CODE NINJAS®

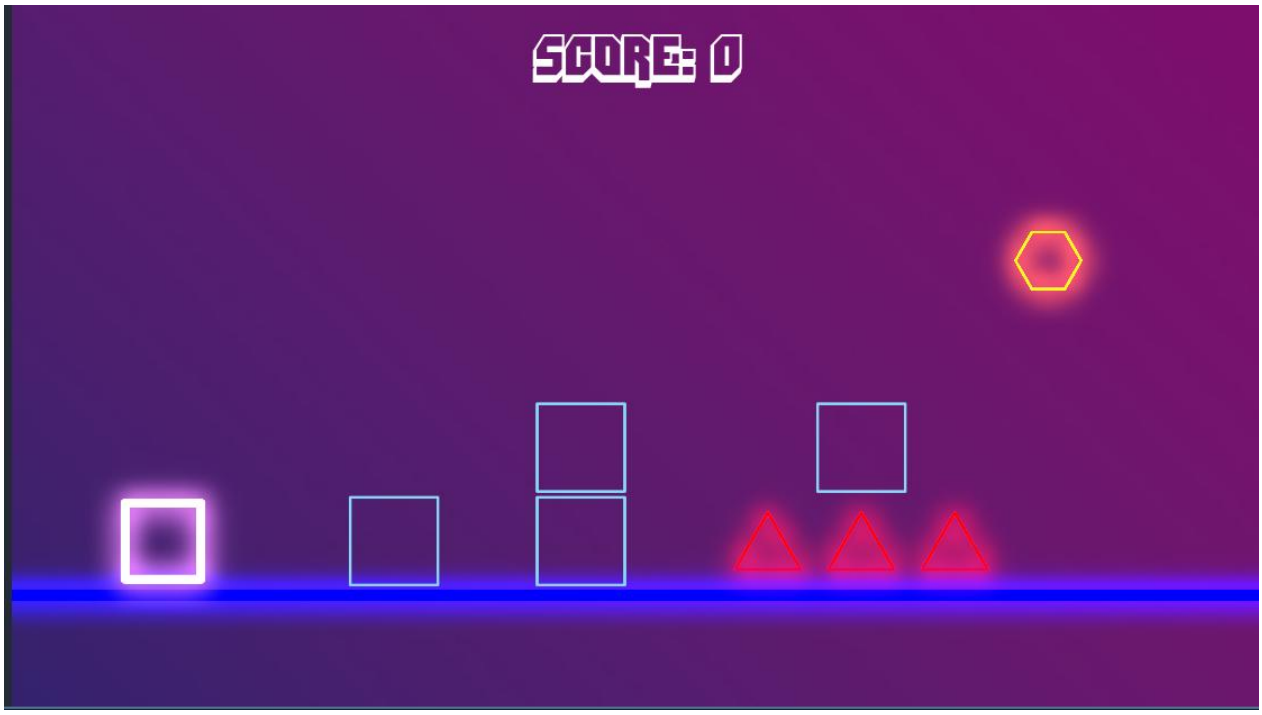
Bronze Belt Ninja Guide

Activity 07: PolyRun

Activity 07: PolyRun

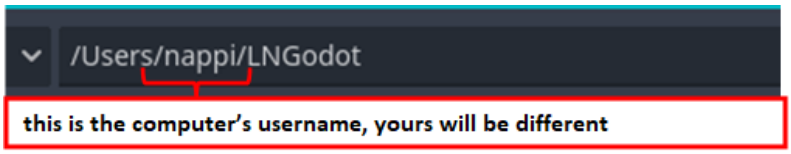
In this activity, you will create an auto-scrolling platformer game where the player must time their jumps to climb over obstacles, avoid spikes, and collect coins. This project will help you learn about global scripts, polygon colliders, and how to manage information flow within a game.

By the end of this activity, you will have explored how to set up a global script and use it as a controller for a game, link nodes across your project to signals from a global, and design custom obstacles using packed scenes as building blocks.

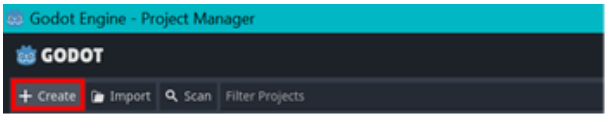


1 Remember all projects will be stored in a path like:
/Users/[MyComputerUsername]/[MyInitials]Godot

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

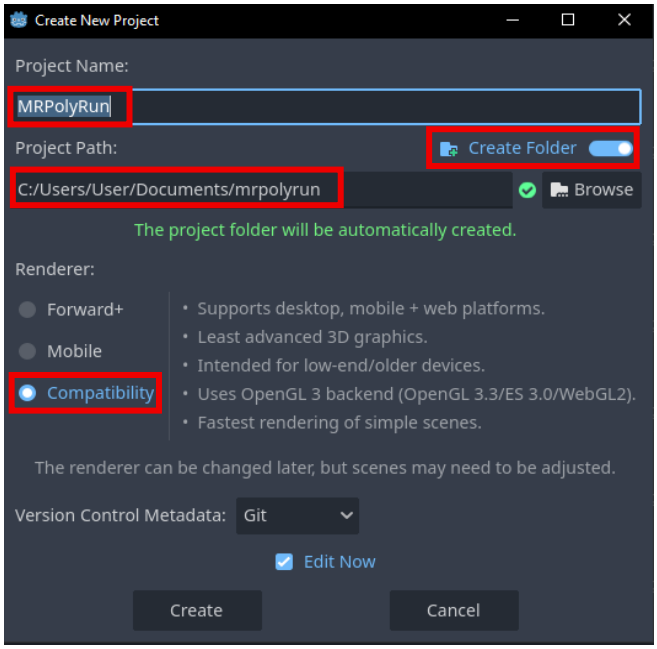


2 In the Godot Project Manager, create a new project.

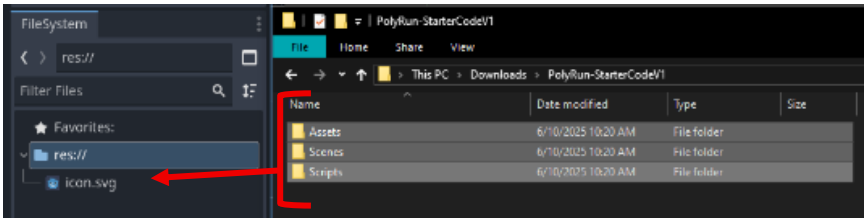


3 Name the project **[MyInitials]PolyRun** and adjust the project path so the project is being saved in the correct folder.

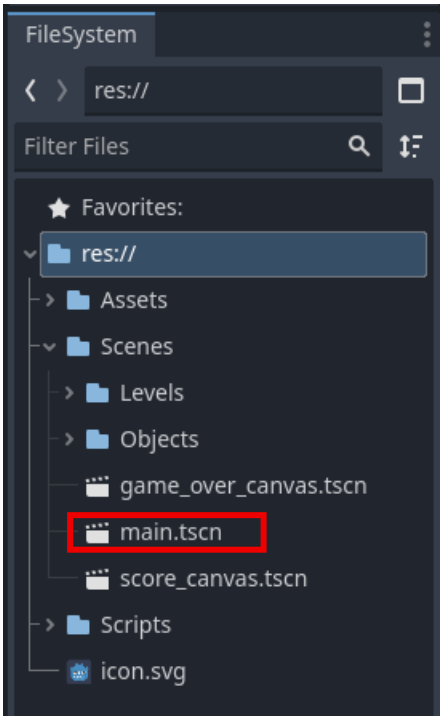
Make sure **Create Folder** is toggled on, set the renderer to **Compatibility**, then click **Create**.



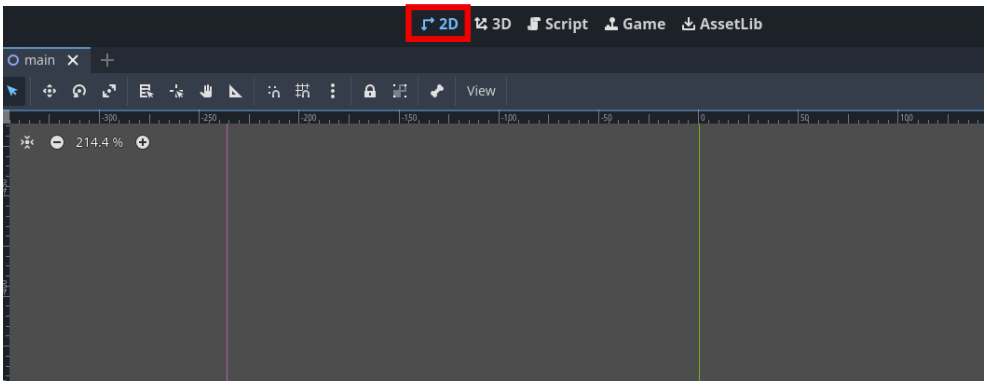
4 Extract **BB Activity 07 – Ninja Starter Pack.zip** and select all folders inside. Drag them into the **res://** folder in **FileSystem**.



