CODE NINJAS®

Bronze Belt Ninja Guide

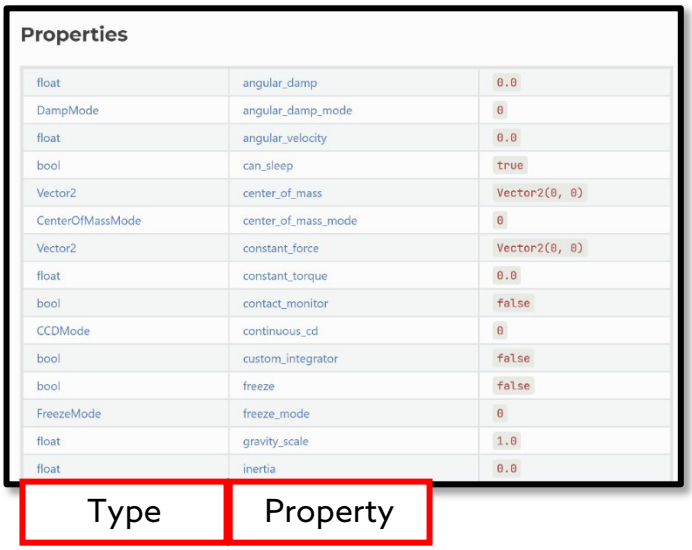
Activity 03: Meany Bird

Classes, Methods, and Inheritance

A **class** is like a blueprint for an object. All nodes in Godot are objects that have their own class, which allows each node to have its own set of properties. Developers can choose different objects based on their coding needs.

The Godot documentation shows all the properties contained in a node class. Some of these properties can be adjusted in the Inspector, others will be saet in the code.

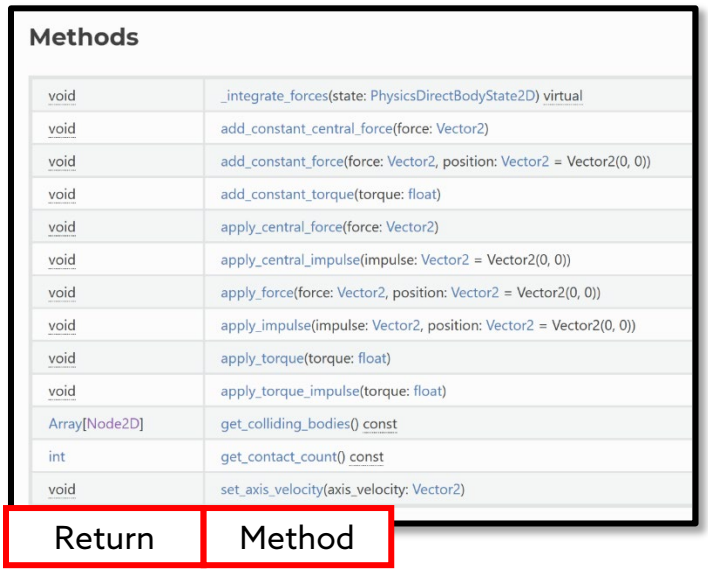
Open the documentation for a **Rigidbody2D** here.



Properties

float	angular_damp	0.0
DampMode	angular_damp_mode	0
float	angular_velocity	0.0
bool	can_sleep	true
Vector2	center_of_mass	Vector2(0, 0)
CenterOfMassMode	center_of_mass_mode	0
Vector2	constant_force	Vector2(0, 0)
float	constant_torque	0.0
bool	contact_monitor	false
CCDMode	continuous_cd	0
bool	custom_integrator	false
bool	freeze	false
FreezeMode	freeze_mode	0
float	gravity_scale	1.0
float	inertia	0.0

Type | Property



Methods

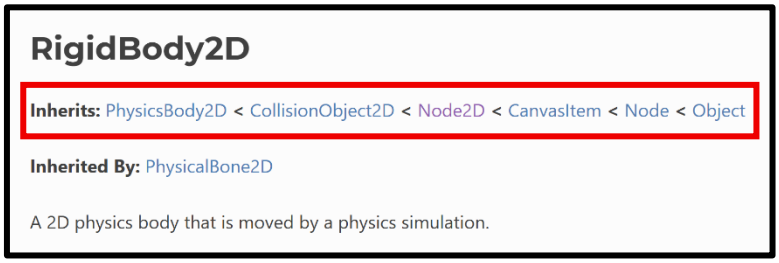
void	_integrate_forces(state: PhysicsDirectBodyState2D) virtual
void	add_constant_central_force(force: Vector2)
void	add_constant_force(force: Vector2, position: Vector2 = Vector2(0, 0))
void	add_constant_torque(torque: float)
void	apply_central_force(force: Vector2)
void	apply_central_impulse(impulse: Vector2 = Vector2(0, 0))
void	apply_force(force: Vector2, position: Vector2 = Vector2(0, 0))
void	apply_impulse(impulse: Vector2, position: Vector2 = Vector2(0, 0))
void	apply_torque(torque: float)
void	apply_torque_impulse(torque: float)
Array[Node2D]	get_colliding_bodies() const
int	get_contact_count() const
void	set_axis_velocity(axis_velocity: Vector2)

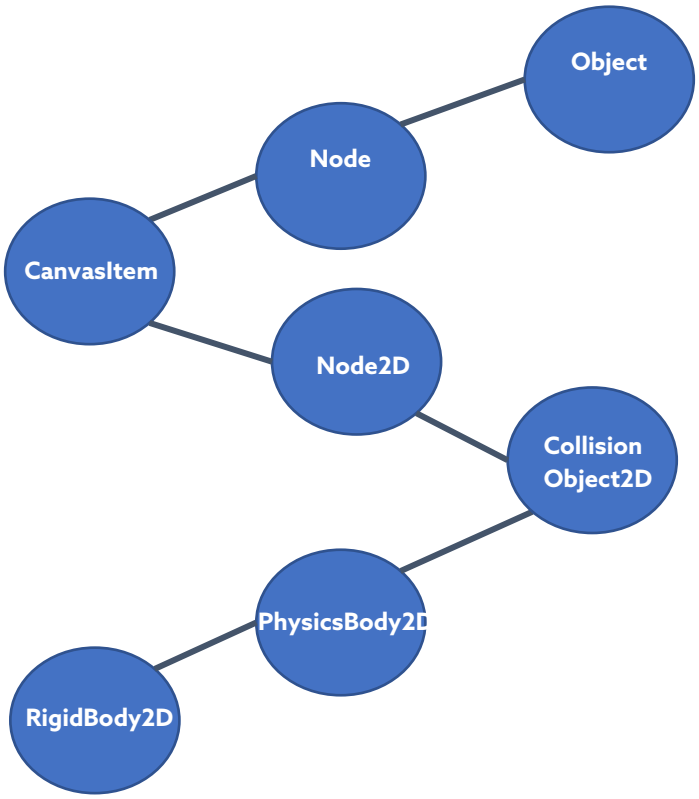
Return | Method

An object's class might contain **methods** as part of that class.

A **method** is a function that is defined in a class. These methods are like built-in functions that developers might often use when working with a node class. The Godot documentation shows the methods defined in the Node2D class, and what each method returns. Methods that don't return anything are void.

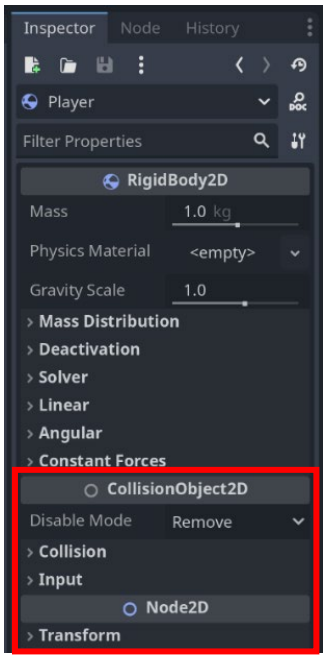
Inheritance allows a node class to **inherit** properties and methods from other node classes. The Godot documentation shows which classes a node inherits and the classes that node is inherited by.





A **RigidBody2D** directly inherits from the **PhysicsBody2D** class and receives all the properties and methods from the class plus its inherited properties.

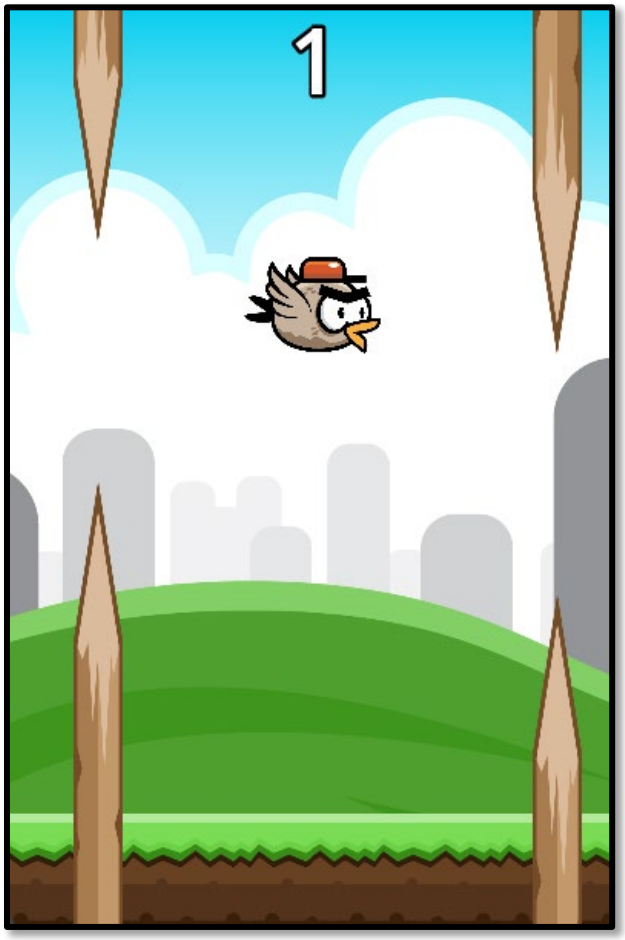
Some of these inherited properties can be seen in the inspector under their respective categories.



Activity 03: Meany Bird

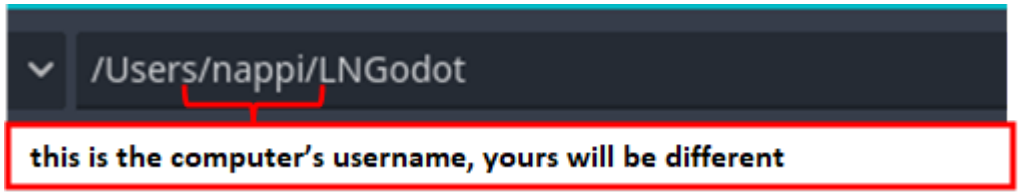
In this project, you will create a side scrolling game where the user must avoid obstacles to earn points! This game will help you learn about basic scripting in Godot.

By the end of this activity, you will have explored inheritance, coded scripts, and connected signals.

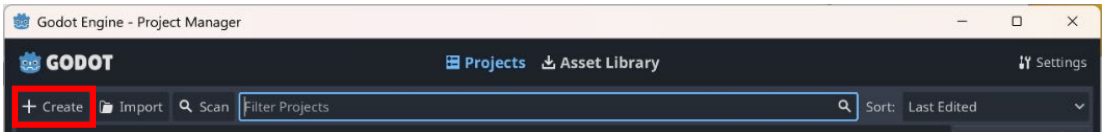


1 Remember all projects will be stored in a path like:
/Users/[MyComputerUsername]/[MyInitials]Godot

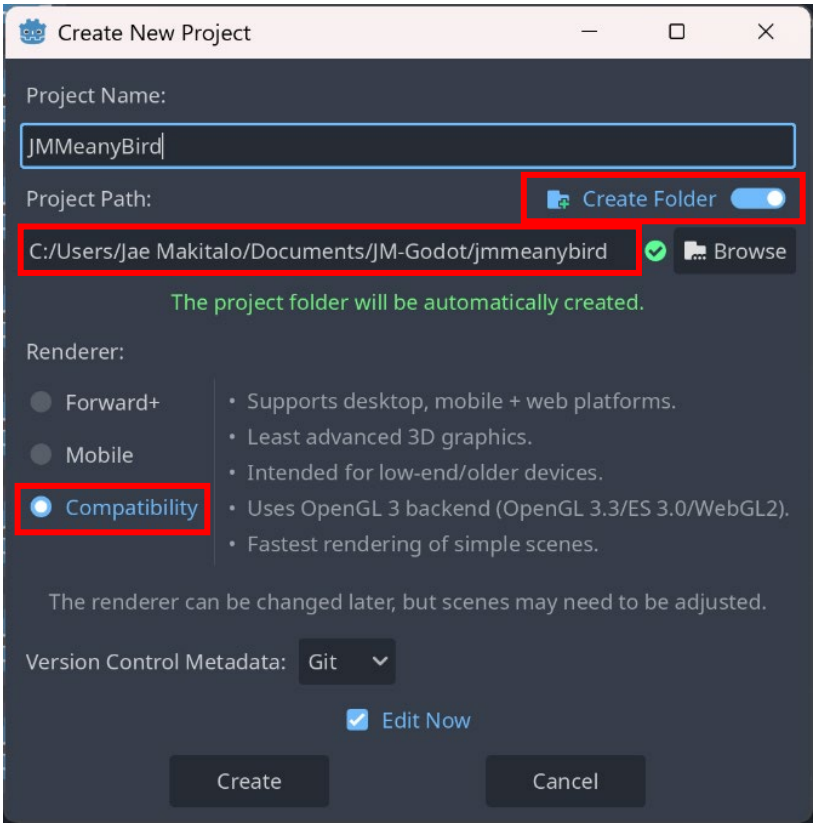
Don't worry if your path looks slightly different from the picture shown! All computers have their own username.



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



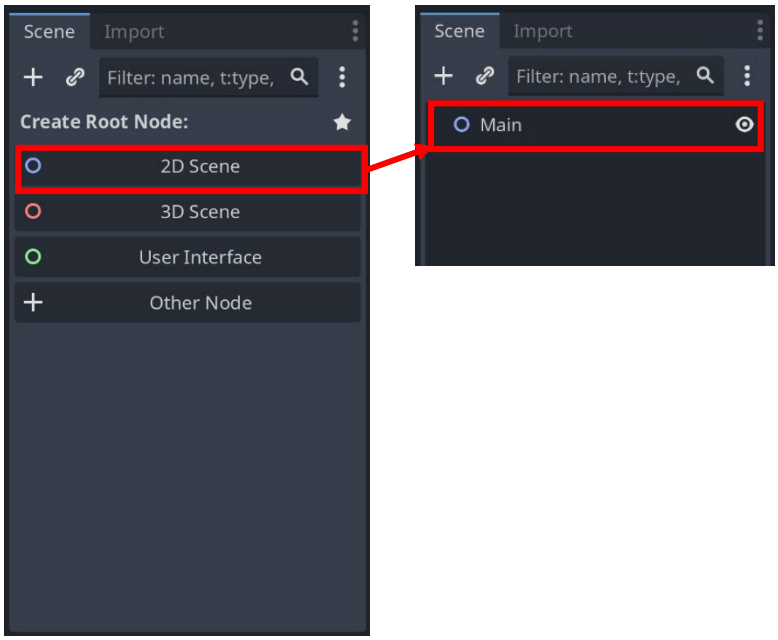
3 Name the project **[MyInitials]MeanyBird** and adjust the project path so the project is being saved in your folder. Make sure **Create Folder** is toggled on, set the renderer to **Compatibility,** then click **Create**.



4

A **Main root node** and **main scene** need to be set for the project. This project is set in a **2D environment** so all nodes will be **2D**.

Select the **2D Scene** as the **root node** for this project. Rename the **Node2D** to **Main**.



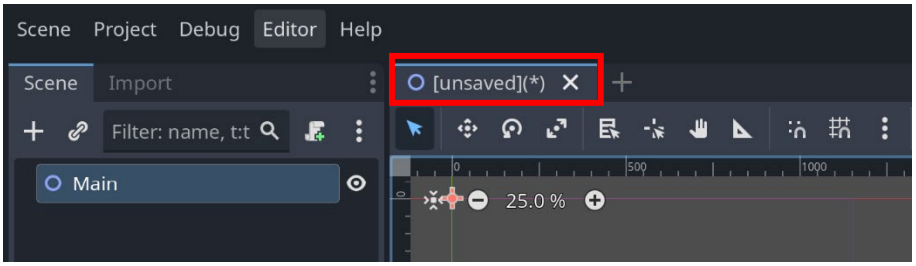
Reminder:

Rename a node by double-clicking on it.

5

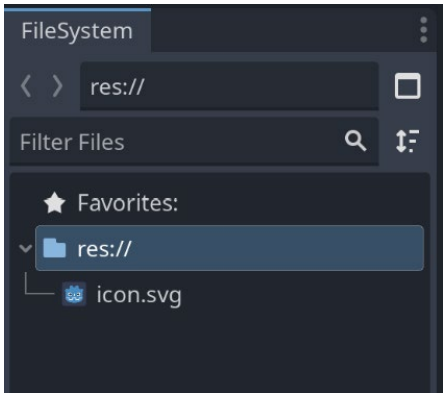
The **main scene** needs to be saved.

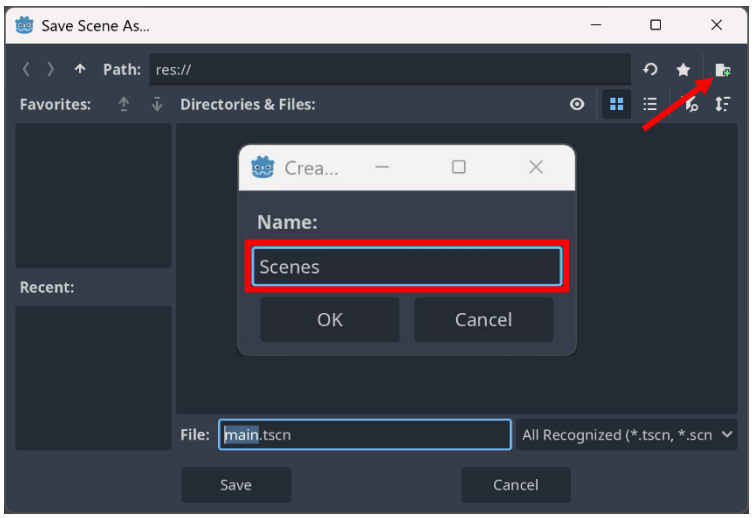
On the keyboard, press **CTRL + S** on to save the scene.



6 This project will contain multiple scenes, so use a **Scenes folder** to stay organized.

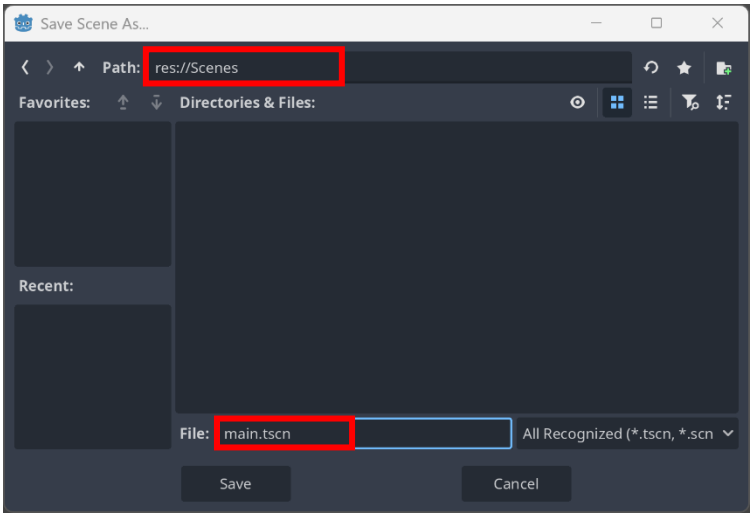
Currently, there are no folders in the **FileSystem**. Click the **new folder icon,** name the new folder **Scenes** and click **Ok**.

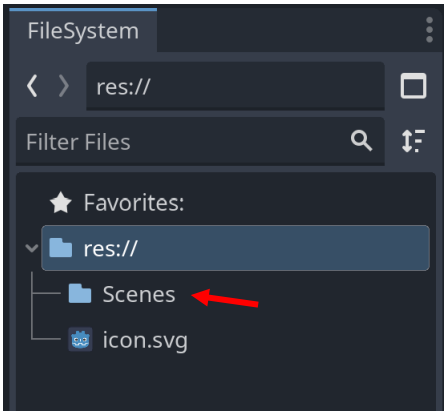




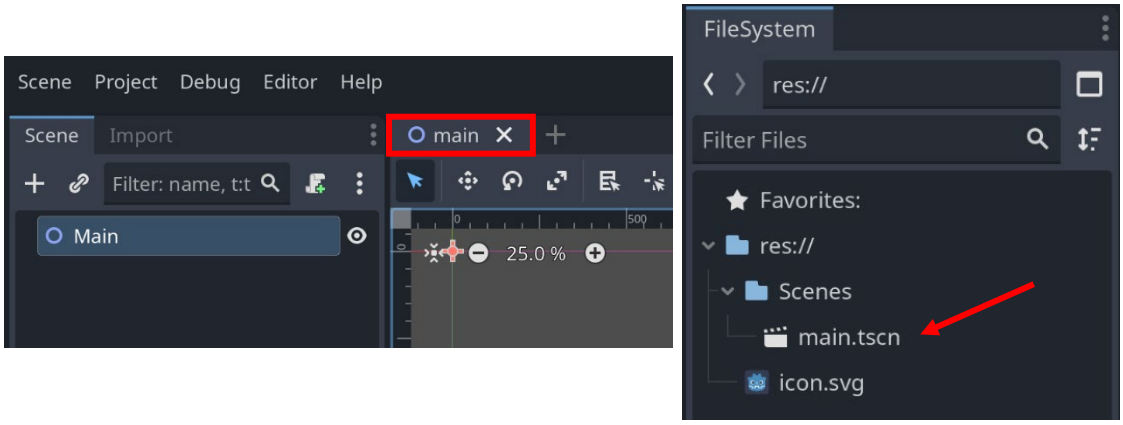
7 The **path** will update from res:// to **res://Scenes** and the Scenes folder will appear in **FileSystem**.

Check that the file is called **main.tscn** and click save.





