Welcome to our beginner-friendly course on creating a 2D platformer game using the Godot engine. Your instructor for this course is Daniel Buckley. Together, we will build a complete 2D platformer game from scratch, incorporating essential game mechanics and polishing elements to make the game engaging and exciting.

Course Overview

Throughout this course, you will learn how to:

- Set up and control a player character using keyboard inputs.
- Create enemies with basic AI movements.
- Implement a health system for the player.
- Design a coin collection system to increase the player's score.
- Manage persistent data to transfer information across different scenes.
- Set up end flags to teleport the player to the next level.

Game Mechanics

The game will feature the following mechanics:

- **Player Movement:** The player will be able to move using the arrow keys and jump using the up arrow or spacebar.
- **Enemy AI:** Enemies will move back and forth between two positions, and the player must avoid them to prevent taking damage.
- **Health System:** The player will have a limited number of hearts, representing their health.
- **Coin Collection:** Coins will be scattered throughout the level, and collecting them will increase the player's score.
- **Persistent Data:** We will manage data transfer across scenes, such as the player's score.
- **End Flags:** These will teleport the player to the next level upon contact.

Prerequisites

To follow along with this course, you should have a basic understanding of the Godot engine and GDScript. While we won't cover fundamental programming concepts like variables, functions, and operators, we will go through the code line by line, making it accessible even for those with limited coding experience.

About Zenva

Zenva is an online learning academy with over a million students. We offer a wide range of courses suitable for beginners and those looking to enhance their skills. You can choose to follow along with video lessons, download course files, and access lesson summaries to reinforce your learning.

Getting Started

Now that you have an overview of what to expect, let's dive into the first lesson and begin creating our 2D platformer game.

Lesson Structure

1. **Introduction:** Overview of the course and what you will learn.
2. **Setting Up the Project:** Creating a new project in Godot and setting up the initial scene.
3. **Player Movement:** Implementing player controls and movement.
4. **Enemy AI:** Creating enemies with basic movement patterns.
5. **Health System:** Adding a health system for the player.

6. **Coin Collection:** Designing a system for collecting coins and increasing the score.
7. **Persistent Data:** Managing data transfer across scenes.
8. **End Flags:** Setting up end flags to teleport the player to the next level.
9. **Polishing:** Adding sound effects and visual enhancements to make the game more engaging.
10. **Conclusion:** Reviewing what you've learned and next steps.

We look forward to seeing the amazing games you will create with the knowledge gained from this course. Happy learning!

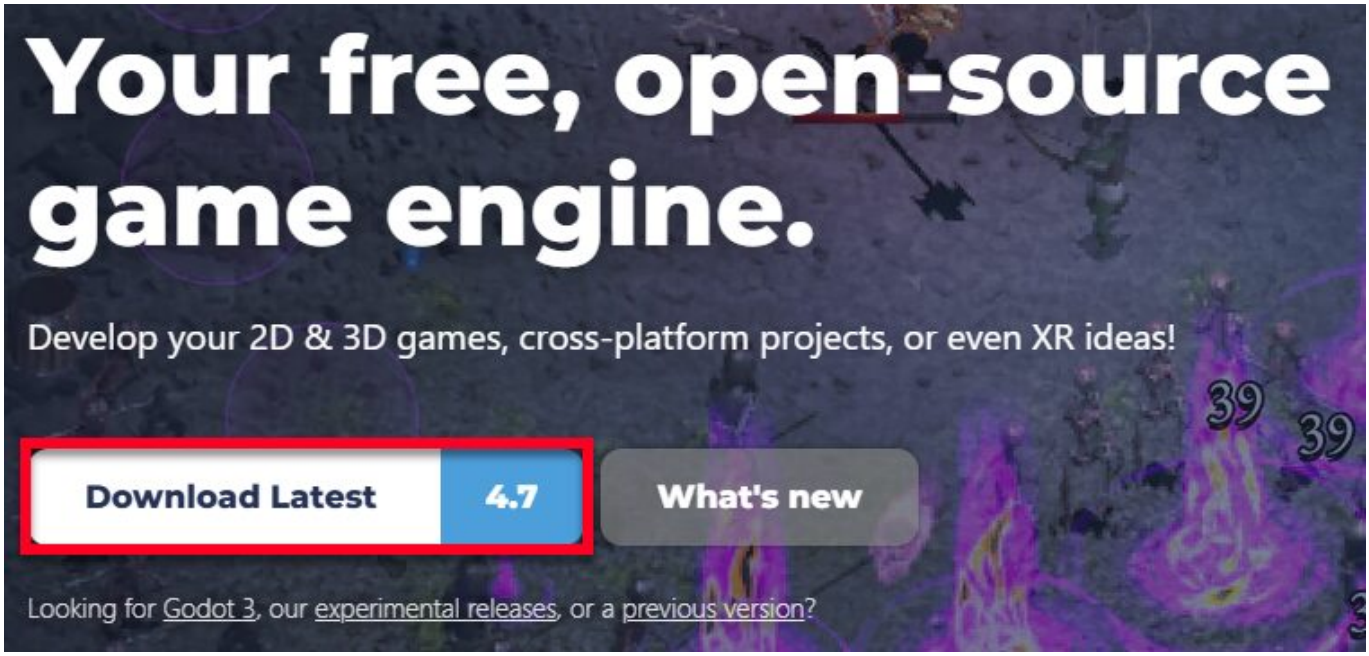
Course Updated to Godot 4.7

We've updated the project files to Godot version 4.7 for this course – the latest stable release.

Please make sure NOT to use newer versions of the software than what we recommend, as the course material provided might not work as expected.

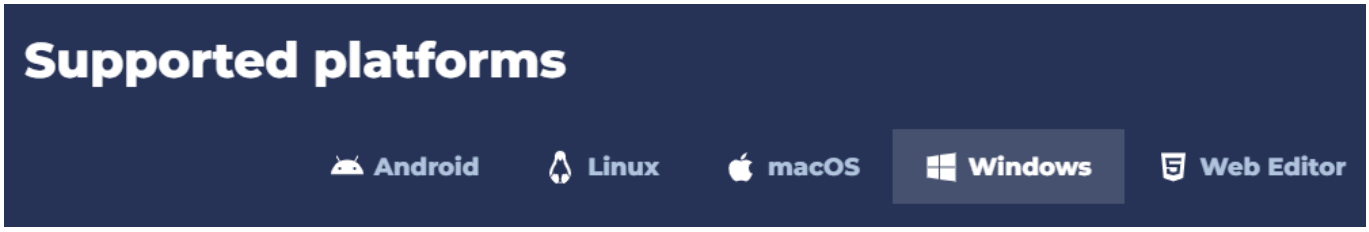
How to Install Version 4.7

You can download the most recent version of Godot by heading to the Godot website (https://godotengine.org/) and clicking the "Download Latest" button on the front page. This will automatically take you to the download page for your operating system.



Once on the download page, just click the option for **Godot 4.7.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application launcher – no further installation steps are required!

If Godot fails to automatically select the correct operating system for you, you can scroll down to the “Supported platforms” section on the download page to manually select the download version you would like to install:



Welcome to this beginner-friendly guide on creating a 2D platformer game using Godot. Before we dive into the actual game creation process, it’s essential to understand what we will be building. This game will feature a player who collects coins and avoids enemies while navigating through various platforms. The primary goal is to reach the end flag of each level.

Player Controls and Movement

The player will be controlled using the keyboard, with the following main controls:

- Movement: Left and right
- Jumping

To enhance the movement experience, we will incorporate:

- **Acceleration:** When the player starts moving
- **Braking:** When the player releases the movement keys

These additions will create a more natural and smoother movement, improving the overall feel of the game compared to basic, jittery pixel-perfect movement.

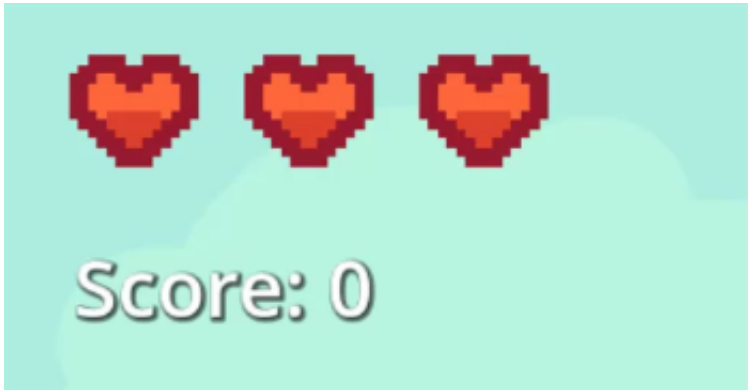
Enemies and Damage System

The game will include enemies that move back and forth between two positions. These parameters can be defined:

- Distance of movement
- Angle of movement

When an enemy collides with the player, the following damage system will be triggered:

1. Player’s health decreases by one
2. A visual UI with hearts displays the health reduction
3. The screen flashes red briefly
4. A screen shake effect is applied
5. A damage sound effect plays

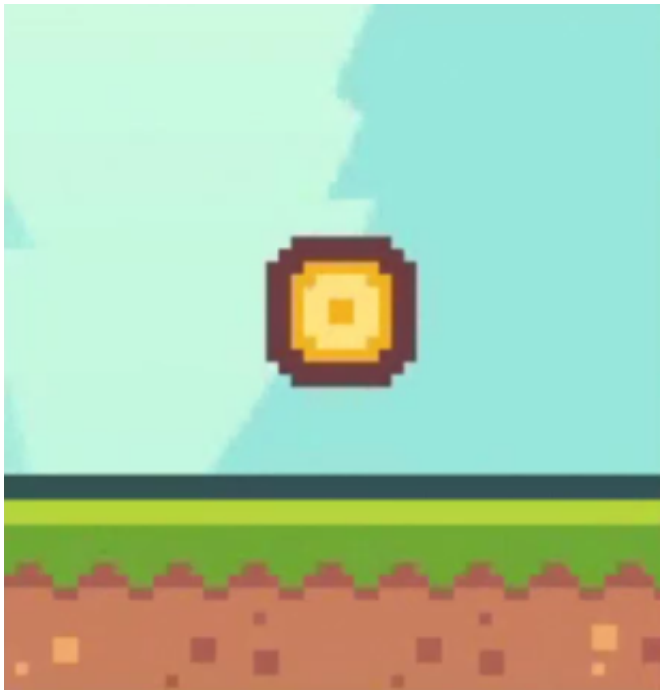


Coins and Collectibles

Coins will be placed throughout the levels as collectibles. When collected, the player’s score increases by one. Additionally, coins will have the following visual and auditory effects:

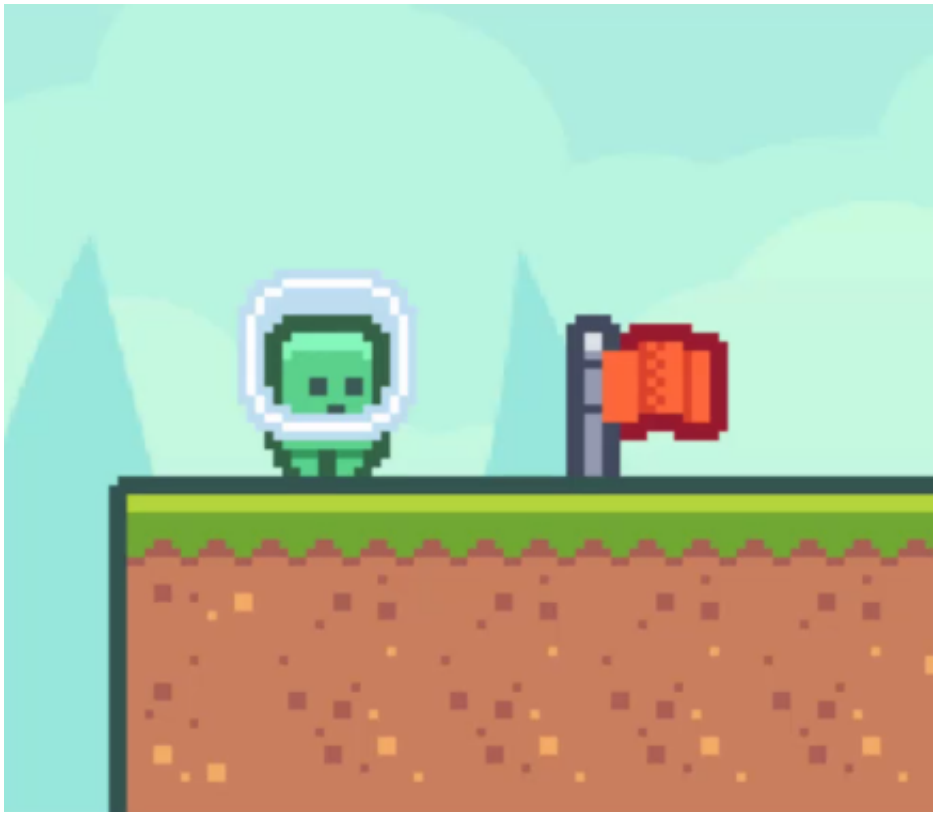
- **Rotating Effect:** The coin appears to spin, achieved by changing its width over time using a sine wave

- **Bobbing Effect:** The coin moves up and down to enhance its visibility
- **Sound Effect:** Plays when the coin is collected



End Flag and Level Transition

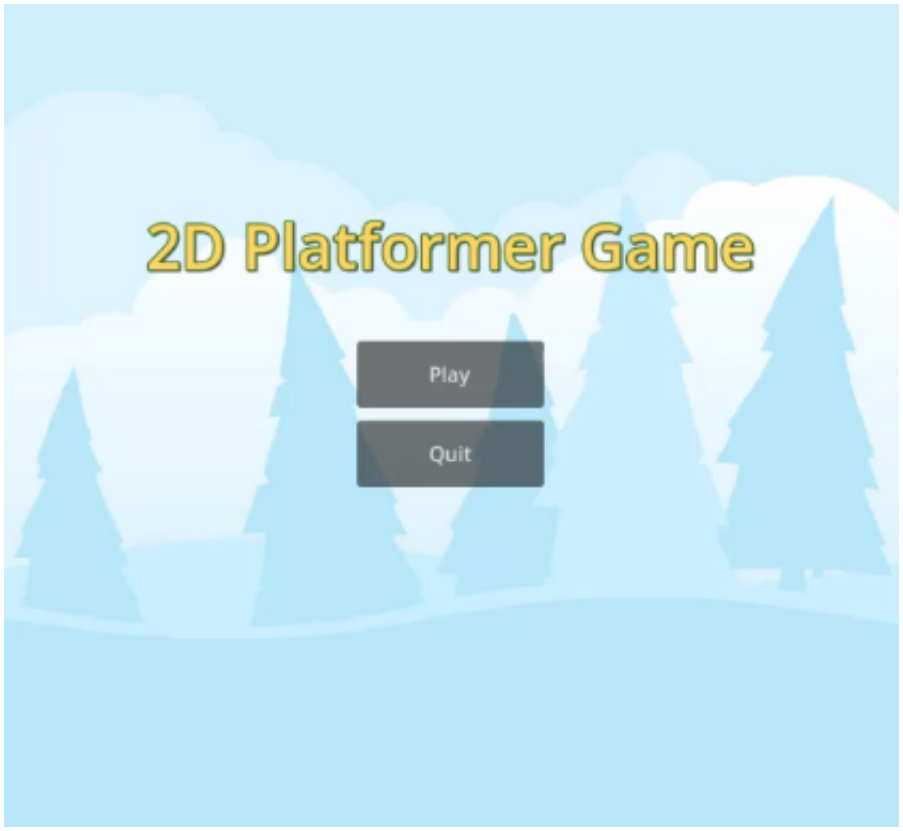
At the end of each level, an end flag will serve as a portal to the next level. When the player reaches the end flag, they will be teleported to the subsequent level (e.g., from Level 1 to Level 2).



Menu Scene

The game will feature a simple menu scene with the following buttons:

- Play Button: Starts the game
- Quit Button: Exits the game



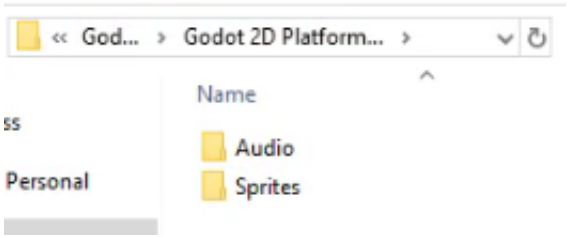
Now that we have a clear understanding of the game's components and mechanics, let's proceed with the first lesson and begin creating our 2D platformer game in Godot.

In this lesson, we will walk through the initial steps of creating a new project, organizing your file system, and setting up your first scene. To begin, open Godot and create a new project. The first task is to set up your file system, which involves importing assets and creating the necessary folders for your project.

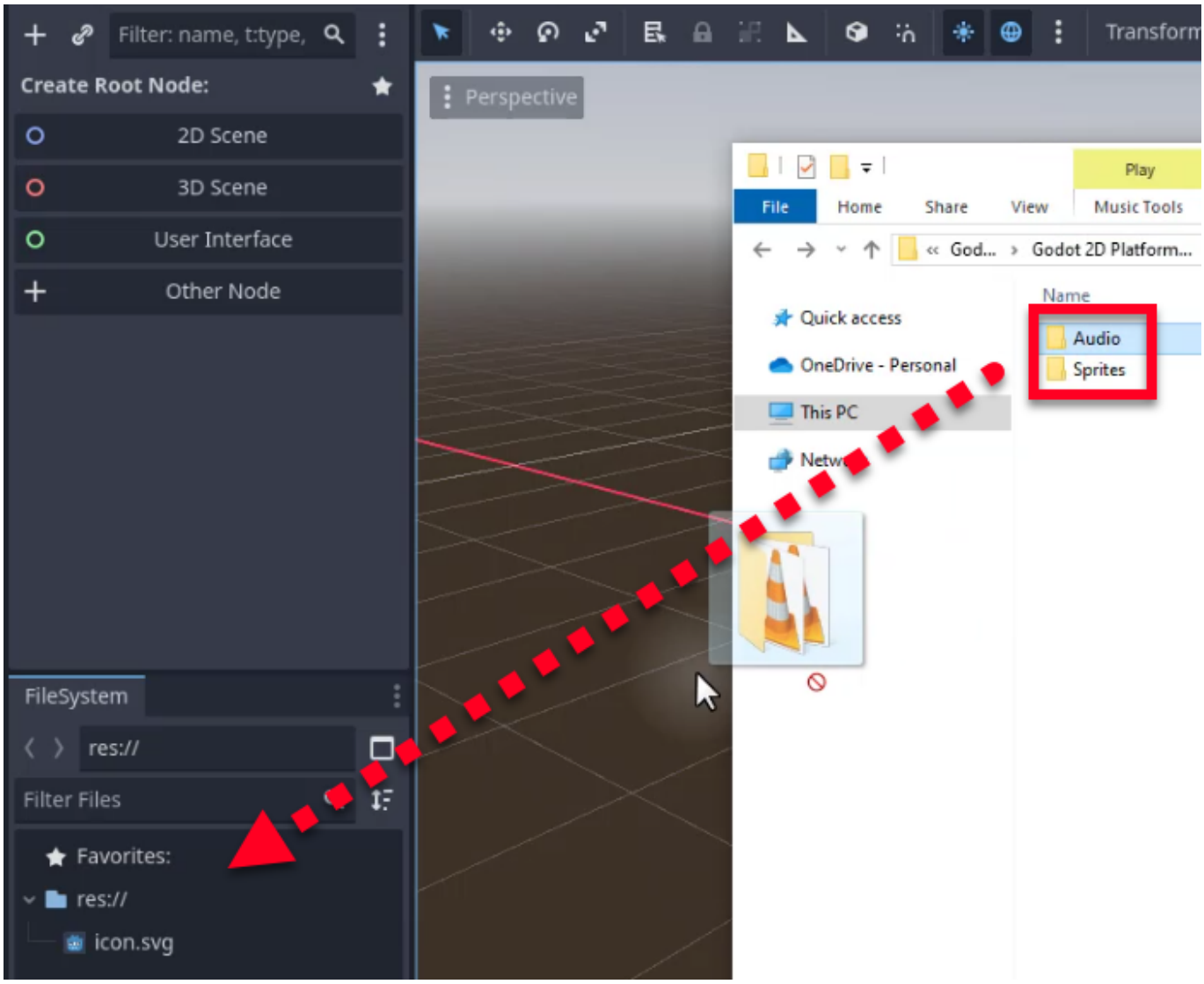
Importing Assets

Let’s begin by importing our assets. First, navigate to the course files tab of this course and download the provided zip file. Then, extract the zip file to your computer.

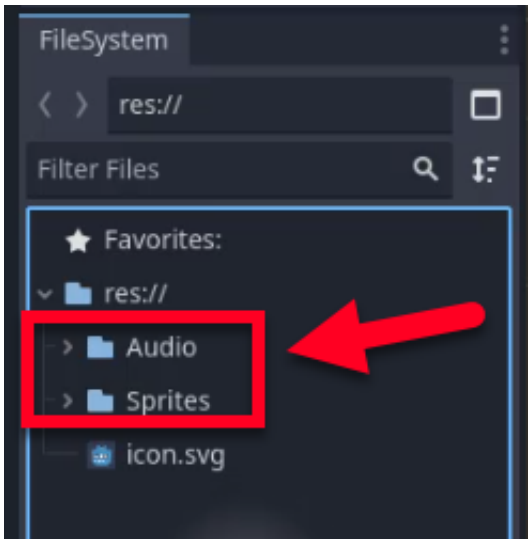
You should see two folders: Audio and Sprites. The Audio folder contains sound effects for picking up coins and taking damage. The Sprites folder includes subfolders for backgrounds, characters, objects, and tiles.



To import these assets into your Godot project, click and drag the Audio and Sprites folders into the FileSystem dock in Godot.



Ensure that the folders are correctly imported and contain the respective files.

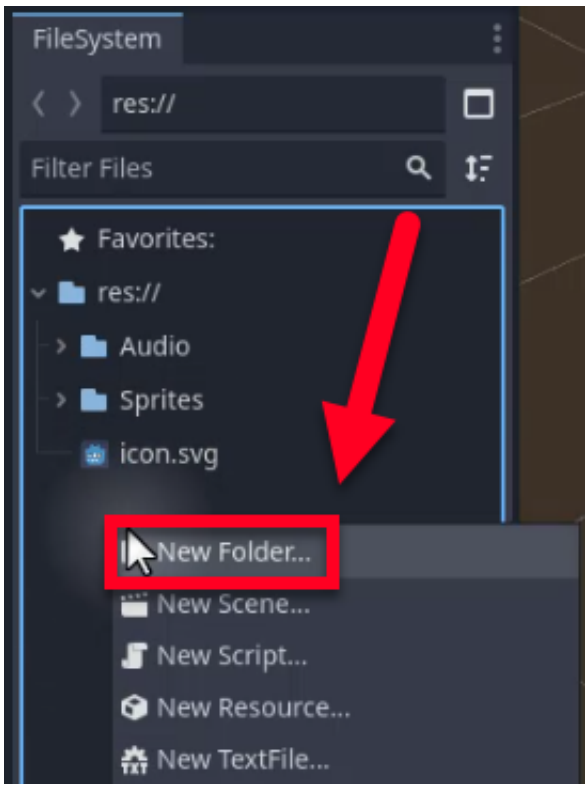


Creating Necessary Folders

Next, create two additional folders for your project:

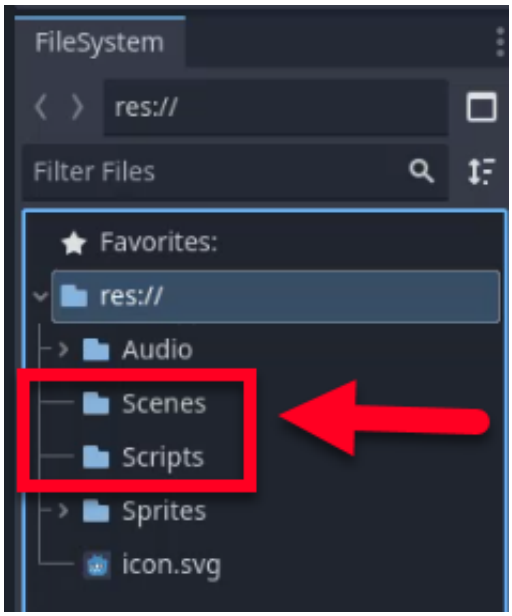
- Scripts: This folder will store all your code files.
- Scenes: This folder will store various scenes, such as the player scene, coin scene, enemy scene, levels, menu, and end flag.

To create these folders, right-click in the FileSystem dock and select New Folder.



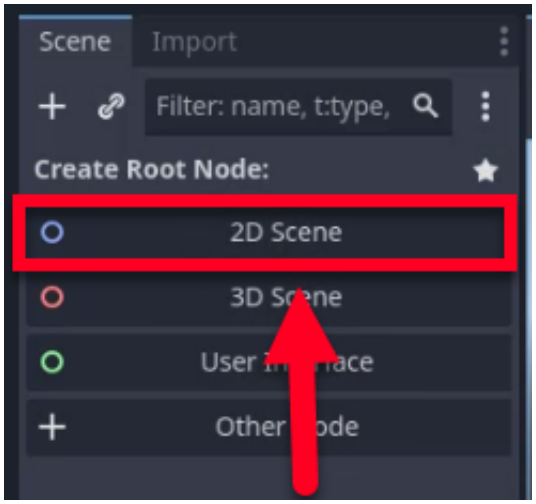
Name the folder and click **OK** to create it.

Per the above, create two folders. Name the first folderScripts, and the second Scenes.

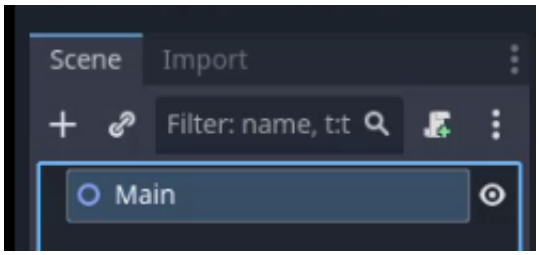


Setting Up Your First Scene

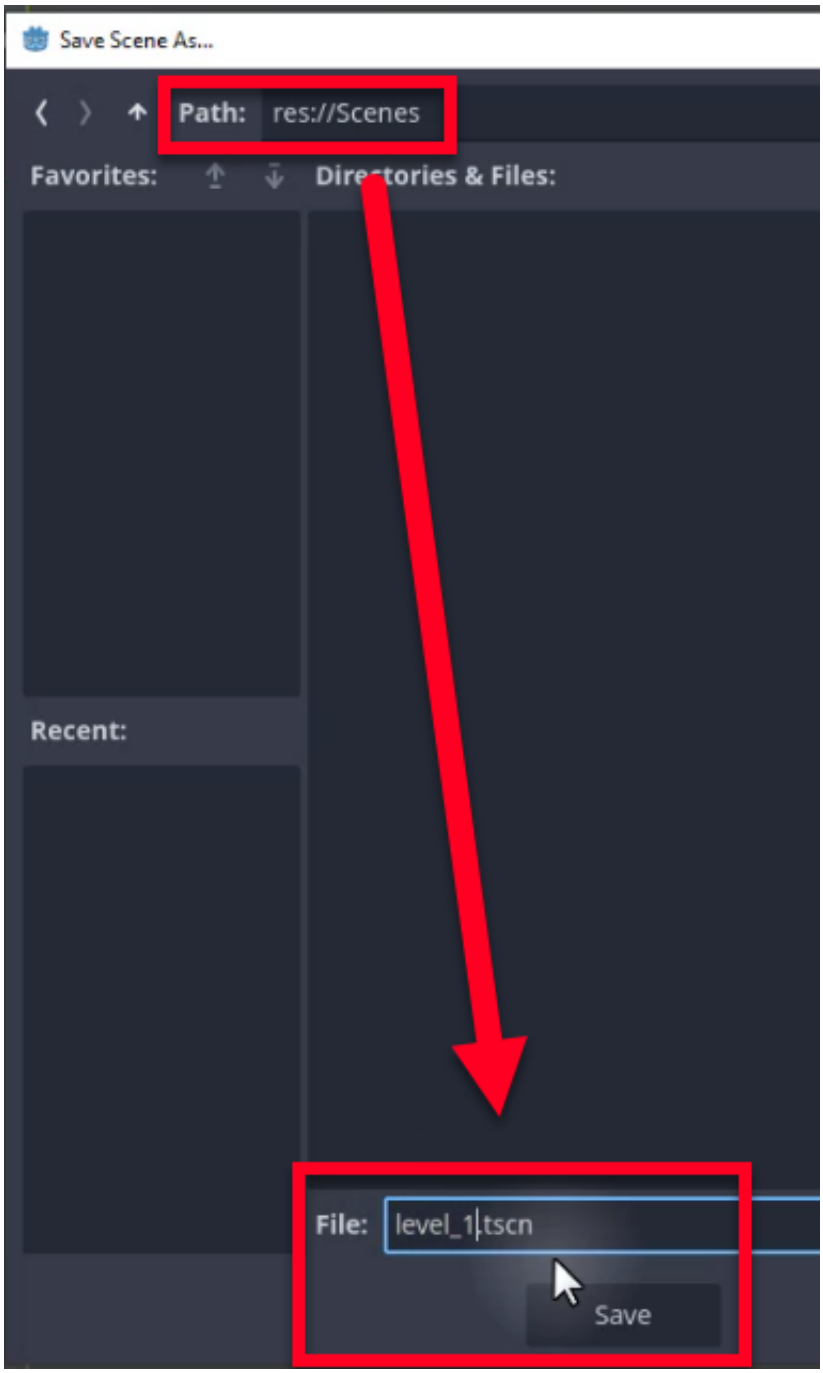
Now, let's create your first scene, which will be for Level 1. In the top left corner, open the Scene dock. Since this is a new project, choose 2D Scene as the root node for your scene.



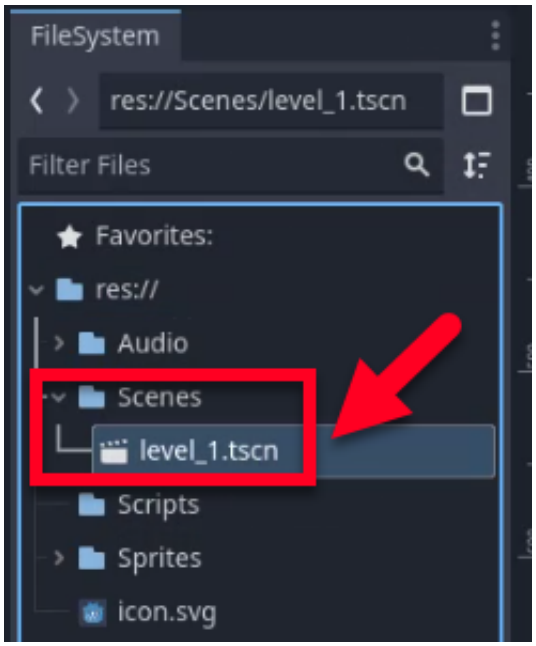
Name the root node Main for consistency.



Save the scene by pressing Ctrl + S or going to Scene > Save Scene. Navigate to the Scenes folder and save the scene as level_1.tscn.



If you need to reopen your scene at any point now, simply navigate to the Scenes folder in the FileSystem dock. Then, double-click on level_1.tscn to open it.



Next Steps

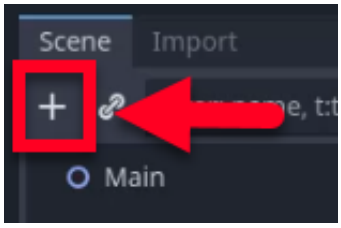
In the next lesson, we will set up the environment for Level 1. This includes adding a background and setting up the tile map level. We will then proceed to add the player, enemies, coins, end flag, and other necessary elements for your game.

Thank you for following along. Stay tuned for the next part of this series, where we will dive deeper into building your game in Godot.

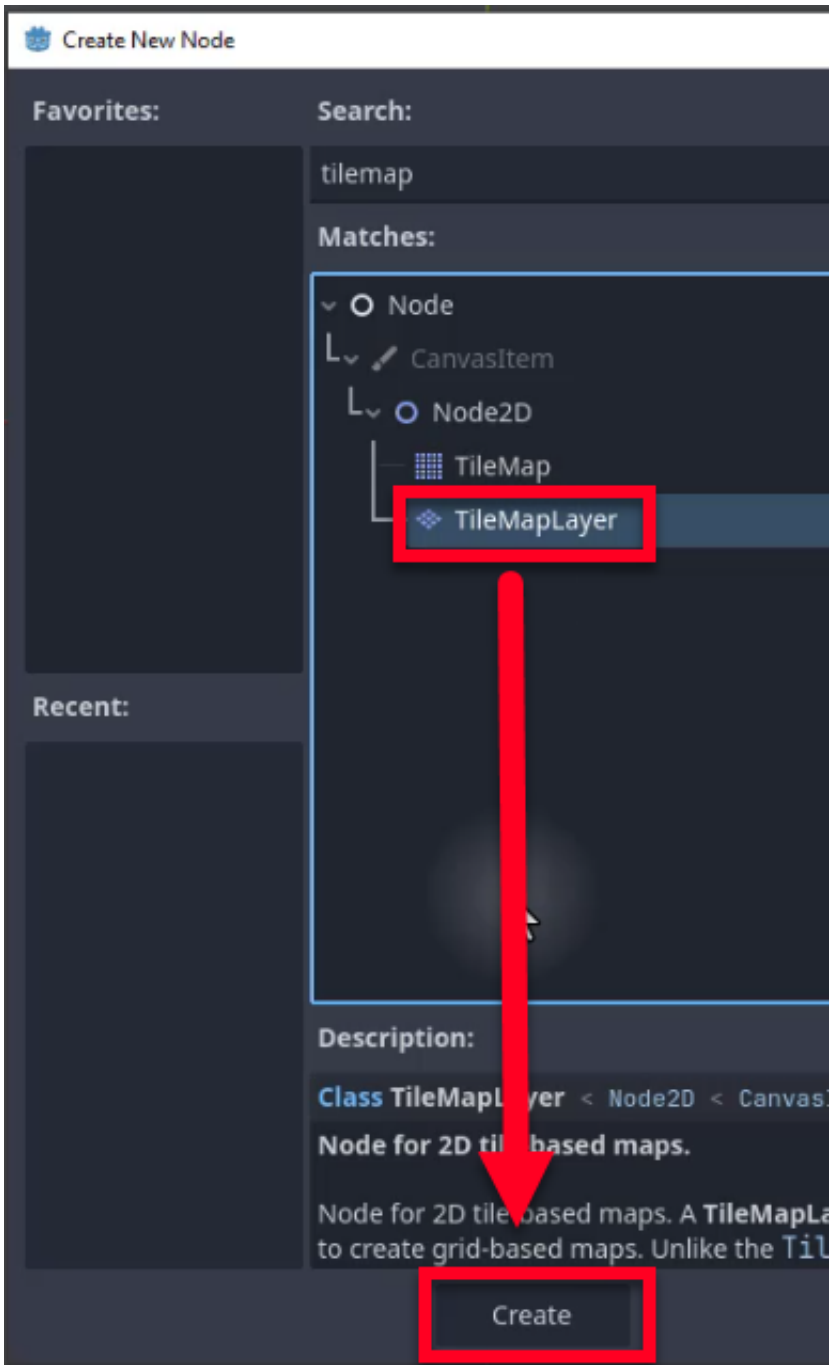
n this lesson, we will focus on creating a tile map and background for your game level. A tile map is essentially a grid where you can paint individual tiles such as grass, dirt, etc., allowing you to lay out your levels as you see fit.

Creating a Tile Map Layer

To start, we need to create a tile map layer. This node is where we will paint our tile map. Click on the plus icon at the top left corner of the Godot interface.

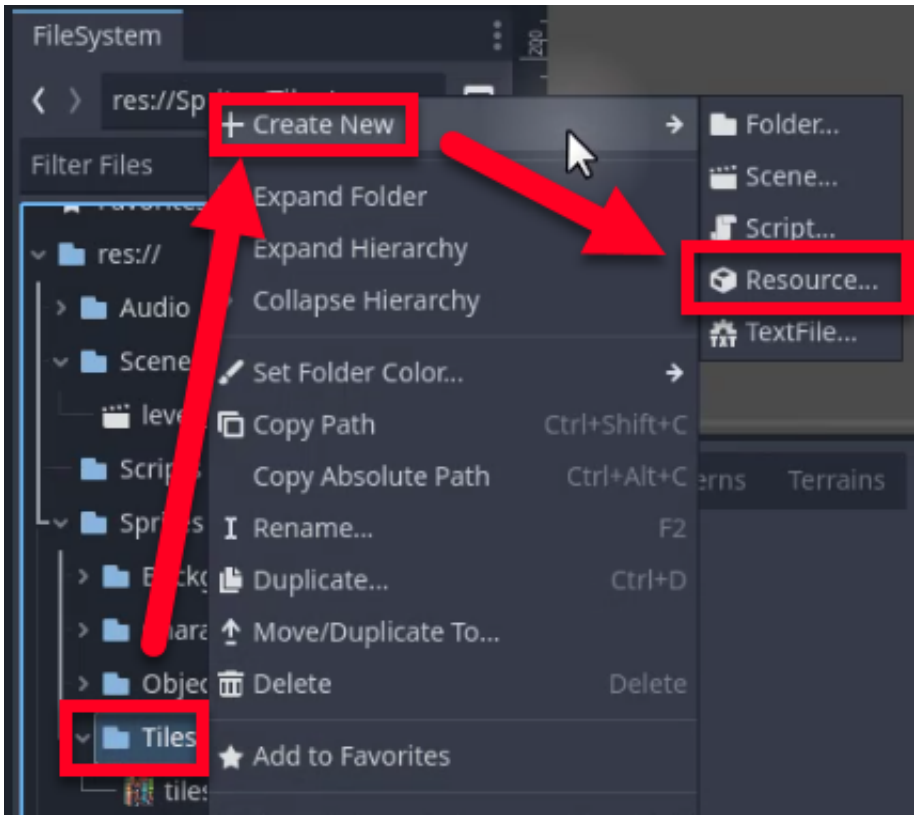


In the ‘Create New Node’ window, search for and select ‘**TileMapLayer**‘ and hit Create. Note that ‘**TileMap**‘ is a deprecated node used in older versions of Godot – so don’t use it!

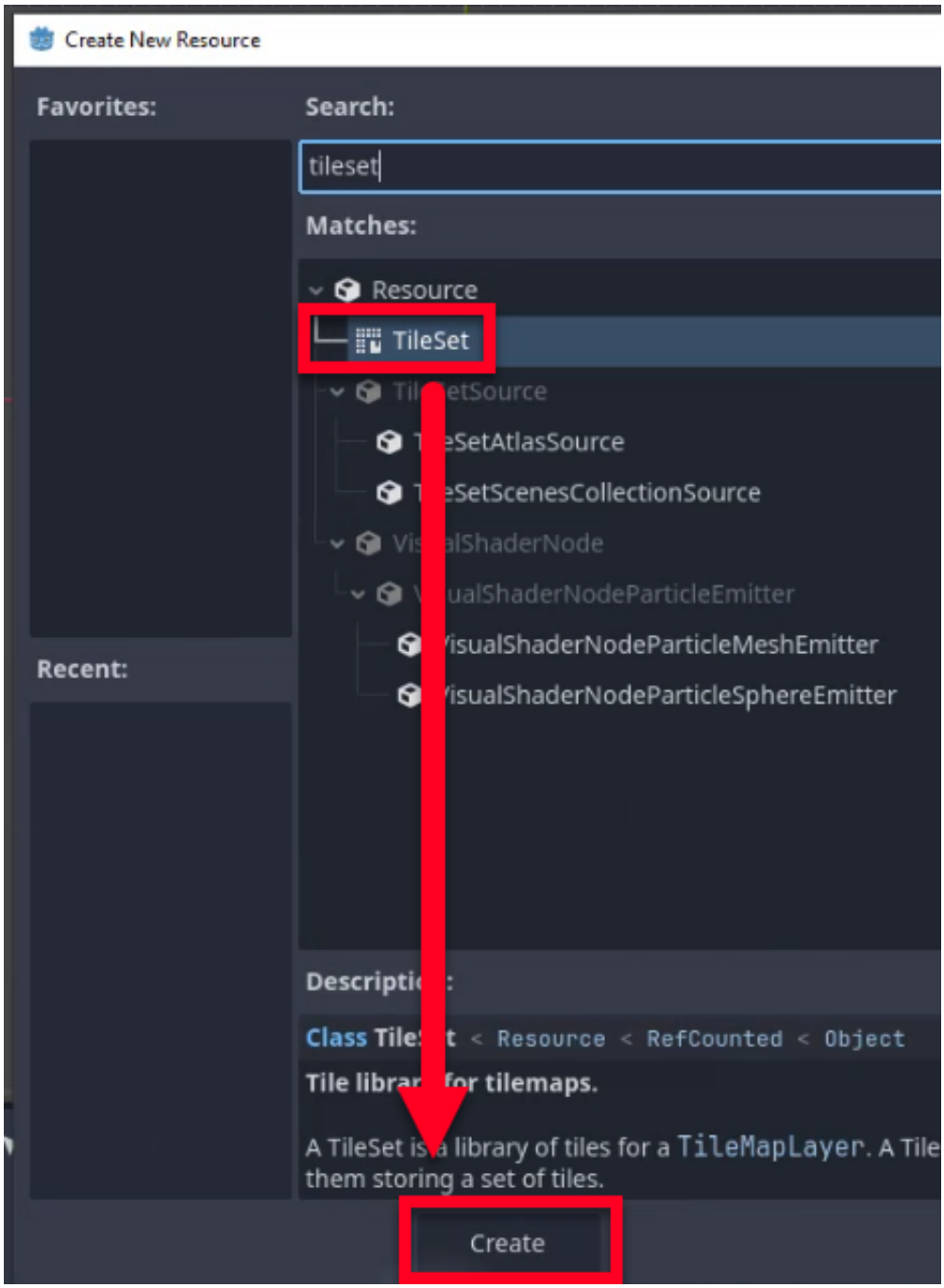


Setting Up the Tile Set

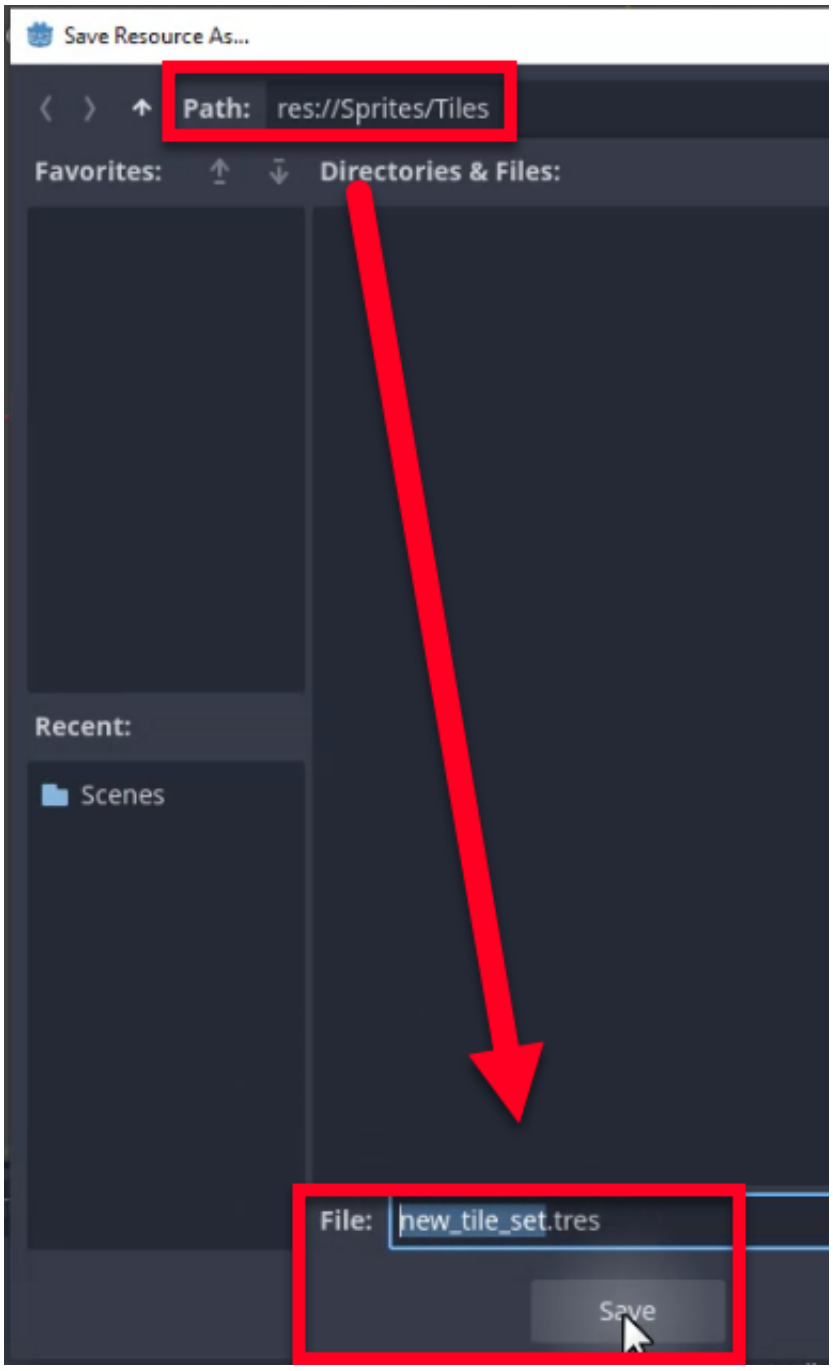
Next, we need to define a tile set, which specifies the tiles we can use. To begin, open the file system and navigate to the ‘Sprites’ folder, then to the ‘Tiles’ folder. Right-click on the ‘Tiles’ folder and select Create New > Resource.



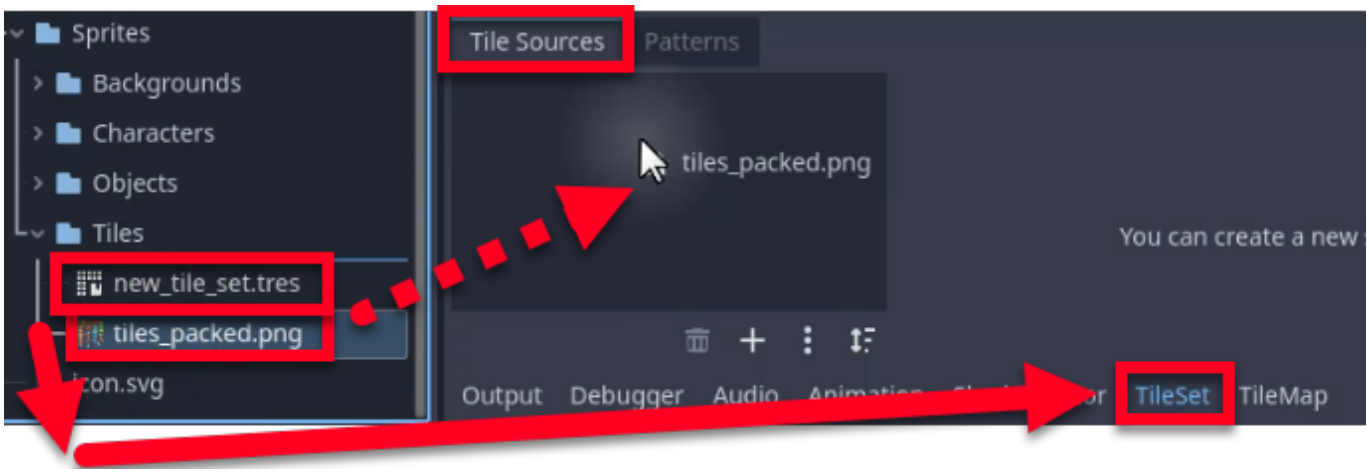
In the ‘Create New Resource’ window, search for and select the TileSet option.



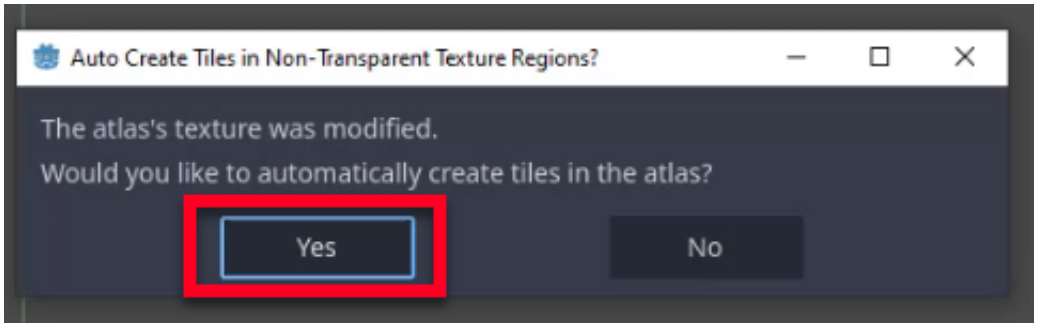
Name your tile set and save it.



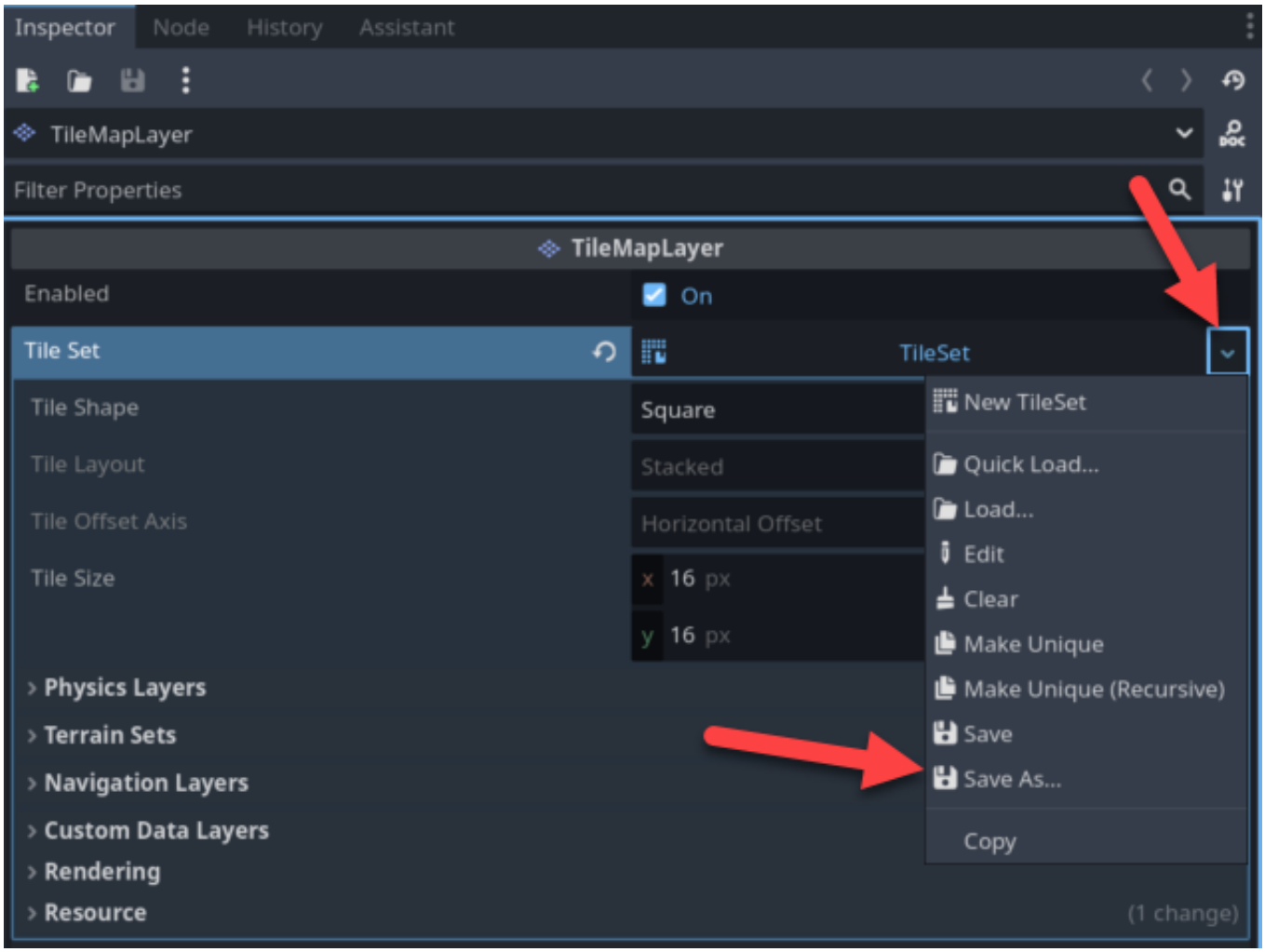
Double-click the tile set to open it in the bottom window. Click the ‘TileSet’ tab button to the left of the TileMap. Then, drag your PNG file (e.g., ’tiles_packed’ tile map) into the ‘Tile Sources’ section.



Confirm the creation of a new atlas for these sprites.

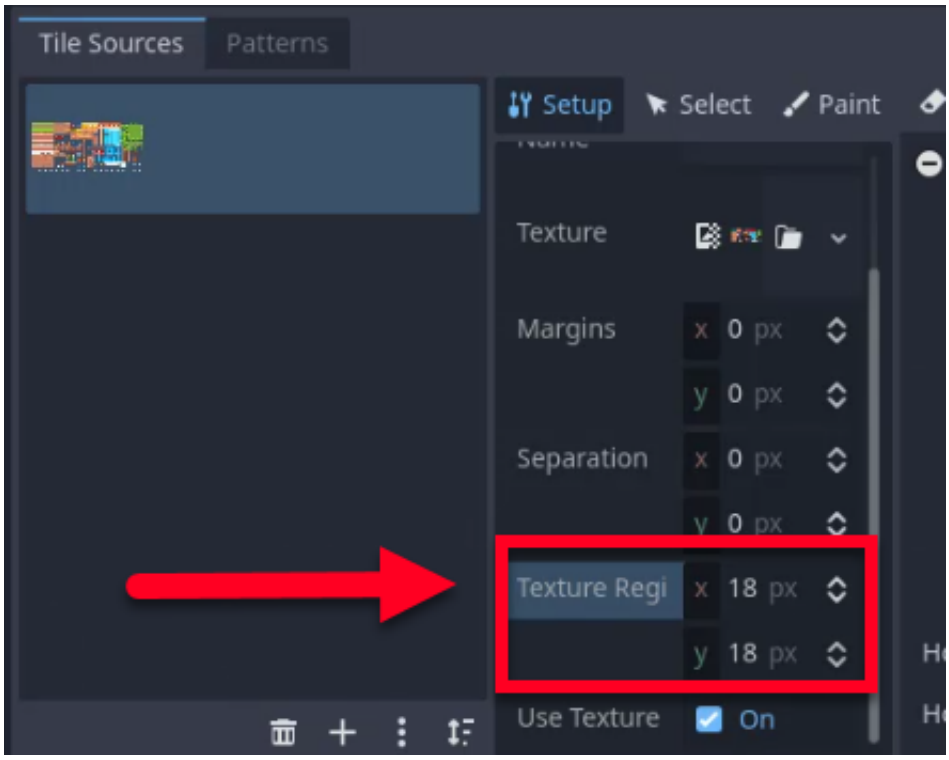


Note: The Tile Set can also be created directly in the Inspector, and it can be saved to disk by expanding the options and clicking on “Save” or “Save as”:

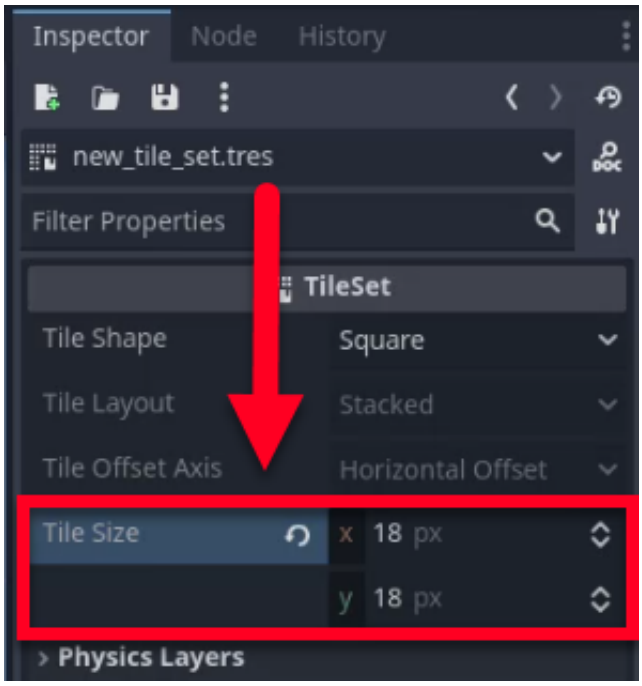


Adjusting Tile Dimensions

You might notice that the tiles are not the correct dimensions. To fix this, go to the ‘Texture Region Size’ in the tile set settings. From here, you can change the dimensions from 16×16 pixels to 18×18 pixels.

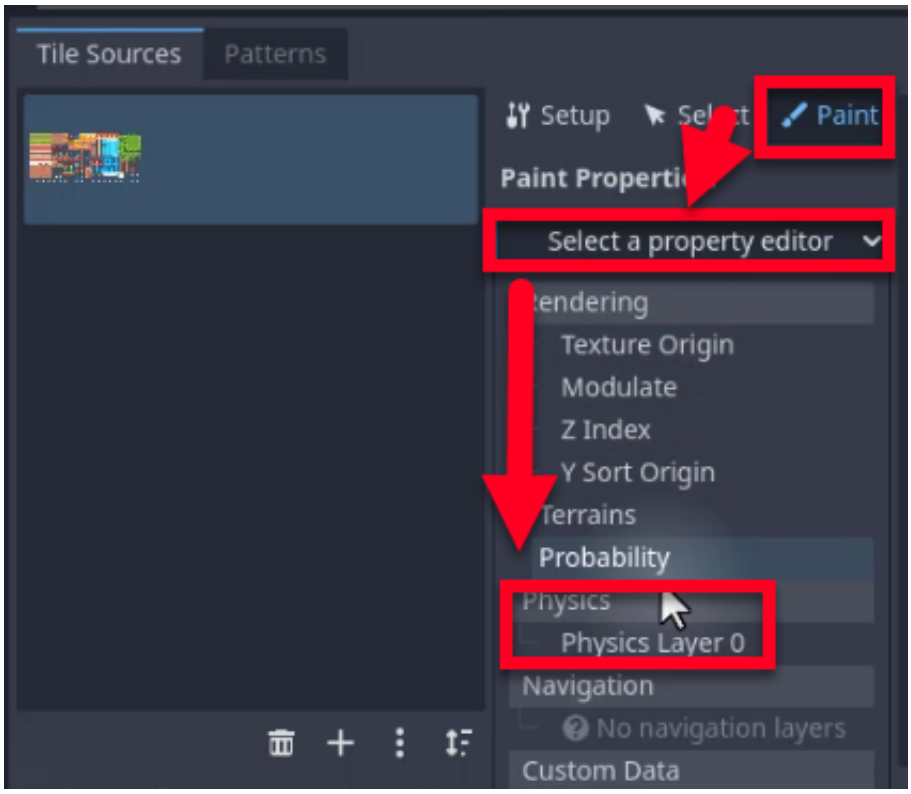


Ensure the tile set property in the inspector is also set to 18×18 pixels.



Adding Collision Properties

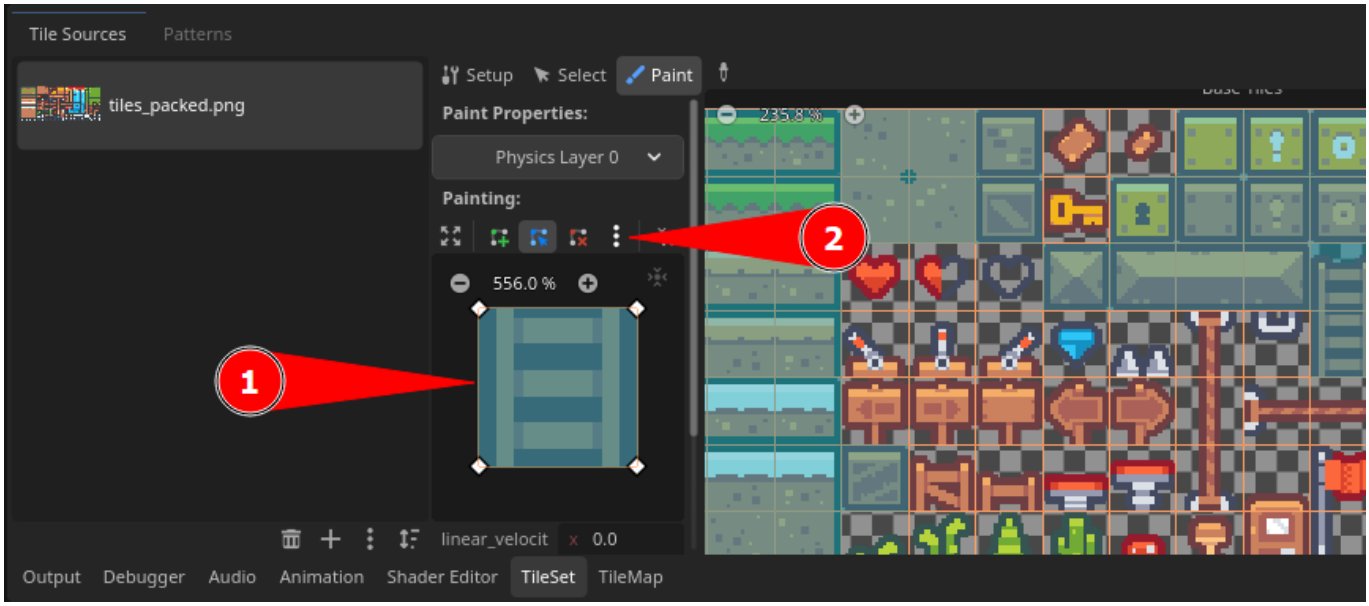
Next, we need to add collisions to our tiles. To do so, in the TileSet window, select Paint. Then, from the ‘property editor’ dropdown, select **Physics Layer 0**.



Now, we can use the paint tool to apply the physics layer to the tiles that need collision. When painting physics polygons onto the tiles, the polygon on the left-hand side of the window is what is applied to the clicked tiles (1 in the image below).

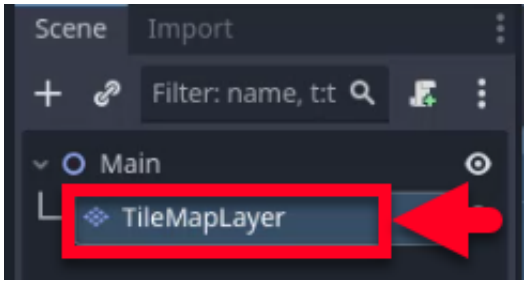
If you accidentally click on a tile and would like to erase it, you can click on the 3-dot menu (2 in the image below) and click on ‘Clear’ to clear the shape. Then you can click on tiles to apply the empty shape.

Once you are done, you can use the ‘Reset to default tile shape’ option in the same menu to continue applying shapes to other tiles.

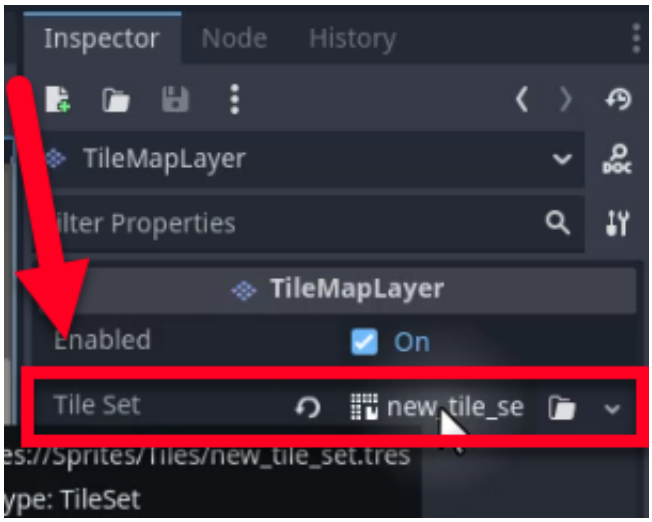


Painting Tiles in the Scene

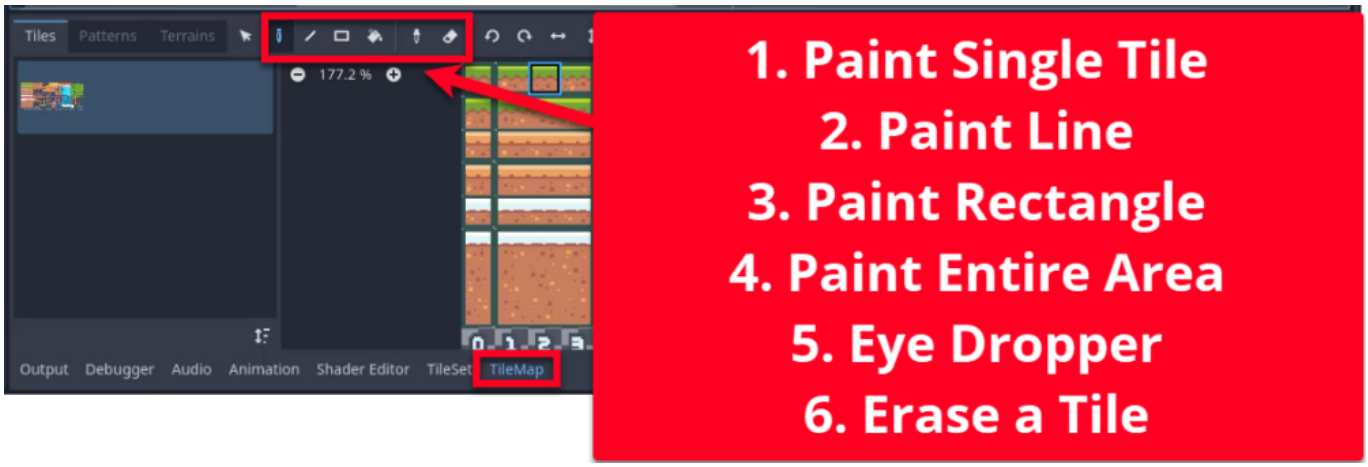
Now, with our tile set setup, let's paint the tiles in our scene. First, select the 'TileMapLayer' node.



In the Inspector, drag your tile set into the 'Tile Set' property.

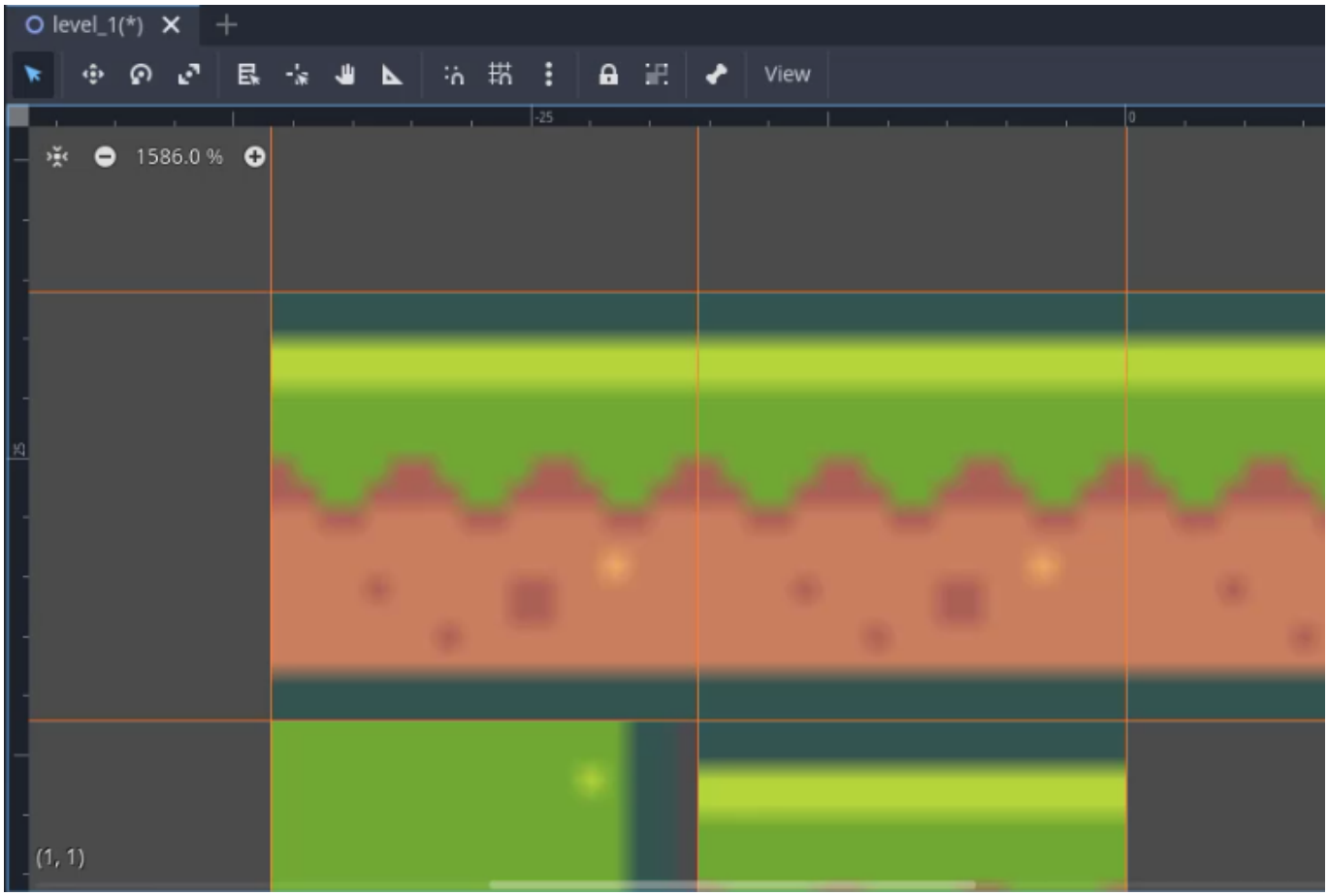


Use the 'TileMap' window at the bottom to select and paint tiles in your scene. Use the eraser tool (shortcut: E) to erase tiles if needed. You can also use the rect tool to paint large areas and the line tool to draw lines between points.

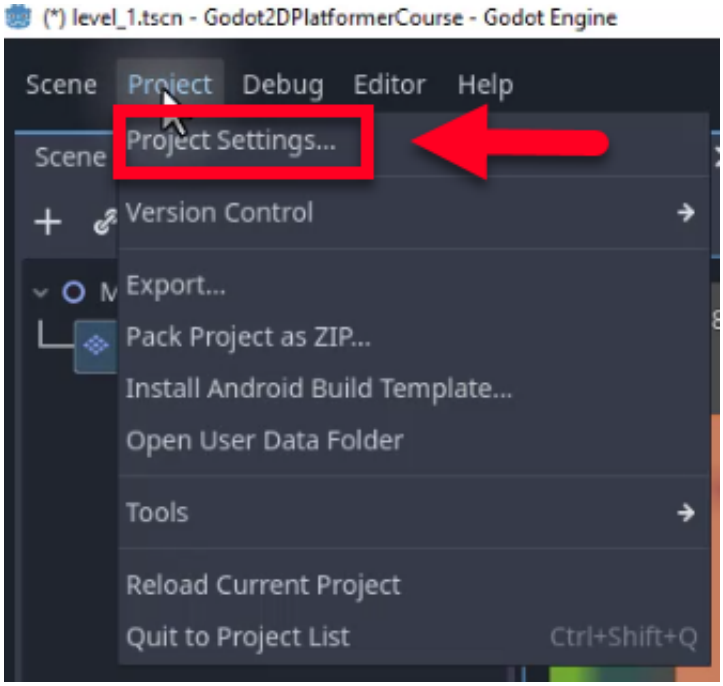


Fixing Blurry Sprites

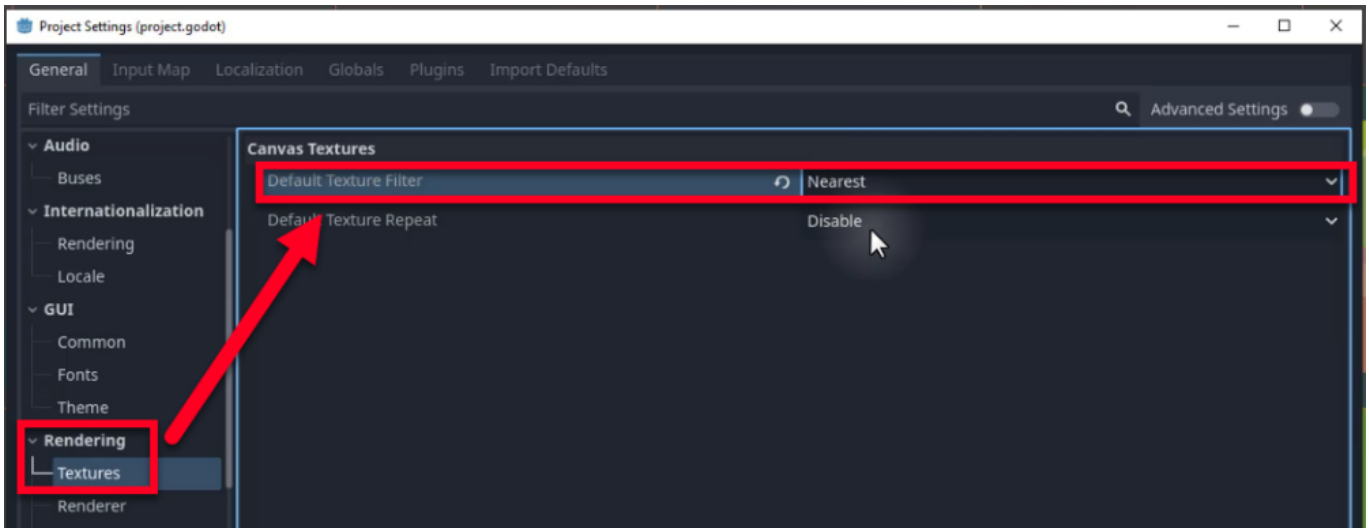
You may notice while painting that your sprites probably look blurry.



To fix this, you need to adjust the texture filter settings. Go to ‘Project’ > ‘Project Settings’.



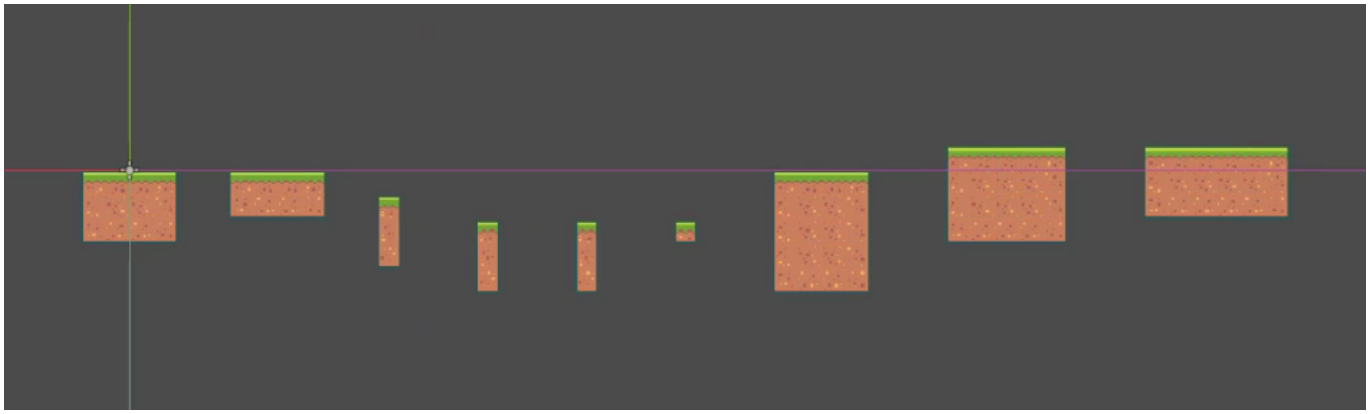
Navigate to ‘Rendering’ > ‘Textures’, and change the ‘Default Texture Filter’ from ‘Linear’ to ‘Nearest’.



