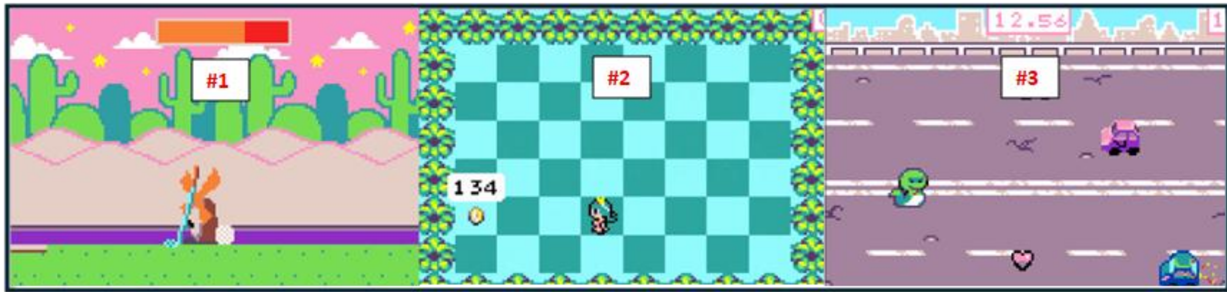
Silver Belt Ninja Guide

Activity 02: Find the Exit

TRANSLATION VS VELOCITY

In Godot, there are two ways to move nodes around. The first one is via **Translation**, which means that the code is explicitly telling the node exactly where to be located each frame. The second is through **Velocity**, which is only available on **Rigidbody2D/3D** and **CharacterBody2D/3D** nodes, and allows the use of Godot's **physics simulations** to calculate the resulting movements and collisions.



Check out these screenshots of games from earlier belts in IMPACT. Can you figure out which games use **Translation** and which ones use **Velocity**?

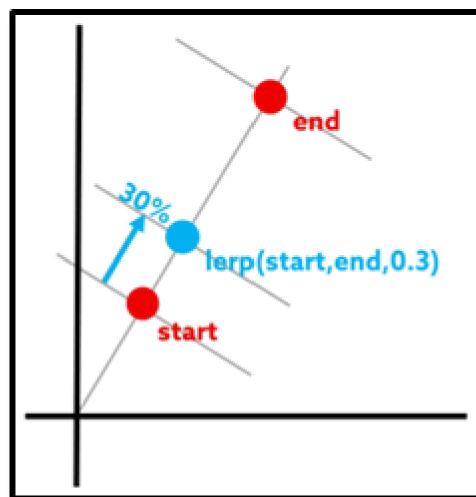
Only **#2** uses **Translation**, and it changes the player's X or Y by 16 pixels based on the direction button being pressed. Both **#1** and **#3** use **Velocity**! In **#1**, the golf ball is set at a **Velocity** based on the charge of the top bar, and in **#3**, the cars are moving at a constant **Velocity** that is set when they are spawned.

Linear Interpolation

Linear Int**erp**olation, or lerp, is a common method used in game design that allows the developer to blend two values together. The method computes a value between the given **start** and **end** values based on the third value, **weight**. **Weight** is the percentage of “blend” from **start** to **end** in decimal form, so its values range from **0** (0%) to **1** (100%).

Lerp is often called with a **weight** that changes over a fixed period of time. For example, it is used in moving platforms, animations, and transitions.

An example of a lerp call is shown below, where the method is called with the weight parameter being 0.3:



Activity 02: Find the Exit

In this activity, you will add movement to a character by *translating* the object at a fixed rate of time. Later, you will switch the character movement to velocity through Rigidbodies.

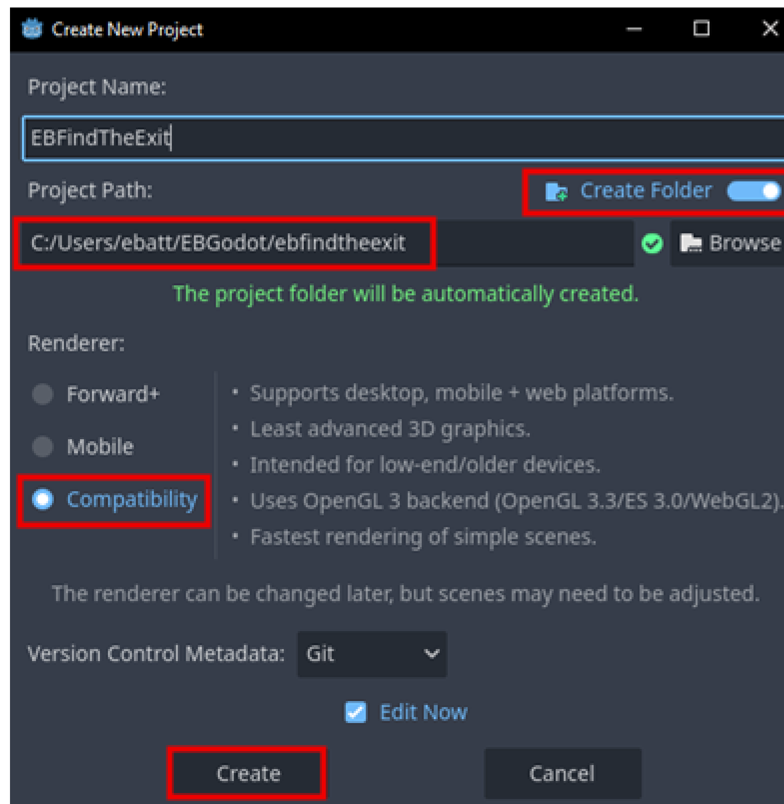
Your mission: Escape the maze in the least amount of time possible. If you press **Play,** you'll discover you are trapped! The player character, Codey, can't move yet because there is no movement code. In this activity, you'll code the player character's movement to enable it to move around the maze, advance through challenges, and reach the exit!

1

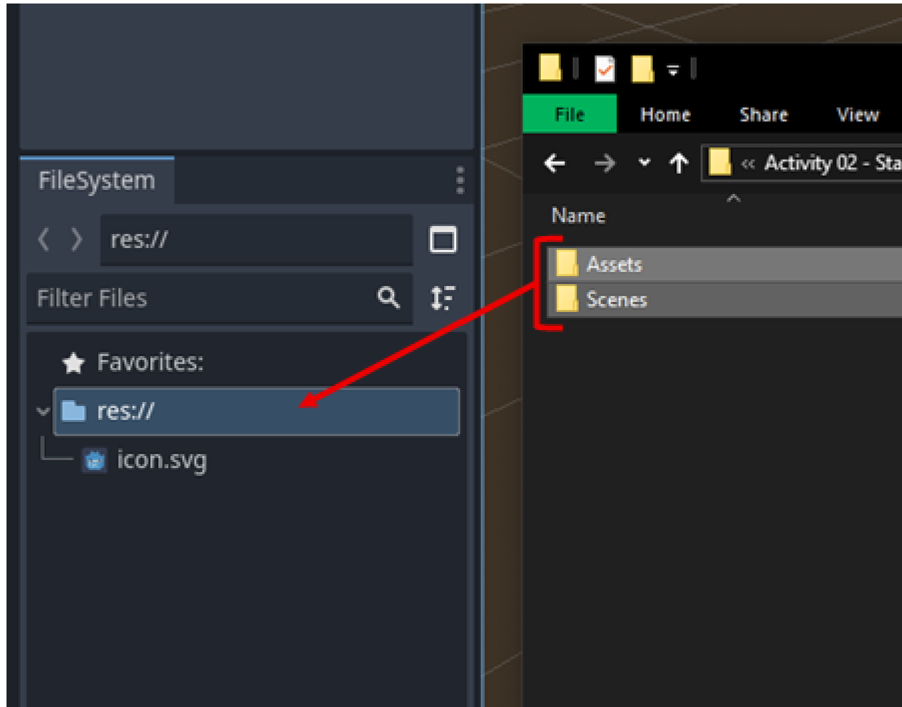
After opening Godot, in the top left corner select **+ Create**.

A **Create New Project** window will pop up. Name the project [**YourInitials]FindTheExit.**

Enable **Create Folder** if it is not already enabled. Make sure the **Project Path** is the same as previously set by a Code Sensei at your center. Ensure that the project is in **Compatibility** mode, then click **Create**.



2 **Extract SB Activity 02 – Ninja Starter Pack.zip** and drag all the folders into res:// in **FileSystem**.



3 So far, you have built many 2D games. From beginning in **Welcome to Godot** up to the previous project, **Robomania**, you've gotten practice with Godot's 2D environment.

In contrast, you have made only a couple of 3D games: **Dropping Bombs** and **Don't Touch the Cubes**. In fact, Dropping Bombs is mostly a 2D game that uses a 3D environment since the main camera is in **Orthographic** mode.

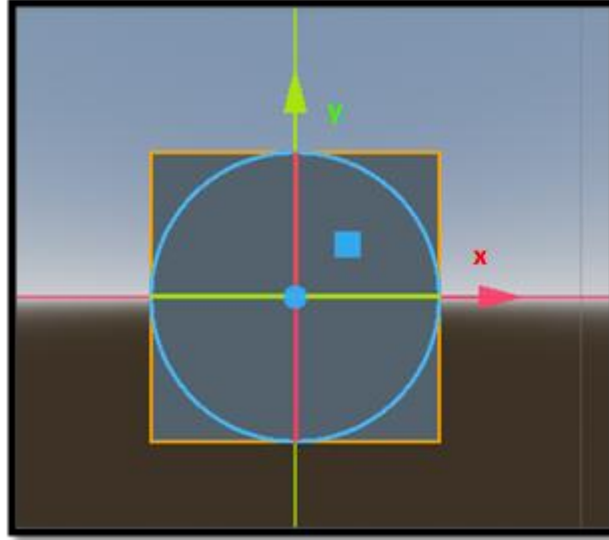
2D Rendered Projects

- Welcome to Godot
- Scavenger Hunt
- Meany Bird
- Sketch Head
- SuperShapes
- PolyRun
- Robomania

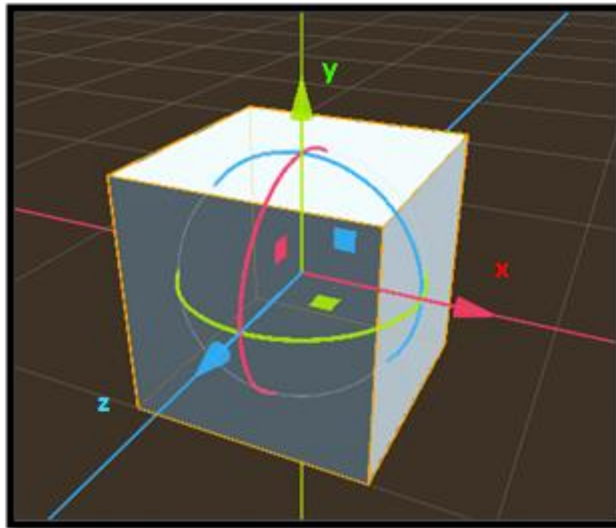
3D Rendered Projects

- Dropping Bombs
- Don't Touch the Cubes

4 2D scenes appear flat on screen with only **X** and **Y axes**.



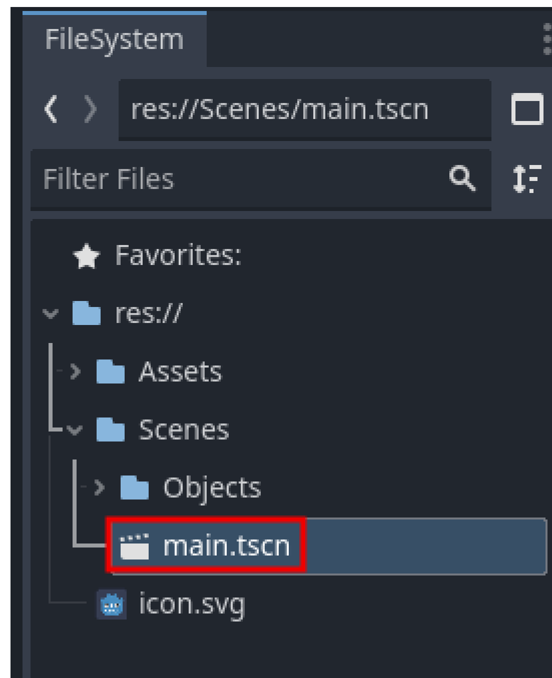
3D scenes appear to have depth and volume with **X, Y,** and **Z axes**, adding the forward/backward dimension.