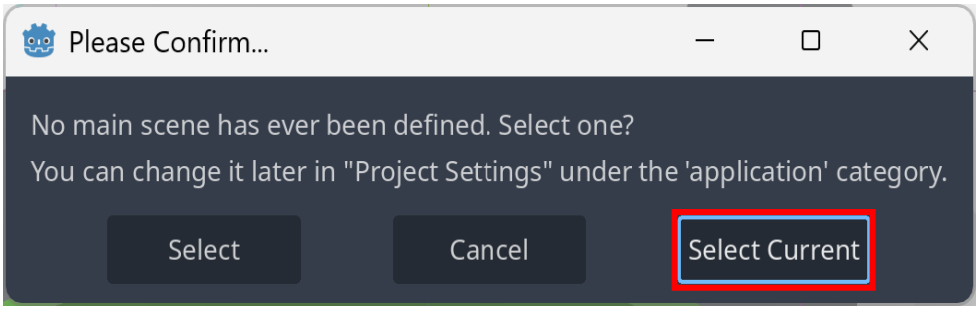
8 The main scene has been saved. Click the **>** arrow beside the **Scenes** folder to see **main.tscn** inside.



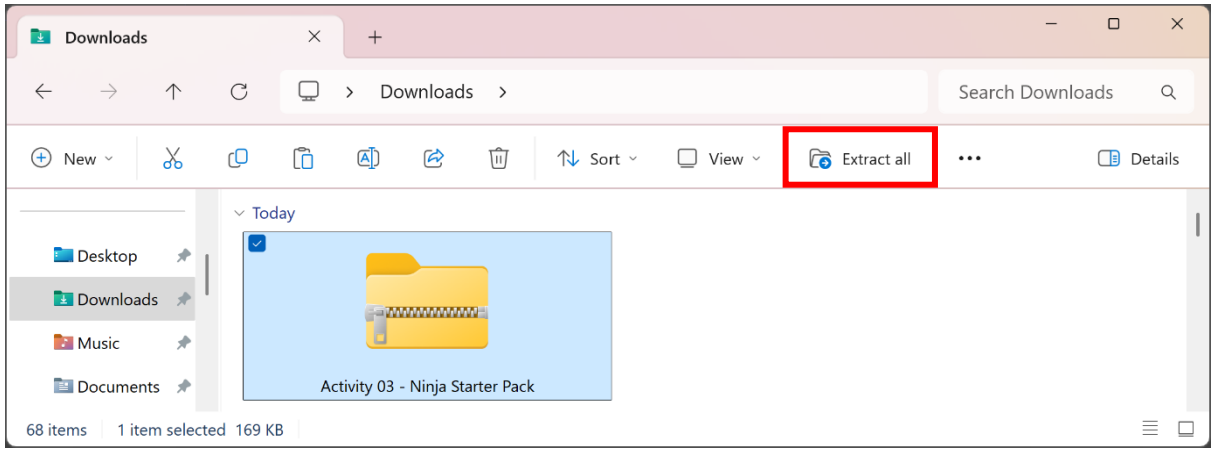
9 Click the **play** button to **playtest** the project.

A window will pop up asking to define the **main scene**. Click **Select Current**.

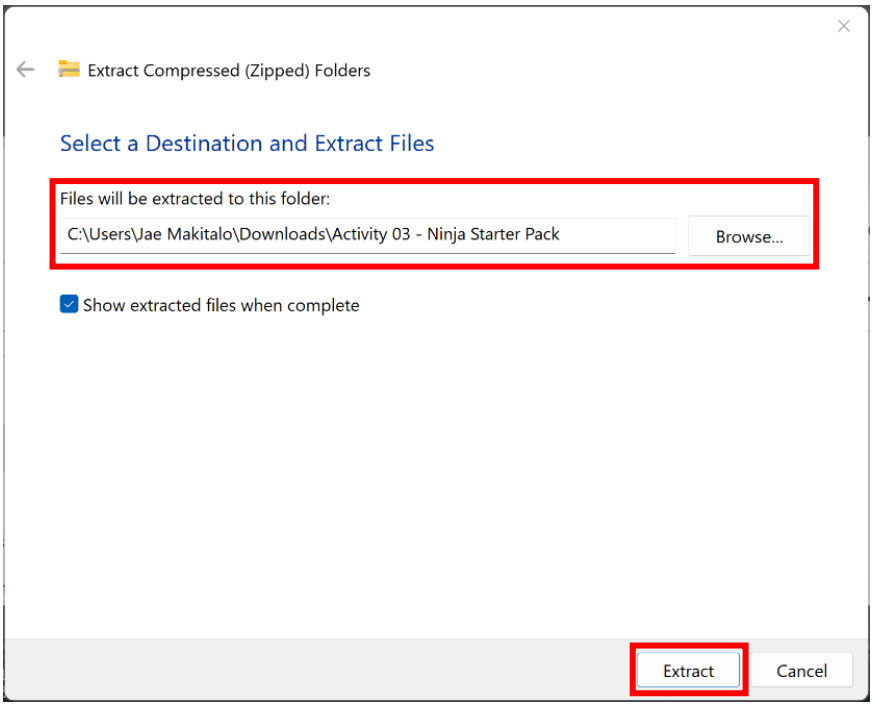
Close the playtest window.



10 Download **BB Activity 03 – Ninja Starter Pack.zip.** Open the file explorer and find the zipped starter code. Select on the folder and click **Extract all**.

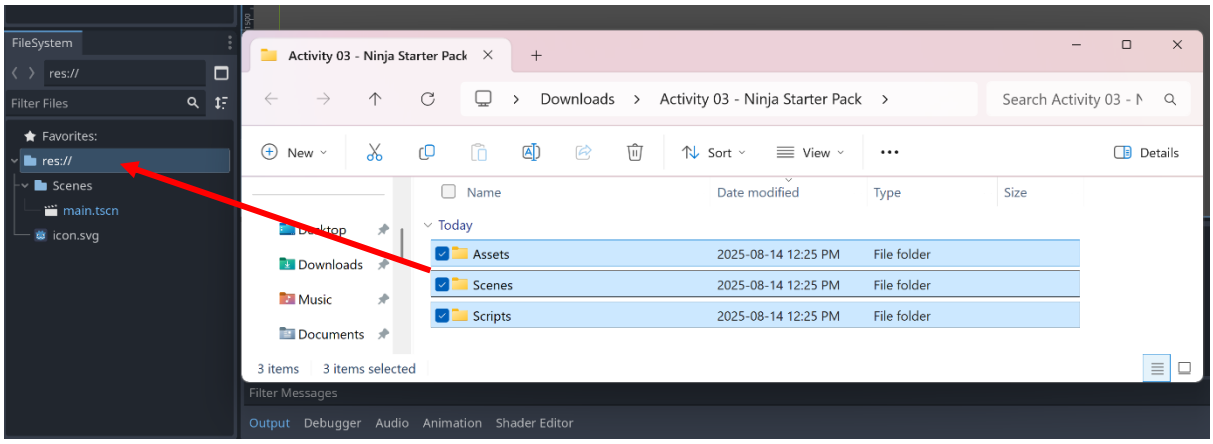


11 Select the file destination by adjusting the path or browsing the file explorer and click **extract**.

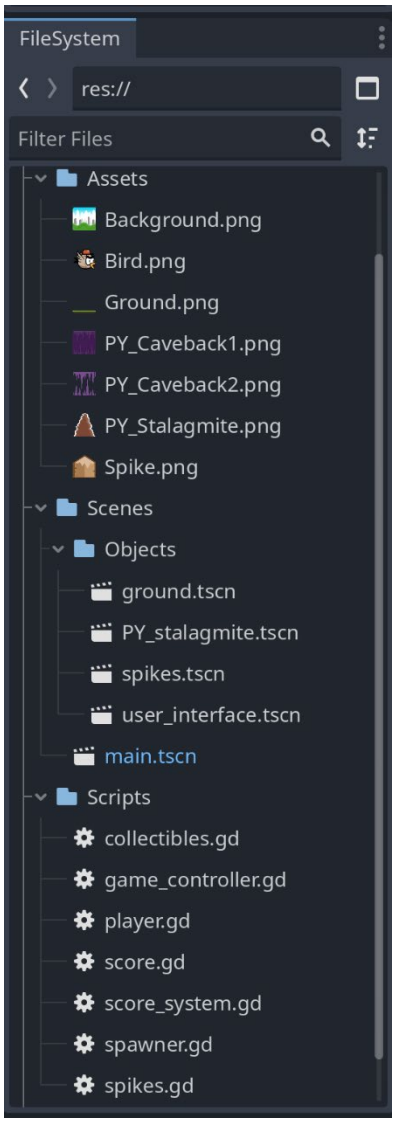


12 In Godot, select the **res://** folder in **FileSystem**. This is where the starter code will go.

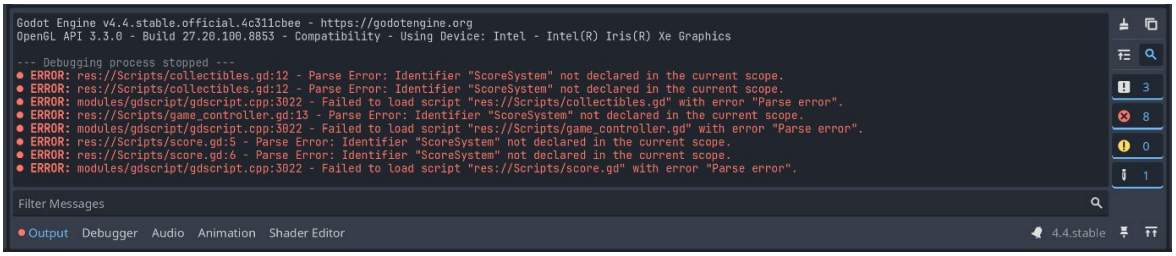
In the file explorer, open the starter code folder, select the **Assets, Scenes** and **Scripts** folders and drag them onto **res://** in **FileSystem**.



13 Open the **Assets**, **Scenes**, **Objects** and **Scripts** folders in **FileSystem** and check to make sure that all the starter code has been imported.



14 Some errors may pop up in Godot's debugger. These can be ignored – they will resolve themselves later.



Pause for **Sensei Stop #1**!

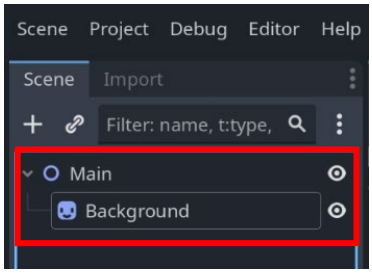
Before continuing, check with a Code Sensei and make sure **main scene** is set up and all files are imported correctly.

Reminder: Save your work!

15

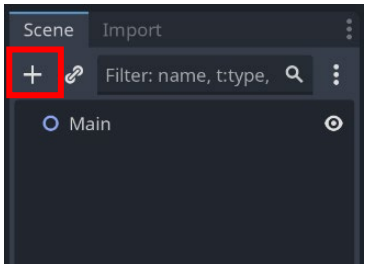
Add a background image to the project.

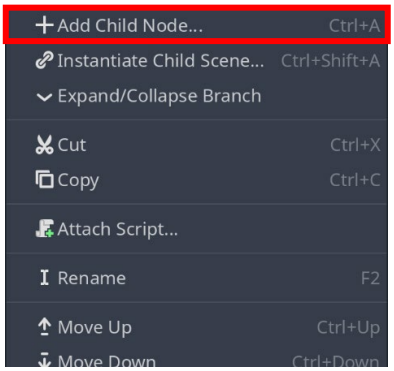
Add a **Sprite2D** as a child node to **Main**. Rename the new node to **Background**.



Remember, Nodes can be added by doing any of the following:

- Selecting the parent node and pressing **CTRL + A** on the keyboard
- Clicking the **+** button
- **Right-clicking** on the parent node and selecting **Add Child Node**

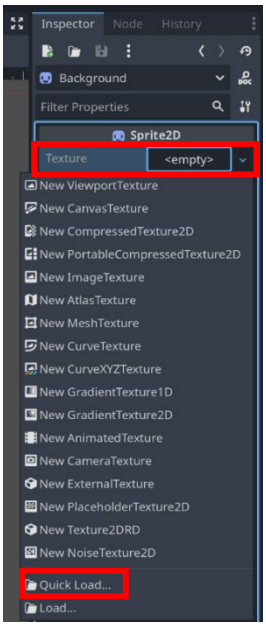




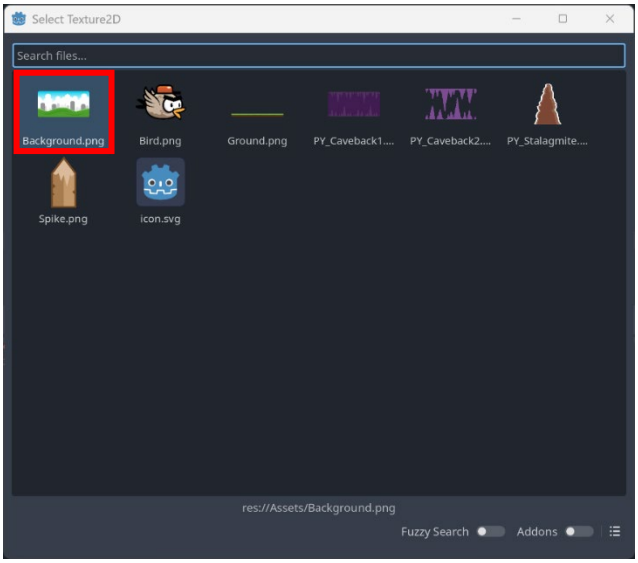
Reminder:

A Sprite2D is a node that displays a 2D texture, such as an image.

16 In the **Inspector** for **Background**, locate **Texture <empty>**. Select the **drop-down arrow** and select **Quick Load. Quick Load** will show all possible textures in the directory.



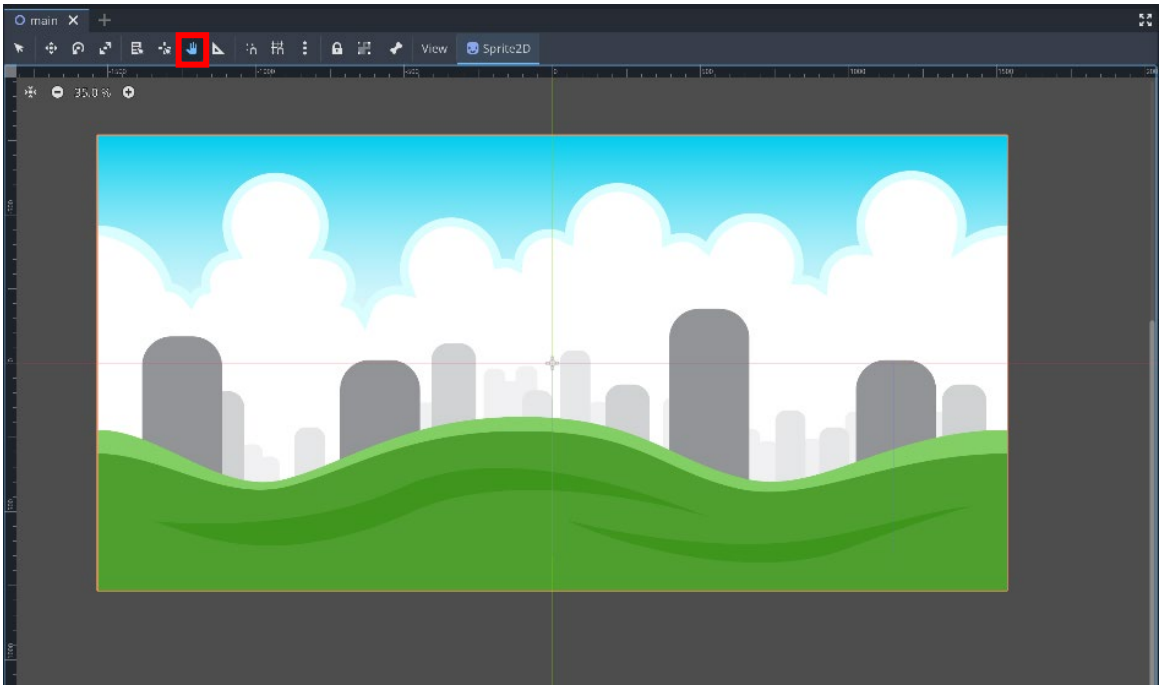
Select the **background** image.



Pro Tip:

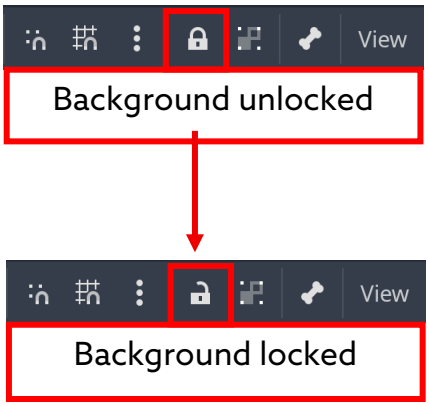
If the wrong texture is loaded, click on the reset arrow beside **Texture** to remove it.

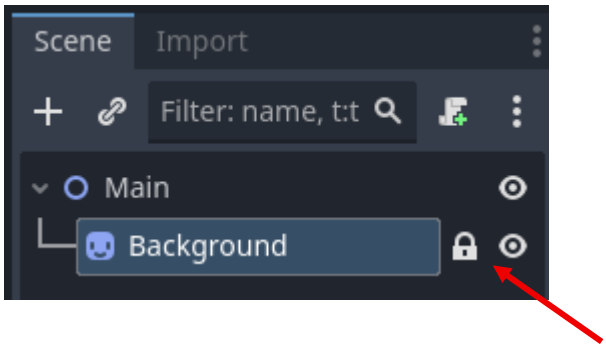
17 Adjust the background image in the game using **Pan Mode** so the whole image is visible. The mouse wheel can be used to zoom in or zoom out as needed.



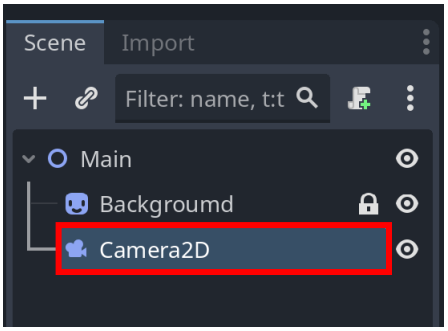
18 The background is currently unlocked.

Click the **lock** icon to prevent accidental movement of the background image. An icon will appear next to the node in Scene to show that it is locked.



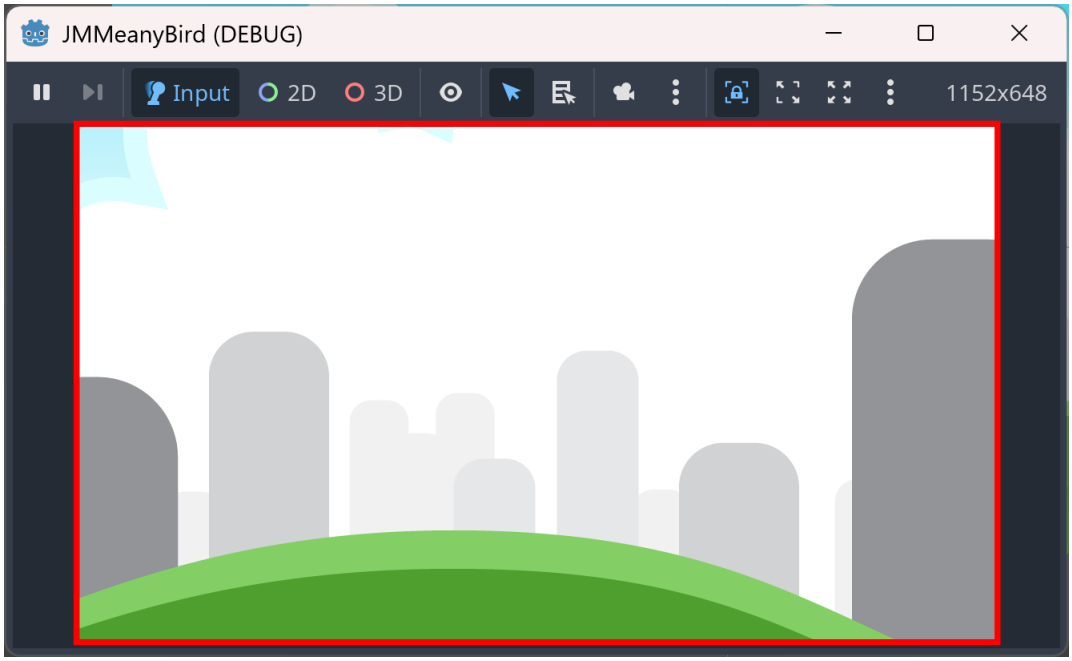


19 In **Scene**, add a **Camera2D** as a child node to **Main**.



20 Playtest the project. What do you notice?

In the playtest window, notice that only a small part of the background image can be seen. The **viewport** dimensions should be narrower to focus on the player's movement, like a phone screen.



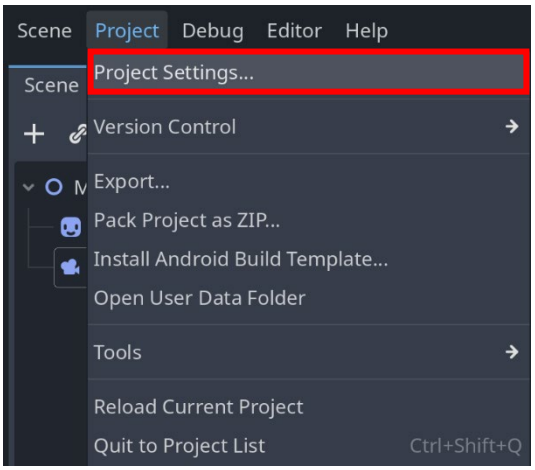
The **viewport** is the section of the project that can be seen in the playtest window.

Close the playtest window.

21

Update the **viewport dimensions**.