Design Your Level

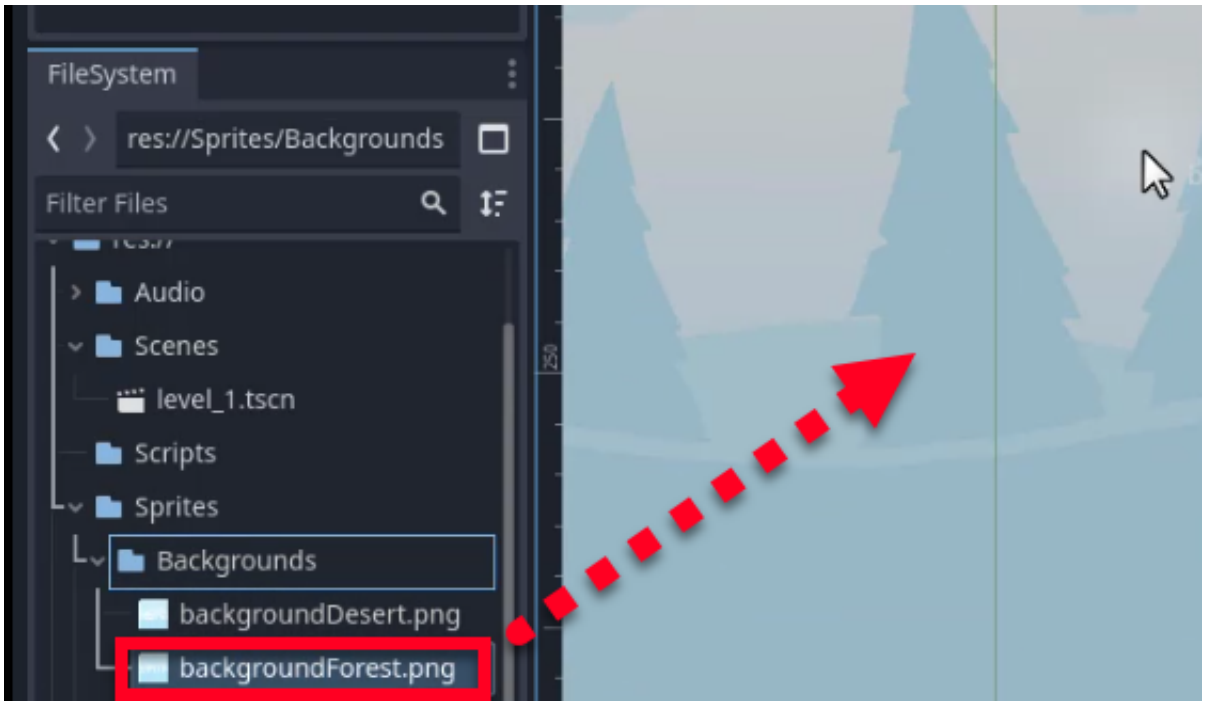
At this point, go forward and design your level however you wish. There are no right or wrong answers!

For example, our level looks like the image below:

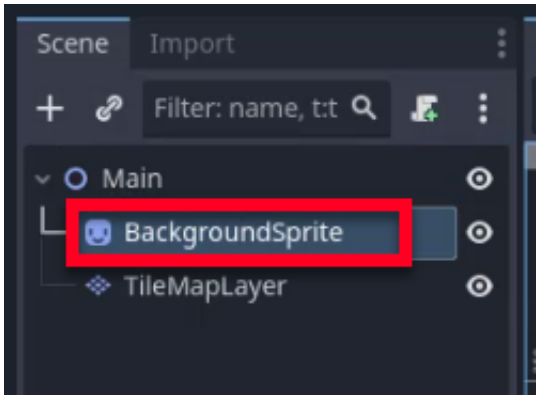


Setting Up the Background

Finally, let's add a background to our scene. Drag your background image (e.g., ' backgroundForest') into the scene.



For organization, rename the background node to ‘BackgroundSprite’.



Congratulations! You have now set up your tile map and background. In the next lesson, we will start creating our player. Stay tuned!

In this lesson, we will guide you through the process of setting up your player character, which will serve as the main interactive element in your game. By the end, you will have a player character that can move and jump within a platformer-style environment.

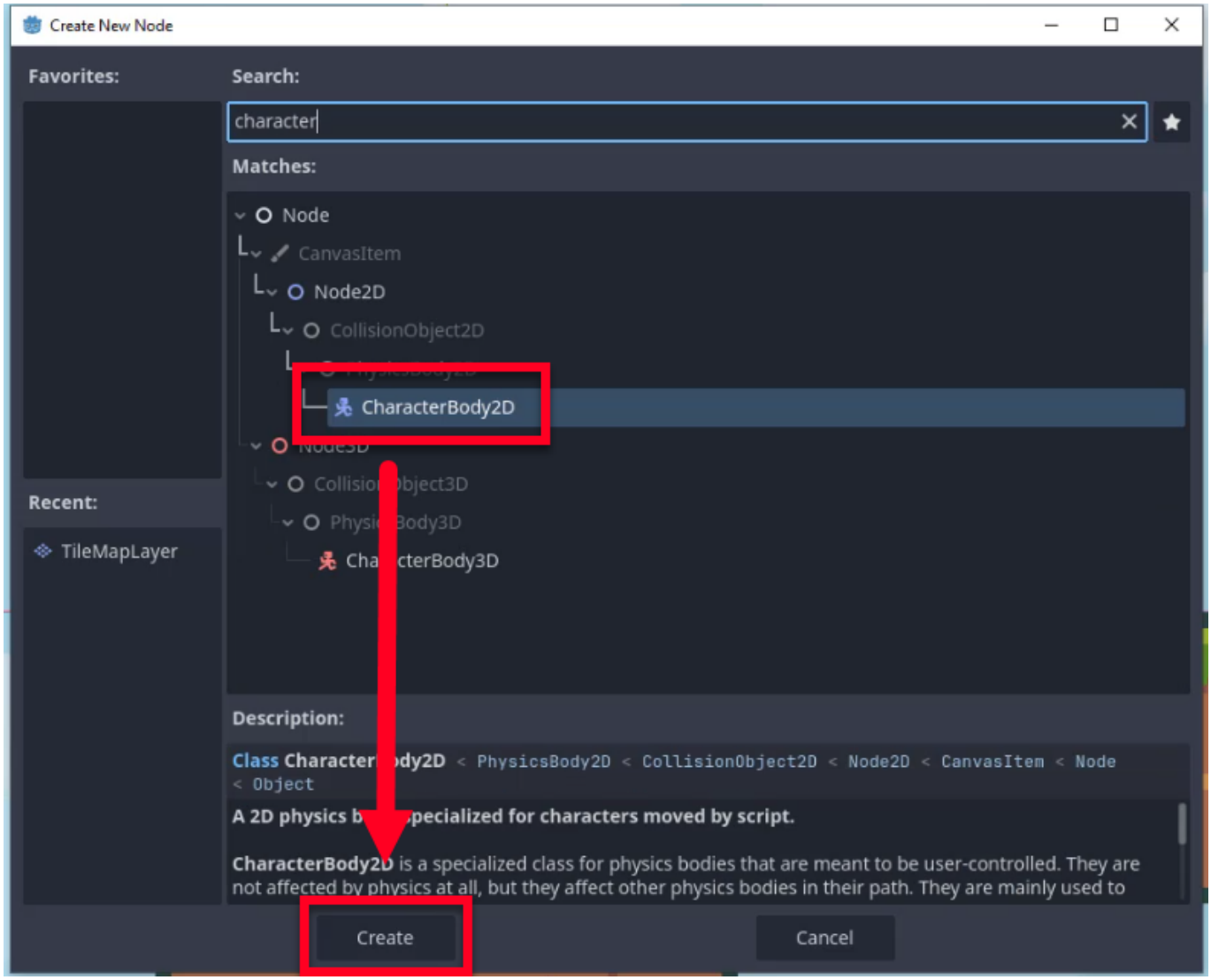
Understanding the Player Scene

The player scene is a crucial component of your game, as it defines the physical object that the player will control. This scene will include various nodes that determine the player's appearance, collision detection, and other properties.

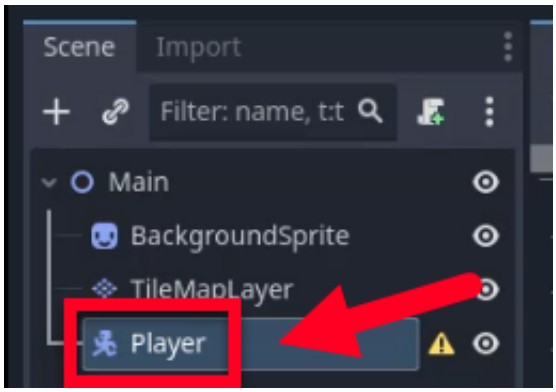
Creating the Player Scene

To begin, we need to determine the root node of our player. For platformer games and characters in general, Godot provides a specialized node called CharacterBody2D. This node is highly useful because it includes built-in features such as collision detection, ground detection, and velocity modification.

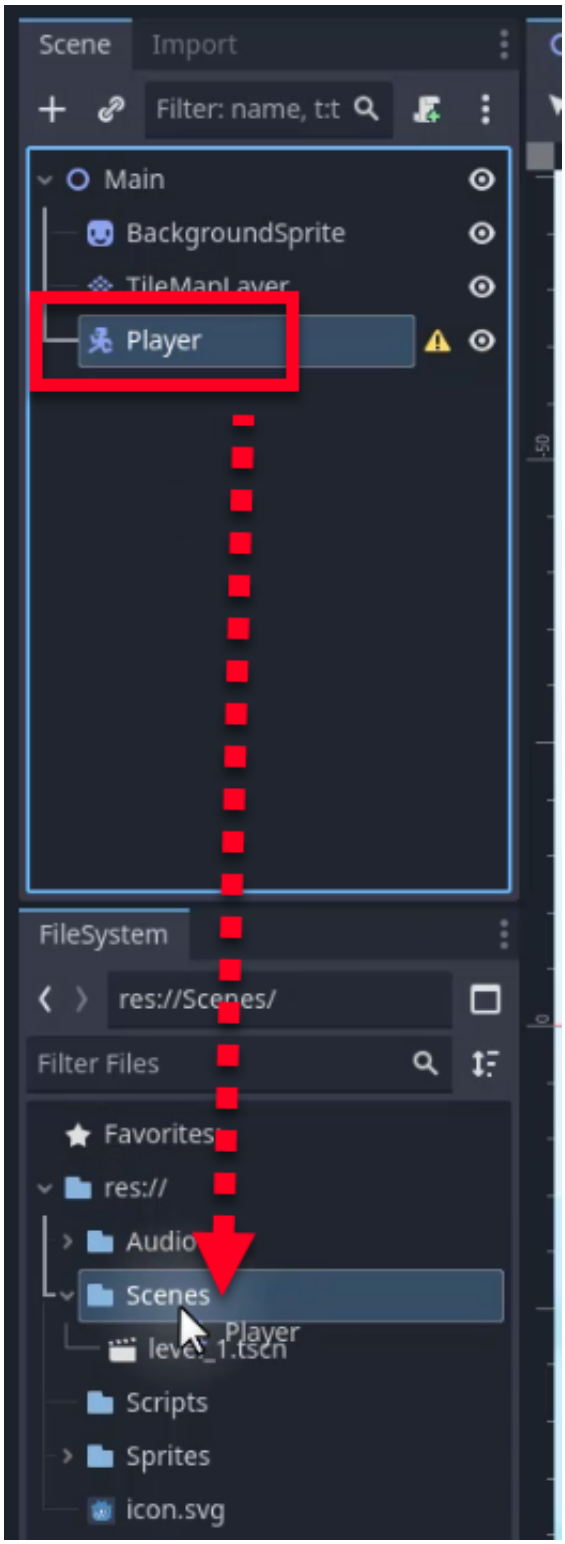
To begin, create a new CharacterBody2D node.

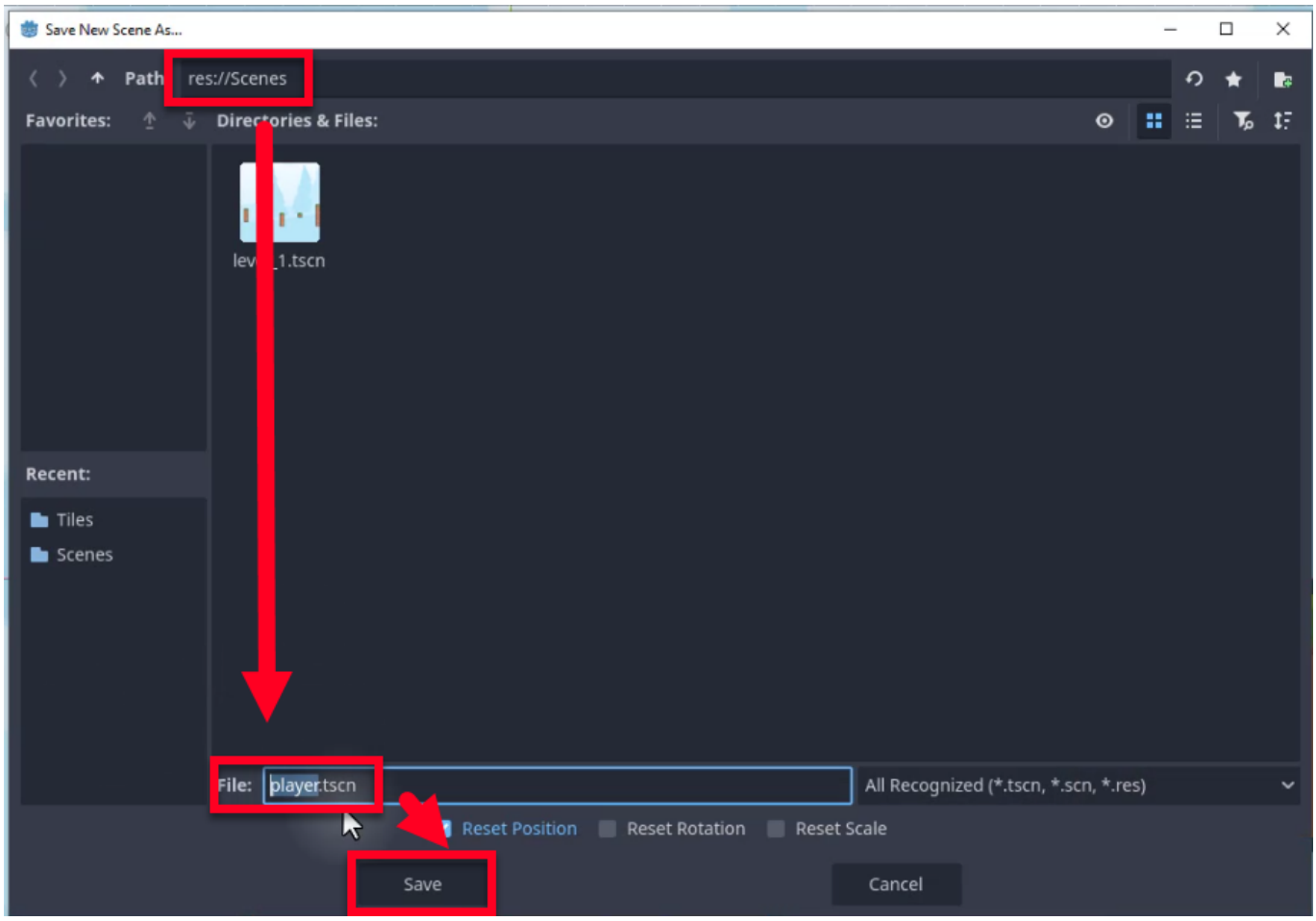


Rename the node to "Player".

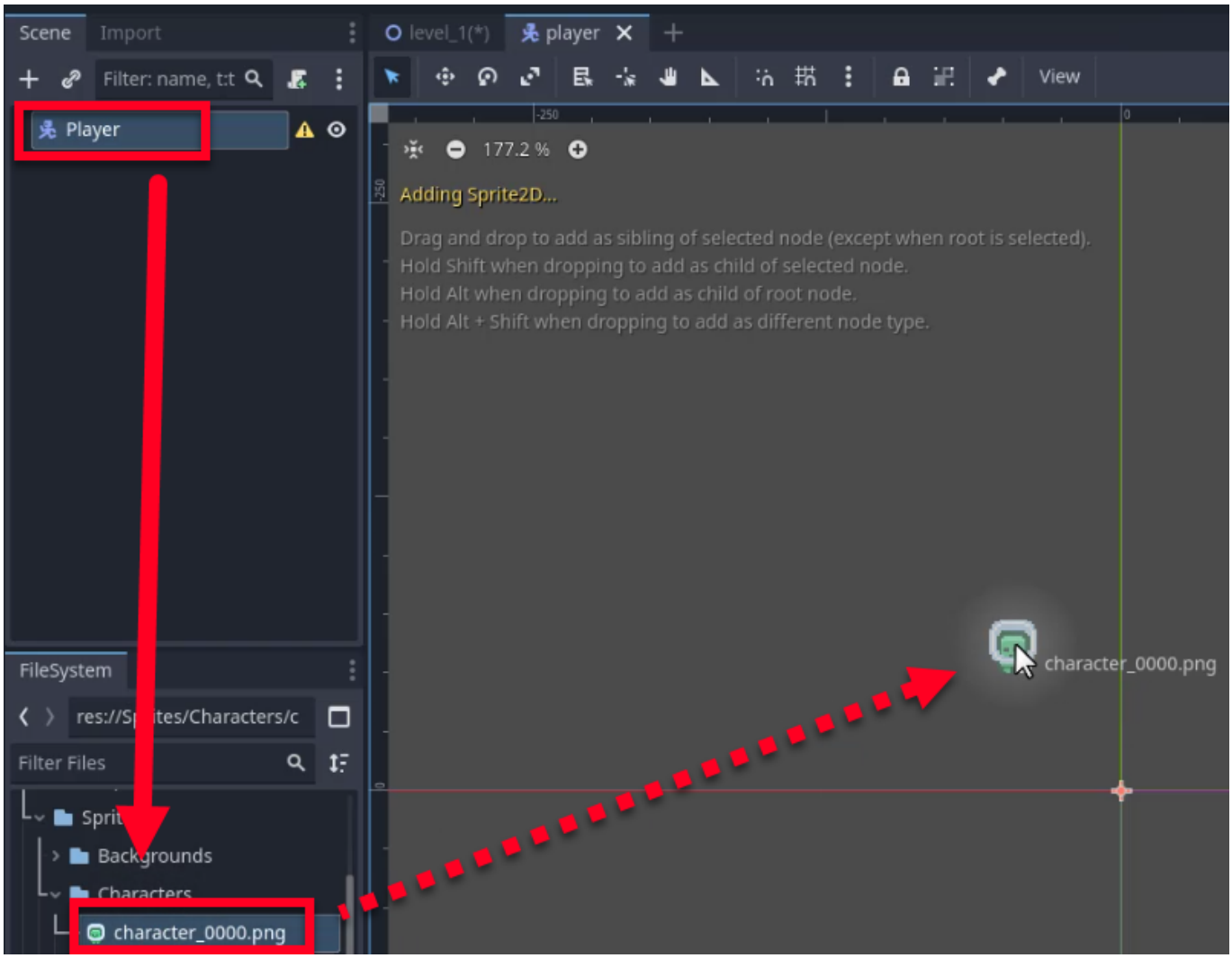


Save the scene by right-clicking on the node and selecting “Save Branch as Scene”, or by dragging the node to the scenes folder and saving it as Player.tscn.

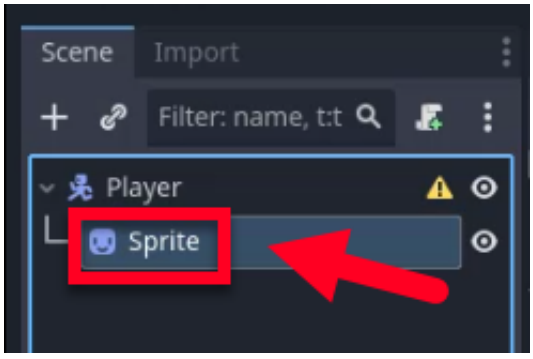




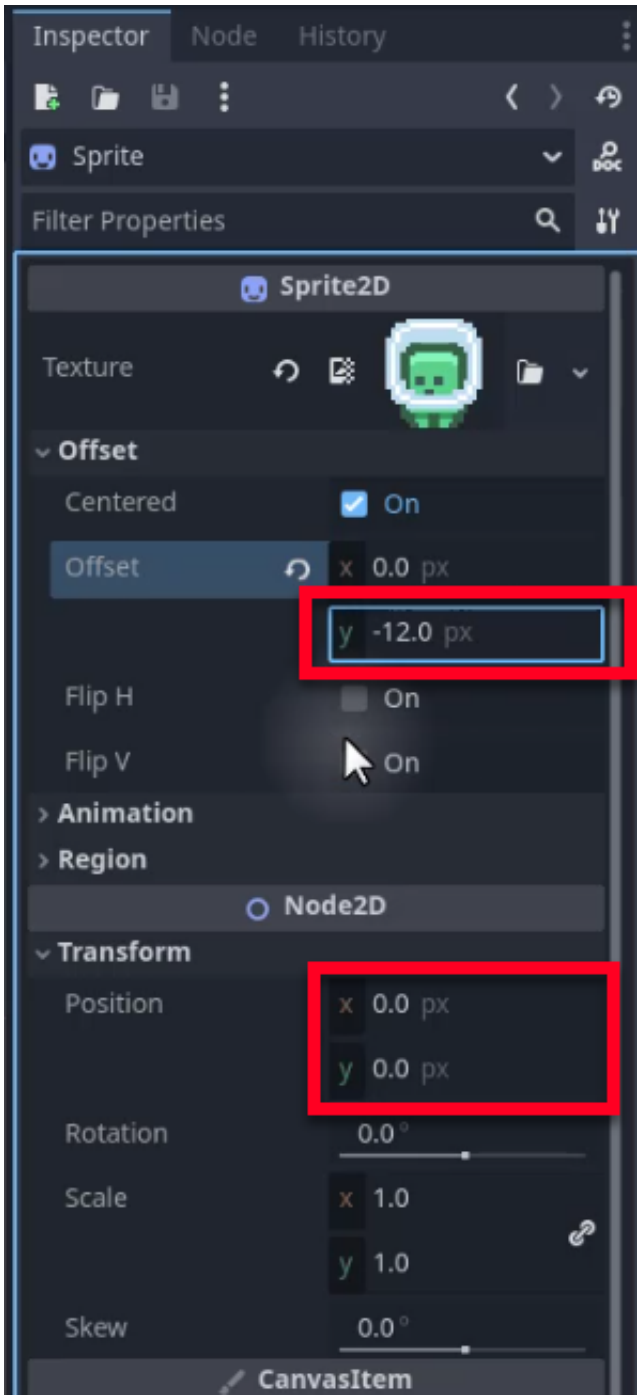
Add a sprite to the player node to visualize the character. You can do this by dragging a sprite from your sprites folder into the player node or into the editor window with the Player node selected.



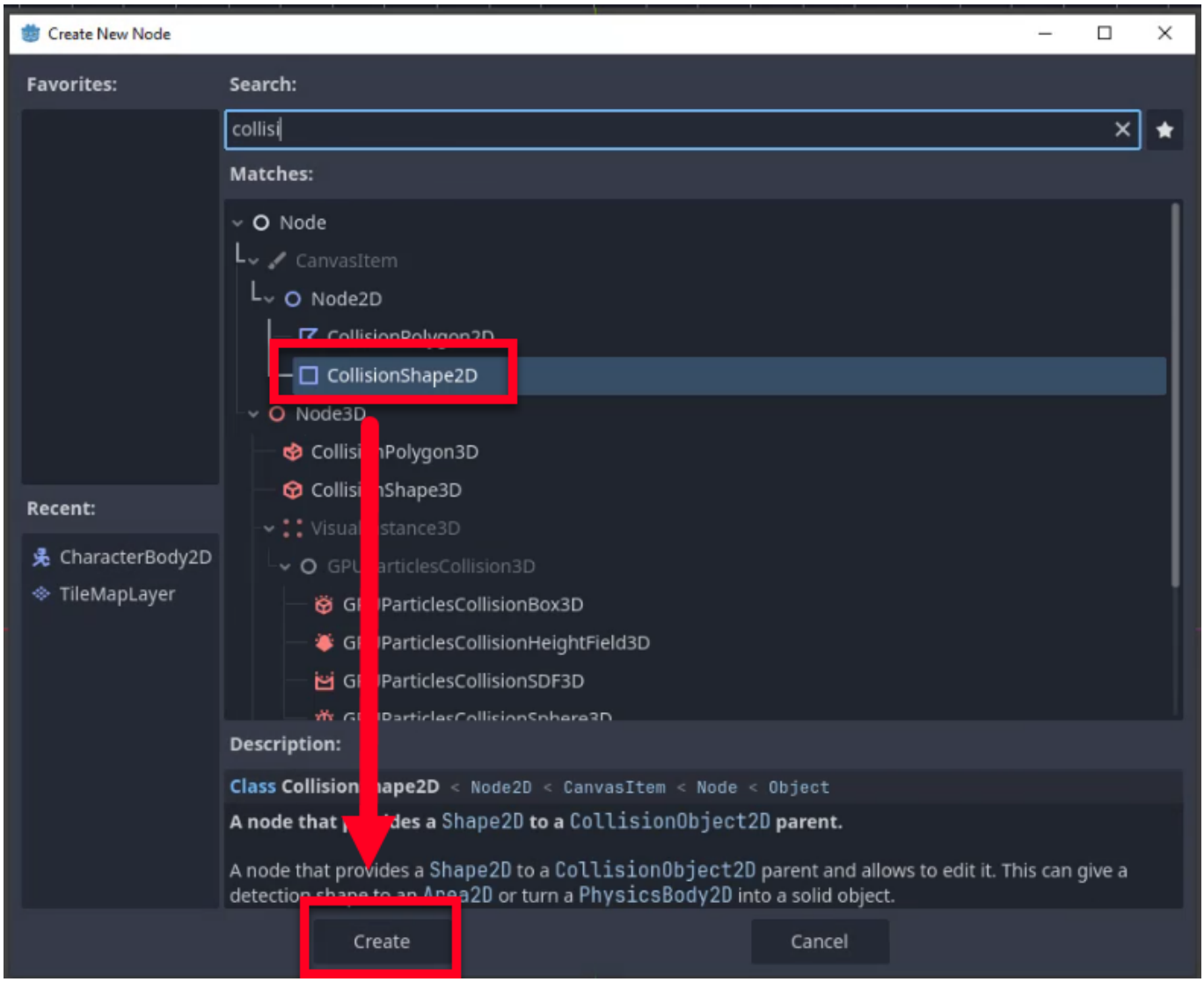
Rename the sprite node to “Sprite” for easy reference.



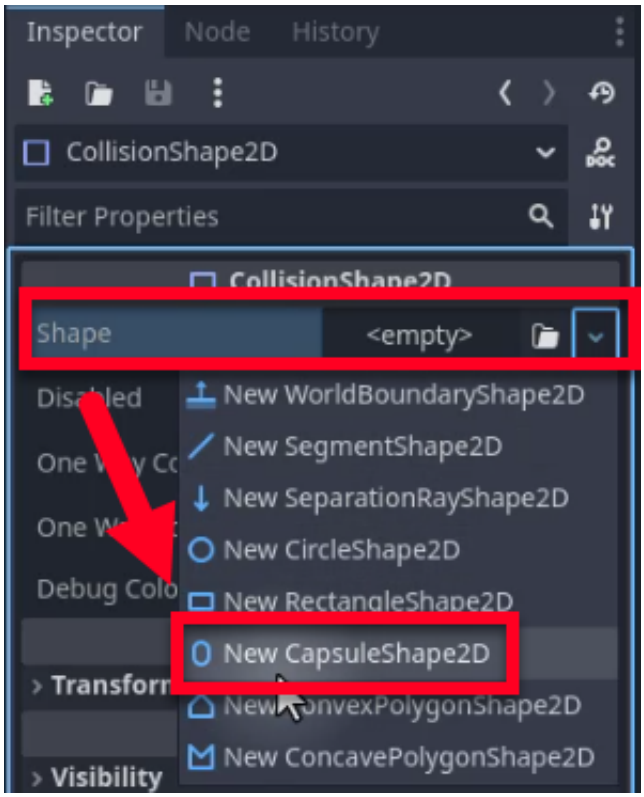
Center the sprite by setting its position to (0, 0) in the transform properties. We also need to adjust the offset of the sprite so that the origin is at the character’s feet. This is done by setting the Y offset to a negative value (e.g., -12).



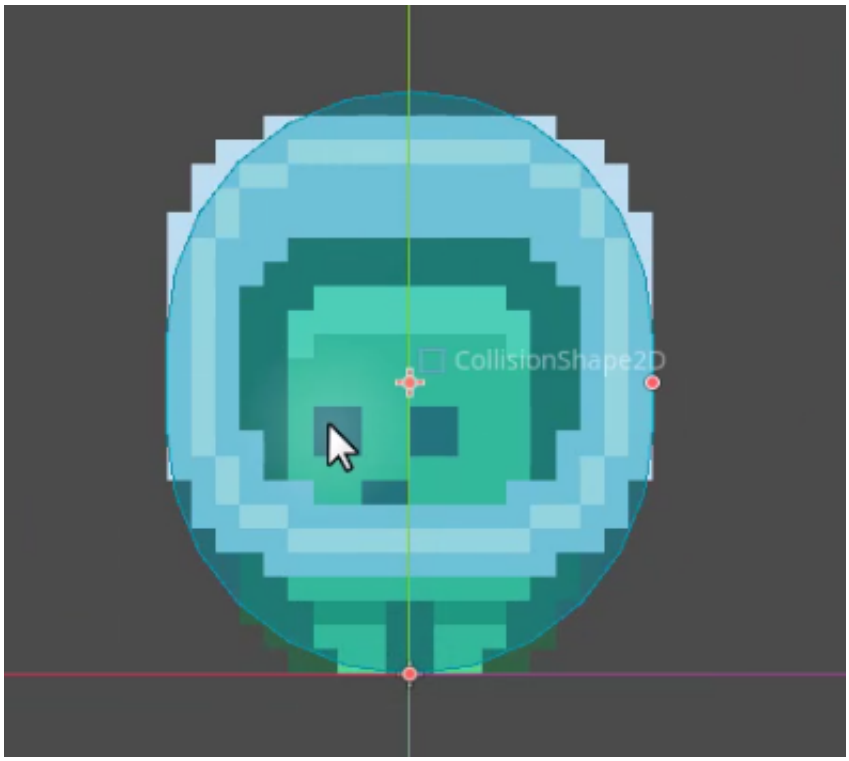
Add a collision shape to the player node by right-clicking on the player node, selecting “Add Child Node”, and choosing “CollisionShape2D”.



Set the shape of the collision shape to “CapsuleShape2D” to ensure smooth movement over bumps.



Adjust the collision shape to fit the sprite's dimensions, ensuring it touches the base of the character's feet.

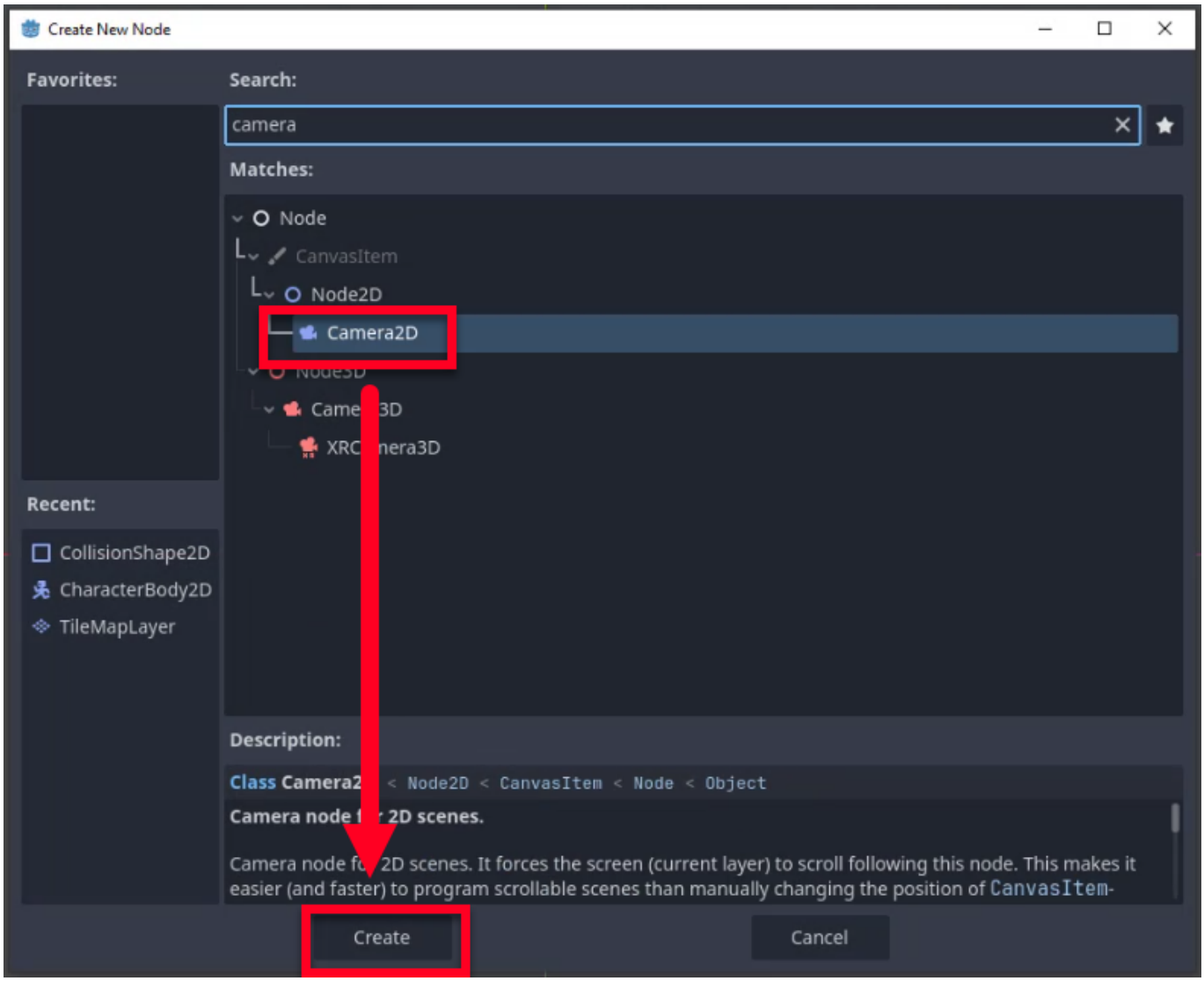


Setting Up the Camera

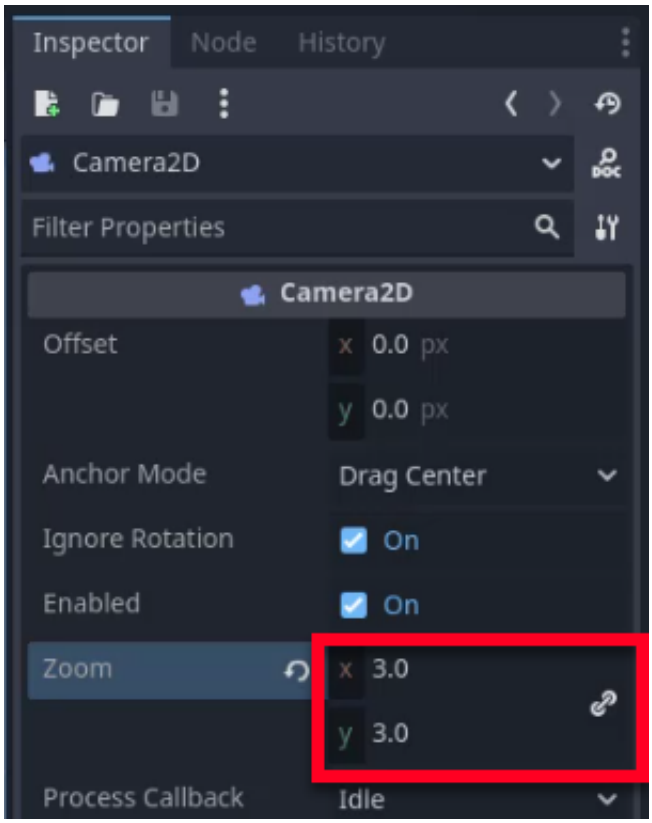
After creating the player scene, you need to set up a camera that follows the player. This ensures that the player remains in view as they move through the game world.

Steps to Set Up the Camera

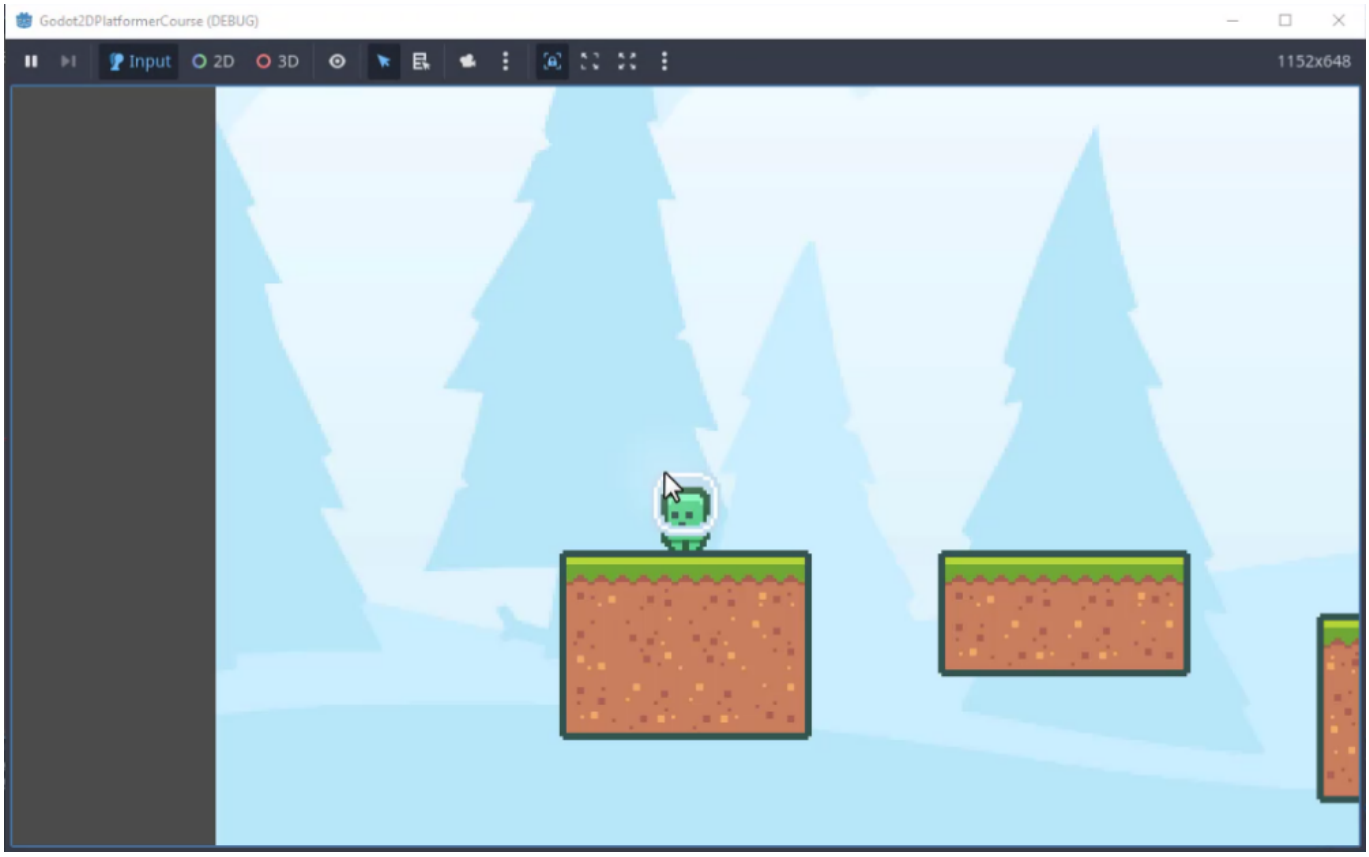
Add a Camera2D node as a child of the player node.



Adjust the zoom property of the camera to ensure it is focused on the player. A zoom value of (3, 3) is a good starting point.



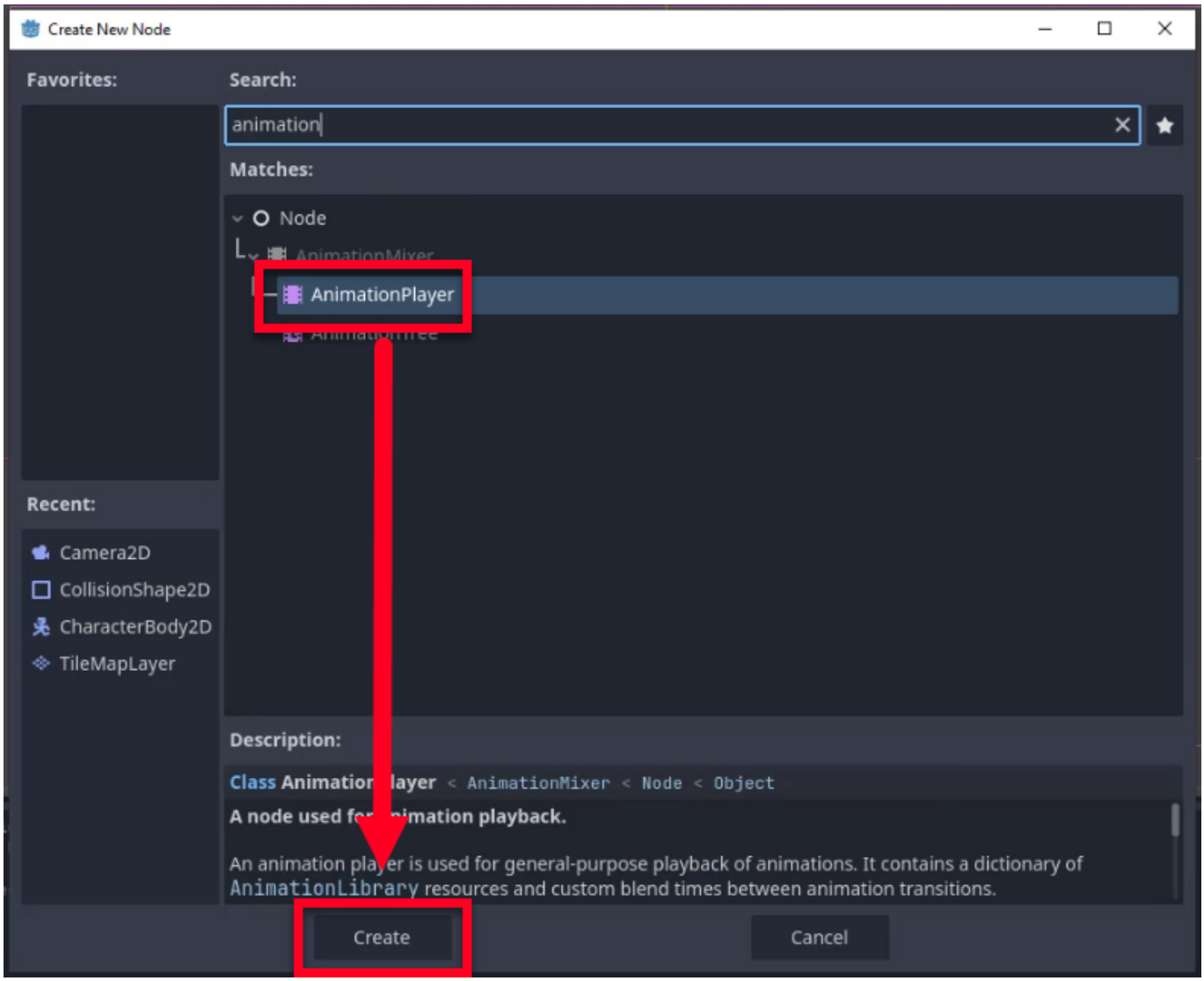
Position the camera slightly above the player to provide a better view of the game world.



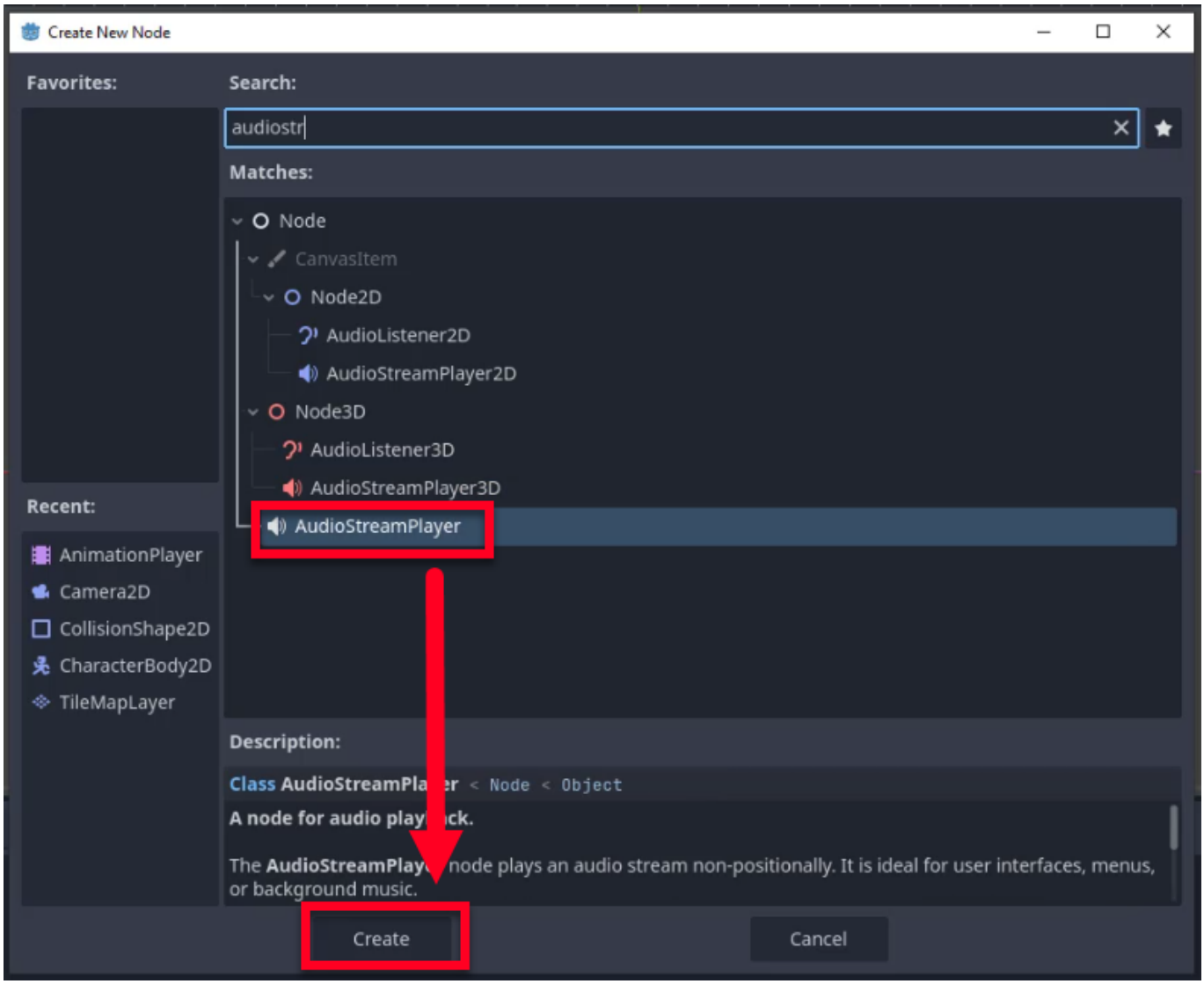
Adding Additional Nodes

To prepare for future lessons, we will add two more nodes to the player scene:

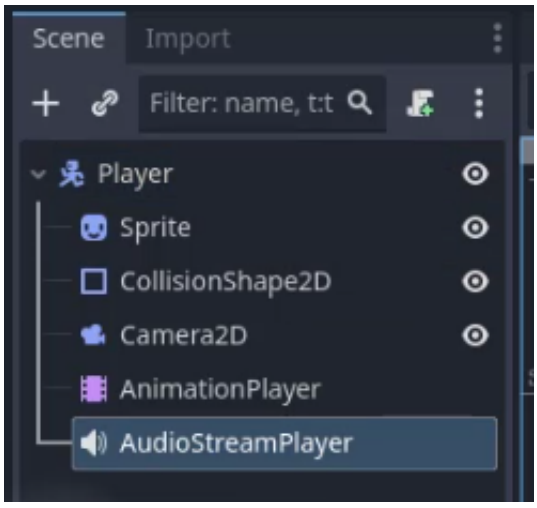
- AnimationPlayer: This node will handle the character's movement animations.



- AudioStreamPlayer: This node will manage sound effects for the character.

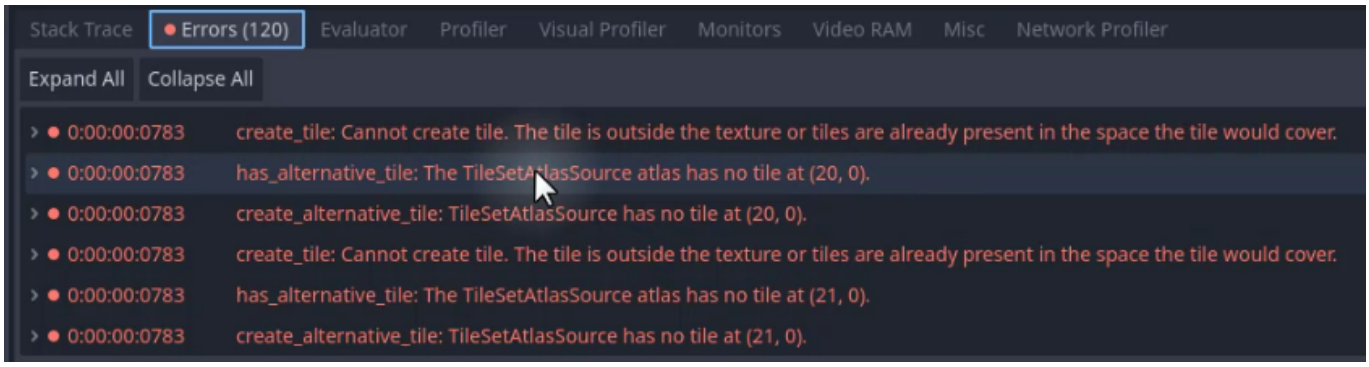


Your Player Scene should now have five child nodes in total.

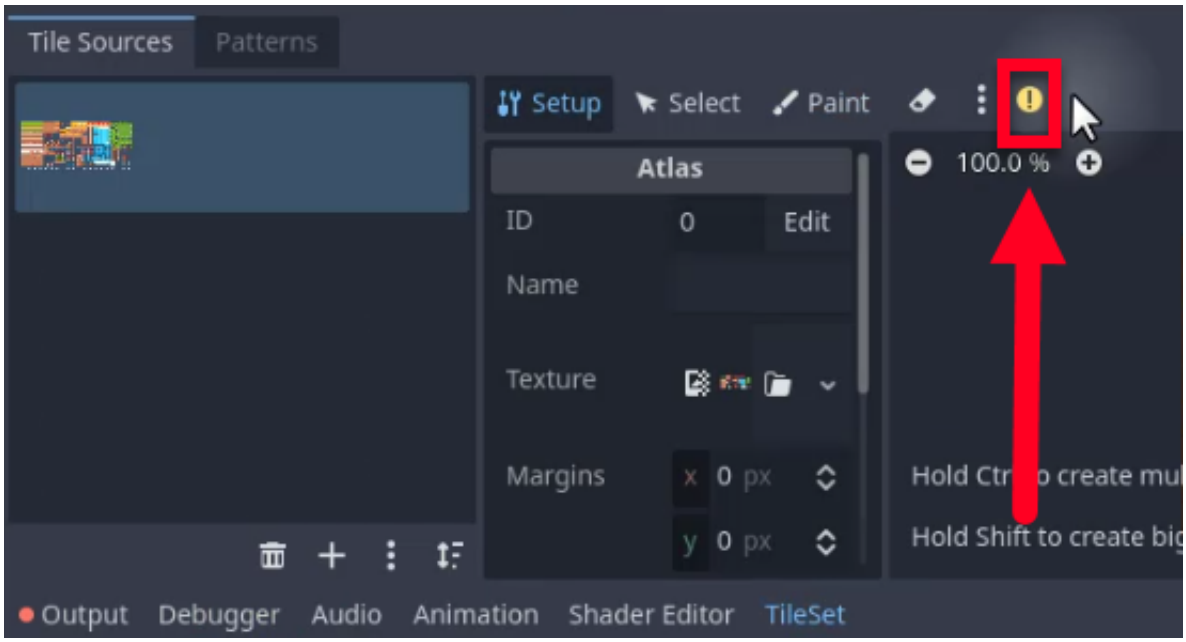


Fixing Common Errors

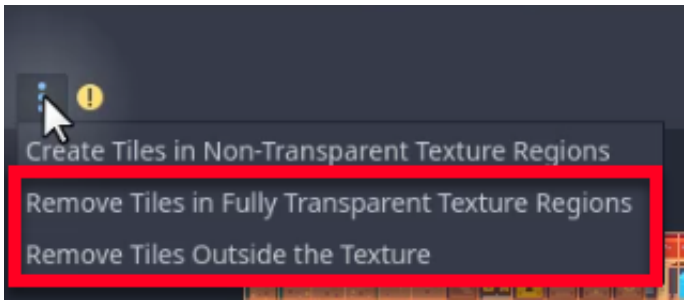
You may encounter errors related to the tile set atlas.



To fix these errors, open your TileSet and look for any yellow exclamation warning signs.



Click on the three dots and select “Remove Tiles Outside the Texture”. Also, select “Remove Tiles in Fully Transparent Texture Regions”.



By following these steps, you will have a fully functional player scene ready for further development. In future lessons, we will explore input handling and scripting to make the player character interactive.

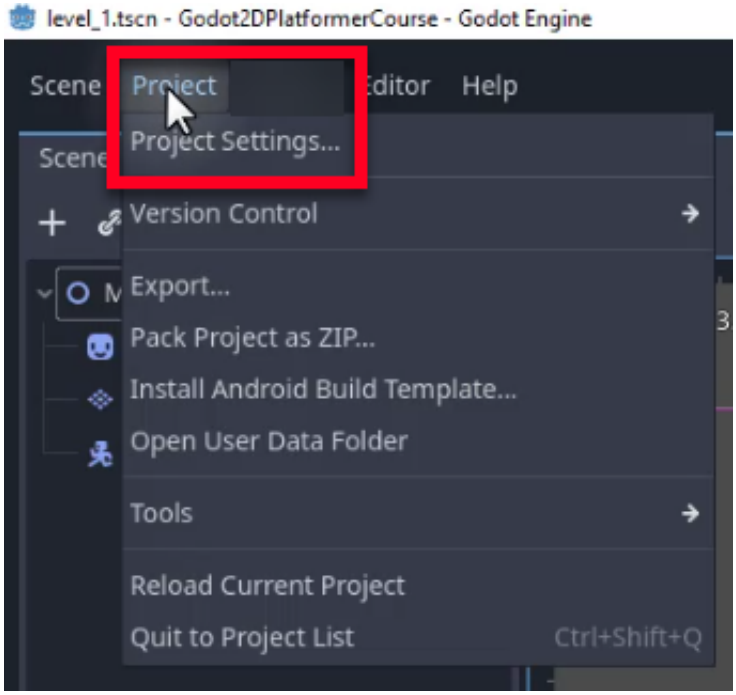
In this lesson, before we dive into scripting our player's movement, jumping, and physics, we need to establish our inputs. Inputs define which buttons on your keyboard, mouse, or controller perform specific actions. Instead of listening for specific keys like the left arrow key or the spacebar, we will listen for actions like 'jump' or 'move_left'. This approach makes our script more flexible and easier to manage.

Why Use Input Actions?

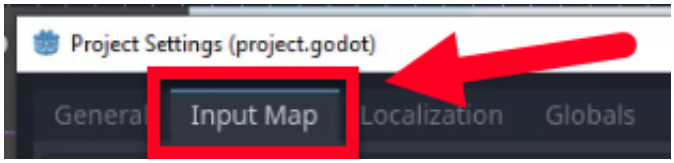
- Abstracts the input method from the script logic.
- Allows for multiple keys to trigger the same action.
- Makes it easier to add or change input methods, such as adding a controller.

Setting Up Input Actions

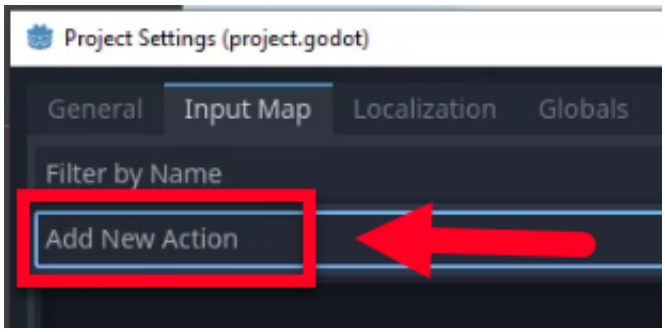
Let's get started by accessing the input map. Go to Project in the top menu and select Project Settings.



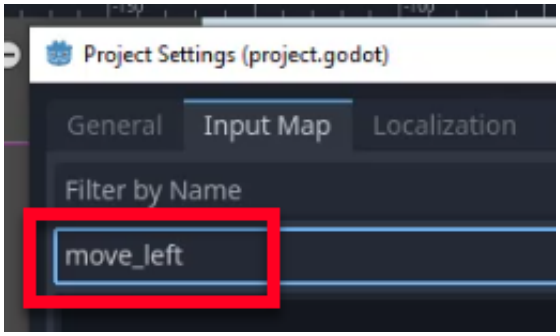
Once in the Project Settings, click on the Input Map tab.



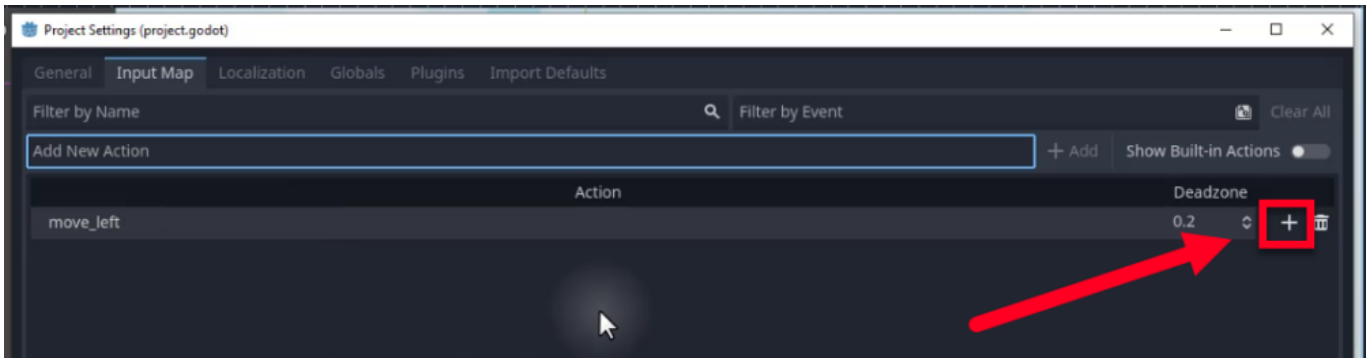
Now we can create new actions. Scroll down to Add New Action.



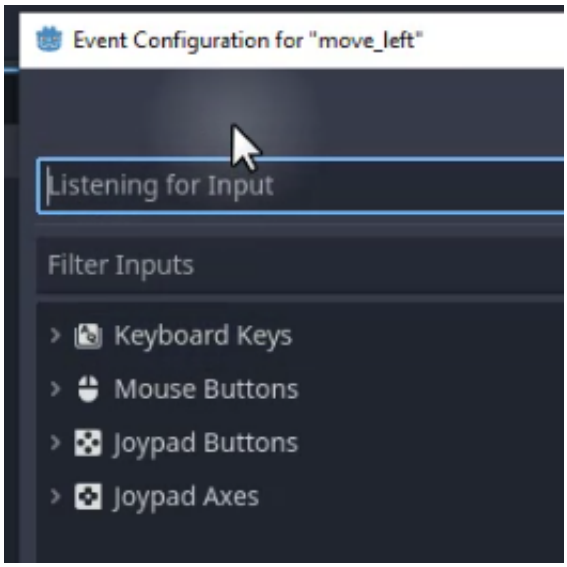
Name your action (e.g., move_left) and press Enter.



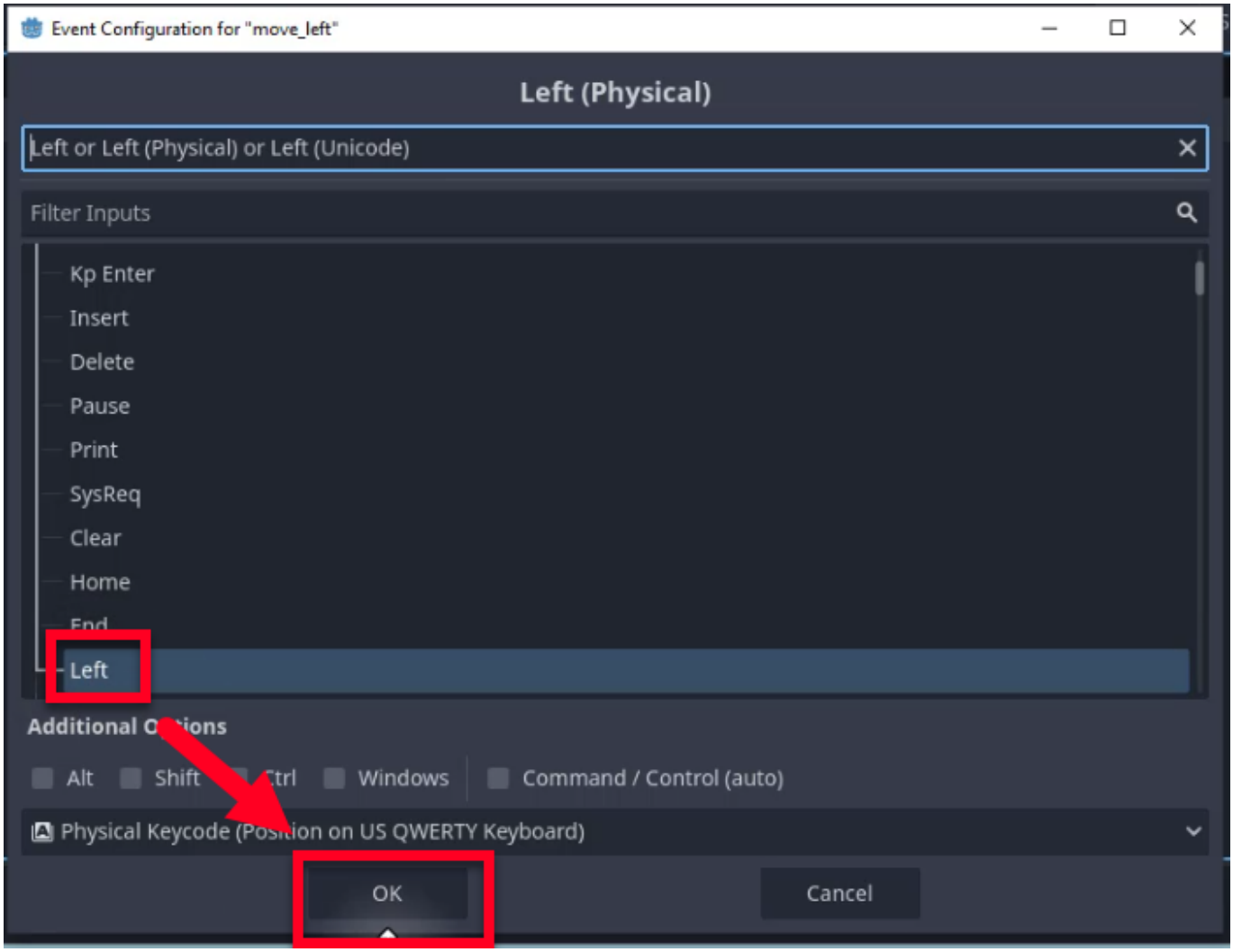
In order to use our action, we need to assign keys to the action. In the action's row, click the plus icon on the right.



Search for the desired key or click on 'Listening for input' and press the key.

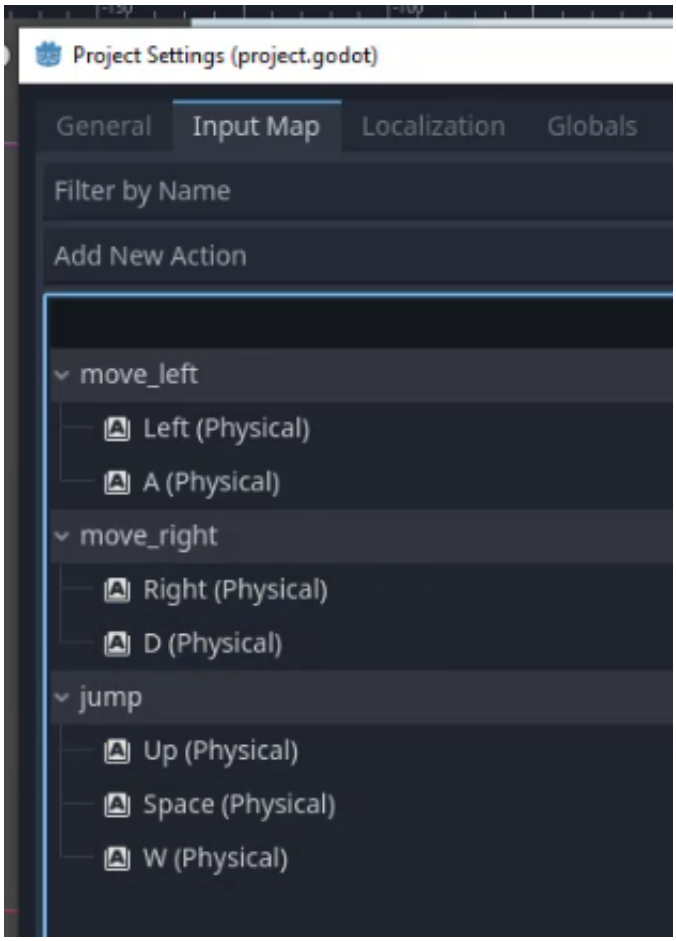


Once the correct key is located, click OK to assign the key.



We will now be able to move left with the left arrow key. Next, repeat the process for other actions (e.g., move_right, jump), as well as add different key options.

- move_left: Left Arrow Key, A Key
- move_right: Right Arrow Key, D Key
- jump: Up Arrow Key, Spacebar, W Key



By setting up your inputs in this way, you can easily manage and update your input methods without changing your script logic. This separation makes your code cleaner and more maintainable.

Next Steps

Now that we have our input actions set up and our player scene ready, the next lesson will focus on creating our player script. This script will handle movement, jumping, and other functionalities.

Godot 2D Platformer - Game Setup [2025]

1. What setting can you adjust to make pixel art appear crisp in Godot?

a. The sprite’s resolution
b. The sprite's collision shape
c. The default texture filter setting in project settings
d. The zoom setting of the camera

2. Which node do we use to paint a tilemap?

a. TileSet
b. Area2D
c. Node2D
d. TileMapLayer

3. True or False: To make sure your tiles have collision properties, you need to apply a physics layer to each tile that requires collision.

a. True
b. False

4. What is the recommended node for creating a player character in Godot for a platformer game?

a. CharacterBody2D
b. Sprite2D
c. Node2D
d. StaticBody2D

5. True or False: The CharacterBody2D node in Godot automatically provides collision detection and velocity management for the player.

a. True
b. False

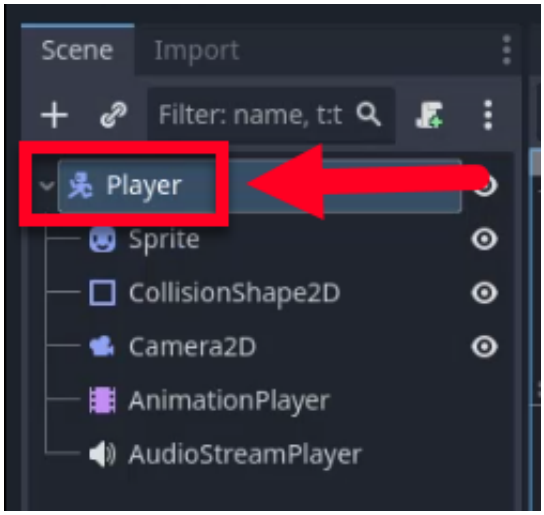
6. Why is it useful to use input actions instead of hardcoding specific keys in Godot?

a. It allows for better graphics in the game
b. It makes the game load faster
c. It simplifies collision detection
d. It makes the input system more flexible and easier to manage across devices

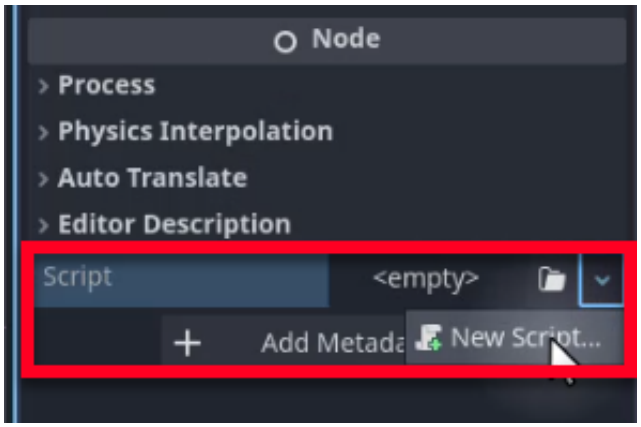
In this lesson, we will focus on setting up the core functionality of our player script, which includes moving and jumping. By the end of this lesson, you will have a player character that can move left and right and fall due to gravity.

Setting Up the Player Script

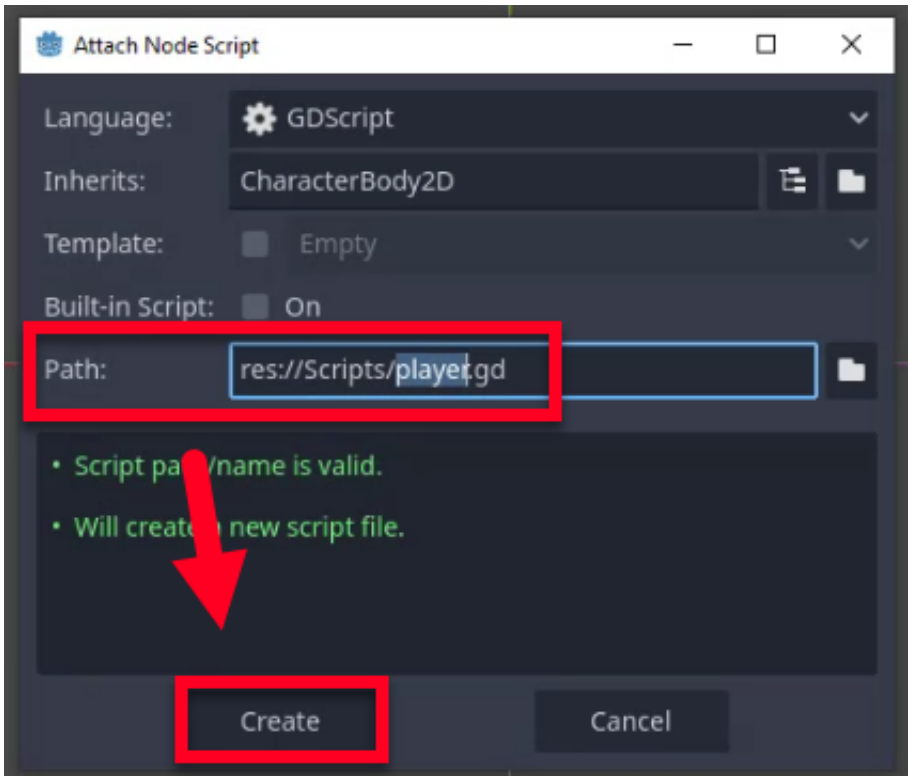
To begin, we need to create a new script for our player. Open your player scene and select the CharacterBody2D node (named Player).



Click on the “Attach Script” icon at the bottom of the Inspector, and select “New Script” from the dropdown.



Save the script as player.gd in the Scripts folder.



Creating Variables

Our player script will need several variables to control movement and physics. We will create variables for move speed, acceleration, braking, gravity, and jump force. Additionally, we will create a variable to store the player's input.

Here is the initial setup of our player.gd script:

```
extends CharacterBody2D

@export var move_speed : float = 100
@export var acceleration : float = 50
@export var braking : float = 20
@export var gravity : float = 500
@export var jump_force : float = 200

var move_input : float
```

- move_speed: The maximum speed at which the player can move.
- acceleration: The rate at which the player accelerates to the move speed.
- braking: The rate at which the player slows down when no movement keys are pressed.
- gravity: The force of gravity applied to the player.
- jump_force: The impulse velocity added when the player jumps.
- move_input: A variable to store the player's input for movement.

Implementing Movement

To implement movement, we will use the _physics_process function, which is called at a consistent rate per second, independent of the frame rate. This ensures that our physics calculations are consistent and deterministic.

Here is the code to set up basic movement:

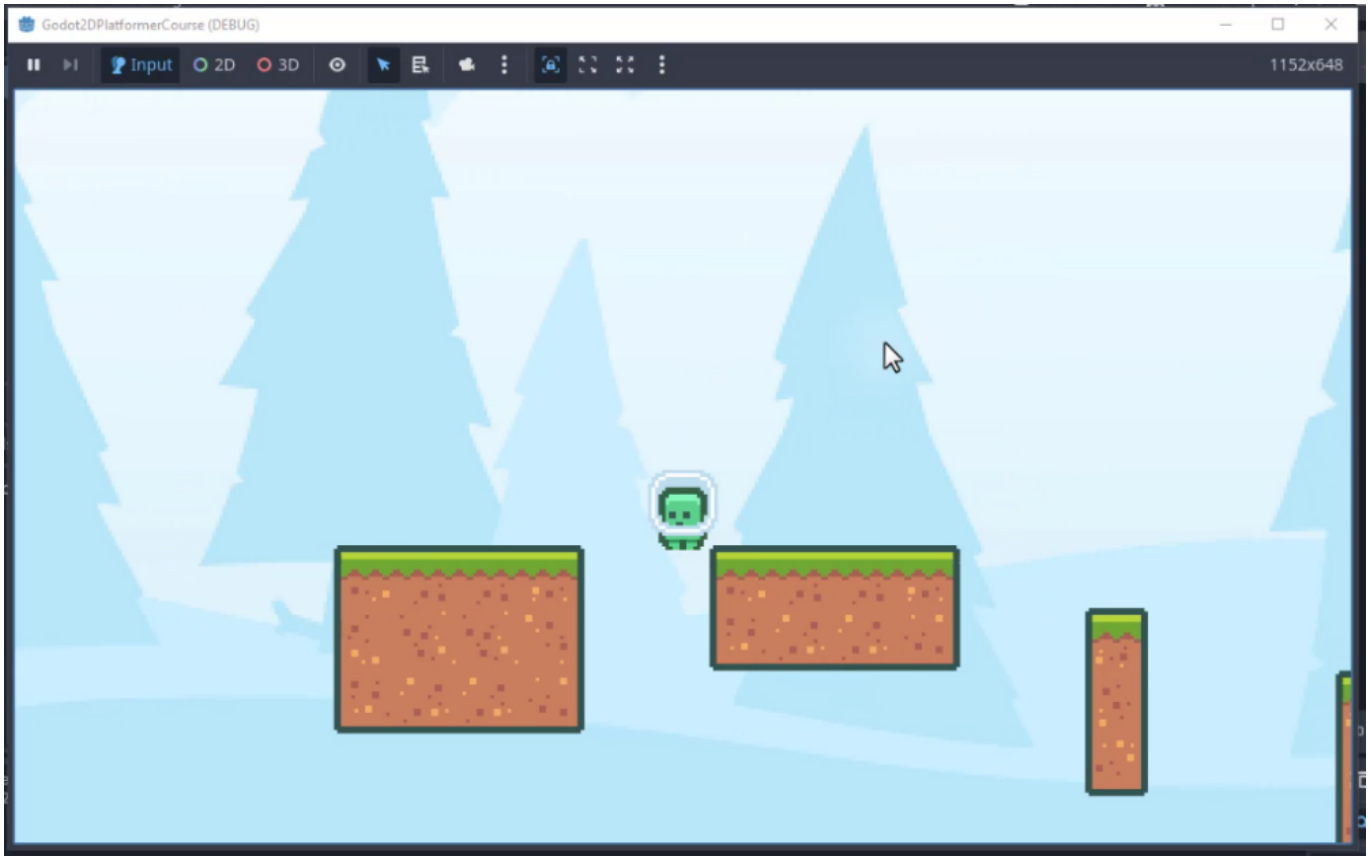
```
func _physics_process(delta):
    # Get the move input
    move_input = Input.get_axis("move_left", "move_right")

    # Movement
    velocity.x = move_input * move_speed

    move_and_slide()
```

In this code:

- Input.get_axis("move_left", "move_right"): This function returns a value between -1 and 1 based on the player's input. If the player presses the left movement key, it returns -1. If the player presses the right movement key, it returns 1. If no keys are pressed, it returns 0.
- velocity.x = move_input * move_speed: This line sets the player's horizontal velocity based on the input and the move speed.
- move_and_slide(): This function applies the velocity to the player's position and handles collision detection.

