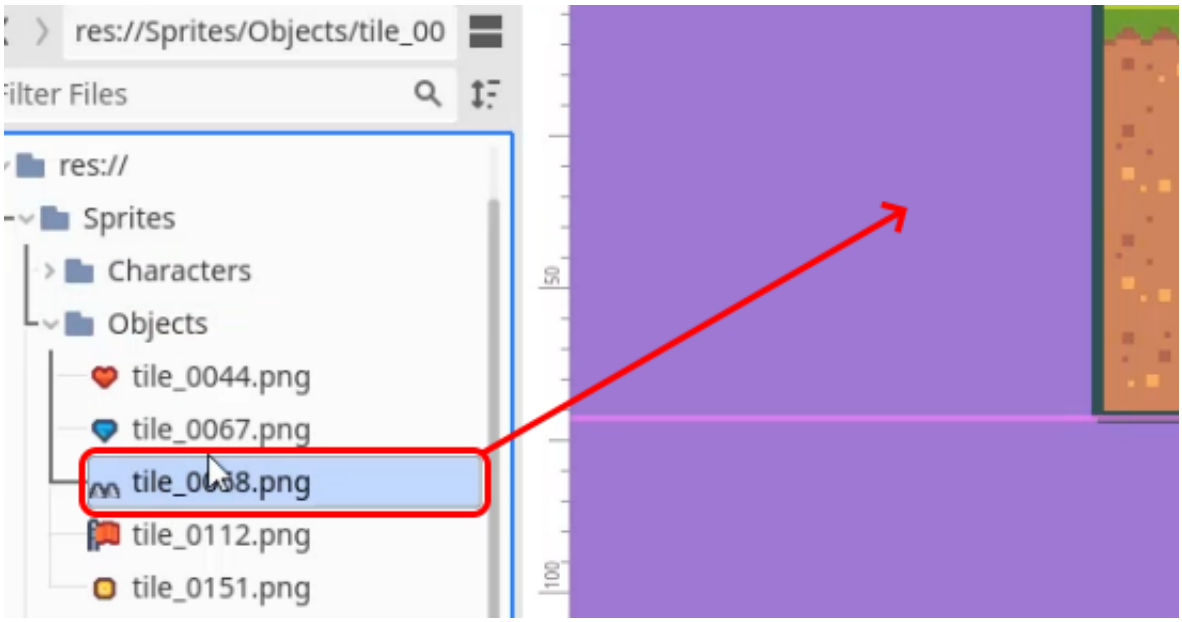
To begin, we need to create an **Area2D** node that will act as the base for our spike.



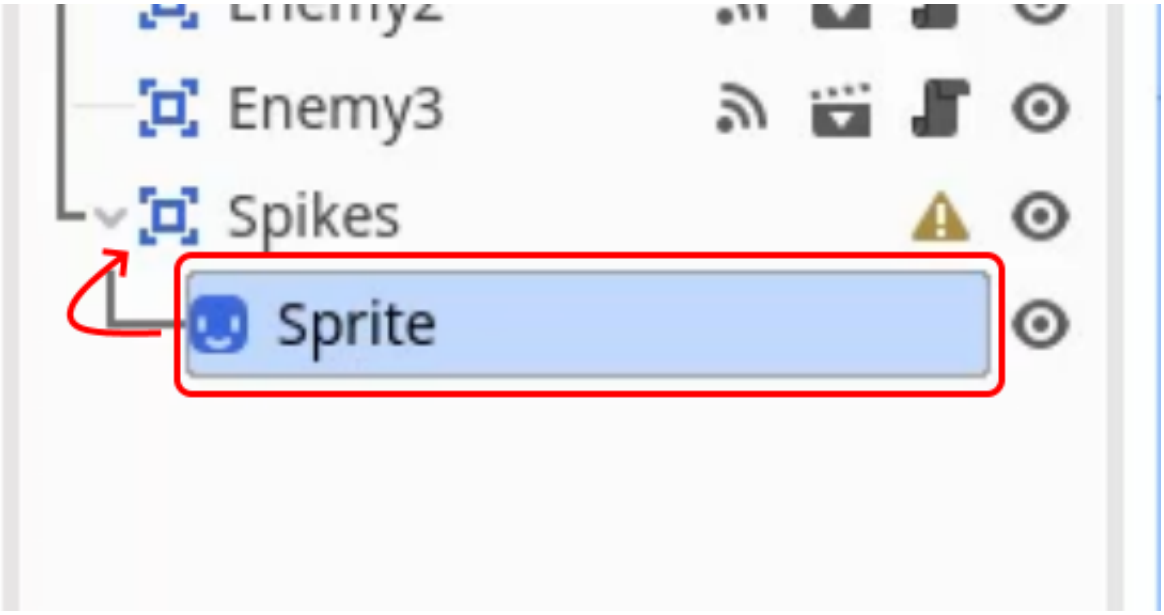
We then double-click the new node to rename it *Spikes*.

Adding a Sprite

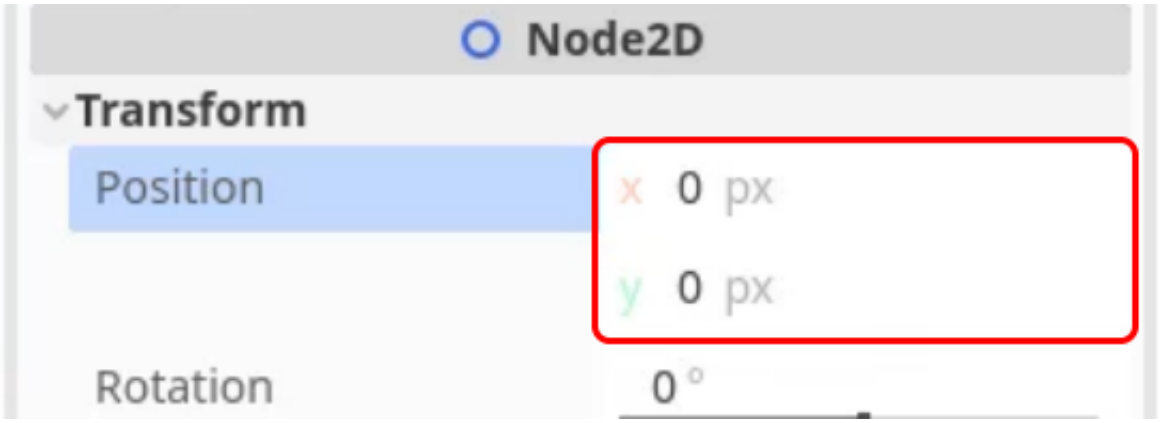
Next, we need to add a Sprite to the node. As with the other sprites, we will do this by dragging a texture into our scene. We will be using the *tile_0068.png* file for our spikes.



Then make the Sprite a child of the spikes node, and rename it to *Sprite*.

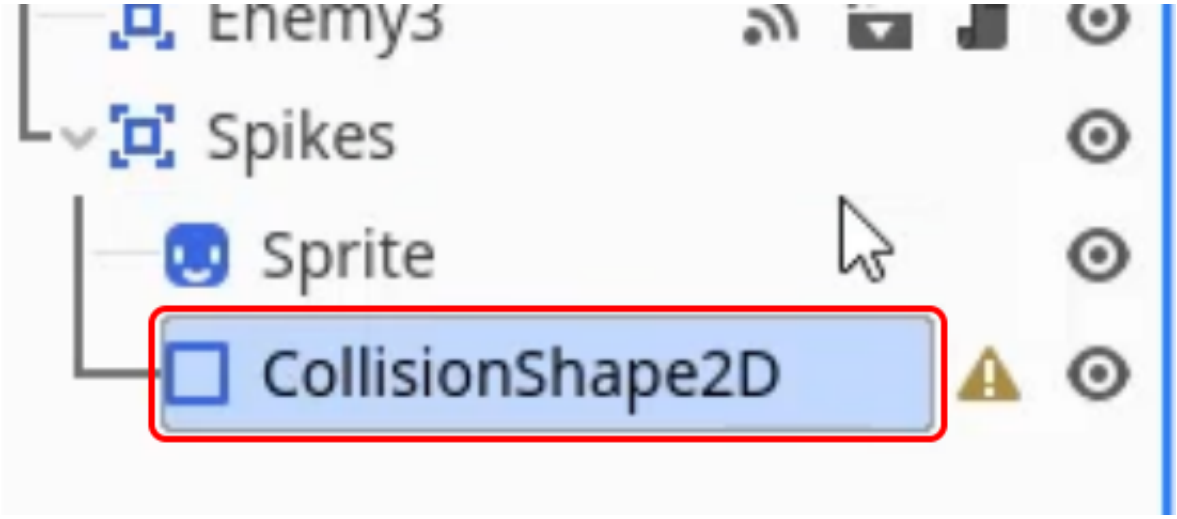


Finally, make sure to set the **position** of the *Sprite* to **(0,0)** to center it inside the *Spikes* node.

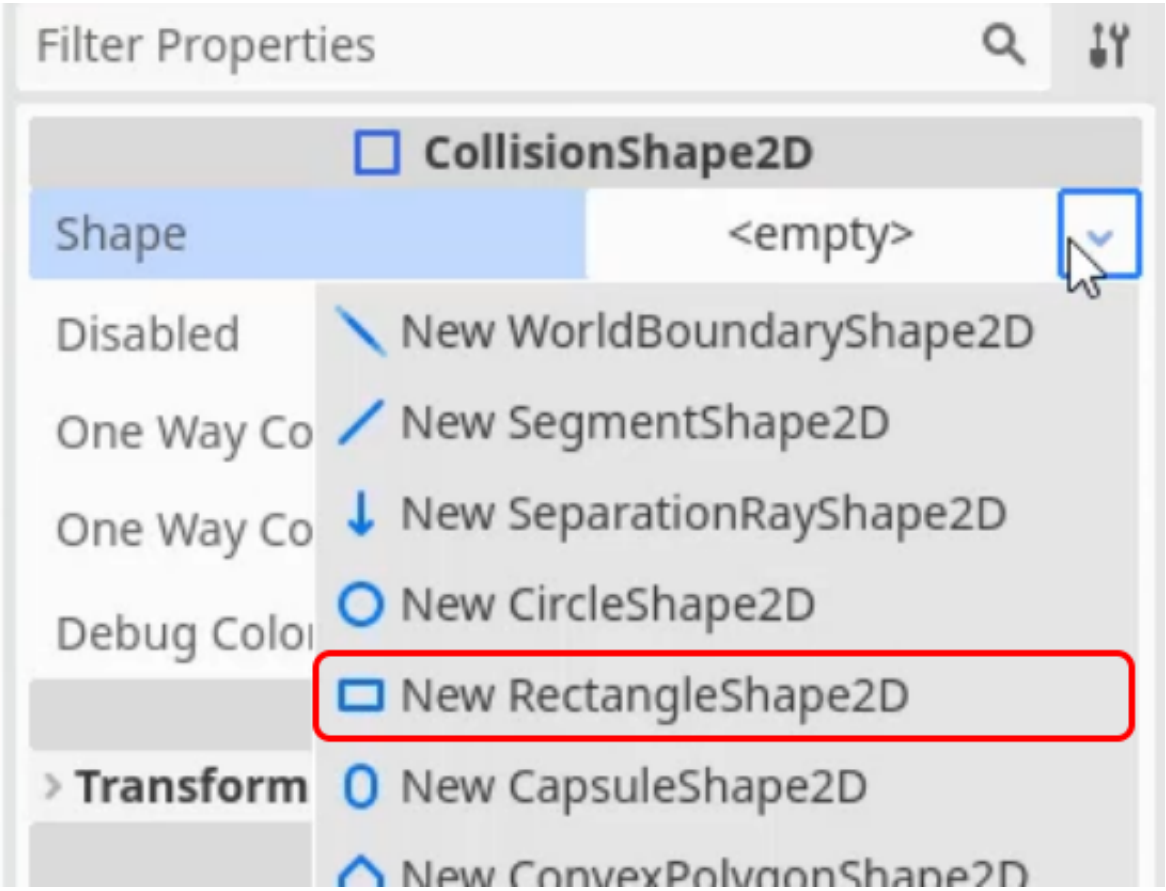


Adding a Collision Shape

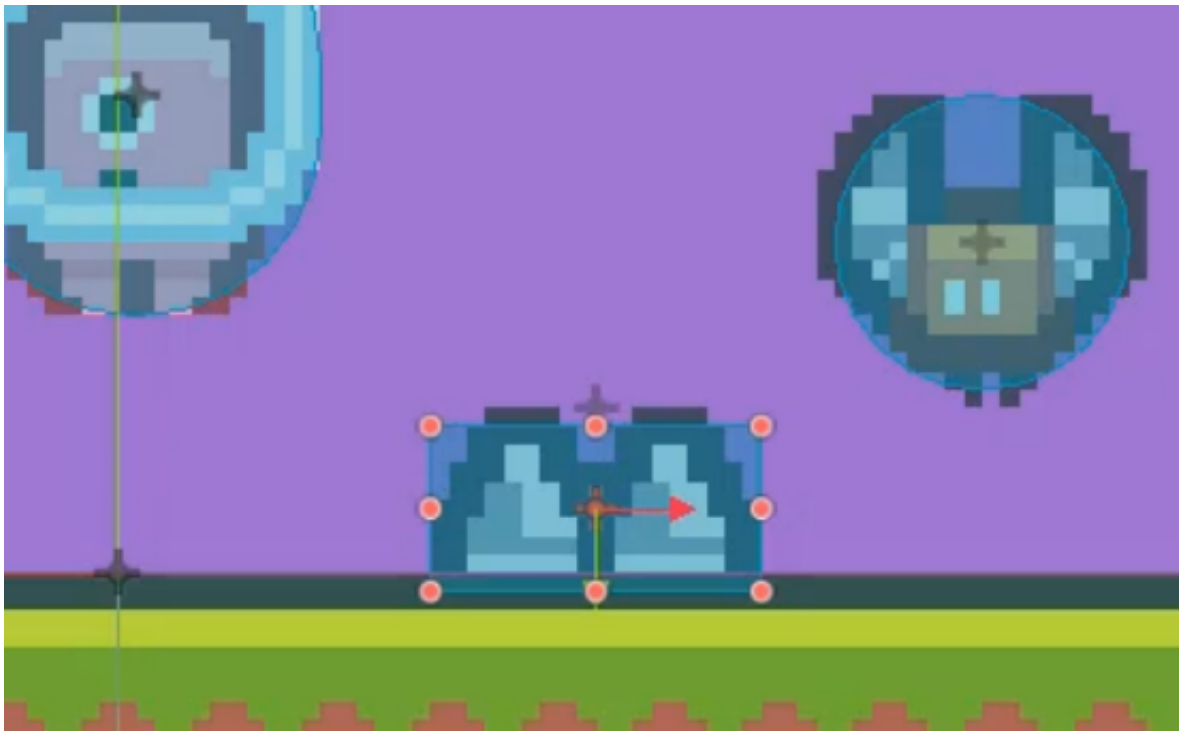
We then need to add a collision shape to the spikes node. *Right-click* on *Spikes* and choose **Add Child Node**, and create a **CollisionShape2D**.



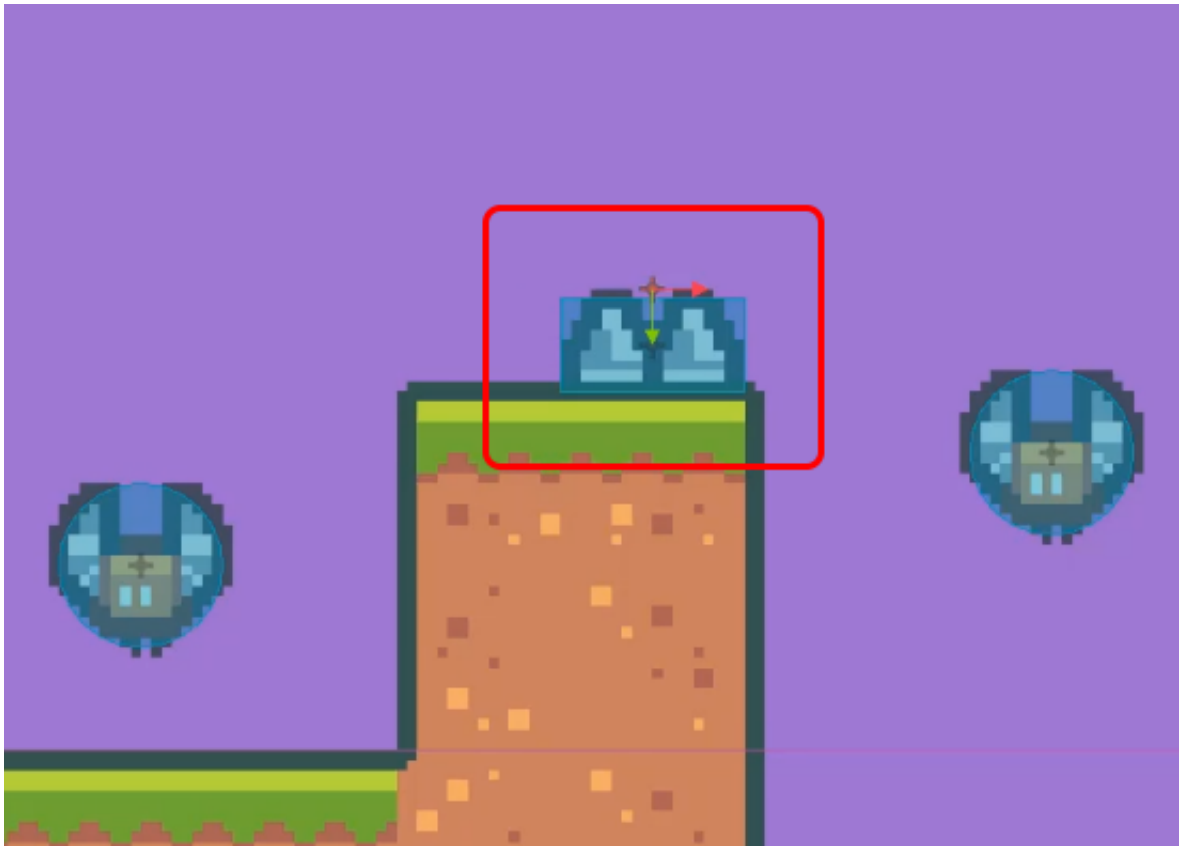
Then choose **New RectangleShape2D** for the **Shape** property.



Finally, resize it to make sure it covers the *Spikes* sprite correctly.