5 In **FileSystem**, open the **Scene** folder and navigate to **main.tscn**. Double click to open the main scene.



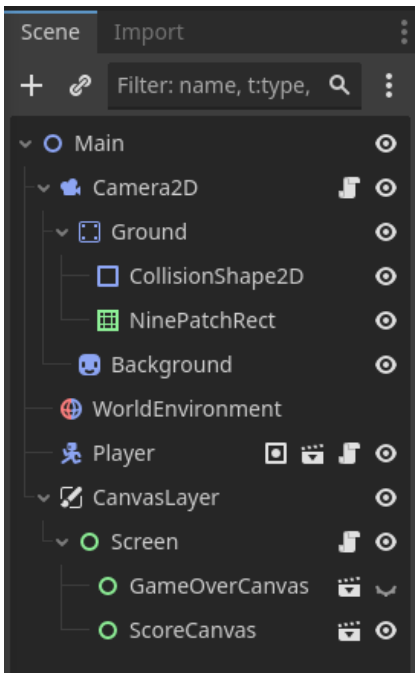
At the top center of the Godot editor, check that **2D** is selected.



Reminder:

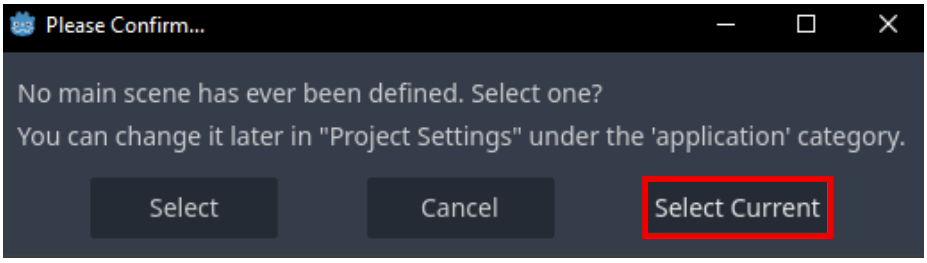
Click the arrows next to the folders to open them.

6 After completing the previous step, the **Main scene** should resemble the image below. Be sure to also verify that the **FileSystem** structure matches the example provided.



7 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. The game will immediately crash and print an error that Globals has not been defined in the current scope. This will be updated in the next few steps.



8

Use a **global script** to manage the sending and receiving of **signals**!

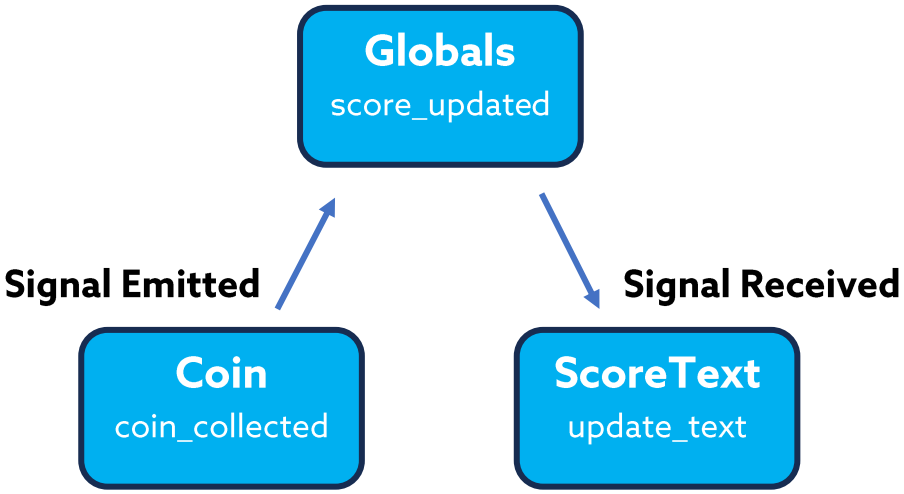
Global Scripts are a type of script that gets attached to a node and added to the scene tree when a Godot project is playtested.

These scripts have a unique property that makes them accessible from anywhere in a project. This property is very useful to quickly access information but can cause lag if used incorrectly.

The global script will create two new signals: **game_end** and **score_updated**.

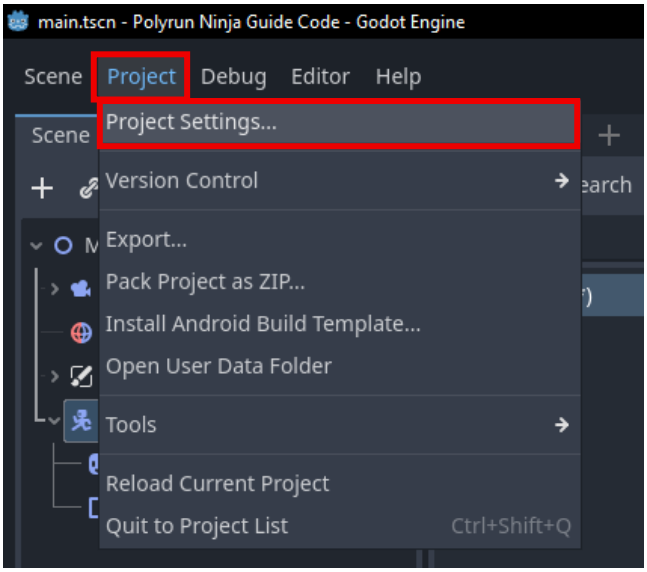
Nodes in the project, as seen in the diagram, will then either call the global script to emit one of those signals or connect one of their functions to the global signals, so when it is called, their function runs.

Note: Not every global script will be called **globals.gd**!



9 Update the project settings so that the **globals.gd** script is autoloaded.

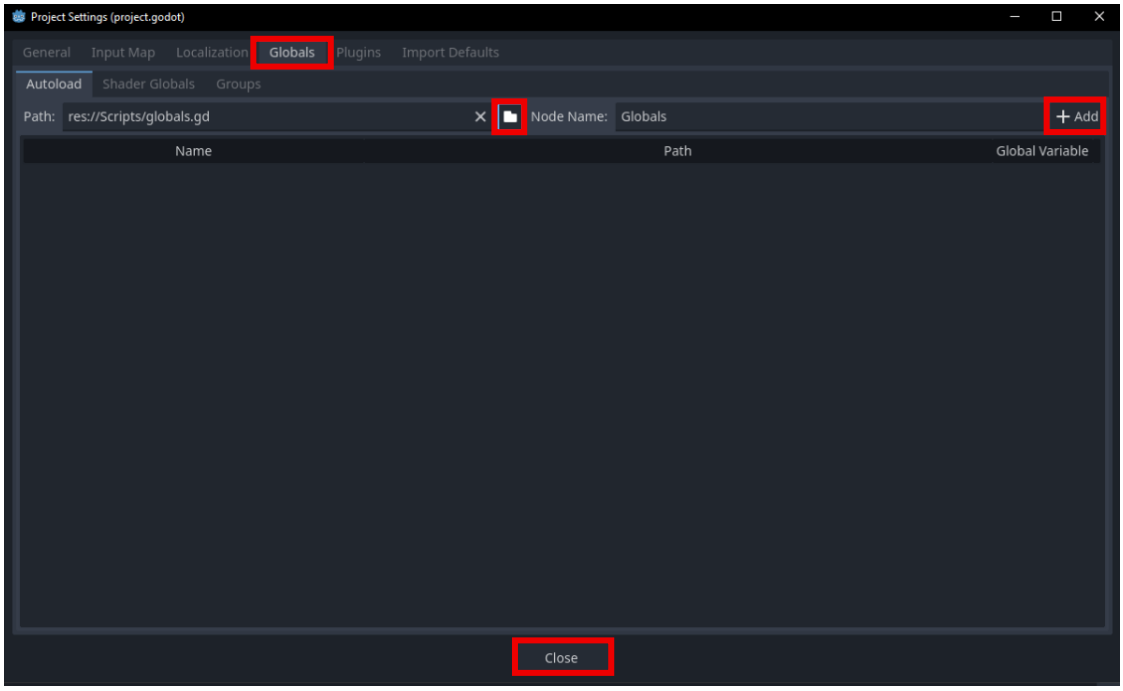
In the top left corner of the editor, click **Project** and select **Project Settings**.



10 Inside **Project Settings**, open the **Globals** tab.

Click on the **file icon**, then open the **Scripts** folder. Select the **globals.gd** script, then click **Add**.

Close the **Project Settings** window.



Pause for **Sensei Stop #1**!

Check in with a Code Sensei before moving on; discuss how a person's eyes, brain, and hands work together to catch a ball. How is that similar to the structure of this global script?

Reminder: Save your work!

11 In **FileSystem**, open the **globals.gd** script.

Locate the **TODO 1** comment and declare a `score` variable of type `int` underneath. Set the variable to `0`.

This variable will keep track of the game score.

```
3  # --------------------
4  # TODO 1
5  # Create the score variable
6  # --------------------
7  var score: int = 0
8
```

12 Underneath the **TODO 2** comment, use the keyword `signal` to add 2 signals to the script: `game_end()` and `score_updated(score: int)`.

```
9  # --------------------
10 # TODO 2
11 # Create the game_end & score_update signals
12 # --------------------
13 signal game_end()
14 signal score_updated(score: int)
15
```

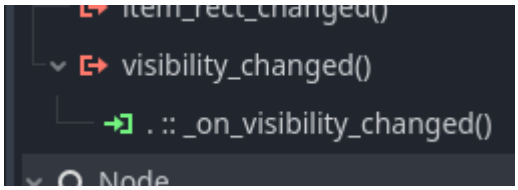
Reminder:

Signals are pieces of code that scripts can connect functions to. When a signal is emitted, all connected functions are called. Signals allow for easy connection across different nodes.

