CODE NINJAS®

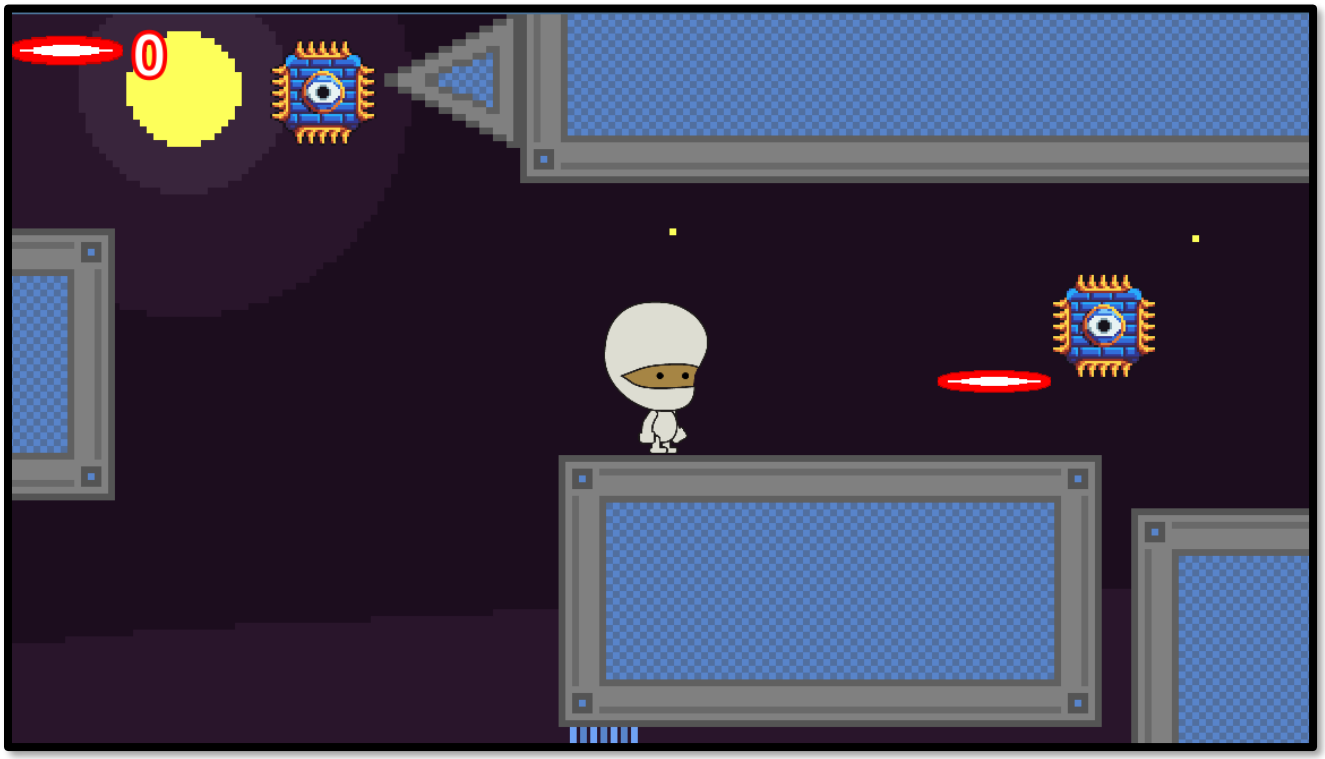
Platinum Belt Ninja Guide

PB Activity 01: Gravity Trails

Contents

PB Activity 01: Gravity Trails

Your goal is to plan, program, and playtest your very own platformer game. You will begin by building level 1 using a premade background. Then, you will plan and build a unique custom level using asset creation!



Requirement #1: Project Setup

Set up the project and the first level's scene.

- ☐ Import the Ninja Starter Pack as a new project in the correct installation path.
- ☐ Rename the project to **[YourInitials]GravityTrails** and open the project.
- ☐ Create a new **2D** scene named **level_1.tscn** in the **Levels** folder.
- ☐ Drag **Assets > background.jpg** to the origin of the scene (set its position in the Inspector manually if needed) and drag **Scenes > Objects > player.tscn** to the top-left platform on the background.
- ☐ Set the Player to always appear in front of the background.

Pause for **Ninja Stop #1**!

Does your project have...

- A level 1 scene?
- The background as a Sprite2D?
- The player at the top-left of the background?

Reminder: Save your work!

Requirement #1 Hints

- Which button imports files as a new project in the Project Manager?
- Where is the starter code stored? What file type is the starter pack?
- Where in the Inspector for Player can the Z-Index be changed to fix the layering?

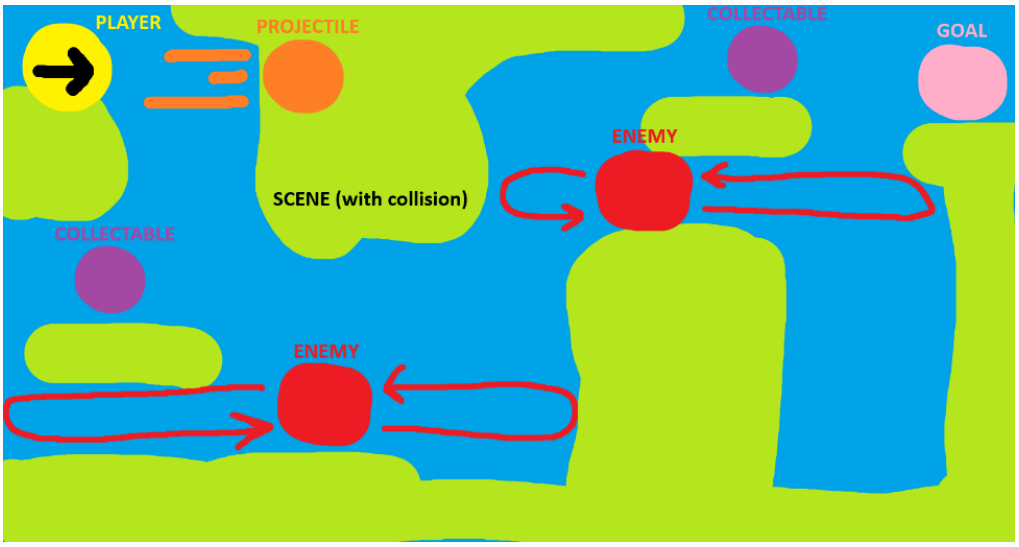
Requirement #1 Resources

- Refer back to Activities 06 – 17 in Silver Belt for help with importing a project.
- The Z Index property can be found in **Inspector > CanvasItem > Ordering > Z Index**.

Requirement #2 (Instructions): Camera & Scene Collision

1 When planning a game, drawing a simple sketch of the final version of the game can be a huge help. It sets the groundwork for all the major features of a game.

This sketch shows the different components of Level 1, including the Player, Enemies, Goal, Collectables, Projectiles, and Scene.



2 The next step is to decide which components have responsibility for certain actions. For example, the Player is the only component that should be responsible for throwing projectiles.

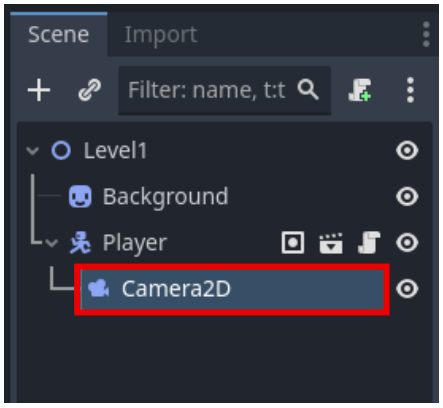
This can greatly help with writing the code for each component. To stay consistent, last step's sketch and this step's responsibility chart will be referenced throughout the project.

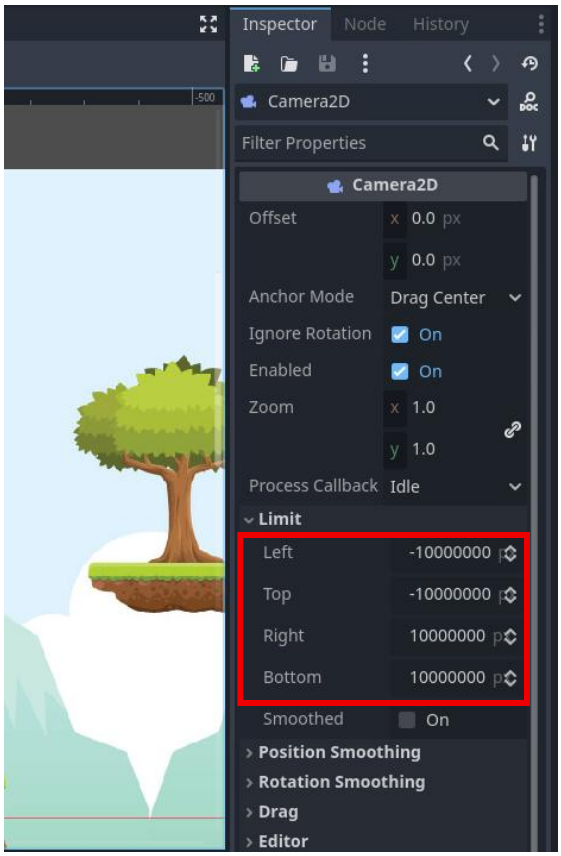
The Player is responsible for... • User input & movement • Increase ammo & update UI when picking up Collectables • Throwing Projectiles • Resetting
The Enemies are responsible for... • Their own movement • Colliding with Player and Scene
The Goal is responsible for... • Checking if all Enemies are cleared • Going to the next scene
The Collectables are responsible for... • Telling the Player when picked up • Destroying themselves
The Projectiles are responsible for... • Their own movement • Colliding with Enemies
The Scene is responsible for... • Providing collision for other components

3 The most useful component to start with is the Scene, as it allows the developer to navigate the environment while playtesting.

Before adding scene collision, the Player should always be on-screen.

Add **Camera2D** as a child node to **Player** and set the pixel values under the **Limit** drop-down menu in the Inspector.



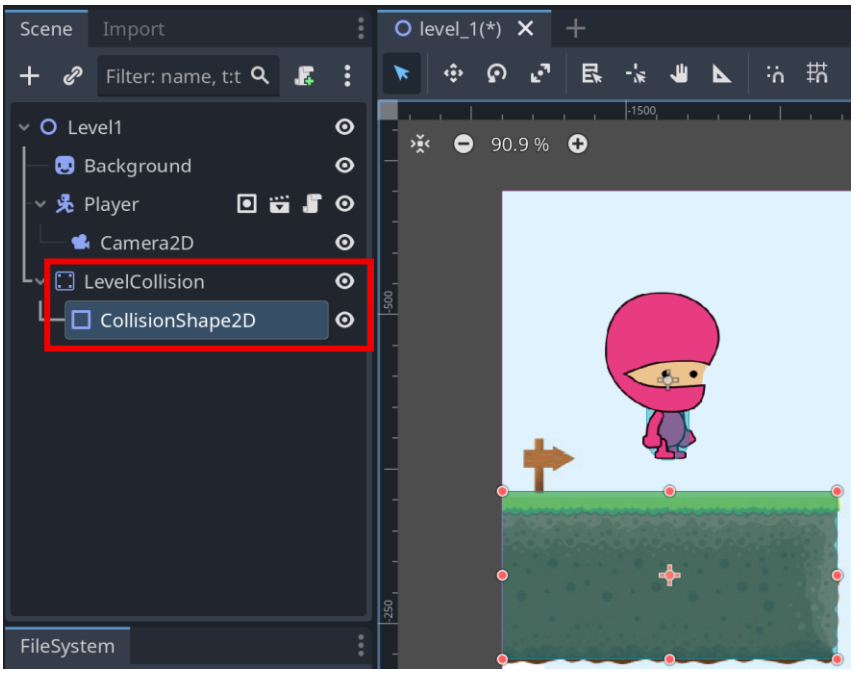


Pro Tip:

Check the pixel resolution of the background image by clicking it in the **Inspector** or hovering over it in **FileSystem**. This is the default size that it will be rendered in as a Sprite2D. Since the Background sprite is centered at the origin it will extend to the Left and Right by half of its width (+/- 1600px) and to the Top and Bottom by half of its height (+/- 600px).

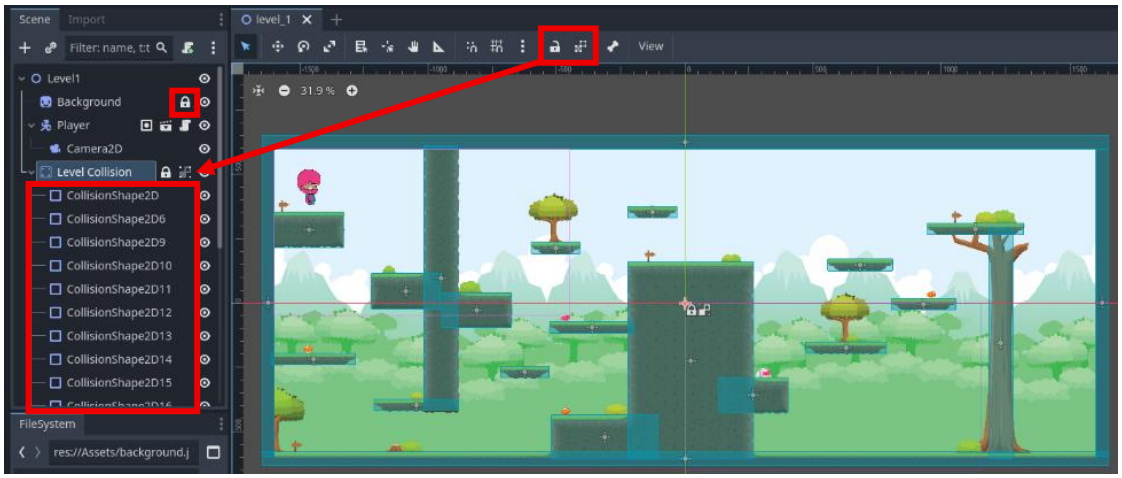
4 Add the scene collision! Add a **StaticBody2D** as a child to the root node and rename it **LevelCollision**.

Add a **CollisionShape2D** as a child node, with its **Shape** set to a **RectangleShape2D** as the platform under the Player.



5 Duplicate (**CTRL+D**) as many **CollisionShape2D's** as needed for the level as children to **LevelCollision**. Ensure there are collision shapes for the outside boundaries of the level!

To avoid accidental movement of any collision shapes in **LevelCollision**, use the toolbar to group the selected nodes of **LevelCollision** and lock them. Similarly, lock the **Background** node.



Reminder:

Don't forget to use "Make Unique" by right clicking the RectangleShape2Ds in the Inspector!

6 Playtest the project! Refer to previous steps if the camera is not properly staying inside of the scene.

Pause for **Sensei Stop #1**!

Check in with a Code Sensei before moving on. Confirm that Requirements #1-2 have been completed.

Reminder: Save your work!

Requirement #3 (Instructions): Player Gravity

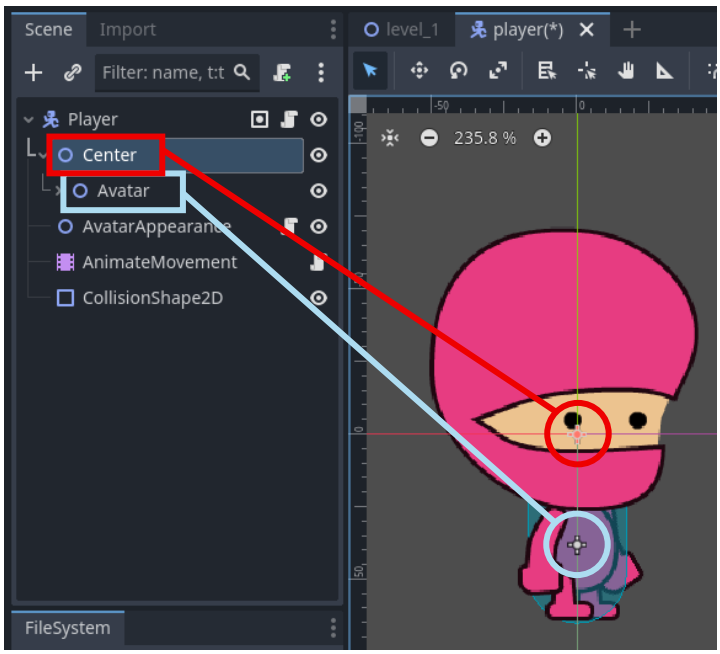
7 Oh no, the Player cannot jump onto platforms to navigate to the end of the level! The Player needs to reverse its gravity when pressing jump to satisfy the **User input & movement** responsibility.

Since the Scene's collision has been implemented, it can be marked off the responsibility planning chart.

The Player is responsible for... • User input & movement • Increase ammo & update UI when picking up Collectables • Throwing Projectiles • Resetting
The Enemies are responsible for... • Their own movement • Colliding with Player and Scene
The Goal is responsible for... • Checking if all Enemies are cleared • Going to the next scene
The Collectables are responsible for... • Telling the Player when picked up • Destroying themselves
The Projectiles are responsible for... • Their own movement • Colliding with Enemies
~~**The Scene is responsible for...**~~ • ~~Providing collision for other components~~

8 Open **player.tscn**. Notice how all the visuals of the player are now inside of another node called “Center”.

This helps flip the Player more accurately than using the Avatar node because the Player’s CollisionShape2D is centered on the origin.



9 Open **player.gd** and navigate to **TODO 1**.

Drag the **Center** node from Scene into the code editor under TODO 1, then hold CTRL and release the drag to create the `@onready` variable.

```
9  # --------------------
10 # TODO 1
11 # Get player center pivot
12 # --------------------
13 @onready var center: Node2D = $Center
14
```

10 Scroll down past the provided starter code to find **TODO 2** inside of `_physics_process()`.

Use `Input.is_action_just_pressed()` to check if the `ui_accept` action was just pressed that frame. Then, use `*=` to multiply both the `gravity_force` variable and `center.scale.y` by `-1` to flip the Player upside-down.

```
47    # --------------------
48    # TODO 2
49    # User input to flip player gravity
50    # --------------------
51    # if ???
52        # gravity_force ???
53        # center.scale.y ???
54
```

11 Check the code! Update the script as needed.

```
47  # --------------------
48  # TODO 2
49  # User input to flip player gravity
50  # --------------------
51  if Input.is_action_just_pressed("ui_accept"):
52      gravity_force *= -1
53      center.scale.y *= -1
54
```

12 Playtest the project! Can the Player navigate the entire scene using space bar to reverse the gravity?

Requirement #4 (Instructions): Crushers

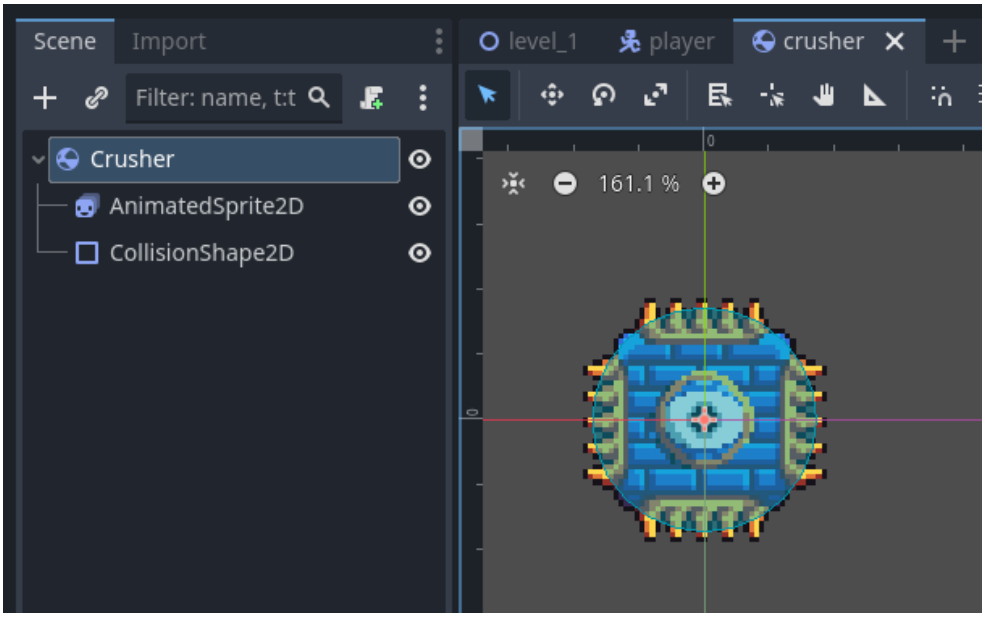
13 Great work! Now that the Player can navigate the scene, it's time to add some Enemies.

The Enemies are responsible for their own movement and running code when colliding with the Player and the Scene.

