CODE NINJAS®

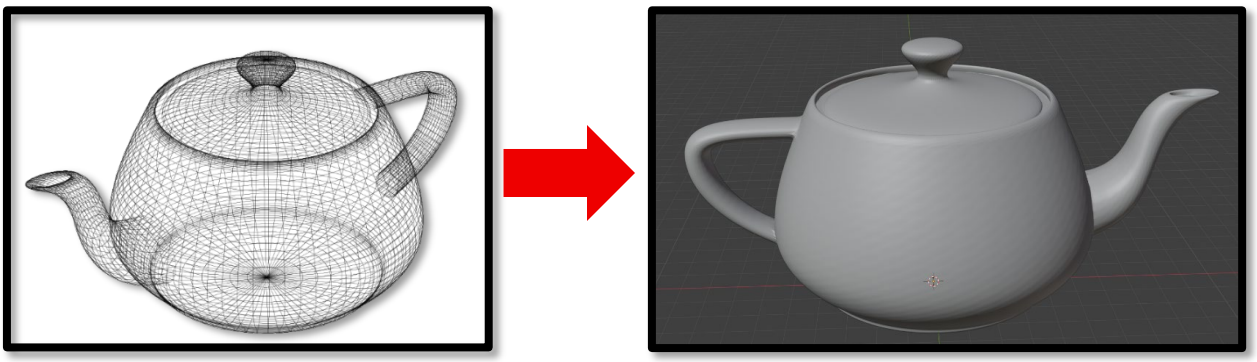
Bronze Belt Ninja Guide

Activity 05: Don't Touch the Cubes

Mesh Instances

When making 3D games, there will often be simple or complex shapes that need to be visually displayed in the 3D space! Think **platforms**, **barrels**, **characters**, **environments**, and more! These shapes are represented using a "Mesh".

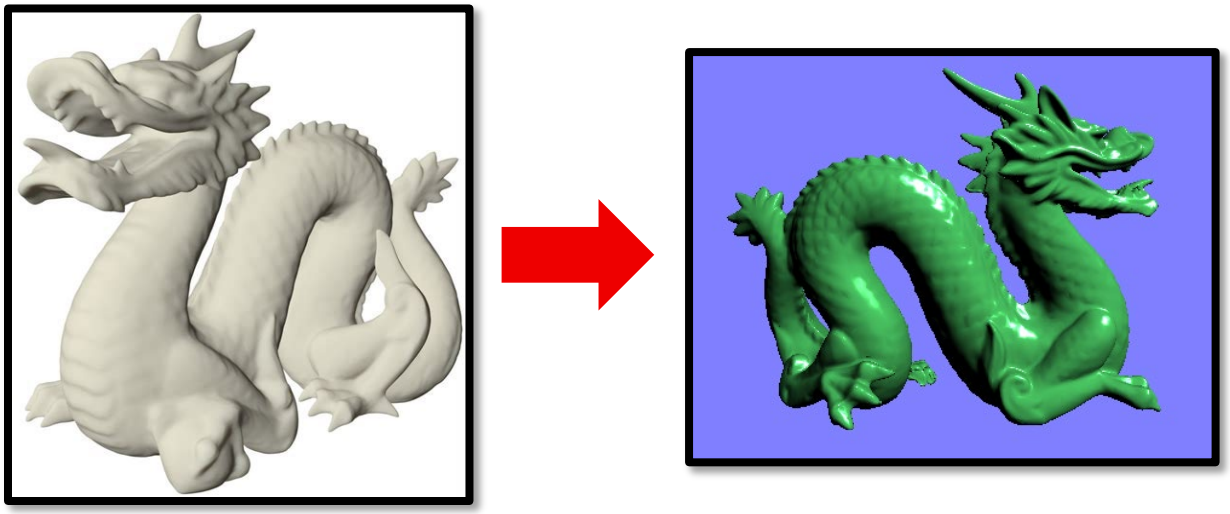
A **Mesh** is like the **skeleton** or **wireframe** for a 3D model. It defines the structure of an object by using a combination of **vertices**, **edges**, and **faces**. This structure allows it to be rendered in 3D space as a solid object!



THE FAMOUS UTAH TEAPOT AS A WIREFRAME AND AS A SOLID OBJECT.

SOURCE: TEAJOURNEY.PUB/UTAH-TEAPOT

MeshInstance3Ds are nodes in Godot that allow meshes to interact with lighting in Godot's 3D game environment. Additionally, they allow **materials** to be assigned to each instance of a mesh so that the **color**, **reflectance**, and **other properties** of the object can be tweaked.

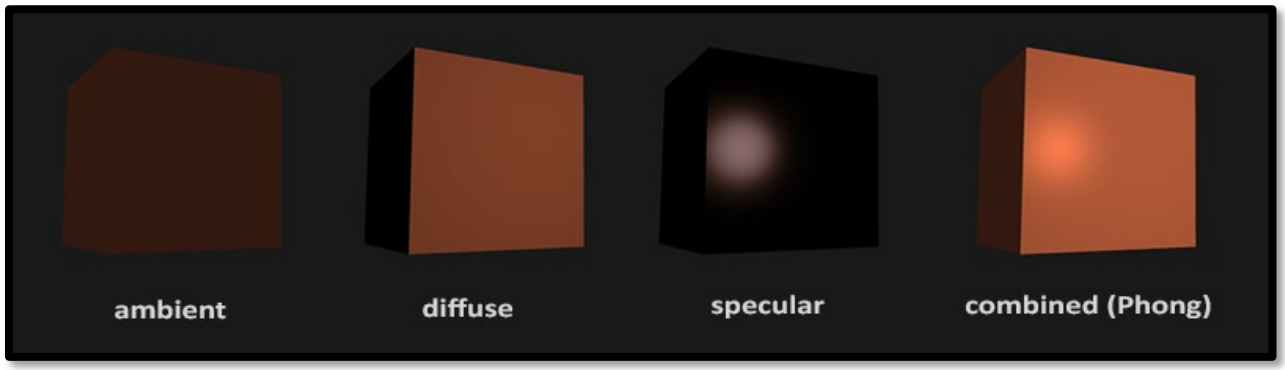


BEFORE AND AFTER OF ADDING A MATERIAL TO THE STANFORD DRAGON MESH.

SOURCE: RESEARCHGATE.NET

3D Lighting Model

In 3D environments lighting is considered an essential piece of the puzzle. It's what makes the difference between dark caves and bright overworlds! There are many different lighting models, but the one that is used in Godot is called the **Blinn-Phong** lighting model. It is composed of three parts that are added together:



AN EXAMPLE OF THE PHONG LIGHTING MODEL ON A CUBE.

SOURCE: LEARNOPENGL.COM/LIGHTING/BASIC-LIGHTING

1. **Ambient lighting** (lighting that exists no matter what from all angles)
2. **Diffuse reflection** (lighting based on the angle between a face and a light source)
3. **Specular highlights** (lighting based on the angle between the viewer, a face, and a light source)

Godot uses a **WorldEnvironment's Background Color** to calculate the **Ambient lighting**. Godot also uses any light sources it can find to calculate the **Diffuse reflection** on all objects in a scene, and the **specular component** must be manually set for each material.

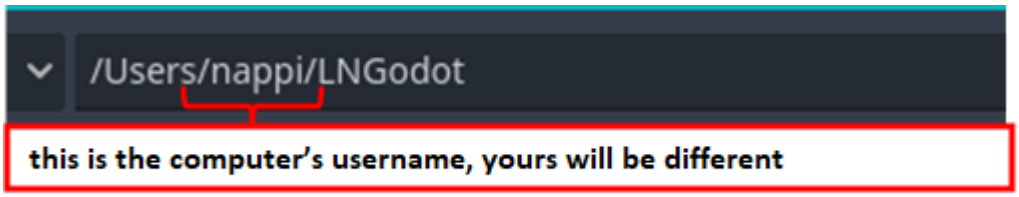
Activity 05: Don't Touch the Cubes

In this activity, you will create an infinite 3D cube-dodging game where the player moves to avoid falling into the incoming cube obstacles. This project will help you learn about Godot's 3D game environment, mesh instances, Vector3s, and what camera frustums are.

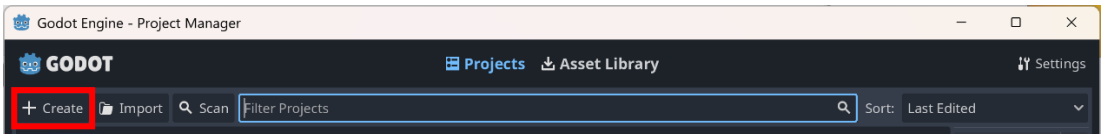
By the end of this activity, you will have explored how to create and use RigidBodies, customize lighting in a game, and use mesh instances to generate collision shapes based on a mesh.

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

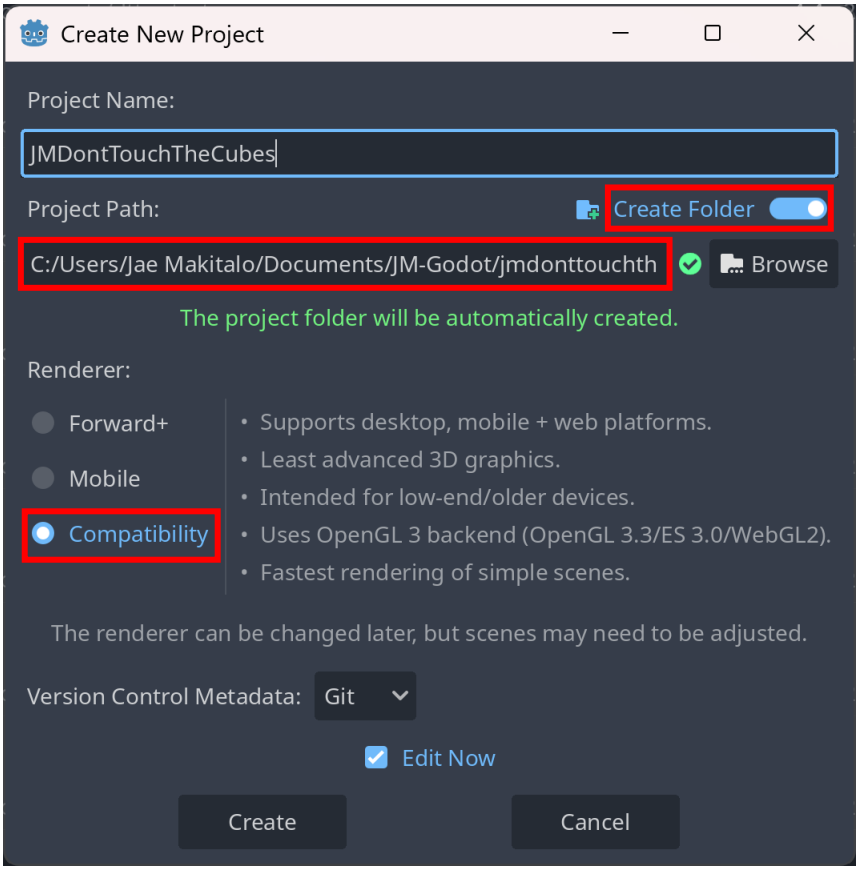
Don't worry if your path looks slightly different from the image shown! All computers have their own username.



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

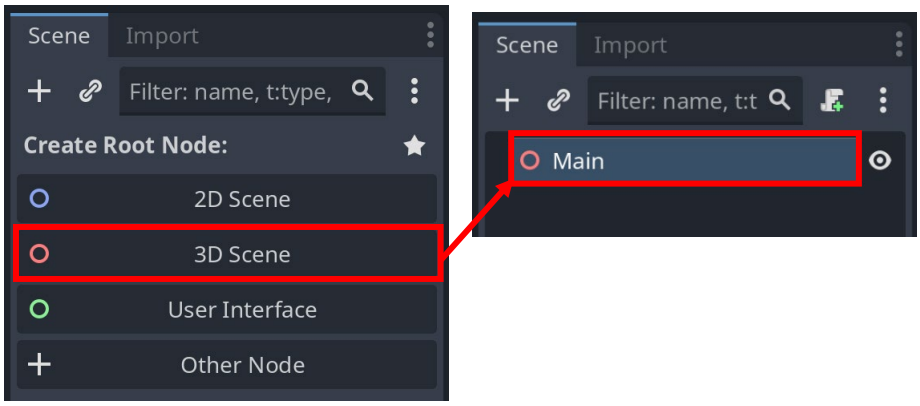


3 Name the project **[MyInitials]DontTouchTheCubes** and adjust the project path so the project is being saved in your folder. Make sure **Create Folder** is toggled on, set the renderer to **Compatibility**, then click **Create**.



4

Select the **3D Scene** as the **root node** for this project. Rename the **Node3D** to **Main**.



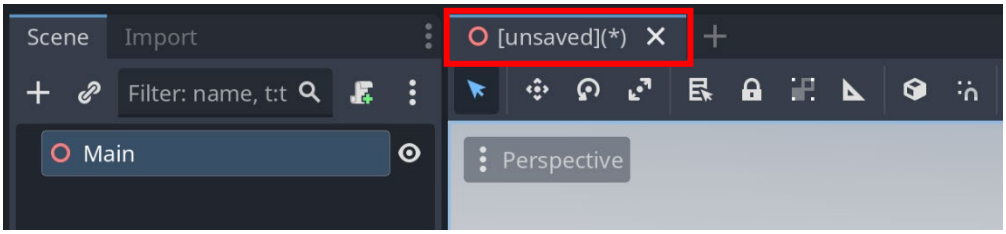
Reminder:

Rename a node by double-clicking on it.

5

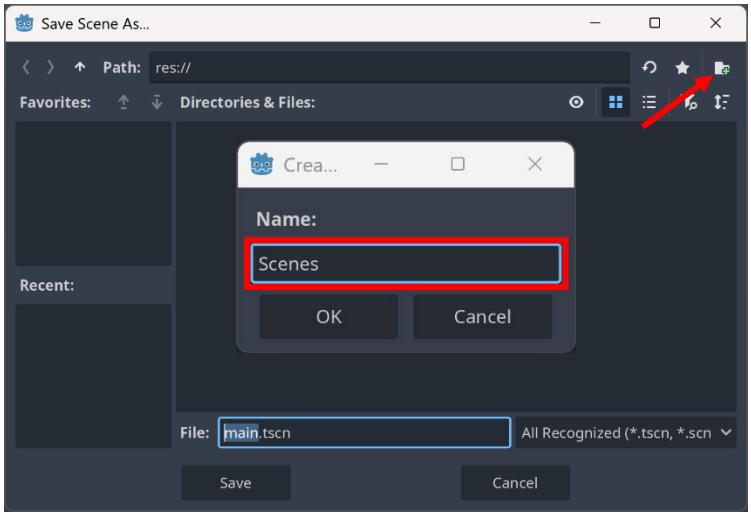
The **main scene** needs to be saved.

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



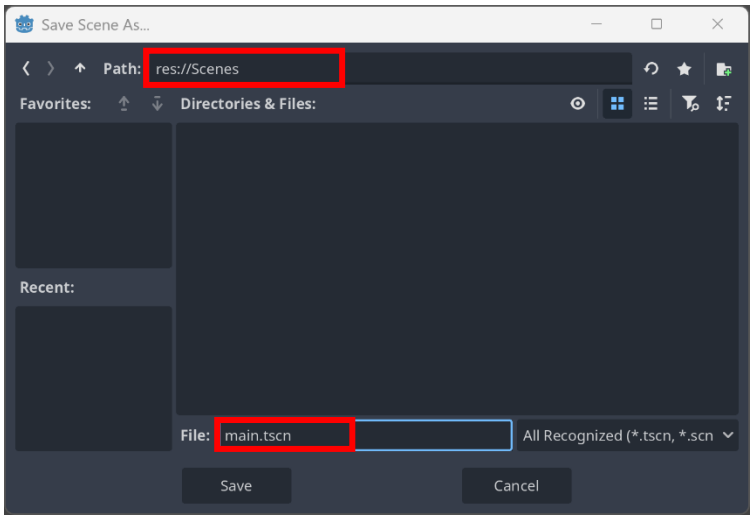
6 This project will contain multiple scenes, so use a **Scenes folder** to stay organized.

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



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

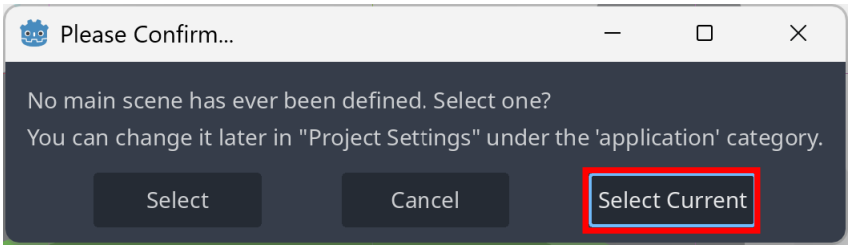
Check that the file is called **main.tscn** and click **Save**.



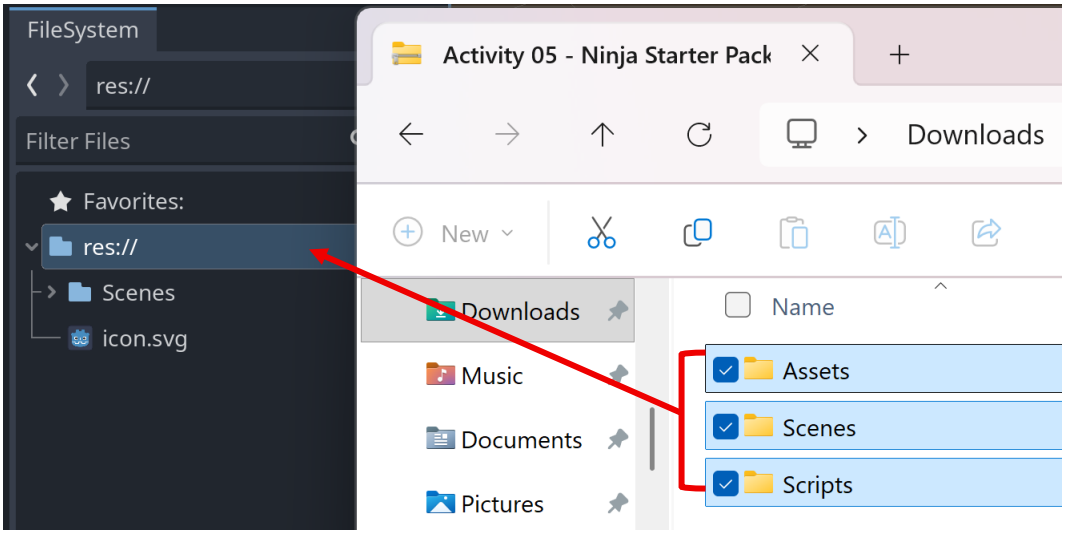
8 Click the **play** button to **playtest** the project.

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

Close the playtest window.



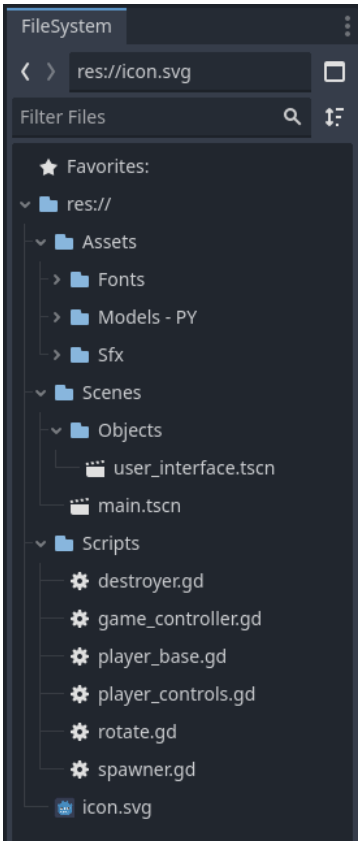
9 Extract **BB Activity 05 - Ninja Starter Pack** and select all folders inside. Drag them into the **res://** folder in **FileSystem**.



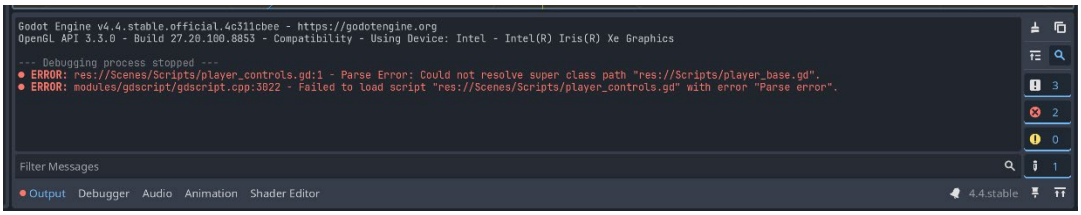
Reminder:

Right-click on the zip file and select **Extract All**. Press CTRL + J on the keyboard to reopen the **File Explorer**.

10 Make sure that **FileSystem** has everything as seen in the image.



11 Notice that there are several errors that appear in the **Output** console. These can be ignored for now.



Pause for **Sensei Stop #1**!

