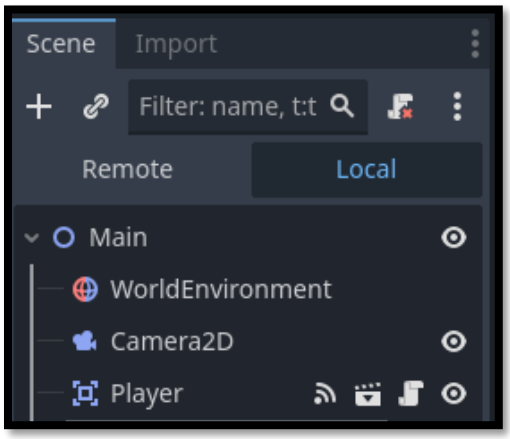
CODE NINJAS®

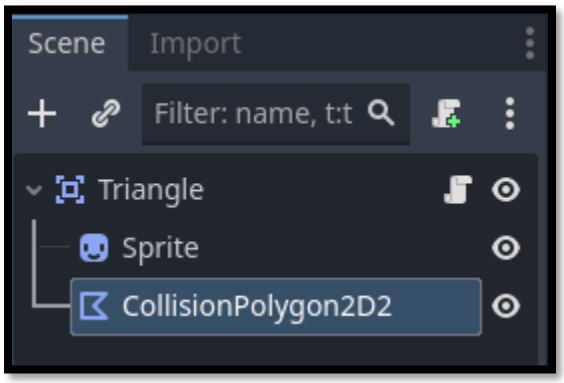
Bronze Belt Ninja Guide

Activity 06: SuperShapes

PackedScenes

Scenes can be created as **sub-trees** in the **Scene menu,** but scenes can also be created in the **FileSystem** as separate trees known as **PackedScenes**.





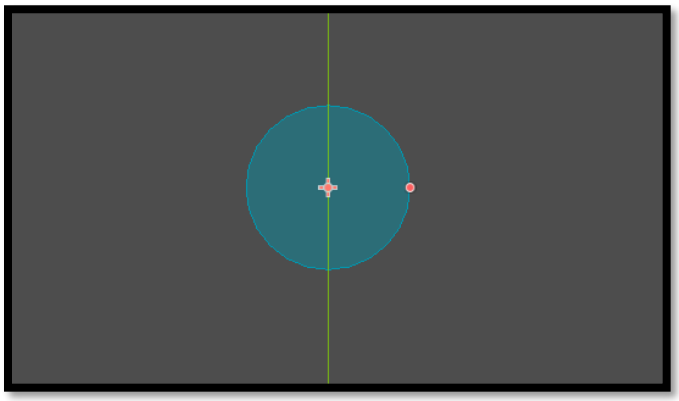
PackedScenes are a **resource** that holds all the data from a saved scene. It does not exist in the **main scene tree** until it is **instanced** when the game is running. This makes it possible to load and spawn scenes dynamically, such as creating enemies or obstacles when needed. To do this in code, first load the PackedScene, then call **instantiate()** it to create a copy of it, then use **add_child()** to add it to an existing scene so it appears in the game.

```
var triangle_scene: PackedScene = preload("res://Scenes/Objects/triangle.tscn")
add_child(triangle_scene.instantiate())
```

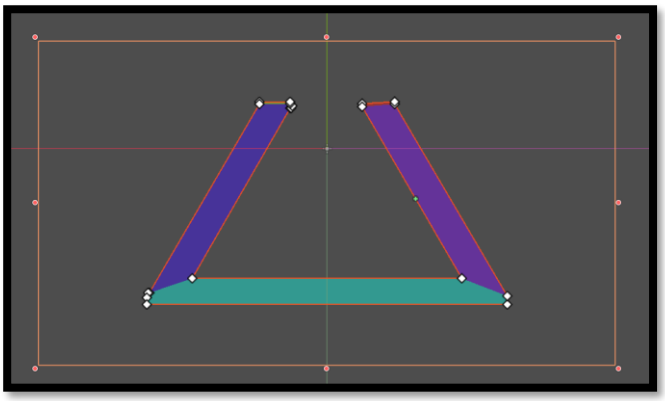
Collision Shapes

Collision shapes define the shape used for detecting interactions and collisions between objects. These are attached to **physics bodies** like **Area2D, StaticBody2D,** or **RigidBody2D** nodes.

Previously, **CollisionShape2D** nodes have been used to create basic **collision shapes.** A **CollisionPolygon2D** allows the user to create a custom polygon shape, rather than a standard rectangle or circle. This helps create more precise collision shapes for sprites and other objects that are irregularly shaped, allowing for more accurate collisions such as better hit detection, and more realistic physics behavior in the game.



CollisionShape2D

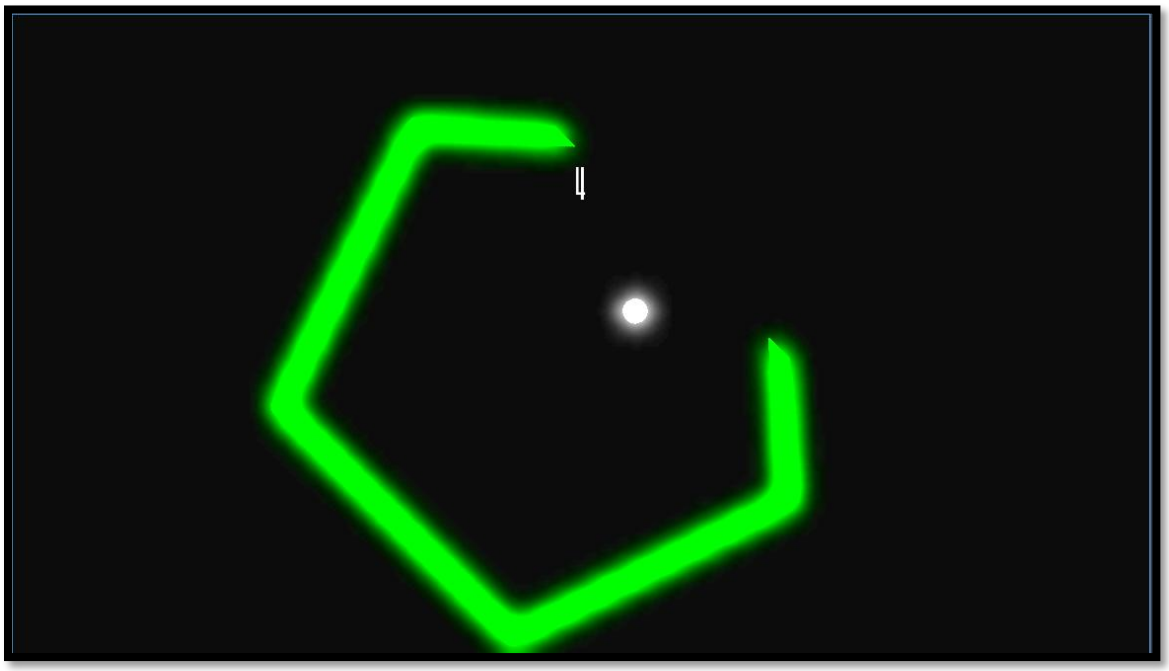


CollisionPolygon2D

Activity 06: SuperShapes

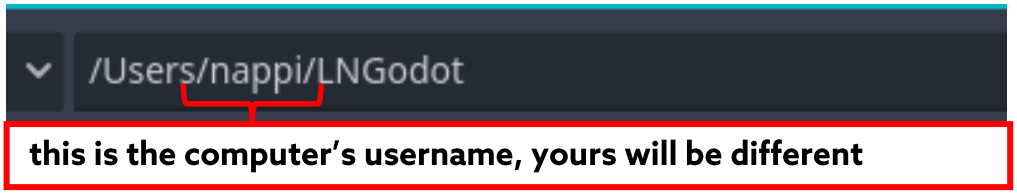
In this activity, you will create a fast-paced dodging game where the player rotates around a center point to avoid incoming polygon shaped obstacles. This project will help you learn about scene management, polygon colliders, and how objects rotate around their local origin points.

By the end of this activity, you will have explored how to create a rotating player, switch between different game scenes and canvases, and use polygon colliders to detect collisions with various obstacle shapes.

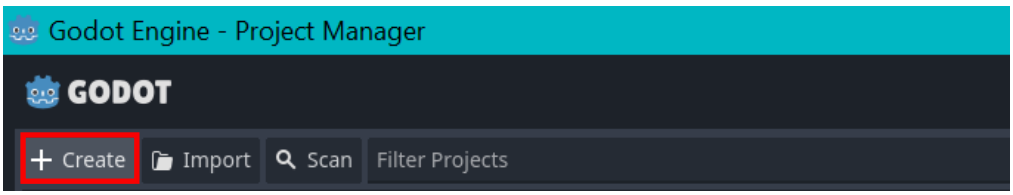


1 All projects will be stored in a path like:
/Users/[MyComputerUsername]/[MyInitials]Godot

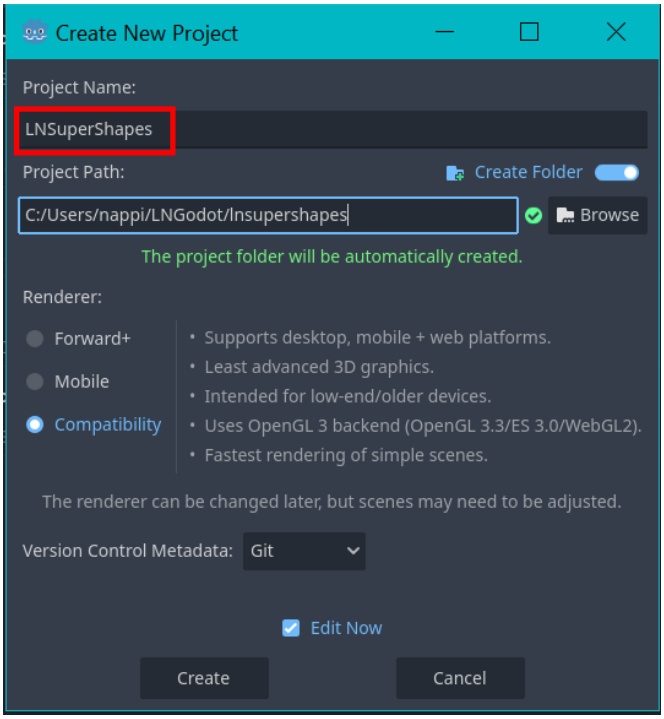
Don't worry if your path looks slightly different from the image shown! All computers have their own username.



2 After opening Godot, in the top left corner select **+ Create.**

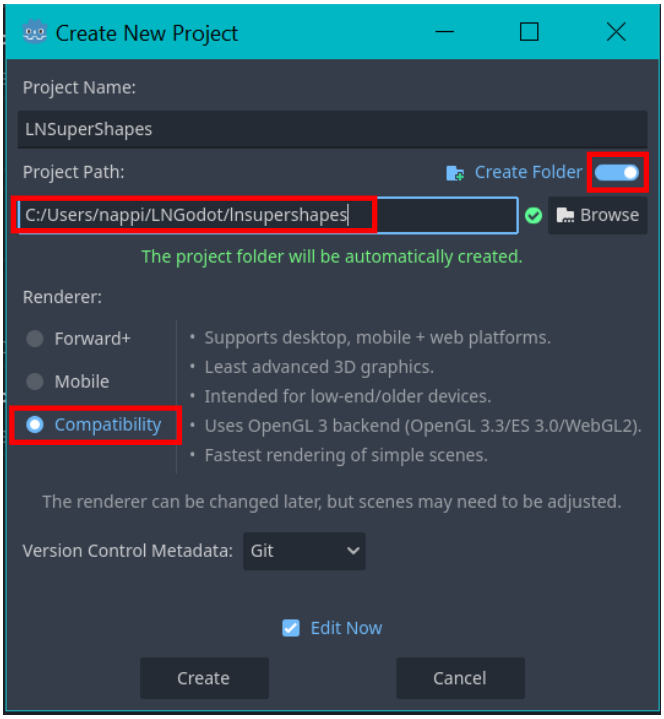


A **Create New Project** window will pop up. Name the project **MyInitialsSuperShapes.**



3 Make sure the **Project Path** matches the image. If not, type in the path specified from **Step 1.**

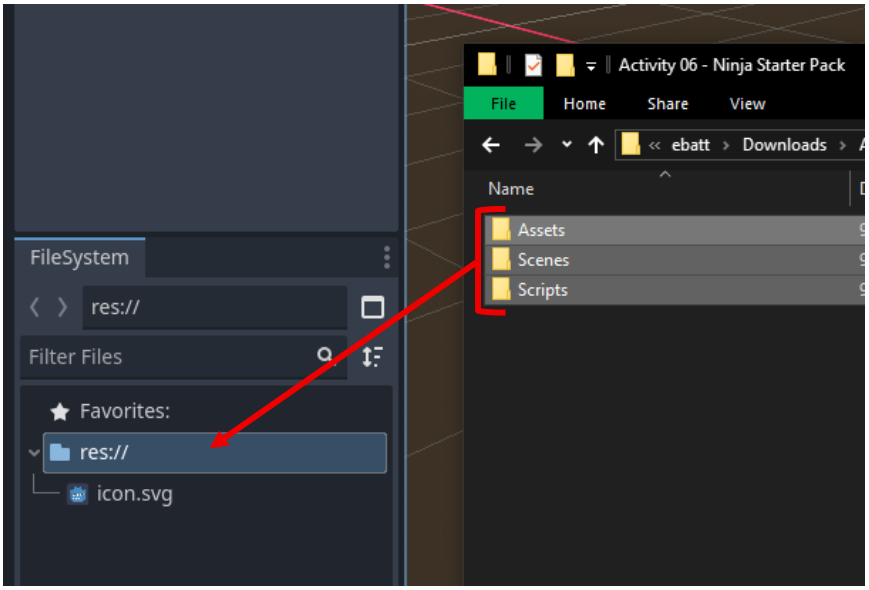
Check that **Create Folder** is turned on, and that the **Compatibility** mode for the renderer is being used. Then click **Create.**



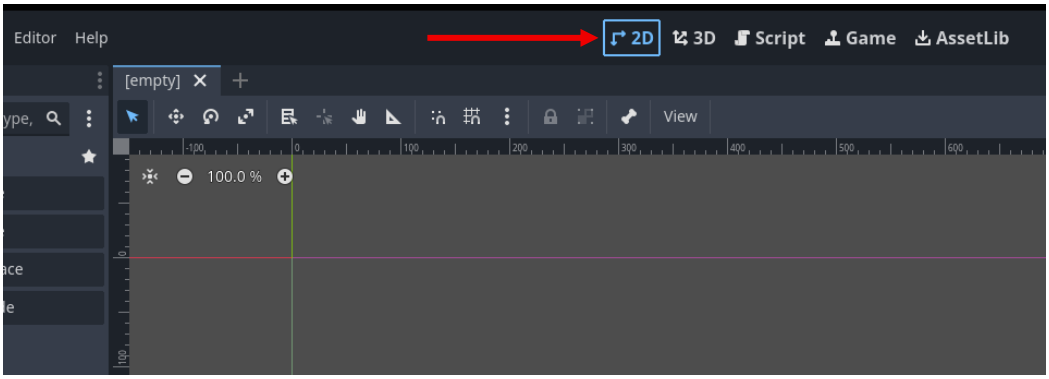
4

Don't create the **main scene** and **Main root** node just yet!

Extract **BB Activity 06 – Ninja Starter Pack.zip** and select all folders inside. Drag them into the **res://** folder in **FileSystem**.



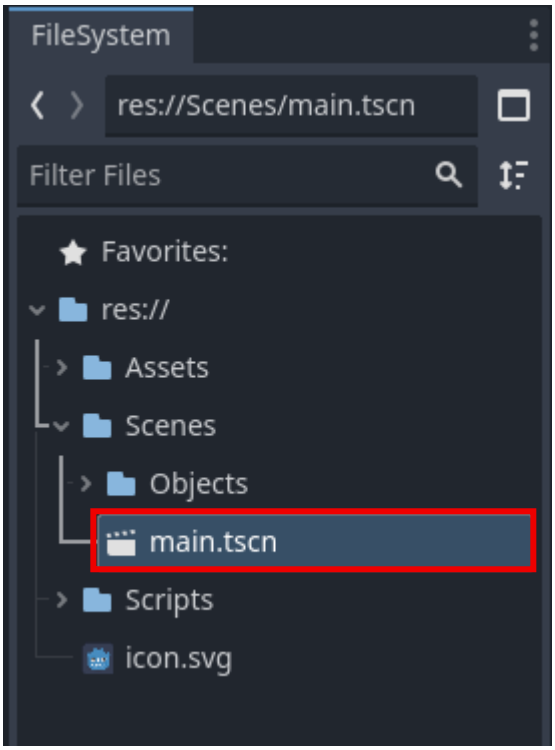
At the top center of the Godot editor, check that **2D** is selected.



Reminder:

Double-click on the folder icon from **Downloads.** Then **right-click** on the zip file and select **Extract All.** Press **CTRL + J** on the keyboard to reopen the **File Explorer.**

5 In **FileSystem**, navigate to **main.tscn** and **open it** by **double-clicking** it.

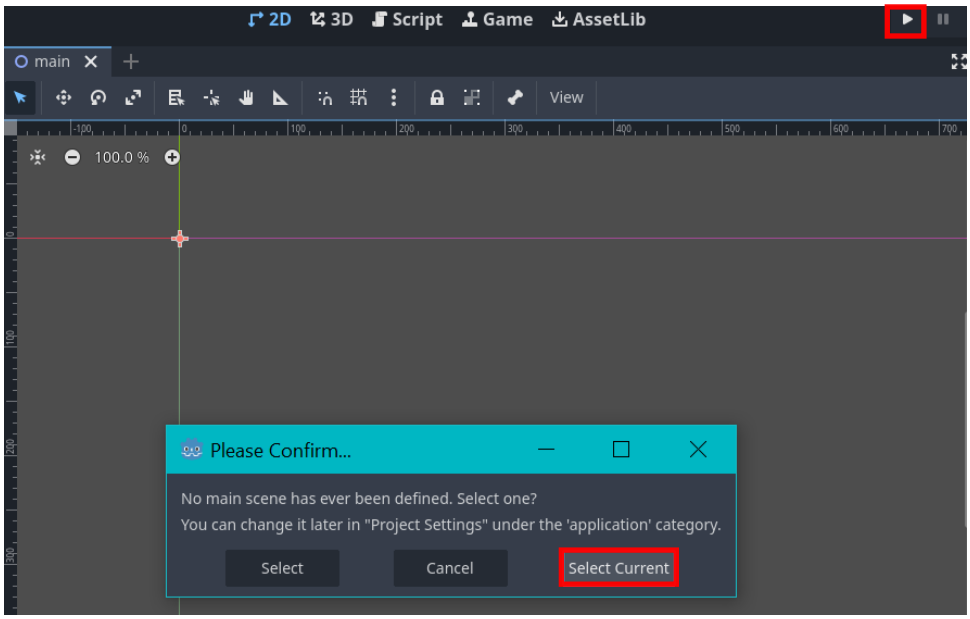


Reminder:

Click the arrows next to the folders to open them.

6 In the top right corner, click the **play** button to run the game.

Click **Select Current** to define the main scene, then close out of the playtest window once it appears.



Notice that the screen is now black; why might this be?

Pause for **Sensei Stop #1**!

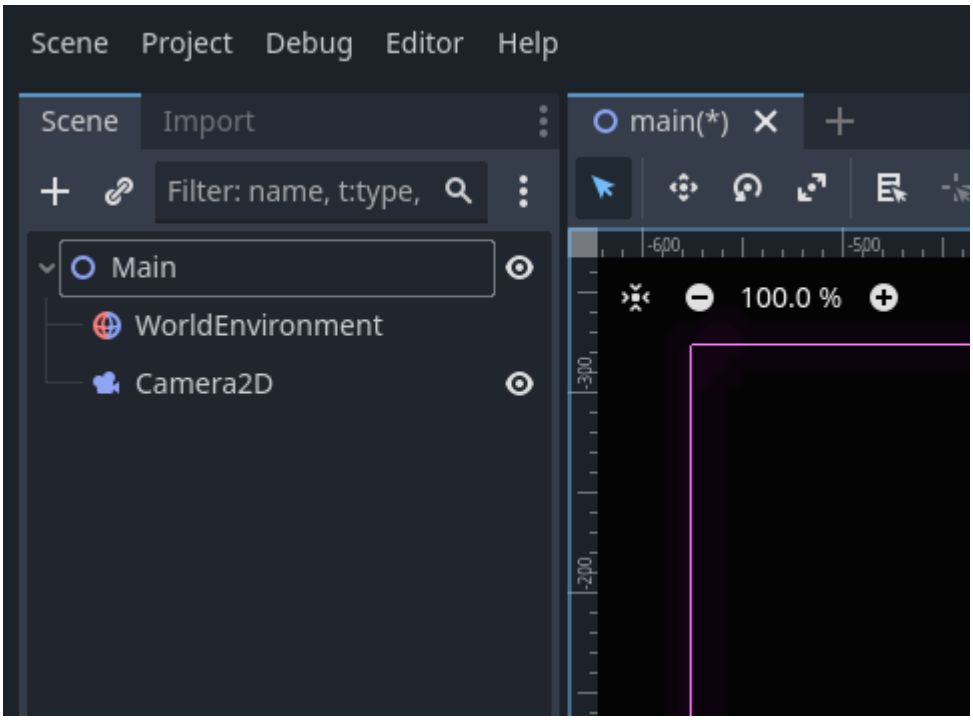
Check in with a Code Sensei before moving on. Make sure the **Ninja Starter Pack** and **main** scene were set up properly.

