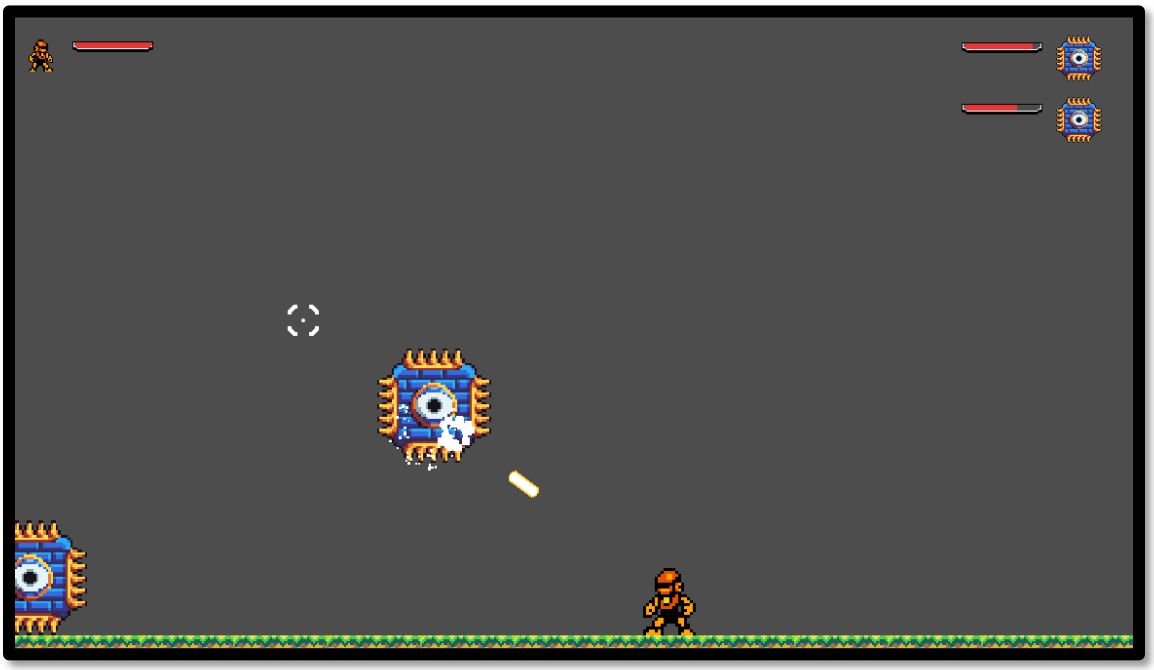
CODE NINJAS®

Silver Belt Ninja Guide
Activity 01: Robomania

Activity 01: Robomania

In this activity, you will create a futuristic robot game where the hero Roboman must overcome two evil crushers to save the day! This project will show you how to create an NPC's (non-playable character) movement.

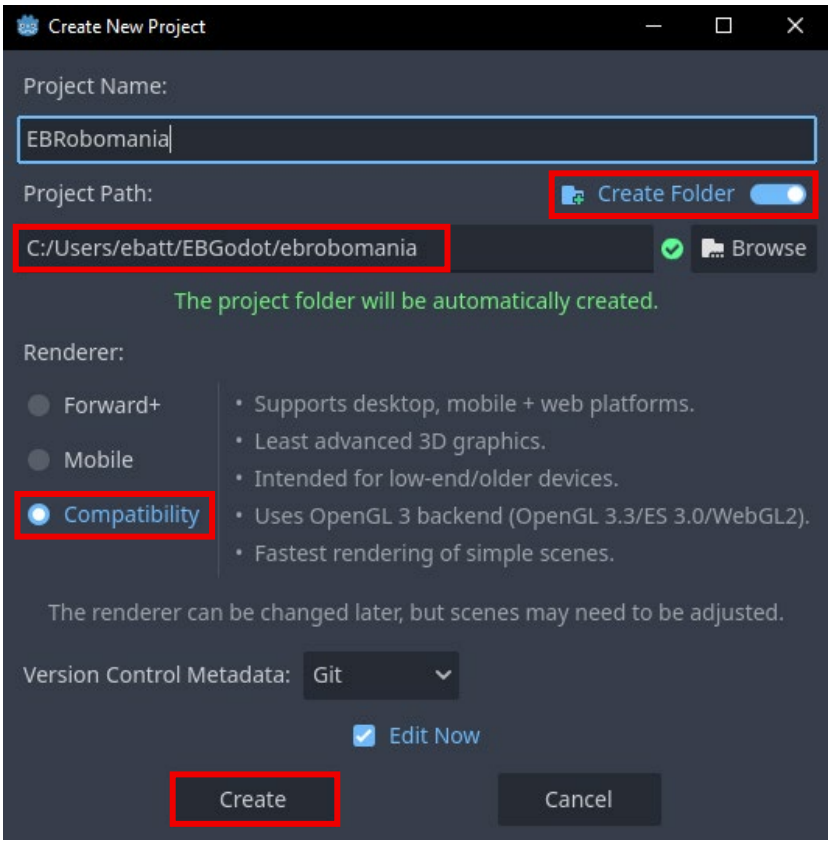
By the end of this activity, you will have explored how to program NPC movement, create physics materials, as well as how to use the power of RigidBody nodes.



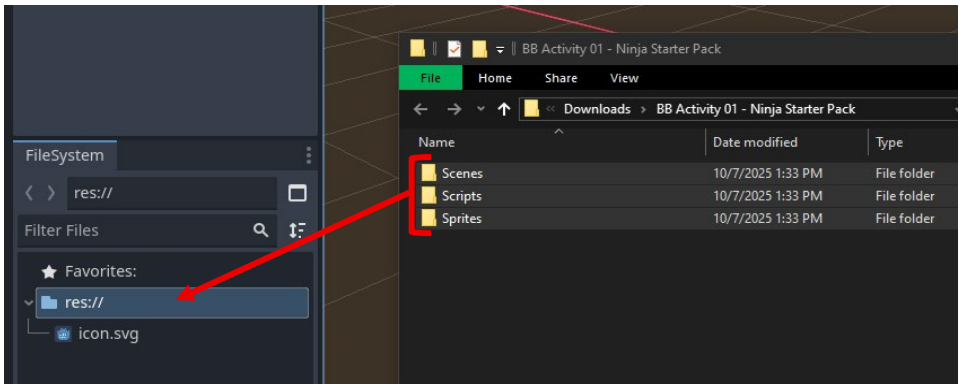
1 After opening Godot, in the top left corner select **+ Create**.

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

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**.