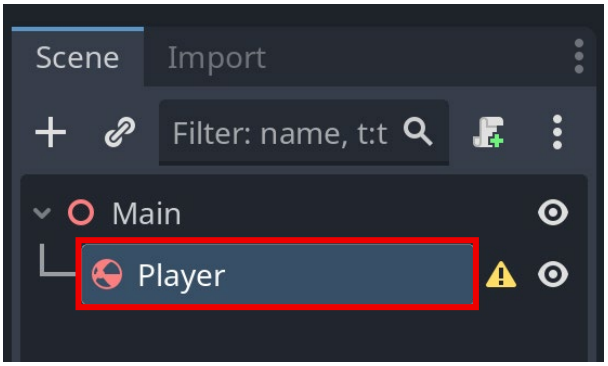
Before continuing, check with a Code Sensei and make sure the **main scene** is set up and all files are imported correctly.

Reminder: Save your work!

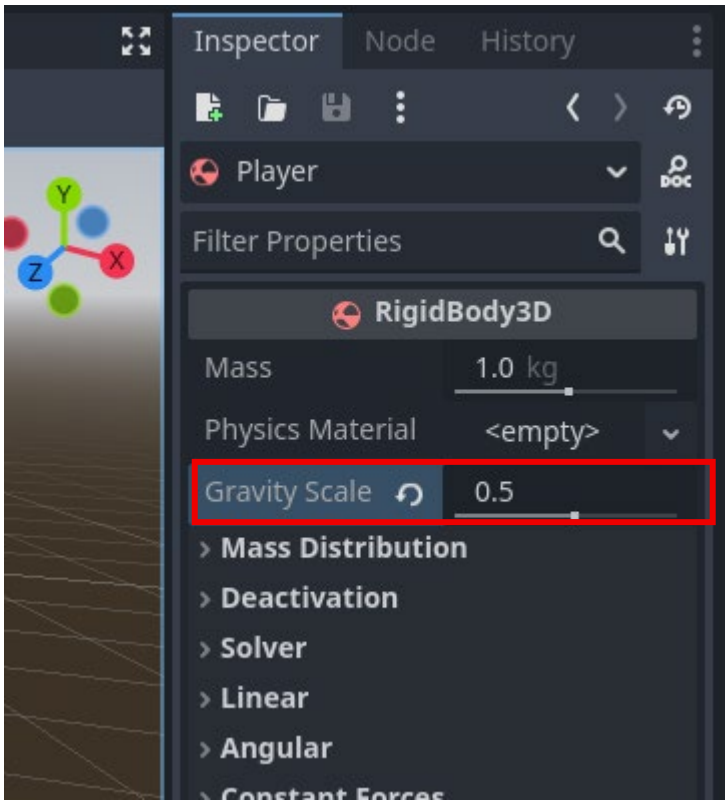
12 Time to make the Player!

Add a **RigidBody3D** node as a child to **Main**. Rename the node to **Player**.

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

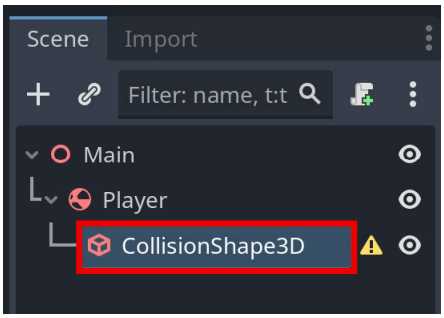


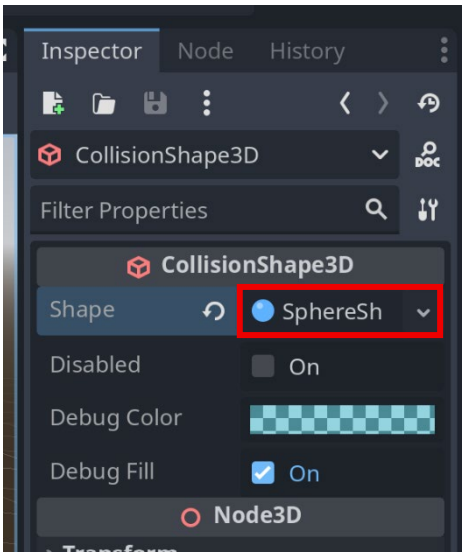
13 To improve the game play, set the Player's **Gravity Scale** to **0.5**. This will make the game more forgiving towards the player when the game starts.



14 To remove the hazard symbol, add a **CollisionShape3D** node as a child to **Player**.

In the **Inspector**, set its shape to **SphereShape3D**.



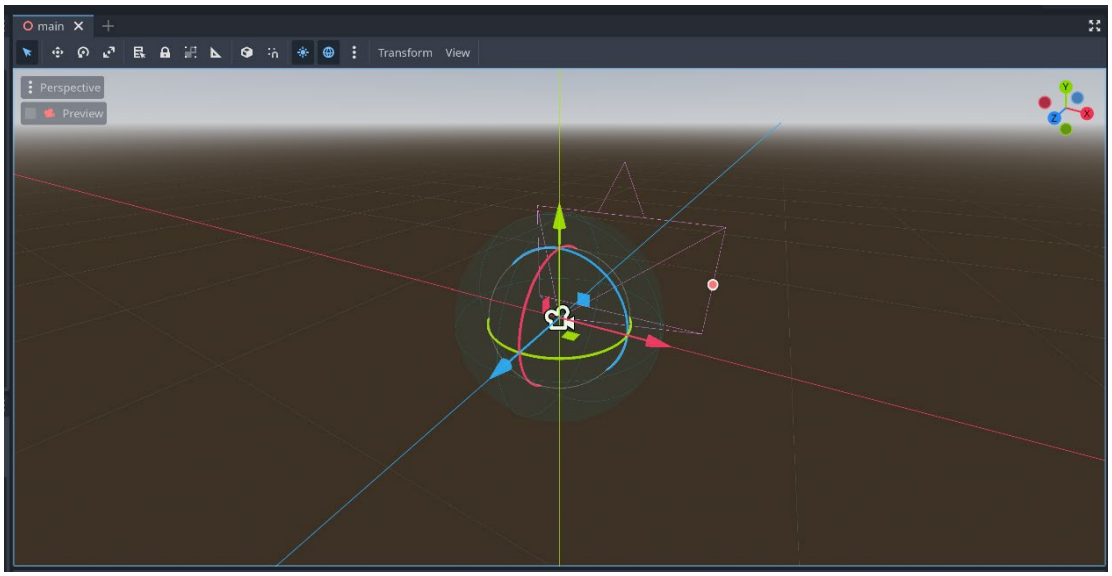


15 This game will be from a first-person perspective, which means the viewport will display the game from the perspective of the player.

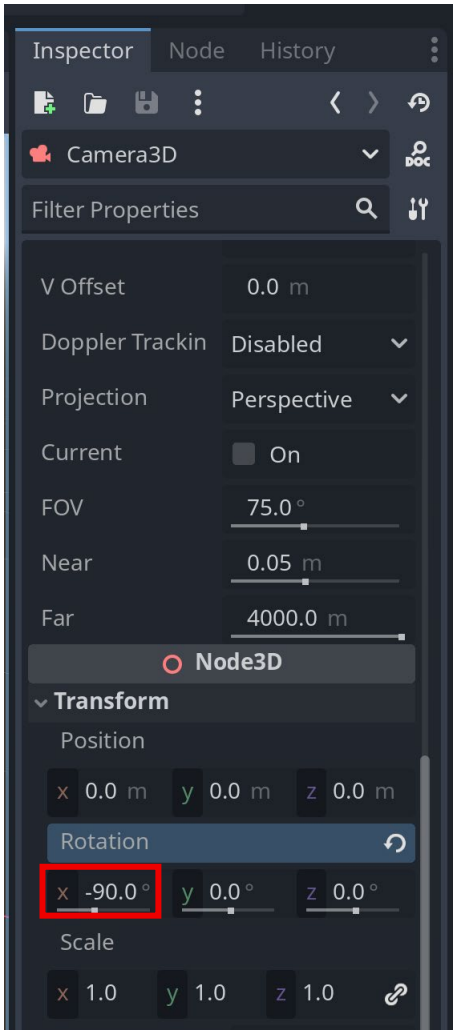
To do this, add a **Camera3D** node as a child to **Player**.

16 Notice how the camera starts off by facing towards the side. In this game, the player falls towards newly spawning cubes which must be avoided.

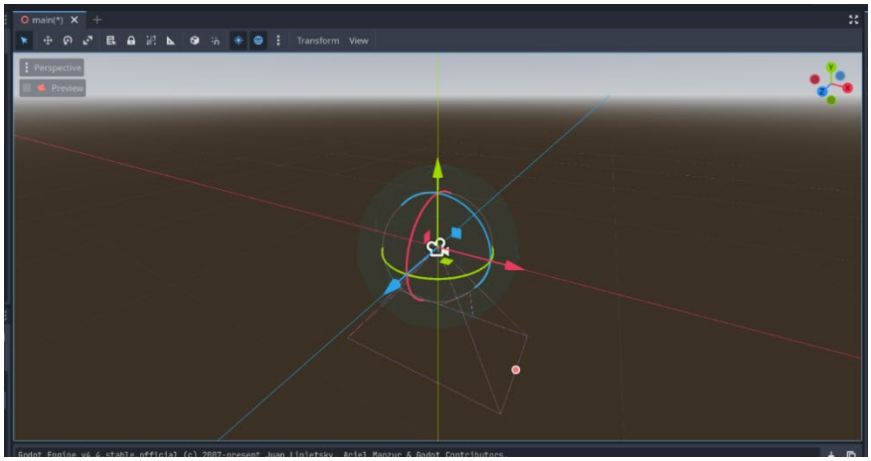
The camera needs to face down so the Player can see the cubes coming and avoid them.



17 In the Inspector for **Camera3D**, open **Transform** to find **Rotation**. Set the **x** value to **-90** degrees.



This should make the camera face downwards as seen in the image.



New Concept: Frustum

A Frustum is the shape that illustrates what a camera can see! It is shown by the pink lines that extend out of Camera3D nodes in Godot. The area enclosed by the pink lines may seem small, but the Camera3D node has "Near" and "Far" parameters that determine which objects can be seen by the camera. Notice how large the "Far" parameter is compared to "Near"! The frustum stretches much further than what the preview in Godot shows.

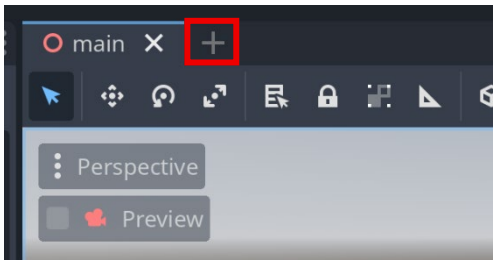
Pause for **Sensei Stop #2**!

Check in with a Code Sensei before moving on and make sure the Player is set up properly.

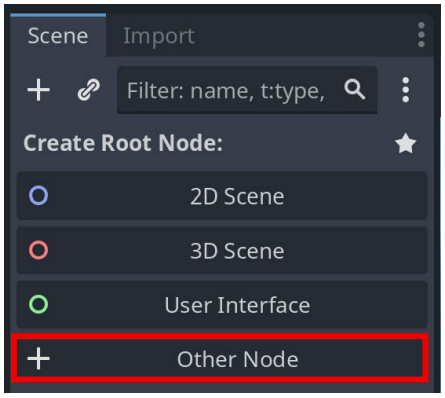
Reminder: Save your work!

18 Time to make the Cube scene!

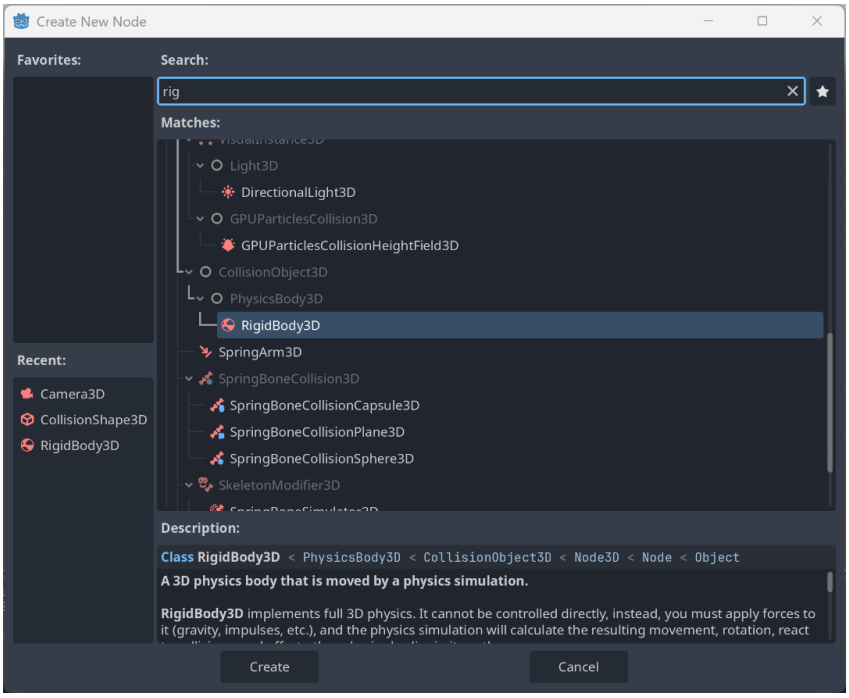
To create a new scene, press **CTRL+A** on the keyboard or click the **+** sign next to the main scene at the top of the screen.



19 The root of this scene will not be a **Node2D**, **Node3D**, or a **Control** node. Instead, select **Other Node**.

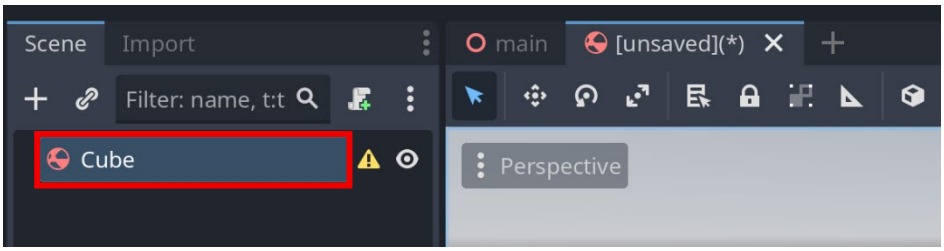


20 Search for **RigidBody3D** and click **Create** to select it as the root node of the **Cube scene**.



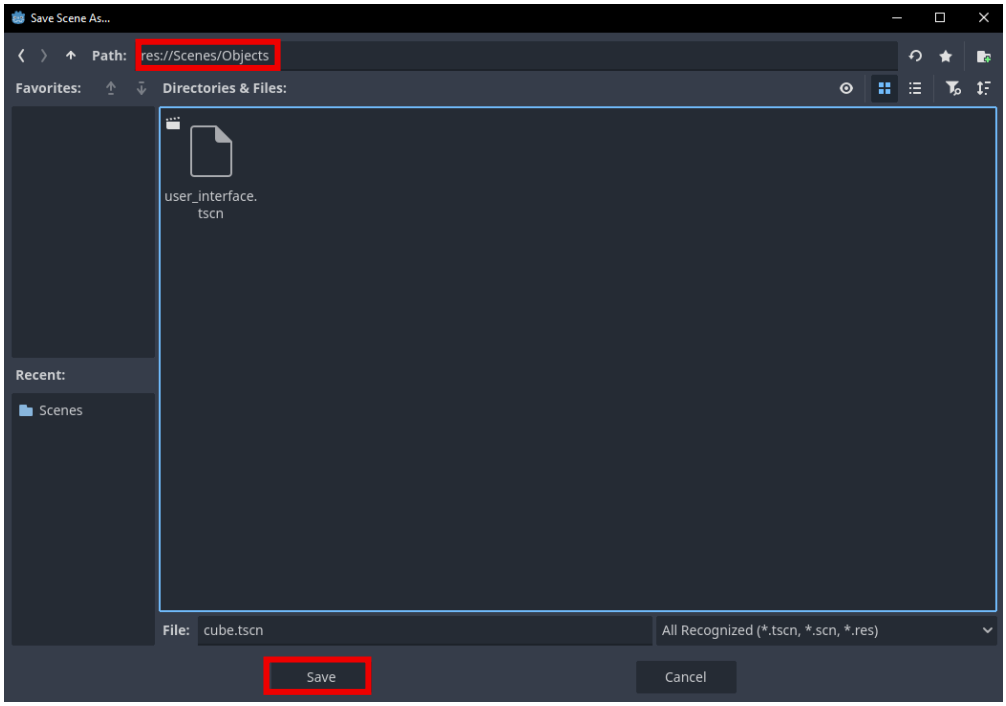
21 Rename the **Rigidbody3D** node to **Cube**.

Once again, ignore the hazard symbol. This will be fixed after giving it a **CollisionShape3D**.

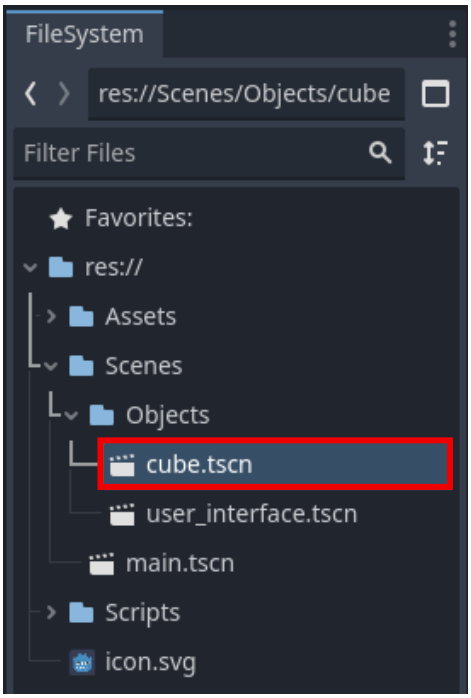


22 Press **CTRL+S** to save the cube scene and open the **Save Scene As...** window.

Navigate to the **Objects** folder and click **Save**.



23 Notice in **FileSystem** the **cube.tscn** scene has been created inside of the **Objects** folder (which is placed inside the **Scenes** folder).



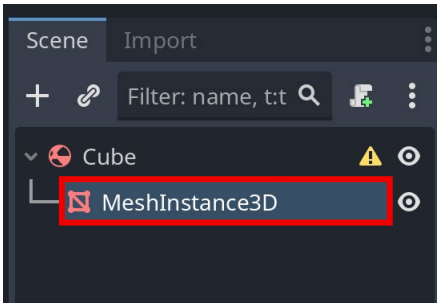
Pro Tip:

In **FileSystem**, click on a folder's arrow to expand its contents below. Click it again to collapse its contents.

24 For the Player to avoid the cubes while falling, the cubes must be visible and have a collision shape.

Adding just a collision shape will allow collisions with the cubes but they would still be completely invisible.

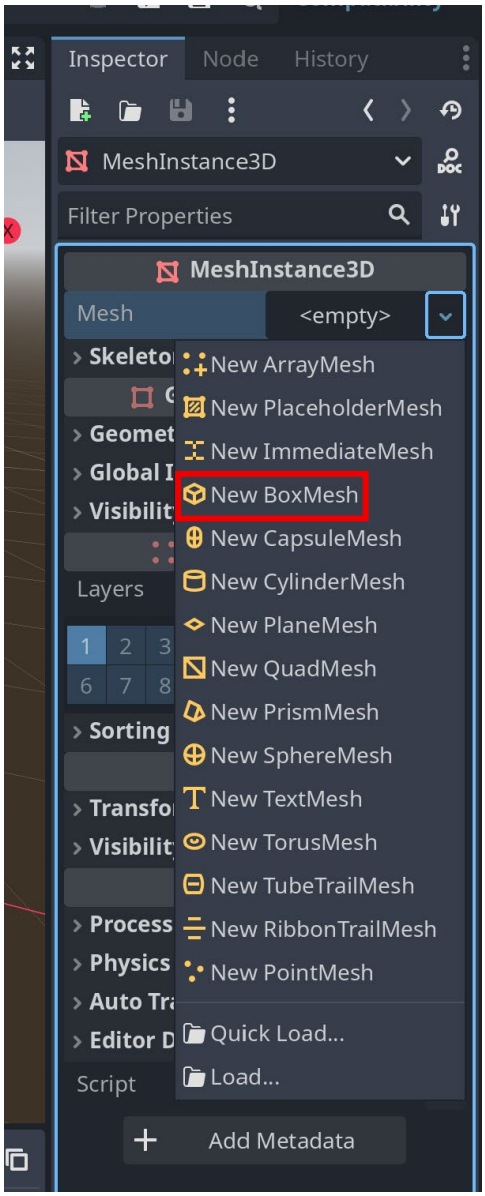
Add a **MeshInstance3D** as a child node to **Cube**.