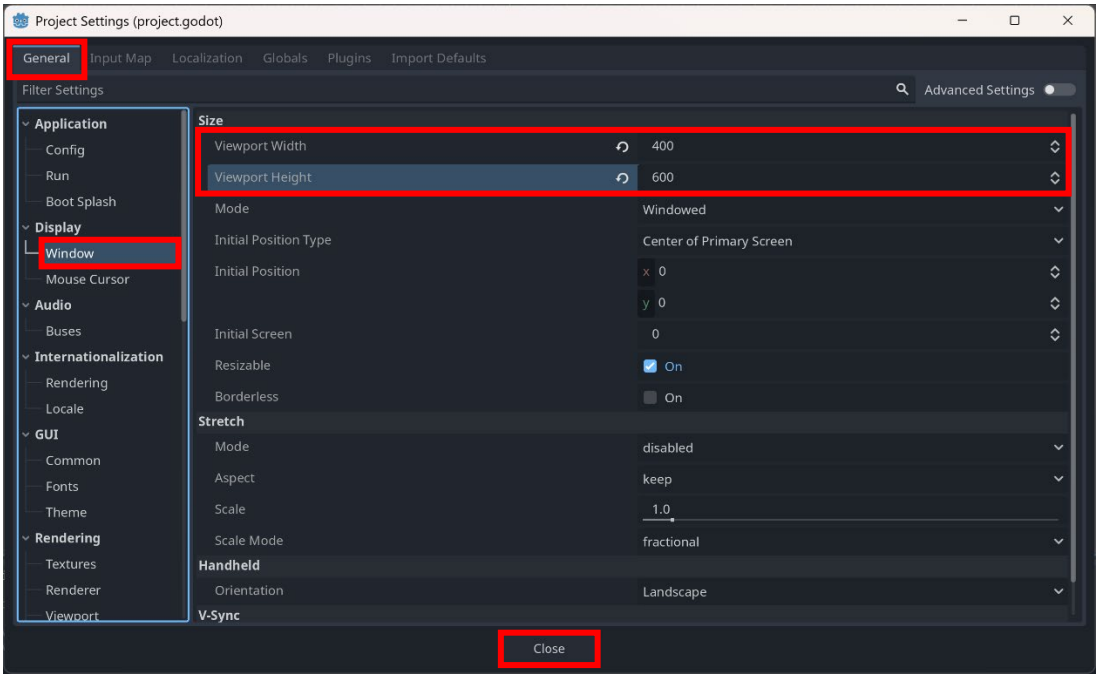
In the top left corner of the editor, click **Project** and open **Project Settings**.



22

Under **General**, find **Display** then **Window** and set the **Viewport Width** to **400**, and the **Viewport Height** to **600**.

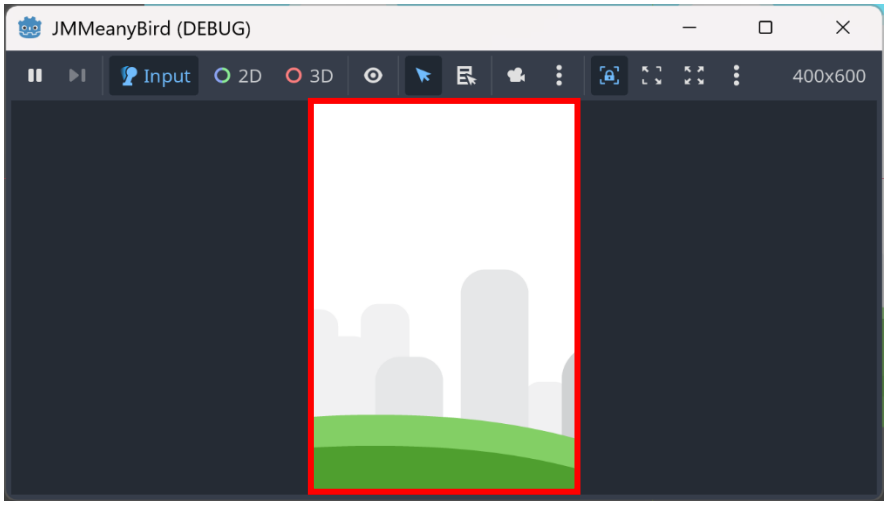
Close the project settings.



23

Playtest the project again. Notice how the dimensions of the viewport have changed.

What else needs to be fixed? Are the sky and grass visible in the viewport?



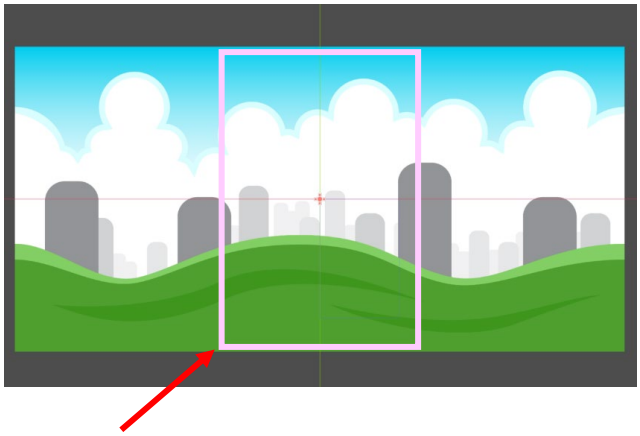
Close the playtest window.

24

Most of the background image can't be seen in the viewport when playtesting. The camera's zoom can be adjusted to fix this.

In the **Inspector** for **Camera2D**, adjust the **Zoom** to include the ground, clouds, and sky within the camera view. The entire background image width should not be included.

Remember, the pink lines in the 2D workspace will show the camera's view.

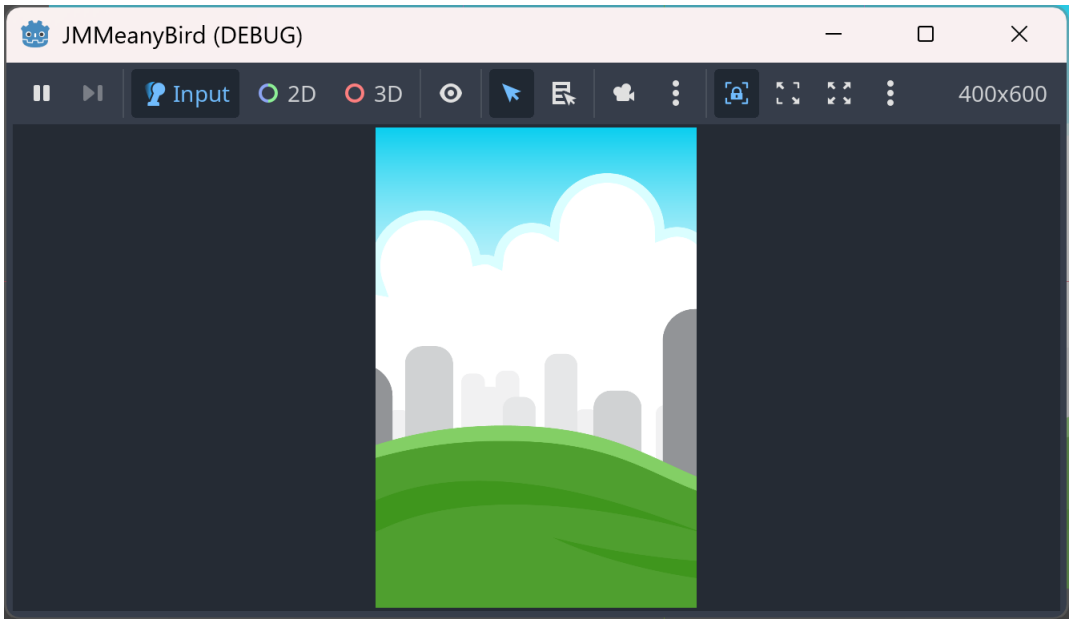


Pro Tip:

Changing the x value of zoom will also change the y value. The **chain icon** symbolizes that the **component ratio** is **locked,** and the values will scale respectively.

25 Playtest the project. Does the camera view look as intended?

Close the Playtest window.



Note: *This playtest window shows the camera view zoomed to 0.4.*

Pause for **Sensei Stop #2**!

Before continuing, check with a Code Sensei to ensure the background and camera are set up correctly.

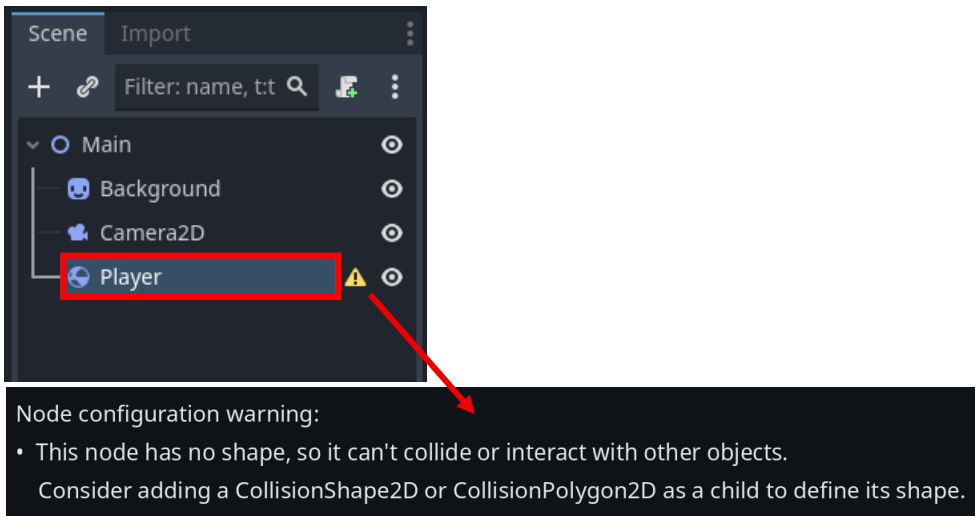
Reminder: Save your work!

26

Let's start setting up the player.

Add a **RigidBody2D** as a child node to Main and rename it **Player**.

The Player node will show a warning symbol. The player does not have a shape currently, preventing it from colliding or interacting with other objects.

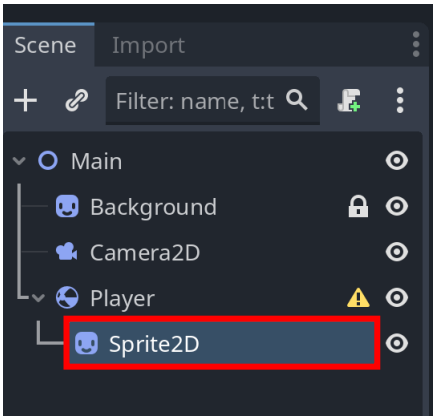


New Concept: RigidBody2D

Remember the RigidBody3D node from Dropping Bombs? A RigidBody2D is the same but for 2D environments. This is used as the base for the player in this project so forces like gravity can be automatically applied to the player.

27

Before giving the Player a shape, let's give it a texture. Add a **Sprite2D** as a child node to Player.



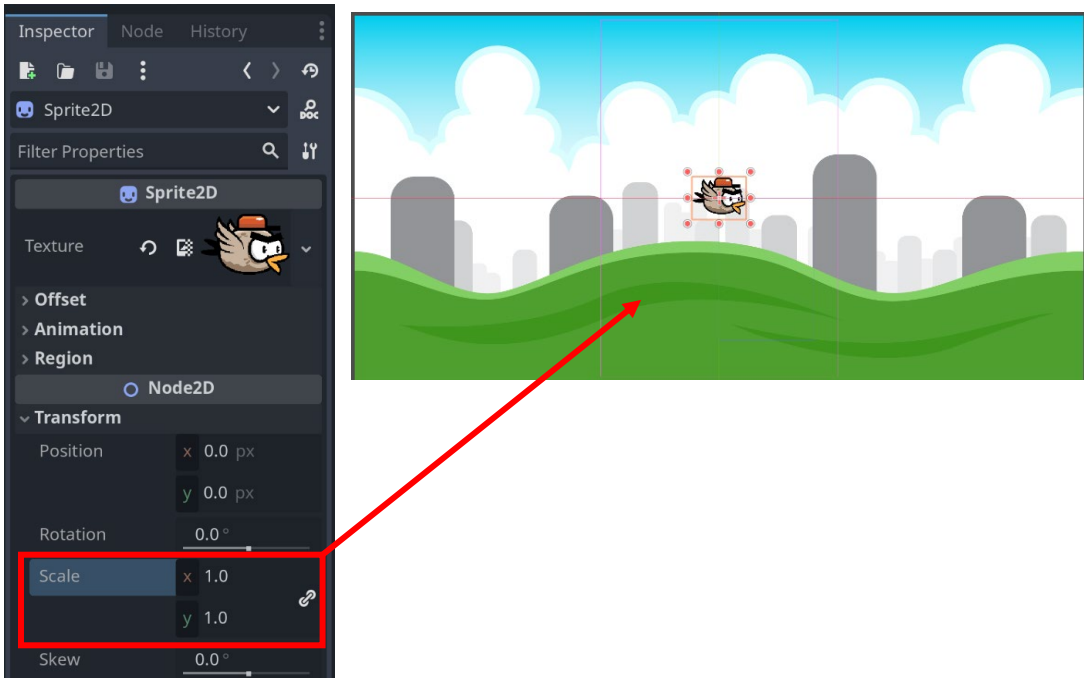
In the **Inspector** for **Sprite2D**, update the texture with the bird image. Refer to **Step 16** as needed.

28

Notice the size of the bird compared to the background image. The bird is too large!

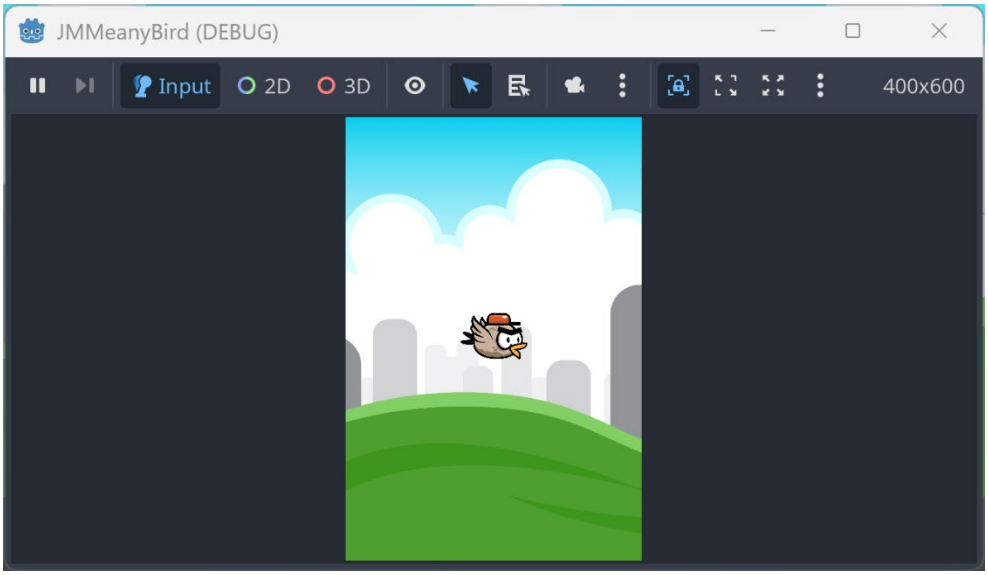
In the **Inspector** for **Sprite2D**, select **Transform** and adjust the **Scale.**

Scale the bird down enough to leave space to add obstacles later.



29

Playtest the project. How does the bird's size appear, relative to the background image?



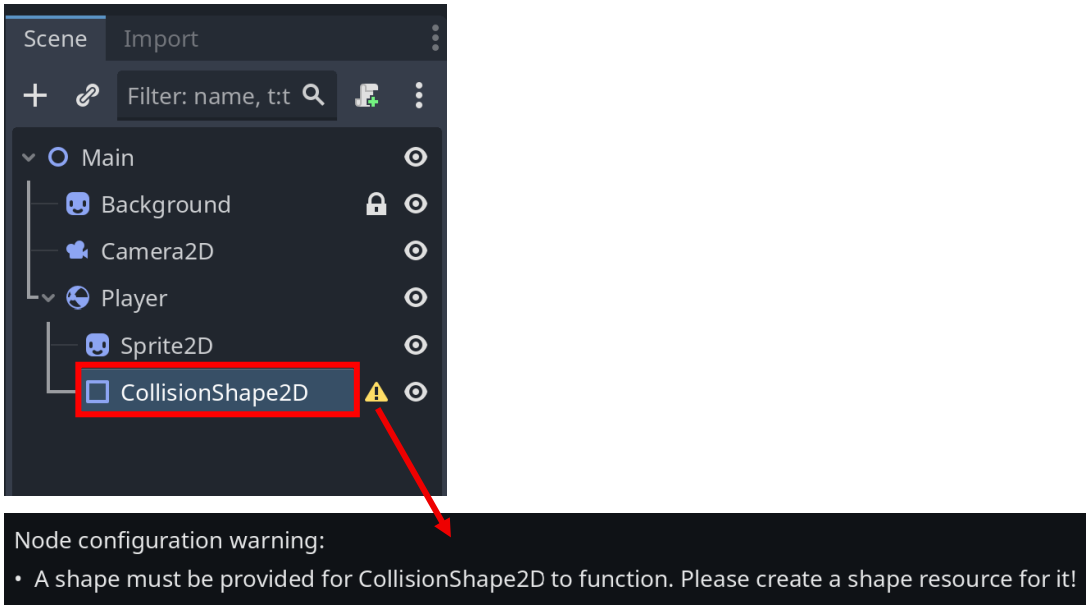
Close the Playtest window.

Note: *This playtest window shows a bird scaled at 0.3.*

30

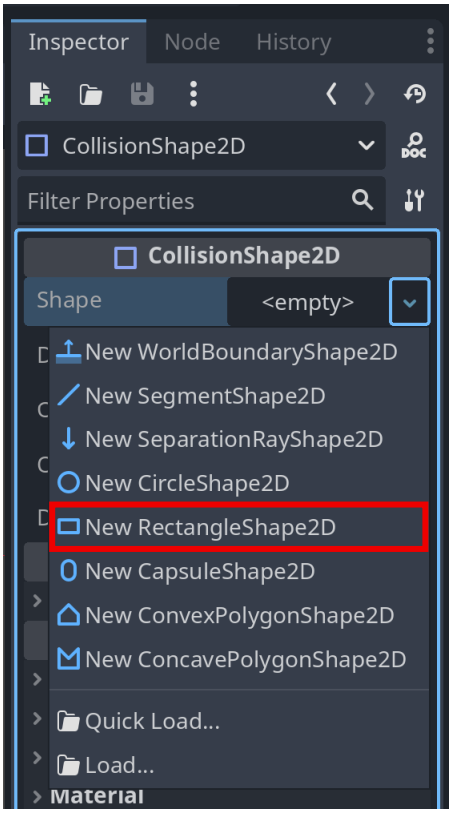
The Player still needs a shape.

In Scene, add a **CollisionShape2D** as a child node to **Player**. The CollisionShape2D node will show the warning symbol until a shape is assigned in the Inspector.



31 In the **Inspector** for **CollisionShape2D,** locate **Shape <empty>** and select the drop-down arrow to choose a new shape for Player from the drop-down menu.

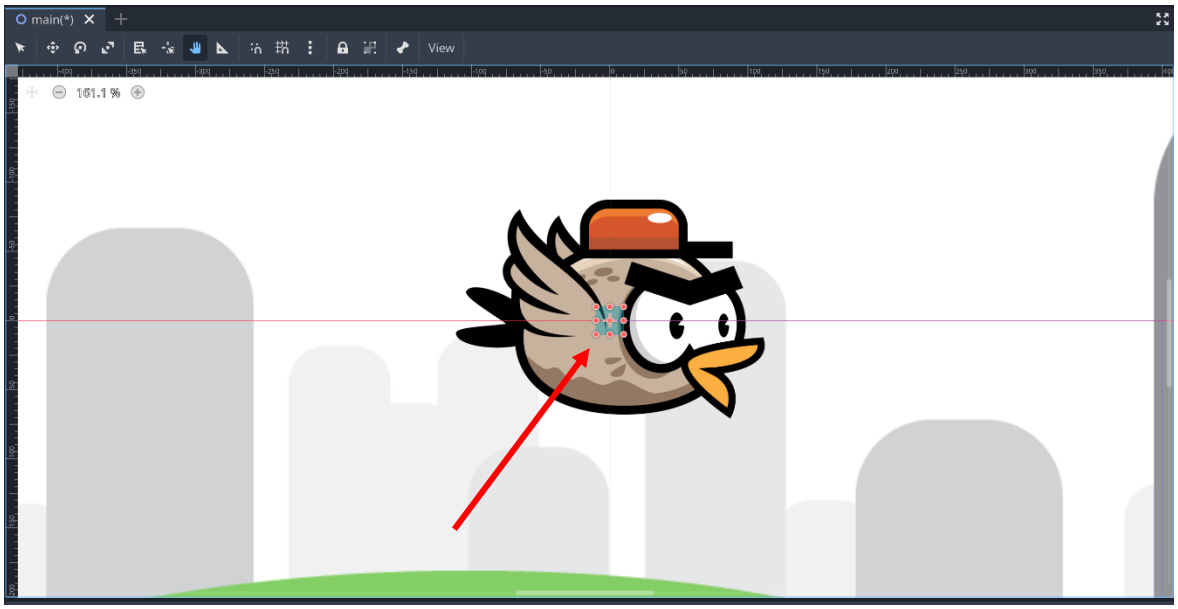
A **rectangle** shape will work best for this project.



32 The shape will need to cover the bird texture so that collisions can be monitored.

Select the **CollisionShape2D** in Scene and use the mouse wheel to zoom in on the bird in the 2D workspace.

A small blue box with red dots will become visible. This is the CollisionShape2D that was added to Player.



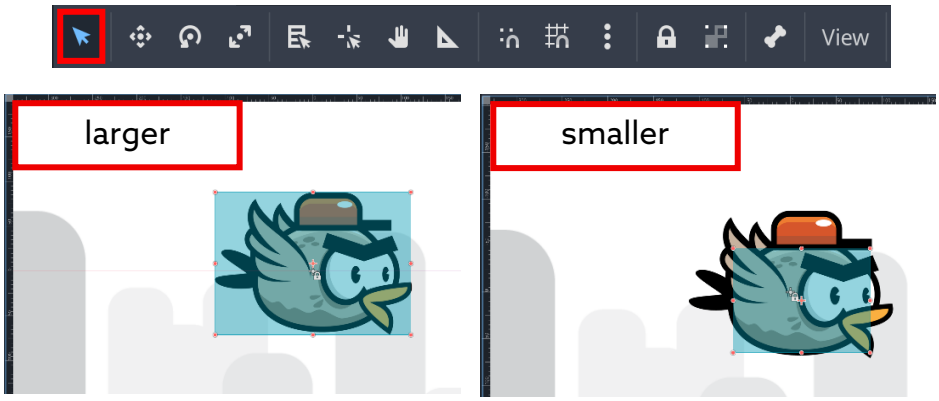
Pro Tip:

Use **Pan Mode** to center the bird texture and CollisionShape2D.