Adding Gravity

If you test the game, you can make your character move left and right, but they can seemingly float in mid-air.



To make the player fall due to gravity, we need to modify the velocity.y property. We will add gravity to the player's vertical velocity only when the player is not on the floor.

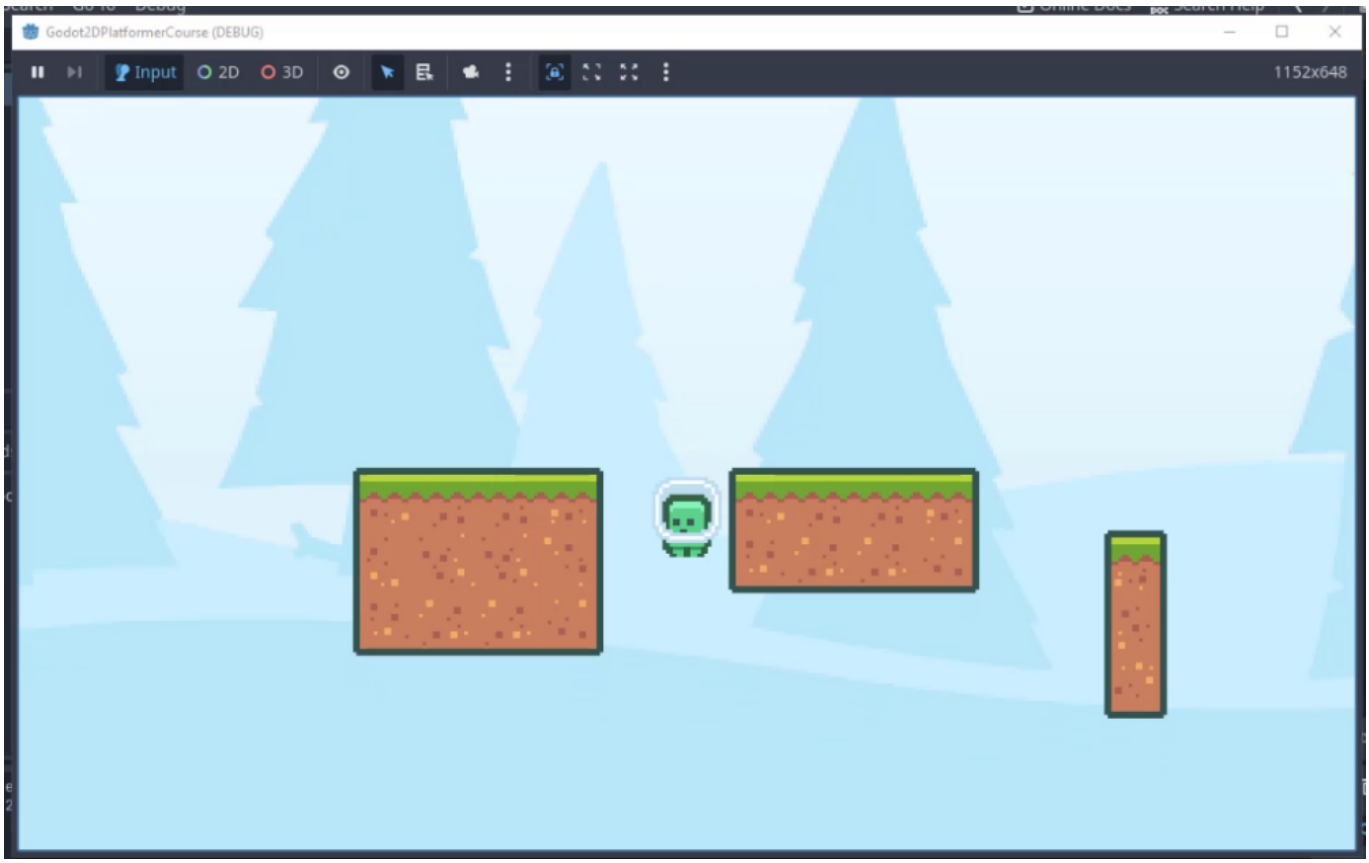
Here is the updated _physics_process function with gravity:

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

    # Get the move input
    move_input = Input.get_axis("move_left", "move_right")

    # Movement
    velocity.x = move_input * move_speed

    move_and_slide()
```

In this code:

- if not is_on_floor(): This condition checks if the player is not on the floor.
- velocity.y += gravity * delta: This line adds gravity to the player's vertical velocity, making the player fall.

Now the character will fall properly with gravity.



Conclusion

In this lesson, we created a player script that handles basic movement and gravity. We set up variables for movement and physics, implemented movement using the _physics_process function, and added gravity to make the player fall. In the next lesson, we will add jumping and refine our movement with acceleration and braking.

In this lesson, we will add jumping functionality to our character's movement. This is a crucial part of our character's control scheme and will make the game more interactive. Let's dive into the steps required to implement jumping in Godot.

Understanding Jumping Mechanics

To implement jumping, we need to consider a few key points:

- Jumping should only occur if the jump key is pressed.
- The character should only jump if they are on the ground.
- The velocity should be modified before calling the move_and_slide function.

Implementing Jumping

We will add the jumping logic in the _physics_process function. As a refresher, this function is called at a fixed rate per second, making it ideal for physics-related updates.

Step-by-Step Guide

1. Add a comment to indicate where the jumping logic will be implemented. This helps in organizing the code and understanding its structure.

   ```
   # Jumping
   ```

2. Check if the jump key is pressed and if the character is on the ground. In Godot, the jump key can be the up arrow, W, or the space bar. We use the is_on_floor function to check if the character is on the ground.

   ```
   if Input.is_action_pressed("jump") and is_on_floor():
   ```

3. Modify the velocity to make the character jump. In Godot's 2D mode, the Y-axis increases downwards. Therefore, to make the character jump upwards, we set the Y velocity to a negative value.

   ```
   velocity.y = -jump_force
   ```

4. Ensure that the move_and_slide function is called after modifying the velocity. This function applies the velocity to the character and handles collision detection.

   ```
   move_and_slide()
   ```

Complete Jumping Code

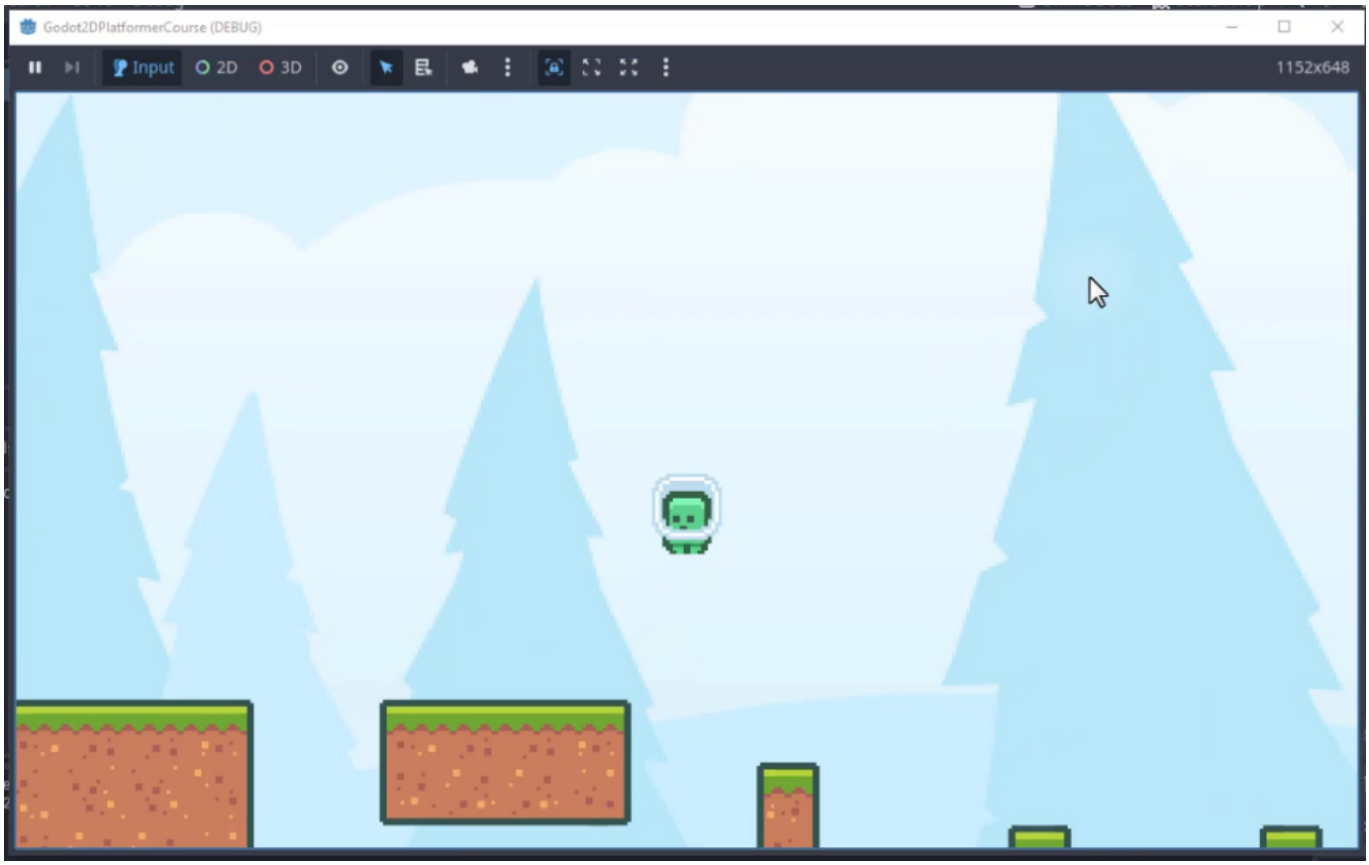
Here is the complete code snippet for implementing jumping:

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

    # get the move input
    move_input = Input.get_axis("move_left", "move_right")

    # movement
```



```
    velocity.x = move_input * move_speed

    # jumping
    if Input.is_action_pressed("jump") and is_on_floor():
        velocity.y = -jump_force

    move_and_slide()
```

Your character should now be able to jump as well as move left and right.



Smoothing the Movement

After implementing jumping, we need to smooth out the character's movement to make it feel more natural. We will use linear interpolation (lerp) to achieve this. In Godot, the lerp function is used to smoothly transition a value from one point to another over time, allowing for a more natural and fluid animation. By specifying a start value, end value, and a weight (or amount of progress), the lerp function calculates a new value that is a percentage of the way between the start and end values, creating a smooth and continuous movement.

Remove the direct assignment of velocity.x and replace it with linear interpolation for acceleration and braking.

```
if move_input != 0:
        velocity.x = lerp(velocity.x, move_input * move_speed, acceleration * delta)
    else:
        velocity.x = lerp(velocity.x, 0.0, braking * delta)
```


Complete Movement Code with Smoothing

Here is the complete code snippet for the character's movement with smoothing:

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

    # get the move input
    move_input = Input.get_axis("move_left", "move_right")

    # movement
    if move_input != 0:
        velocity.x = lerp(velocity.x, move_input * move_speed, acceleration * delta)
    else:
        velocity.x = lerp(velocity.x, 0.0, braking * delta)

    # jumping
    if Input.is_action_pressed("jump") and is_on_floor():
        velocity.y = -jump_force

    move_and_slide()
```

Summary

In this lesson, we implemented jumping for our character and smoothed out the movement to make it feel more natural. We used the _physics_process function to handle physics-related updates and ensured that the move_and_slide function was called after modifying the velocity.

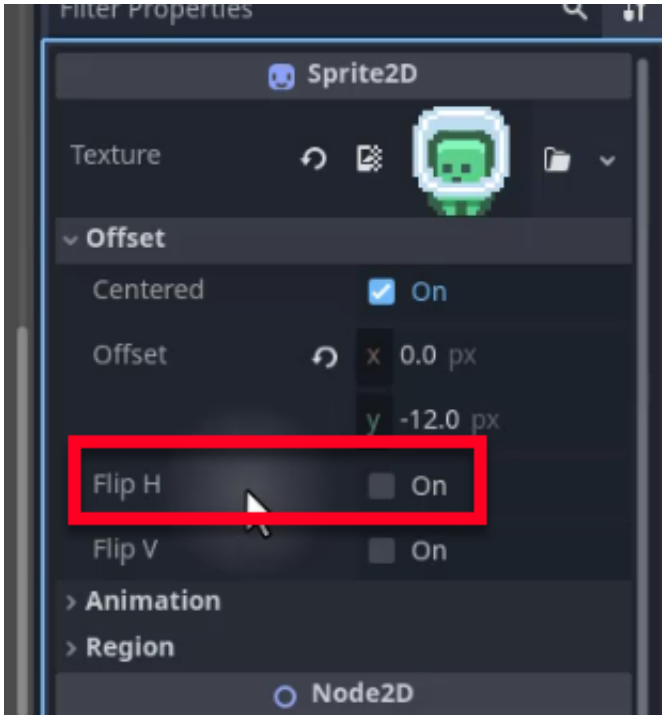
In the next lesson, we will start working on the character's visuals, including animations and sprite flipping to face the direction of movement.

In this lesson, we will begin setting up our player's animations. This process involves two main parts: animating the player with different sprites for various actions (such as moving and jumping) and ensuring the sprite faces the direction of movement. Let's dive into the details.

Setting Up the Player Sprite

To start, we need to ensure that our player sprite faces the direction of movement. This involves flipping the sprite horizontally based on the player's velocity. Here's how we can achieve this:

1. **Create a Reference to the Sprite Node:** First, we need to create a variable to reference the sprite node in our player script. This allows us to manipulate the sprite properties easily.
2. **Flip the Sprite Based on Velocity:** We will use the flip_h property of the sprite to flip it horizontally based on the player's movement direction.



Step-by-Step Implementation

1. **Open the Player Script:** Navigate to your player script. This is where we will add the code to handle the sprite flipping.
2. **Create a Variable for the Sprite Node:** At the top of your script, create a variable to reference the sprite node. Use the @onready keyword to initialize this variable when the node is ready.

```
extends CharacterBody2D

@onready var sprite : Sprite2D = $Sprite
```

3. **Flip the Sprite in the _process Function:** In the _process function, add the code to flip the sprite based on the player's velocity. The _process function is called every frame, making it ideal for updating the sprite's orientation.

```
func _process(delta):
        sprite.flip_h = velocity.x > 0
```

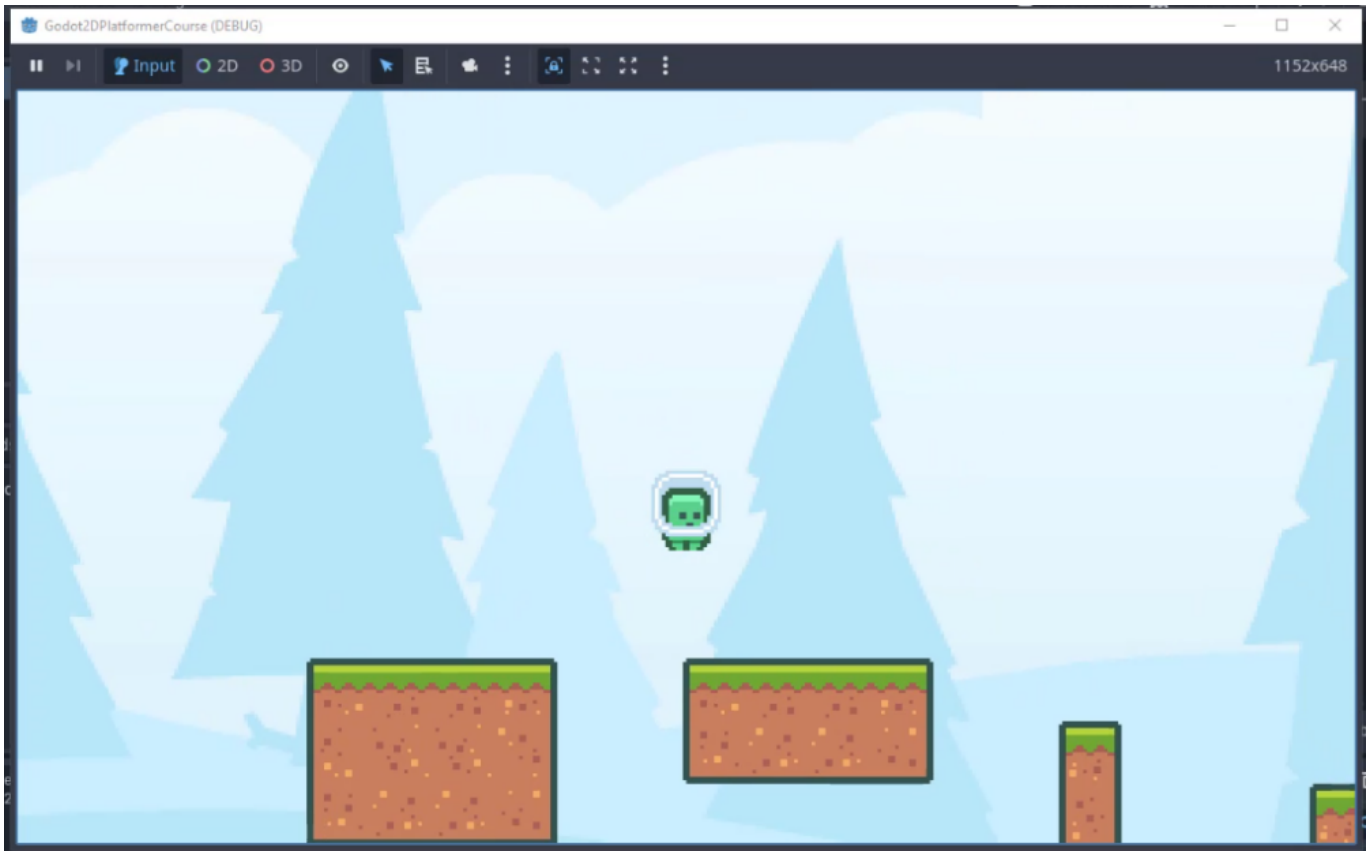
This code checks the player's velocity in the x-direction. If the velocity is greater than 0 (meaning the player is moving to the right), the sprite will flip horizontally to face right. If the velocity is less than 0 (meaning the player is moving to the left), the sprite will face left.

Handling Edge Cases

Depending on the default orientation of your sprite, you might need to adjust the condition. If your sprite defaults to facing right, you should flip it when the velocity is less than 0. Additionally, to prevent the sprite from flipping back to its original orientation when the player stops moving, you can add an if statement to only flip the sprite when the player is moving:

```
func _process(delta):
    if velocity.x != 0:
        sprite.flip_h = velocity.x > 0
```

This ensures that the sprite only flips when the player is actively moving, preventing any unwanted flipping when the player is idle. You should be able to test the game now and see the sprite flipped based on how you're moving.



Next Steps

In the next lesson, we will begin setting up our animations using the AnimationPlayer node. We will

create animations for idle, moving, and jumping states and switch between them based on the player’s current state.

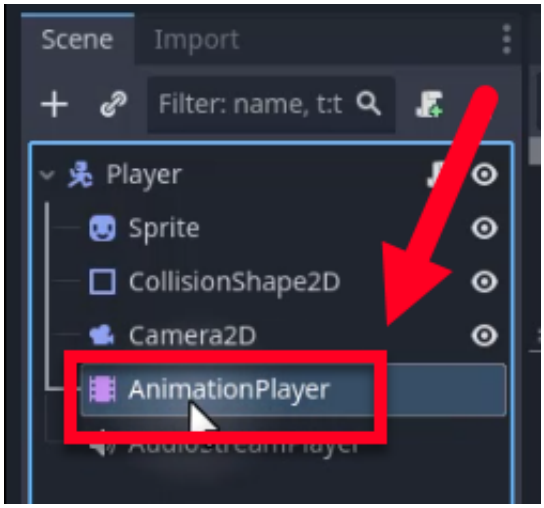
In this lesson, we will begin setting up animations for our player character in Godot. Animations are crucial for bringing your game characters to life. For this first part, we will focus on creating three essential animations: idle, moving, and jumping. Let's dive right in!

Understanding Animations in Godot

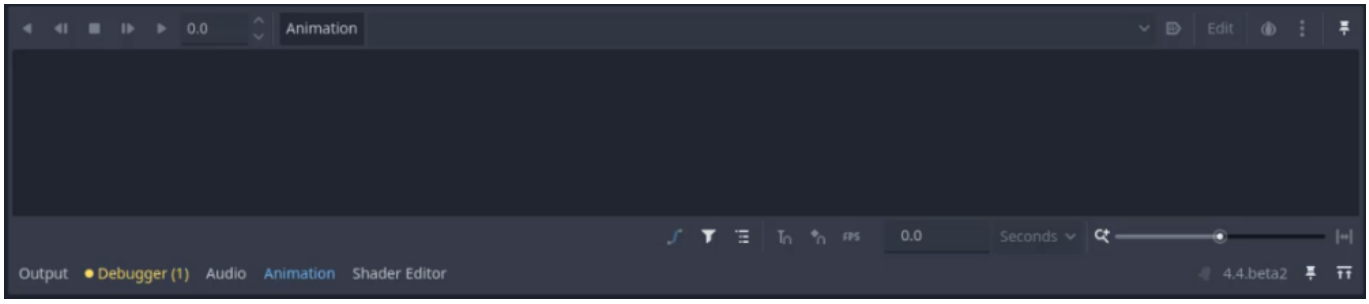
In Godot, an animation is essentially a timeline that starts from 0 seconds and ends at a specified duration. Along this timeline, you can add **keyframes** that change the properties of your sprite or any other node. For our player, we will be animating the texture property to switch between different sprites.

Setting Up the Animation Player

To begin, ensure you have an **AnimationPlayer** node in your player scene. If not, create one.

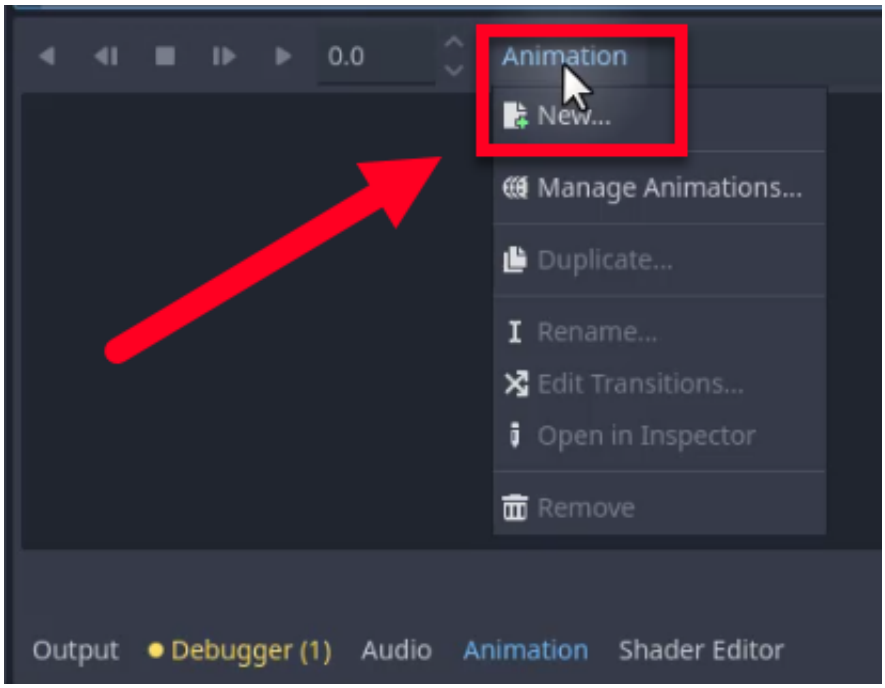


When you select the AnimationPlayer node, you should see the animation window at the bottom of the editor.

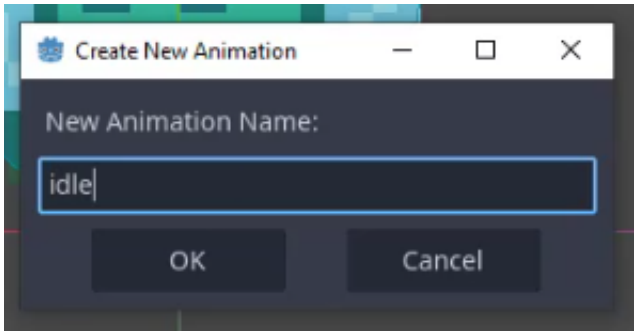


Creating the Idle Animation

Let's start by creating the Idle animation. In the animation window, click on **Animation > New**.

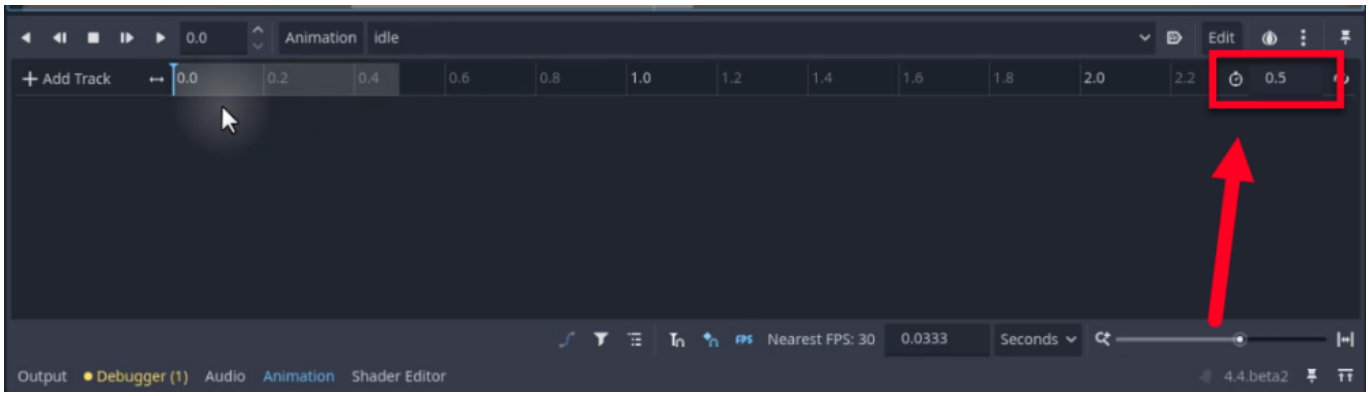


Name the animation “idle”.

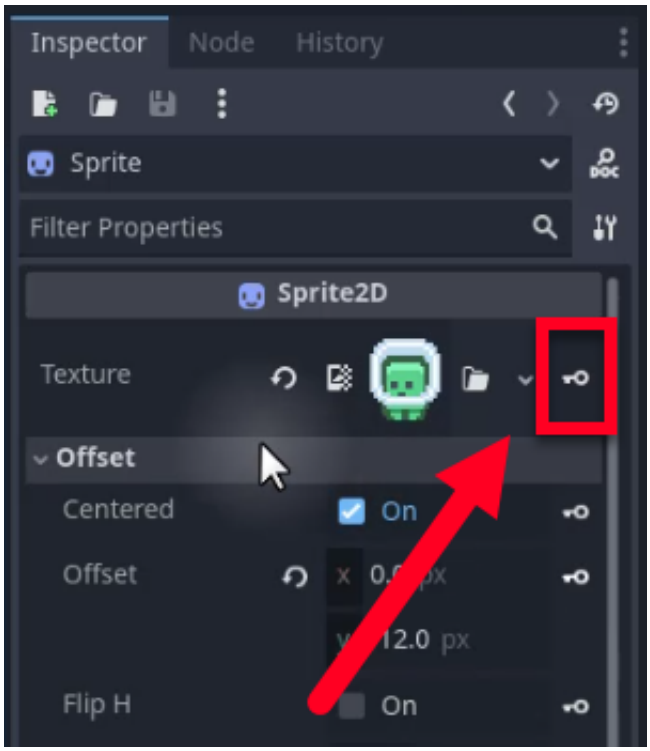


Zoom in on the timeline using the magnifying glass icon to work in smaller increments.

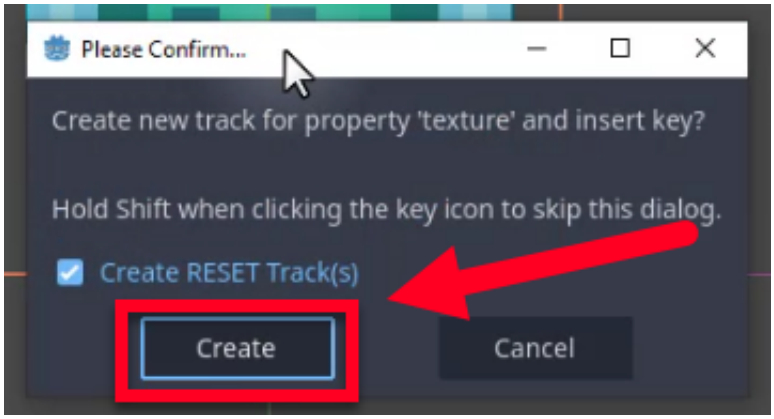
By default, most animations are set to last one second. We only want this animation to last half a second. To do this, click the duration field near the top right and alter the value to be 0.5.



In order to animate anything, we need to configure the animation to access tracks, which control the properties of our objects. Keeping the animation window open, select your sprite node. Click the key icon next to the **Texture** property to add a new track.



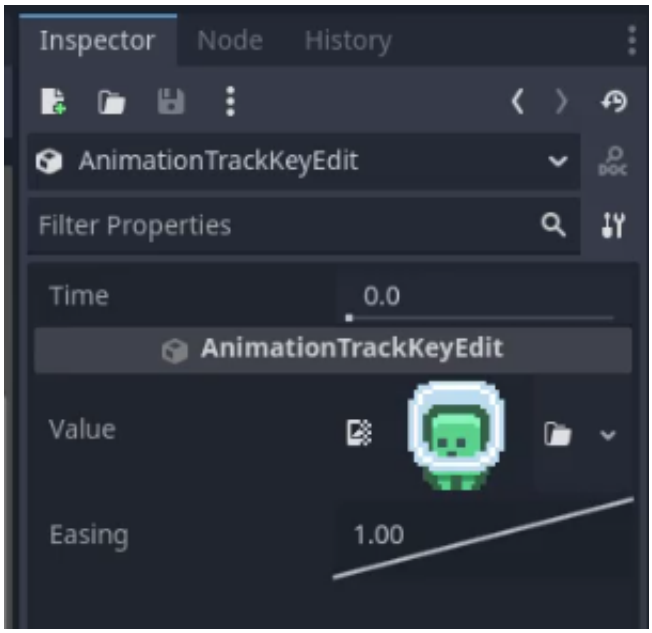
On the pop-up window, hit “Create” to confirm the creation of the track. Ensure **Create Reset Tracks** is enabled (we’ll discuss RESET tracks in a bit).



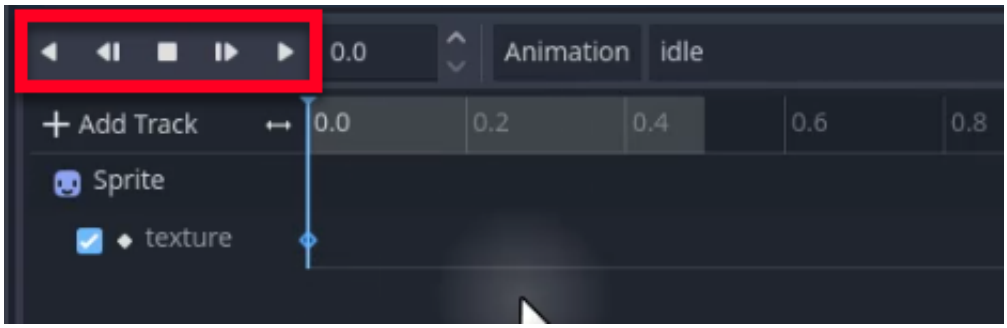
On the Animation timeline, you should now see the track for the Sprite's texture property.

For animations, once we have a property we can access in the timeline, we need to create keyframes. Keyframes control the key points in the animation where something needs to happen. In this case, we'd be changing the texture in each keyframe we create.

You can see at the 0.0 mark that we already have a keyframe for the current sprite texture created automatically.



For the idle animation, we don't need any other keyframes. You can test the idle animation using the play, reverse, and pause buttons in the animation window.



Understanding Reset Tracks

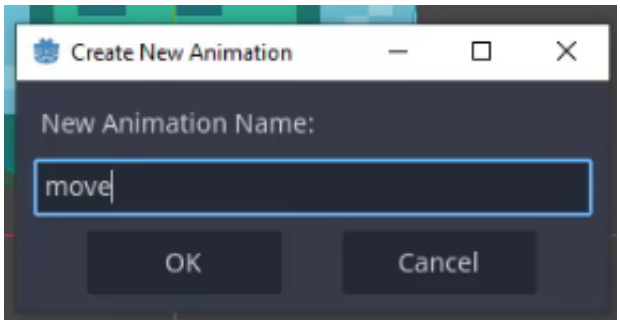
When creating our idle animation, we also created a RESET track. Reset tracks are essential for returning your character to its default state after an animation. Here's how to use them:

- The reset animation contains default values for all modified properties.
- It ensures that any changes made during an animation can be reset to their original state.
- You can find the reset animation in the dropdown menu next to the animation button.

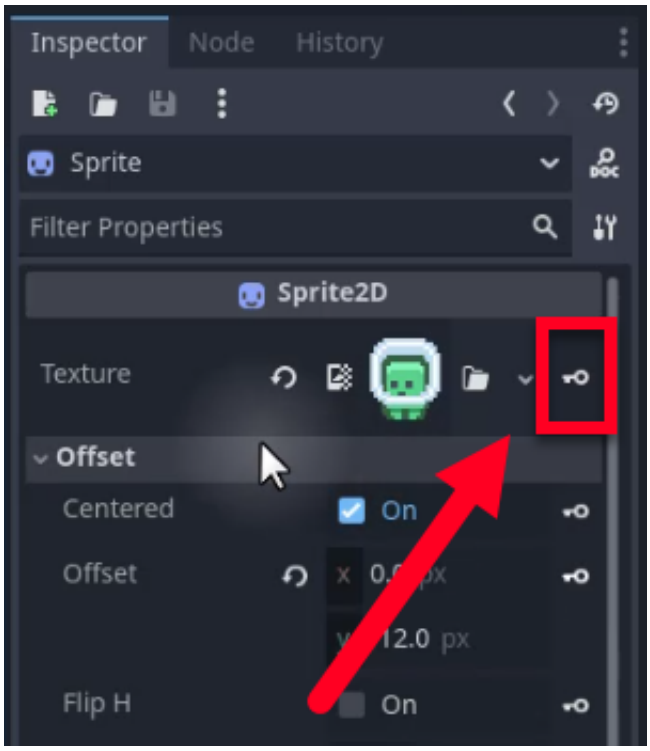
Creating the Move Animation

Now that we have the idle animation set up, let's set up the animation that will play when the character moves. Once again, click on **Animation** and select **New**.

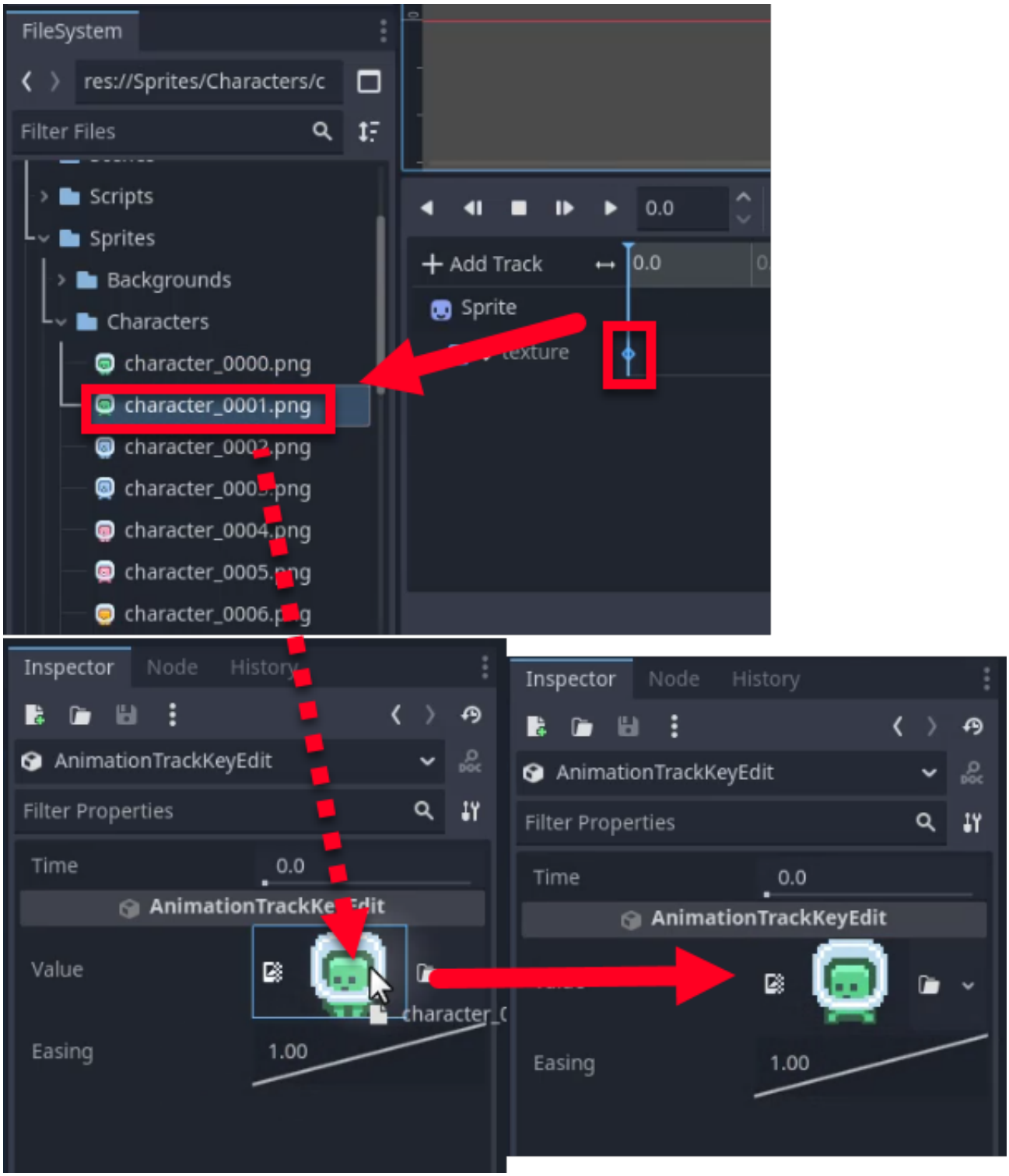
We'll name this animation "move".



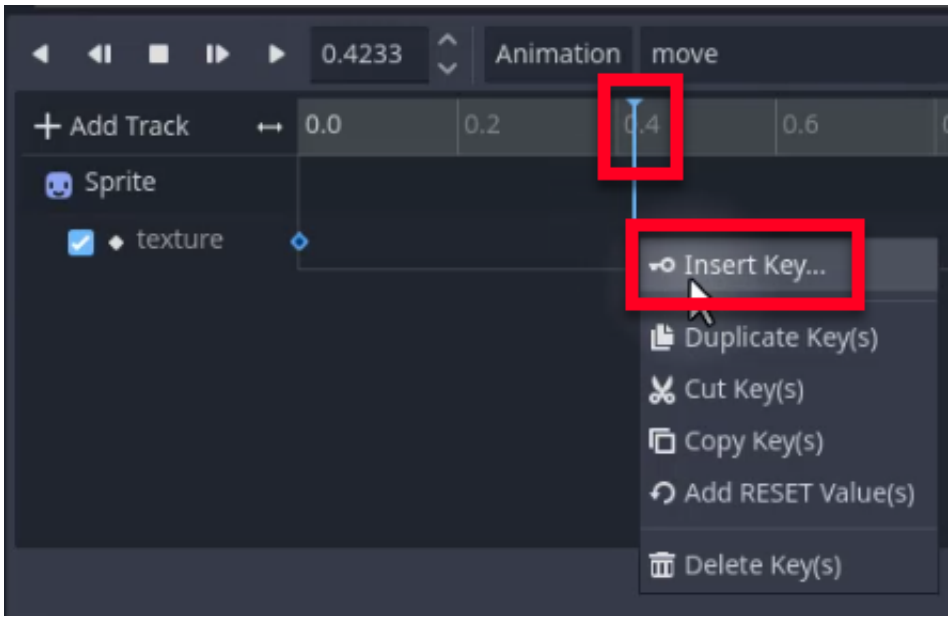
As before, add the texture property track.



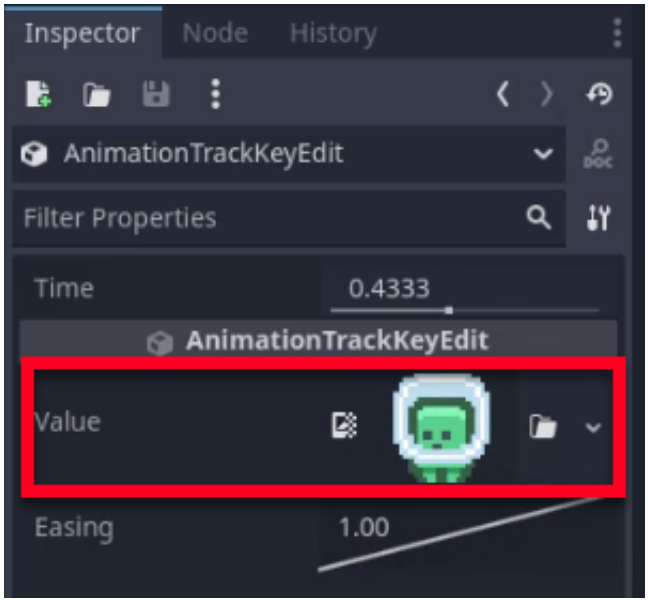
Next, set the first keyframe to the sprite where the character's legs are up. You can do this by dragging the appropriate sprite from the FileSystem into the Value property on the keyframe's data in the Inspector.



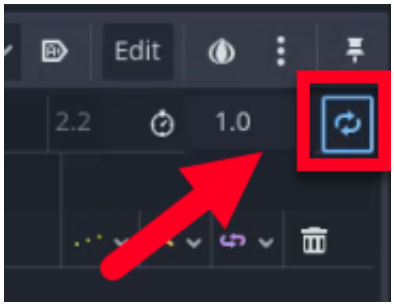
Let’s actually create a new keyframe this time. Move to about halfway on the timeline, right-click on the track, and select **Insert Key**.



Set the second keyframe to the sprite where the character's legs are down.

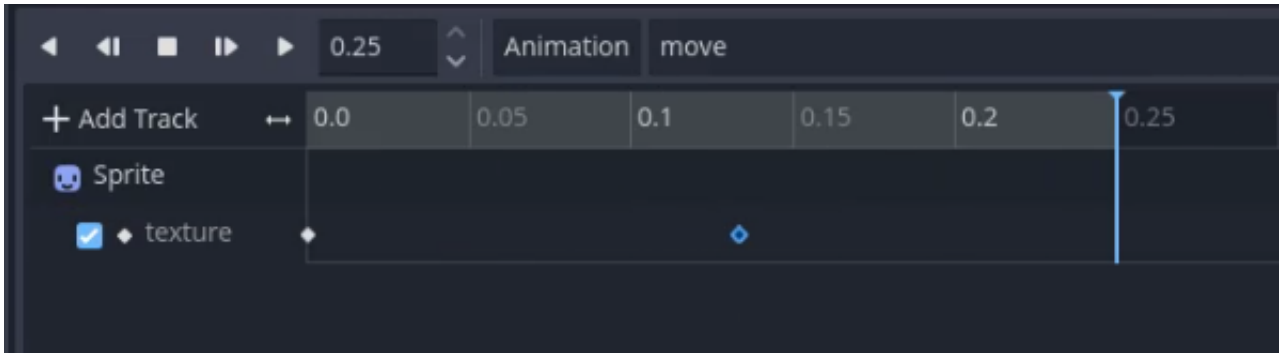


In order for our running effect to work properly, we need this animation to loop. Enable animation looping by clicking the loop icon on the right.



Our current animation is a bit slow, so we'll adjust the animation duration to speed up the movement. 0.25 is a good value, but you can customize it how you see fit.

With the animation shorter now, you'll notice that the second keyframe is now outside the animation duration – thus it would never be seen. In order to fix this, zoom in and drag the second keyframe to be about halfway between the beginning and end again.



You can further tweak the keyframes to achieve the desired running effect.

Creating the Jump Animation

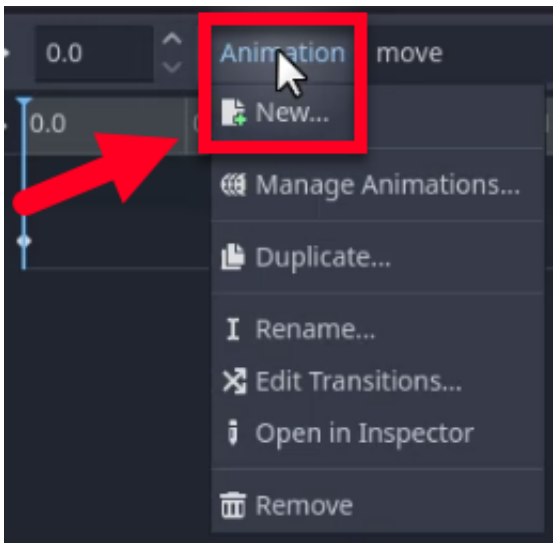
As a challenge, try creating the jump animation using the sprite with the character's legs in the air. This animation will be a static pose with one leg in the air, indicating that the character is jumping.

We'll go over the solution to this in the next lesson!

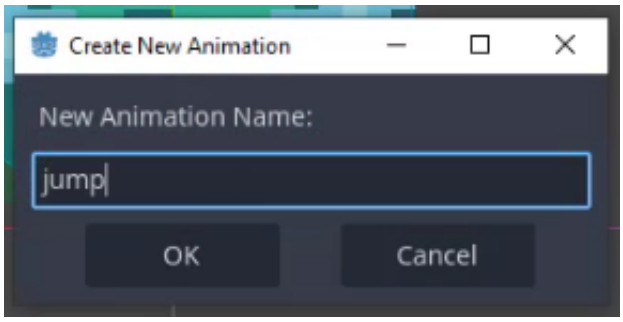
In this lesson, we will focus on creating the jump animation and managing animations for our player character in Godot.

Creating the Jump Animation

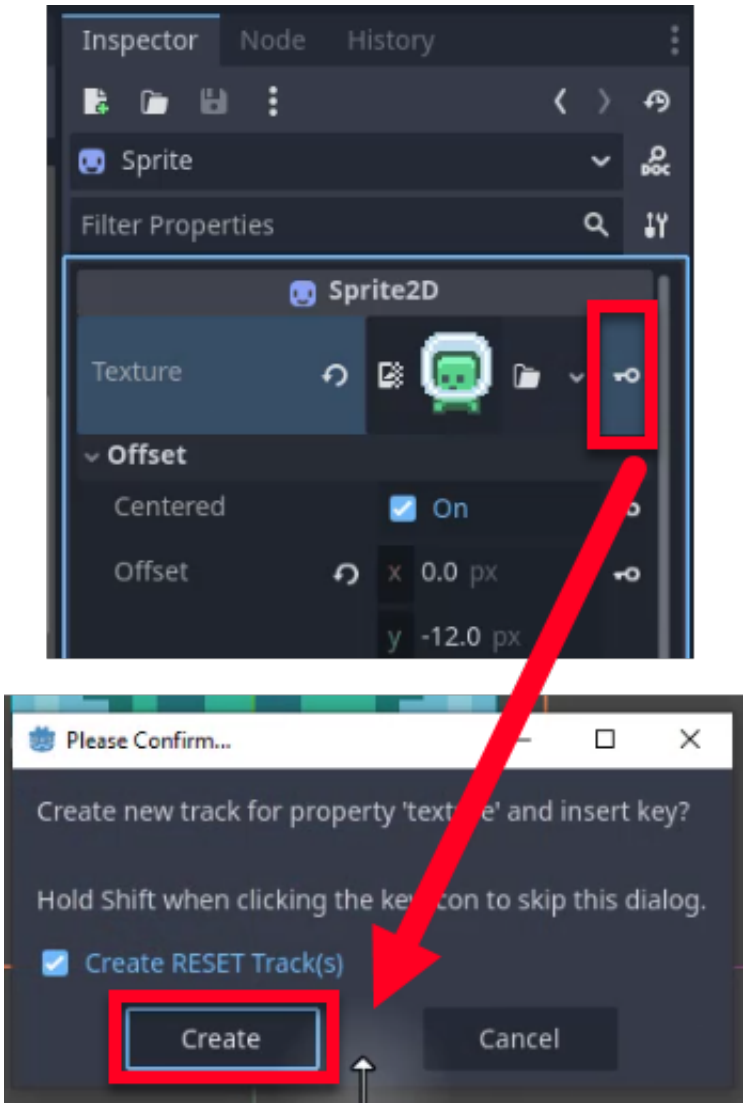
Let's start by creating the jump animation. First, click on the **Animation** button in the Godot editor and select **New**.



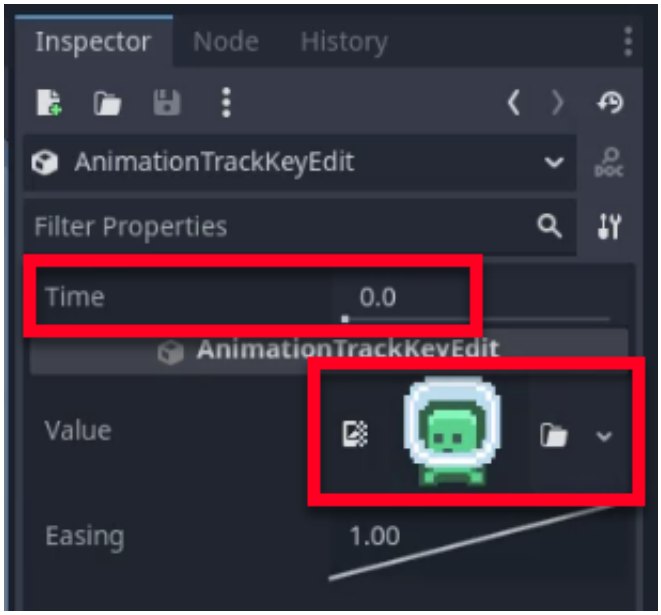
Name the animation jump.



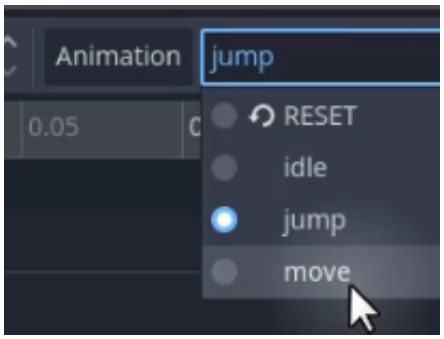
Add a track by selecting our sprite, clicking on the key icon in the Texture property row, and then clicking **Create**.



Select the first keyframe and change its sprite to the one with its legs in the air.



Once done, ensure that you have all three animations: idle, jump, and move.



Next, we will script the logic to play these animations based on the player's state.

Scripting the Animation Logic

In the player.gd script, create a new function called _manage_animation.

Here is the code for the _manage_animation function:

```
func _manage_animation():
    if not is_on_floor():
        anim.play("jump")
    elif move_input != 0:
        anim.play("move")
    else:
        anim.play("idle")
```

This function checks the player's state and plays the appropriate animation:

- If the player is not on the floor (i.e., jumping or falling), it plays the jump animation.
- If the player is moving, it plays the move animation.
- Otherwise, it plays the idle animation.

Adding the AnimationPlayer Reference

To reference the AnimationPlayer node, add the following variable at the top of your script:

```
@onready var anim : AnimationPlayer = $AnimationPlayer
```

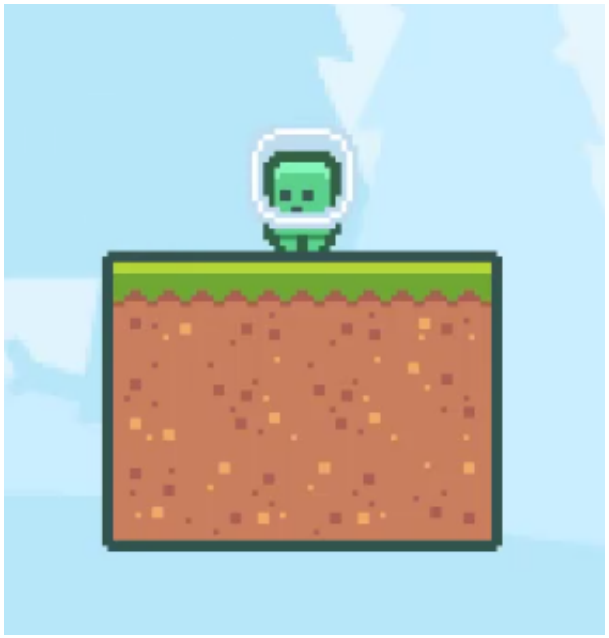
Calling the Animation Function

Finally, call the _manage_animation function inside the _process function to ensure the animations are updated every frame:

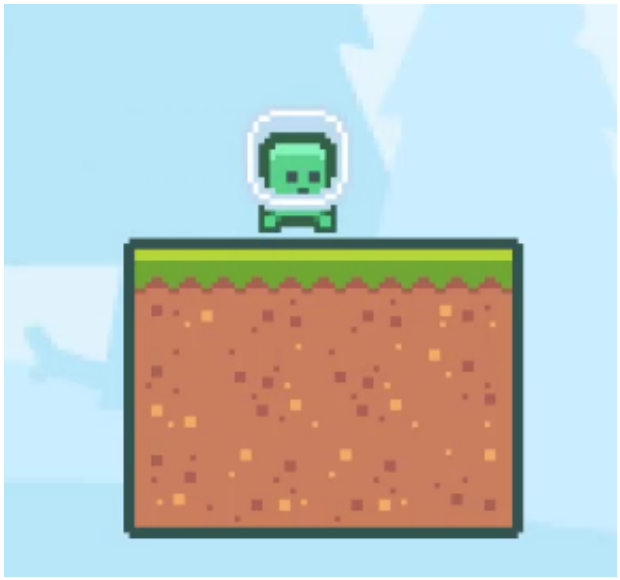
```
func _process(delta):
    if velocity.x != 0:
        sprite.flip_h = velocity.x > 0

    _manage_animation()
```


Testing the Game

Now, let’s test the game. Go back to your level scene and press the play button. Observe the player character. By default, it should display the idle animation.



When you move the character, it should play the move animation.



When you jump, the jump animation should play.

If everything is working correctly, your player character should seamlessly transition between the idle, move, and jump animations based on its state. This adds a layer of realism and engagement to your game.

In the next lessons, we will move on from our player and start working on our enemy obstacles.

Godot 2D Platformer - Player [2025]

1. What is the purpose of the _physics_process function in Godot?

a. It runs once when the game starts
b. It is used to load assets into the scene
c. It toggles whether physics runs or not in the game
d. It processes physics-related updates at a fixed rate per second

2. What does Input.get_axis(“move_left”, “move_right”) return when no movement key is pressed?

a. 0
b. 1
c. -1
d. It depends on the player's last movement direction

3. What function moves the player and handles collision detection?

a. run()
b. walk()
c. move_and_slide()
d. jump()

4. What condition must be met for the player to jump?

a. The player must be moving
b. The player must be in the air
c. The jump key must be pressed and the player must be on the floor
d. The move speed must be greater than zero

5. How does linear interpolation (lerp) improve movement?

a. It makes the character move instantly
b. It smooths out acceleration and braking
c. It increases jump height
d. It applies gravity more effectively

6. What does sprite.flip_h = velocity.x > 0 do?

a. Flips the sprite vertically
b. Makes the sprite invisible when moving left
c. Rotates the sprite 180 degrees
d. Flips the sprite horizontally based on movement direction

7. True or False: The AnimationPlayer node is used to handle physics calculations.

a. True
b. False

8. True or False: We can manage animations so the correct one is played based on the player’s state.

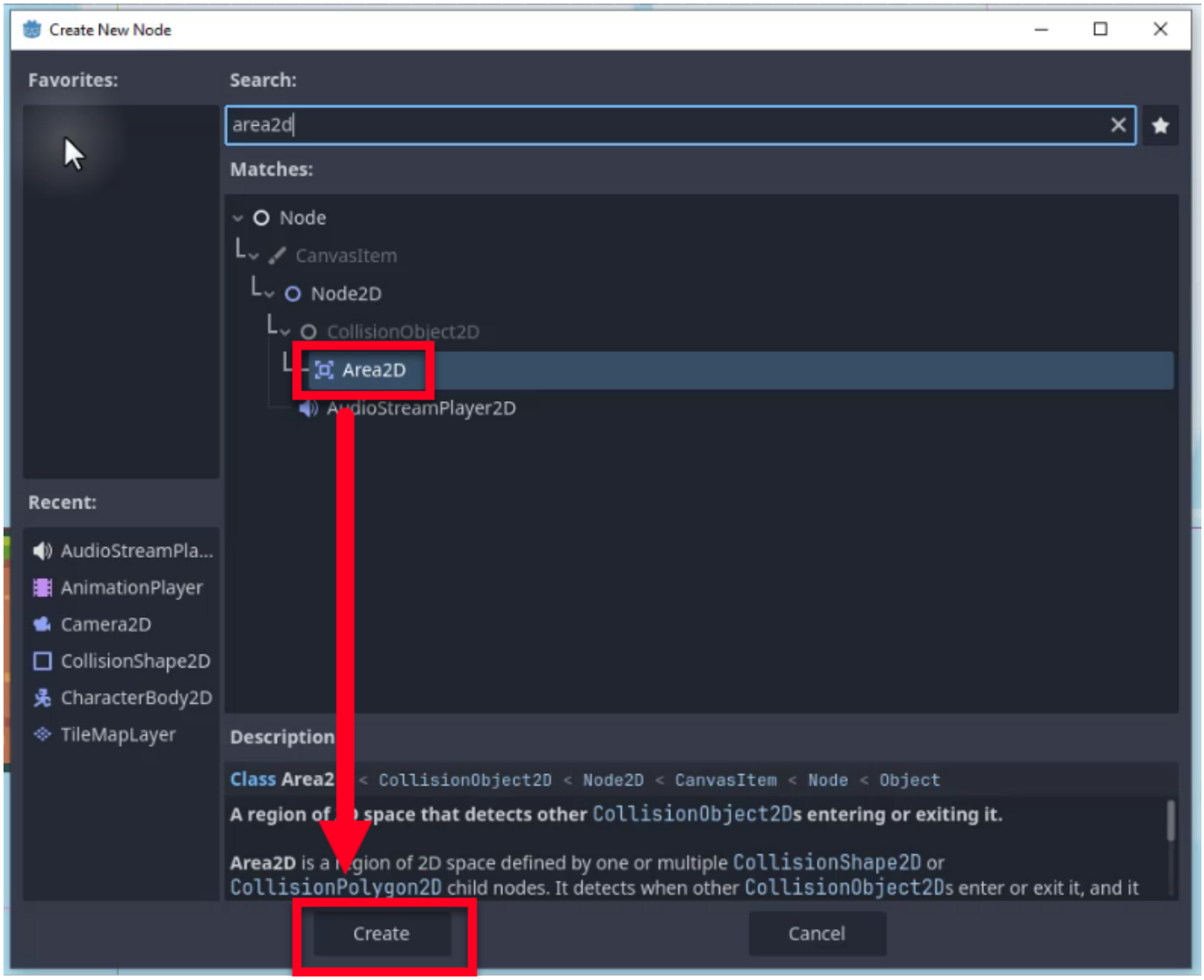
a. True
b. False

In this lesson, we will begin creating our enemy character. The enemy will move from one point to another in a back-and-forth motion. The player's objective will be to avoid these enemies. If the player collides with an enemy, they will take damage. Let's dive into the steps to create our enemy's root node and set up its movement.

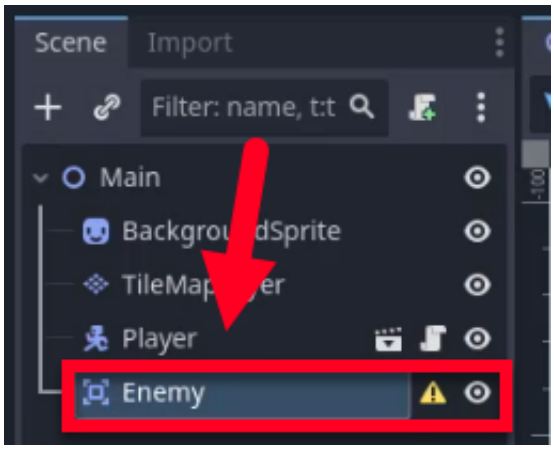
Creating the Enemy's Root Node

The enemy's root node will be of type Area2D. An Area2D node has a collider and can detect collisions, but it does not physically collide with other objects. This allows the player to pass through the enemy while still detecting the collision and dealing damage.

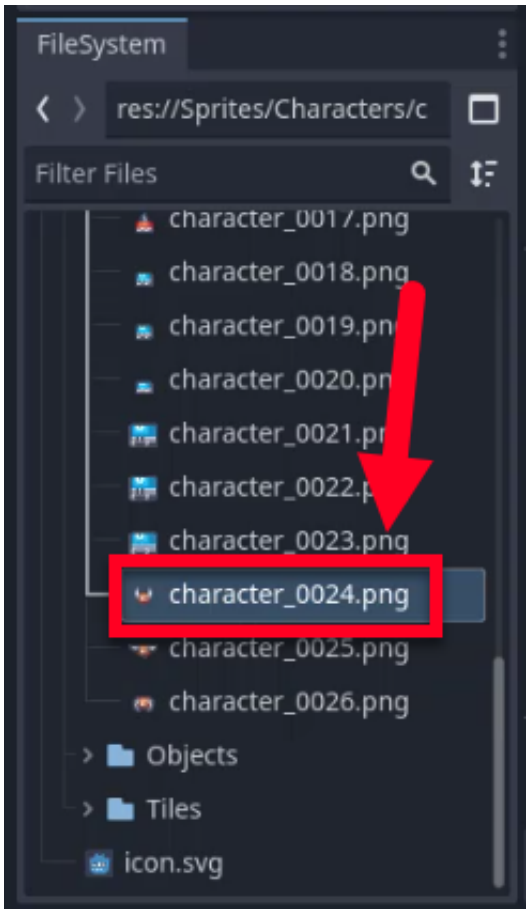
First, let's create the new Area2D node itself.



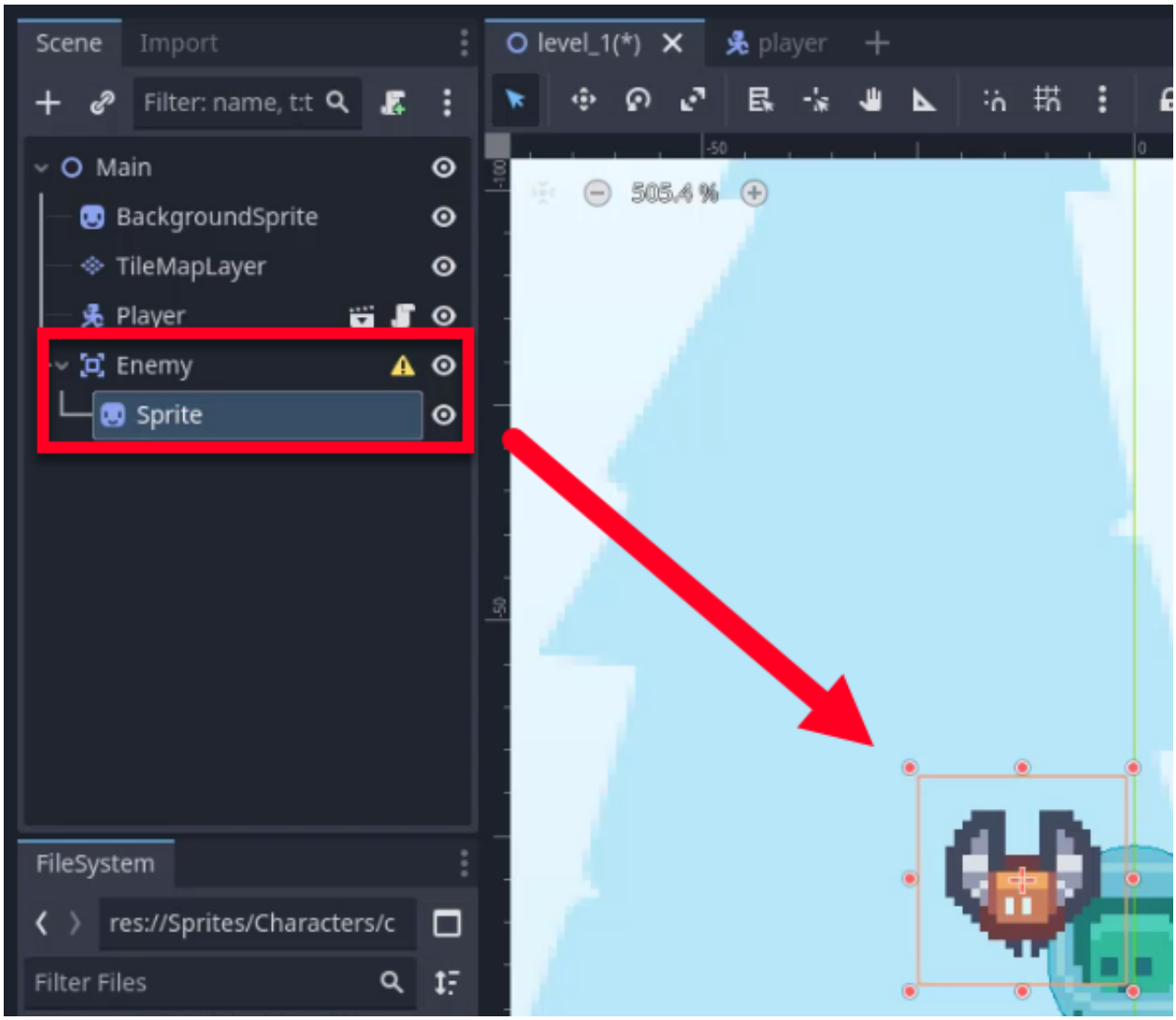
Rename this new node to Enemy.



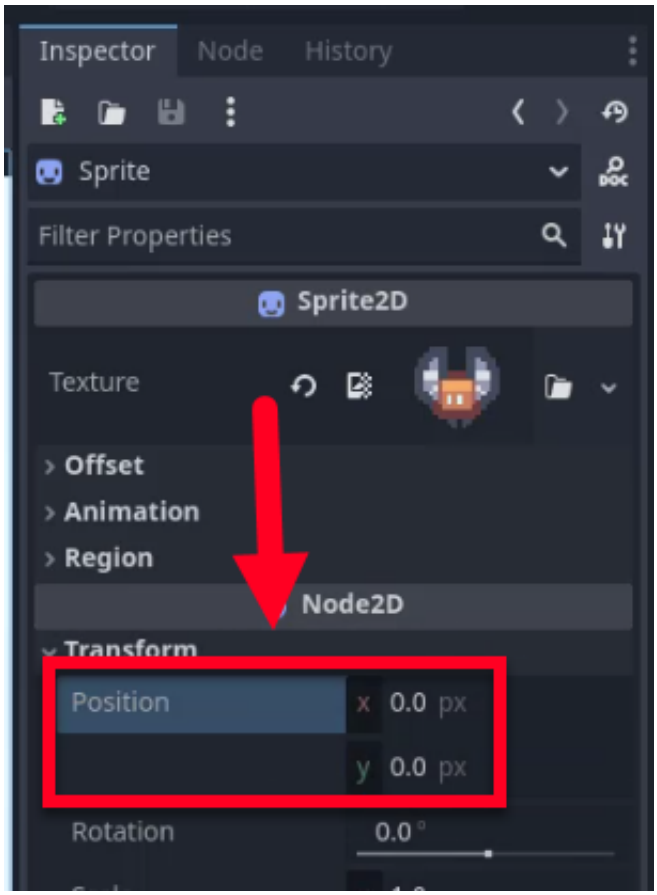
Next, we'll add our character sprite. Go to the Objects folder, navigate to the Characters folder, and find the bat sprite.



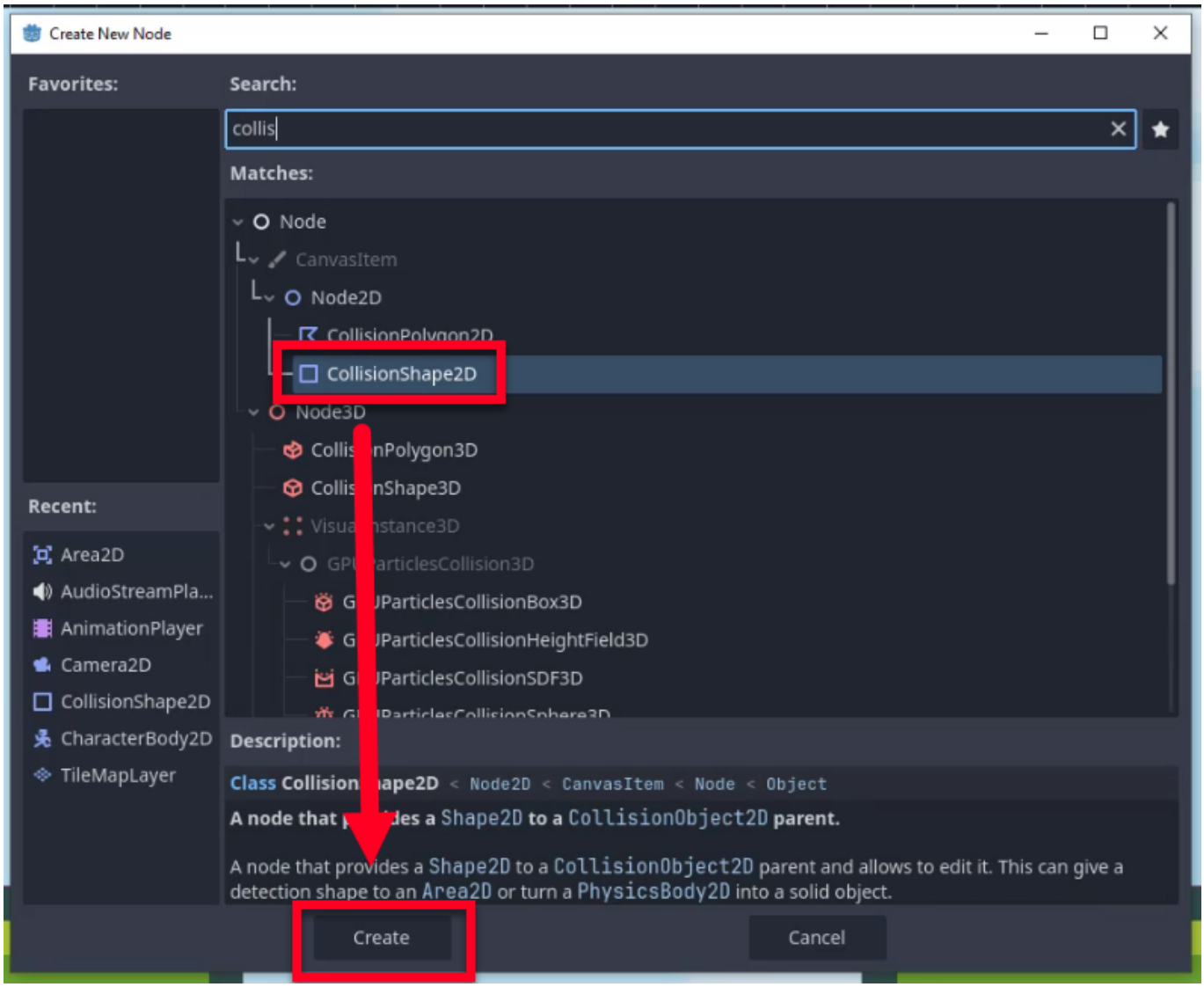
Drag the bat sprite into the Enemy node and rename it to Sprite.



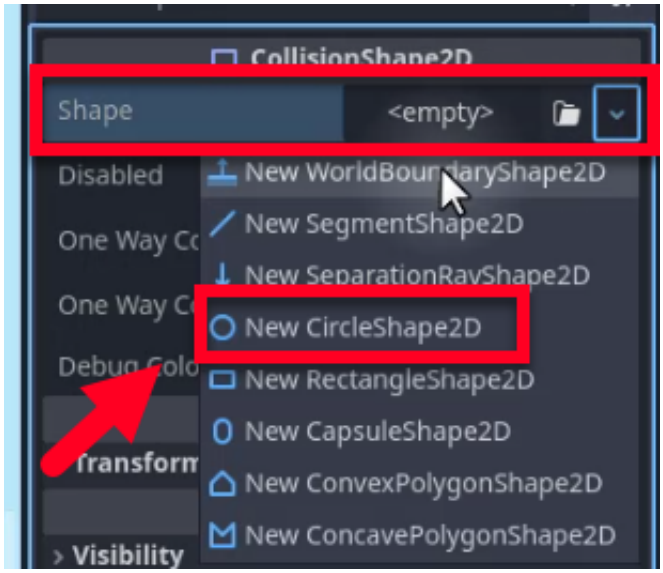
To prevent any weird positioning, with the sprite selected, set its position to (0, 0) in the Inspector.



In order for our node to work, we need to add a collision shape to it. Right-click on the Enemy node and add a child node of type CollisionShape2D.

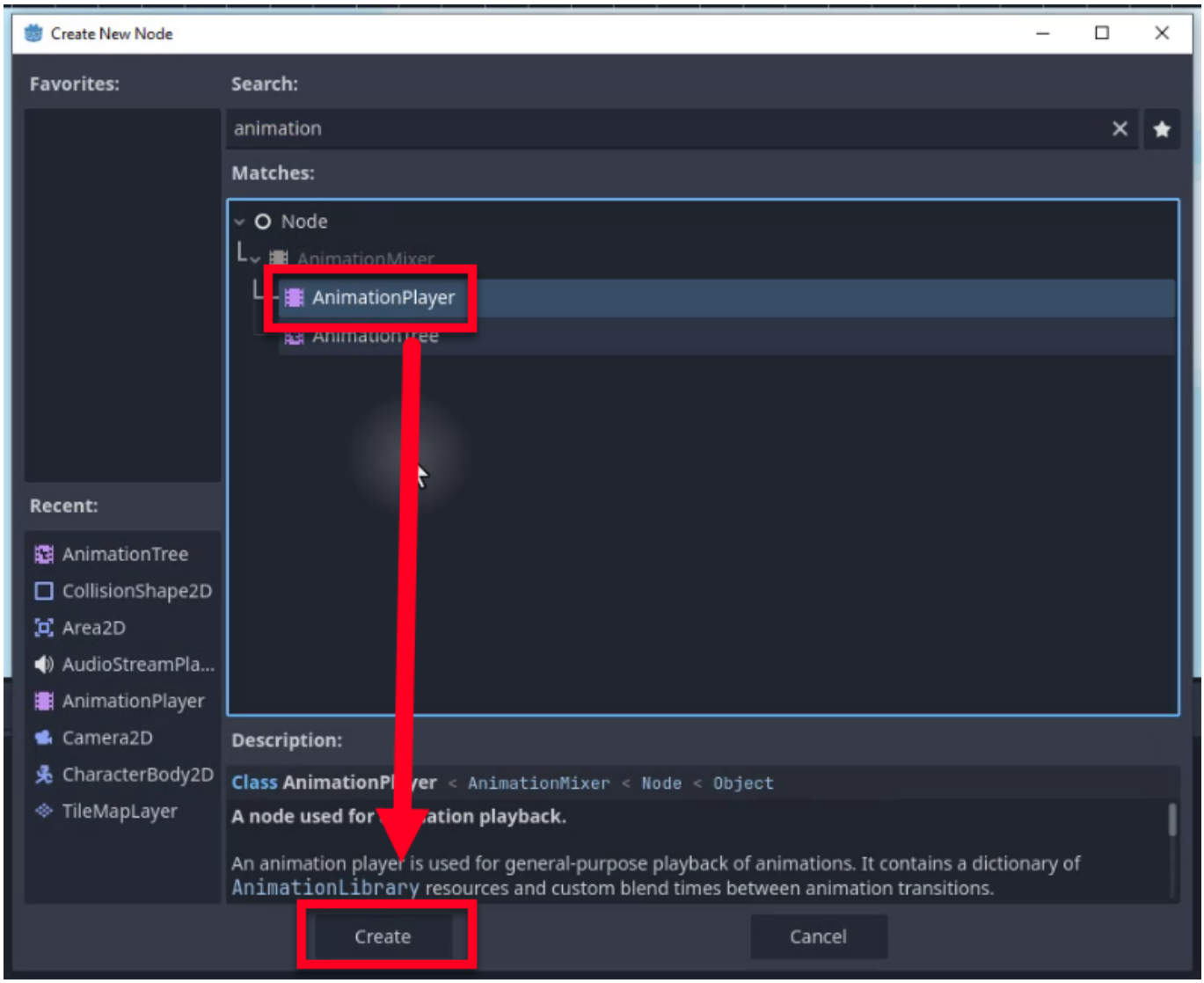


Set the collision shape to be a circle.

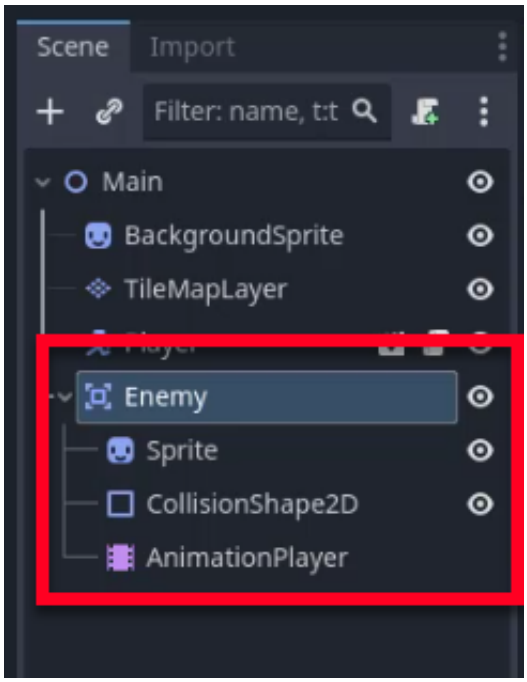


Next, adjust the size to give the player some leeway in dodging.

Finally, as we want this enemy to be animated, we'll add an animation player. Right-click on the Enemy node and add a child node of type AnimationPlayer.

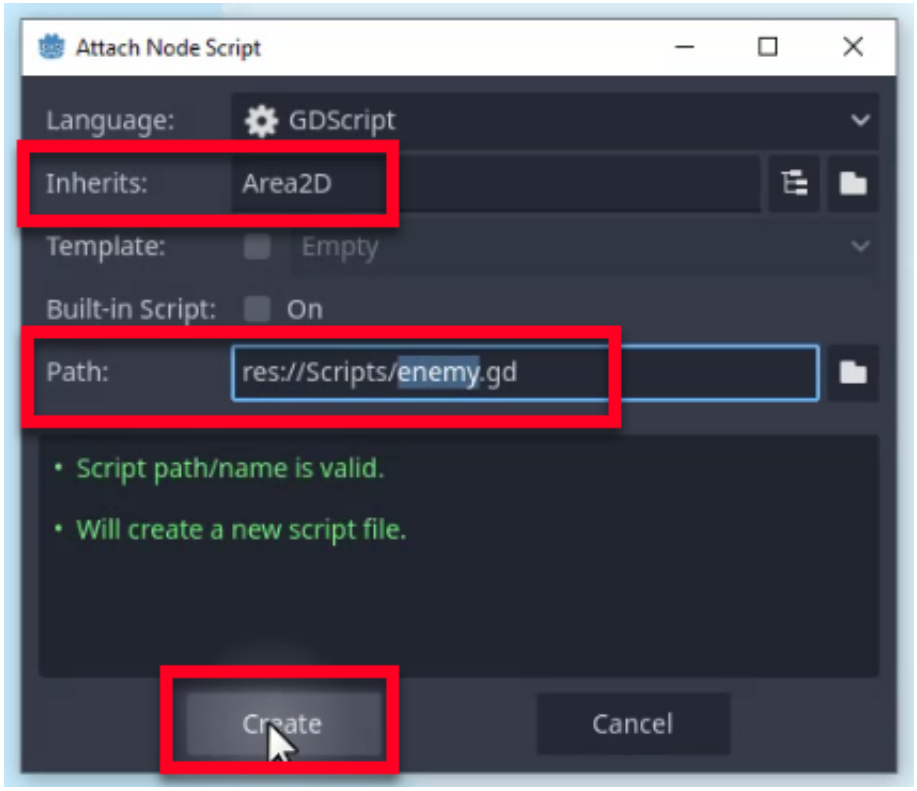


Now the foundations of our Enemy scene are set up!



Creating the Enemy Script

Now that we have set up the enemy's node structure, let's create a script to control its movement. We will define variables to manage the enemy's movement direction, speed, and positions. To begin, ,create a new script on the Area2D node and name it enemy.gd.



Next, define the variables described above:

```
extends Area2D
```



```
@export var move_direction : Vector2
@export var move_speed : float = 20

@onready var start_pos : Vector2 = global_position
@onready var target_pos : Vector2 = global_position + move_direction
```

Detecting Target Position

To make the enemy move back and forth between two points, we need to detect when the enemy has reached its target position and then change the target position accordingly:

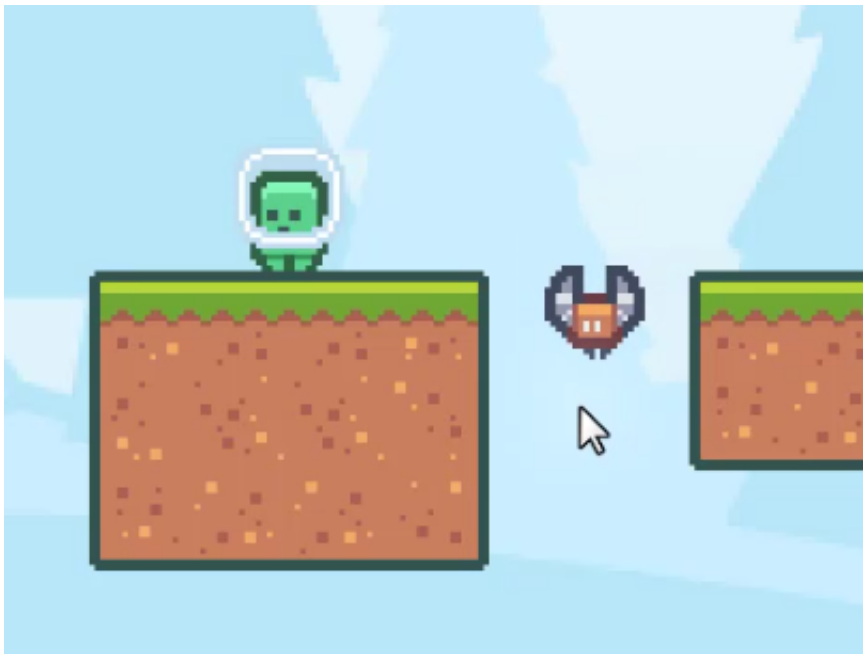
```
func _physics_process(delta):
  global_position = global_position.move_toward(target_pos, move_speed * delta)

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

In the _physics_process function, which is called every frame by Godot, the enemy's global_position is updated to move towards the target_pos at a speed of move_speed multiplied by delta. delta is a built-in Godot variable that represents the time elapsed since the last frame, which helps to make the movement smooth and frame-rate independent.