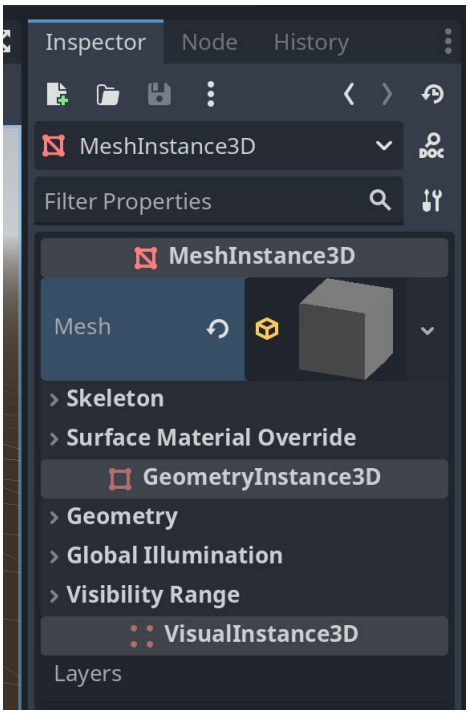


Reminder:

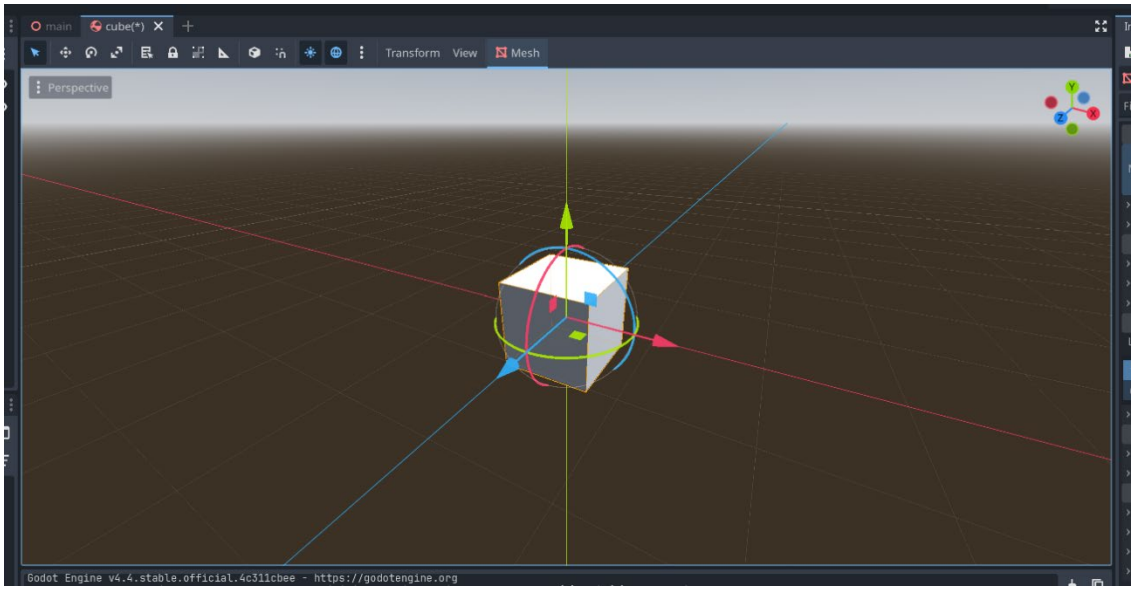
A **MeshInstance3D** allows the developer to import any object/mesh. The node can calculate an optimal **CollisionShape3D** to fit the object's mesh.

25 In the **Inspector** for **MeshInstance3D**, select **BoxMesh** as its mesh in the dropdown.

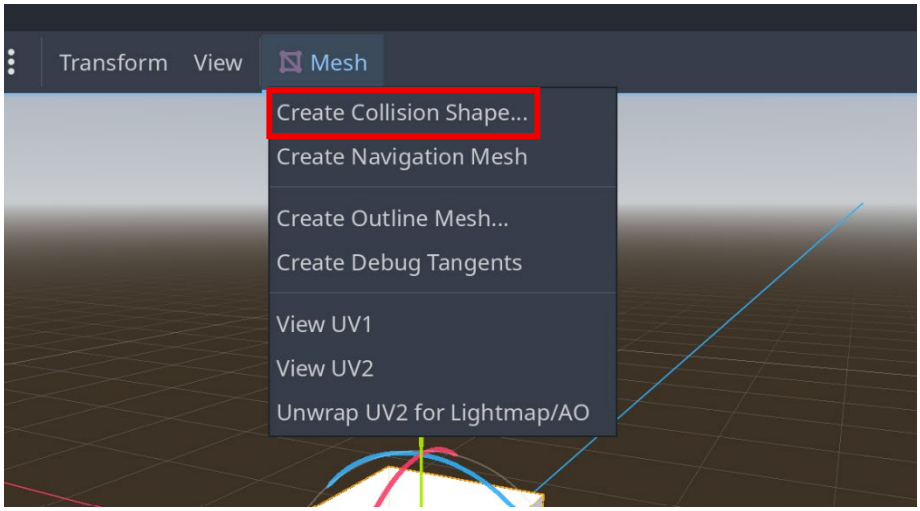




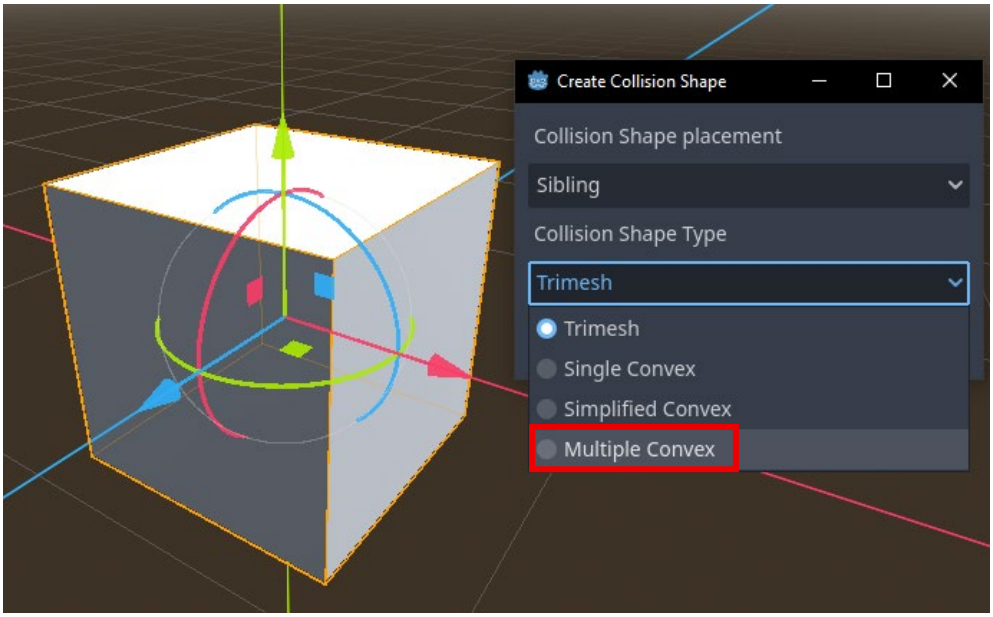
26 Notice that a Cube object appears. However, it is missing the blue **CollisionShape** that allows detection of collisions.

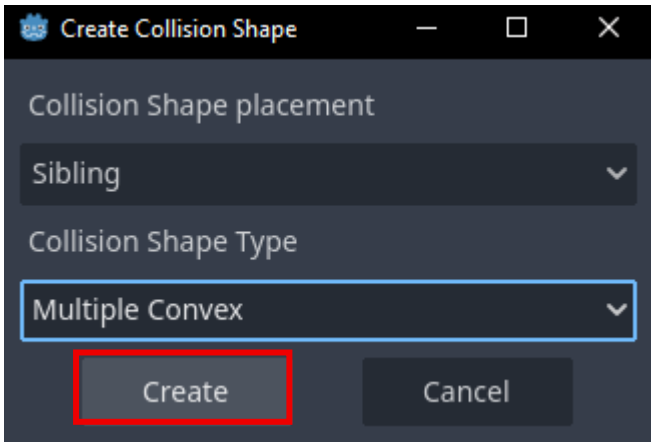


27 At the top of the game window click on **Mesh**. Select **Create Collision Shape...** from the drop-down menu.



28 Keep the Collision Shape placement set to Sibling. Underneath, set the **Collision Shape Type** to **Multiple Convex**. Click **Create**.



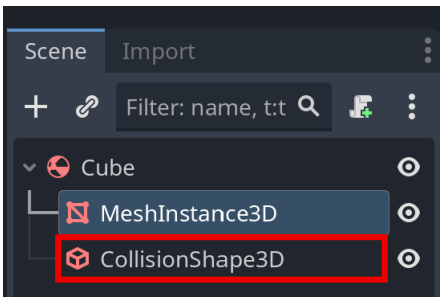


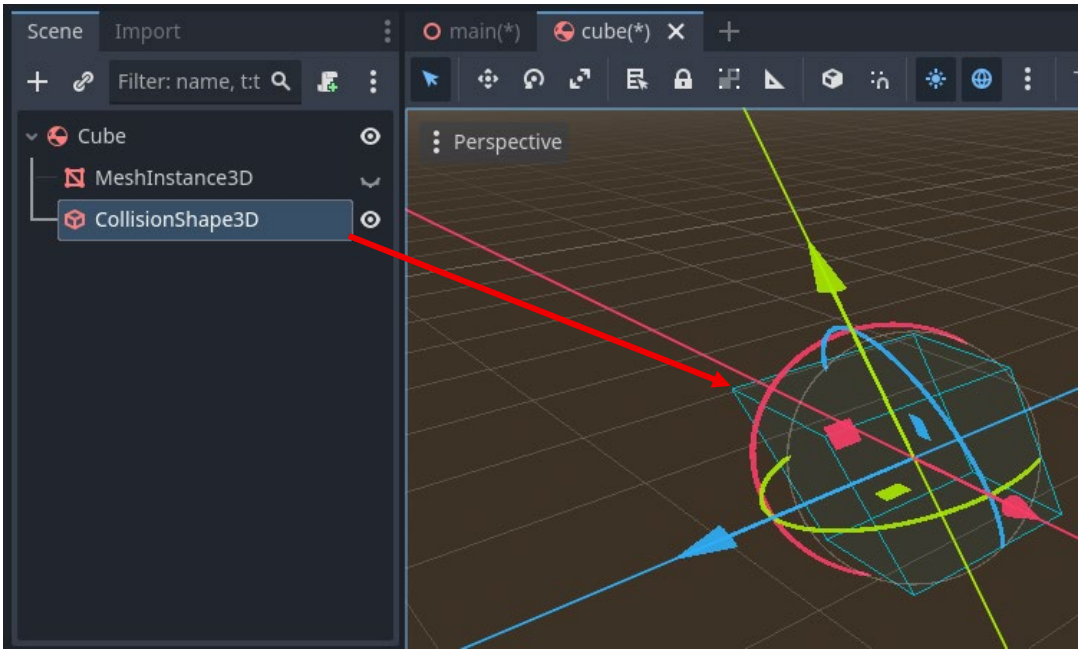
New Concept: Multiple Convex

Convex is a combination of the **Trimesh** and the **Single Convex** generation algorithms for **CollisionShape3Ds**. **Trimesh** offers the best accuracy with the worst performance, while **Single Convex** offers the best performance with the worst accuracy.

29 Notice that a **CollisionShape3D** gets generated in the Scene, which is based on the selected **Mesh** from **MeshInstance3D**.

To see the **CollisionShape** more clearly, try setting the **MeshInstance3D** to be **invisible**.





Pro Tip:

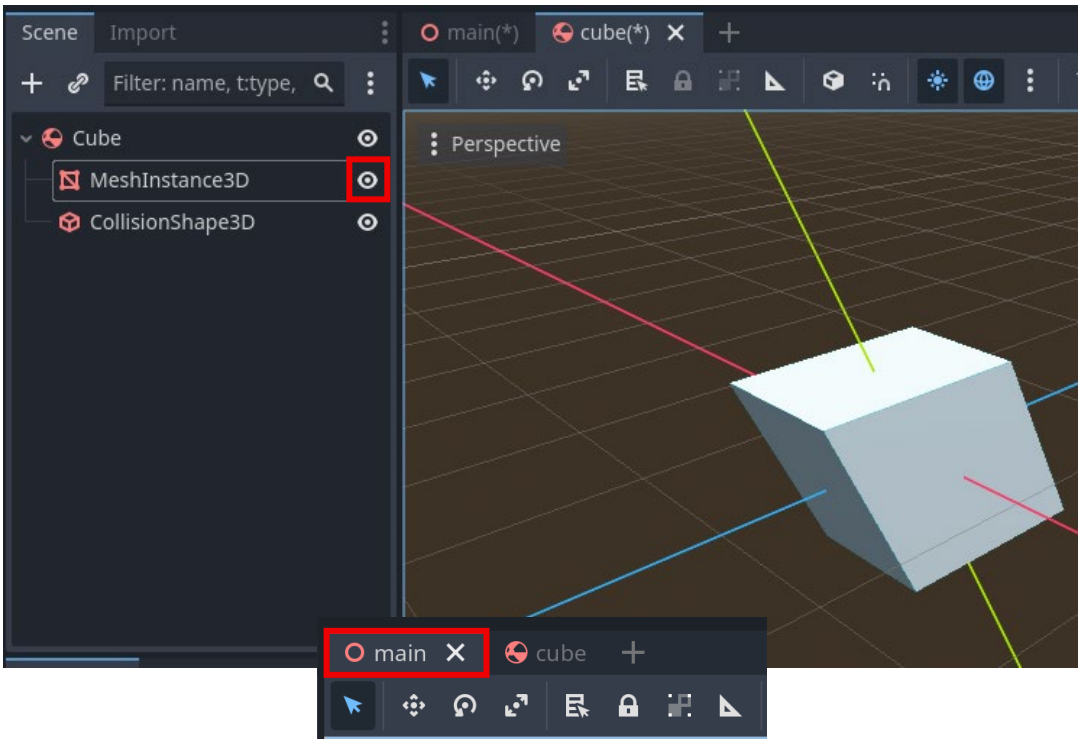
To toggle visibility, click the eye symbol next to a node in the **Scene** window.

Pause for **Sensei Stop #3**!

Check in with a Code Sensei before moving on and make sure the Cube Scene is set up properly.

Reminder: Save your work!

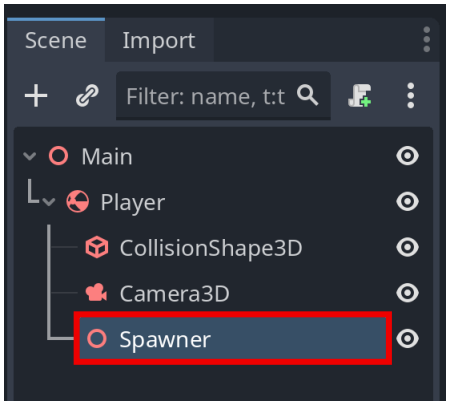
30 Reset the **MeshInstance3D** so that it is visible again, then navigate back to the **main** scene.



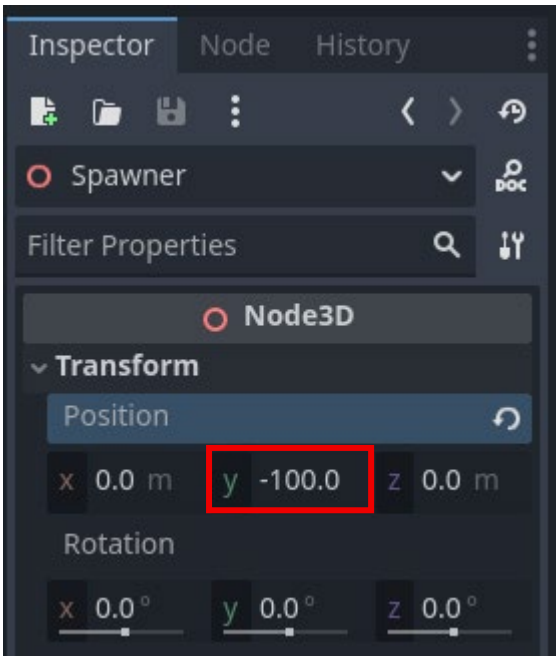
31 Time to start spawning the cubes!

To make sure that the cubes will always spawn below the Player, even when it is falling, the cubes need to be spawned relative to the Player's position. The Spawner node will be placed as a child to Player to achieve this.

Add a **Node3D** as a child to **Player** and rename it to **Spawner**.



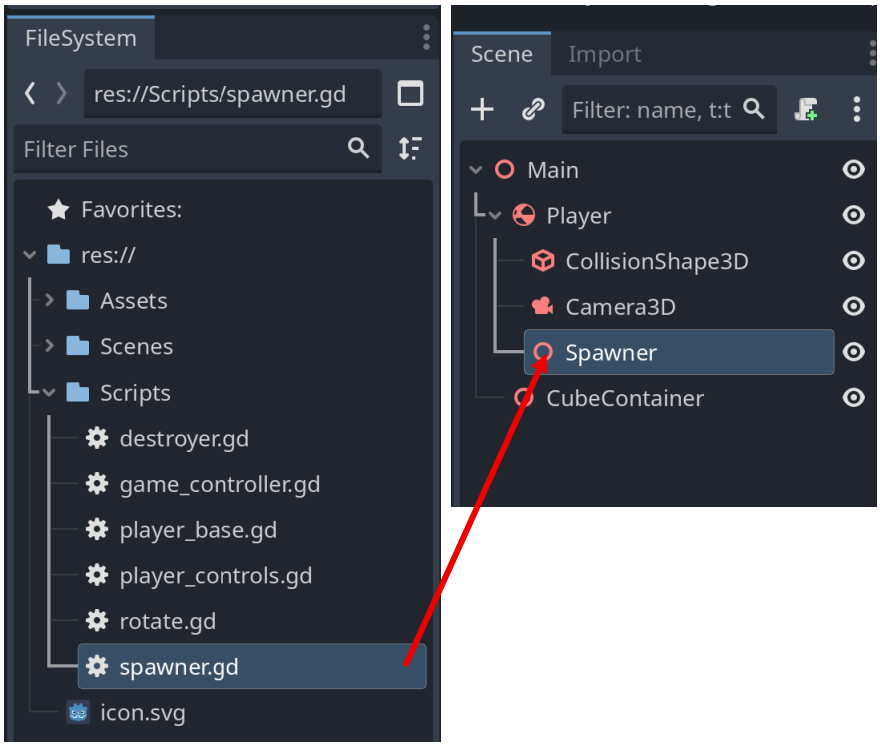
32 Set the Inspector for **Spawner,** set the **y-position** to **-100**. This will ensure the cubes spawn 100 units below the Player.



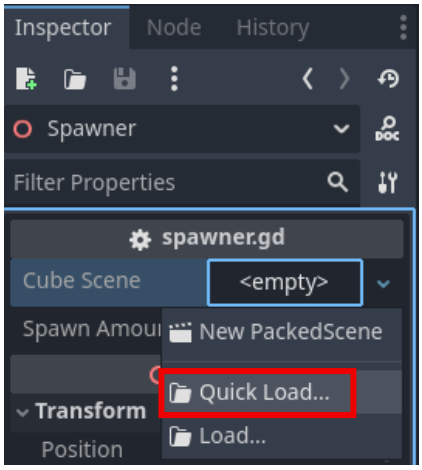
33 To keep things organized and prevent the cubes from moving with the Player's movements, the cubes will be placed outside of the Player's subtree.

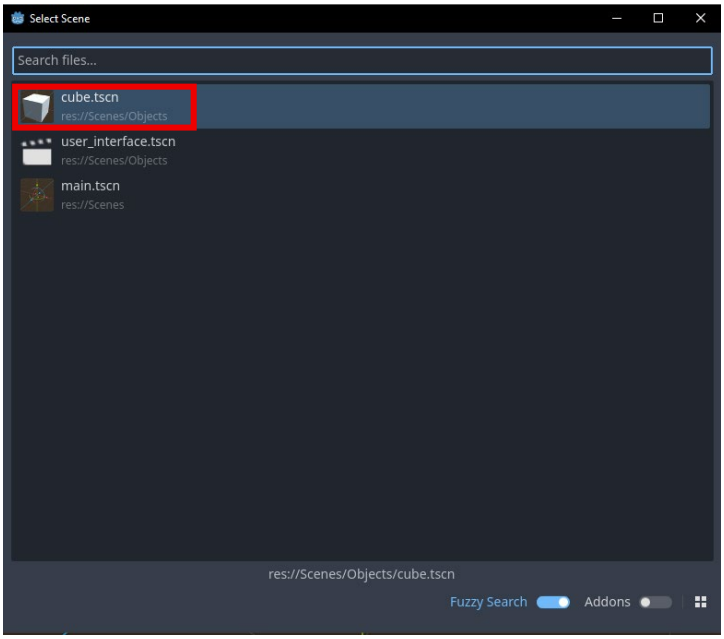
Add a new **Node3D** as a child to **Main** and rename it **CubeContainer**.