The Player is responsible for... • ~~User input & movement~~ • Increase ammo & update UI when picking up Collectables • Throwing Projectiles • Resetting
The Enemies are responsible for... • Their own movement • Colliding with Player and Scene
The Goal is responsible for... • Checking if all Enemies are cleared • Going to the next scene
The Collectables are responsible for... • Telling the Player when picked up • Destroying themselves
The Projectiles are responsible for... • Their own movement • Colliding with Enemies
~~The Scene is responsible for...~~ • ~~Providing collision for other components~~

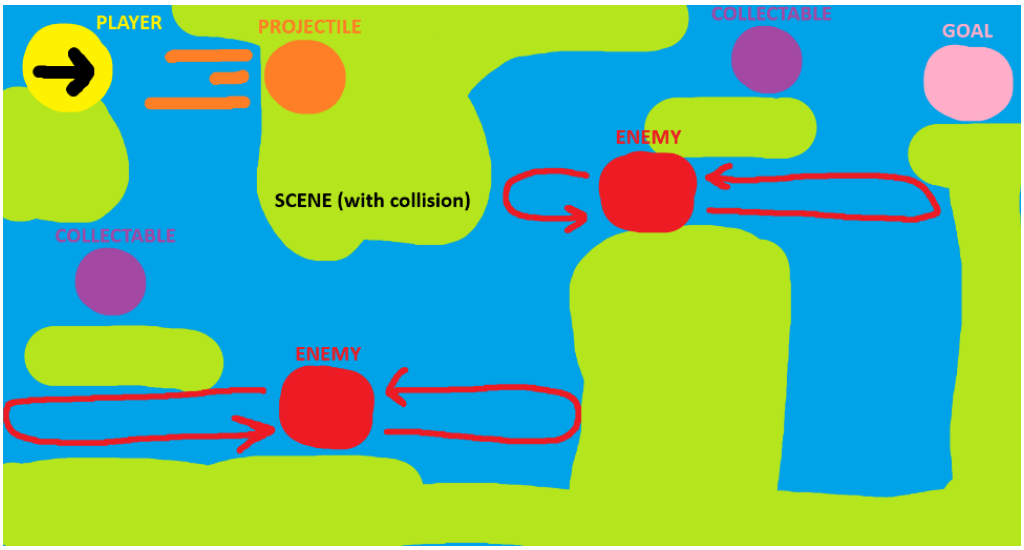
14 In **FileSystem**, navigate to **Scenes > Objects** and open **crusher.tscn**.

It is a simple scene with a **RigidBody2D** as the root node, an **AnimatedSprite2D** to run the blinking animation, and a **CollisionShape2D**.



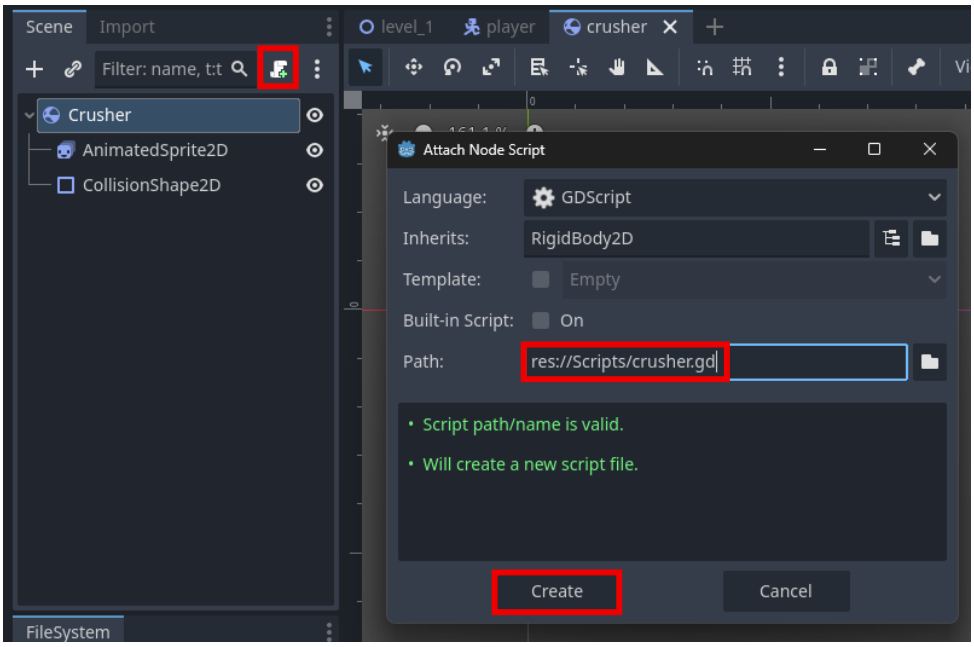
15 Recall the planning sketch from the start of the project. The Crushers will bounce off the scene's collision as automatic movement.

One way to code this is to set a starting speed and direction. Then, when hitting the scene's collision, reverse the direction.



16

Create a new **crusher.gd** script attached to the **RigidBody2D**.



Inside the script, create two `@export` variables: `speed` of type `float` with a default value of `100` and `direction` of type `Vector2` with a default value of `Vector2.RIGHT`.

```
1   extends RigidBody2D
2
3   # export speed ???
4   # export direction ???
5
```

17

Underneath, define the `_ready()` method, then inside, set the RigidBody2D's `linear_velocity` property to `direction * speed` when it enters the scene tree.

```
1   extends RigidBody2D
2
3   # export speed ???
4   # export direction ???
5
6   # _ready() ???
7       # linear_velocity ???
```

18

Check the code! Update the script as needed.

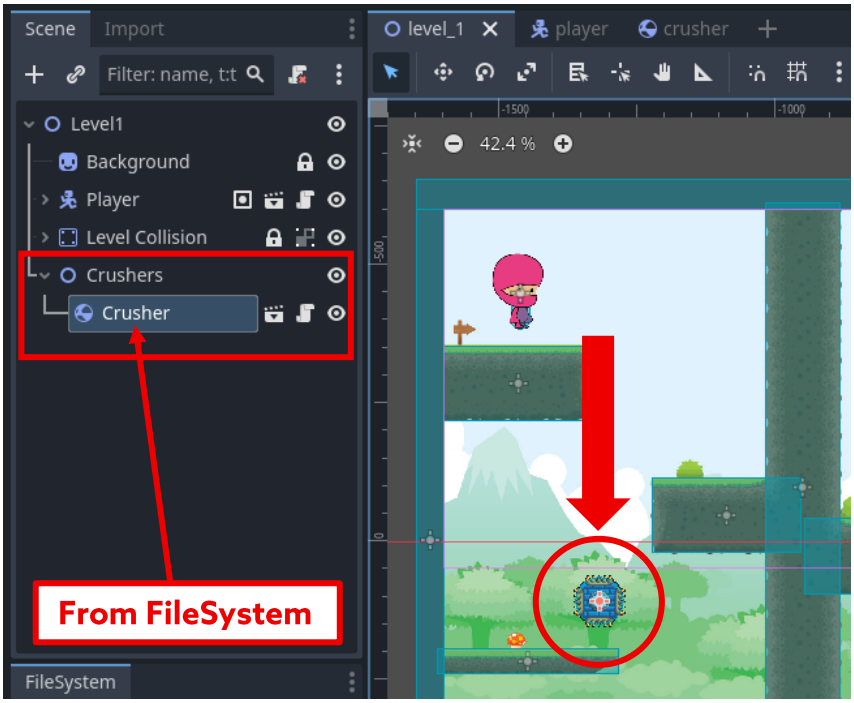
```
1   extends RigidBody2D
2
3   @export var speed: float = 100
4   @export var direction: Vector2 = Vector2.RIGHT
5
6   func _ready() -> void:
7       linear_velocity = direction * speed
8
```

19

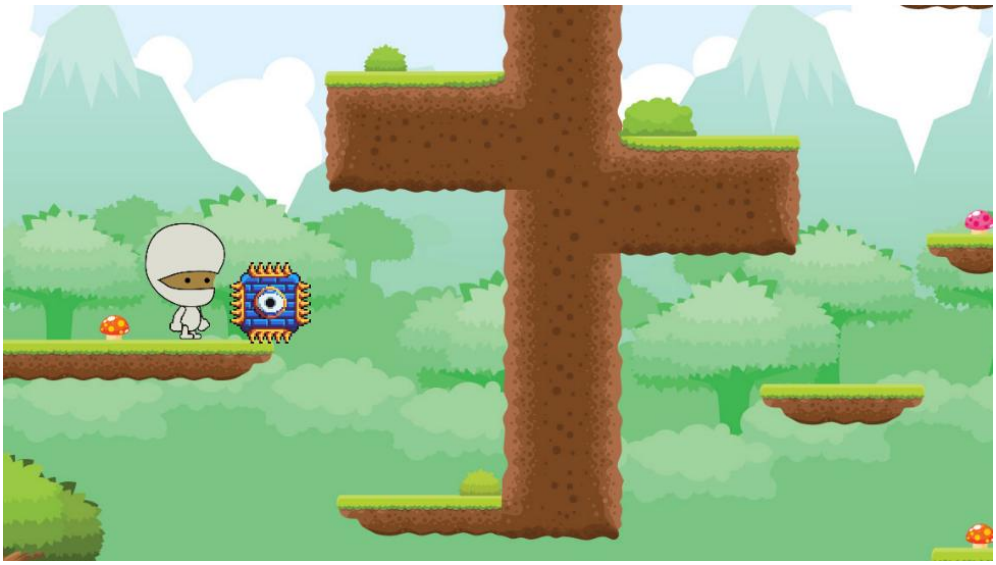
Return to the **level_1** scene.

Create a new empty **Node2D** as a child to the root node and rename it to **Crushers**.

Then, add a new crusher using **crusher.tscn** as a child to the **Crushers** node and place it in the air somewhere near the start of the level so it can be playtested.



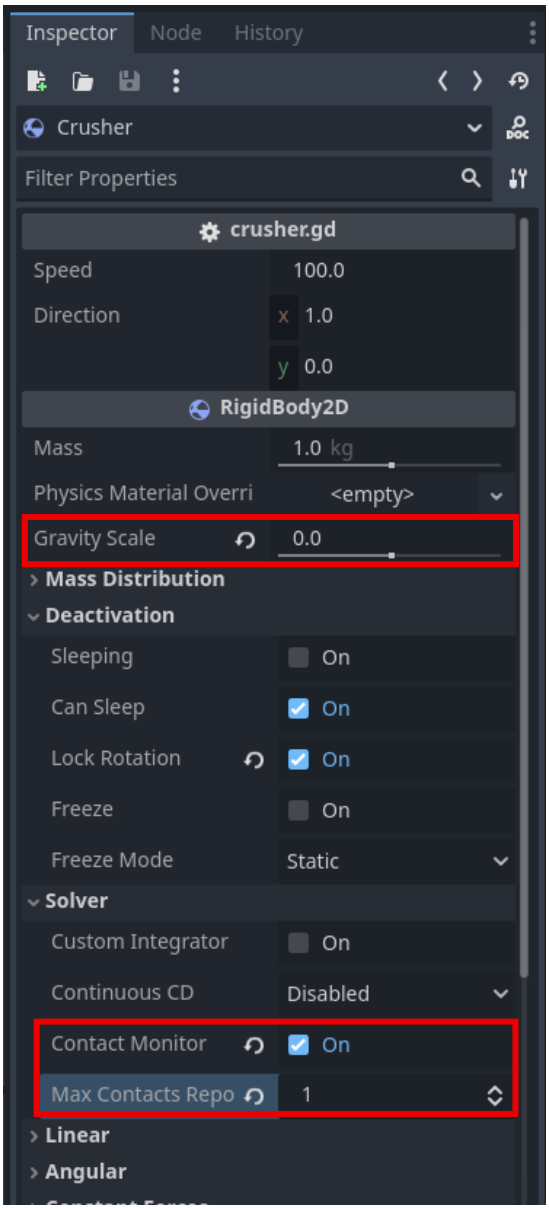
20 Playtest the project. Does the crusher move as expected?



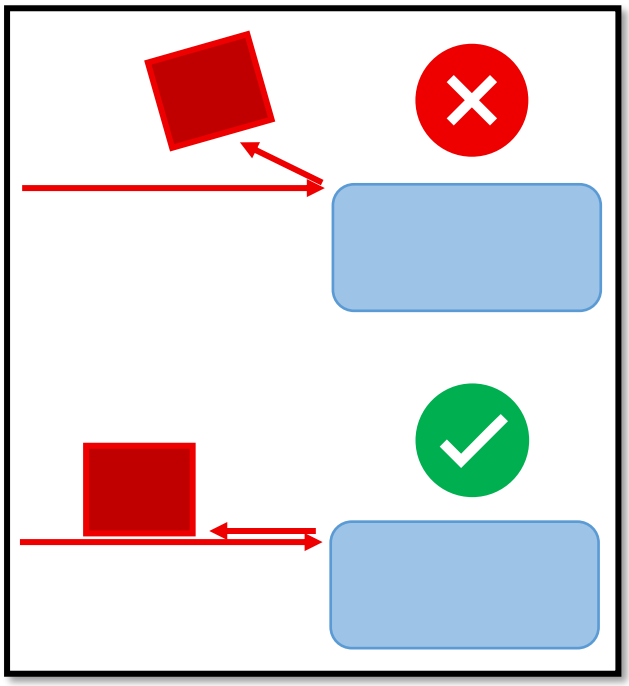
21 Oh no! The crusher drops to the ground and stops moving. The RigidBody2D needs to have zero gravity for this to work properly.

Return to the crusher scene and set Crusher's **Gravity Scale** property to **0** in the **Inspector**.

Open the **Solver** drop-down menu and enable **Contact Monitor** to set the **Max Contacts** to 1 so the Crusher will be able to interact with the Player and Scene.



22 Fixing the gravity alone is not enough. To ensure the Crusher always stays on-track and doesn't move up or down (for example, bouncing off a wall at an angle), the **y-velocity** can be overridden every physics step to 0.

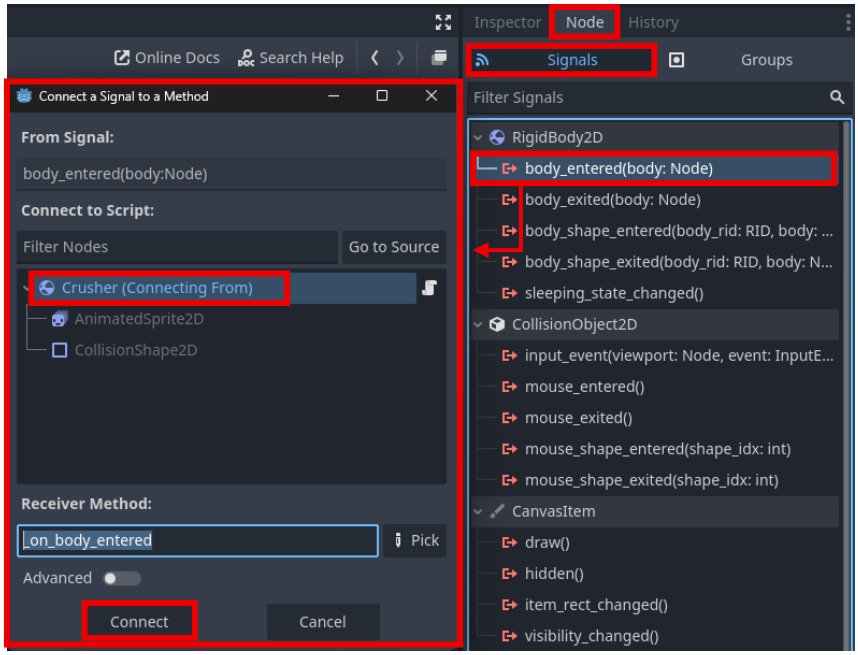


In **crusher.gd**, define the `_integrate_forces()` method to override the physics. Inside, use the `state` parameter to set the `y` component of the `linear_velocity` property to `0`.

```
9  # _integrate_forces() ???
10     # state.linear_velocity.y ???
```

23 In **Scene**, ensure that Crusher is selected. Then, toggle **Inspector** to **Node** and open the **Signals** section.

Connect the **body_entered()** signal to a new `_on_body_entered()` receiver method in Crusher's attached script.



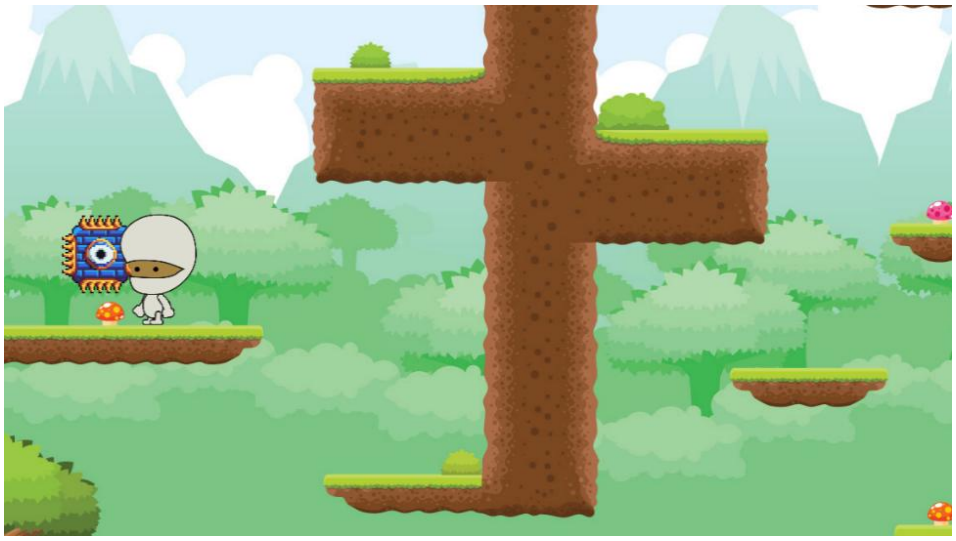
24 Inside of the new `_on_body_entered()` receiver method, reverse the `direction` by multiplying it by `-1` then set the `linear_velocity` to the updated `direction` multiplied by `speed`.

```
9  # _integrate_forces() ???
10     # state.linear_velocity.y ???
11
12 # _on_body_entered() ???
13     # direction ???
14     # linear_velocity ???
```

25 Check the code! Update the script as needed.

```
9  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
10     state.linear_velocity.y = 0
11
12 func _on_body_entered(body: Node) -> void:
13     direction *= -1
14     linear_velocity = direction * speed
15
```

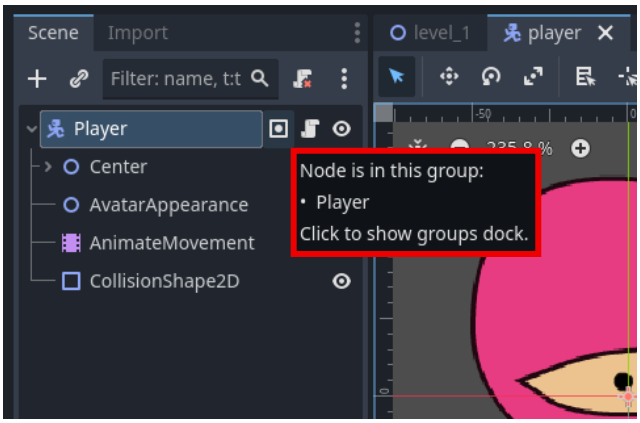
26 Playtest the project. Is the Crusher bouncing off walls properly? What happens when the Player interacts with the Crusher?



27

Uh oh, the crusher seems to bounce off the player, too!

Groups can be used to check for this in code. Notice that the Player already has a "Player" group attached to it from the Starter Pack.



Return to **crusher.gd**. In the `_on_body_entered()` method, add an `if`-statement that uses `is_in_group()` to check if the `body` is in the `"Player"` group. Then, call the body's `reset()` method. Else, use the logic to bounce off the walls.

```
12 func _on_body_entered(body: Node) -> void:
13     # if ???
14         # body ???
15     # else ???
16         direction *= -1
17         linear_velocity = direction * speed
18
```

28

Open **player.gd** and navigate to **TODO 3**.

Define a `reset()` method that has no parameters and returns `void`. Inside, use `get_tree().call_deferred()` to call the `reload_current_scene()` method at the end of the current frame. It's best practice to use `call_deferred()` to ensure that scene transitions occur safely.

```
61  # --------------------
62  # TODO 3
63  # Player hit function (called from the enemy)
64  # --------------------
65  # reset() ???
66      # get_tree ???
67
```

Pro Tip:

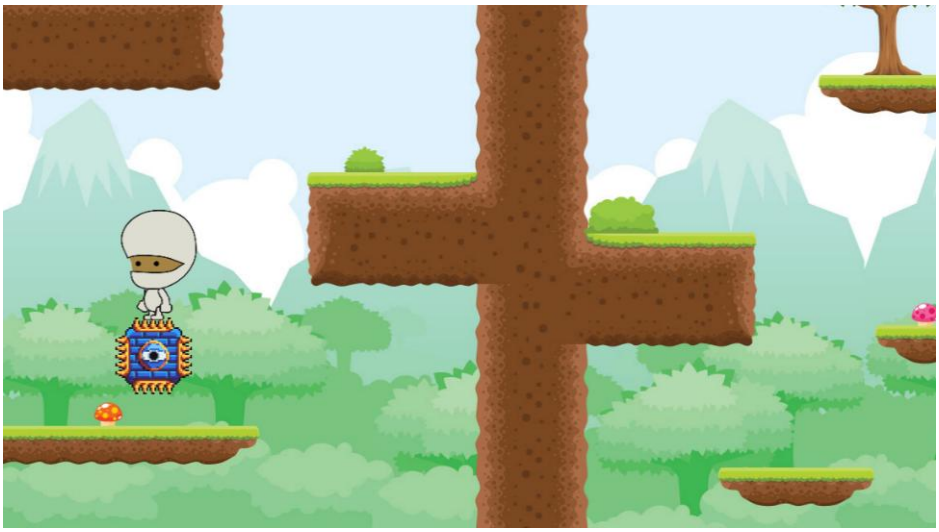
The method name inside of `call_deferred()` must be a string with no parentheses at the end. For example, when calling the `foo()` method at the end of the current frame, use `call_deferred("foo")`.