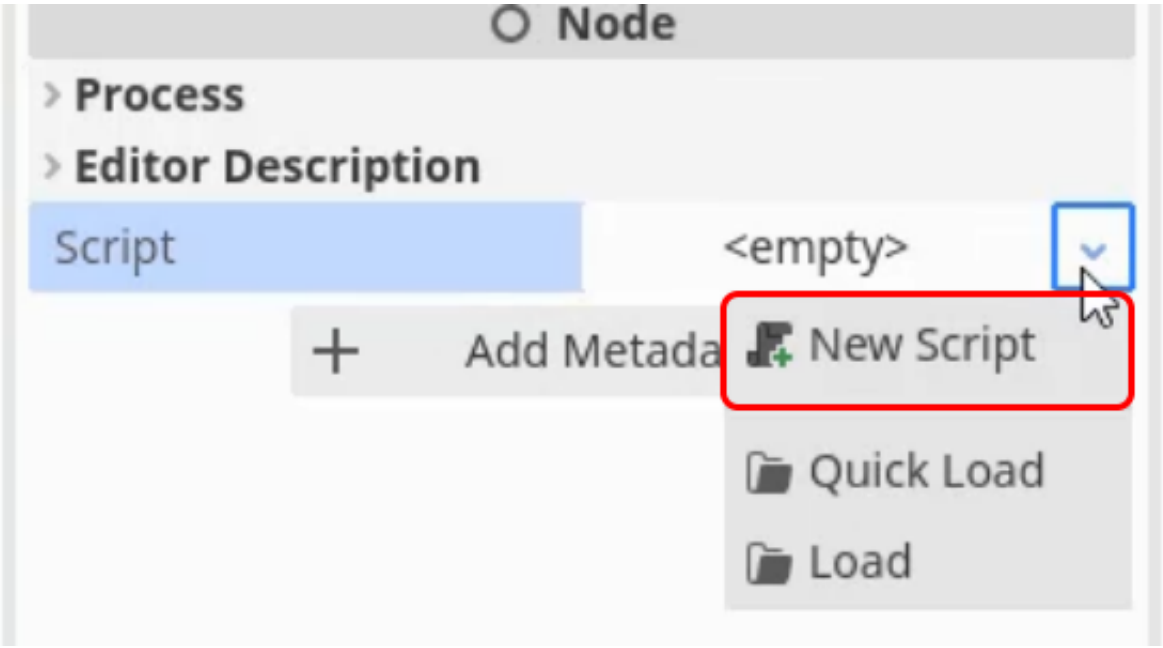
Select the *Spikes* parent node and move the object to somewhere you want in your level. For us, we will place ours on top of a hill section.



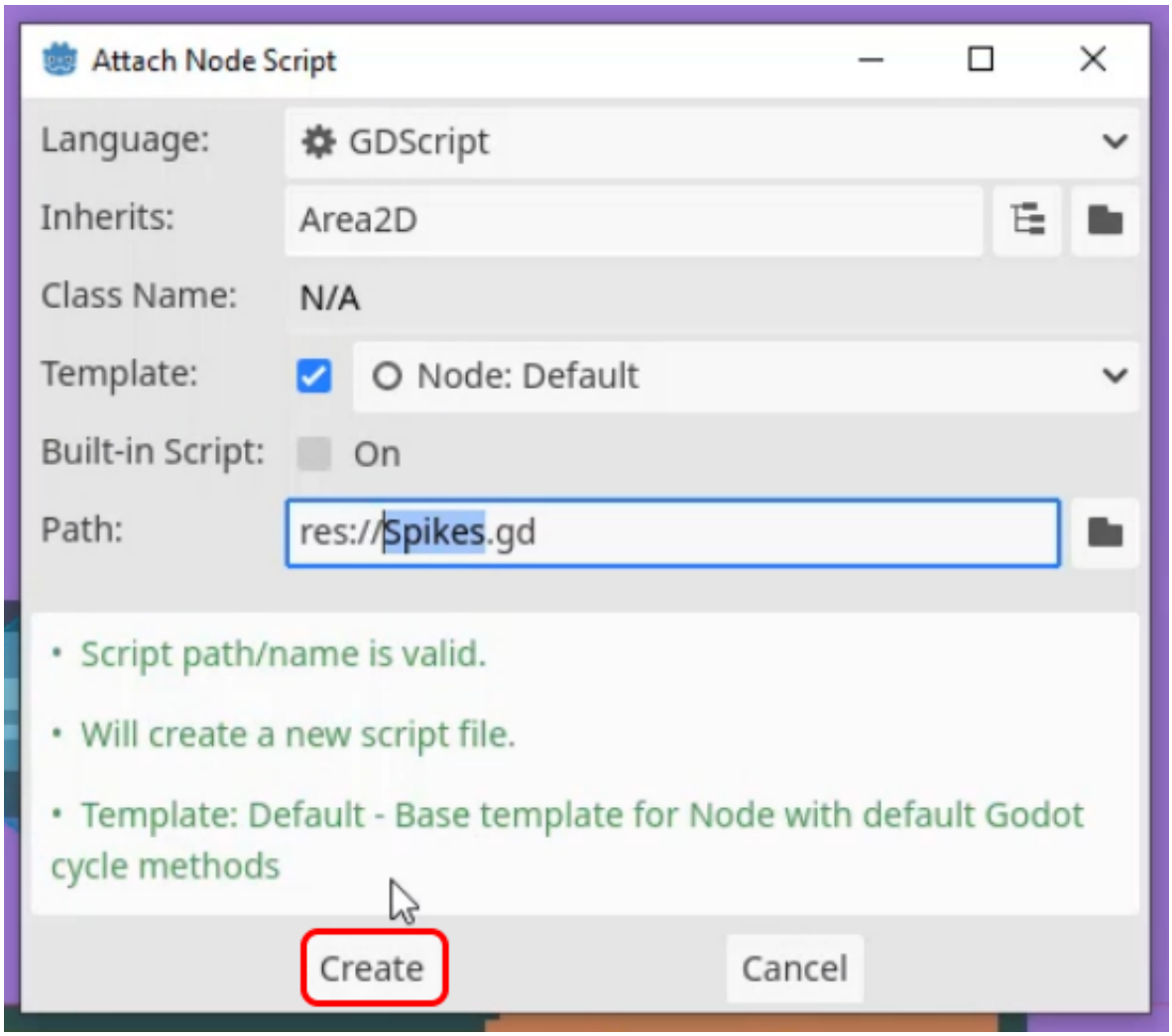
Adding a Script

Finally, we need to attach a script to the spikes node. With the *Spikes* node selected, choose **New**

Script in the **Script** property, as we have done previously.



Make sure the script is named *Spikes.gd* and that it inherits from *Area2D*. Then press **Create**.



We can then remove the default functions, leaving only the Area2D inheritance call. This should leave your *Spikes.gd* script looking like the following:

```
extends Area2D
```

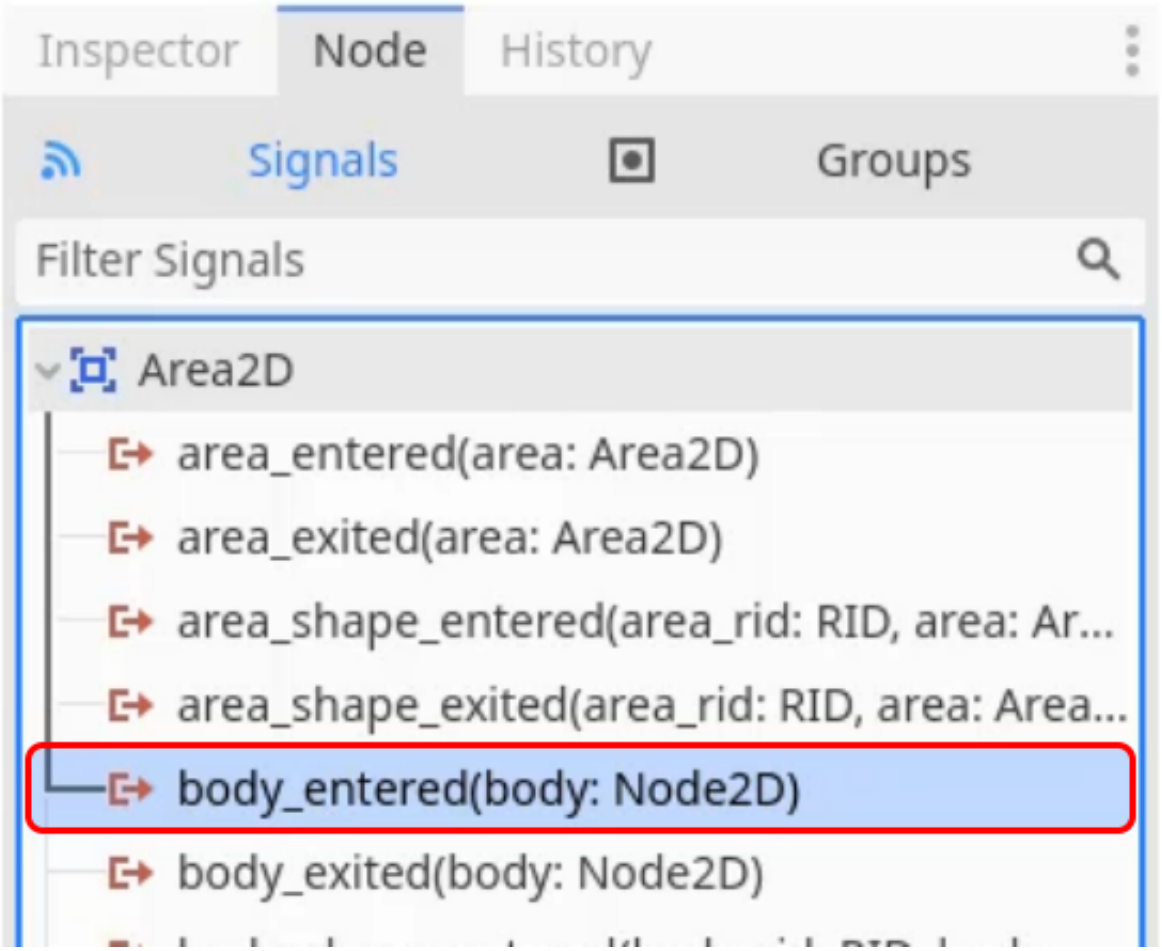
Functionality

The functionality of the script is similar to the enemy script, and we challenge you to create it on your own. In the *Enemy.gd* script we have an *_on_body_entered* function, which gets called from the signal in the node panel. When this is called, we check to see if the *body* is in the *Player* group. If so, we call the *game_over* function. We will cover the solution to this in the next lesson.

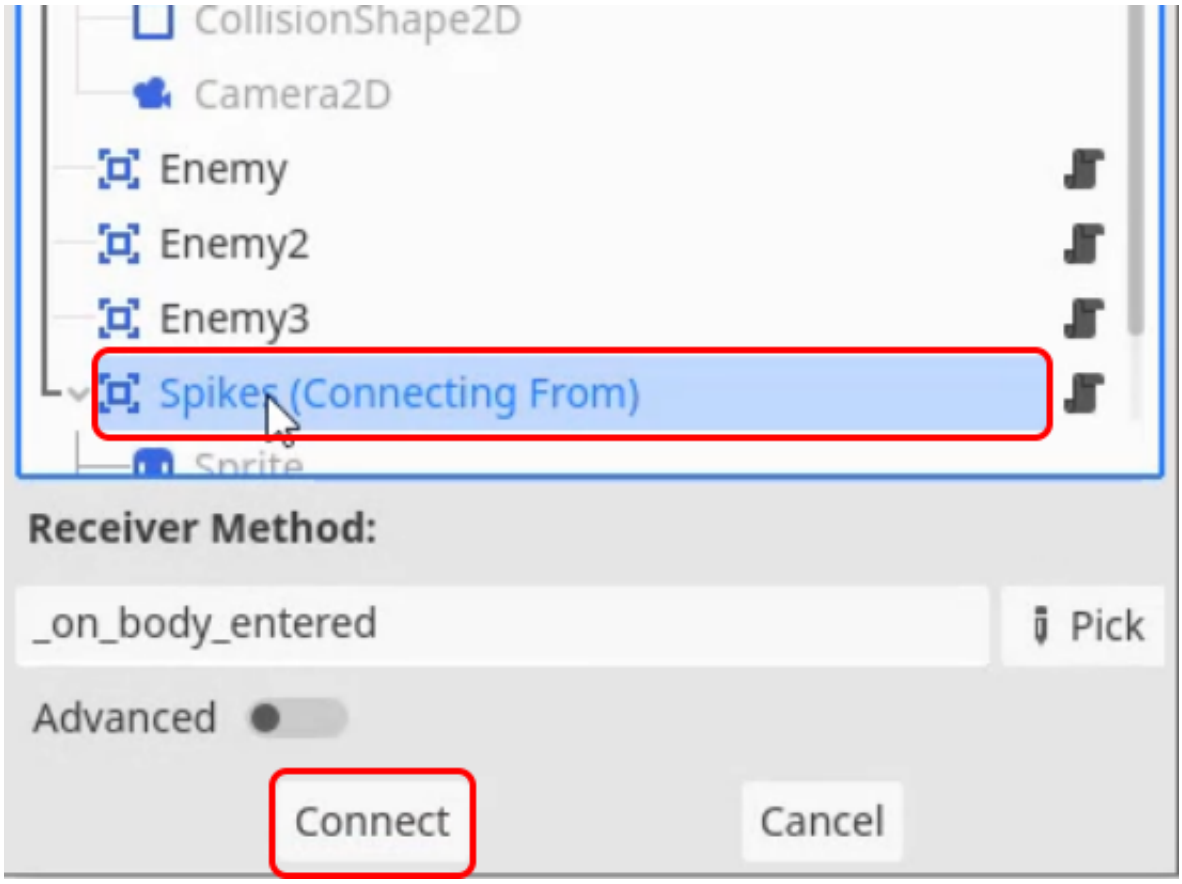
In this lesson, we will be looking at the solution to setting up our *Spikes* node.

Setting up the Spikes

With the *Spikes* node selected, in the *Node* panel, select the *Signals* section and *double-click* the **body_entered** signal.



In the next window, select the **Spikes** node and press **Connect** to create an *_on_body_entered* function in our *Spikes.gd* script.



Then in the **Script** tab make sure *Spikes.gd* is open. Then add the following code to our new *_on_body_entered* function.