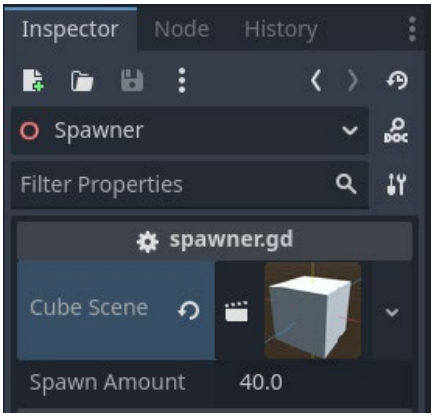
34 In **FileSystem**, navigate to the **Scripts** folder and drag the **spawner.gd** script onto the **Spawner** node.



35 In the **Inspector** for the **Spawner** node, find the **spawner.gd** section. Next to **Cube Scene**, select **Quick Load...** and select **cube.tscn**.







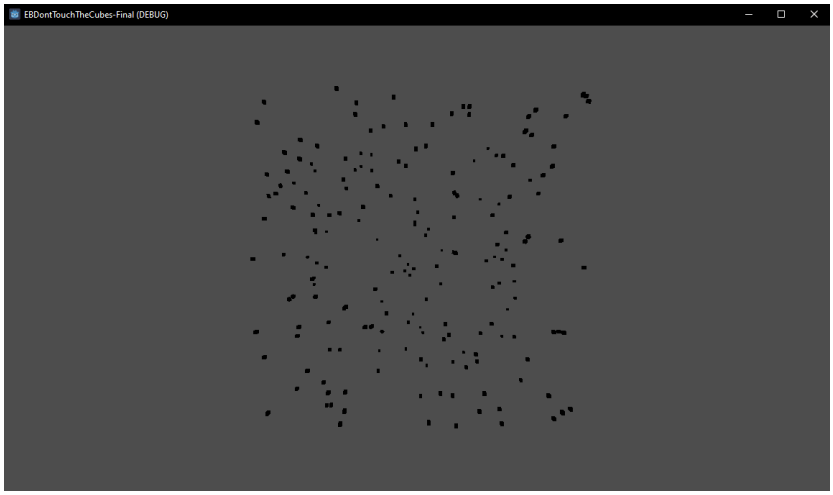
Pro Tip:

Don't worry if the preview doesn't look the same as in the image! The preview just shows a screenshot of the game window when the scene was last saved.

36 Playtest the game to see the cubes spawning *below* the player.

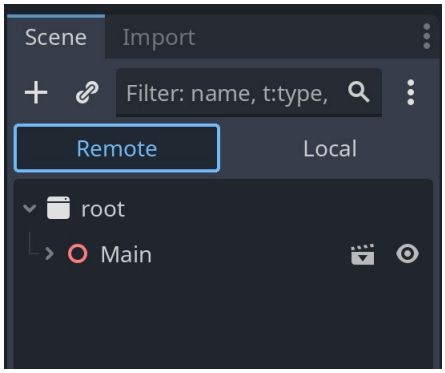
Great work! If the cubes aren't spawning in as expected, refer to the previous steps for additional guidance.

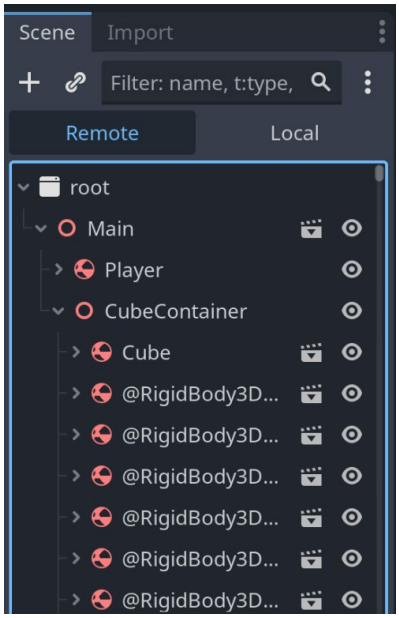
There seems to be a problem though. What might be wrong?



37 The player is supposed to fall past the cubes, but the cubes start falling away faster than the player can get past!

With the Playtest window still open, navigate to **Scene** and click on **Remote**. Expand the scene to notice all the cubes spawning inside of **CubeContainer** as expected.

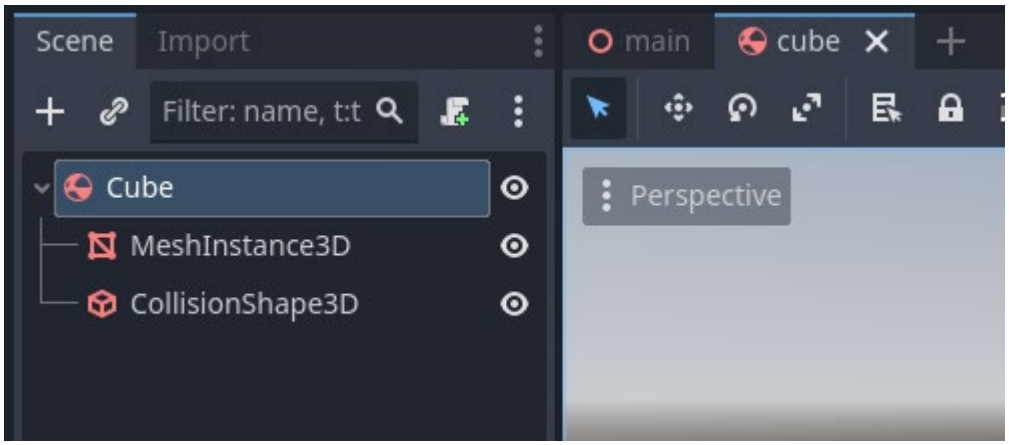




38 Close the Playtest window.

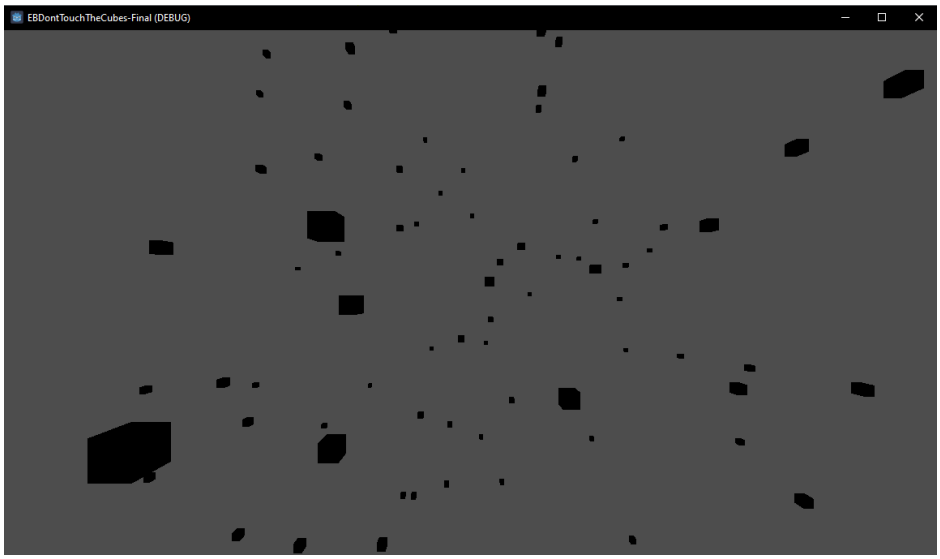
Navigate back to the **Cube scene** and select the **Cube node**.

In the **Inspector**, set its **Gravity Scale** to **0**. This keeps the cubes floating still in space so the player can fall past them.



39 Playtest the project again.

Notice how the cubes are floating still in space, and the Player can pass right by them!



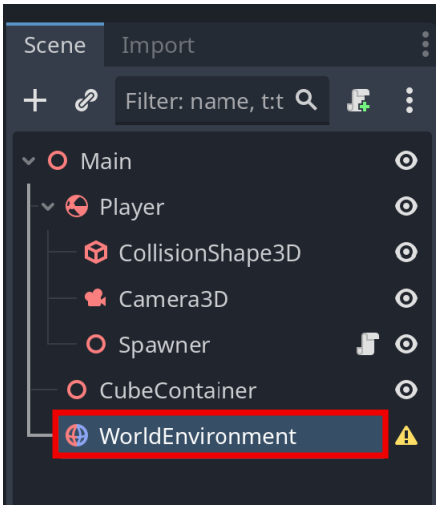
Pause for **Sensei Stop #4**!

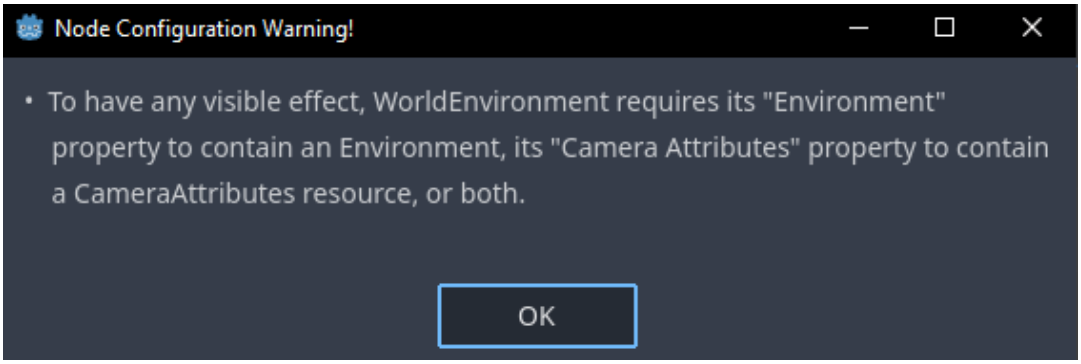
Check in with a Code Sensei before moving on and make sure that the cubes are spawning in and the Player is falling past them.

Reminder: Save your work!

40 Notice that the cubes appear black. This is because a **DirectionalLight** hasn't been added yet!

Return to the **Main** scene. Before adding a **DirectionalLight**, add a **WorldEnvironment** node as a child to the **main root**.



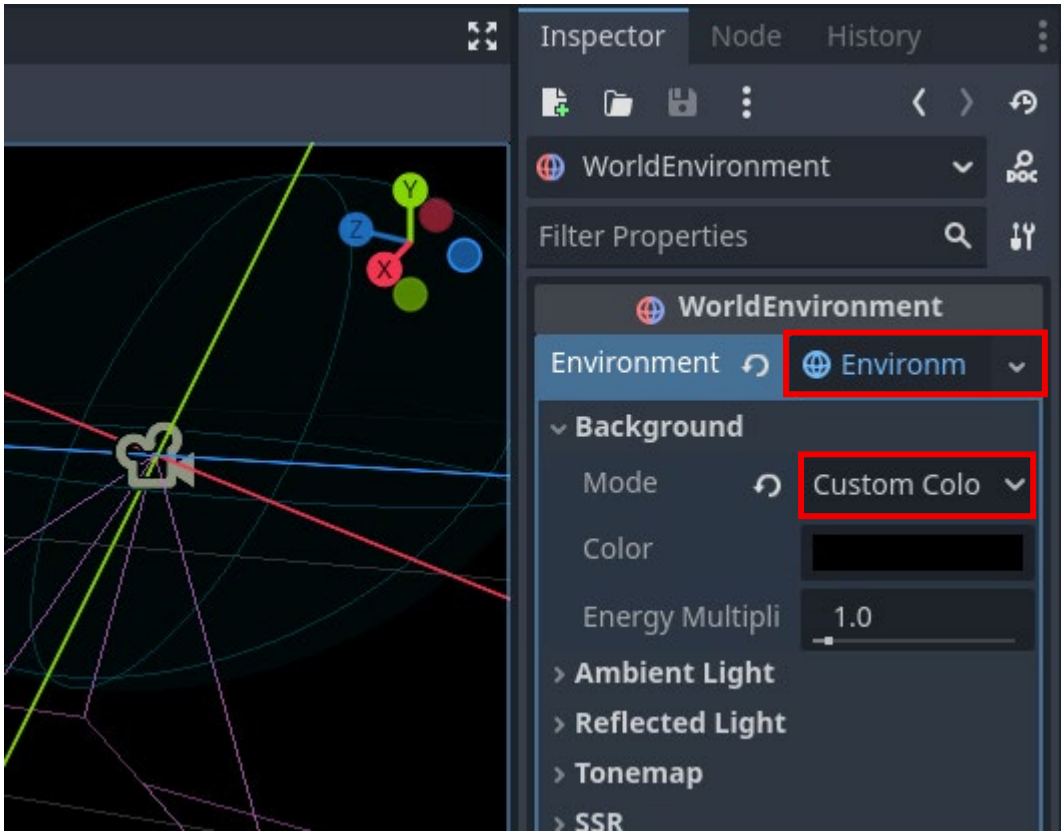


Reminder:

The **WorldEnvironment** determines how the world is rendered and can enable lots of cool effects like glow, fog, reflections, and more!

41 In the **Inspector** for **WorldEnvironment**, set Environment to **New Environment**. Select **Environment** and in **Background**, set the **Mode** to **Custom Color**.

Click on the **Color** property to tinker with the shade of the background environment!

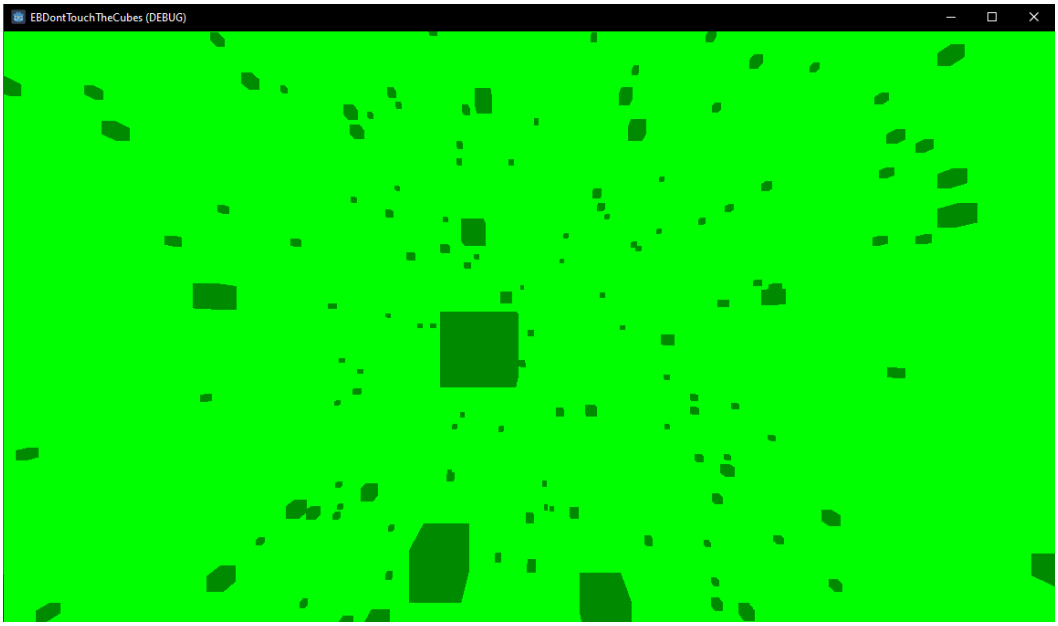


42 Playtest the project.

If the color was changed, the cubes should now be visible from the WorldEnvironment's ambient light (see the discussion on 3D Lighting Models at the start of this project).

Notice the cubes appear to be gone if black was left as the default color.

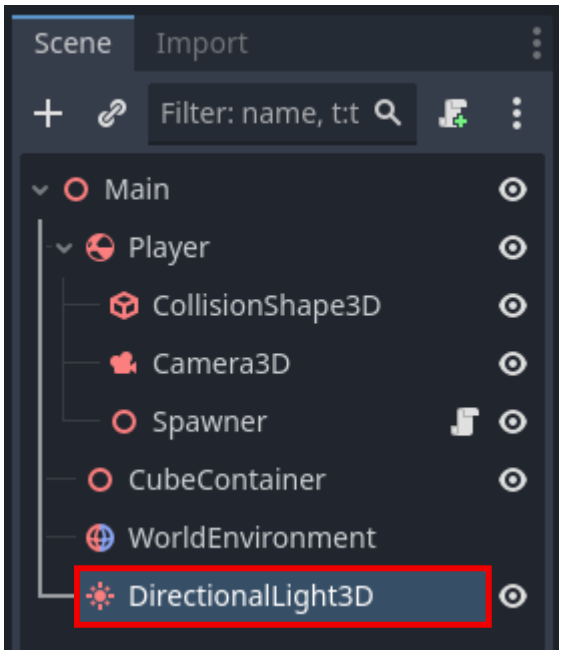
They really are there! They're still black because no light sources have been added yet.



43

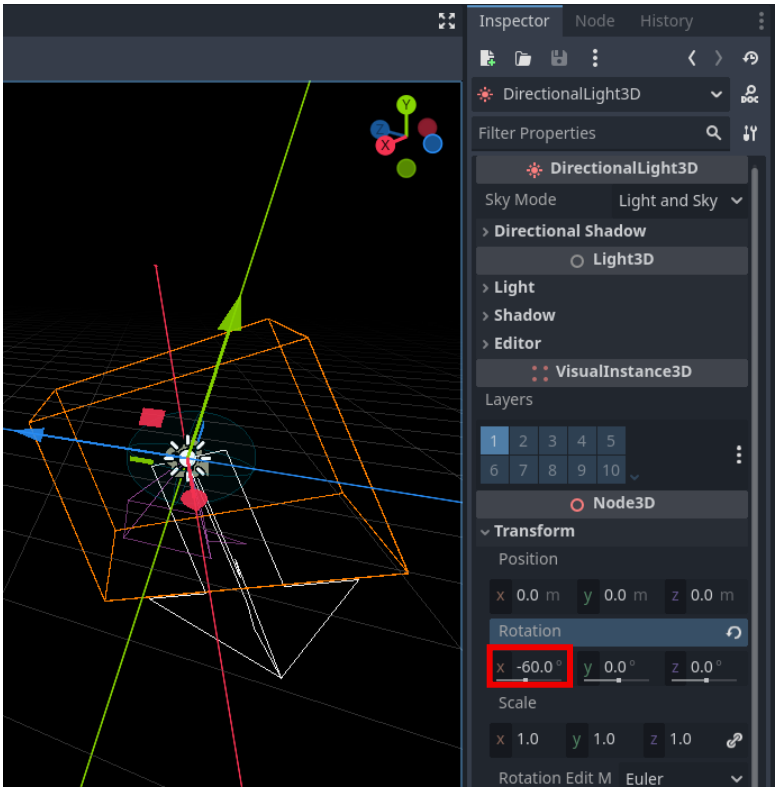
Add a light source to help see the cubes!

Add a new **DirectionalLight3D** node as a child to **Main**.



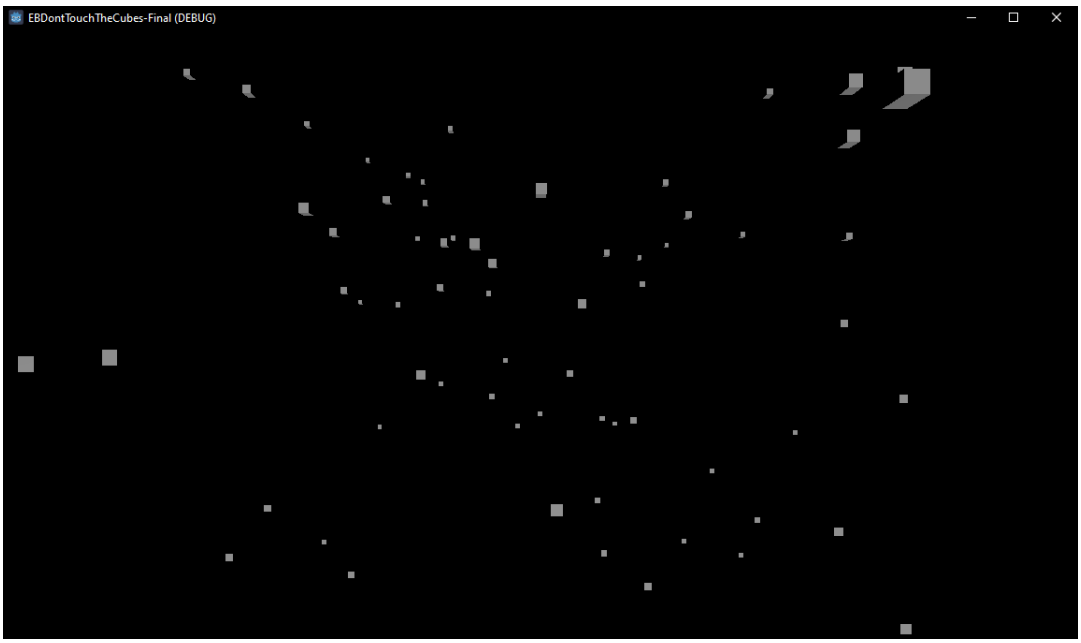
44

In the **Inspector** under the **Node3D** section, set the **DirectionalLight3D's x rotation** to **-60** degrees.



45 Playtest the game.

Tinker with the rotation values of **DirectionalLight3D**. How does it change the appearance of the cubes?