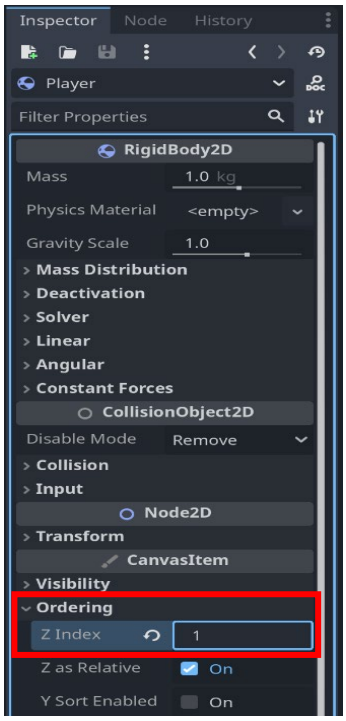
33 The shape needs to be resized. The shape will not fit the bird texture exactly. Choose whether to resize the shape so it's slightly larger than the texture, or slightly smaller.

A larger collision shape could make the game harder to play, while a smaller collision shape would make the game easier and more rewarding to play.

Click back to the cursor tool and drag the red dots to resize the shape to fit the bird texture.



34 In the **Inspector** for **Player**, select the **Ordering** drop-down menu and change the **Z Index** from 0 to **1**.



All nodes are automatically assigned a **Z Index** of 0. Changing the **Z Index** of Player from 0 to 1 ensures the Player can always be seen **in front** of the background.

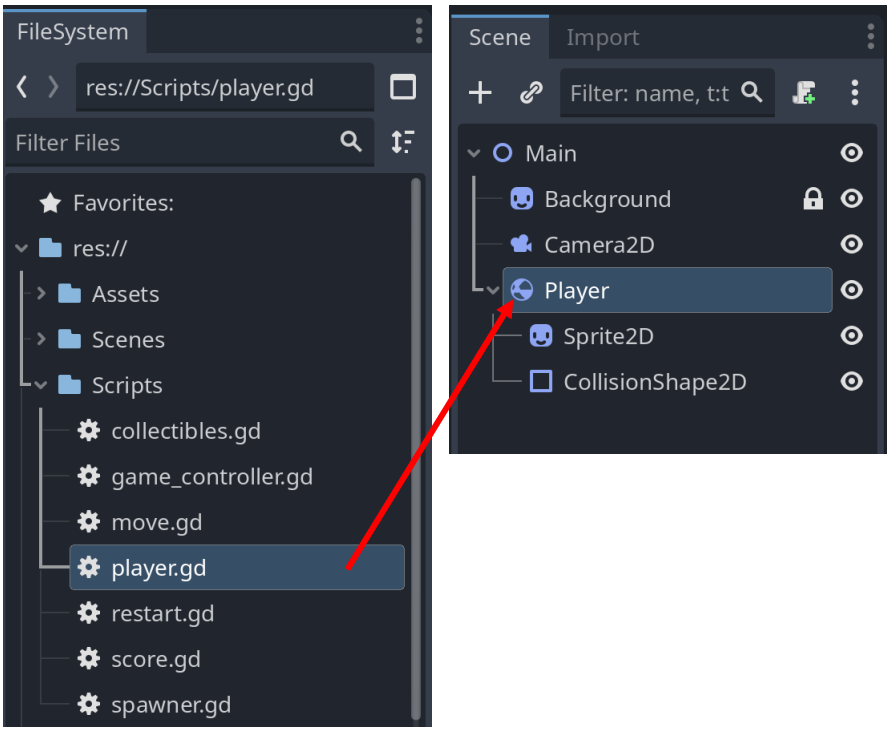
Pause for **Sensei Stop #3**!

Before continuing, check with a Code Sensei and make sure the Player setup is correct.

Reminder: Save your work!

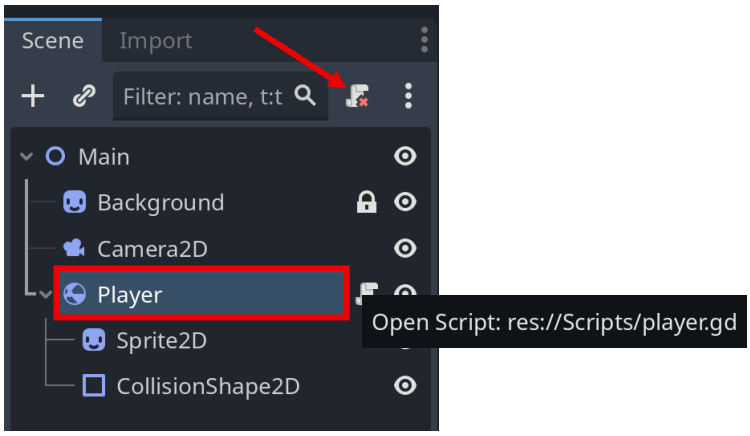
35 Now that the Player has been set up, let's add in some controls.

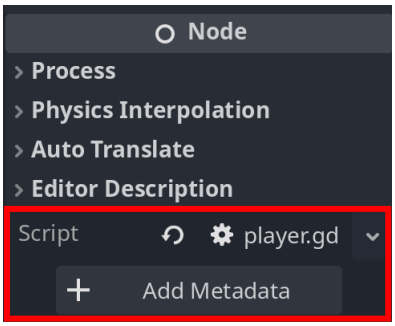
In the **FileSystem**, find **player.gd** script in the **Scripts** folder. Drag the script to the Player node to attach the script.



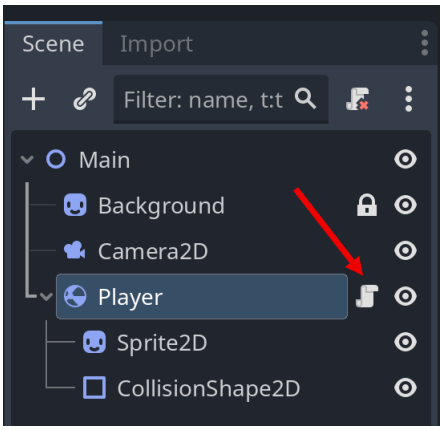
36 Check which script is attached by hovering the cursor over the **script** icon beside the Player node in Scene or, in the **Inspector** for **Player**, scroll down to the bottom and find **Script**.

If the incorrect script is attached, select **Player** in Scene and click the **detach script** icon or click the  icon beside **Script** in the **Inspector**.





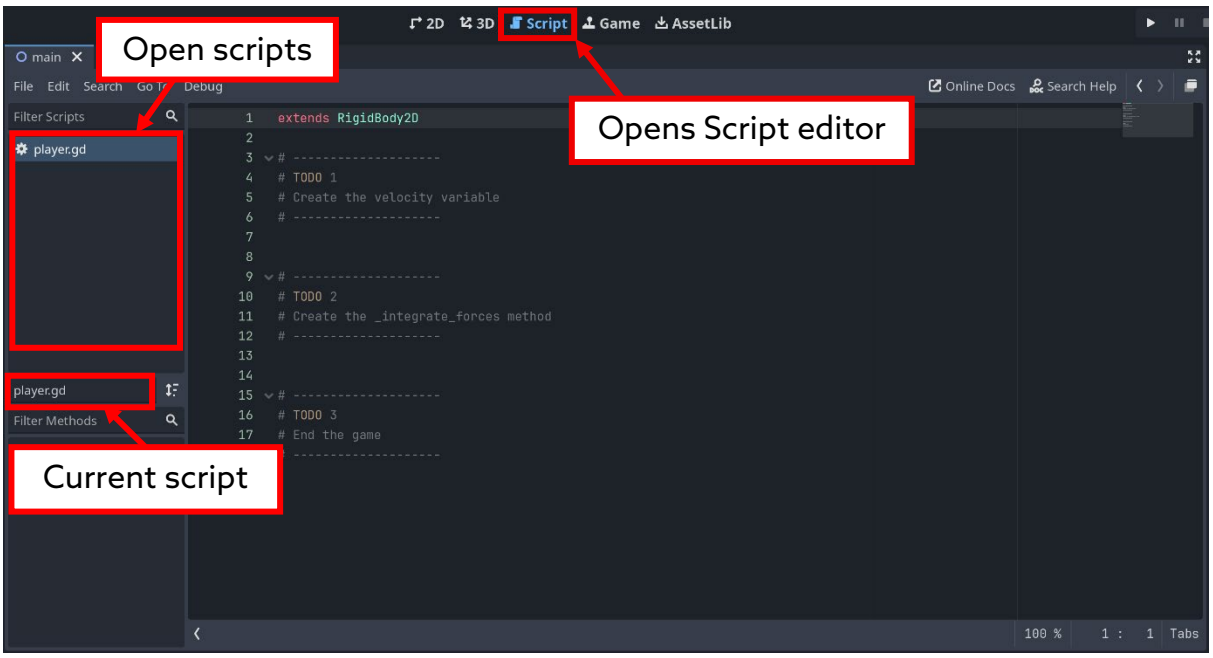
37 Click the **script** icon beside **Player** to open the **player.gd** script.



38 The top of the editor shows the script editor is open.

The player script is open and contains some comments to show where different parts of the code will go in the script.

To the left of the code editor, there is a section which shows the scripts that have been opened and edited, as well as the script currently open in the editor.



The player script will be coded to push the bird up when the left mouse button is clicked.

39 The first line of code in the script reads `extends RigidBody2D`.

This allows the script to **inherit** methods from the **RigidBody2D class** so they can be referenced or called when writing the code.

```
1  extends RigidBody2D
2
3  # ---------------------
```

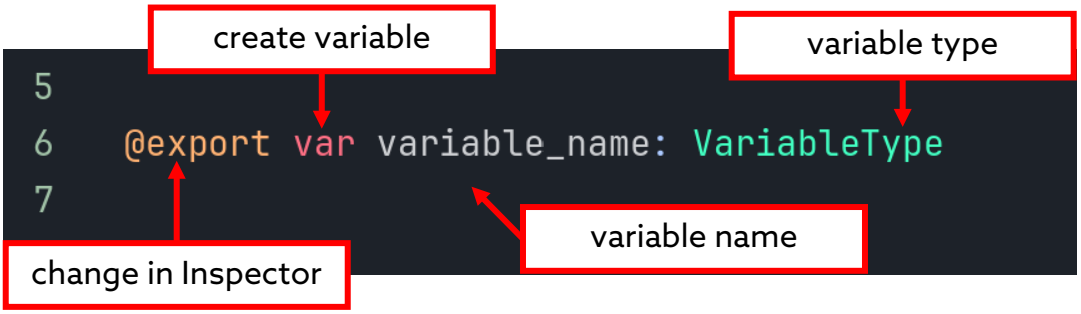
40

The player script needs a variable.

When creating a variable that can be accessed and changed in the **Inspector**, `@export` is used.

To create a variable in GDScript, the key word **var** is required, followed by the **variable name**.

Godot requires the **type** of variable to be declared (String, Integer, Node2D, etc).

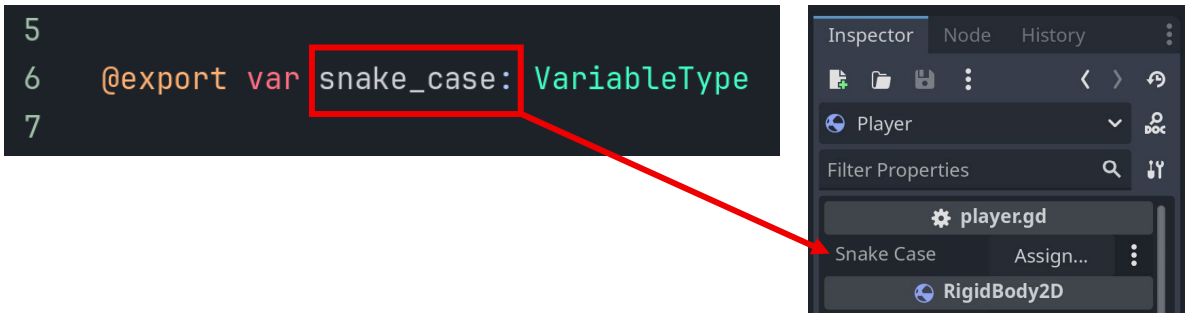


In Godot, variables should be named in lower case, using an underscore to separate words. This is called **snake_case**.

One-word names remain lower case. Example: snake

In the Inspector, Godot automatically capitalizes variable names in snake_case.

Nodes in **Scene** do not follow snake_case, but instead use **PascalCase** where the first letter of each word is capitalized.



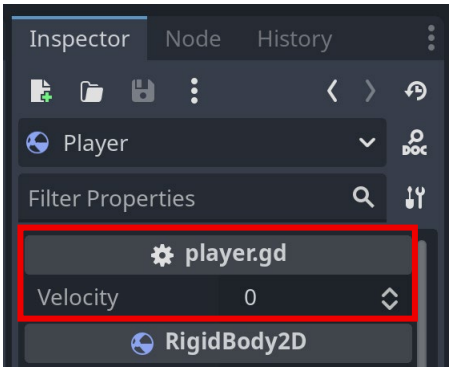
41

Try creating a variable!

This script will need an `@export` variable named `velocity` of type `int` **(integer)**. Add code to create the `velocity` variable under **TODO 1**.

```
1  extends RigidBody2D
2
3  # --------------------
4  # TODO 1
5  # Create the velocity variable
6  # --------------------
7
8
```

Save the script and check in the Inspector for Player. Does the variable show up in the inspector?



42

Check the code! Update the script as needed.

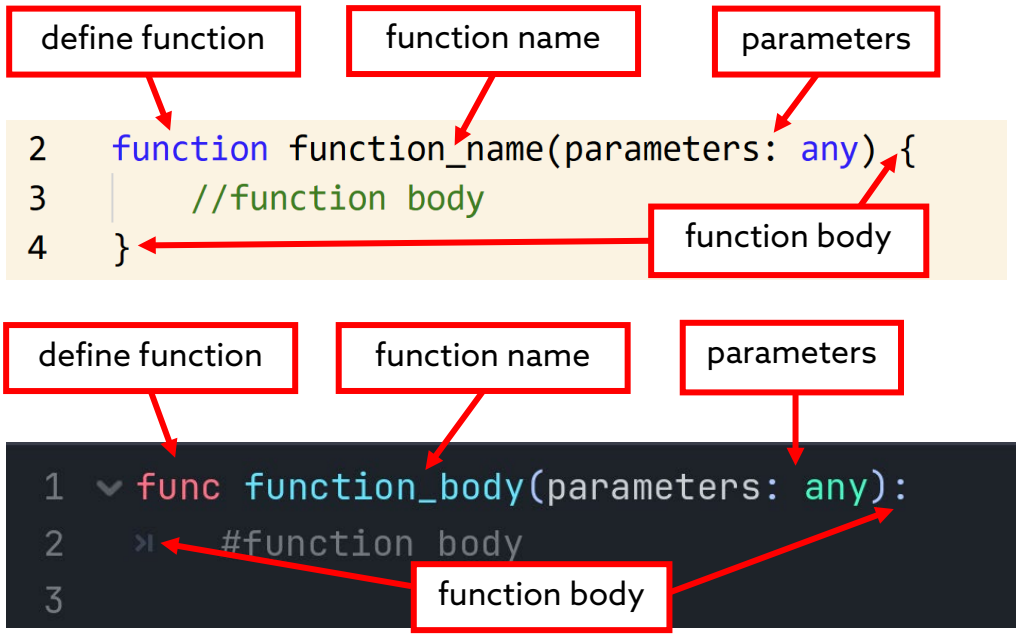
```
1   extends RigidBody2D
2
3   # --------------------
4   # TODO 1
5   # Create the velocity variable
6   # --------------------
7   @export var velocity: int
8
```

43

Before continuing with the code, let's look at how GDScript and JavaScript syntax differ when creating functions.

GDScript is very similar to Python and relies on **indents** and **colons** instead of brackets to organize code into functions, loops, and conditionals.

In both GDScript and JavaScript, a keyword to **define** the **function** is used, followed by a **function name** and a set of **soft brackets ()** for any parameters.




```
2   function function_name(parameters: any) {
3       //function body
4   }
```

```
1   func function_body(parameters: any):
2       #function body
3
```

44 Under **TODO 2**, use the keyword **func** to define a new function, followed by the method name, `_integrate_forces`.

Double click the code completion to complete the method.

The `_integrate_forces()` method has a **PhysicsDirectBodyState2D** parameter.

The `-> void` shows what the method returns. This method is **void**, meaning it does not return anything.

Make sure the `:` **(colon)** is added after the method definition (the method's name, parameters and return type).

```
9
10  # --------------------
11  # TODO 2
12  # Create the _integrate_forces method
13  # --------------------
14  func _integr
15      fo _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
16
```

45 The `_integrate_forces()` method is called at the same time the engine calculates physics, which happens a set number of times per second.

Since rigid bodies are controlled by **Godot's built-in physics engine**, their properties can't just be updated since these changes may conflict with built-in physics. The `_integrate_forces()` method is used to "break" the physics engine and update properties (like applying velocity) to a rigid body.

> `_integrate_forces()`: Part of the RigidBody2D/3D classes, this method is called during the physics processing step of the main loop. This method modifies the simulation state of the object.
>
> **Parameters:**
>
> 1. `state` ***(PhysicsDirectBodyState2D/3D)*****:** provides direct access to a physics body, allowing changes to physics properties.
>
> **Returns *(void)*:** this method is void, it returns nothing.

46 After adding in the method, the **script editor** will send an error. This is because the method is missing the indented code for the body.

The script editor will always send an error when indentation or code for the body of a function or method is forgotten.

```
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15
16
```

Error at (14, 65): Expected indented block after function declaration. 1

Methods vs Functions

Methods are built-in functions that are part of a Node class.
Functions are user-defined functions that don't have ties to a class.

47 Inside the `_integrate_forces()` method, start an `if`**-statement**. Don't forget to **indent** the code inside the method!

```
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15      if
16
17
```

48 `If`**-statements** in GDScript are formatted similarly to functions in GDScript with a **colon** and **indentation** for the statement body, unlike JavaScript which uses curly brackets**.**

In GDScript, **parentheses** `()` *can* be used but are not needed to hold the condition that needs to be true for the if-statement.

```
7   if (condition) {
8       // then
9   }
```

JavaScript

```
11  if condition:
12      #then
```

GDScript

49

With the help of the `if`-statement, the `_integrate_forces()` method will be coded to:

- Check if the click action is pressed
- If it's true, give the player a velocity boost

Using the keyword `Input`, code the `if`-statement to check if the **click** action is pressed.

Input:
The key word `Input` accesses the Input class which handles key presses, mouse buttons and movement, gamepads, and input actions. Actions and their events can be set in the **Input Map** tab in the **Project Settings**.

> `Input.is_action_just_pressed()`: returns true when the user has started pressing the action event in the current frame or physics tick. Will only return true in the frame or tick that the user pressed down the button. This is useful for code that needs to run only once when the action is pressed, instead of every frame while it's pressed.
>
> **Parameters:**
>
> 1. `action` ***(StringName):*** the name of the action event.
>
> **Returns** ***(boolean):*** whether the action was initiated in the current frame

```
10  # --------------------
11  # TODO 2
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15      if Input.is_action_just_pressed("click"):
16
17
```

50 Using the **assignment operator** `=`, give the player a velocity boost by updating the `linear_velocity` parameter inside the `if`-statement.

```
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15      if Input.is_action_just_pressed("click"):
16          linear_velocity =
17
```

New Concept: Linear Velocity

Linear Velocity measures how fast a RigidBody changes its position in a straight line.

51 Assign `linear_velocity` to `Vector2.UP`.

`Vector2()` creates a 2D vector with an x and y coordinate of 0 unless specified otherwise.

Using the constant `.UP` creates a vector where **x = 0** and **y = -1**. Since the **positive Y** axis is **down** in 2D, a **negative Y** value will point **up**.

```
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15      if Input.is_action_just_pressed("click"):
16          linear_velocity = Vector2.UP
17
```

52 A vector like `Vector2.UP` is usually used to represent a direction and multiplied by some other value to specify how far in that direction to move.

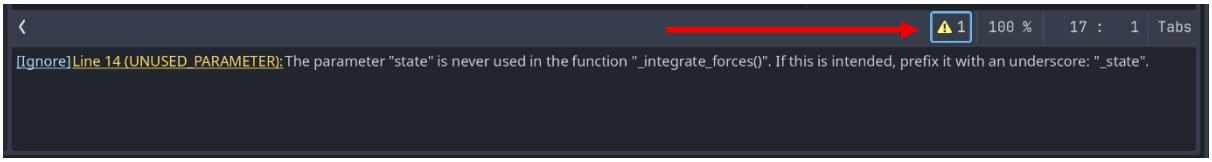
Using the **multiplication operator** `*`, multiply `Vector2.UP` by the `velocity` variable. This will move the player up by the value of velocity.

```
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(state: PhysicsDirectBodyState2D) -> void:
15      if Input.is_action_just_pressed("click"):
16          linear_velocity = Vector2.UP * velocity
17
```

Save the script.

53 Look in the script editor. What do you notice?

There is a **warning** symbol! Click on the warning symbol to view the warning.



What might it mean? How could it be fixed?

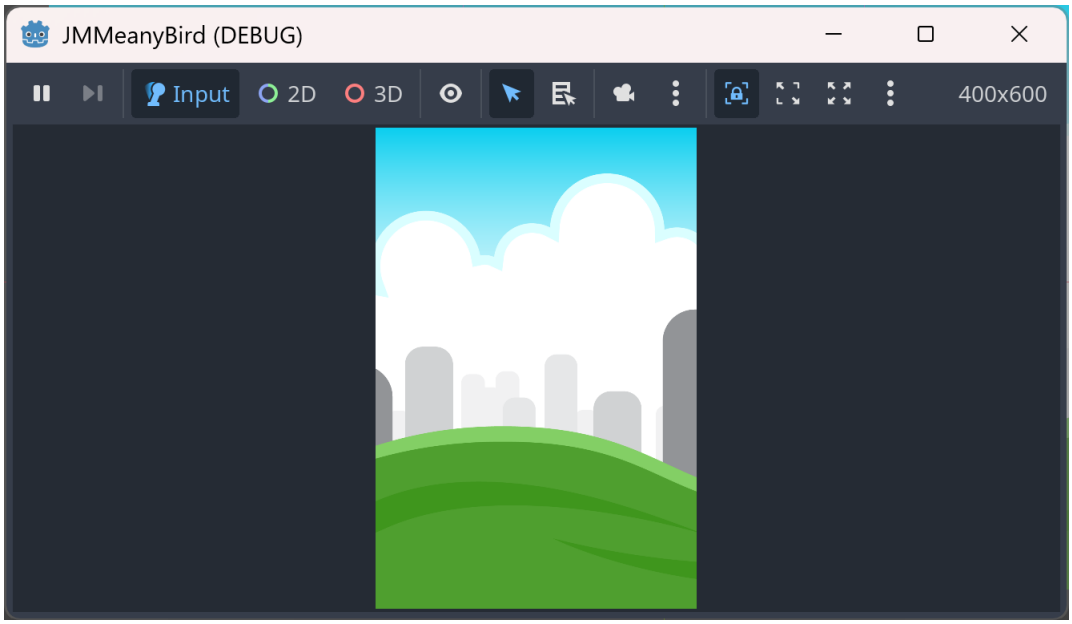
54 The parameter `state` is not used in the `_integrate_forces()` method.