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        body.game_over()
```

You can then **save** the script (**CTRL+S**) and return to the **2D** scene view.

Testing the Spikes

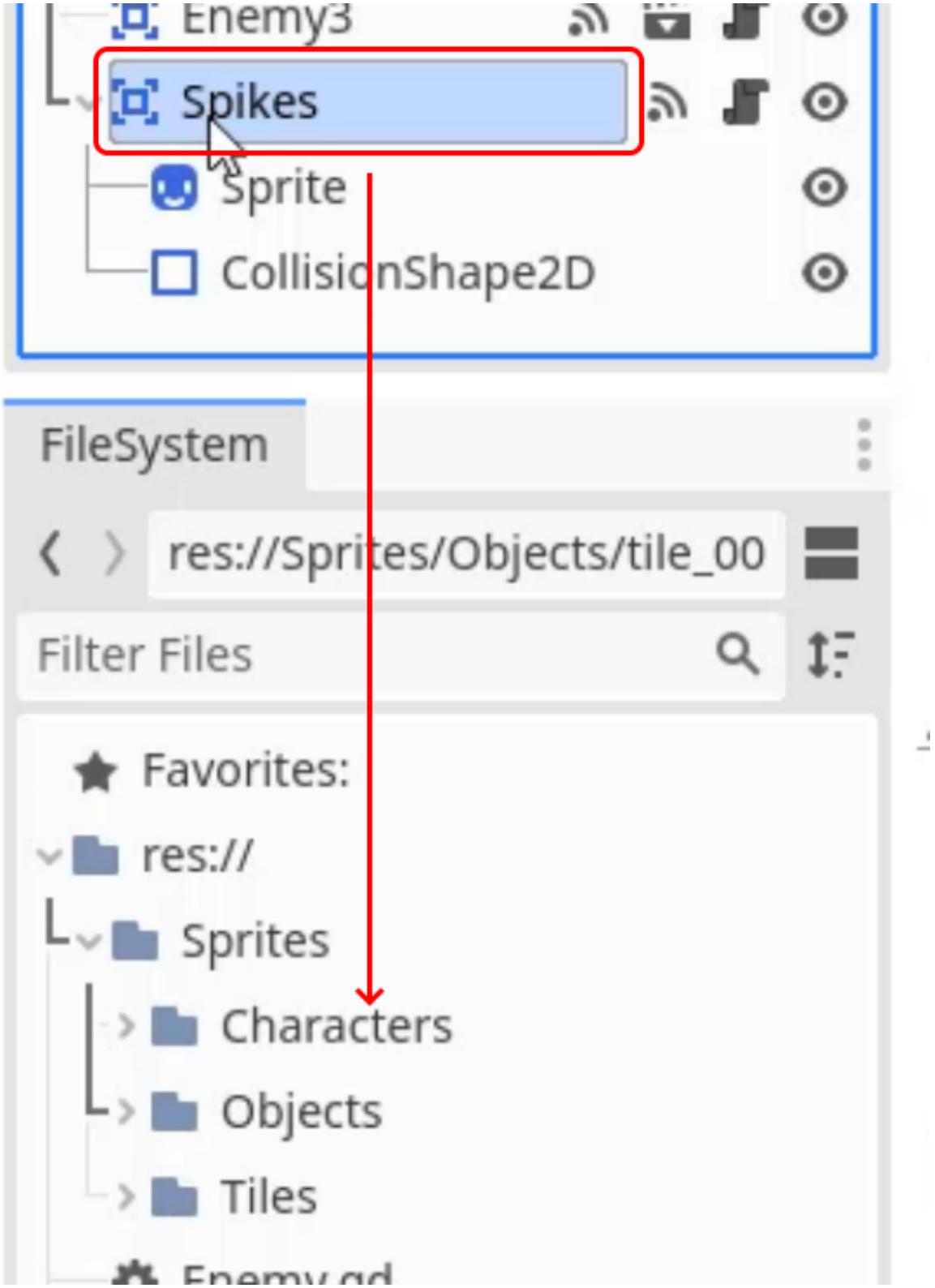
Once we have saved the script, we can go back into our 2D scene and press **Play**. We should be able to move our enemies and jump over them. If we collide with the spikes, the scene should restart.

In the next lesson, we will be looking at setting up coins. This will allow us to collect coins to increase our score, and have some text on the screen to display the current amount.

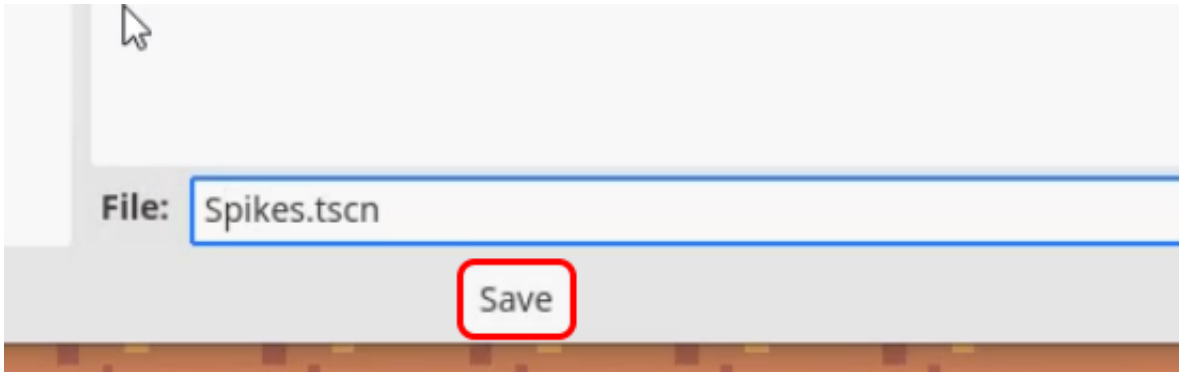
In this lesson, we will be looking at setting up a scoring system for our game. This system will allow us to keep track of the player's score and display it on the screen.

Setting up our Spikes Scene

Firstly, we need to drag our *Spikes* node into the *FileSystem* tab to save it as a new **Scene**.



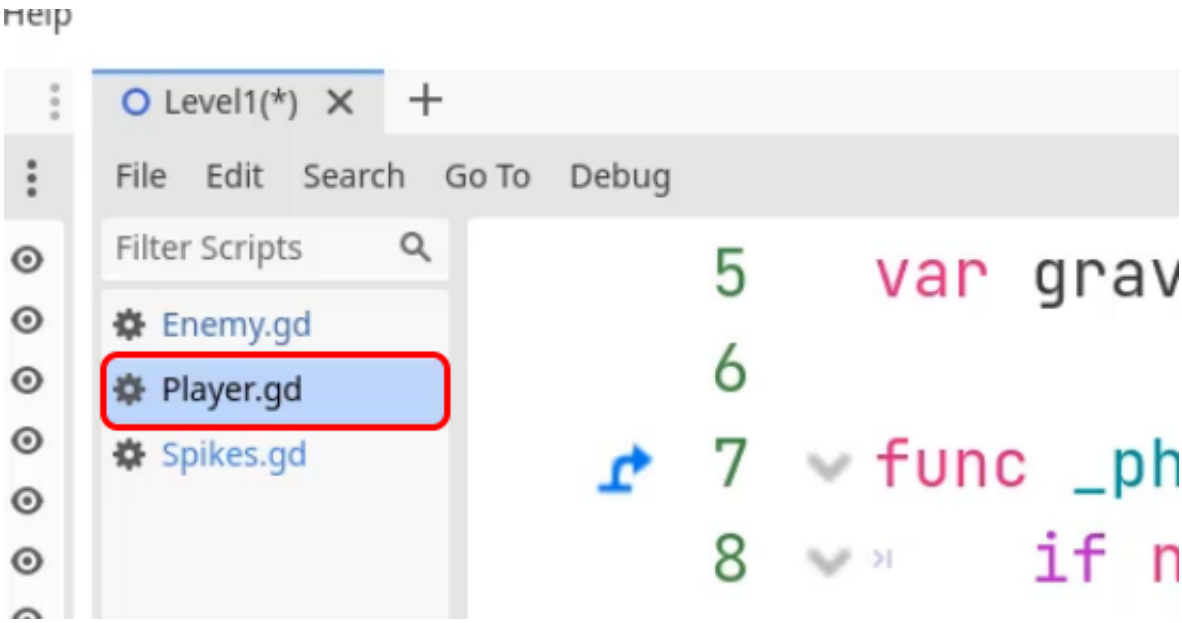
Make sure to save the file as *Spikes.tscn* and press **Save**.



Like with our *Enemy* scene, this will allow us to create multiple instances of the *Spikes* scene but still edit it using the parent scene at a later date.

Creating the Score Variable

The first step is to create a variable to store the score in our **Player.gd** script.



We will call this variable **score**, it will be of type **int** and give it a default value of **0**. This can be placed below the last *gravity* variable.

```
var score : int = 0
```

We can then create an **add_score** function that will take a parameter named **amount**. This can be placed below our *game_over* function.

```
func add_score(amount):
```

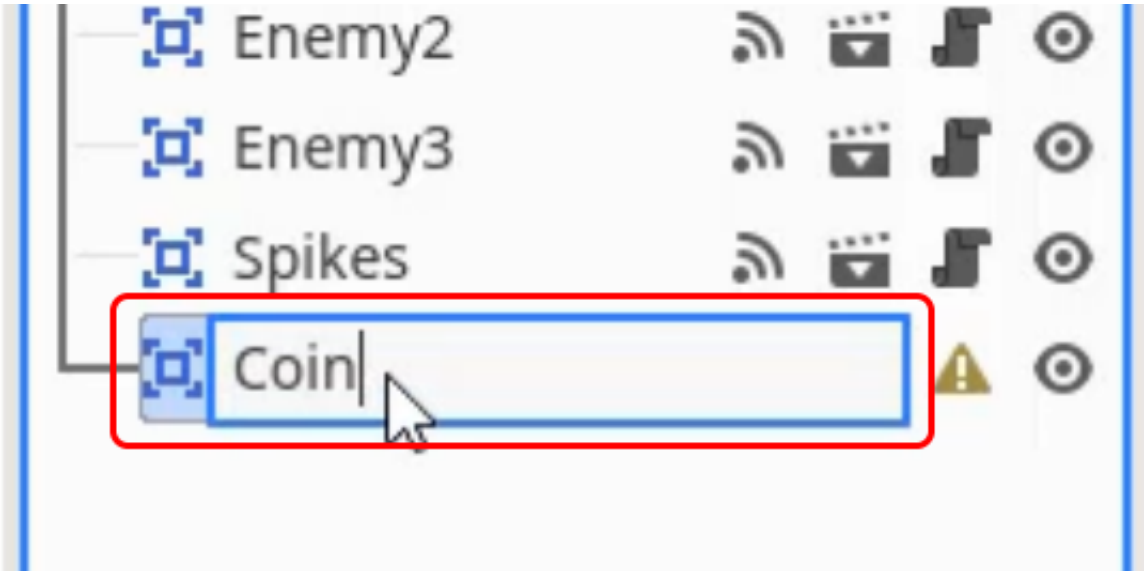
We will then use this function to add the *amount* value to our *score* variable. This will not be

everything in the function, however, this will be the base functionality for now.

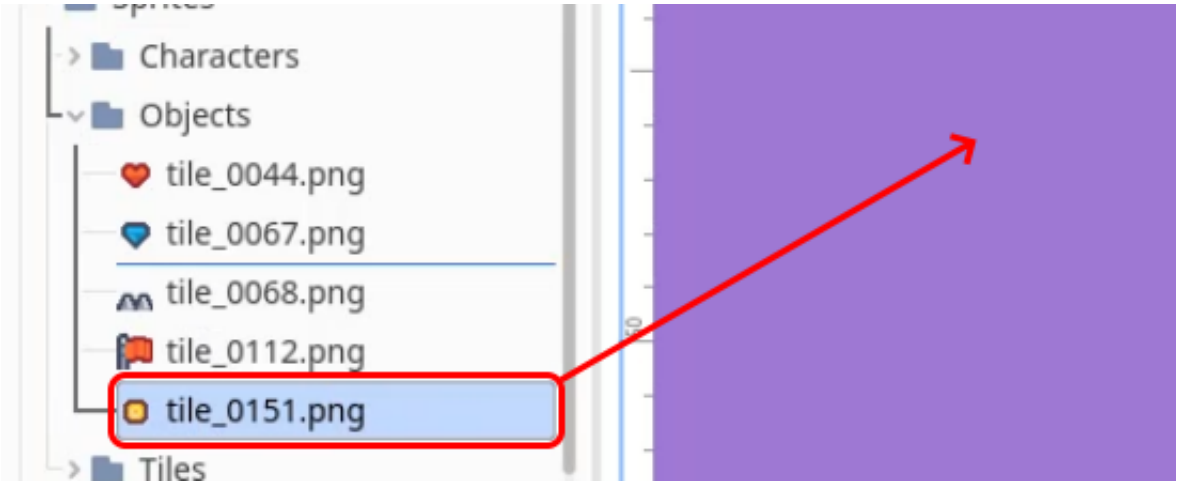
```
func add_score(amount):
    score += amount
```

Creating the Coin Node

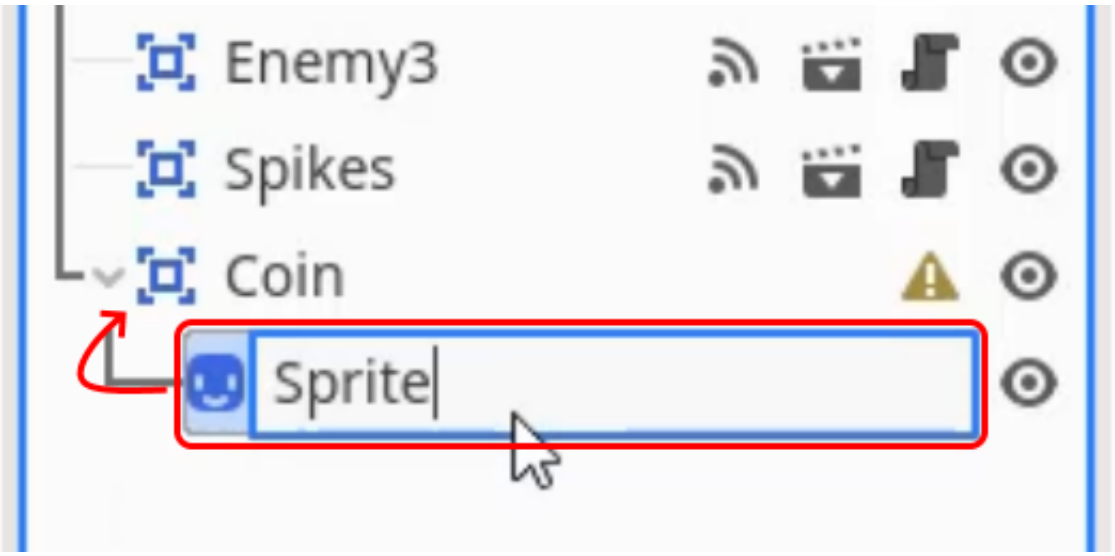
Like with our *Enemy* and *Spike* scenes, we will begin by creating an **Area2D** node and naming it *Coin*.



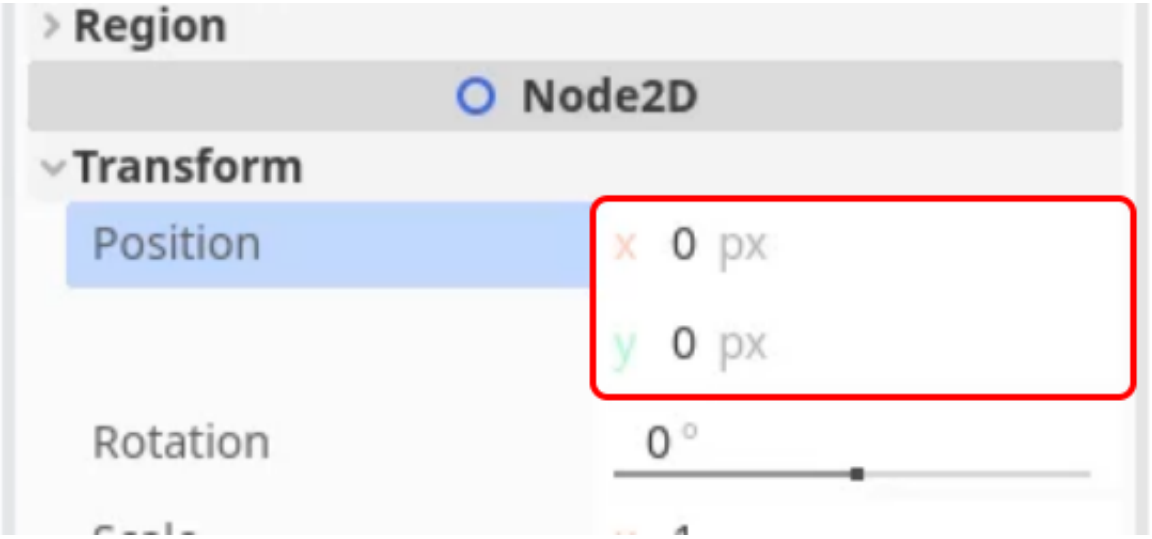
From the *Sprites* folder, drag in a sprite to use for our coin. From our sprite pack, we will be using a sprite named *tile_0151.png*.



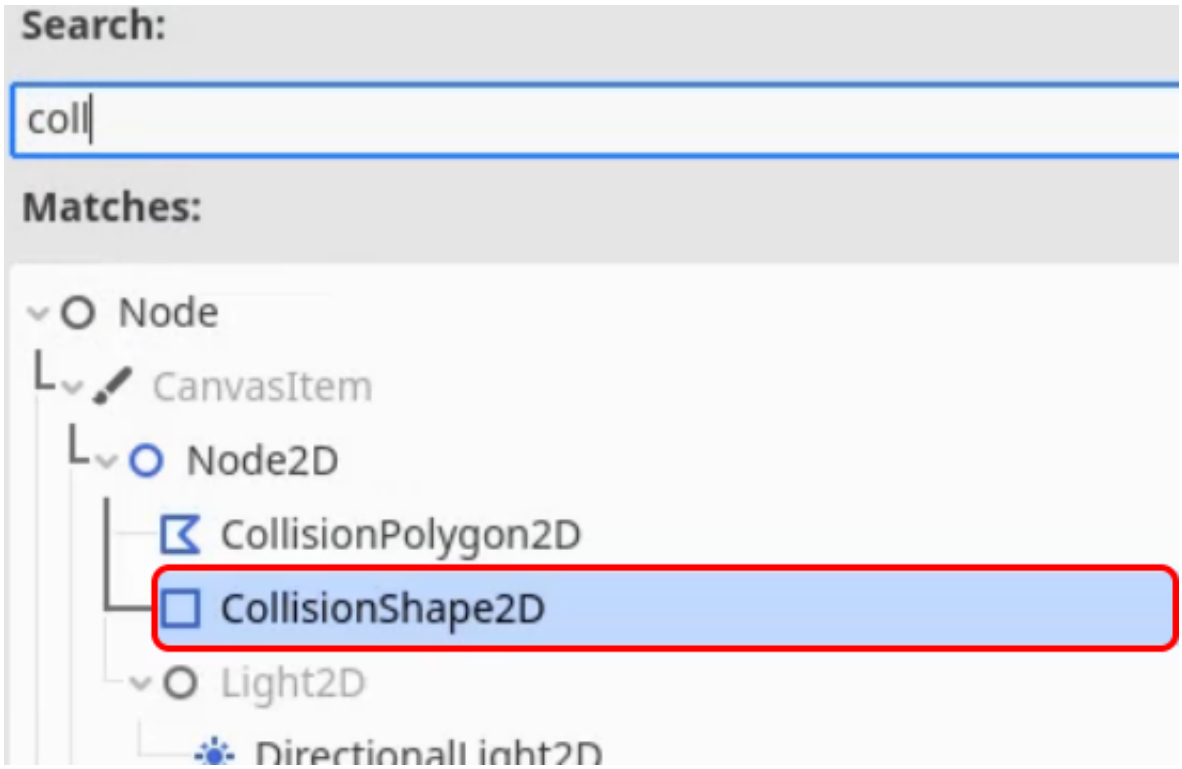
Make the sprite a child of *Coin* and rename it to *Sprite*.



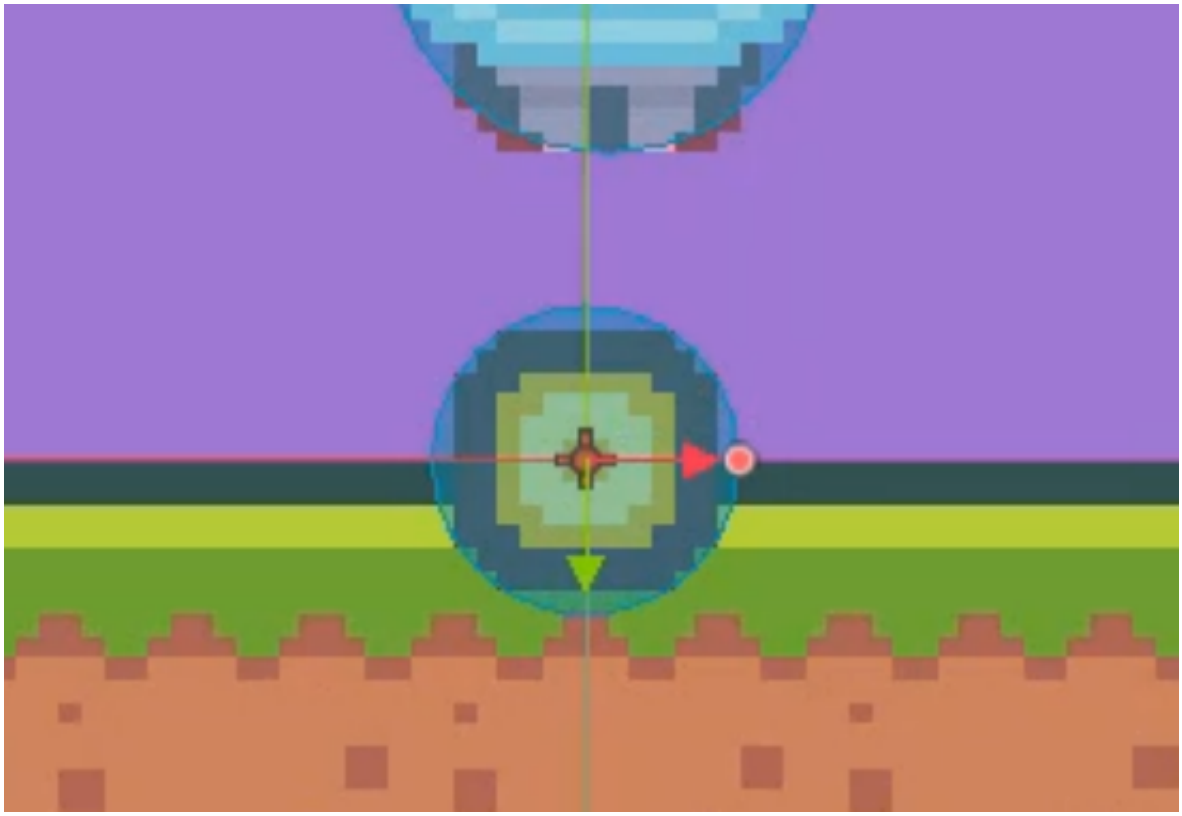
Make sure to center the *Sprite* by setting its position to **(0,0)**.



Create a **CollisionShape2D** as a child node of *Coin*.



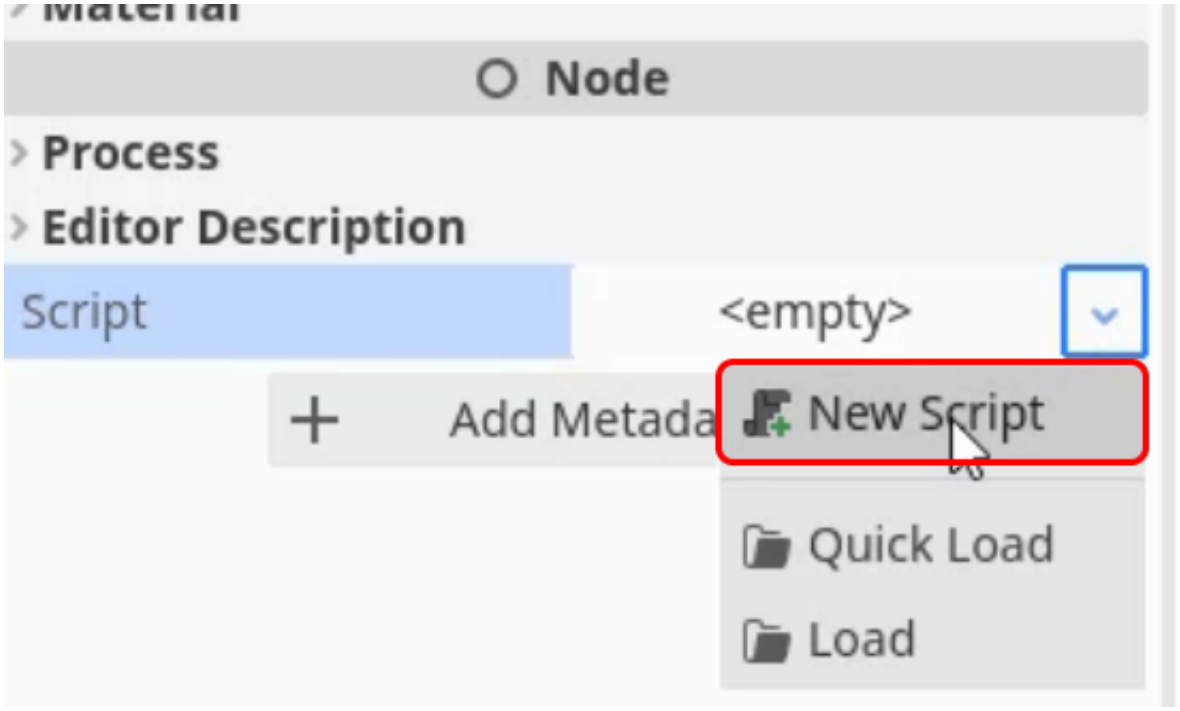
Then create a new **CircleShape2D** for the **Shape** property and resize it to the *Coin* sprite.



Finally, select the coin and **move** it to a good position on your level.

Adding Movement to the Coin

We want the coin to move up and down. To do this we will create a **New Script** on our *Coin* node, making sure to name it *Coin.gd*.



Firstly, you can remove the *_ready* function as we won't be needing it in this script. From here, we need to create some variables:

```
var bob_height : float = 5.0
var bob_speed : float = 5.0

@onready var start_y : float = global_position.y
var t : float = 0.0
```

- *bob_height* – This will show how many pixels our coin will move up and down. In this case, we use a default value of 5.0.
- *bob_speed* – This will be the speed at which our coin bobs up and down.
- *start_y* – This will keep track of our starting y position. To set it when the script begins running, we use the @onready tag. This will then allow us to use the global_position value, as using the @onready tag is like setting something in the _ready function.
- *t* – This will be the time value, a number that we will increase every frame.

Now, inside of *_process* we are ready to write the functionality for our coin movement. We will begin by **increasing t** by **delta** to keep track of the time that has passed. We can then pass the *t* value into a **sin** function which will give us a value along a **sin wave** between 1 and 0. We will save this in a variable called **d** for sin delta.

```
func _process(delta):
    t += delta

    var d = (sin(t * bob_speed) + 1) / 2
```

This equation will move along a sign wave at a rate of *t* multiplied by *bob_speed* and then we add 1,

then divide by 2 to turn the values between -1 and 1 into values between 0 and 1.

From here we can then add the line:

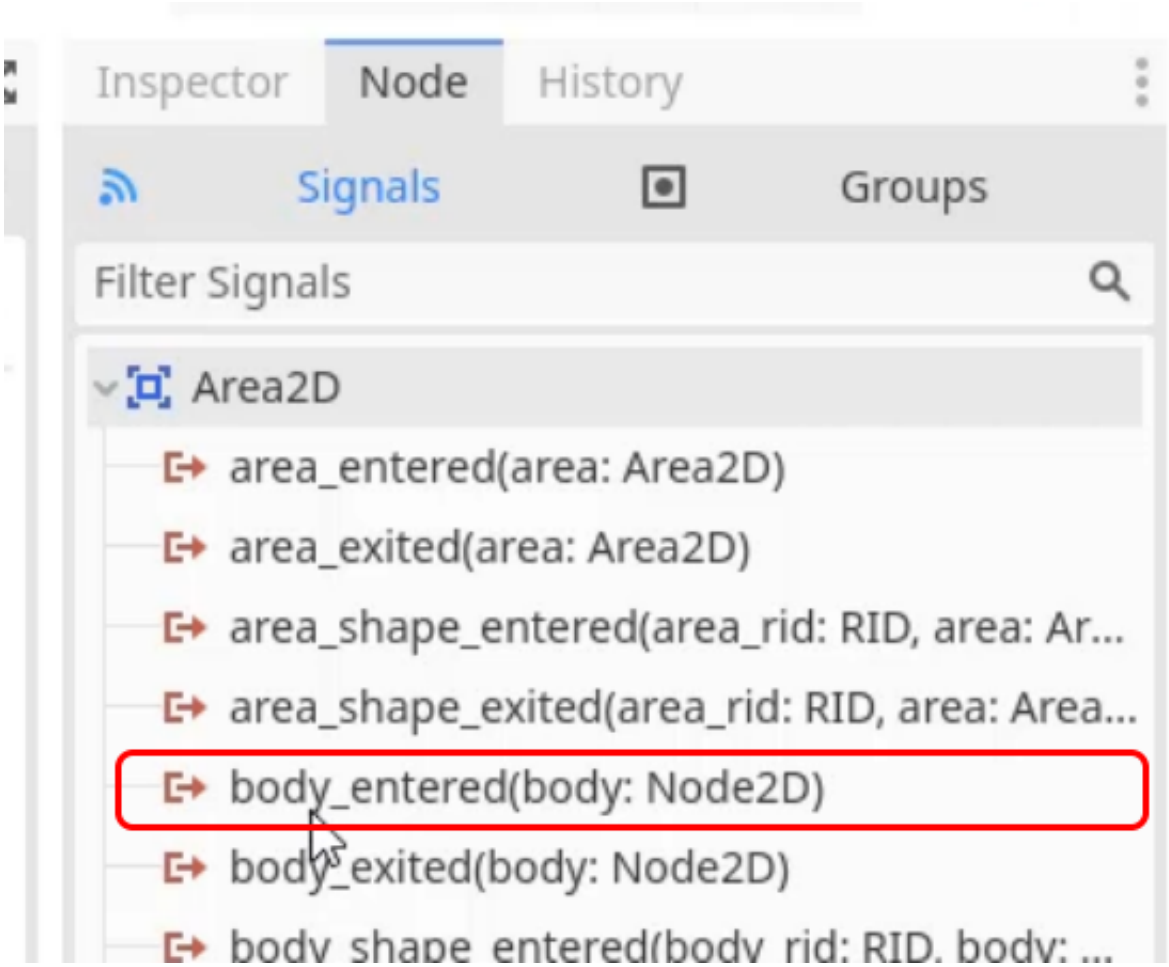
```
func _process(delta):
    ...