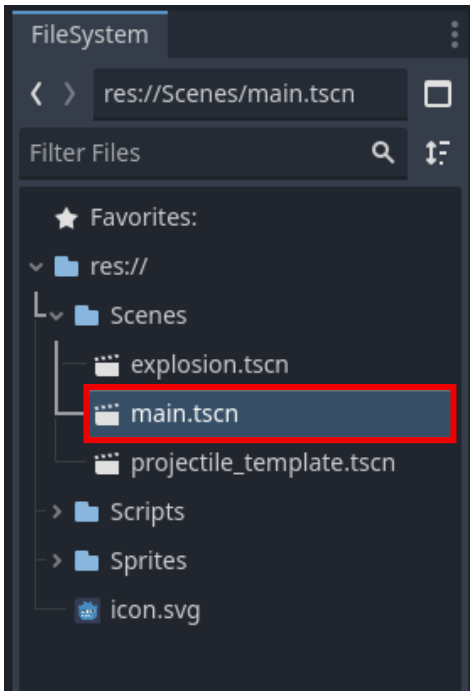


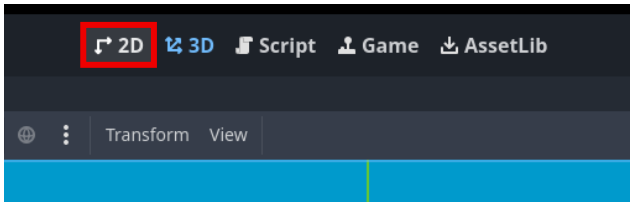
2 Import the **SB Activity 01 - Ninja Starter Pack** by dragging the contents from the zip folder into **res://** in the **FileSystem**.



3

In **FileSystem**, navigate to the **Scenes** folder and double-click to open **main.tscn**. Select **2D** in the workspace selector.

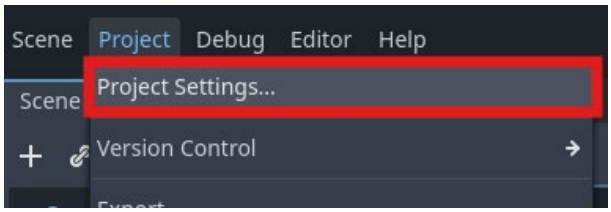




4

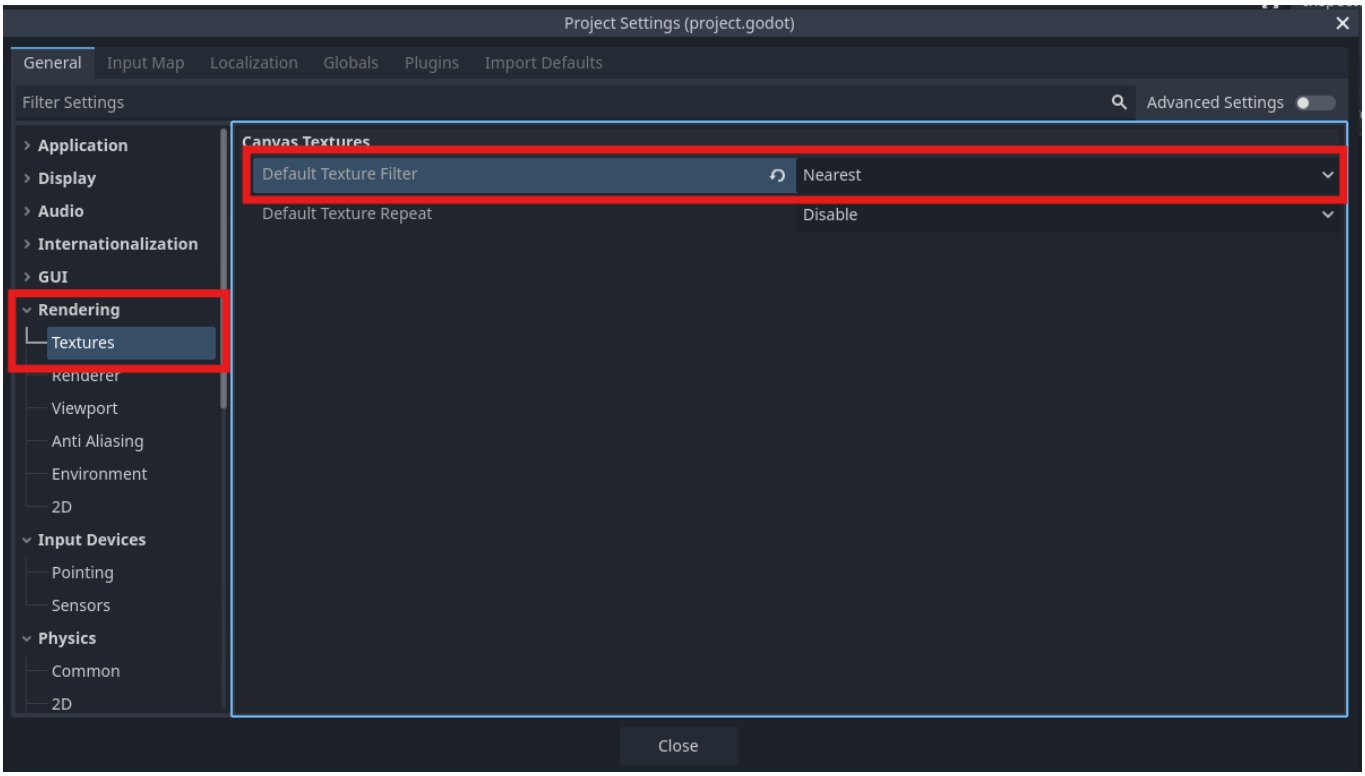
Oh no! The pixel art appears blurry! This is because by default, Godot tries to smooth out 2D art. This works well for many games, but not for pixel art.

To fix this, the texture filter mode will have to be changed. Open **Project** then click **Project Settings**.



5 In **Project Settings**, navigate to **General -> Rendering -> Textures** and change **Default Texture Filter** to **Nearest.** Click **Close**.

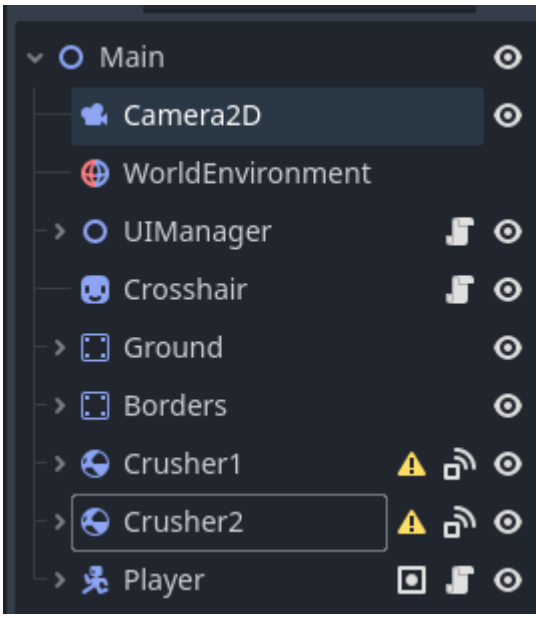
Now the sprites are looking sharp!