This does not cause any issues in the code or prevent the game from working. To prevent the warning from showing up, add an **underscore** `_` in front of the parameter `state`. Godot will ignore the parameter and the warning will disappear!

```
10  # --------------------
11  # TODO 2
12  # Create the _integrate_forces method
13  # --------------------
14  func _integrate_forces(_state: PhysicsDirectBodyState2D) -> void:
15      if Input.is_action_just_pressed("click"):
16          linear_velocity = Vector2.UP * velocity
17
```

Save the script.

55 Playtest the project. What happens to the bird when the mouse button is clicked?



The bird falls off the screen!

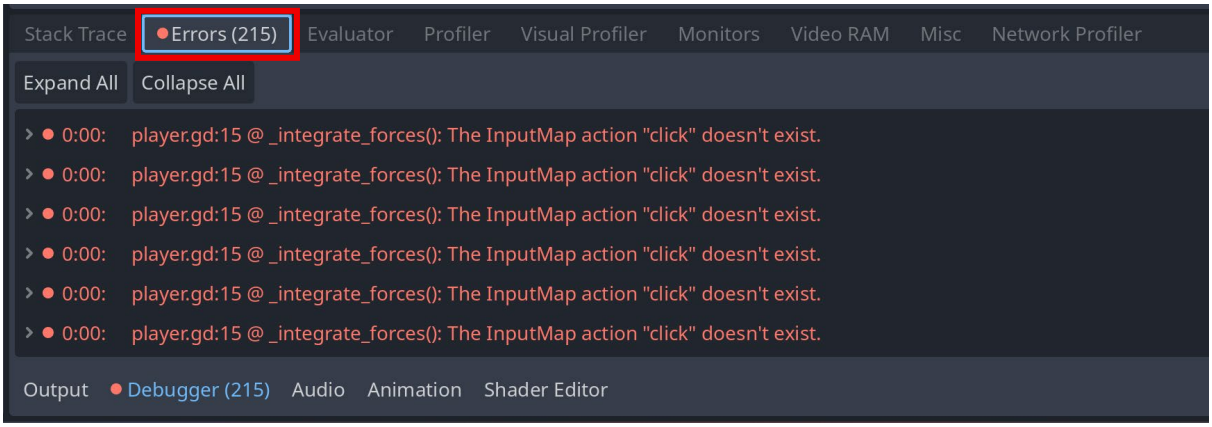
Close the playtest window.

56 After playtesting, some errors can be seen in the **Debugger**.

Click on the **Debugger** in the bottom panel.

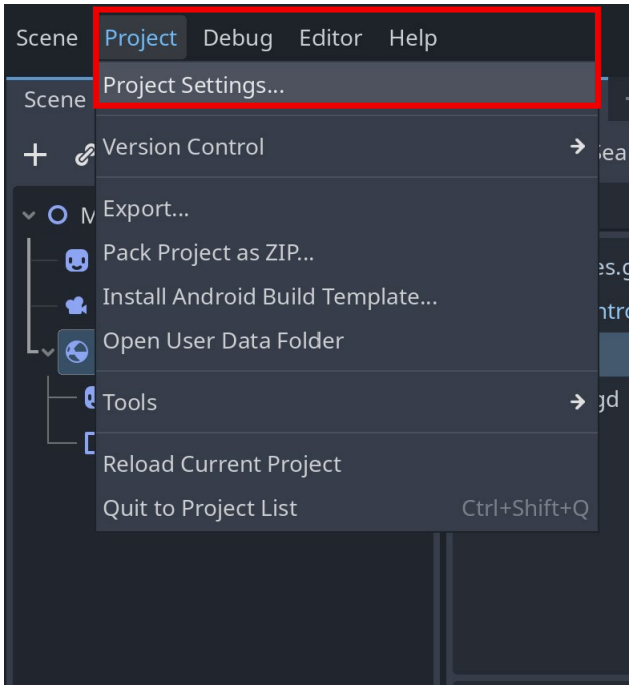
57 In the **Debugger**, toggle to **Errors** and read the errors.

What might the errors mean? How might the errors be resolved?

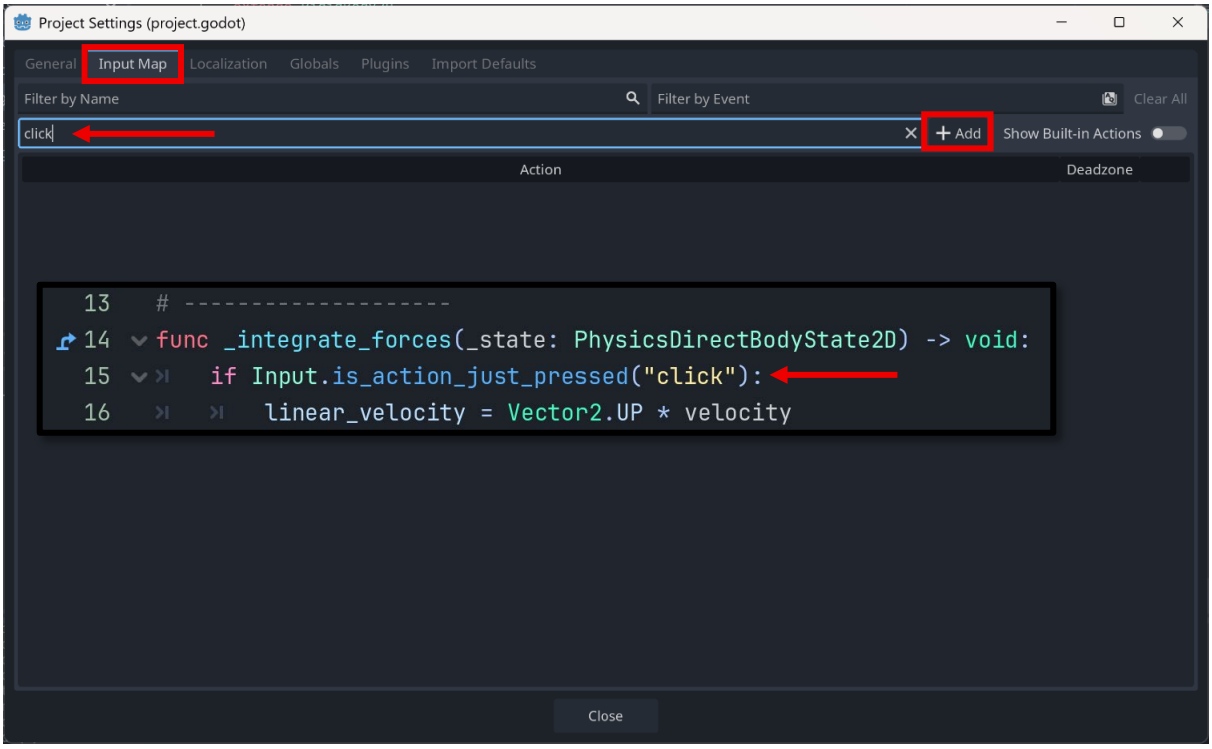


58 The `click` action used in the **player.gd** script does not exist. The action needs to be added to the **Input Map** in the project settings.

In the top left corner, find **Project** and open **Project Settings**.



59 In **Project Settings**, toggle to **Input Map**. Type the name of the action, **click**, and click **+ Add**.



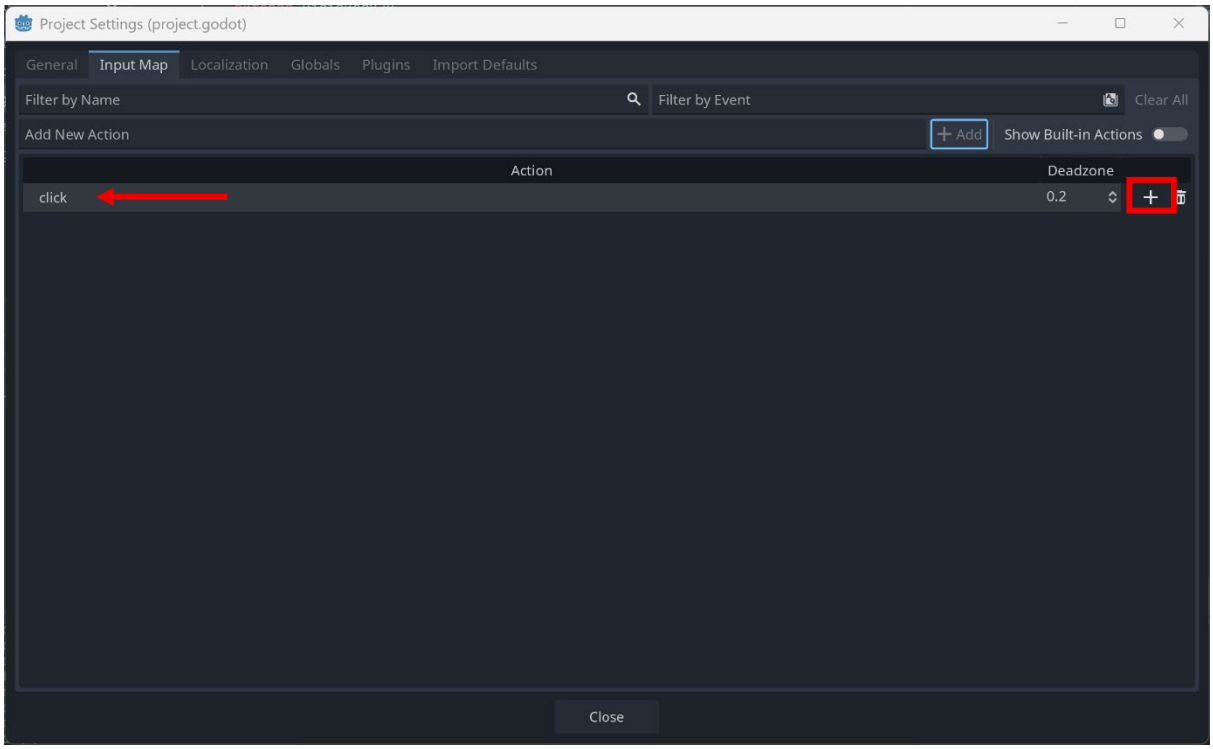
Pro Tip:

The name of the action must match the spelling and the capitalization used in the code!

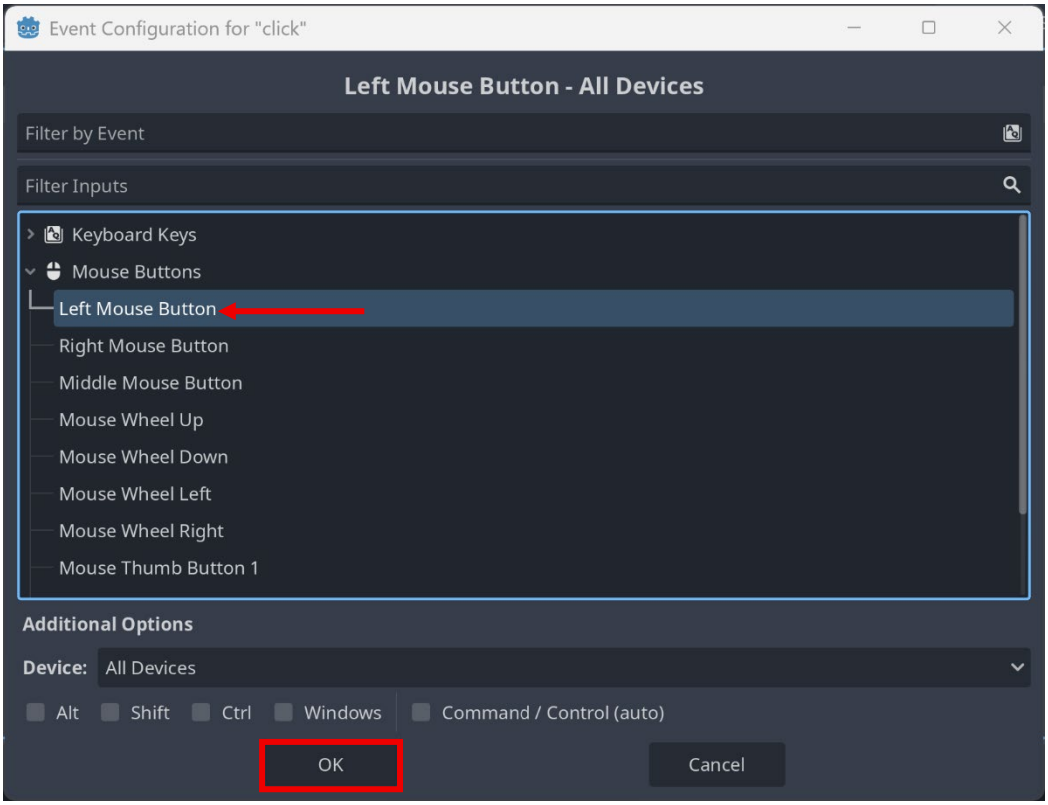
60

The **click** action has been created but is not linked to any input yet.

On the right side of the click action, select **+**.



61 Under **Mouse Buttons**, select **Left Mouse Button** and click **OK**.

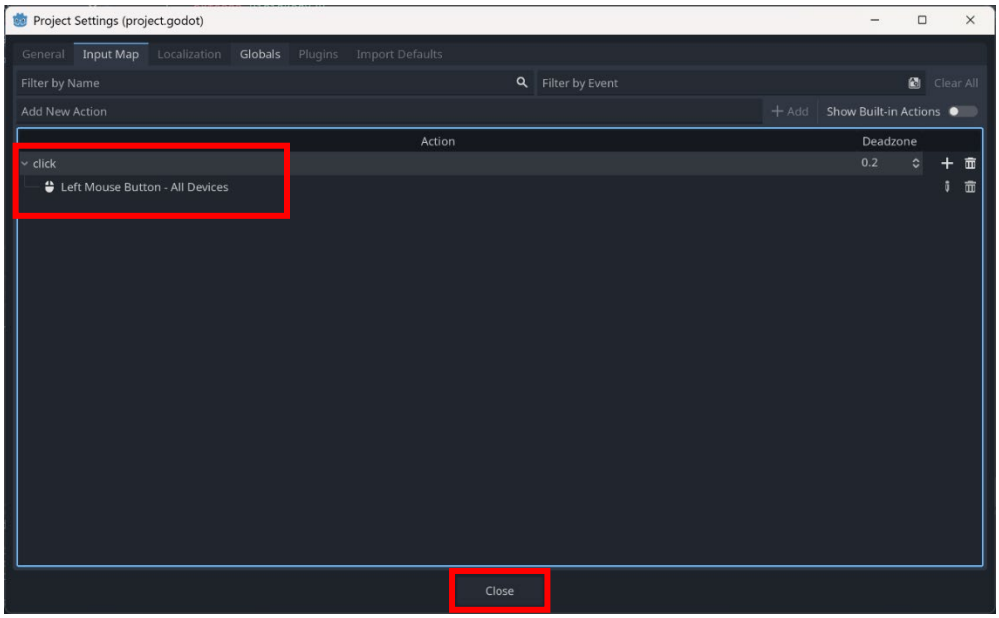


Pro Tip:

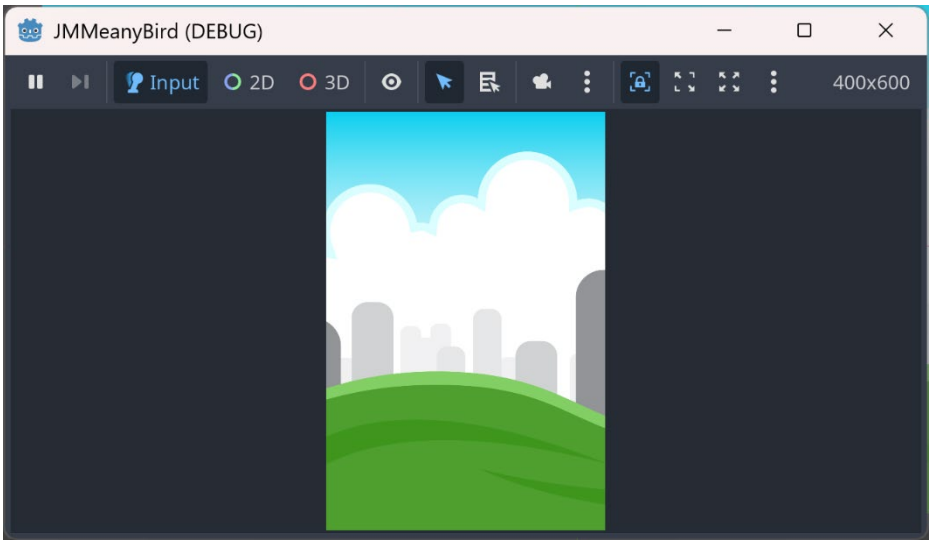
Expand the Input types by clicking on the > arrow!

62 The Input Map now shows the click action and its input!

Click **Close** to exit the project settings and save the scene.



63 Playtest the project. What happens when the left mouse button is clicked?

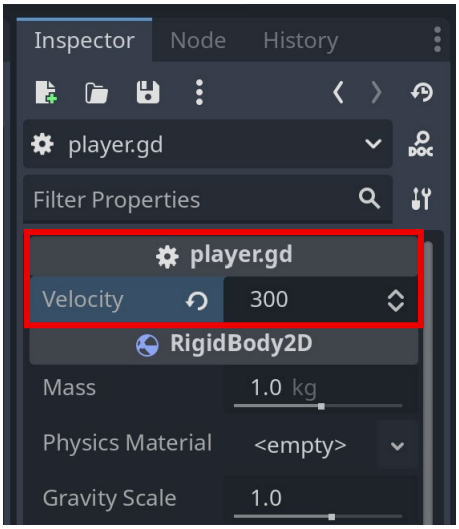


The bird still falls off the screen! Why might that be?

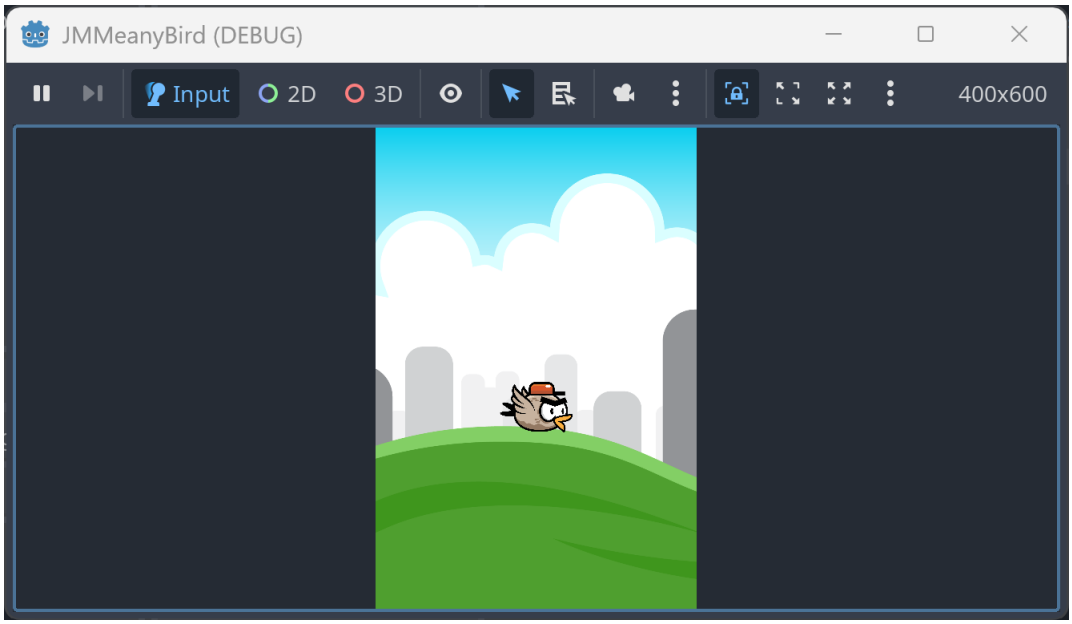
Close the playtest window.

64 In the **Inspector** for **Player**, check the value of **Velocity**. Has the value been updated yet? It has not!

Set **Velocity** to a value around **300** and save the scene.



65 Playtest the project. Click with the mouse in the playtest window to test the velocity.



Does the bird move up enough? Try changing the value of velocity in the Inspector until you can keep the bird mid-screen without clicking too often.

Close the playtest window.

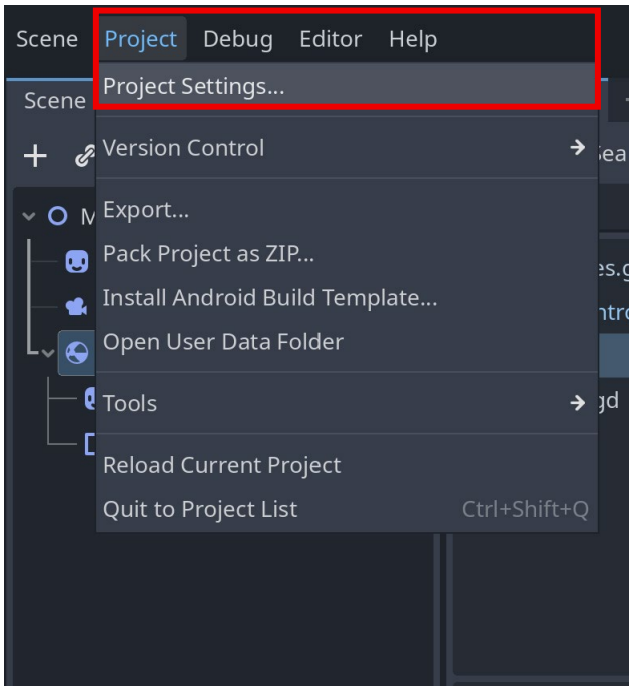
Pause for **Sensei Stop #4**!

Before continuing, check with a Code Sensei and make sure the player script code and click action are correct.

