7
Creating a 2D Mini-Game in Godot – Part 2

In the previous chapter, we guided you through the process of creating your first complete 2D platformer level in Godot. We built the level using a **TileMap**, set up the player character with animations, and cleaned up the default basic movement script. Now, it's time to take things further by adding more complex mechanics and refining how the player interacts with the game world.

In this chapter, you'll learn how to trigger player animations directly through code. We'll also dive into custom behaviors such as wall-sliding, double-jumping, and falling through platforms—common mechanics in many platform games. We will also add collectible items and a patrolling enemy. Finally, we'll script a checkpoint system for completing levels, allowing players to progress through the game.

In this chapter, we're going to cover the following main topics:

- Controlling player animations with code
- Wall-sliding and double-jumping mechanics
- Falling through platforms
- Adding collectible items
- Adding a patrolling enemy
- Implementing level completion

By the end of this chapter, you'll have a deeper understanding of how to bring your game mechanics to life with code and create a more dynamic and engaging platformer experience.

Technical requirements

By this point in the book, you should know how to do the following:

- Create nodes
- Create scenes

You should also know about variables and functions for use in GDScript (see ***Chapter 4***).

This chapter's code files are available here in the book's GitHub repository: `https://github.com/PacktPublishing/Godot-4-for-Beginners/tree/main/ch7`

Visit this link to check out the video of the code being run: `https://packt.link/goVK2`

The game assets used in this project are released under a **Creative Commons Zero** (**CC0**) license by Pixel Frog. Pixel Frog has allowed us to distribute, remix, adapt, and build upon the material in any medium or format, even for commercial purposes.

They can be found here:

- **Pixel Adventure 1**: https://pixelfrog-assets.itch.io/pixel-adventure-1
- **Pixel Adventure 2**: https://pixelfrog-assets.itch.io/pixel-adventure-2

Controlling player animations with code

Animations play a crucial role in bringing your game to life, making player actions feel smooth and responsive. In this section, we'll explore how to trigger different player animations in code. By learning how to control animations programmatically, you'll gain the ability to seamlessly transition between movement states such as idle, running, and jumping, enhancing the overall feel and polish of your game.

Take a moment to consider the conditions that will trigger each animation that we created in the previous chapter. We should try to imagine what will cause the animation to play so that we can implement it in the code:

- The *idle* animation is triggered when the player is stationary (both `x` and `y` velocities are `0`).
- The *run* animation is triggered when the player moves left or right. To ensure the sprite faces the correct direction, we'll flip it based on movement direction.
- The *fall* animation is triggered when the player is moving downward (the `y` velocity is increasing) and is not wall-sliding.
- The *jump* animation plays when the player is moving upward (the `y` velocity is decreasing) during their first jump (to prevent the player from spamming *jump* to *float* or *fly*).
- The *double-jump* animation occurs when the player is already airborne, jumps again, and is not wall-sliding.
- The *wall-slide* animation triggers when the player is sliding down a wall and pressing toward the wall. The sprite will flip to face away from the wall they are sliding on.

In the previous chapter, we imported and named all the animations, and we now have a clear understanding of the conditions under which each animation should be triggered. However, to implement these animations effectively, there are some missing pieces:

- We have no way of knowing whether the player is currently wall-sliding
- We have no way of knowing whether this is the player's first jump
- We have no way of knowing whether this is a double jump

These checks are crucial because our animations need to accurately represent the player's actions and state. For instance, triggering the wall-slide animation requires us to detect when the player is sliding against a wall. Similarly, distinguishing between the first jump and a double jump ensures that the correct animation plays for each action.

Helper variables

By addressing these gaps with **helper variables**, we can create a more dynamic and responsive player character, where the animations align seamlessly with the gameplay mechanics established in the previous chapter.

To ensure our character animations reflect the player's actions, we need to track certain states, such as whether the player is wall-sliding or how many jumps they've performed. To do this, we'll introduce two new variables.

Add the following variables at the top of your `Player` script, just after the constants:

```
var jump_count = 0
var is_wall_sliding = false
```

These variables will help us manage and detect the conditions needed to trigger the correct animations.

The preceding variables will solve our problems; we just need to use them in the correct place. Starting with the `jump_count` variable. When the player jumps or double jumps, we increase `jump_count` by `1`. When the player is on the floor again, we reset `jump_count` to `0`.

In the `handle_jump()` function, increment `jump_count` as follows:

```
func handle_jump():
    # Reset jump count if the player is on the floor
    if is_on_floor():
        jump_count = 0
    # Jump from the floor (first jump)
    if Input.is_action_just_pressed("ui_accept") and is_on_floor():
        velocity.y = JUMP_VELOCITY
        jump_count += 1
    # Double jump in the air
    elif Input.is_action_just_pressed("ui_accept") and jump_count < 2:
        velocity.y = JUMP_VELOCITY
        jump_count += 1
```

In the preceding code, when the player jumps, `jump_count` is increased by `1`, and when the player lands on the floor, `jump_count` resets to `0`.

Now that we've added the necessary variables to track the player's state, we can start implementing mechanics that use them in the next section.

Wall-sliding and double-jumping mechanics

In this section, we will create dedicated functions to handle *wall-sliding* and *double-jumping*, ensuring smooth and responsive movement for the player. These mechanics will not only enhance gameplay but also tie directly into triggering the appropriate animations. We will also create an *animate* function that calls all the specific animations.

Wall-sliding and double-jumping are unique player movements that require additional logic to function correctly. By separating these actions into their own functions, we can keep the code organized and easier to maintain while ensuring that each mechanic behaves as intended. Let's start by creating these dedicated functions.

Double-jump function

The **double jump** is a classic mechanic in platformer games, giving players an extra boost and greater control over their movement. To implement this feature, we need to track when the player is in the air and determine whether a second jump is allowed.

This section will walk you through creating a function to handle the double-jump logic, ensuring a smooth and responsive experience for the player.

Let's look at the following code:

```
func double_jump():
    # Handle double jump
    if Input.is_action_just_pressed("ui_accept") and !is_on_floor() and jump_count <
        velocity.y = JUMP_VELOCITY
        jump_count += 1
```

In the preceding double-jump function, first, check whether the player has pressed the *jump* key and whether they are currently in the air (not on the floor) and the jump count is less than `2` (to prevent flying). If all these conditions are true, then the player can jump a second time.

Wall-slide function

Wall-sliding depends on another value to work: **friction**. This is so that we can slide down the wall at a slower rate than if we fall. To see that in action, add a new constant at the top of the program as follows:

```
const SPEED = 150.0
const JUMP_VELOCITY = -400.0
const FRICTION = 100
```

The preceding constant called `friction` will be the speed at which we slide down the wall. The value of `100` is chosen because it delivers the best effect; however, you could choose any number you feel looks best to you when testing.

Now, implement the `wall_slide` function:

```
func wall_slide(delta):
    if is_on_wall() and !is_on_floor():
        if Input.is_action_pressed("ui_left") or Input.is_action_pressed("ui_right")
            is_wall_sliding = true
        else:
            is_wall_sliding = false
    else:
        is_wall_sliding = false
    if is_wall_sliding:
        velocity.y = min(velocity.y, FRICTION)
```

The `wall_slide` function enables the wall-sliding mechanic for the player by checking specific conditions and modifying their movement. To better understand how this function works, let's break it down step by step in the following sections.