13 These global signals work the same way as signals connected through the inspector. However, they must be connected through code instead.

Pre-built signals, such as those on **Rigidbody** or **Area** nodes, also have mechanisms built into their nodes that cause them to emit. When a custom signal is created through code, the mechanism to emit that signal must also be programed.

```
4  func _on_visibility_changed() -> void:
5      pass # Replace with function body.
6
```



Connecting signals through the editor

Creating, connecting, and emitting signals with code

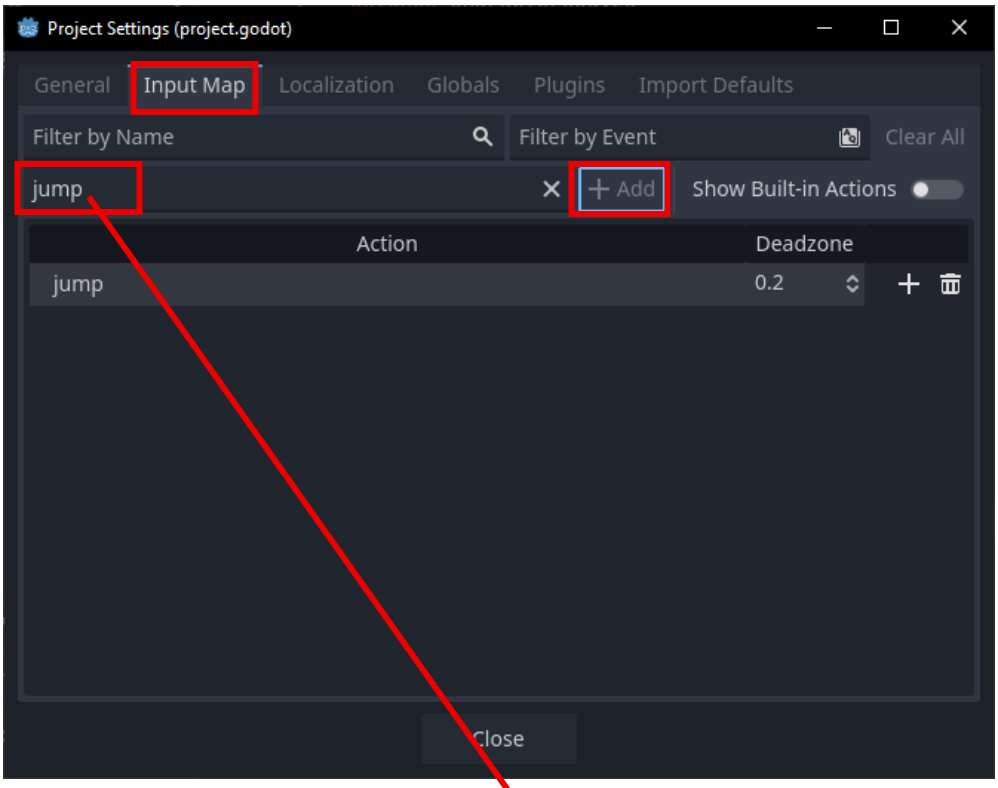
```
4  # Signal creation
5  signal custom_signal()
6
7  #Signal connection to a function
8  func _ready() -> void:
9      custom_signal.connect(listening_function)
10
11 #Function to connect to signal
12 func listening_function():
13     pass
14
15 #Emitting the signal
16 func _process(delta: float) -> void:
17     if score > 10:
18         custom_signal.emit()
```

14

Use custom input mapping to enable the player to jump!

Select **Project**, then open **Project Settings**. Select **Input Map**, then type **jump** into the **Add New Action** search bar. Select **+ Add.**

This creates a new action, **jump**, which is used in the **player.gd** script.




```
func _physics_process(delta: float) -> void:
    velocity.x = current_speed

    if is_on_floor():
        if Input.is_action_just_pressed("jump"):
            velocity.y = sqrt(2 * current_gravity * jump_height) * -1
```

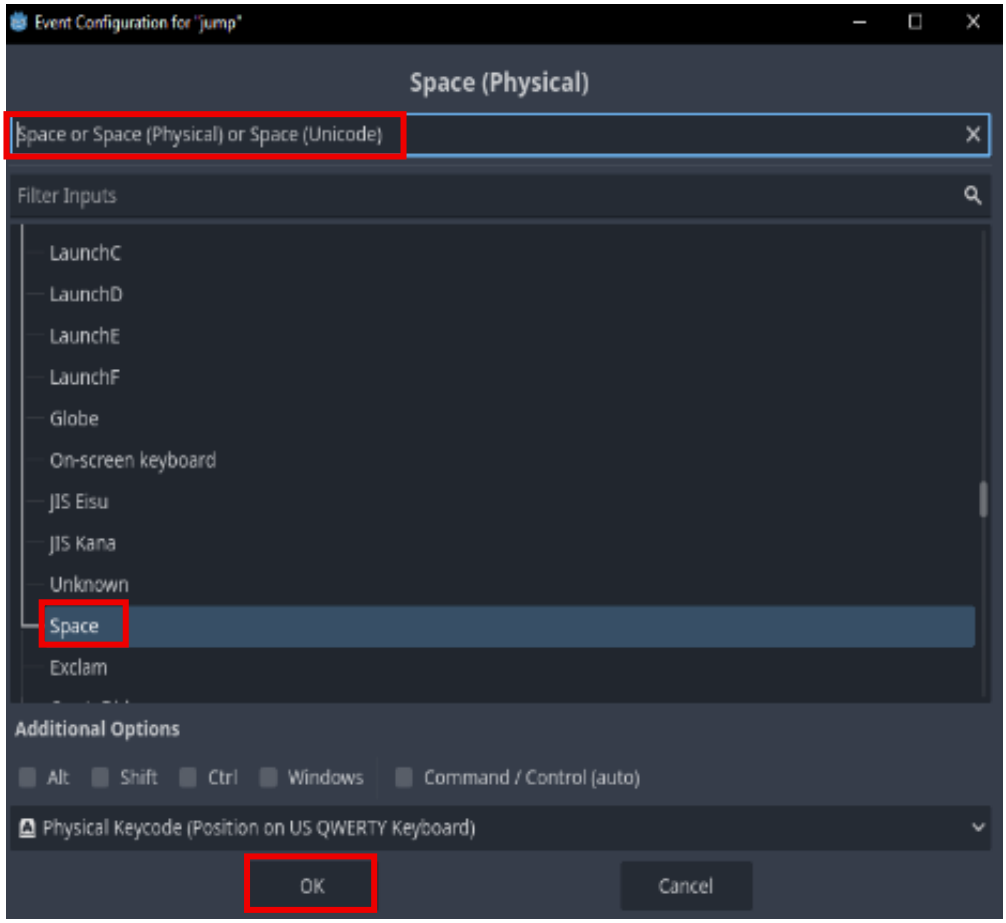
Pro Tip:

The name of the action must match the name used in the code exactly.

15 The jump action exists but still needs an associated key.

To the right of the **jump** action, click **+**. Press the **space bar** on the keyboard, select **Space**, then click **OK**.

Close the **Project Settings** window.



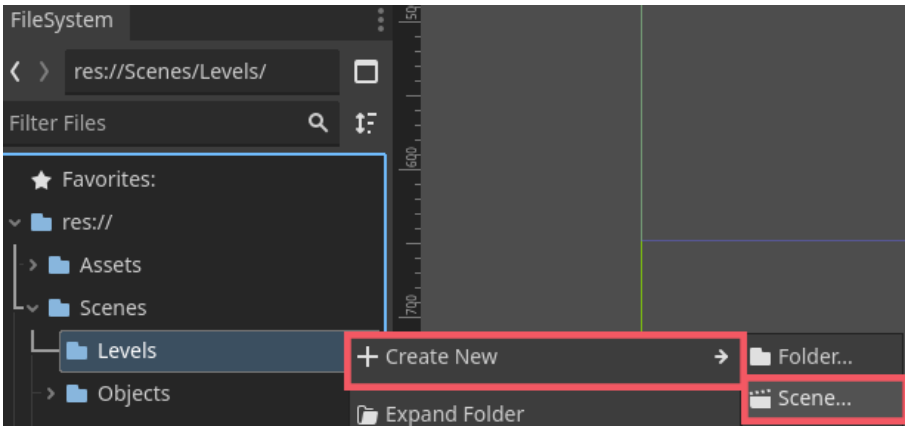
16 Playtest the game and see the player jump when pressing one of the inputs mapped to the **jump** action.

Notice that the Player can jump and the score variable is displayed at the top of the game window. What else might be added for the player to jump over and collect?