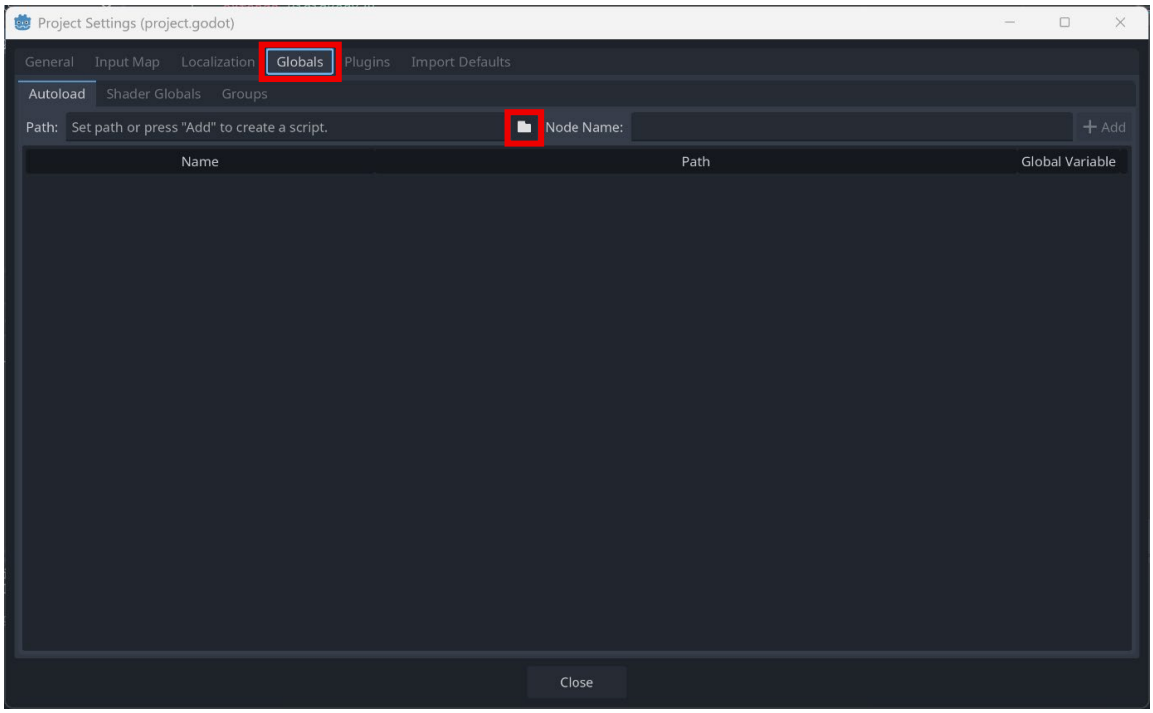
Reminder: Save your work!

66 Before continuing with the project, a global variable needs to be added to the project settings.

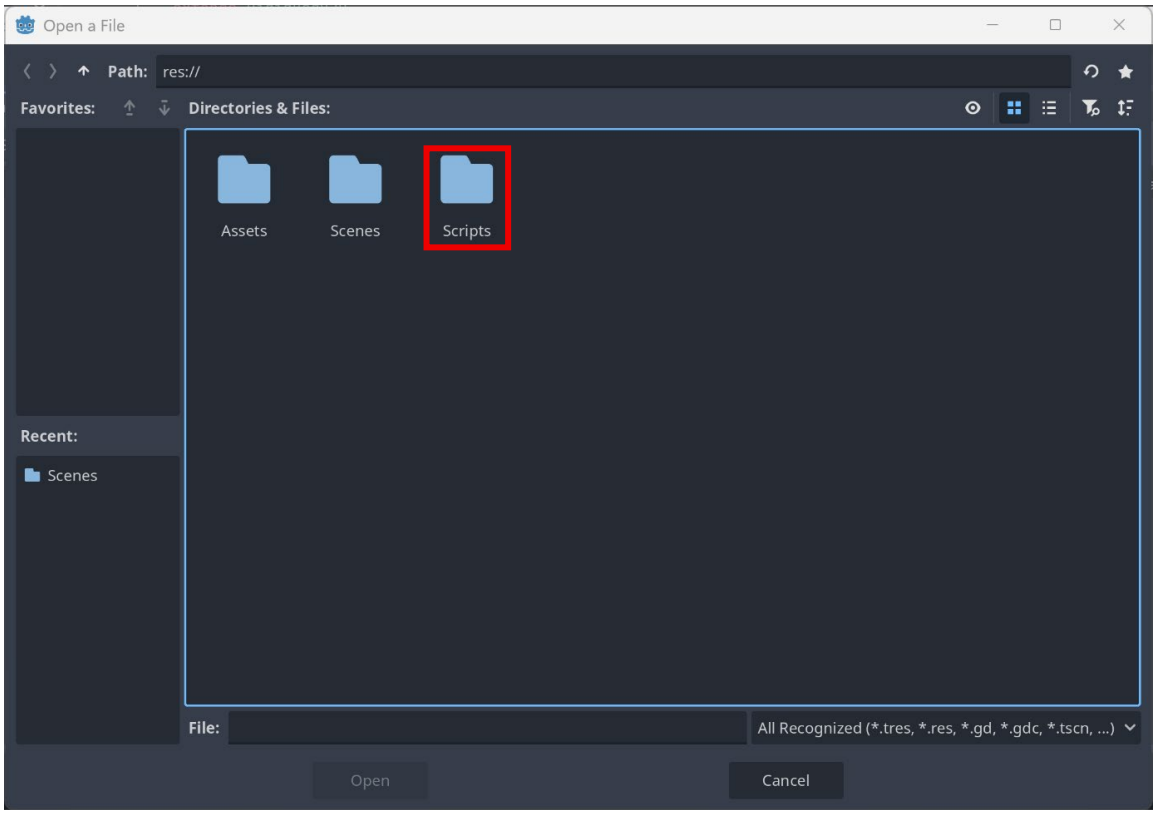
In the top left corner, find **Project** and open **Project Settings**.



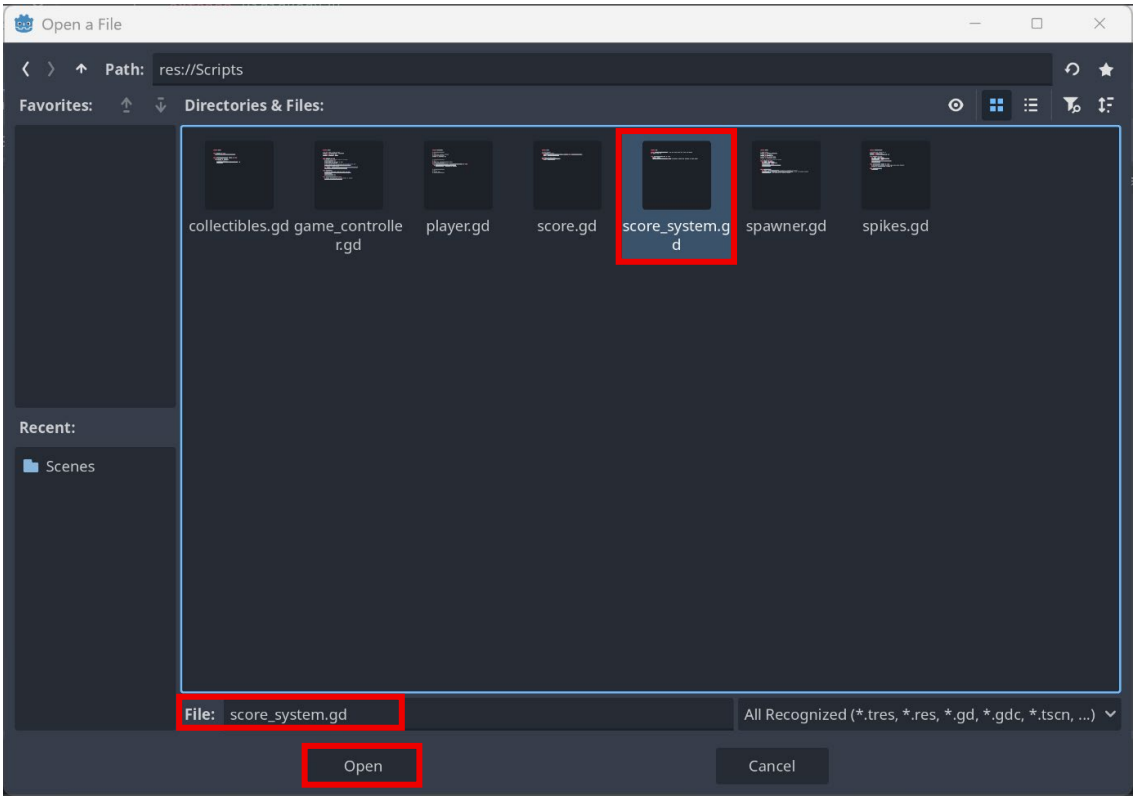
67 Toggle to **Globals** and click the **file icon**.



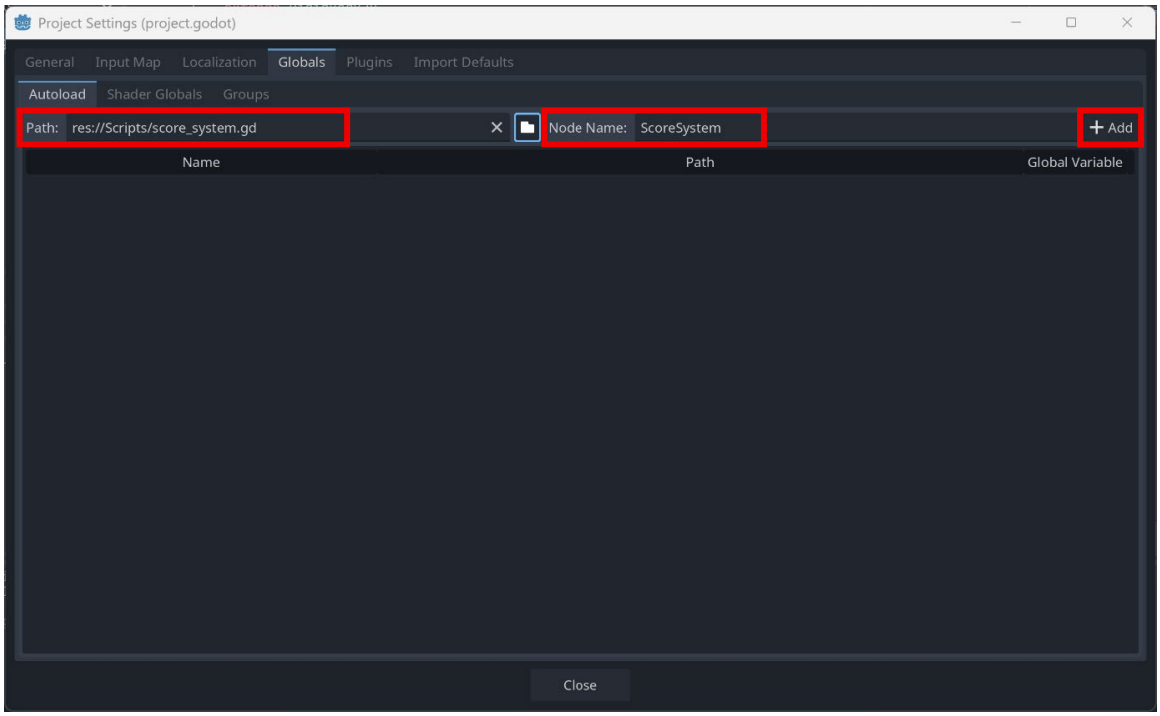
68 In Directories & Files, open the **Scripts** folder.



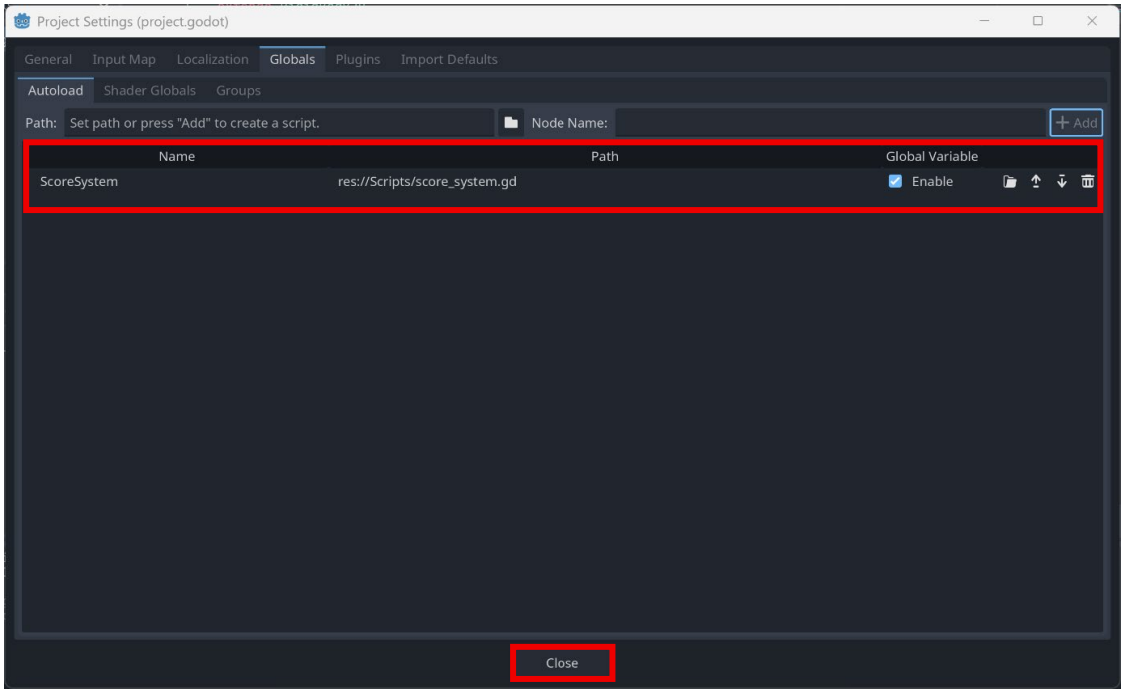
69 Select **score_system.gd** and click **Open**.



70 Check that *Path* is set to **res://Scripts/score_system.gd** and **Node Name** is set to **ScoreSystem**, then click **+Add**.



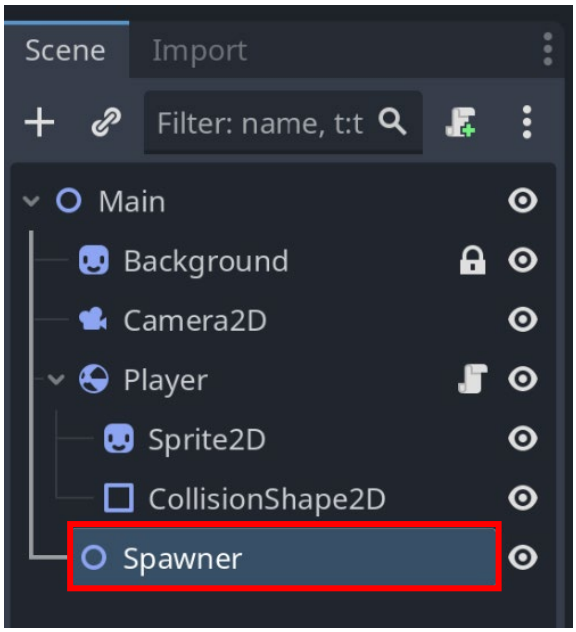
71 A new global variable **ScoreSystem** will appear. Make sure **Enable** is checked and click **Close**.



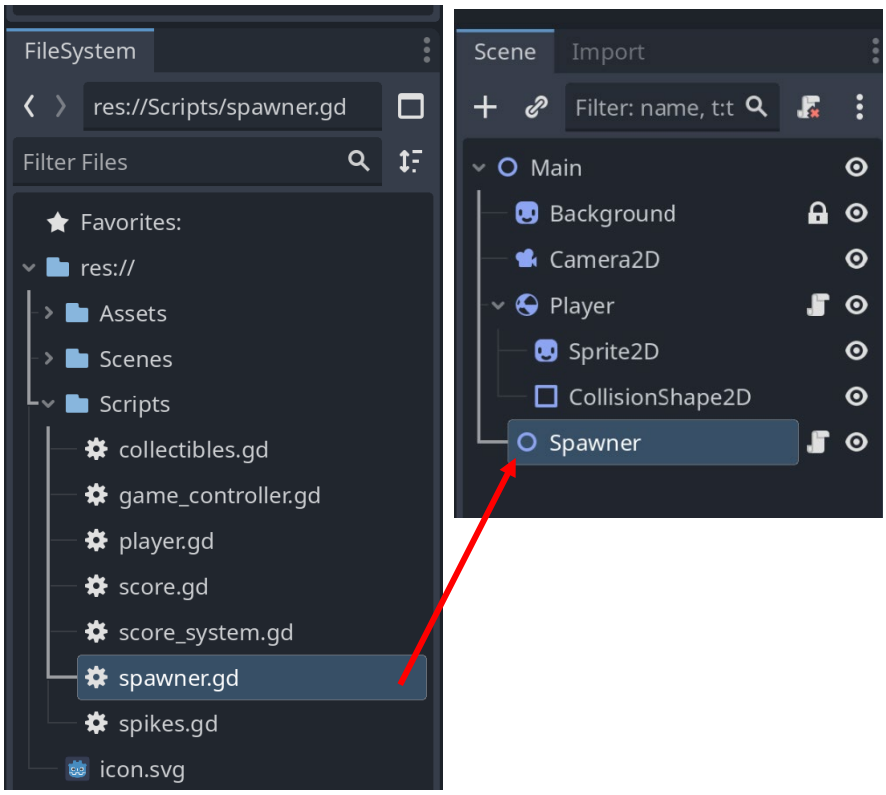
72 It's time to spawn some obstacles for the Player to avoid!

In Scene, add a **Node2D** as a child to Main and rename the new node **Spawner**.

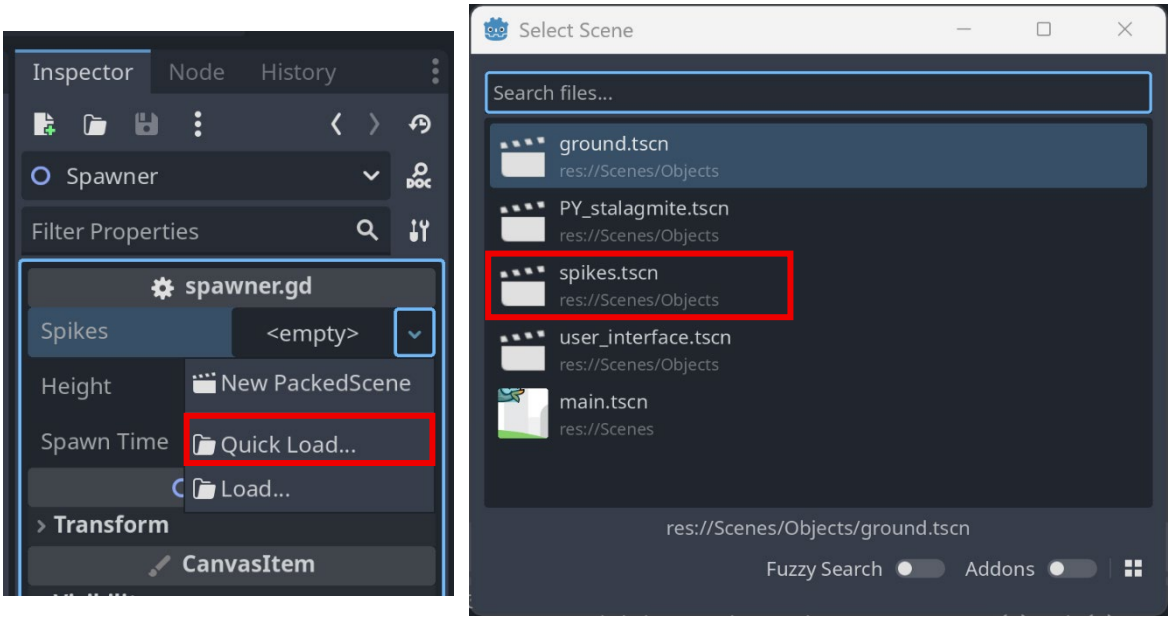
Check that **Spawner** is a **child** of **Main**, not Player.



73 In **FileSystem**, find **spawner.gd** in the Scripts folder and attach the script to the **Spawner** node.



74 In the **Inspector** for **Spawner**, locate **Spikes <empty>.** Select the **drop-down arrow** and select **Quick Load** to view all scenes in the FileSystem, then click the **spikes.tscn** scene.

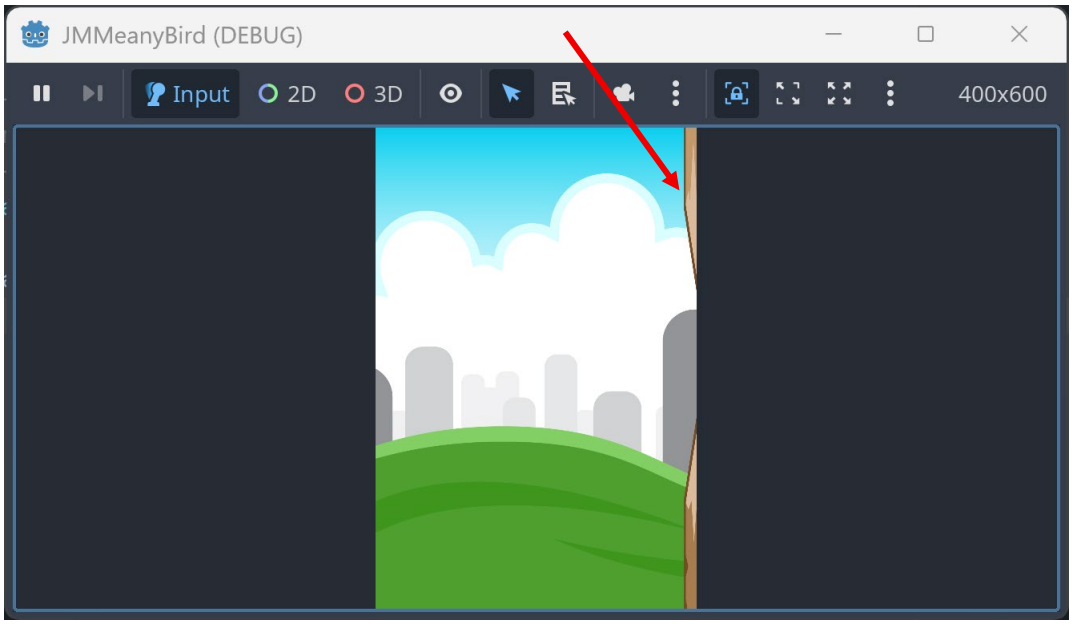


75 Underneath Spikes, set **Height to 200** and **Spawn Time to 2.0** to start. These values can be customized later.

Save the scene.