6 Review the nodes already in the main scene.

Note the major objects:

- **Crusher1 & Crusher2 –** These will be the two enemies in the game. What do you notice about each?
- **Player –** The main player of the game with animations and collisions.
- **Crosshair –** A UI Sprite that will follow the cursor during the game to show where the player is aiming.
- **UIManager** – All of the UI for the game, including the health bars, win and lose screens, and a flawless screen!

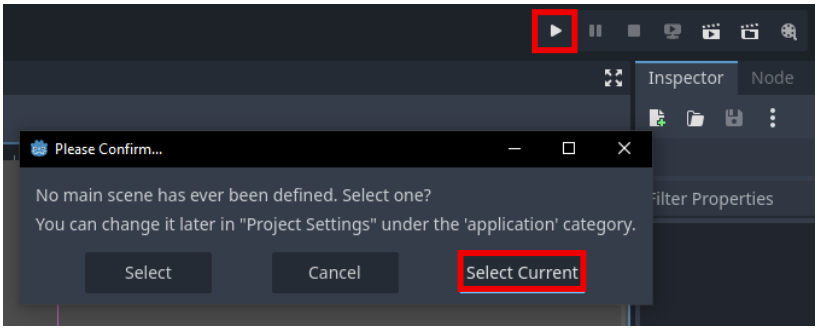


Reminder:

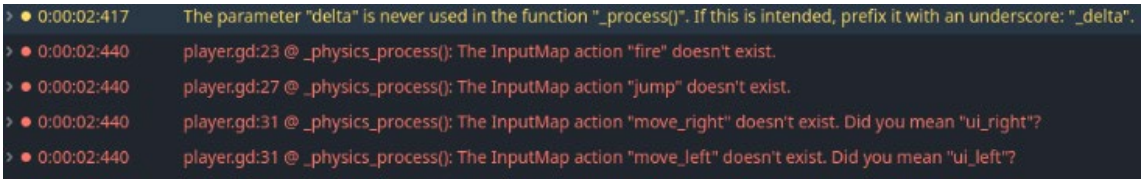
Click on the arrow next to each node to expand or condense their child nodes.

7 Press **play** and click **Select Current** to define the main scene. Test the game to see what is already completed!

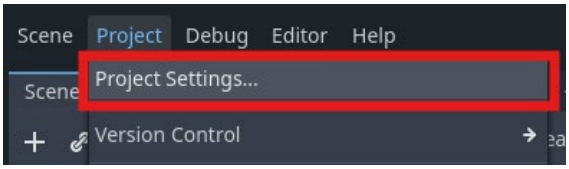
Close the playtest window.

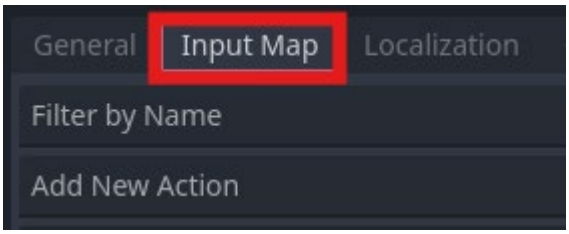


8 Oh no, there are errors in the Debugger! The errors are appearing because the input has not been set up yet.



9 Open **Project Settings** and go to the **Input Map** tab.





10 Add the following actions to create custom inputs in the game:

- **fire** – Left Mouse Button
- **jump** – Space
- **move_left** – Left and A
- **move_right** – Right and D

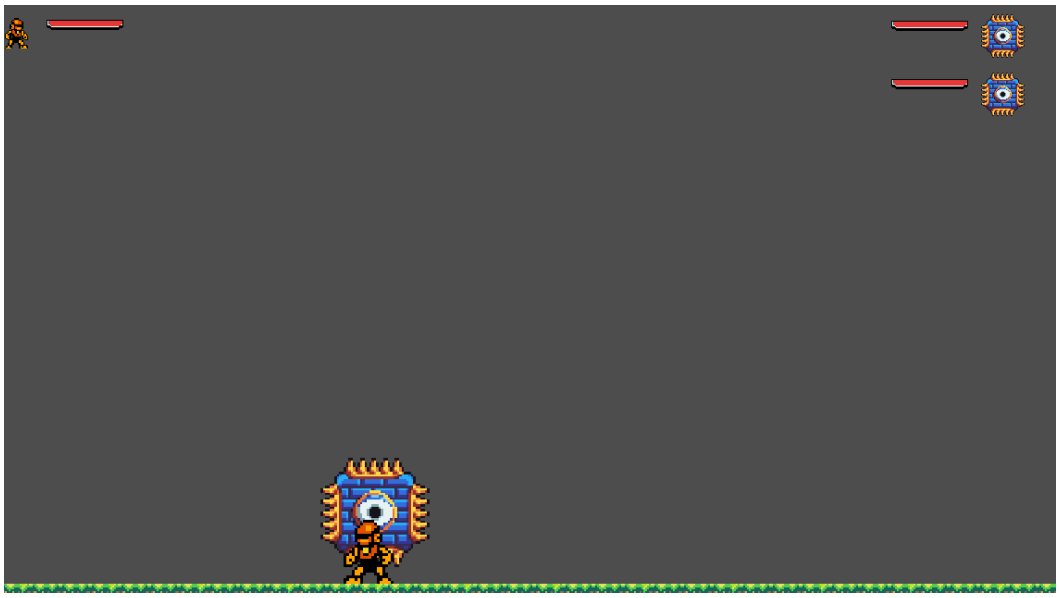
Reminder:

Use the **Add New Action** field to create new inputs, then use **+** to attach the user's input.

11 Playtest the game again to test out the new inputs.

Move **RoboMan** with the **A & D** or **Left & Right** direction buttons. Press the **space bar** to make **RoboMan** jump and fire the ProtonBlaster by aiming the cursor and clicking the **left mouse button.**

Notice there is only one enemy crusher. Later, the other crusher will be added to the game! Try to destroy the crusher! Currently, the crusher just floats on the screen. What happens when the **ProtonBlaster** is fired at the crusher and when **RoboMan** runs into the crusher?

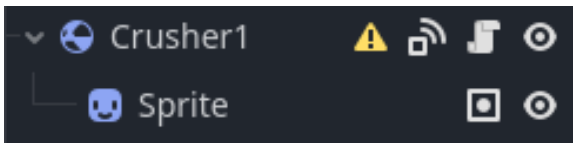


Close the playtest window.

12

Currently, the crusher does not do anything!