    global_position.y = start_y + (d * bob_height)
```

This will set the y position of the coin to be increased by the sin delta multiplied by the bob_height. So when the sin delta (*d*) returns 0 it will be at the *start_y* position, and when it returns 1 it will be at the maximum position set by *bob_height*.

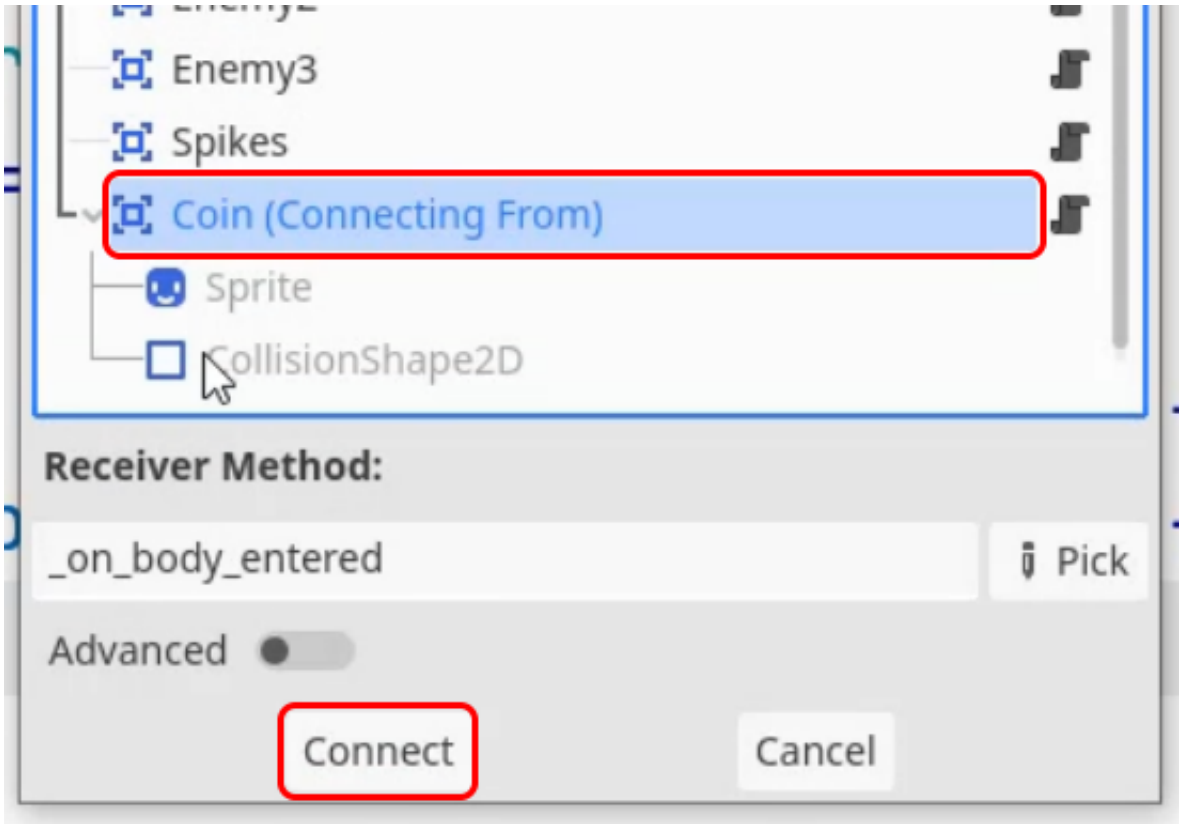
Now if you **save** the script (**CTRL+S**) and press **play** you will be able to see the coin bob up and down as intended. All of the variables can of course be edited to change factors like the height and the speed of the bobbing.

Collecting the Coin

Finally, we need to add code to detect when the player has collected the coin. Like with the *Enemy,* we will do this by connecting a **body_entered** signal from the *Signals* section under the *Node* tab.



This can then be connected to the *Coin* node and press **Connect**.



Then, in the *Coin.gd* script we want to add the following code to the new *_on_body_entered* function.

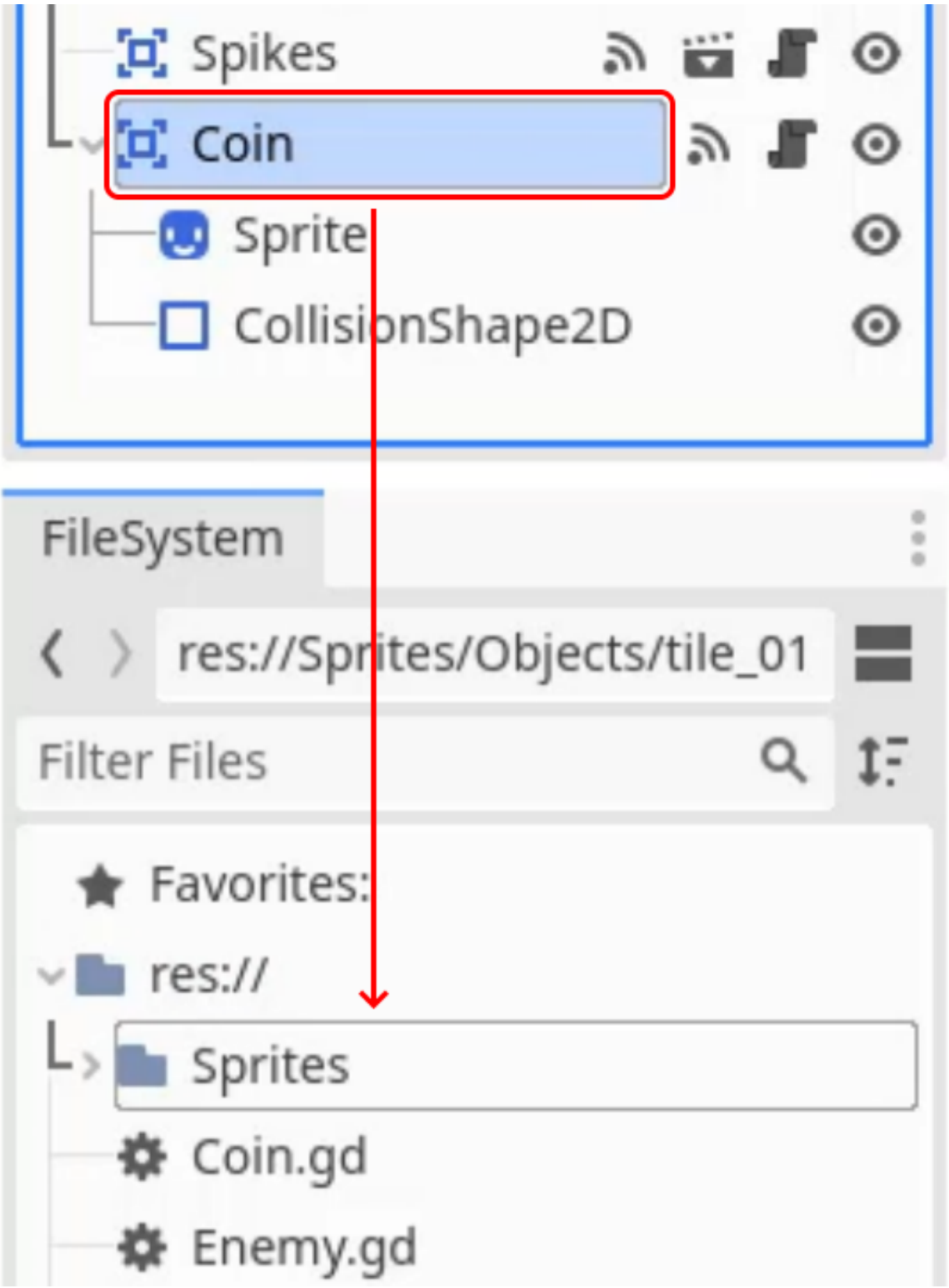
```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        body.add_score(1)
        queue_free()
```

This will check if the body collided with our *Player* node, and if it is, call the *add_score* function with a value of 1. Then we will use the *queue_free* function to destroy the coin node after it has been collected.

Now you can **save** the script and press **Play** and you will be able to collect the coin.

Creating a Scene from the Coin Node

We finally need to create a scene from the *Coin* node, so that we can have multiple instances of this. Do this by dragging the *Coin* node into the *FileSystem* tab.



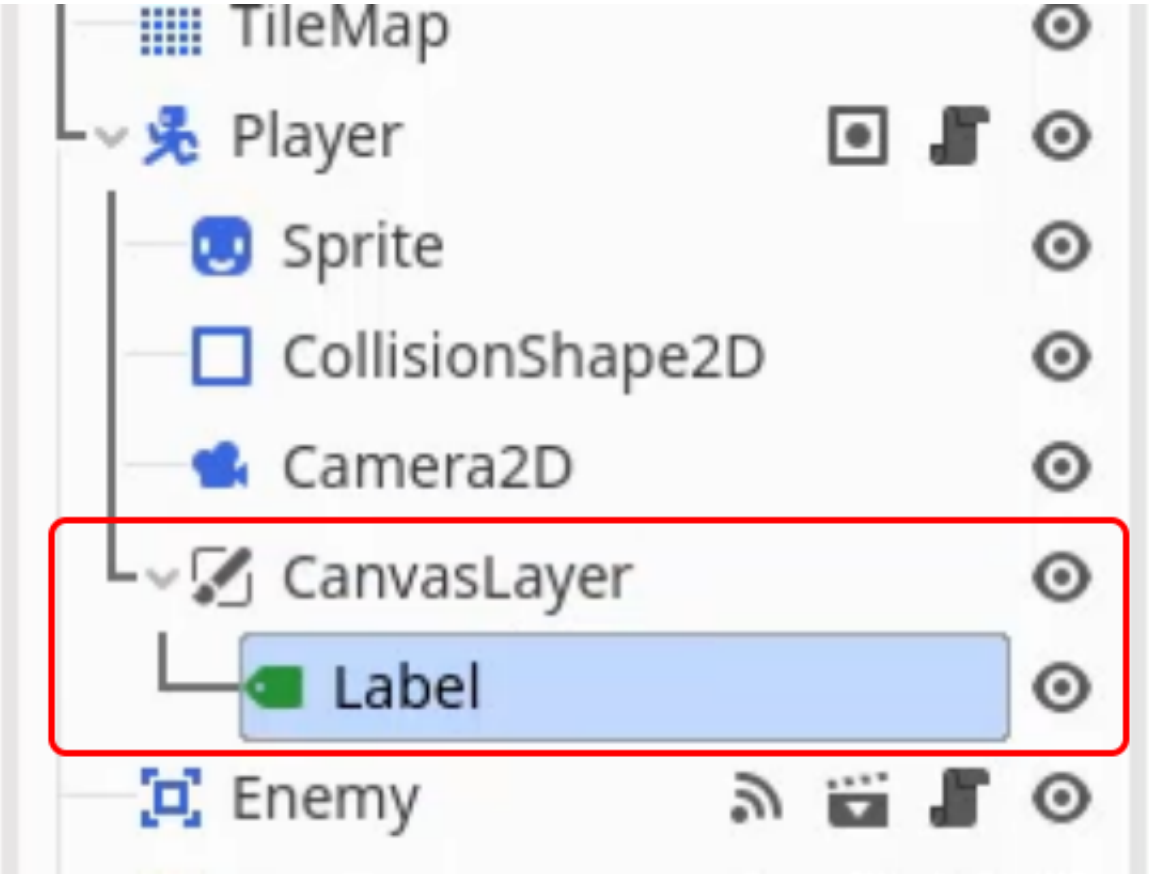
From here, this can be named *Coin.tscn* and **Saved**. You can now use this scene to create coins across your level for the player to collect and increase their score.

In the next lesson, we will look at displaying the player's score on the screen.

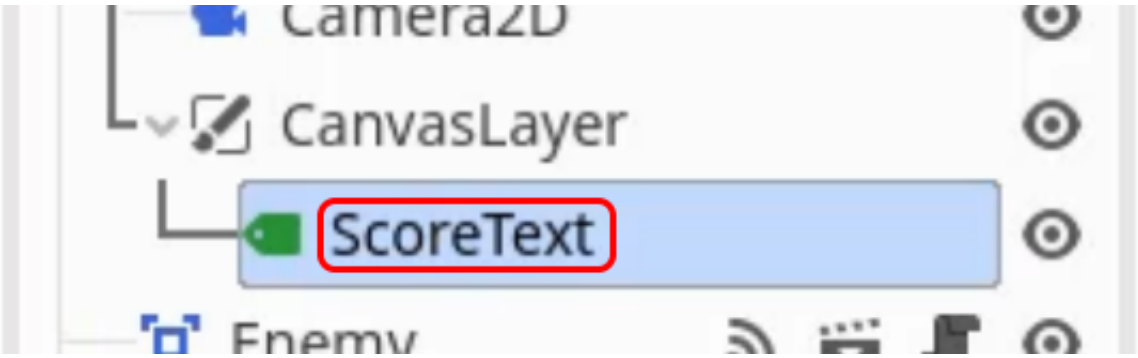
In this lesson, we will be setting up our score text for our game. This will allow us to see our score as we progress through the game.

Creating the Score Text

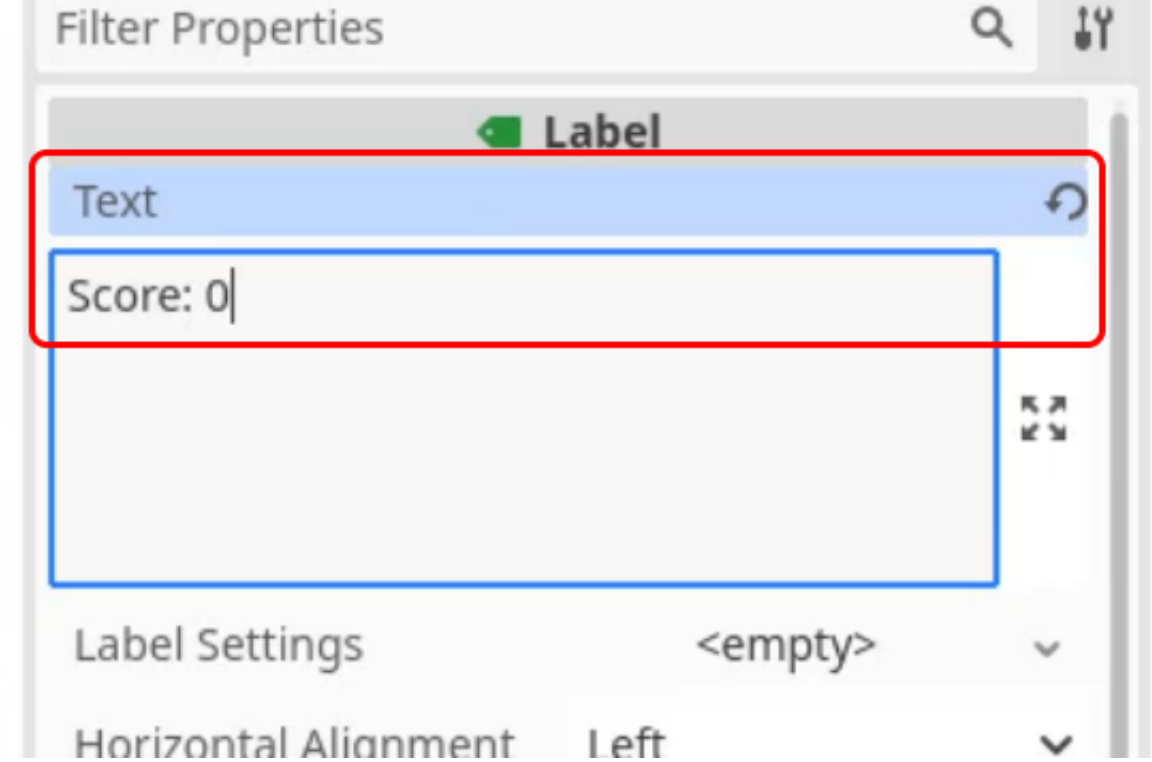
First, we need to create a **Canvas Layer** as a child node of the **Player** node, this is a node that renders UI elements on our screen. We will also add a **Label** node as a child to the *Canvas Layer*.



We can then rename the label to *ScoreText*.

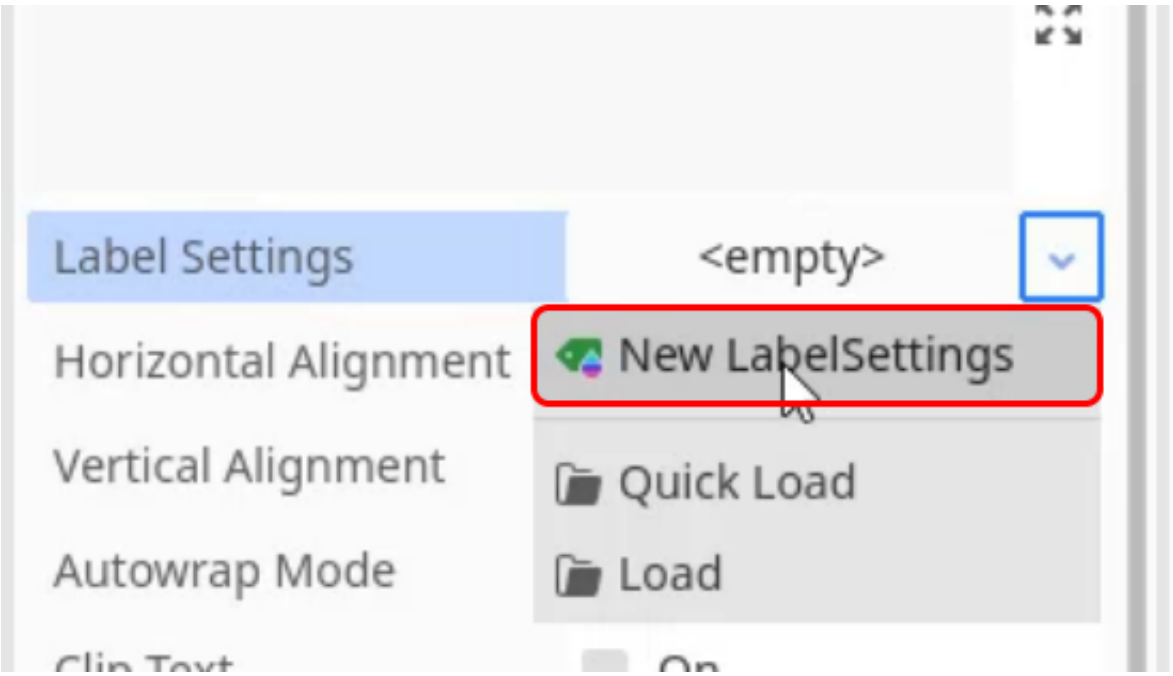


Next, select the *ScoreText* node and press the **F** key to focus on it. In the inspector, add the text *Score: 0*. This will be the text that is displayed on the screen.

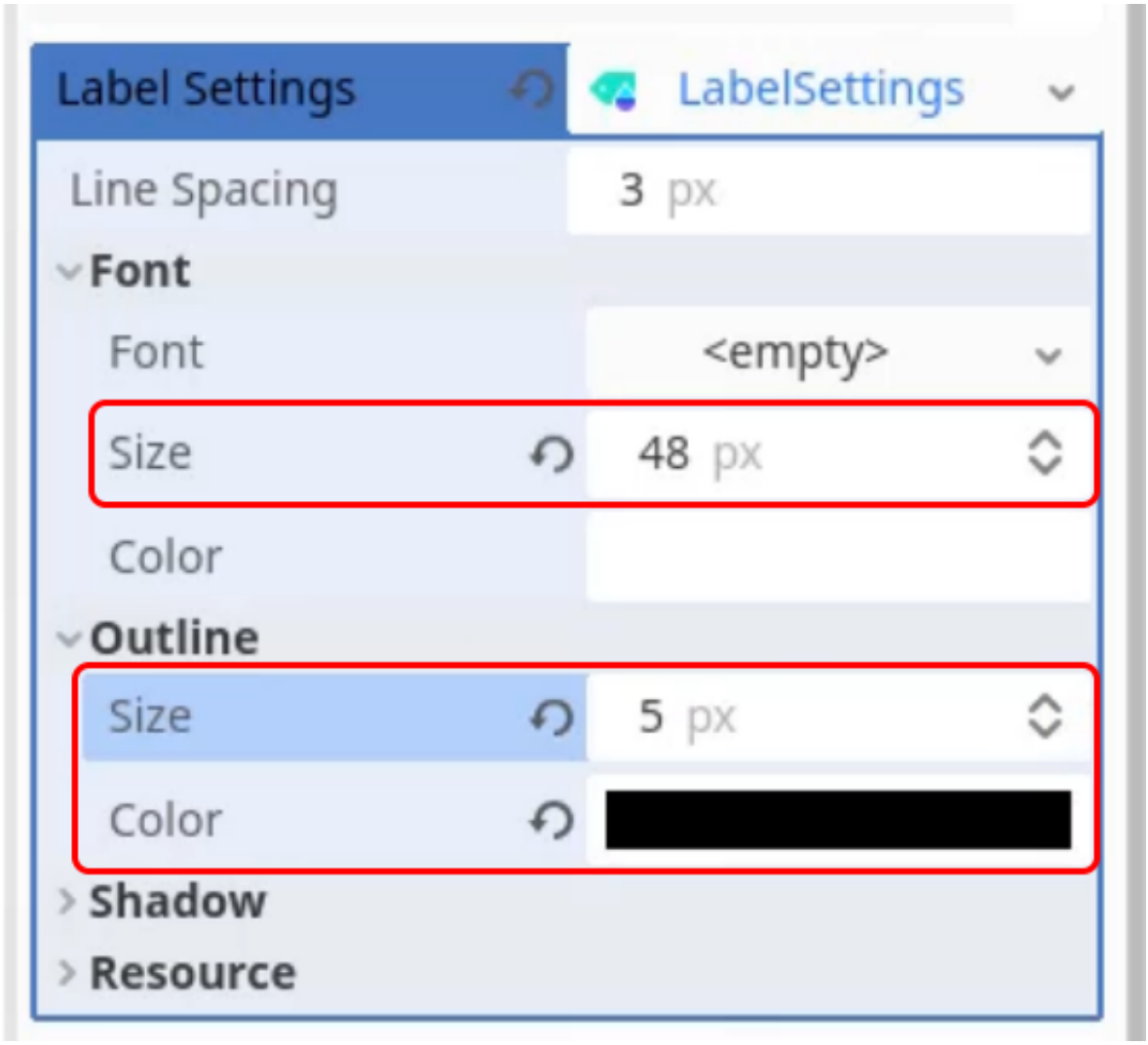


We can position the *Score Text* wherever we want relative to the screen. To do this, scroll out and you will see a faint blue rectangle. This represents the screen resolution. We can then use the *move* tool arrows to position it in the space.

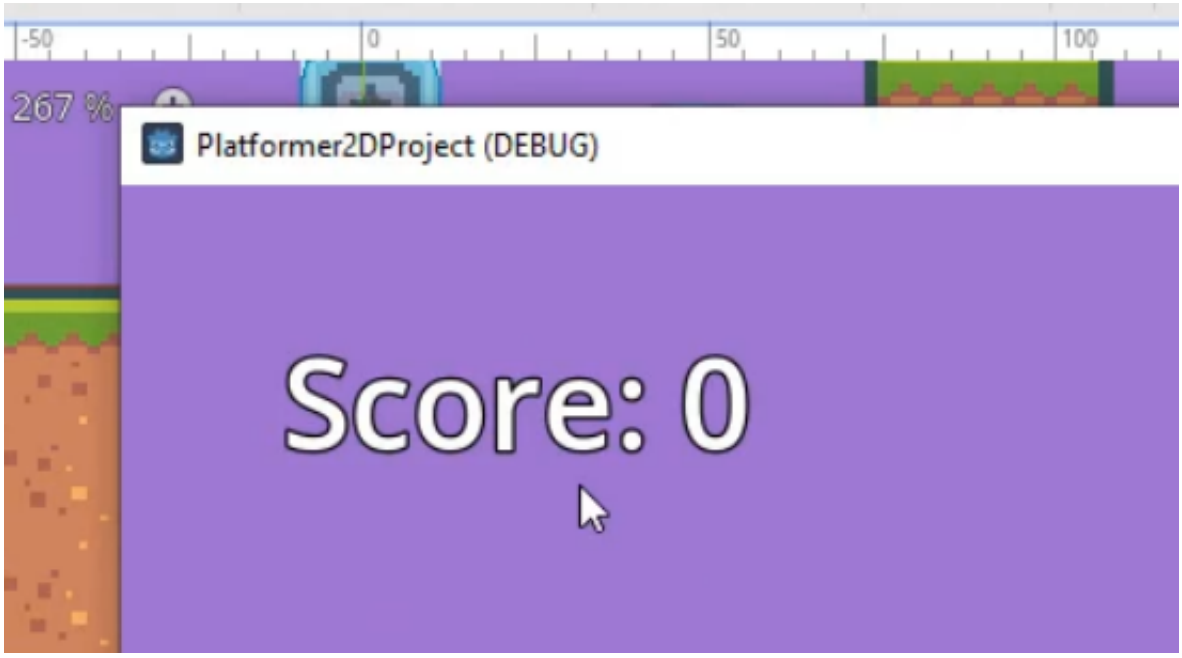
We can also customize the *Score Text* by changing the font size, outline, and shadow. To do this, in this *Inspector* choose **New LabelSettings** in the **Label Settings** property.



We will increase the **font size** to **40** pixels and add a **black outline** that is **5** pixels wide. You can change any of the values you like to your own taste of course.

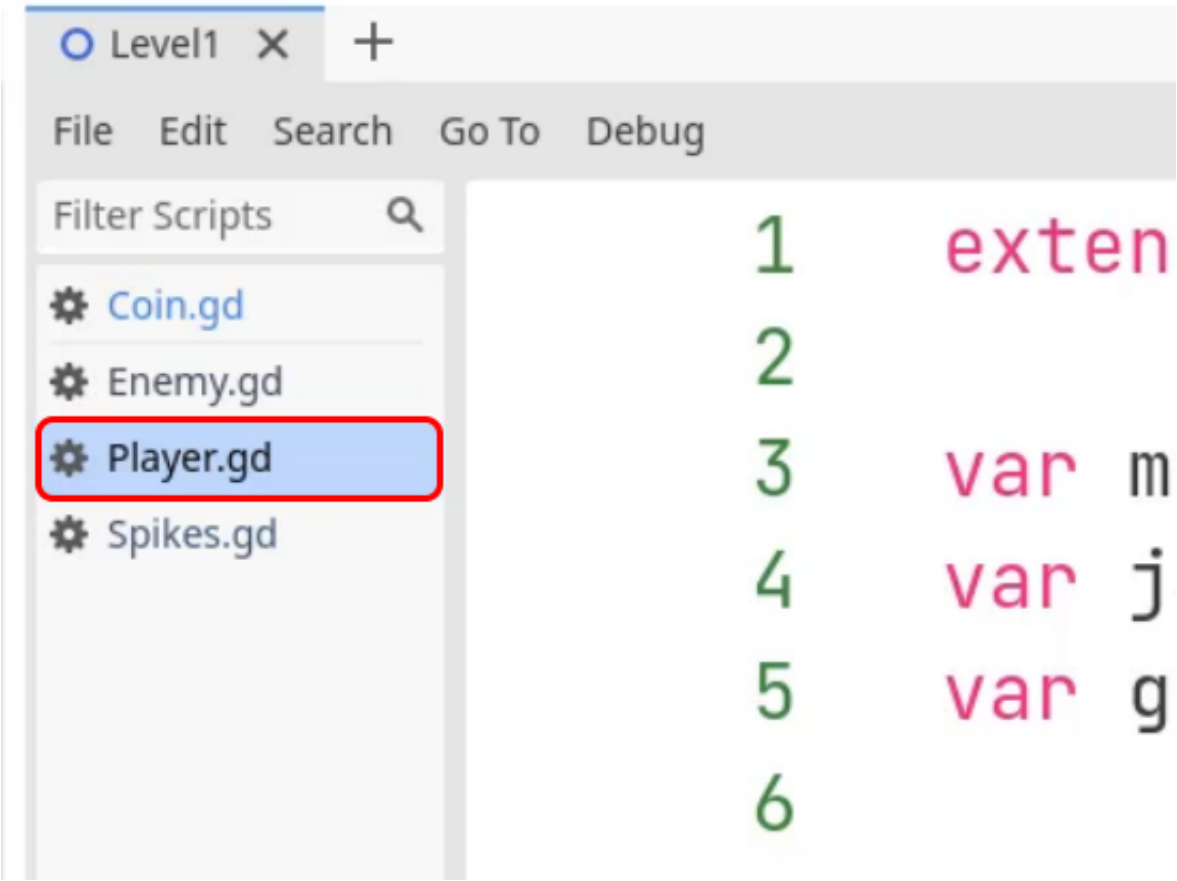


This will make the *Score Text* display on our screen, however, we still haven't added any functionality to it.



Adding the Score Text to the Script

Now, we need to add the score text to our *Player.gd* script and hook it up to the function we created previously.



Underneath the score variable, we will add a new variable to keep track of our *Score Text*. We will use the @onready tag and the get_node function to set it to our *Score Text* node when the script is loaded.