Reminder: Save your work!

7 In **Scene** notice the nodes that are already in the **main scene.**

The **WorldEnvironment** node is already set up. In this case, it is included to give the game a glowing effect and display a black background.

The **Camera2D** is the typical camera node that's been used in previous activities. By this point, you already have ample experience setting up Camera2D nodes, so it may be included in StarterPacks in future projects.



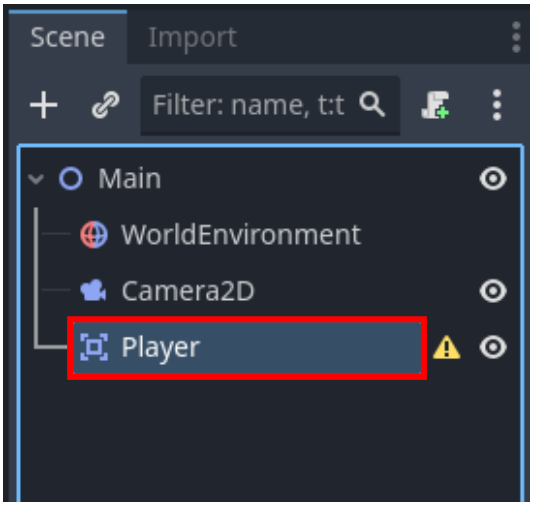
Reminder:

The **WorldEnvironment** node allows the lighting, effects, and background to change in a scene.

8 For this project, the **player scene** will be made from scratch. This means a sub-tree of nodes for the player will need to be added to the **main scene.**

Add an **Area2D** node as a child to the **Main root** node. Rename the node to **Player**.

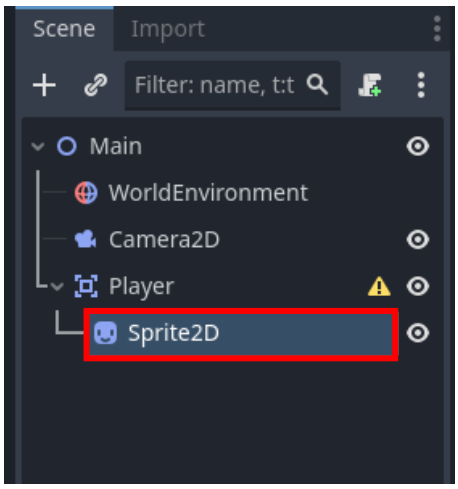
Ignore the hazard symbol for now, this will be fixed later.



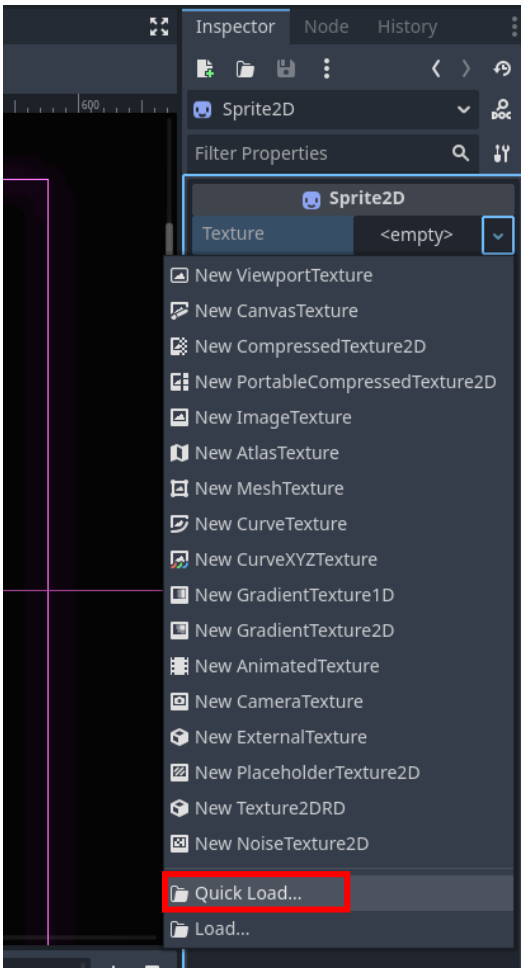
Reminder:

An **Area2D** node is a region of 2D space defined by its collision shape.

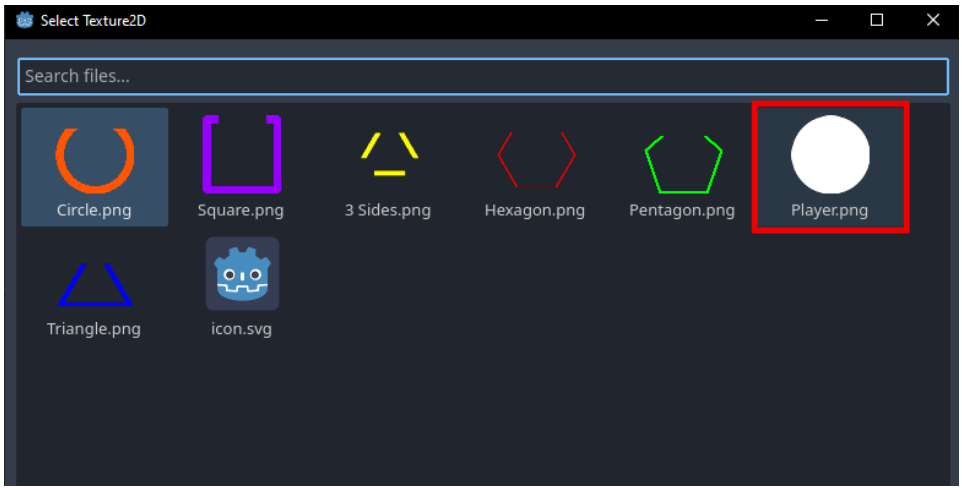
9 The player sprite still needs a texture. Add a **Sprite2D** node as a child to the **Player.**



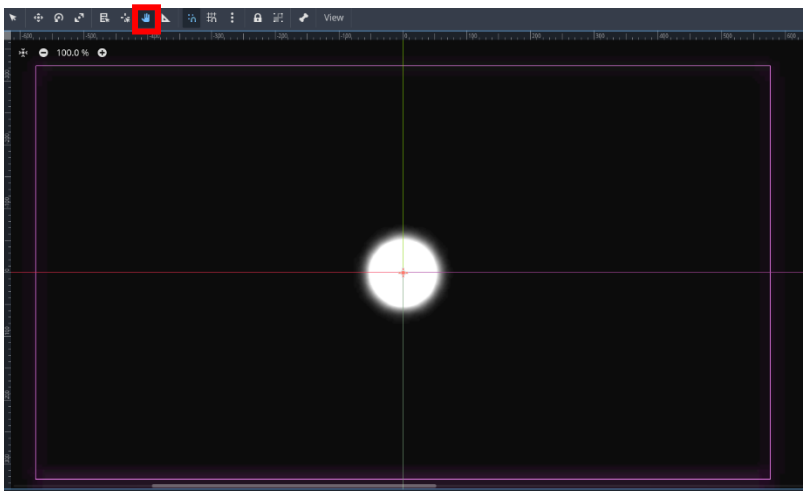
10 In the **Inspector,** click the dropdown arrow to open the **Texture** menu, then select **Quick Load.**



11 Double-click on the **Player.png** to select the **Texture2D** for the sprite.

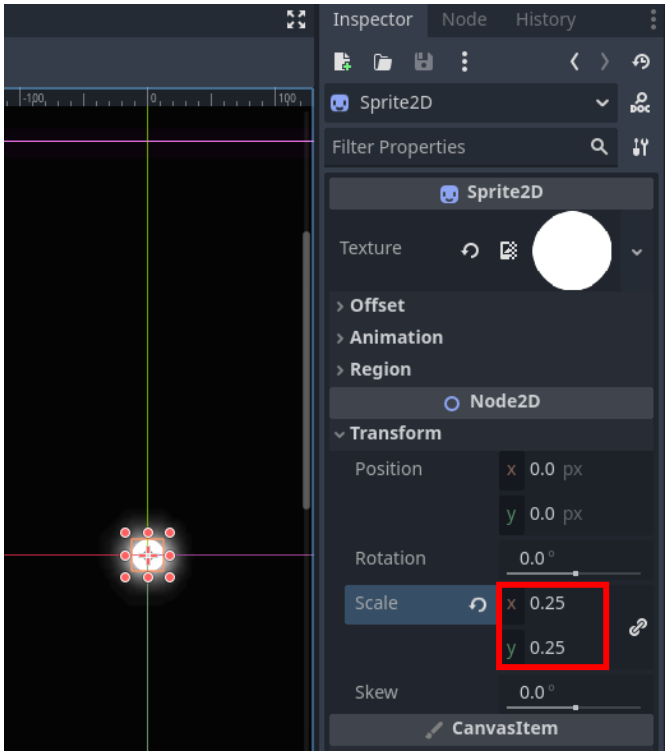


12 Find the Player sprite. To navigate the game window, use the mouse to **zoom out** and click on the **Pan** tool to move around.



The Player sprite seems a bit too big to be dodging shapes!

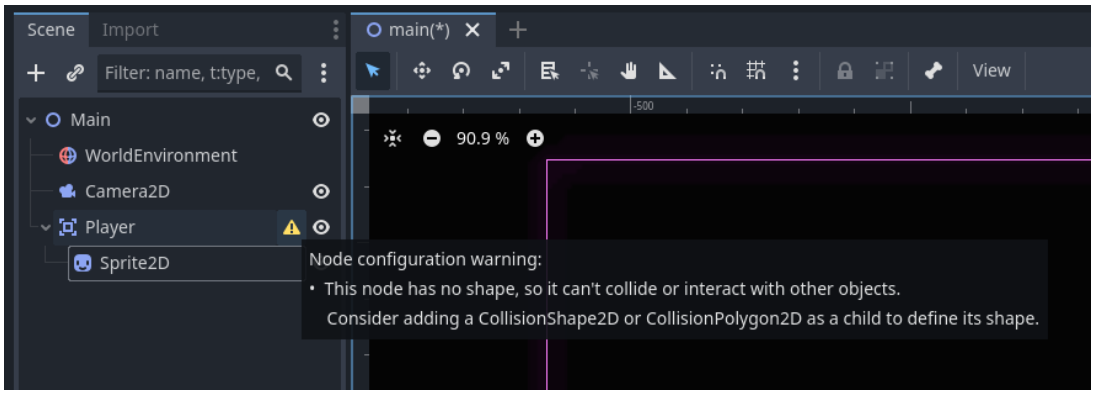
In the **Inspector** for **Sprite2D** under **Transform,** set the **scale** to **0.25** to make it just the right size. Either the **x** or the **y** value can be changed**.**



13

Fix the hazard symbol next to the **Player**.

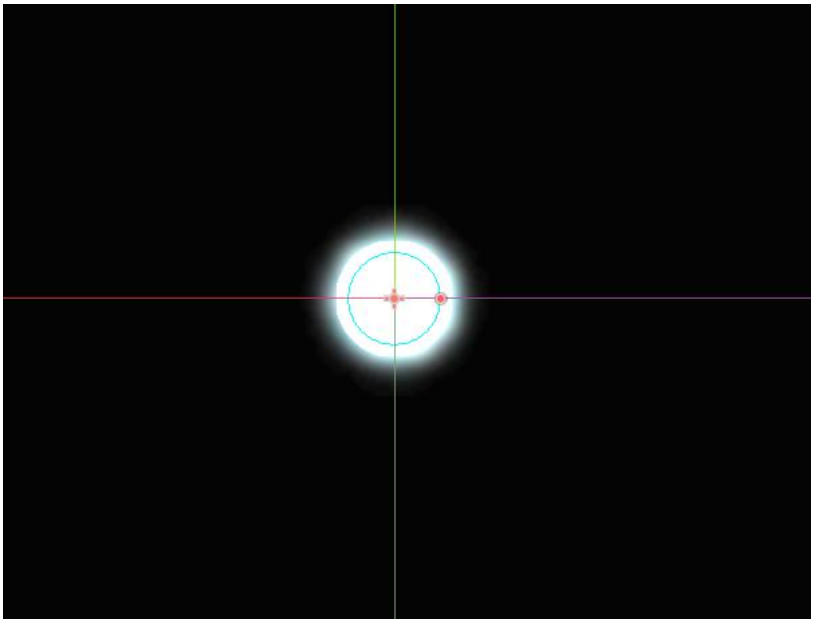
Add a **CollisionShape2D** node as a child to the player. Assign **CircleShape2D** as the shape in the **Inspector.**



14

Zoom in on the **Player** in the game window**.**

Notice the size of the **collision shape.** It should be a little bit smaller than the Player's **texture**. If not, adjust it to match the **Player's Sprite2D** as shown in the image.



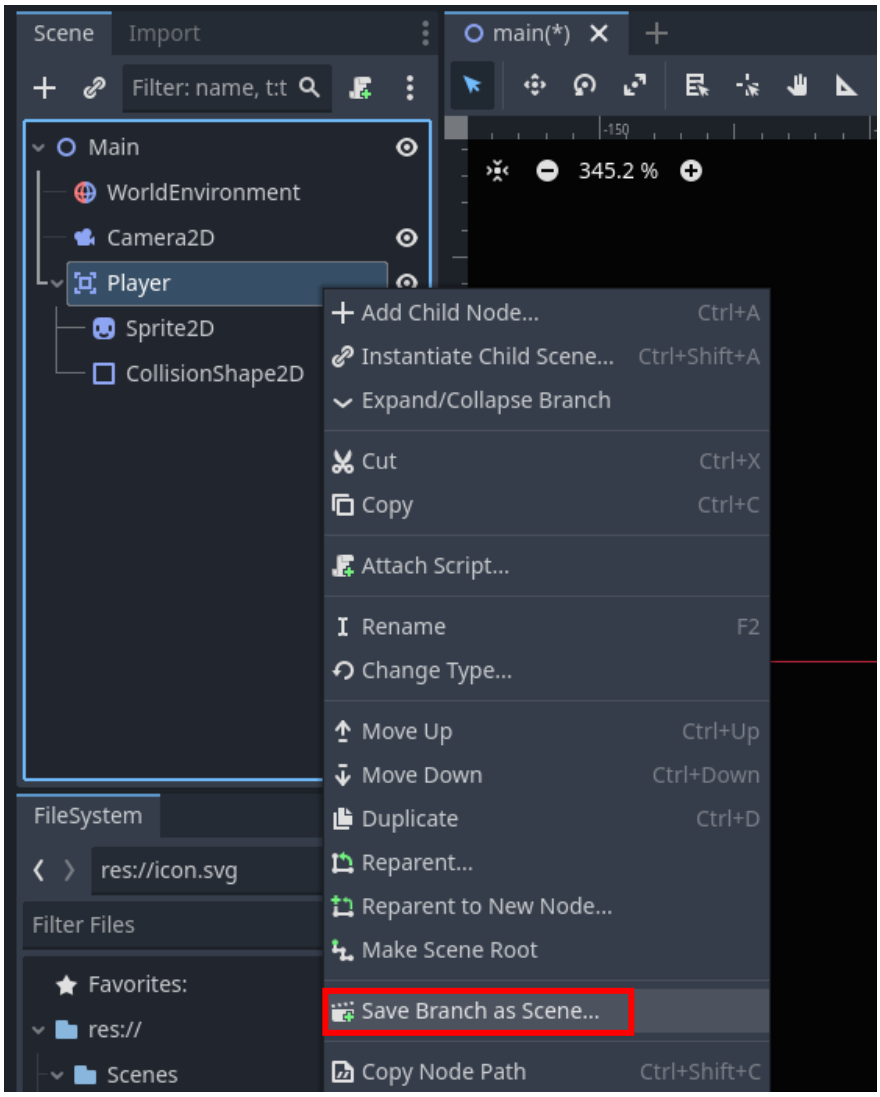
Reminder:

Collision shapes can be resized by **dragging** the red dot.

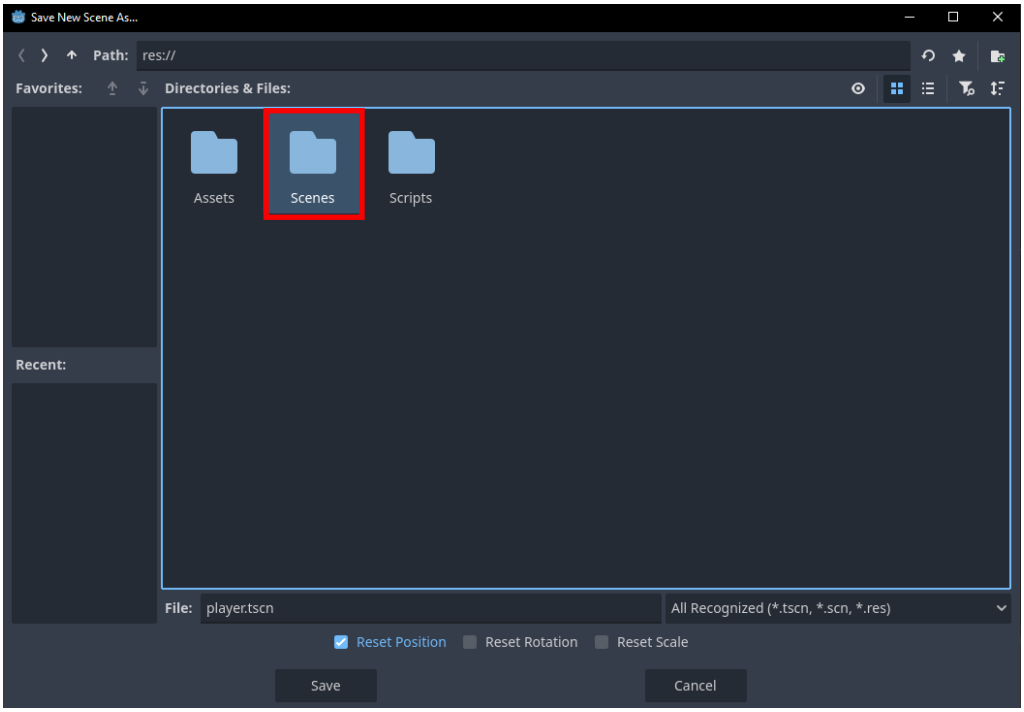
15 In the **Scene** menu, notice the sub-tree of nodes for the **Player**.