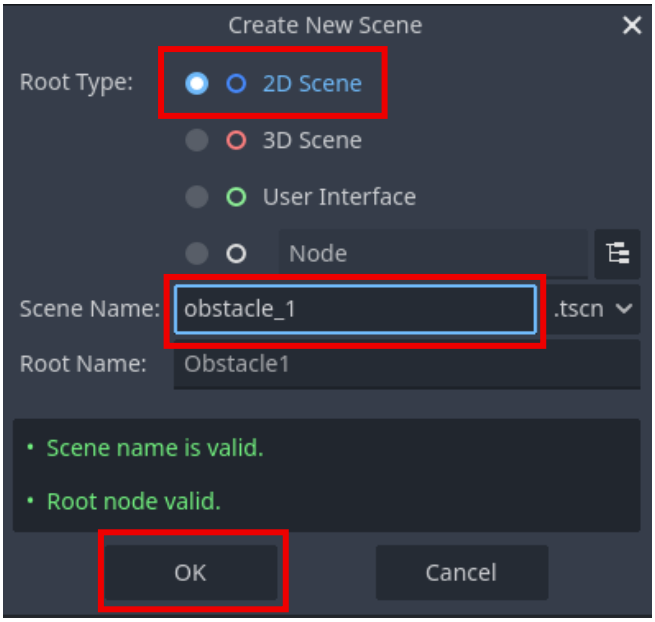
17 Create a new scene to add Obstacles to the game!

In **FileSystem**, open the **Scenes** folder then **right click** the **Levels** sub folder.

Click **Create New**, then select **Scene** from the drop-down menu.



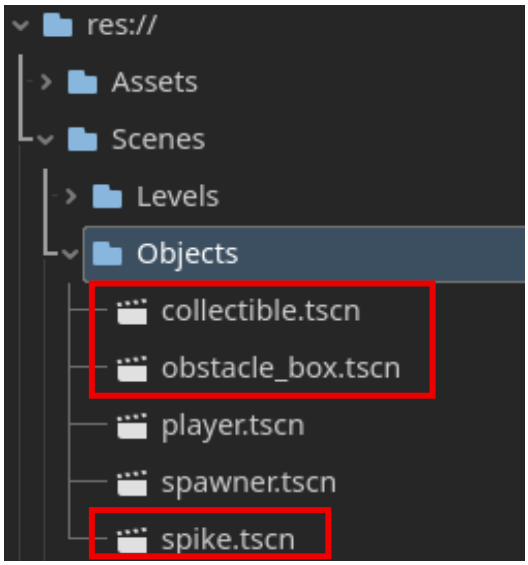
Select **2D Scene** as the Root type, then name the scene **obstacle_1**. Click **OK**.



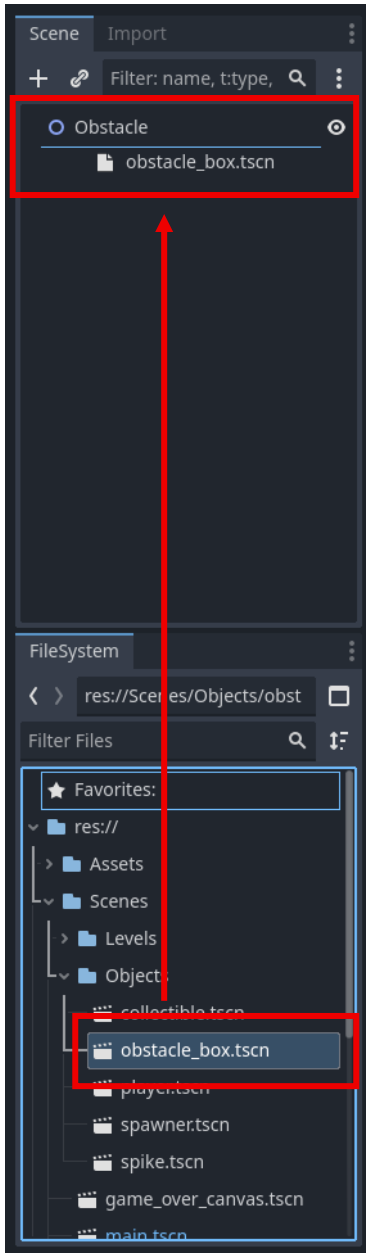
18 Locate the **Objects** subfolder of **Scenes** in the **FileSystem**. Notice that there are five objects inside: **collectible**, **spike**, **obstacle_box, player,** and **spawner**.

The following three will be used in the Obstacle Scene:

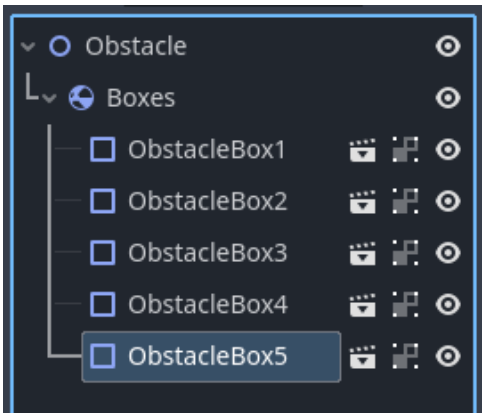
- **Obstacle_box** is the base box for the scene. It is the safe object for the player to touch.
- **Collectible** is an object, like a coin, for the player to collect and gain points.
- **Spike** is a game ending object.



19 In the **Objects** folder, select **obstacle_box.tscn** and drag it over the **Obstacle** root node to add it to the scene.



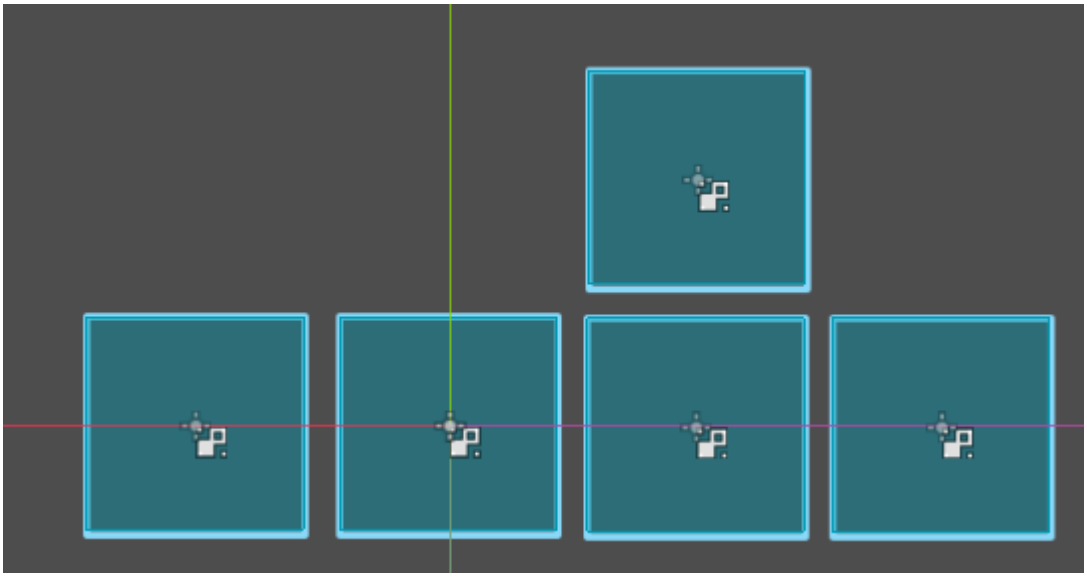
20 Select the **obstacle_box** node in **Scene**, then press **Ctrl + D** to duplicate it. Repeat this until there are 5 **obstacle_box** nodes in the scene.



21 Navigate to the **2D** workspace.

Once all 5 **obstacle_box** nodes are in the scene, use the **Move Tool** to reposition them into the shape shown in the image below.

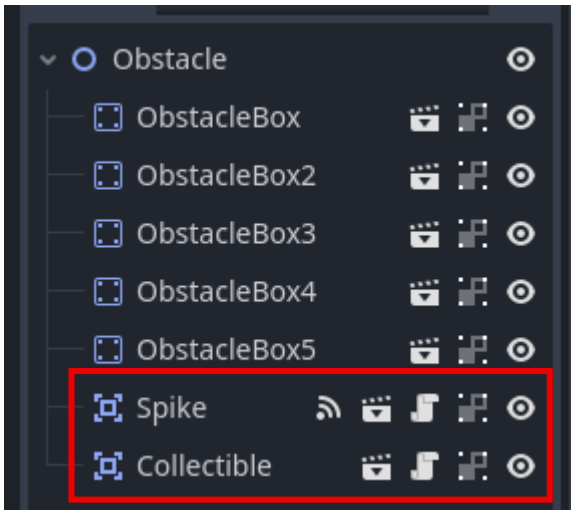
When moving the obstacle boxes, keep the first layer of boxes along the y-axis. This will make the obstacles line up with the ground correctly.



Reminder:

When using the editor Move Tool, click and drag the different axis arrows to move in one direction at a time!