29 Check the code! Update the scripts as needed.

```
12  func _on_body_entered(body: Node) -> void:
13      if body.is_in_group("Player"):
14          body.reset()
15      else:
16          direction *= -1
17          linear_velocity = direction * speed
18
```

```
61  # --------------------
62  # TODO 3
63  # Player hit function (called from the enemy)
64  # --------------------
65  func reset() -> void:
66      get_tree().call_deferred("reload_current_scene")
67
```

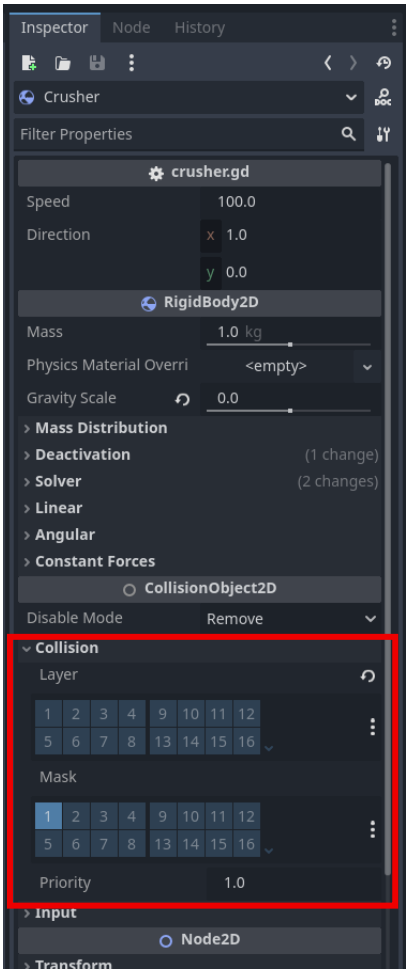
30 Playtest the project. Does the scene reset when the enemy collides with the player, as expected?

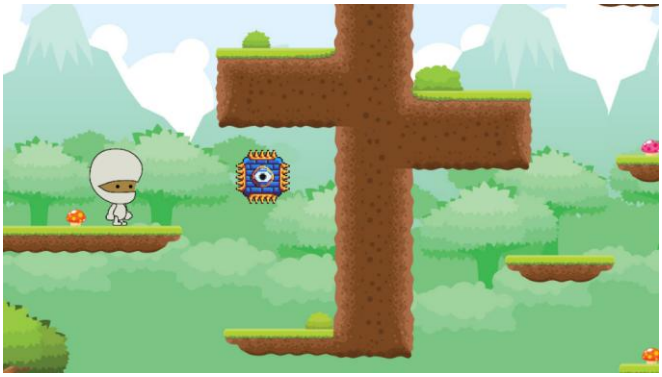


31 Hmm, sometimes the player can stand on top of the crusher or even walk against it! To fix this issue, remove the enemy from the collision layers that the player scans for.

Return to the crusher scene and open the crusher's Inspector. In this case, the only collision layer it exists on is Layer 1, so click on the **Collision** drop-down and click **1** under **Layer** to disable it.

Playtest the project again. Can the player still stand on top of or walk against the crusher?





Pause for **Sensei Stop #2**!

Check in with a Code Sensei before moving on. Confirm that Requirements #3-4 have been completed.

Reminder: Save your work!

Requirement #5 (Instructions): Collectables

32

Amazing job! Because the goal requires that all Enemies are cleared, the player must pick up collectables to throw at the Enemies.

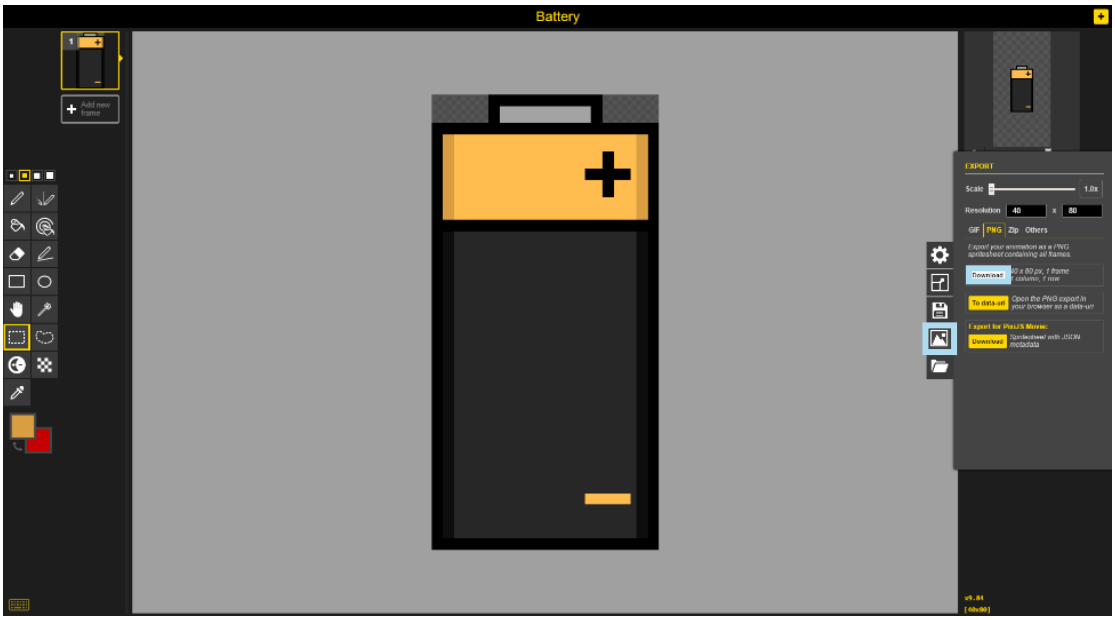
The Collectables are responsible for telling the Player when they are picked up and destroying themselves, while the Player must increase the ammo and update the UI to reflect the change.

The Player is responsible for... • ~~User input & movement~~ • Increase ammo & update UI when picking up Collectables • Throwing Projectiles • ~~Resetting~~
~~**The Enemies are responsible for...**~~ • ~~Their own movement~~ • ~~Colliding with Player and Scene~~
The Goal is responsible for... • Checking if all Enemies are cleared • Going to the next scene
The Collectables are responsible for... • Telling the Player when picked up • Destroying themselves
The Projectiles are responsible for... • Their own movement • Colliding with Enemies
~~**The Scene is responsible for...**~~ • ~~Providing collision for other components~~

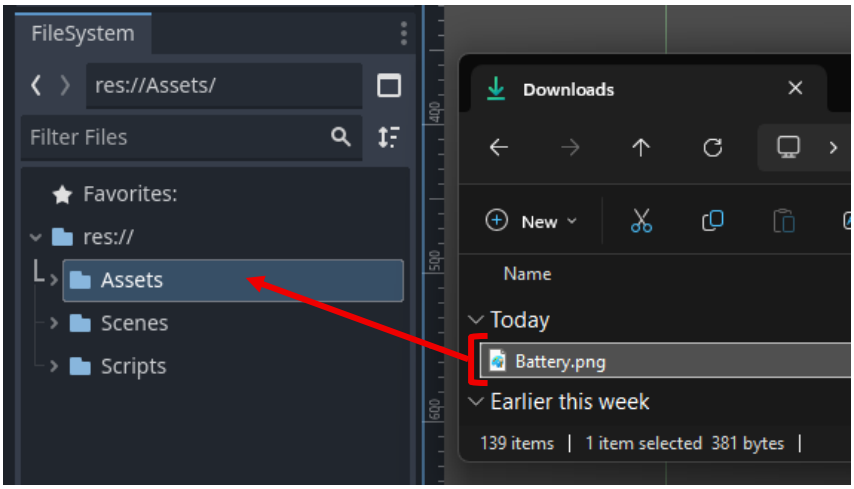
33 Draw the collectable sprite using **Piskel**! If you don't have access to Piskel, use the assets provided in the Starter Pack.

Refer to the **PB Activity 00 Ninja Guide** for help with navigating Piskel.

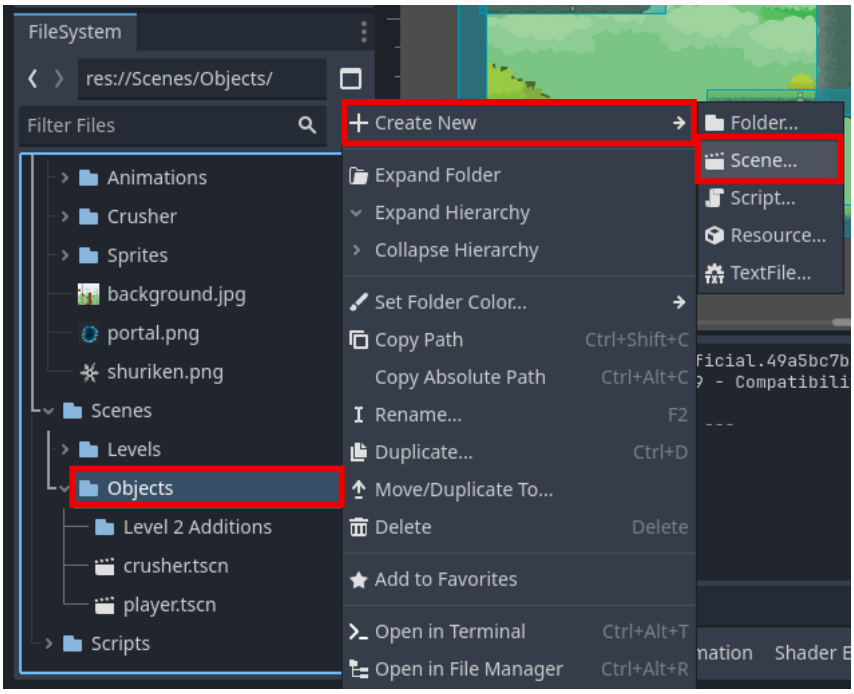
Once you're done with the asset, go to **Export**, ensure **PNG** is selected, then click **Download**.



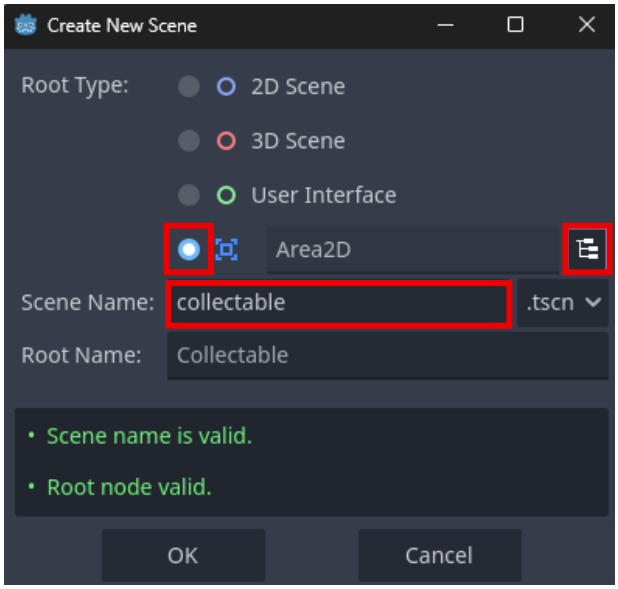
34 Navigate to the Downloads folder on your computer. Select the two newly created **.png** files and drag them into the **Assets** folder.



35 Create a new scene inside the **Scenes > Objects** folder to represent a collectable.



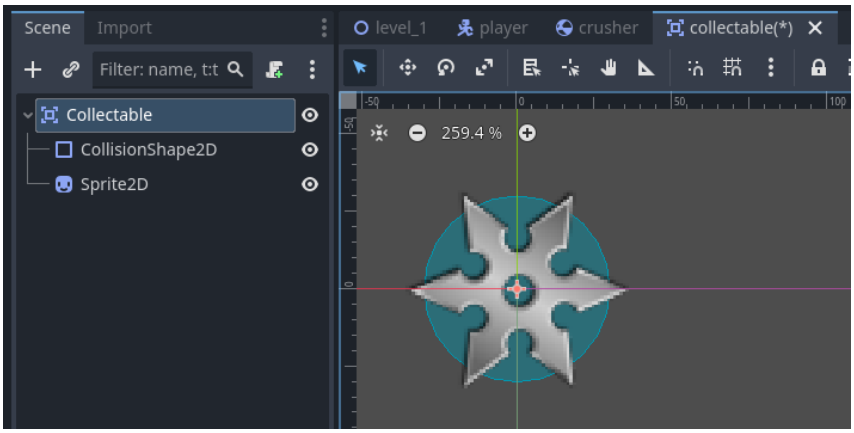
Set the **Root Type** to **Node** and set the type to **Area2D**. Name the scene **collectable** and click **OK**.



36

Set up the nodes in the Collectable scene.

Add a **CollisionShape2D** as a child node to the root with a **CircleShape2D** selected. Set its radius to 30px. Then, add a **Sprite2D** and set its **texture** to the custom asset or the provided **shuriken.png** in the **Assets** folder. Scale the Sprite2D to roughly fit the **CollisionShape2D**.

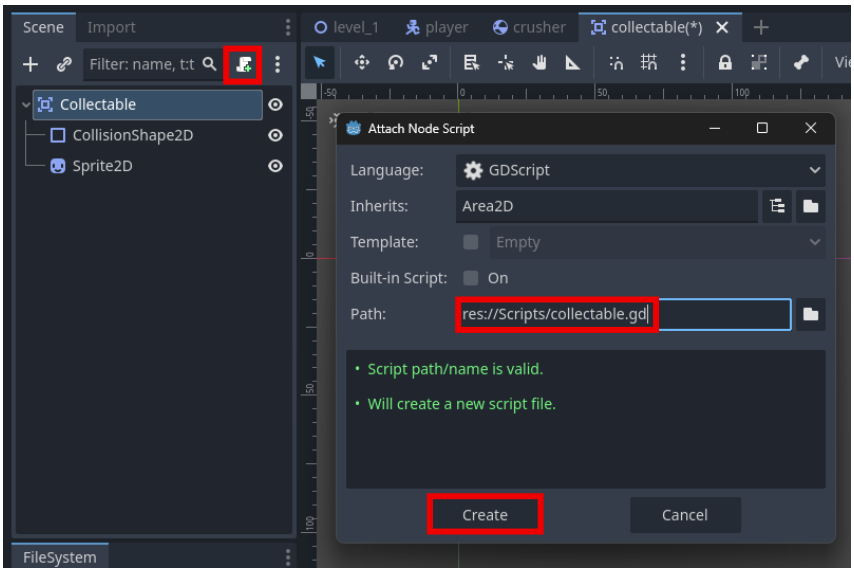


Reminder:

Click the CircleShape2D in the Inspector to change its radius.

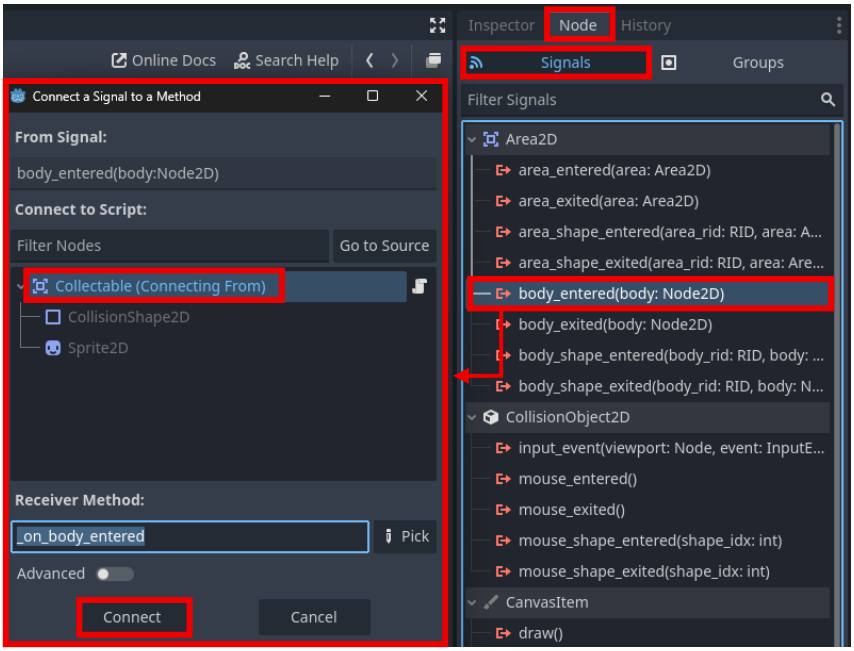
37

Create a new **collectable.gd** script attached to the **Area2D**.



38 In **Scene**, ensure that Collectable is selected. Then, toggle **Inspector** to **Node** and open the **Signals** section.

Connect the **body_entered()** signal to a new `_on_body_entered()` receiver method in Collectable's attached script.



39 The Collectables have the responsibility of telling the Player when they are picked up.

Inside the `_on_body_entered()` receiver method, write an `if`-statement to check if the `body` parameter is in group `"Player"`. Then, call the body's `pickup_collectable()` method and then `queue_free()`.

```
4  func _on_body_entered(body: Node2D) -> void:
5      # if ???
6          # body ???
7          # queue_free ???
```

40 Check the code! Update the script as needed.

```
4  func _on_body_entered(body: Node2D) -> void:
5      if body.is_in_group("Player"):
6          body.pickup_collectable()
7          queue_free()
```

41 Write the `pickup_collectable()` method!

Open **player.gd** and navigate to **TODO 4**.

Create a `collectable_counter` `@export` variable of type `Label` and a `collectable_ammo` variable of type `int` with a default value of `0`.

```
15 # --------------------
16 # TODO 4
17 # Get collectable count UI and store collectable ammo
18 # --------------------
19 # collectable_counter ???
20 # collectable_ammo ???
21
```

42 Scroll down to find **TODO 5**.

Define a `pickup_collectable()` method that has no parameters and returns **void**. Inside, increase the `collectable_ammo` variable by `1` then update the `collectable_counter`'s `text` property to the `str` conversion of `collectable_ammo`.

```
69 # --------------------
70 # TODO 5
71 # Pick up a collectable (called from the collectable)
72 # --------------------
73 # pickup_collectable() ???
74     # collectable_ammo ???
75     # collectable_counter.text ???
76
```

43

Check the code! Update the script as needed.

```
15  # --------------------
16  # TODO 4
17  # Get collectable count UI and store collectable ammo
18  # --------------------
19  @export var collectable_counter: Label
20  var collectable_ammo: int = 0
21
```

```
69  # --------------------
70  # TODO 5
71  # Pick up a collectable (called from the collectable)
72  # --------------------
73  func pickup_collectable() -> void:
74      collectable_ammo += 1
75      collectable_counter.text = str(collectable_ammo)
76
```

44

Since the label has not been added yet, comment out the line of code that uses it in the **`pickup_collectable()`** method.

```
69  # --------------------
70  # TODO 5
71  # Pick up a collectable (called from the collectable)
72  # --------------------
73  func pickup_collectable() -> void:
74      collectable_ammo += 1
75      # collectable_counter.text = str(collectable_ammo)
76
```

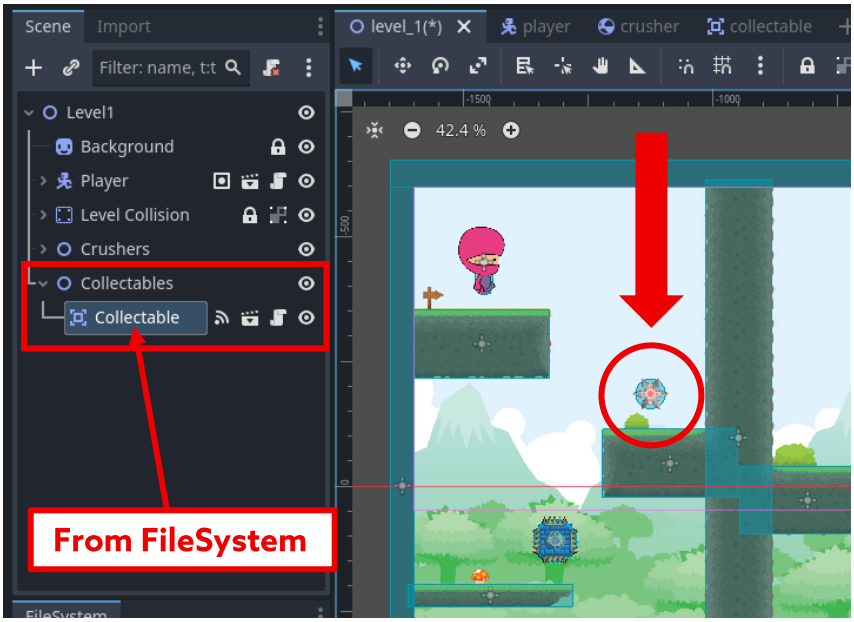
Pro Tip:

If the line of code isn't commented out, the game will crash during playtest because it will try to set the text property of a Label that has not been defined, so it will be type **Nil**.

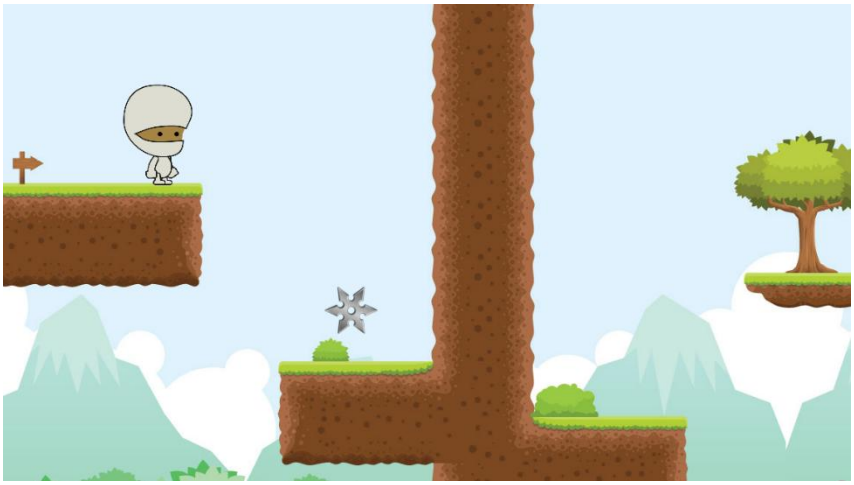
45 Return to the **level_1** scene.

Create a new empty **Node2D** as a child to the root node and rename it to **Collectables**.

Add a new collectable using **collectable.tscn** as a child to the **Collectables** node and place it somewhere near the start of the level so it can be playtested.