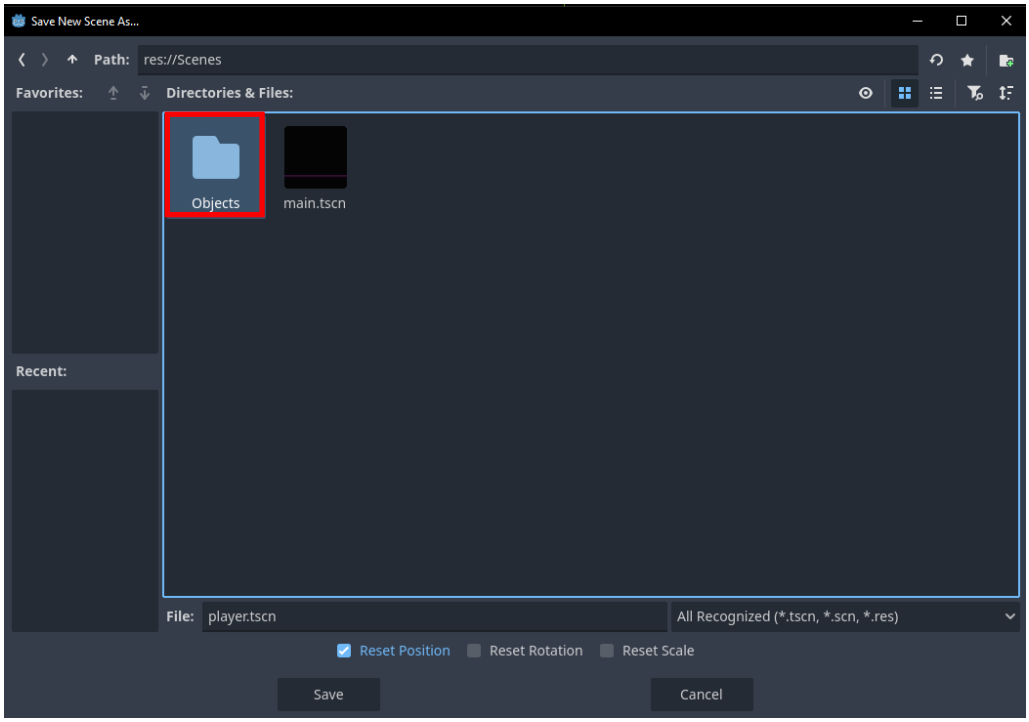
Time to create a **player scene!**

Right-click on the **Player** node and select **Save Branch as Scene.**



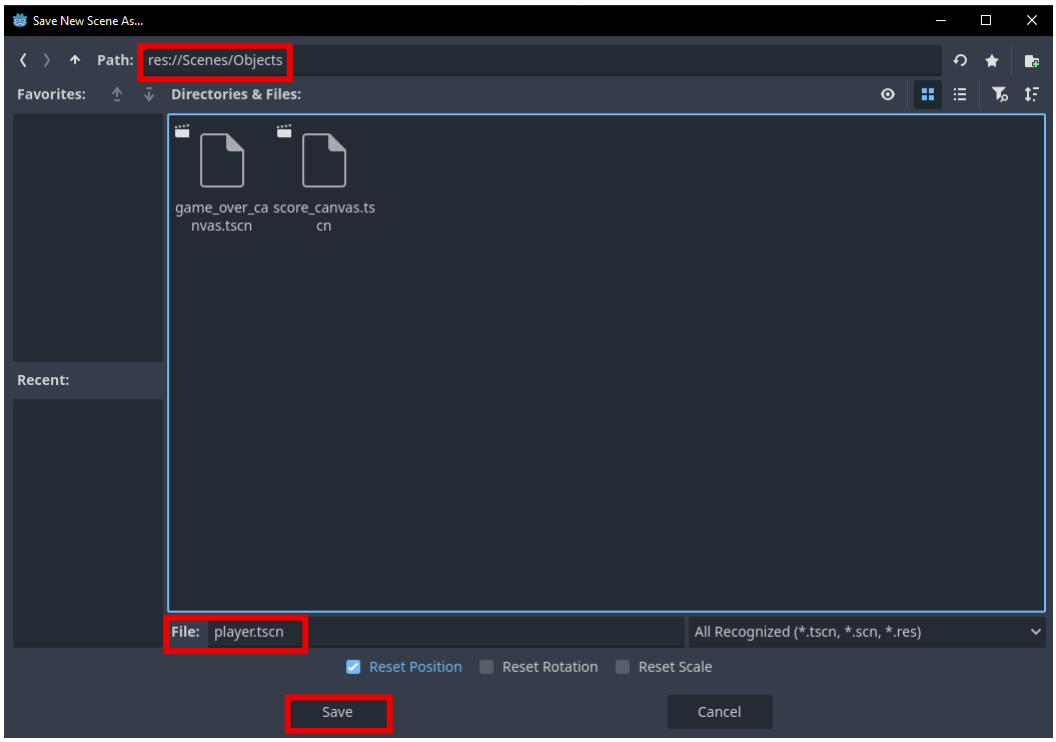
16 **Double-click** on the **Scenes** folder, then the **Objects** folder.





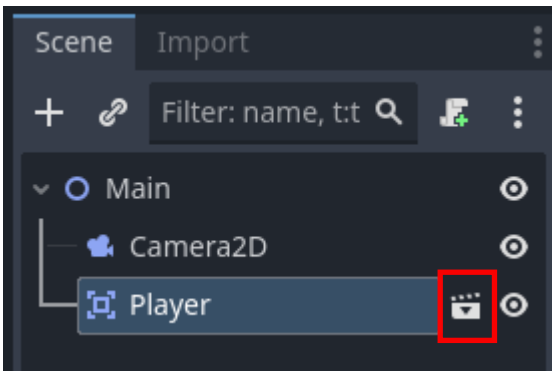
17 At the top of the window, check that the path is the same as shown in the image.

Check that the file name is named **player.tscn.** Once the path and file name are checked, click **save.**



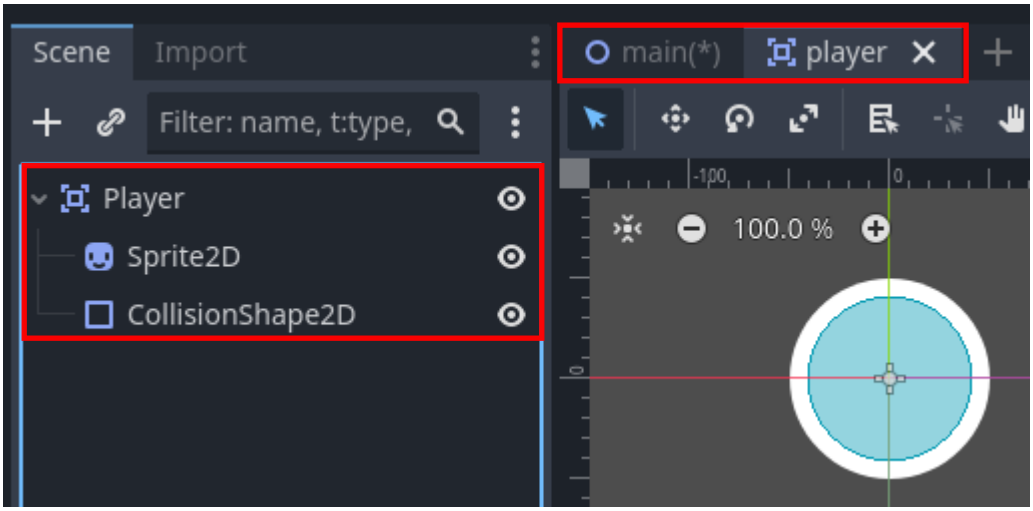
18 In the **Scene** menu, notice how the scene symbol now appears next to the **Player** node.

Click the **scene** icon.



19 Notice that the sub-tree of nodes for the **Player** now has its own **player scene**, which is separate from the **main scene**.

When viewing the **Player** scene, the glow seems to disappear. This is because the glow effect comes from the **WorldEnvironment** node which only exists in the **main scene**. But don't worry, the glow remains when the project is playtested since the **main** scene was selected as the "main scene" for playtests!



Pause for **Sensei Stop #2**!

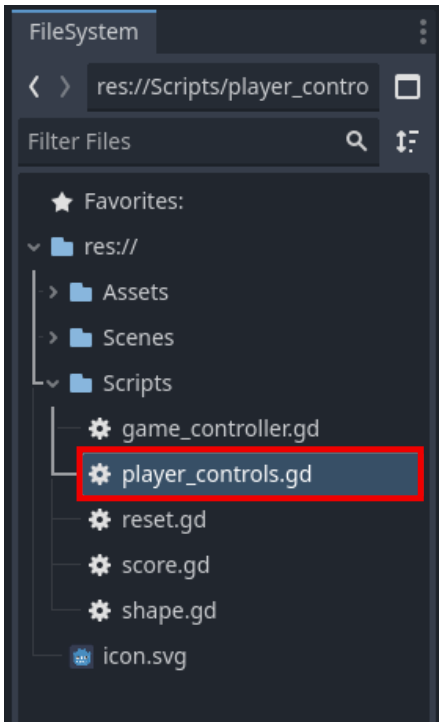
Check in with a Code Sensei before moving on. Make sure the **Player** scene is set up properly.

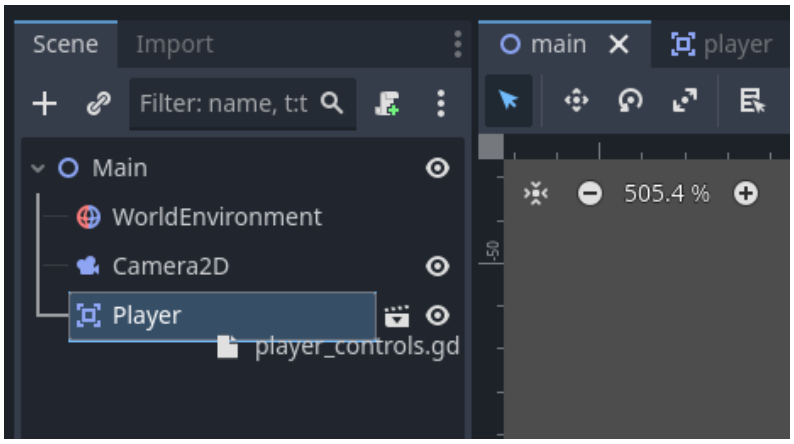
Reminder: Save your work!

20 Navigate to the **main** scene.

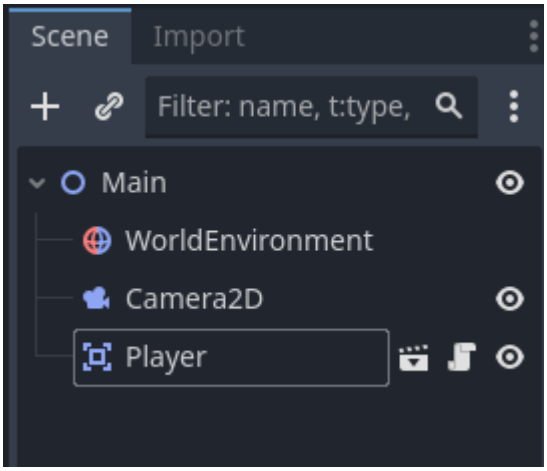
In **FileSystem**, find **player_controls.gd** in the **Scripts** folder.

Drag and drop the script onto **Player** in **Scene.**





21 Open the script that is attached to the **Player** node.



22 Review the code that is already in the script.

Find the `move_speed` variable. Notice it uses `@export` so that its value can be tinkered with in the **Inspector.** Underneath, notice the `movement` variable which will help compute which direction to rotate the player.

Notice the `_ready()` method, which resets the game's time scale to `1`. This will help when the player wants to restart the game.

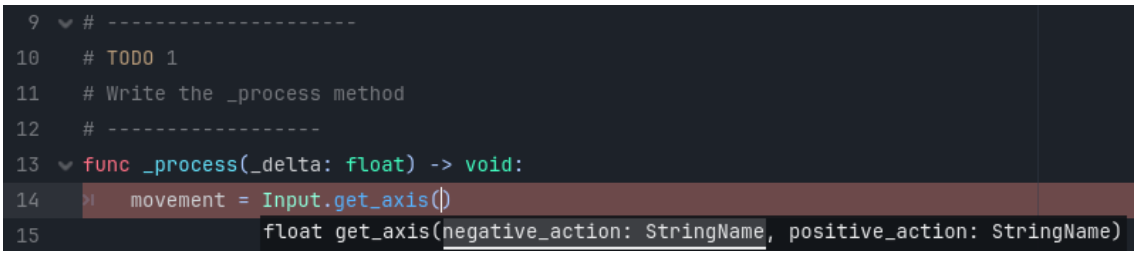
```
1   extends Area2D
2
3   @export var move_speed: float = 10
4   var movement: float = 0
5
6   func _ready():
7       Engine.time_scale = 1
8
9   # ---------------------
10  # TODO 1
11  # Write the _process method
12  # ------------------
13
```

23

Add code to control the player via user input.

Under **TODO 1,** define a `_process()` method with a `_delta` parameter**.** Inside the method, set the player's `movement` variable to `Input.get_axis()`**.**

Inside the parentheses of the `Input.get_axis()` method, type the names of the input actions for negative and positive directions. Separate the parameters with a comma.



`Input.get_axis()`: checks if two inputs are being pressed and returns a number to show the direction of that input.

Parameters:

1. `negative_action` ***(String)****:* the name of the input action for negative direction **("ui_left")**
2. `positive_action` ***(String)****:* the name of the input action for positive direction **("ui_right")**

Returns ***(float):*** number between -1 and 1

24 Make sure the `_process()` method matches the code in the image shown.

This will set the movement variable to a **float** of either **-1, 0, or 1** depending on what button the user is pressing.

- If the **left direction button** is inputted, then movement is **-1**.
- If the **right direction button** is inputted, then movement is **1**.
- If **nothing** is inputted, then movement is **0.**

```
9  # ---------------------
10 # TODO 1
11 # Write the _process method
12 # ------------------
13 func _process(_delta: float) -> void:
14     movement = Input.get_axis("ui_left", "ui_right")
15
```

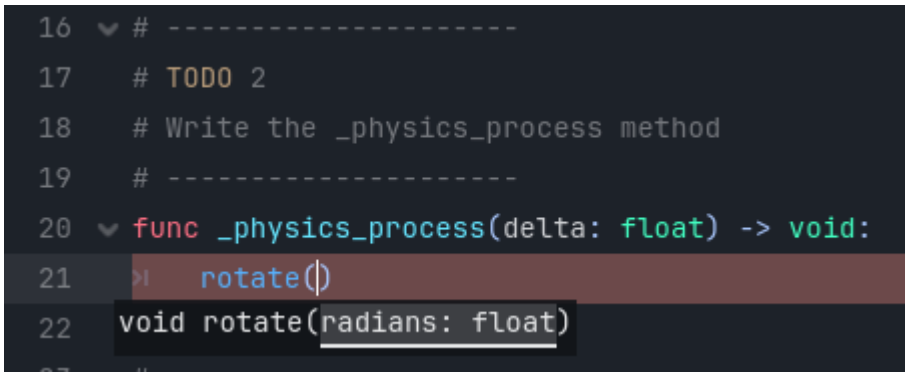
25

Time to code the player's movement.

Under **TODO 2,** define the `_physics_process()` method with a `delta` parameter**.**

Inside this method, call the `rotate()` method. This built-in Godot method has one **parameter** and returns **void** (nothing) as shown.

Inside the parentheses of the `rotate()` method, multiply the values of the `movement`**,** `delta`**,** and `move_speed` variables together using the multiply `*` operator.



`rotate()`: part of the Node class, rotates a node by the given angle in radians around its local origin

Parameters:

1. `radians` ***(float)*****:** the angle to rotate in radians. Positive values rotate clockwise; negative values rotate counterclockwise.

Returns *(void)*: nothing

26 Check the `_physics_process()` method and update as needed.

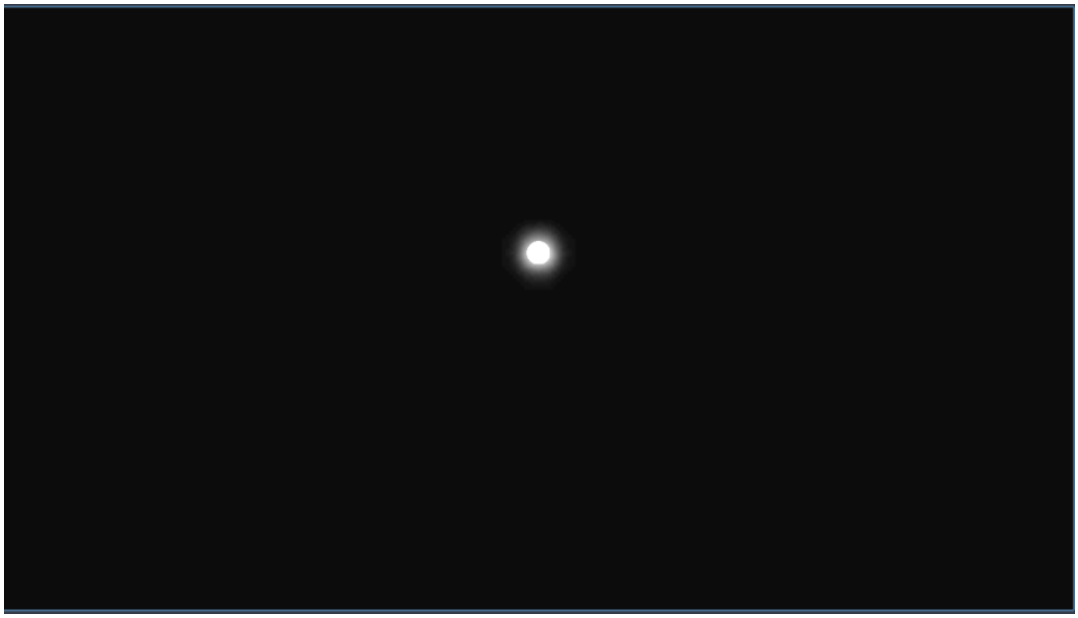
This rotates the **Player** over time **(`delta`)** based on input direction (**movement)**, and the rotation speed (**`move_speed`).**

Now when the **right direction button** is pressed, movement is set to **1**, causing the **Player** to rotate **clockwise.** When the **left direction button** is pressed, movement is set to **-1**, causing the **Player** to rotate **counterclockwise.**

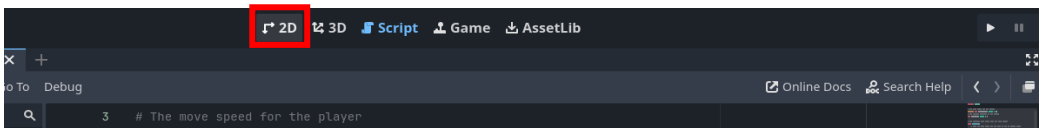
```
16  # ---------------------
17  # TODO 2
18  # Write the _physics_process method
19  # ---------------------
20  func _physics_process(delta: float) -> void:
21      rotate(movement * move_speed * delta)
22
```

27 Playtest the project and press the left and right direction buttons to move the player.

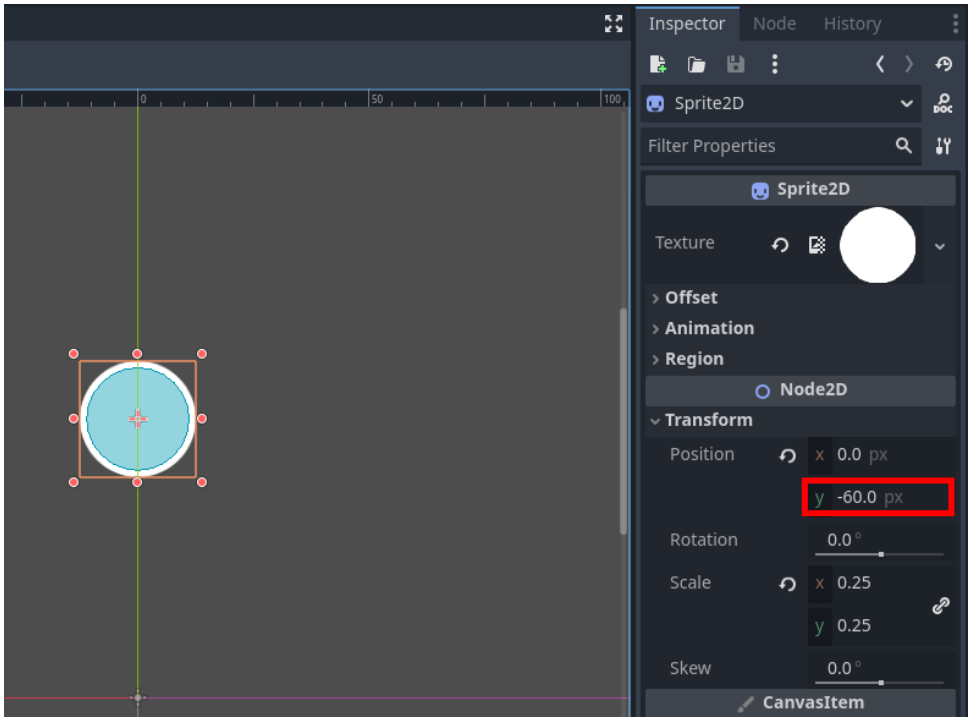
Hmmm, the **Player** seems to not be moving at all! Why might this be?



28 At the top of the **game window,** switch to **2D.** In the **player scene,** select the Player's **Sprite2D** child node.