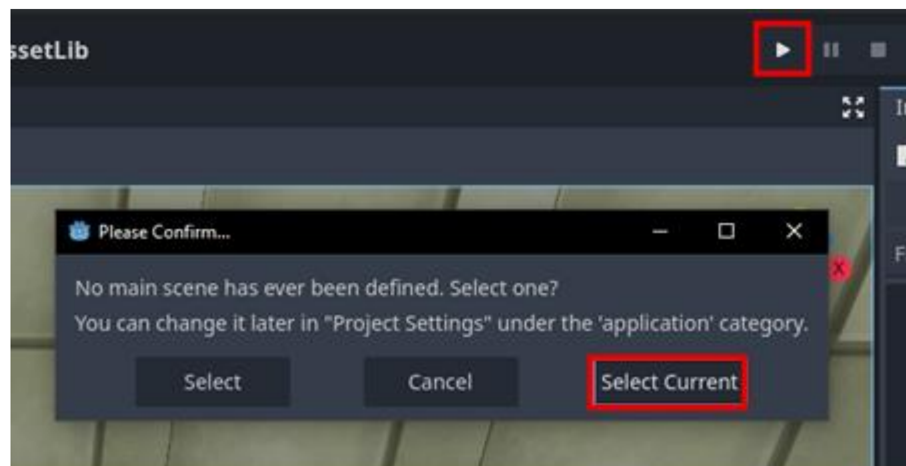


Since the two **environments** are so different, most **Nodes** are specific to the currently used type of **environment**.

This project will be created in the **3D environment**, so most **Nodes** that will be used are for **3D only**.

5 In **FileSystem**, navigate to **main.tscn** and double-click to open the scene. In the top right corner, start a playtest and **Select Current** as the main scene.





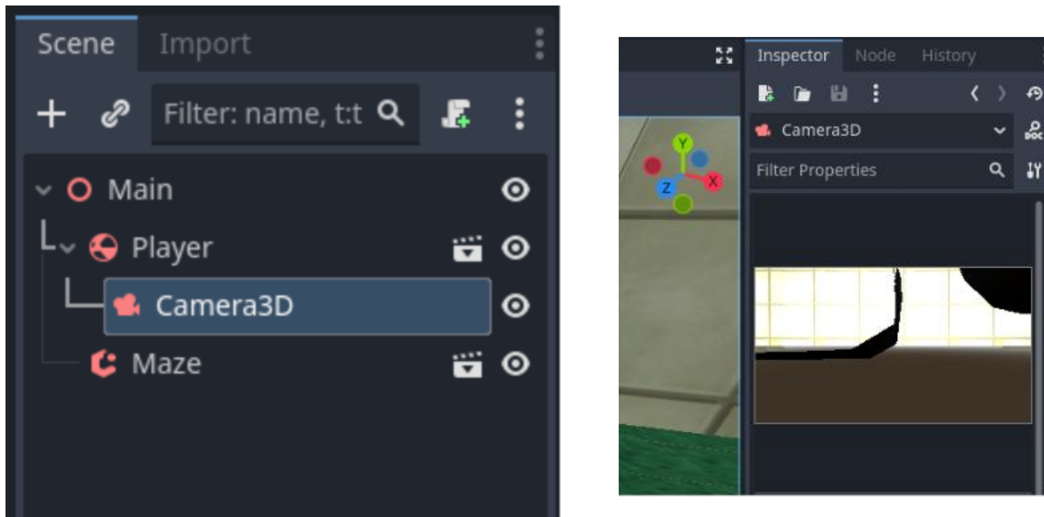
6 Uh oh! The game is just a gray screen. This is because there isn't a **Camera3D** node yet.

Pause for **Sensei Stop #1**!

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

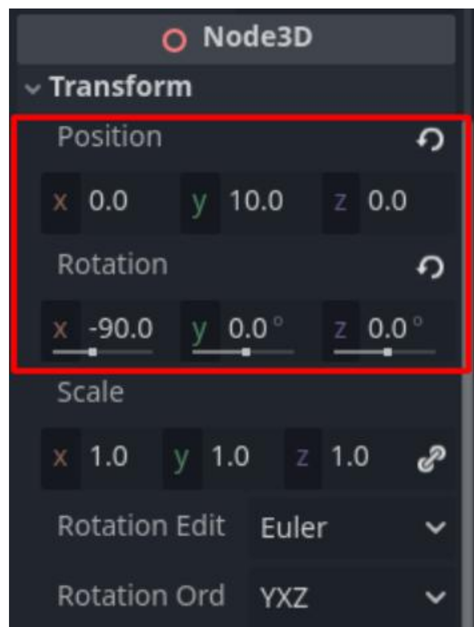
Reminder: Save your work!

7 Add a **Camera3D** as a child to **Player**. Notice that in the **Camera3D's Inspector**, there is a preview of what the camera sees. So far, the view doesn't seem too great.

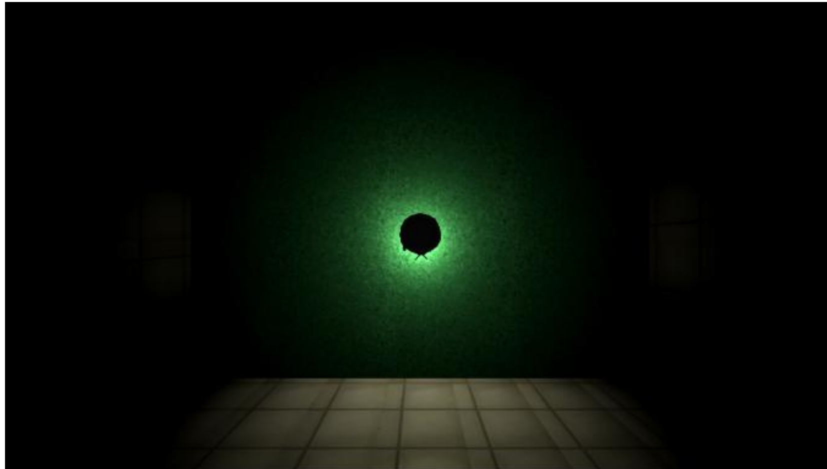


8 In the **Inspector,** scroll to **Node3D** and open the **Transform** drop-down menu. Update the values below:

Position (x, y, z): 0, 10, 0
Rotation (x, y, z): -90, 0, 0

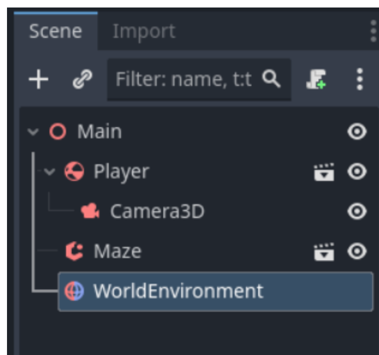


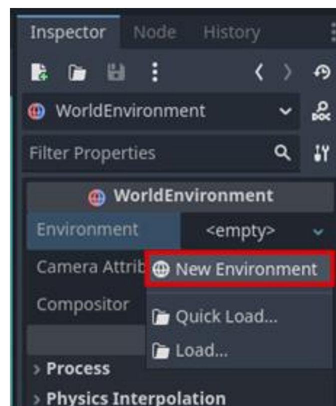
9 Playtest the game. Notice how the areas far from the player are pitch black.



10 The maze is too dark! This maze takes place outdoors, so there should be some ambient moonlight.

Add a **WorldEnvironment** node as a child node to **Main**. In the **Inspector**, set its Environment to **New Environment**.



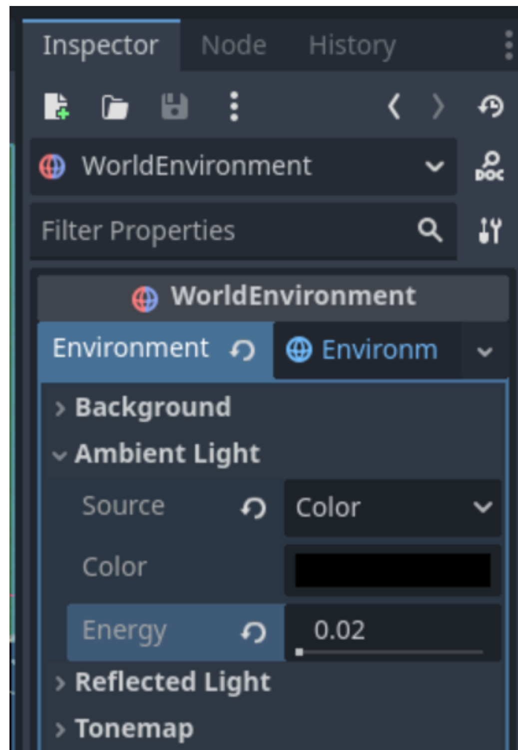


Reminder:

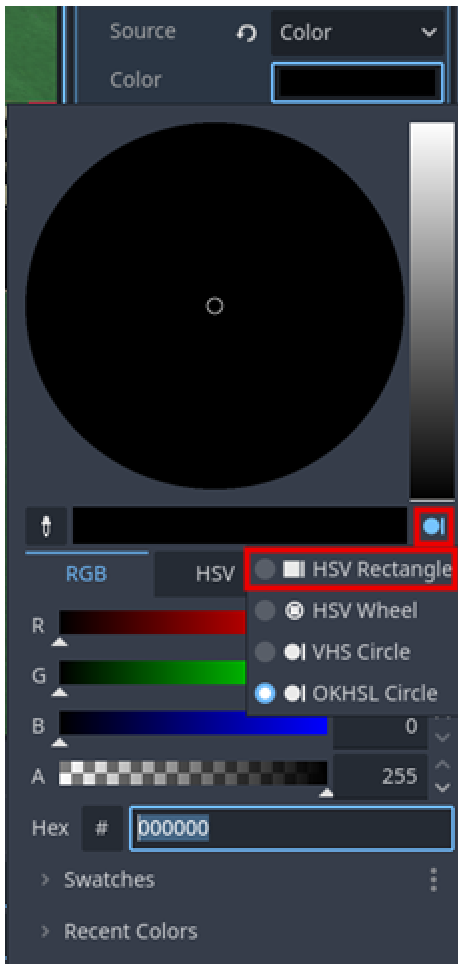
In **Don't Touch the Cubes**, a **WorldEnvironment** was used to add Ambient Lighting!

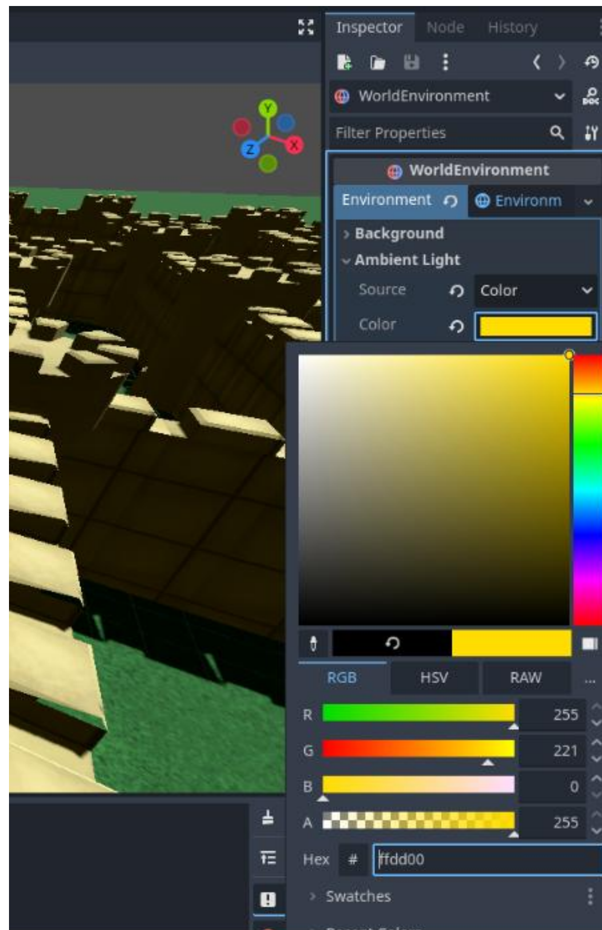
11 Click on the new **Environment** to edit it and open the **Ambient Light** drop-down menu.