What could be done to get the crusher to respond to **RoboMan** and the **ProtonBlaster**?



Think back to previous games where nodes needed to collide with other nodes.

Reminder:

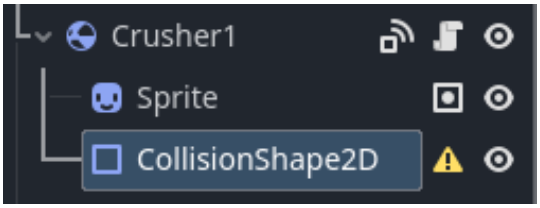
Click on the hazard symbol next **Crusher1**.

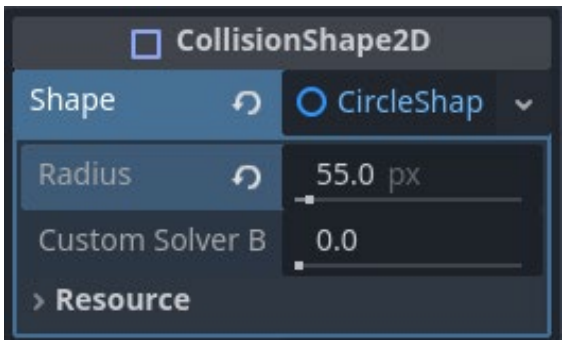
13

A collision shape needs to be added for **Crusher1** to be able to interact with other nodes!

In **Scene**, add a **CollisionShape2D** node as a child to **Crusher1**.

Add a new **CircleShape2D** as the shape. Select **CircleShape2D** and change its radius to **55.0 px**.

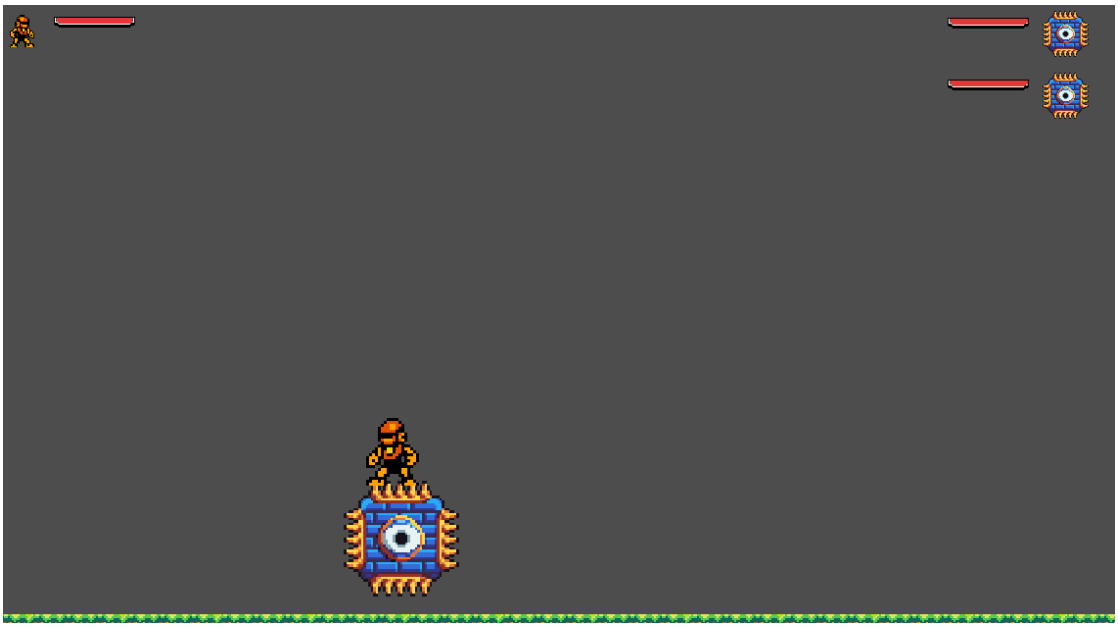




14 Playtest the project again.

How has adding the collision shape changed the game? What happens when the **ProtonBlaster** is fired at the crusher and when **RoboMan** runs into the crusher?

Close the playtest window.



15 While the player can interact with the crusher, the projectiles go right through them! This is because the projectiles are on a different **collision layer**.

Collision layers allow developers to tell objects what they are supposed to collide with. The projectile is on a different layer, so it doesn't collide with the crusher!

Open the **Inspector** for **Crusher1**. Navigate to **CollisionObject2D** and open the **Collision** drop-down menu. In the **Layer** section, uncheck **1** and then select **4**. Set the **Mask** to **1**.

16 Playtest the project again. How has adding the collision layers changed the game? What happens when the **ProtonBlaster** is fired at the crusher?

Close the playtest window.

Pause for **Sensei Stop #1**!

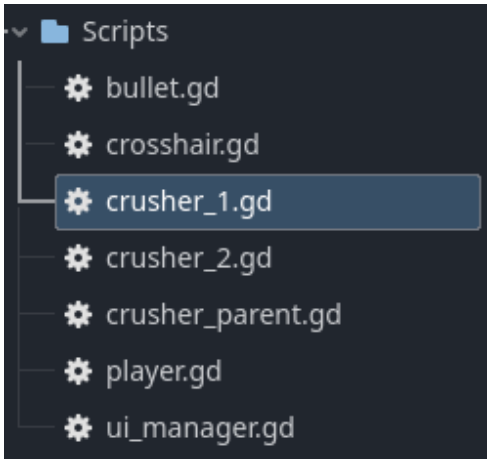
Check in with a Code Sensei before moving on. Ensure the **Roboman** takes damage when touching the **Crusher** and the **ProtonBlaster** successfully hits the **Crusher**.

Reminder: Save your work!

17

How could the game be made more challenging? What if... the crusher could move?!

In **FileSystem**, open the **crusher_1** script. Notice that the script is already extending a parent script that includes functionality for damage.



18

Start by defining a `_physics_process()` method.