In the **Inspector** change the **y value** to **-60.**



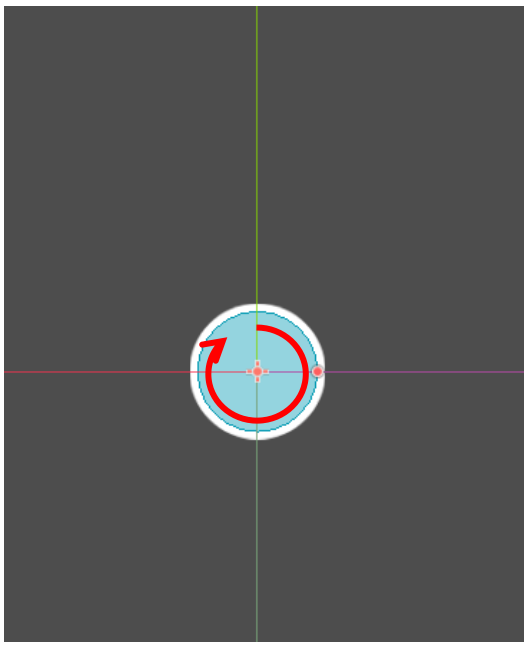
Repeat this same step for the **CollisionShape2D** node and ensure the **same y value** is used for both nodes.

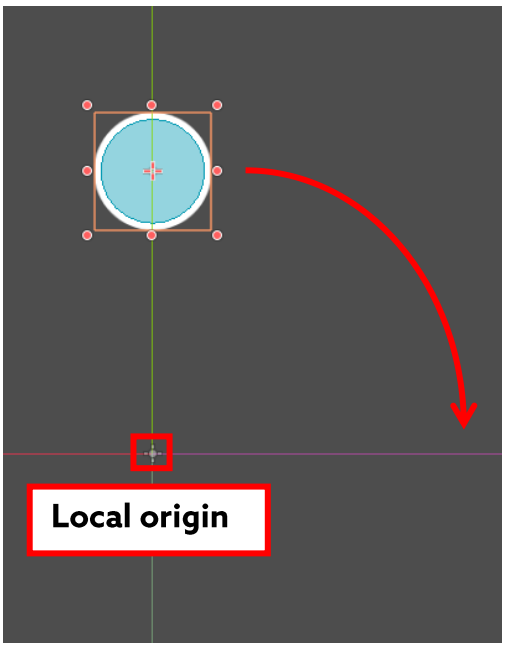
29 Playtest the project again and notice the Player now rotates based on the input!

Before when the Player's position wasn't changed and `rotate()` was called, it caused the Player to spin in place**.**

When the Player is moved *away* from its **local origin,** it's able to rotate in a **circle path** around the **local origin.**

The **local origin** is the original position that an object is in. Always be sure to **offset** an object by moving its position away from the **local origin** if using the `rotate()` **method.**





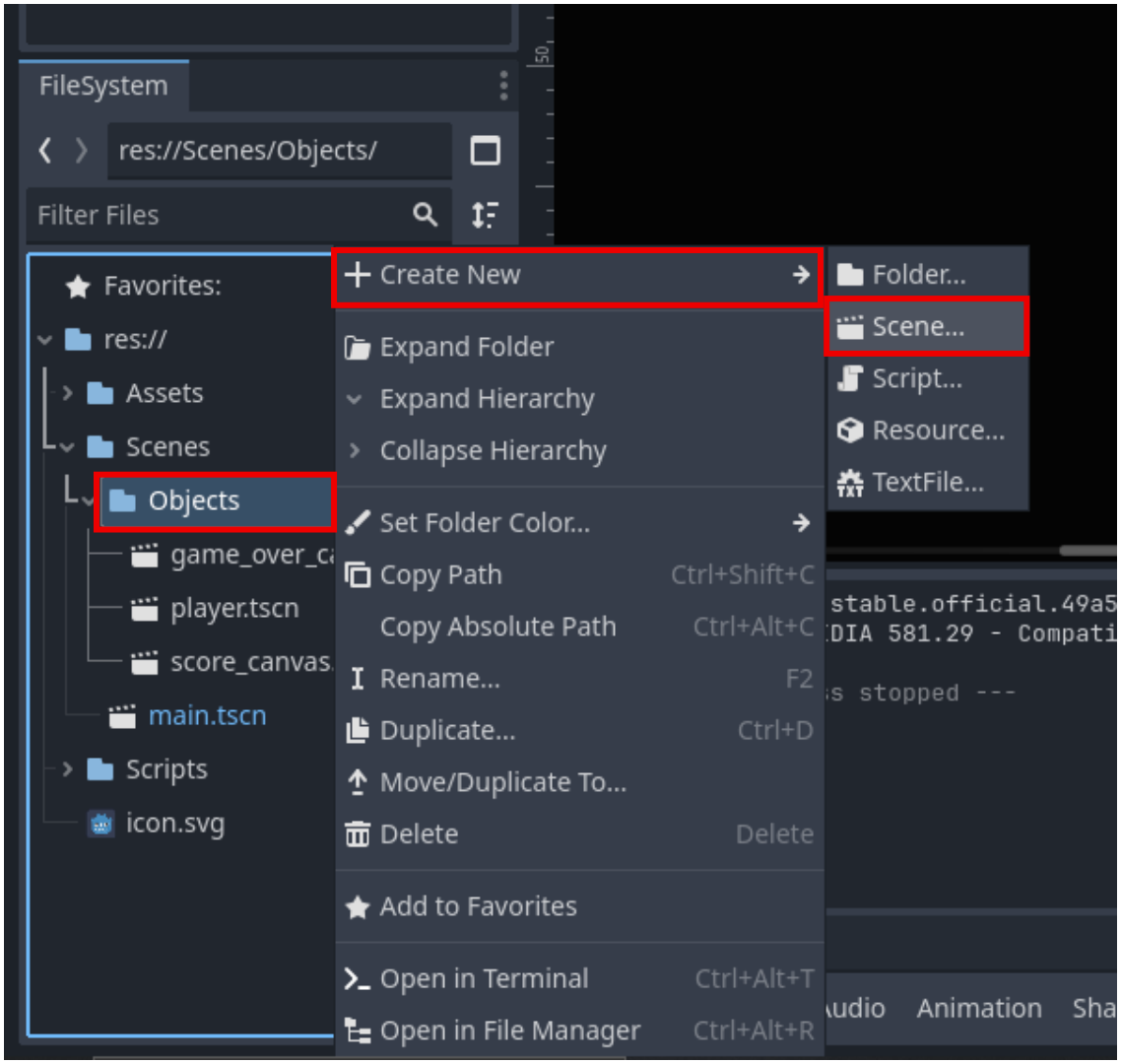
Pause for **Sensei Stop #3**!

Check in with a Code Sensei before moving on. Make sure the Player moves in a rotation before moving on.

Reminder: Save your work!

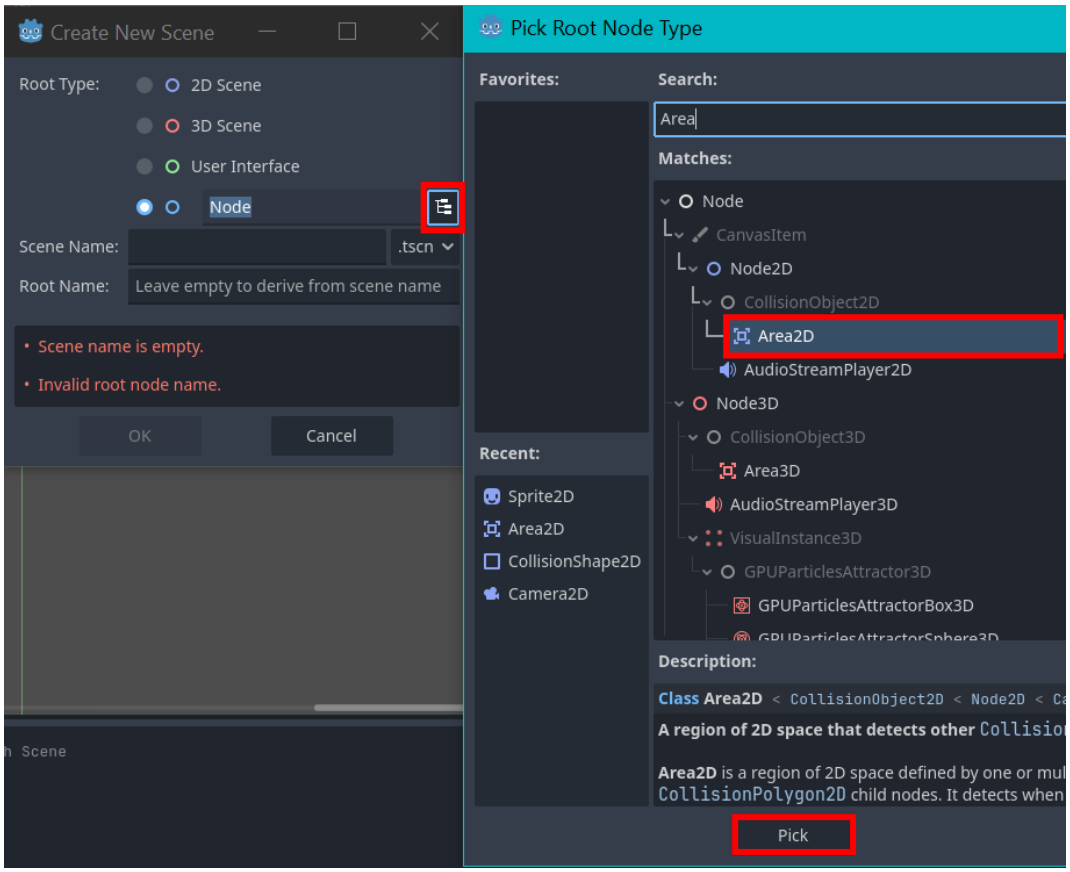
30 Time to add some shapes to dodge! The first shape that will be created is the **3 sides shape.**

Another way to create a new scene is in the **FileSystem.** Right-click on the **Objects** folder, hover over **+ Create New** then select **Scene.**

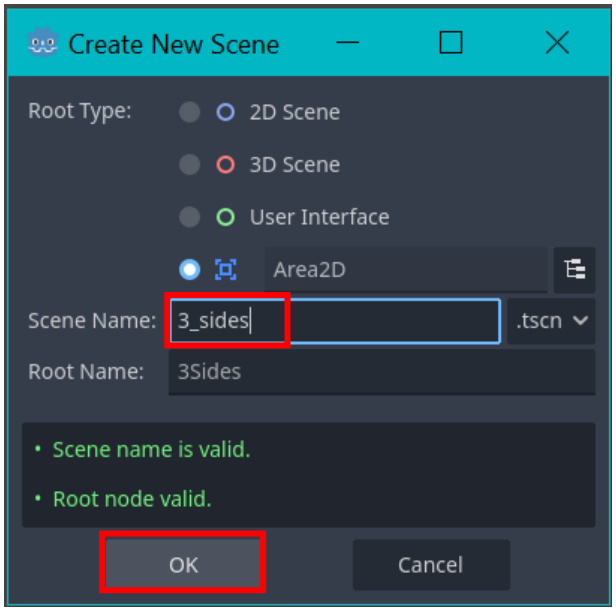


31 Select the **branch** symbol next to **Node** to change the **Root Type.**

In the **Pick Root Node Type** window, select the **Area2D** node and click **Pick.**

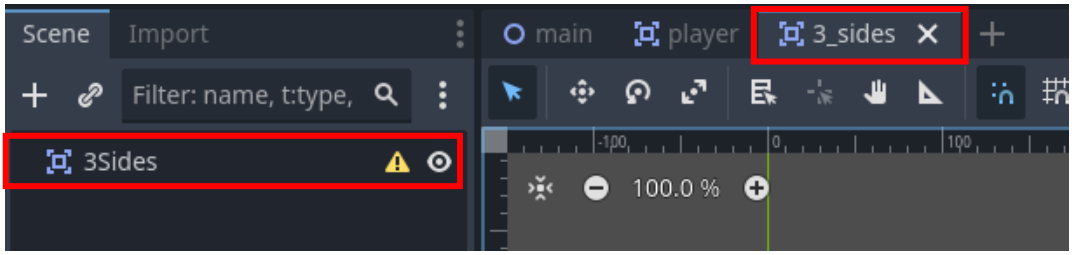


32 Rename the **Scene Name** to **3_sides** then select **OK.**



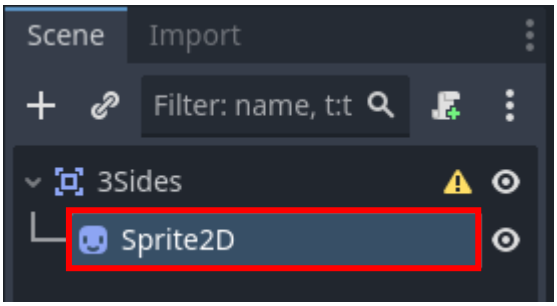
33 Notice the new scene that was created and how it's not a sub-tree of the **main scene** but an entirely **different** tree!

Ignore the hazard symbol for now, this will be fixed later.

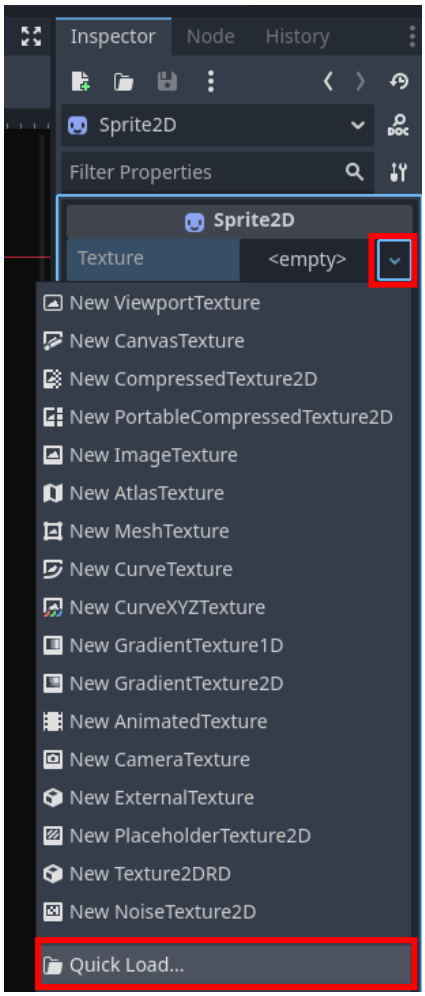


34 The **3Sides** sprite needs a texture.

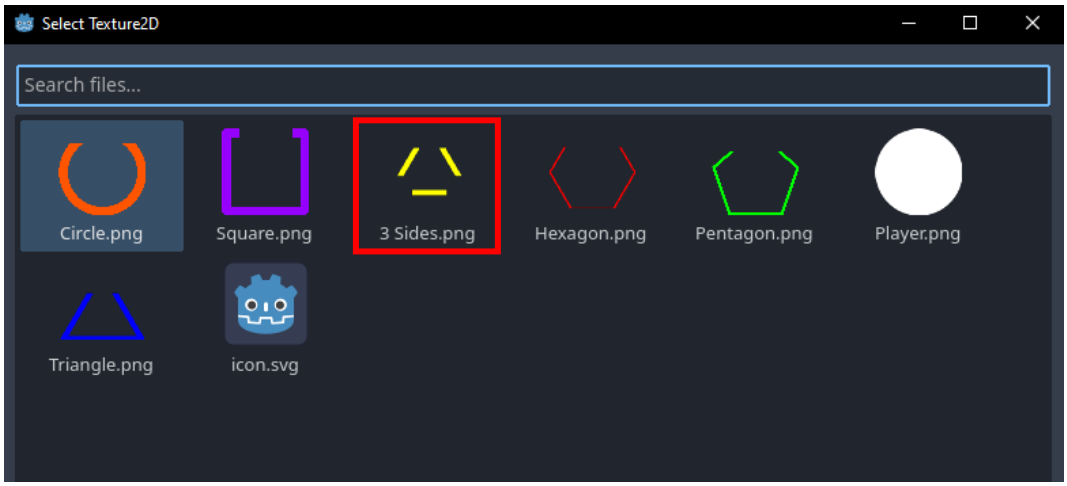
Add a **Sprite2D** node as a child to the **3Sides root.**



35 In the **Inspector,** click the **dropdown arrow** and select **Quick Load.**

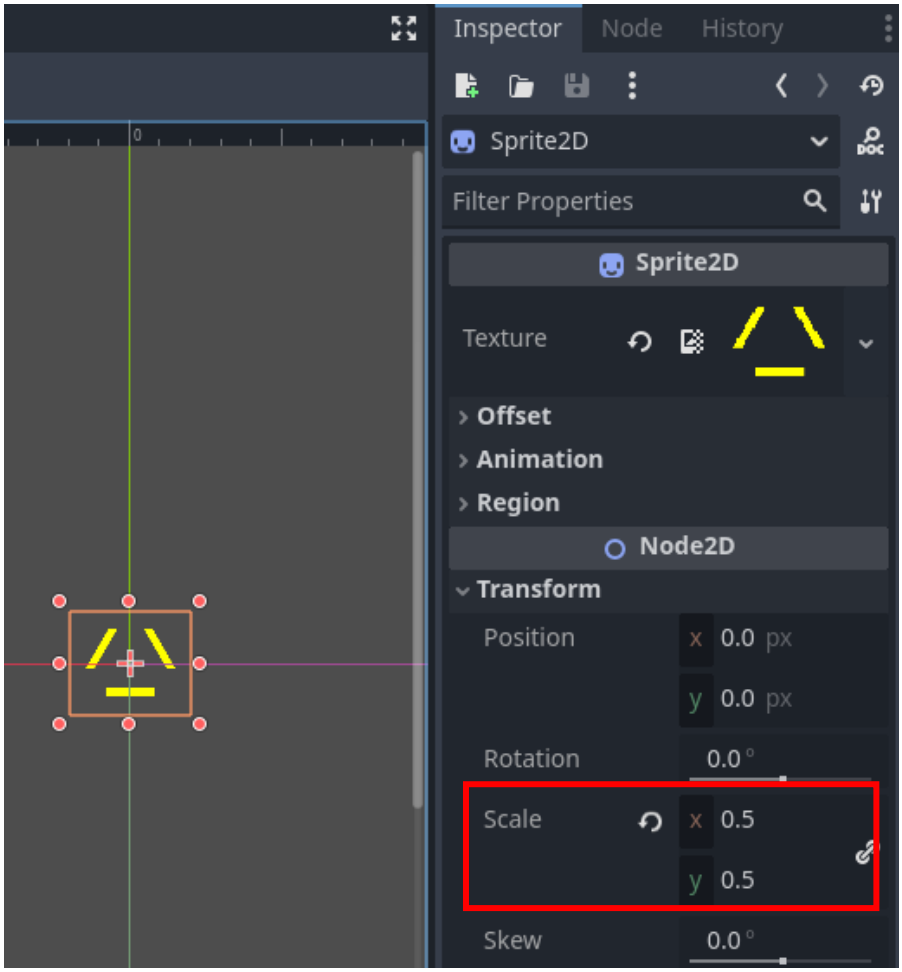


36 Double-click on the **3 Sides.png** to select the **Texture** for the sprite.



37 The image file is originally too big for the game to play properly.

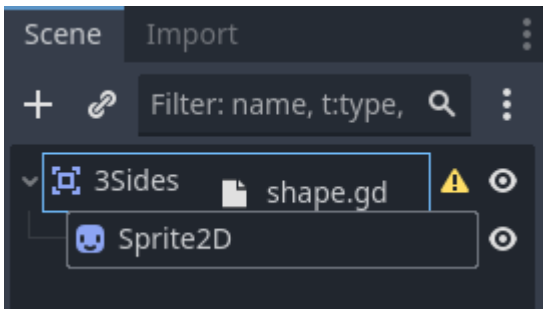
In the **Inspector**, set the **Scale** to **0.5** to make the shape smaller.



38 In **FileSystem**, find **shape.gd** in the **Scripts** folder.

Drag and drop the script onto the **3Sides** root node.

Continue to ignore the hazard symbol for now, as this will be fixed later.



39

Create the scenes for the rest of the shapes!

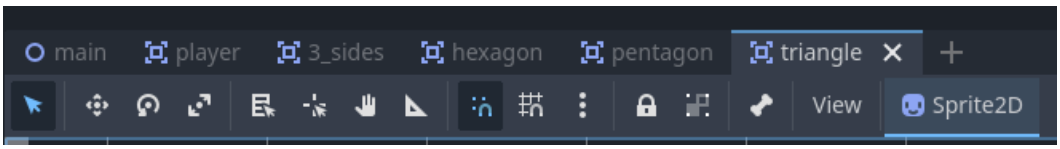
Refer to the previous steps for additional guidance.

Set the scales for the other scenes' **Sprite2Ds** to the following:

- Hexagon: **0.2**
- Pentagon: **0.4**
- Triangle: **0.5**

Ignore the hazard symbols for now while creating these scenes.