Set the **Source** to Color and the **Energy** to 0.02.



12 In the **Color** window, click the **Select a picker shape** icon and choose **HSV Rectangle**. From here, select any color!

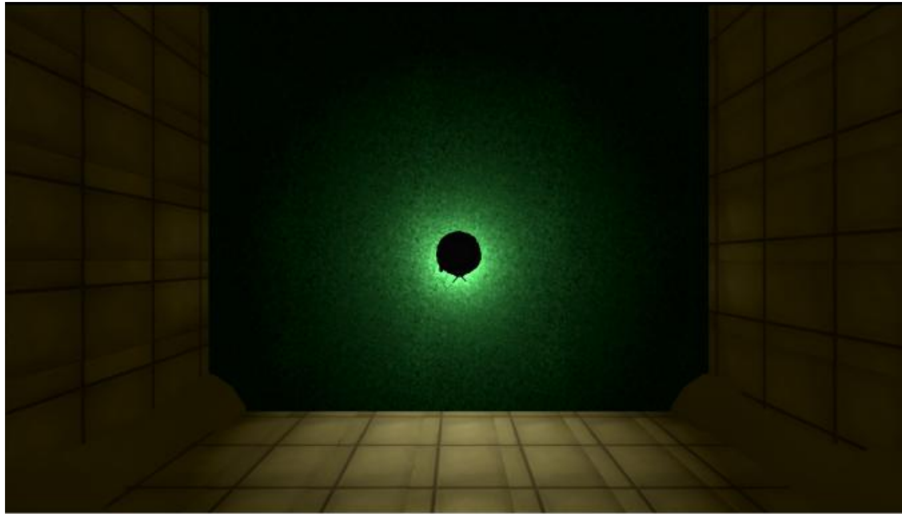




Pro Tip:

For the best appearance, try a mix of **blue** and **white**! That will make it look like the maze is illuminated by moonlight!

13 Playtest the game. Check that the ambient light has been added. If not, review the previous steps.

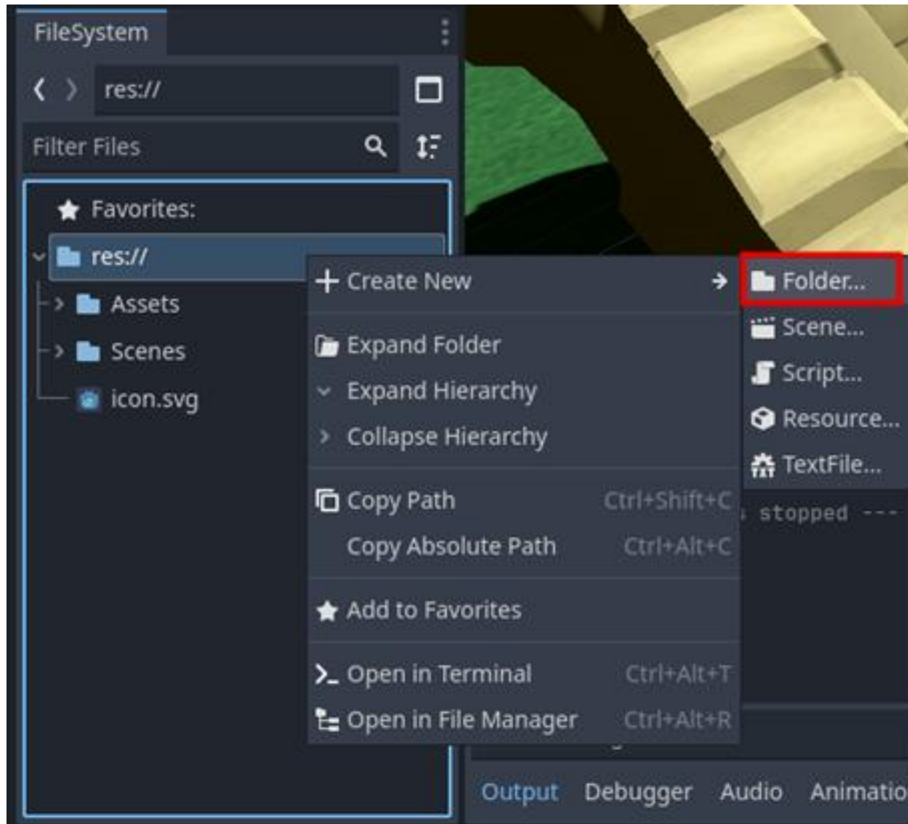


Pause for **Sensei Stop #2**!

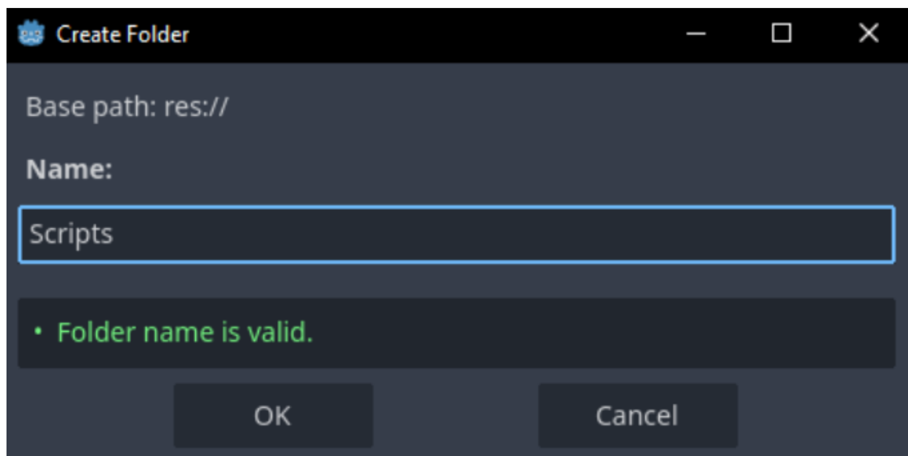
Check in with a Code Sensei before moving on. Ensure the camera is **top-down** and the **ambient light** is added.

Reminder: Save your work!

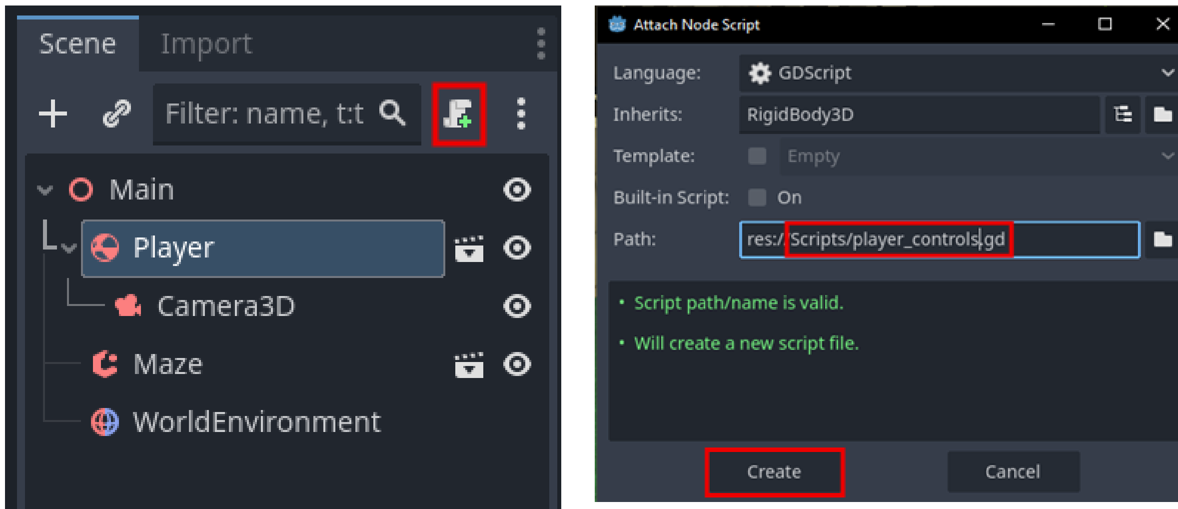
14 In **FileSystem**, right-click **res://** and hover over **Create New** to create a new **Folder**.



Name the new folder **Scripts** and click OK.



15 In **Scene**, select **Player** and attach a new script. Name the script **player_controls.gd** and ensure it is saved to the Scripts folder.



16 Declare a `move_speed` export variable of type `float`. Set the default value to `10.0`.

```
1    extends RigidBody3D
2
3    @export var move_speed: float = 10.0
```

Reminder:

Export variables allow values to be easily tweaked in the **Inspector**.

17

Time to make Codey move! For now, test if setting Codey's position every frame is good enough of a solution.

Define the `_physics_process()` method and change Codey's position (`global_position`) every frame by the following four values multiplied together:

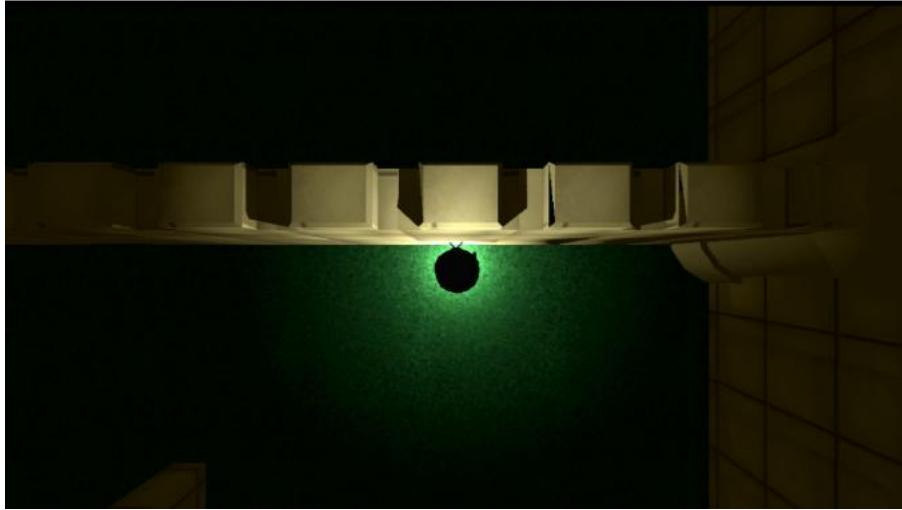
1. `Vector3.FORWARD`
2. `global_basis`
3. `move_speed`
4. `delta`

```
1  extends RigidBody3D
2
3  @export var move_speed: float = 10.0
4
5  func _physics_process(delta: float) -> void:
6      global_position += Vector3.FORWARD * global_basis * move_speed * delta
```

Pro Tip:

Multiplying the above four values allows Codey to move forward because it scales the **`FORWARD` vector** to the Player's transform, making the movement point in the Player's forward-facing direction. The rest of the variables, **`move_speed`** and **`delta`**, are number values used to scale how quickly the **`global_position`** moves.

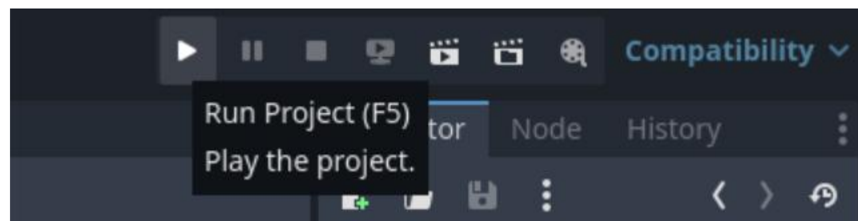
18 Playtest the game. Codey should walk forwards and successfully stop at a wall.