46 Playtest the project. Are the collectables picked up when the player overlaps with them?



Requirement #6 (Instructions): Projectiles

47

Now that the Collectables have been implemented, the Projectiles can be added.

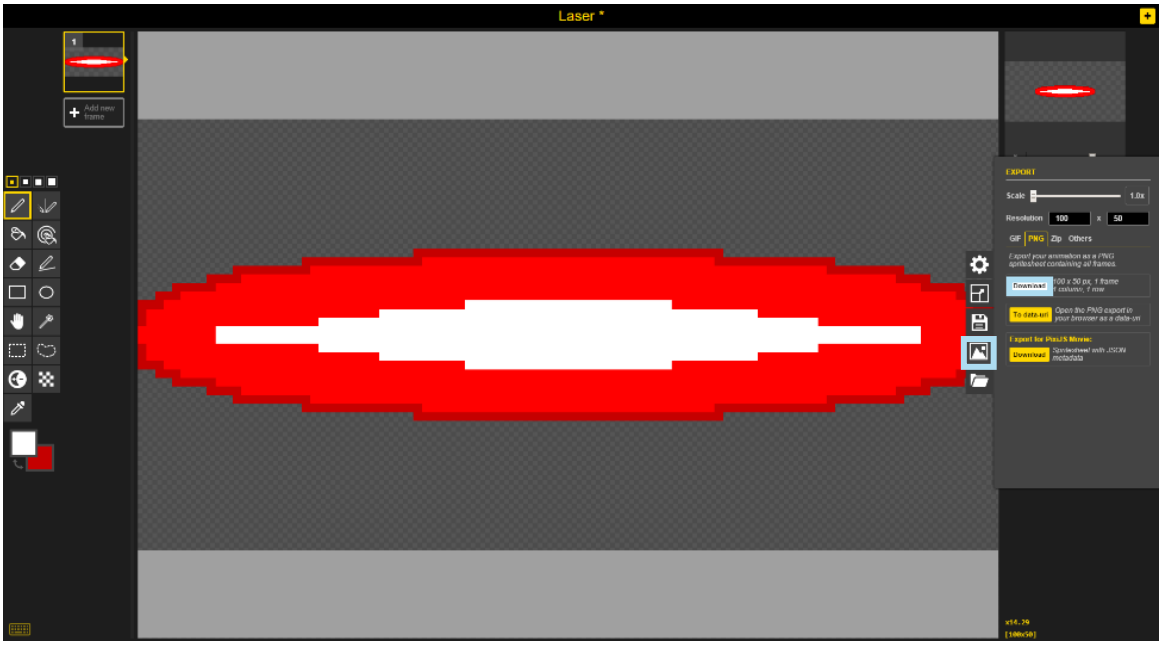
The Projectiles are responsible for their own movement and colliding with Enemies, while the Player must instantiate them and set off their movement.

The Player is responsible for... ~~• User input & movement~~ ~~• Increase ammo & update UI when picking up Collectables~~ • Throwing Projectiles ~~• Resetting~~
~~**The Enemies are responsible for...**~~ ~~• Their own movement~~ ~~• Colliding with Player and Scene~~
The Goal is responsible for... • Checking if all Enemies are cleared • Going to the next scene
~~**The Collectables are responsible for...**~~ ~~• Telling the Player when picked up~~ ~~• Destroying themselves~~
The Projectiles are responsible for... • Their own movement • Colliding with Enemies
~~**The Scene is responsible for...**~~ ~~• Providing collision for other components~~

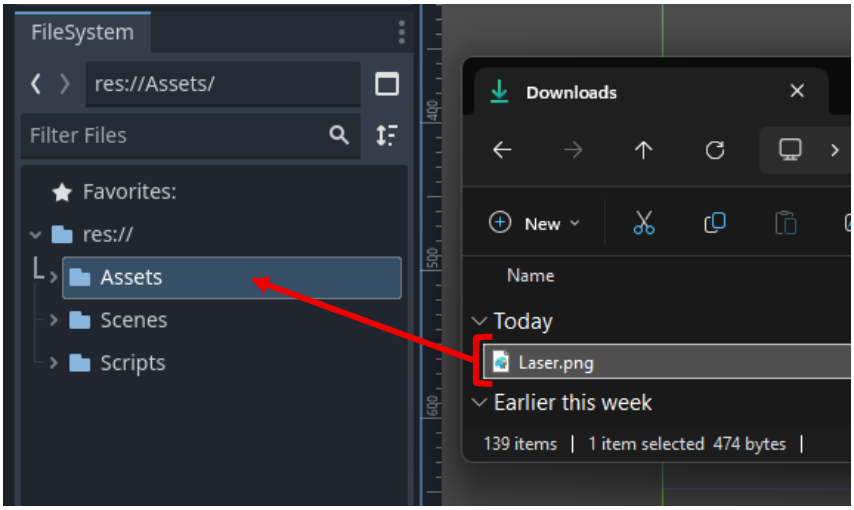
48 Draw the projectile sprite using **Piskel**! Alternatively, reuse the sprite made for the collectable.

Refer to the **PB Activity 00 Ninja Guide** for help with navigating Piskel.

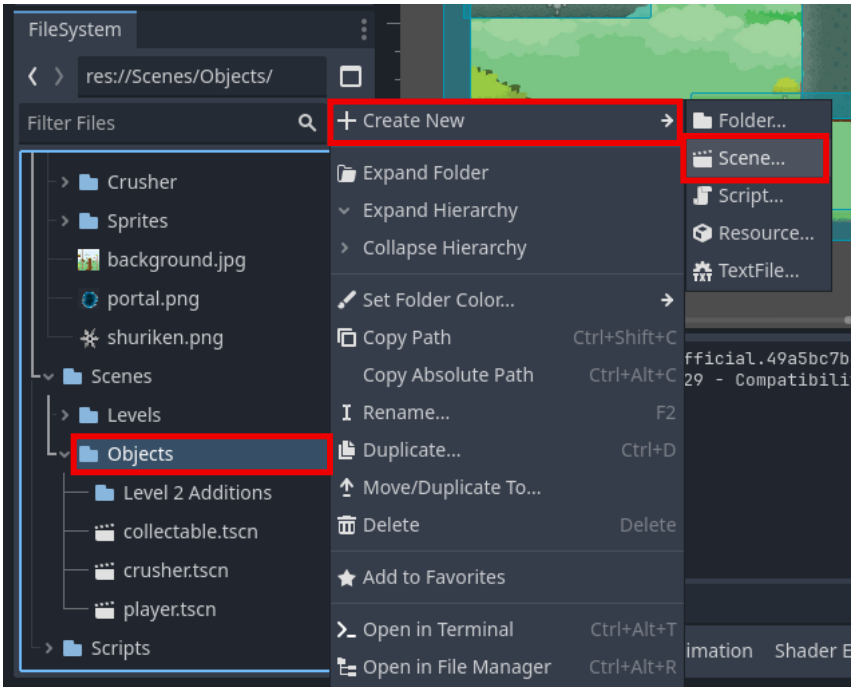
Once you're done with the asset, go to **Export**, ensure **PNG** is selected, then click **Download**.



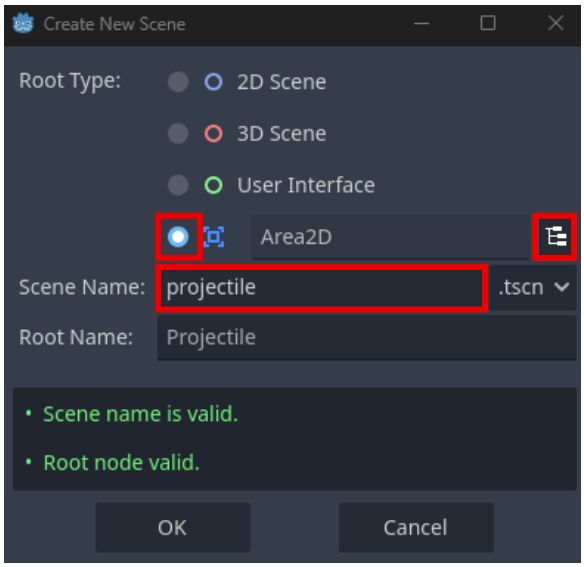
49 Navigate to the Downloads folder on your computer. Select the two newly created **.png** files and drag them into the **Assets** folder.



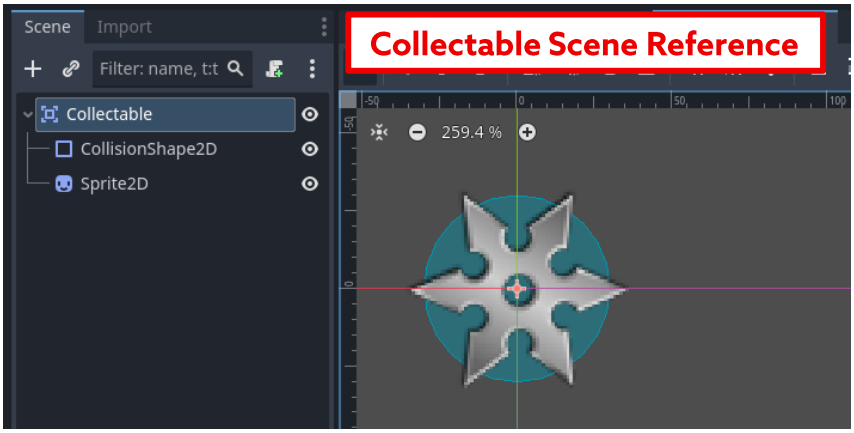
50 Create a new scene inside the **Scenes > Objects** folder to represent a projectile.



Set the **Root Type** to **Node** and set the type to **Area2D**. Then, set the **Scene Name** to **projectile** and click **OK**.



51 Set up the nodes in the Projectile scene to be the same as the Collectable scene. Use the new custom projectile asset as the texture.

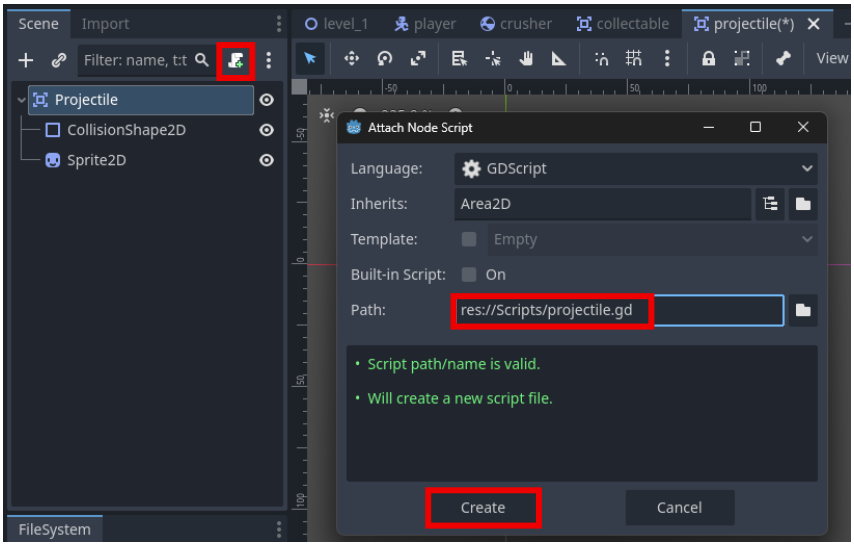


Reminder:

Add a **CollisionShape2D** as a child node to the root with a **CircleShape2D** selected. Set its radius to 30px. Then, add a **Sprite2D** and set its **texture** to the custom asset or the provided **shuriken.png** in the **Assets** folder. Then, scale it to roughly fit the **CollisionShape2D**.

52

Create a new **projectile.gd** script attached to the **Area2D**.



Inside the script, create a `speed` `@export` variable of type `float` with a default value of `300` and a `velocity` variable of type `Vector2`.

```
1   extends Area2D
2
3   # speed ???
4   # velocity ???
5
```

53

Next, define a new `set_direction()` method so the player can kick off the projectile's movement once it is instantiated. It will have a `direction` parameter of type `int` and will return `void`.

Inside of `set_direction()`, set `velocity` to `Vector2(direction, 0) * speed`.

```
1   extends Area2D
2
3   # speed ???
4   # velocity ???
5
6   # set_direction() ???
7       # velocity ???
8
```

Pro Tip:

`Vector2(direction, 0)` creates a new Vector2 with the x-component equal to **direction** and the y-component equal to **0**.

54 Below, define the `_physics_process()` method.

Inside, increase the Area2D's `position` by `velocity * delta` every frame.

```
1   extends Area2D
2
3   # speed ???
4   # velocity ???
5
6   # set_direction() ???
7       # velocity ???
8
9   # _physics_process() ???
10      # position ???
```

55 Check the code! Update the script as needed.

```
1   extends Area2D
2
3   @export var speed: float = 300
4   var velocity: Vector2
5
6   func set_direction(direction: int) -> void:
7       velocity = Vector2(direction, 0) * speed
8
9   func _physics_process(delta: float) -> void:
10      position += velocity * delta
```

56 Instantiate the projectiles!

Open **player.gd** and navigate to **TODO 6**.

The script should know what the projectile scene is so it can be instantiated. Drag **projectile.tscn** from **FileSystem** into the code editor under TODO 6, then hold **CTRL** and release the drag to create the `PROJECTILE` constant.

```
22  # --------------------
23  # TODO 6
24  # Load projectile scene
25  # --------------------
26  const PROJECTILE = preload("res://Scenes/Objects/projectile.tscn")
27
```

57 Remember the Instantiation Code Pattern from Silver Belt?

1. Instantiate to memory
2. Handle object-specific stuff
3. Add to the scene tree
4. Handle scene-specific stuff

This is what will be followed in this project, too.

Scroll down to find **TODO 7**. Underneath, define a new `throw_projectile()` method that has no parameters and returns `void`. Inside, create a `projectile_instance` variable and set it equal to `PROJECTILE.instantiate()`.

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  # throw_projectile() ???
82      # projectile_instance ???
83
```

58 The Projectiles need to know which way the Player is looking so their movement can be set off. The `sign()` helper function can be used here.

> `sign()`: returns the sign of the argument
>
> **Parameters:**
>
> 1. `argument` ***(Variant)***: an int, float, or Vector2/2i/3/3i/4/4i
>
> **Returns (*Variant*)**: 1 or -1 based on whether the argument is positive or negative. For vectors, returns another vector with 1 or -1 in each component

59 Create a `direction` variable and set it equal to `sign(scale.y)`.

Then, call the `set_direction()` method to set `projectile_instance`'s direction to `direction`.

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  # throw_projectile() ???
82      # projectile_instance ???
83      # direction ???
84      # projectile_instance ???
85
```

Pro Tip:

In a strange twist of fate, Godot's transform system can only horizontally flip a node by negating its **scale.y** and rotating it 180 degrees.

60 Check the code! Update the script as needed.

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  func throw_projectile() -> void:
82      var projectile_instance = PROJECTILE.instantiate()
83      var direction = sign(scale.y)
84      projectile_instance.set_direction(direction)
85
```

61 The object-specific stuff has been handled. Now it's time to add the projectile to the scene tree!

From **`get_tree().current_scene`**, use **`add_child()`** and pass **`projectile_instance`** as the parameter.

Now to handle the scene-specific stuff. Start by creating an **`offset`** variable and set it equal to **`direction * 64`**. Then, set the **`global_position`** property of **`projectile_instance`** to this node's (the player's) **`global_position + Vector2(offset, 0)`**.

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  func throw_projectile() -> void:
82      var projectile_instance = PROJECTILE.instantiate()
83      var direction = sign(scale.y)
84      projectile_instance.set_direction(direction)
85
86      # get_tree() ???
87
88      # offset ???
89      # projectile_instance ???
90
```

62 Decrease the `collectable_ammo` variable by `1` and update the `text` property of `collectable_counter` to the `str` conversion of `collectable_ammo`.

Keep the line of code that updates the text commented out for now; later when the UI is implemented and connected to the Player this will be uncommented

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  func throw_projectile() -> void:
82      var projectile_instance = PROJECTILE.instantiate()
83      var direction = sign(scale.y)
84      projectile_instance.set_direction(direction)
85
86      # get_tree() ???
87
88      # offset ???
89      # projectile_instance ???
90      # collectable_ammo ???
91      # collectable_counter ???
92
```

Pro Tip:

Copy the previous code under TODO 5 to update the **collectable_counter**'s text.

63

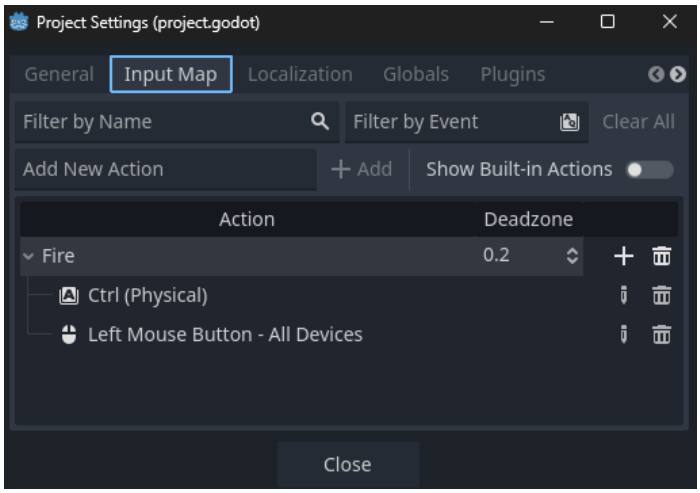
Check the code! Update the script as needed.

```
77  # --------------------
78  # TODO 7
79  # Throw the projectile
80  # --------------------
81  func throw_projectile() -> void:
82      var projectile_instance = PROJECTILE.instantiate()
83      var direction = sign(scale.y)
84      projectile_instance.set_direction(direction)
85
86      get_tree().current_scene.add_child(projectile_instance)
87
88      var offset = direction * 64
89      projectile_instance.global_position = global_position + Vector2(offset, 0)
90      collectable_ammo -= 1
91      # collectable_counter.text = str(collectable_ammo)
92
```

Keep commented for now!

64

Navigate to **Project Settings > Input Map** and notice that the Fire action has already been set up to enable the player to throw projectiles.



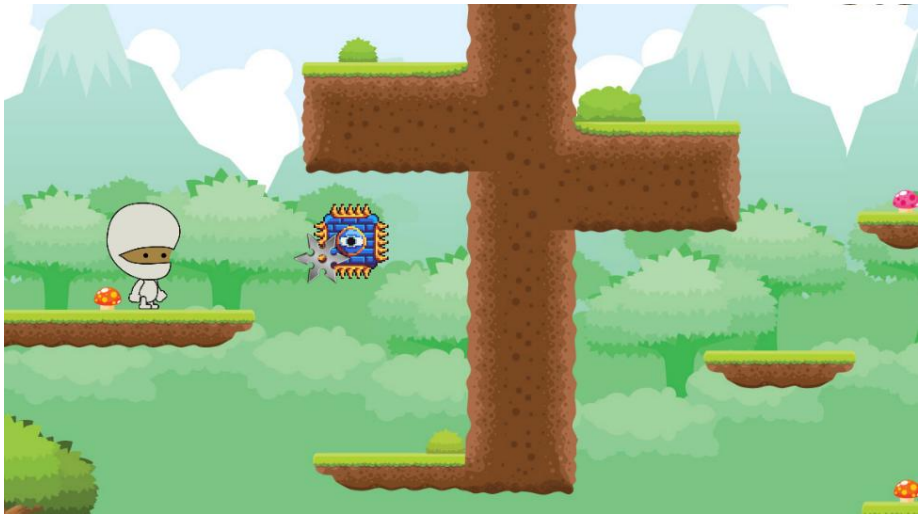
Navigate to **TODO 8**. Write an `if`-statement to check if the `"Fire"` action was just pressed. Inside, write a nested `if`-statement that checks if `collectable_ammo` is greater than `0`. Then, call `throw_projectile()`.

```
56      # --------------------
57      # TODO 8
58      # User input for throwing projectile
59      # --------------------
60      # if ???
61          # if ???
62              # throw_projectile()
63
```

65 Check the code! Update the script as needed.