Make sure to **save** each of the new scenes by pressing **CTRL + S**!

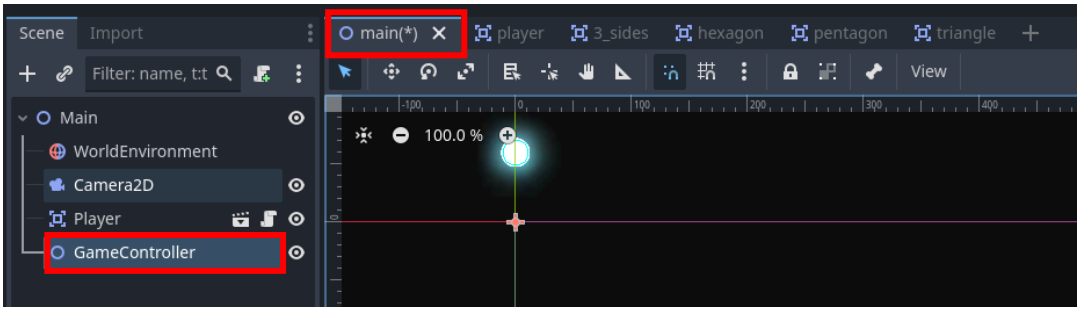


Pause for **Sensei Stop #4**!

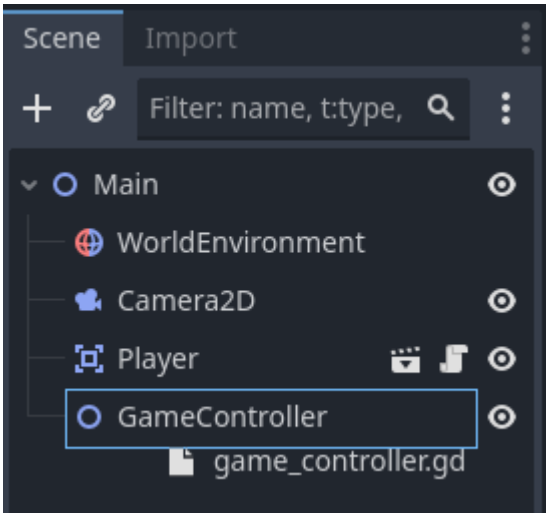
Check in with a Code Sensei before moving on. Make sure all shape scenes were created correctly before moving on.

Reminder: Save your work!

40 Navigate to the **main scene** and add a new **Node2D** as a child to the **Main** root node**.** Rename it **GameController**.



41 In **FileSystem,** find **game_controller.gd** in the **Scripts** folder and drag and drop it onto the **GameController** node.



42 Open the **game_controller.gd** script. Review the starting code that is already present.

Notice the `shape_scenes` variable, which is an array that will hold all the shapes. Declaring an array in Godot is a bit different from JavaScript. In Godot, arrays are declared as regular variables and can be explicitly set to an Array type if needed.

The `spawn_delay` variable will store the number of seconds it takes before a shape begins to spawn and the `spawn_time` variable will store the number of seconds between shapes spawning.

Find the `_ready()` method. So far it uses Godot's built in `randomize()` function and the `spawn()` function to pick a random shape to spawn.

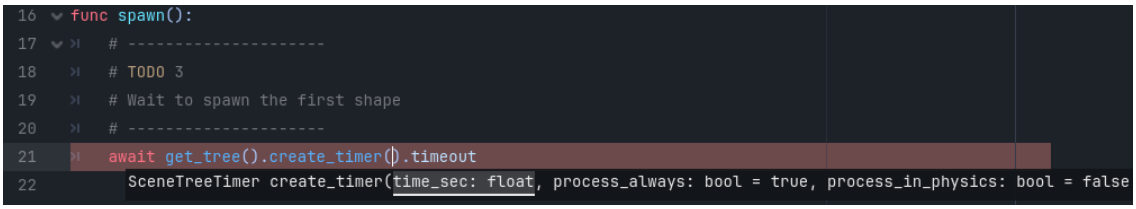
What else might be added to this script to spawn in the different shapes?

```
1   extends Node2D
2
3   @export var shape_scenes: Array[PackedScene]
4   @export var spawn_delay: float = 2
5   @export var spawn_time: float = 3
6   @export var game_over_canvas: CanvasLayer
7
8   var repeating: bool = true
9
10  # The function that runs when the game is loaded
11  func _ready():
12      randomize()
13      spawn()
14
15  # The function acts like a coroutine, running to make shapes continuously spawn
16  func spawn():
17      # ---------------------
18      # TODO 4
19      # Wait to spawn the first shape
20      # ---------------------
21      pass
22
```

43

Scroll down to the `spawn()` function**.**

Under **TODO 3** in the `spawn()` function, replace `pass` with a new line of code. Use the keyword `await` to begin adding a **Timer** node**.** Use the code completion to add the `get_tree()` and `create_timer()` methods. Use the `spawn_delay` variable as the parameter to the `create_timer()` method. Then use the dot `.` operator to select `timeout` from the code completion.



A `get_tree()` method must be used when a script needs to interact with the entire game, not just the node it's attached to. It gives access to the **SceneTree** which manages adding or removing nodes, loading or changing scenes, and pausing or resuming the game.

The **SceneTree** is also the **only Node** that allows the use of the convenience method `create_timer()`**.**

> `create_timer()`: part of the SceneTree class, creates and starts a temporary **Timer** node that runs for the specified number of seconds. It is added to the game and is removed after it finishes.
>
> **Parameters:**
>
> 1. `time_sec` ***(float)*****:** duration of the timer in seconds
> 2. `pause_mode_process` ***(bool, optional)*****:** if **true,** timer will pause when the game is paused
>
> **Returns *(node)*:** A temporary **Timer** node

44 Create a `while` loop using the `repeating` variable declared earlier in the script.

The loop will run while **repeating** is **true.** This means new shapes will keep spawning until the game is over, since **repeating** is set to **false** in the **game_over()** function.

```
16  func spawn():
17      # --------------------
18      # TODO 3
19      # Wait to spawn the first shape
20      # --------------------
21      await get_tree().create_timer(spawn_delay).timeout
22      while repeating:
23          # --------------------
24          # TODO 4
25          # Instantiate a random shape scene, then wait until spawning the next shape
26          # --------------------
27
```

45 Inside the `while` loop, under **TODO 4,** instantiate a random shape scene from the `shape_scenes` array.

Declare a `random_int` variable as type `int` and set it to `randi_range()`.

```
22  while repeating:
23      # --------------------
24      # TODO 4
25      # Instantiate a random shape scene, then wait until spawning the next shape
26      # --------------------
27      var random_int: int = randi_range()
28                            int randi_range(from: int, to: int)
```

`randi_range()` is the same as MakeCode's **randint(min, max)** function.

```
1   let shapeScenes: number[] = []
2
3   let randomInt = randint(0, shapeScenes.length-1)
```

Set the minimum value to 0 and the maximum value to the last index in the `shape_scenes` array. Use the `.size()` function to get the `length` of the array.

Notice how this is done in MakeCode and try to replicate it in Godot. What should the maximum value in the range be set to?

> `randi_range()`: generates a random integer between the given from and to values including both ends.
>
> **Parameters:**
>
> 1. `from` ***(int)*****:** the minimum value in the range
> 2. `to` ***(int)*****:** the maximum value in the range
>
> **Returns** ***(int)*****:** A random number within the range.

46 On the next line, use the `add_child()` method to add one of the shapes to the scene.

Inside the method, use the brackets `[ ]` to access a random scene from the `shape_scenes` array using the `random_int` variable, then call `.instantiate()` to create an instance of that scene.

```
23    # --------------------
24    # TODO 4
25    # Instantiate a random shape scene, then wait until spawning the next shape
26    # --------------------
27    var random_int: int = randi_range(0, shape_scenes.size()-1)
28    add_child()
29  void add_child(node: Node, force_readable_name: bool = false, internal: Node.InternalMode = 0)
```

47 Create another **Timer** node and pass the `spawn_time` variable as the parameter for the `create_timer()` method.

Copy **(Ctrl + C)** the code highlighted and paste **(Ctrl + V)** it on the line underneath the **TODO 5.** Remember, the **parameter** must be changed to `spawn_time` and **NOT** `spawn_delay`**.**

Make sure to copy this line *inside* the `while` loop by indenting with **tab** on the keyboard

```
16  func spawn():
17      # --------------------
18      # TODO 3
19      # Wait to spawn the first shape
20      # --------------------
21      await get_tree().create_timer(spawn_delay).timeout
22      while repeating:
23          # --------------------
24          # TODO 4
25          # Instantiate a random shape scene, then wait until spawning the next shape
26          # --------------------
27          var random_int: int = randi_range(0, shape_scenes.size()-1)
28          add_child(shape_scenes[random_int].instantiate())
29          # --------------------
30          # TODO 5
31          # Wait until spawning the next shape
32          # --------------------
33
```

Pause for **Sensei Stop #5**!

Check in with a Code Sensei before moving on. Make sure the **game_controller.gd** script was coded correctly.

Reminder: Save your work!

48

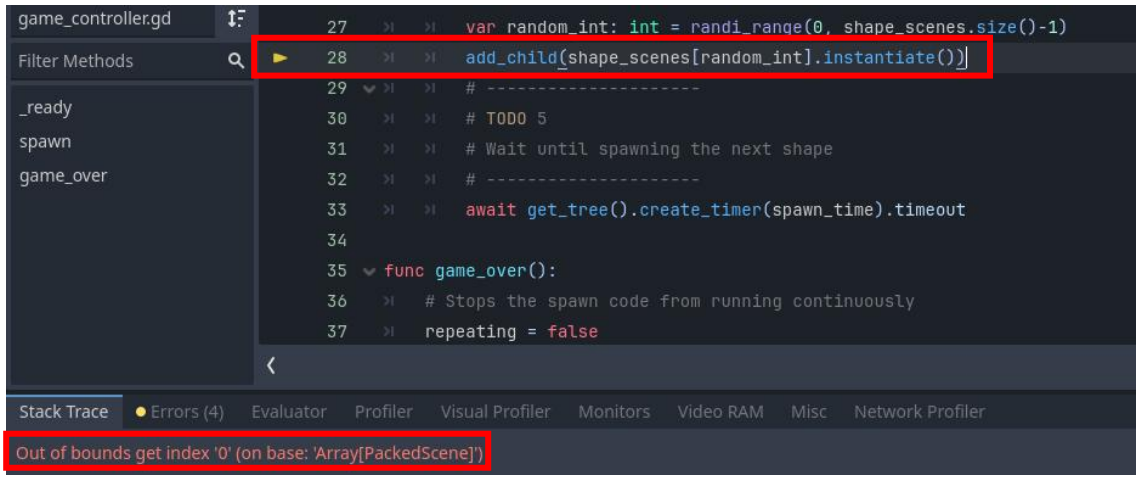
Playtest the project to see what happens.

Oh no, the game crashes!

Notice the **Out of Bounds Error** in the Debugger and the **yellow arrow** indicating where the error happened.

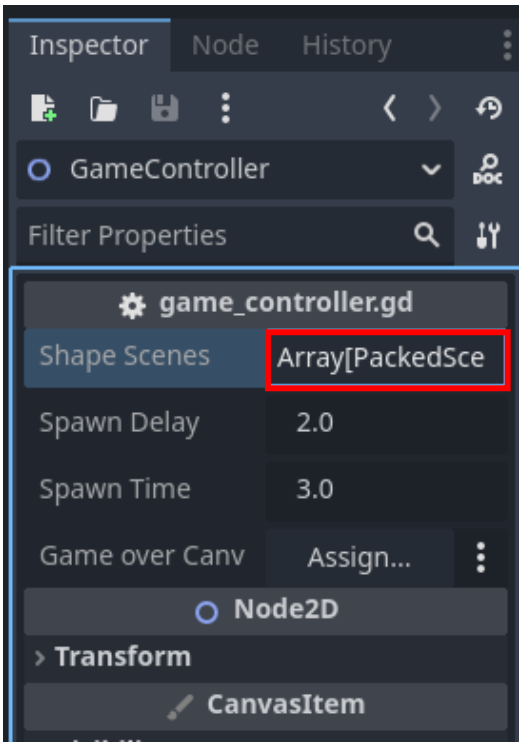
Why might this error be occurring?

Think back to MakeCode Arcade, what happens when a loop iterates through an element that has an index larger than the length of the array?



49 This happens when the code iterates through an index of an array that doesn't exist yet. In this case, let's check the `shape_scenes` array.

In the **Inspector** for the **GameController** node**,** click on **Array[PackedScene]** in **Shape Scenes.**



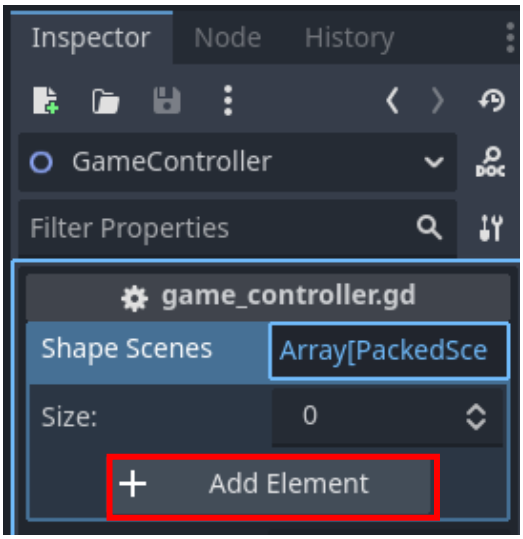
Reminder:

Select the node in the **Scene** to view it in the **Inspector.**

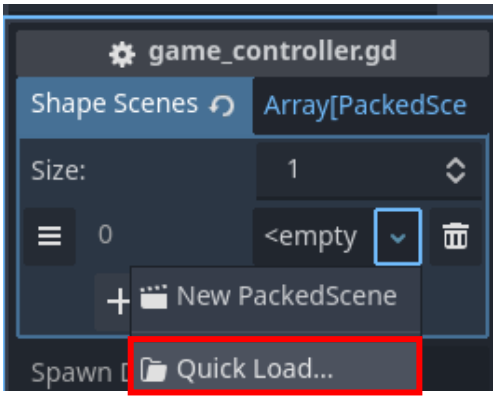
50 Notice an empty **shape_scenes** array is declared in the script, but elements are never added to it and the size of the array is 0...

The **shape_scenes** array is currently empty! This is why the **Out of Bonds Error** occurs.

Let's add the shape scenes to the array. Select **Add Element.**



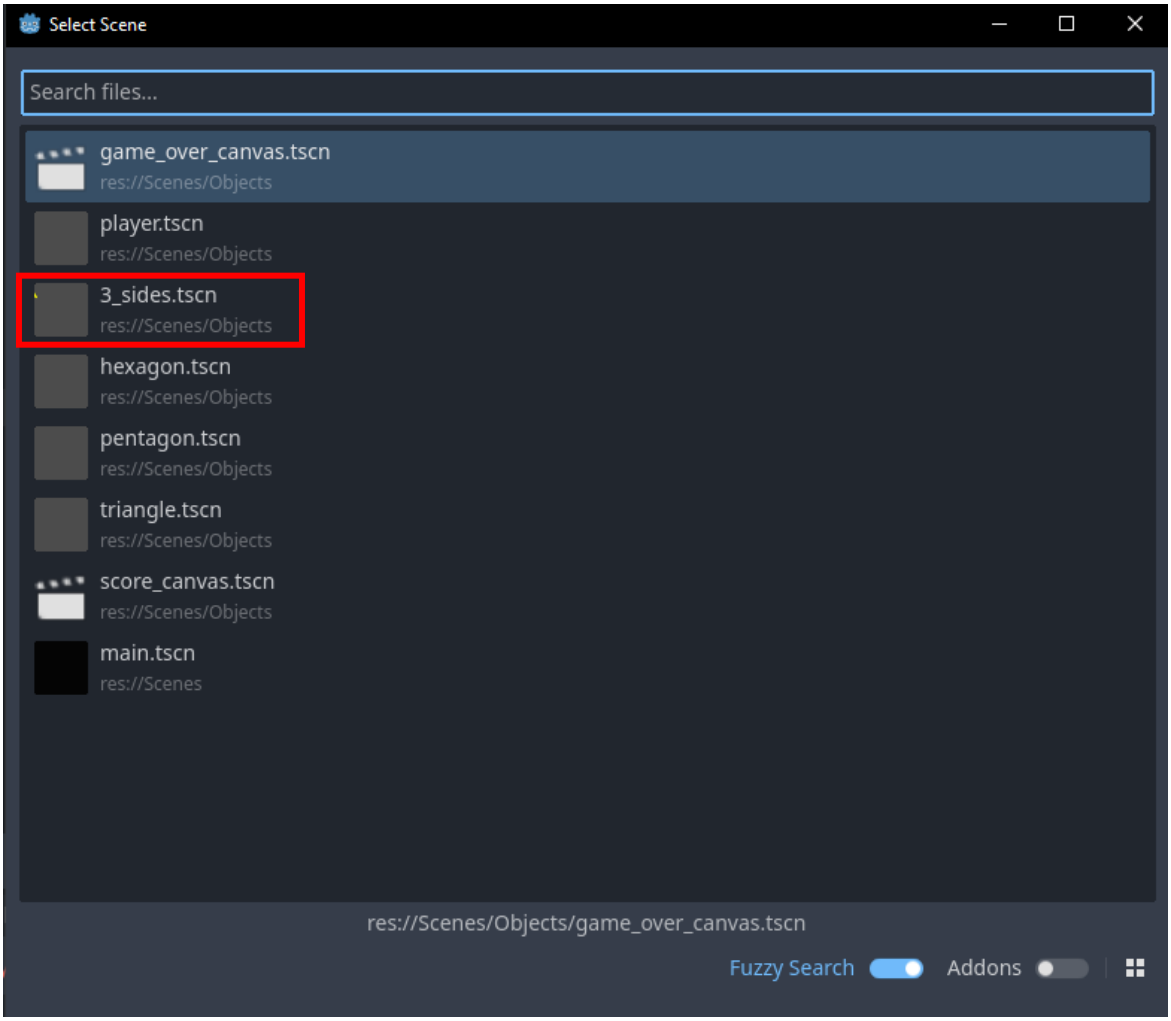
Click the **dropdown arrow** next to **empty** and select **Quick Load.**



Reminder:

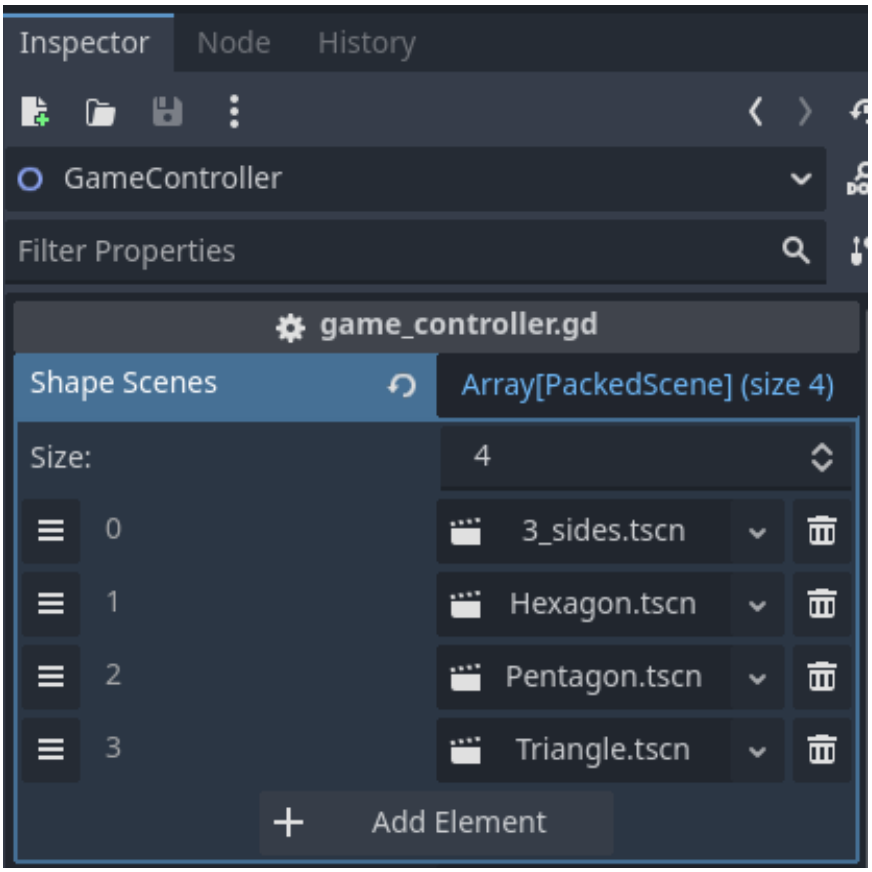
PackedScenes hold the data from a saved scene, that does not exist in the **main scene tree** until it is **instanced** when the game is running. Here, the **shape scenes** just created are **PackedScenes** and will be **instanced** in the **game_controller.gd** script.