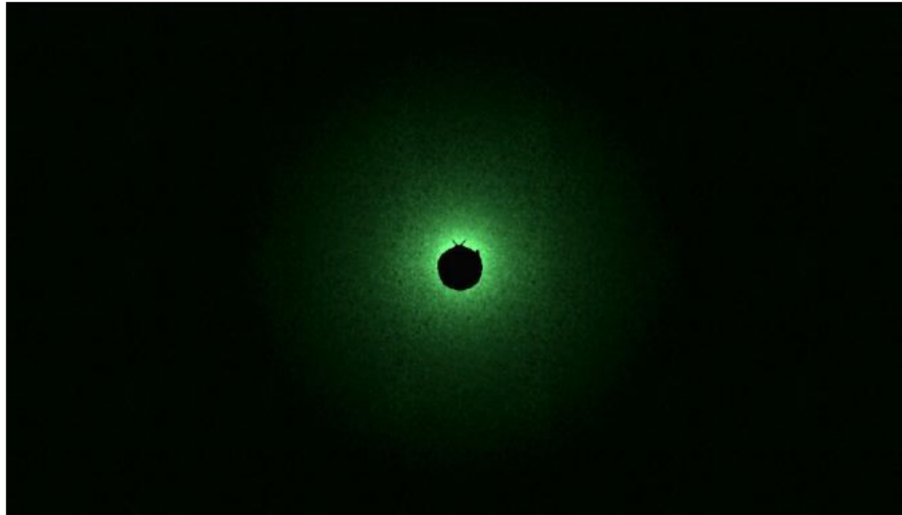
19 Stop the Playtest.

In the **Inspector** for **Player**, tinker with the **Move Speed** export variable and set it to a high value, like 100-200.

20 **Playtest** the project again. What happens?



21 Oh no, Codey can move through walls at high speeds! They should speed out of the maze, onto the surrounding grass, then out of the map over a gray background.



When Codey has walked up to a wall, in the next frame, Godot calculates Codey's next position to be on the other side of the wall.

Typically, in game development, this issue is avoided by using velocities instead of directly changing the position.

Pause for **Sensei Stop #3**!

Check in with a Code Sensei before moving on. Make sure that the Player moves so fast that they **phase through walls** and move **out of the maze**.

Reminder: Save your work!

22 Back in the **player_controls** script, use **#** to comment out the previous line so that it does not run.

```
1  extends RigidBody3D
2
3  @export var move_speed: float = 10.0
4
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis * move_speed * delta
7
```

23 **RigidBody3D** nodes have a property called **`linear_velocity`** that is a very useful alternative to updating a **Node3D's** **`global_position`** property**.**

On the line below, set the **`linear_velocity`** property to the **`FORWARD`** vector multiplied by the **`move_speed`** export variable.

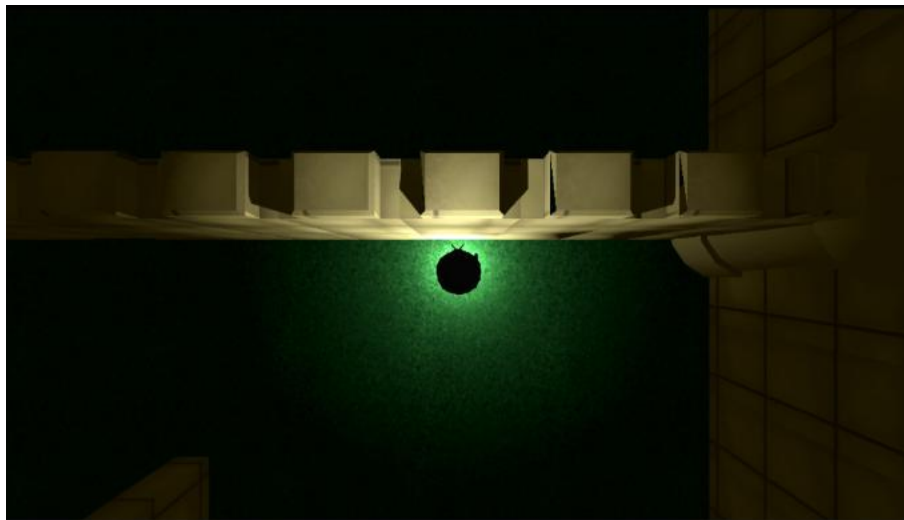
```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * globa
7      
```

24 Check the code! Update the script as needed.

```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis
7      linear_velocity = Vector3.FORWARD * move_speed
```

25 Playtest the project again with the `move_speed` variable still set to 100-200.

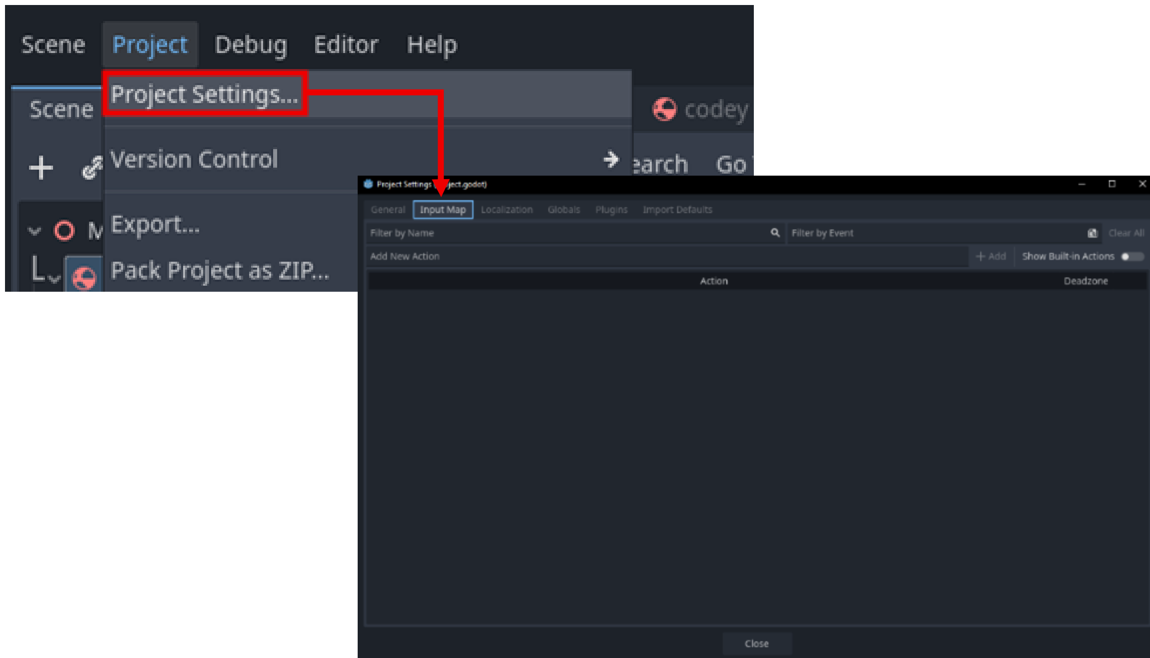
Notice that Codey still hits the wall but stays inside the maze because of Godot's handling of the RigidBody!



26

Add custom controls to move Codey around!

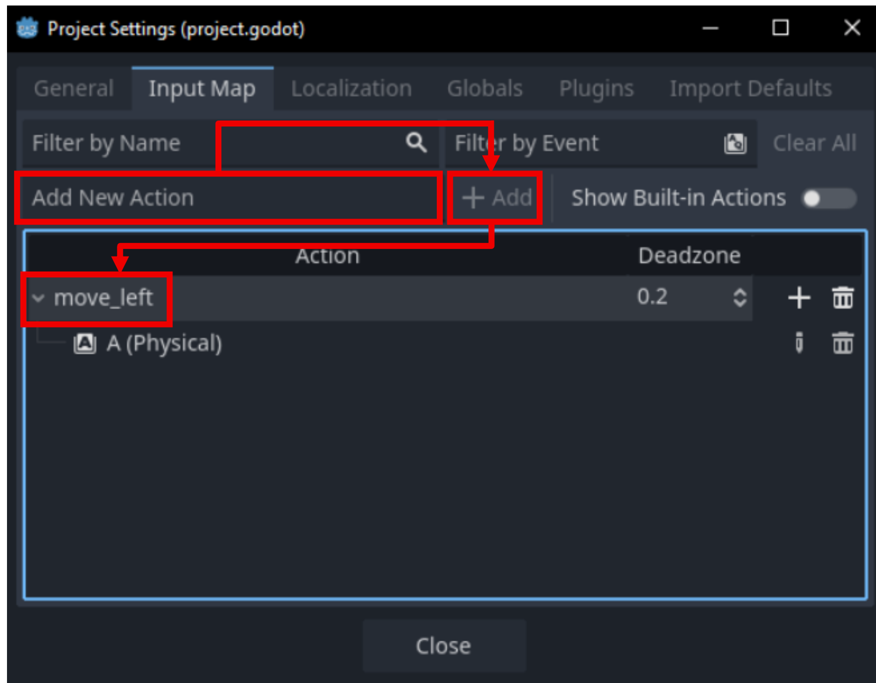
Navigate to the **Input Map** for the project which can be found in **Project Settings**.



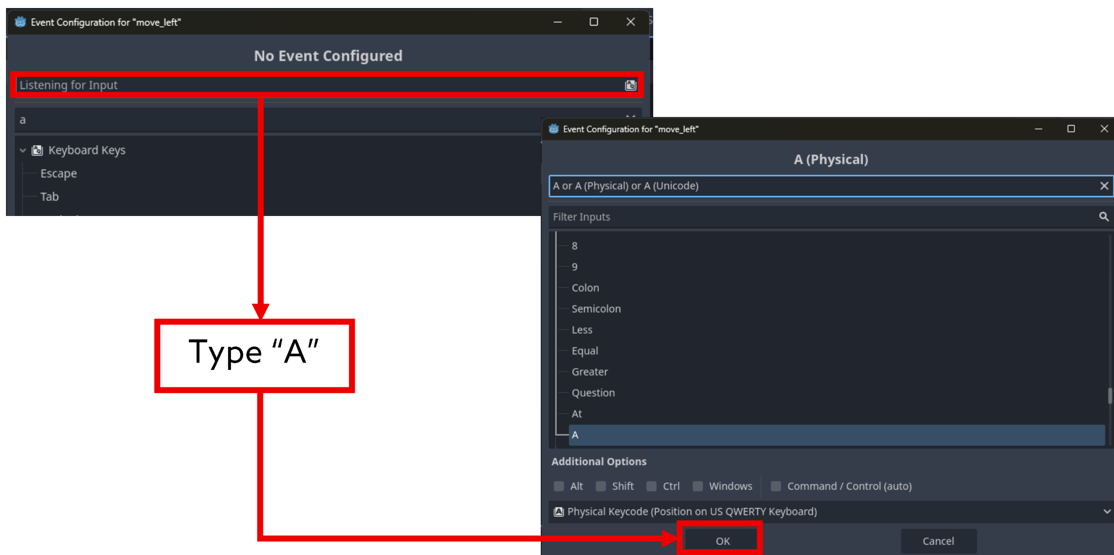
27

Create a **move_left** action mapped with the A button!

In the **Add New Action** text box, type **move_left** then click **+ Add**. An empty **move_left** action should appear.



Press the **+ sign** to the right of it. In the pop-up window, select the "**Listening for Input**" field and press **only** the **A** key so Godot knows what input to map the action to. When **A** is selected, click **OK**.

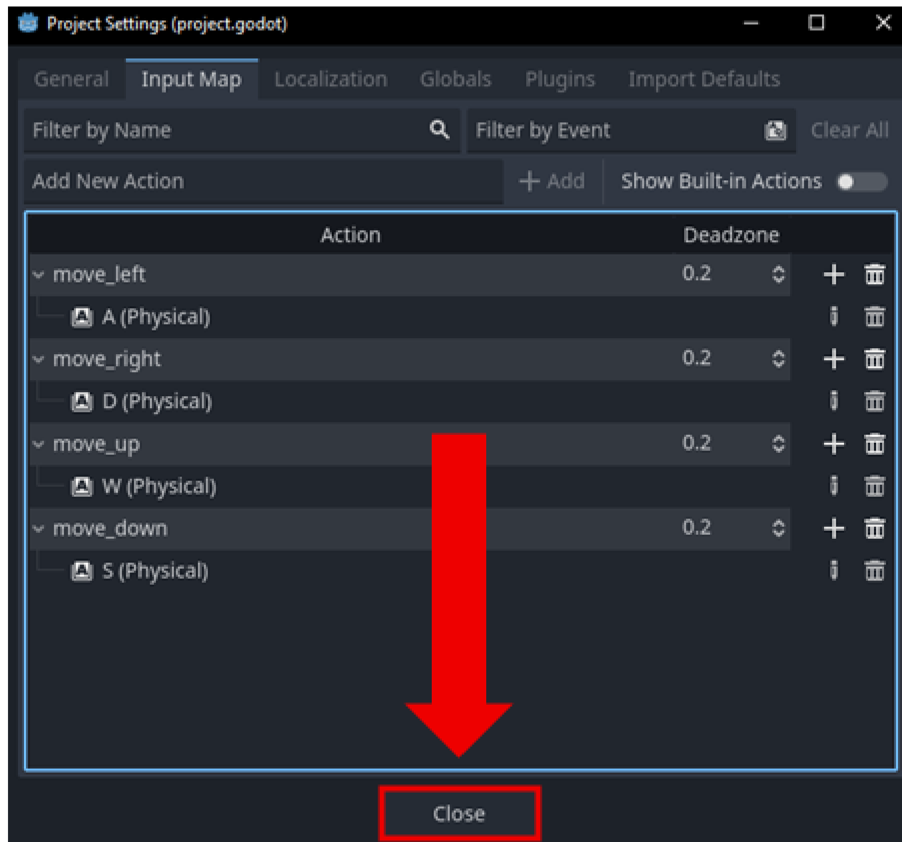


The input mapping should appear in the **Input Map** window.