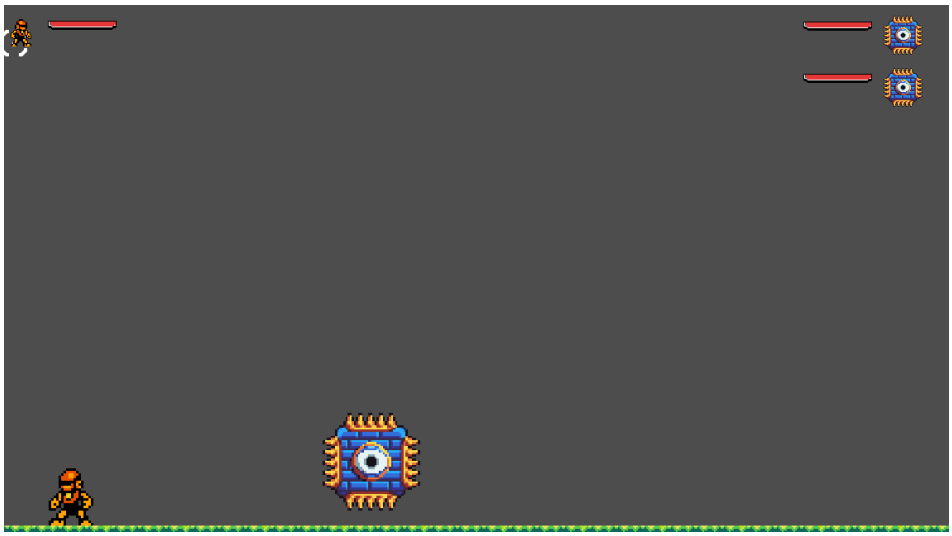
Inside the `_physics_process()` method, update the Crusher's `position`. Add the current position and `Vector2.RIGHT`, multiplied by `speed` and `delta` (the seconds between the last update).

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   func _physics_process(delta: float) -> void:
4       position = position + Vector2.RIGHT * speed * delta
5
```

Reminder:

`_physics_process()` is a special built-in Godot method that runs 60 times per second.

19 Playtest the project. Which direction is the crusher moving in?



20 Currently **Crusher1** is moving to the right, but it should also move to the left at some point!

Set the direction of the `Vector2` in the script to be `LEFT` instead of `RIGHT`.

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   func _physics_process(delta: float) -> void:
4       position = position + Vector2.LEFT * speed * delta
5
```

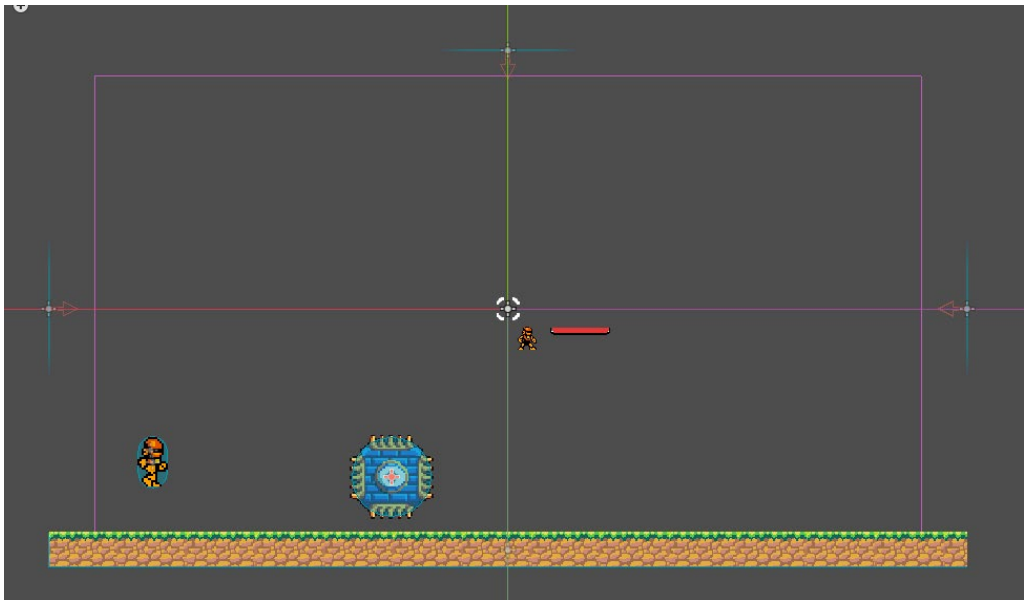
21 Playtest the game again to see if **Crusher1** moves to the left.

What happens when **Crusher1** reaches the edge of the screen?

22 Hmm, **Crusher1** seems to fly right off the screen!

Currently the **main scene** is set up so that the **Camera2D** node is at the center of the scene with an **x position** of **0.**

The **LeftWorldBoundary** has an **x position** of **-550** and the **RightWorldBoundary** has an **x position** of **550.**



Reminder:

WorldBoundary nodes are colliders on the edge of the screen where no colliders (with physics) can pass.

23

If **Crusher1** goes off the edge of the screen, the direction will have to change. Because direction will be changing in the middle of the game, use a variable to update its value.

At the top of the **Crusher1** script, create a `direction` variable of type `Vector2` and set it to `Vector2.LEFT` by default.

Replace `Vector2.LEFT` in the code that sets the Crusher's position with the new `direction` variable.

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   var direction: Vector2 = Vector2.LEFT
4
5   func _physics_process(delta: float) -> void:
6       position = position + direction * speed * delta
7
```

24

To keep **Crusher1** within the boundaries of the game, there needs to be a check to see when a **WorldBoundary** is hit.

At the start of the `_physics_process()` method, before the `position =` line, add an `if`-statement that checks to see if the **x position** of **Crusher1** is **less than** the **x position** of the **LeftWorldBoundary.**

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   var direction: Vector2 = Vector2.LEFT
4
5   func _physics_process(delta: float) -> void:
6       if position.x < -550:
7
8
9       position = position + Vector2.LEFT * speed * delta
10
```

Reminder:

To view the **x position** of the **LeftWorldBoundary** node navigate to **Inspector > Transform > Position.**

25

Now, whenever **Crusher1** hits the left side of the screen, the code inside this `if`-statement will execute.

What could be changed about the `direction` of **Crusher1** to get it to move in the *opposite* direction?

To reverse the direction, `direction` needs to be changed to `Vector2.RIGHT`!

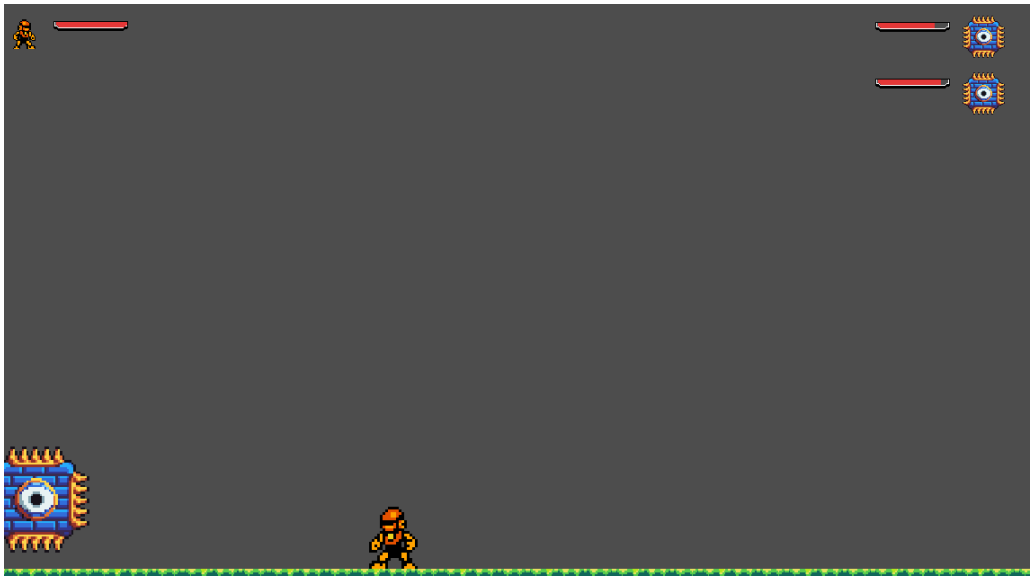
Inside the `if`-statement, update the value of `direction` to `Vector2.RIGHT`.