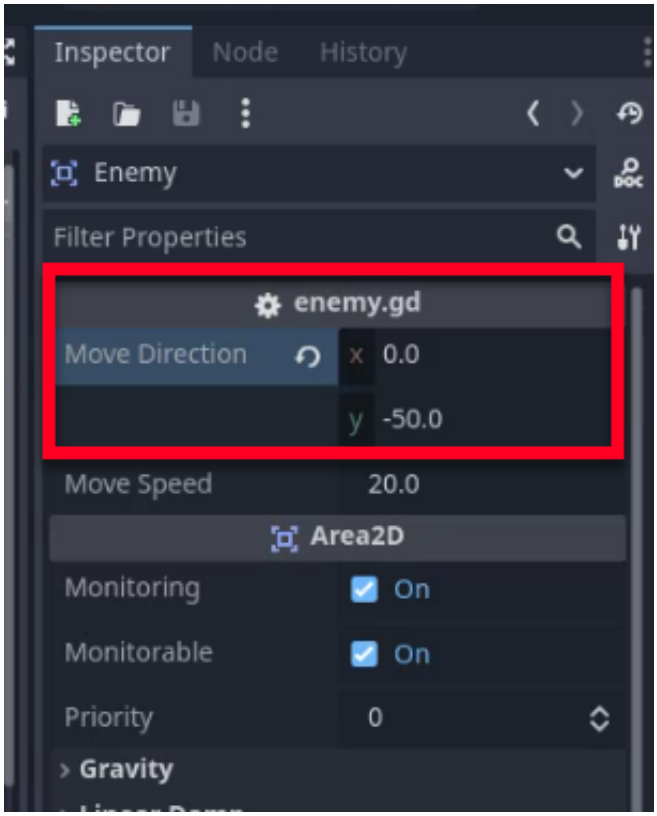
The code then checks if the enemy has reached the target_pos. If it has, it checks if the target_pos is equal to the start_pos. If it is, the enemy's target_pos is updated to be the start_pos plus the move_direction, effectively moving the enemy in the specified direction. If the target_pos is not equal to the start_pos, the enemy's target_pos is reset to the start_pos, causing the enemy to move back to its starting position.

Testing the Enemy Movement

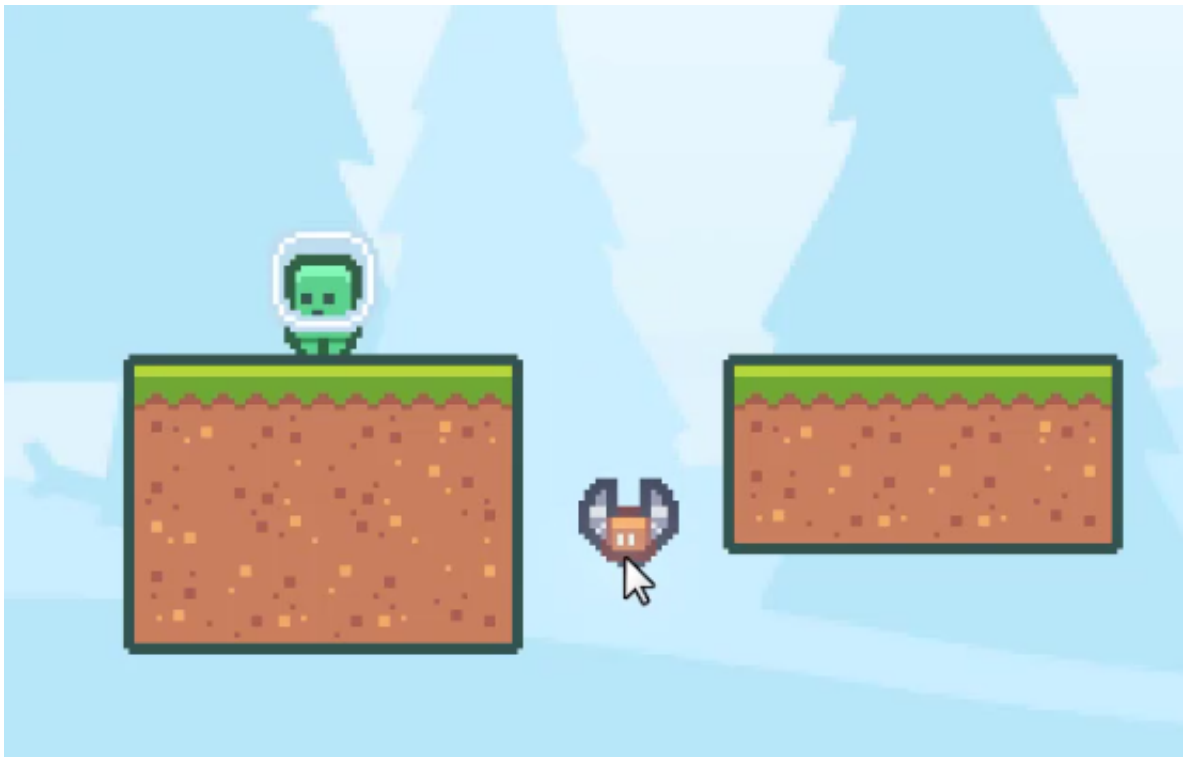
Let's test the enemy movement to ensure it works as expected. Select the enemy node in the 2D mode and move the enemy to a starting position.



In the Inspector, set the move_direction to (0, -50) to make the enemy move up and down.



Press play to see the enemy move!

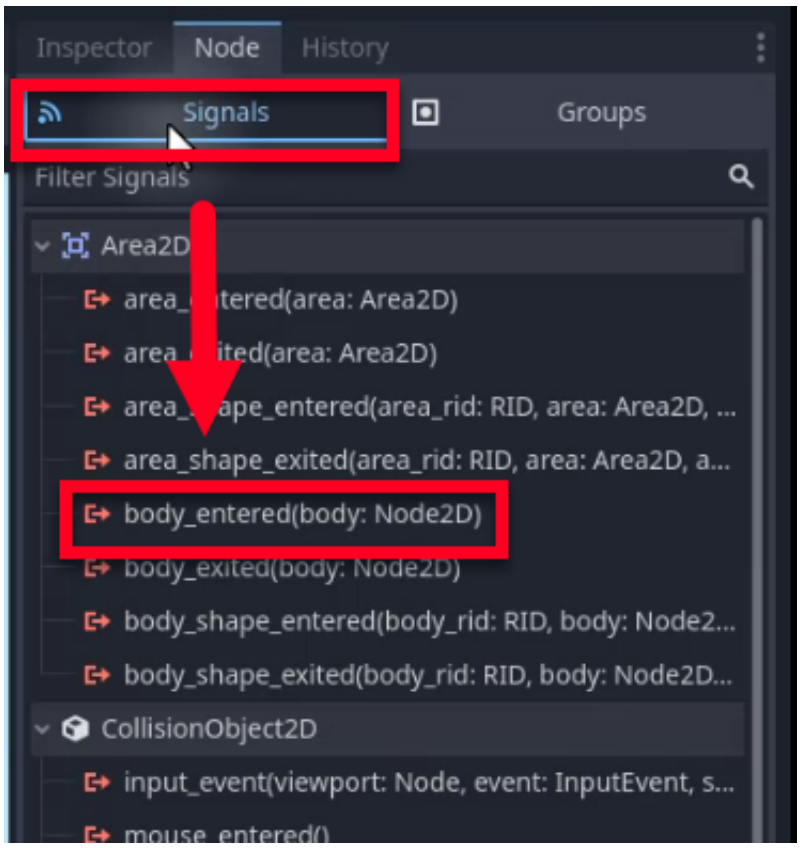


That’s it for this lesson! In the next lesson, we will focus on improving our enemy a bit more.

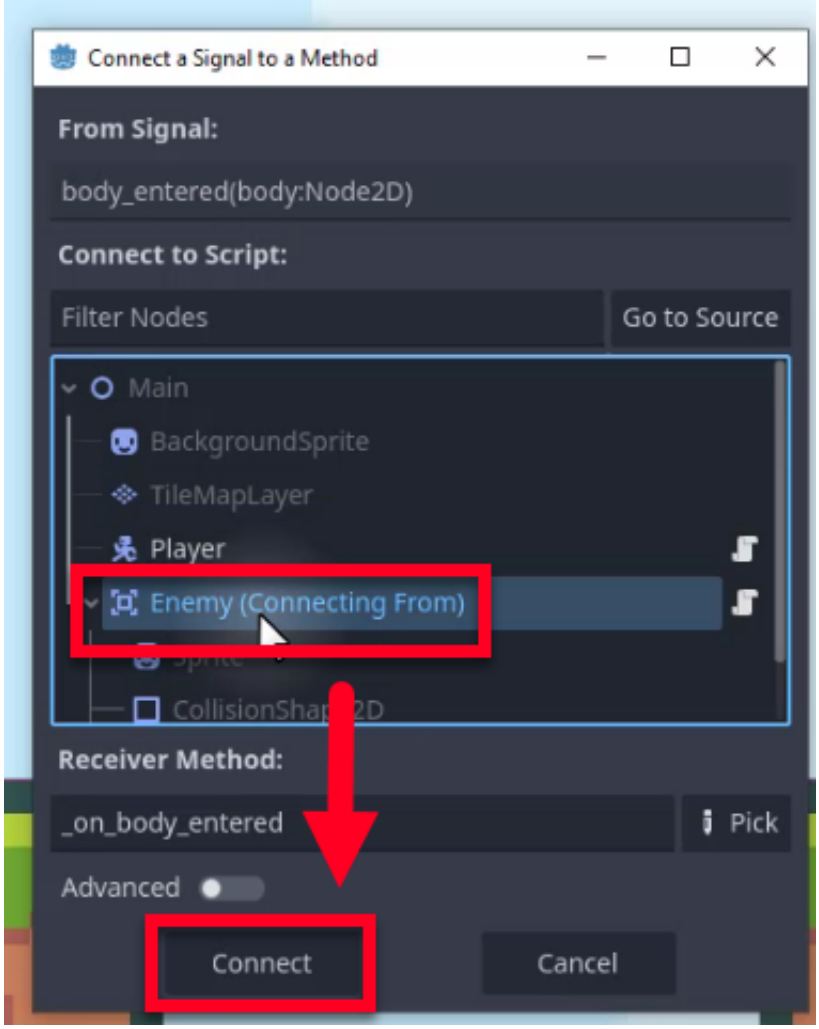
In this lesson, we will implement the functionality for the enemy to detect the player. Specifically, the enemy will detect when the player's collider overlaps or enters the enemy's collider. When this happens, we want certain actions to occur, such as the player taking damage. To achieve this, we will use signals in Godot and write some script to handle the collision detection.

Setting Up the Enemy Node

First, let's connect a signal from our enemy to our script. Select the enemy node, which is an Area2D. With the enemy selected, go to the Node panel and click on the "Signals" tab. Look for the body_entered signal. This signal is triggered when a body enters the enemy's collider.



Double-click on the body_entered signal and connect it to the enemy node.



This will automatically add a new method to our script that connects to this built-in Godot signal.

Writing the Collision Detection Script

Now, let's write the script to handle the collision detection. In the enemy script, add the _on_body_entered functionality to handle the collision detection:

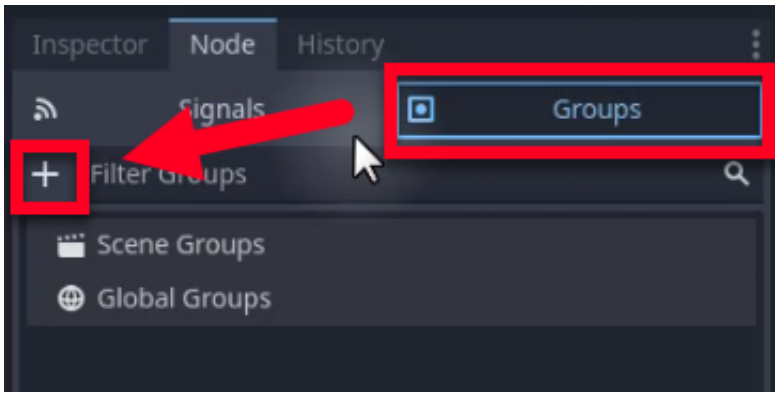
```
func _on_body_entered(body):
    if not body.is_in_group("Player"):
        return

    print("Deal damage to player")
```

In this script:

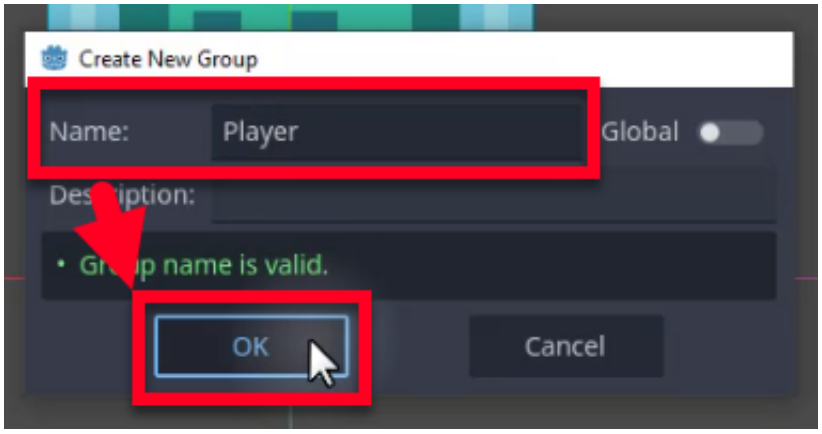
- The _on_body_entered function is called when a body enters the enemy's collider.
- We check if the body that entered is in the "Player" group using body.is_in_group("Player").
- If the body is not the player, we return and do nothing.
- If the body is the player, we print "Deal damage to player" to the output. This is a placeholder for where we will eventually deal damage to the player.

Assigning the Player to a Group

To ensure that the enemy can identify the player, we need to assign the player to a group. Select the player node, go to the Node panel, and click on the “Groups” tab. Click the plus icon to create a new group.



Name the group “Player” and click OK.



Now, the player node is assigned to the “Player” group, and our enemy script can check for this group to identify the player.

Testing the Collision Detection

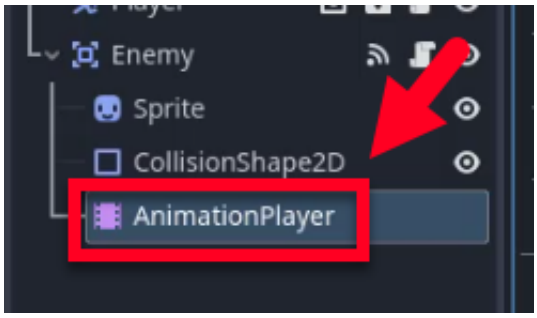
Let’s test our collision detection:

1. Press the play button to run the scene.
2. Move the player to collide with the enemy.
3. Check the output to see if “Deal damage to player” is printed each time the player collides with the enemy.

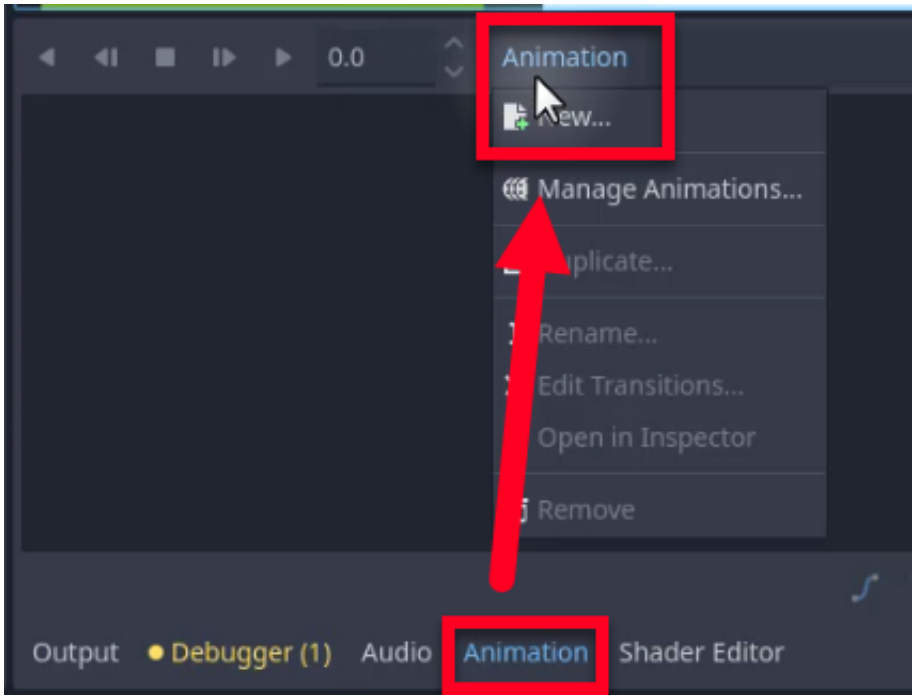
If you see the message in the output, the collision detection is working correctly.

Animating the Enemy

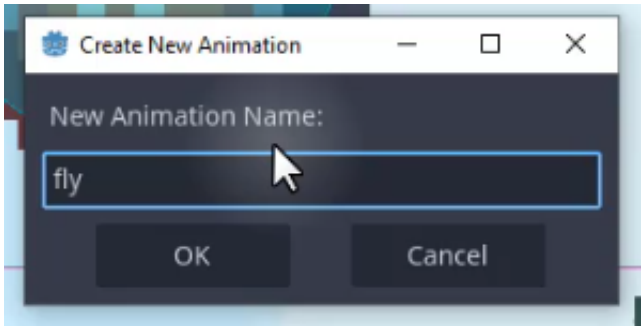
Now that our enemy can detect the player, let’s add some animation to make the enemy more dynamic. We will create a simple flying animation for the enemy. Select the enemy node and then select the AnimationPlayer node.



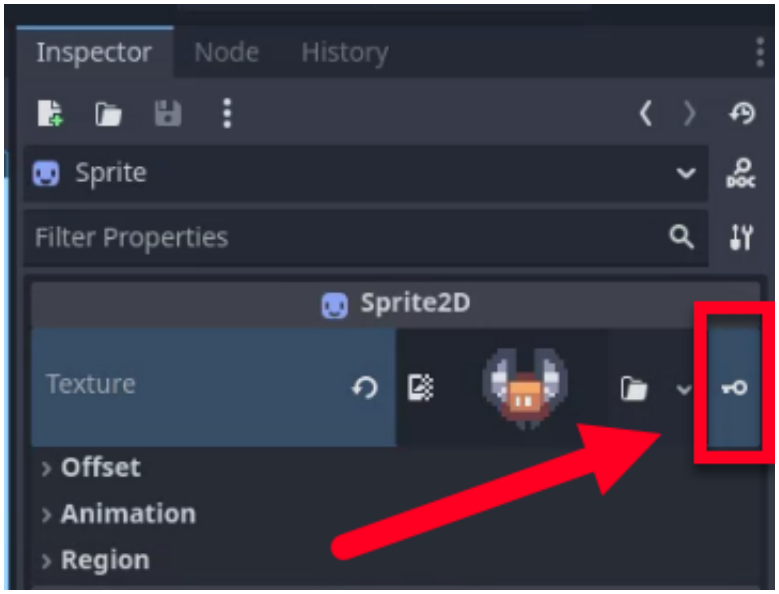
Go to the Animation panel, click on the Animation button, and select “New” to create a new animation.



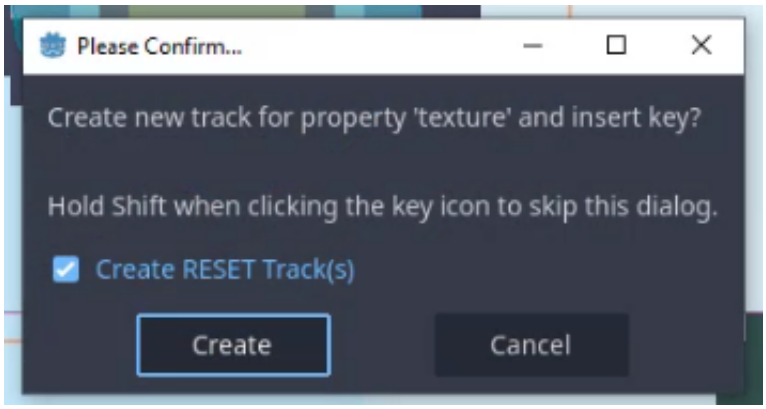
Name the animation “fly” and hit OK to create the animation.



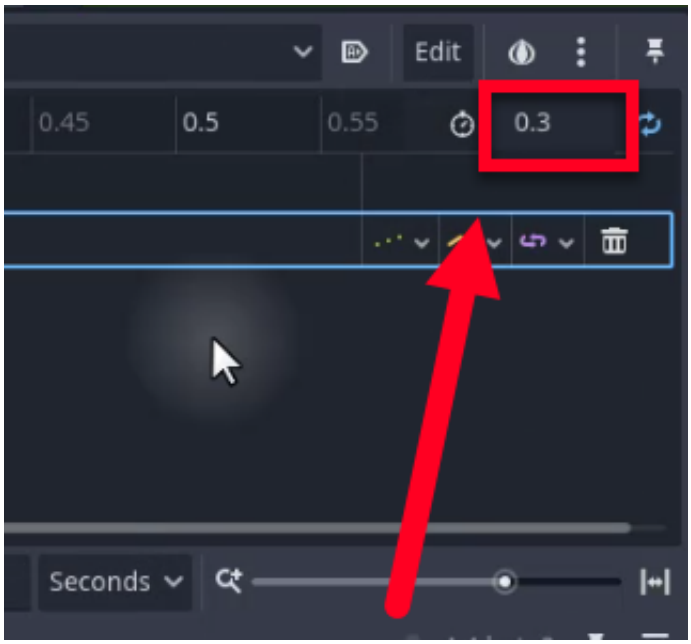
Next, like with the player, we want to set up a track for the Sprite’s texture. You can do this by selecting the bat sprite and hitting the key icon next to the Texture property field.



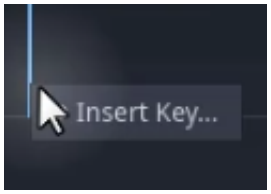
In the pop-up that appears, hit Create – and remember to leave “Create RESET Track(s)” checked.



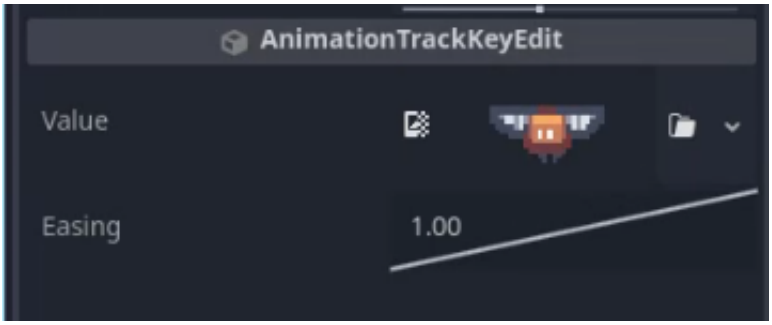
By default, animations are set to 1 second. We don’t need that long of an animation, so set the duration to 0.3.



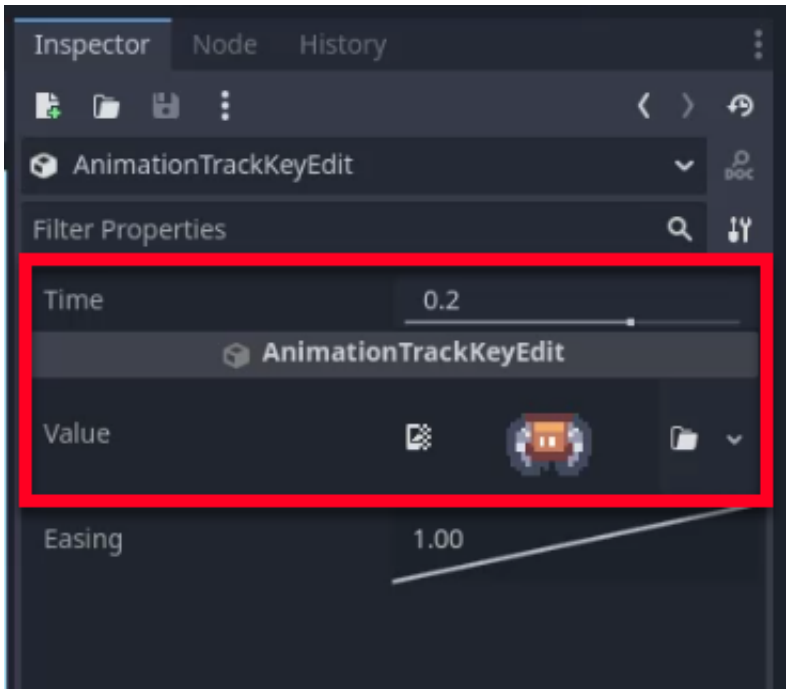
Now we can create the animation itself. Our keyframe at 0 seconds is already set, as we want the sprite with wings up to begin the animation. Move the timeline to about 0.1 seconds and create a new keyframe. You can do so by right-clicking on the track and selecting “Insert Key”…



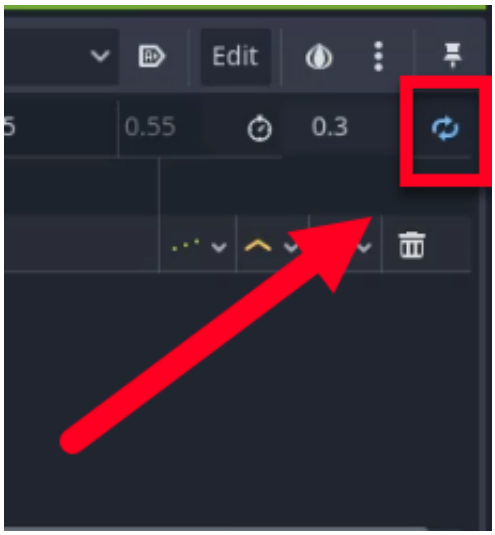
With the keyframe selected, swap out the character texture so it’s the bat sprite with its wings horizontal in the Inspector.



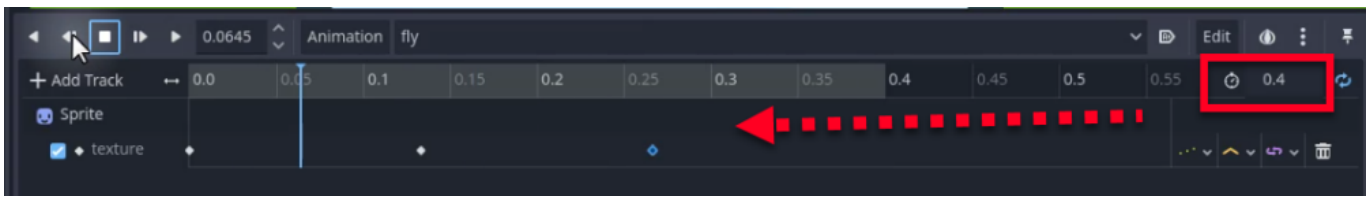
Repeat these steps, adding a new keyframe at around 0.2 seconds and changing the texture to the bat sprite with its wings down.



Last, we need to turn on looping for this animation by using the button near the upper right of the timeline.



Now you can play the animation and see it in action. Feel free to adjust the duration and timing to your preference. For example, we decided the animation was a little too fast, so we changed the duration to 0.4 and moved the keyframes accordingly.



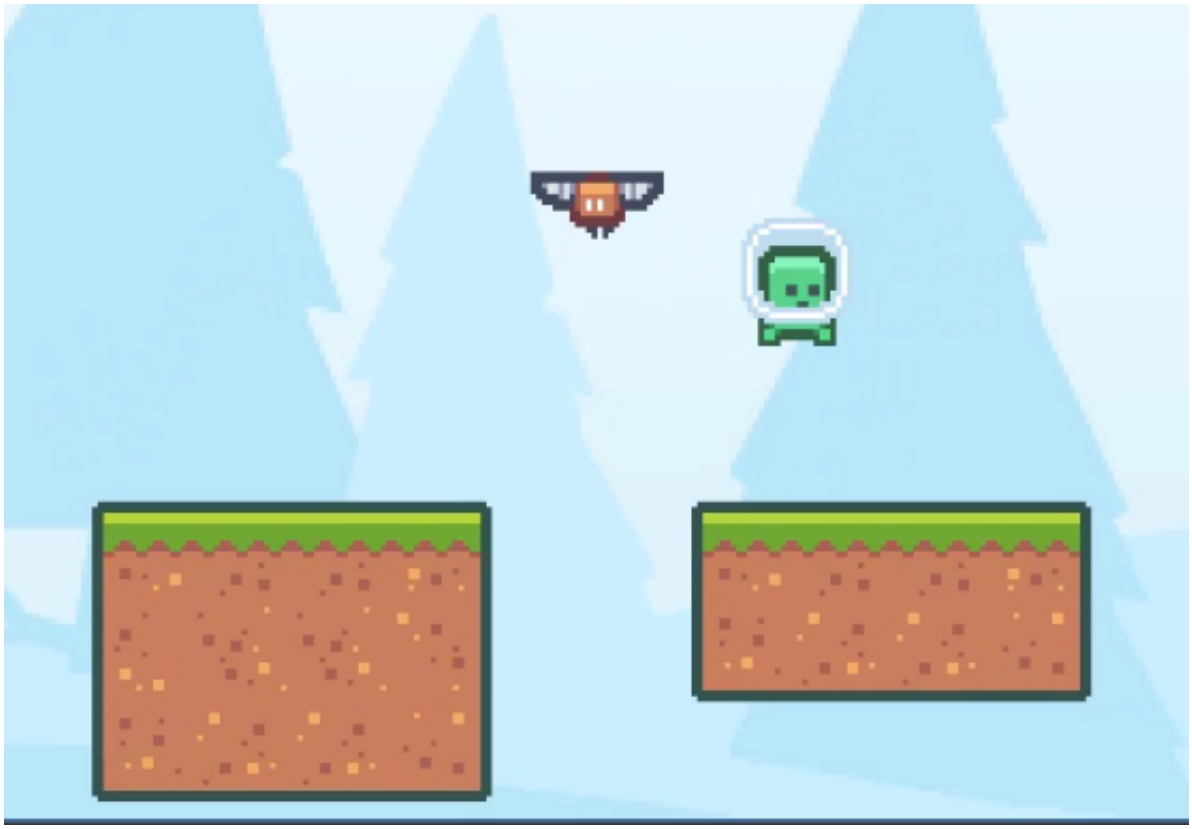
Playing the Animation

To ensure the animation plays when the enemy is in the scene, we need to add some code to our script to play the animation. In the enemy script, add the _ready function to play the animation:

```
func _ready():
    $AnimationPlayer.play("fly")
```

In this script, the _ready function is called when the enemy node is added to the scene. It plays the “fly” animation using the AnimationPlayer.

Now, within the game, you should see the enemy automatically animate as it moves.



Conclusion

In this lesson, we implemented the functionality for the enemy to detect the player and added a simple flying animation to the enemy. In the next lesson, we will implement the ability for the player to take damage and reset the level when the player’s health reaches zero.

In this lesson, we will add the ability for our player to take damage. Once the player's health reaches zero, the scene will reset. Let's dive into the steps required to achieve this functionality.

Setting Up Player Health

First, we need to create a variable to track the player's health. Open the player.gd script and add the following line at the top:

```
@export var health : int = 3
```

This variable, health, is of type int and is initialized to 3. This means the player can take three hits before being defeated.

Creating the Take Damage Function

Next, we will create a function called take_damage that will be called when the player takes damage. This function will reduce the player's health by a specified amount and check if the health has reached zero.

```
func take_damage(amount : int):
    health -= amount
    if health <= 0:
        game_over()
```

In this function:

- amount is the amount of damage taken.
- The player's health is reduced by the amount of damage.
- If the health is less than or equal to zero, the game_over function is called.

Implementing the Game Over Function

The game_over function will reload the current scene. Add the following code to your script:

```
func game_over():
    get_tree().change_scene_to_file("res://Scenes/level_1.tscn")
```

This function uses get_tree().change_scene_to_file to reload the current scene, effectively resetting the game.

Updating the Enemy Script

Now, we need to update the enemy script to call the take_damage function when the player collides with an enemy. Open the enemy.gd script and modify the _on_body_entered function as follows:

```
func _on_body_entered(body):
    if not body.is_in_group("Player"):
        return
    body.take_damage(1)
```

This code checks if the body that entered the enemy's collision area is the player. If it is, it calls the take_damage function on the player, dealing one damage.

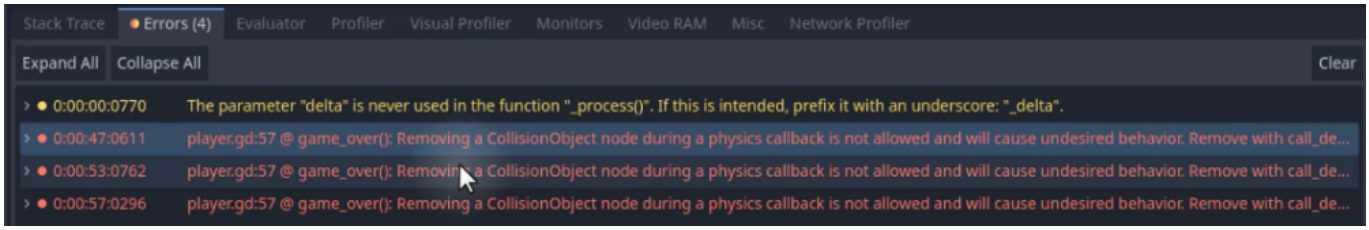
Testing the Implementation

While we won't be able to see the full effect of the decrease in player health yet (as we lack a UI still), we can test it by seeing if we can get the scene to reset.

1. Press play to test the game.
2. Collide with the enemy to decrease the player's health.
3. Once the health reaches zero, the scene should reset.

Fixing the Physics Callback Error

You might encounter an error stating that removing a collision node during a physics callback is not allowed.



This happens because the game_over function is called within the physics process loop. To fix this, we use the call_deferred function to delay the call to game_over until the end of the frame.

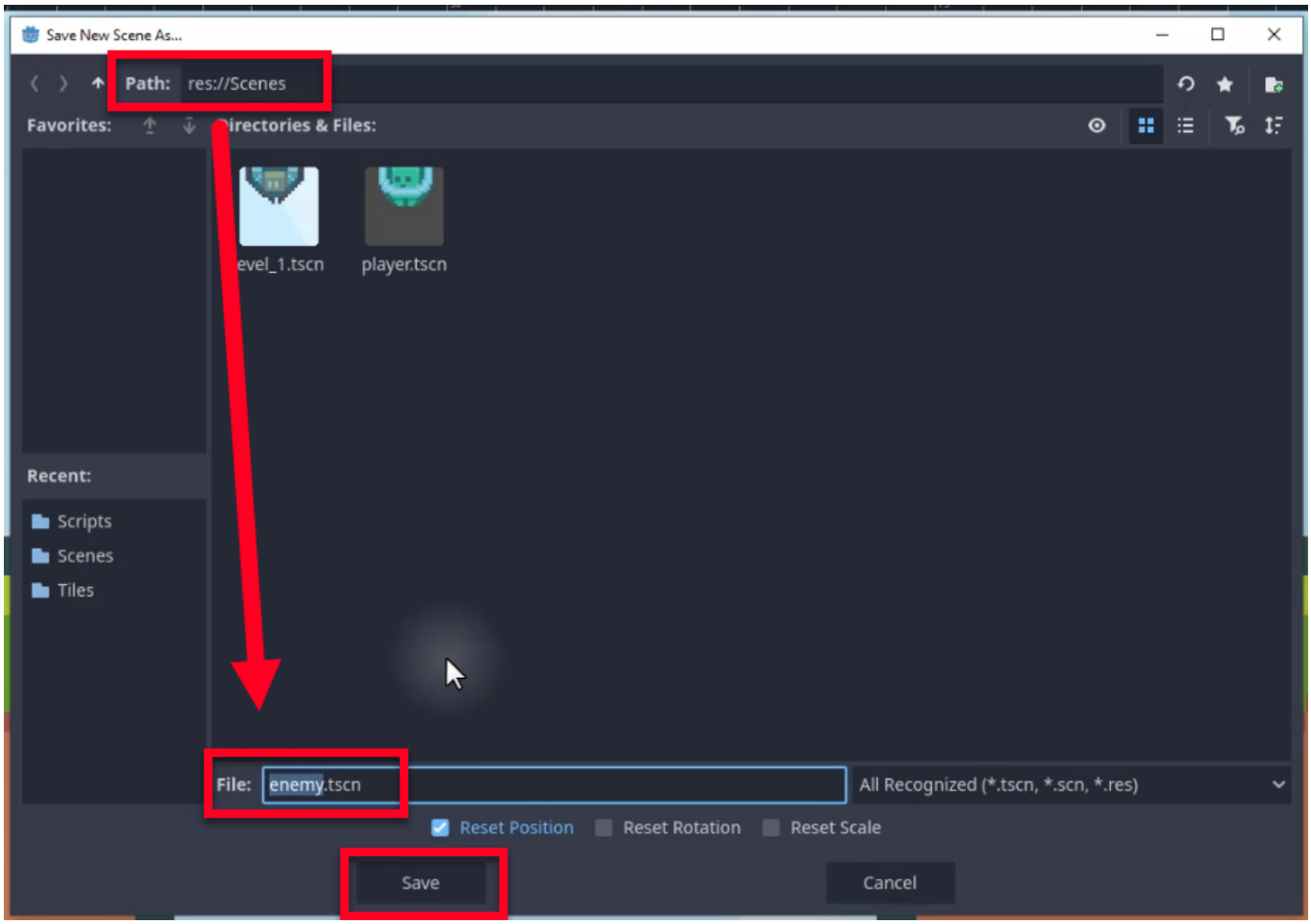
Update the take_damage function as follows:

```
func take_damage(amount : int):
    health -= amount
    if health <= 0:
        call_deferred("game_over")
```

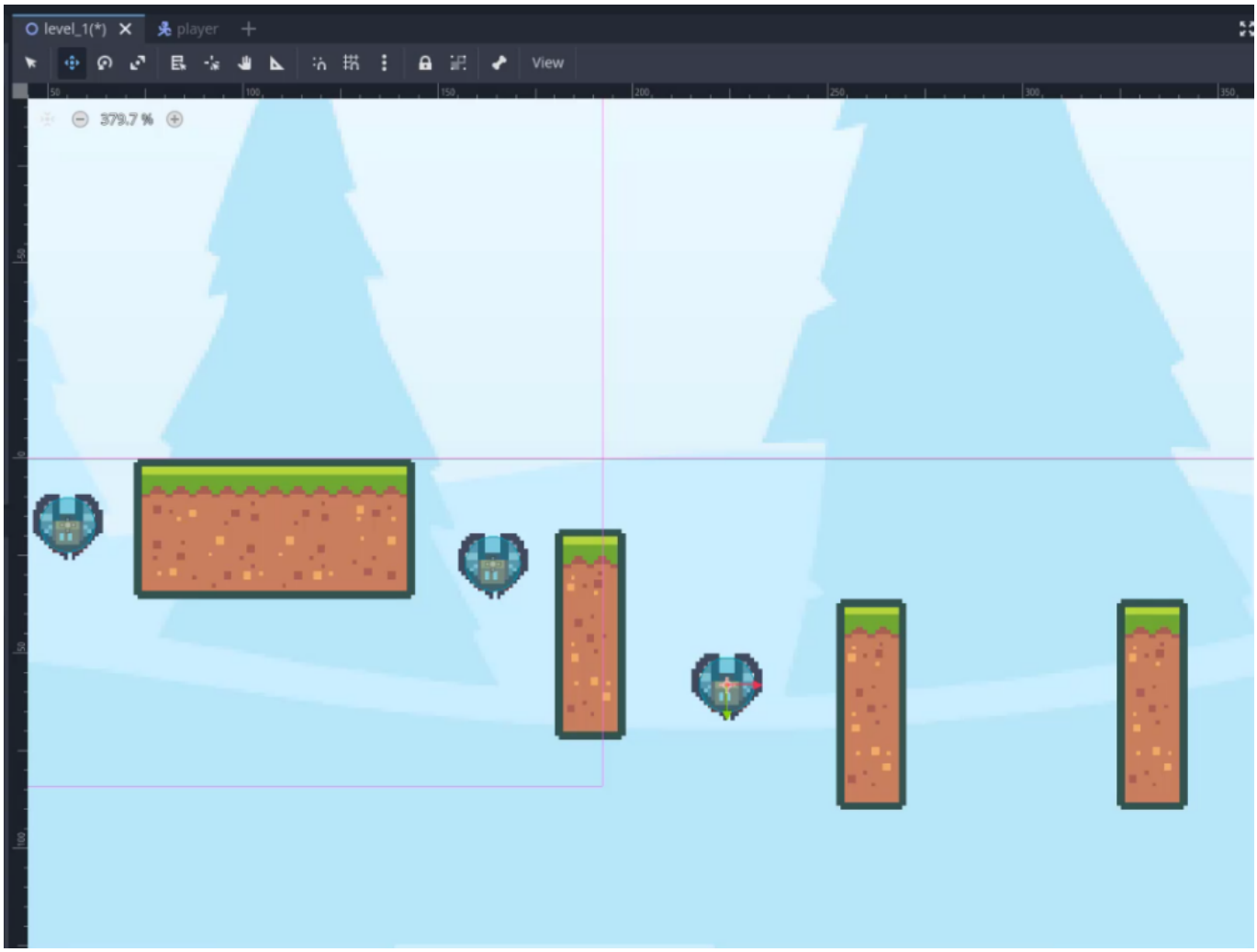
The call_deferred function ensures that game_over is called after the current frame has ended, avoiding the physics callback error.

Saving and Duplicating Enemies

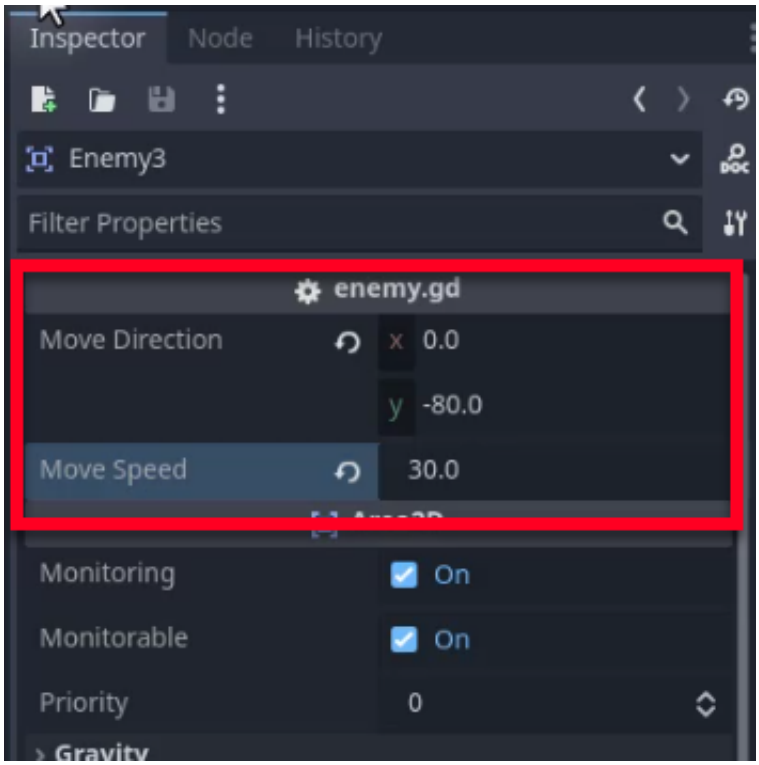
Finally, let's save our enemy as a scene and duplicate it around the level. Open the Scenes folder and drag the enemy into the folder and save it as enemy.tscn.



Now that we can reuse our enemy, duplicate the enemy around the level to create multiple enemies.



Be sure to adjust the properties of the duplicated enemies as needed (e.g., move speed, position). There is no right or wrong answer here, so feel free to get creative in how the enemies behave!

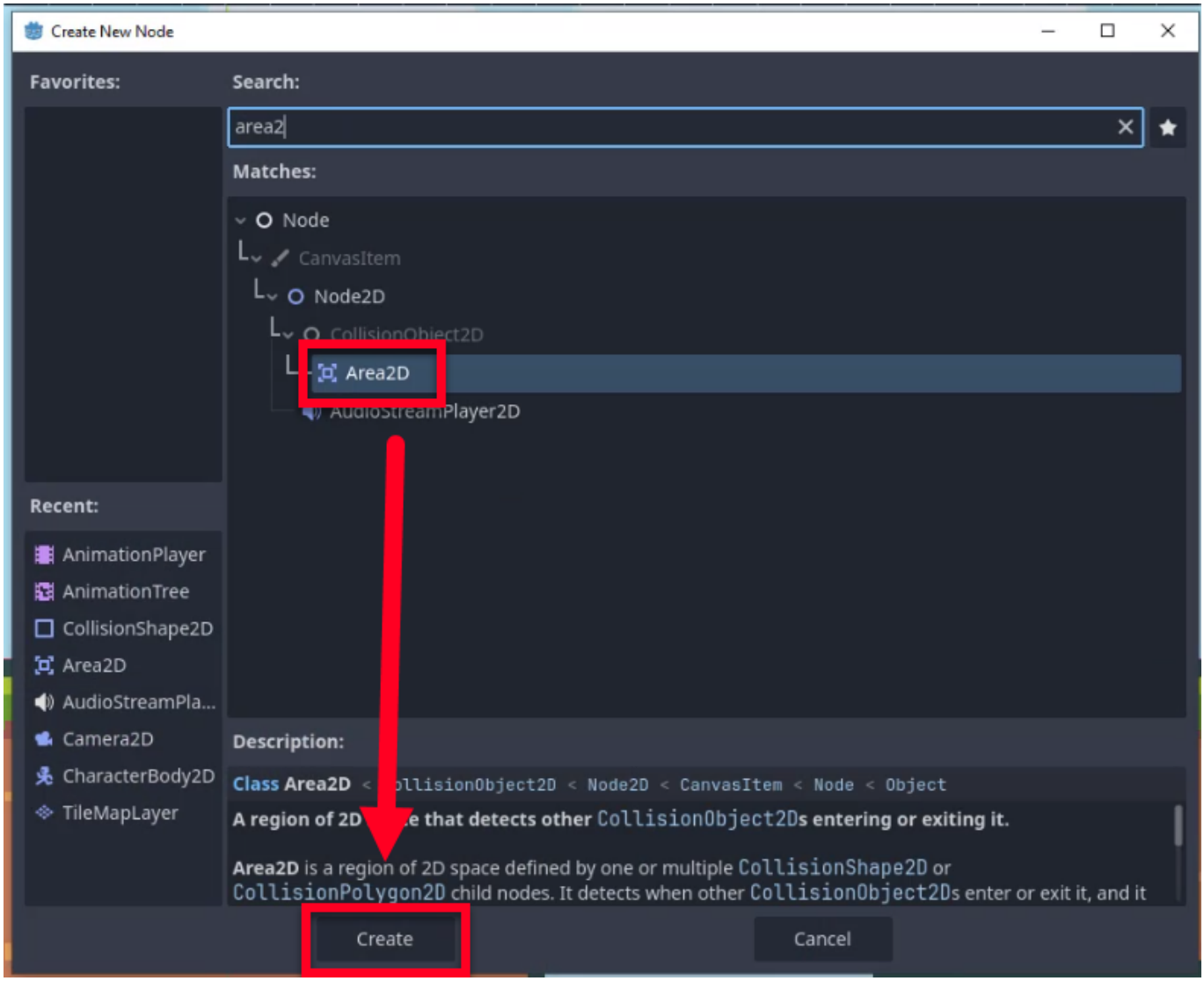


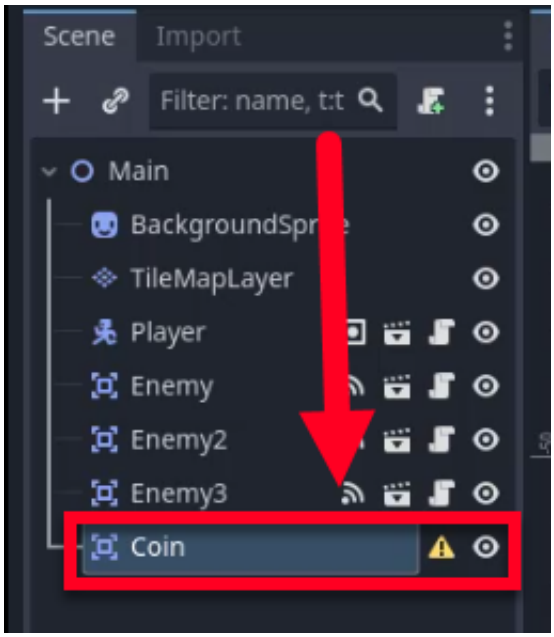
By following these steps, you have successfully implemented the functionality for the player to take damage and reset the scene when their health reaches zero. This enhances the gameplay experience and adds a layer of challenge to your game.

In this lesson, we will focus on creating and animating a coin object for our game. The coin will serve as a collectible item that increases the player's score when collected. We'll start by setting up the coin scene, adding a sprite, and configuring its properties. Then, we'll write a script to make the coin bob up and down and rotate, enhancing its visual appeal.

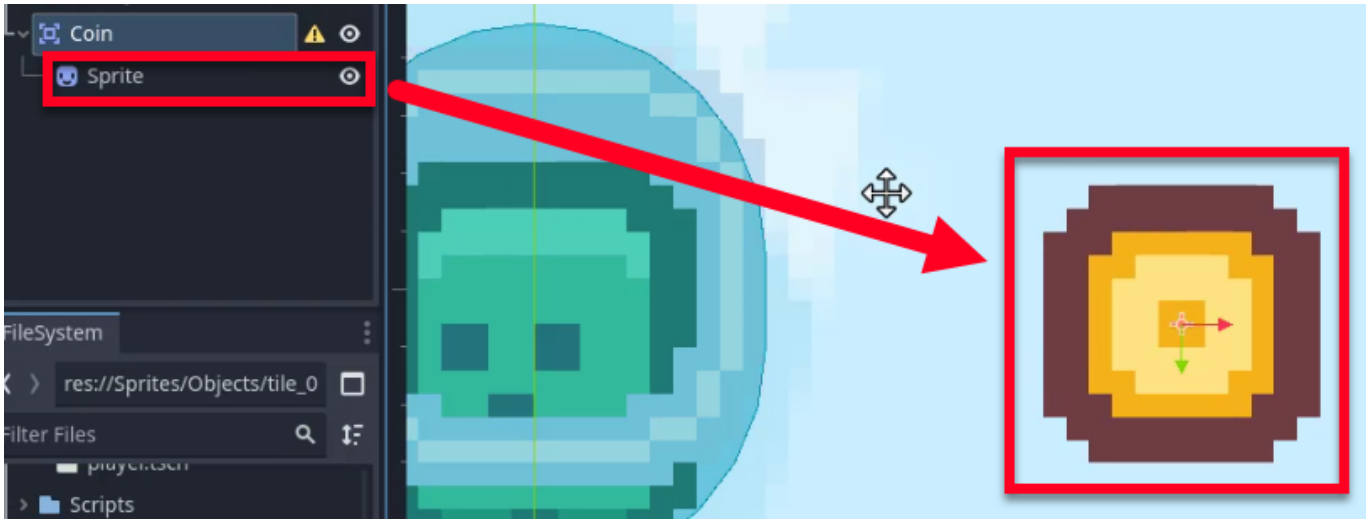
Setting Up the Coin Scene

Let's begin by creating a new scene for the coin. Create a new Area2D node and name it Coin.

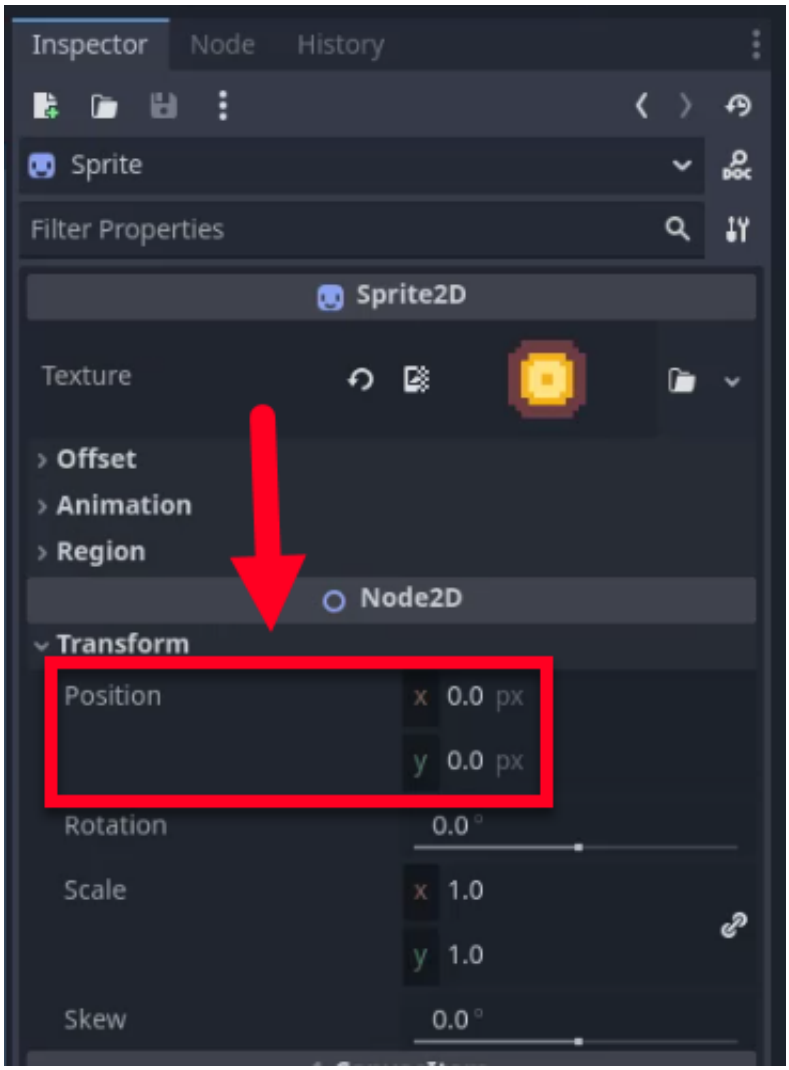




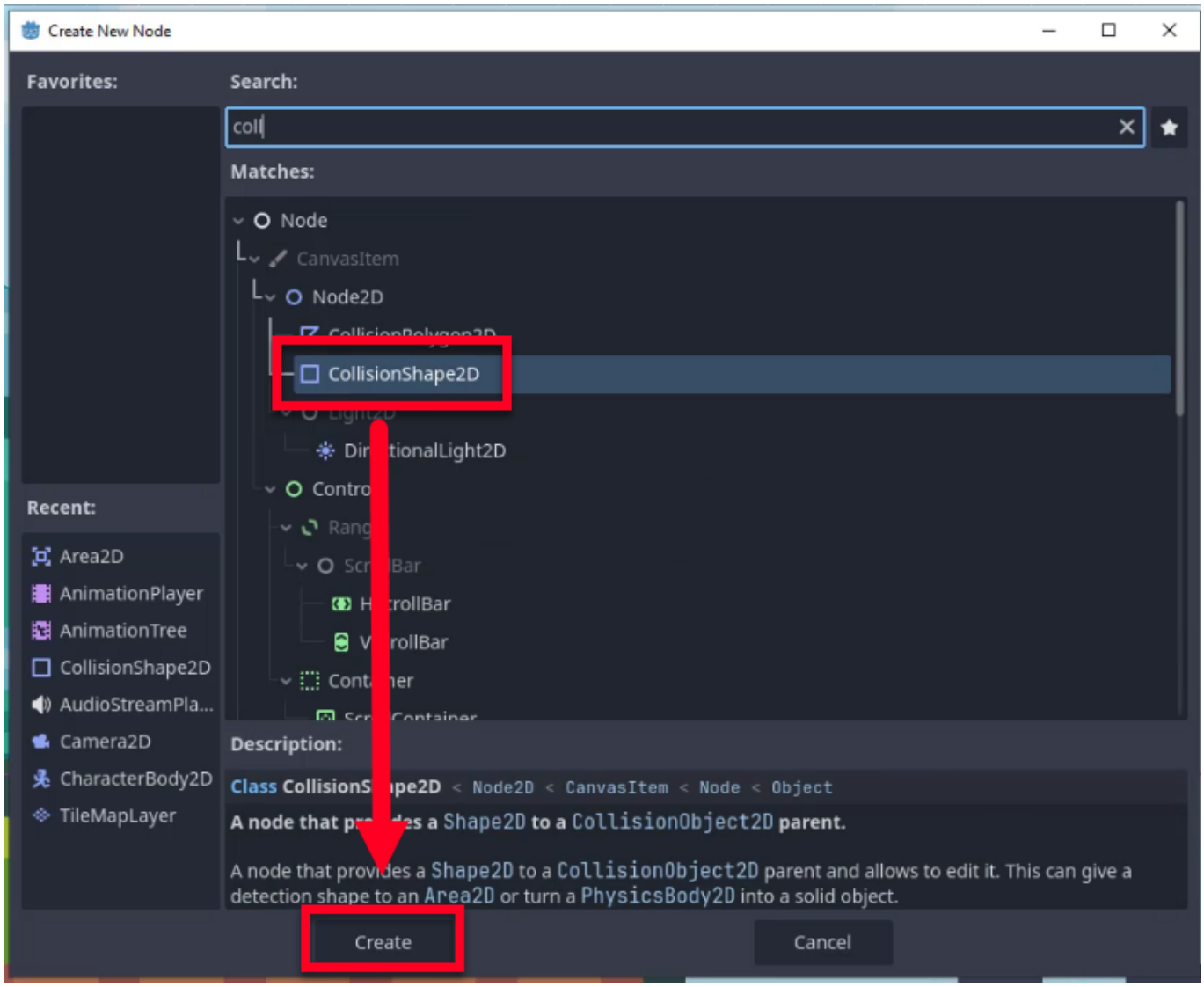
Add a Sprite node as a child of Coin and assign it the coin texture from your assets. Make sure to rename the Sprite node to Sprite for clarity.



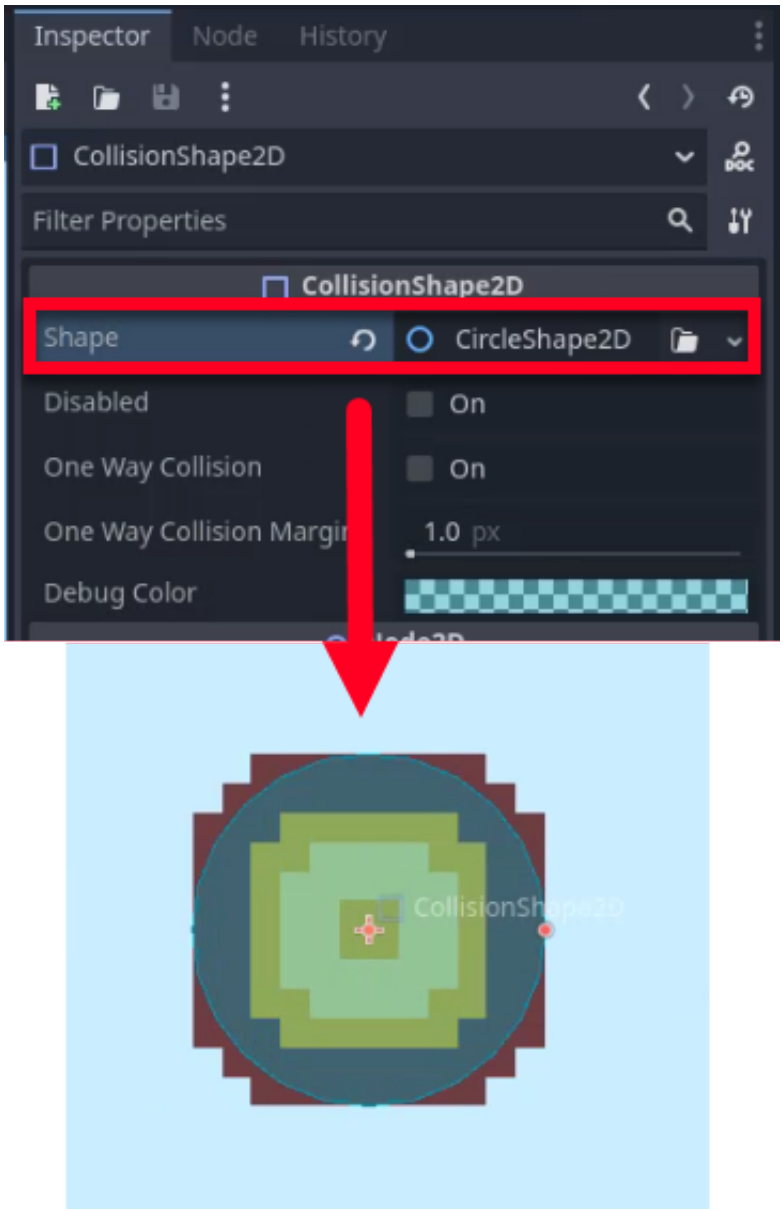
Set the position of the Sprite node to (0, 0).



Next, we'll add a collision shape to the coin so that the player can interact with it. Add a CollisionShape2D node as a child of Coin.

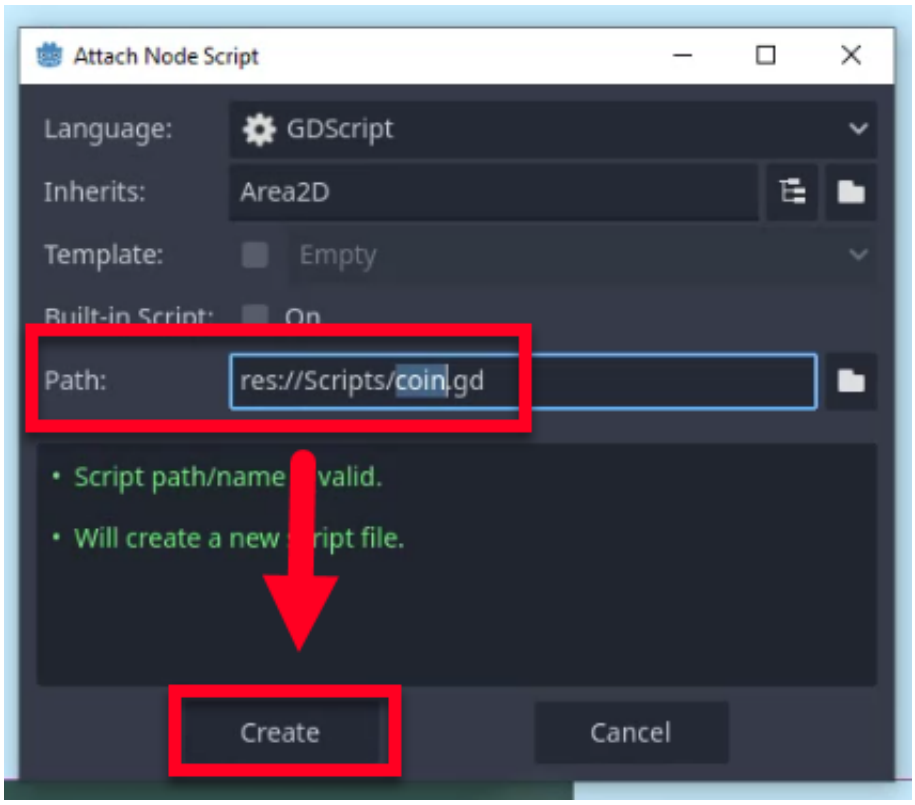


Create a circular collision shape and fit it to the coin’s size.



Animating the Coin

To make the coin visually appealing, we'll add two animations: a bobbing motion and a rotation effect. Let's start by creating a new script for the coin. Create a new script on the Coin node and name it coin.gd.



Now, open the script and set up the initial variables:

```
extends Area2D

var rotate_speed : float = 3.0
var bob_height : float = 5.0
var bob_speed : float = 5.0

@onready var start_pos : Vector2 = global_position

@onready var sprite : Sprite2D = $Sprite
```

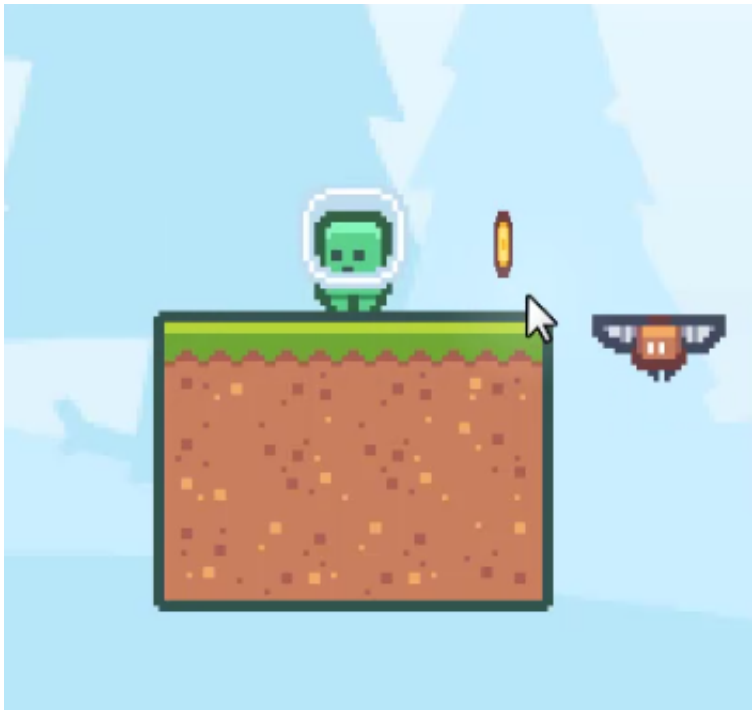
Next, we'll implement the _physics_process function to handle the coin's movement and rotation:

```
func _physics_process(delta):
 var time = Time.get_unix_time_from_system()

 # rotate
 sprite.scale.x = sin(time * rotate_speed)

 # bob up and down
 var y_pos = ((1 + sin(time * bob_speed)) / 2) * bob_height
 global_position.y = start_pos.y - y_pos
```

Let's break down the code:

- rotate_speed, bob_height, and bob_speed are variables that control the speed and height of the coin's rotation and bobbing motion.
- start_pos stores the initial position of the coin.
- sprite is a reference to the Sprite node.
- In the _physics_process function, we use the current time to create a sine wave effect for both the rotation and bobbing motion.
- The coin's x-scale is modified using the sine wave to create a rotation illusion over time.
- The coin's y-position is modified using the sine wave to create a bobbing motion. Note that here we add additional values to the calculation to affect how far and fast the coin bobs.

With this setup, the coin will rotate and bob up and down, making it more visually appealing and engaging for the player.



In the next lesson, we'll implement the functionality for the player to collect the coin and increase their score.

In this lesson, we will continue setting up our coin functionality. Specifically, we will make it so that when the player collides with the coin, their score increases. Although we will not fully implement the scoring system in this lesson, we will lay the groundwork for it. Let's dive in!

Setting Up the Player Script

First, we need to create a function in our player script that will handle the score increase. This function will be called increase_score and will take a parameter to specify the amount by which the score should increase.