51 Click on **3_sides.tscn** and notice the shape scene is now the first index in the `shape_scenes` array.



52 Continue adding the rest of the shape scenes to the **array.** The preview of each scene may appear instead of its file name.

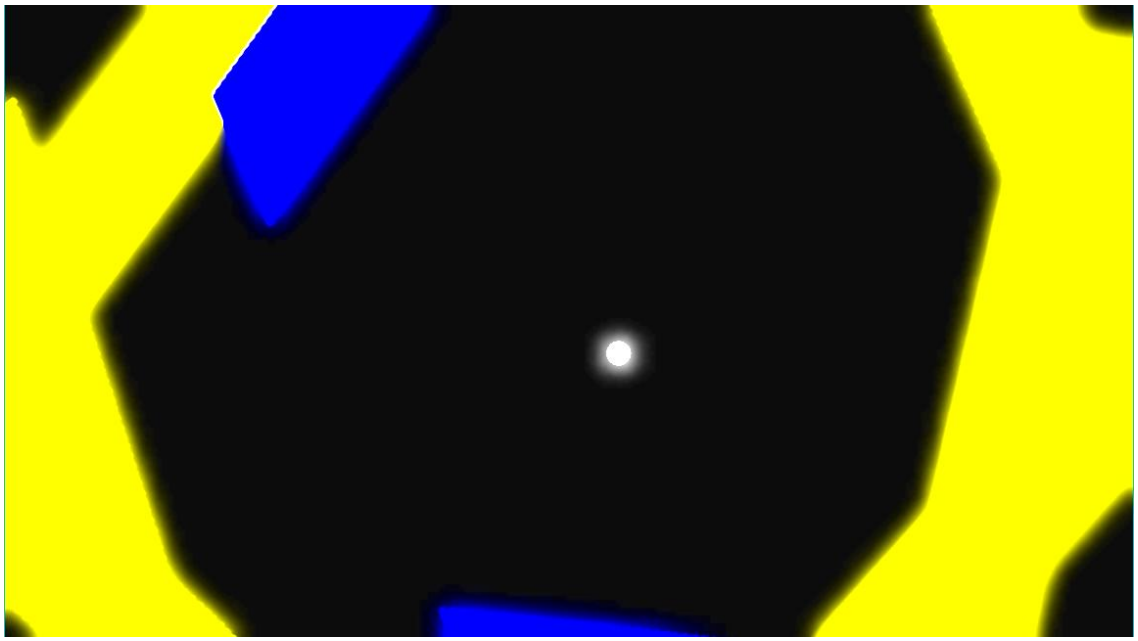
Repeat the previous steps for additional guidance.



53 Playtest the project again and wait a few moments.

The game doesn't crash and the shapes are spawning in!

Hmmm, but it seems the shapes are not moving how they're supposed to.



54 Add code to decrease the size of each shape, when spawned.

Double click to open the **shape.gd** script in the **Scripts** folder.

Under **TODO 6,** add the `_process()` method with `delta` as the parameter.

Inside the function, use the code completion to type the line as shown. This line of code shrinks the shape over time equally in **x** and **y** values at the given `shrink_speed`**.**

```
15  # ---------------------
16  # TODO 6
17  # Write the _process method
18  # ---------------------
19  func _process(delta: float) -> void:
20      scale -= Vector2.ONE * shrink_speed * delta
21
```

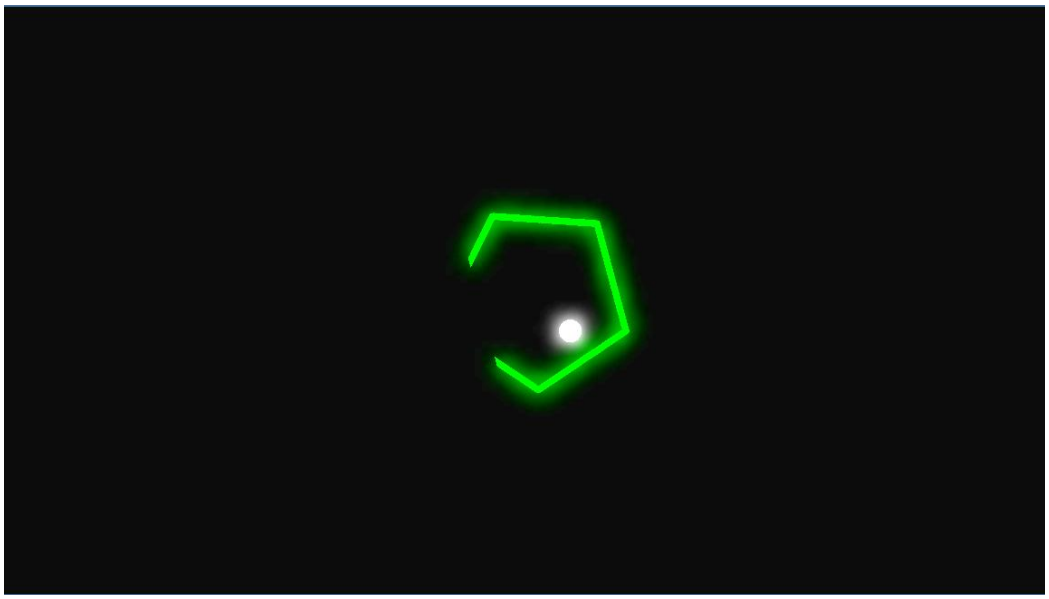
55 Playtest the project again.

Different shapes should be randomly spawning and going from very large to small then back to large again.

Nothing seems to happen when the Player and shapes collide, this will be added next.

Common Bugs:

- If a shape spawns but is not moving, make sure the **shape.gd** script is attached to that scene.
- If no shapes are spawning, navigate back to the **game_controller.gd** script and make sure everything in the `spawn` function is **inside** the `while` **loop**.

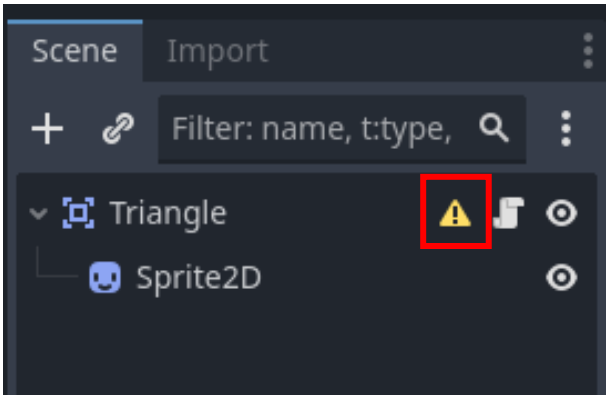


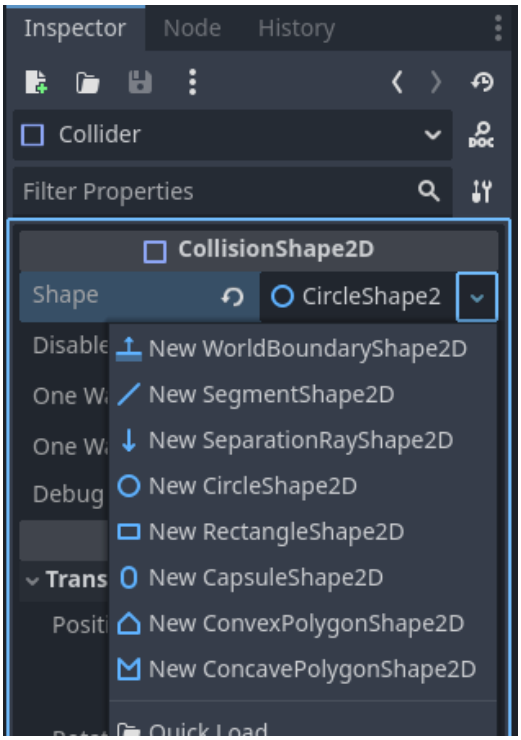
Pause for **Sensei Stop #6**!

Check in with a Code Sensei before moving on. Make sure the different **shape scenes** are spawning in randomly and shrinking over time.

Reminder: Save your work!

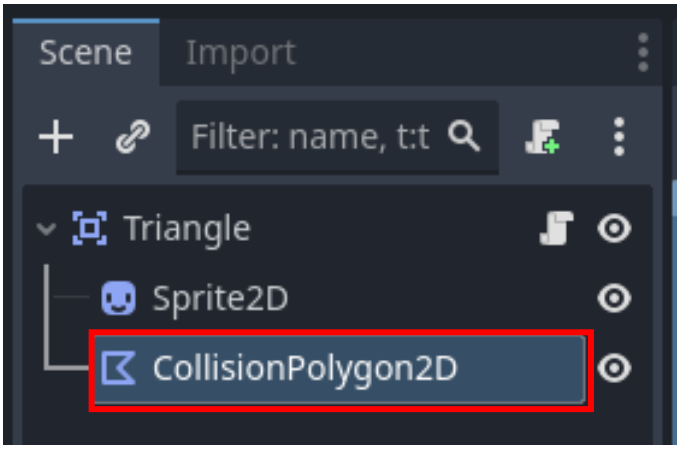
56 Navigate to the **triangle scene.** Notice the **hazard symbol.** A **CollisionShape2D** is needed but there isn't a shape option that properly fits the triangle.





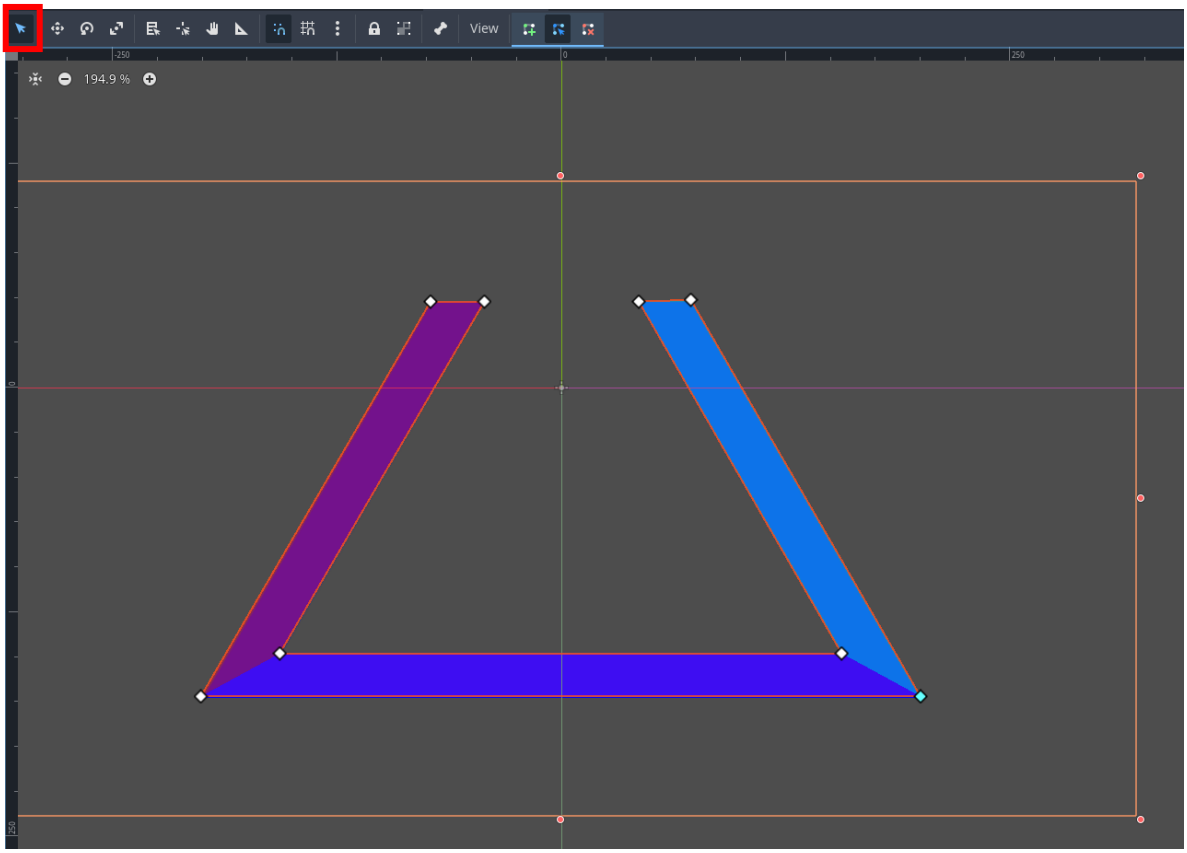
57 Add a **CollisionPolygon2D** node as a child to the **Triangle root.**

This allows a polygon collision shape to be created for the polygon!



58 At the top of the game window, click **2D** view then use the **Select** tool.

Click on the different corners of the polygon to create a **polygon collision shape** as shown.



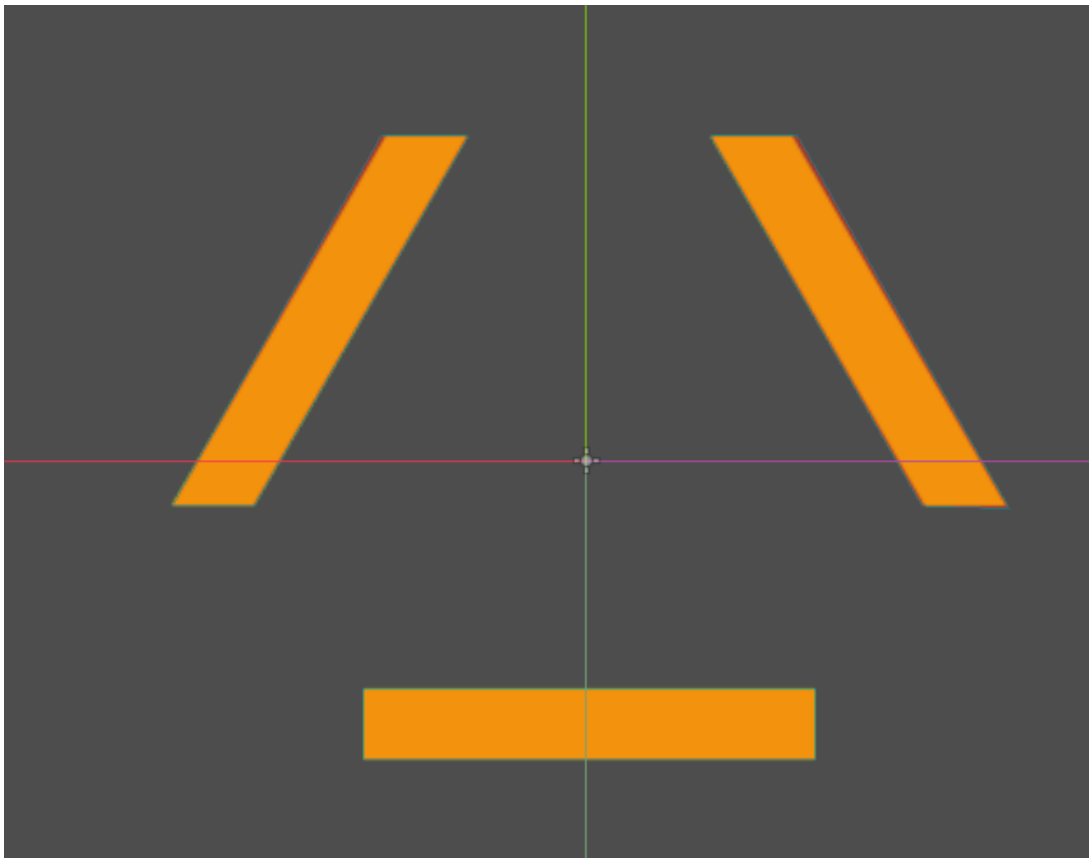
Pro Tip:

It might be helpful to **zoom in** on the workspace to see the edges of the polygon better. After connecting the entire polygon, you can drag the points around to fine-tune the shape!

59 Create **collision polygons** for the rest of the **scene shapes.**

Repeat the previous steps for help.

Hint: Some **scenes** might need more than one **CollisionPolygon2D** node.



60

Add code to make something happen when the **Player** collides with a shape**.**

Open the **player_controls.gd** script and navigate to the bottom.

Under **TODO 7**, define a `collide_into_shape()` method which takes a `body` parameter of type `Area2D` and returns `void`.

Inside the method, use the code completion to add the `get_node()` method. Inside, type `"/root/Main/GameController"`**.** Then use the dot operator and call `game_over()`**.** This finds the **GameController** node in the **main** scene and then runs a `game_over()` method in its attached script.

What will happen now when the Player collides with a shape?

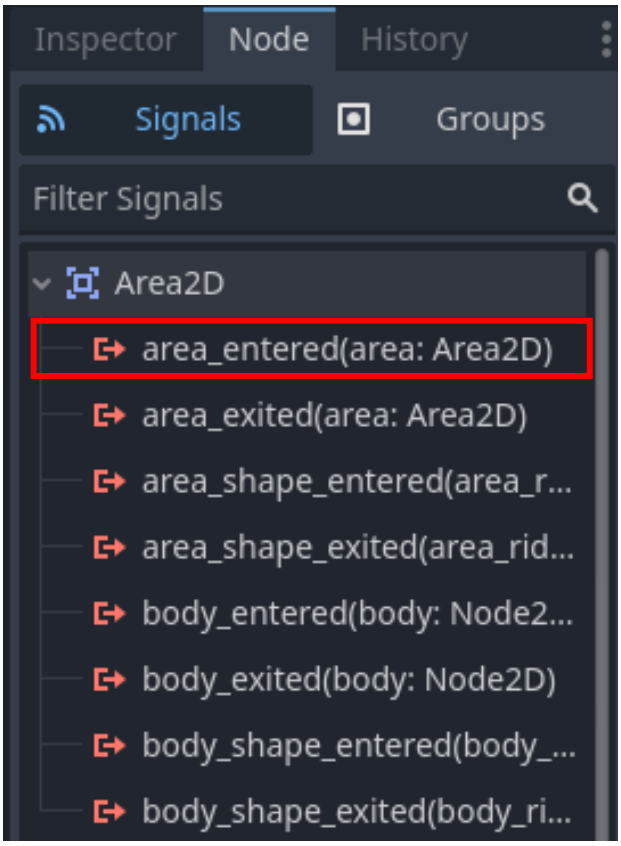
```
23  # ---------------------
24  # TODO 7
25  # Write the collide_into_shape method
26  # ---------------------
27  func collide_into_shape(body: Area2D) -> void:
28      get_node("/root/Main/GameController").game_over()
29
```

Reminder:

Always save the script before moving on!