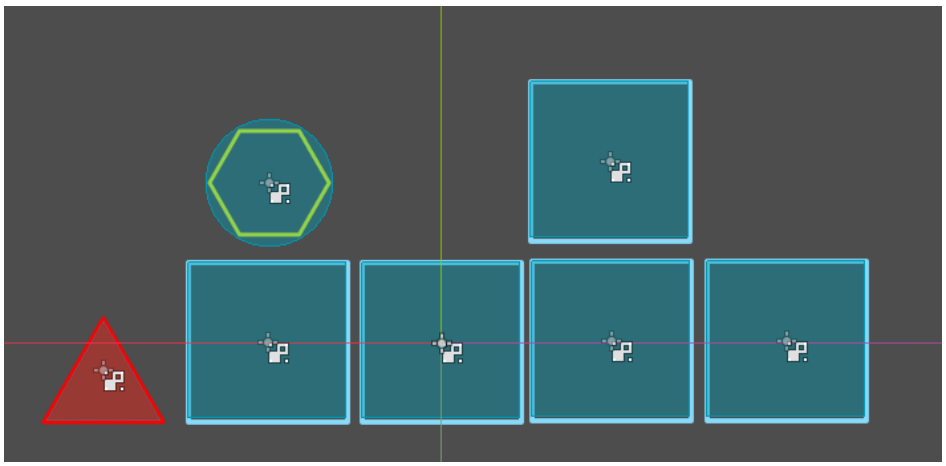
22 Add **spike.tscn** and **collectible.tscn** from the **Objects** folder as child nodes to **Obstacle**.



Reminder:

Drag the file in the Objects folder on top of the Obstacle's child nodes in Scene.

23 Once the spike and collectible are added, position them as shown.



Pause for **Sensei Stop #2**!

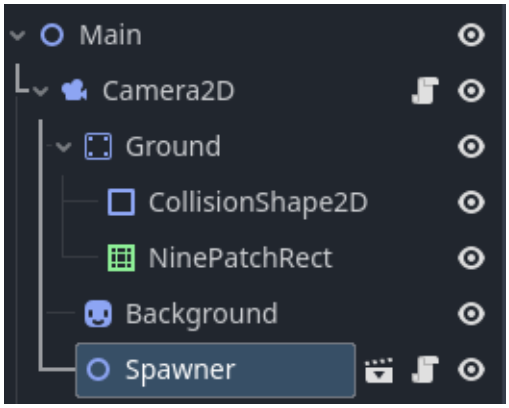
Check in with a Code Sensei before moving on to make sure the **Obstacle** scene node structure and object positions are set up correctly.

Reminder: Save your work!

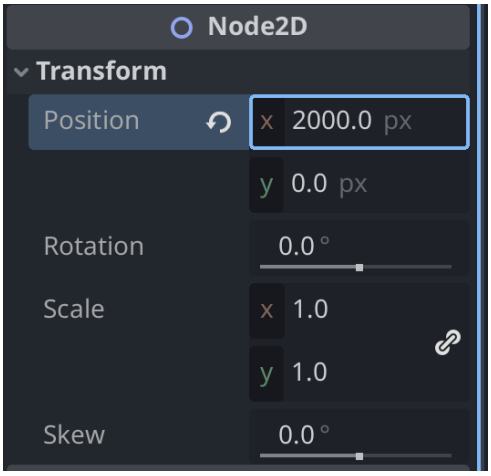
24 With the obstacles created, a spawner is needed to place it into the game scene.

Navigate to the **main** scene.

From the **Objects** folder, add **spawner.tscn** as a child node to **Camera2D**.



In the **Inspector** for the **Spawner** node, set its **x position** to **2000 px** under **Transform**.



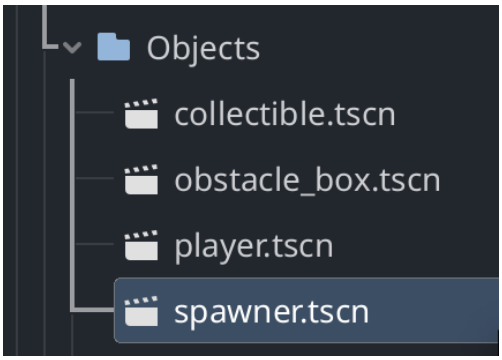
This will make the obstacles spawn in front of the player, outside the view of the camera.

25

Tell the spawner which obstacles to spawn!

From the **Objects** folder, open **spawner.tscn** to modify it.

Note: As the scene is modified and saved, the spawner added to the **main** scene will be updated with changes made as it is still linked to the scene file in the **Objects** folder.



26

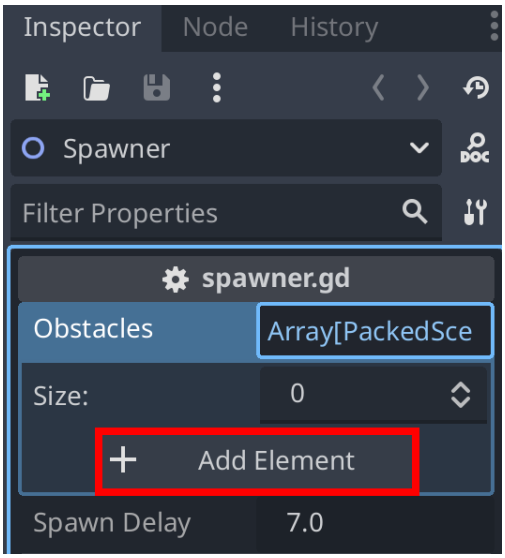
Inside **spawner.tscn**, select the **Spawner** root node.

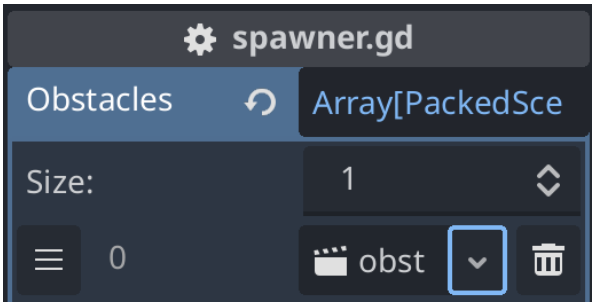
In the **Inspector**, click **Array[PackedScene] (size 0)** next to **Obstacles**.

Click the **Add Element** button.

Drag **obstacle_1.tscn** from the **Levels** folder over the box that says **<empty>**.

This will add **obstacle_1.tscn** to the list of obstacles that can spawn. More obstacles will be added later. For now, it will be only this obstacle.



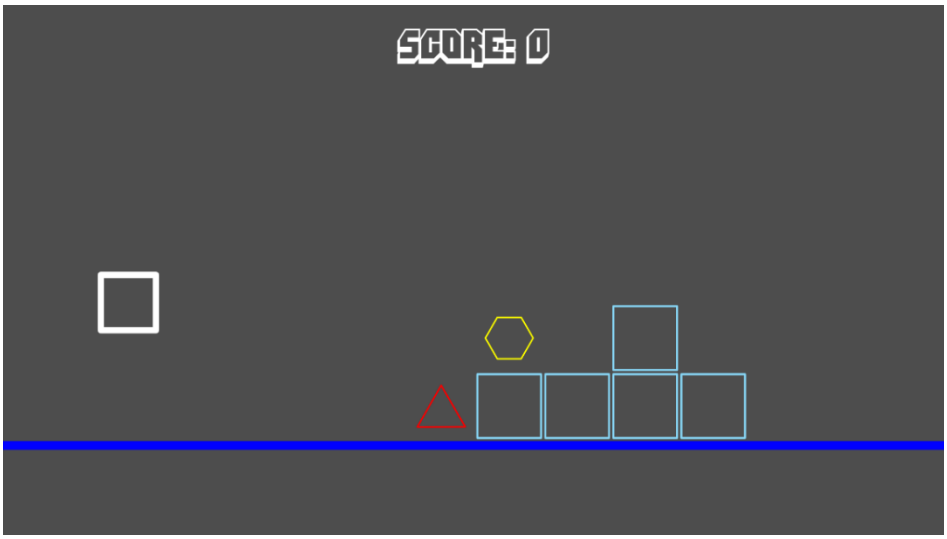


27

Playtest the game!

A new obstacle should spawn every 7 seconds from the right of the screen for the player to platform over.

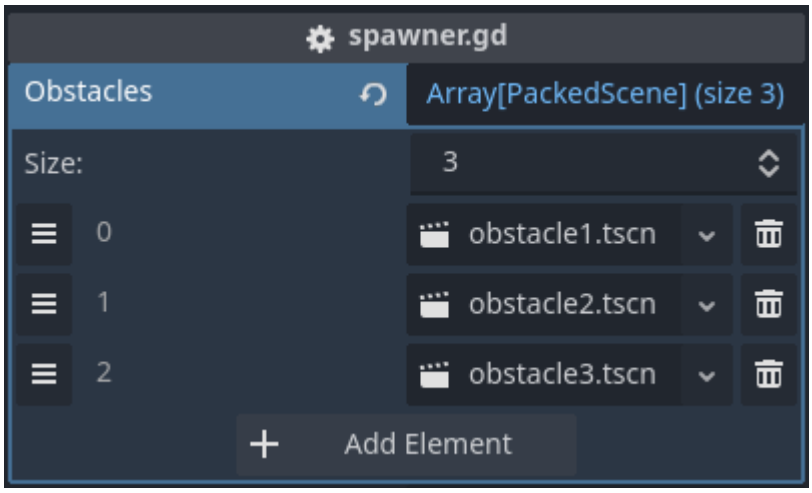
Practice timing jumps to clear obstacles!