Checking conditions

The first `if` condition checks whether the player is touching a wall (`is_on_wall()`) and is not grounded (`!is_on_floor()`).

This verifies two things:

- Whether the player is touching a wall using the `is_on_wall()` function
- Whether the player is not grounded, as determined by the `!is_on_floor()` condition

Detecting input

Inside the wall detection block, the following line checks for player input:

```
if Input.is_action_pressed("ui_left") or Input.is_action_pressed("ui_right")
```

This condition determines whether the player is pressing either the *left* (`ui_left`) or *right* (`ui_right`) movement keys. This leads to two scenarios:

- If either key is pressed, the code sets `is_wall_sliding = true`, enabling the wall slide
- If no key is pressed, the variable is set to `false`, stopping the wall slide

Resetting the wall slide

The `else` block ensures that if the player is not on a wall or is standing on the floor, `is_wall_sliding` is set to `false`. This prevents unintended sliding.

Limiting downward speed

The final check, `if is_wall_sliding:`, applies the line `velocity.y = min(velocity.y, FRICTION)` inside its block.

Here, the `min()` function compares the current downward velocity (`velocity.y`) with the predefined `FRICTION` value. If `velocity.y` is greater than `FRICTION` (meaning the player is falling too fast), `min()` returns the smaller of the two—the `FRICTION` value. This effectively caps the falling speed during a wall slide, ensuring the player descends at a controlled, slower rate.

This results in a smoother and more manageable wall-sliding experience, preventing rapid, uncontrolled drops down the wall.

By combining these checks and actions, the `wall_slide` function ensures the player can seamlessly transition into and out of a controlled wall slide based on their input and environment. This sets the foundation for intuitive gameplay mechanics.

Remember to call our new functions in the process function so that we can use them:

```
func _physics_process(delta):
    apply_gravity(delta)
    double_jump()
    wall_slide(delta)
    move_and_slide()
```

The `_physics_process(delta)` function is where all our custom movement and physics-related functions are called. By calling them, we are asking Godot to run these functions as part of the physics engine. `_physics_process()` is part of Godot's game loop and runs at a fixed interval, ensuring that the player's actions and interactions with the game world are updated consistently and in real time.

It's important to use `_physics_process()` for physics-based logic because it's synced with the physics engine. If we used `_process()` instead—which runs every rendered frame and can vary depending on frame rate—it could lead to jittery or inconsistent movement, especially on lower-performance machines. This is because `_process()` does not guarantee consistent timing, which is crucial for physics calculations.

Animate function

The next step is to rewrite the animation trigger conditions in the code. Each animation trigger is described with a comment.

The `AnimatedSprite2D` node will play the corresponding animation, but we need a reference to the node to call its functions. We can get the reference as the scene is ready (loaded), and so we give the variable the `@onready` annotation, as shown in the following code:

```
extends CharacterBody2D
@onready var animations = $AnimatedSprite2D
```

Next, we'll declare a new function and call it `animate`:

```
#Handle Animations
func animate():
    # Stationary player
    if velocity.x == 0 and velocity.y == 0:
        animations.play("idle")
    # Player moving horizontally (left or right)
    elif velocity.y == 0:
        animations.play("run")
    # Player is falling
    elif velocity.y > 0 and !is_wall_sliding:
        animations.play("fall")
    # Player is jumping (first jump)
    elif velocity.y < 0 and jumpCount == 0:
        animations.play("jump")
    # Player is double-jumping
    elif !is_on_floor() and jumpCount > 0 and !is_wall_sliding:
        animations.play("double_jump")
    # Player is sliding on wall
    elif is_wall_sliding:
        animations.play("wall_slide")
    # Flip the sprite based on movement direction
    animations.flip_h = velocity.x < 0
```

Remember to call the `animate()` function in the `_physics_process()` function:

```
func _physics_process(delta):
    apply_gravity(delta)
    standard_player_movement()
    double_jump()
    wall_slide(delta)
    animate()
    move_and_slide()
```

In the preceding code, we have again made sure to call all our custom functions to run repeatedly in the `physics_process` function.

Now, test the game and your player should animate! Our next step is to set up the *pass-through platforms* so that the player can jump up through a platform from below and press down to fall through the platform from above.

Falling through platforms

We created two different physics layers on the TileMap in ***Chapter 6*** so that the player can jump up and fall down through certain platforms. Because we made these tiles as *one-way tiles*, we can jump through them from below, with no need for coding.

Now, we need to set up the code that will allow our player to fall through these platforms when we are standing on them and pressing *down*. Our code should check whether the player is pressing *down* and then simply turn off **Collision Mask** number `3` (the pass-through layer). This means that the player will stop detecting collisions on the `pass_through` layer and will fall until it reaches a layer that it is detecting collisions for, which is the ground layer. This is demonstrated in the `check_pass_through()` function as shown here:

```
func check_pass_through():
    if Input.is_action_pressed("ui_down"):
        set_collision_mask_value(3, false)
    else:
        set_collision_mask_value(3, true)
```

As can be seen in the preceding code, a new function has been created named `check_pass_through()`. It then checks whether the user is pressing the *down* key and, if so, it sets the value of **Collision Mask** layer number `3` to `false` (disabled). If the player is not pressing the *down* key, the value will be reset to `true` (enabled).

Remember to add `check_pass_through()` to your list of function calls in the `_physics_process()`; otherwise, the function will not run!

```
func _physics_process(delta):
    apply_gravity(delta)
    standard_player_movement()
    double_jump()
    wall_slide(delta)
    animate()
    check_pass_through()
    move_and_slide()
```

The player can now run, jump, double-jump, and wall-slide around the level as well as fall through special platforms. Our next step is to scatter some items around the level for the player to collect.

Adding collectible items

In many games, **collectible items** play a vital role in enhancing gameplay by providing rewards, power-ups, or essential resources to the player. These items can create a sense of progression, encourage exploration, and add layers of strategy to the game. In this section, we'll explore how to create a collectible item system in Godot.

You'll learn how to design a collectible item, detect when the player interacts with it, and implement logic to grant rewards or effects. By the end of this section, you'll have a reusable system that can be extended to include a variety of collectibles for your levels.

For the collectible item, we will use the Strawberry found in the `Items` | `Fruits` folder. However, you may use any item you wish as long as you adapt some of the steps that follow.

Strawberry scene (our collectible item)

For our game, the player will collect Strawberries. Once the player has collected all of the Strawberries on a level, a flag will appear which the player will touch to complete the level.