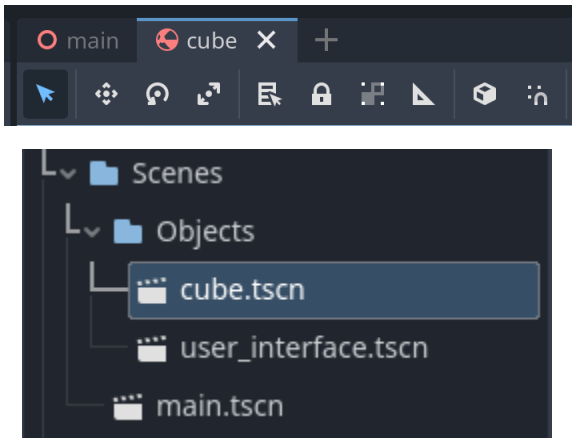
Pause for **Sensei Stop #5**!

Check in with a Code Sensei before moving on. Ensure that the background is no longer gray and the cubes are being lit up.

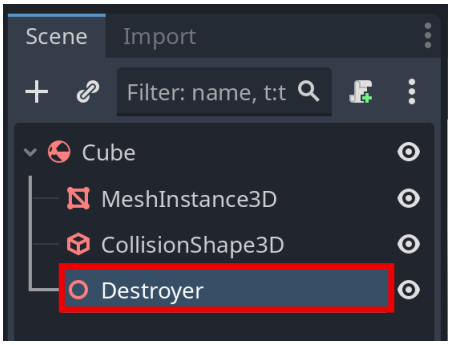
Reminder: Save your work!

46 If the game is left open for too long, the performance drops and it starts to lag and stutter. This Is because there are too many cubes spawning at once!

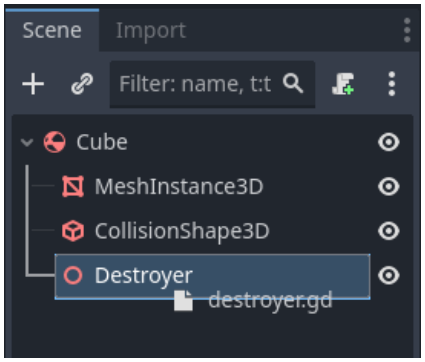
Navigate to the **cube** scene at the top of the game window or by double-clicking **cube.tscn** in **FileSystem**.

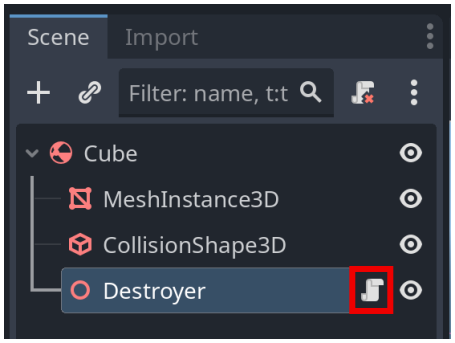


47 Add a **Node3D** as a child to **Cube** and rename it to **Destroyer**.



48 In **FileSystem**, drag the **destroyer.gd** script onto the **Destroyer** node. Click the script icon to open it, once attached.





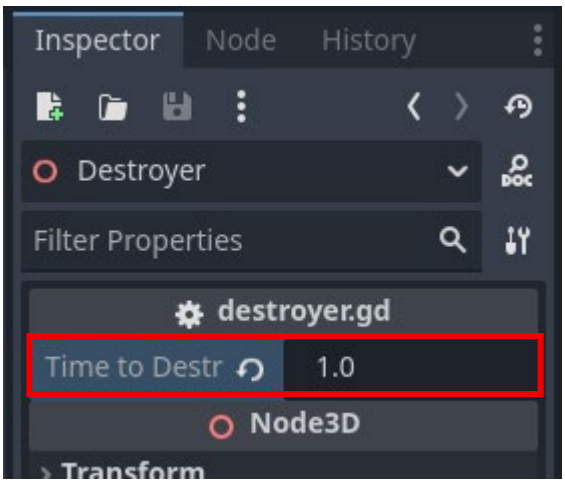
49 Review the code already in the script.

Recall how the Spikes in Meany Bird were spawned and destroyed by updating the value of Spawn Time and Time To Destroy in the Inspector. Review the code in the _ready() method, which shows how a timer is used here in a similar way.

```
1   extends Node3D
2
3   @export var time_to_destroy: float = 10
4
5   # Called when the node enters the scene tree for the first time.
6   func _ready() -> void:
7       var timer = Timer.new()
8       timer.wait_time = time_to_destroy
9       timer.autostart = true
10      timer.one_shot = true
11      timer.timeout.connect(destroySelf)
12      add_child(timer)
13
14  func destroySelf():
15      get_parent().queue_free()
16
```

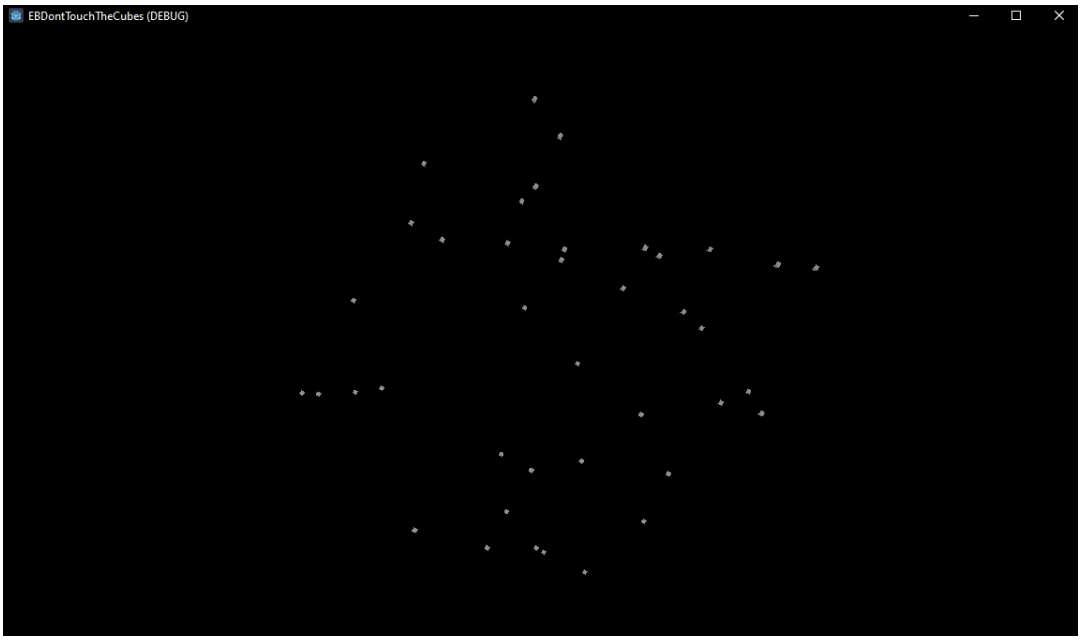
50 What happens if the Time to Destroy is too short?

In the **Inspector**, set the **Time to Destroy** export variable to just 1.0 seconds.



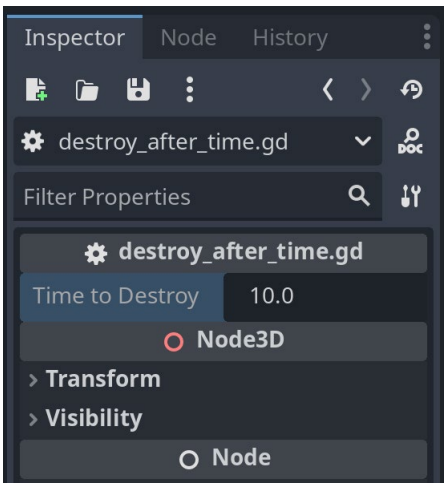
51 Playtest the project.

Watch the cubes get destroyed way before they pass the camera.



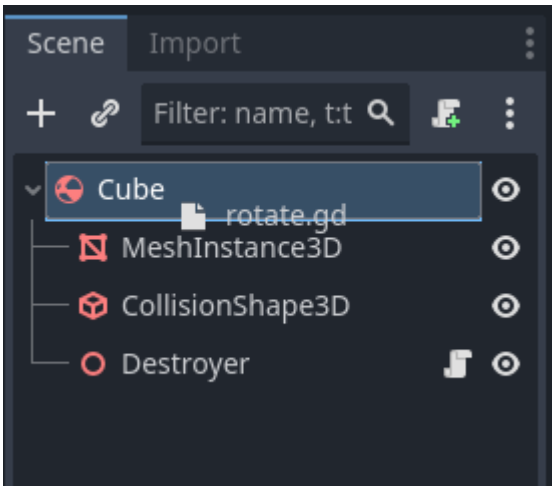
52 Wow, that was too short!

In the **Inspector**, reset the **Time to Destroy** variable back to **10** seconds.

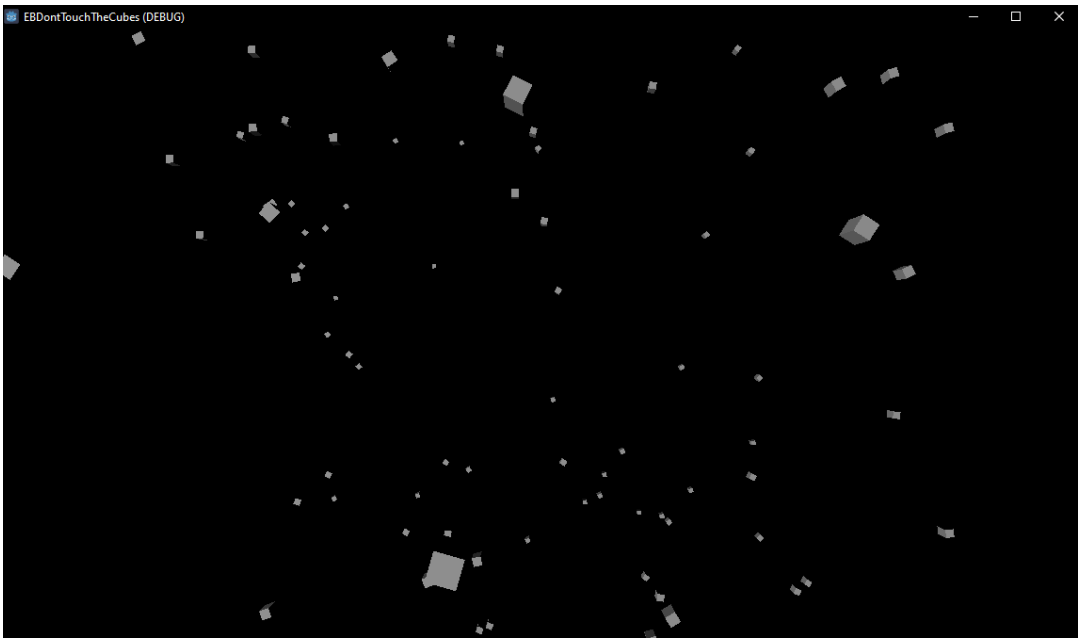


53 To improve the user experience, make the cubes rotate.

From **FileSystem**, drag the **rotate.gd** script onto the Cube node.



Playtest the game. Notice how the cubes rotate as they move towards the camera! Feel free to review the code in **rotate.gd**; it's short, sweet, and simple.

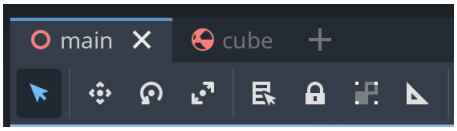


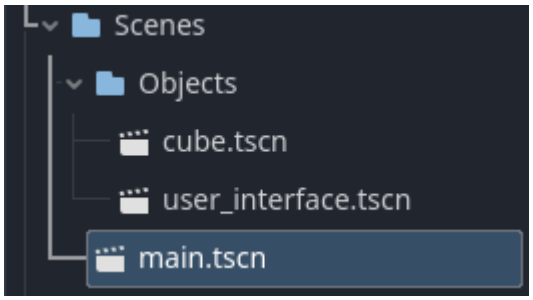
Pause for **Sensei Stop #6**!

Check in with a Code Sensei before moving on. Make sure that the **destroyer.gd** and **rotate.gd** scripts are attached to the correct nodes.

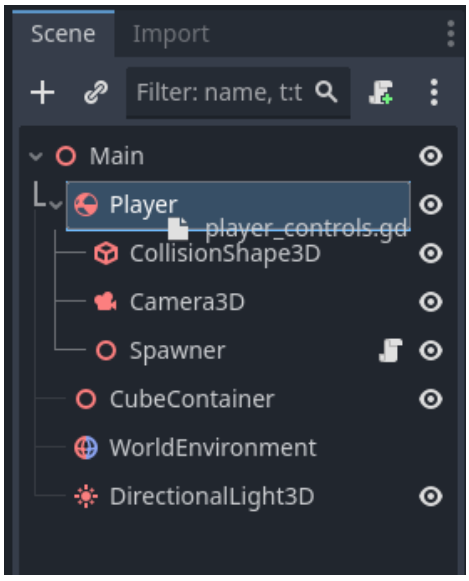
Reminder: Save your work!

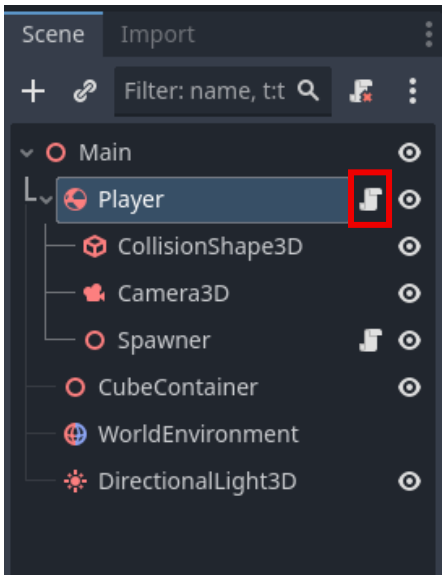
54 Return to the main scene.





55 From **FileSystem**, attach the **player_controls.gd** script onto the **Player** node. Click on the script icon to open it.





56

Time to code the player's movement!

Three things must happen for the implementation to be complete:

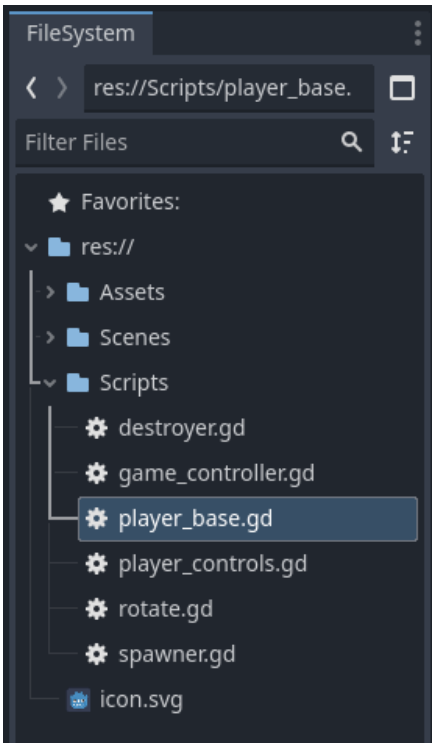
1. Determine the direction to move based on the user's input
2. Normalize the direction so it has a length of 1
3. Update the Player's position based on the user's input

```
1   extends "res://Scripts/player_base.gd"
2
3   func handle_movement(delta: float) -> void:
4       # --------------------
5       # TODO STEP 56
6       # Write code for handling player input
7       # --------------------
8       pass
9
10      # --------------------
11      # TODO STEP 71
12      # Normalize input direction so that it's a unit vector
13      # --------------------
14
15      # --------------------
16      # TODO STEP 73
17      # Set the position of our player depending on the input
18      # --------------------
19
```

57 Before getting into the code, it's important to emphasize how this script is an extension of another script.

Notice the first line of code says that it extends the **player_base.gd** script. That allows all the code from **player_base.gd** to act as if it was pasted at the start of **player_controls.gd**, but it's really stored in the separate file **player_base.gd**.

```
1  extends "res://Scripts/player_base.gd"
2
3  func handle_movement(delta: float) -> void:
4      # --------------------
```



58 Inside of the **handle_movement()** method, delete **pass**. It only served the purpose of having *something* inside of the method so that Godot doesn't throw an error.

On the same line, declare an **input_dir** variable and set it to **Vector3.ZERO**. This is a constant value that is already made for developers to use; it is simply a **Vector3** containing the numbers **(0, 0, 0)**.

Recall that a **Vector2** was used in Meany Bird to give the bird velocity and push it up when the right mouse button was clicked.

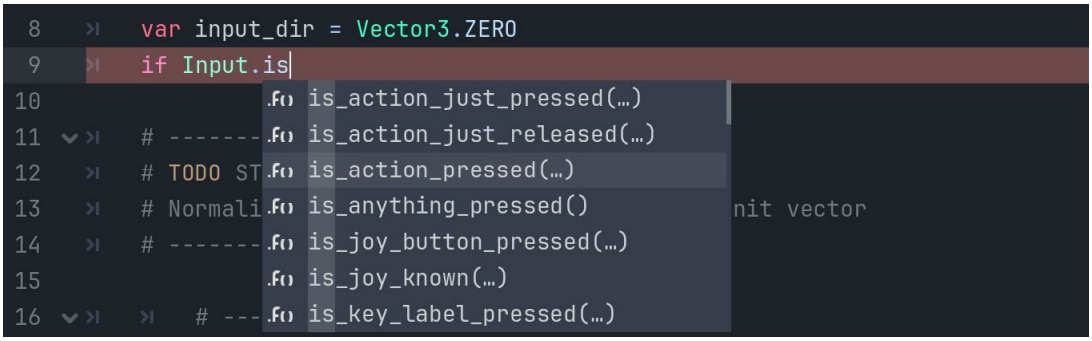
```
3  func handle_movement(delta: float) -> void:
4      # --------------------
5      # TODO STEP 56
6      # Write code for handling player input
7      # --------------------
8      var input_dir = Vector3.ZERO
9
```

New Concept: Vector3

A Vector3 can store three values which are typically in the order of x, y, z. They are used for all sorts of 3D space manipulation. Think back to IMPACT, where tiles have "locations" which are a combination of row and column values. Vector3s are combinations of 3 values instead!

59 On the next line, start typing **if Input.is** and the code completion will appear. Use the direction buttons to navigate to **is_action_pressed(...)** and press **TAB** to select it.

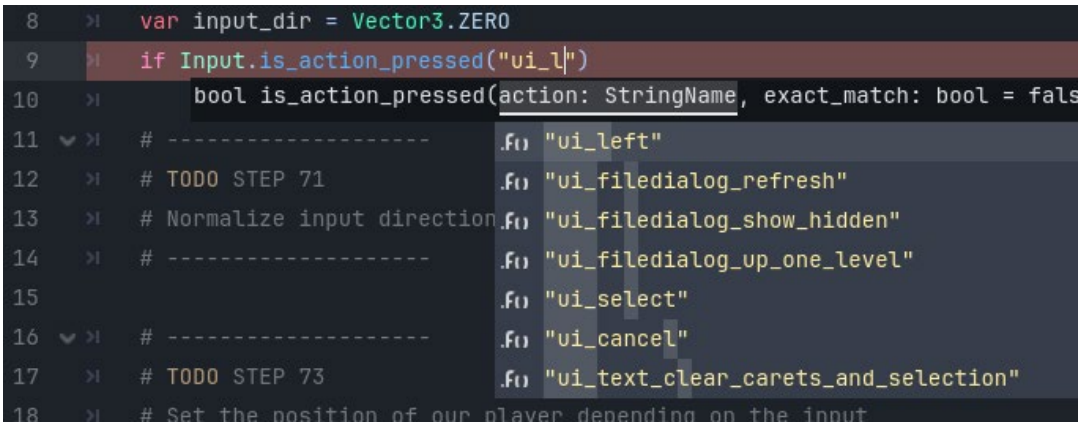
The **is_action_pressed()** method checks if a button is currently being held.



Reminder:

Make sure that the "**I**" in "**Input**" is capitalized! Otherwise, the code completion will not recognize "**input**" correctly and won't show the correct options.

60 Using the code completion, add **"ui_left"** inside the parentheses. Make sure quotation marks surround the text, and all of it is placed inside the parentheses.