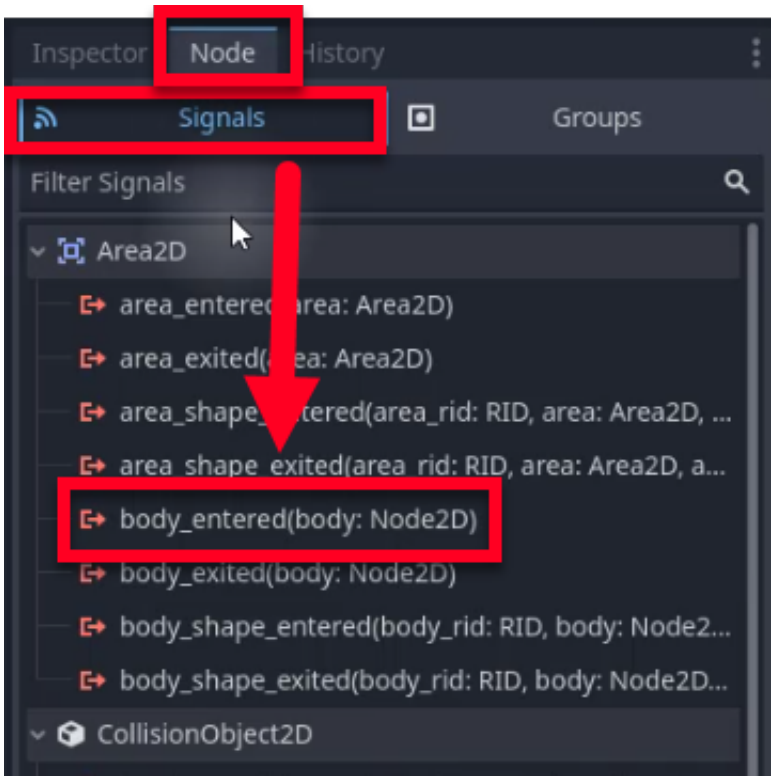
Open the player.gd script and add the following function:

```
func increase_score(amount: int):
    print("increase score")
```

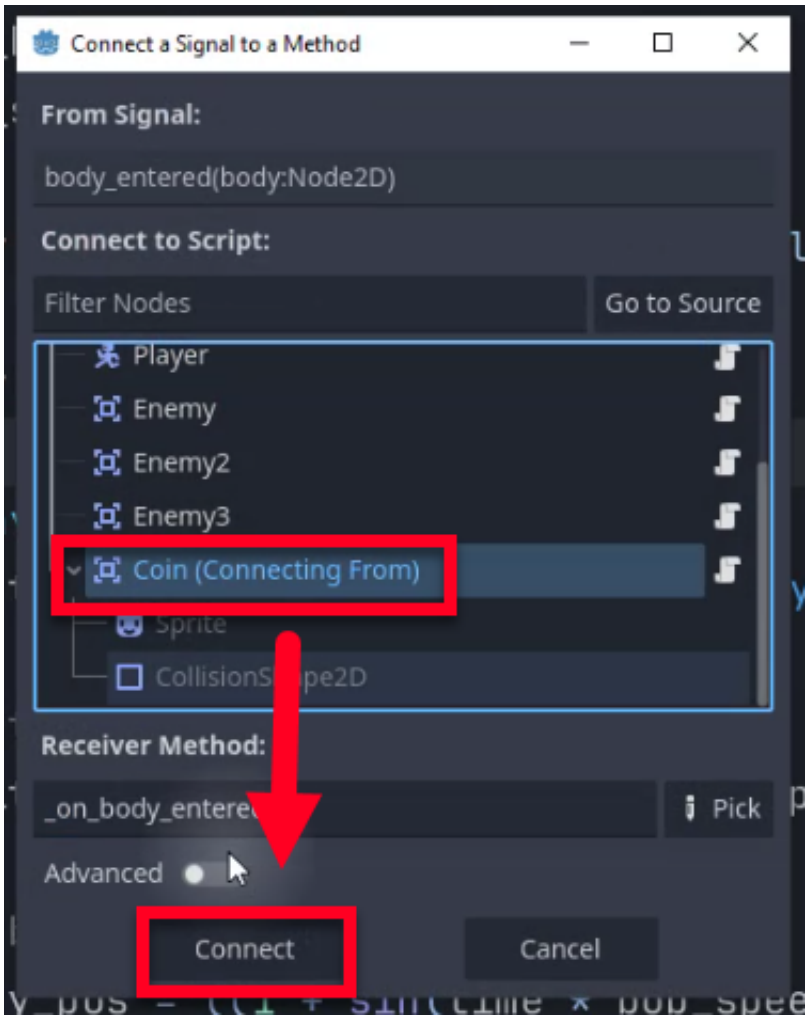
Note that we are not yet implementing the full scoring logic. For now, we will just print a message to the output. The actual scoring logic will be covered in the next lesson.

Handling Coin Collision

Next, we need to handle the collision between the player and the coin. We will use the body_entered signal to detect when the player collides with the coin. Select the coin node in the scene tree, go to the Node tab, and find the body_entered signal.



Double-click on the body_entered signal to connect it. Select the coin node again and click "Connect".



In the coin.gd script, add the following code inside the _on_body_entered function that was generated:

```
func _on_body_entered(body):
    if not body.is_in_group("Player"):
        return
    body.increase_score(1)
    queue_free()
```

This code checks if the body that entered the coin's area is the player. If it is, it calls the increase_score function on the player and then removes the coin from the scene using queue_free().

Testing the Implementation

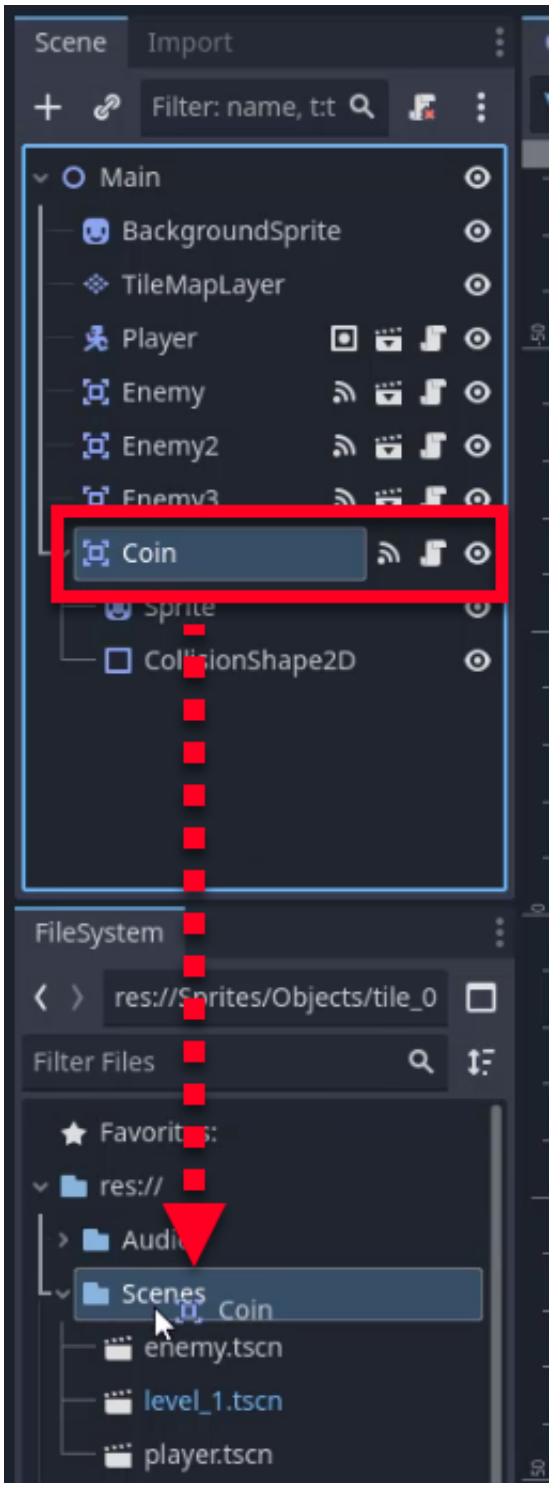
Let's test our implementation to ensure it works as expected:

1. Press the play button to run the scene.
2. Move the player to collect the coin.
3. Check the output to see if the message "increase score" is printed.

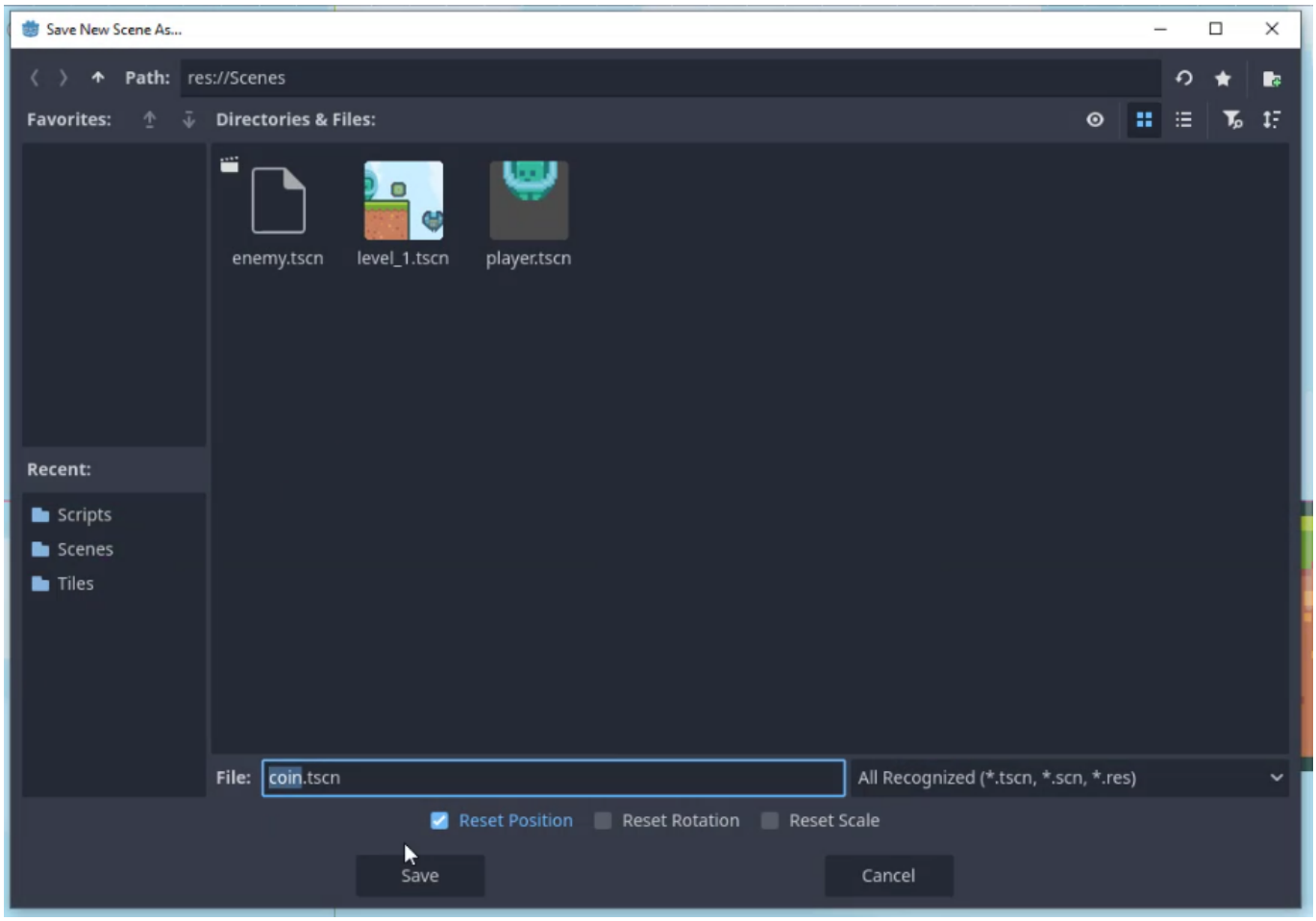
If everything is set up correctly, you should see the coin disappear when the player collects it, and the message "increase score" should be printed in the output.

Saving the Coin as a Scene

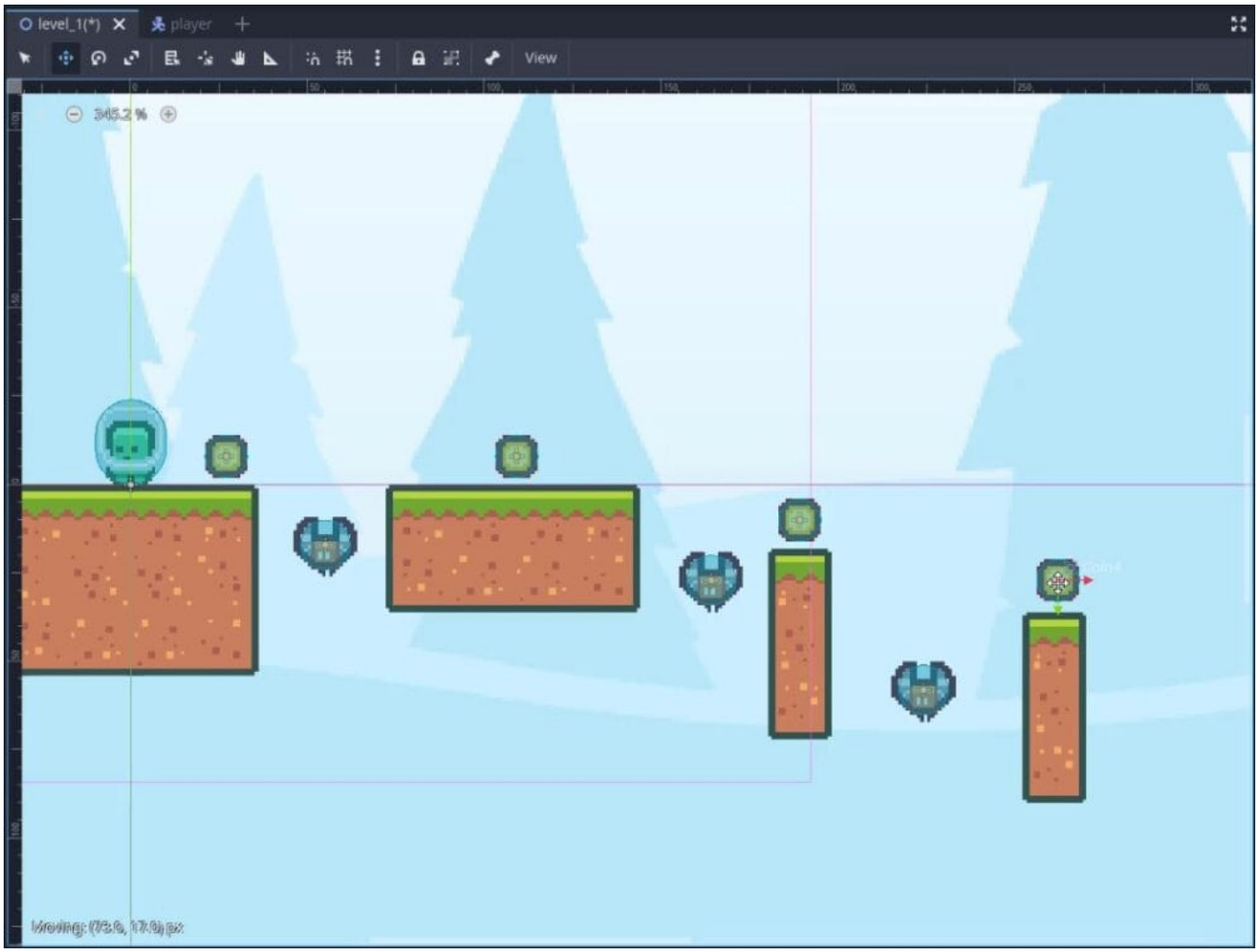
Finally, we will save the coin as a scene so that we can easily duplicate and place it throughout our level. Select the coin node in the scene tree and drag it into the “Scenes” folder in the FileSystem tab.



Save the scene as coin.tscn.



Now you can duplicate the coin and place it throughout your level as needed.



Conclusion

In this lesson, we laid the groundwork for increasing the player's score when they collect a coin. We created a function in the player script to handle the score increase and set up the coin to call this function when the player collides with it. In the next lesson, we will fully implement the scoring system and learn about autoload scripts.

In this lesson, we will set up a scoring system for our game. Unlike health points, which reset with each level, we want our scoring system to persist across all levels. To achieve this, we will use an auto-load script, also known as a singleton. This script will exist outside of our scene hierarchy and maintain its values across different scenes.

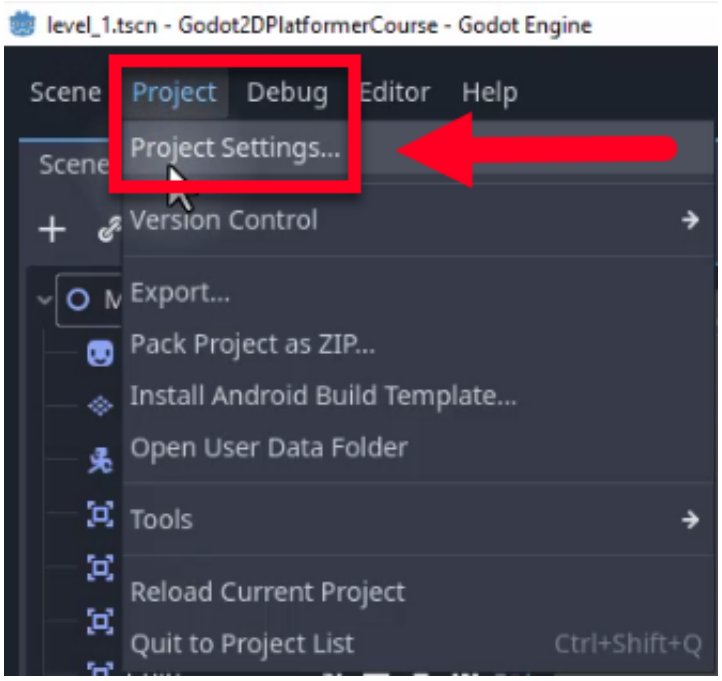
Why Use an Auto-Load Script?

Using an auto-load script is crucial for maintaining persistent data across scenes. Here's why:

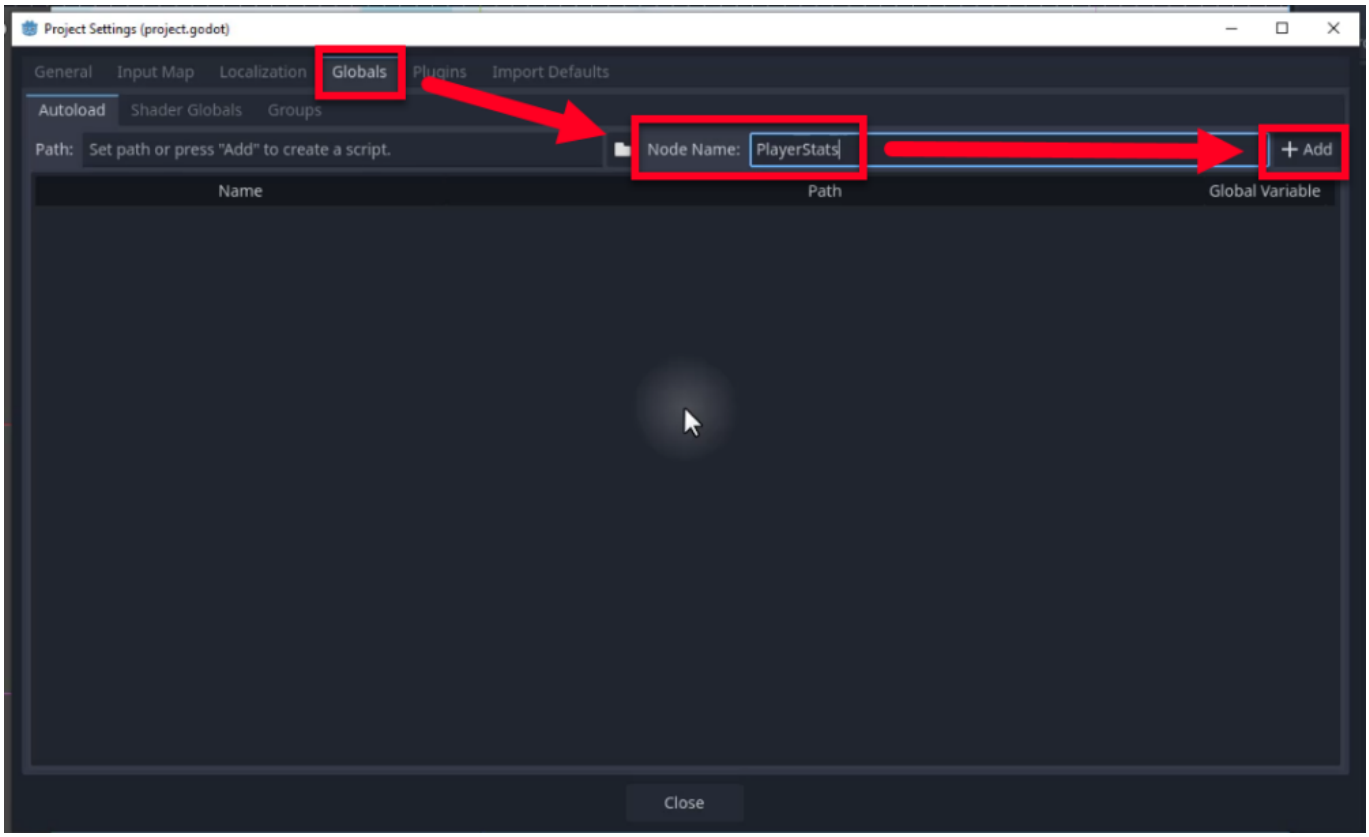
- If we store the score in the player script, it will reset every time we load a new scene.
- An auto-load script exists outside the scene hierarchy, ensuring that the score remains consistent across all levels.

Creating an Auto-Load Script

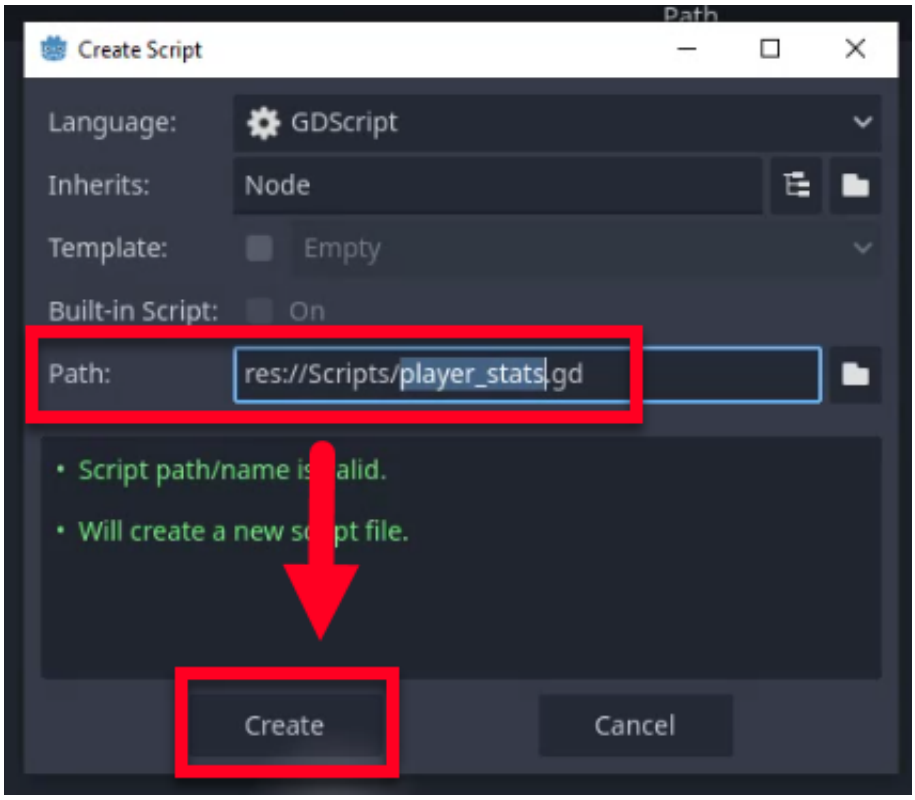
To create an auto-load script, go to **Project** -> **Project Settings**.



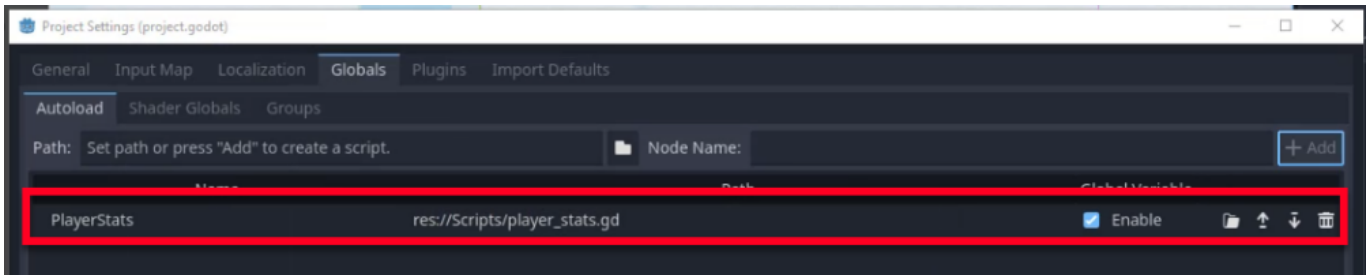
From here, navigate to the **Globals** tab. In the **Node Name** field, enter PlayerStats. This name will be used to access the script in our code. Click **Add** to add the script.



When prompted, choose the **Scripts** folder to save the new script.



This will create a new script file named PlayerStats.gd. and add it to the Autoload list.



Defining the Score Variable

Open the newly created PlayerStats.gd script and define a score variable:

```
extends Node

var score : int = 0
```

Accessing the Score Variable

To access and modify the score variable from another script, such as the player script, follow these steps:

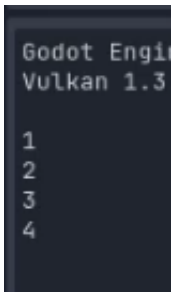
1. Open the player script (player.gd).
2. Locate the increase_score function and modify it to update the score variable in the PlayerStats script:

```
func increase_score(amount : int):
 PlayerStats.score += amount
 print(PlayerStats.score)
```

Testing the Scoring System

Let’s test our scoring system to ensure it works as expected. First, play the game and collect coins. Observe the output to see the score incrementing.

Restart the scene by letting the player character die (e.g., by hitting an enemy three times). Collect more coins and verify that the score continues to increment from the previous value, indicating that the score persists across scenes.



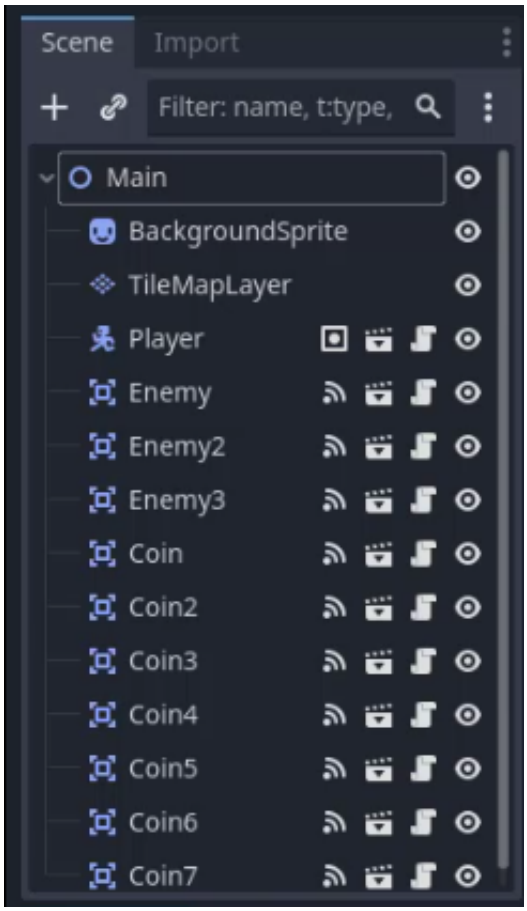
By following these steps, you have successfully created a persistent scoring system using an auto-load script. This ensures that the player's score is maintained across different levels, enhancing the gameplay experience.

In future lessons, we will explore how to reset the score when the game restarts and implement additional features to enhance our scoring system.

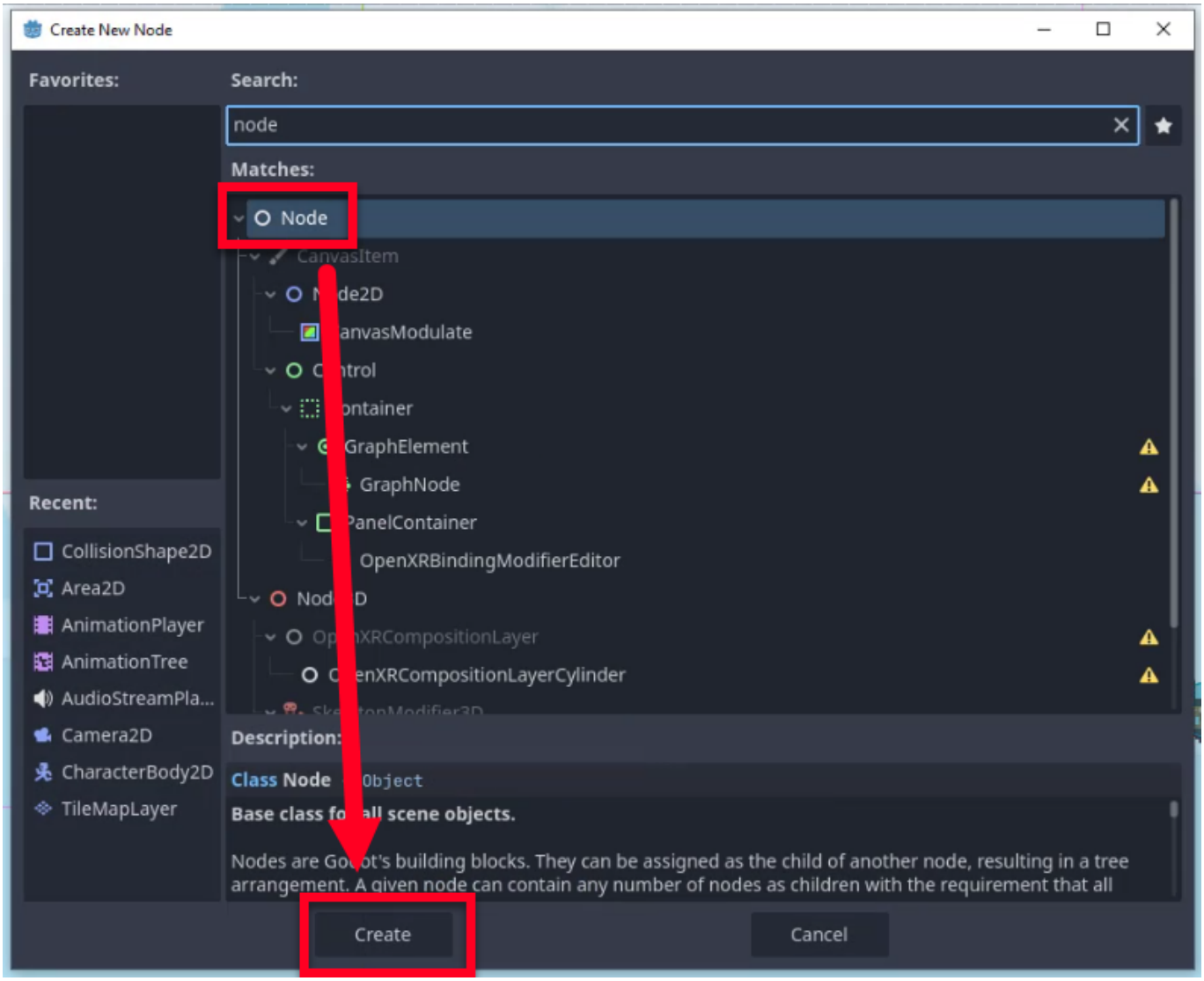
In this lesson, we will set up the end flag for our game. The end flag will serve as the goal that the player aims to reach at the end of each level. To accomplish this, we will organize our scene hierarchy, create the end flag, and write a script to handle the scene transition when the player reaches the end flag.

Organizing the Scene Hierarchy

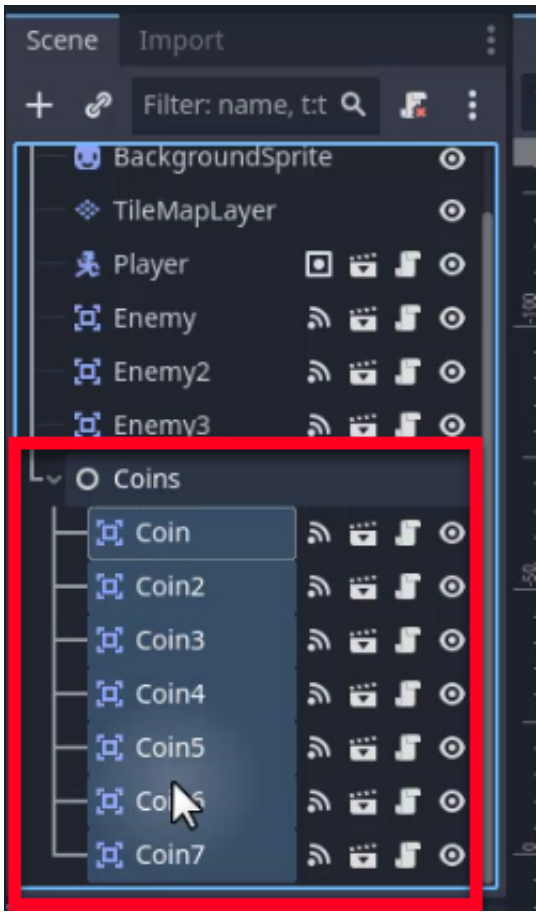
First, let's clean up our scene hierarchy to make it more manageable. Currently, our scene contains multiple enemies and coins, which can quickly become disorganized.



We will create containers to store these nodes. Right-click on the main node and select **Add Child Node**. Search for an **Empty Node** and add it to the scene.

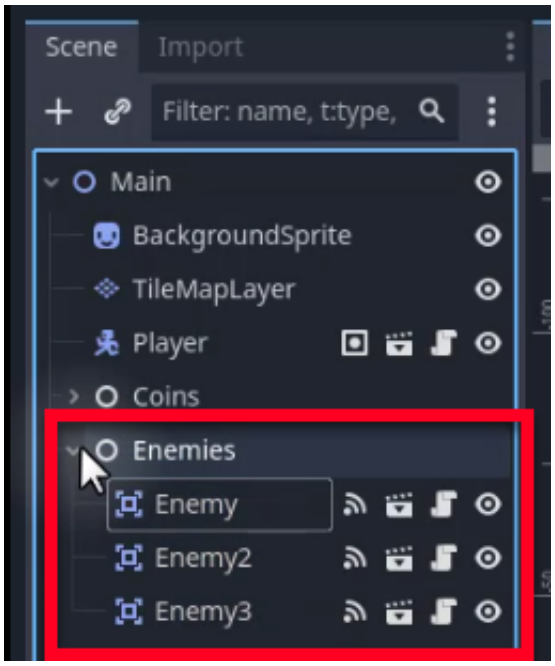


Rename it to Coins. Then, select all the coin nodes and drag them into the Coins node.



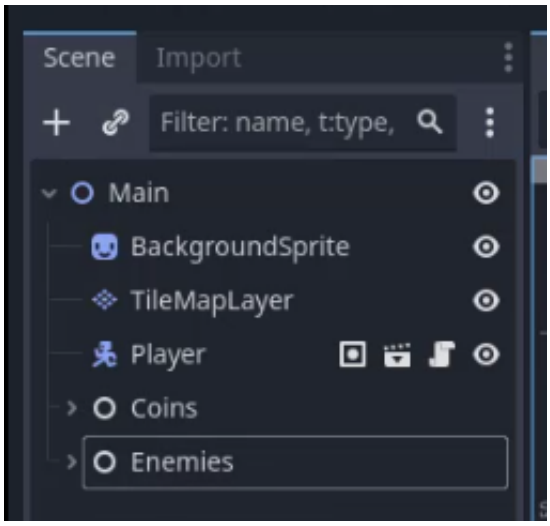
Repeat the process for the enemies:

- Create another **Empty Node** and rename it to Enemies.
- Select all the enemy nodes and drag them into the Enemies node.



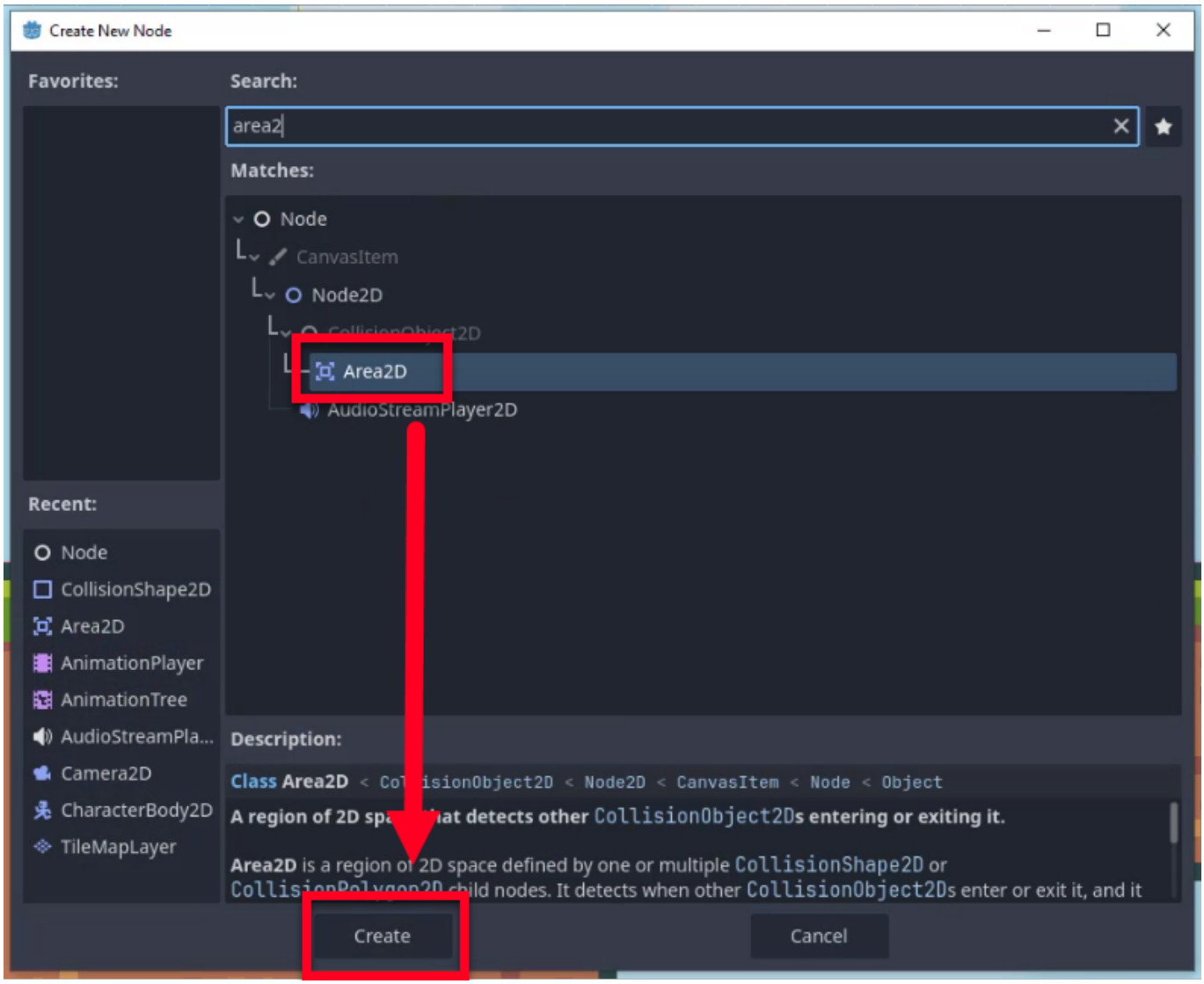
Now our scene is more organized with dropdowns that allow us to access the coins and enemies as

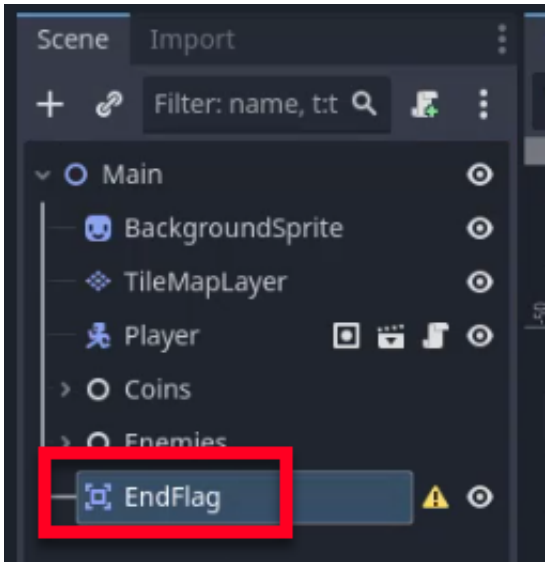
needed.



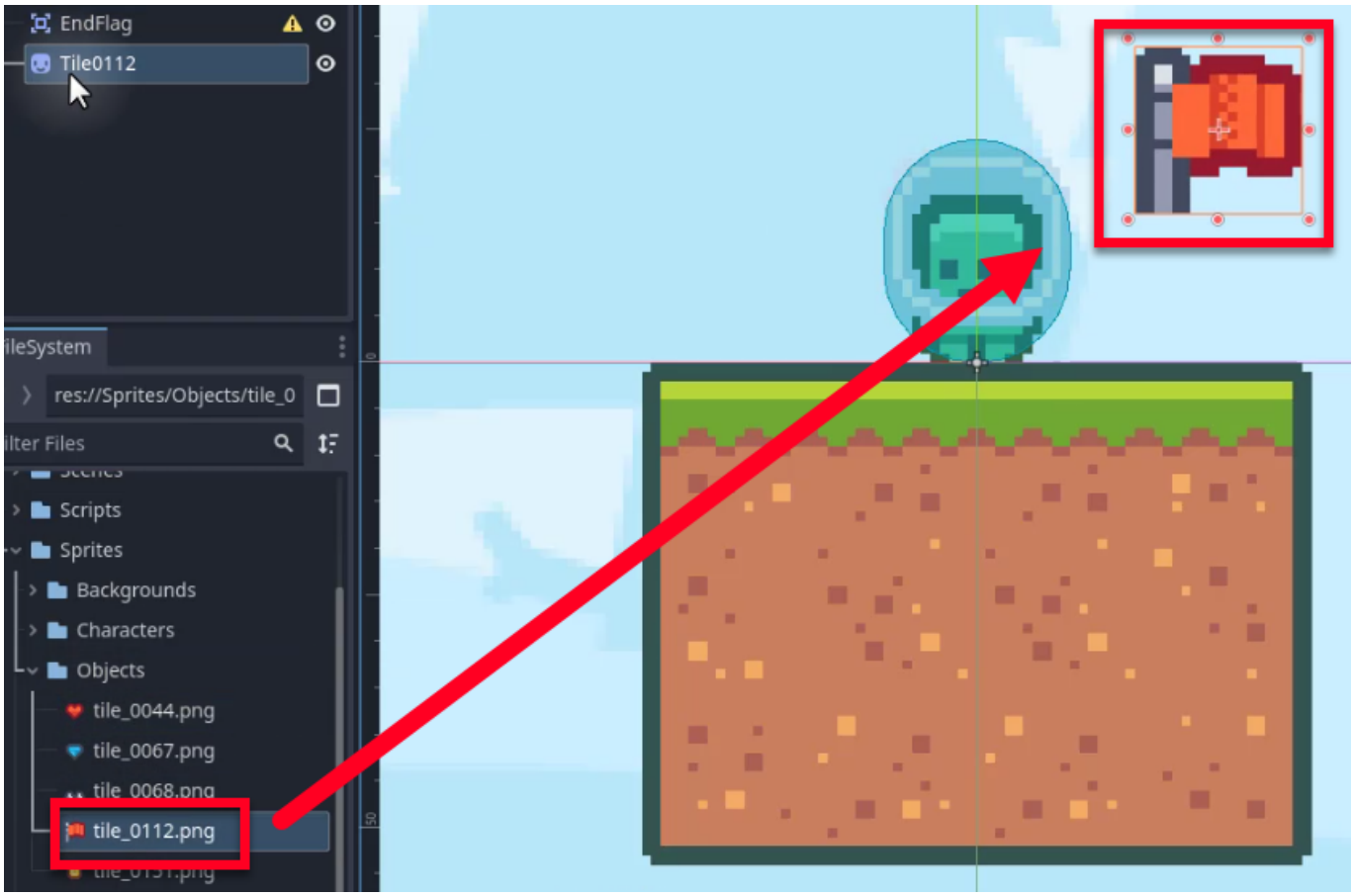
Creating the End Flag

Now that our scene hierarchy is organized, we can create the end flag. The end flag will be an Area2D node with a sprite and a collision shape. First, delete any coin node that is located where you want to place the end flag. Next, as we've done with out objects in this game, create a new Area2D node and rename it to EndFlag.



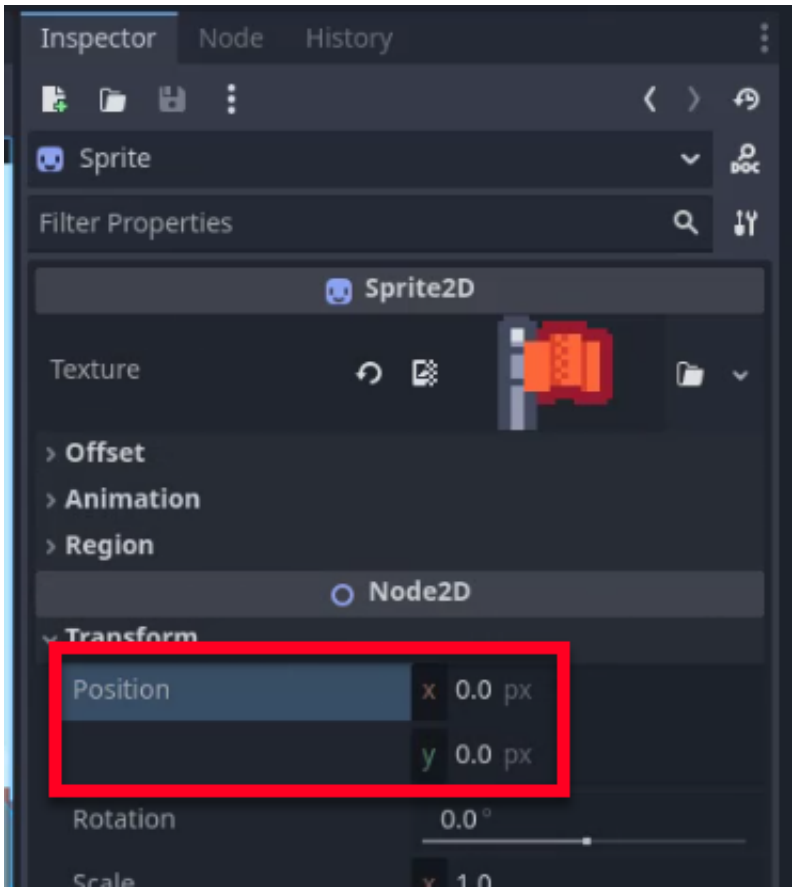


Open the **Sprites** folder, go to the **Objects** subfolder, and drag the flag image into the scene.

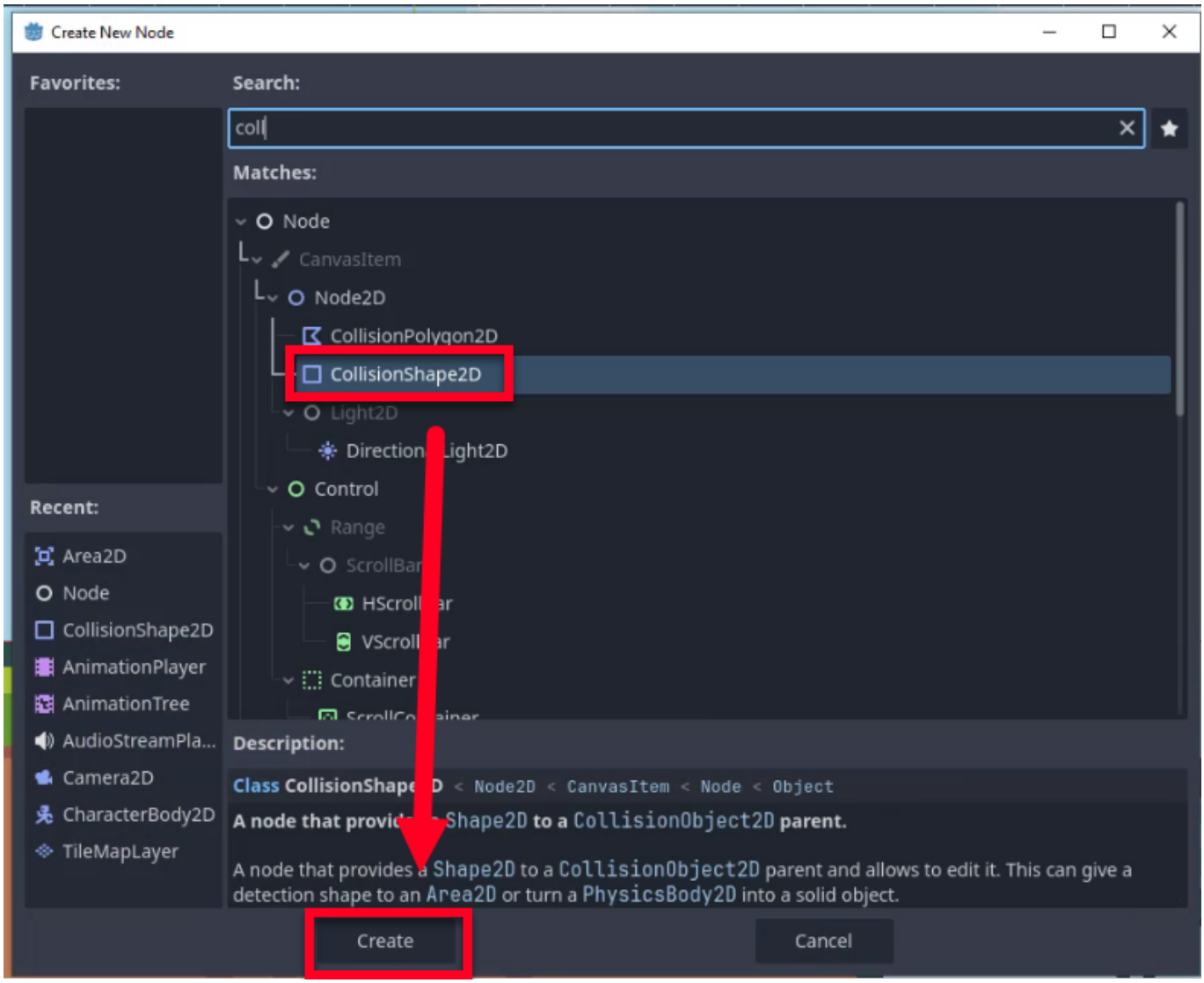


Rename the Sprite node to Sprite, and make it a child of the End Flag node.

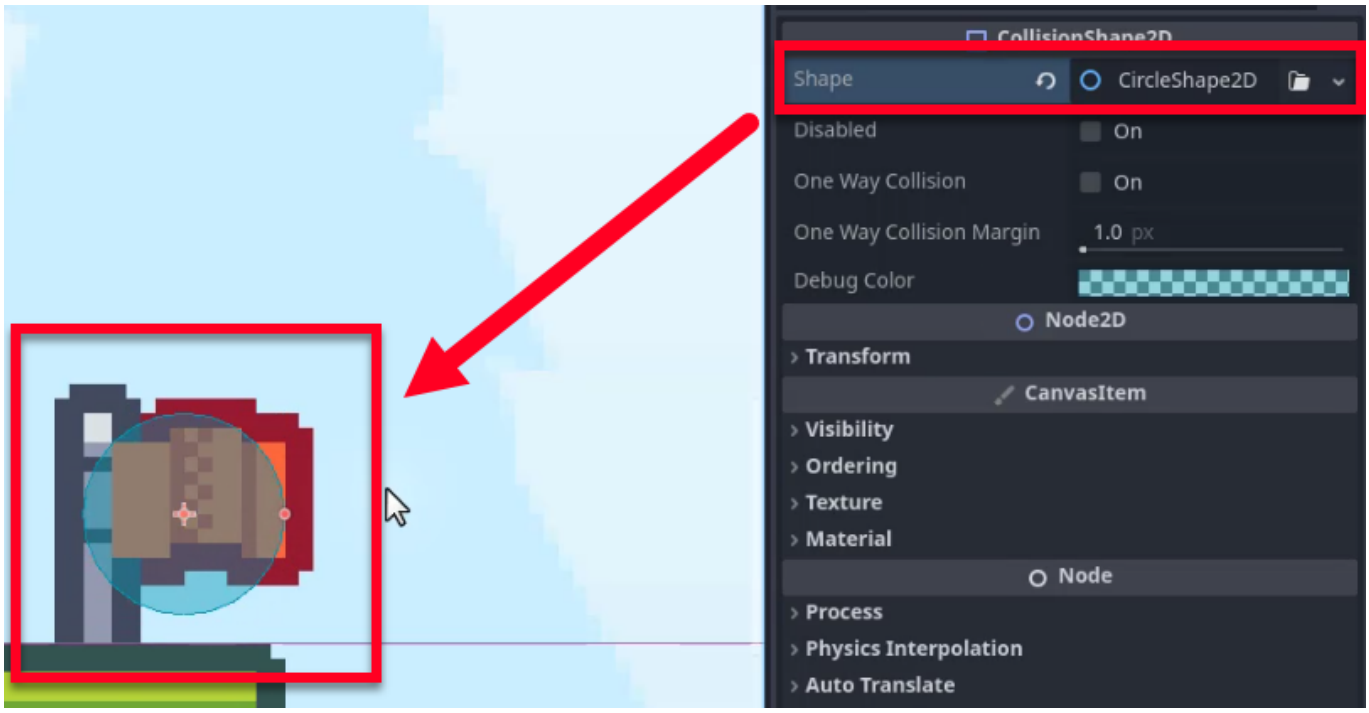
Per usual, set the position of the Sprite node to (0, 0).



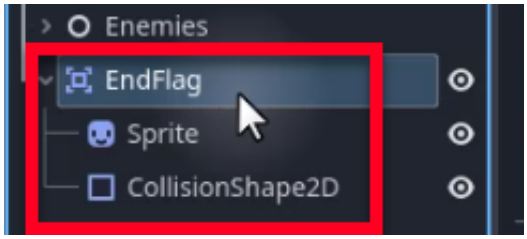
Add a CollisionShape2D node as a child of EndFlag.



Set its collision shape to circle and resize it to fit the flag appropriately.

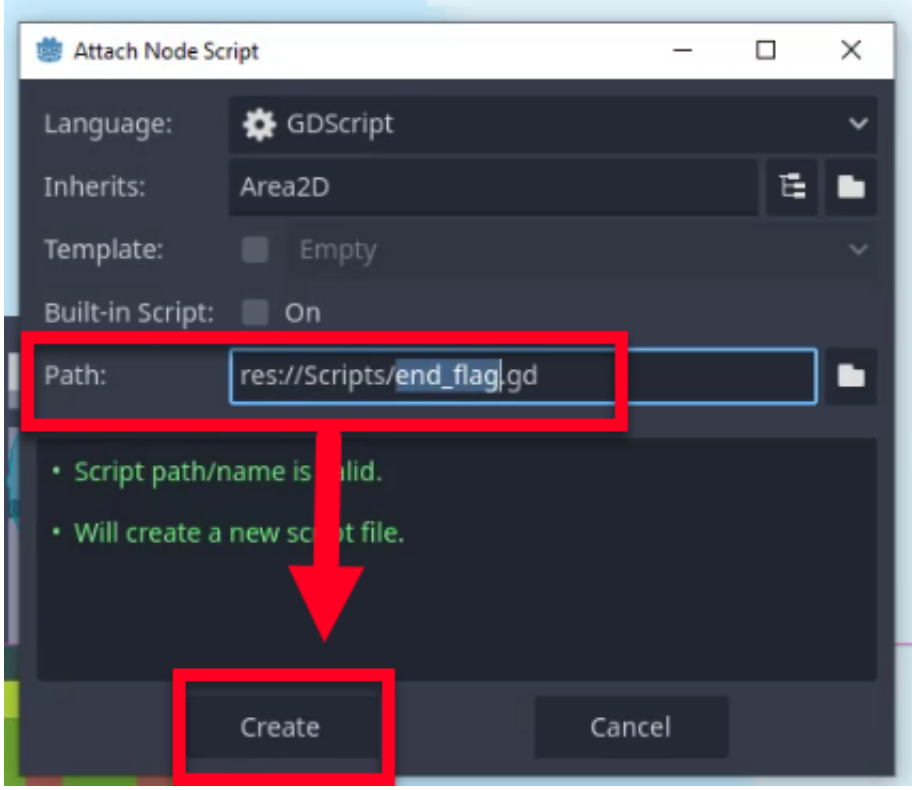


Our flag object is set up and ready to go for logic.

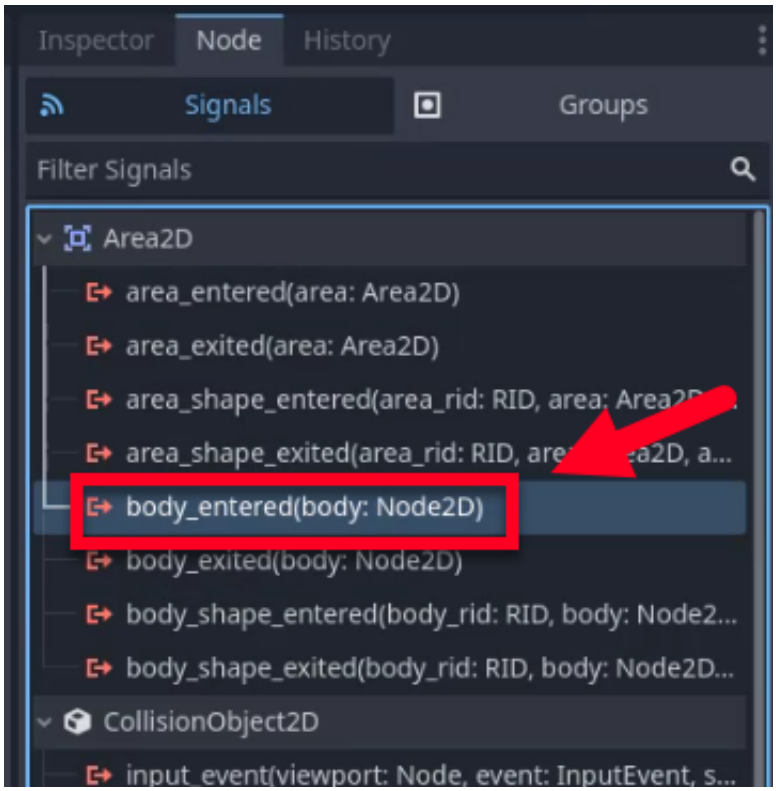


Scripting the End Flag

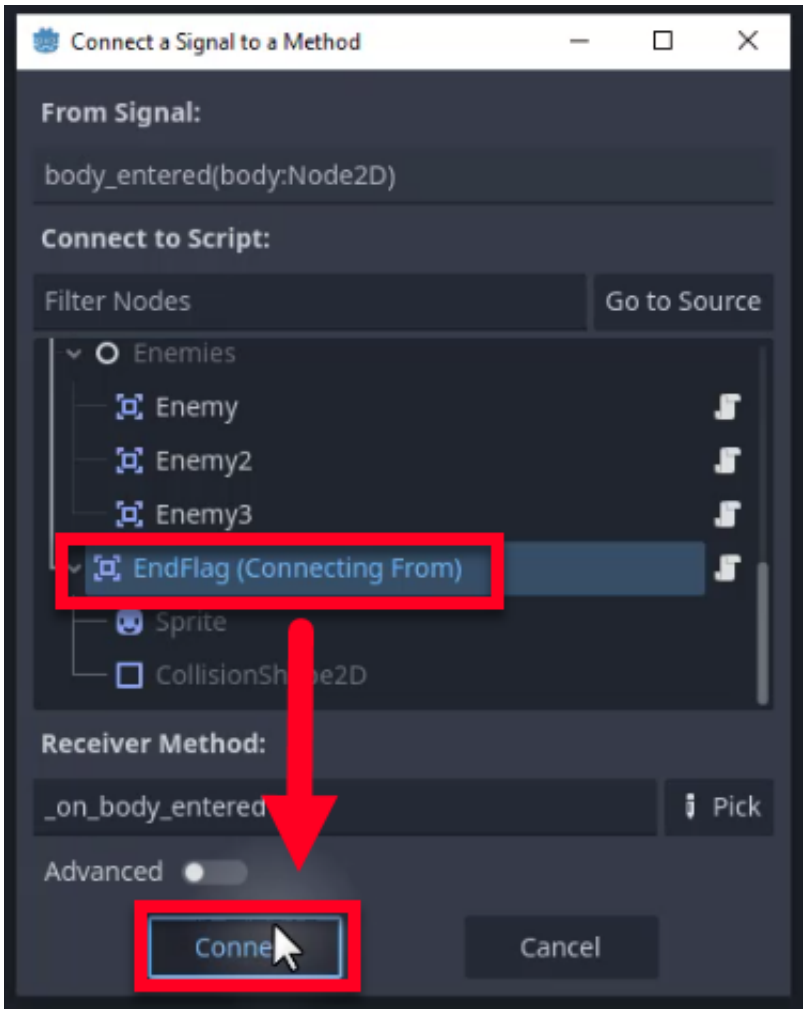
Next, we will create a script for the end flag that will handle the scene transition when the player reaches it. Create a new script attached to the EndFlag node named end_flag.gd.



As we need to detect overlaps, we need to set up the same signal we've used for enemies and coins. Select the EndFlag node, go to the **Node** tab, and find the body_entered signal.



Double-click it, select the EndFlag object, and click Connect. This will add the signal to the script.



Now, let's open the script and add the following code:

```
extends Area2D

@export var scene_to_load : PackedScene

func _on_body_entered(body):
 if not body.is_in_group("Player"):
  return

 get_tree().change_scene_to_packed(scene_to_load)
```

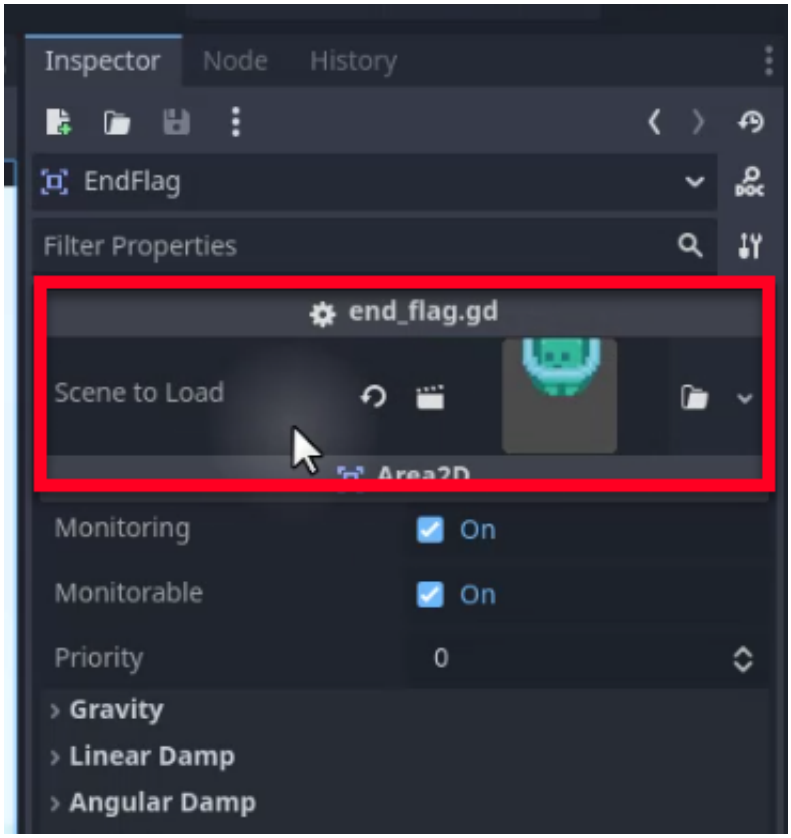
We first add an exported variable scene_to_load of type PackedScene, which allows us to specify the next level's scene in the Godot editor. This makes it super easy to add new levels to the game.

In the _on_body_entered function, we check if the body that entered is in the group "Player". If it's not, the function returns without doing anything. If the body is the player, the script uses the get_tree().change_scene_to_packed function to load the next level's scene, which is stored in the scene_to_load variable. This effectively transfers the player to the next level.

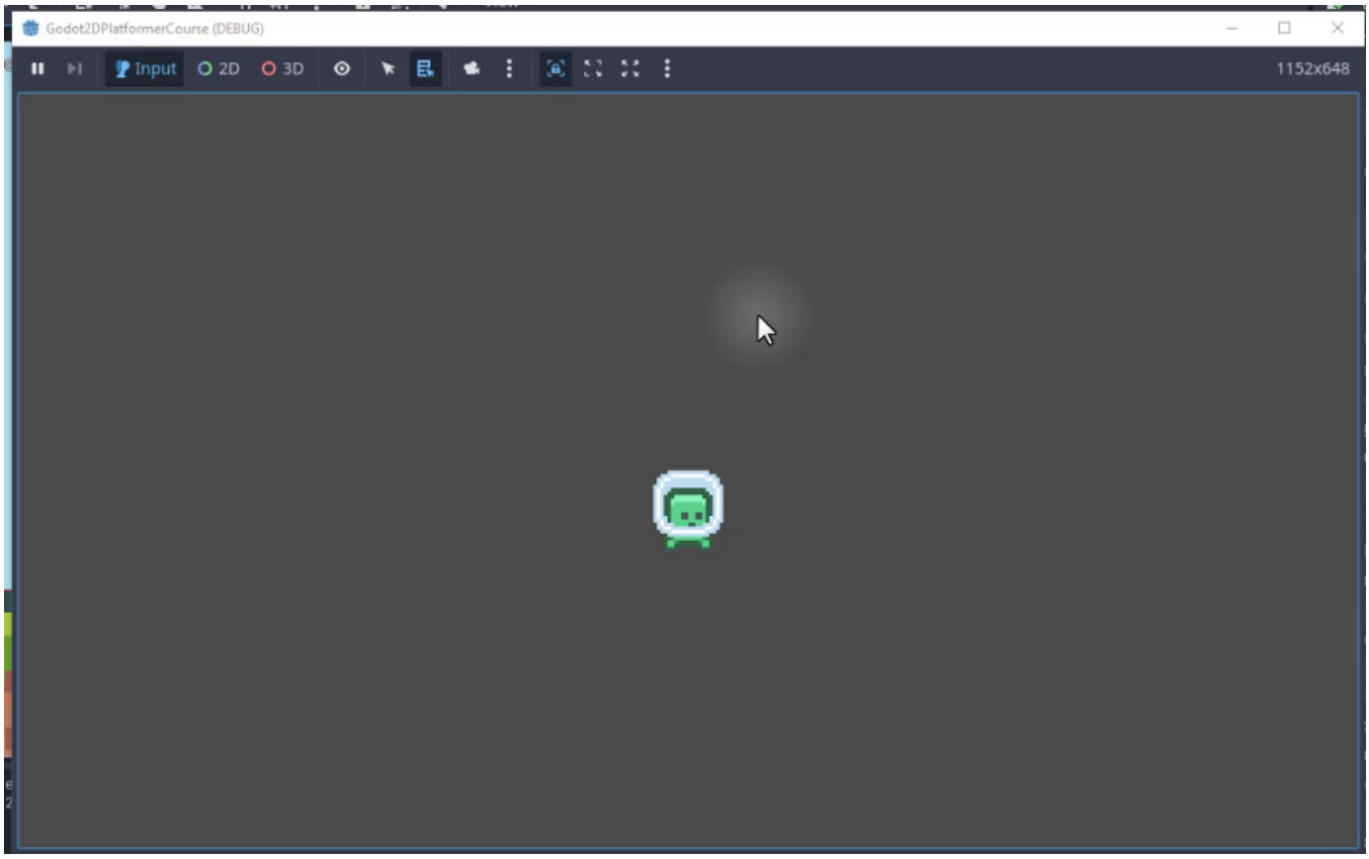
Testing the End Flag

Finally, let’s test the end flag to ensure it works as expected. Select the EndFlag node. In the Inspector, set the scene_to_load variable to the scene you want to load when the player reaches the end flag. As we don’t have a 2nd level yet, any scene will do for these testing purposes.

Note: Godot prevents loading the same scene that is currently loaded to avoid infinite loops. To test the scene transition, use a different scene temporarily. We will make a second level later in the course.



Press the play button to test the game by running into the end flag to see if the scene transitions correctly.



Congratulations! You have successfully set up the end flag for your game. In the next lesson, we will continue to enhance our project by adding UI elements for health and score display.

Godot 2D Platformer - Enemy & Scoring [2025]

1. True or False: In Godot, an “Area2D” node can be used to detect collisions without blocking the movement of other nodes.

a. True
b. False

2. How do we enable the enemy to process the player colliding with it?

a. By using a Timer node to check for proximity
b. By enabling physics collision response
c. By connecting the body_entered signal to a function that checks for the player
d. By setting the player as a child of the enemy node

3. What is the main reason for assigning the player to a “Player” group?

a. To allow the player to collect coins
b. To enable the enemy script to check if the colliding object is the player
c. To store player health in a global script
d. To automatically apply animations to the player

4. Why is call_deferred used in the take_damage function?

a. To delay the health reduction for a few seconds
b. To ensure the game_over function is called after the current physics process completes
c. To execute game_over only when the player collects a coin
d. To prevent the enemy from detecting multiple collisions at once

5. What is the function of queue_free() in the coin script?

a. It removes the coin from the scene after the player collects it
b. It resets the coin’s position
c. It resets the player's score
d. It duplicates the coin in another location

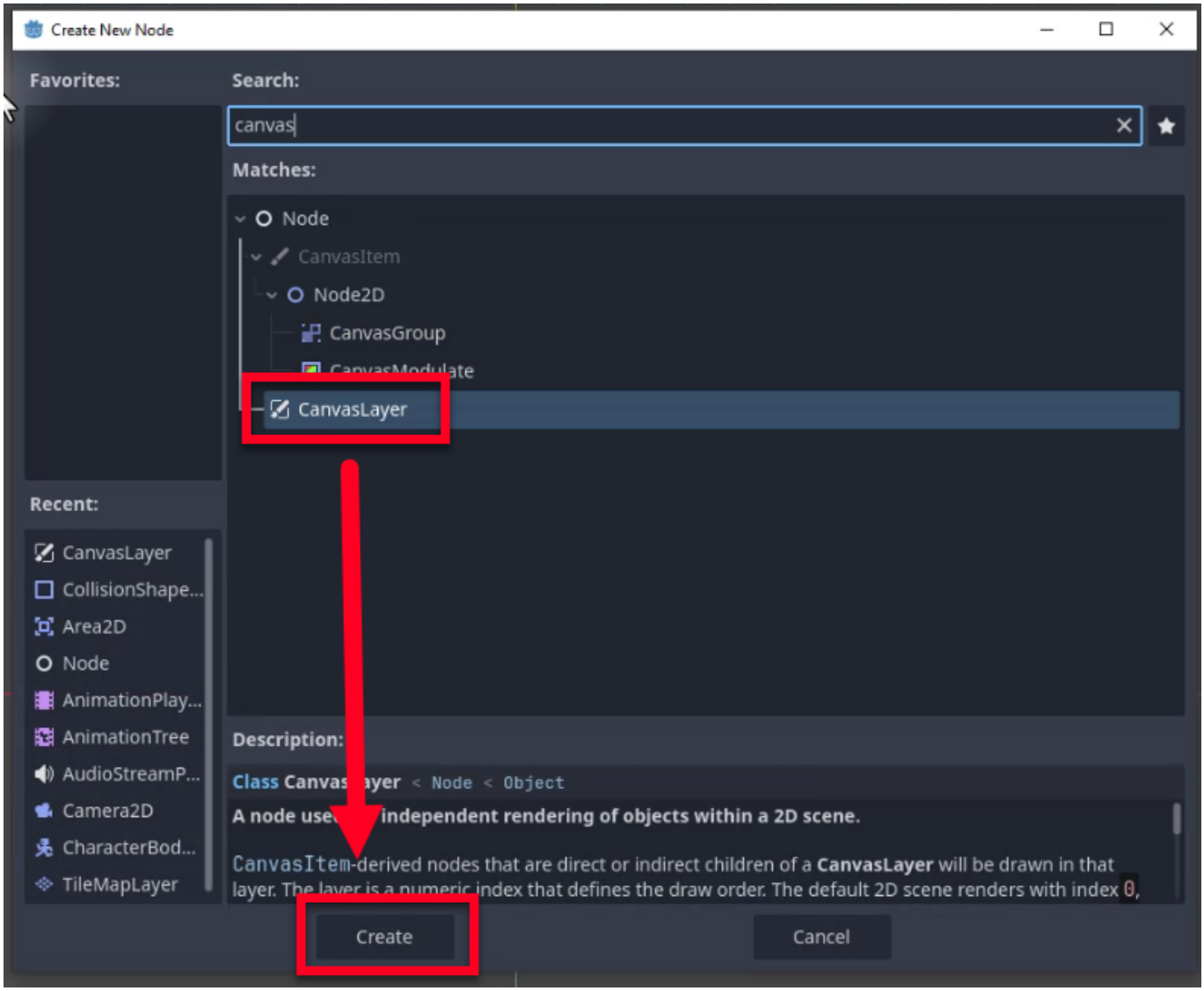
6. Why is an Autoload script used for tracking the player’s score?

a. To reset the score every time a new level loads
b. To allow score values to persist across multiple scenes
c. To store the player's health permanently
d. To create an instance of the player in every level

In this lesson, we will begin working on our User Interface (UI). The UI will consist of two main elements: hearts at the top of the screen representing our health and a text display underneath showing our score.