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   var direction: Vector2 = Vector2.LEFT
4
5   func _physics_process(delta: float) -> void:
6       if position.x < -550:
7           direction = Vector2.RIGHT
8
9
10      position = position + direction * speed * delta
11
```

26

Playtest the game and let **Crusher1** move across the screen.

Does **Crusher1** bounce off the left side of the screen? What about the right side?



27

Oh no, **Crusher1** still flies off the right side of the screen!

Navigate back to the **crusher1** script. Notice the `if`-statement only checks when the **x position** of **Crusher1** is at the **x position** of the **LeftWorldBoundary**!

Add another `if`-statement. This statement should check if the x position is greater than the **RightWorldBoundary's** x position, and if so, set `direction` to `Vector2.LEFT`.

```
1   extends "res://Scripts/crusher_parent.gd"
2
3   var direction: Vector2 = Vector2.LEFT
4
5   func _physics_process(delta: float) -> void:
6       if position.x < -550:
7           direction = Vector2.RIGHT
8
9       if position.x > 550:
10          direction = Vector2.LEFT
11
12      position = position + direction * speed * delta
13
```

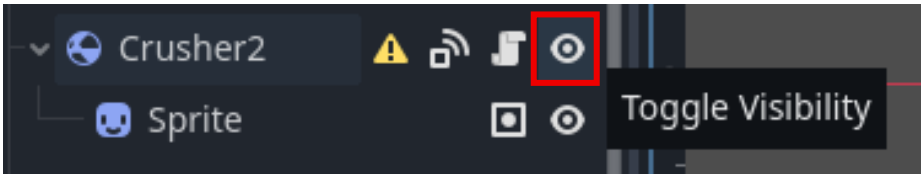