61 The last step is to add a **signal** to know when the collision occurs.

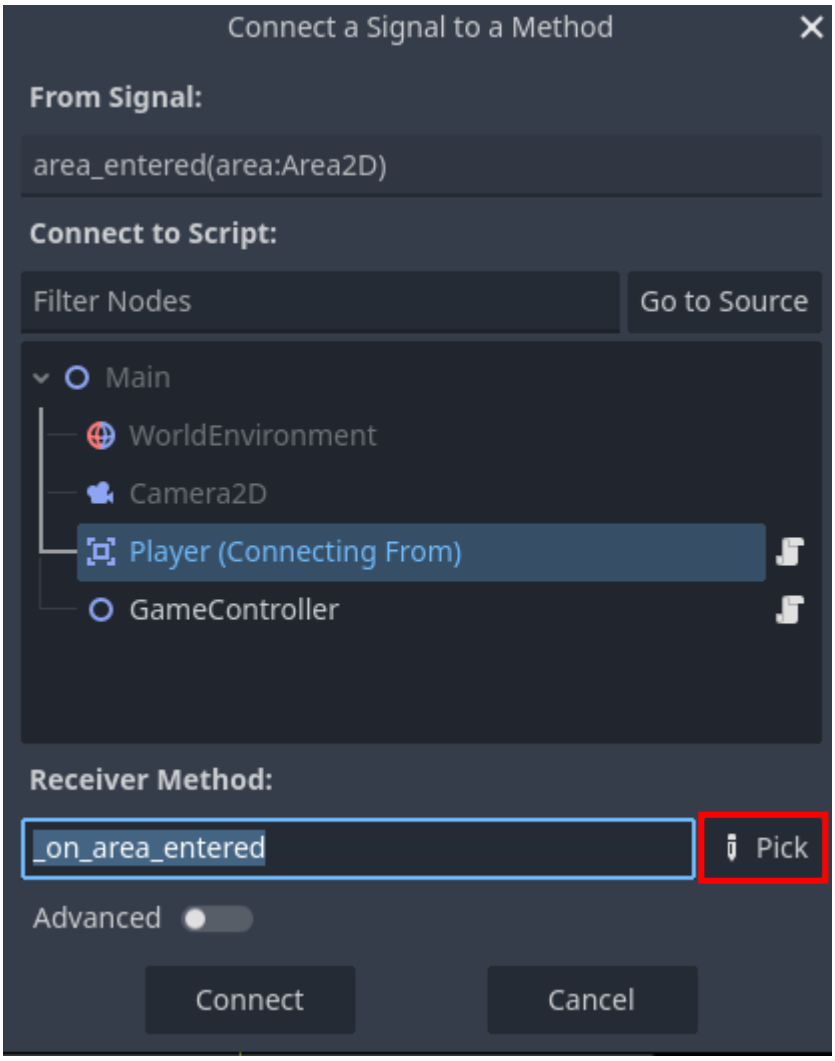
Navigate to the **main scene** and select the **Player** node**.** In the **Inspector** of the **Player,** click on the **Node** tab, then **double-click** the **area_entered signal.**



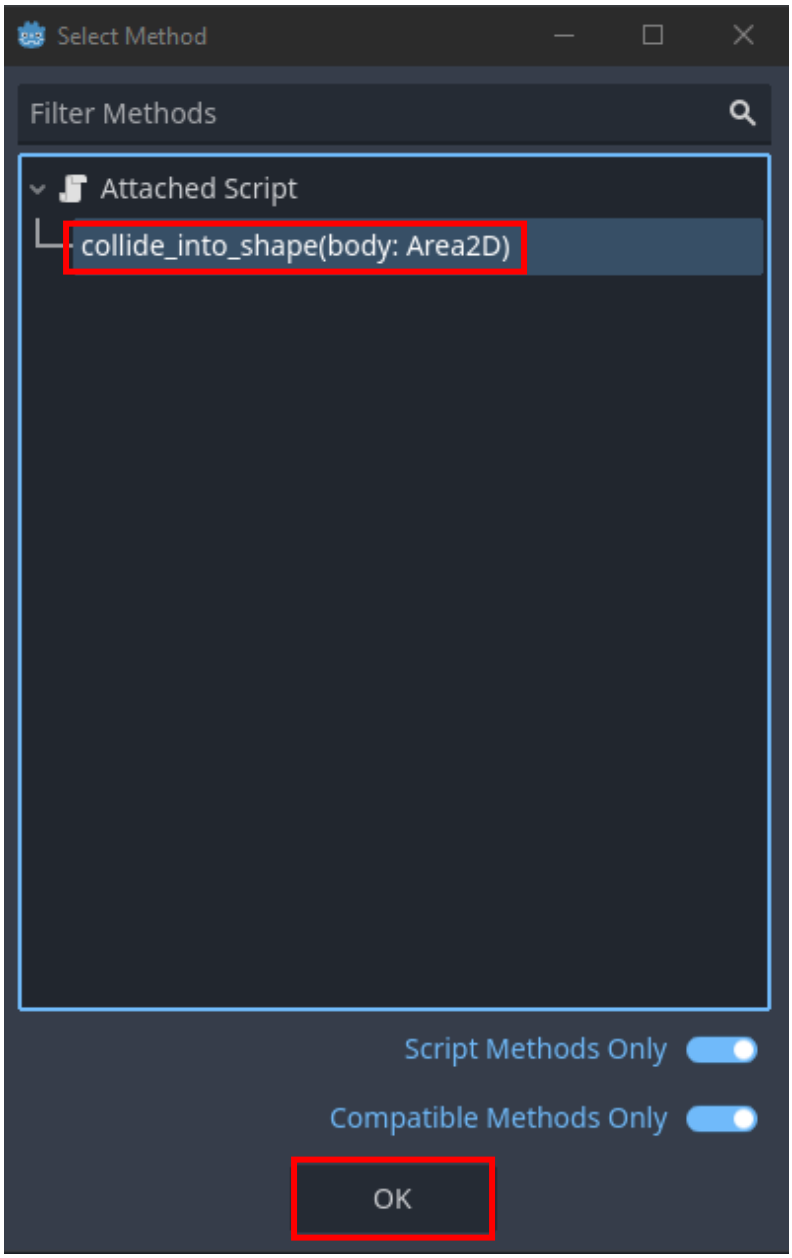
Pro Tip:

Make sure to add the signal to the **Player** node in the **main scene** and **NOT** the **player scene**.

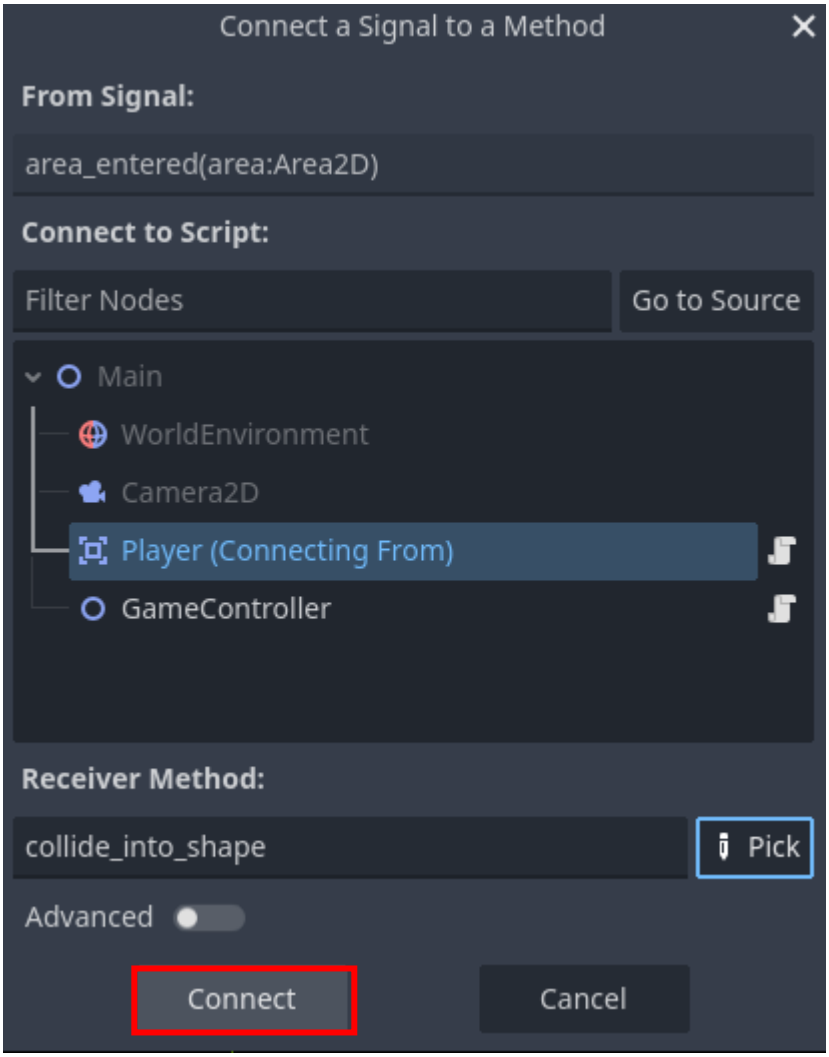
62 In the **Connection** window, click on the **Pick** button.



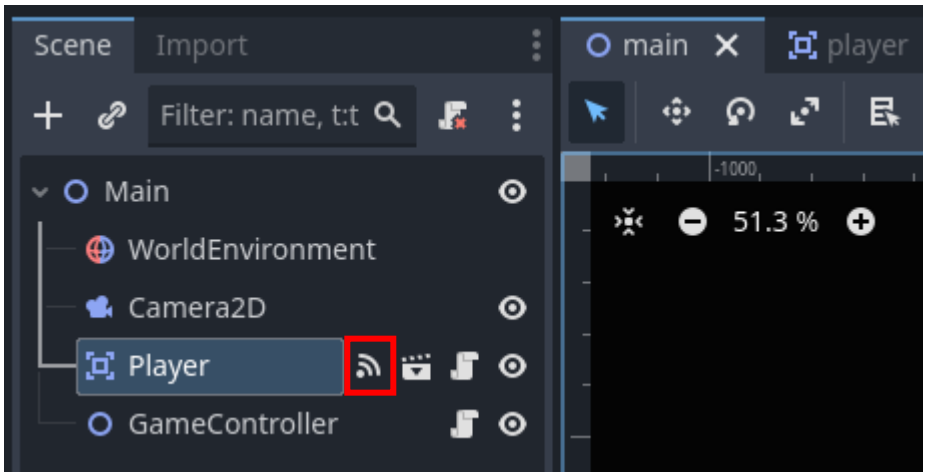
63 Select the newly made **`collide_into_shape()` function** then click **OK.**

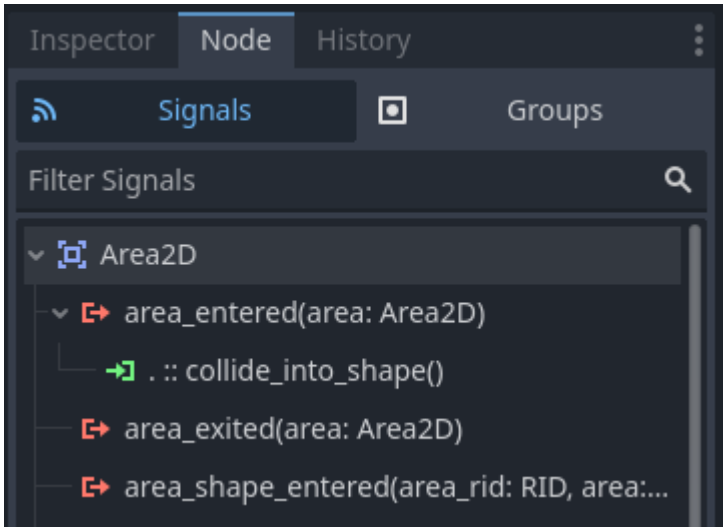


64 Ensure `collide_into_shape` is used as the **Receiver Method** and is connecting from the **Player**. Select **Connect.**



65 A **signal** icon should now be displayed next to the **Player sub-tree** in the **main scene**. The signal should also appear in **Node**.





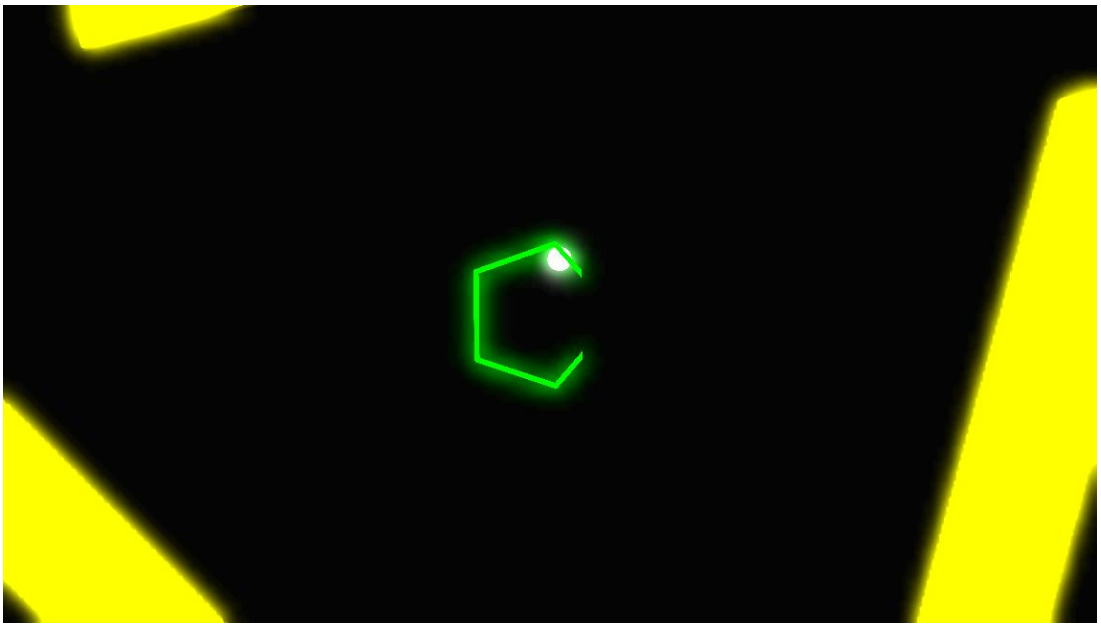
If the signal was added in the **Player scene,** redo the previous steps so it is in the **main scene.**

Pause for **Sensei Stop #7**!

Check in with a Code Sensei to make sure the **CollisionPolygon2D** nodes and the **Player** signal were set up correctly.

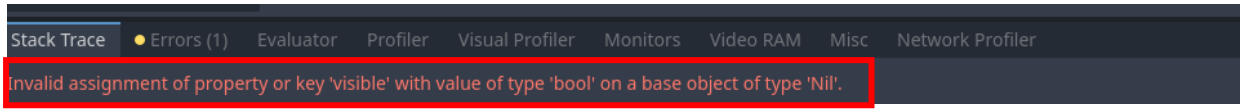
Reminder: Save your work!

66 Playtest the project and have the **Player** collide with a **shape.**



When the two collide, the game crashes!

Review the **Invalid Assignment Error** at the bottom.



67 Notice the **yellow arrow** indicates which line is causing the **error**.

The error says that **key visible** cannot be assigned to a value of type **bool** when it is currently of type **Nil.**

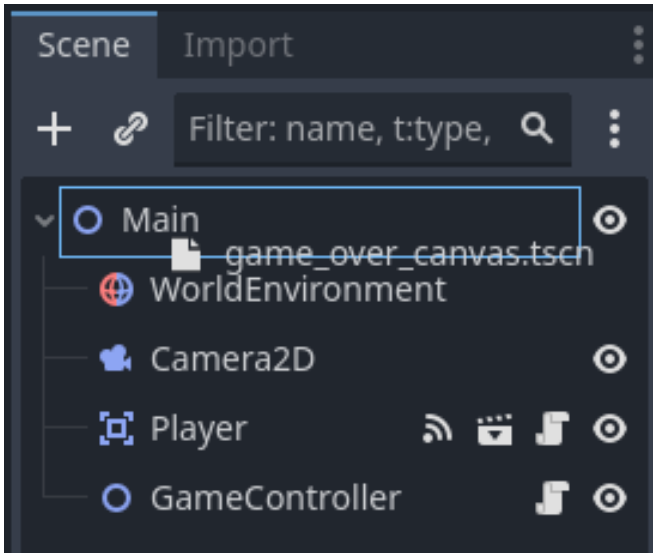
Nil refers to a value that does not exist yet.

What might be done to fix this?

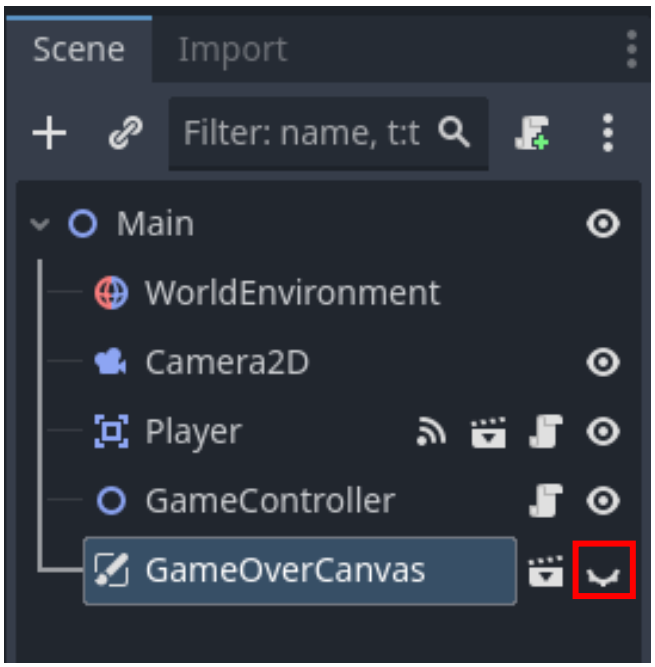
```
44
45  # The function that begins the game over part of the program
46  func game_over():
47      # Stops the spawn code from running continuously
48      repeating = false
49      # Sets the game over canvas to be visible so the player can reset the game
50      game_over_canvas.visible = true;
51      # Freezes the time scale of the game so the player cant move around while the game is over
52      Engine.time_scale = 0
53
```

68 Currently a **GameOverCanvas** node doesn't exist yet!

In **FileSystem**, navigate to the **game_over_canvas.tscn** scene in the **Objects** folder and drag it onto the **Main root** node.



Notice the eye symbol next to the node. Remember, the eye tells the visibility of a node. Since it is closed, the **GameOverCanvas** subtree is currently invisible.

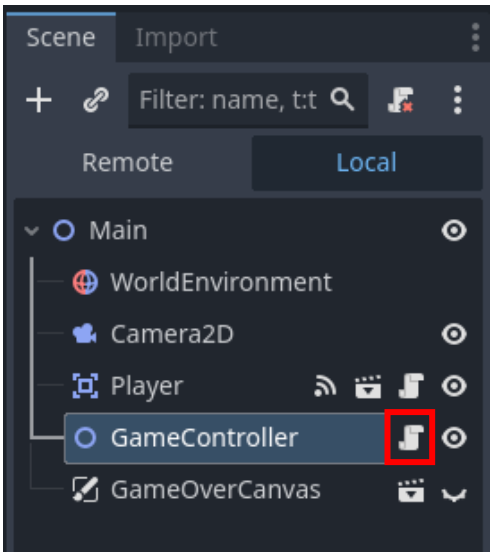


Playtest the project to test if the game over screen appears or not.

69 Hmm the error still seems to be occurring.

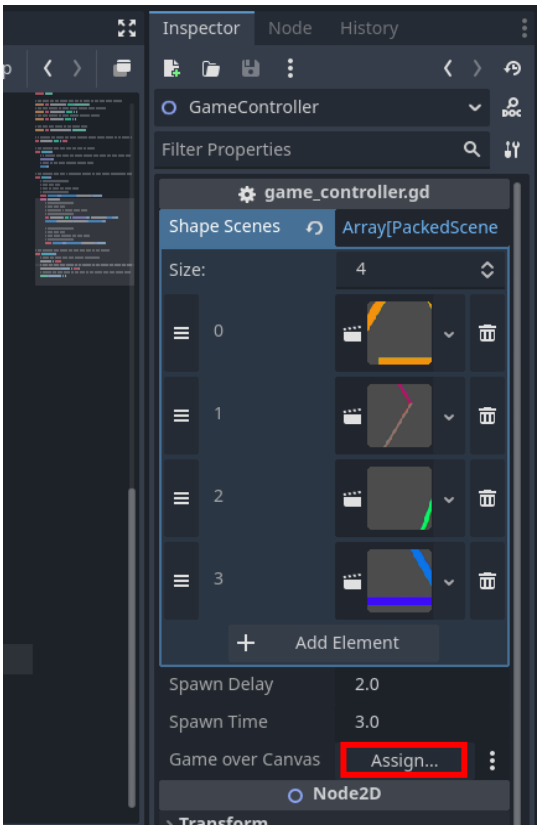
In **Scene**, open the **GameController** node's script and find the `game_over_canvas` variable declaration.

Since it is exported, the variable's value must be set in the **Inspector** too!