Setting Up the CanvasLayer

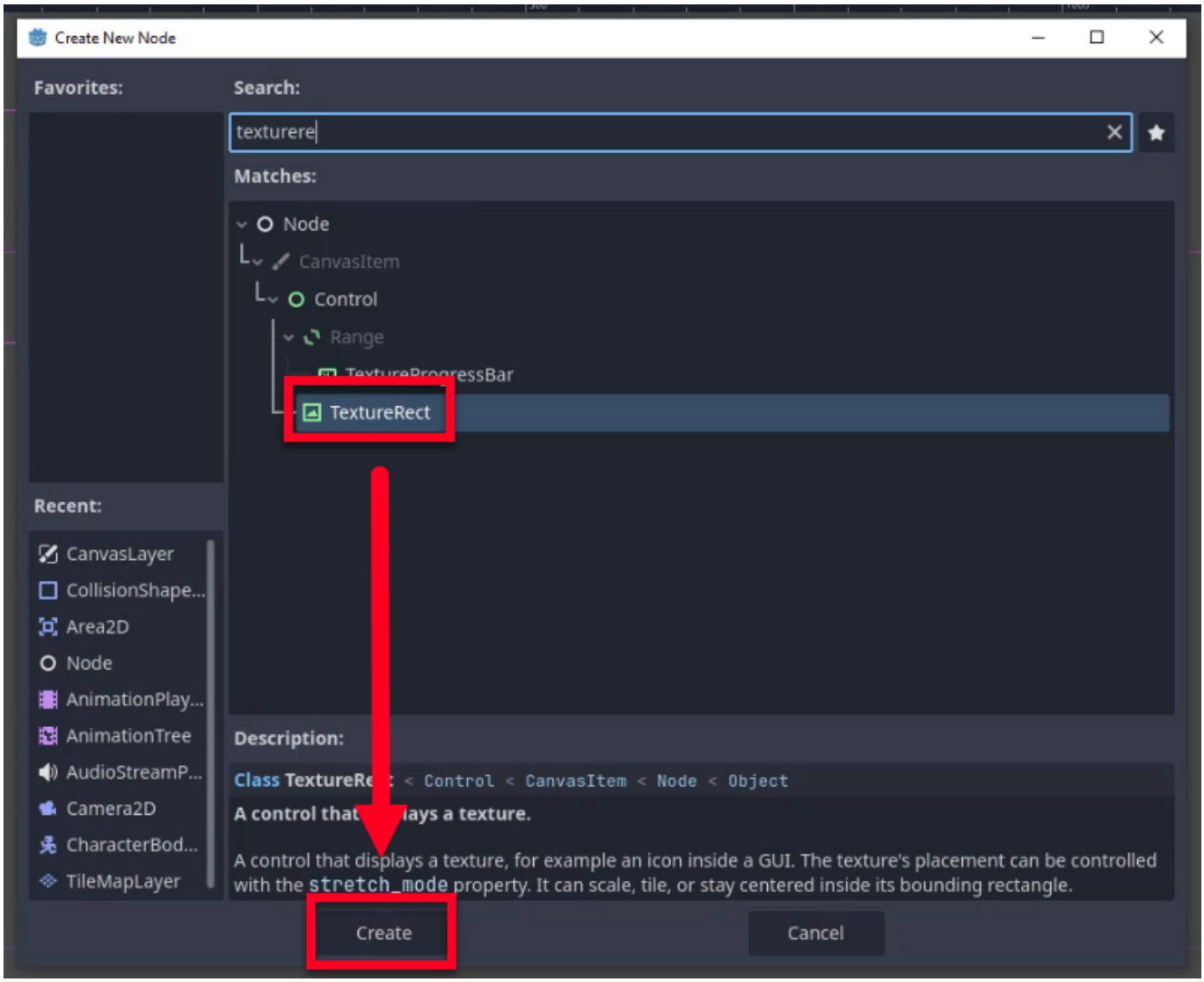
We will be working in the player scene and start by creating a new node called CanvasLayer.



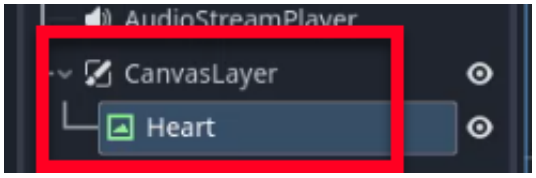
The CanvasLayer node is essential as it ensures that everything within it is rendered directly to the screen, similar to sticking it to your monitor. This is crucial for UI elements that need to remain visible regardless of the player's position in the game world.

Creating the Health Display

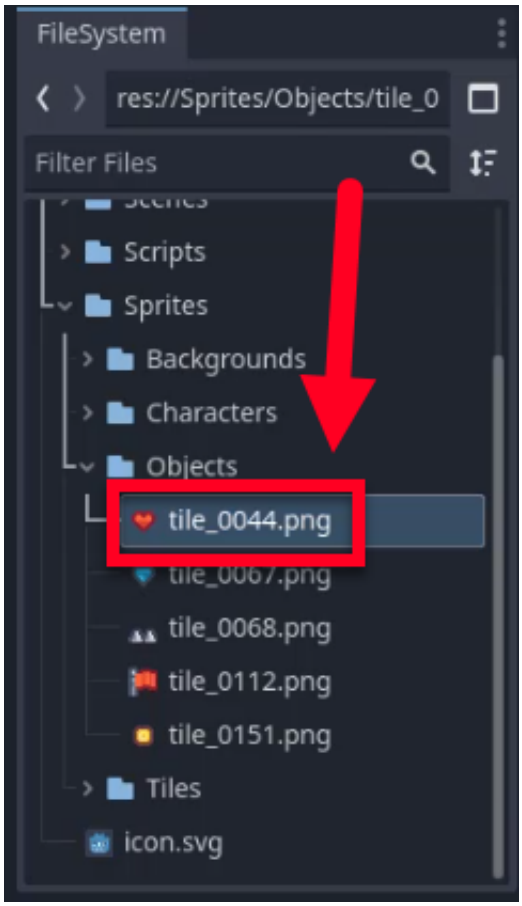
Our health display will be composed of multiple TextureRect nodes, each representing a heart. These nodes will be automatically aligned horizontally using a HBoxContainer. To begin, create a new node as a child of CanvasLayer called TextureRect.



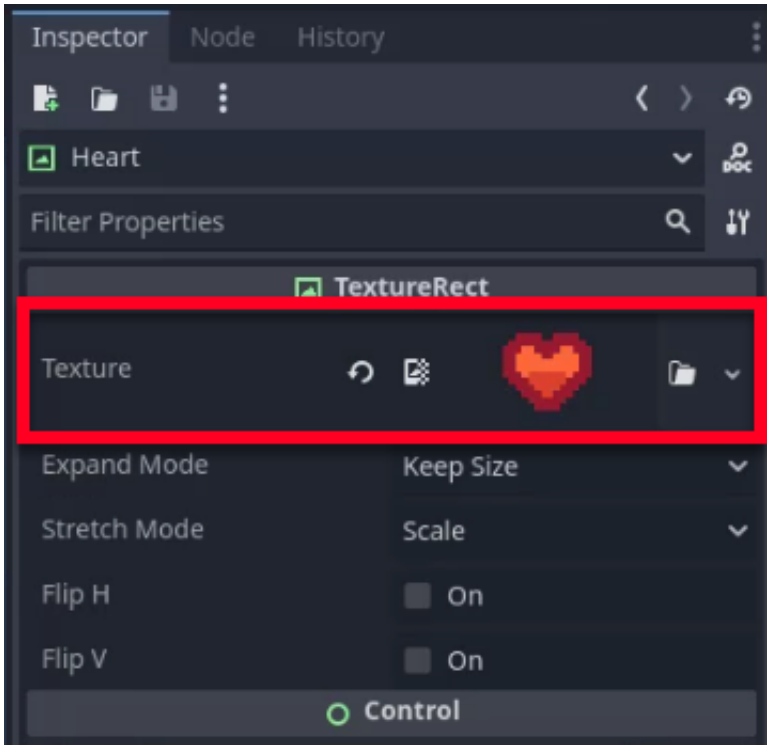
Rename this TextureRect node to Heart.



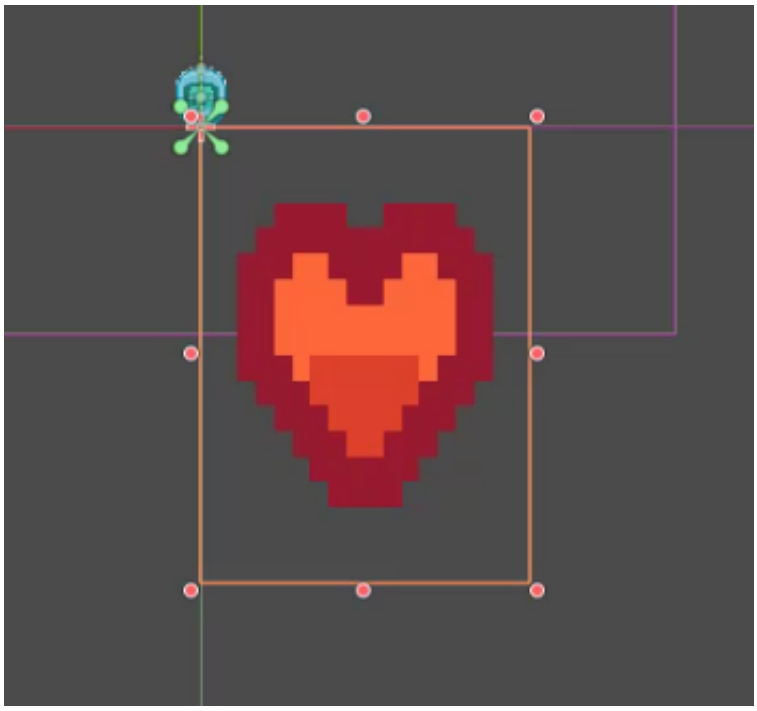
Next, we need to assign a texture to the Heart node. Go to the Sprites folder and locate the heart texture in the Objects folder.



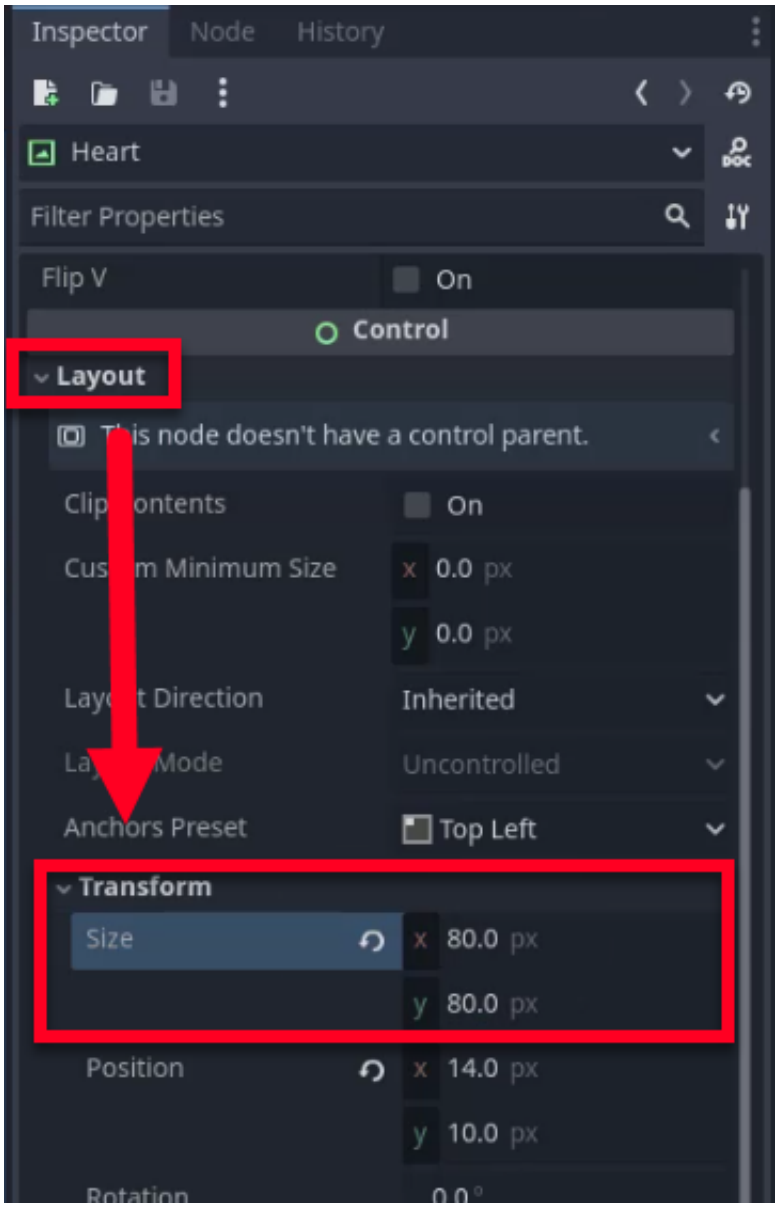
Drag the heart texture into the TextureRect node.



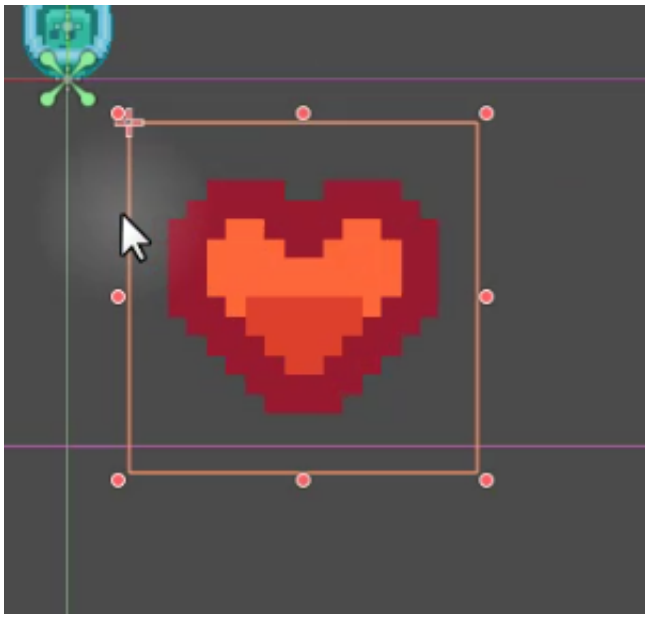
Right now, the heart looks stretched and squished in the editor view.



To fix this, we need to define the size of the heart. In the Inspector, go to the Layout tab. Under Transform, set the size to 80×80.

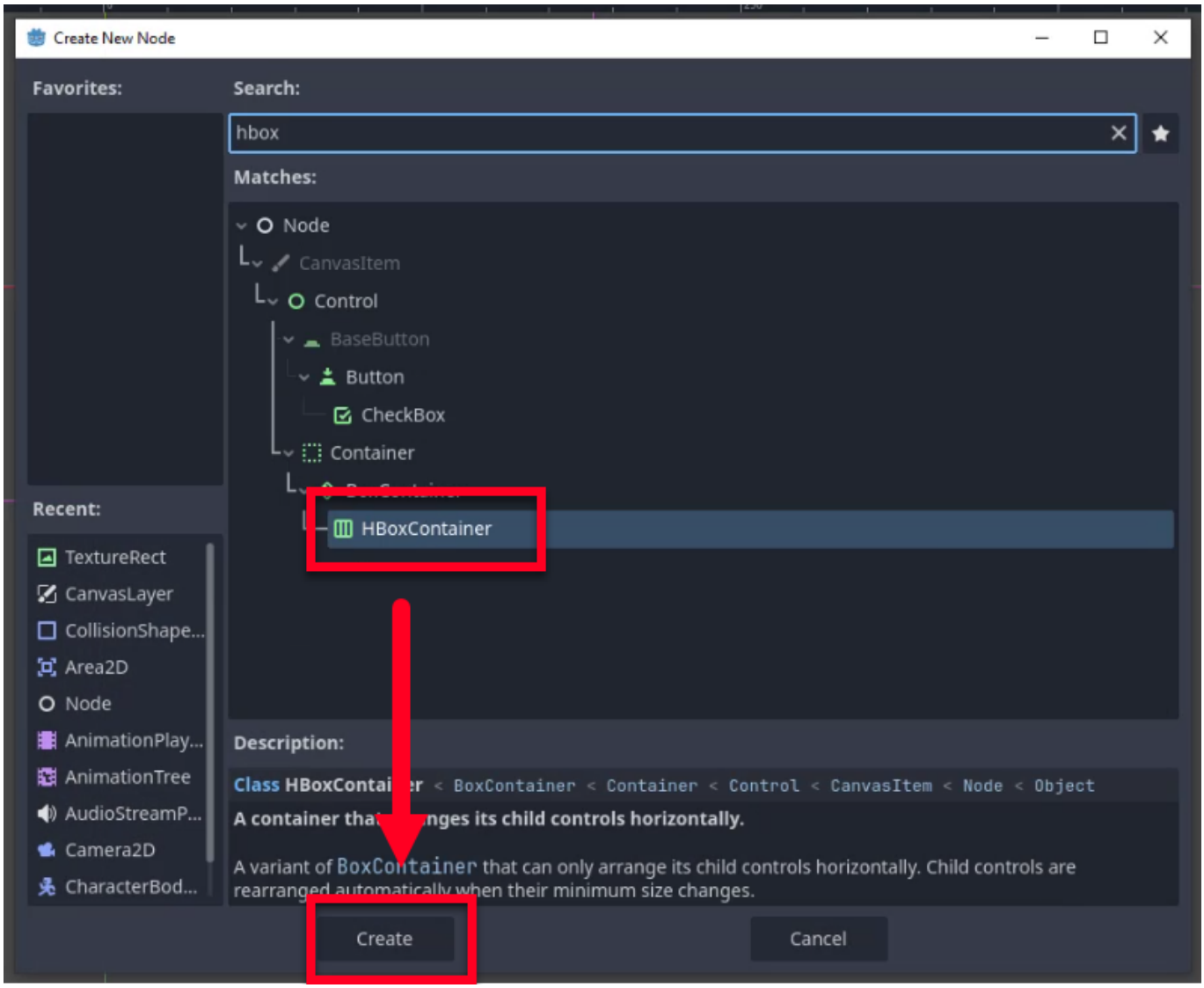


This should give us a perfectly square heart.

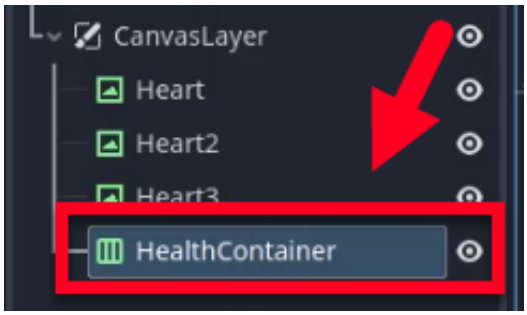


Duplicating Hearts and Using HBoxContainer

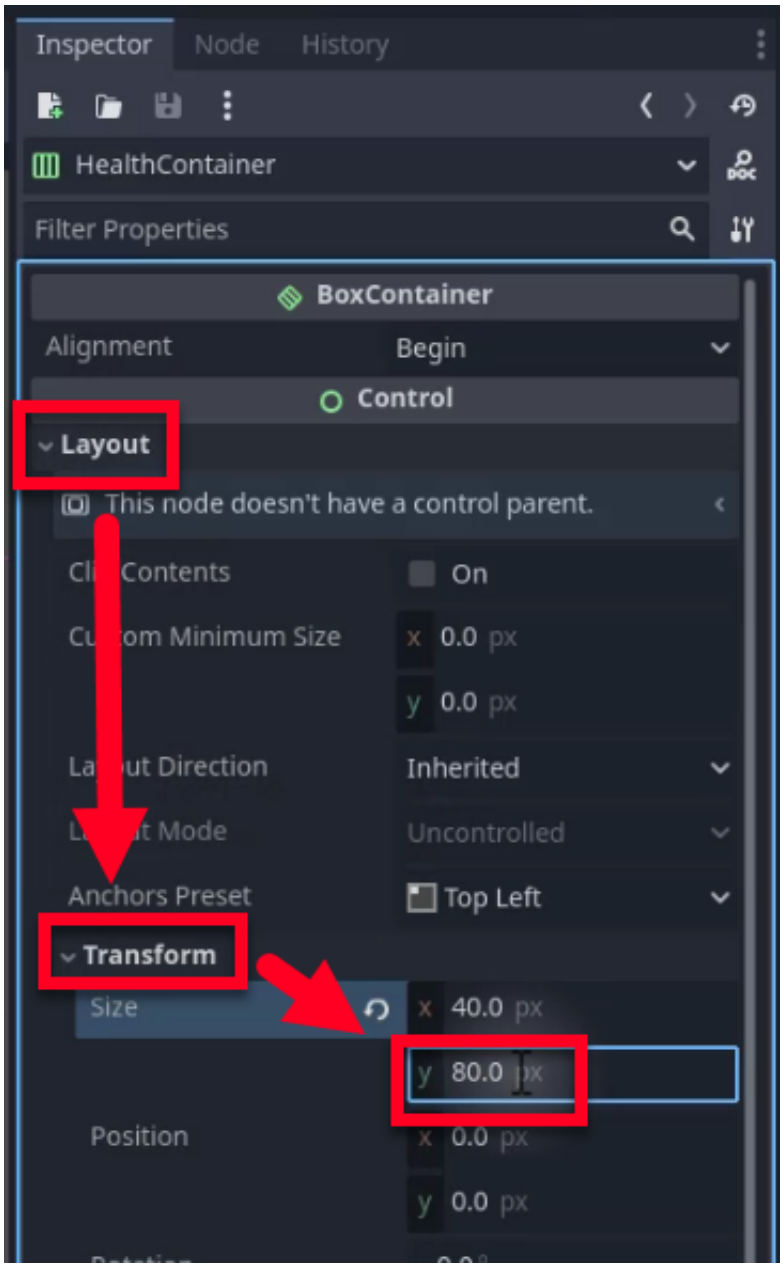
To efficiently manage multiple hearts, we will use an HBoxContainer to automatically position and size them. Create a new node called HBoxContainer as a child of CanvasLayer.



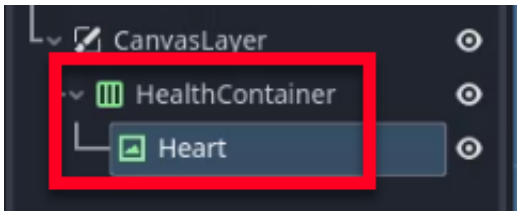
Rename this node to HealthContainer.



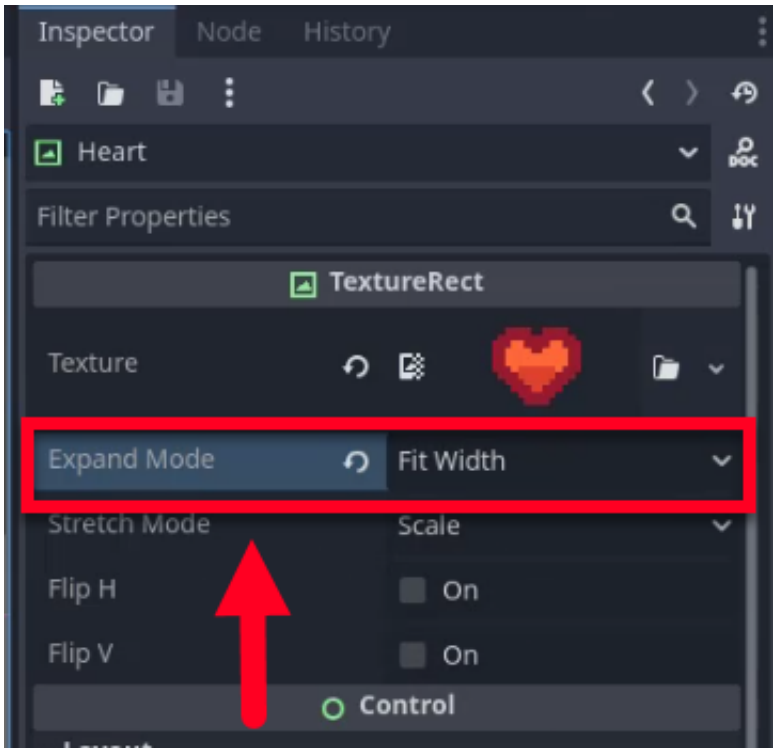
In the Inspector, set the Y size of HealthContainer to 80 to match the hearts. You can find this under the Layout tab in the Transform section.



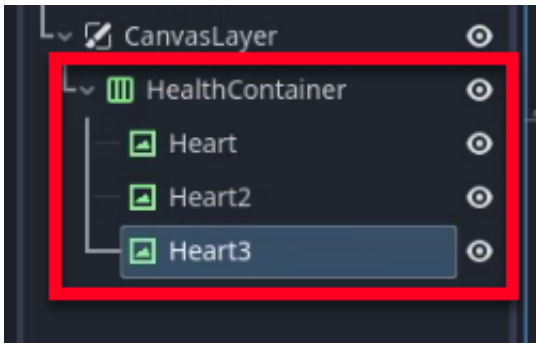
Drag the Heart node to be a child of HealthContainer.



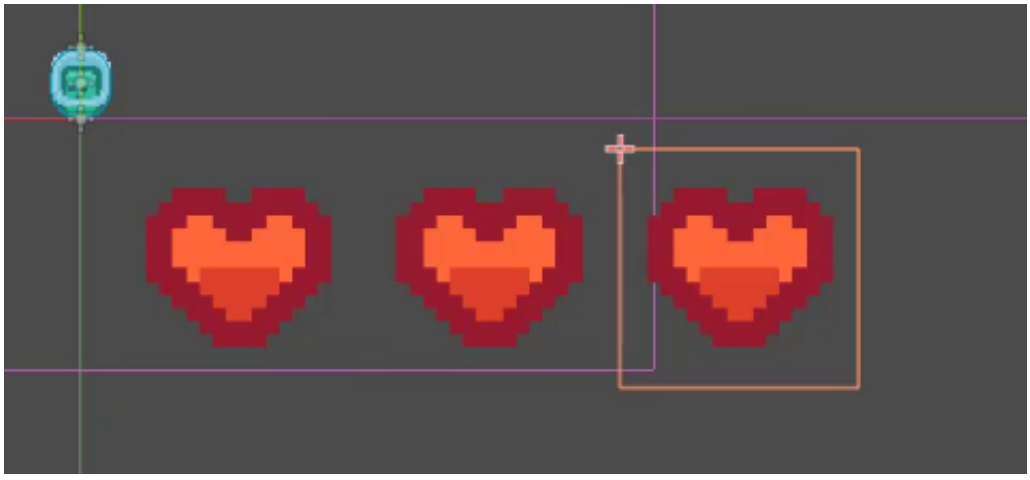
With the Heart selected, change the expand mode of the Heart node from Keep Size to Fit Width in the Inspector.



Lastly, duplicate the Heart node to create additional hearts.

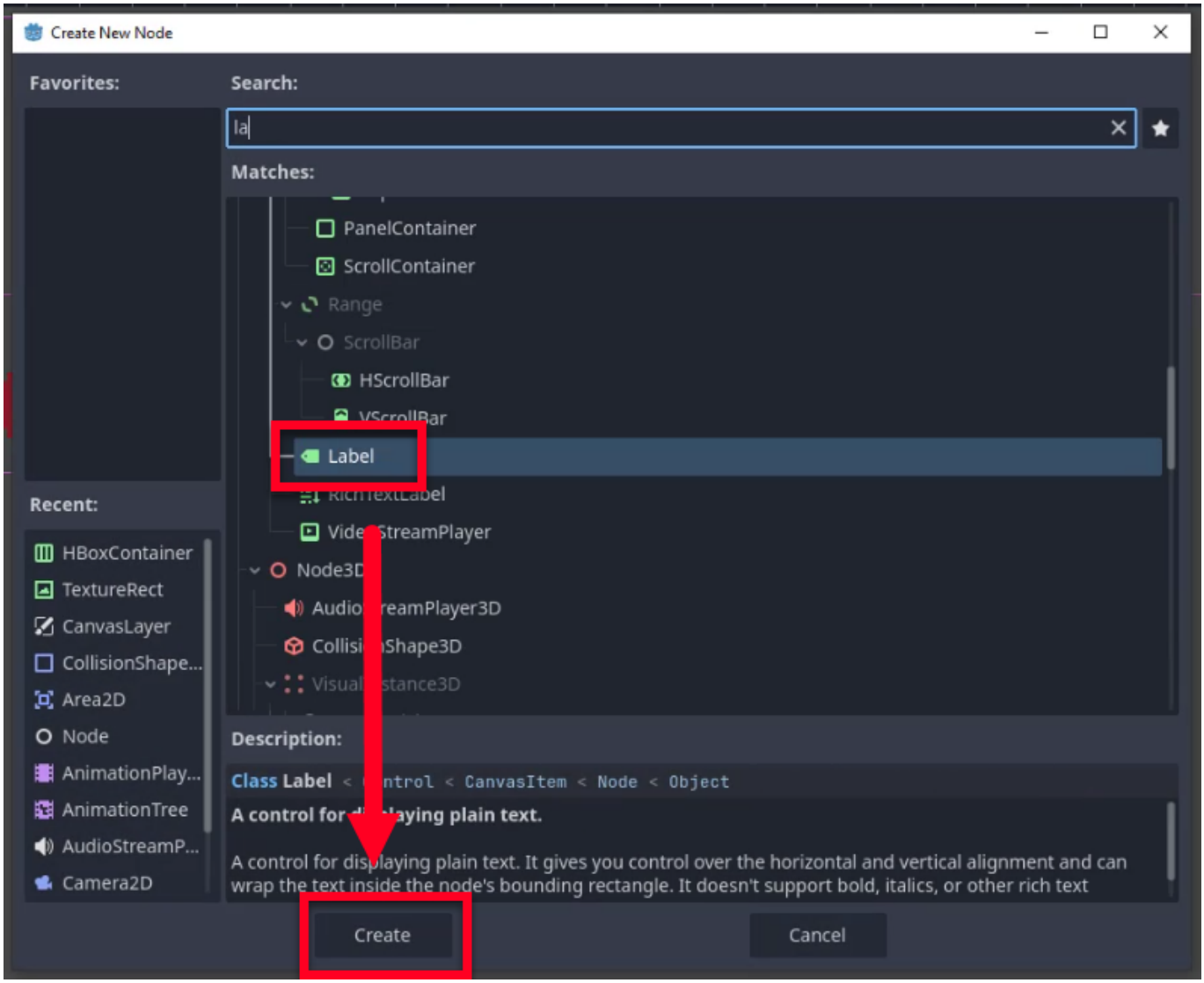


The HBoxContainer will automatically arrange the hearts horizontally.

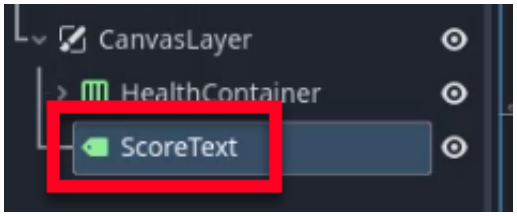


Creating the Score Text

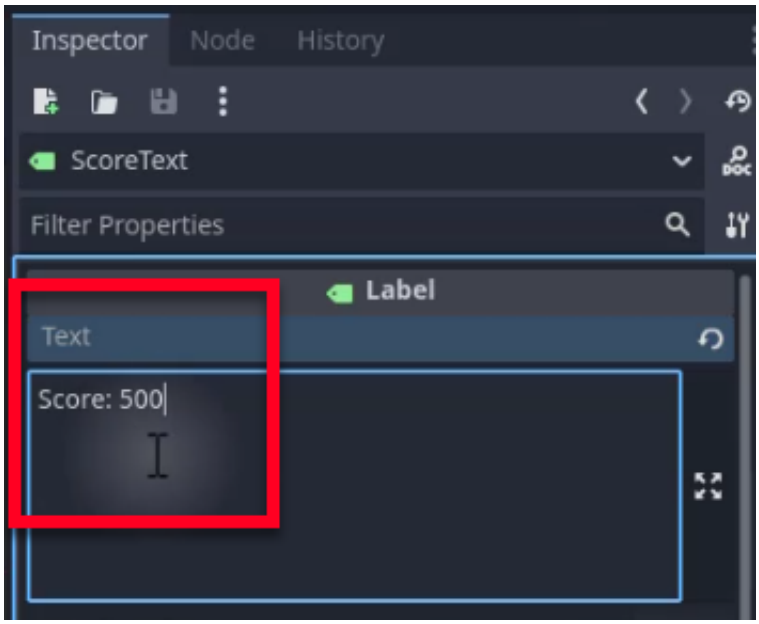
Next, we will create a label to display the player's score. Create a new node called Label as a child of CanvasLayer.



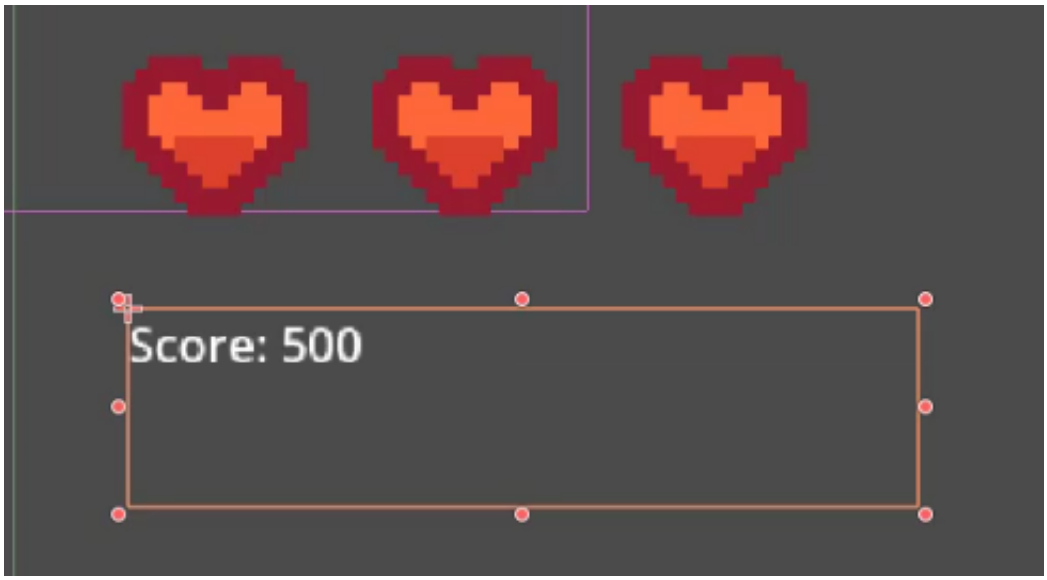
Rename this node to ScoreText.



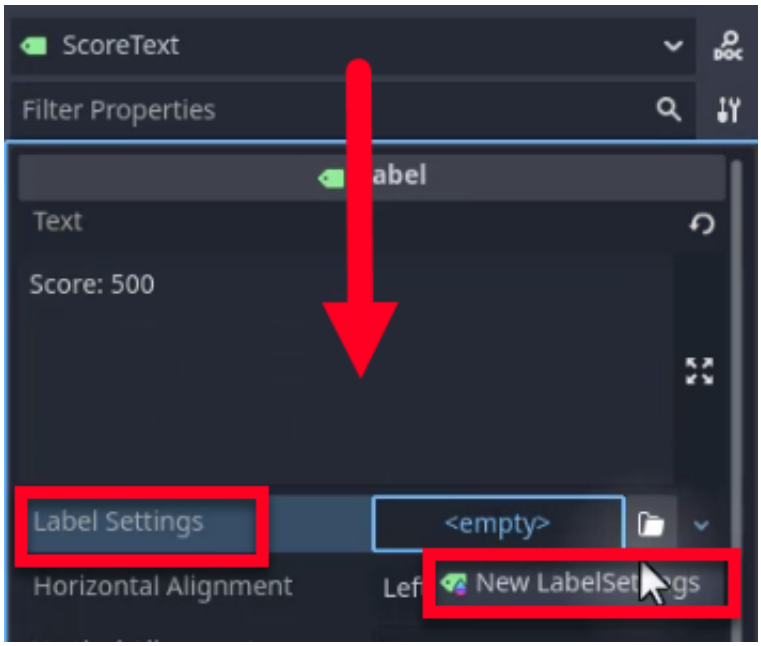
In the Inspector, set the text of ScoreText to Score: 500. We'll update this dynamically via script. For now we just need a placeholder.



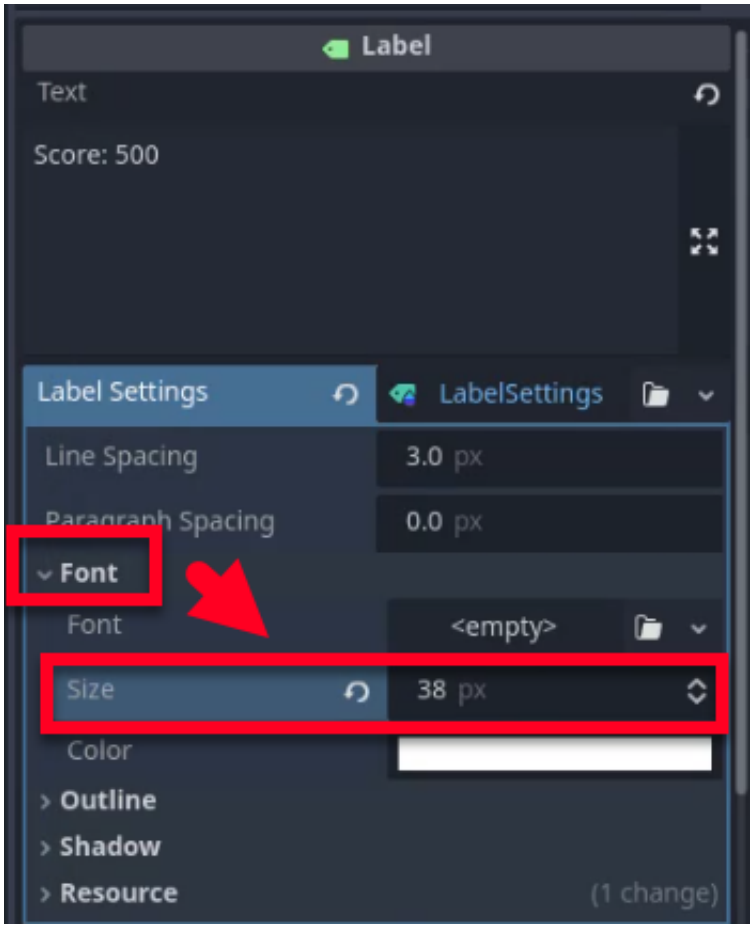
Right now, our text is a little small on our Canvas.



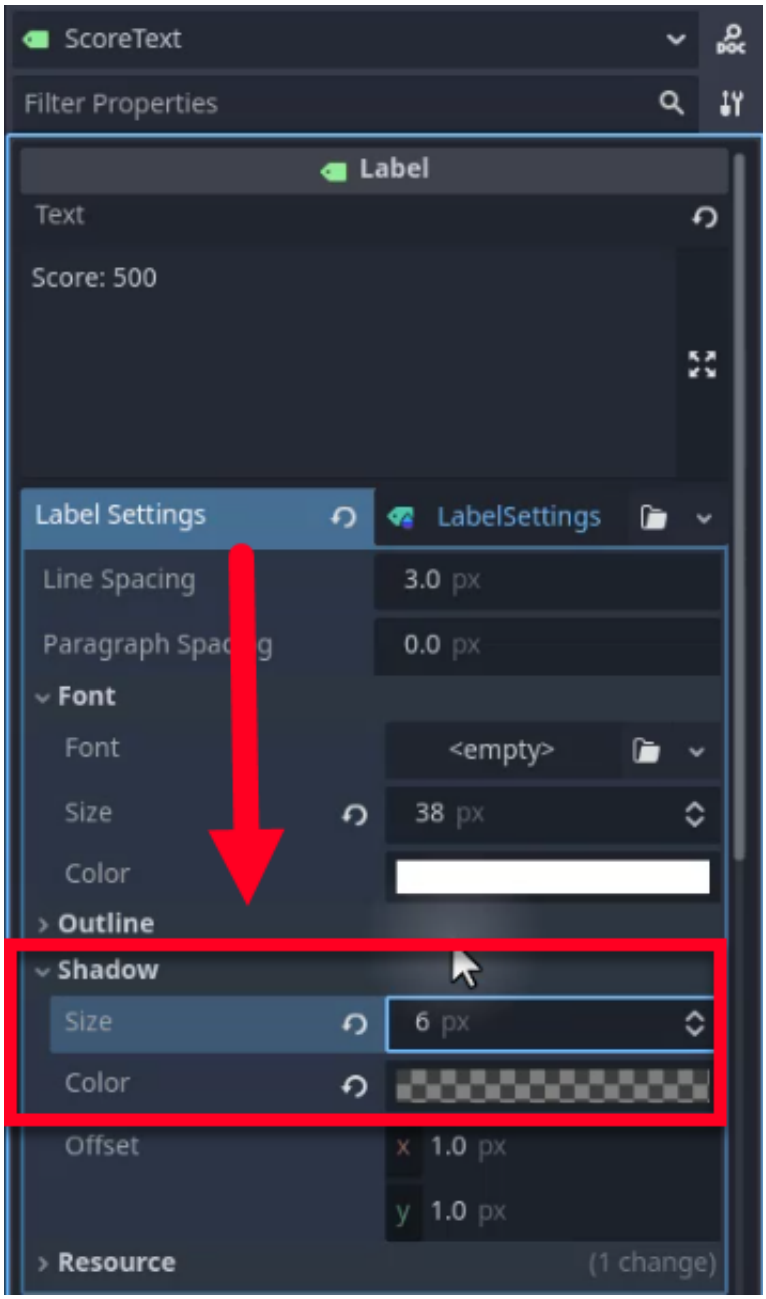
To fix this, we can adjust the font settings. First, let’s increase the font size. In the Inspector, go to the Label settings and select “New LabelSettings”.



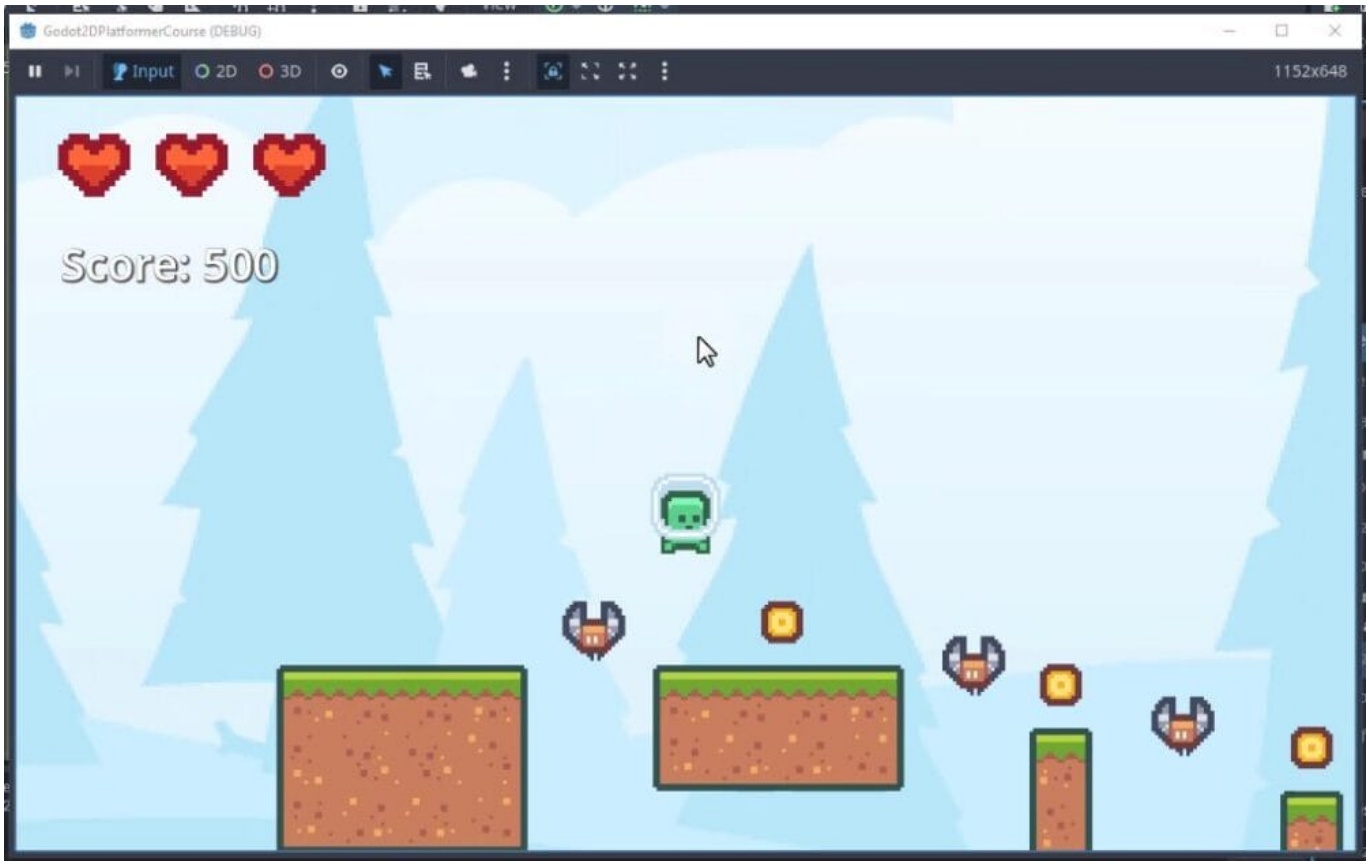
Click the LabelSettings to open the text setting properties. We can now locate the Font section and change its size to something like 38px.



Let’s also add a shadow to the text for better contrast. Go to the Shadow settings, and increase the alpha value to make the shadow more visible. Then, set the size to 6 for better contrast against the background.

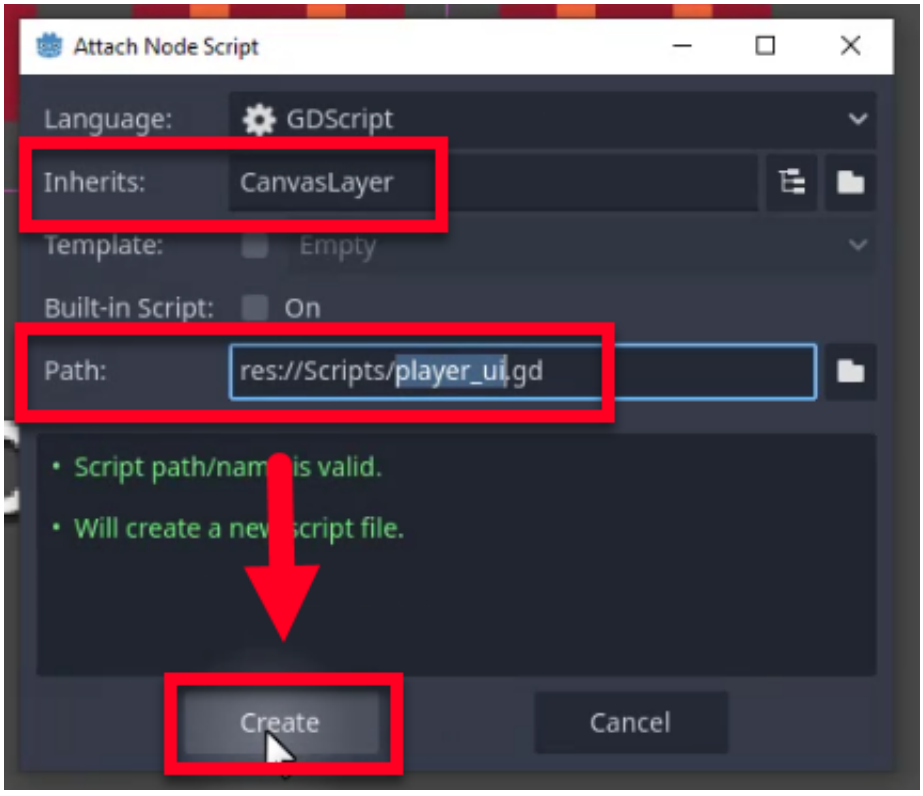


If you haven't done so already, position both the hearts and the score text in the upper left part of the CanvasLayer. If you press play, you should be able to see the UI now on the screen.



Managing UI with Scripts

To manage the UI elements, we will create a script called player_ui.gd and attach it to the CanvasLayer.



Updating UI Elements Efficiently

Instead of updating the UI every frame, we will use custom signals to update the UI only when the player's health or score changes. This approach is more efficient and prevents performance issues.

1. Open the player.gd script.
2. Define two new signals at the top of the script:

```
signal OnUpdateHealth (health : int)
signal OnUpdateScore (score : int)
```

3. Emit these signals when the player's health or score changes:
    - In the take_damage function, emit the OnUpdateHealth signal:

```
func take_damage (amount : int):
    health -= amount
    OnUpdateHealth.emit(health)

    if health <= 0:
        call_deferred("game_over")
```

    - In the increase_score function, emit the OnUpdateScore signal:

```
func increase_score (amount : int):
    PlayerStats.score += amount
    OnUpdateScore.emit(PlayerStats.score)
```

In the next lesson, we will begin filling in the player_ui.gd script to listen for these signals and update the UI accordingly.

In this lesson, we will continue working on our player UI script. This script will manage the display of the player's health and score, ensuring that the user interface updates in real-time as the player's health and score change.

Setting Up Variables

To begin, we need to create several variables that will help us manage the player's health and score display.

1. **Health Container:** This variable will reference the container that holds our health icons (hearts).
2. **Hearts Array:** This array will store the individual heart icons. An array is a variable that can hold multiple values. Think of it like a filing cabinet with different files, where each file represents a value. In our case, each value in the array will represent a heart icon.
3. **Score Text:** This variable will reference the label that displays the player's score.
4. **Player Reference:** This variable will reference the player object itself.

Let's start by declaring these variables in our script:

```
extends CanvasLayer

@onready var health_container = $HealthContainer
var hearts : Array = []

@onready var score_text : Label = $ScoreText

@onready var player = get_parent()
```

Ready Function

Next, we will create the _ready function. This function will initialize our variables and set up the necessary connections to update the UI when the player's health or score changes.

```
func _ready():
    pass  # We will fill this in later
```

Update Functions

We need two functions to update the health and score displays:

1. **_update_hearts:** This function will update the visibility of the heart icons based on the player's current health.
2. **_update_score:** This function will update the score text label with the player's current score.

Let's define these functions:

```
func _update_hearts(health : int):
    pass  # We will fill this in later

func _update_score(score : int):
    pass  # We will fill this in later
```

Connecting Signals

To ensure that our UI updates in real-time, we need to connect these functions to the player's signals. The player object emits signals when the health or score changes, and we will connect our update functions to these signals.

Besides connecting our signals, we will assign our hearts array to be the hearts in our health container. We'll also want to update our hearts and scores on load so they display correctly from the get-go.

In the _ready function, we will set up these aspects:

```
func _ready():
    hearts = health_container.get_children()

    player.OnUpdateHealth.connect(_update_hearts)
    player.OnUpdateScore.connect(_update_score)

    _update_hearts(player.health)
    _update_score(PlayerStats.score)
```

Updating Hearts

Now, let's implement the _update_hearts function. This function will loop through the hearts array and set the visibility of each heart icon based on the player's current health.

```
func _update_hearts(health : int):
    for i in range(len(hearts)):
        hearts[i].visible = i < health
```

Here's how it works:

- The loop runs through each index of the hearts array.
- For each heart, it sets the visibility to true if the index is less than the player's health, and false otherwise.

Updating Score

Finally, let's implement the _update_score function. This function will update the score text label with the player's current score.

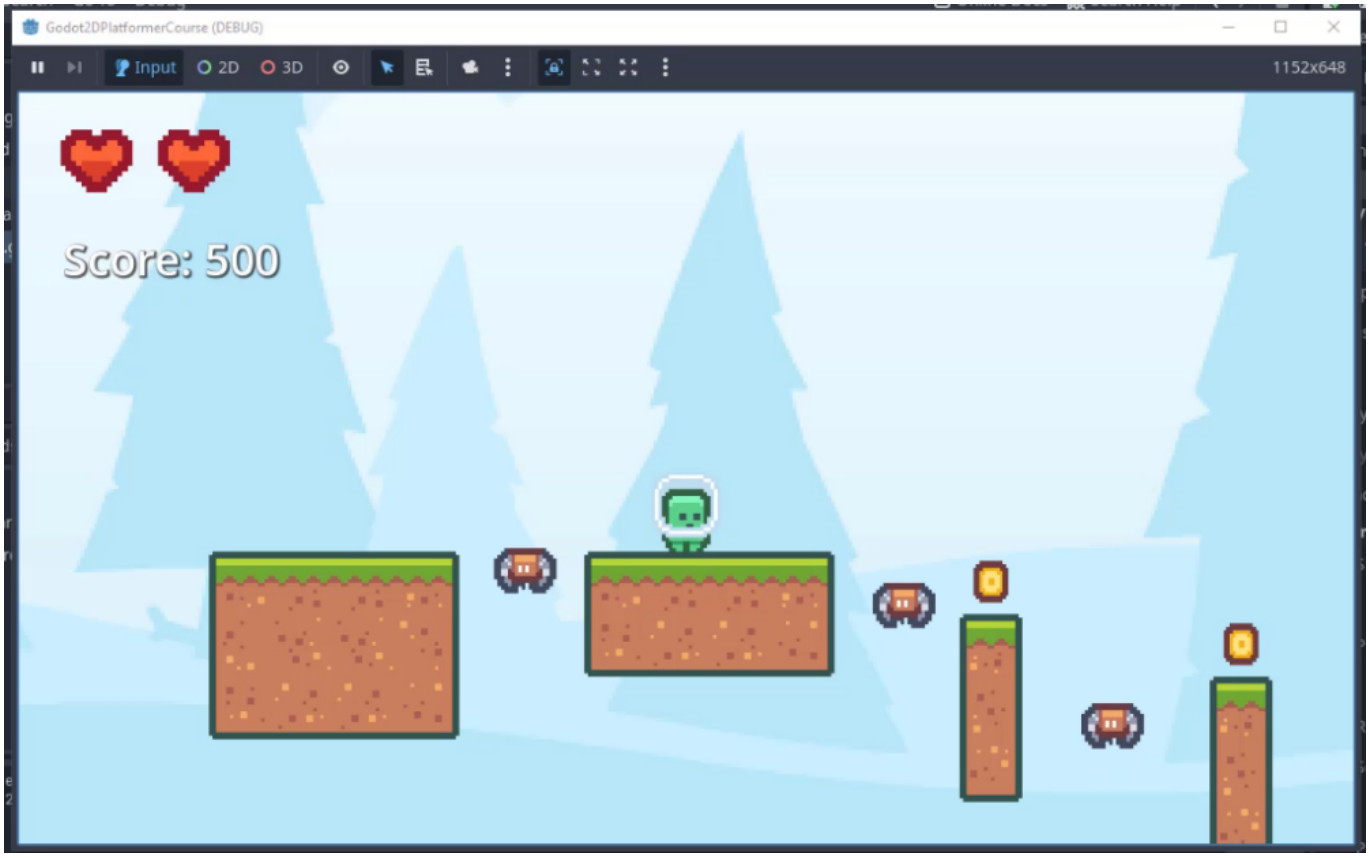
```
func _update_score(score : int):
    score_text.text = "Score: " + str(score)
```

Here's how it works:

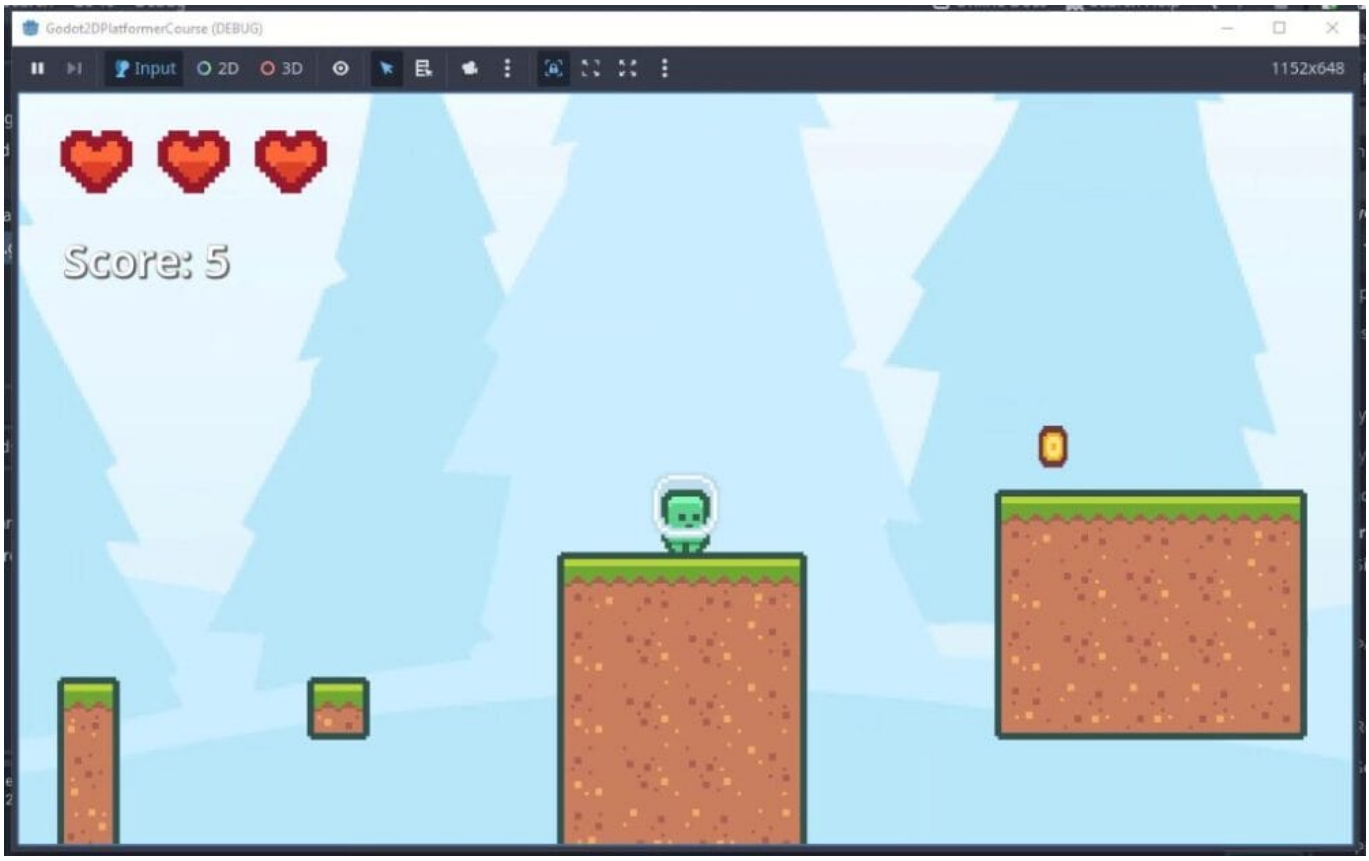
- The function converts the score to a string and concatenates it with the text "Score: ".
- It then sets the text property of the score text label to this new string.

Testing the UI

With our UI script complete, we can now test it in the game. As the player's health changes, the heart icons should update to reflect the current health.



Similarly, as the player's score changes, the score text label should update to display the current score.



Congratulations! You have successfully implemented a dynamic UI that updates in real-time based on the player's health and score. In the next lesson, we will enhance the "game feel" of our damage to provide the player with more feedback.

In this lesson, we will enhance the player's damage feedback by implementing a damage flash effect and a screen shake effect. Currently, when the player takes damage, it is only indicated in the top left corner of the screen. To make the damage more visually impactful, we will make the player flash red briefly and add a screen shake effect.

Implementing the Damage Flash Effect

First, let's create a new function called _damage_flash in the player script. This function will change the player's sprite color to red and then revert it back to its normal color after a short delay.

Here's how you can do it:

1. Open the player script.
2. Add the following function at the bottom of the script:

```
func _damage_flash():
    sprite.modulate = Color.RED
    await get_tree().create_timer(0.05).timeout
    sprite.modulate = Color.WHITE
```

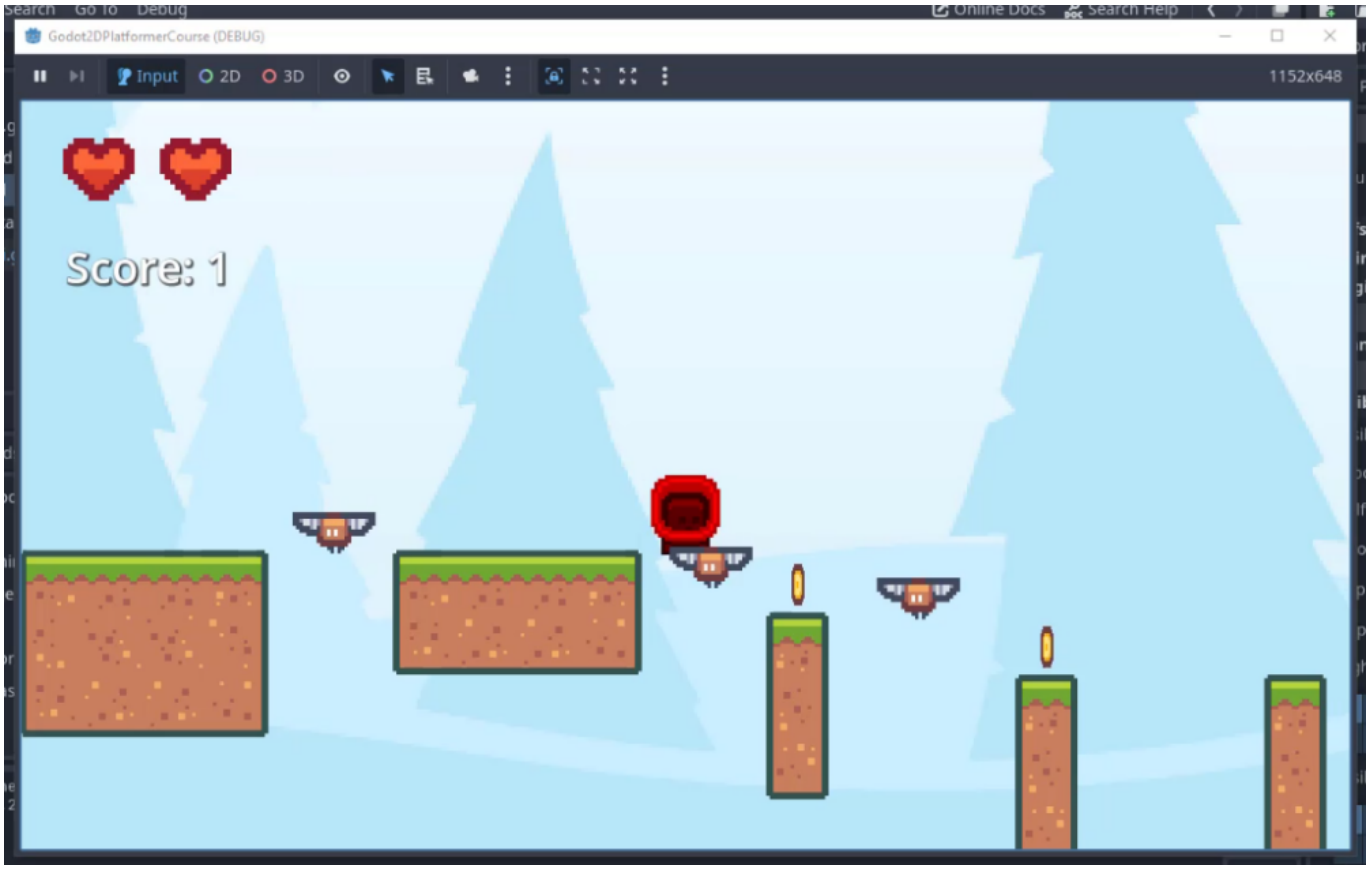
In this function:

- We set the sprite's modulate property to Color.RED to make the player flash red.
- We then use the await keyword to pause the function for 0.05 seconds.
- After the pause, we set the sprite's modulate property back to Color.WHITE to revert the player to its normal color.

Next, we need to call this function whenever the player takes damage. Inside the take_damage function, add the following line:

```
_damage_flash()
```

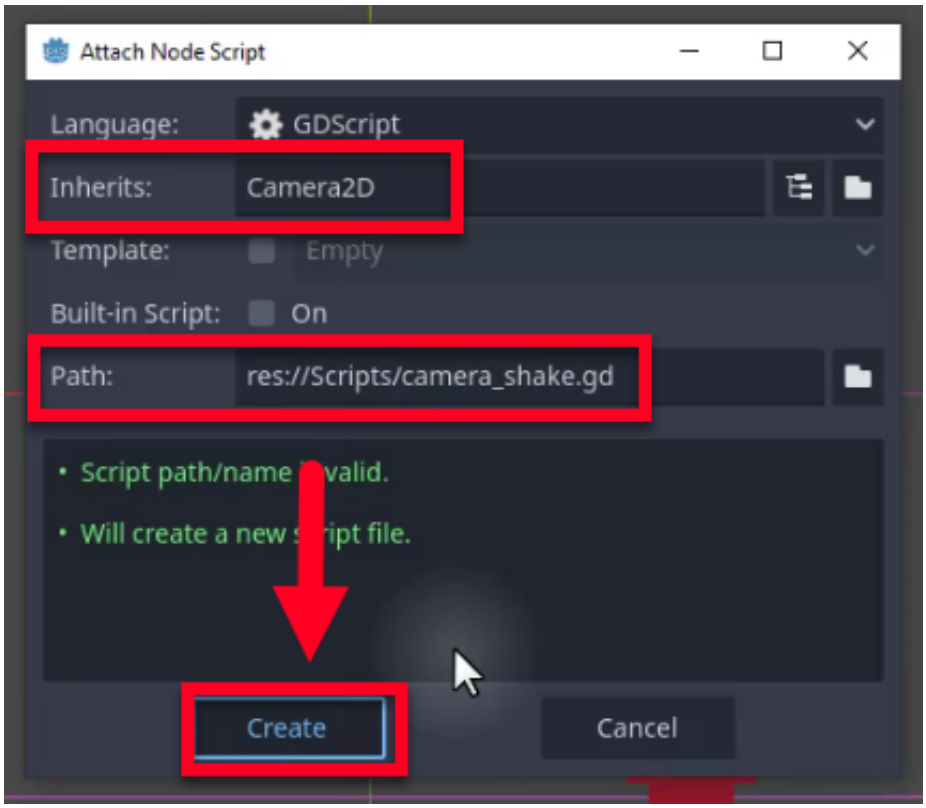
Now, when the player takes damage, the player sprite will flash red briefly, providing a clear visual indication of the damage taken.



Implementing the Screen Shake Effect

Next, let’s add a screen shake effect that triggers whenever the player takes damage. This effect will make the camera shake briefly, adding to the impact of taking damage.

First, we need to create a new script for our camera. Select the Camera2D node in the player scene and attach a new script to the Camera2D node. name it camera_shake.gd.



Open the camera_shake.gd script and add the following code:

```
extends Camera2D

var intensity : float = 0

func _ready():
    get_parent().OnUpdateHealth.connect(_damage_shake)

func _damage_shake(health : int):
    intensity = 3

func _process(delta):
    if intensity > 0:
        intensity = lerpf(intensity, 0, delta * 10)
        offset = _get_random_offset()

func _get_random_offset() -> Vector2:
    var x = randf_range(-intensity, intensity)
    var y = randf_range(-intensity, intensity)
    return Vector2(x, y)
```

In this script:

- We define a variable intensity to control the strength of the shake effect.
- In the _ready function, we connect the OnUpdateHealth signal to the _damage_shake function.
- In the _damage_shake function, we set the intensity to 3 whenever the player takes damage.

- In the _process function, we gradually reduce the intensity to 0 over time using the lerpf function.
- We also set the camera's offset to a random value based on the current intensity, creating the shake effect.
- The _get_random_offset function returns a random offset vector based on the current intensity.

Now if you test the game, the camera should shake to help provide more "juice" for when the player is hit.

Preventing Infinite Falling

Finally, let's add a simple check to prevent the player from falling infinitely. In the player script, modify the _process function to include the following check:

```
if global_position.y > 200:
    game_over()
```

In this check, if the player's global_position.y exceeds 200, we call the game_over function to end the game.

With these implementations, you have now added a damage flash effect, a screen shake effect, and a fall prevention check to your game. These enhancements provide clear visual feedback to the player and improve the overall gameplay experience.

Godot 2D Platformer - UI & Visuals [2025]

1. What is the role of the CanvasLayer node in the UI setup?

a. It is used for positioning background elements
b. It ensures that UI elements are always visible, regardless of the player's position in the game world
c. It manages player movement
d. It handles animations for UI elements

2. True or False: A TextureRect node allows us to display sprites in the UI, such as hearts for health.

a. True
b. False

3. Why is an HBoxContainer used in the health display setup?

a. It automatically aligns the hearts vertically
b. It holds the background image for the health display
c. It ensures that hearts are spaced evenly and aligned horizontally
d. It stores player score data

4. True or False: It’s always better for performance to check UI updates frame-by-frame instead of using custom signals.

a. True
b. False

5. True or False: We can use the modulate property to change the player’s sprite color to red briefly to indicate damage.

a. True
b. False

6. What effect does the intensity variable in the camera_shake.gd script control?

a. The background color
b. The speed of the player movement
c. The amount of camera shake when the player takes damage
d. The size of the health hearts

7. Why are we using lerpf(intensity, 0, delta * 10) for the camera shake effect?

a. To gradually reduce the shake intensity over time
b. To randomly reset the intensity each frame
c. To make the shake effect permanent
d. To increase the shake intensity after each hit

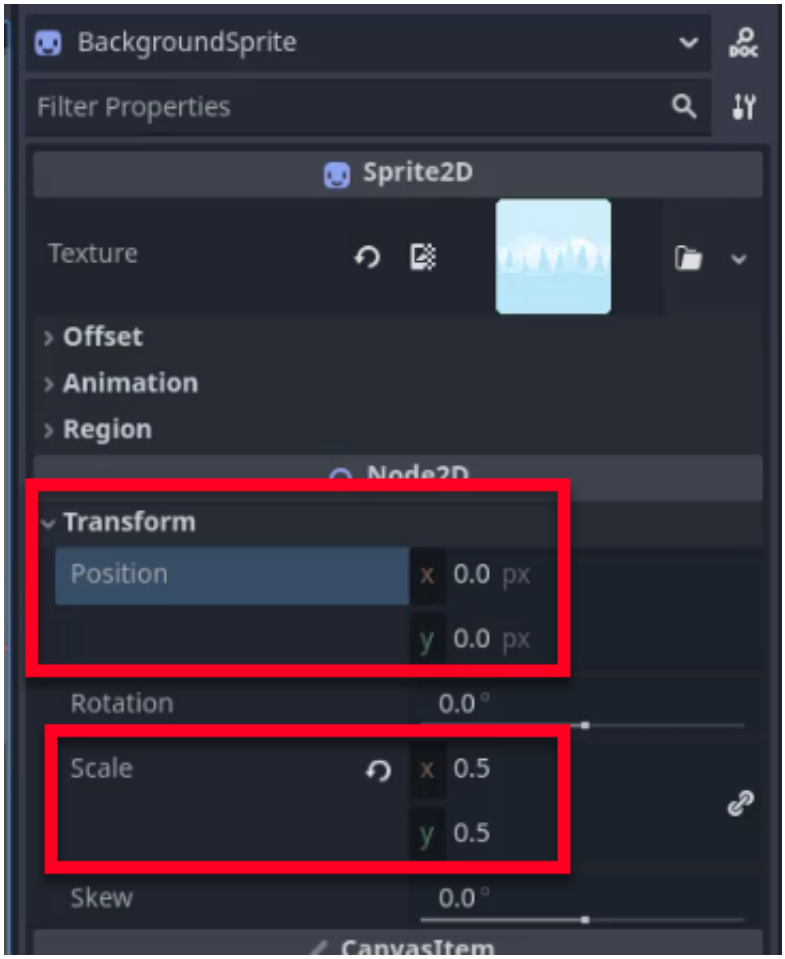
8. True or False: The ‘if global_position.y > 200: game_over()’ condition check prevents the player from falling infinitely.

a. True
b. False

In this lesson, we will focus on setting up the background for our 2D platformer game. While we already have a background in place, it needs some adjustments to look more polished and dynamic. Specifically, we will address two main issues: the background being clipped on the sides and the static nature of the background. By the end of this lesson, you will have a background that moves at a different rate than the foreground, creating a parallax effect that simulates depth.

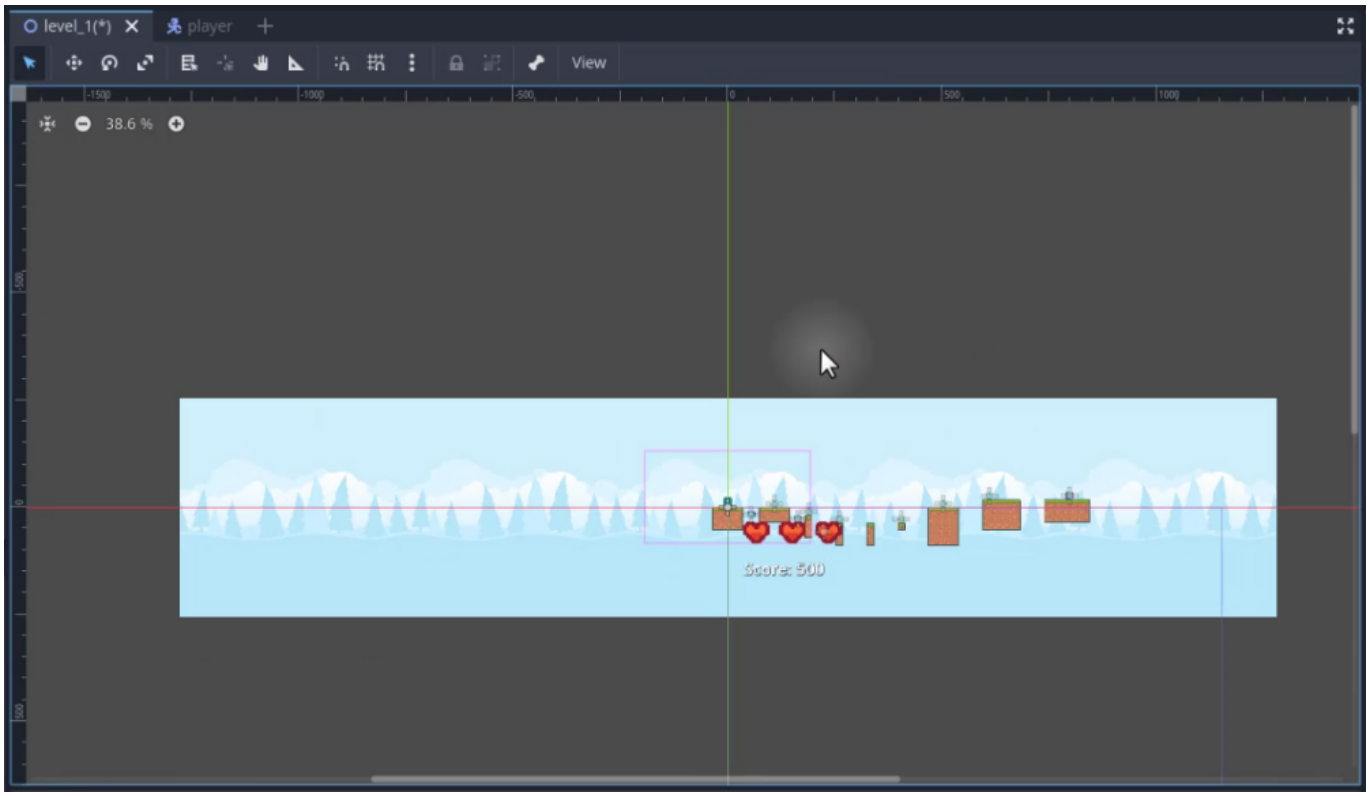
Resizing and Positioning the Background

First, let's resize the background to make it look more visually appealing. Select the background sprite in your scene. In the Inspector, go to the Transform section. Let's set the Scale to 0.5. This will make the background less pixelated and more concise. We'll also set the Position to (0, 0) to center the background.