Pause for **Sensei Stop #2**!

Check in with a Code Sensei before moving on. Ensure the **Crusher** bounces off the left and right sides of the screen.

Reminder: Save your work!

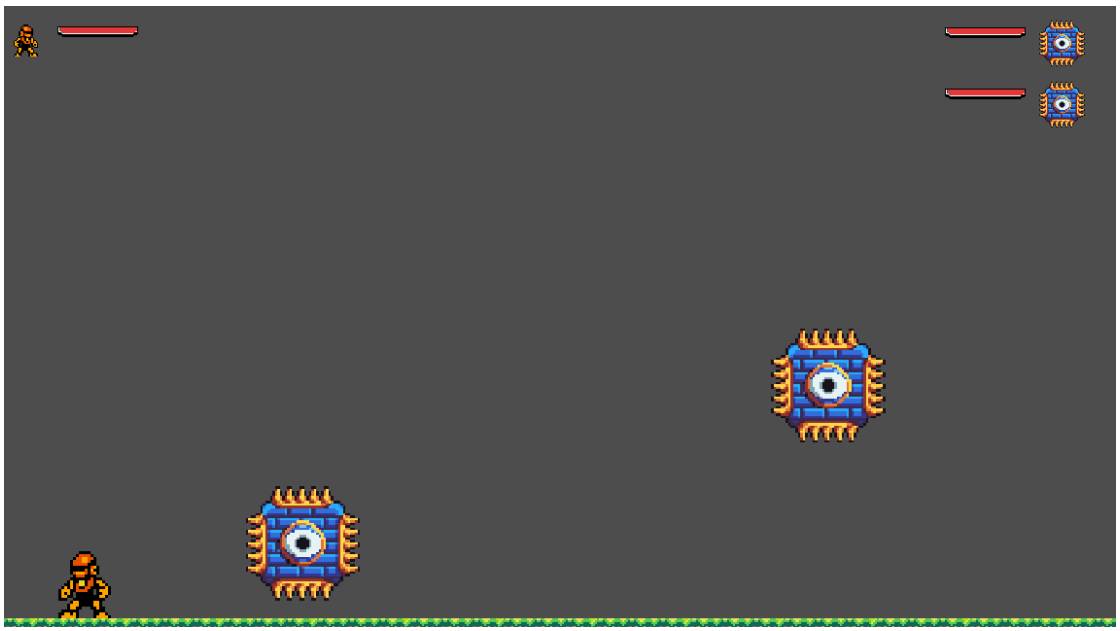
28 Now is an important time to explore how **RigidBodies** can save lots of time when creating NPC movement.

In **Scene,** notice the **closed eye symbol** next to the **Crusher2** node. Click the **eye symbol** to toggle the node's **visibility** to **on.**



29 Playtest the project and see there are now two crushers in the game!

Does **Crusher2** move around like **Crusher1**?



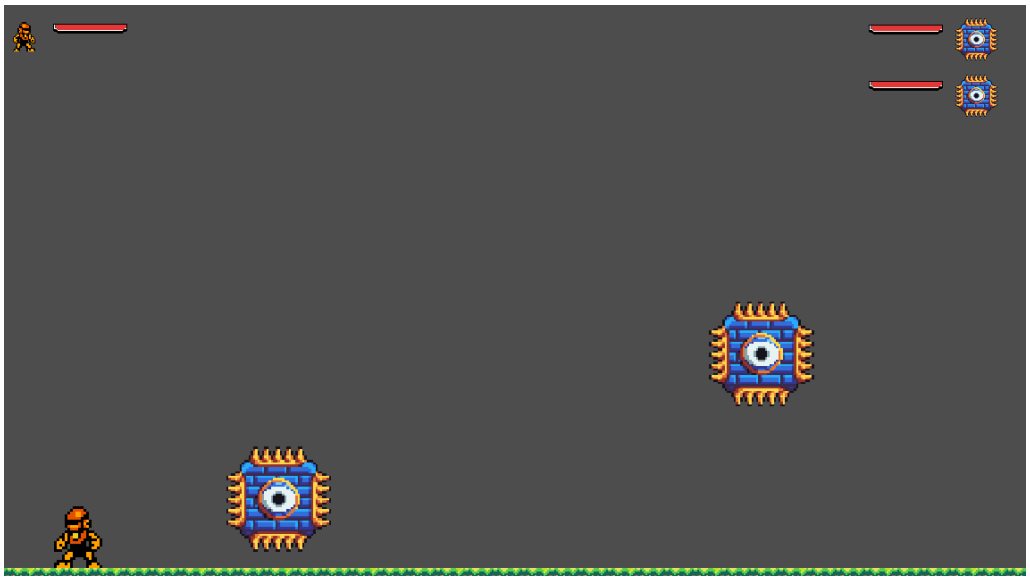
30 **Crusher2** also needs a collider and layers!

Set up **Crusher2** the same way as **Crusher1**, with the following:

- ☐ CollisionShape2D
- ☐ CircleShape2D with radius of 55.0 px
- ☐ Layer 4, Mask of 1

Refer to **steps 13-15** for additional guidance.

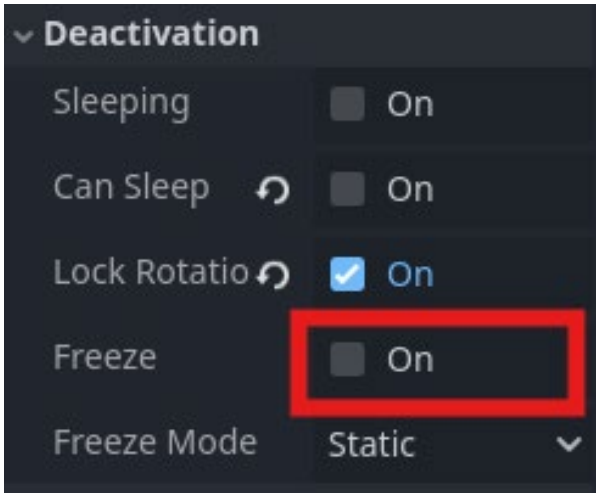
31 Playtest the game again. How does **Crusher2** behave?



Close the playtest window.

32 Currently, physics on **Crusher2** are disabled! This is stopping the crusher from moving, and from objects interacting with it (like the projectile).