Follow these steps to create the collectible item scene:

1. Create a new scene and add **Area2D** as the root node. **Area2D** is an invisible node that defines a region of 2D space, which detects when objects have collided or stopped colliding with it.
2. Add an **AnimatedSprite2D** node as a child node.
3. Recall that to animate a sprite, we use multiple frames that depict the sprite at various stages of motion. These frames are often grouped together in a single image known as a **sprite sheet**.
4. To animate the sprite, we first import the sprite sheet and then divide it into individual frames. To do this, select the **AnimatedSprite2D** node, and in the **Sprite Frames** property in **Inspector**, click on **New SpriteFrames**. This is shown in *Figure 7.1*:

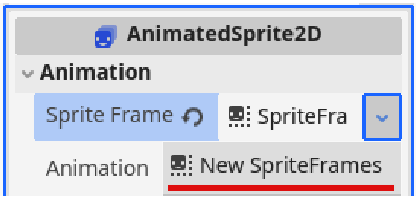


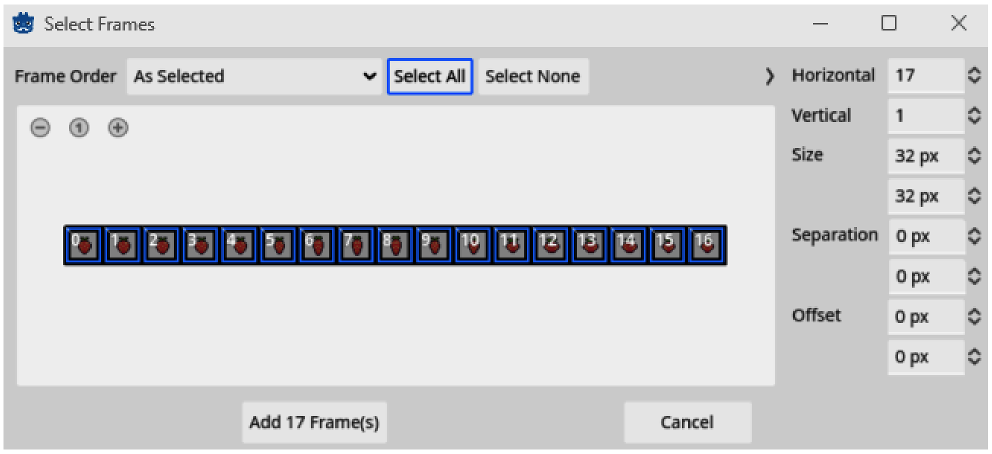
Figure 7.1 – Creating new SpriteFrames

5. Clicking on **New SpriteFrames** in **Inspector**, as shown in *Figure 7.1*, will open the **SpriteFrames** area at the bottom of the screen. This is the space where we can import frames and create and name new animations. This is shown in *Figure 7.2*:

Figure 7.2 – The SpriteFrames area

As shown in *Figure 7.2*, we currently have only the default animation and no images or frames associated with it. We will now create animations for the `idle` animation and then import the sprites associated with it from the sprite sheet.

6. Start by renaming the default animation to `idle`.
7. In **FileSystem**, create a new folder called `Collectibles`.
8. Now, find the `Items` | `Fruits` folder in the assets you downloaded from Pixel Frog (see the *Technical requirements* section) and drag the images for your Strawberry into the **Collectibles** folder in **FileSystem**.
9. Select the **idle** animation and then click on **Add frames from a Sprite Sheet** button or press *Ctrl* + *Shift* + *O*. Now, select the `Idle` image file in your `Collectibles` folder.
10. Now, you must count the player images and update the **Horizontal** and **Vertical** frame properties (see *Figure 7.3*) as required.
11. Then, click on the **Select All** button followed by the **Add x Frame(s)** button. This is shown for the idle frames of the Strawberry in *Figure 7.3*:



Horizontal 17
Vertical 1
Size 32 px 32 px
Separation 0 px 0 px
Offset 0 px 0 px

Figure 7.3 – Setting the Horizontal and Vertical frames and adding them to the animation

Now that you have cut the individual frames from the image (sprite sheet) as in *Figure 7.3*, your **SpriteFrames** window will update with the images to be used for the `idle` animation. This is shown in *Figure 7.4*:

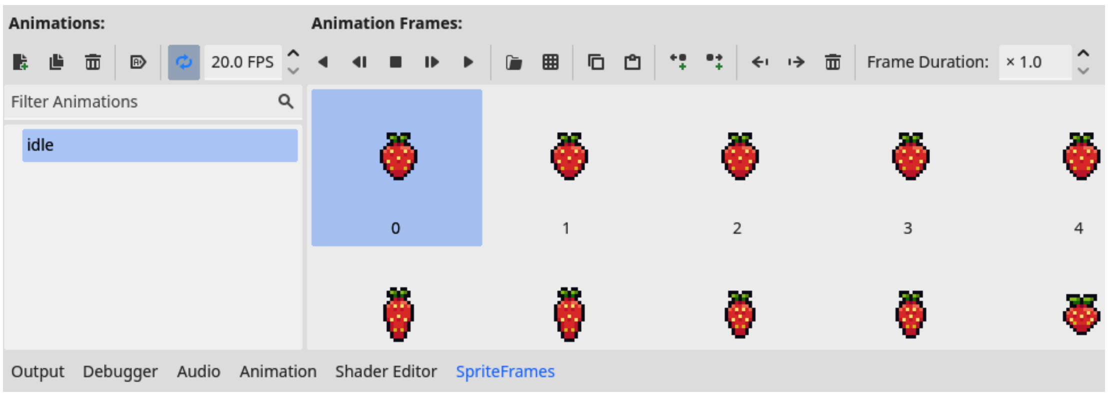


Figure 7.4 – The frames used for the idle animation

12. As you can see in *Figure 7.4*, there are controls for the animations in the bar at the top. I have set the idle frame rate to **20 FPS**, and looping is on. You can play with these settings to find the ones you like. You can see this in *Figure 7.5*:

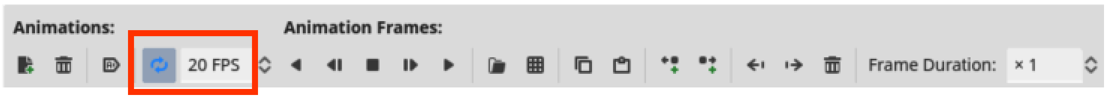


Figure 7.5 – Animation controls with looping and FPS highlighted

13. To complete the setup of the Strawberry collectible, we need to add a **CollisionShape2D** node as a child of the **Area2D** node. For the **Shape** property of the node, select **CapsuleShape2D** and adjust it in the Viewport to fit the Strawberry. This is shown in *Figure 7.6*:

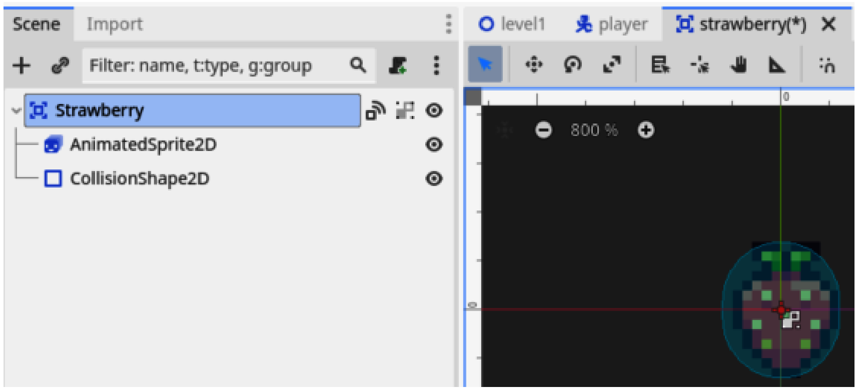


Figure 7.6 – Adjusting the collision to fit the Strawberry

The Strawberry can now collide with objects and detect when an object has collided with it. We will need to write code to animate the Strawberry and to do something when the player collides with it, such as removing the Strawberry and adding *1* to the player's Strawberry collection score.

Implementing the Strawberry script

The Strawberry only needs to do two things: animate and disappear when the player collects it. We will now program the Strawberry to do these two things.