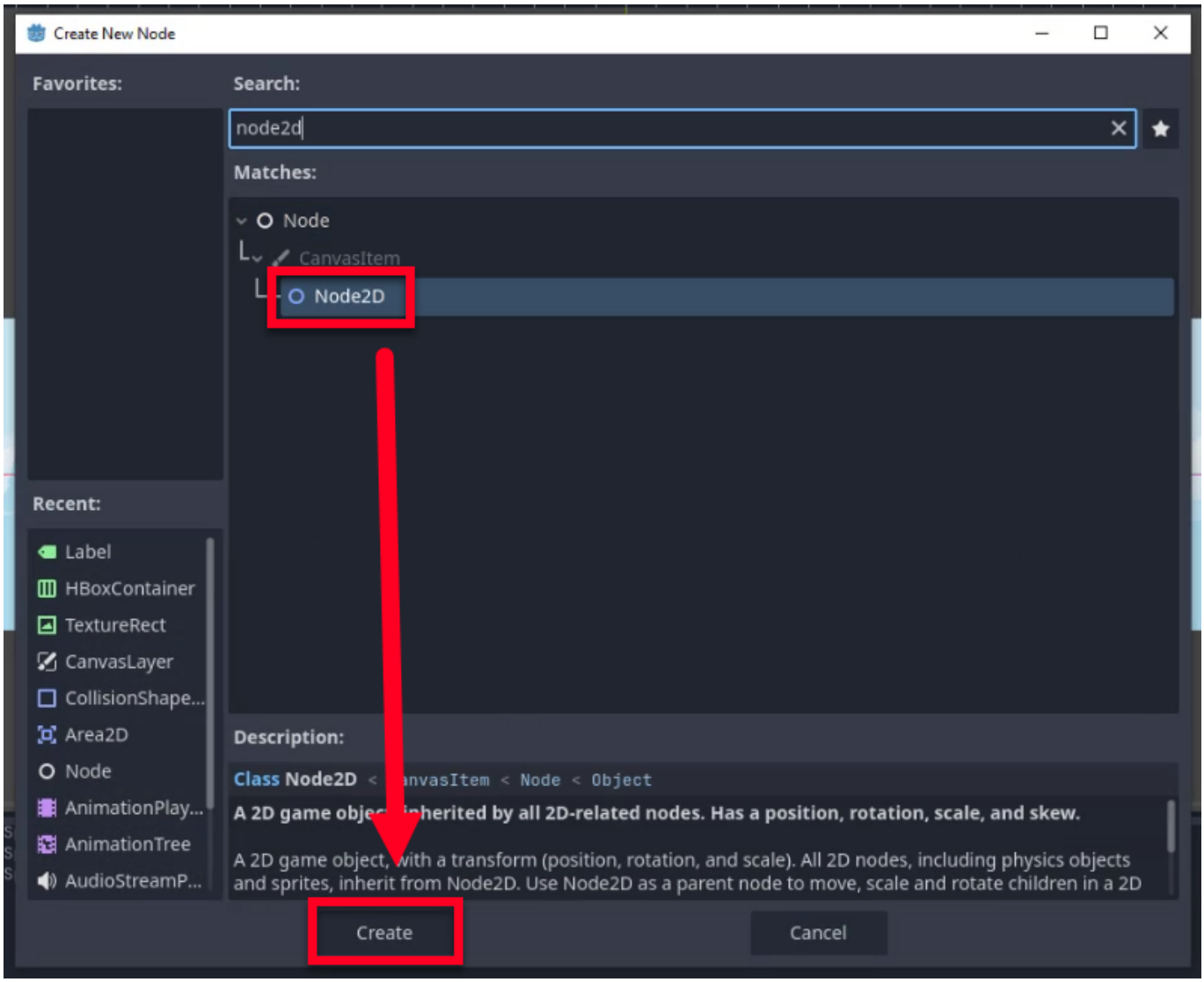
Creating a Tiling Background

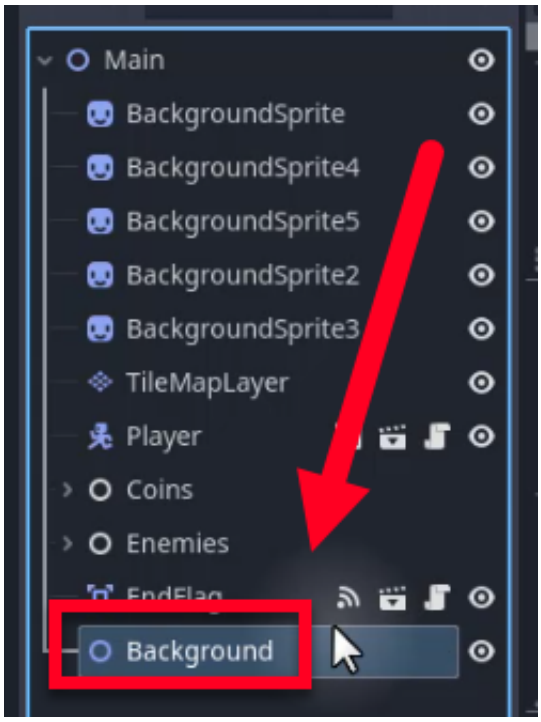
Next, we need to create multiple copies of the background to form a long strip that covers the entire level. This will ensure that the background can move smoothly as the player progresses through the level. To do this, simply duplicate the background sprite and move the duplicate to the right by holding the Shift key and adjusting its position. For the included backgrounds, use an offset of 512 units. Repeat the duplication process to cover the entire level. Make sure to adjust the position of each duplicate accordingly (e.g., +512, -512, etc.).



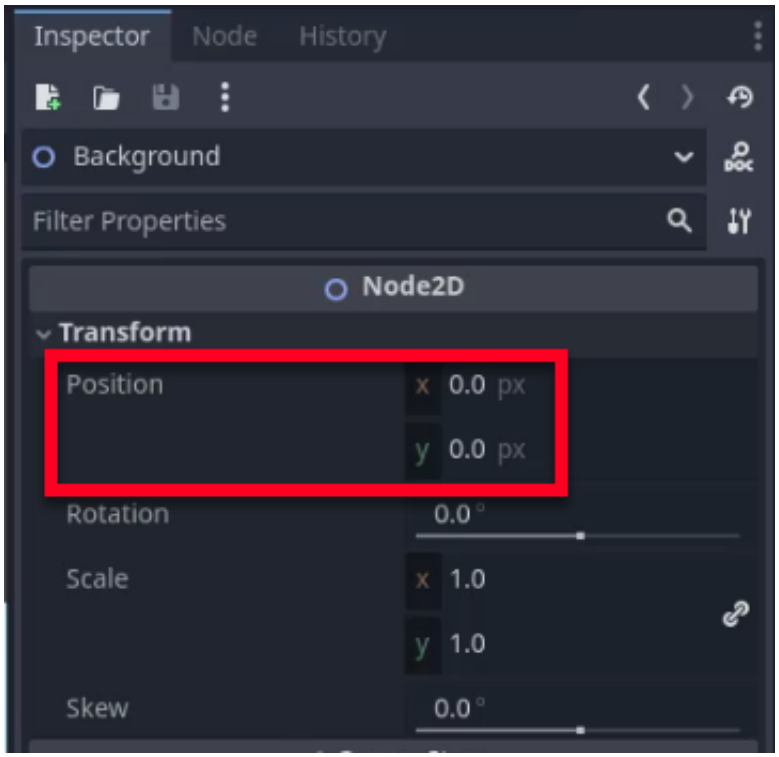
Organizing the Background Sprites

To manage the background sprites more efficiently, we will group them under a single node. Create a new Node2D and name it Background.

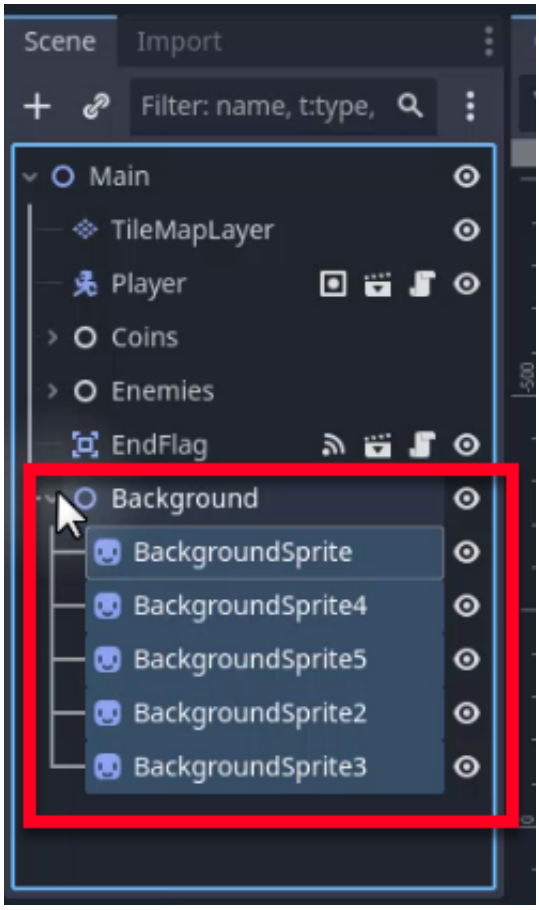




Set the position of the Background node to (0, 0).

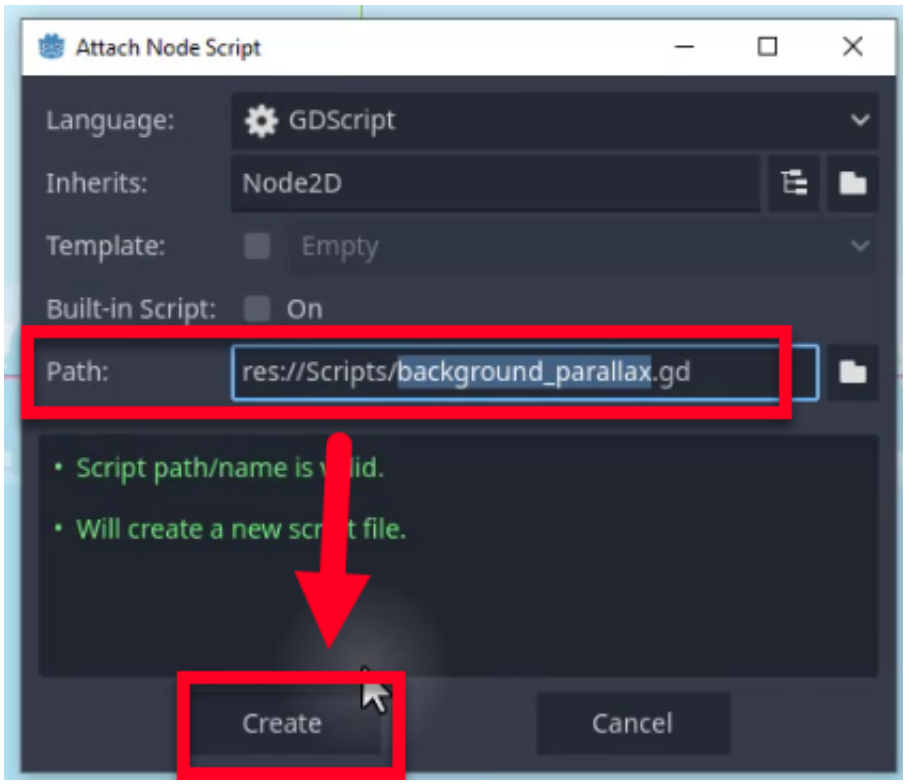


Finally, drag all the background sprite nodes into the Background node as children.



Implementing Parallax Effect

Now, let's implement the parallax effect. This will make the background move at a different rate than the foreground, creating a sense of depth. Create a new script for the Background node and name it background_parallax.gd. As usual, save the script in the Scripts folder.



Open the script and add the following code:

```
extends Node2D

var parallax : float = 0.7
@onready var player = $"../Player"

func _process (delta):
 global_position = player.global_position * parallax
```

Here's what the code does:

- parallax: This variable determines the rate at which the background moves in relation to the player. A value of 0.7 means the background will follow the player's movement at 70% of the player's speed.
- player: This variable references the player node in the scene.
- _process: This function runs every frame and updates the background's position based on the player's position and the parallax value.

Testing the Parallax Effect

Finally, let's test the parallax effect to ensure it works as expected.

1. Save the script and return to the Godot editor.
2. Press the play button to run the scene.
3. Move the player character and observe the background. It should now move at a different rate than the foreground, creating a parallax effect that simulates depth.

Congratulations! You have successfully set up a dynamic background with a parallax effect for your 2D platformer game. This will enhance the visual appeal and immersion of your game. In the next lessons, we will continue to build on this foundation to create a more engaging and polished gaming experience.

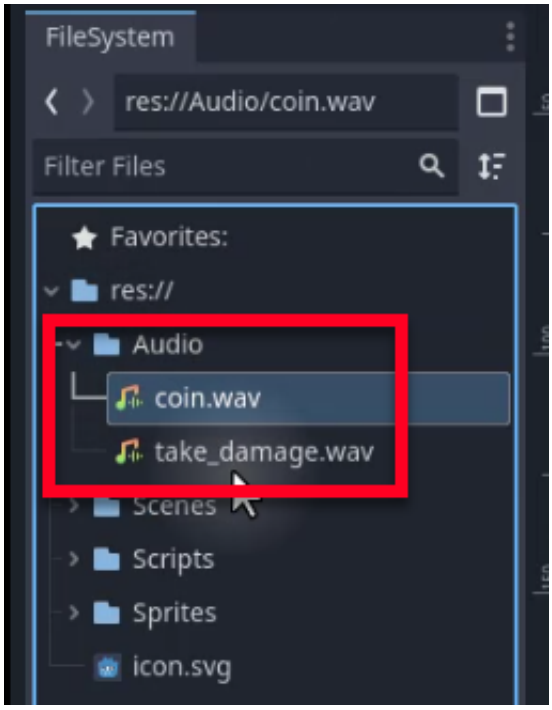
In this lesson, we will be implementing audio in our game. Specifically, we will add sound effects that play when certain events occur, such as collecting a coin or taking damage. By the end of this lesson, you will have a basic understanding of how to integrate sound effects into your Godot project.

About Audio

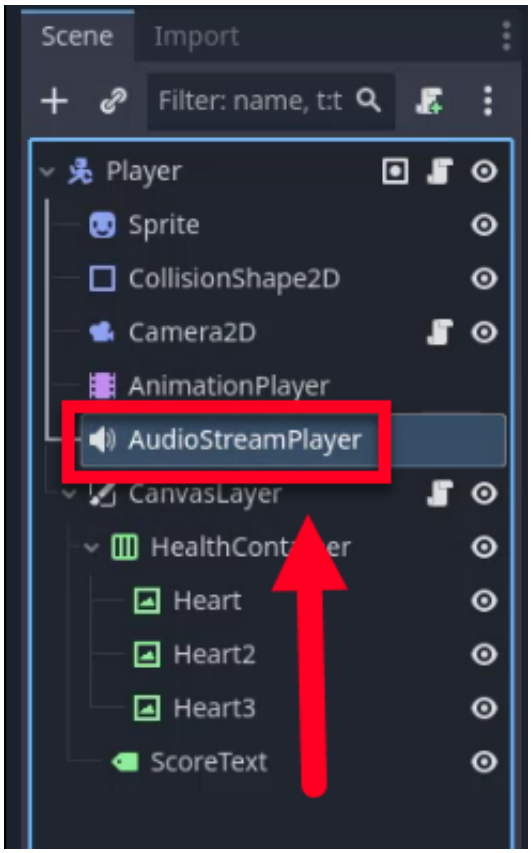
To begin, let's navigate to our audio folder. You will notice that we have two sound effects:

- Coin.wav
- Take Damage.wav

Our goal is to play these sound effects when the corresponding events happen in the game.



If you open the Player scene, you will see that we already have an AudioStreamPlayer node. This node will be responsible for playing our sound effects.



Modifying the Player Script

In order to use our audio, we need to modify our player script to handle the audio. Open the player.gd script and follow these steps:

Step 1: Declare the AudioStreamPlayer Node

Add a new variable to reference the AudioStreamPlayer node:

```
@onready var audio : AudioStreamPlayer = $AudioStreamPlayer
```

Step 2: Preload the Sound Effects

Create variables to preload the sound effects. This ensures that the sound effects are loaded into memory before they are needed, preventing any delays or glitches when they are played:

```
var take_damage_sfx : AudioStream = preload("res://Audio/take_damage.wav")
var coin_sfx : AudioStream = preload("res://Audio/coin.wav")
```

Step 3: Create the Play Sound Function

Create a function to play the sound effects. This function will set the stream property of the AudioStreamPlayer node to the sound effect we want to play and then call the play method:

```
func play_sound(sound : AudioStream):
    audio.stream = sound
```



```
    audio.play()
```

Step 4: Call the Play Sound Function

Finally, call the play_sound function in the appropriate places within our script:

1. In the take_damage function, add the following line to play the take damage sound effect:

```
func take_damage(amount : int):
    health -= amount
    OnUpdateHealth.emit(health)
    _damage_flash()
    play_sound(take_damage_sfx)

    if health <= 0:
        call_deferred("game_over")
```

2. In the increase_score function, add the following line to play the coin sound effect:

```
func increase_score(amount : int):
    PlayerStats.score += amount
    OnUpdateScore.emit(PlayerStats.score)
    play_sound(coin_sfx)
```

Testing the Implementation

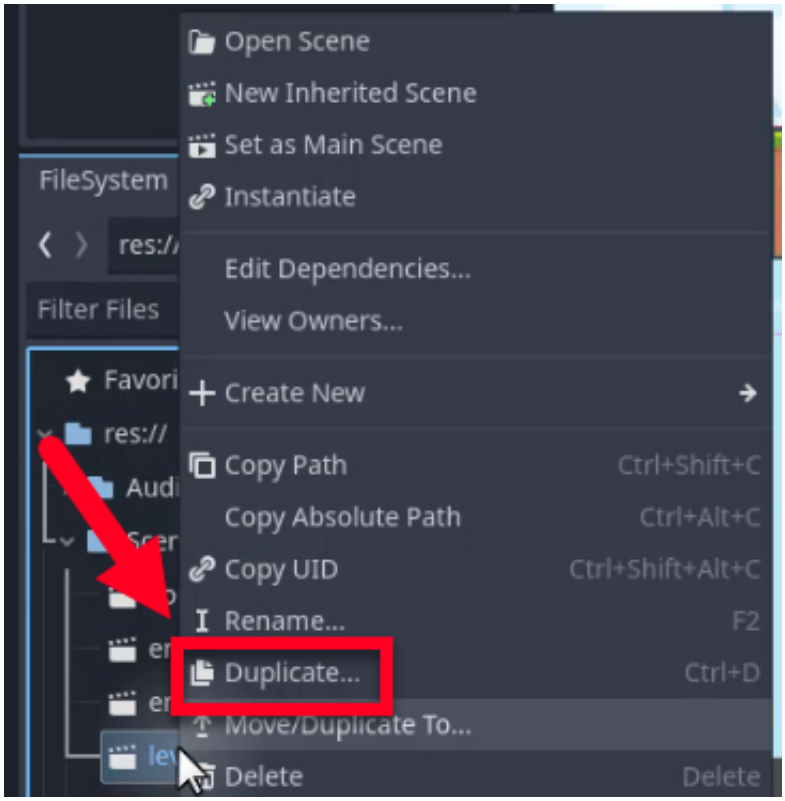
Now that we have implemented the audio, it’s time to test our changes. Press the play button and try collecting a coin or taking damage. You should hear the corresponding sound effects play.

In the next lesson, we will continue to enhance our game by adding more features and improving the existing ones.

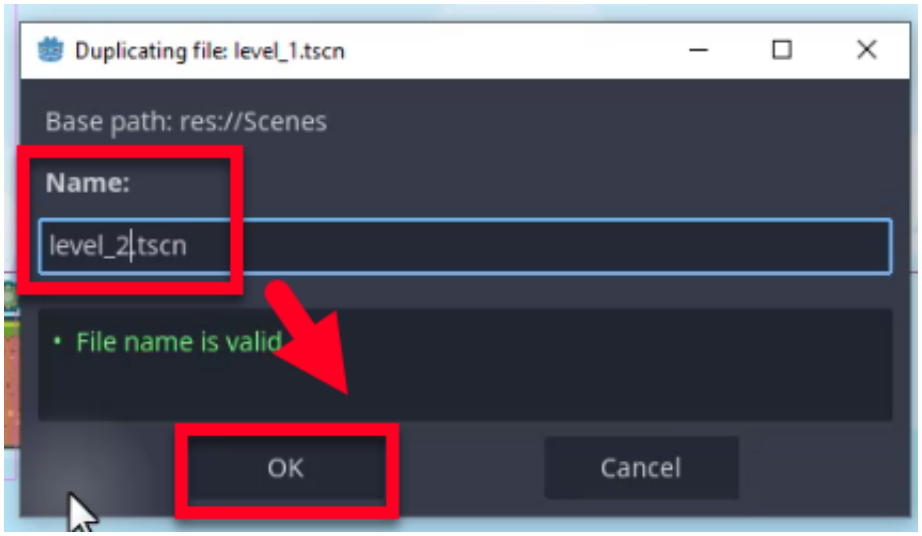
In this lesson, we will focus on creating a second level for our game. This process involves duplicating the first level and modifying it to create a unique and engaging new environment. Let's dive into the steps required to achieve this.

Duplicating the First Level

The easiest way to create a second level is by duplicating the first level and then modifying the nodes. Navigate to the **Scenes** folder, right-click on **Level 1**, and select **Duplicate**.



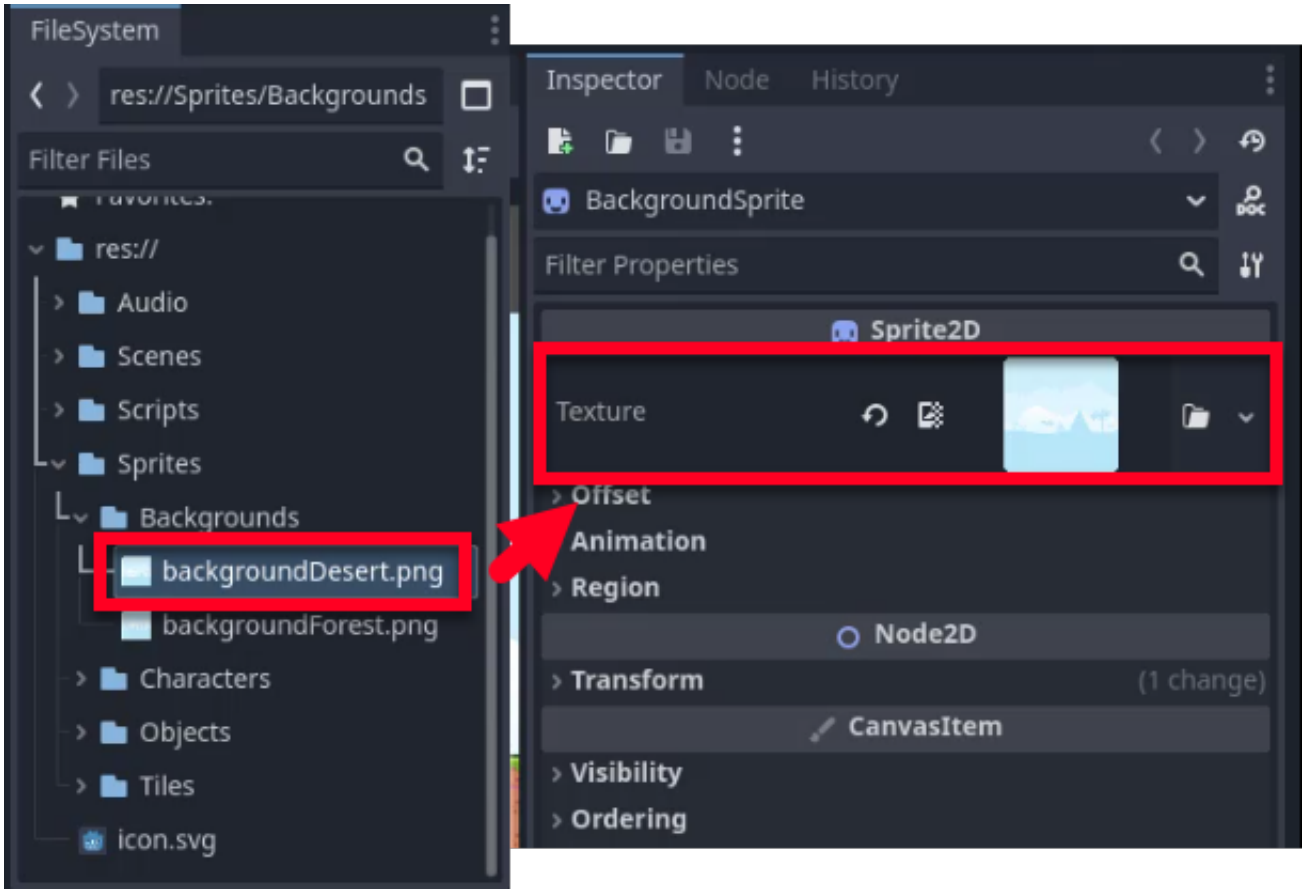
Rename the duplicated scene to **Level 2** and click **OK**.



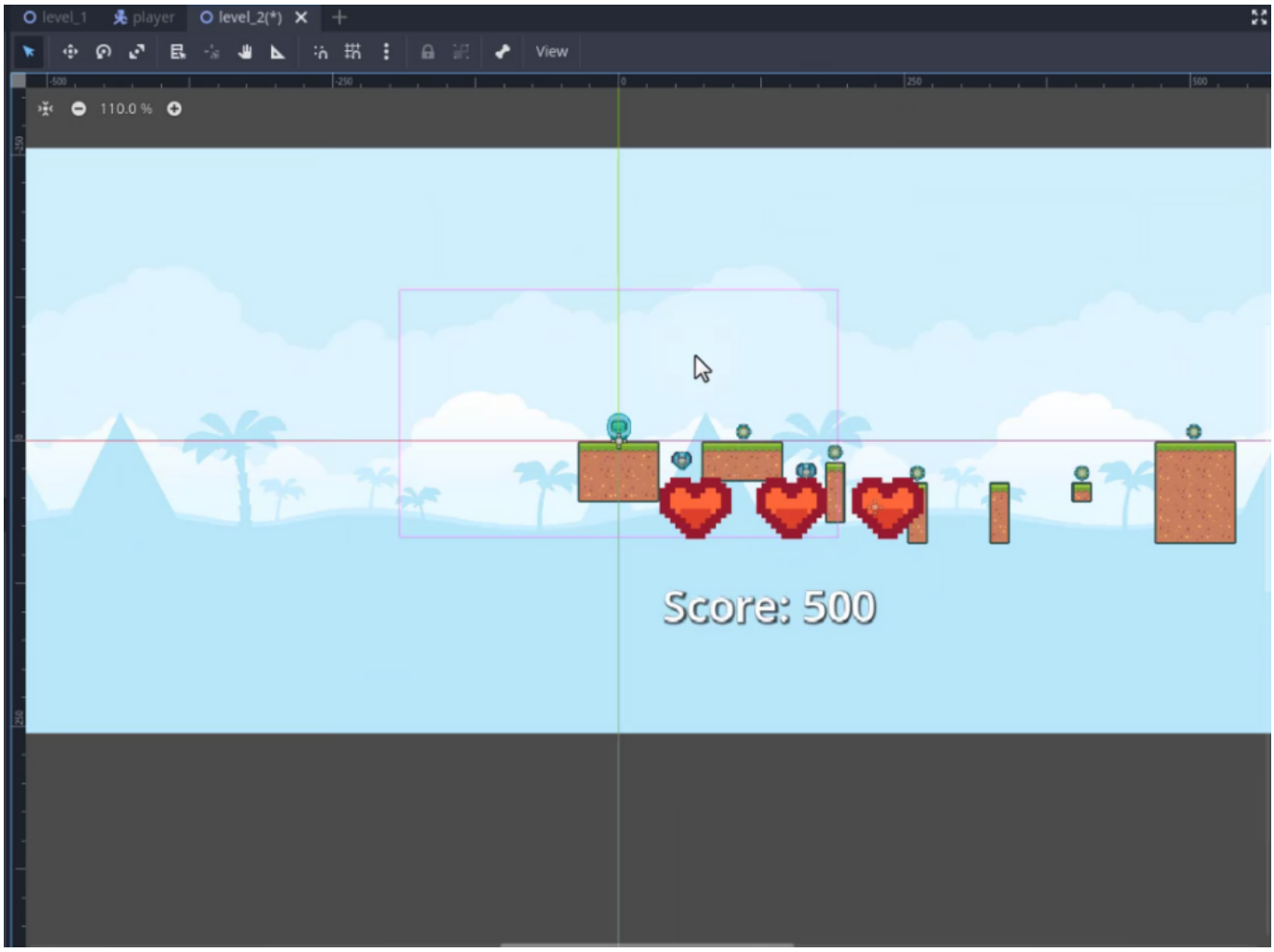
By doing this, you will have a copy of Level 1 that you can modify to create Level 2.

Modifying the Background

To make Level 2 visually distinct from Level 1, we will change the background. Select the background sprites and replace them with the desert background sprites from the **Sprites** folder.

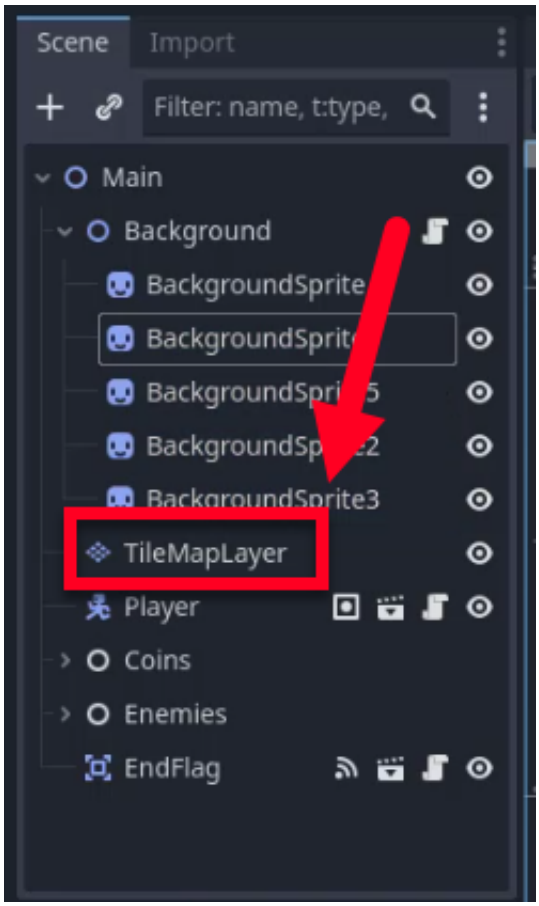


This will give Level 2 a desert theme, differentiating it from the forest theme of Level 1.

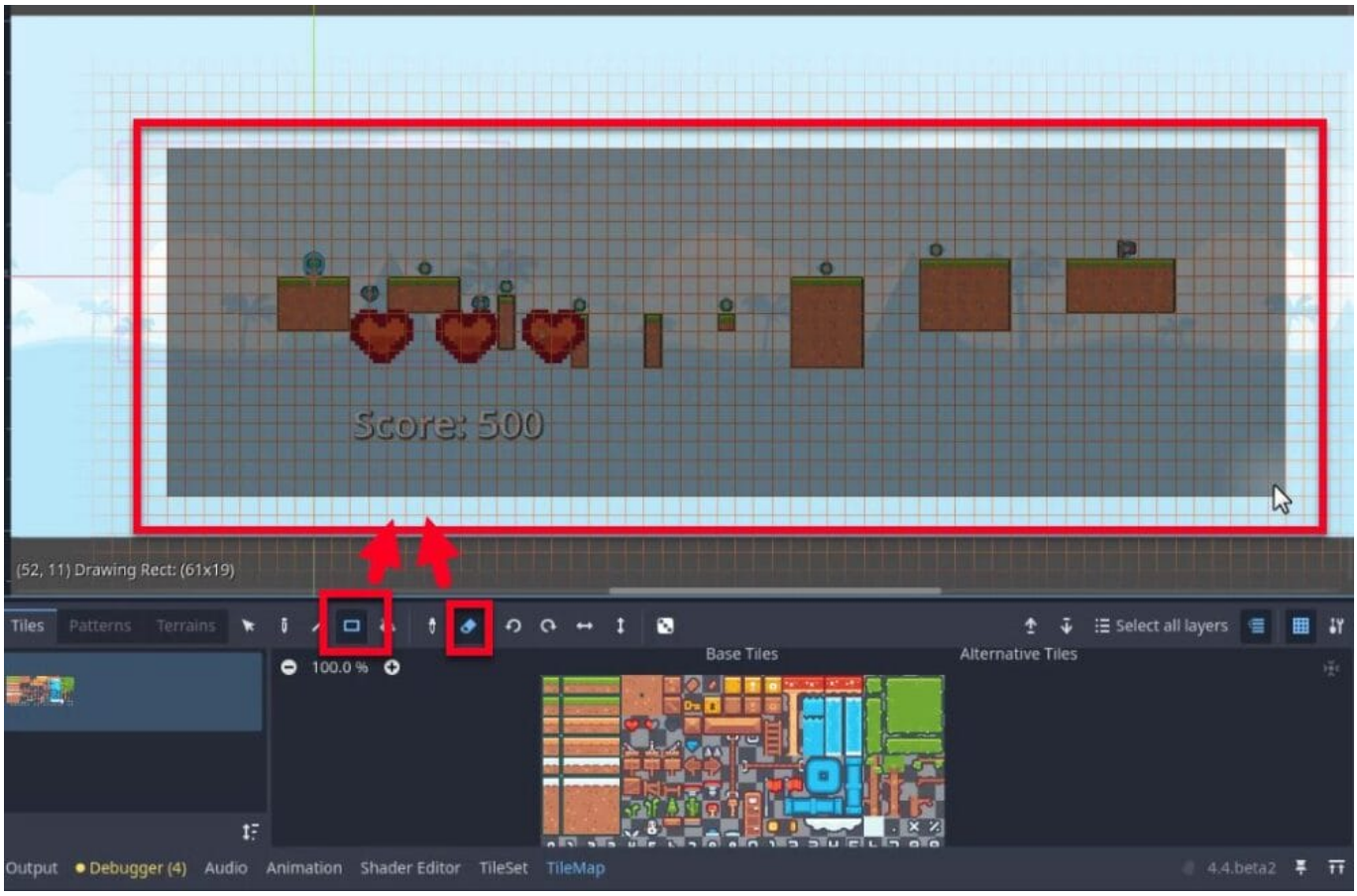


Editing the Tile Map

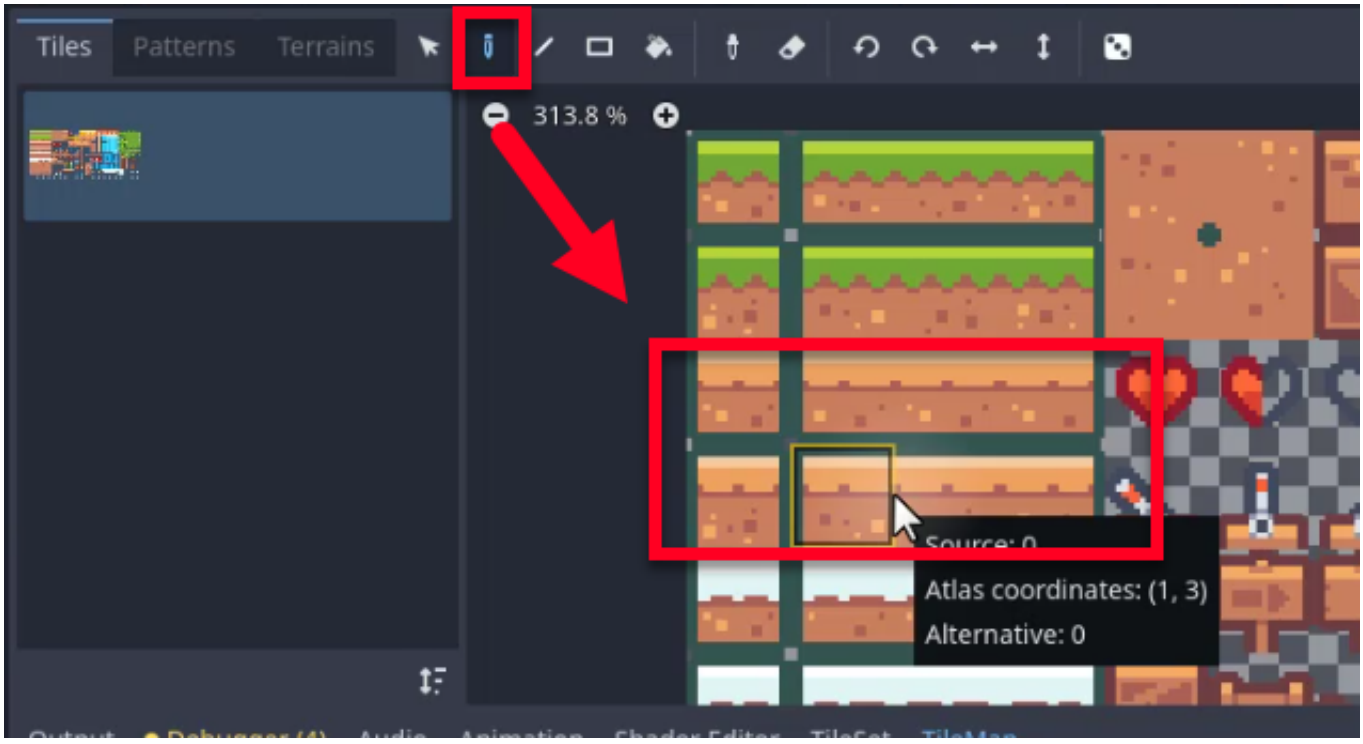
Next, we will edit the tile map to create a new layout for Level 2. First, select the **TileMapLayer**.



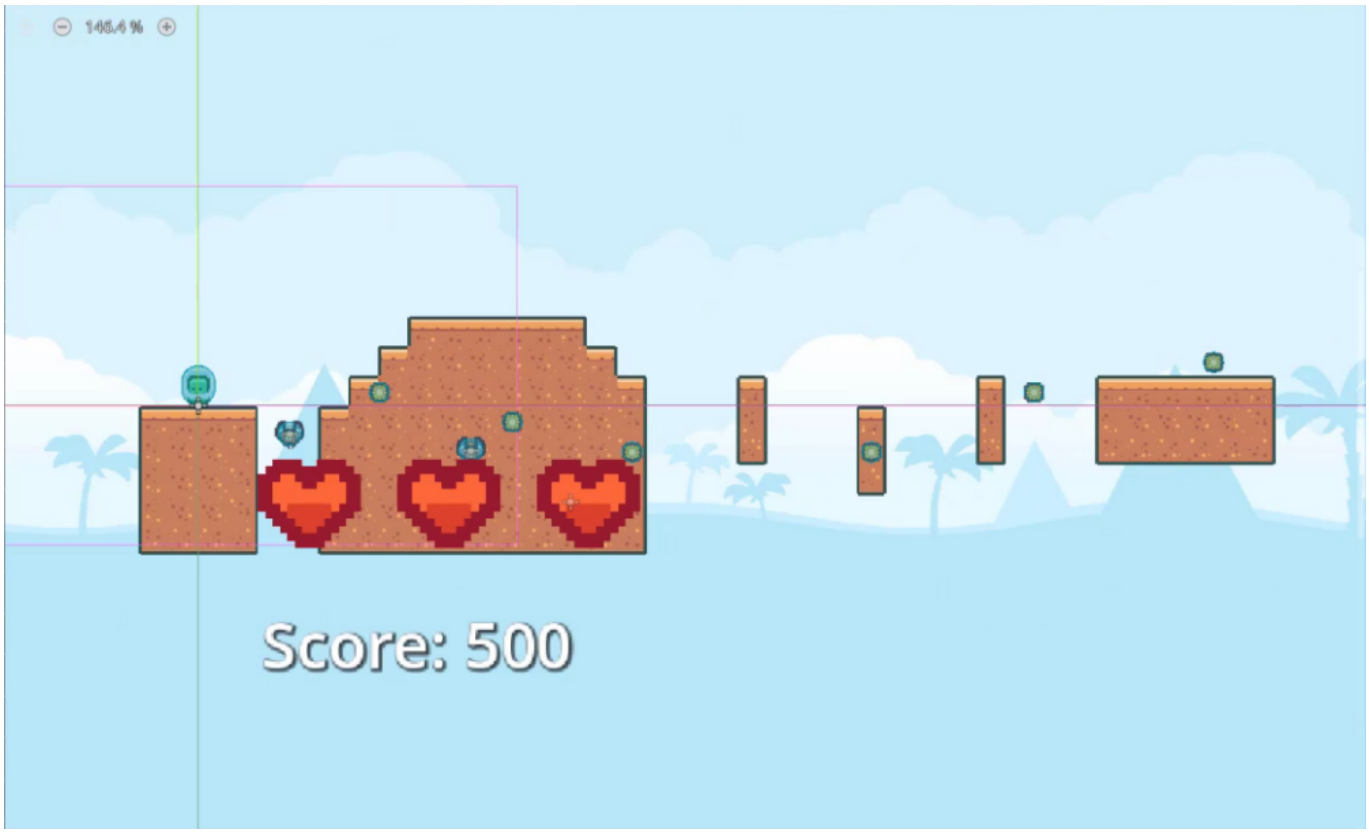
Use the eraser tool in the tile editor to delete the existing tiles. You can make this easier by selecting the rectangle options to delete many tiles at once.



Switch to the pen tool and create a new level layout using the desert tiles. This step ensures that the level design is unique and fits the desert theme.



The level design is up to you, so feel free to experiment until you get a level you feel is fun to play.

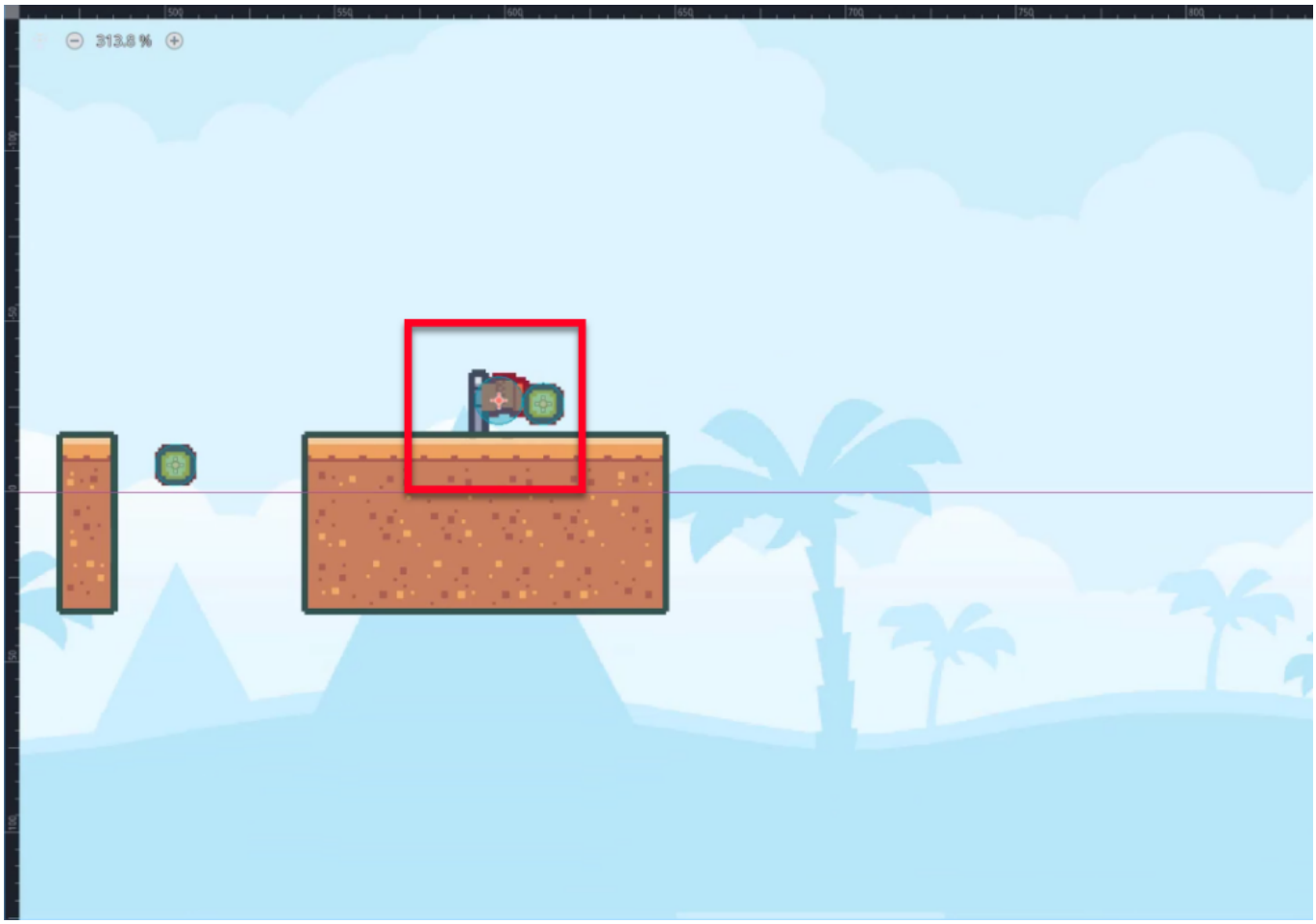


Positioning Game Elements

Now, let’s position the essential game elements such as the end flag, coins, and enemies.

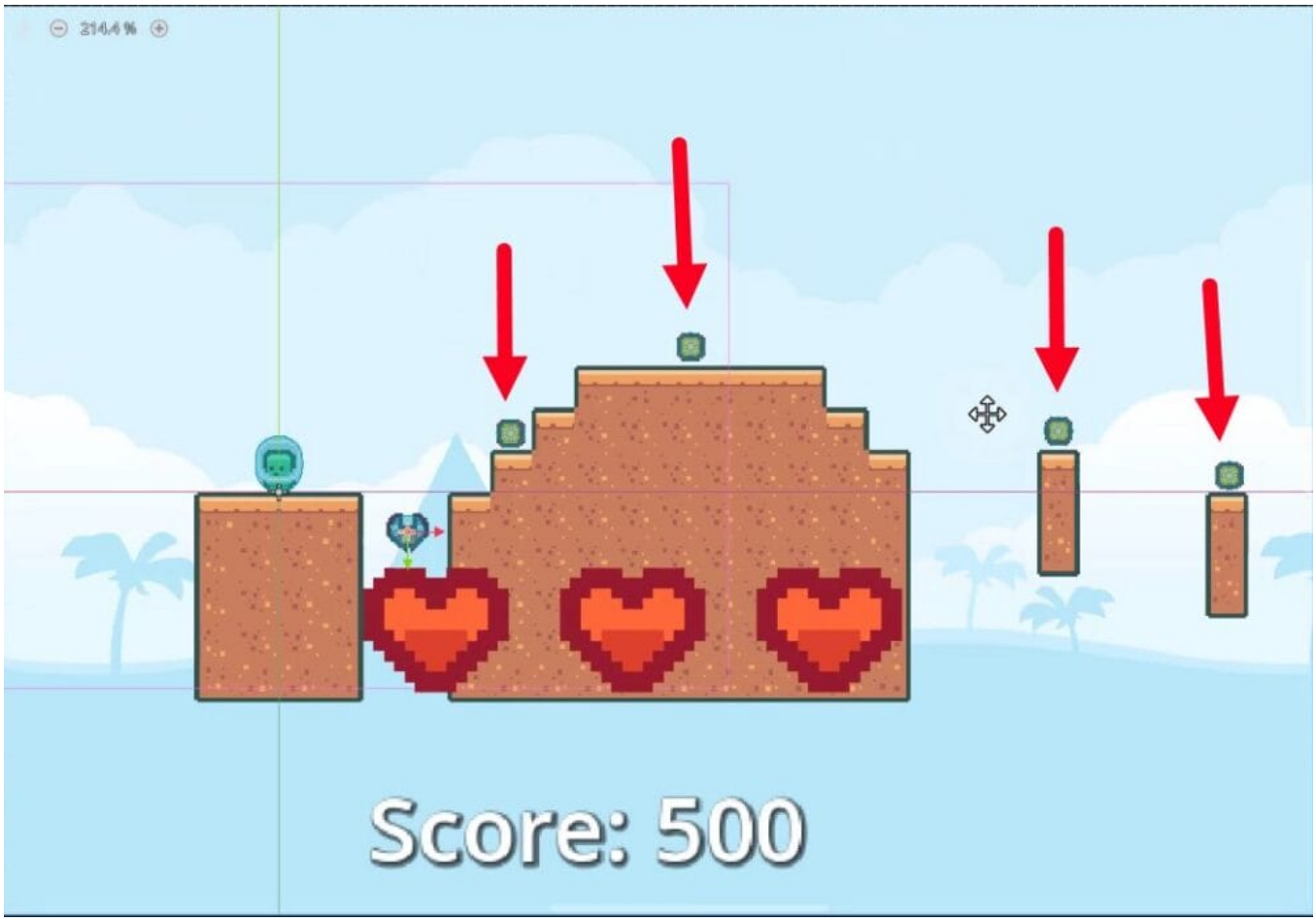
End Flag

Place the end flag at the desired location in the level.



Coins

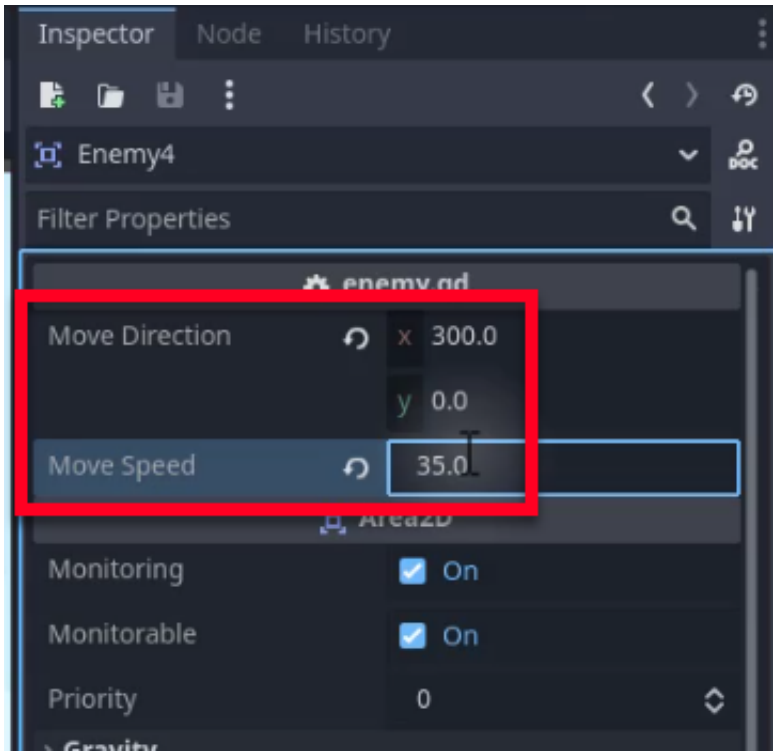
Next, position the coins strategically throughout the level. A simple way to do it is to delete all coins except for one. Then you can duplicate the remaining coin and place everything in various locations.



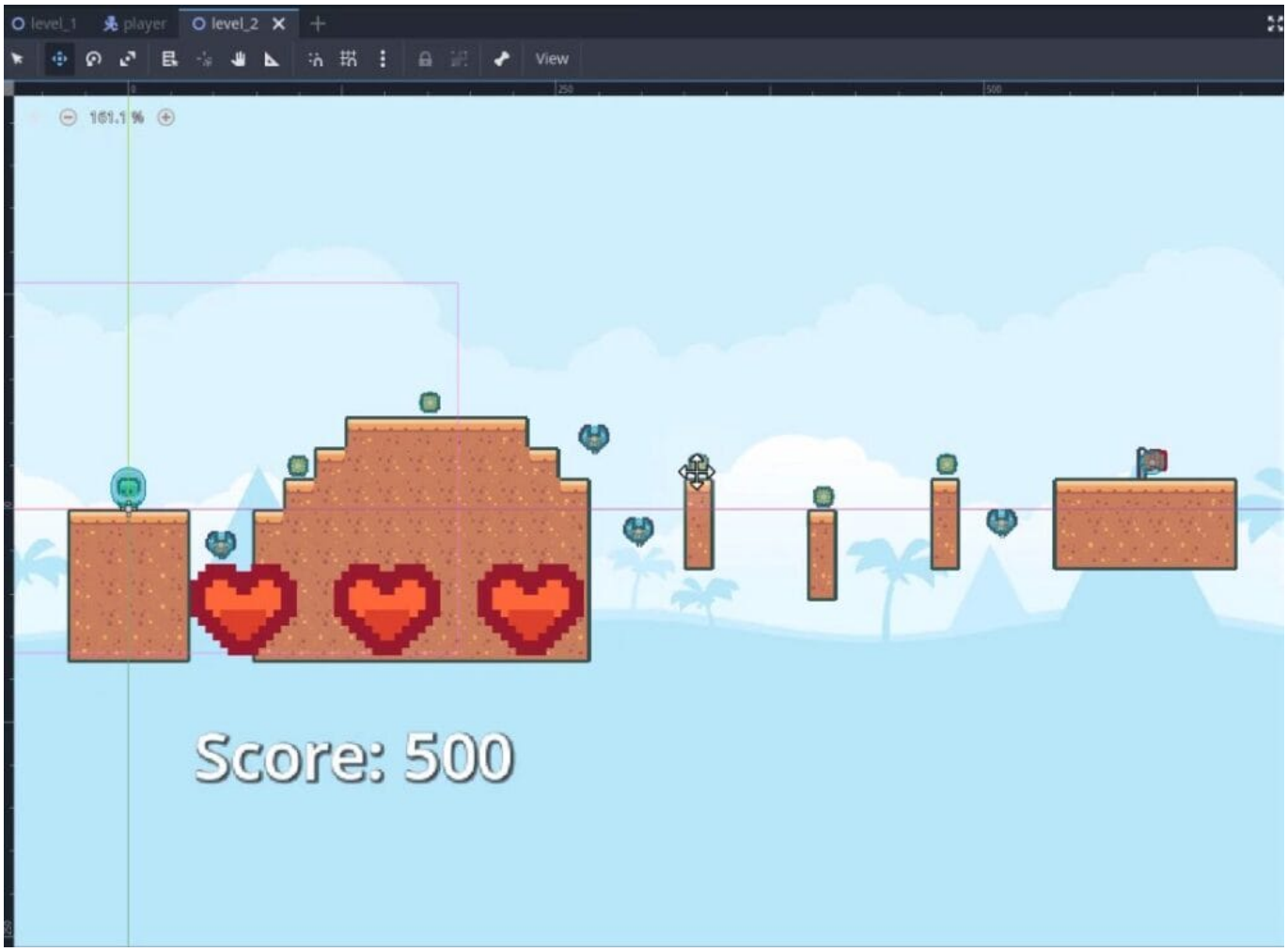
Enemies

Similarly, position the enemies to create challenging gameplay.

You can also modify their movement patterns by adjusting the movement direction and speed in the properties panel. For example, you can make an enemy move horizontally by setting its move direction on the Y-axis to zero and adjusting its position on the X-axis.

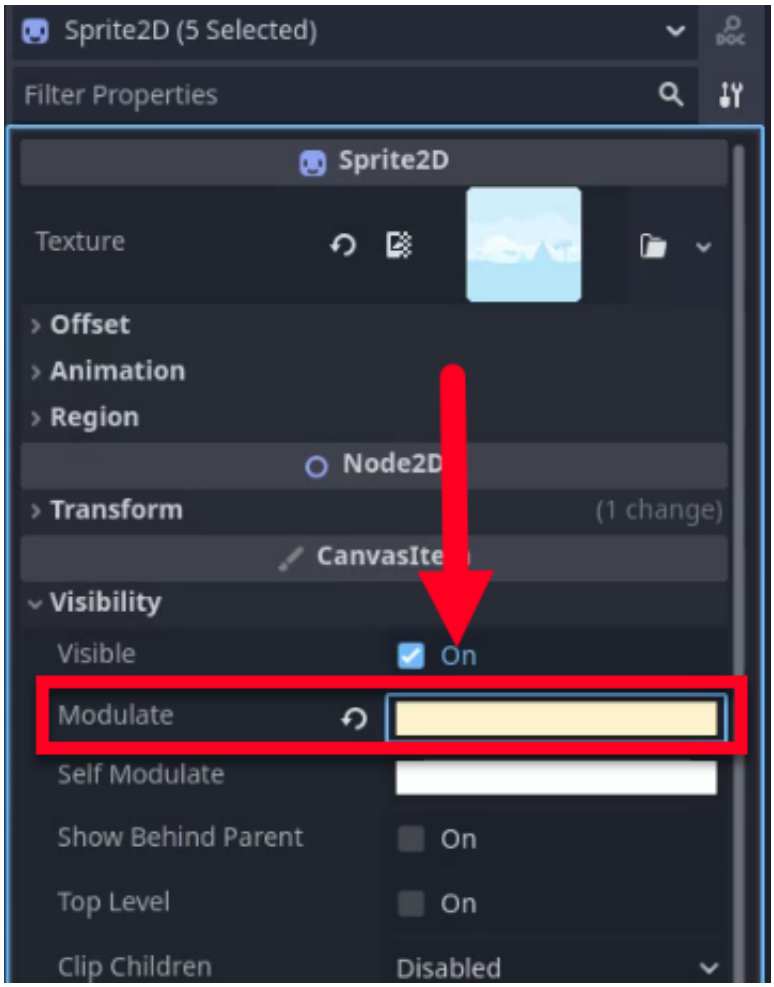


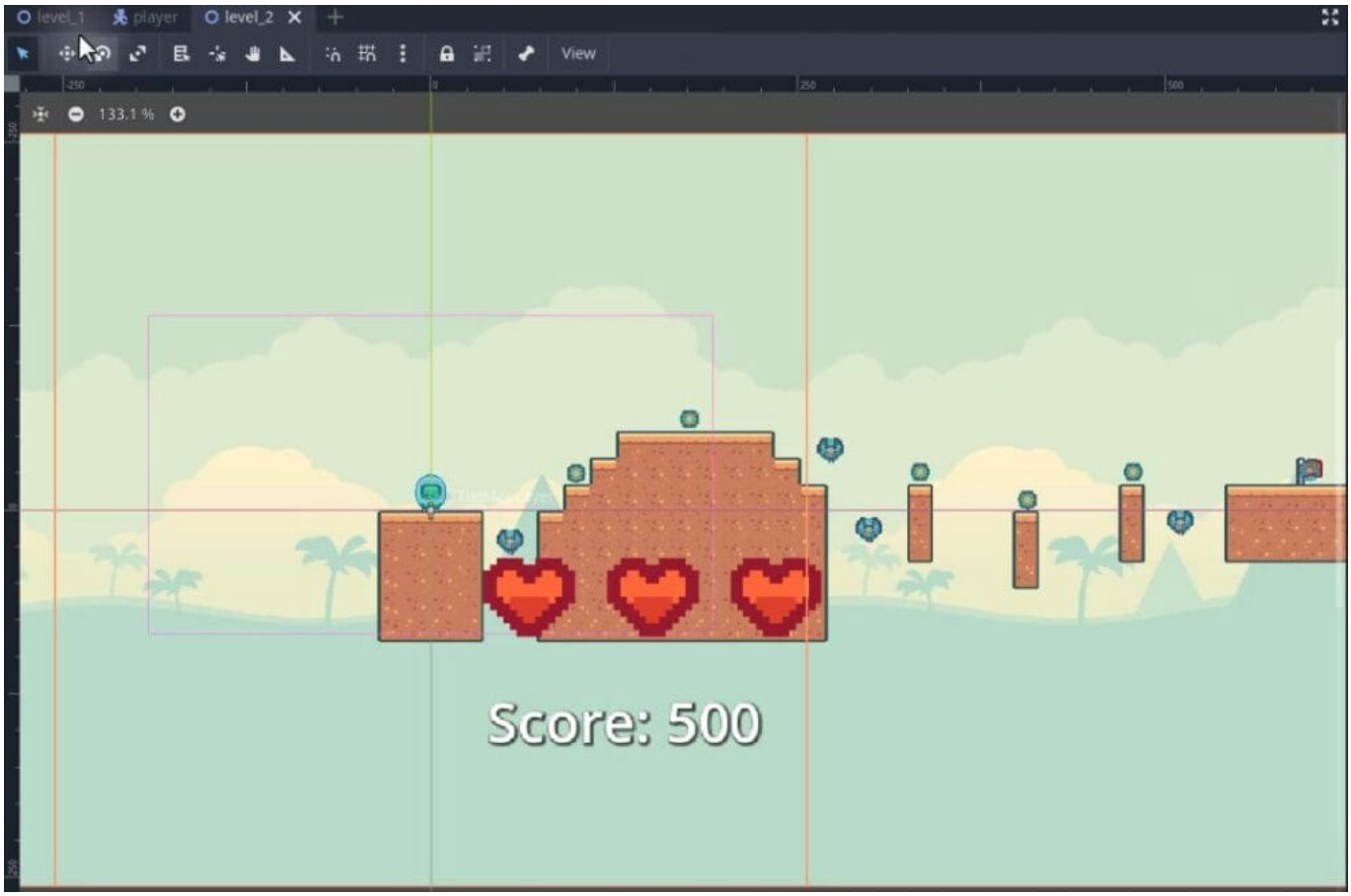
Once again, the ultimate design is up to you!



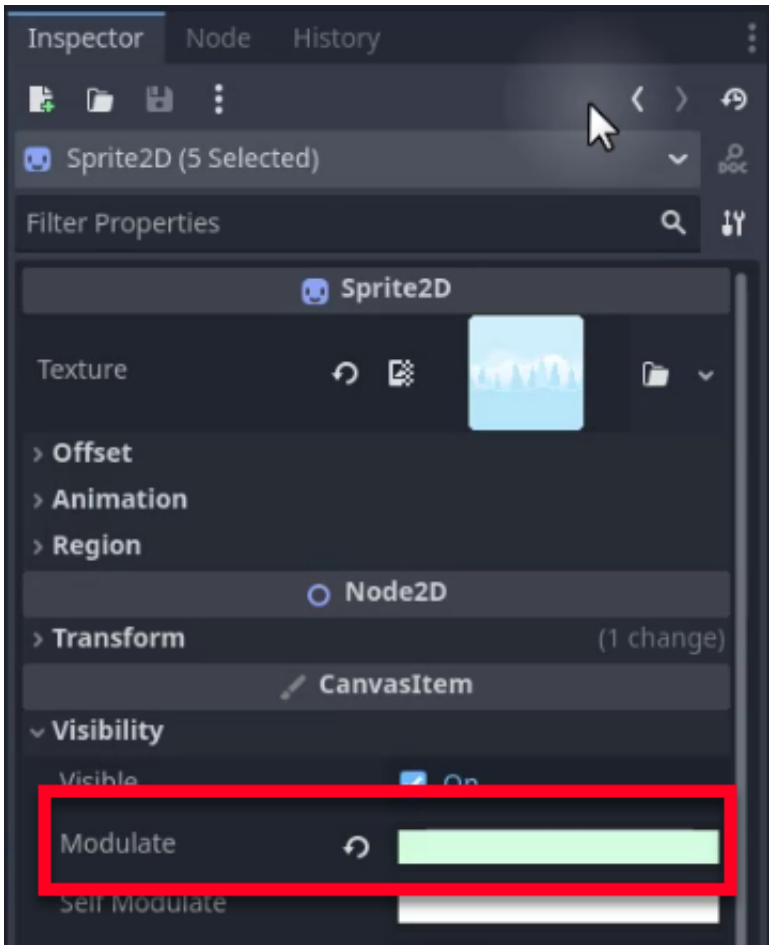
Changing Background Colors

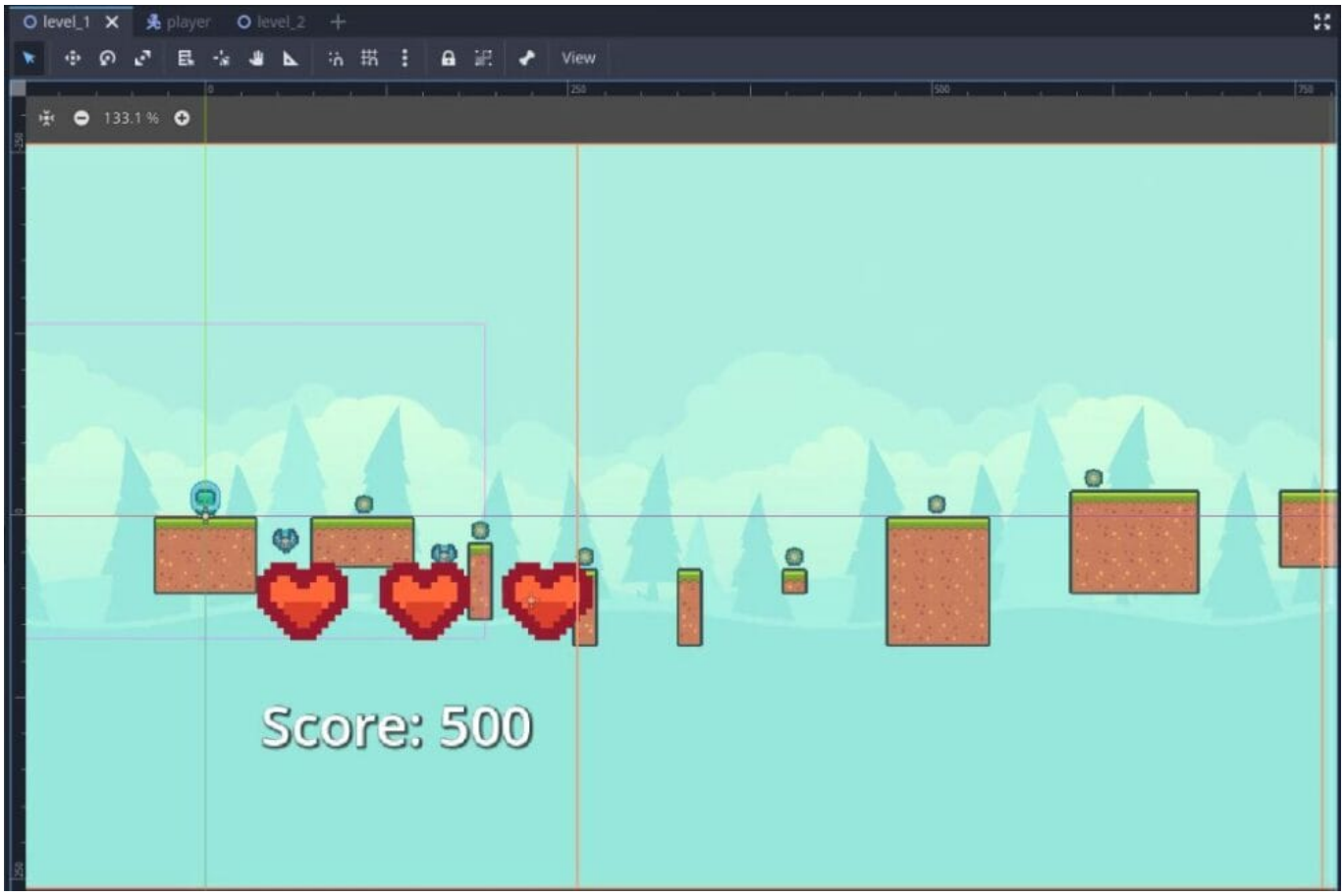
To further differentiate the levels, we will change the background colors. Select the backgrounds for Level 2 and go to the **Visibility** panel. Click on **Modulate** and adjust the colors to give a more desert-like appearance.





For Level 1, you can modulate the backgrounds to have a greener hue, enhancing the forest theme.





Final Touches

Review both levels to ensure they have distinct themes and challenging gameplay elements. This will provide a varied and engaging experience for players.

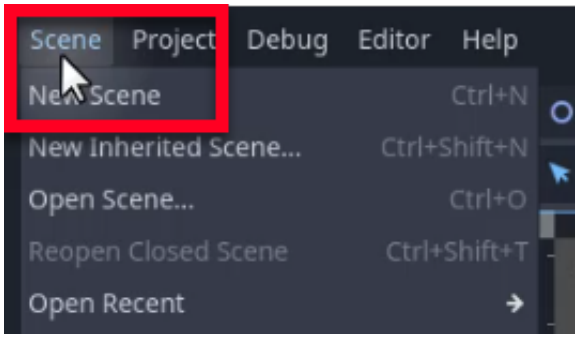
In the next lesson, we'll work on creating a menu scene.

The instructions in the video for this particular lesson have been updated. Please see the lesson notes below for the corrected version.

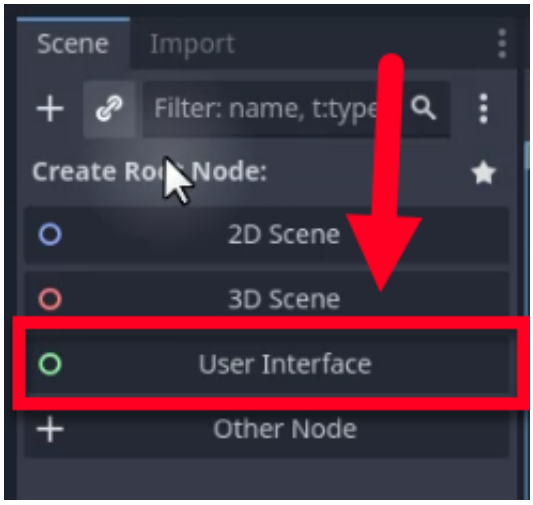
In this lesson, we will be setting up a main menu scene. The menu is the first screen that players will see when they launch the game. It will feature a play button and a quit button. Let's get started!

Creating the Menu Scene

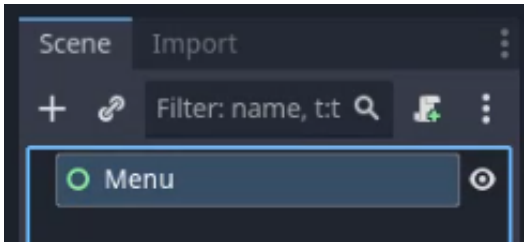
To create the menu scene, click on **Scene** in the top menu, then select **New Scene**.



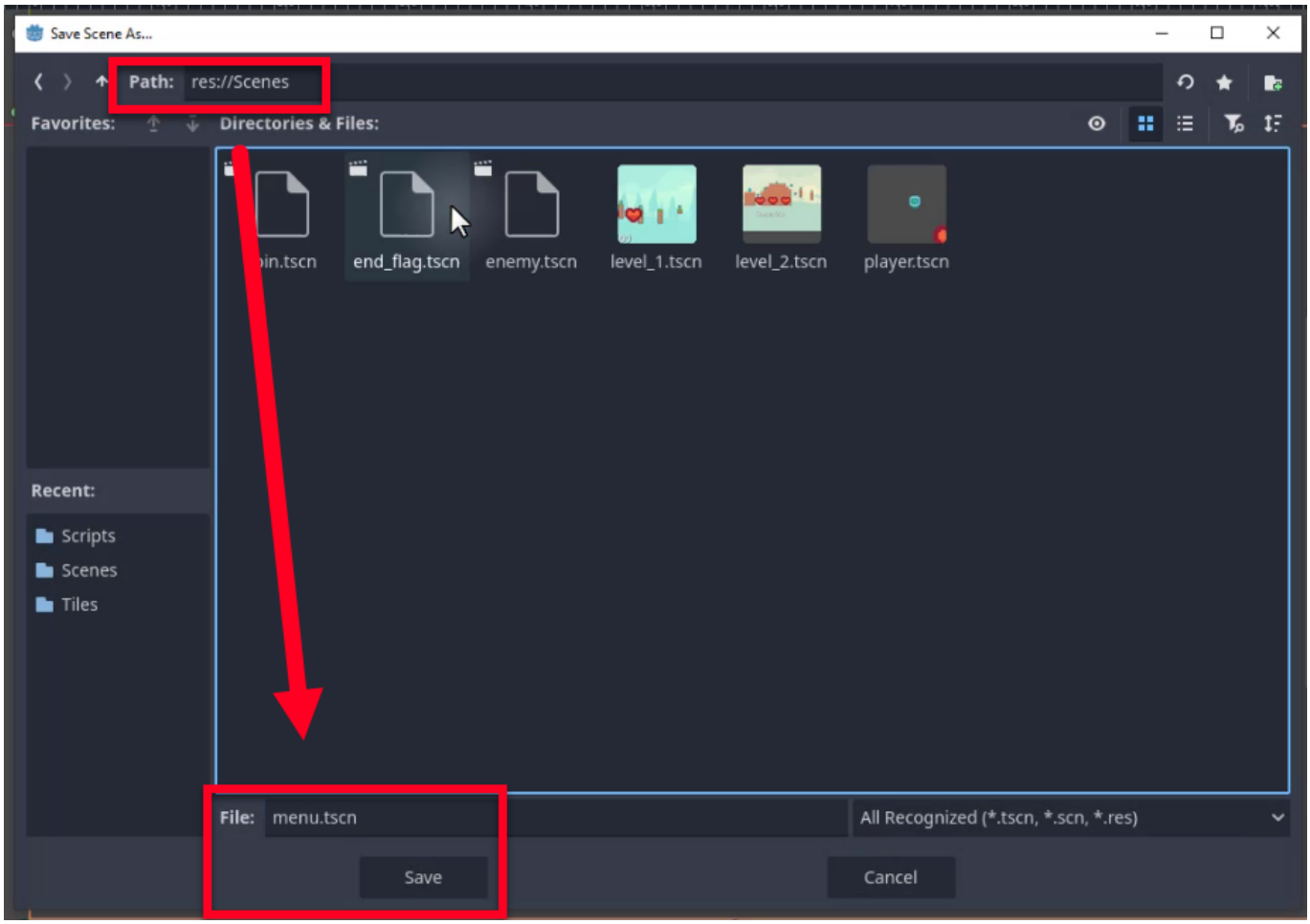
In the new scene dialog, select **User Interface** as the root node.



Name the scene **Menu**.

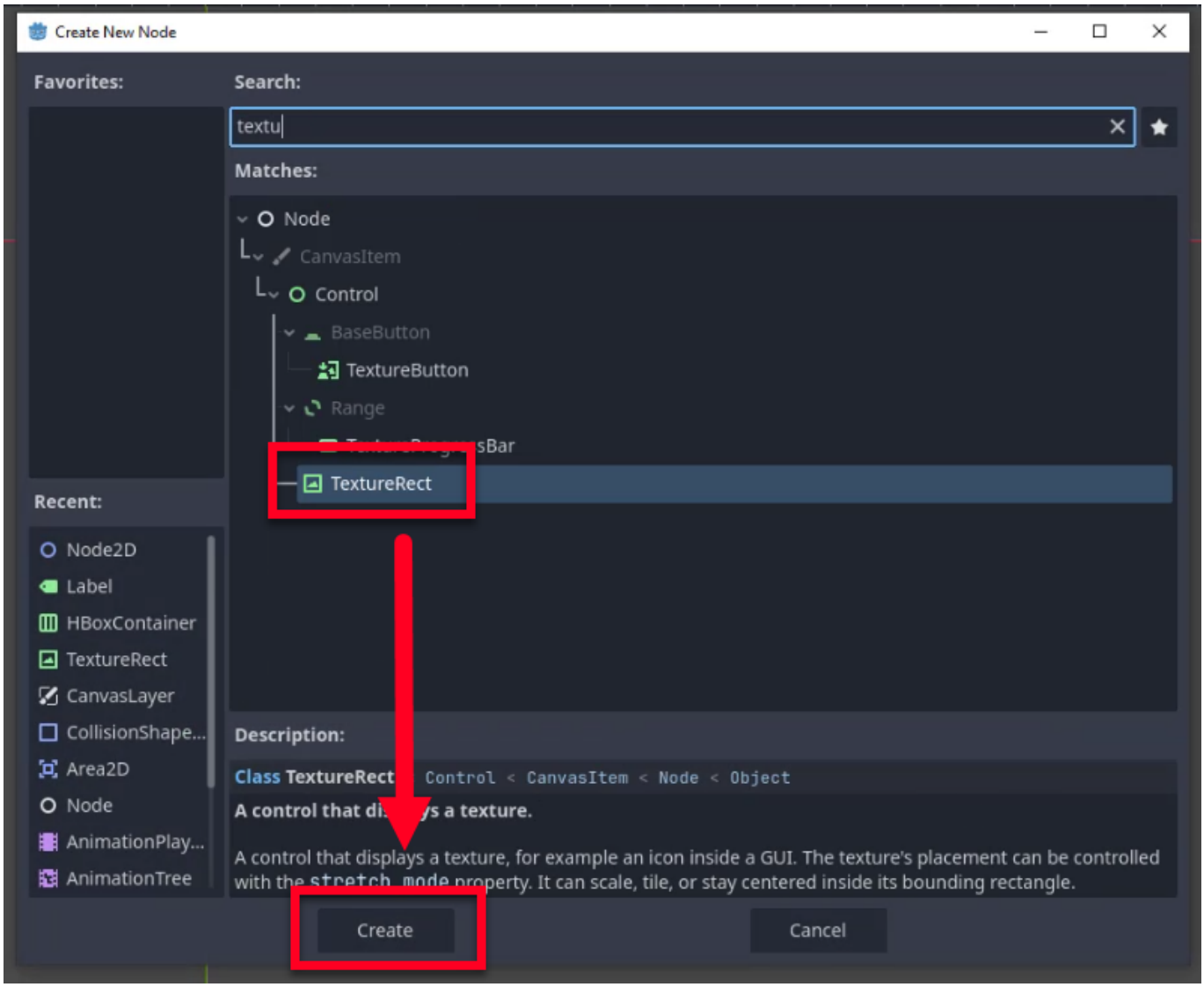


Save the new scene as **menu.tscn** in the **Scenes** folder.