```
56  # --------------------
57  # TODO 8
58  # User input for throwing projectile
59  # --------------------
60  if Input.is_action_just_pressed("Fire"):
61      if collectable_ammo > 0:
62          throw_projectile()
63
```

66 Playtest the project. Can the player throw projectiles using CTRL or Left-Click? Can the Player only throw as many projectiles as the number of collectables they have picked up?

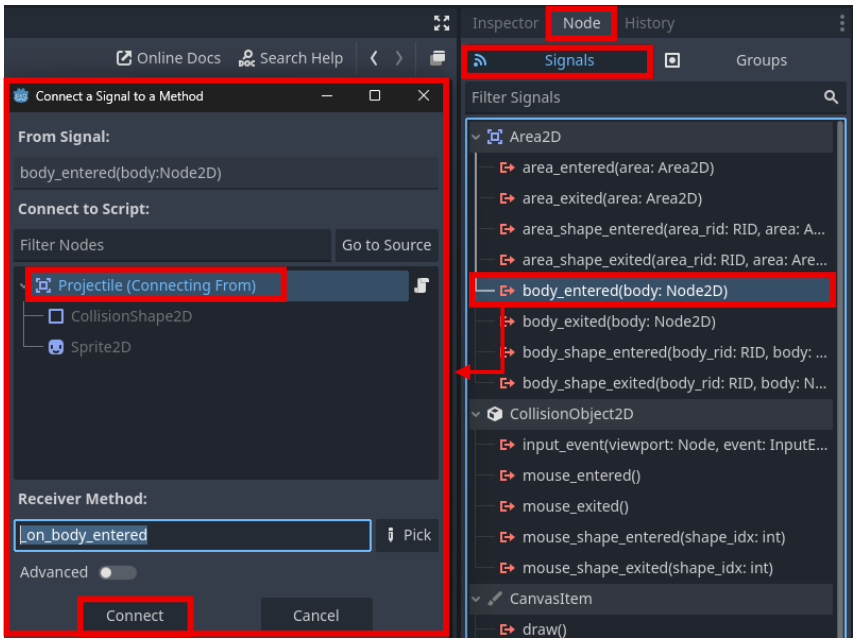


67

Great job! However, the Projectiles don't interact with Enemies yet.

In the **Projectile** scene, select the root node. Toggle **Inspector** to **Node** then open the **Signals** section.

Connect the **body_entered()** signal to a new `_on_body_entered()` receiver method in Projectile's attached script.



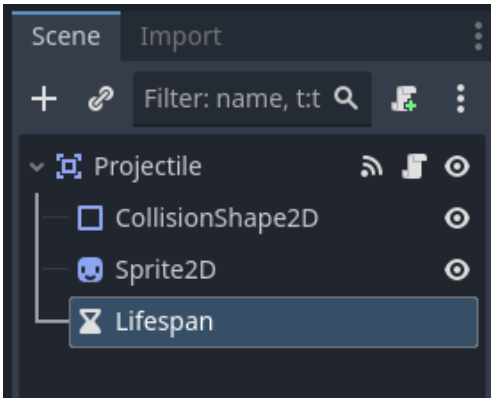
68

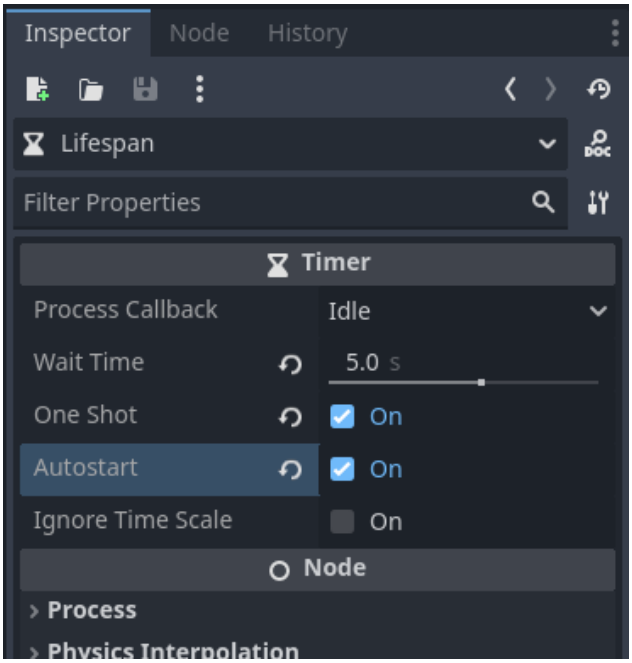
In the `_on_body_entered()` receiver method, write some code to call `queue_free()` on the `body` that the projectile entered and `queue_free()` on the `projectile`.

```
12 func _on_body_entered(body: Node2D) -> void:
13     body.queue_free()
14     queue_free()
15
```

69 It's also important for the projectiles to disappear after a time to save on memory.

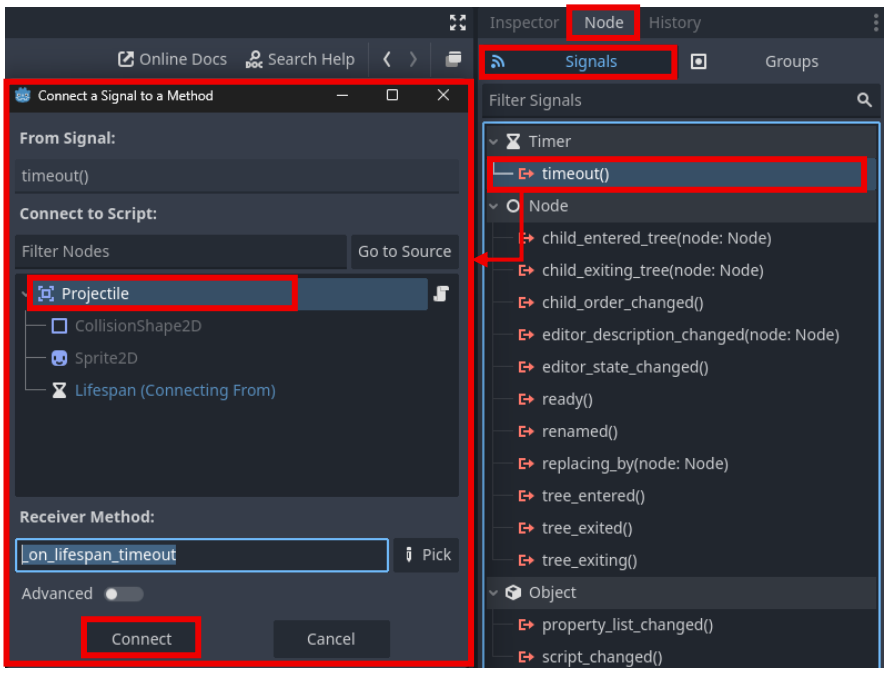
In **Scene**, add a **Timer** as a child node to the root node, rename it to **Lifespan**, and set its values in the **Inspector** as shown in the image.





70 Toggle **Inspector** to **Node** then open the **Signals** section.

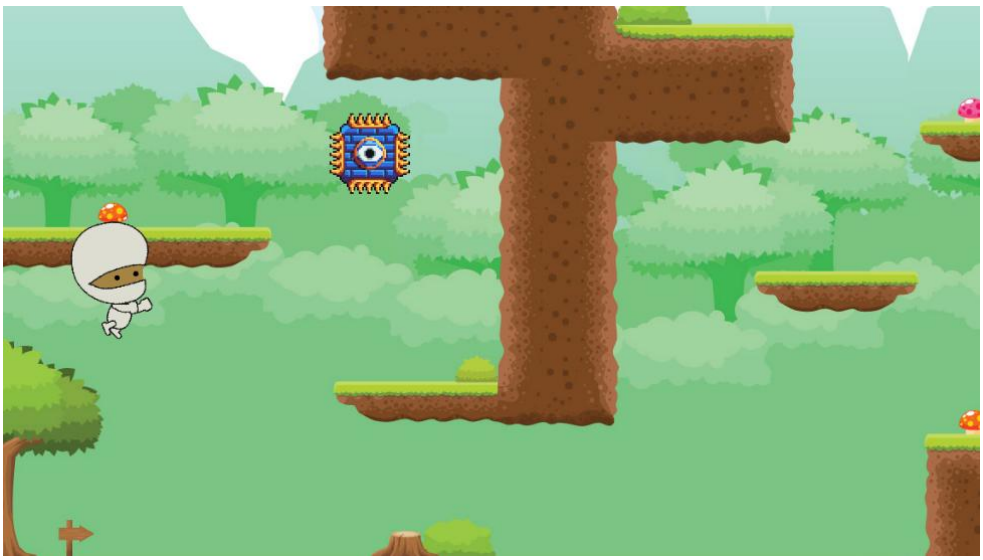
Connect the **timeout()** signal to a new `_on_lifespan_timeout()` receiver method in Projectile's attached script.



71 In the `_on_lifespan_timeout()` receiver method, call `queue_free()` on the projectile.

```
12 func _on_body_entered(body: Node2D) -> void:
13     body.queue_free()
14     queue_free()
15
16 func _on_lifespan_timeout() -> void:
17     queue_free()
18
```

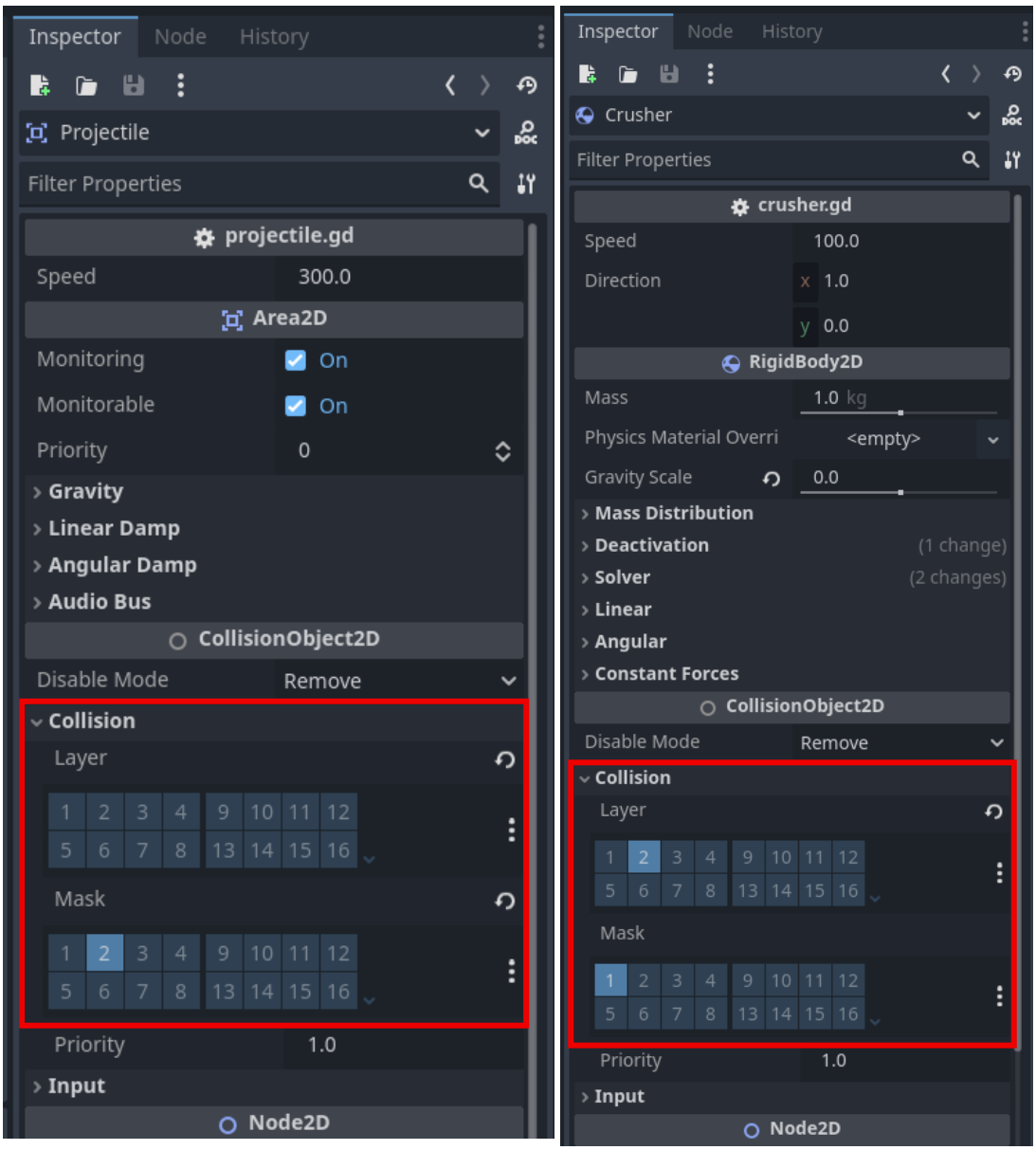
72 Playtest the project. What happens when the projectile overlaps with something?



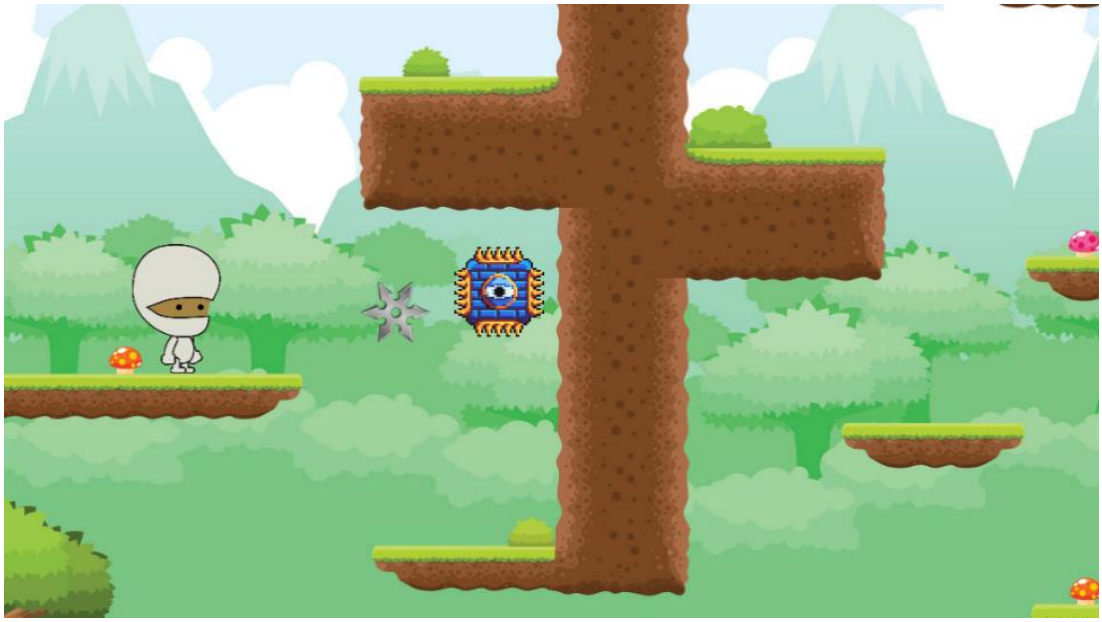
73 Oh no, the projectile can destroy the Player and LevelCollision!

The Projectiles should only interact with the Enemies. The best way to solve this is by using Collision Layers! The Enemies can be the only objects that exist on Layer 2 while the projectiles are the only objects that scan Layer 2!

Set the **Projectile** node (Area2D) in the Projectile's scene to **exist on none** of the Layers and **only scan for Layer 2**. Then, set the **Crusher** node (RigidBody2D) in the Crusher's scene to **exist on Layer 2** and **only scan for Layer 1** (for player and scene collision).



74 Playtest the project again. Do the projectiles collide with the crushers? Does the lifespan timer work properly?



Pause for **Sensei Stop #3**!

Check in with a Code Sensei before moving on. Confirm that Requirements #5-6 have been completed.

Reminder: Save your work!

Requirement #7: Ammo Counter UI

Create the UI to display the number of collectables held by the player.

- ☐ Create a **CanvasLayer** as a child to the **Level1** root node, then add **TextureRect** and **Label** child nodes.
    - Use the collectable's sprite as the Texture for the **TextureRect** and set the **Label's** Text to "0".
    - Rename both the **TextureRect** and **Label** to fit with the context of the game, like "AmmoSprite" and "AmmoCounter".
- ☐ Anchor the **TextureRect** and **Label** to the top-left corner and move the **Label** by a pixel offset so it does not overlap with the **TextureRect's** texture.
- ☐ Set the **Label**'s **Label Settings** property to a **New LabelSettings** to customize what the text looks like!
- ☐ Connect the **Label** to the "Collectable Counter" **@export** variable for Player.
- ☐ In player.gd, **uncomment** the previously commented lines of code in TODOs 5 & 7 so the text is properly updated.

Pause for **Ninja Stop #2**!

Does your project have...

- A **CanvasLayer** node?
- Renamed and anchored **TextureRect** and **Label**?
- Dynamic text updates when the value changes?

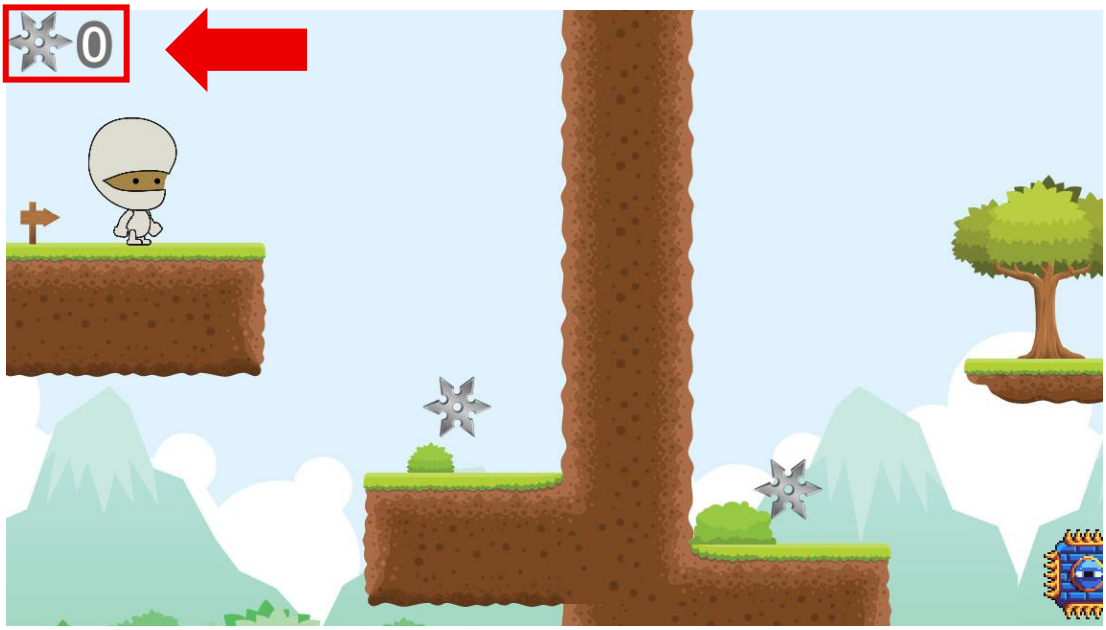
Reminder: Save your work!

Requirement #7 Hints

- If you created a custom collectable sprite, reuse that as the Texture for the **TextureRect**.
- Where in the Godot Editor can Control nodes be anchored with **Anchor Presets**?
- Where in the Inspector for the **Label** can the Transform's Position be moved?
- Do not change the Pivot Offset property in the Inspector for the **Label**.
- Where in the Godot editor can a node's `@export` variables be found?

Requirement #7 Resources

- Example UI layout:



- Anchor Presets: https://docs.godotengine.org/en/4.4/tutorials/ui/size_and_anchors.html#anchor-presets
  - Refer to Activities 12 – 14 in Silver Belt for help with Anchor Presets.

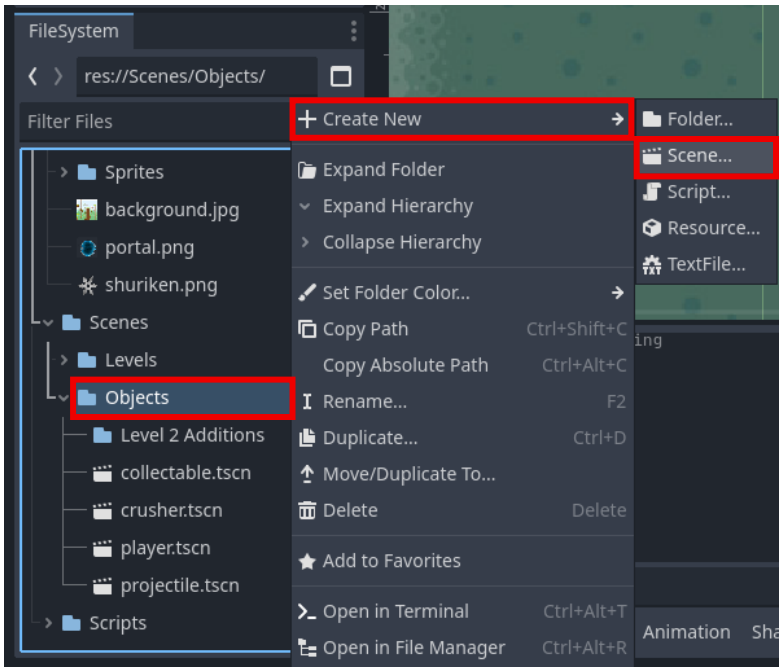
Requirement #8 (Instructions): Goal

75 Amazing work so far – you're crushing it! There's only one more component of the game left to implement: the Goal.

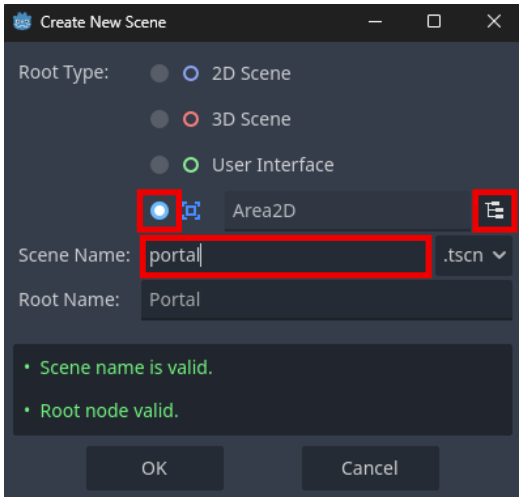
The Goal is responsible for knowing when the Player is overlapping it, then going to the next scene if all Enemies have been cleared.

~~**The Player is responsible for…**~~ • ~~User input & movement~~ • ~~Increase ammo & update UI when picking up Collectables~~ • ~~Throwing Projectiles~~ • ~~Resetting~~
~~**The Enemies are responsible for…**~~ • ~~Their own movement~~ • ~~Colliding with Player and Scene~~
The Goal is responsible for… • Checking if all Enemies are cleared • Going to the next scene
~~**The Collectables are responsible for…**~~ • ~~Telling the Player when picked up~~ • ~~Destroying themselves~~
~~**The Projectiles are responsible for…**~~ • ~~Their own movement~~ • ~~Colliding with Enemies~~
~~**The Scene is responsible for…**~~ • ~~Providing collision for other components~~

76 Create a new scene inside the **Scenes > Objects** folder to represent the goal.



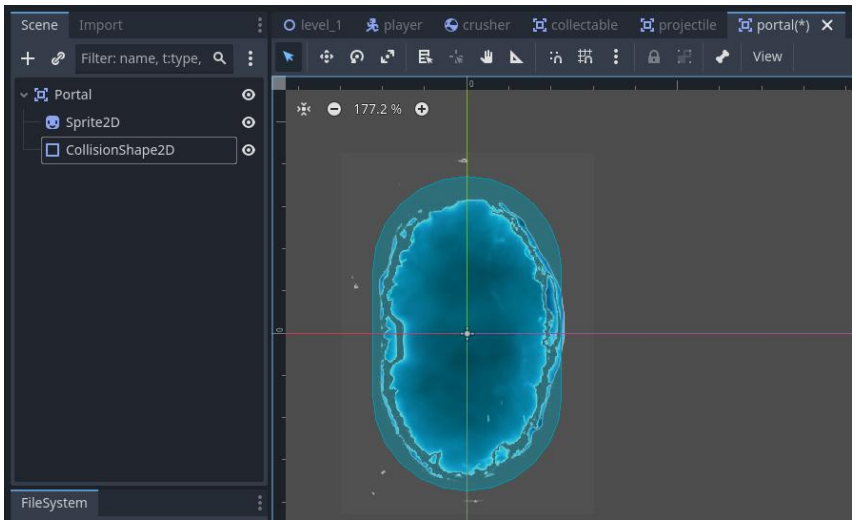
Set the **Root Type** to **Node** and set the type to **Area2D**. Name the scene **portal** and click **OK**.



77 Set up the nodes in the Portal scene.

Add a **Sprite2D** to load the portal texture and scale it to **0.25**.

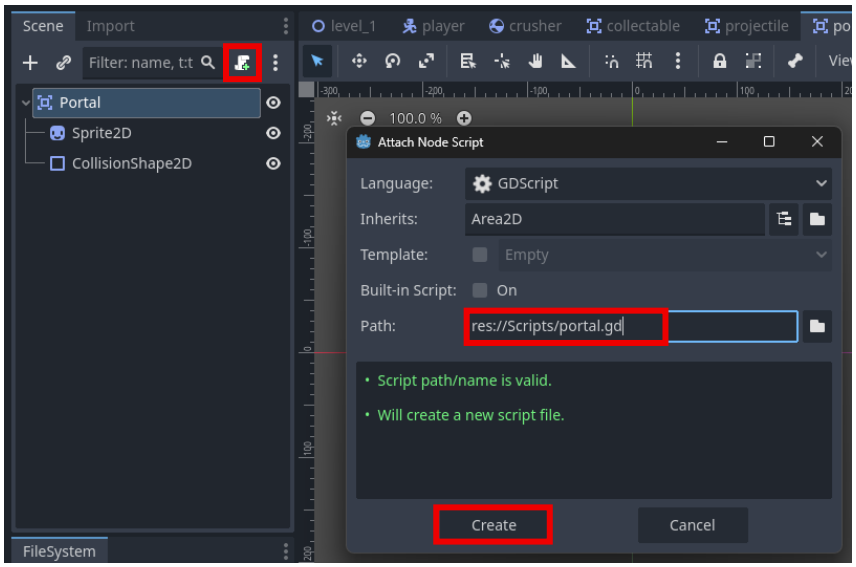
Add a **CollisionShape2D** as a child node to the root with a **CapsuleShape2D** selected and adjust its size to fit the portal's texture.



Reminder:

Select the CollisionShape2D in Scene to make the red dots appear. Drag the red dots around to scale the CapsuleShape2D until it fits.

78 Create a new **portal.gd** script attached to the **Area2D**.

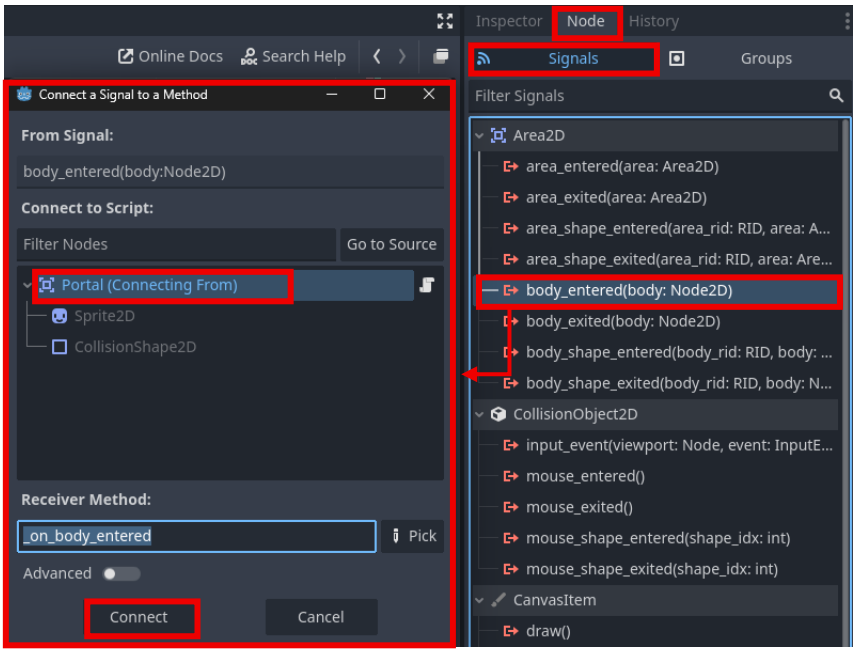


Inside the script, create a **next_scene** **@export** variable of type **PackedScene**.