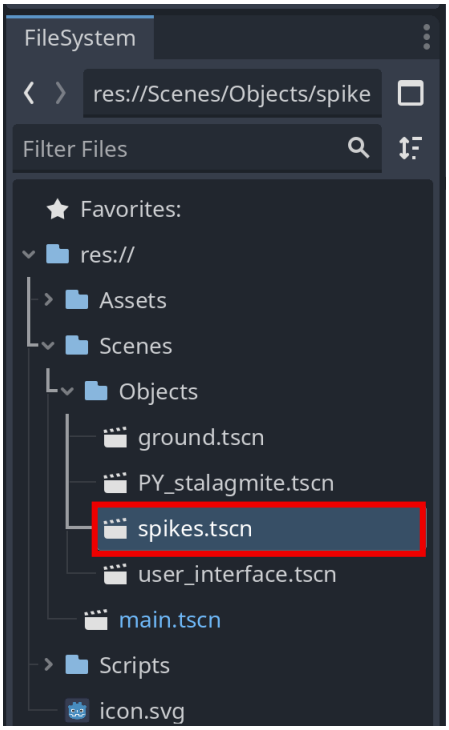
76 Playtest the project. Are there spikes spawning in the scene?

Notice that the spikes spawn on the right side of the viewport before disappearing, but they aren't moving across the screen!

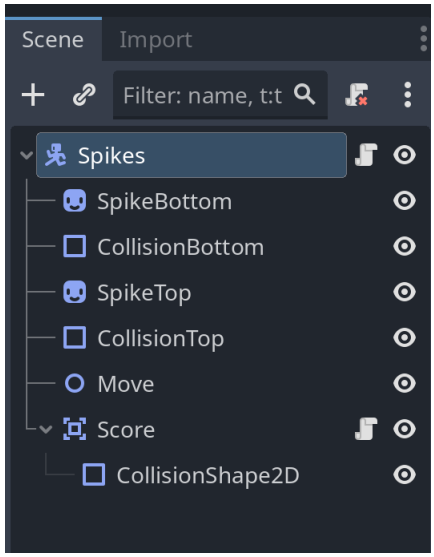


Close the playtest window.

77 In **FileSystem** under **Scenes**, find the **spikes.tscn** scene inside the **Objects** folder. Double click on the file to open the scene.

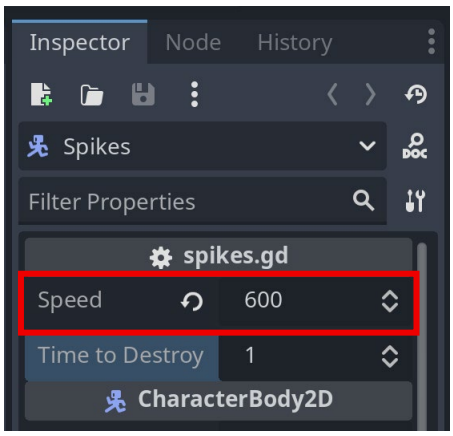


78 The spikes.tscn scene should open automatically and display a **new node tree** with a **new root node**, **Spikes**.



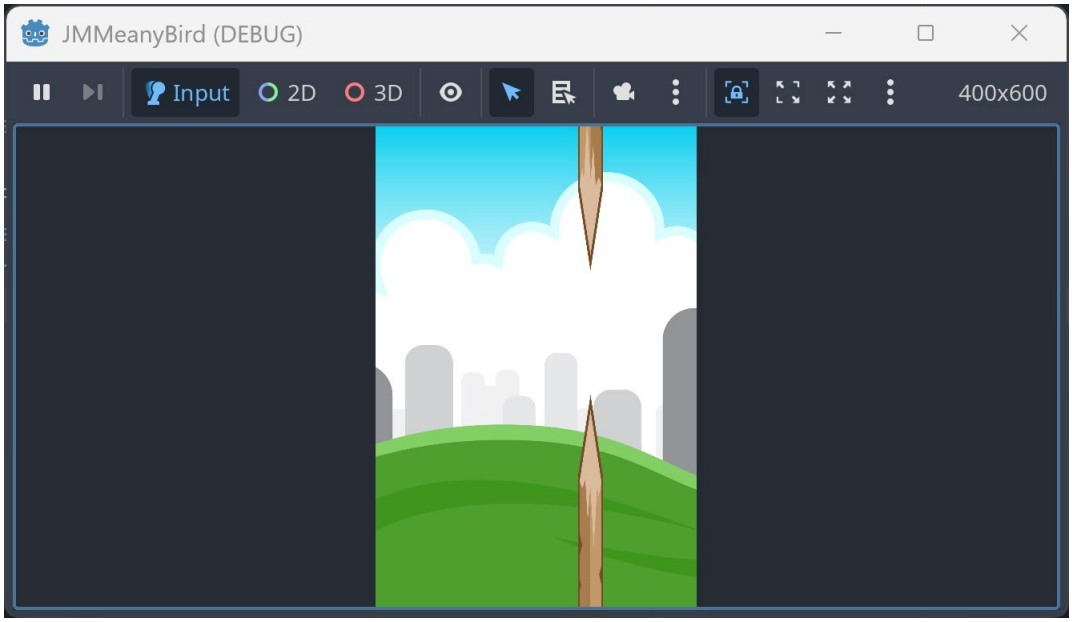
If new node tree has not appeared in Scene, toggle the scene from main to spikes.

79 In the **Inspector** for **Spikes**, set **Speed to 600** and save the scene.



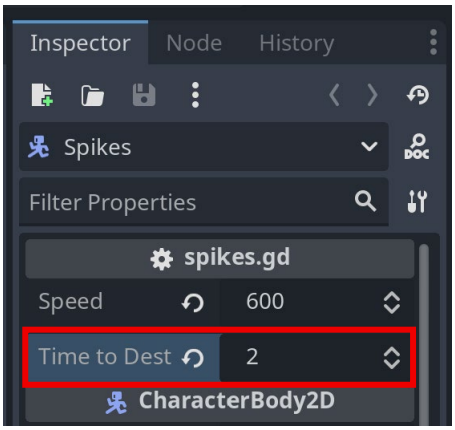
80 Playtest the project. What are the spikes doing in the scene?

The spikes now move across the screen but disappear before reaching the left side!

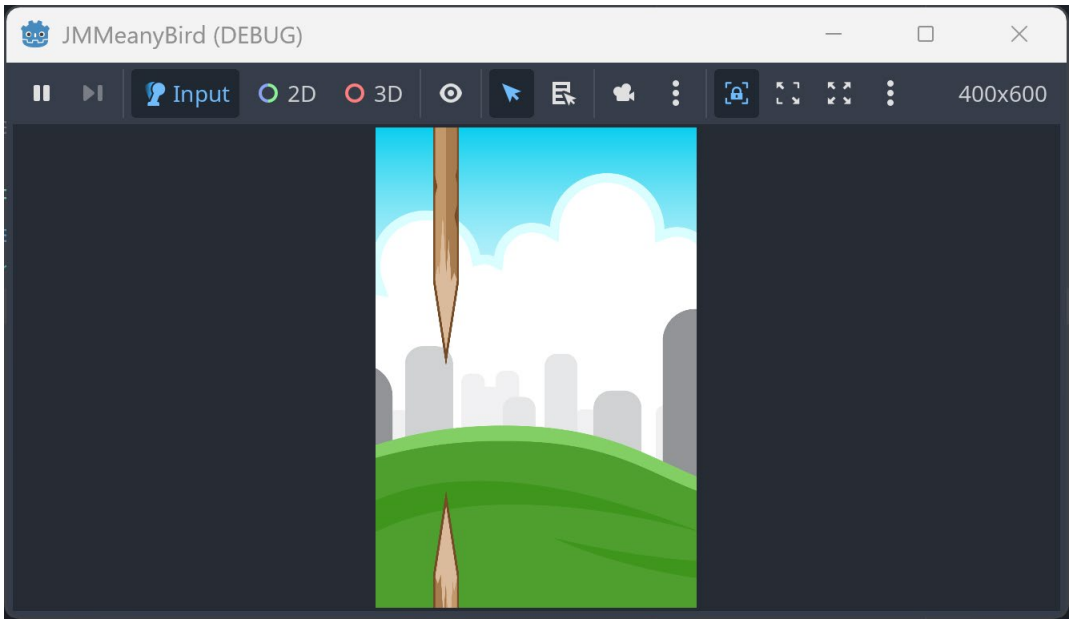


Close the playtest window.

81 In the **Inspector** for **Spikes**, set **Time to Destroy to 2** and save the scene.

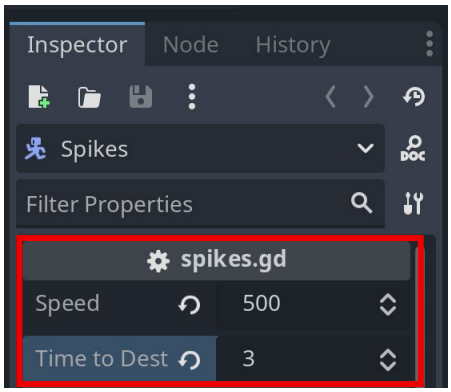


82 Playtest the project. Do the spikes move off the screen before disappearing?



Close the playtest window.

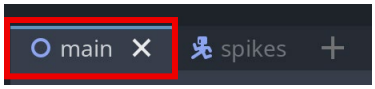
83 Customize the values of **Speed** and **Time to Destroy**.



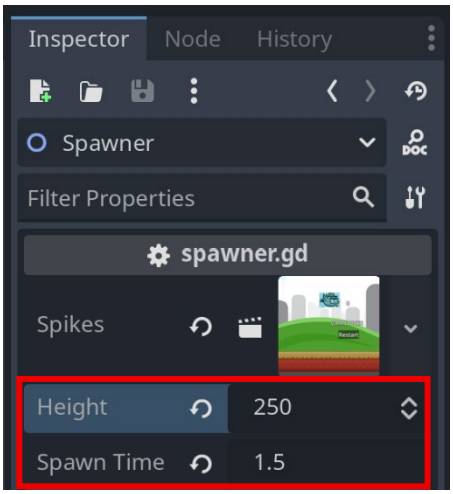
Speed: Choose a value that adds some difficulty to the game without making it impossible to avoid the spikes.

Time to Destroy: Choose a value that allows the spikes to leave the viewport before destroying them. Spikes moving at a slower speed will require a larger Time to Destroy.

84 Toggle back to the **main** scene.

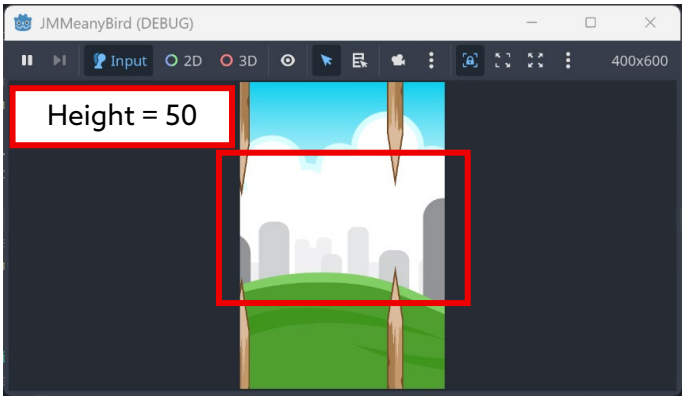


In the **Inspector** for **Spawner**, adjust the **Height** and **Spawn Time**.

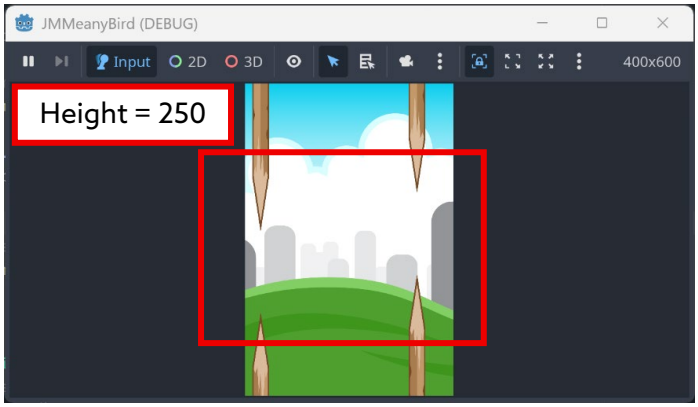


85 The spikes will spawn at a **random y position** from `-height` **to** `+height`**.**

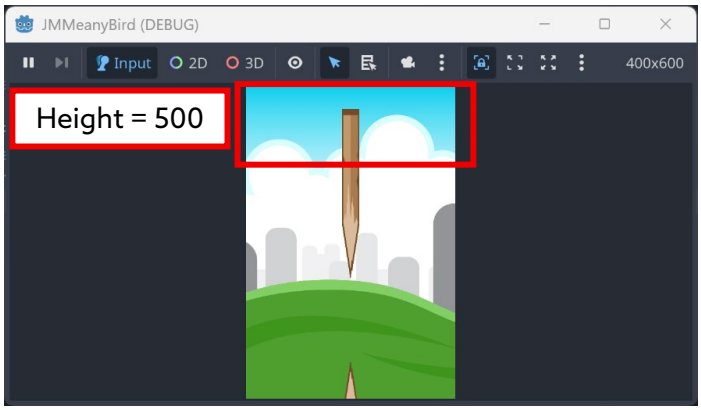
A smaller height value such as 50 will create level spikes with only a small height difference.



A larger height value such as 250 will create spikes with a larger height difference and provide a more challenging game.

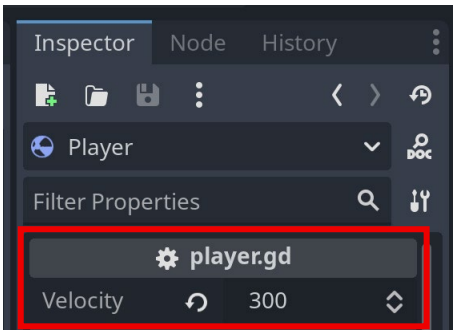


If the height value is too large, the bottom of the spikes can be seen.



When playtesting, make sure the bottom of the spikes cannot be seen.

86 If needed, return to the **Inspector** for **Player** and adjust the value of **Velocity**.



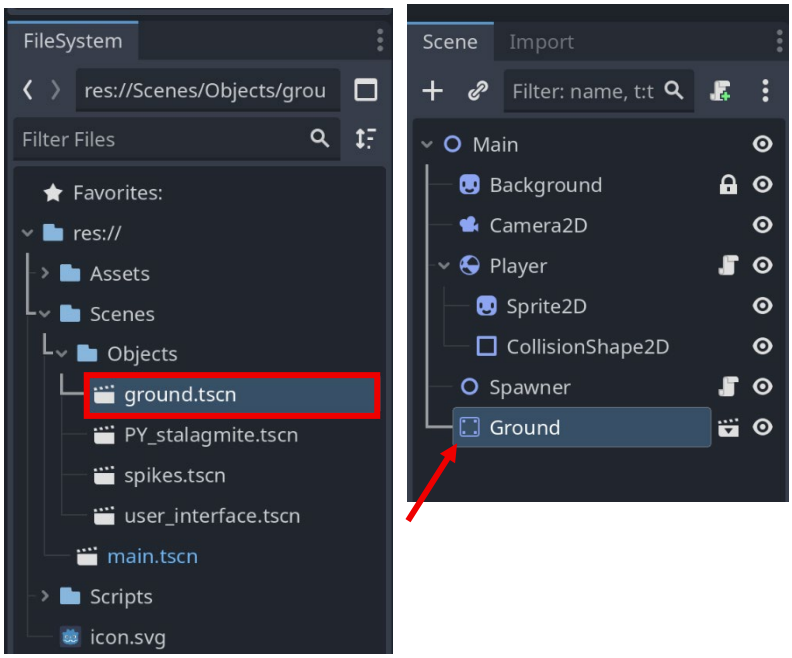
If there is a large difference in the spikes spawn height, a smaller velocity value could make it easier to avoid the low spawning spikes.

Pause for **Sensei Stop #5**!

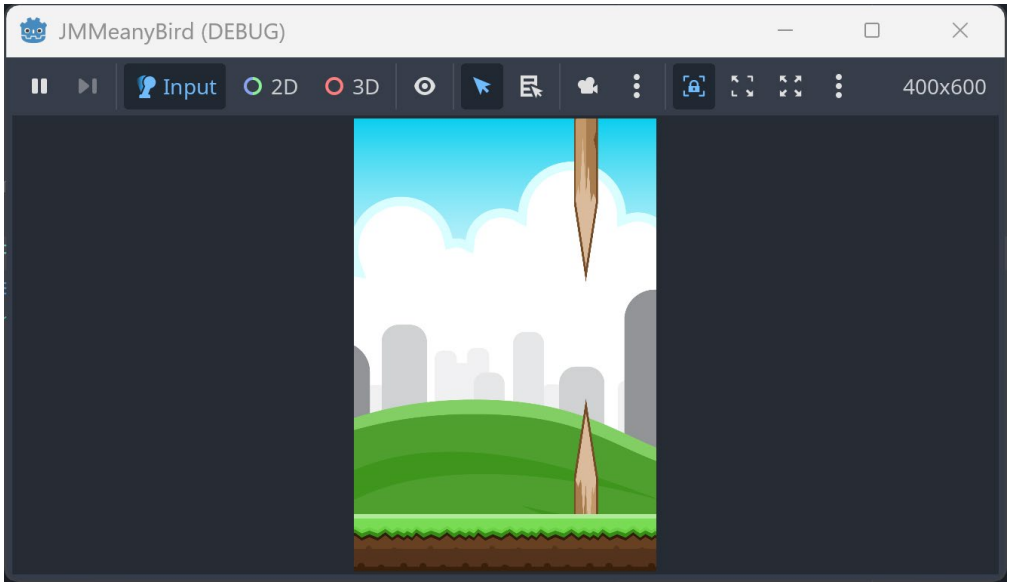
Before continuing, check with a Code Sensei and make sure the global script was added and the spawner is set up correctly.

Reminder: Save your work!

87 In **FileSystem** under **Scenes**, find **ground.tscn** inside the **Objects** folder. Drag the **ground.tscn** scene onto the **Main** node in Scene to make **Ground** a child to **Main**.



88 Playtest the project. What can be noticed about the ground and the spikes?

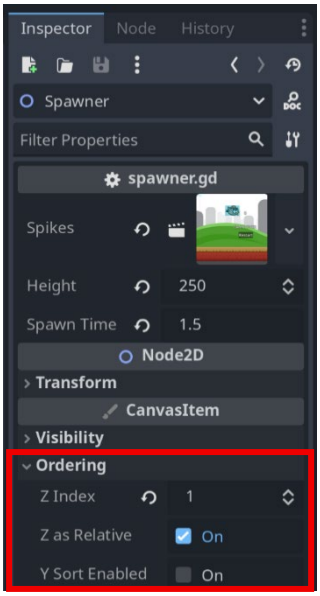


Close the playtest window.

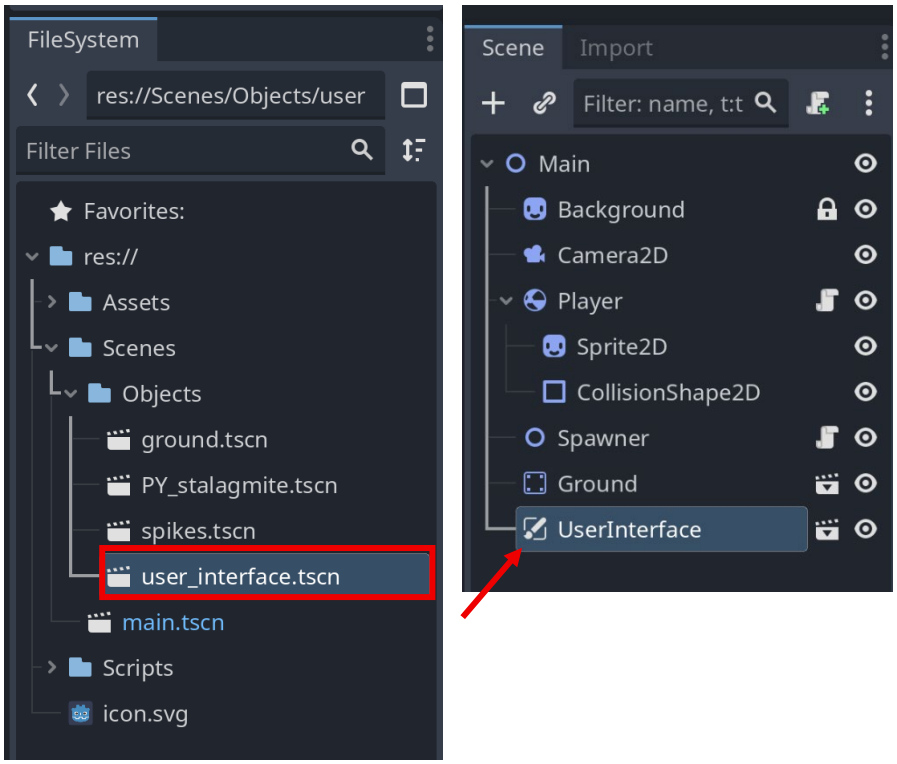
89 Notice that the spikes spawn *behind* the ground, instead of *in front* of it. This makes it hard to spot the bottom spike sometimes.

In the **Inspector** for **Spawner**, open the **Ordering** drop-down menu.