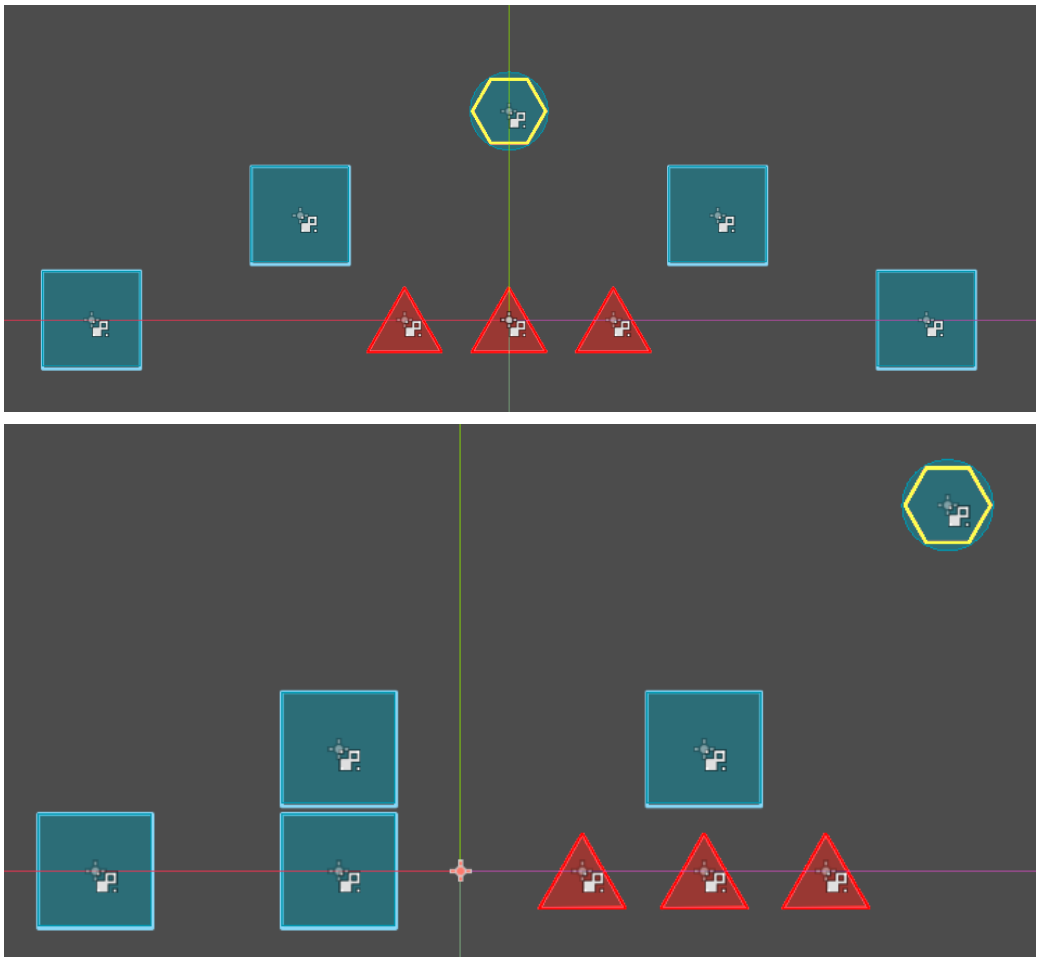
28

Repeat the obstacle steps to create two new custom obstacles!

Using your knowledge of scene creation and how to utilize packed scenes, create an **obstacle_2.tscn** and an **obstacle_3.tscn** and add them to the spawner.



Here are some examples of other obstacles!



Pro Tip:

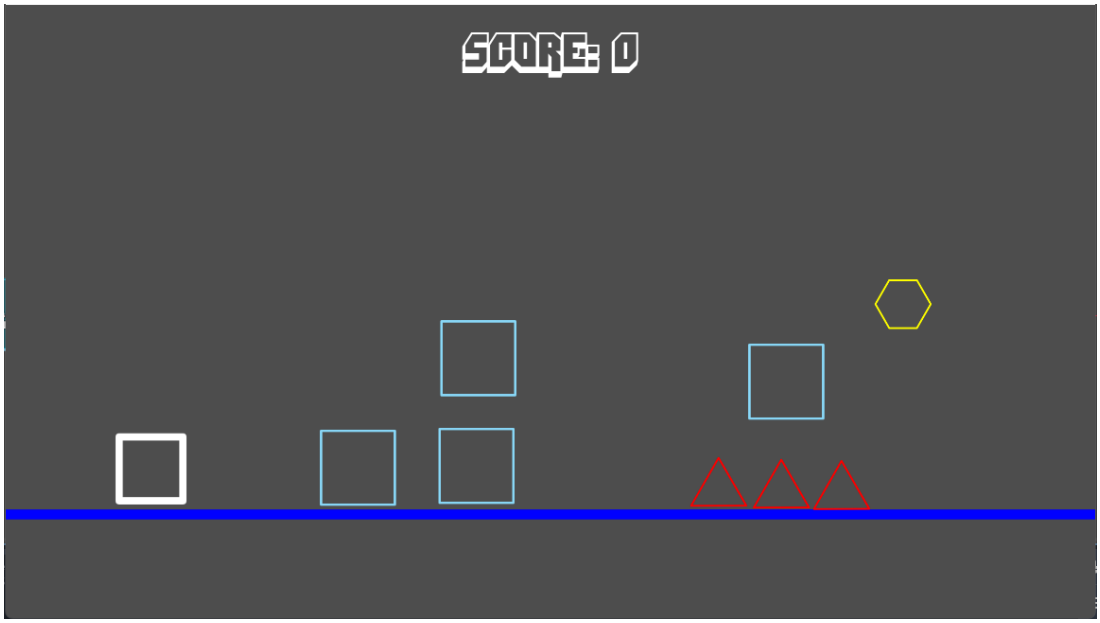
Make sure the new obstacles are possible to navigate with playtesting.

29

Playtest the game!

All three obstacles will now spawn and will be randomized from the spawners list!

Notice that although the obstacles are spawning, the spikes don't end the game and the coins don't increase the score!

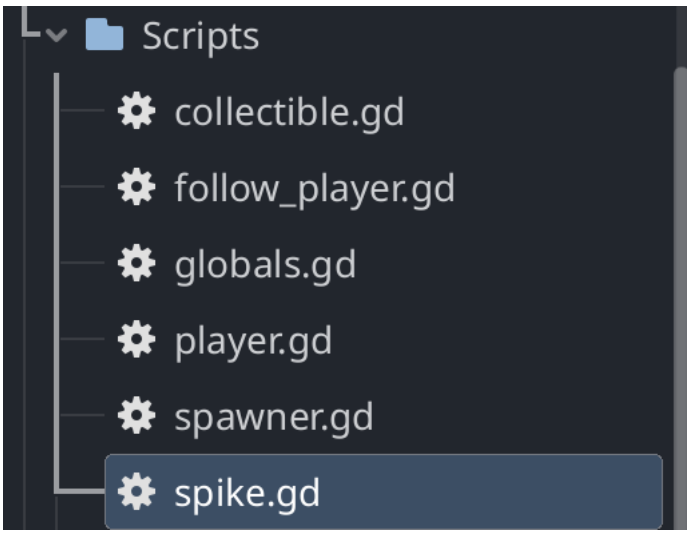


30

Code what happens when the player collides with a spike!

In the **Filesystem**, open the **spike.gd** script located in the **Scripts** folder.

Hint: If the script is not visible, make sure that script workspace is selected.



31 A **game_over** signal should be emitted when the Player collides with a spike.

Inside **spike.gd**, locate the **TODO 3** comment. Underneath, define an `_on_body_entered()` method with the parameter `body`.

```
1  extends Area2D
2
3  # --------------------
4  # TODO 3
5  # Create the _on_body_entered method below
6  # --------------------
7
```

Reminder:

_on_body_entered is a method that default signals create when connecting them through the inspector. The spikes **body_entered** signal is already connected to this function name, it just needs to be implemented.

32 Inside the `_on_body_entered()` method, create an `if` statement that calls the `is_in_group()` method with the parameter `"Player"` to check if the overlapping body is a player.

```
3  # --------------------
4  # TODO 3
5  # Create the _on_body_entered method below
6  # --------------------
7  func _on_body_entered(body):
8      if body.is_in_group("Player"):
9
```

`is_in_group()`: returns **true** if this node has been added to the given **group**.

Parameters:

1. `group` ***(String):*** the group to be checked (`"Player"`).

Returns ***(Boolean):*** true, if the node is found in the group.

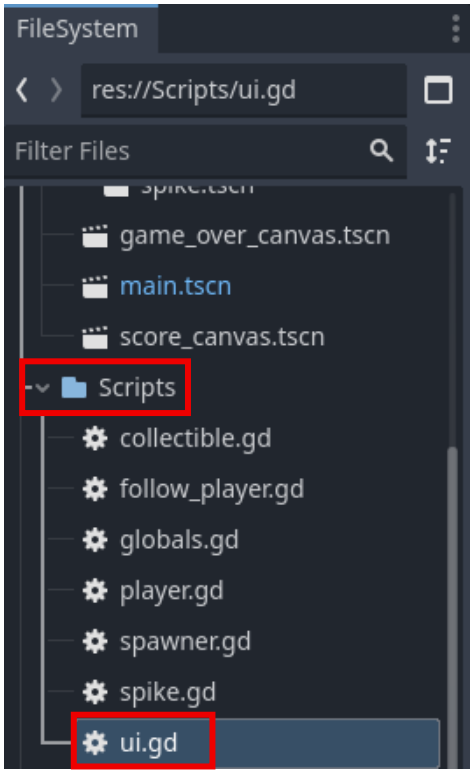
33 Inside the `if` statement access the `Globals` class. Then access the signal `game_end` and call the method `emit()`.