28

Repeat the previous step for **move_right**, **move_up**, and **move_down** so that the WASD keys control all the cardinal directions!

When done, click **Close**.



29

One of the most helpful tools to debug code is the `print()` method! `print()` allows for logging of information to the **Output Bottom Panel** as the game is being run, and it has no return value.

For example, when developing a game that has a **score** variable but doesn't have a UI to show the score yet, a convenient way to keep track of the **score** is to print it to the **Output Panel** with the `print()` method!

> `print()`**:** displays given arguments as text in the Output Bottom Panel of the editor while the game is running.
>
> **Parameters:**
>
> 1. `argument` ***(any)*:** any number of arguments of any type may be sent to a print call.
>
> **Returns *(void)*:** No return value but displays all arguments together as nicely as possible to the Output Bottom Panel, followed by a newline character.

30

Navigate back to the **player_controls** script.

Add a `print()` call inside of `_physics_process()` so that the horizontal button being pressed is printed to the **Output Bottom Panel**. In other words, print the return value of `Input.get_axis()` with parameters **`"move_left"`** and **`"move_right"`**.

```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis * move
7      linear_velocity = Vector3.FORWARD * move_speed
8
9      print(Input.get_axis("move_left", "move_right"))
10
```

What number will be printed in **Output** when the A key is pressed? What about the D key? What will be printed when neither key is pressed?

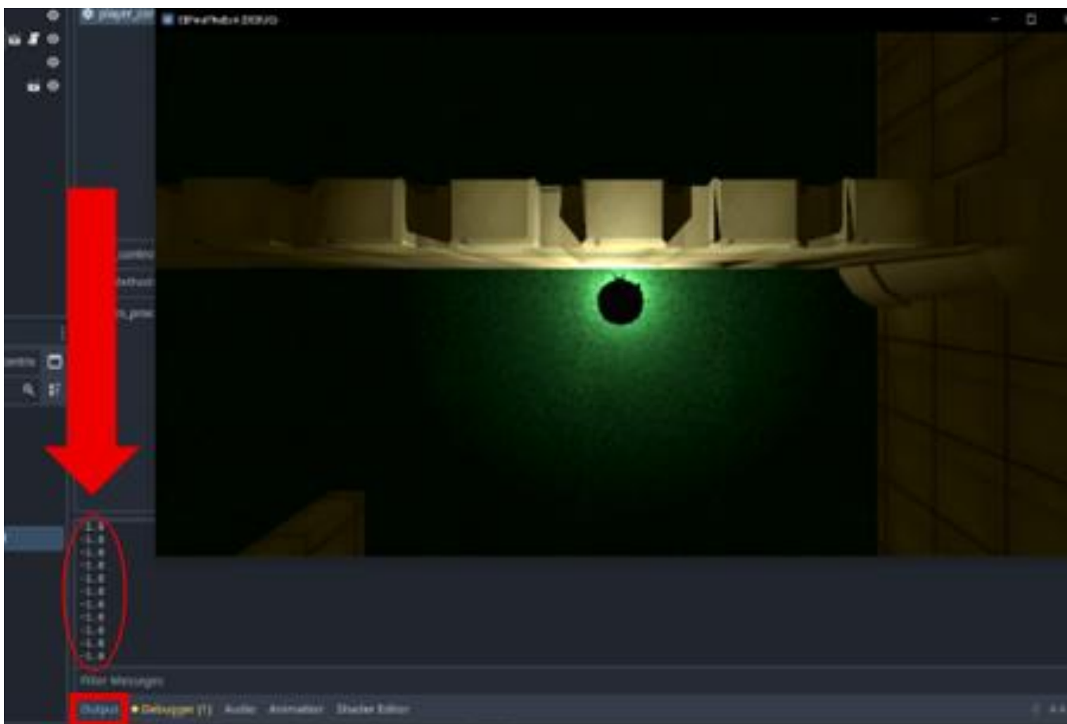
Reminder:

`Input.get_axis()` was previously used in **SuperShapes** where pressing left/right would cause the player to rotate counterclockwise/clockwise.

31 Playtest the game.

Look at **Output** while the game runs to see the result of `Input.get_axis()`.

Test pressing the A key, the D key, both, and neither. What numbers are displayed when each button is pressed?



32

When **A** is pressed, `Input.get_axis()` returns **-1**. When **D** is pressed, `Input.get_axis()` returns **1**. When **neither** or **both** keys are pressed, `Input.get_axis()` returns **0**.

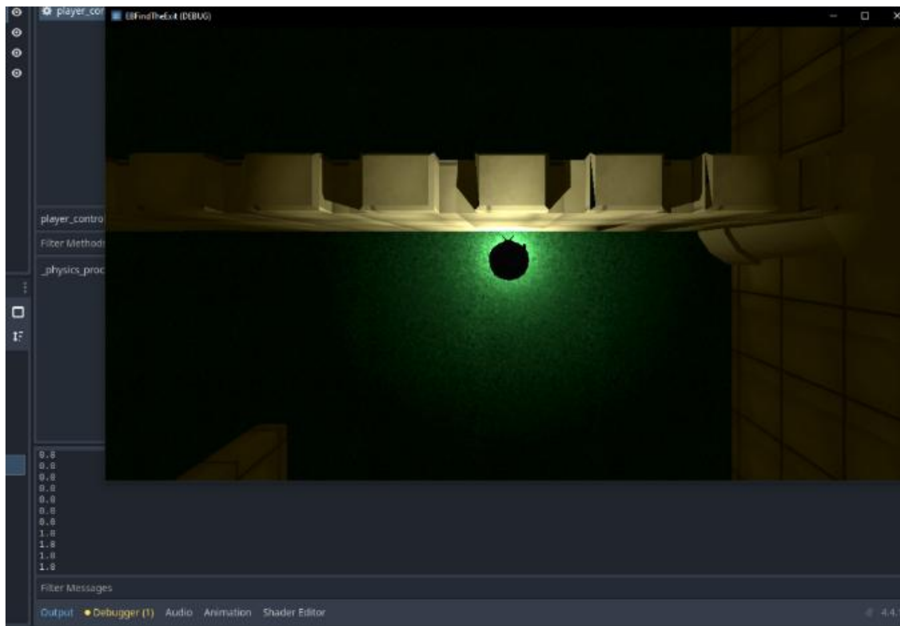
Navigate back to the script.

Update the print statement inside `_physics_process()` and pass **"move_down"** and **"move_up"** as parameters.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis
7       linear_velocity = Vector3.FORWARD * move_speed
8
9       print(Input.get_axis("move_down", "move_up"))
10
```

33

Playtest the game and test out the **W** and **S** keys. What happens?



34 When **S** is pressed, **`Input.get_axis()`** returns **-1**. When **W** is pressed, **`Input.get_axis()`** returns **1**. When **neither** or **both** keys are pressed, **`Input.get_axis()`** returns **0**.

From these experiments, notice that in **`Input.get_axis()`**, the first parameter **`negative_action`** returns **-1** while the second parameter **`positive_action`** returns **1**.

```
Method Input.get_axis(negative_action: StringName, positive_action: StringName)
-> float const
Get axis input by specifying two actions, one negative and one positive.

This is a shorthand for writing Input.get_action_strength("positive_action") -
Input.get_action_strength("negative_action").
```

35 Navigate back to the script, delete the print statement, and use **#** to comment out the code that sets the **`linear_velocity`**.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis
7       #linear_velocity = Vector3.FORWARD * move_speed
8
```

Declare **`horizontal`** and **`vertical`** variables, of type **`float`**, inside the **`_physics_process`** method. Use **`Input.get_axis()`** so **`horizontal`** keeps track of the A and D keys, and **`vertical`** keeps track of the S and W keys.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis
7       #linear_velocity = Vector3.FORWARD * move_speed
8       var horizontal: float =
9       var vertical: float =
10
```

36 Check the code! Update the script as needed.

```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis * move_speed * d
7      #linear_velocity = Vector3.FORWARD * move_speed
8      var horizontal: float = Input.get_axis("move_left", "move_right")
9      var vertical: float = Input.get_axis("move_down", "move_up")
10
```

37 On the next line, create an **`input_dir`** variable which is set to a new **`Vector3`**. Pass the following values in this order, separated by commas: **`horizontal`**, **`0`**, **`vertical`**

```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis * move_speed * d
7      #linear_velocity = Vector3.FORWARD * move_speed
8      var horizontal: float = Input.get_axis("move_left", "move_right")
9      var vertical: float = Input.get_axis("move_down", "move_up")
10     var input_dir = Vector3()
11
```

38 Check the code! Update the script as needed.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis * move_speed * d
7       #linear_velocity = Vector3.FORWARD * move_speed
8       var horizontal: float = Input.get_axis("move_left", "move_right")
9       var vertical: float = Input.get_axis("move_down", "move_up")
10      var input_dir = Vector3(horizontal, 0, vertical)
11
```

39 Lastly, set the RigidBody3D's `linear_velocity` property to the normalized scaling of `input_dir`. Use the dot operator to access the `Vector3.normalized()` method, then multiply by the `move_speed` variable.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis * move_speed *
7       #linear_velocity = Vector3.FORWARD * move_speed
8       var horizontal: float = Input.get_axis("move_left", "move_right")
9       var vertical: float = Input.get_axis("move_down", "move_up")
10      var input_dir = Vector3(horizontal, 0, vertical)
11      linear_velocity = |
```

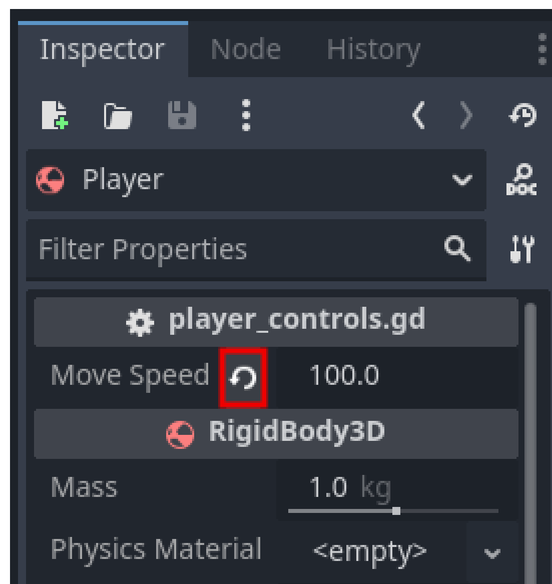
Reminder:

Normalizing is important to ensure that the vector's magnitude isn't too large, and the player won't move unnaturally fast when they're going in two directions at the same time!

40 Check the code! Update the script as needed.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis * move_speed *
7       #linear_velocity = Vector3.FORWARD * move_speed
8       var horizontal: float = Input.get_axis("move_left", "move_right")
9       var vertical: float = Input.get_axis("move_down", "move_up")
10      var input_dir = Vector3(horizontal, 0, vertical)
11      linear_velocity = input_dir.normalized() * move_speed
```

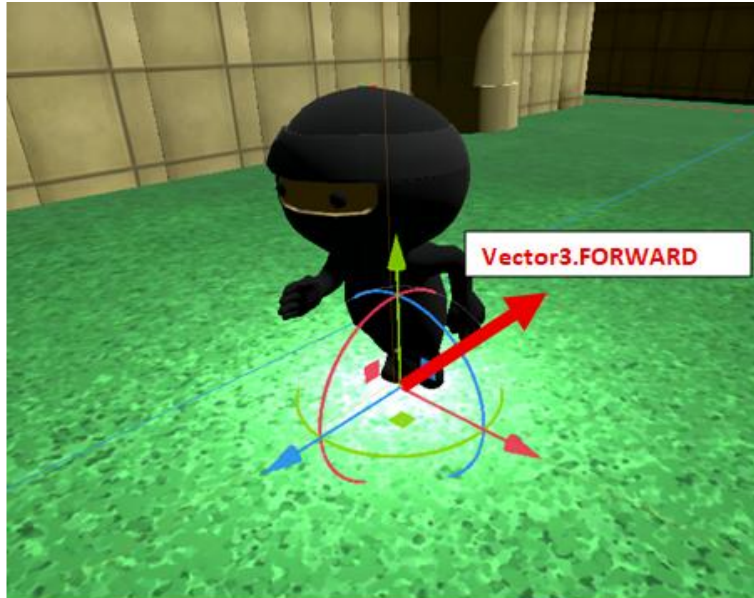
41 In the Player's Inspector, **reset** the **Move Speed** export variable to **10.0**.