61 Add a **colon :** to the end of the **if** statement.

On the next indented line, type **input_dir.x -= 1**. This will set the **input_dir Vector3** to **(-1, 0, 0)**.

```
8   var input_dir = Vector3.ZERO
9   if Input.is_action_pressed("ui_left"):
10      input_dir.x -= 1
11
```

62 Repeat the previous two steps. Use the code completion to add code that checks if **ui_right** is pressed, then change the **input_dir** with **+=**.

```
8   var input_dir = Vector3.ZERO
9   if Input.is_action_pressed("ui_left"):
10      input_dir.x -= 1
11  if
12
```

63

Check the code! Update the script as needed.

```
var input_dir = Vector3.ZERO
if Input.is_action_pressed("ui_left"):
    input_dir.x -= 1
if Input.is_action_pressed("ui_right"):
    input_dir.x += 1

```

64

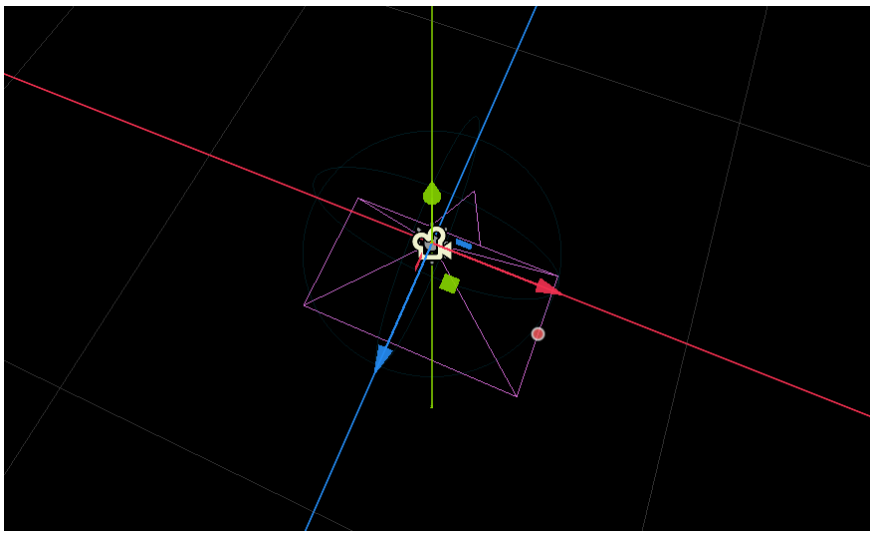
Add code that checks if **ui_up** is pressed, but don't update the **input_dir** value yet. Since left/right is on a different axis than up/down, consider what value from **input_dir** will be changed now (**X, Y, or Z**).

```
    input_dir.x += 1
if Input.is_action_pressed("ui_up"):
    

# --------------------
# TODO STEP 71
# Normalize input direction so that it's a unit vector
# --------------------
```

In the below image, the 3D scene and the coordinate axes are shown. Red, Green, and Blue are X, Y, and Z respectively.

If the player should move up relative to where the camera is looking, which coordinate value will be changed and how will it be changed? Without looking at the next step, try updating **input_dir**.



65 Check the code! Notice that the **Z** coordinate is being set to a negative value because that aligns with the camera's positioning and goes in the opposite direction of the blue arrow.

```
12        input_dir.x += 1
13    if Input.is_action_pressed("ui_up"):
14        input_dir.z -= 1
15
```

66 Without looking at the next step, try updating **input_dir** for when **ui_down** is pressed.

```
13    if Input.is_action_pressed("ui_up"):
14        input_dir.z -= 1
15    if
16
```

67 Check the code! Update the script, as needed.

```
13    if Input.is_action_pressed("ui_up"):
14        input_dir.z -= 1
15    if Input.is_action_pressed("ui_down"):
16        input_dir.z += 1
```

68 Make sure your code matches the code in this image.

Note: An error may appear because the **delta** parameter is not used from the **handle_movement()** function. This can be ignored for now, delta will be used later in the code.

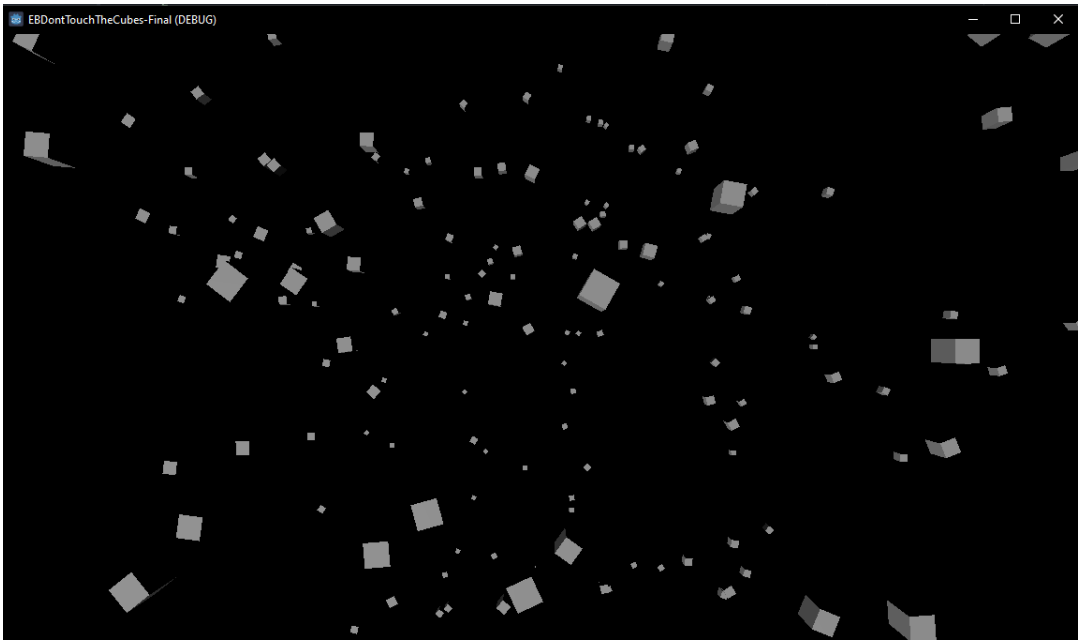
```
1   extends "res://Scripts/player_base.gd"
2
3   func handle_movement(delta: float) -> void:
4       # --------------------
5       # TODO STEP 56
6       # Write code for handling player input
7       # --------------------
8       var input_dir = Vector3.ZERO
9       if Input.is_action_pressed("ui_left"):
10          input_dir.x -= 1
11      if Input.is_action_pressed("ui_right"):
12          input_dir.x += 1
13      if Input.is_action_pressed("ui_up"):
14          input_dir.z -= 1
15      if Input.is_action_pressed("ui_down"):
16          input_dir.z += 1
17
```

69 Playtest the project.

Notice that the whole screen seems to rotate now, which is because there is code inside of **player_base.gd** that rotates the Player node (and therefore, the camera with it).

Try moving with the direction buttons.

Uh oh! It doesn't seem to work yet. That's because the code hasn't used the **input_dir** variable for anything yet!

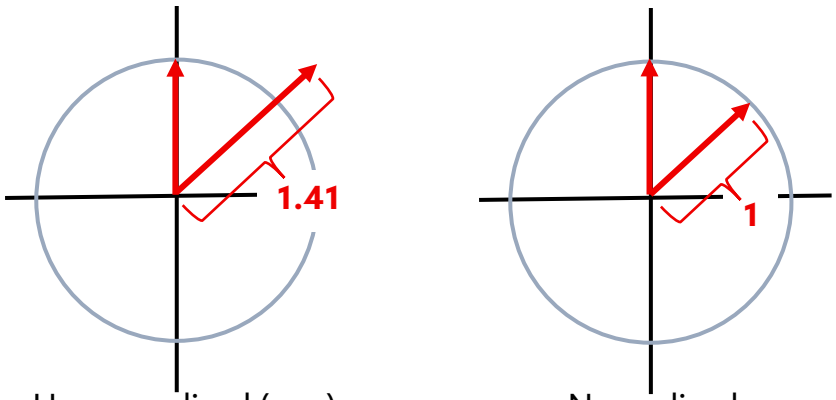


70 The input direction needs to be normalized.

This is because when two buttons are pressed at the same time, **input_dir**'s magnitude would be too large and the player would move unnaturally fast.

When one direction button is pressed, input_dir's length as a vector would equal 1. However, if two direction buttons are pressed, input_dir's length as a vector would equal 1.41.

So, the player would move ~41% faster when pressing two buttons rather than just one!



71 To get a normalized vector in Godot, use **normalized().** This method returns the normalized scaling of a Vector so the length is 1.

> `normalized()`: Part of the Vector2/3 classes, this method returns the normalized scaling of the given vector so the length is 1.
>
> **Parameters: None**
>
> **Returns *(Vector):*** normalized scaling of the target of length 1.

72

Close the playtest window.

Navigate back to the **player_controls.gd** script and find the TODO for Step 71.

Set **input_dir** to the normalized version of itself by calling **Vector3's normalized()** function.

```
18  # --------------------
19  # TODO STEP 71
20  # Normalize input direction so that it's a unit vector
21  # --------------------
22  input_dir = input_dir.normalized()
23
```

73

Since the player always rotates, the current code isn't enough.

global_basis is a property for every Node that stores its rotation and scale information relative to world space.

The player's movement must be based on its rotation so that pressing up will always make the player move up **relative to where the camera is pointing.**

To achieve this, multiply **input_dir** by the node's **global_basis** property.

```
18  # --------------------
19  # TODO STEP 71
20  # Normalize input direction so that it's a unit vector
21  # --------------------
22  input_dir = input_dir.normalized()
23  input_dir = global_basis * input_dir
```

74

Now it's time to update the position of the player!

global_position is a property for every Node that stores its position information relative to world space.

Navigate to the **TODO** for Step 73.

Add code to declare a **target_position** variable. Set the variable to **global_position** plus three values multiplied in this order: **input_dir**, **move_speed**, and **delta**.

```
25  # --------------------
26  # TODO STEP 73
27  # Set the position of our player depending on the input
28  # --------------------
29
```

75

Check the code! Update the script, as needed.

```
25  # --------------------
26  # TODO STEP 73
27  # Set the position of our player depending on the input
28  # --------------------
29  var target_position = global_position + input_dir * move_speed * delta
```

Pro Tip:

Godot's order of operations is very similar to the one taught in school. The order is:

1. Parentheses
2. Negation
3. Times, divide, and remainder (left-to-right)
4. Add, subtract (left-to-right)

Parentheses are often used for clarity but aren't required for many math operations.

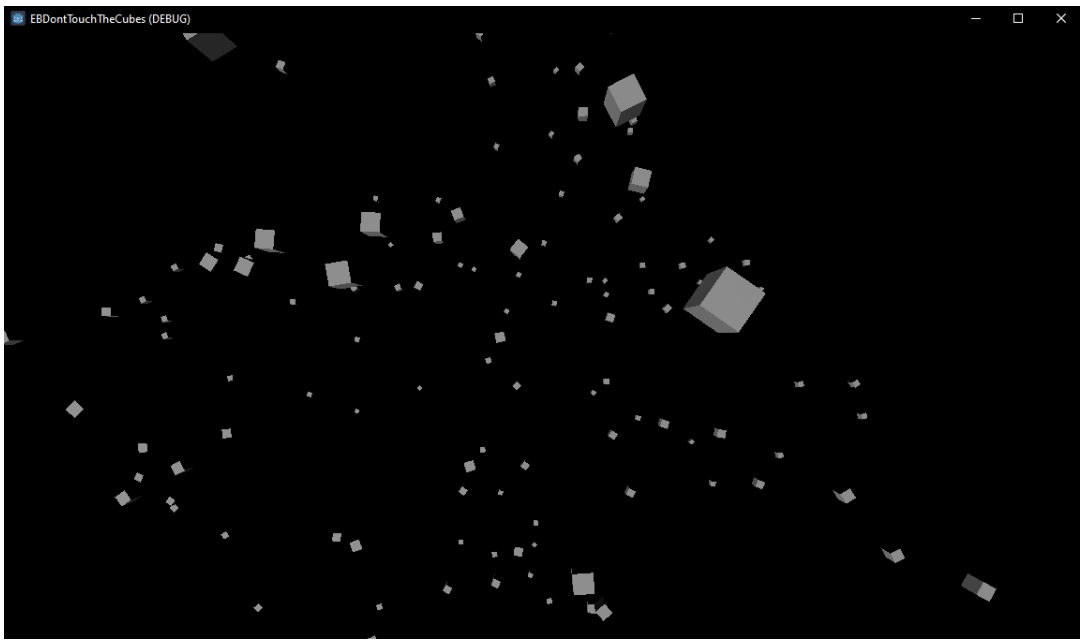
76 Underneath, set the **global_position** to be **target_position**.

```
25  # --------------------
26  # TODO STEP 73
27  # Set the position of our player depending on the input
28  # --------------------
29  var target_position = global_position + input_dir * move_speed * delta
30  global_position = target_position
```

Note: When Godot sees **global_position =**, it knows to set the global position. Whenever it sees **global_position** otherwise it knows to get the current value of global position. This works for all properties of a node.

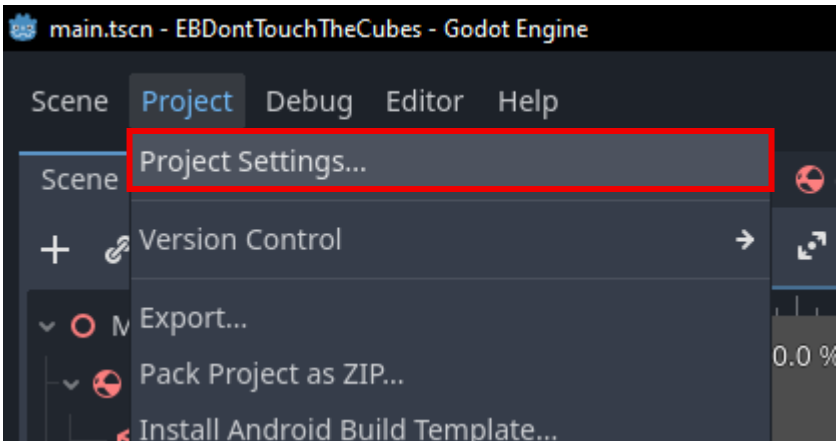
77 Playtest the project.

Try pressing the direction buttons to control the player's movement!

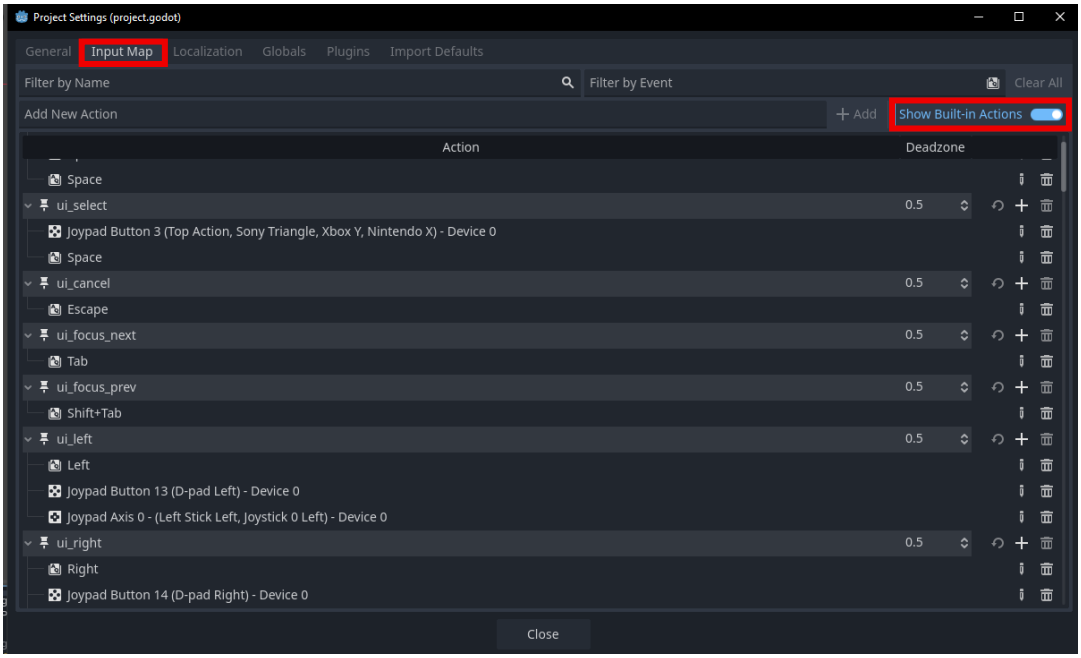


78 Add WASD keys as alternatives to the direction buttons!

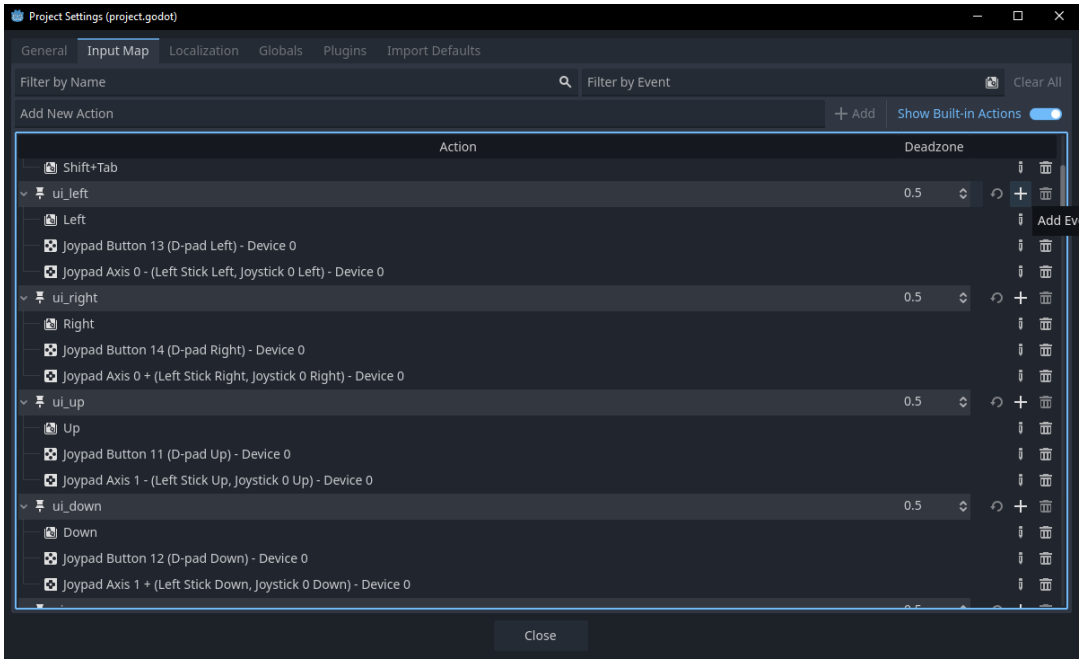
In the top left corner of the editor, open the **Project Settings** window.



79 At the top, navigate to **Input Map** and toggle the **Show Built-in Actions** slider **on**.

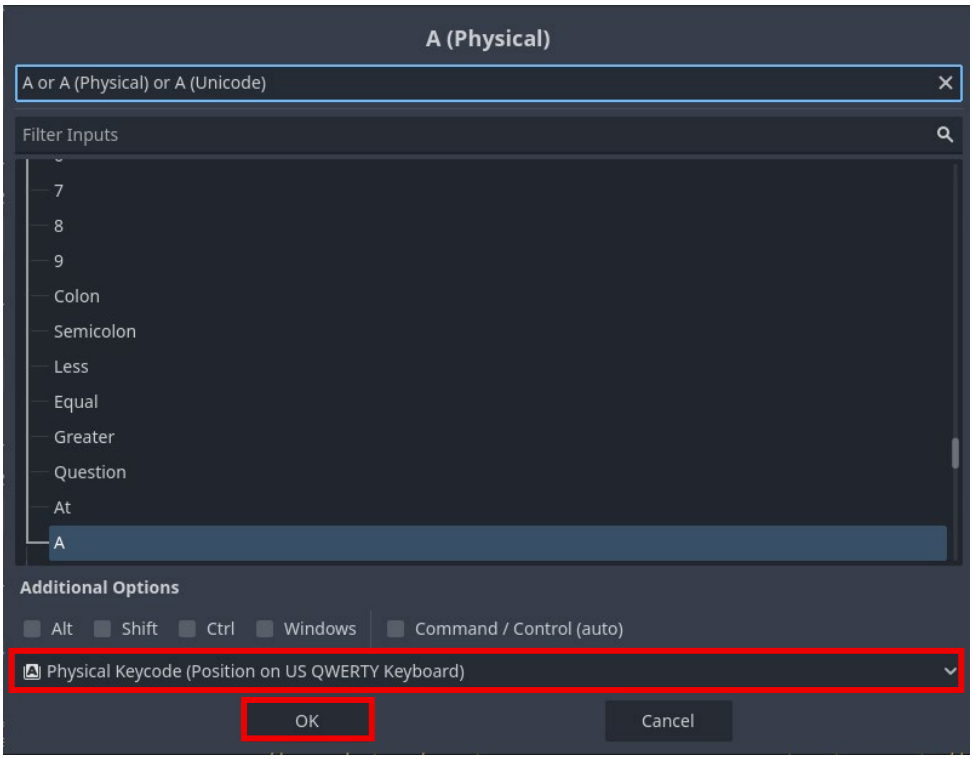


80 Scroll down to find **ui_left**, **ui_right**, **ui_up**, and **ui_down**. Click **+** to the right of **ui_left**.



81 Press **A** to record it as the keystroke. Click **Ok**.

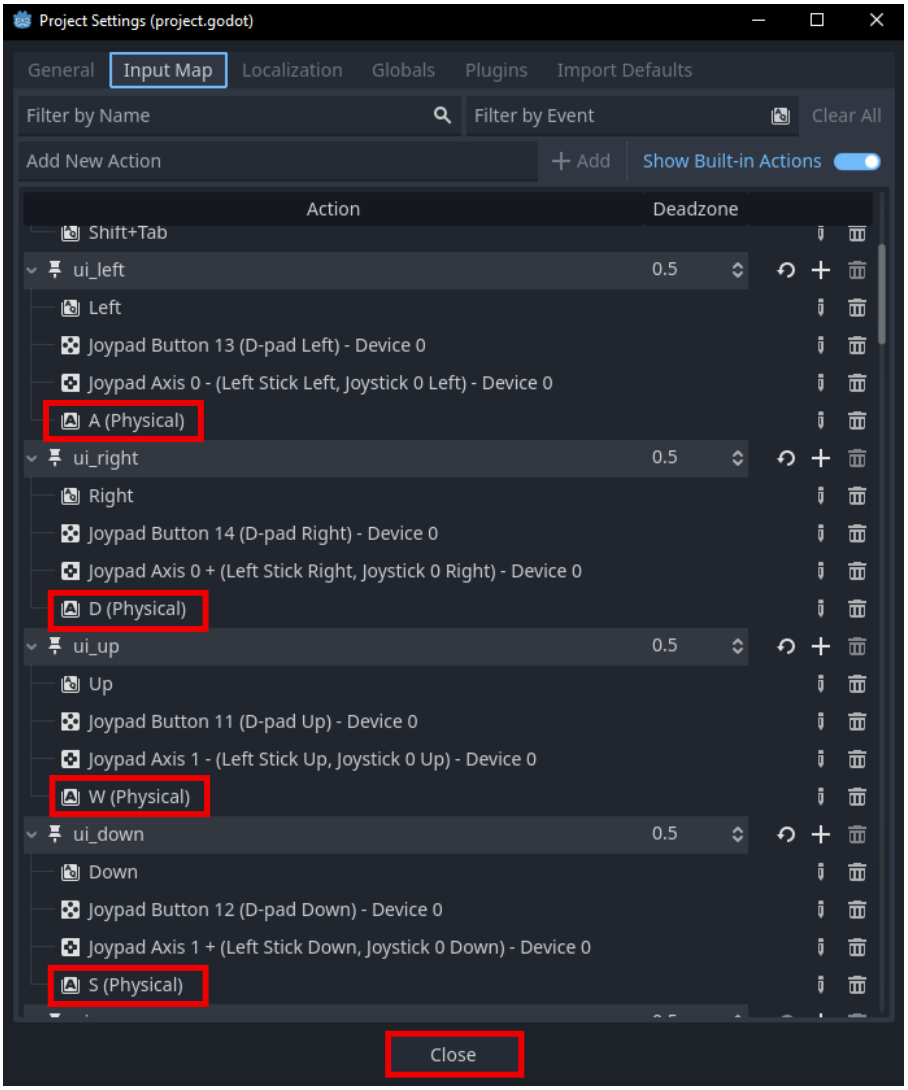
The box at the bottom of the window will show which **Physical Keycode** has been recorded.