```
@onready var score_text : Label = get_node("CanvasLayer/ScoreText")
```

Finally, in the *add_score* function, we want to update our score_text. We will use the *str* function to combine our integer *score* variable with a string.

```
func add_score(amount):
    score += amount
    score_text.text = str("Score: ", score)
```

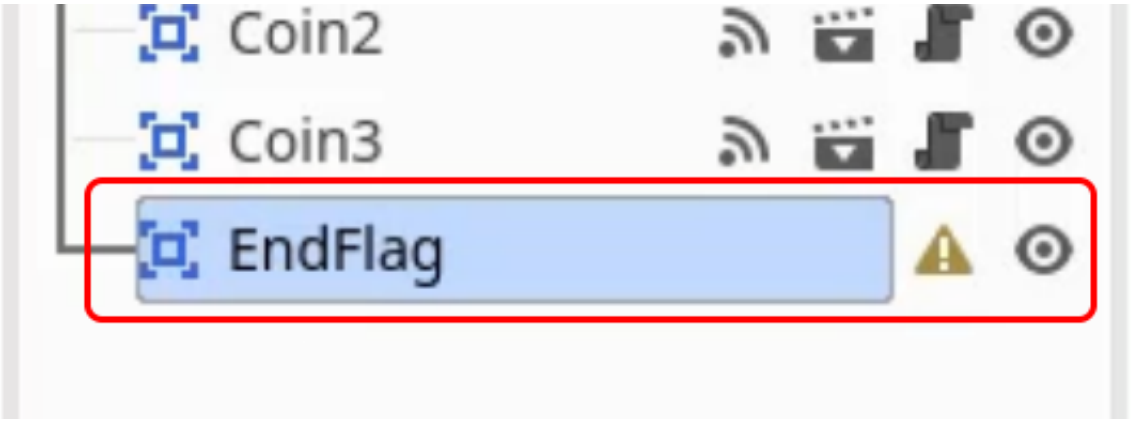
Now you can press **Play** and collect the coins in your level and you will see the score increase. Feel free to duplicate the coin (or drag the *Coin.tscn* scene into the level from the *FileSystem*) to create more coins across your level. Due to the way we handle the *game_over* state of the game, when we restart the score will also be reset as intended.

In the next lesson, we will be looking at setting up multiple levels.

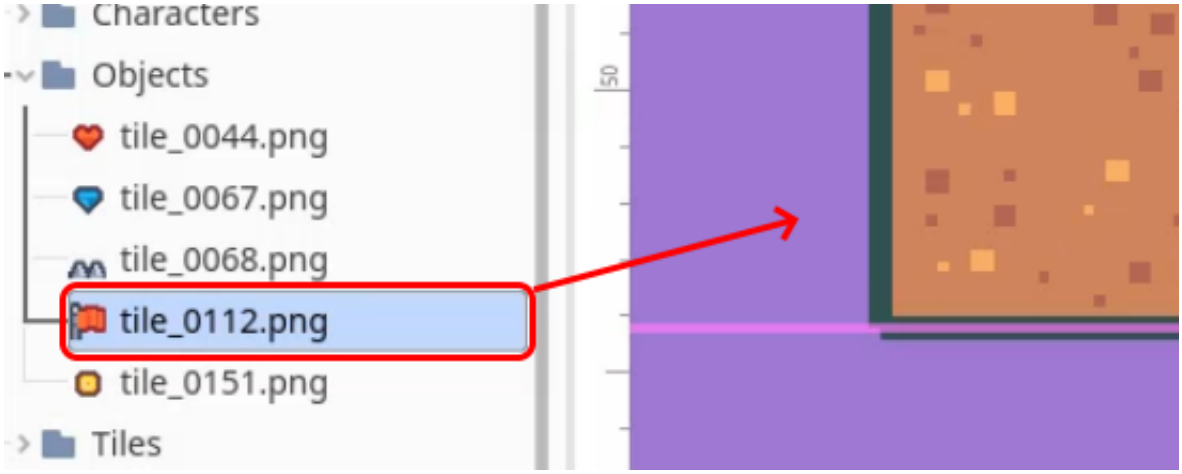
In this lesson, we will set up an end flag for our game. An end flag is a node that, when interacted with, triggers a change in the game. In our case, we will be using it to change to a new level.

Creating the Node

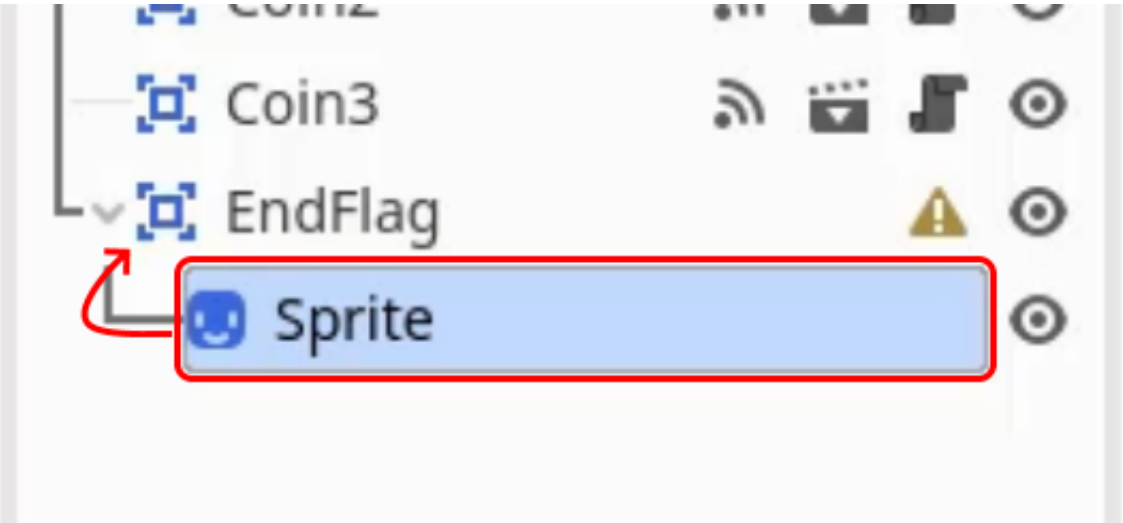
To begin with, we will create a new **Area2D** node, as we have previously, and rename it to *EndFlag*.



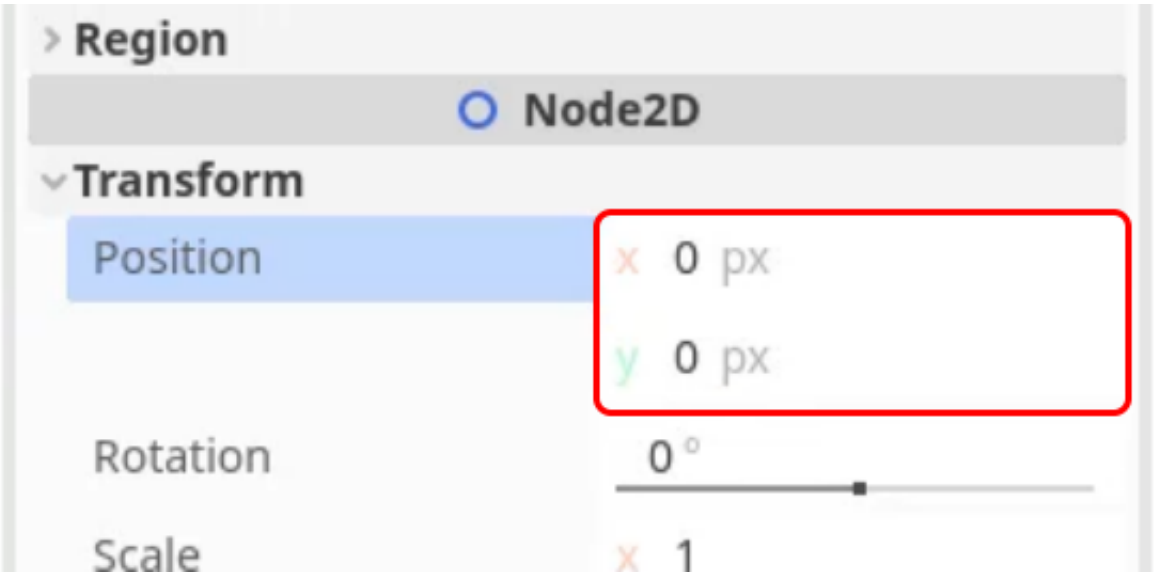
We will then add a sprite from the *FileSystem*, in our case, we will use the *tile_0112.png* sprite from our asset pack.



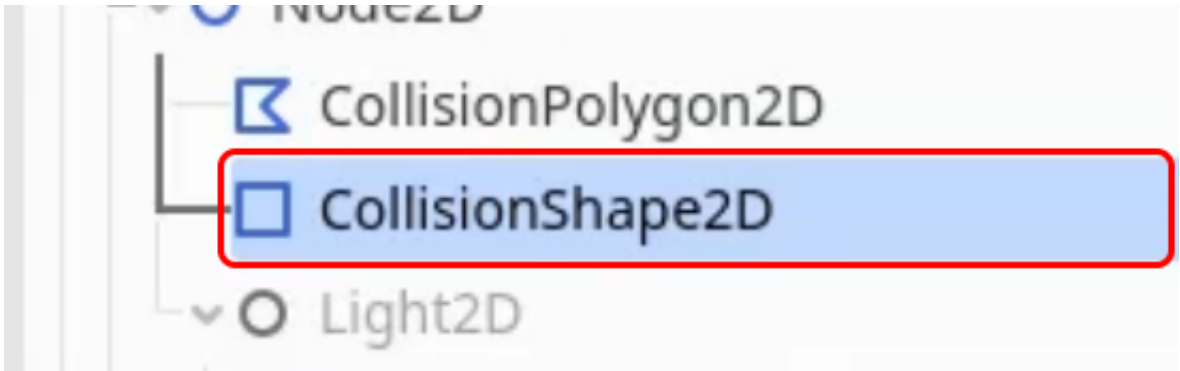
Finally, make it a child of the *EndFlag* node and rename it to *Sprite*.



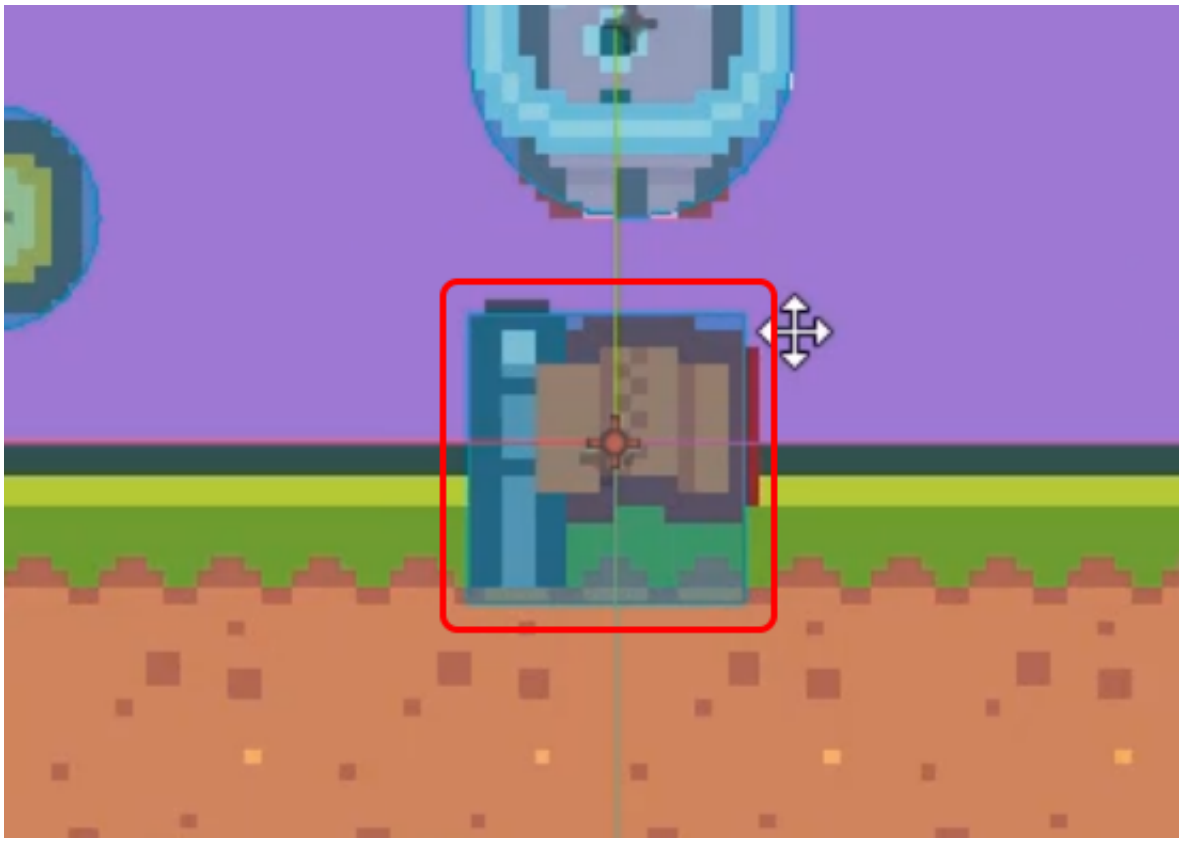
Remember to center the *Sprite* by setting the **Position** value to **(0,0)**.



Next, add a **CollisionShape2D** node as a child of our *EndFlag*.

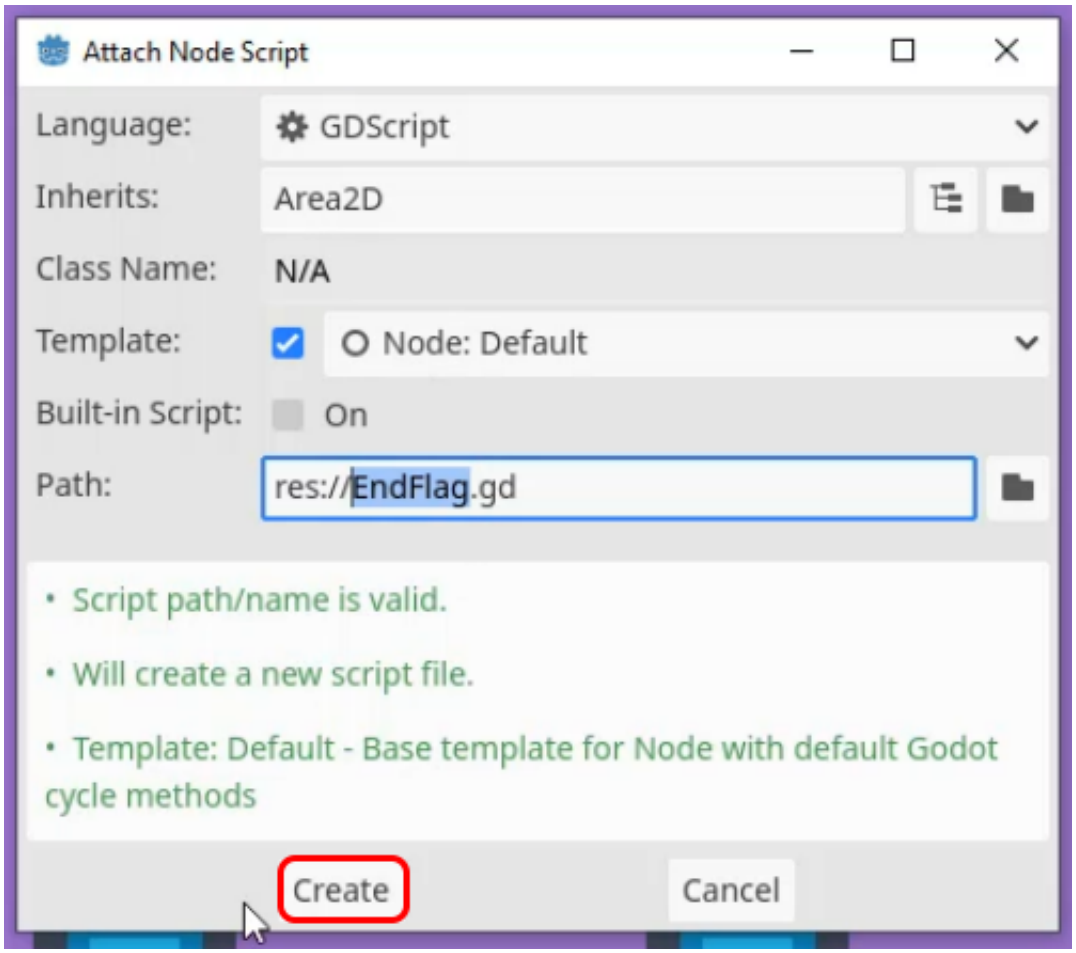


This can then be a **New RectangleShape2D** for the **Shape** property, and make sure to **scale** the collider to the size of your sprite.



Creating the Script

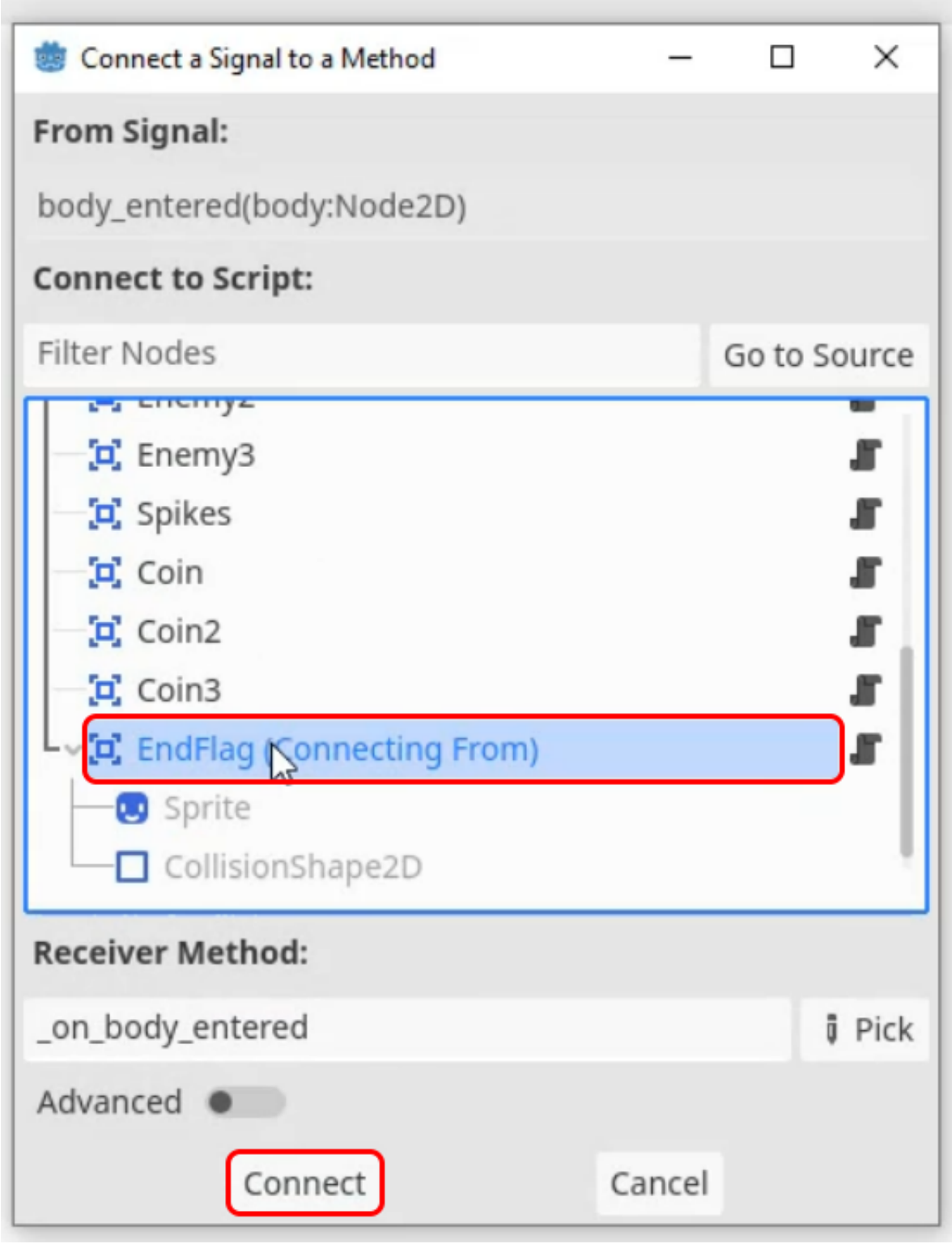
Next, select the *End Flag* node and create a **new script**. Call it *EndFlag.gd* and make sure it extends from area2d.



Delete the default functions, which will leave you with the below code in the file.