```
1   extends Area2D
2
3   # next_scene ???
```

79 Ensure the root node is selected. Then, toggle **Inspector** to **Node** and open the **Signals** section.

Connect the **body_entered()** signal to a new `_on_body_entered()` receiver method in Portal's attached script.



80 Inside the new `_on_body_entered()` receiver method, set a new `enemy_count` variable to `get_tree().get_node_count_in_group("Enemy")` to count the current number of existing Enemies.

Underneath, write an `if`-statement that checks whether the body is in group `"Player"` `and` whether `enemy_count` is equal to `0`. Then, use `get_tree().call_deferred()` to call the `change_scene_to_packed()` method at the end of the current frame, with its parameter being `next_scene`.

```
1   extends Area2D
2
3   # next_scene ???
4
5   func _on_body_entered(body: Node2D) -> void:
6       # enemy_count ???
7       # if ???
8           # get_tree ???
9
```

Pro Tip:

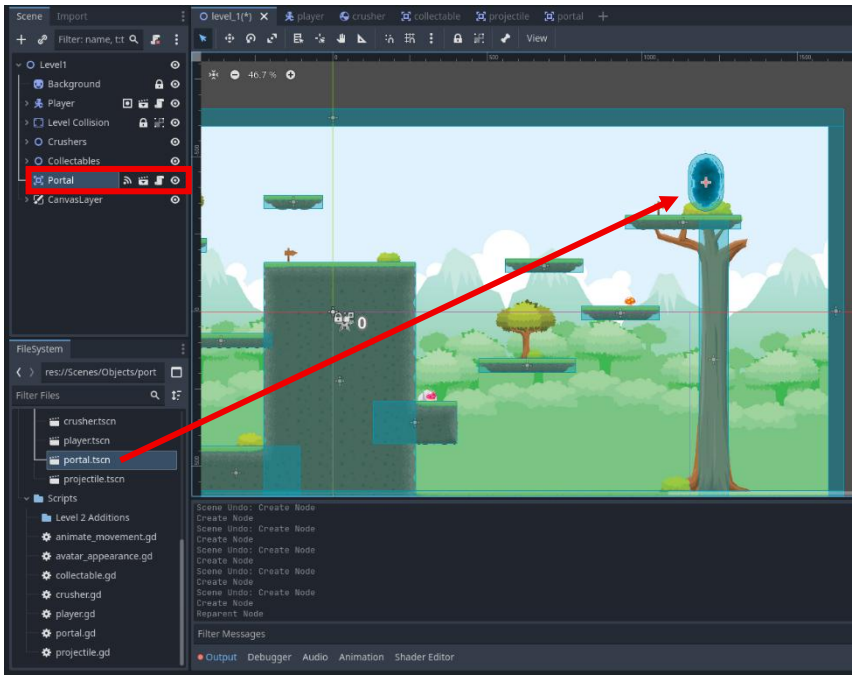
Sending parameters when calling a method using **`call_deferred()`** is done by separating with commas. For example, when calling the **`foo()`** method with the parameter **bar** at the end of the current frame, use **`call_deferred("foo", bar)`**.

81 Check the code! Update the script as needed.