Reminder:

Do not press any other buttons as it will record all other buttons pressed as part of the button combo!

82 Do the same for the other directional ui inputs: **D** for **ui_right, W** for **ui_up**, and **S** for **ui_down**. Then, close the **Project Settings** window.

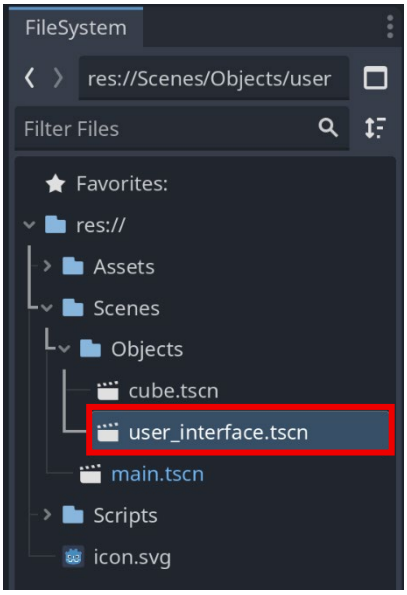


Pause for **Sensei Stop #7**!

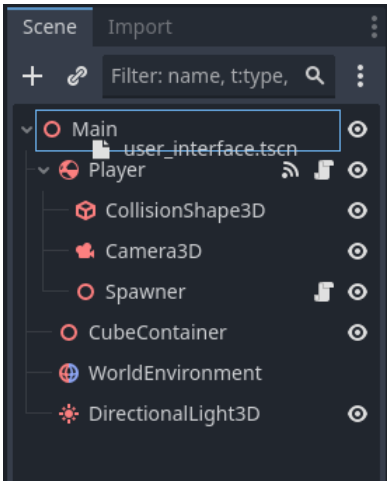
Check in with a Code Sensei before moving on. Make sure that the player controls are working properly and the WASD keys are connected.

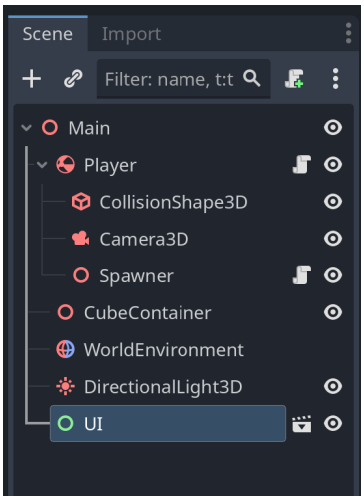
Reminder: Save your work!

83 In **FileSystem**, navigate to **Scenes** > **Objects** > **user_interface.tscn**.

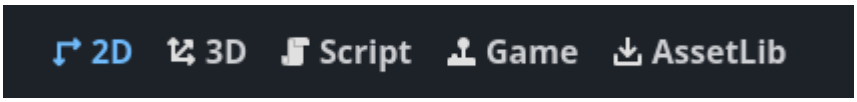


84 Drag **user_interface.tscn** onto **Main** to add the subtree as seen in the image.

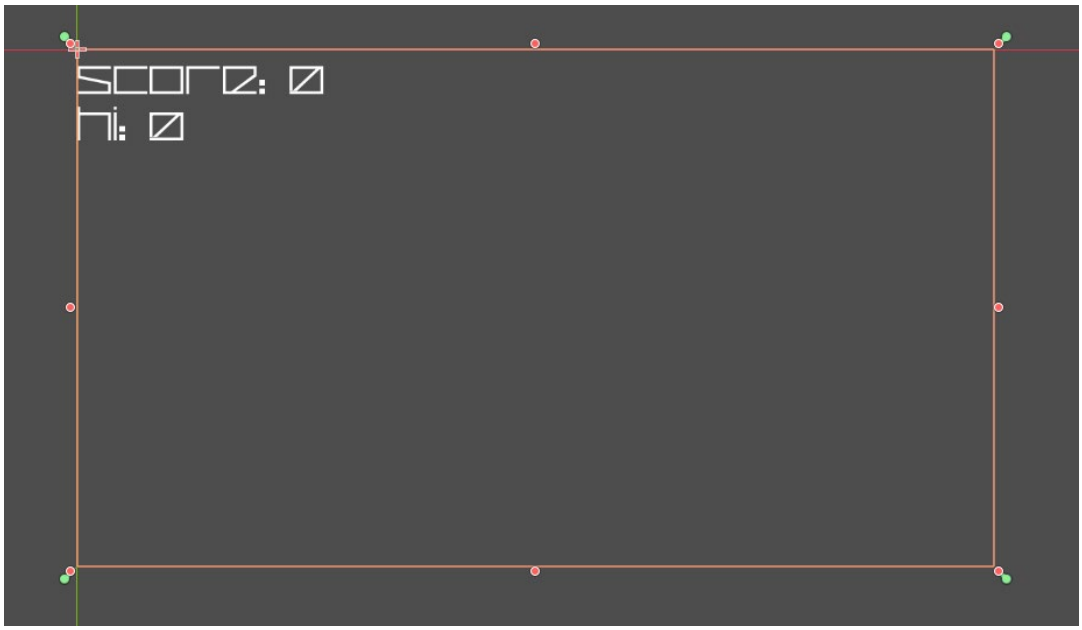




85 Switch from the **Script** workspace to the **2D** workspace.

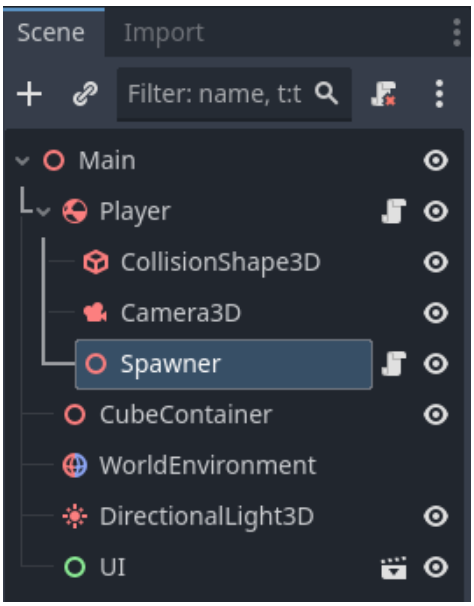


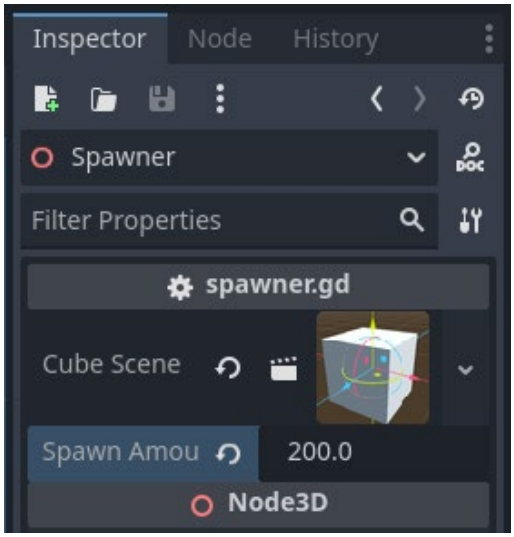
Explore what's inside the UI scene. How does it work?



86 Test what happens when the player collides with a cube!

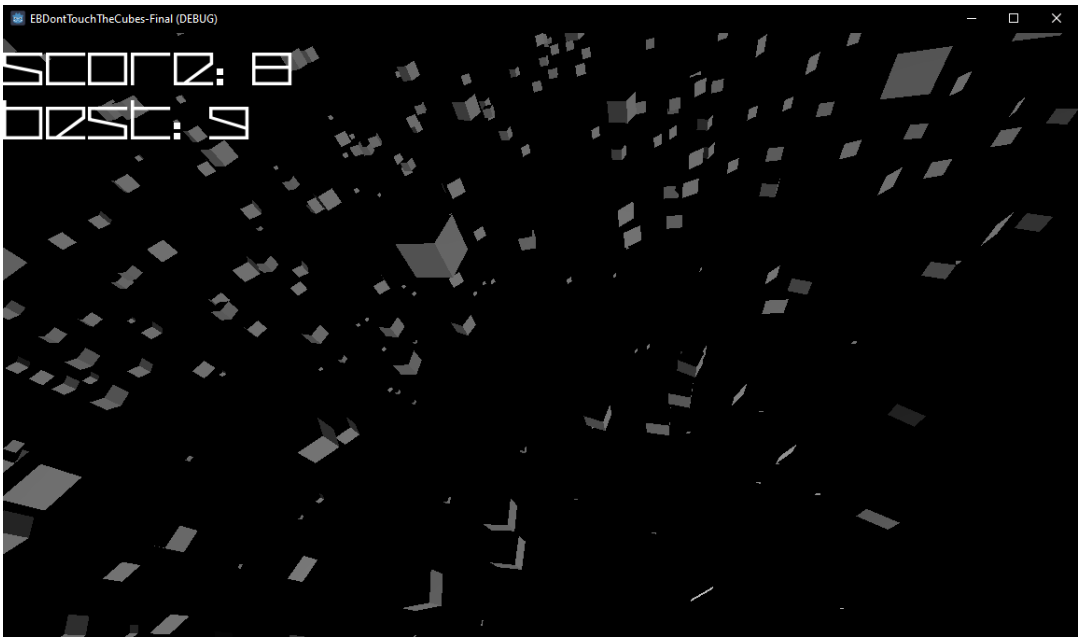
Navigate back to the **main scene**, and select the **Spawner** node. In the **Inspector**, increase the spawn amount to **100-200**, so that even more cubes spawn in.





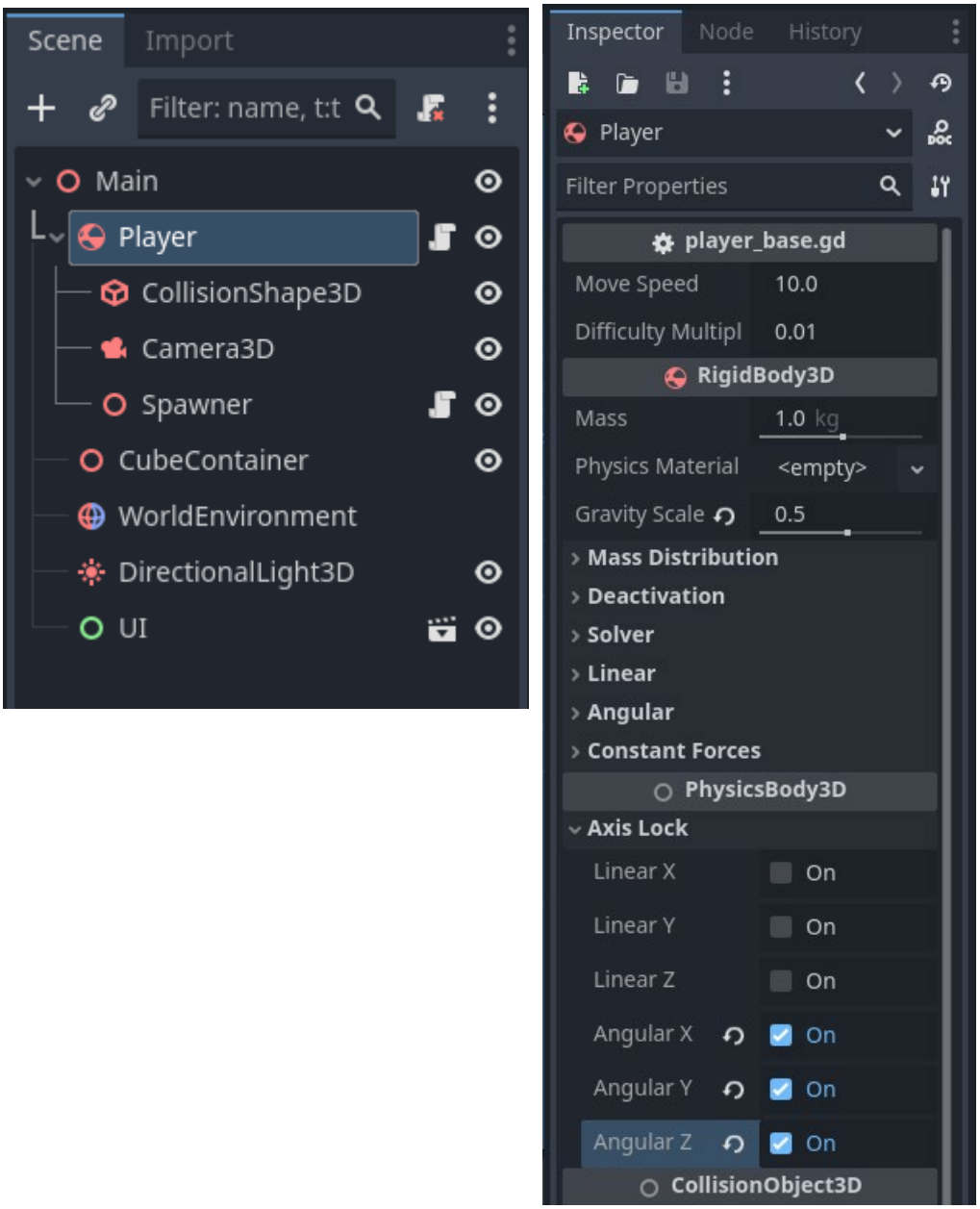
87 Playtest the project and try to get hit by a cube!

Uh oh! The player gets flung and spins around! Also, the game doesn't end when the player collides with the cube, since there isn't a collide signal on the player yet!



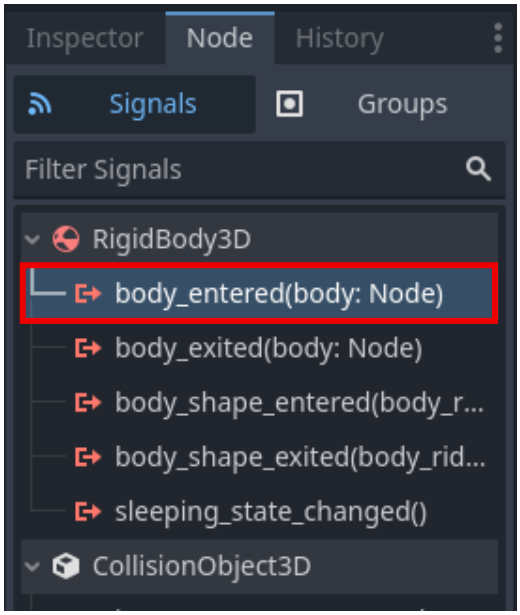
88 First, fix the player so that it doesn't spin out.

In the **Inspector** for the **Player** node, open the **Axis Lock** drop-down and select the **Angular X**, **Y**, and **Z** options.



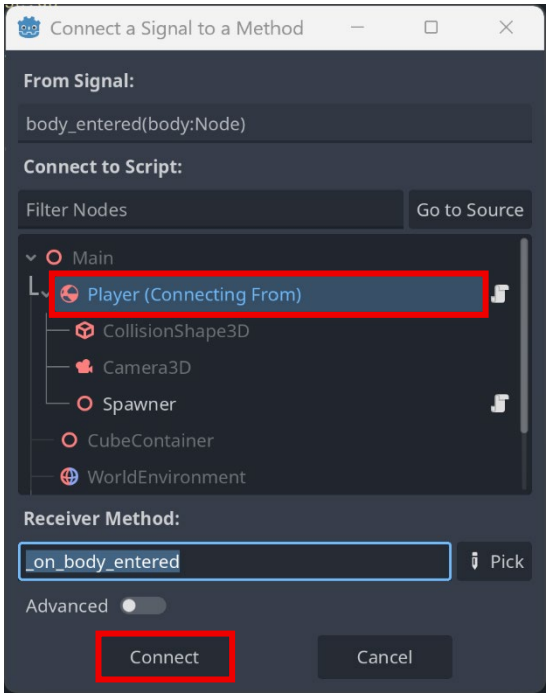
89 Connect the signal!

At the top of the **Inspector**, toggle to the **Node** window and double-click on **body_entered()**.

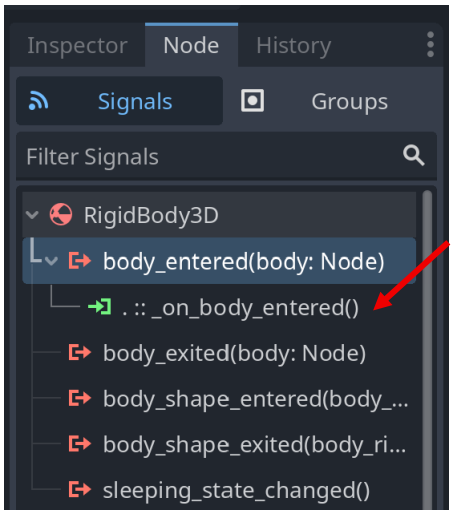


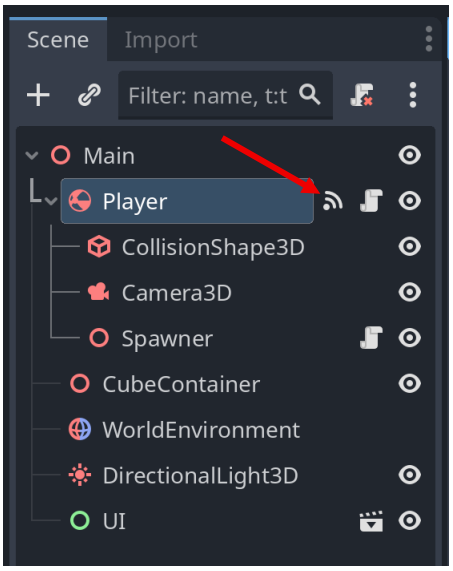
90 In the **Connect a Signal to a Method** window, ensure that **Player** is selected and the receiver method is **_on_body_entered**. Then, click **Connect**.

The **_on_body_entered** method is already coded inside the **player_base.gd** script.