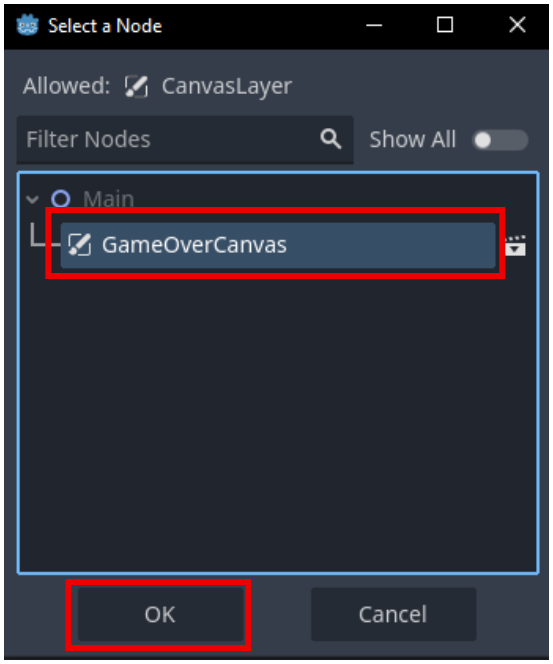



```
1  extends Node2D
2
3  @export var shape_scenes: Array[PackedScene]
4  @export var spawn_delay: float = 2
5  @export var spawn_time: float = 3
6  @export var game_over_canvas: CanvasLayer
7
```

70 Select the **GameController** node, and in the **Inspector** click "**Assign...**" next to Game over Canvas.

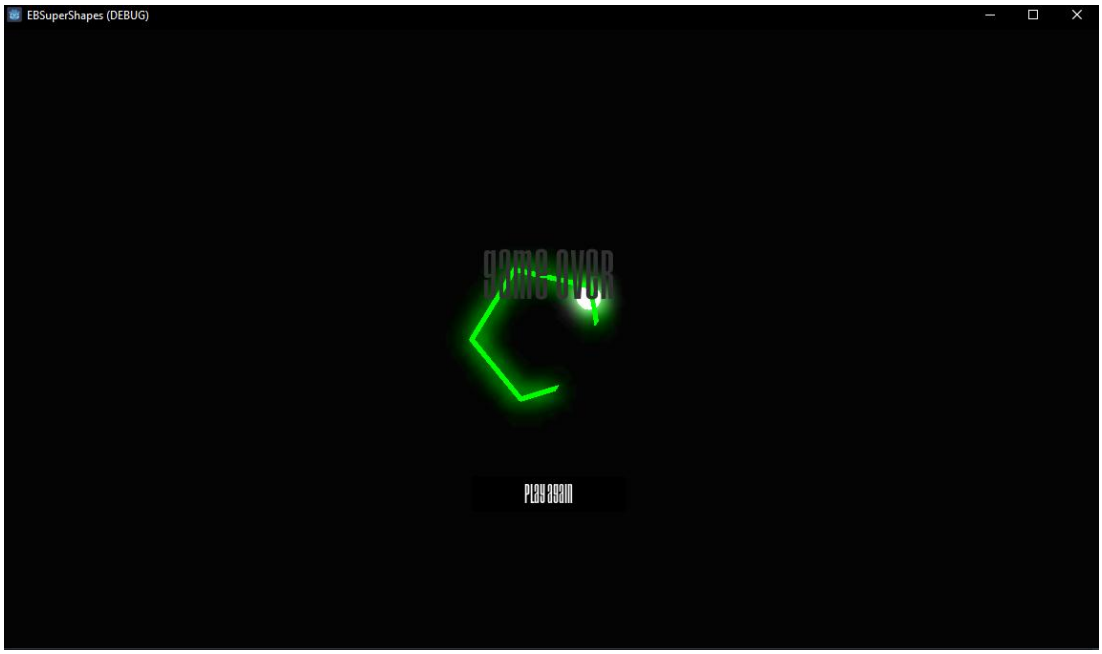


In the pop-up window, choose **GameOverCanvas** and click **OK**.



71 Playtest the game.

Whenever the Player collides with a shape, the game ends and the Player can play again!

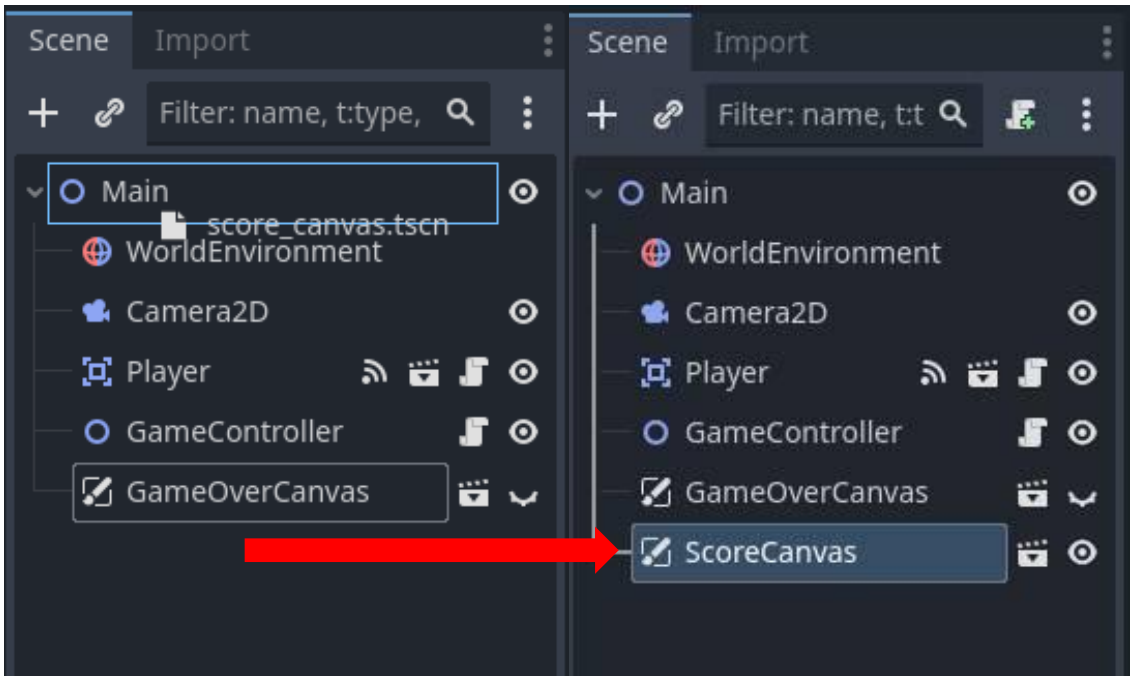


Pause for **Sensei Stop #8**!

Check in with a Code Sensei before moving on. Ensure the Game Over screen appears when the Player collides with a shape.

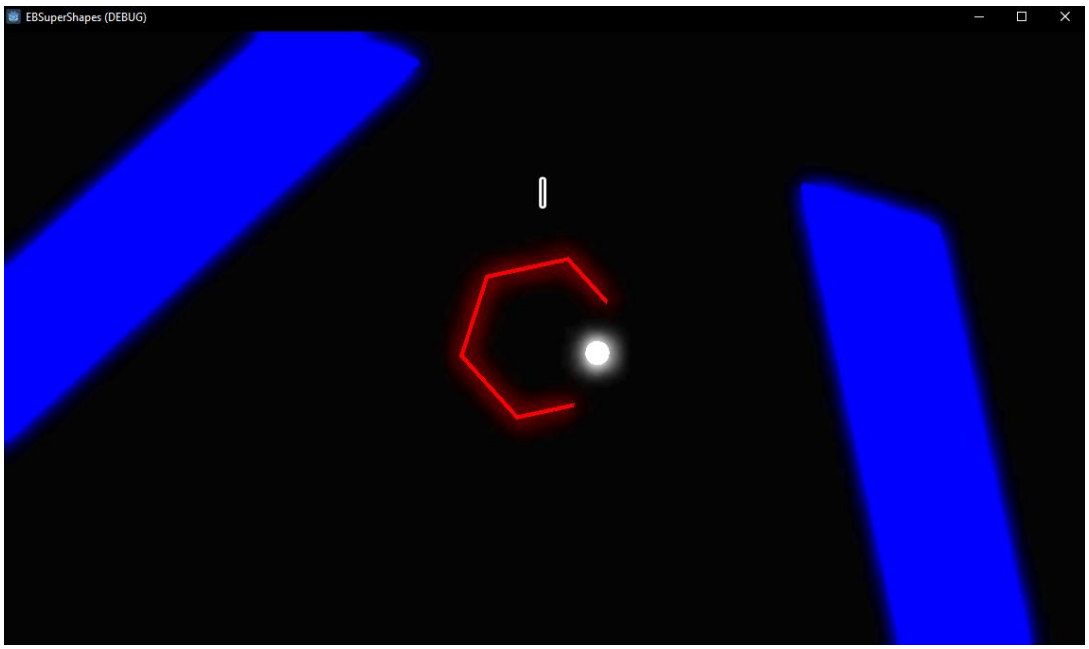
Reminder: Save your work!

72 In **FileSystem**, navigate to the **score_canvas.tscn** scene in the **Objects** folder and drag it onto the **Main root node**.



Playtest the project to see the score UI has been added.

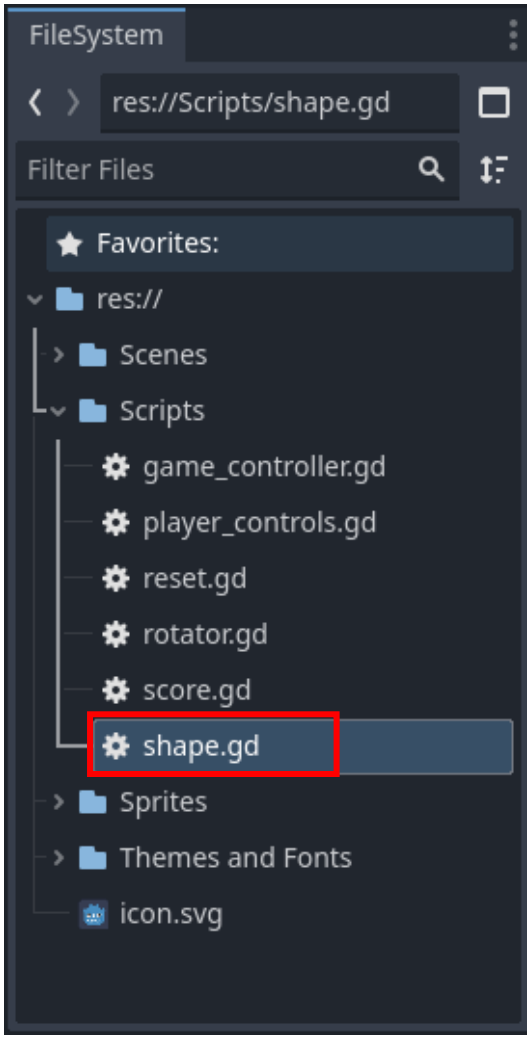
Does the score increase after dodging a shape?

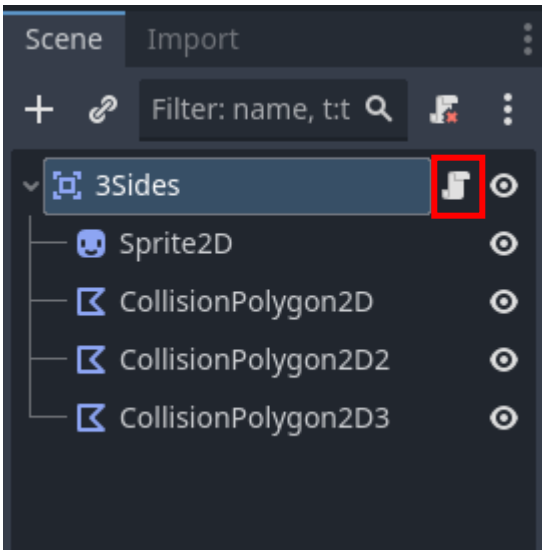


73 It looks like the score doesn't increase. In fact, the shapes are still shrinking then growing again when they should disappear when they get too small.

Both issues can be fixed at once.

Open the **shape.gd** script.





Pro Tip:

Scripts can be opened by either finding it in **FileSystem** or opening the script attached to the root of any of the shape scenes (3_sides, triangle, pentagon, hexagon).

74 Once the shape gets too small, the score should increase and the shape should disappear.

Navigate to the `_process()` method under **TODO 6**. On the next line, write an `if`-statement that checks if the `scale's x` value is smaller than `0.05`.

Inside the if statement, add `Score.score += 1` to increase the score variable in the Score script by one, and call `queue_free()` to delete the node that the script is attached to.

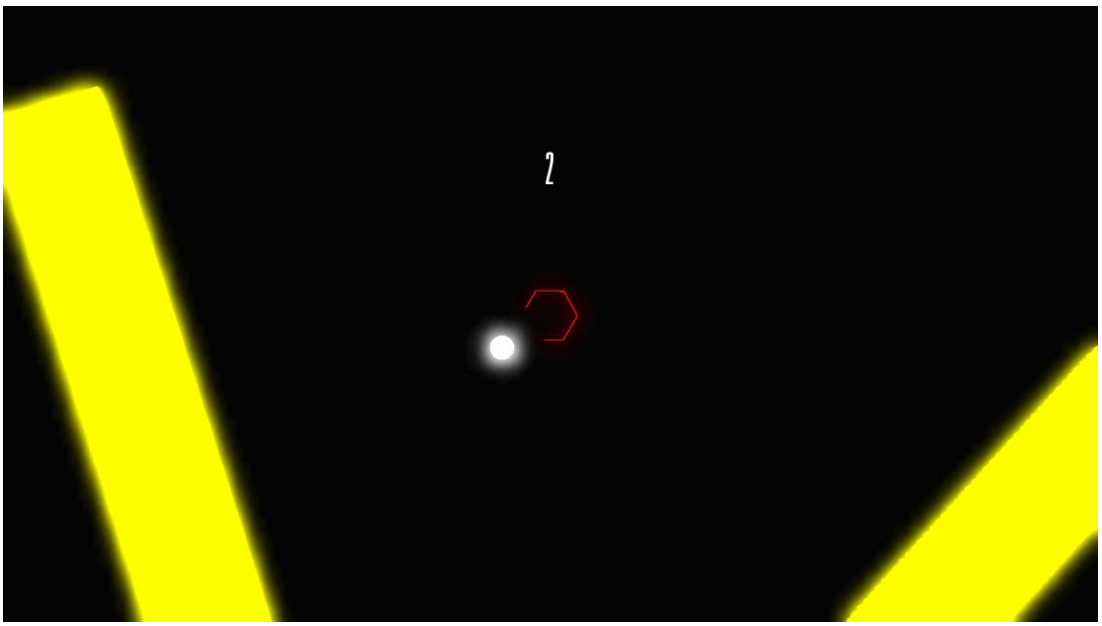
```
15  # ---------------------
16  # TODO 6
17  # Write the _process method
18  # ---------------------
19  func _process(delta: float) -> void:
20      scale -= Vector2.ONE * shrink_speed * delta
21
22      if scale.x < 0.05:
23          Score.score += 1
24          queue_free()
25
```

Think back to MakeCode. The `.sx` used to check a sprite's x scale is just like `scale.x` in Godot. To increase the score by one, `info.changeScoreBy(1)` serves the same purpose as `Score.score += 1` and `shape.destroy()` removes the sprite from the game just like calling `queue_free()` on a node in Godot.

```
6       if (shape.sx < 0.05) {
7           info.changeScoreBy(1)
8           sprites.destroy(shape)
9       }
```

75 Playtest the project to see the score increase and the shapes disappear!

If it's not working, refer back to the previous steps and make sure none of them were skipped.



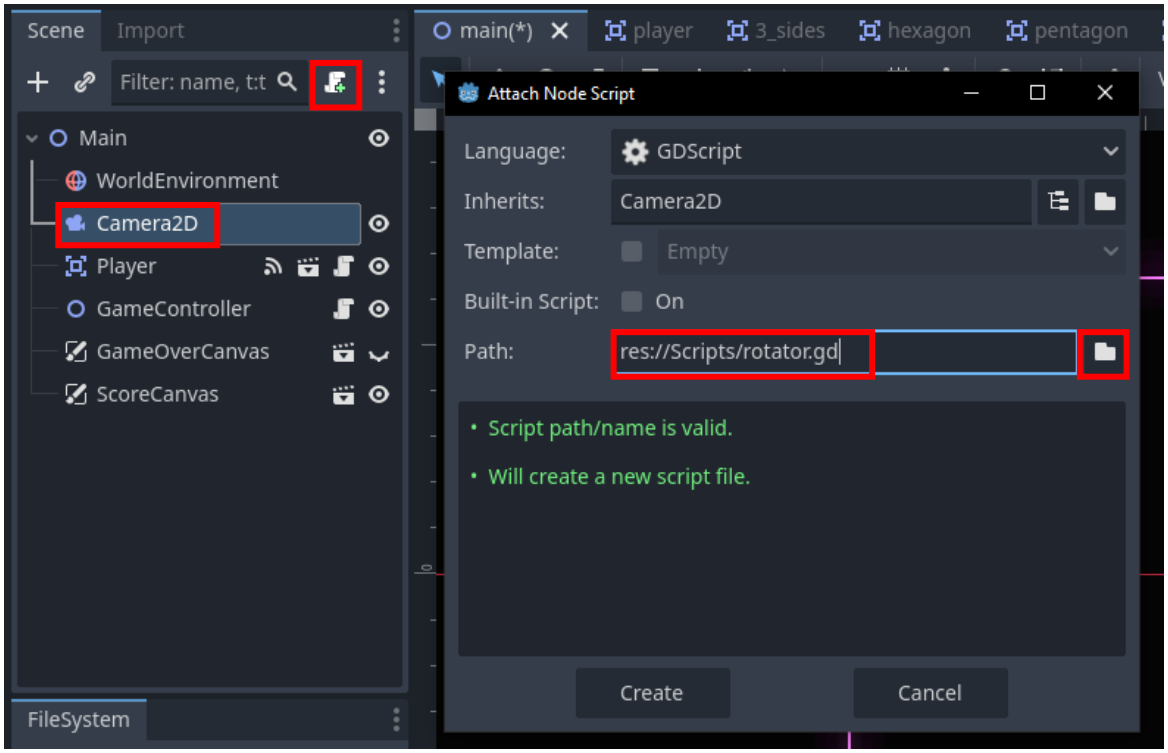
Pause for **Sensei Stop #9**!

Check in with a Code Sensei before moving on. Ensure the Score increases when the Player dodges a shape.

Reminder: Save your work!

76 Now that everything's working, it's time to spice up the game! Rotate the camera to make the user experience more interesting.

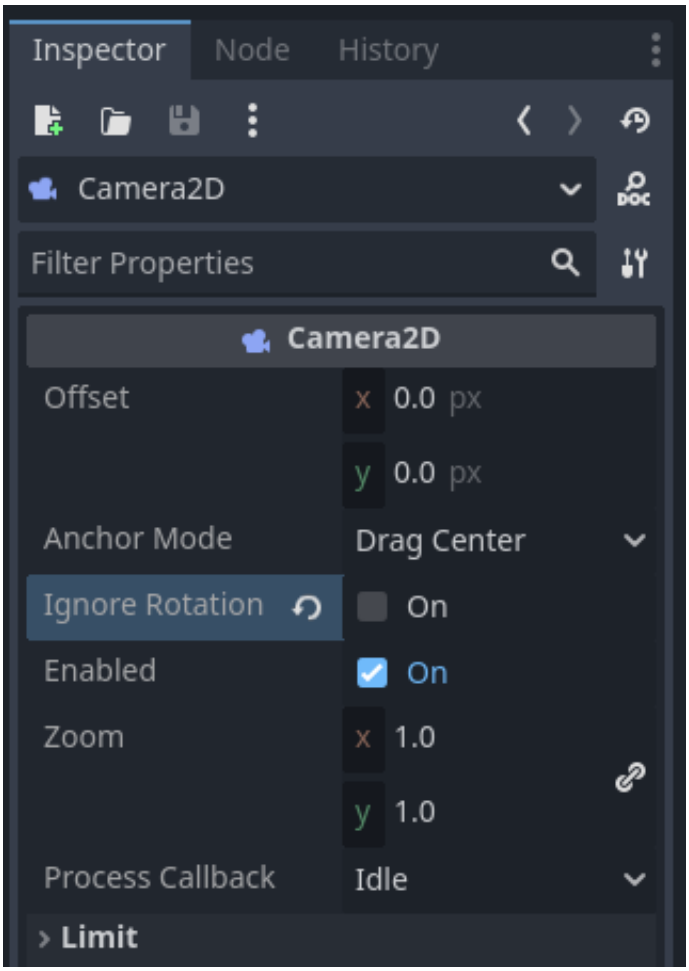
Select the **Camera2D** node and attach a new Script to it. Use the script path **res://Scripts/rotator.gd** by typing or clicking the folder icon. Then click **Create**.



77 In the new script, define the `_process` function. Inside the function, call the `rotate()` function, using `delta` as the parameter.

```
1   extends Camera2D
2
3   func _process(delta):
4       rotate(delta)
5
```

In the **Inspector** for **Camera2D**, uncheck **"Ignore Rotation"**. This will allow the viewport to rotate according to the **Camera2D's** orientation.



78 The game is complete! Try it out. What's your high score?

Pause for **Sensei Stop #10**!

Congratulations on creating your first game with polygon colliders in Godot! Great job!

Before submitting, check in with a Code Sensei to make sure the score and rotation works then reflect on the following:

- ☐ What did you learn about polygon colliders?
- ☐ What did you enjoy most when creating this project?
- ☐ What was something you found difficult and why?

Reminder: Save your work!