Playtest the game. Do the controls work as expected?

42 Oh no! The W and S keys make Codey move the **wrong way**!

This is because `Vector3.FORWARD` points to the negative **z** direction, which is the opposite of what the editor's transform gizmo shows. Notice how it equals (0, 0, -1)!



Constant Vector3.FORWARD: `Vector3 = Vector3(0, 0, -1)`

Forward unit vector. Represents the local direction of forward, and the global direction of north. Keep in mind that the forward direction for lights, cameras, etc is different from 3D assets like characters, which face towards the camera by convention. Use `Vector3.MODEL_FRONT` and similar constants when working in 3D asset space.

In the `Input.get_axis("move_down", "move_up")` code, `move_down` is mapped to -1 which causes the player to **move upwards** from the camera's **top-down view**.

Pro Tip:

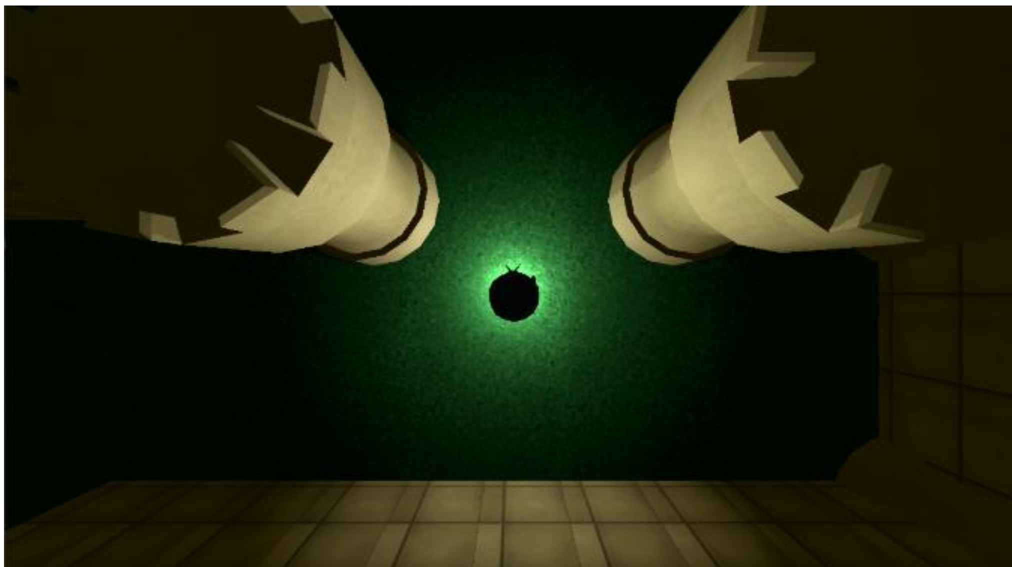
Transform Gizmo is the technical term that refers to the arrows, circles, and squares that appear near objects in the Godot Editor when they are selected.

43 There are a couple ways to resolve the issue. One way is to rotate everything in the game except for Codey by 180 degrees, so that "`move_up`" means up.

The better way of fixing the issue is to swap `move_down` and `move_up` in the code. Navigate to the script and in the `vertical` variable's declaration, swap `move_down` and `move_up` as seen in the image.

```
5  func _physics_process(delta: float) -> void:
6      #global_position += Vector3.FORWARD * global_basis * move_speed *
7      #linear_velocity = Vector3.FORWARD * move_speed
8      var horizontal: float = Input.get_axis("move_left", "move_right")
9      var vertical: float = Input.get_axis("move_up", "move_down")
10     var input_dir = Vector3(horizontal, 0, vertical)
11     linear_velocity = input_dir.normalized() * move_speed
12
```

44 Playtest the game. Try making it to the end of the maze!



Pause for **Sensei Stop #4**!

Check in with a Code Sensei before moving on. Make sure that the **player movement** is coded properly.

Reminder: Save your work!

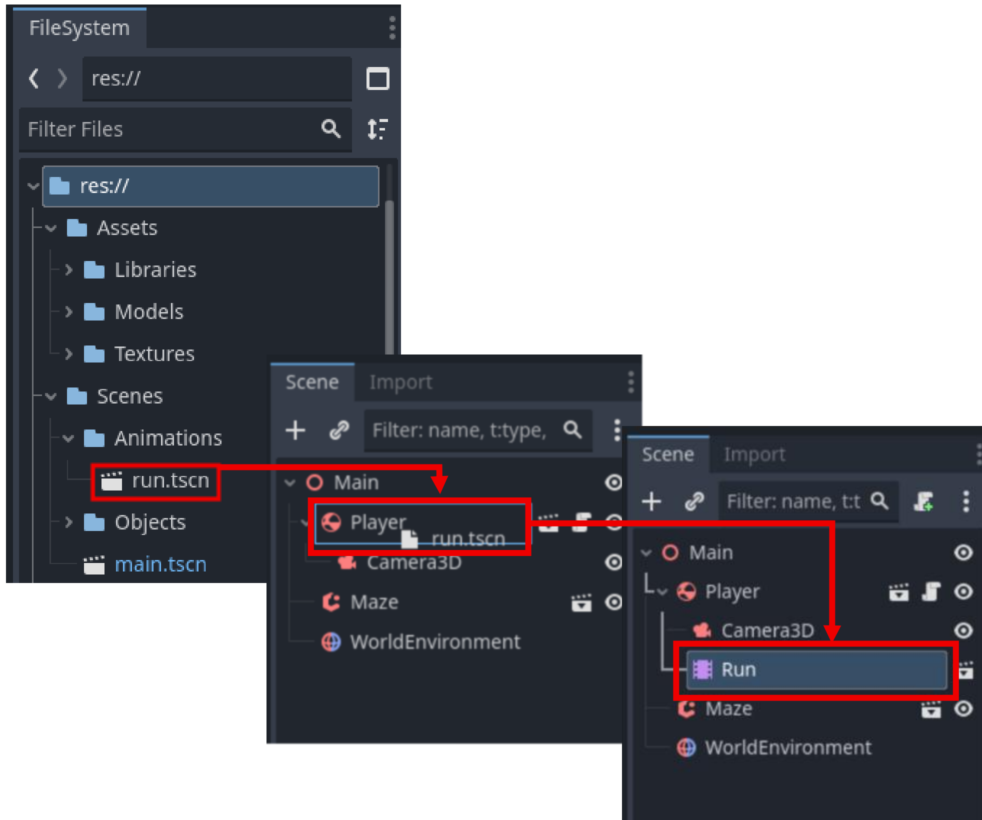
45 As Codey makes their way through the maze, they're stuck in the same stance! This probably isn't fun for Codey and it also doesn't give the user any visual feedback on their inputs.

Return to the **3D** workspace.

46

Don't worry, a running animation has already been set up for Codey! In **FileSystem**, navigate to **run.tscn**.

In **Scene**, drag **run.tscn** onto **Player** to create a new **AnimationPlayer** node called "Run".



Reminder:

AnimationPlayer nodes offer precise control of animations through callable methods in GDScript.

47

Run already has all the animation data loaded into it, so test it out!

In **Scene,** select **Run**. Move the scene view's camera near Codey. Open the **Animation Bottom Panel** and click the **Play Animation** icon to see Codey run!

48 Open the **player_controls** script through either **Scene** or **FileSystem**.

Between the **`export`** variable and the **`_physics_process()`** method, create an **`onready`** variable called **`anim_player`** of type **`AnimationPlayer`**. Set **`anim_player`** to **`get_node("Run")`**.

```
1   extends RigidBody3D
2
3   @export var move_speed: float = 10.0
4
5   @onready var |
6
7 ⌄ func _physics_process(delta: float) -> void:
8 ⌄ >|   #global_position += Vector3.FORWARD * global_b
9   >|   #linear_velocity = Vector3.FORWARD * move_spee
```

Reminder:

`@onready` variables are set right before a node's **`_ready()`** method is called.

49 Check the code! Update the script as needed.

```
1   extends RigidBody3D
2
3   @export var move_speed: float = 10.0
4
5   @onready var anim_player: AnimationPlayer = get_node("Run")
6
7   func _physics_process(delta: float) -> void:
8       #global_position += Vector3.FORWARD * global_basis * move_s
9       #linear_velocity = Vector3.FORWARD * move_speed
```

50 The **AnimationPlayer** node has two useful methods: `target.play("[animation_name]")`, which enables a specific animation, and `target.stop()`, which disables all animations.

Codey should *only* play the running animation when they are moved by the user.

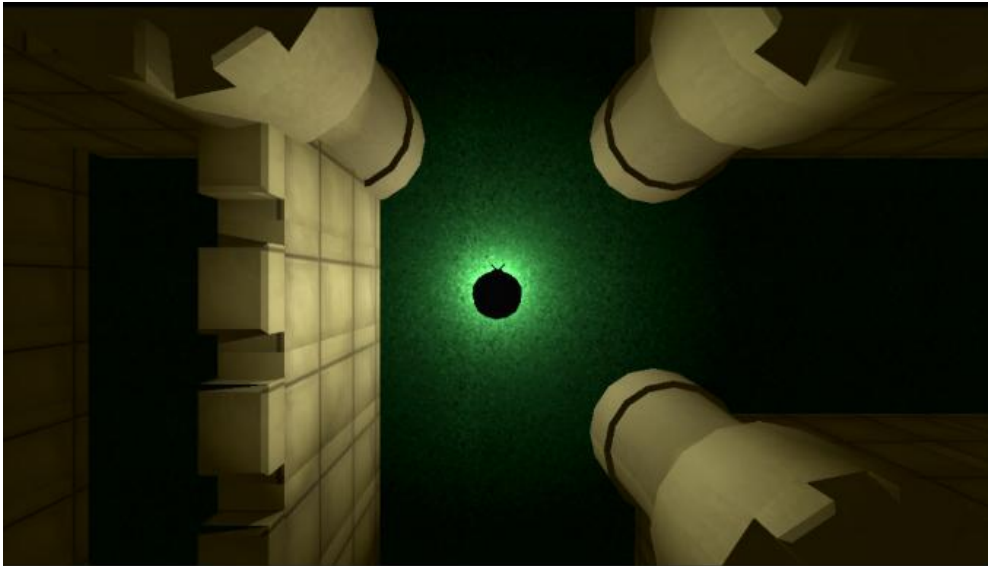
Inside the `_physics_process()` method, write an if statement that checks if `input_dir` is not equal to `Vector3.ZERO`, then use `anim_player` to play the Run animation. Else, stop all animations.

```
5   func _physics_process(delta: float) -> void:
6       #global_position += Vector3.FORWARD * global_basis * move_speed *
7       #linear_velocity = Vector3.FORWARD * move_speed
8       var horizontal: float = Input.get_axis("move_left", "move_right")
9       var vertical: float = Input.get_axis("move_up", "move_down")
10      var input_dir = Vector3(horizontal, 0, vertical)
11      linear_velocity = input_dir.normalized() * move_speed
12
13      if |
```

51 Check the code! Update the script as needed.

```
12  	var input_dir = Vector3(horizontal, 0, vertical)
13  	linear_velocity = input_dir.normalized() * move_speed
14  
15  	if input_dir != Vector3.ZERO:
16  		anim_player.play("Run")
17  	else:
18  		anim_player.stop()
```

52 Playtest the game. Does Codey animate while moving now?



Pause for **Sensei Stop #5**!