```
extends Area2D
```

Under the *Node* tab, we can then add a **body_entered** signal to our *EndFlag* node, like with previous nodes.



Then, create a variable with the *@export_file* tag of type with the *"*.tscn"* modifier. We will then call this *next_scene* to show it stores the next scene.

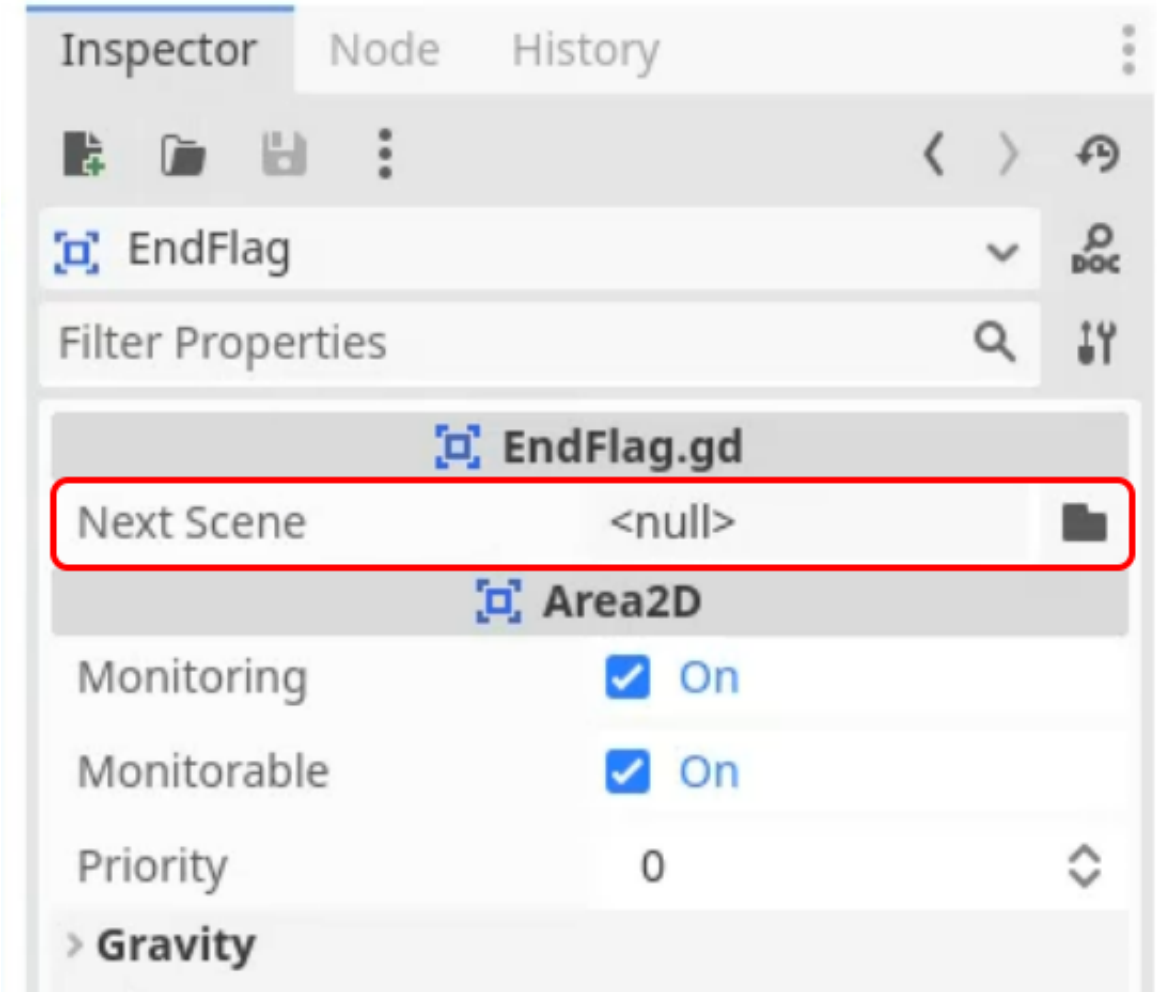
```
@export_file("*.tscn") var next_scene
```

In the *_on_body_entered* function, we will then add the code to get the tree and change the scene to

the file stored in the variable, after checking whether the collided body is the player node.

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        get_tree().change_scene_to_file(next_scene)
```

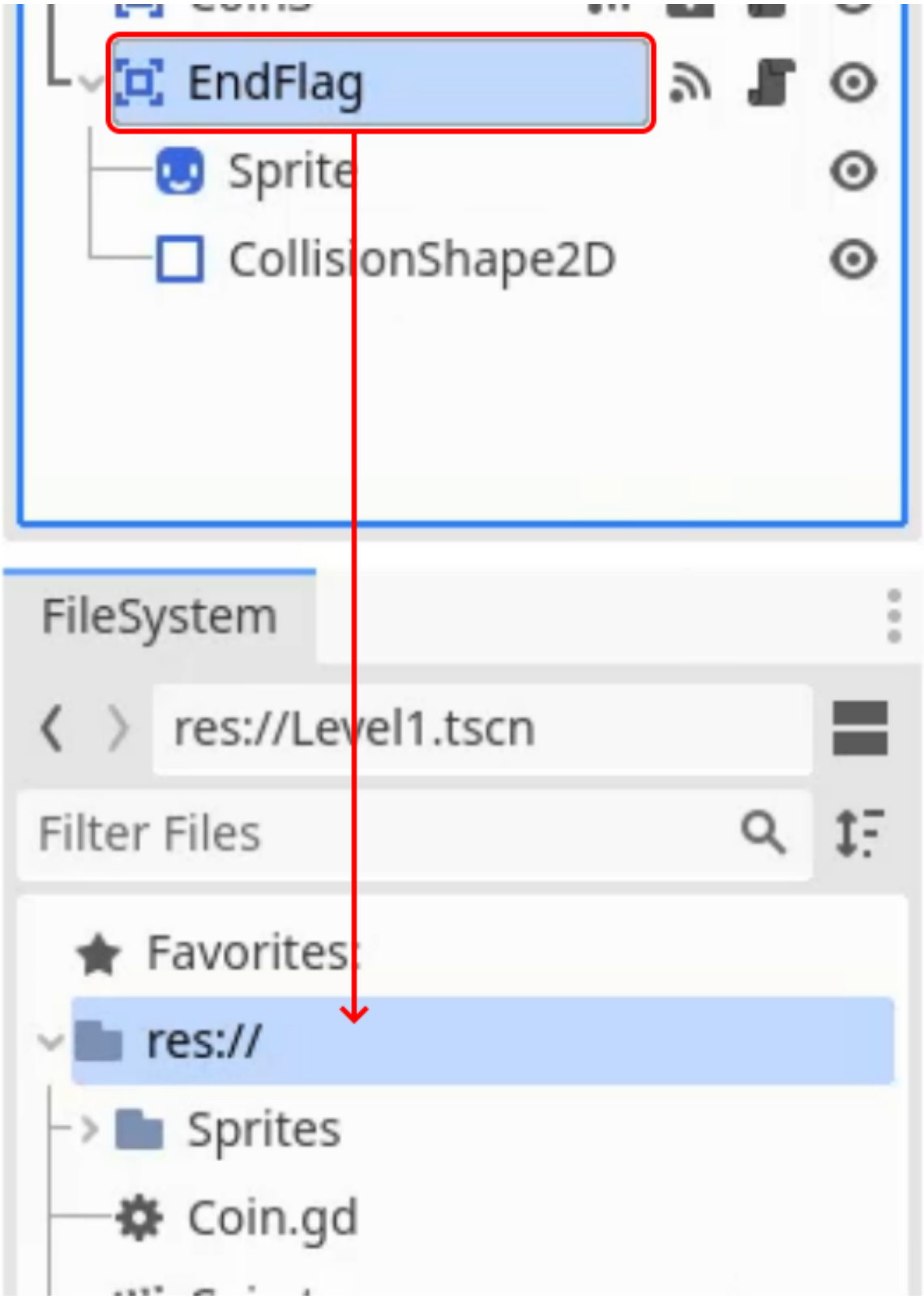
Now, return to the **2D** scene view and you will find the **Next Scene** property is set to null.



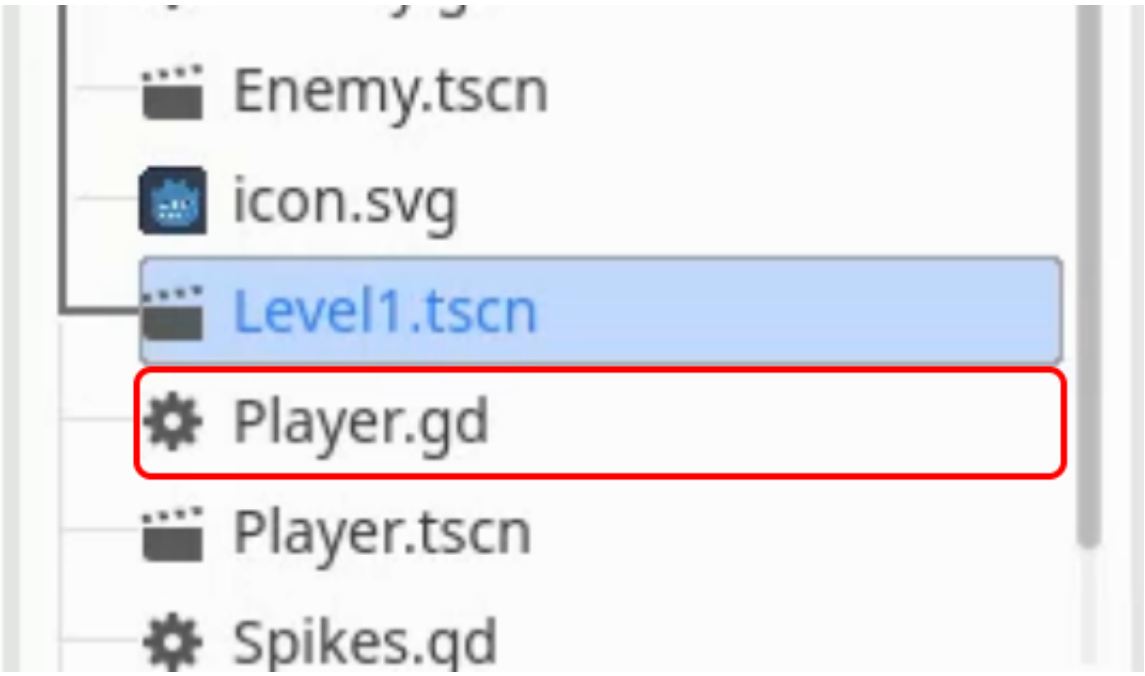
We now need to create a new scene so that we can set this value. To test the *EndFlag* we will set the *Next Scene* property to be our *Level1.tscn* scene that already exists in the FileSystem.

Now if you press **Play** and let your player character reach the *EndFlag* node, you will notice the scene restarts, as we load into the *Next Scene* value, which we set as our *Level1.tscn* scene.

Finally, we need to drag the *EndFlag* node into the *FileSystem* to create a scene out of it, allowing us to have multiple instances of the flag.



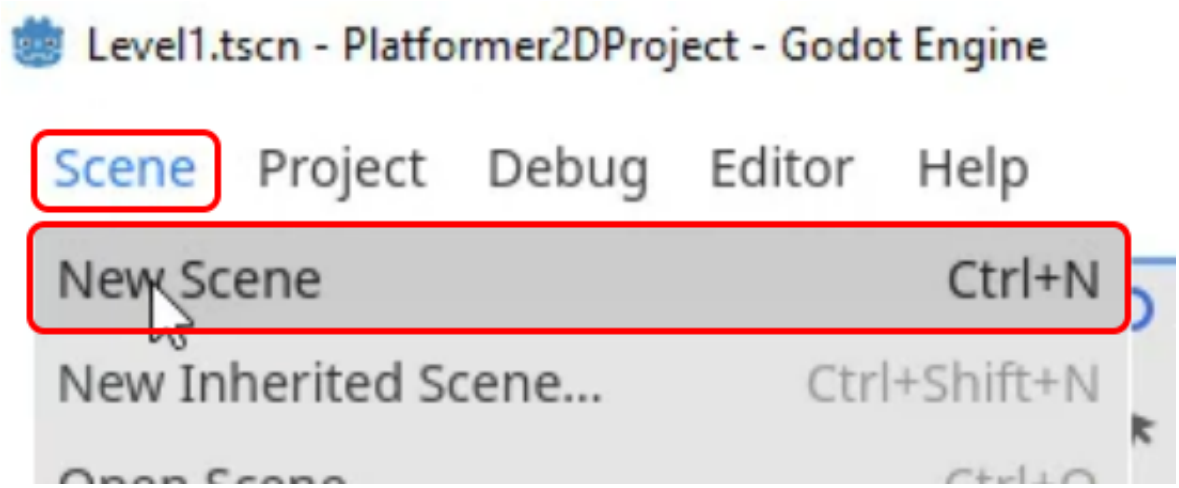
We also need to turn our *Player* node into a scene.



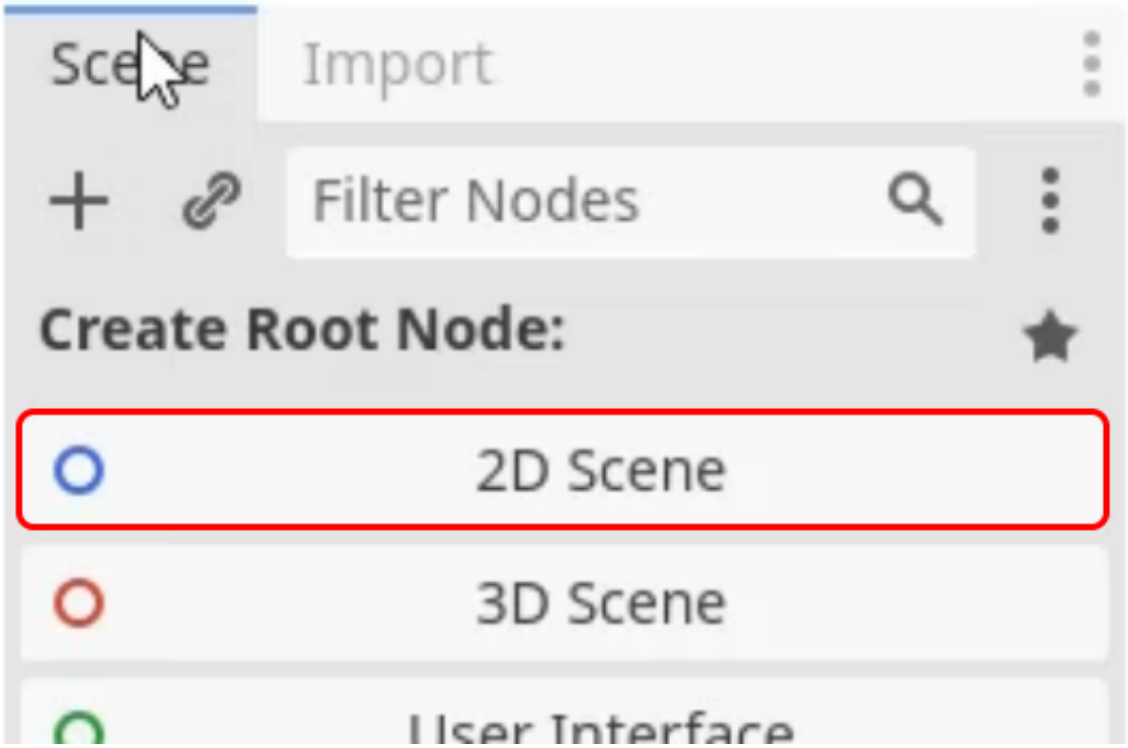
Having everything as a scene will allow us to construct a new level easily.