In the **Strawberry** scene, select your **Strawberry** node and click on the **Attach New Script** button/icon, as shown in *Figure 7.7*:

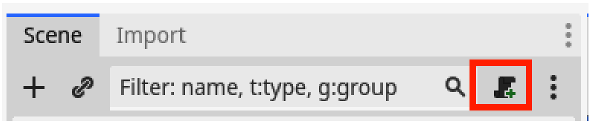


Figure 7.7 – The Attach New Script button/icon

The `idle` animation must play repeatedly; we have already set it to loop, so, in the `ready` function, we just need to start the animation. To make it easier to reference other nodes in a scene, Godot provides a built-in shortcut. By preceding a node's name with a dollar sign (`$`), we can quickly access that node and its functions directly.

This shortcut is particularly useful when frequently interacting with other nodes in a scene, such as controlling a character's animations or updating UI elements, as it keeps the code more readable and efficient. Use this shortcut to play the `idle` animation of the Strawberry, as shown in the following code:

```
func _ready():
    $AnimatedSprite2D.play("idle")
```

When our player collides with or overlaps a Strawberry, we want to *signal* the Strawberry to execute a function. In the function, we can remove the Strawberry from the level and add 1 to the number of Strawberries collected by the player. **Area2D** nodes can respond to a variety of built-in signals, one of which is when a body enters the area of the node.

To set this up, select the **Strawberry** node, then click on the **Node** tab (to the right of the **Inspector** tab). In the **Signals** category, under **Area2D**, select the **body_entered** signal and click on **Connect** in the bottom-right corner, or simply double-click the signal name. This is shown in *Figure 7.8*:

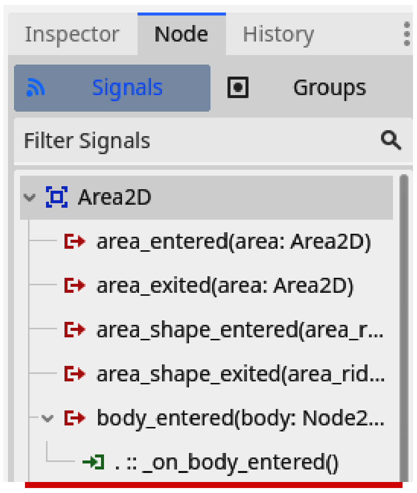

Inspector
Node
History
Signals
Groups
Filter Signals
Area2D
area_entered(area: Area2D)
area_exited(area: Area2D)
area_shape_entered(area_r...
area_shape_exited(area_rid...
body_entered(body: Node2...
. :: _on_body_entered()

Figure 7.8 – The body_entered signal of Area2D

A new window will pop up, which is shown in *Figure 7.9*. You need not worry about the details now and simply click on the **Connect** button.

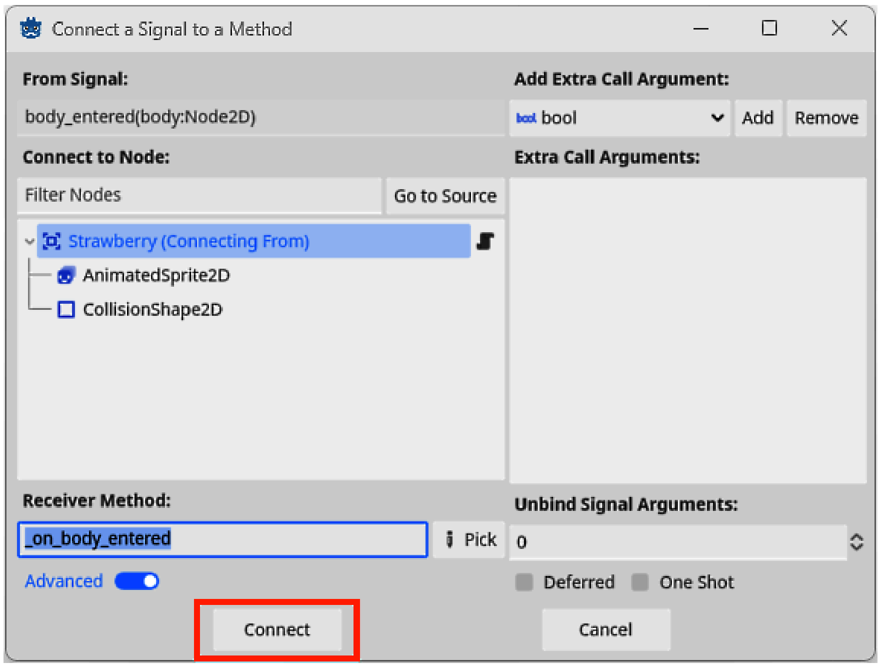


Figure 7.9 – Connecting a signal to a method (function)

Clicking on the **Connect** button, as shown in *Figure 7.9*, will create a new function in the `Strawberry` script called `_on_body_entered(body)`. The parameter called `body` contains a reference to the object that collided with the Strawberry. To determine whether that *body* is the *player*, we can tag the player as part of a group called `Player`.

Here's how you do it:

1. Switch to your **Player** scene and select the **Player** node.
2. Again, click on the **Node** tab to the right of **Inspector**, then choose the **Groups** sub-tab.
3. Click on the + symbol to add a group.
4. Name the group `Player`.

These steps are shown in *Figure 7.10*.

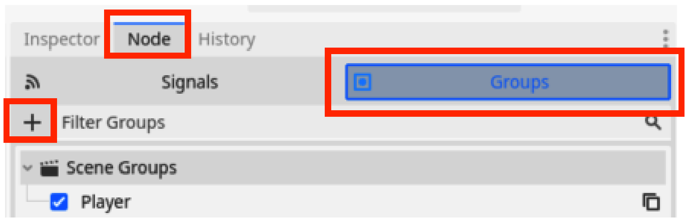


Figure 7.10 – The Node tab, Groups sub-tab, and the + symbol to add a new group

With the group ready, we can check whether the body that entered the Strawberry is in the **Player** group and if so, react accordingly. We can do this in the `Strawberry` script as shown:

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        #add to player score
        #remove the strawberry
```

`Player` will keep track of how many Strawberries it has collected. We need a variable in the `Player` script to keep count of the number of Strawberries the player has collected, and we need a function in the `Player` script to add one to the score each time a Strawberry is collected.

Return to the `Player` script and add the variable and function as shown in the code:

```
var jumpCount = 0
var is_wall_sliding = false
var strawberry_count = 0
func add_score(amount):
    strawberry_count += amount
```

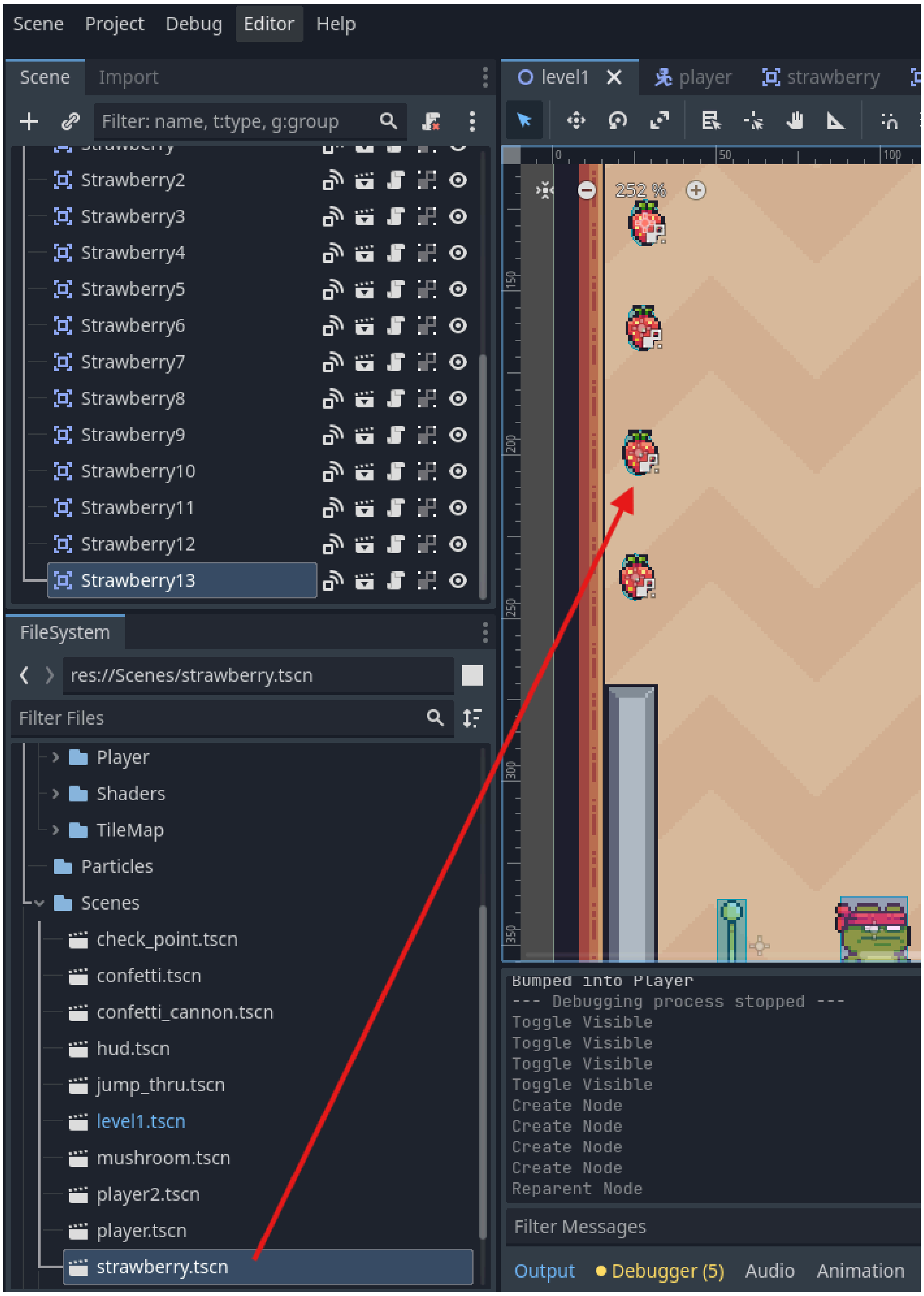
The preceding code will increase the `strawberry_count` variable by an amount each time it is called. We will call it in the `_on_body_entered(body)` function of the Strawberry and remove the Strawberry from the scene once collected, as shown here:

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

The built-in `queue_free()` function deletes a node from memory at the end of the current frame as soon as it is safe to do so. It is the recommended method for removing nodes from the scene.

With this code in place, when the Strawberry signals that another body has entered its space, a value of `1` will be added to the Strawberry count of the player, and the Strawberry will be removed from the scene.

Feel free to drag and drop the `strawberry.tscn` file into your level multiple times so that the player has something to collect, as shown in *Figure 7.11*:



```
Bumped into Player
--- Debugging process stopped ---
Toggle Visible
Toggle Visible
Toggle Visible
Toggle Visible
Create Node
Create Node
Create Node
Create Node
Reparent Node
```

Figure 7.11 – Dragging and dropping Strawberries into the level

The player can move through the level and collect Strawberries. To add a challenge, we need an enemy to patrol.

Adding a patrolling enemy

This is a reminder that the enemies can be found here: https://pixelfrog-assets.itch.io/pixel-adventure-2.

We will be using the **Mushroom** enemy. Create a new scene and set it up with **CharacterBody2D**, **AnimatedSprite2D**, and **CollisionShape2D** with **CapsuleShape2D** only covering part of the mushroom, as shown in *Figure 7.12*:

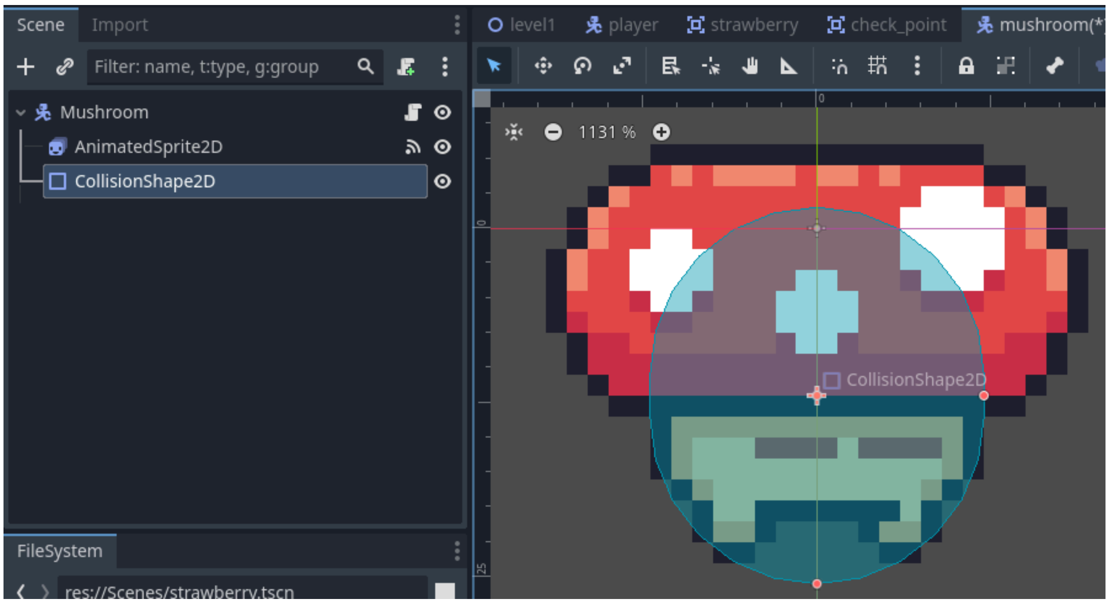


Figure 7.12 – Setting up the Mushroom scene

Use the sprite sheets provided in the `Mushroom` enemy folder and set up the SpriteFrames and animations for **idle** and **run** to be looped at **20 FPS**. I created my own **death** animation using one frame from **idle** and rotating it, as shown in *Figure 7.13*:

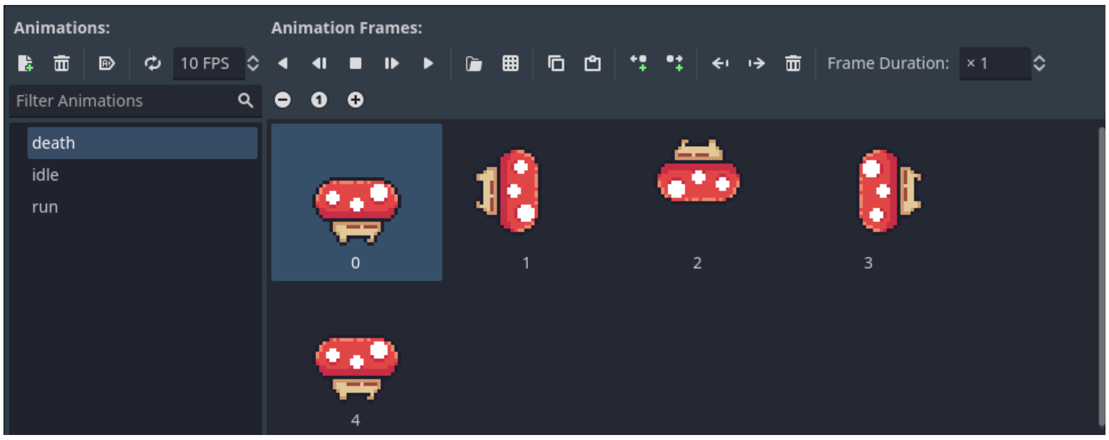
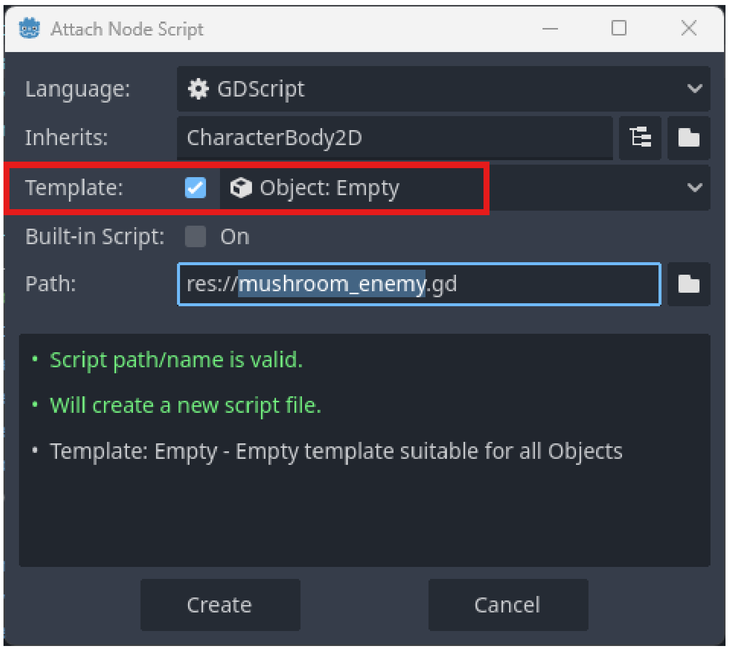


Figure 7.13 – Animations for the Mushroom enemy

Attach a new script to the **Mushroom** node and ensure that the template used is an empty object, as shown in *Figure 7.14*:



Language: GDScript
Inherits: CharacterBody2D
Template: ✓ Object: Empty
Built-in Script: On
Path: res://mushroom_enemy.gd

- Script path/name is valid.
- Will create a new script file.
- Template: Empty - Empty template suitable for all Objects

Figure 7.14 – Using the Object: Empty template for the Mushroom enemy

Using the **Empty** template will ensure that our script is created with no default or added code and is as simple as possible.

Add code to set values for the speed, direction, and health of the Mushroom, as shown in the code:

```
extends CharacterBody2D
const SPEED = 75
var direction = 1
var health = 1
```

Note that the `direction` variable will have values of `-1` or `1` to indicate left or right move-ment, respectively.

Now, we have variables to control the speed, direction, and health of the Mushroom enemy.

Also, add a function to apply gravity to the Mushroom, allowing it to fall and touch the ground, as shown in the following code:

```
func add_gravity(delta):
    # Add gravity.
    if not is_on_floor():
        velocity += get_gravity() * delta
```

If the Mushroom is not on the ground, gravity will pull it downward.

Now, create a function to make the Mushroom move. I have called mine `update_direction()` and all it does is make the enemy move to the right at a constant speed, as shown in the code:

```
func update_direction():
    # Move enemy at constant speed
    velocity.x = SPEED * direction
```

The velocity of the Mushroom in the horizontal plane is multiplied by speed and direction.

Both functions must also be called in the `_physics_process(delta)` function along with `move_and_slide()`, as shown in the code:

```
func _physics_process(delta):
    add_gravity(delta)
    update_direction()
    move_and_slide()
```

We must also set collisions on the Mushroom so that it can collide with the floor and the player. Do this by turning on **Collision Layer 1** and turning on **Collision Mask 1**, **2**, and **3**.

If you drop a Mushroom into the level now and run the game, the Mushroom will begin moving to the right until it collides with a wall.

We will detect when a Mushroom has collided with a wall using the built-in `is_on_wall()` function and reverse its direction, as shown in the code:

```
func reverse_direction():
    #Reverse direction when hitting a wall
    if is_on_wall():
        direction = -direction
```

Note that by inverting or negating the value of direction, we can determine whether the Mushroom moves to the left or the right.

Make sure to call `reverse_direction()` after you have called `move_and_slide()`, as shown in the following code, because otherwise, collisions will not be detected correctly:

```
func _physics_process(delta):
    add_gravity(delta)
    moveEnemy()
    move_and_slide()
    reverse_direction()
```

If you place your Mushroom on a floating platform, when it reaches the edge of the platform, it will fall off. To prevent this, use a **RayCast2D** node to detect when the Mushroom is near the edge and then reverse direction. Add a **RayCast2D** node as a child of **Mushroom**, as shown in *Figure 7.15*:

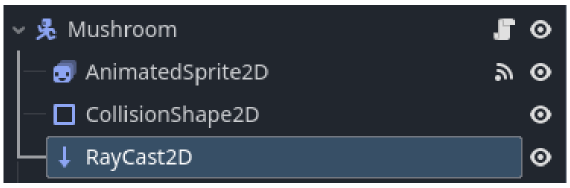


Figure 7.15 – Adding a RayCast node to our Mushroom node

RayCast2D is simply an invisible ray cast by an object that looks for collisions in the direction in which it was sent. In this case, we are looking for collisions below us to determine when we are no longer on the floor. Our ray is cast a little to the right of our Mushroom (I used **Target Position x**: `0`, **y**: `14` and **Transform Position x**: `15` and **y**: `7`) so that we can see when we are approaching the *no floor zone* and reverse the direction before we fall off! Place your ray as shown in *Figure 7.16*:

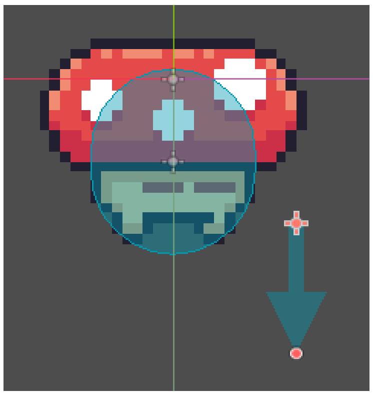


Figure 7.16 – Casting a ray downward from the right of the Mushroom

Now, we can write a function to reverse the direction of the Mushroom when the ray is not colliding with the floor and to move the ray to the opposite side so that it is always looking for no floor in the direction the Mushroom is moving, as shown:

```
func platform_edge():
    if not $RayCast2D.is_colliding():
        direction = -direction
        $RayCast2D.position.x *= -1
```

In the preceding code, as soon as the ray is no longer colliding with the ground, the Mushroom's movement direction is reversed and the ray is moved to the other side of the Mushroom to detect collisions in the direction that the Mushroom is moving. We must all turn on **Collision Mask 2** and **3** for the **RayCast2D** node so that it will detect the floor.

Remember to call this function after `move_and_slide()` in the `_physics_process()` function so that it runs after collisions have been detected. Finally, place a Mushroom on a platform to test that it patrols the platform and does not fall off.

Mushroom stomping

The player should be able to hurt the Mushroom only by attacking it from above. We will create a *hurt zone* or *death zone* for the Mushroom, which will consist of an **Area2D** and **CollisionShape2D** node with a **RectangleShape2D** node placed on top of the Mushroom.

The Mushroom will hurt the player if it attacks from the side. We will create a *hurt player* zone on either side of the Mushroom, again made up of **Area2D** and **CollisionShape2D** with **RectangleShape2D**. This is shown in *Figure 7.17*:

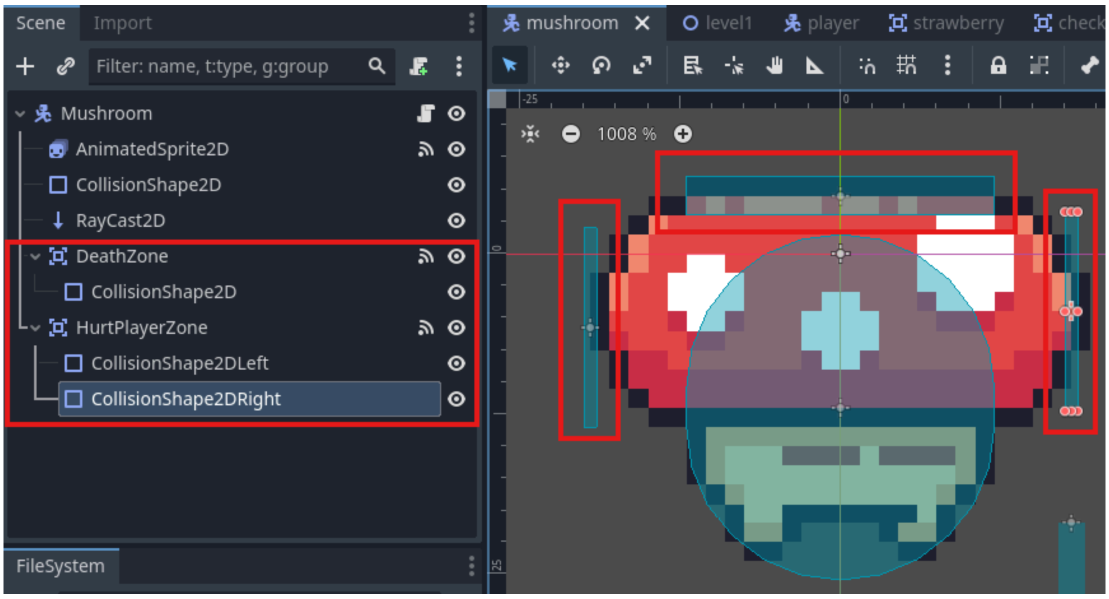


Figure 7.17 – Creating the death zone and hurt player zone

With these in place, we can make use of signals to detect when the player has entered one of the zones and then react accordingly.

Connect the `_on_death_zone_body_entered()` and `_on_hurt_player_zone_body_entered()` signals to the `Mushroom` script. Then, write code in the empty functions, as shown in the code:

```
func _on_death_zone_body_entered(body):
    if "Player" in body.name:
        body.velocity.y = -500
        health -= 1
func _on_hurt_player_zone_body_entered(body: Node2D):
    if "Player" in body.name:
        body.hurt()
```

There is no `hurt()` function in the `Player` script yet. We should create that now.

Open the `Player` script and add the following function so that the player can be hurt by the Mushroom enemies:

```
func hurt():
    print("Player hurt")
```

Take note that, for now, we just use a technique called **stub testing** in which we simply display a message to verify that the function was called correctly. Later, we will complete the full implementation of this function.

The `_on_death_zone_body_entered(body)` function runs whenever a body enters the *death zone* area (jumps onto the Mushroom from above). We check to see whether the body that entered the zone has the word `Player` in its name. This is an alternative and equivalent option to checking whether `body.is_in_group("Player")`. If the body does have the word `Player` in its name, we add `500` to the velocity of the player to make it appear to bounce off the Mushroom and we subtract `1` from the health of the Mushroom.

The `_on_hurt_player_zone_body_entered()` function runs when a body enters the Mushroom from the sides. Again, if this body is the player, we run the player's `hurt()` function, reducing the player's health.

When the Mushroom's health is `0`, it should disable collisions, stop detecting collisions with the player, and play the `death` animation. This is achieved with the custom function shown as follows:

```
func mushroom_death():
    if health <= 0:
        $CollisionShape2D.disabled = true
        $HurtPlayerZone.monitoring = false
        $AnimatedSprite2D.play("death")
```

The `mushroom_death()` function should be called in the `_physics_process()` function, as shown in the code:

```
func _physics_process(delta):
    add_gravity(delta)
    moveEnemy()
    move_and_slide()
    reverse_direction()
    platform_edge()
    mushroom_death()
```

Connect the `_on_animated_sprite_2d_animation_finished()` signal of the `AnimatedSprite2D` node of the Mushroom. When the `death` animation is finished, it will trigger the function to remove the Mushroom from the scene, as shown in the code:

```
func _on_animated_sprite_2d_animation_finished():
    if $AnimatedSprite2D.animation == "death":
        queue_free()
```

Add some more Mushrooms and collectibles to your level. To prevent crowding the scene tree, add a new child node of the **Node** type. Rename it `Enemies` and drag all of the Mushrooms under it. You can do the same for **Collectibles**, as shown in *Figure 7.18*:

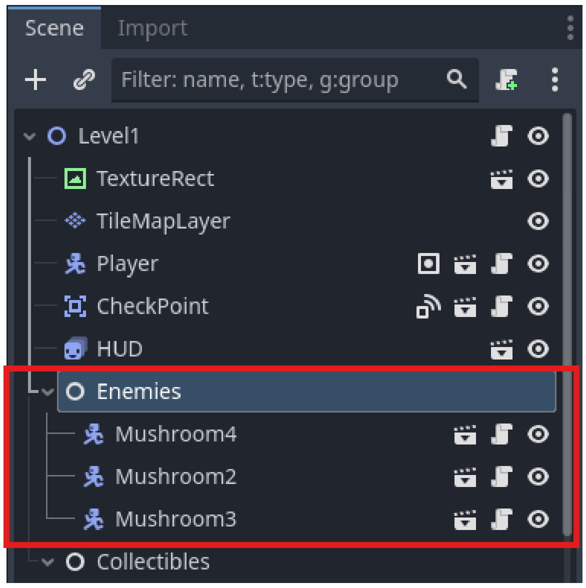


Figure 7.18 – Organizing the scene tree by grouping nodes

Now that the level has enemies and collectibles, we need a *win* condition. For our purposes, it will be that once the player has collected all of the Strawberries, a checkpoint flag will fly. Touching that flag will end the level. Let's learn more about level completion in the next section.

Implementing level completion

Level completion mechanics are a vital part of any game, providing players with a sense of accomplishment and closure. In this section, we'll create a system in which collecting all the Strawberries triggers a sequence of events: a checkpoint flag animates and flies out, signaling the completion of objectives. The player then moves to the flag, touching it to end the level.

This system not only adds visual and interactive flair but also reinforces the player's progress and goal achievement. By the end of this section, you'll have a polished and rewarding level completion sequence to enhance your game's overall experience.

Create a new scene with the following:

- **Area2D** as the root node (renamed to `CheckPoint`)
- **AnimatedSprite2D**
- **CollisionShape2D** with **RectangleShape2D**

The scene tree is shown in *Figure 7.19*:

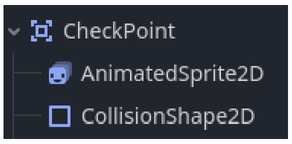


Figure 7.19 – Scene tree for the checkpoint scene

Create three animations for the flag, named `flag_fly`, `idle`, and `trigger`. In the `assets` folder from Pixel Frog, navigate to the `Items` | `Checkpoints` | `Checkpoint` folder.

- For the `idle` animation, use the single frame called `Checkpoint (No Flag).png`.
- For the `trigger` animation, use the `Checkpoint (Flag Out).png` sprite sheet. Do not loop the animation and run it at **20 FPS**.
- For the `flag_fly` animation, use the `Checkpoint (Flag Idle).png` sprite sheet. Loop the animation and run it at **20 FPS**.

An image of the `flag_fly` animation is shown as an example in *Figure 7.20*:

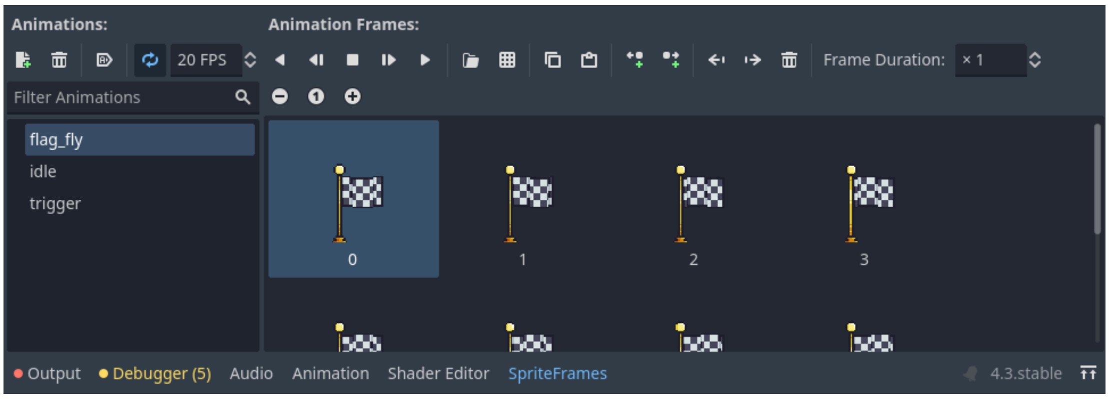


Figure 7.20 – Setting up the flag animations for the checkpoint

Attach a script to the **CheckPoint** node to customize its behavior. We will create our own signal to trigger the flag to fly out. Once the player has collected all of the Strawberries in the level, the player will emit the `trigger` signal. The `CheckPoint` code is shown and explained as follows:

```
extends Area2D
signal trigger
var level_complete = false
# signal that runs when the animation is finished playing
func _on_animated_sprite_2d_animation_finished():
    $AnimatedSprite2D.play("flag_fly")
    level_complete = true
# our own custom signal
func _on_trigger():
    $AnimatedSprite2D.play("trigger")
# A signal that runs when a body enters
func _on_body_entered(body):
    if level_complete:
        get_tree().quit()
```

As shown, we create our own signal and call it `trigger`. Then, in **Inspector**, we connect that signal to our script. We also have a Boolean variable called `level_complete` to determine when the level is complete. The `trigger` function linked to our signal plays the `trigger` animation and then signals that the animation is complete, which switches to the `flag_fly` animation and sets `level_complete` to `true`. For now, when the player touches the flag, the game will exit.

However, we need to emit the `trigger` signal from the `Player` script once they have collected all the Strawberries. Return to the `Player` script and add a new variable called `level_s-`

`trawberries` to keep track of the number of Strawberries in the level, as shown:

```
var strawberry_count = 0
var level_strawberries
```

First, we should create a group called `checkpoint` and add the **CheckPoint** node to that group. To do this, select the **CheckPoint** node in **Inspector**, select the **Node** tab, then choose the **Groups** sub-tab and add a new group called `checkpoint`. This is shown in *Figure 7.21*:

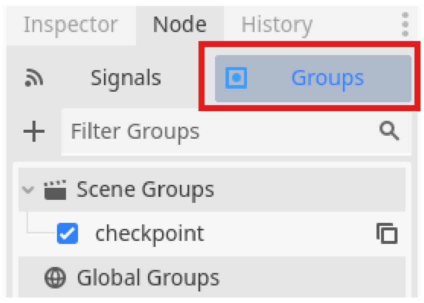


Figure 7.21 – Creating the checkpoint group

We also need a custom function that will check to see whether the level is complete and then emit the `trigger` signal. If `strawberry_count` (which is the number of Strawberries that the player has collected) matches `level_strawberries` (which is the total number of Strawberries in the level), then we get a reference to the checkpoint via its group name and emit the checkpoint's `trigger` signal. Finally, we reset the Strawberries to `0` so that the signal is only emitted once and not constantly. This is shown in the code as follows:

```
func complete_level():
    if strawberry_count == level_strawberries:
        var checkpoint =
        get_tree().get_first_node_in_group("checkpoint")
        checkpoint.emit_signal("trigger")
        strawberry_count = 0
```

Remember to call the new `complete_level()` function in the `_physics_process(delta)` function.

Since all the Strawberries are in the **Level1** scene, we need to attach a script to the **Level1** node so that it can count the Strawberries in the scene and update the `Player` variables. This is shown in *Figure 7.22*:

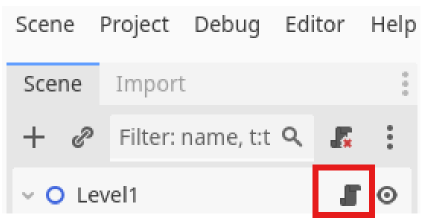


Figure 7.22 – Attaching a script to the level node

Counting the Strawberries in the level script is shown in the code as follows:

```
extends Node2D
@onready var player = $Player
@onready var strawberries = $Collectibles
func _ready() -> void:
    player.level_strawberries =
    strawberries.get_child_count()
```

In the preceding code, we get a reference to the `Player` scene as well as the `Collectibles` group of scenes (multiple Strawberries.) In the `_ready` function, we update the player's `level_strawberries` variable by setting it to the count of all the Strawberries in the level.

Now, add a **CheckPoint** scene to the level, and you have a way to complete it.

With that, the level is now playable!

Summary

In this chapter, we expanded our 2D platformer with essential gameplay mechanics and refined player interactions. We controlled player animations through code, ensuring smooth visual feedback for actions such as running, jumping, and wall-sliding. We also implemented wall-sliding and double-jumping, adding versatility to player movement, and introduced falling-through platforms for dynamic level navigation. Alongside these, we added collectible items to encourage exploration, a patrolling enemy to introduce a challenge, and a checkpoint for tracking level completion.

In the next chapter, we will transfer our 2D skills and make a 3D platformer!