```
1  extends Area2D
2
3  @export var next_scene: PackedScene
4
5  func _on_body_entered(body: Node2D) -> void:
6      var enemy_count = get_tree().get_node_count_in_group("Enemy")
7      if body.is_in_group("Player") and enemy_count == 0:
8          get_tree().call_deferred("change_scene_to_packed", next_scene)
9
```

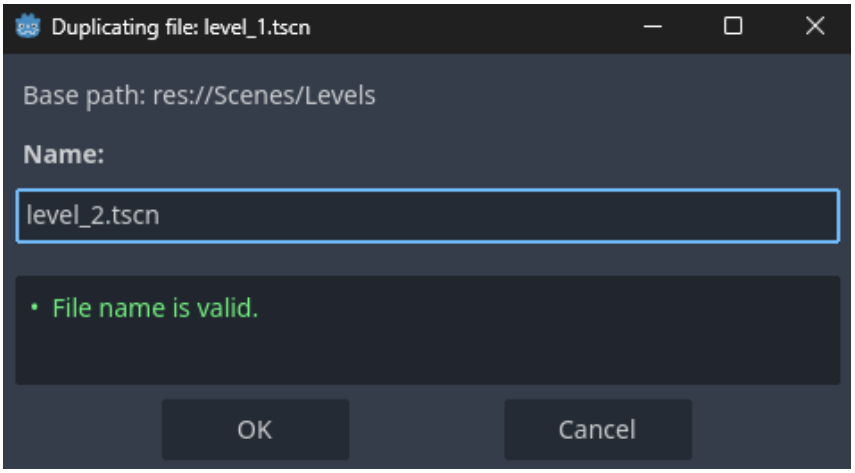
82 Return to the **level_1** scene.

Drag **portal.tscn** from **FileSystem** anywhere onto the scene, and check that it is a child to the root node in **Scene**.

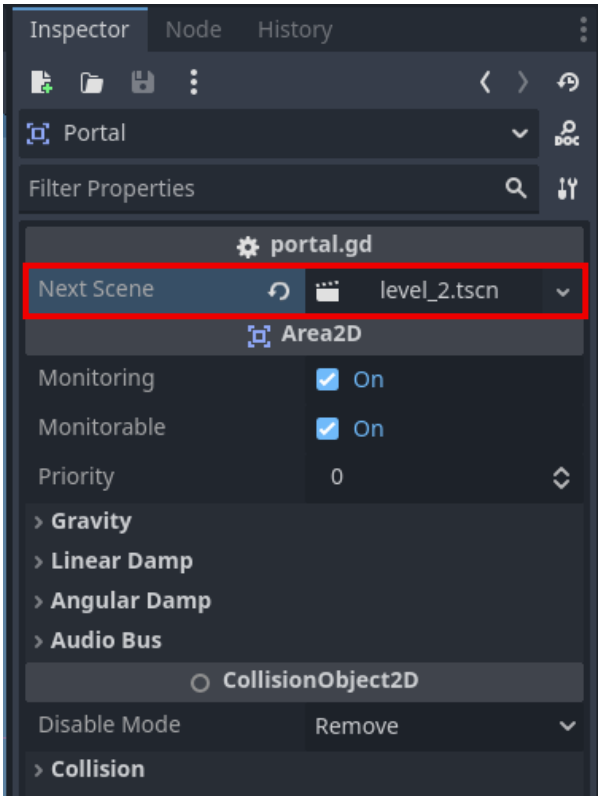


83 The portal has no destination yet!

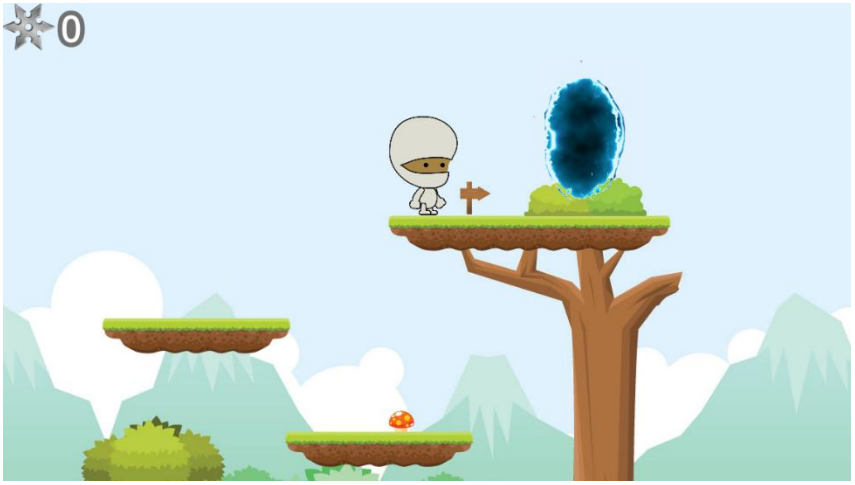
Select **level_1.tscn** in **FileSystem**, and press **CTRL+D** to duplicate it. In the prompt that shows up, set the name to **level_2.tscn** then click OK.



Make sure the currently opened scene is level_1.tscn, then open the Portal's **Inspector**. Set the **Next Scene** export variable to **level_2.tscn**.

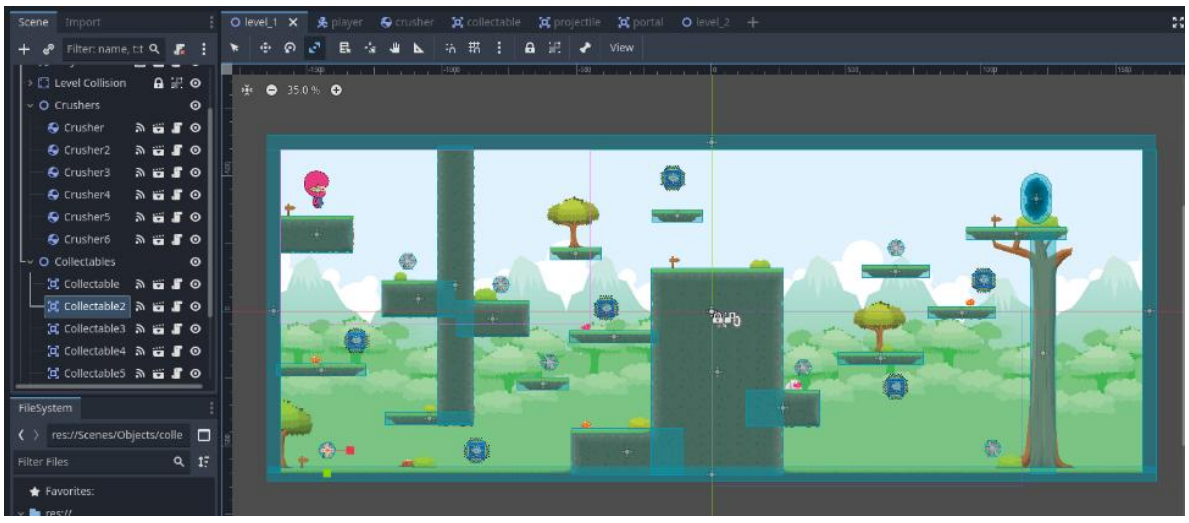


84 Playtest the project. Does the portal teleport the player only when there are no Enemies left?



85 Now that all the components for level 1 are complete, **fill the scene** with collectables and Enemies! Make sure each Crusher is a child to the **Crushers** node and each Collectable is a child to the **Collectables** node.

There must be at least as many Collectables as there are Enemies, or the game will become impossible to beat!



Pause for **Sensei Stop #4**!

Check in with a Code Sensei before moving on. Confirm that Requirements #7-8 have been completed.

Reminder: Save your work!

Requirement #9 (Instructions): Level 2

86

Level 2 should be interesting – think of some more systems that could be added!

For example, you can make Spikes, Doors with Switches, and more!

Spike

- Its own scene that is instantiated in Level 2
- **Area2D** with **Sprite2D** and **CollisionShape**/**Polygon2D** as children
- Only masks the Player's collision layer
- Script attached to the **Area2D** with **_on_body_entered()** signal connected to reset the body if it is the Player.

Door & Switch

- Separate nodes in Level 2
- Door is **StaticBody2D** and Switch is **Area2D**
- Both have **Sprite2D** and **CollisionShape2D** as children
- Switch only masks the Player's collision layer
- Script attached to the Switch's **Area2D** that gets the Door node with an `@export` variable.
- Connect Switch's **body_entered()** signal to destroy the Door if the body is Player and it hasn't already been flipped.

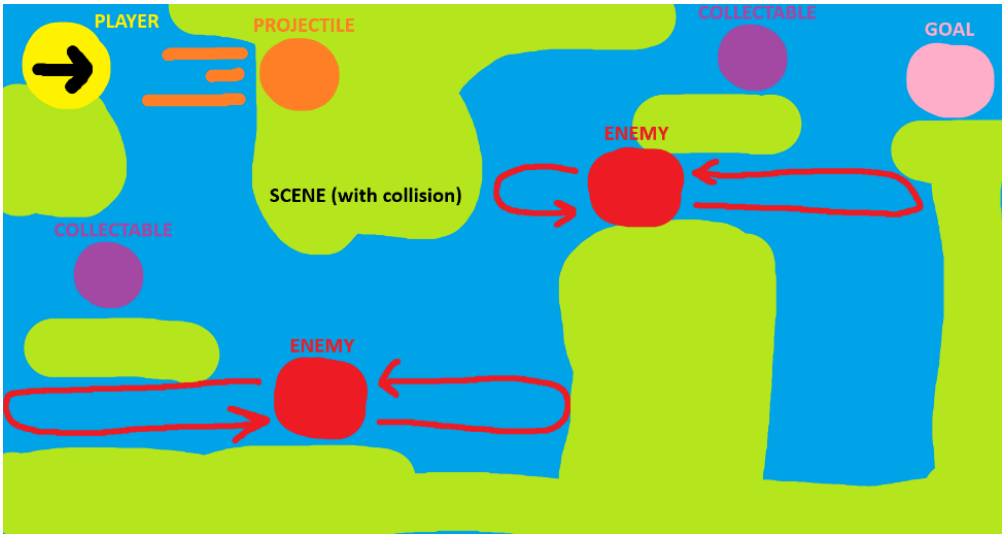
...and more!

- What ideas do you have?

87 Recall the beginning of this project.

When planning a game, drawing a simple sketch of the final version of the game can set the groundwork for all the major features of a game.

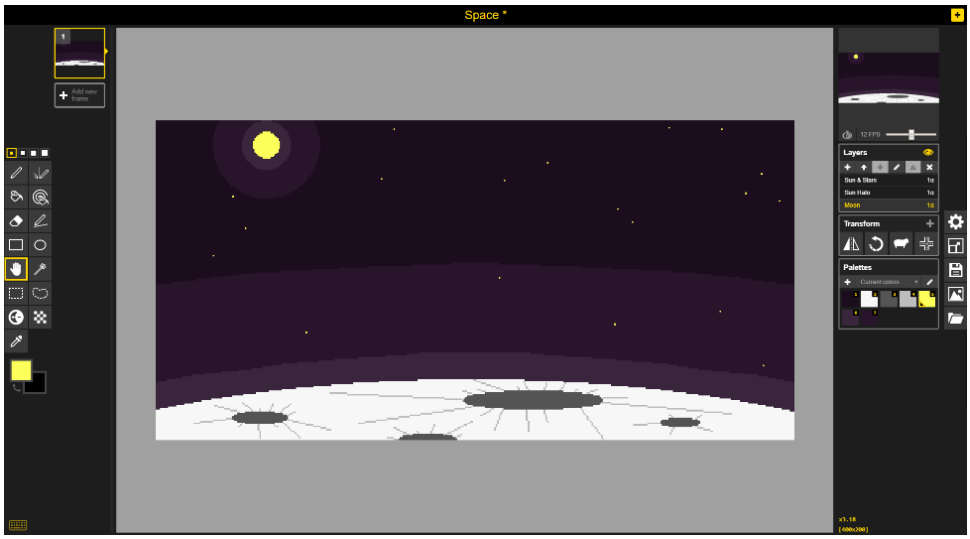
Try making a quick sketch in Paint or Piskel on your plans for Level 2. Include any new features that you brainstormed and try to color-code everything, so features are easily distinguishable.



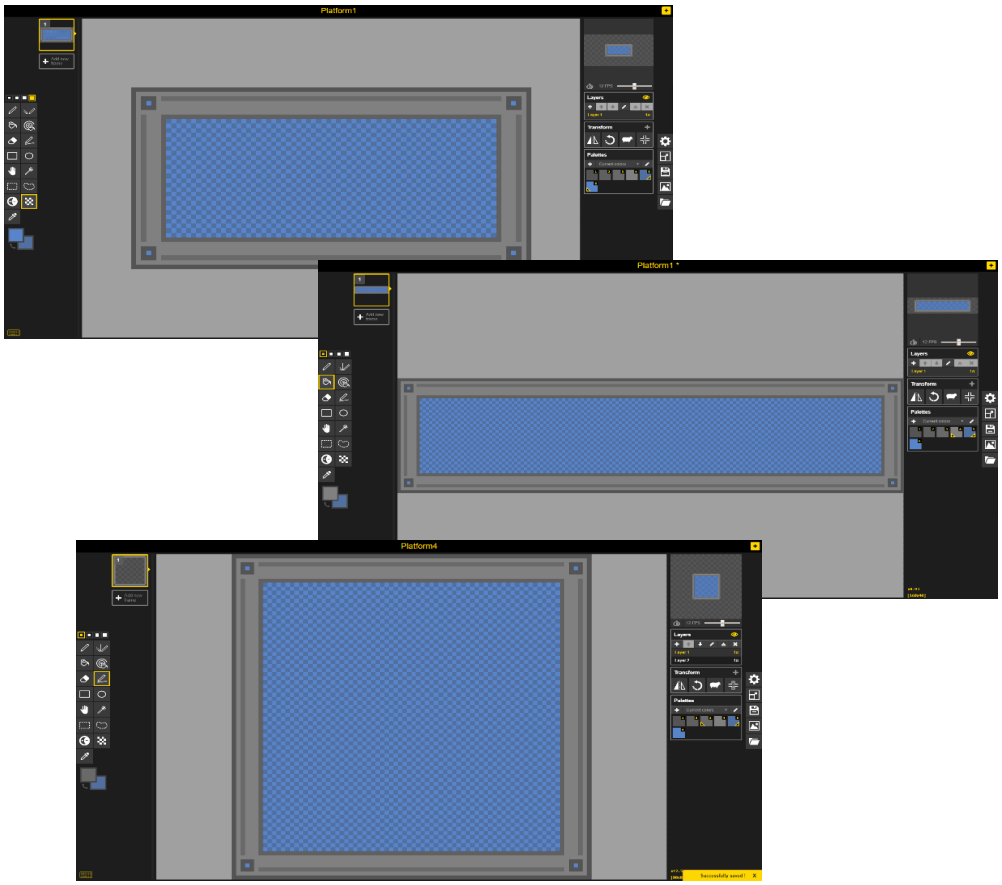
88 Draw the Level 2 background using **Piskel**! Level 2 will ***not*** have platforms as part of the background image. Instead, you will build the platforms separately.

Refer to the PB Activity 00 Ninja Guide for help with navigating Piskel.

Once you're done with the asset, go to **Export**, ensure **PNG** is selected, then click **Download**.



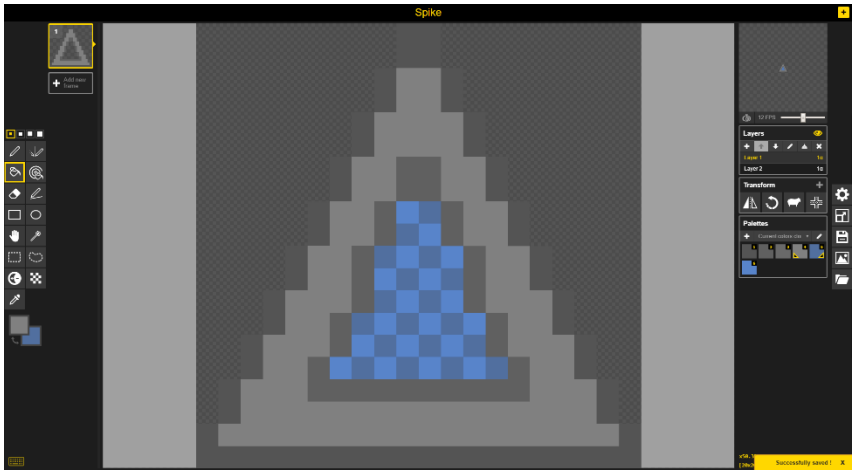
89 Draw the Level 2 platforms using Piskel! Keep the design simple and make a couple variants so the level doesn't feel dull.

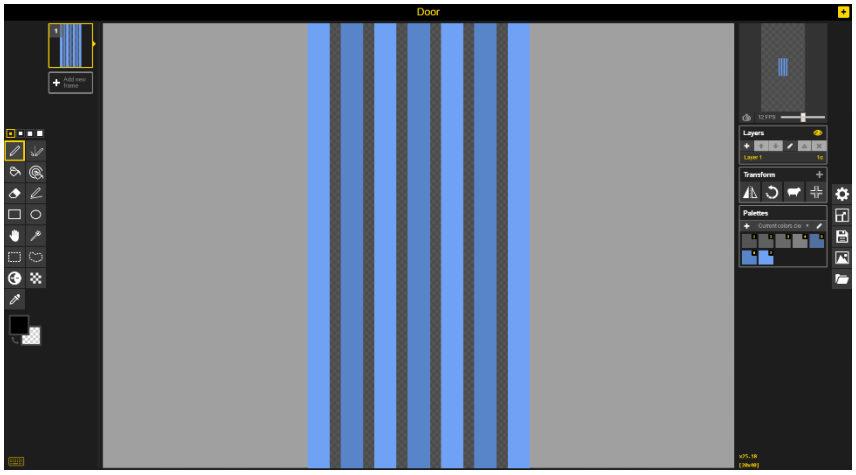


Once you're done with each asset, go to Export, ensure PNG is selected, then click Download.

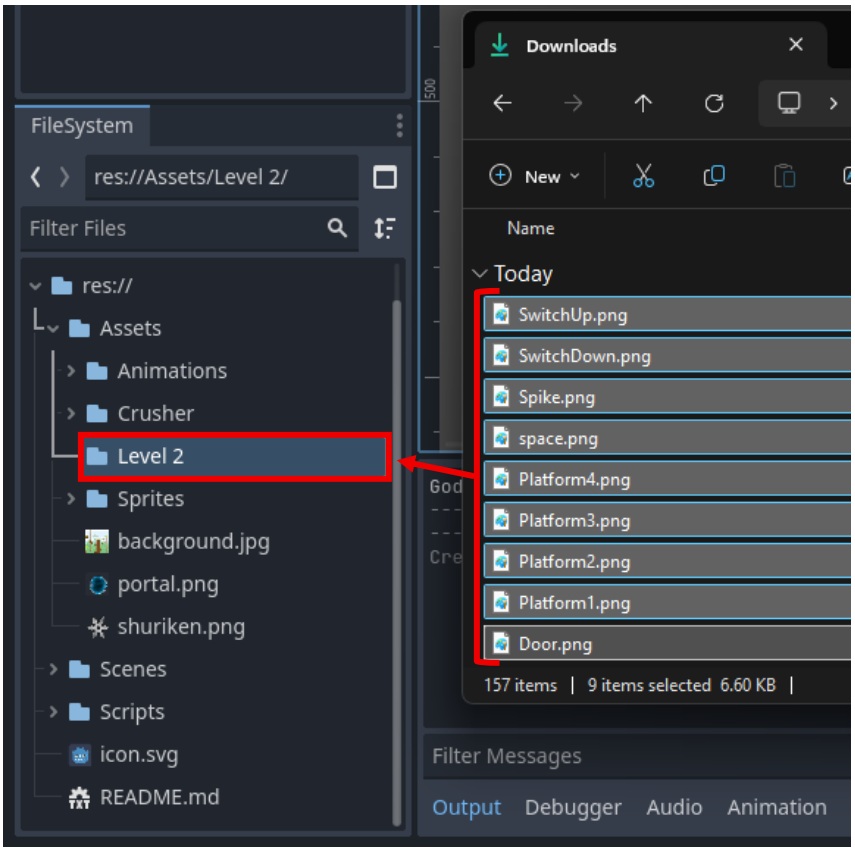
90 Draw assets for any other game elements using **Piskel**!

Once you're done with each asset, go to **Export**, ensure **PNG** is selected, then click **Download**.



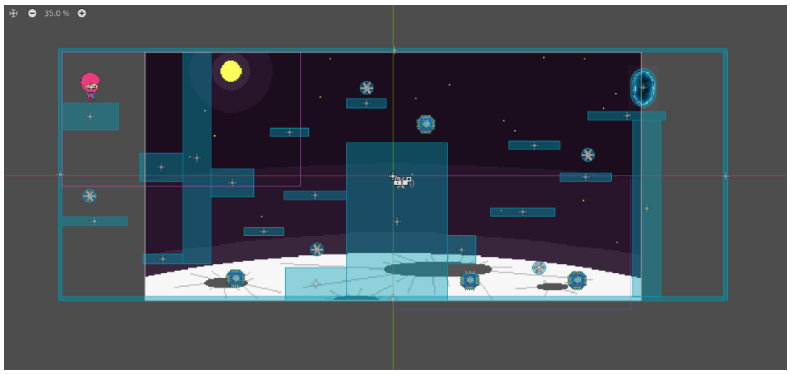


91 In Assets, create a new **Level 2** folder. Then, navigate to your computer's Downloads folder and drag all new assets into the **Level 2** folder.



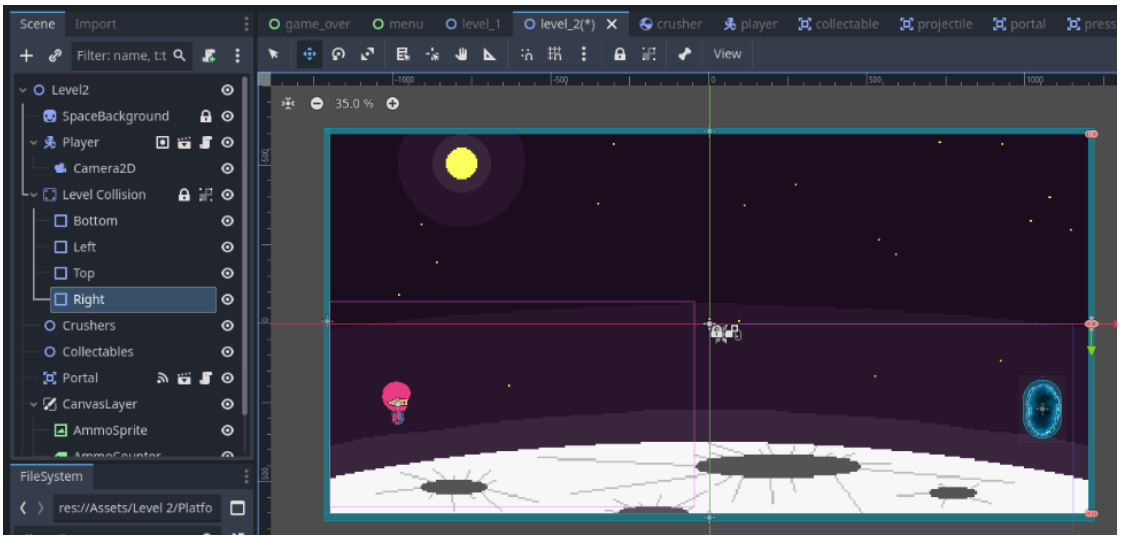
92 Open level_2.tscn. Select the Background node and change its Sprite2D to the newly created asset.

Scale the asset to fit within the scene, and remember the exact scaling used so the same values can be used for the other artwork!



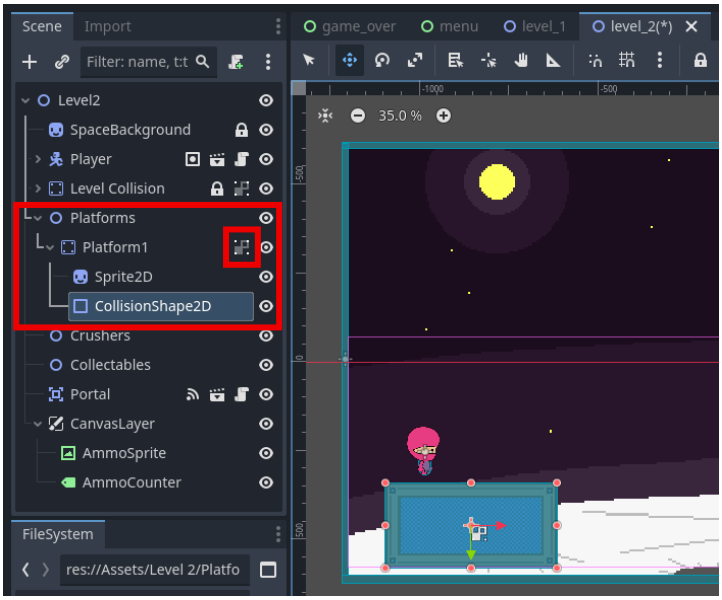
93 Remove all level collision *except for* the outside borders and remove all Crushers and Collectables. Adjust the position and scaling of each outer wall to fit the new background.

In **Scene**, navigate to **Player > Camera2D** and set the limits to fit the new dimensions of the scene.

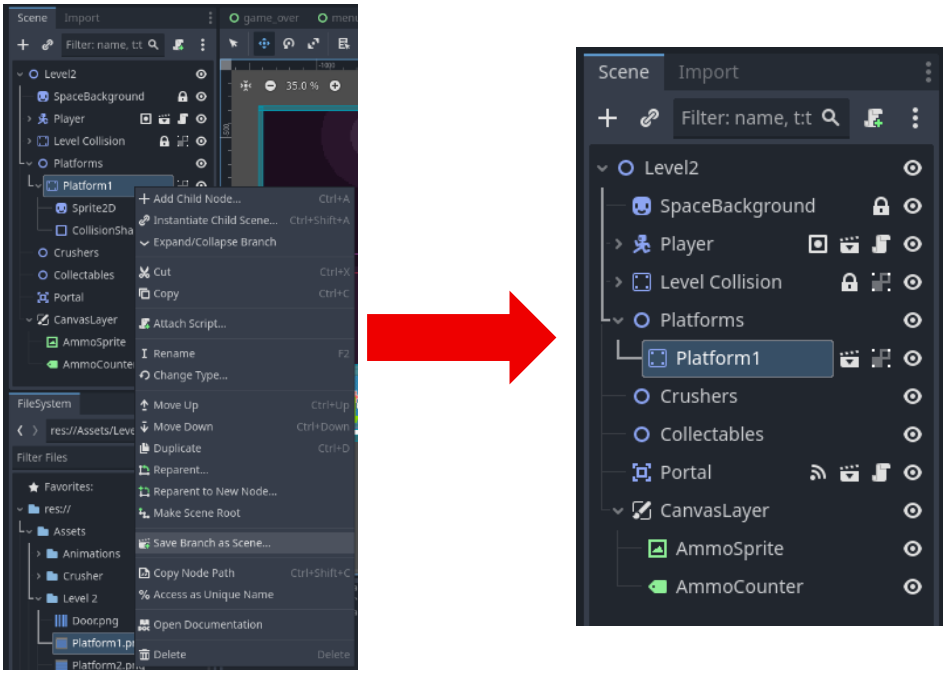


94 Build the first platform! Create a new **Platforms** Node2D as a child to the root node with a **Platform1** StaticBody2D as a child.

Add a **Sprite2D** and a **CollisionShape2D** as children to Platform1, set the Sprite2D's **texture** to your base platform asset, and match its **shape** using CollisionShape2D.

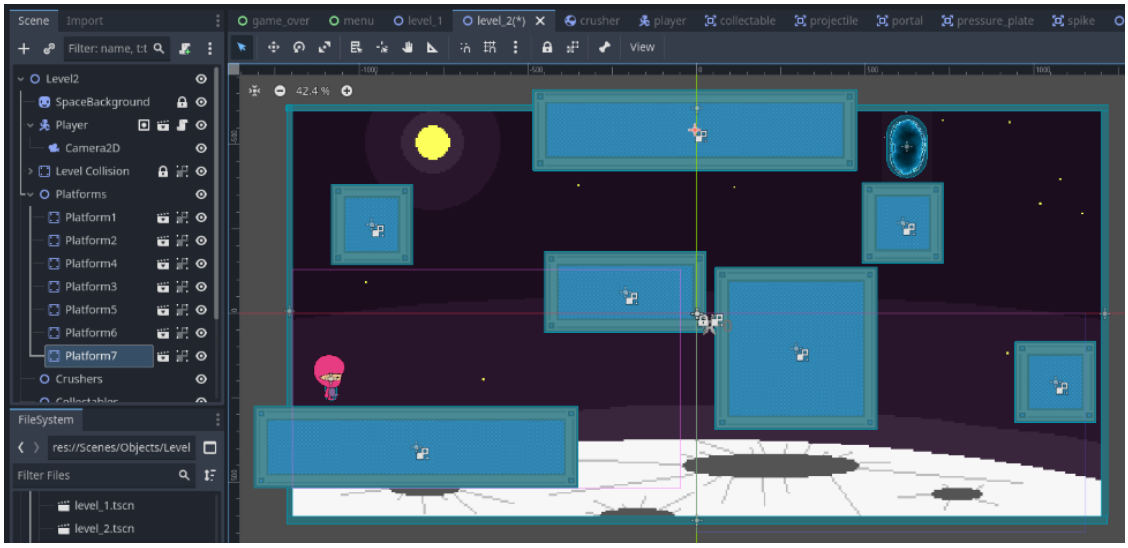


Once you're done, select Platform1 and press CTRL+G to Group Selected Nodes. Then, right-click Platform1 and select **Save Branch as Scene...**. In the popup, navigate to **Scenes > Objects > Level 2 Additions** and click Save.

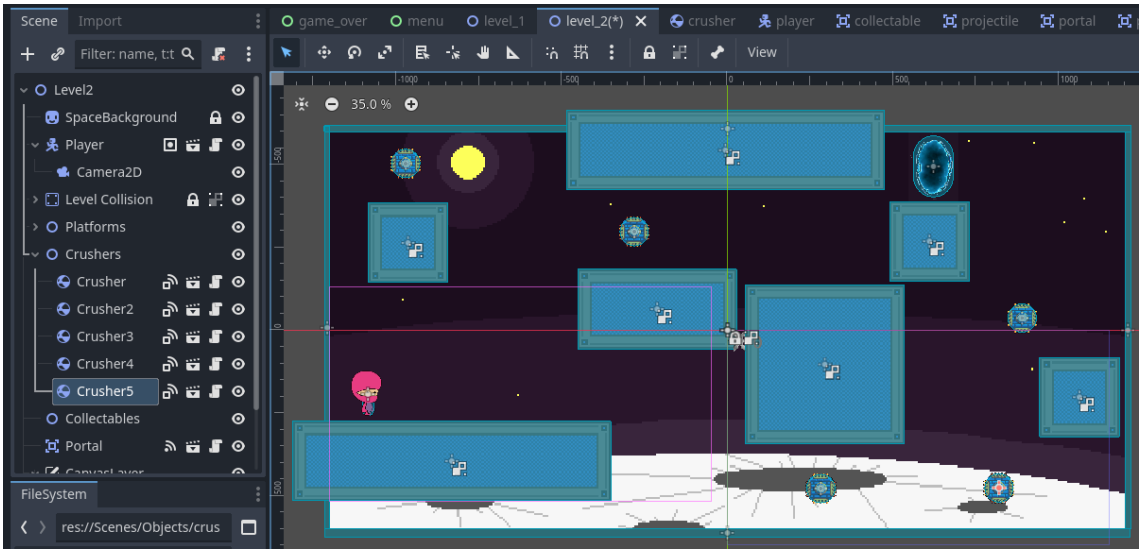


95 In **FileSystem > Scenes > Objects > Level 2 Additions**, duplicate platform_1.tscn and adjust the Sprite2D's **texture** and CollisionShape's **shape** to fit for all other platform variants you made in Piskel!

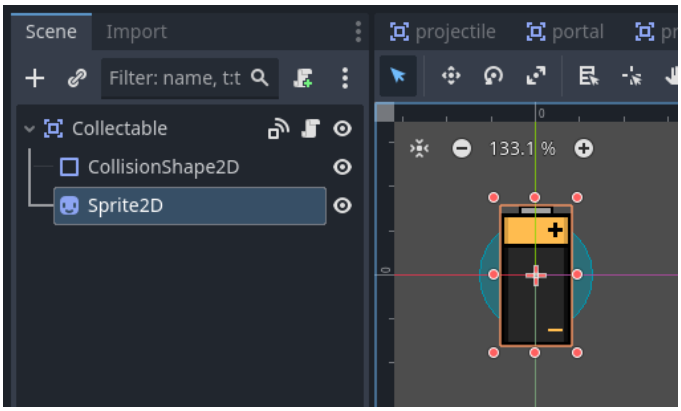
Then, build the layout for Level 2 using a combination of all platforms, all placed as children to the Platforms node. Adjust the positions of the Player and Portal where necessary.



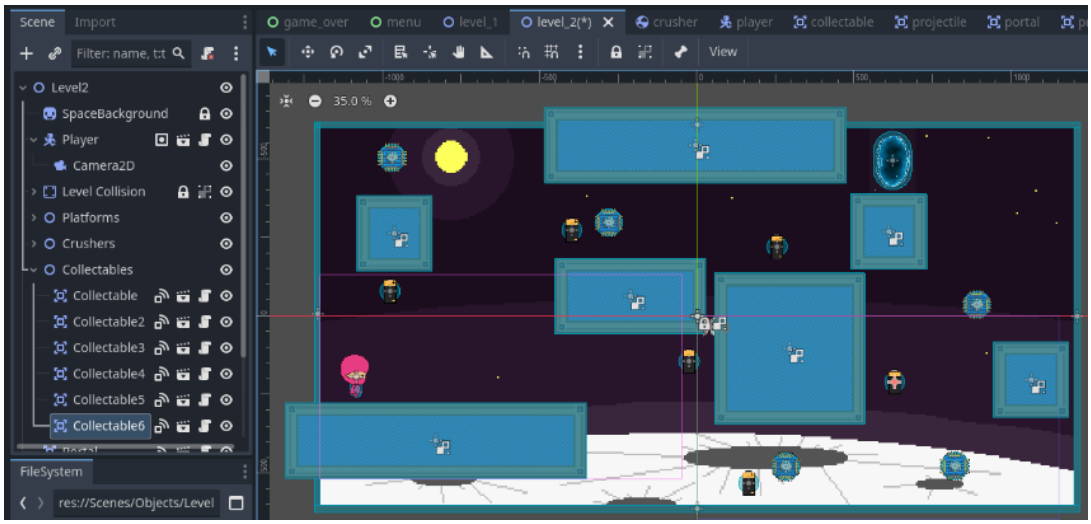
96 Add the crushers into the new scene, all placed as children to the **Crushers** node.



97 In **FileSystem > Scenes > Objects**, duplicate collectable.tscn into **collectable_lv2.tscn** and place it into the **Level 2 Additions** folder. Then, set the Sprite2D's texture to your new collectable asset.



Add the new collectables to the scene, all placed as children to the **Collectables** node. There must be at least as many collectables as there are crushers so the level can be completed.



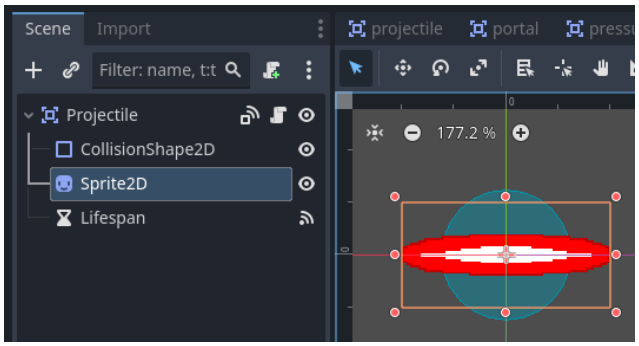
98 To allow for the projectile to change across levels, change the `PROJECTILE` `const` to `@export` `var` of type `PackedScene` in player.gd.

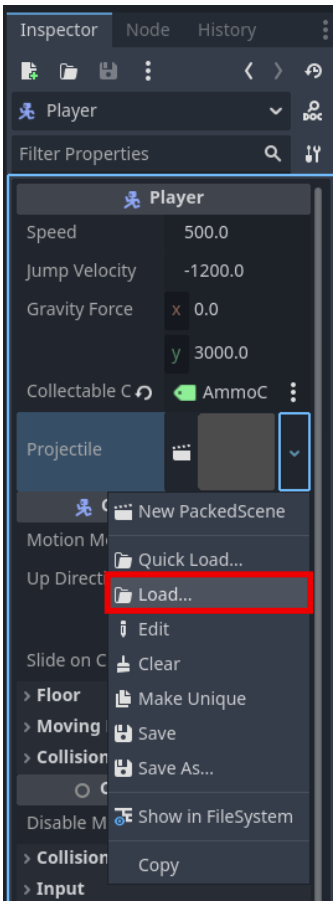
```
22  # --------------------
23  # TODO 6
24  # Load projectile scene
25  # --------------------
26  @export var PROJECTILE: PackedScene = preload("res://Scenes/Objects/projectile.tscn")
27
```

Now, the projectile can be changed based on the current level, with the default value always being **projectile.tscn**.

99 In **FileSystem > Scenes > Objects**, duplicate projectile.tscn into **projectile_lv2.tscn** and place it into the **Level 2 Additions** folder. Then, set the Sprite2D's texture to your new projectile asset.

In the Player's Inspector for Level 2, update the Projectile scene to **projectile_lv2.tscn**.





100 Build your own game elements for Level 2! Think back to the planning phase – which nodes should you use and how should you combine scripting and signals to make everything work together?

The hints from the planning phase of Level 2 are provided below.

Spike

- Its own scene that is instantiated in Level 2
- **Area2D** with **Sprite2D** and **CollisionShape/Polygon2D** as children
- Only masks the Player's collision layer