Creating Multiple Scenes

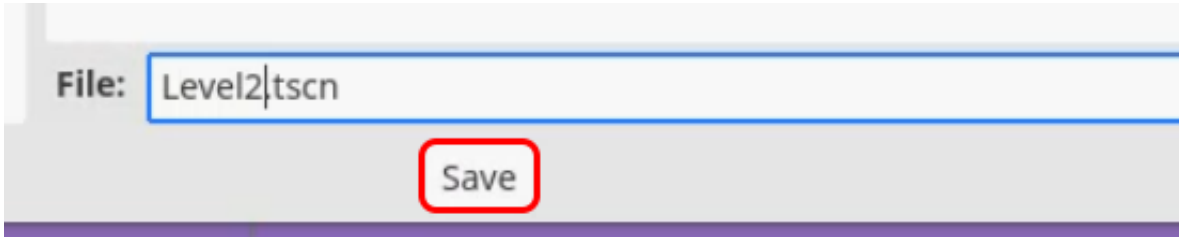
To begin with, press **Scene** and select **New Scene** to create a new level.



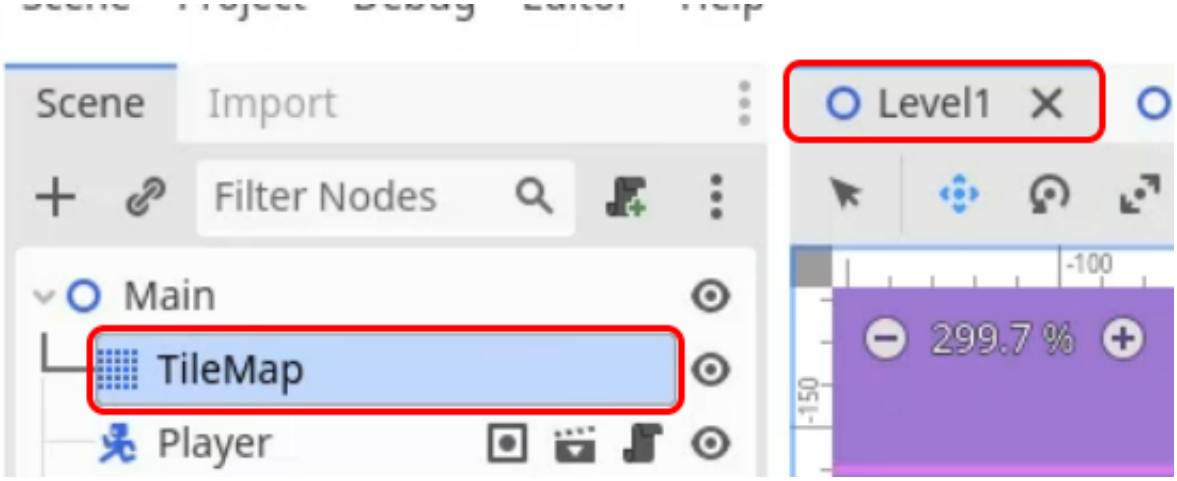
From here select **2D Scene** and rename the root node to *Main* again.



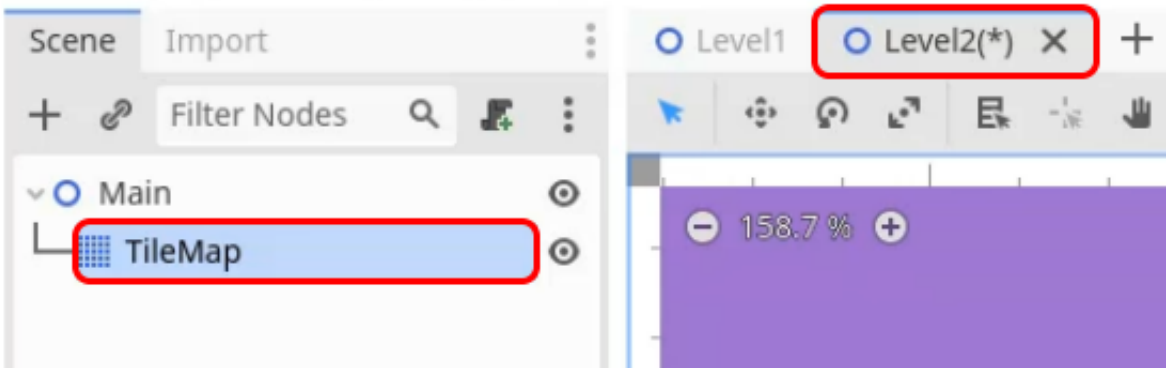
We can now save the scene as *Level2.tscn* (**CTRL+S**).



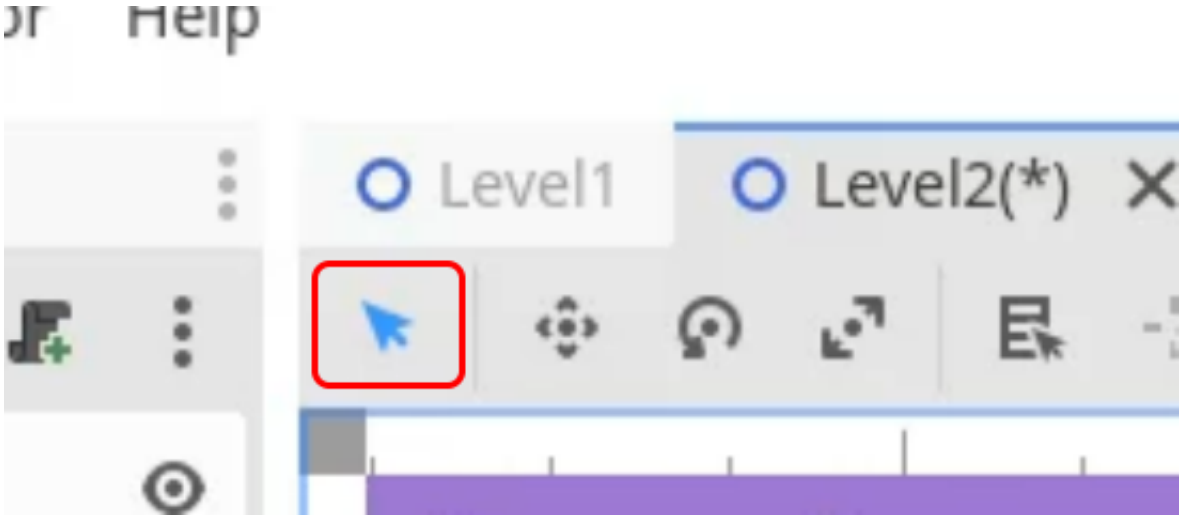
Now return to the **Level1** scene tab and copy the *TileMap* node (**CTRL+C**).



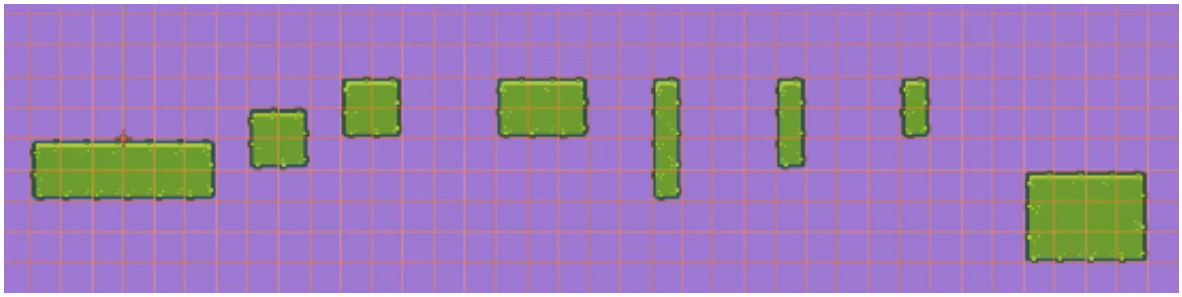
We can then paste that in the **Level2** scene tab and paste the *TileMap* node (**CTRL+V**).



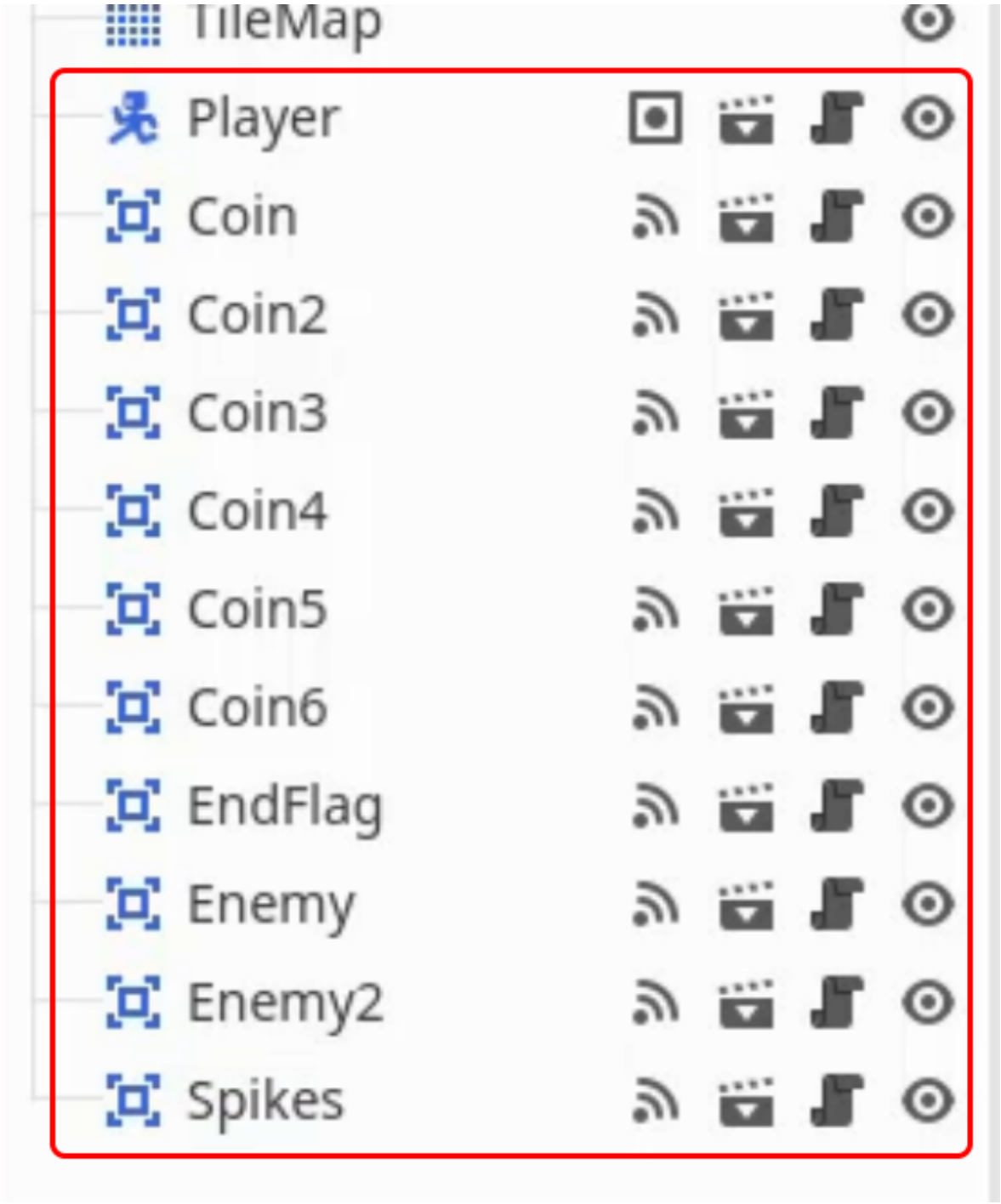
Then using the **Select** tool to draw on the tilemap, replace the pasted tiles to redraw your *TileMap* for your new level.



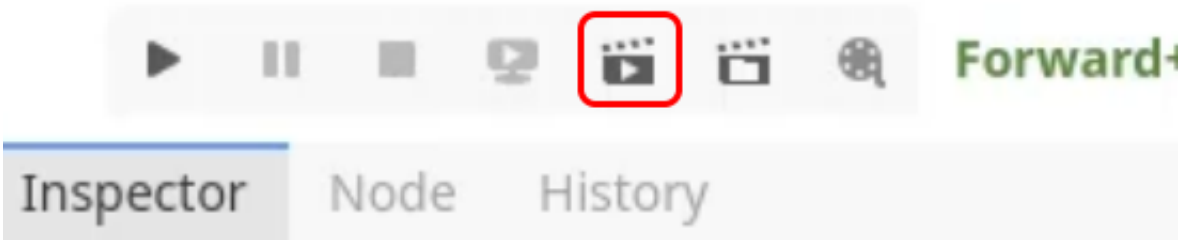
This will give us a new map to play on.



From here, drag the **Player**, **Enemy, Spikes, Coin,** and **EndFlag** scenes into your level to create some gameplay.



You can press the **Run Current Scene** (**F6**) button to test this scene, the normal *Play* button will simply run the default scene, which is currently set to *Level1*.



Connecting Level1 to Level2

To connect our levels, we want the *EndFlag* on *Level1* to have a **Next Scene** property of **Level2.tscn**.

Now if you press **Play** you can touch the *EndFlag* and you will be teleported to the next level. You now have the skills to set up as many levels as you like and link them together using the *EndFlags* we set up.

Creating a 2D Platformer

Creating a 2D platformer is a great way to learn game development. In this course, we have looked at the basics of creating a 2D platformer using the Godot engine. From what we have shown you, you now have the knowledge required to expand upon this project or create your very own game.

- The first step in creating a 2D platformer is to create the player character. This character is able to move around with the arrow keys and jump with the spacebar.
- The next step is to create enemies for the player to interact with. These enemies can be either stationary (spikes) or mobile (enemies that move from one point to another). Both types of enemies will call the game over function in the player's script when they come into contact with the player.
- Coins are a great way to add an extra layer of interactivity to your game. To create coins, we used a sine wave to make them bob up and down. When the player collects a coin, it increases their score and updates the UI label that displays their current score.
- The end flag is used to string different levels together. When the player reaches the end flag, they should be taken to the next level.

About Zenva

Zenva is an online learning academy with over 1 million students. They offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are versatile, allowing you to learn in whatever way you want, whether it is by viewing the online video tutorials, reading the included lesson summaries, or following along with the instructor with included course files.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Spikes.gd

Found in the Project root folder

This code is for a 2D game where the player should avoid hitting spikes. If the player's object comes into contact with the spikes (i.e., enters the same 'Area2D'), the game will end.

```
extends Area2D

# if the player hits the spikes, game over!
func _on_body_entered(body):
  if body.is_in_group("Player"):
    body.game_over()
```

Coin.gd

Found in the Project root folder

This code controls a coin in the game that bobs up and down over time. When the player's object intersects with the coin, it is removed from the scene and the player is awarded one point.

```
extends Area2D

var bob_height : float = 5.0
var bob_speed : float = 5.0

@onready var start_y : float = global_position.y
var t : float = 0.0

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
  # increase 't' over time.
  t += delta

  # create a sin wave that bobs up and down.
  var d = (sin(t * bob_speed) + 1) / 2

  # apply that to the coin's Y position.
  global_position.y = start_y + (d * bob_height)

# when the player hits the coin, add score and destroy the coin.
func _on_body_entered(body):
  if body.is_in_group("Player"):
    body.add_score(1)
    queue_free()
```

Player.gd

Found in the Project root folder

This code defines a 2D character with specific movement capabilities (left, right and jump controlled by player's input), gravity effects, and a scoring system. The game ends, i.e., current scene gets reloaded, if the character falls below a certain level. Each time a coin is collected, the player's score increases.

```
extends CharacterBody2D

var move_speed : float = 100.0
var jump_force : float = 200.0
var gravity : float = 500.0

var score : int = 0
@onready var score_text : Label = get_node("CanvasLayer/ScoreText")

func _physics_process(delta):
  # gravity.
  if not is_on_floor():
    velocity.y += gravity * delta

  velocity.x = 0

  # move left.
  if Input.is_key_pressed(KEY_LEFT):
    velocity.x -= move_speed
  # move right.
  if Input.is_key_pressed(KEY_RIGHT):
    velocity.x += move_speed

  if Input.is_key_pressed(KEY_SPACE) and is_on_floor():
    velocity.y = -jump_force

  move_and_slide()

  # game over if we fall below the level.
  if global_position.y > 100:
    game_over()

# restarts the current scene.
func game_over ():
  get_tree().reload_current_scene()

# called when we collect a coin.
func add_score (amount):
  score += amount
  score_text.text = str("Score: ", score)
```


Enemy.gd

Found in the Project root folder

This code controls a 2D area within a game that moves towards a specified target position at a set speed, reversing its direction when it reaches the target. If this area intersects with the player, it ends the game.

```
extends Area2D

@export var move_speed : float = 30.0
@export var move_dir : Vector2

var start_pos : Vector2
var target_pos : Vector2

# Called when the node enters the scene tree for the first time.
func _ready():
  start_pos = global_position
  target_pos = start_pos + move_dir

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
  # move towards the 'target_pos'.
  global_position = global_position.move_toward(target_pos, move_speed * delta)

  # when we arrive at 'target_pos' switch it around.
  if global_position == target_pos:
    if global_position == start_pos:
      target_pos = start_pos + move_dir
    else:
      target_pos = start_pos

# when the enemy hits the player, game over.
func _on_body_entered(body):
  if body.is_in_group("Player"):
    body.game_over()
```

EndFlag.gd

Found in the Project root folder

This code represents a game level flag. When the player touches the flag, the game transitions to the next level, loading the new scene file specified in the ‘next_scene’ variable.

```
extends Area2D

@export_file("*.tscn") var next_scene

# when the player hits the flag, load the next level.
func _on_body_entered(body):
  if body.is_in_group("Player"):
```



```
    get_tree().change_scene_to_file(next_scene)
```