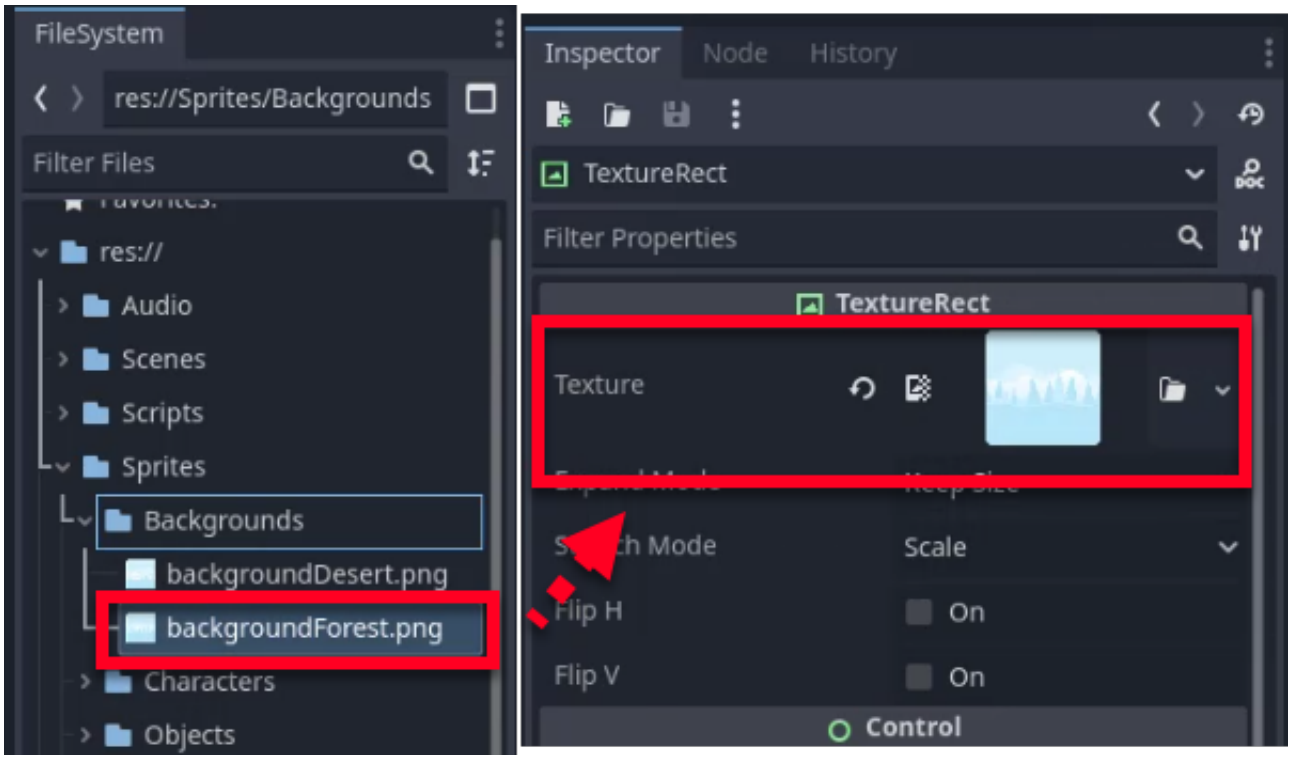


Adding a Background Image

Let’s add a background to our menu scene to make it less plain. Create a new **TextureRect** node as a child of the **Menu Control** node.

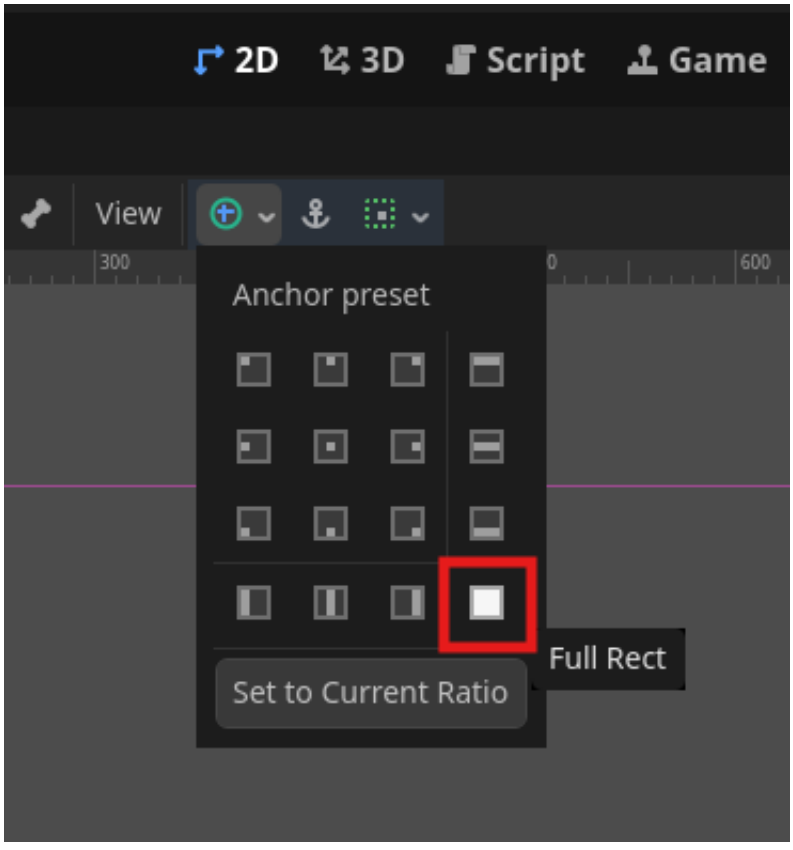


In the FileSystem dock, navigate to the **Sprites/Backgrounds** folder and drag the **forest_background.png** image into the **Texture** property of the **TextureRect** node.

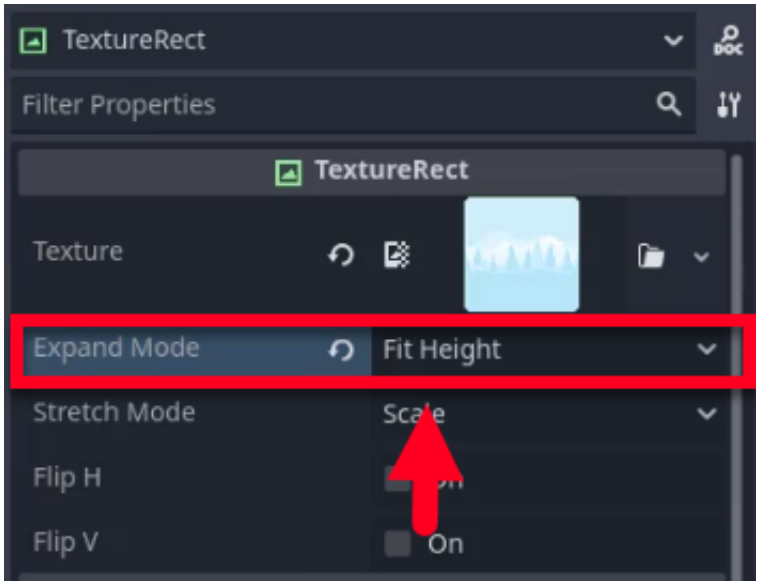


The following instructions have been updated and differ from the video:

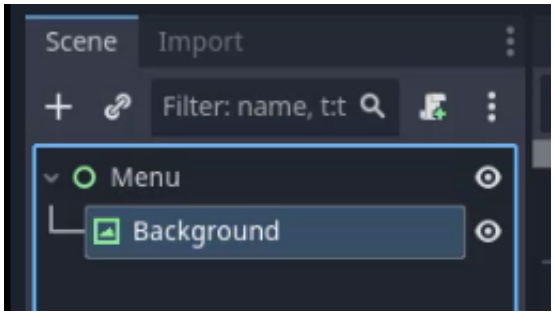
Set the A**nchor preset** using the green circle button on the top bar and clicking on the bottom right most option (Full Rect) to have to **TextureRect** be anchored to the entire game window.



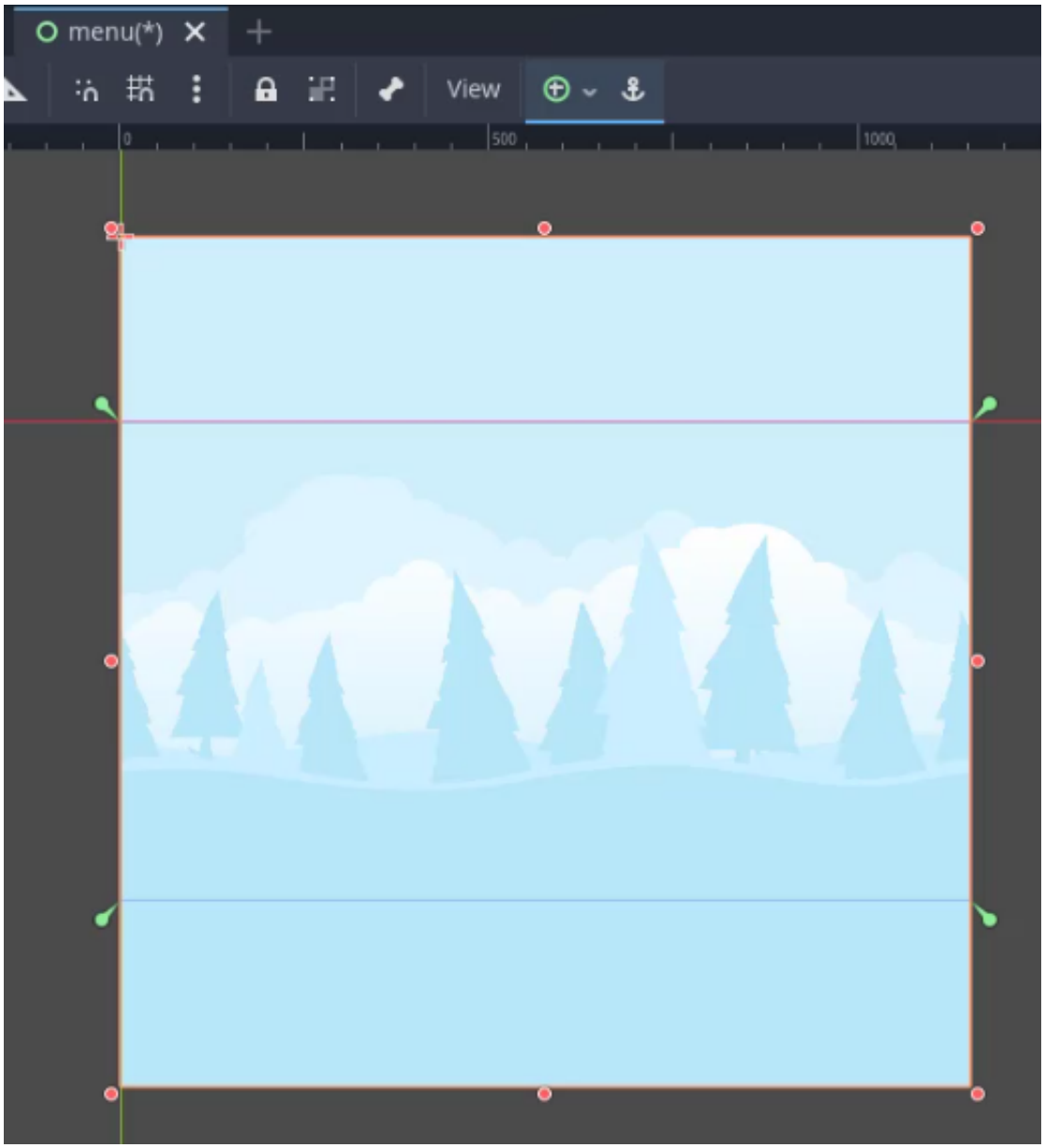
Change the **Expand Mode** property to **Fit to Height**. This will allow the image to resize based on the resolution.



Rename the **TextureRect** node to **Background**.

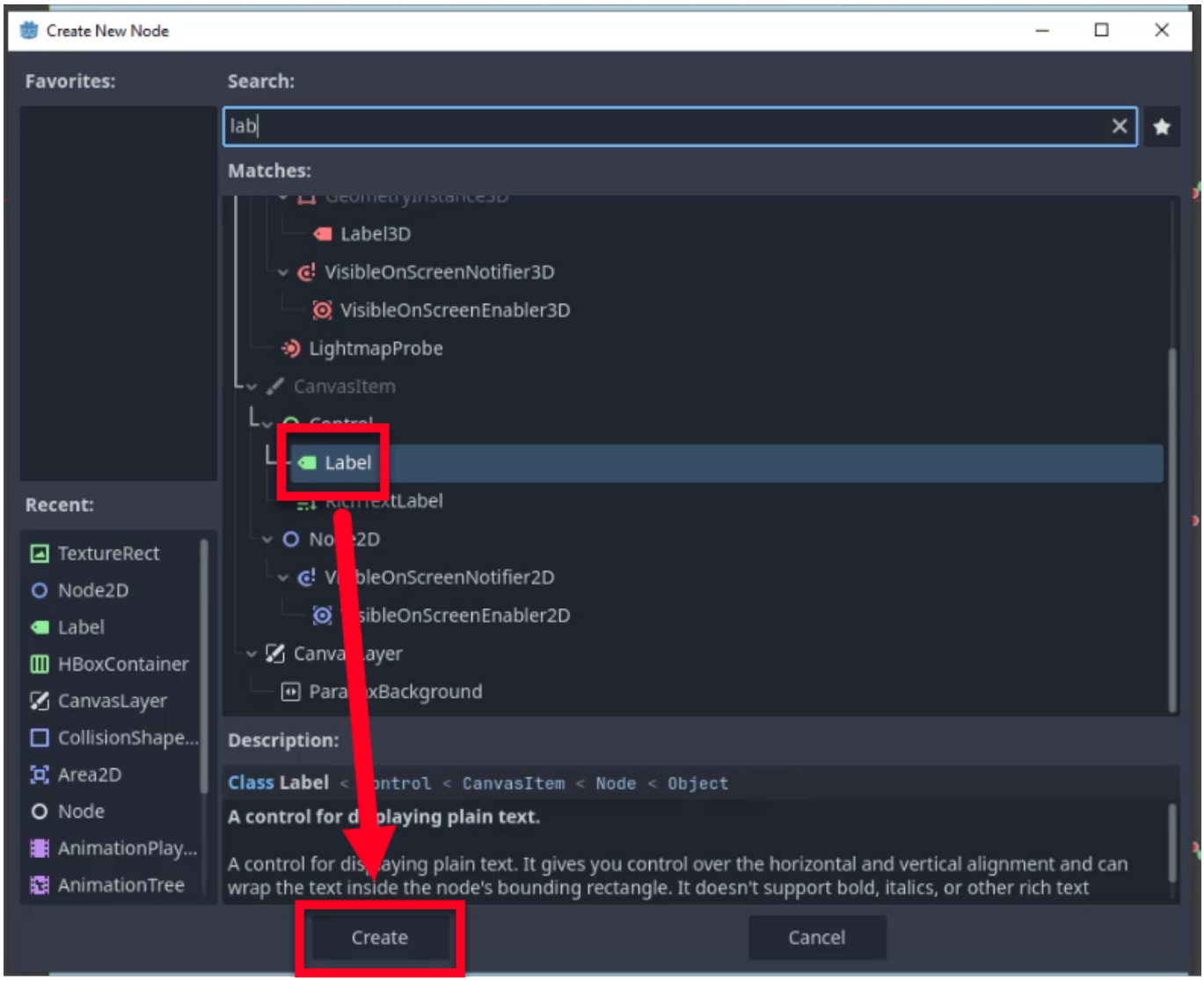


Ensure that the **TextureRect** node is resized to fill the full screen resolution (blue box).

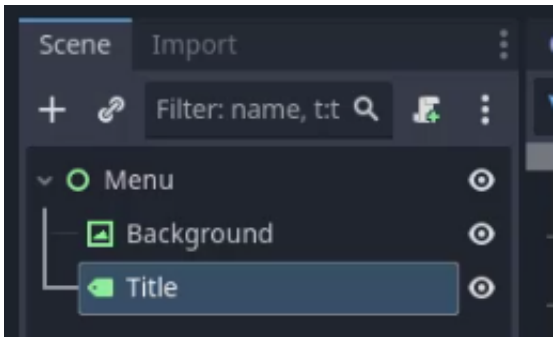


Adding a Title

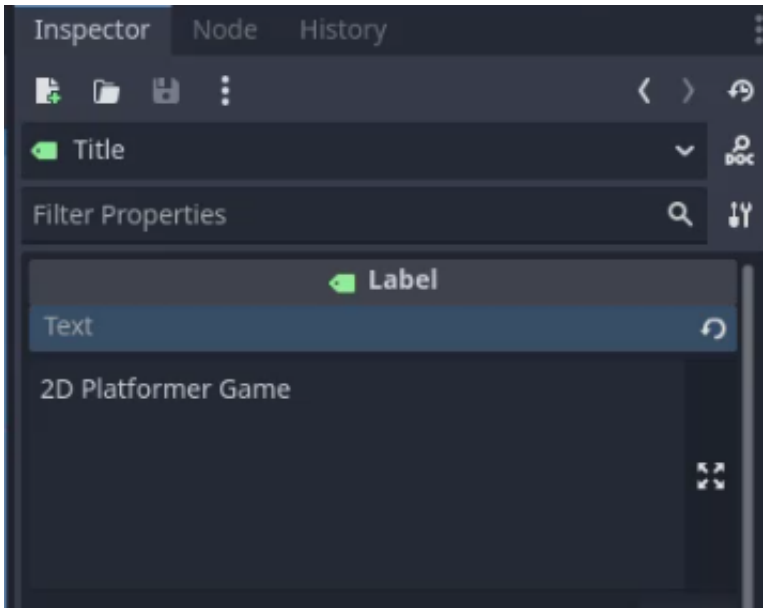
Next, we'll add a title to our menu. Create a new **Label** node as a child of the **Menu Control** node.



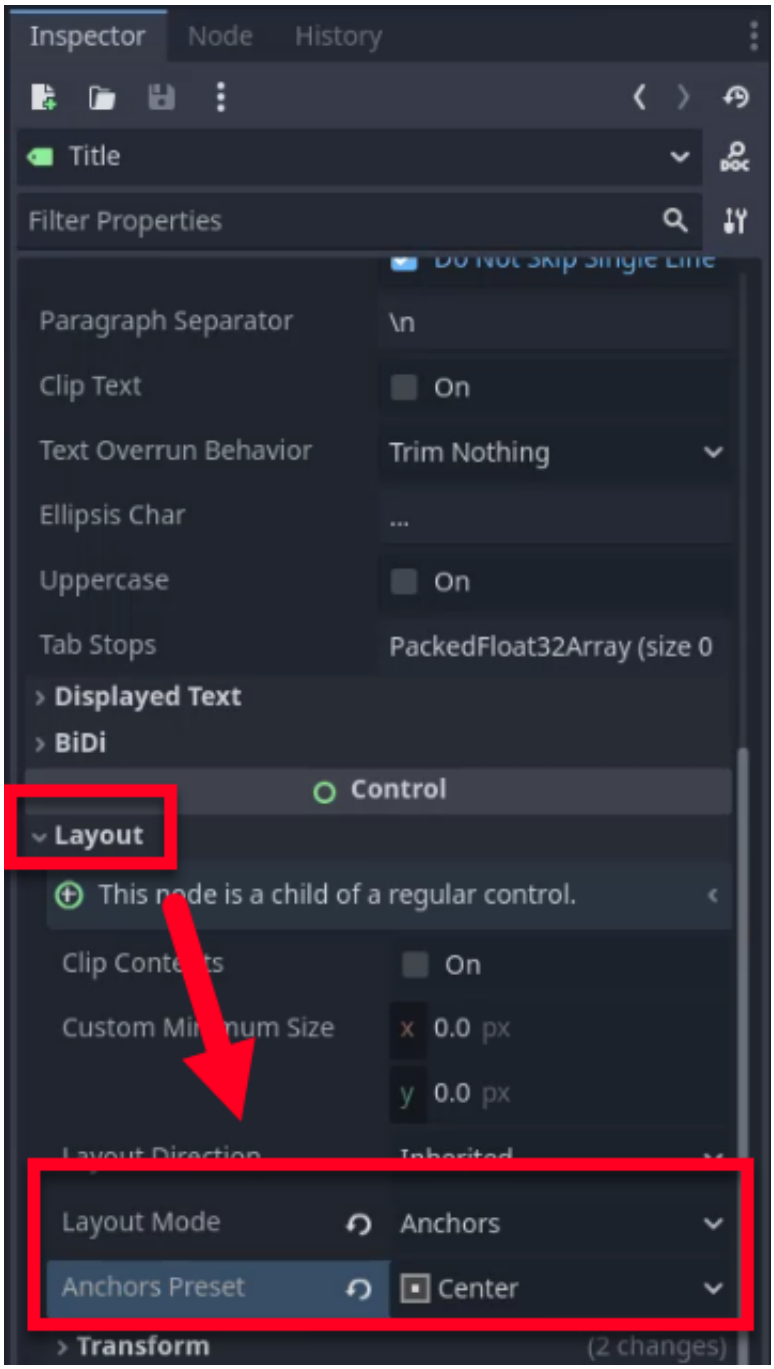
Rename the **Label** node to **Title**.



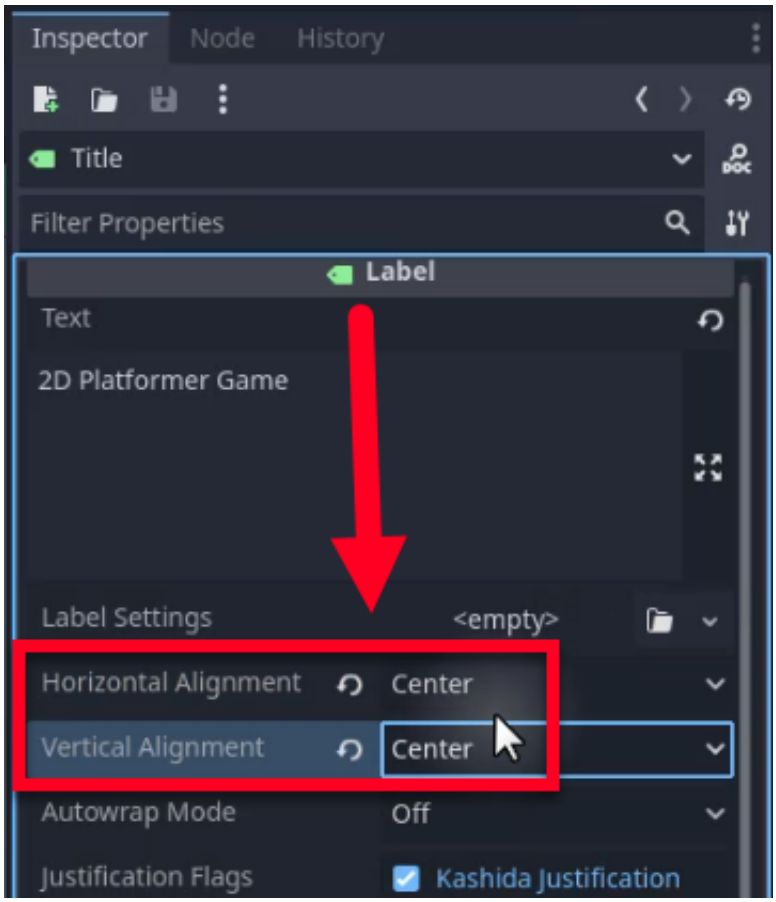
Set the **Text** property to **2D Platformer Game** (or the title of your game).



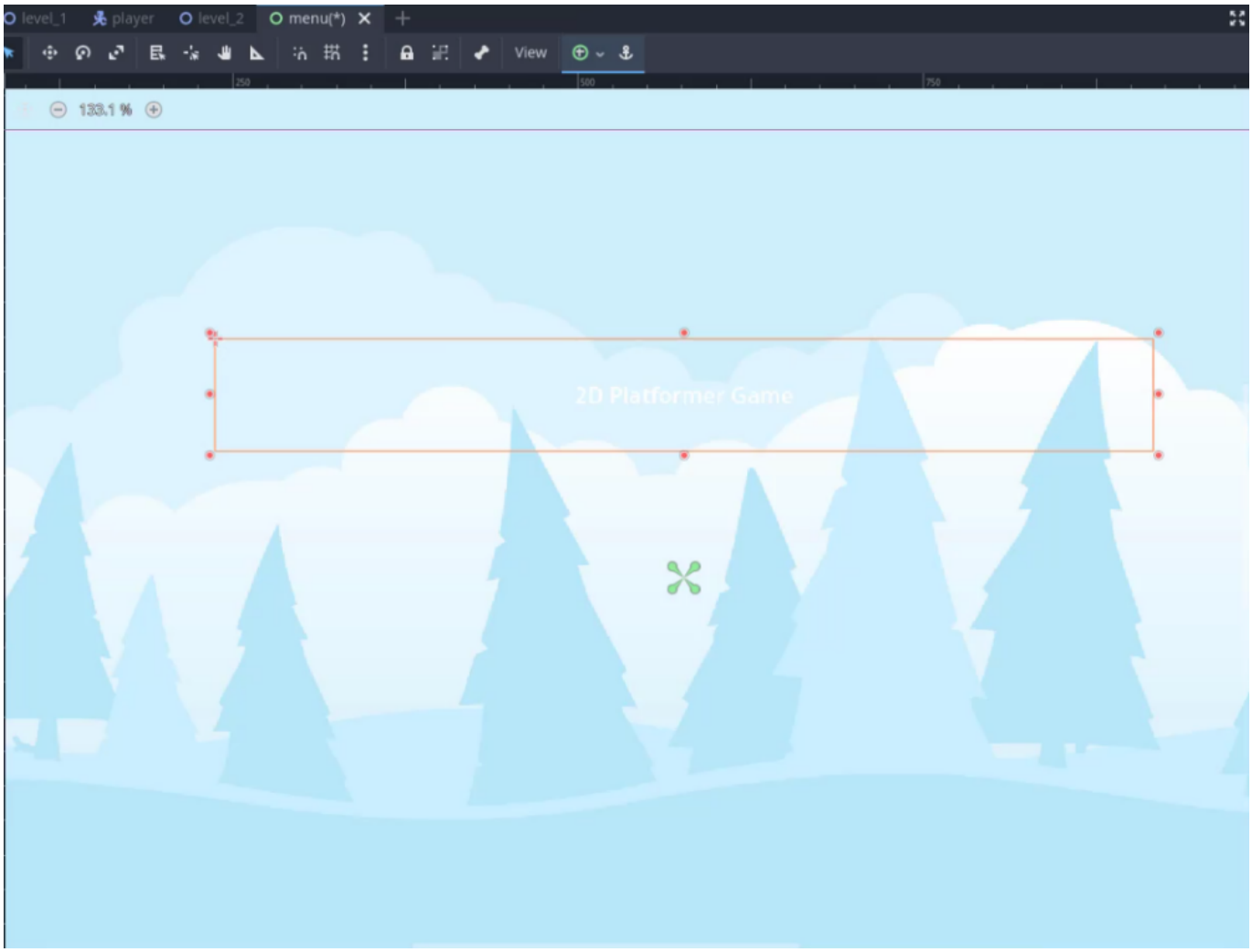
For responsiveness, we want our text anchored to the center of the screen. Scroll down to the **Layout** section. Change the **Layout Mode** to **Anchors** and set the **Anchors Preset** to **Center**.



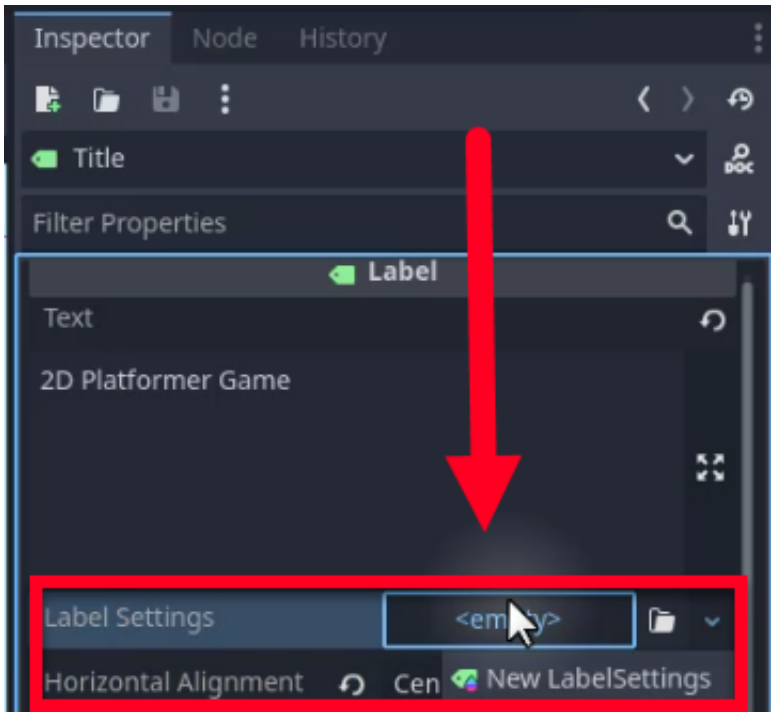
To truly center the text, we also need to change the **Horizontal Alignment** and **Vertical Alignment** properties to **Center**.



Our text is on screen now in the right spot, but barely visible.



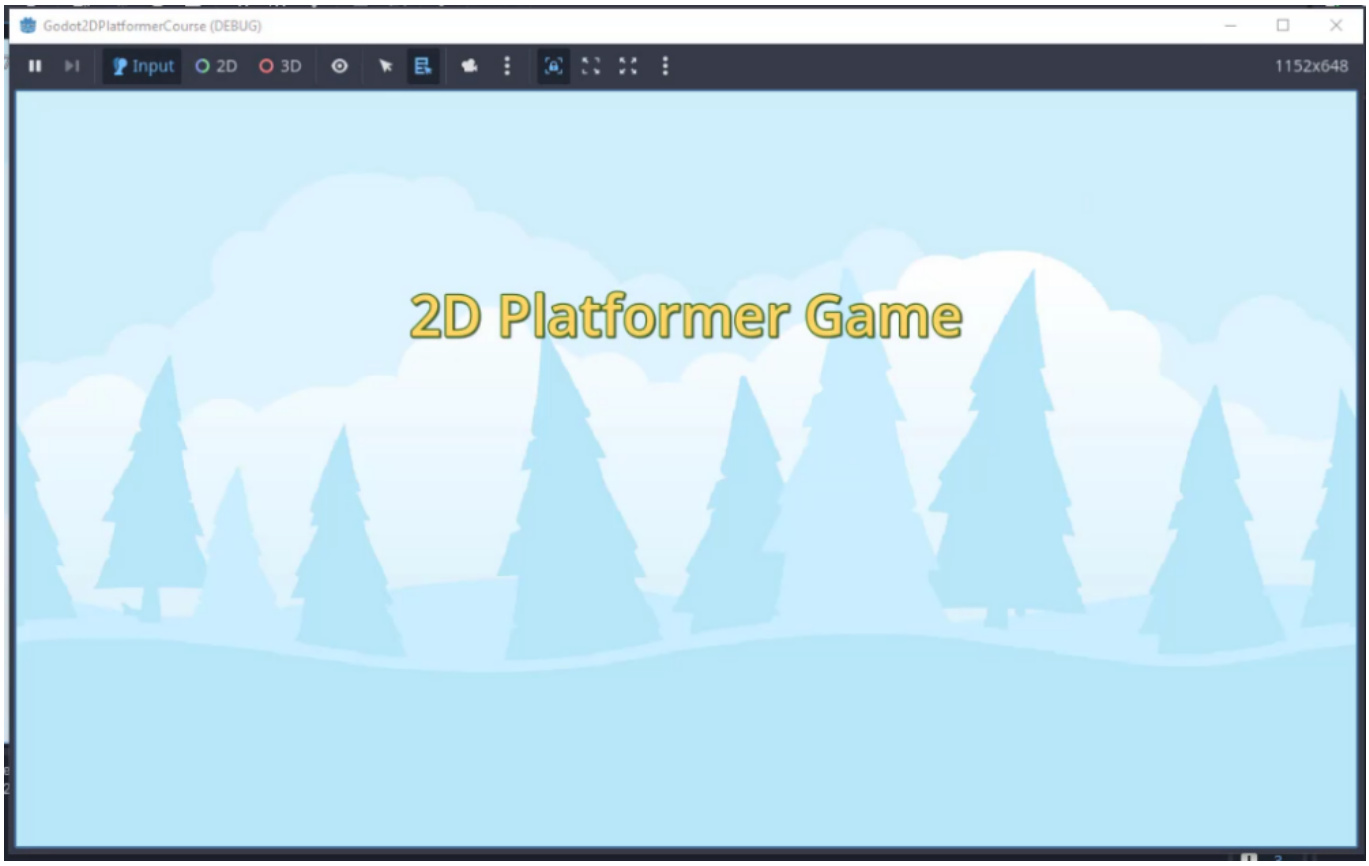
To fix this, first create a new **Label Settings** resource by selecting **New LabelSettings** from the Label Settings dropdown.



Now, click on the LabelSettings to open the customizations available. You can change as you see fit, but we made the following alterations:

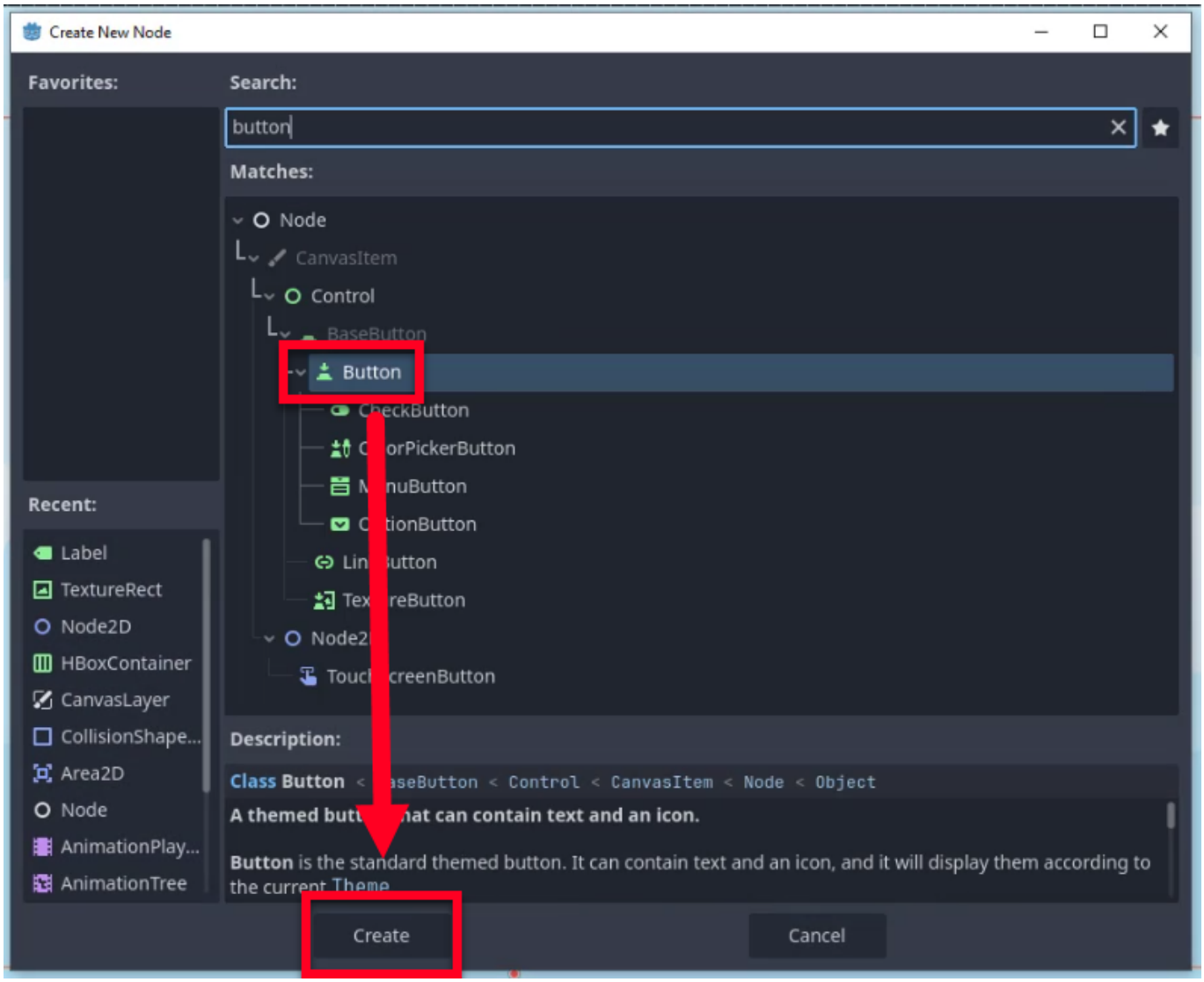
- Set the Font size to 48px
- Set the font color to a light yellow
- Set the outline to 5px
- Set the outline color to a forest green
- Set the shadow size to 5px

Now our text is much more visible on the menu.

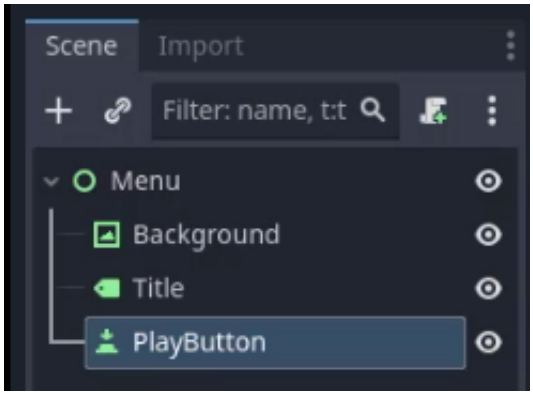


Adding Buttons

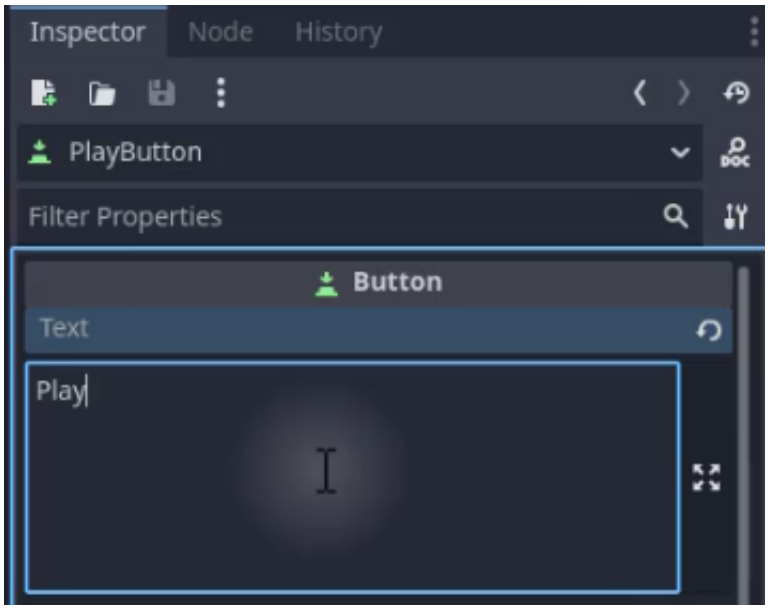
Next, we need to add buttons to our menu so players can actually get to the game. Create a new **Button** node as a child of the **Menu Control** node.



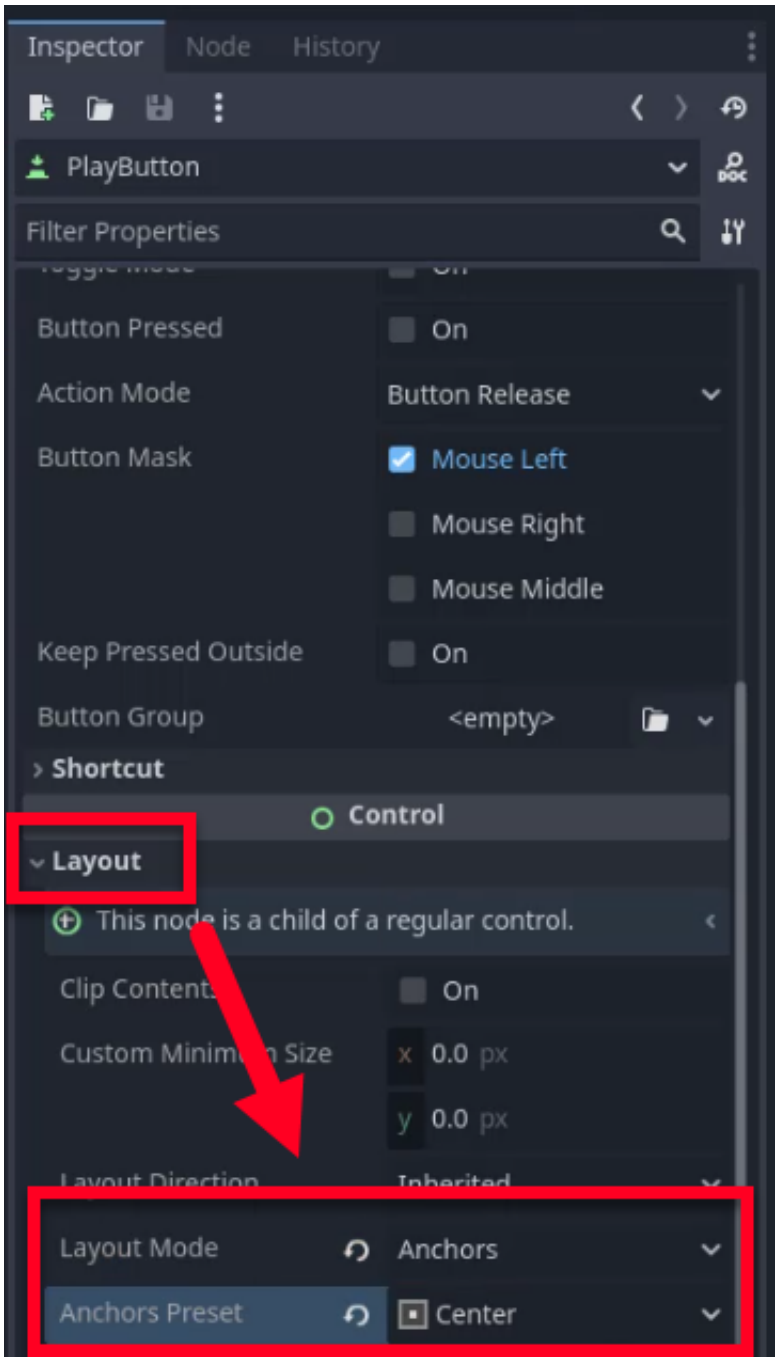
Rename the **Button** node to **PlayButton**.



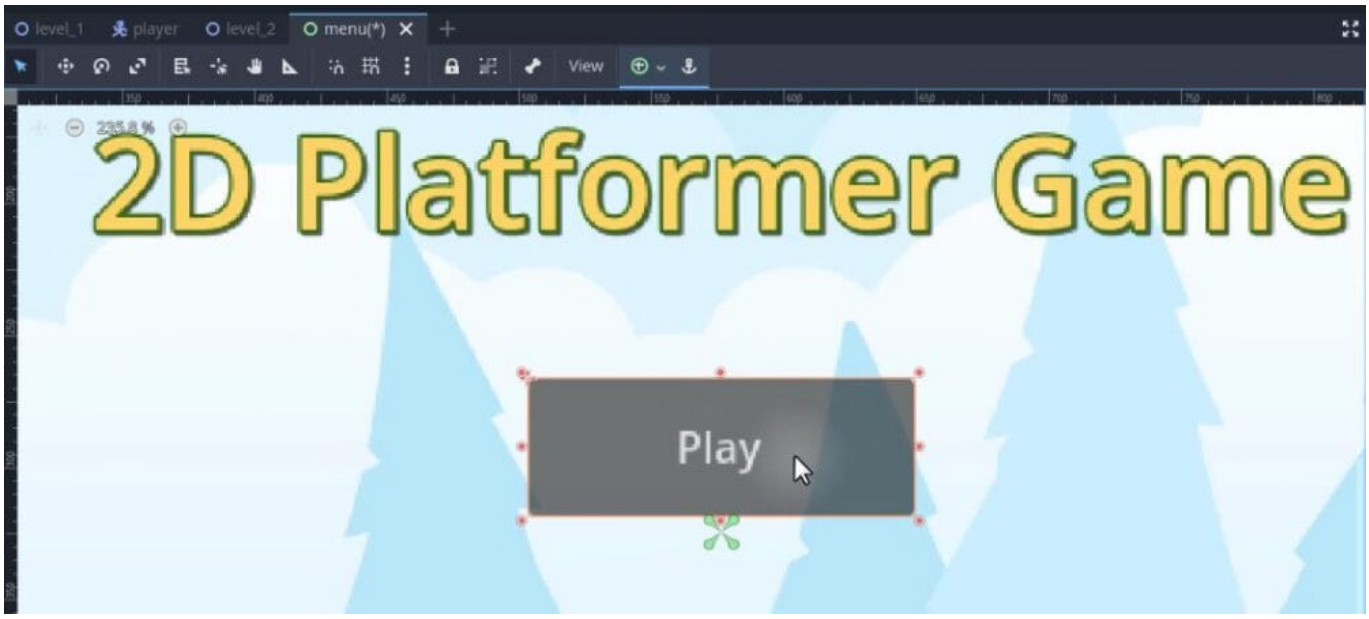
Set the **Text** property to **Play**.



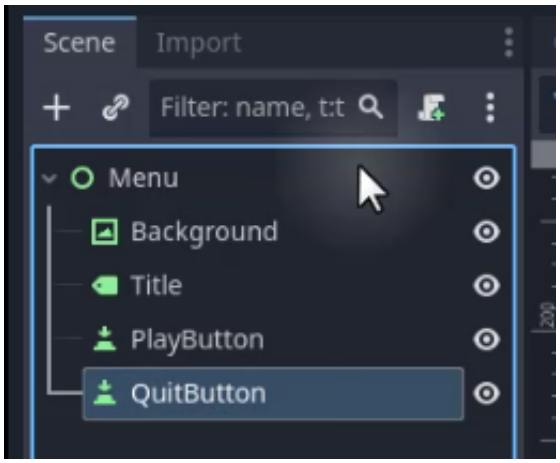
As we also want this anchored to the center, repeat what we did with the title: change the **Layout Mode** to **Anchors** and the **Anchors Preset** to **Center**.



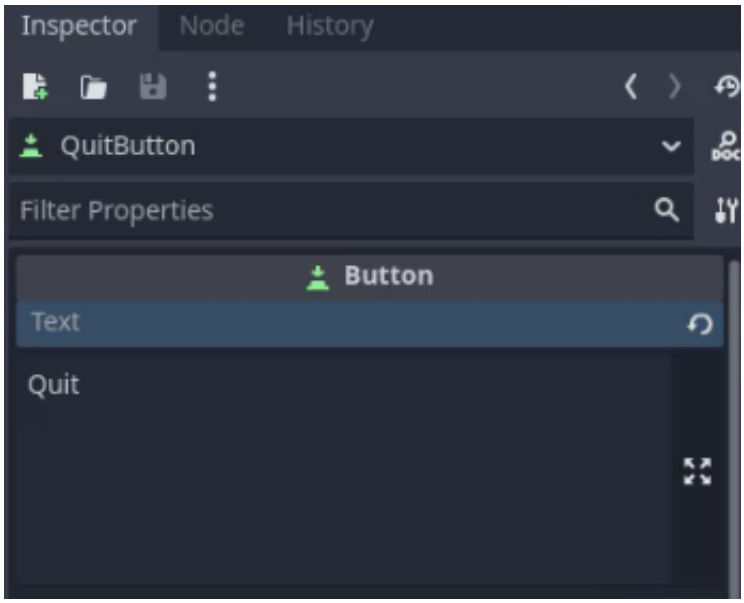
Position the title and button until you have a look you like. Also, feel free to resize the button as desired as well.



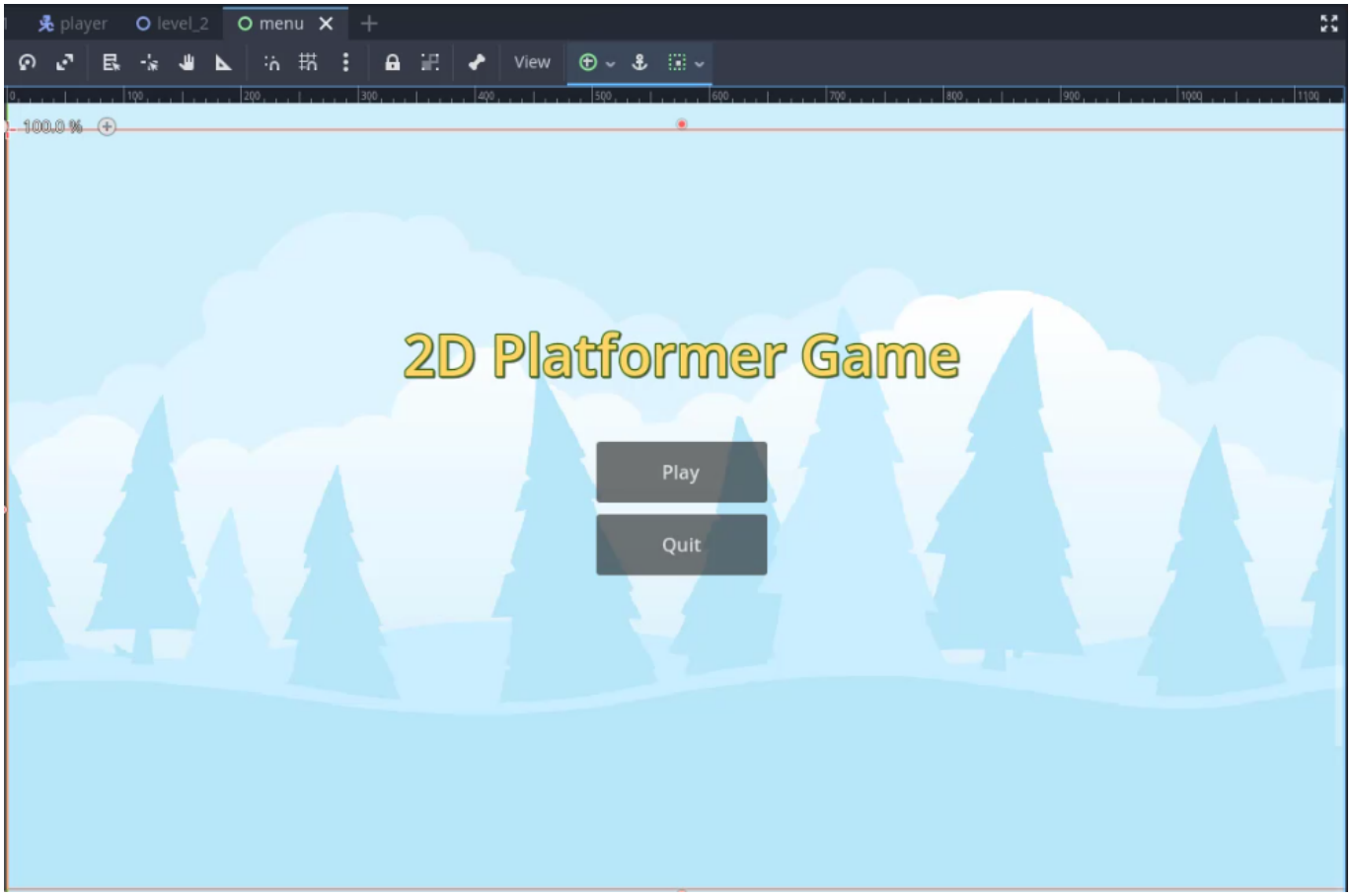
Duplicate the **PlayButton** node and rename the duplicate to **QuitButton**.



Set the **Text** property of the **QuitButton** to **Quit**.

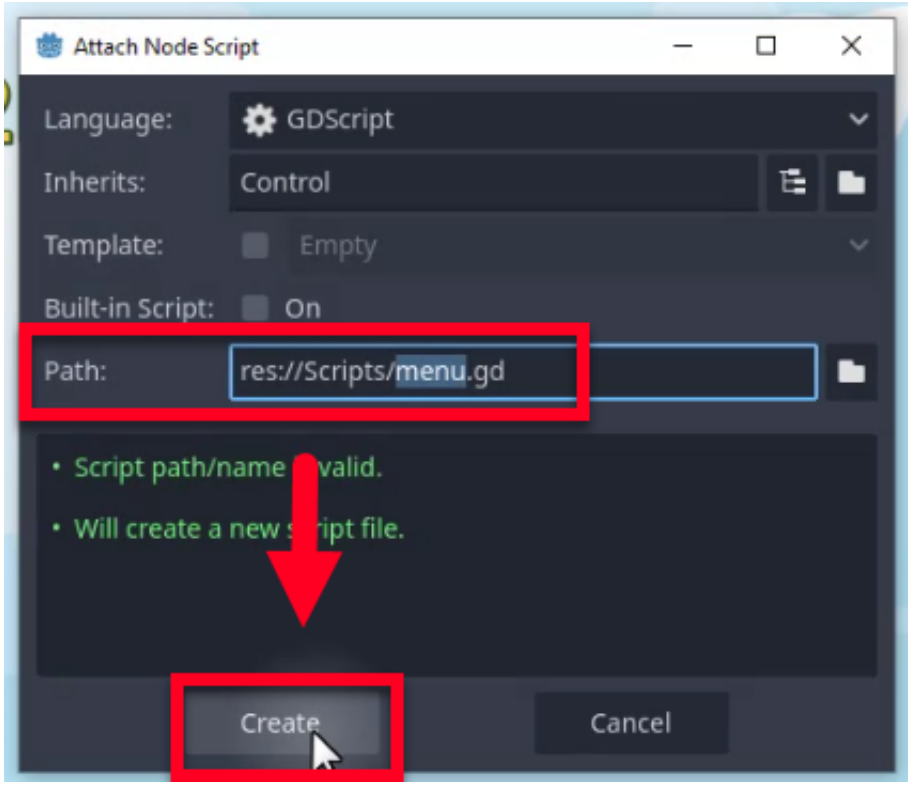


Position the **QuitButton** below the **PlayButton**. Once again, go with what you think looks best!

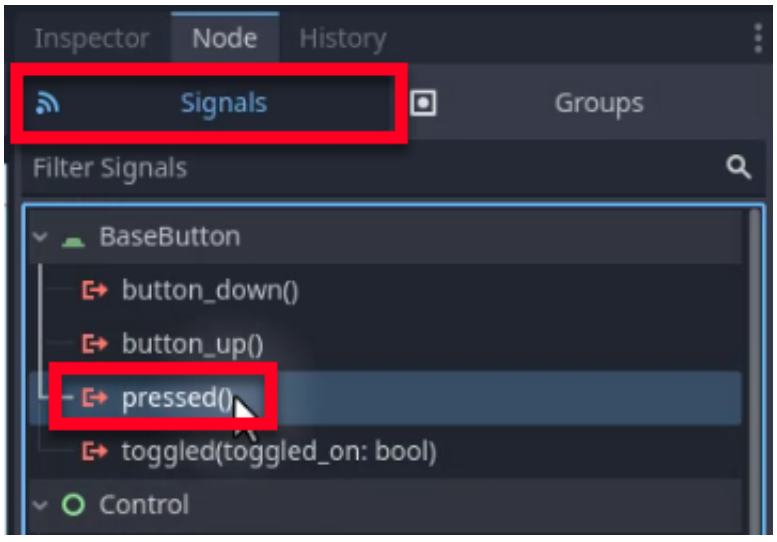


Scripting the Menu

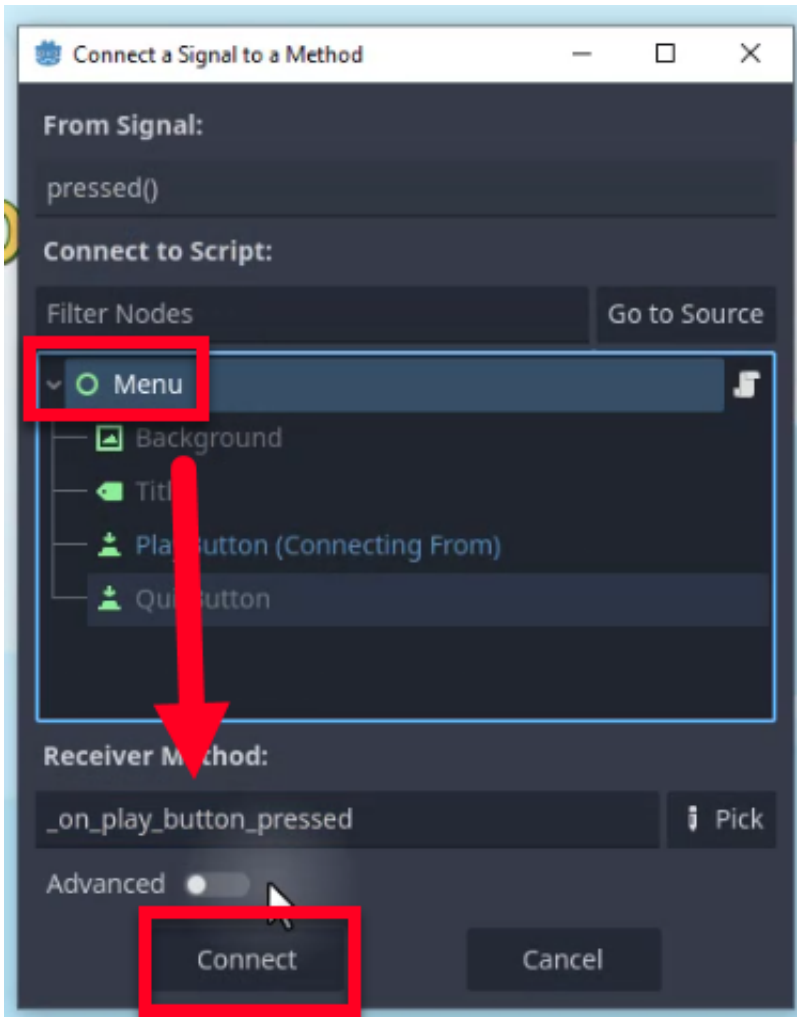
Now, let's make our menu functional with some logic. Create a new script on the **Control** node and save it as **menu.gd** in the **Scripts** folder.



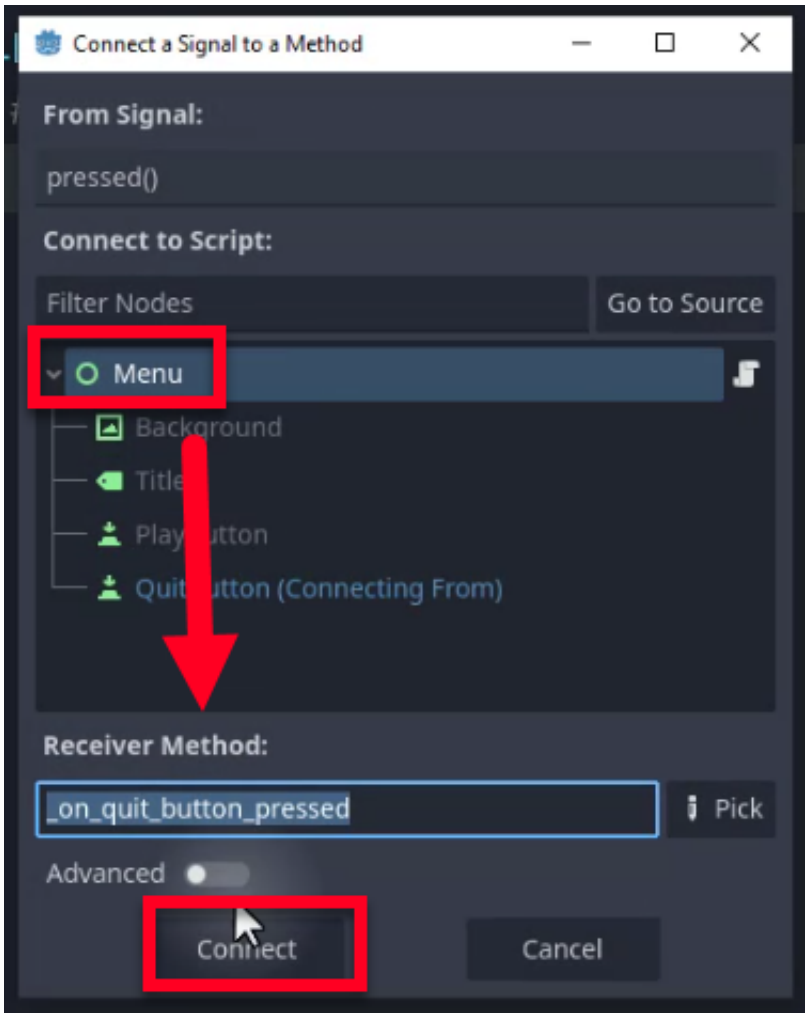
Connect the **pressed** signal of the **PlayButton** to the script. You can do this by selecting the PlayButton, selecting Node > Signals (in the same pane as the Inspector), and double-clicking **pressed()**.



On the prompt, select the Menu node and hit connect.



Next, repeat the steps above and connect the **pressed** signal of the **QuitButton** to the script



With our signals connected, we can now add our code logic. The logic here will be very simple, as there are only a limited number of things the buttons need to do.

```
extends Control

func _on_play_button_pressed():
 PlayerStats.score = 0
 get_tree().change_scene_to_file("res://Scenes/level_1.tscn")

func _on_quit_button_pressed():
 get_tree().quit()
```

With the play button, we reset the score to 0 and then change to our level 1 scene. For the quit button, we quit the game.

Testing the Menu

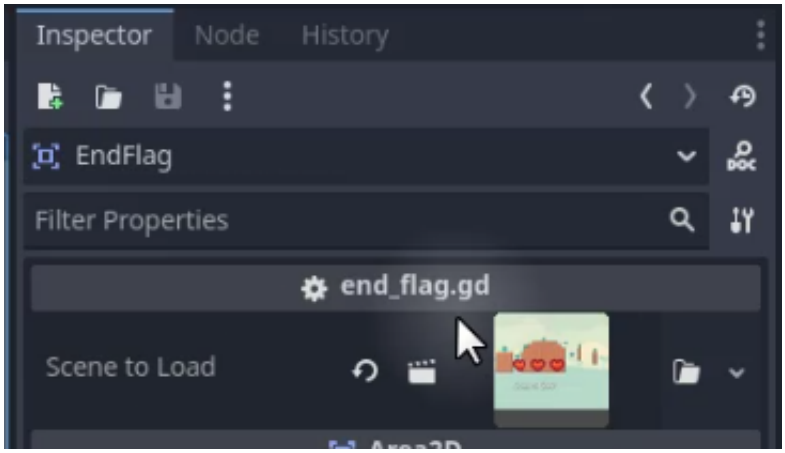
To test the menu, follow these steps:

1. Save the scene and click the play button.
2. Test the **Quit** button to ensure it closes the game.
3. Test the **Play** button to ensure it loads the first level.

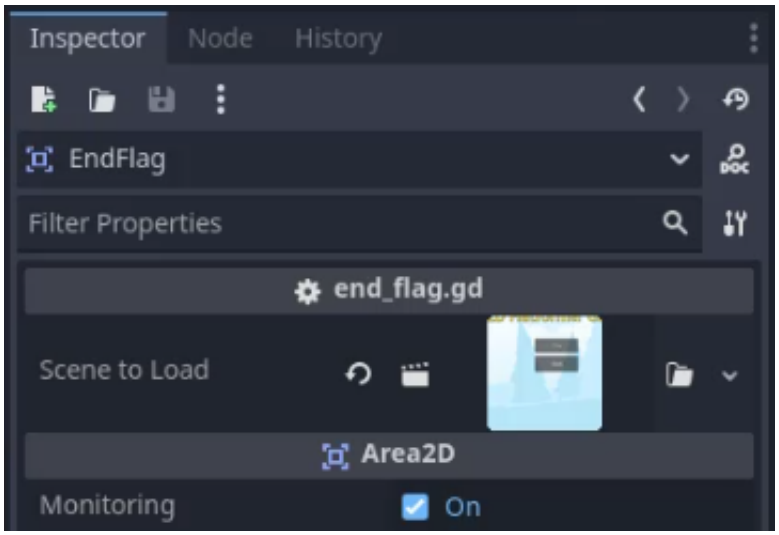
Connecting End Flags

To ensure the game can be played through from start to finish, connect the end flags in each level to the appropriate scenes:

- In **Level1**, connect the end flag to **Level2**.



- In **Level2**, connect the end flag to **Level3** (if created) or back to the **Menu**.



Updating Game Over Logic

To ensure the game returns to the menu when the player falls off the edge, update the game over logic in the player script (**player.gd**). In this case, we want to change the scene that is loaded in the **game_over** function to the menu scene:

```
func game_over():
 get_tree().change_scene_to_file("res://Scenes/menu.tscn")
```

That’s it for this lesson! You now have a functional menu scene with play and quit buttons!

Godot 2D Platformer - Background & Menu [2025]

1. How does the background's movement relate to the player's movement in a parallax system?

a. The background moves at the same speed as the player
b. The background moves in the opposite direction of the player
c. The background does not move at all
d. The background moves at a slower rate than the player to simulate depth

2. Why do we connect the end flags in each level?

a. To play a special sound effect
b. To allow the player to move between levels seamlessly
c. To increase the player's score
d. To activate an enemy spawn event

3. True or False: We can use buttons, such as in a menu, to transition to other scenes in our game.

a. True
b. False

4. True or False: In Godot, no special node is required to play sound effects.

a. True
b. False

5. Why do we preload sound effects in the player script?

a. To reduce memory usage
b. To ensure the sounds loop continuously
c. To prevent delays or glitches when playing the sounds
d. To decrease the file size of the game

6. True or False: To play sounds more efficiently, we can use a function to assign a sound file to the AudioStreamPlayer and then play it.

a. True
b. False

Final Challenge: Altering the End Flag

We've learned a lot over this course – from fundamental 2D platformer mechanics to providing game polish that makes our 2D platformer stand out from the crowd. However, before we close out this course, we have one final challenge for you.

At current, when your player collides with the end flag, it will produce the following error in Godot:

```
end_flag.gd:9 @ _on_body_entered(): Removing a CollisionObject node during a physics callback is not allowed and will cause undesired behavior. Remove with call_deferred() instead.
```

Your challenge is to fix this error so that the end flag is bug-free! **Hint:** Check the Player Script as we had to do something similar to prevent errors with running our game_over function.

In the next lesson, we'll go over the solution!

Final Challenge Solution

We hope you've had a go at that challenge! Let's talk about the solution.

In order to fix the bug with our **end flag**, we need to refer back to how we handled the take_damage function in our Player script. Recall that when calling the game_over function, we had to wrap this call in a special Godot function called call_deferred.

```
func take_damage (amount : int):
  ...

  if health <= 0:
    call_deferred("game_over")
```

call_deferred instructs Godot to call whatever function we need at **the end of the frame**. In so doing, this prevents errors caused by Godot trying to calculate physics on objects that may have already been moved out of the current scene.

To fix our **end_flag.gd** script, we first want to make a new function called _load_new_scene and move our scene loading code into that function:

```
func _load_new_scene ():
  get_tree().change_scene_to_packed(scene_to_load)
```

Then, in our _on_body_entered function, we call this new function using call_deferred:

```
func _on_body_entered(body):
  if not body.is_in_group("Player"):
    return

  call_deferred("_load_new_scene")
```

And that's it! You should now be able to run the game without error. Congratulations for getting this far!

Congratulations on completing your 2D platformer game setup in Godot! Throughout this course, we've covered a range of topics to help you create a dynamic and engaging game. Let's recap the key steps and concepts you've learned along the way.

Setting Up the Player

We began by establishing our player character and implementing their movement mechanics:

- Added smooth movement with acceleration and braking.
- Incorporated jumping mechanics.
- Set up the player's input map, allowing multiple keys to be assigned to a single action for flexibility.

Creating Enemies

Next, we focused on setting up enemies to challenge the player:

- Designed enemies as scenes that move between points.
- Implemented damage mechanics for when the player hits an enemy, including:
    - Damage flash effect.
    - Sound effects.
    - Screen shake.

Collectible Coins

To add more interactivity, we set up collectible coins with engaging visual effects:

- Created a bobbing up and down effect.
- Added a rotating effect by modifying the horizontal scale.

Level Transition with End Flags

To enable progression between levels, we set up end flags:

- Allowed players to transfer between levels upon reaching the flag.

About Zenva

Zenva is an online learning academy with over a million students. We offer a wide range of courses suitable for:

- Beginners starting their journey in game development.
- Those looking to learn new skills or enhance existing ones.

You can choose to follow along with the included lesson summaries or use the provided course files to work alongside the instructor.

Conclusion

Thank you for following along with the course. We wish you the best of luck with your future Godot games!

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.

background_parallax.gd

Found in folder: **/Scripts**

This code makes the background node move in relation to the player's position. The background's position is calculated by multiplying the player's position by a parallax value, creating a scrolling effect. The parallax value determines how fast the node moves in relation to the player.

```
extends Node2D

var parallax : float = 0.7
@onready var player = $"../Player"

func _process (_delta):
 global_position = player.global_position * parallax
```

camera_shake.gd

Found in folder: **/Scripts**

This code extends the Camera2D class and adds a shake effect when the player's health is updated. The shake effect's intensity is initially set to 3 when the health is updated, and then gradually decreases over time. The camera's offset is randomly changed based on the current intensity, creating a shaking motion.

```
extends Camera2D

var intensity : float = 0

func _ready ():
 get_parent().OnUpdateHealth.connect(_damage_shake)

func _damage_shake (_health : int):
 intensity = 3

func _process (delta):
 if intensity > 0:
  intensity = lerpf(intensity, 0, delta * 10)
  offset = _get_random_offset()

func _get_random_offset () -> Vector2:
 var x = randf_range(-intensity, intensity)
 var y = randf_range(-intensity, intensity)
 return Vector2(x, y)
```


coin.gd

Found in folder: **/Scripts**

This code controls the movement and behavior of a coin game object. It makes the coin's sprite rotate and bob up and down over time, creating a dynamic visual effect. When the coin collides with the player, it increases the player's score and removes itself from the game.

```
extends Area2D

var rotate_speed : float = 3.0
var bob_height : float = 5.0
var bob_speed : float = 5.0

@onready var start_pos : Vector2 = global_position

@onready var sprite : Sprite2D = $Sprite

func _physics_process(_delta):
 var time = Time.get_unix_time_from_system()

 # rotate
 sprite.scale.x = sin(time * rotate_speed)

 # bob up and down
 var y_pos = ((1 + sin(time * bob_speed)) / 2) * bob_height
 global_position.y = start_pos.y - y_pos

func _on_body_entered(body):
 if not body.is_in_group("Player"):
  return

 body.increase_score(1)
 queue_free()
```

end_flag.gd

Found in folder: **/Scripts**

This script for the end flag extends the Area2D node and listens for a body to enter its area. When a body enters, it checks if the body is in the "Player" group, and if so, it changes the scene to the one specified in the `scene_to_load` variable. If the body is not in the "Player" group, the function does nothing and returns.

```
extends Area2D

@export var scene_to_load : PackedScene

func _on_body_entered(body):
 if not body.is_in_group("Player"):
```

```
  return

 call_deferred("_load_new_scene")

func _load_new_scene ():
 get_tree().change_scene_to_packed(scene_to_load)
```

enemy.gd

Found in folder: **/Scripts**

This code controls the movement of an enemy object in a 2D space. It moves the enemy in a specified direction at a set speed, and when it reaches its target position, it reverses direction and moves back to its starting position. The enemy object also deals damage to the player when they collide with it.

```
extends Area2D

@export var move_direction : Vector2
@export var move_speed : float = 20

@onready var start_pos : Vector2 = global_position
@onready var target_pos : Vector2 = global_position + move_direction

func _ready ():
 $AnimationPlayer.play("fly")

func _physics_process(delta):
 global_position = global_position.move_toward(target_pos, move_speed * delta)

 if global_position == target_pos:
  if target_pos == start_pos:
   target_pos = start_pos + move_direction
  else:
   target_pos = start_pos

func _on_body_entered(body):
 if not body.is_in_group("Player"):
  return

 body.take_damage(1)
```

menu.gd

Found in folder: **/Scripts**

This code handles the behavior of two buttons on the main menu. When the play button is pressed, it resets the player’s score to 0 and changes the scene to the first level. When the quit button is pressed, it closes the game.

```
extends Control

func _on_play_button_pressed():
 PlayerStats.score = 0
 get_tree().change_scene_to_file("res://Scenes/level_1.tscn")

func _on_quit_button_pressed():
 get_tree().quit()
```

player.gd

Found in folder: **/Scripts**

This script controls the player character's movement, health, and score. It handles user input for movement and jumping, applies gravity, and updates the character's animation and sound effects accordingly. The script also manages the character's health and score, emitting signals when either value changes and triggering a game over when the character's health reaches zero.

```
extends CharacterBody2D

signal OnUpdateHealth (health : int)
signal OnUpdateScore (score : int)

@export var move_speed : float = 100
@export var acceleration : float = 50
@export var braking : float = 20
@export var gravity : float = 500
@export var jump_force : float = 200

@export var health : int = 3

var move_input : float

@onready var sprite : Sprite2D = $Sprite
@onready var anim : AnimationPlayer = $AnimationPlayer
@onready var audio : AudioStreamPlayer = $AudioStreamPlayer

var take_damage_sfx : AudioStream = preload("res://Audio/take_damage.wav")
var coin_sfx : AudioStream = preload("res://Audio/coin.wav")

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

 # get the move input
 move_input = Input.get_axis("move_left", "move_right")

 # movement
 if move_input != 0:
  velocity.x = lerp(velocity.x, move_input * move_speed, acceleration * delta)
 else:
```



```
  velocity.x = lerp(velocity.x, 0.0, braking * delta)

 # jumping
 if Input.is_action_pressed("jump") and is_on_floor():
  velocity.y = -jump_force

 move_and_slide()

func _process(_delta):
 if velocity.x != 0:
  sprite.flip_h = velocity.x > 0

 if global_position.y > 200:
  game_over()

 _manage_animation()

func _manage_animation ():
 if not is_on_floor():
  anim.play("jump")
 elif move_input != 0:
  anim.play("move")
 else:
  anim.play("idle")

func take_damage (amount : int):
 health -= amount
 OnUpdateHealth.emit(health)
 _damage_flash()
 play_sound(take_damage_sfx)

 if health <= 0:
  call_deferred("game_over")

func game_over ():
 get_tree().change_scene_to_file("res://Scenes/menu.tscn")

func increase_score (amount : int):
 PlayerStats.score += amount
 OnUpdateScore.emit(PlayerStats.score)
 play_sound(coin_sfx)

func _damage_flash ():
 sprite.modulate = Color.RED
 await get_tree().create_timer(0.05).timeout
 sprite.modulate = Color.WHITE

func play_sound (sound : AudioStream):
 audio.stream = sound
 audio.play()
```

player_stats.gd

Found in folder: **/Scripts**

This code defines a global script that can be carried across scenes. It declares a single variable named “score” with a data type of integer and initializes it to 0.

```
extends Node

var score : int = 0
```

player_ui.gd

Found in folder: **/Scripts**

This code manages the display of a player’s health and score. It connects to the player’s health and score update signals to update the display in real-time. The health is displayed as a series of hearts that are shown or hidden based on the player’s current health, and the score is displayed as a text label.

```
extends CanvasLayer

@onready var health_container = $HealthContainer
var hearts : Array = []

@onready var score_text : Label = $ScoreText

@onready var player = get_parent()

func _ready ():
 hearts = health_container.get_children()

 player.OnUpdateHealth.connect(_update_hearts)
 player.OnUpdateScore.connect(_update_score)

 _update_hearts(player.health)
 _update_score(PlayerStats.score)

func _update_hearts (health : int):
 for i in len(hearts):
  hearts[i].visible = i < health

func _update_score (score : int):
 score_text.text = "Score: " + str(score)
```