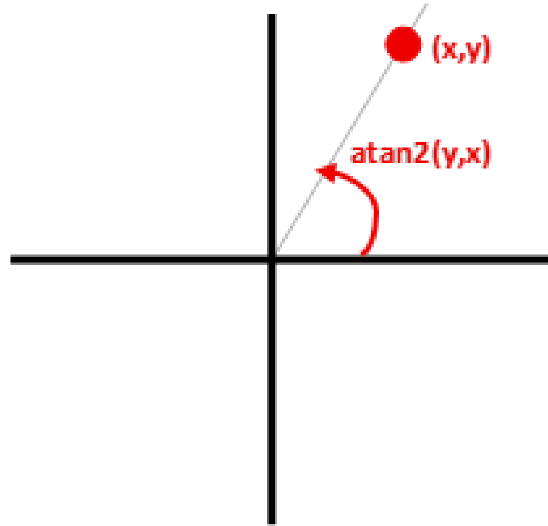
Check in with a Code Sensei before moving on. Ensure that Codey is **animating** but **not rotating** in the direction of movement.

Reminder: Save your work!

53 Almost there! However, Codey doesn't turn in the direction they're moving in, giving the appearance of running in place.

To determine the direction that Codey should be facing, set the y-rotation to face the movement direction.

A useful method for this is called **atan2**([y],[x]).



New Concept: `atan2(y,x)`

Calculates the angle between the positive x axis and the point (x,y), measured with the origin as the pivot.

54 Underneath the line of code that sets the animation to Run, set the Player's `rotation.y` property to `atan2()` of `input_dir.x` and `input_dir.z`.

```
15  if input_dir != Vector3.ZERO:
16      anim_player.play("Run")
17      rotation.y = 
18  else:
19      anim_player.stop()
20
```

55 Check the code! Update the script as needed.

```
15    if input_dir != Vector3.ZERO:
16        anim_player.play("Run")
17        rotation.y = atan2(input_dir.x, input_dir.z)
18    else:
19        anim_player.stop()
20
```

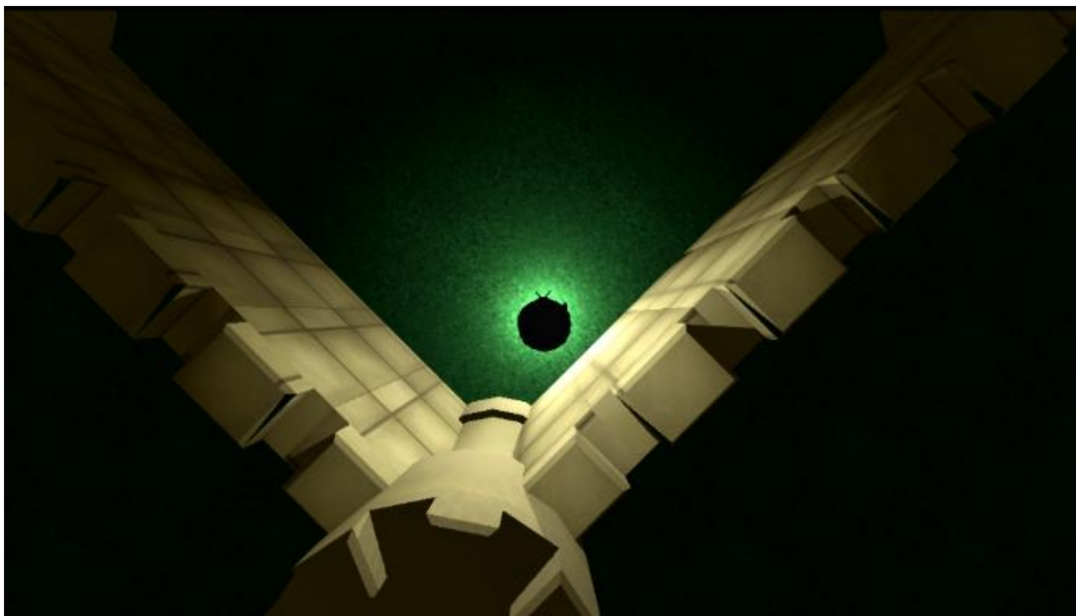
New Concept: RemoteTransform3D

A node that sends its own transform information to another node. It can be adjusted so that only some information is sent, instead of all of it!

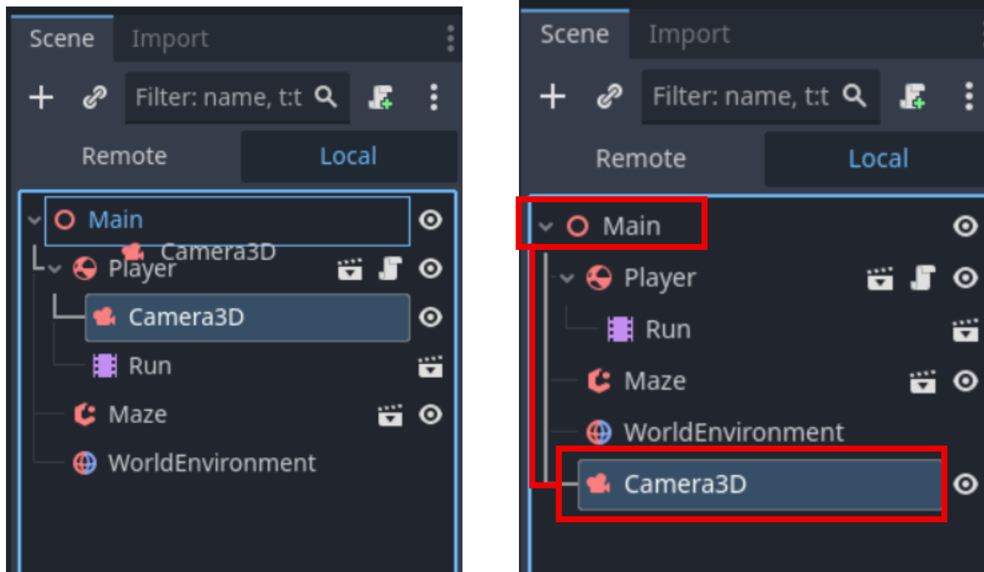
56 Playtest the game. What happens when Codey moves?

Since **Camera3D** is a child node to **Player**, whenever Codey rotates, the camera does too! If only there was some way to send only the position data of **Player** to **Camera3D**...

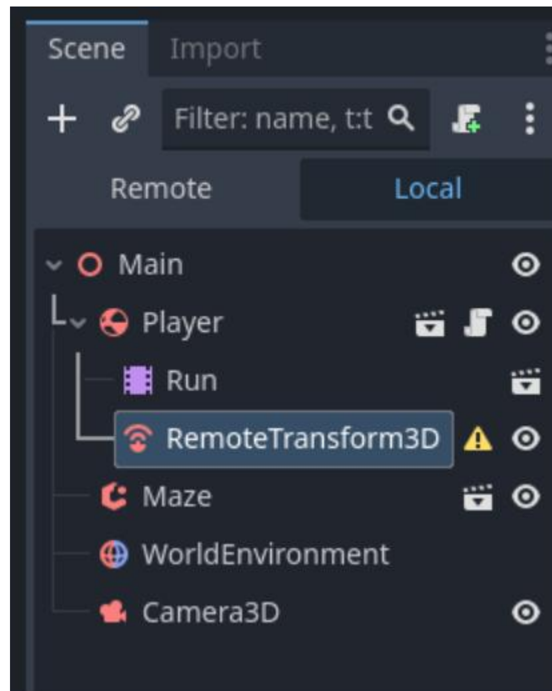
Oh wait, there is! It's another node called **RemoteTransform3D**. This will help the camera follow Codey without rotating.



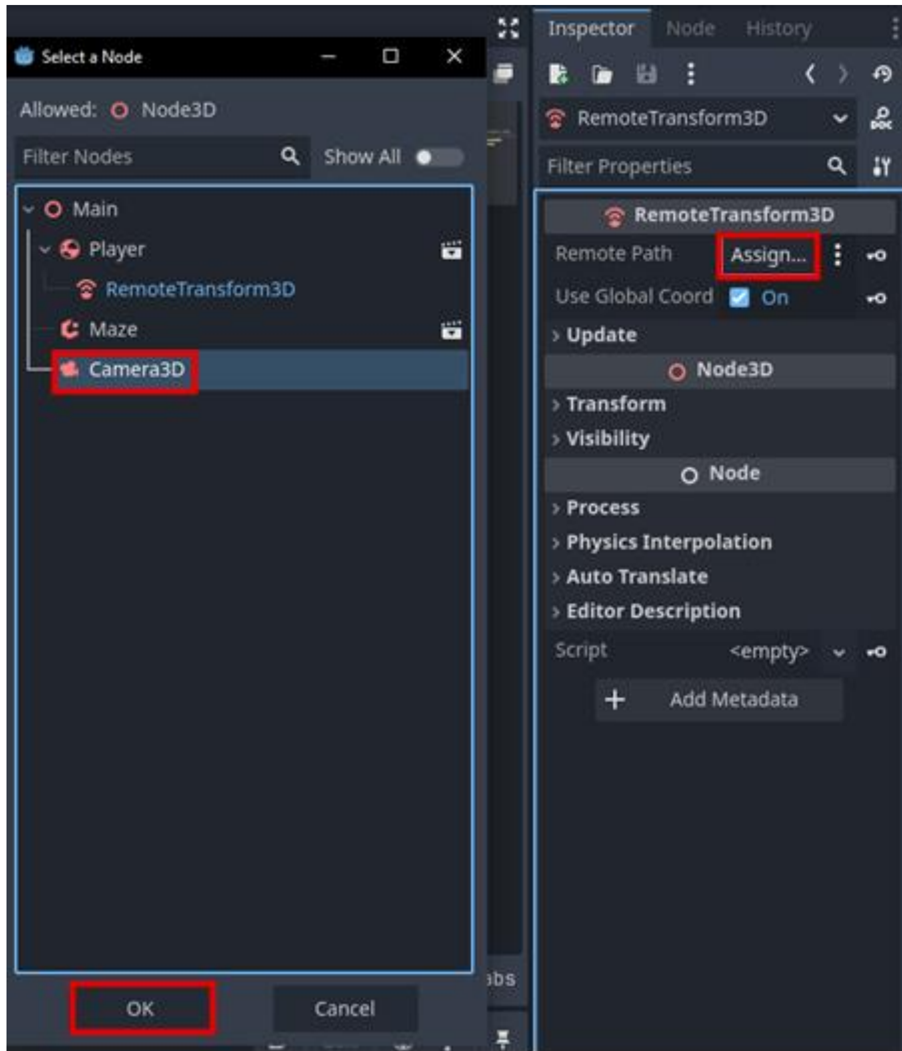
57 Drag the **Camera3D** node out of **Player** and onto **Main** so that **Player** and **Camera3D** are sibling nodes.