What is the purpose of calling `emit()` here? How does it relate to the structure of the global script?

```
3  # --------------------
4  # TODO 3
5  # Create the _on_body_entered method below
6  # --------------------
7  func _on_body_entered(body):
8      if body.is_in_group("Player"):
9          Globals.game_end.emit()
10
```

34 Update the score when the player collides with a spike!

In **Filesystem**, open the **ui.gd** script located in the **Scripts** folder.



The **ui.gd** script will handle displaying the score and game over message.

35 The **UI script** loads the two canvas nodes `$ScoreCanvas` and `$GameOverCanvas` when the game is run.

These lines will save the two child canvases into the respective variables when the game loads.

```
@onready var score_canvas : Control = $ScoreCanvas
@onready var game_over_canvas : Control = $GameOverCanvas
```

36 Note the function `ui_game_over()` inside the **UI script.** This function is used to end the game when called.

This function freezes the speed of the game, by setting the `time_scale` to `0`, resets the `score`, and makes the `game_over_canvas` visible.

To call `ui_game_over()` when the **global script** emits the `game_end()` signal, the **UI script** must connect the signal and the function in `_ready()`.

```
func ui_game_over() -> void:
    Engine.time_scale = 0
    Globals.score = 0
    game_over_canvas.visible = true
```

Reminder:

The `_ready()` function runs when the node is loaded into the scene.

37 Find the **TODO 4** comment in **ui.gd**.

Underneath, use the code completion to define a `_ready()` function.

```
6  # --------------------
7  # TODO 4
8  # Create the _ready method
9  # --------------------
10 func _ready() -> void:
11
```

38 Inside the `_ready()` function, access the `game_end` signal of `Globals` with `Globals.game_end` and connect that signal to `ui_game_over` with `.connect(ui_game_over)`.

```
10 func _ready() -> void:
11     Globals.game_end.connect(ui_game_over)
```

Pro Tip:

Every signal has a function called `connect()` that takes a callable as a parameter. When called, it will link the signal emitting to the function passed as a parameter.

39 Restart the game when the button on the game over screen is pressed. Use the **pressed** signal to connect to the **restart_game** function. Connect the **pressed** signal of **Button** to the **restart_game** function of the **global script**.

Underneath `Globals.game_end`, call `game_over_canvas`. Use the `get_node(path: NodePath)` method to access the `Button` child of `game_over_canvas`.

Access the `pressed` signal of the **Button** node and connect it to the `restart_game` function of `Globals`.

What will happen when the button node is pressed on the game over screen?

```
10 func _ready() -> void:
11     Globals.game_end.connect(ui_game_over)
12     game_over_canvas.get_node("Button").pressed.connect(Globals.restart_game)
```

40 The **UI script** needs to end the game when the `restart_game()` method is called.

Navigate to the **globals.gd** script.

Underneath the signals below **TODO 5**, define a `restart_game()` function with return type `void`.

```
16  # --------------------
17  # TODO 5
18  # Create the restart_game method
19  # --------------------
20
```

41 Inside the `restart_game()` function, set the `Engine` global script's `time_scale` property to **1**.

Note: Engine is an example of a different global script. It is a default script within every project that Godot uses to manage engine logic.

```
16  # --------------------
17  # TODO 5
18  # Create the restart_game method
19  # --------------------
20  func restart_game():
21      Engine.time_scale = 1
22
```

42 Underneath `Engine.time_scale`, use the code completion to call `get_tree()` to access the scene root.

Call the `reload_current_scene()` method of the scene root. This method resets the scene back to its default state.

```
16  # --------------------
17  # TODO 5
18  # Create the restart_game method
19  # --------------------
20  func restart_game():
21      Engine.time_scale = 1
22      get_tree().reload_current_scene()
23
```

43 Navigate to the **player.gd** script.

Find the **TODO 6** comment.

Below, within the `_physics_process` function, create an `if` statement that checks if the player `is_on_wall()`.

```
26    # --------------------
27    # TODO 6
28    # Connect game_end & score_update signal
29    # --------------------
30    if is_on_wall():
31
```

`is_on_wall()`: Part of the CharacterBody2D/3D classes, returns true if the body collided with a wall on the last call of `move_and_slide()`. Otherwise, returns false. The `up_direction` and `floor_max_angle` are used to determine whether a surface is "wall" or not.

Parameters: None

Returns *(Boolean):* true, if the body last collided with a wall.

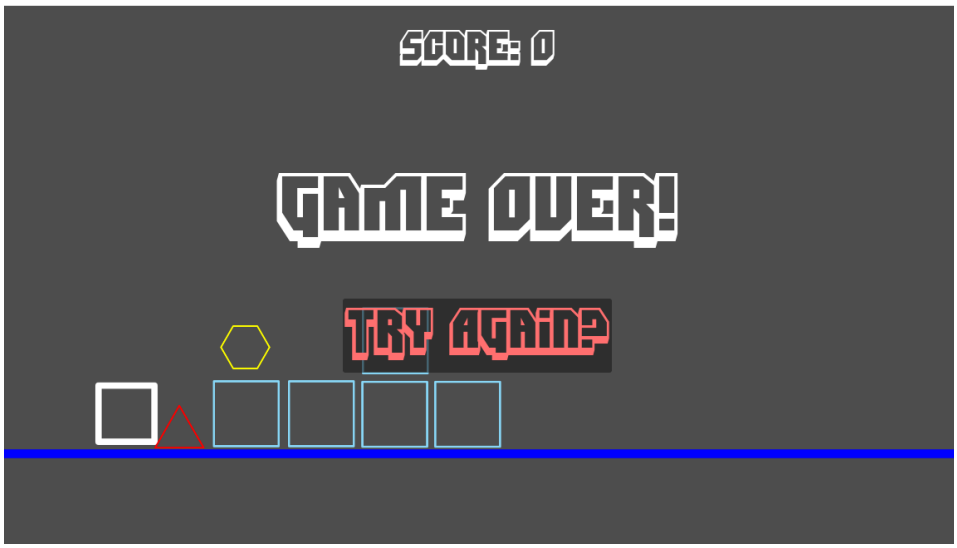
44 Inside the `if` statement, access the `game_end` signal of `Globals` and call the `emit()` method.

```
26    # --------------------
27    # TODO 6
28    # Connect game_end & score_update signal
29    # --------------------
30    if is_on_wall():
31        Globals.game_end.emit()
32
```

45 Playtest the game!

When the player touches a wall, the game will stop and the game over screen will appear.

What happens when the player collides with a collectible? Does the score go up? If not, consider what could be done to fix that!



Pause for **Sensei Stop #3**!

Why might the game over signal be called inside the `if is_on_wall` statement? How does it relate to the previously created signals?

Reminder: Save your work!

46 The **globals script** needs a function to change the score and tell other nodes it has changed by emitting `score_updated`.

Navigate to **globals.gd**. Find the **TODO 7** comment.

Underneath, define a `change_score` function that takes a `value` parameter of type `int`.

```
24  # --------------------
25  # TODO 7
26  # Create the change_score method
27  # --------------------
28  func change_score(value: int):
29
```

47 Inside the `change_score` function, increase `score` by `value`.

This method will be called from other places in the code, such as the collectible. When it is called, an integer `value` must be passed as a parameter. This integer will be added to score.

```
24  # --------------------
25  # TODO 7
26  # Create the change_score method
27  # --------------------
28  func change_score(value: int):
29      score += value
30
```

Pro Tip:

If `value` is negative, the score will decrease!

48 Inside the `change_score` function, call `score_updated`. Then, have it `emit` `score`.

When the `score_updated` signal is emitted, every method that has connected to it will run.

The connected functions can then perform different logic. In this game the score canvas updates. In other games a high score or leaderboard might be updated instead!

```
24  # --------------------
25  # TODO 7
26  # Create the change_score method
27  # --------------------
28  func change_score(value: int):
29      score += value
30      score_updated.emit(score)
31
```

49 The collectible currently has no functionality!

In **FileSystem**, open the **collectible.gd** script and navigate to **TODO 8**.

Create a new function called `_on_body_entered()` that takes a `body` parameter**.**

This function will be connected to a default signal of **Area2D** to trigger logic when touched.

```
3  # --------------------
4  # TODO 8
5  # _on_body_entered will auto create below when connected
6  # --------------------
7  func _on_body_entered(body):
8
```

50 The score should only go up when the **Player** overlaps the collectible!

Inside the `_on_body_entered()` function, create an `if` statement that calls the method `is_in_group()` with the parameter `"Player"` to check if the overlapping body is a player.