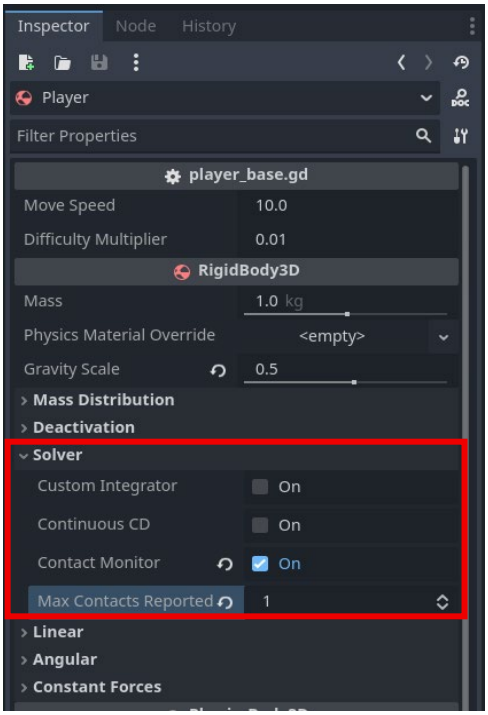
91 Notice that the signal connection is shown in the **Node** window and the **Scene** window.





92 If the game were playtested right now, the collisions would not work. That's because **RigidBodies** need the **Contact Monitor** property enabled to emit signals on collision.

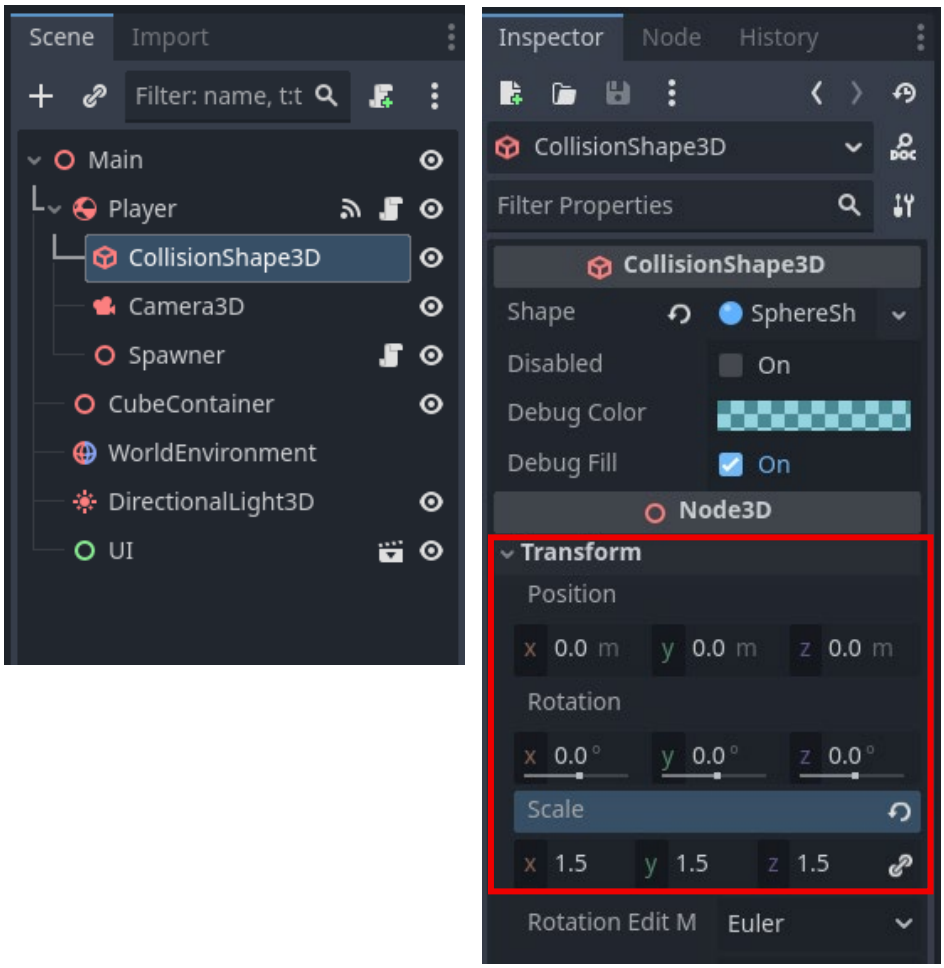
In the **Inspector** for **Player**, open the **Solver** drop-down. Then, enable **Contact Monitor** and set the **Max Contacts Reported** to **1**.



93 Make the player slightly bigger so that collisions happen more consistently!

Select the Player's **CollisionShape3D**. In the **Inspector** and under the **Node3D** section, open the **Transform** drop-down. Set the **Scale** values to **1.5**.

Try playtesting the project and check that the collisions are working!



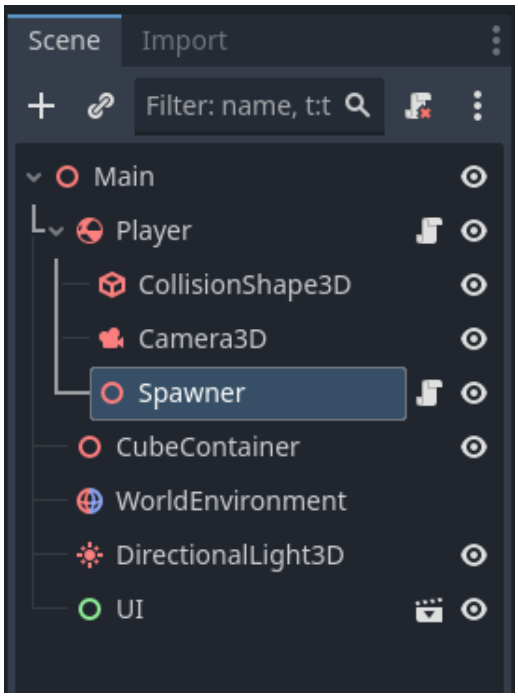
Pause for **Sensei Stop #8**!

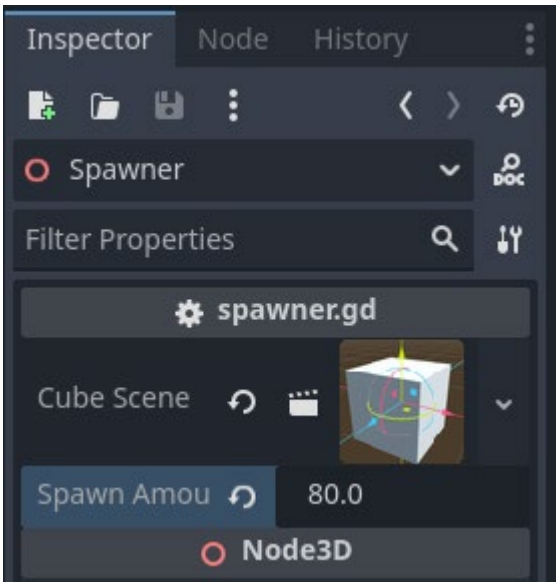
Check in with a Code Sensei before moving on. Make sure that score works, the player collides with the cubes properly, and the game resets.

Reminder: Save your work!

94 Time to customize the game!

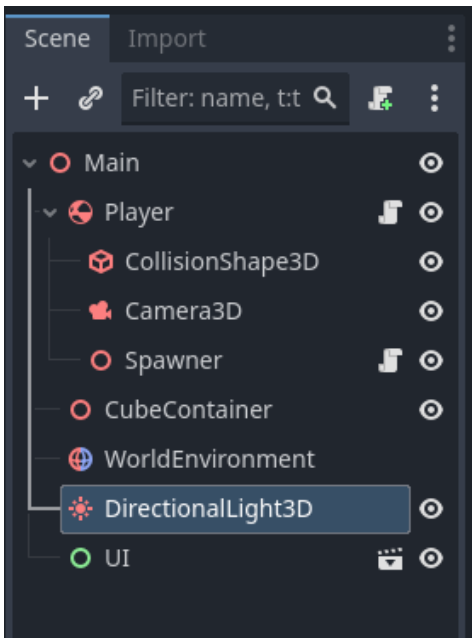
Select the **Spawner** node and tinker with the **Spawn Amount** value to see how that changes the number of cubes that spawn.

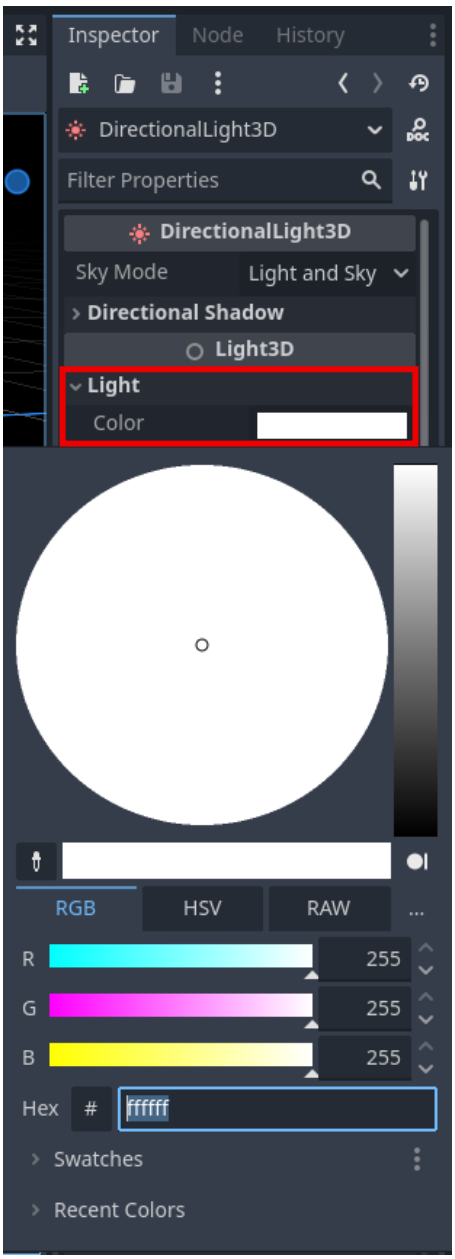




95 Customize the color of the cubes by changing the **DirectionalLight3D**!

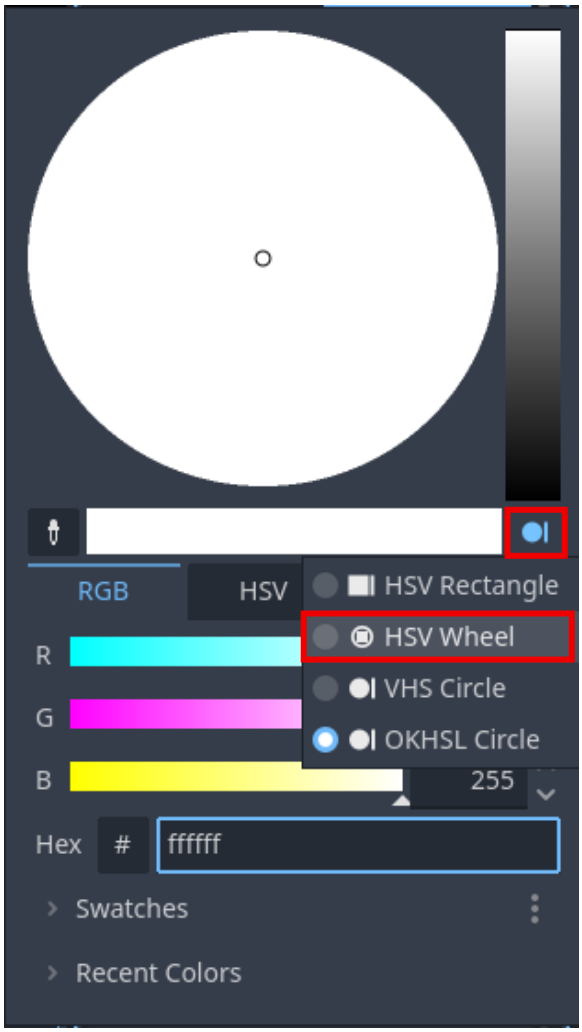
In the **Inspector** for the **DirectionalLight3D,** click the Light drop-down. Select **Color**.





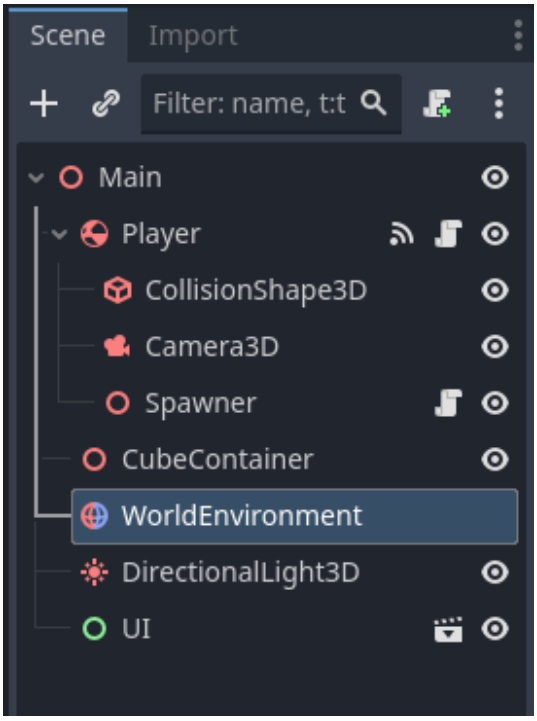
96 Click the  icon and select **HSV wheel**. This will make it easy to select the desired lighting color.

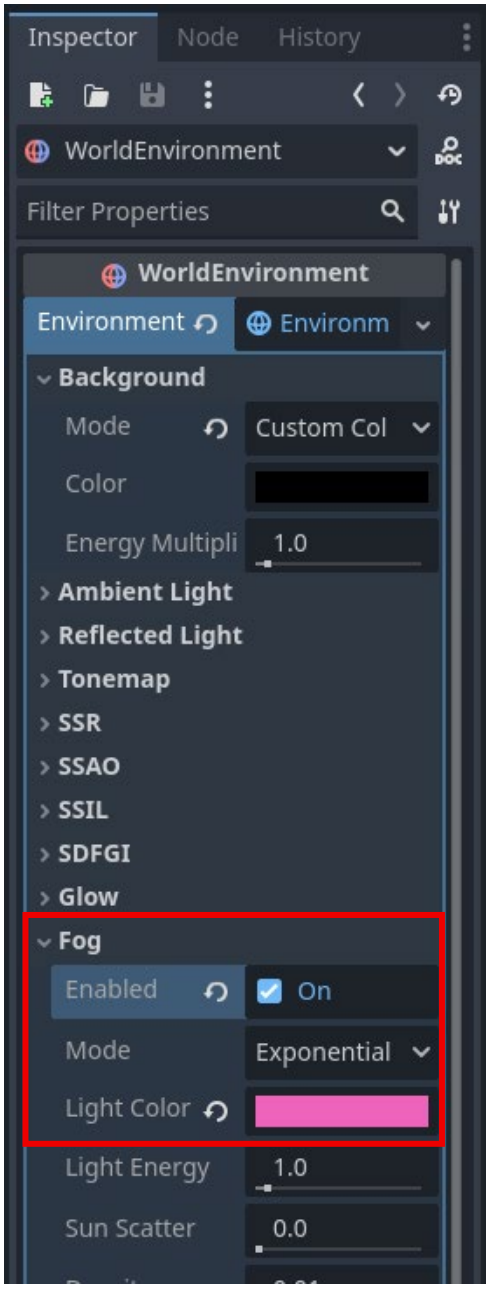
Choose a color, playtest the game, then tweak as needed.



97 The **WorldEnvironment** node has a cool effect called **Fog**. It works just like fog in real life; the further away something is from the camera, the more it fades into the fog.

Select the **WorldEnvironment** node and select **Environment**. Open the Fog drop-down and **enable** it. Then, set the **Light Color** value to the desired color.



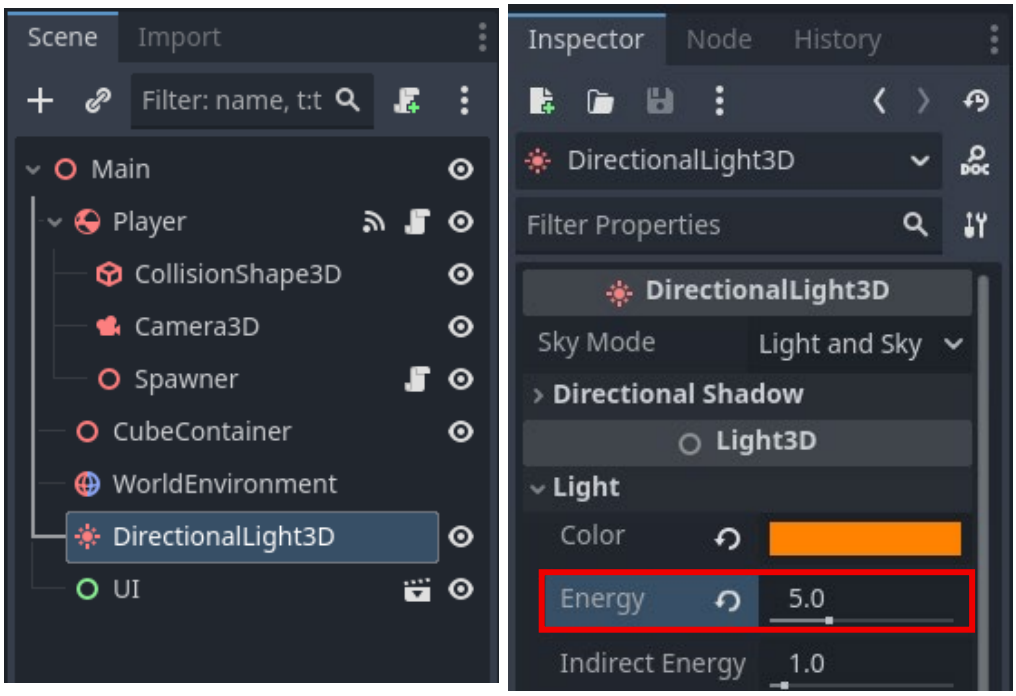


New Concept: Ambient Lighting

The Fog's Light Color setting will override the Background's Color setting for the coloring of the skybox, but it will **not** override the ambient lighting provided by the Background's Color setting. **Ambient Lighting** is a certain amount of lighting that will appear on objects from every angle **no matter what**, even without any actual light sources being present in the scene. If the Background Color was set to Red but the Fog Color was set to Black, there would be a black fog on reddish cubes.

98 If the fog's color is bright enough, it will make the cubes look unnaturally dark.

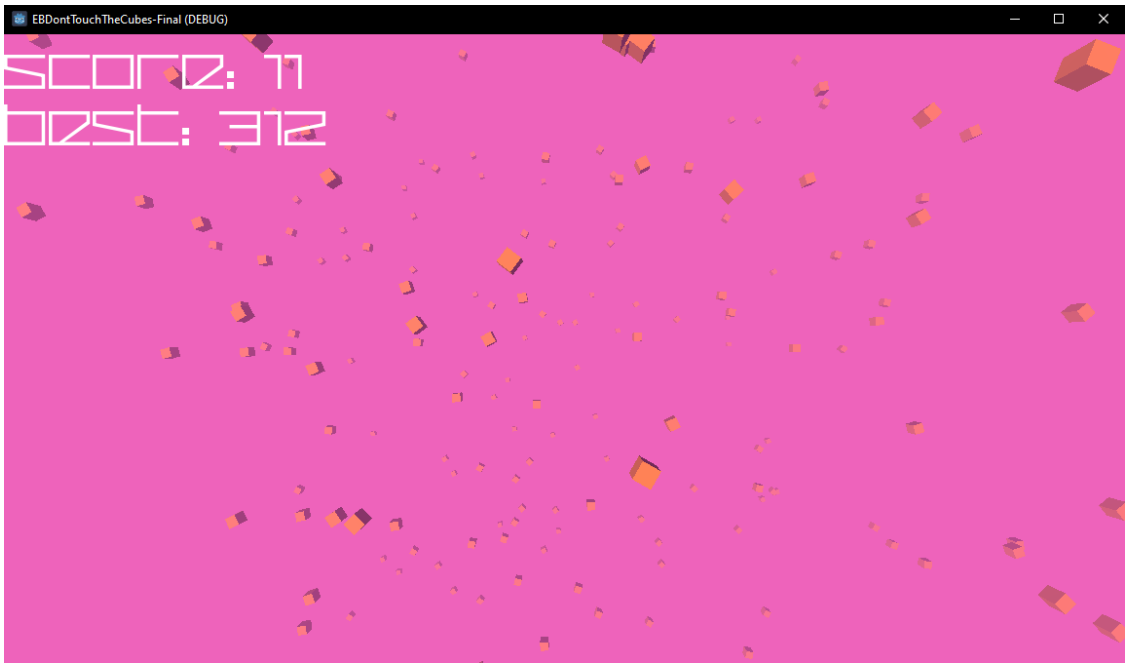
In the **Inspector** for **DirectionalLight3D** set the **Energy** to a high value like **5**.



Pro Tip:

The **Light3D** node (which DirectionalLight3D extends) has an **Energy** property which determines how strong the light source is.

99 The game is now complete! Playtest the game. What is your best score?



Pause for **Sensei Stop #9**!

Congratulations on creating your first game with mesh instances and lighting in Godot! Great job!

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

- What did you learn about mesh instances and lighting?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!