58 Add a **RemoteTransform3D** node as a child to **Player**.

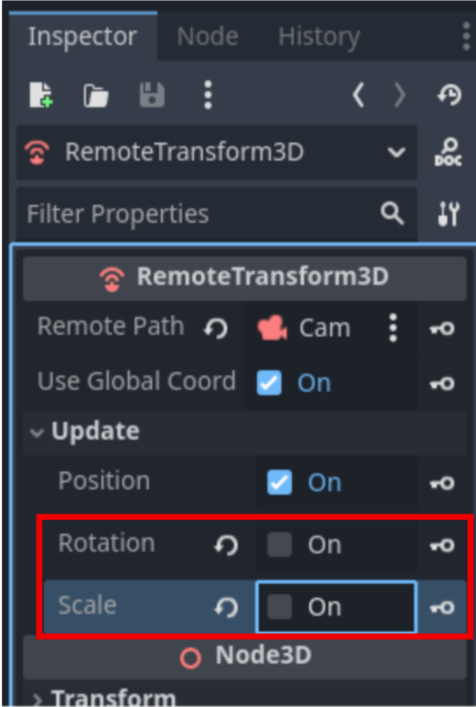


59 In the **Inspector**, find **Remote Path**. Select **Assign...** and choose **Camera3D**.

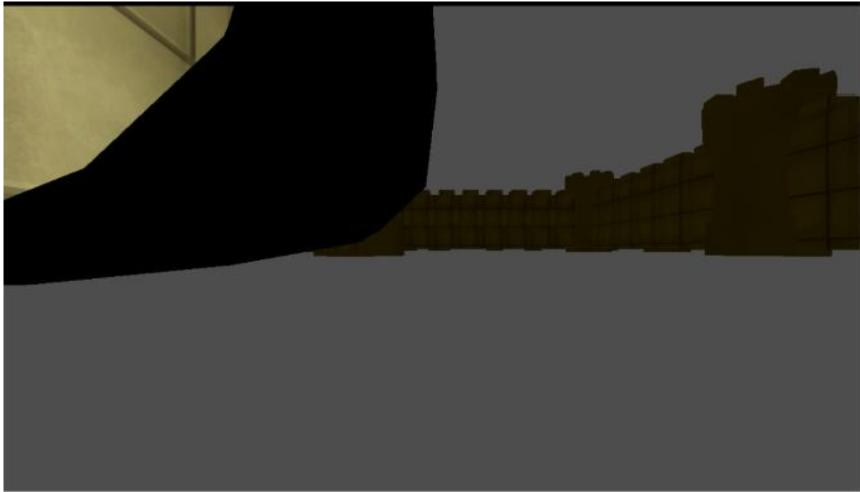


60 Open the **Update** drop-down menu and disable **Rotation** and **Scale**.

Make sure **Position** is kept enabled, otherwise the Camera will always stay in place!

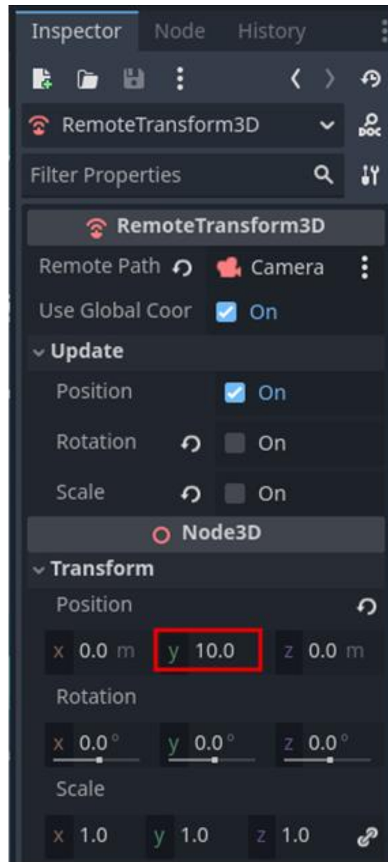


61 Playtest the game. Observe what the camera displays on the viewport.

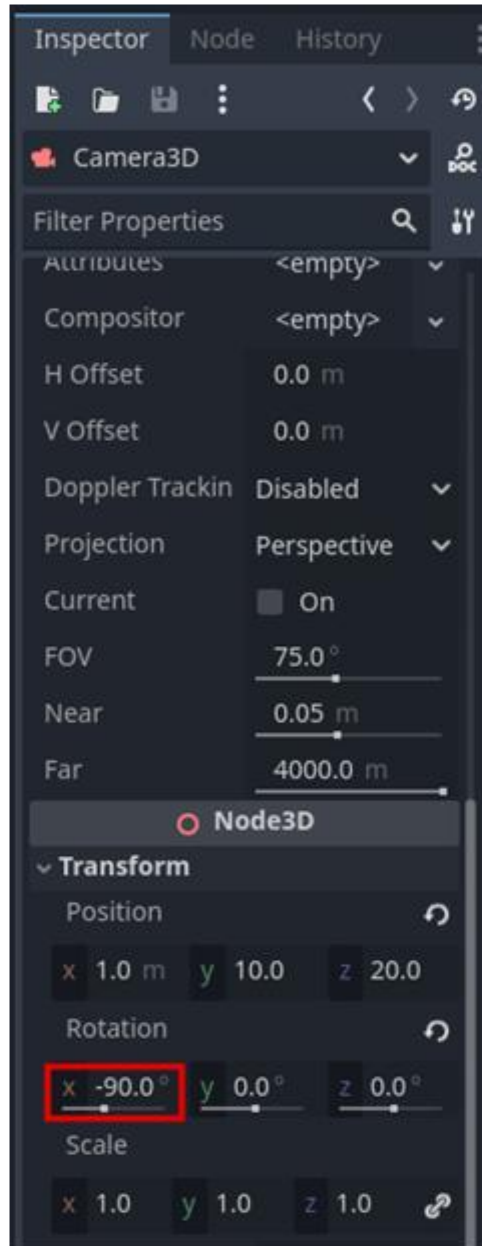


Uh oh, since the **RemoteTransform3D** sent its rotation information to the **Camera3D** the moment it was created, the camera was automatically rotated to face forwards. Additionally, the **RemoteTransform3D** is located at the bottom of Codey. Let's fix both issues.

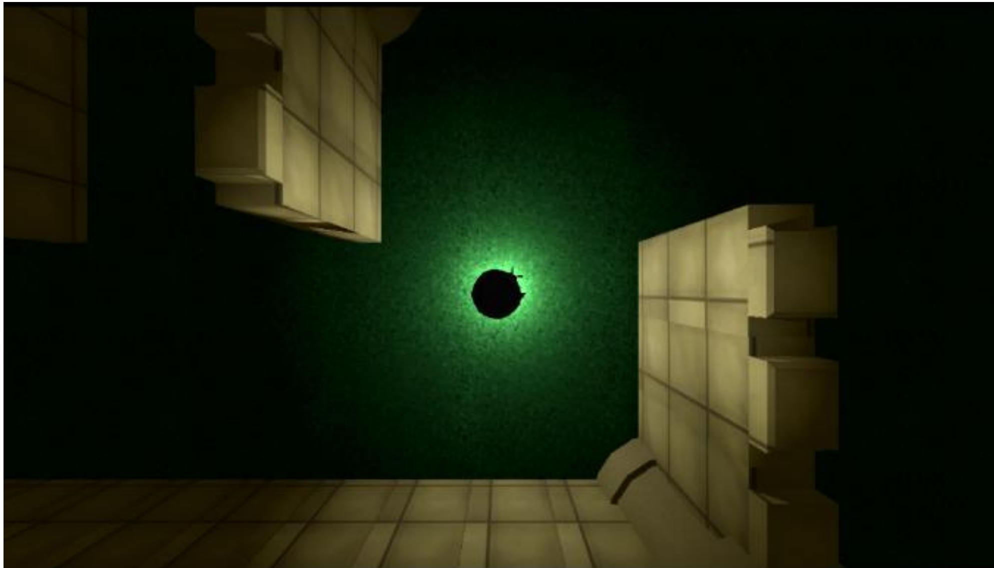
62 With **RemoteTransform3D** selected, in the **Inspector** open the **Transform** drop-down menu and set the y-position to **10**. This is how high the camera was at the start of the project.



63 Open the **Inspector** for the **Camera3D** and navigate to the **Transform** drop-down menu. Set the x-rotation to **-90** so the camera points down again.



64 Playtest the game. Codey should now be able to turn in any direction and the camera will not rotate concurrently!



Pause for **Sensei Stop #6**!

Check in with a Code Sensei before moving on. Make sure that Codey rotates when running around and the camera doesn't rotate with Codey.

Reminder: Save your work!

65 Currently, Codey will suddenly snap into the direction of movement. There happens to be a way to smooth out the transition between directions, and it's a method called `lerp_angle()`.

`lerp_angle()`**:** computes the angle value between the given from and to values given by weight

Parameters:

1. `from` ***(int)*****:** the start angle
2. `to` ***(int)*****:** the destination angle
3. `weight` ***(float)*****:** decimal (0<weight<1) representing percentage traveled between **from** and **to**

Returns *(float)*: The angle value that is (weight*100) percent of the way between **from** and **to**, starting at **from**.

Reminder:

Linear Int**erp**olation, or **lerp**, is a way to blend two values together and is often used for moving platforms, animations, and transitions.

66 At the top of the **player_controls** script, declare a `turn_rate` export variable of type `float`, and set its default value to `0.1`.

```
1   extends RigidBody3D
2
3   @export var move_speed: float = 10.0
4   @export var turn_rate: float = 0.1
5
6   @onready var anim_player: AnimationPlayer = $Run
7
8   func _physics_process(delta: float) -> void:
9       #linear_velocity = Vector3.FORWARD * move_spe
10      var horizontal: float = Input.get_axis("move_
```

67 Inside the `_physics_process()` method and inside the if statement, **remove** the line of code setting `rotation.y` to the `atan2()` call. In its place, create a variable of type `float` named `target_angle` that stores the result of the `atan2()` call.

```
15  if input_dir != Vector3.ZERO:
16      anim_player.play("Run")
17      var target_angle: float = atan2(input_dir.x, input_dir.z)
18  else:
19      anim_player.stop()
```

68 Set `rotation.y` to the result of a call to `lerp_angle()` with parameters `rotation.y`, `target_angle`, and `turn_rate`.

```
15  if input_dir != Vector3.ZERO:
16      anim_player.play("Run")
17      var target_angle: float = atan2(input_dir.x, input_dir.z)
18      rotation.y = lerp_angle()
19  else:        float lerp_angle(from: float, to: float, weight: float)
20      anim_player.stop()
```

Reminder:

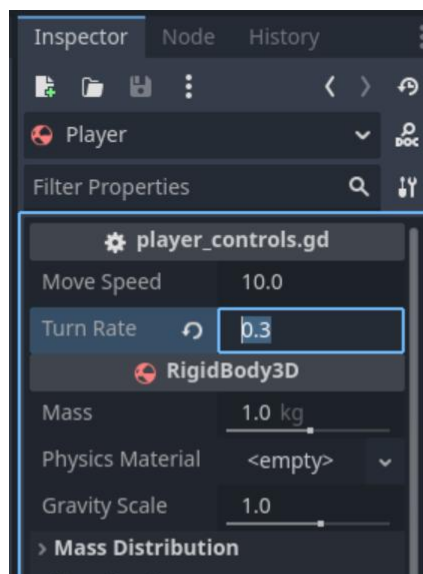
Placing a property before an assignment operator = will tell Godot to set it to whatever follows. Placing a property anywhere else will ask Godot to retrieve its current value!

69 Check the code! Update the script as needed.

```
15      if input_dir != Vector3.ZERO:
16          anim_player.play("Run")
17          var target_angle: float = atan2(input_dir.x, input_dir.z)
18          rotation.y = lerp_angle(rotation.y, target_angle, turn_rate)
19      else:
20          anim_player.stop()
```

70 Playtest the game. Codey should now gradually ease into the current direction of movement!

71 In the **Player's Inspector**, experiment with **Turn Rate** to see different outcomes! Remember, its effective values range between 0 and 1.



Pro Tip:

Why does `lerp_angle()` smoothly change Codey's rotation instead of linearly changing it? This is because of a cheap trick the code is using in this scenario!

The `from` is the *current rotation*, and the `to` is the *target rotation*. Every frame, the code computes the angle 10% (if Turn Rate = 0.1) of the way between `from` and `to` and sets the new *current rotation* to that angle. The next frame, it uses the new *current rotation* computed in the last frame to get 10% closer to `to`. Over 10 frames, instead of fully interpolating between `from` and `to`, Codey only rotates (1-(0.9^10)) = 0.652... = 65.2% of the way between `from` and `to`, not 100%!

This means that Codey never actually faces exactly in the direction of movement, instead Codey will get infinitely close to it.

Pause for **Sensei Stop #7**!

Congratulations on creating your first maze game with RigidBody controls in Godot! Great job!

Before submitting, check in with a Code Sensei to make sure the player gradually turns towards the direction of movement then reflect on the following:

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

Reminder: Save your work!