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

Pro Tip:

All nodes have a method **`is_in_group`** that takes a string and returns a **`true`** boolean if that node is in the group of the given string.

51 Inside the if statement, call **`Globals`** and the **`change_score`** method. Change the score by **`1`**.

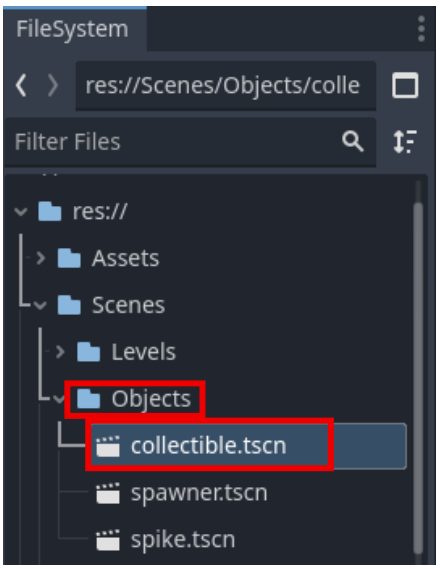
Calling this function will both add **1** to the score and emit the signal **score_updated** as discussed in step 48.

```
func _on_body_entered(body):
    if body.is_in_group("Player"):
        Globals.change_score(1)
```

52 Underneath **`Globals.change_score(1)`**, call **`queue_free()`** to destroy the collectible when the Player has collided with it.

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

53 In the **FileSystem**, navigate to the **Objects folder.** Open **collectible.tscn.**



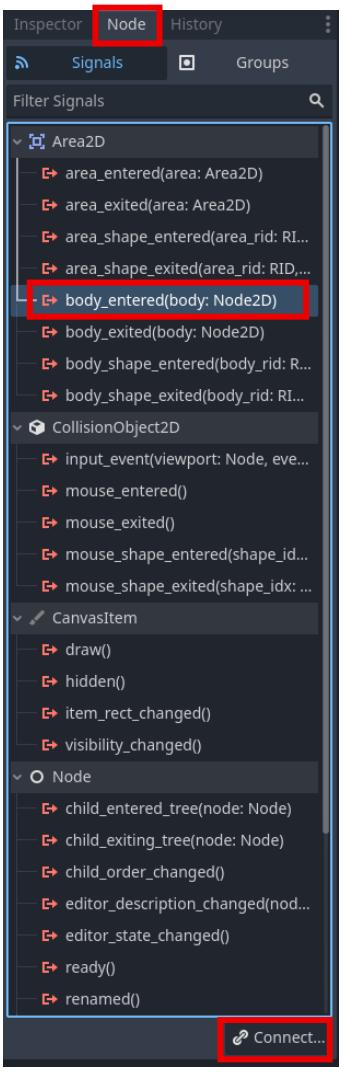
Reminder:

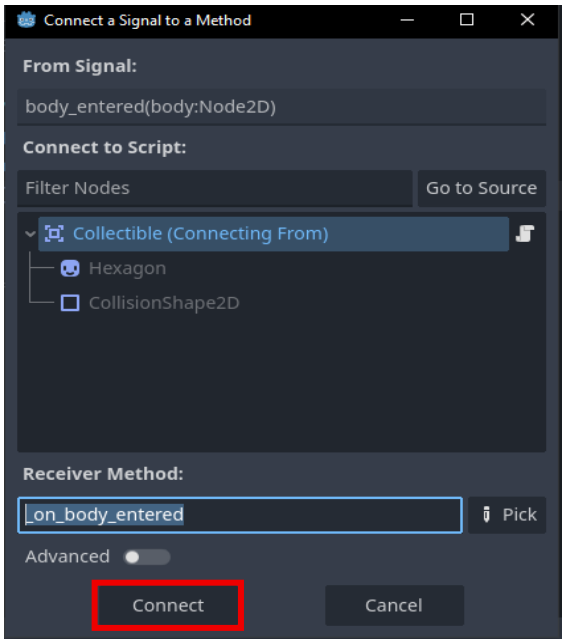
In Godot, a game can have many different scenes, such as levels, menus, or cutscenes. In **FileSystem**, all scenes are saved in the Objects or Levels folders. Double click to open any file that ends in **.tscn**. Use the scene tab at the top of the scene viewport to navigate between all active assets.

54 Select the **Collectible** node in **Scene**. In the **Inspector**, select **Node.**

Under **Area2D**, locate **body_entered.** Click on it. The click **Connect...**

In the **Connect a Signal to a Method** window, click **Connect.**



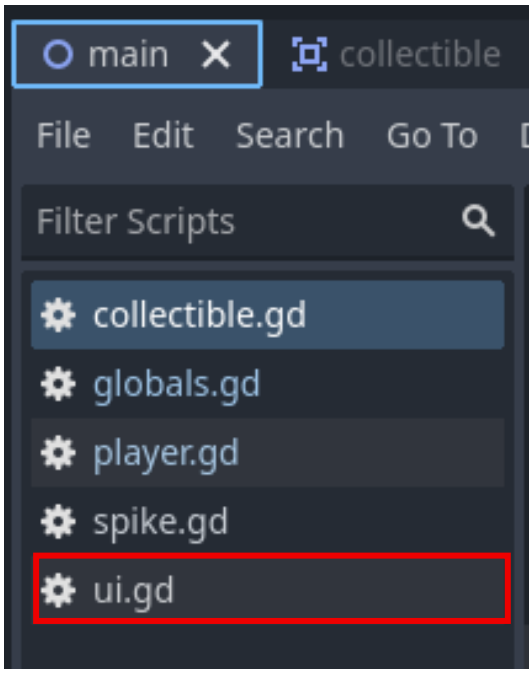


55 The function `_on_body_entered()` should now have a green icon next to it. This icon shows that a signal has been connected to the code.

The signal just connected is a property of **Area2D** nodes that detects when a **PhysicsBody** has entered its bounds. The connected method will now run when that signal emits from the **Area2D**.

```
7  func _on_body_entered(body):
8      if body.is_in_group("Player"):
9          Globals.change_score(1)
10         queue_free()
```

56 Navigate to the **ui.gd** script.



57 Locate the `ui_update_score()` function.

This function will be connected to the `score_updated` signal of the **global script**.

When `score_updated` is emitted, the function will access the child node **Label** and update the text to display the current score.

```
20  func ui_update_score(score: int) -> void:
21      score_canvas.get_node("Label").text = "Score: " + str(score)
22
```

58 Locate the previously created `_ready()` method in the script.

Access the `score_updated` signal in `Globals` and connect it to `ui_update_score`.

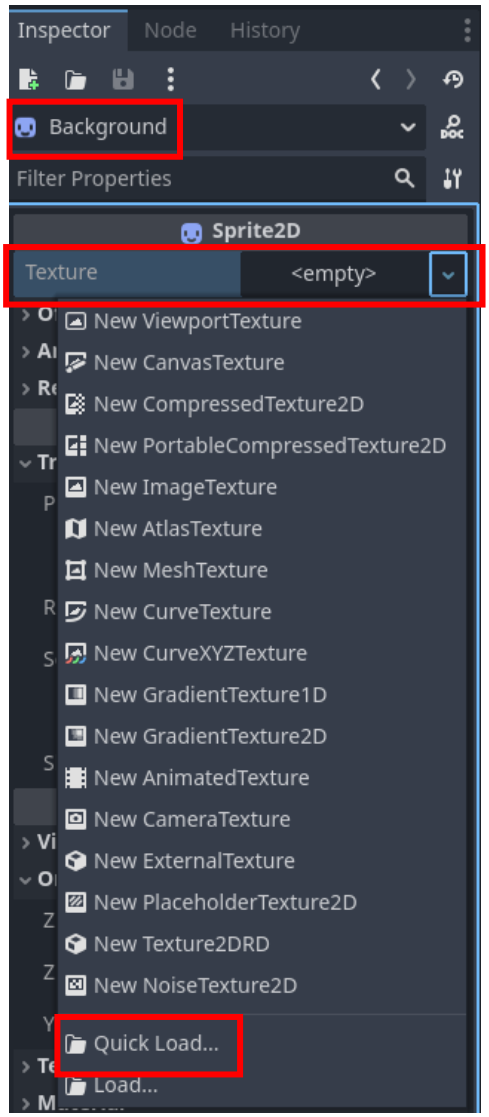
```
10  func _ready():
11      Globals.game_end.connect(ui_game_over)
12      game_over_canvas.get_node("Button").pressed.connect(Globals.restart_game)
13
```

59 Check that the signal connection from the previous step was correctly implemented.

```
10  func _ready() -> void:
11      Globals.game_end.connect(ui_game_over)
12      game_over_canvas.get_node("Button").pressed.connect(Globals.restart_game)
13      Globals.score_updated.connect(ui_update_score)
```

60 Complete the project by adding texture to the Background.

Return to the **main** scene and select the **Background** node. In the **Inspector**, find **Texture**. Click **<empty>** and select **Quick Load**. Select **PolyRunBackground** as the texture.



Pause for **Sensei Stop #4**!

Before submitting, check in with a Code Sensei to make sure the obstacles, score, and game over functionality work correctly then reflect on the following:

- How might global scripts and signals enable the implementation of unique features in a project?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!