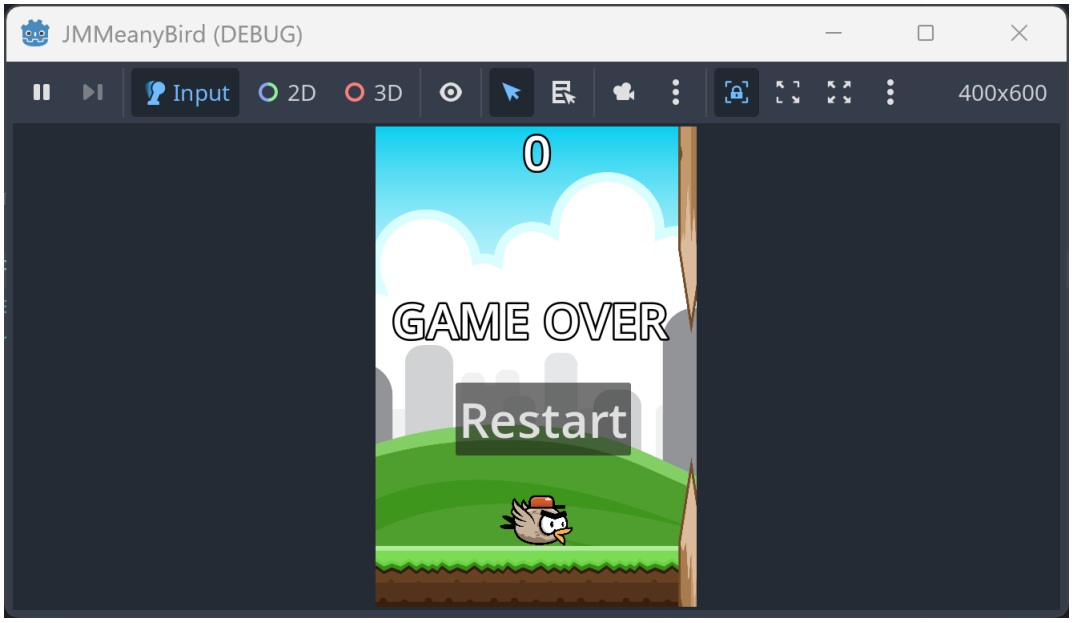
Change the value of **Z Index** from 0 to 1. This will make sure the spikes are always visible in front of the ground.



90 In **FileSystem** under **Scenes**, find **user_interface.tscn** inside the **Objects** folder. Drag the **user_interface.tscn** scene onto the **Main** node in Scene to make **UserInterface** a **child** to **Main**.



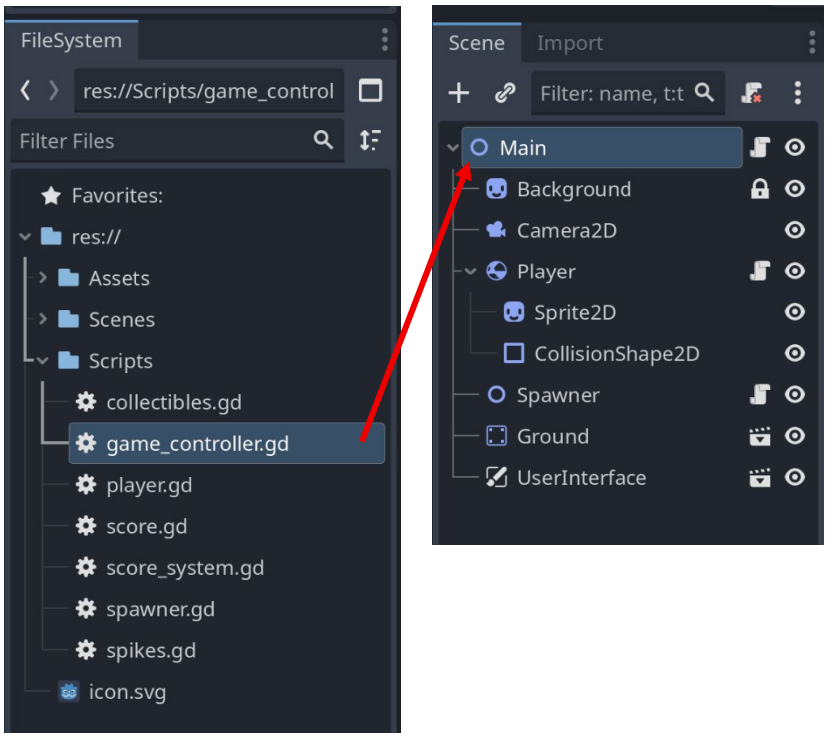
91 Playtest the project. Is something wrong with the user interface?



Close the playtest window.

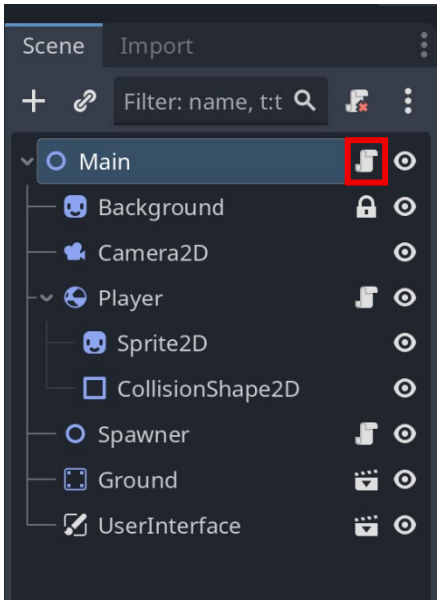
92 The user interface displays the game over canvas throughout the whole game! This can be managed through a game controller script.

In **FileSystem,** find the **game_controller.gd** script inside the Scripts folder and attach it to the **Main** node.

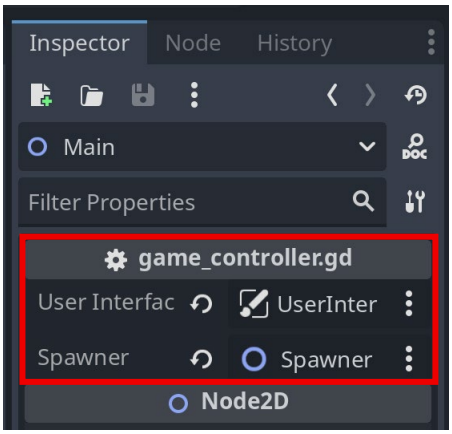


Click on the script icon to open the **game_controller.gd** script.

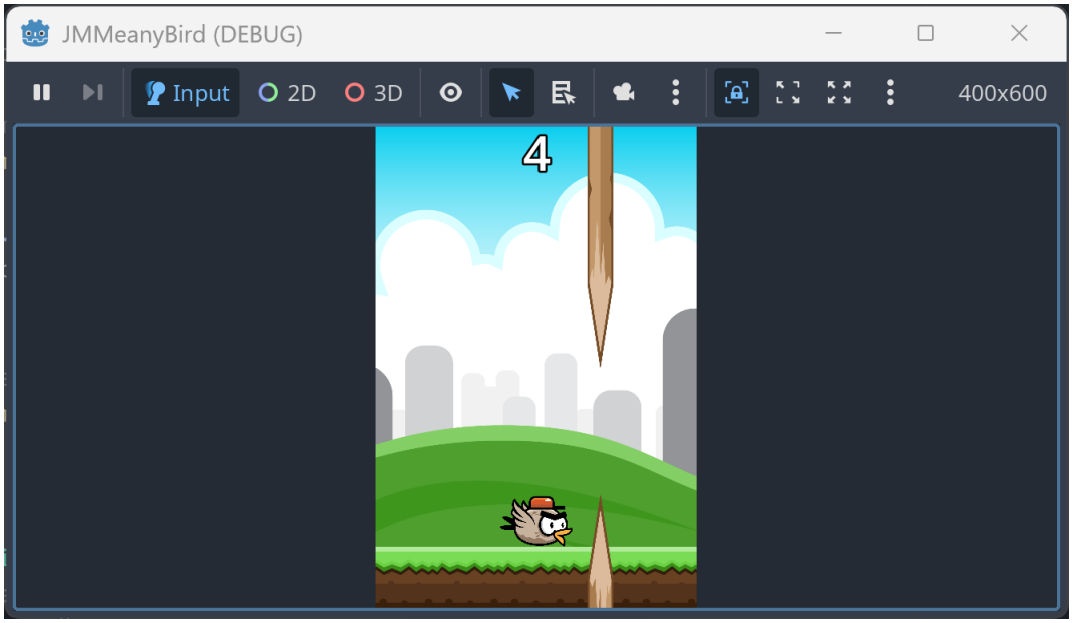
Read through the script – it's okay if you don't fully understand everything.



93 In the **Inspector** for **Main**, set **User Interface** to the UserInterface node and **Spawner** to the Spawner node.



94 Playtest the project. The game over screen is now gone, but what happens when the player collides with the ground or a spike?

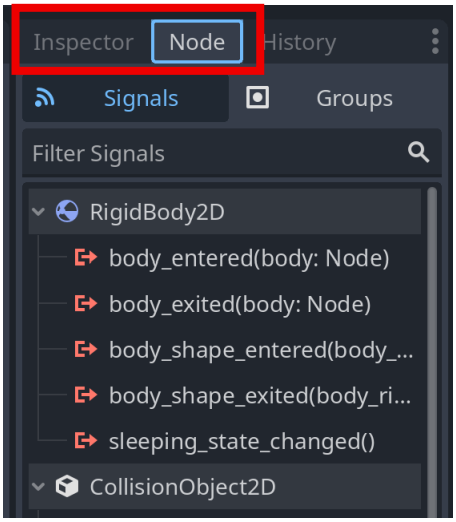


Close the playtest window.

95 When the player hits the ground or a spike, nothing happens!

A **signal** needs to be connected to make something happen when the Player collides with objects.

At the top of the **Player** node's Inspector, toggle the window to **Node**. What signal might be used?

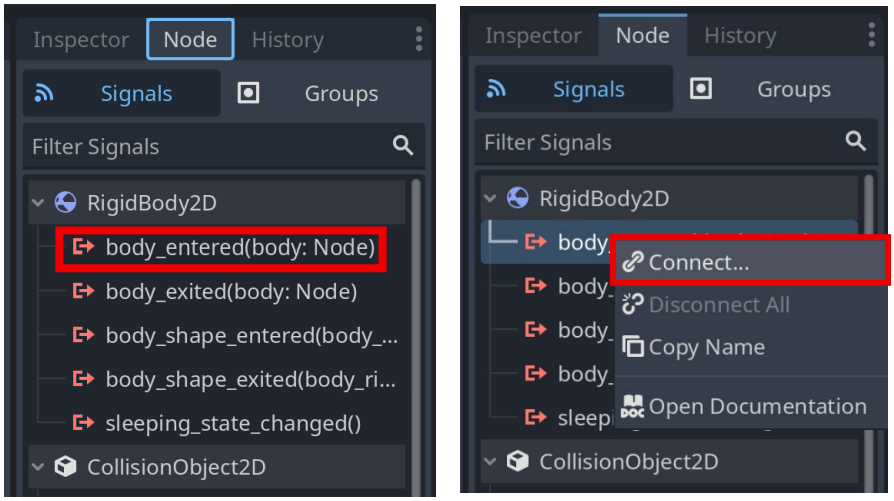


Reminder:

Signals are messages that nodes emit when something specific happens to them. Other nodes can connect to the signal and call a function when the event occurs.

96 The player's **body_entered** signal will be used to send a message when **Player** (a RigidBody2D) collides with the body of any other node.

To connect the **body_entered** signal, **double click** on the signal *or* **right click** on the signal and click **Connect**.

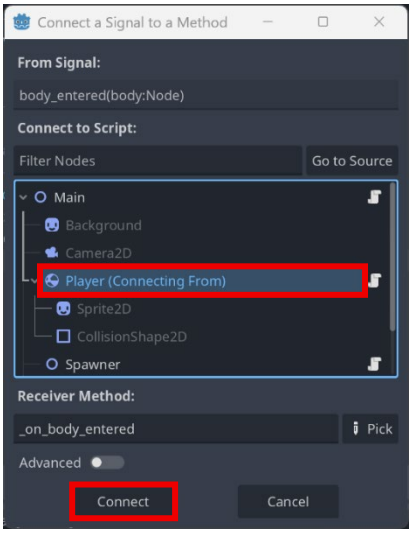


97 A signal needs to be connected to a **Receiver Method** in a script.

The **body_entered** signal will **connect** to the player.gd script which is attached to the **Player** node.

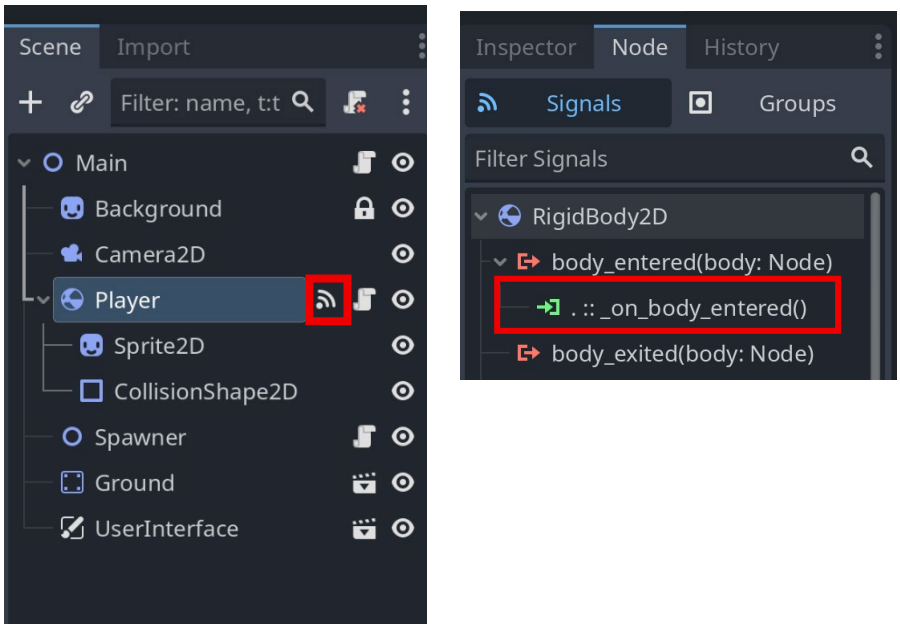
The player script does not contain a receiver method yet, so one will be automatically created.

Select **Player** and click **Connect**.



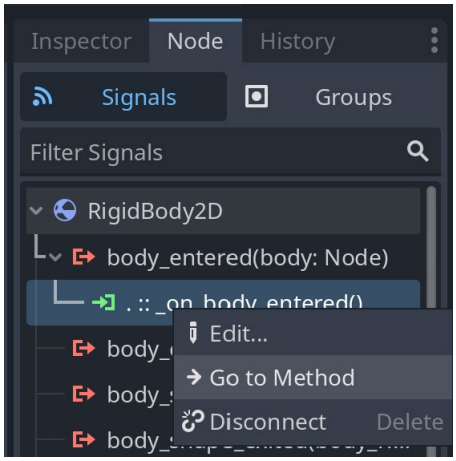
98 In Scene, the **signal symbol** will appear next to the Player node. This shows which nodes have connected signals.

In the Player's node window, the **green symbol** shows that the signal is connected to the receiver method **`_on_body_entered()`**.



99 After connecting the signal, the **player.gd** script should open automatically in the script editor.

The script can also be opened by **right-clicking** on the **receiver method** and selecting **Go to Method**.



100 In the **player.gd** script, the receiver method `_on_body_entered()` will appear below **TODO 3**. The same **green symbol** can be seen beside the method declaration. This shows the method is connected to a signal.

```
18
19  # --------------------
20  # TODO 3
21  # End the game
22  # --------------------
23
24  func _on_body_entered(body: Node) -> void:
25      pass # Replace with function body.
26
```

Pro Tip:

Some empty lines may appear between the TODO 3 comment and the method declaration. These empty lines *can* be removed but don't need to be.

101 The `_on_body_entered()` method will end the game when the Player collides with another object by calling the `game_over()` function from **game_controller.gd,** which is attached to the main node. A variable is needed to access the main node and its script!

Under **TODO 1** below the velocity variable, create a new `game_controller` variable of type `Node2D`, which begins with `@export` so it can be assigned in the Inspector.

```
3  # --------------------
4  # TODO 1
5  # Create the velocity variable
6  # --------------------
7  @export var velocity: int
8
9
```

Save the script.

102

Check the code! Update the script as needed.

```
3  # --------------------
4    # TODO 1
5    # Create the velocity variable
6    # --------------------
7    @export var velocity: int
8    @export var game_controller: Node2D
9
```

103

Inside the `_on_body_entered()` method below **TODO 3** there is a line of code that says `pass`. This line allows the method to be added to the script without causing errors.

```
20 # --------------------
21   # TODO 3
22   # End the game
23   # --------------------
24 func _on_body_entered(_body: Node) -> void:
25     pass # Replace with function body.
26
```

Remove this line of code.

104

The `_on_body_entered()` method will call a function, `game_over()`, which is inside **game_controller.gd**. This function triggers the end of the game.

Inside the `_on_body_entered()` method add in the code to access **game_controller.gd** by using the `game_controller` variable and call its `game_over()` function.

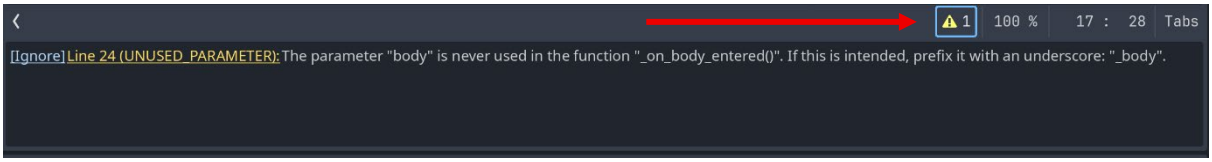
```
20 # --------------------
21   # TODO 3
22   # End the game
23   # --------------------
24 func _on_body_entered(body: Node) -> void:
25     game_controller.game_over()
26
```

access script

call function

105

A warning will appear in the script editor. Click on the warning symbol and read the warning.



What might it mean? How could it be fixed?

106

The parameter `body` is not used in the `_on_body_entered()` method.

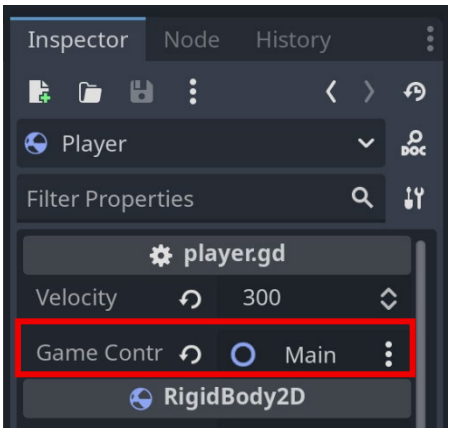
This does not cause any issues in the code or prevent the game from working. To prevent the warning from showing up, add an **underscore** _ in front of the parameter `body`. Godot will ignore the parameter and the warning will disappear!

```
20  # --------------------
21  # TODO 3
22  # End the game
23  # --------------------
24  func _on_body_entered(_body: Node) -> void:
25      game_controller.game_over()
26
```

Save the script.

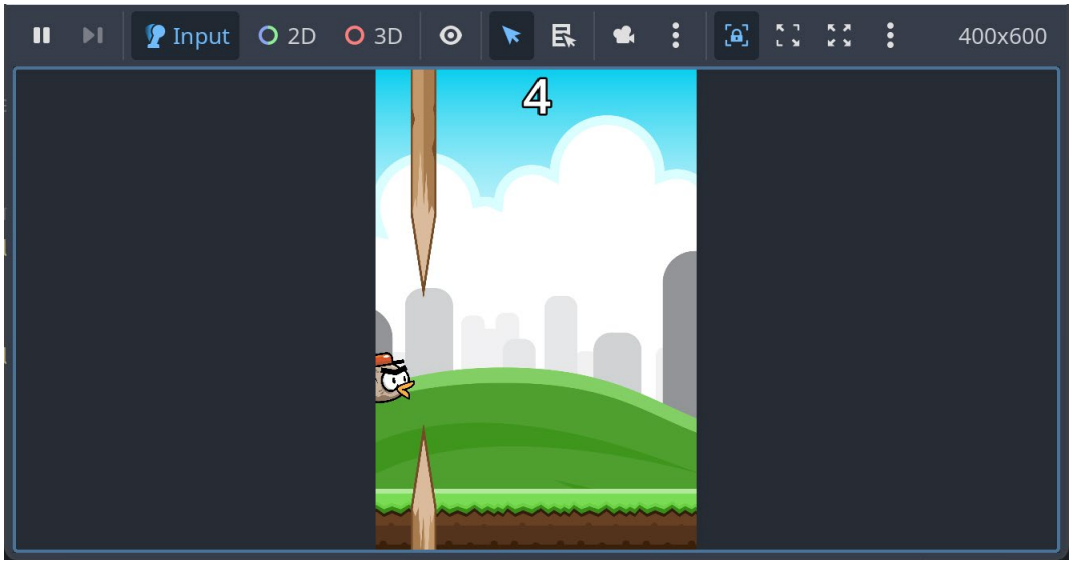
107

In the **Inspector** for **Player**, set **Game Controller** to the **Main** node.



108

Playtest the project. What happens when the Player hits the ground or the spikes?



Close the playtest window.

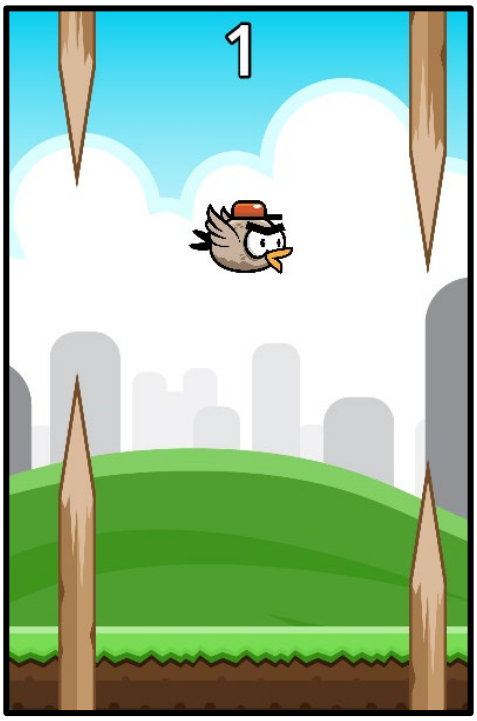
109

The game still won't end when the player hits the ground or a spike.

This is because, by default, the player's RigidBody2D does not emit signals when colliding with other bodies.

Return to the **Inspector** for **Player** and select the **Solver** drop-down menu. In Solver, turn on **Contact Monitor** and set **Max Contacts** to 1.

110 Playtest the project. Congratulations! You've completed another project!



Pause for **Sensei Stop #6**!

Congratulations on building your first infinite side-scrolling arcade game in Godot!

- ☐ How can collision shapes be fitted around image textures?
- ☐ What are some differences between coding in JavaScript and GDScript?
- ☐ Why might Signals be useful?
- ☐ What was needed to make the Bird collisions work after the signal was connected and coded

Reminder: Press CTRL+S to save your work and submit!