- Script attached to the **Area2D** with **_on_body_entered()** signal connected to reset the body if it is the Player.

Door & Switch

- Separate nodes in Level 2
- Door is **StaticBody2D** and Switch is **Area2D**
- Both have **Sprite2D** and **CollisionShape2D** as children
- Switch only masks the Player's collision layer
- Script attached to the Switch's **Area2D** that gets the Door node with an `@export` variable.
- Connect Switch's **body_entered()** signal to destroy the Door if the body is Player and it hasn't already been flipped.

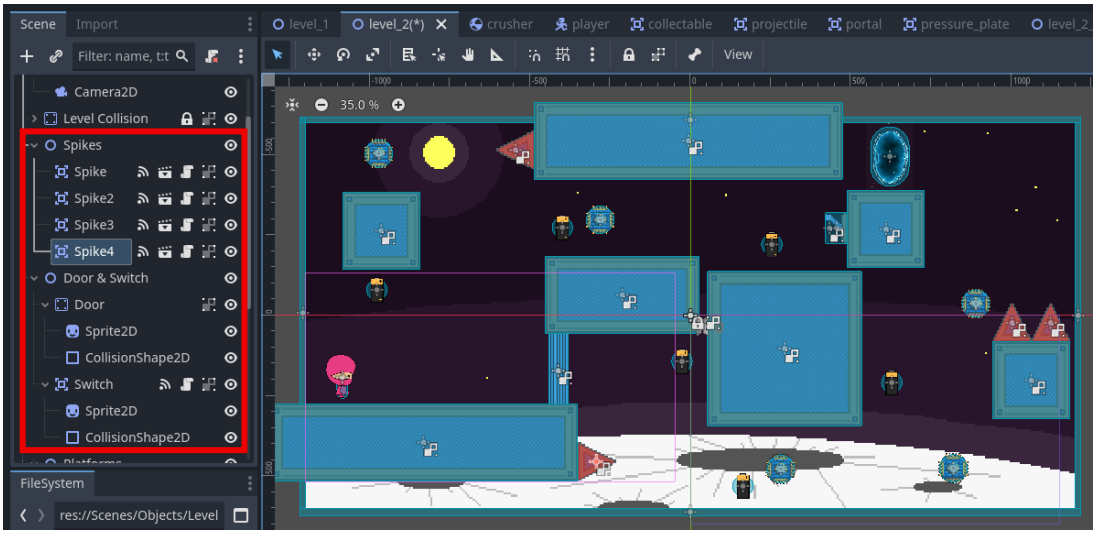
...and more!

- What ideas do you have?

101

Place the newly created game elements into Level 2!

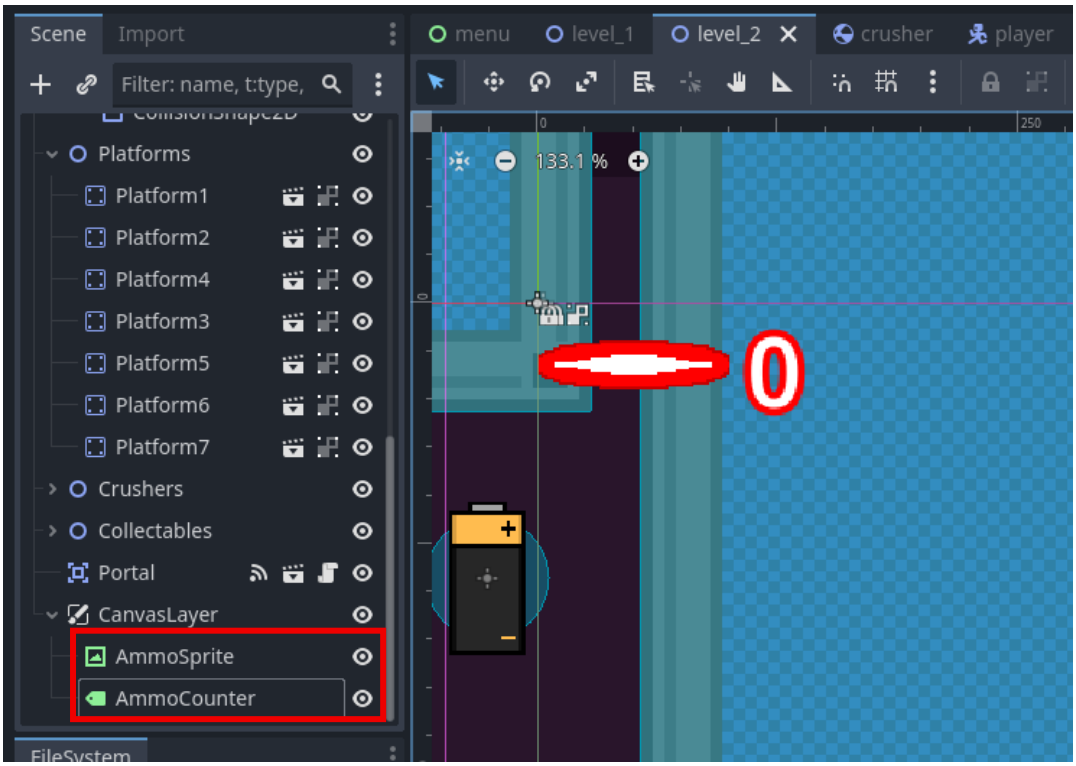
Set the portal to load the Level 1 scene for now. This will be changed once a game over screen is added.



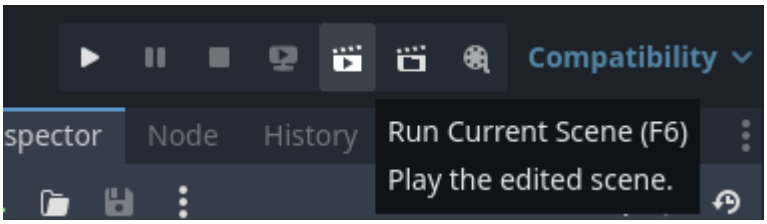
102

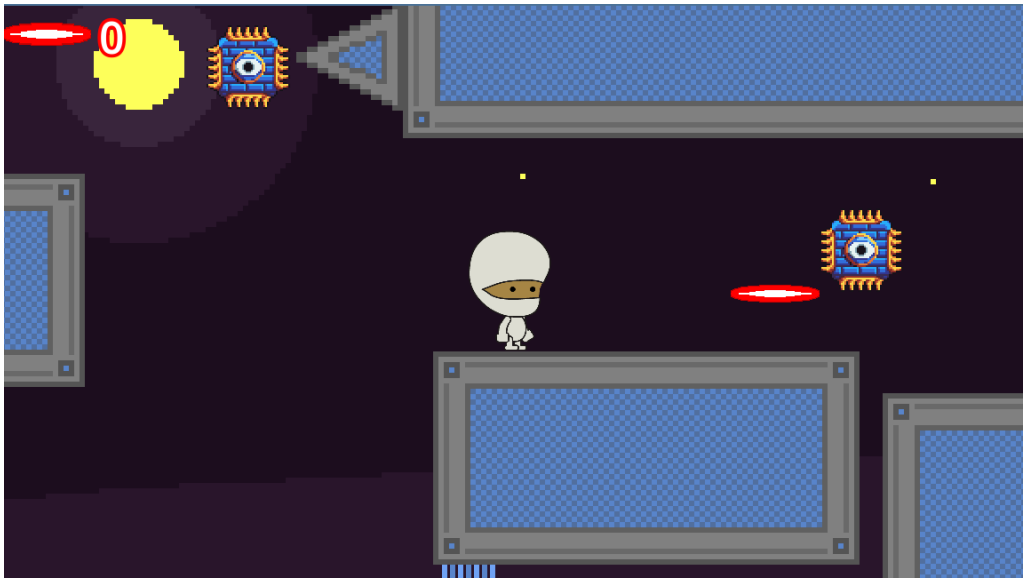
Update the UI to show the new projectile sprites!

Set the **Texture** and change the **colors** and **borders** of the label to match the new projectile asset.



103 Playtest the project using **Run Current Scene (F6)**! Does everything work as intended?





Reminder:

Set the portal to load the Level 1 scene for now. This will be changed once a game over screen is added.

Pause for **Sensei Stop #5**!

Check in with a Code Sensei before moving on. Confirm that Requirement #9 has been completed.

Reminder: Save your work!

Requirement #10: Main Menu Scene

Design the UI in a new main menu scene.

- ☐ Create a new **menu.tscn** scene in **FileSystem > Scenes > Levels** with a **Control** node as the root. Set its anchor preset to **Full Rect**.
- ☐ Set menu.tscn as the new main scene in **Project Settings**.
- ☐ Add the background used in Level 1 as the background image of the main menu using a **Sprite2D**.
- ☐ Add 3 additional nodes as children to the main root: an empty **Control** node to organize the player controls, a **Label** for the title, and a **Button** to start the game. Rename the empty Control node to **PlayerControls**, the Label to **Title**, and the Button to **StartGame**.
- ☐ Set PlayerControls' anchor preset to **Full Rect,** Title to **Center Top**, and StartGame to **Center**.
- ☐ Set the **text** properties in the **Inspector** of Title and StartGame, and in **Control > Theme Overrides** customize the **Color**, **Constant**, and **Font Size** properties.
- ☐ Add a **Label** and an empty **Control** node as children to PlayerControls. Use the label to say "Player Controls:" then rename the empty Control node to **Left**. Set its anchor preset to **Center**.
    - Add a **Label** and a **ColorRect** as children to Left, with a 2nd **Label** as a child to the new ColorRect. The 1st label will be the control: "Move Left". The 2nd label will be the button pressed: "<".
    - Set the 2nd label's anchor preset to **Full Rect**, and, in the Inspector, set its Horizontal and Vertical alignments to Center. In the Inspector, use the **Theme Overrides** drop-down to customize the 2nd label's color so it is visible against the white ColorRect.
    - Adjust the 1st label's position using Move Mode (W) so it does not overlap with the ColorRect and 2nd label.
    - Duplicate **Left** three times to show the other controls:
        - Control Nodes: Left, Right, Flip, Throw
        - First Label: "Move Left", "Move Right", "Gravity Flip, "Throw"
        - Second Label: "<", ">", "space", "left CTRL"

- **Move** each control around, **adjust the size** of each ColorRect, and **tweak** the values in **Theme Overrides** to make the player controls appear visually pleasing.

- [ ] Attach a new script to the StartGame button. Then, connect the button's **_pressed()** signal to the new script. Use `get_tree().change_scene_to_file()` to change the scene to **level_1.tscn**.

- [ ] **Adjust the positions** of PlayerControls, Title, and StartGame's using Move Mode (W) to be visually appealing.

Pause for **Ninja Stop #3!**

Does your project have...

- A **menu** scene?
- A **title**, **player controls**, and **background**?
- A "Start Game" **button** that loads level_1.tscn?

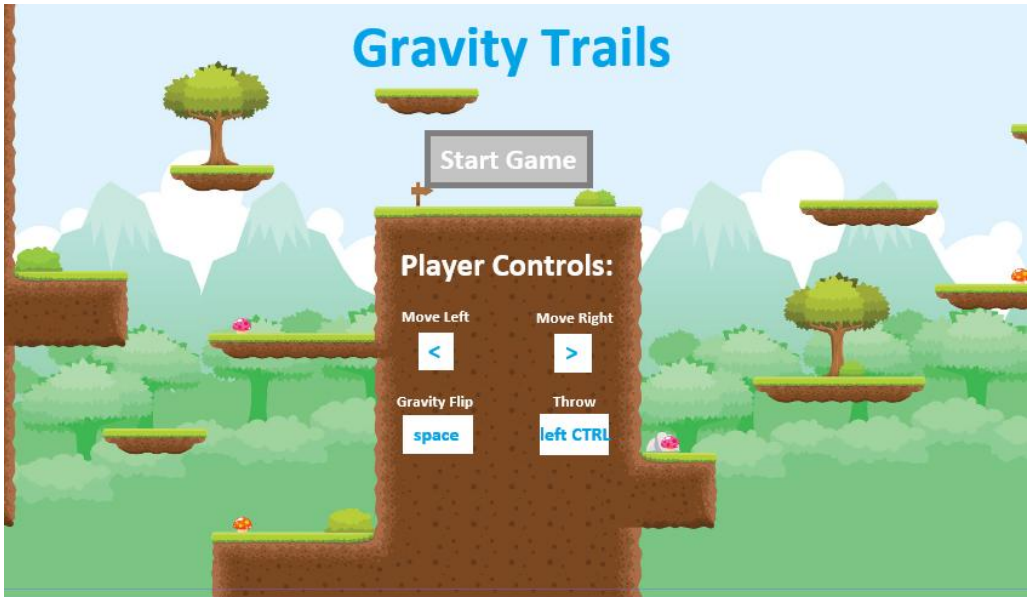
Reminder: Save your work!

Requirement #10 Hints

- The main scene can be set in **Project Settings > General > Application > Run**.
- Where in the Godot Editor can **Control** nodes be anchored with **Anchor Presets**?
- Where in the Inspector for **Control** nodes can the Transform's Position be moved?
- If you are having trouble selecting and adjusting the **ColorRects**, select them in Scene and press **CTRL+G** to Group Selected Nodes.
- Use the Theme Overrides drop-down menu in the Inspector to customize the font size and outline for each Label.
- The order of nodes matters in Scene. The button should be last, so it captures mouse input:
    - Menu
        - Background *(Sprite2D)*
        - PlayerControls *(Control)*
        - Title *(Label)*
        - StartGame *(Button)*
- To get a scene's path for use with `get_tree().change_scene_to_file()`, select a scene in FileSystem and press **CTRL+SHIFT+C** or right-click it and select **Copy Path**.

Requirement #10 Resources

- Example main menu:



- Anchor Presets: https://docs.godotengine.org/en/4.4/tutorials/ui/size_and_anchors.html#anchor-presets
  - Refer back to Activities 12 – 14 in Silver Belt for help with Anchor Presets.

Requirement #11: Game Over UI

Build the Game Over UI so it appears when the Player exits Level 2!

- ☐ Create a **game_over.tscn** scene in **Scenes > Levels** with the root type of User Interface (Control). Inside the scene, add three nodes as children to the root: Sprite2D, Label, and Button.
- ☐ Rename the nodes to something that makes sense, like **Background**, **GameOverText**, and **ReplayButton**.
- ☐ Set the **Text** for the Label to **You Win!** and set its **Font Size** anywhere within **40-60px**. Set the **Text** for the Button to **Replay** and set its **Font Size** anywhere within **30-40px**.
- ☐ Set the Sprite2D's texture to the background used in Level 2 and transform it so it covers the entire viewport.
- ☐ Anchor the Label and Button to **Center** and adjust their positions using **Move Mode (W)** so they are not overlapping.
- ☐ Attach a new script to ReplayButton. Then, connect the button's **_pressed()** signal to the new script. Use `get_tree().change_scene_to_file()` to change the scene to **menu.tscn**.
- ☐ In **level_2.tscn**, set the portal's next scene to **game_over.tscn**.

Pause for **Ninja Stop #4**!

Test your project! Does it have...

- A game over UI that only appears at the end?
- Level 2's portal loads the game over scene?
- A replay button that loads menu.tscn?

Reminder: Save your work!

Requirement #11 Hints

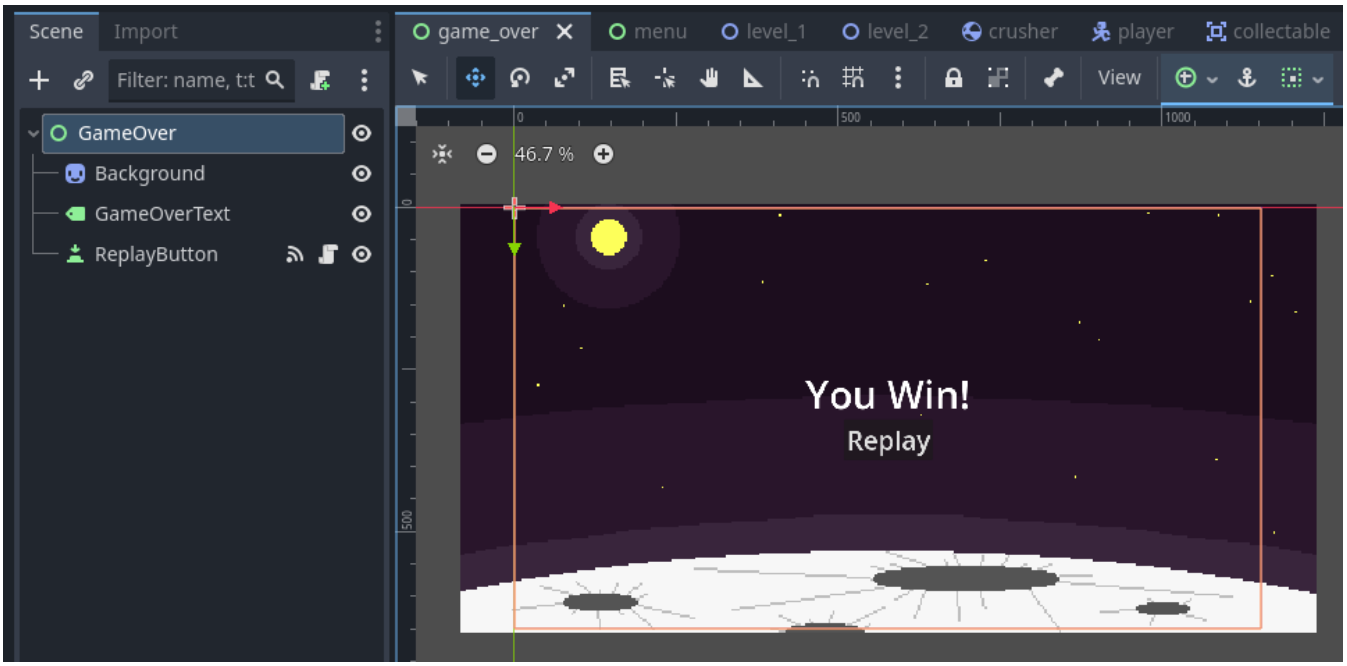
- Where in the Godot Editor can **Control** nodes be anchored with **Anchor Presets**?

- Use the Theme Overrides drop-down menu in the Inspector to customize the font size and outline for each Label.

- To get a scene's path for use with `get_tree().change_scene_to_file()`, select a scene in FileSystem and press **CTRL+SHIFT+C** or right-click it and select **Copy Path**.

Requirement #11 Resources

- Anchor Presets: https://docs.godotengine.org/en/4.4/tutorials/ui/size_and_anchors.html#anchor-presets
    - Refer back to Activities 12 – 14 in Silver Belt for help with Anchor Presets.
- Example Scene hierarchy and UI layout:



Pause for **Sensei Stop #6**!

Congratulations on building your very own platformer game in Godot! Great job!

Before submitting, check in with a Code Sensei to confirm that Requirements #10-11 have been completed, then reflect on the following:

- What did you learn about level design?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!