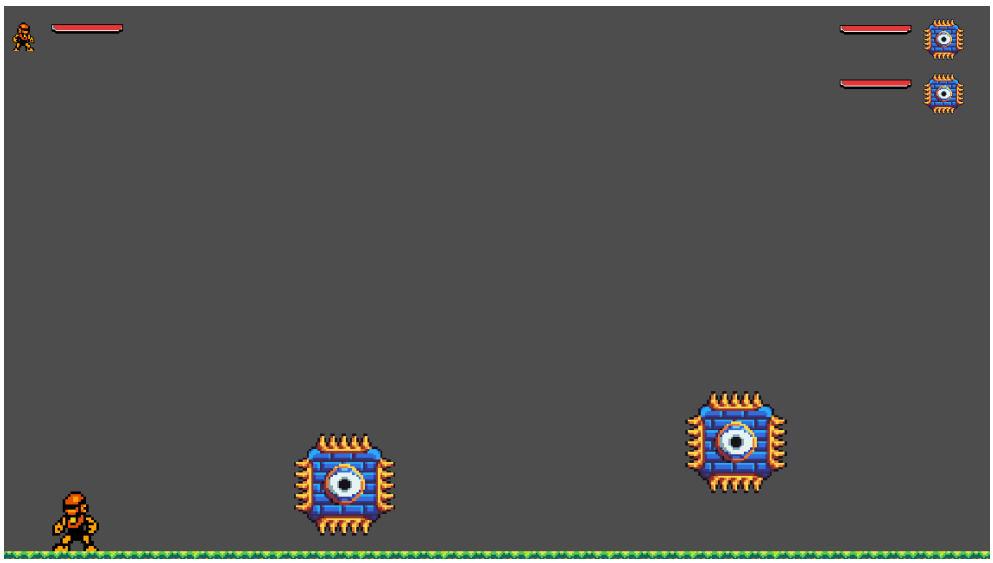
In the **Inspector** for **Crusher2**, under the **Deactivation** drop-down menu, uncheck **Freeze**.



Pro Tip:

Take a look back at **Crusher1**, what type of **Freeze Mode** is it set to? Hover over the "Freeze Mode" text in the Inspector to see its description. Why do you think this allows **Crusher1** to work?

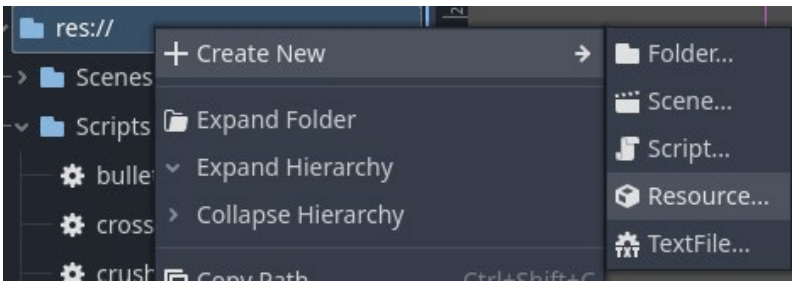
33 Playtest the game again. How does **Crusher2** interact with the projectile and environment?



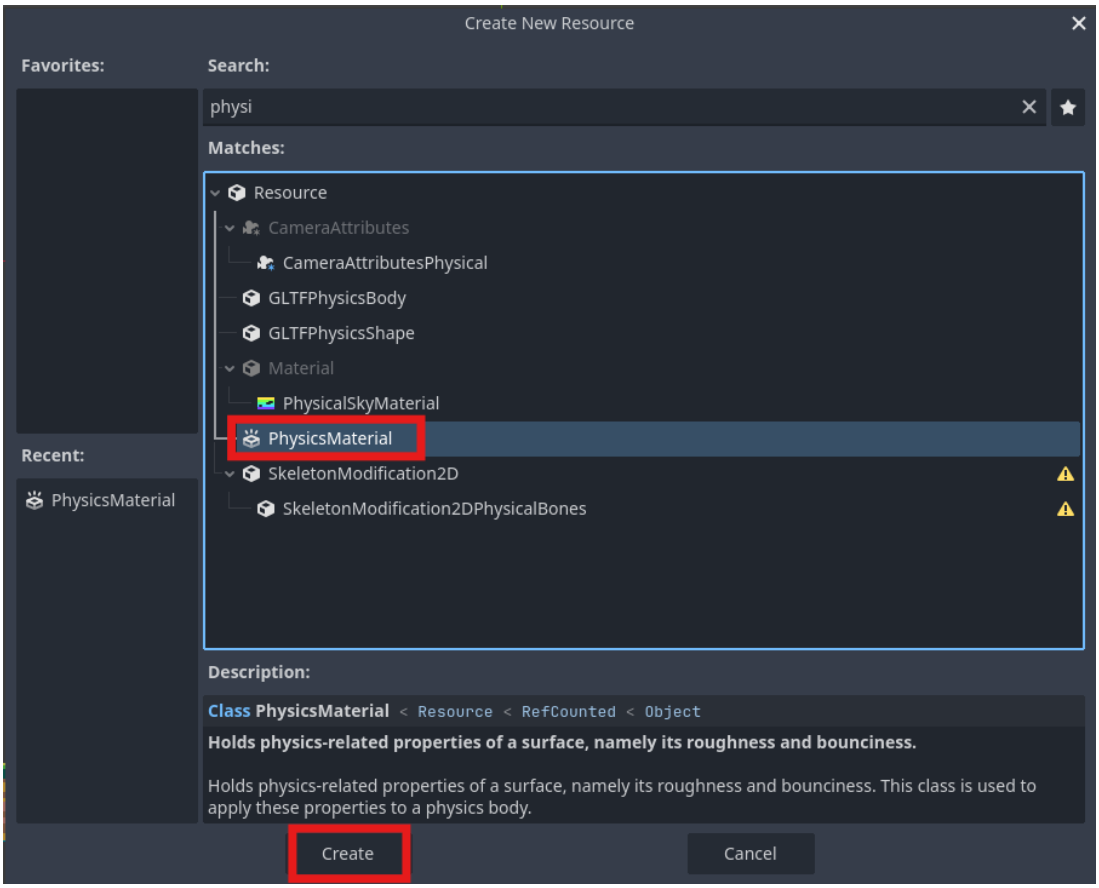
Close the playtest window.

34 To tell Godot how a **RigidBody** node should behave, it needs a **physics material**. This has properties to determine how bouncy a node is and how much friction applies to it.

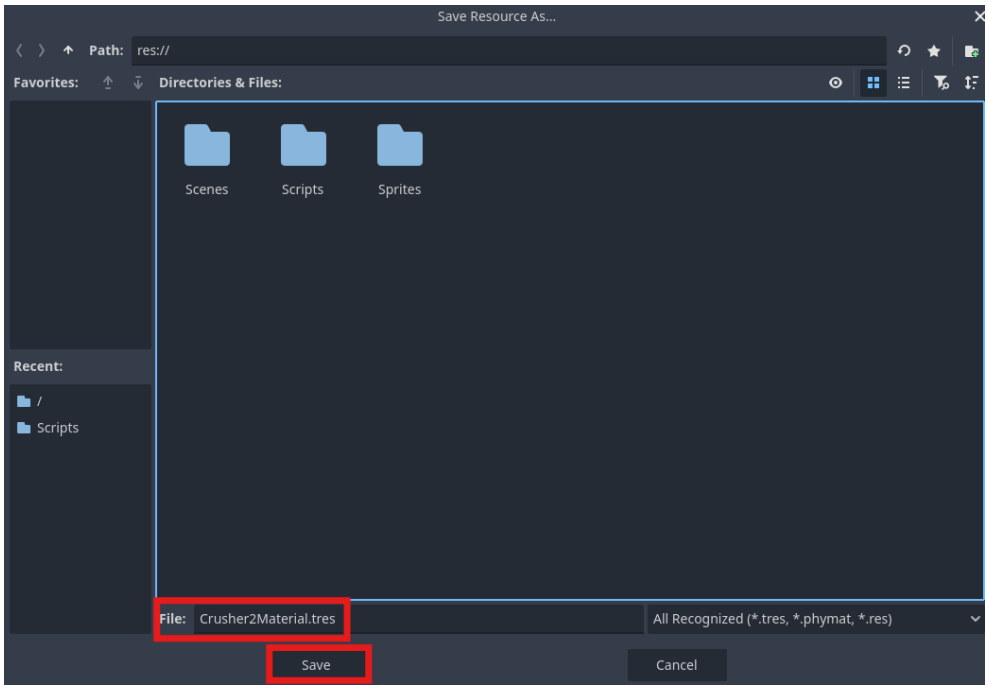
In **FileSystem**, create a new physics material by clicking **res:// > Create New > Resource.**



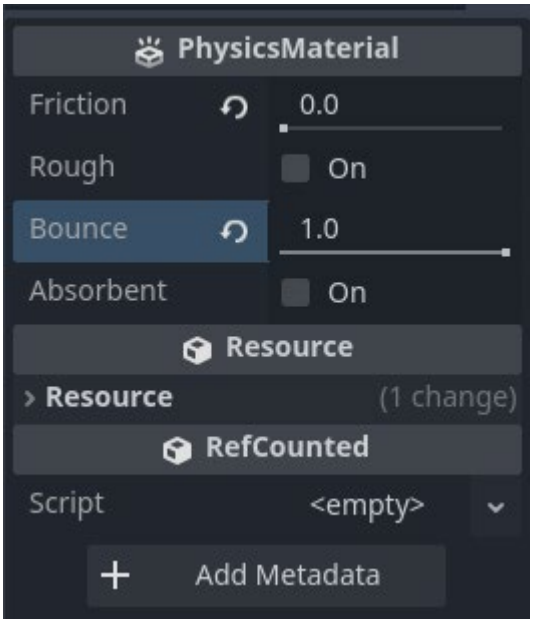
35 Select **PhysicsMaterial** and then click **Create**.



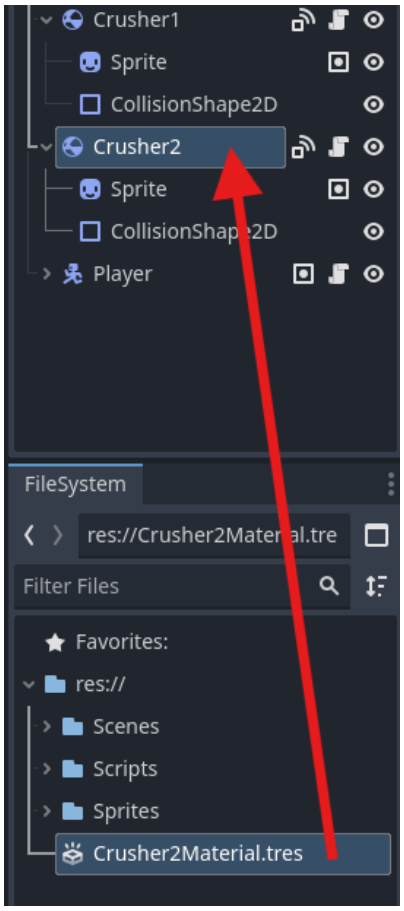
36 Name the resource **Crusher2Material** and select **Save**.



37 In **FileSystem**, double-click to select **Crusher2Material.tres**. In its **Inspector**, set **Friction** to **0** and **Bounce** to **1**.



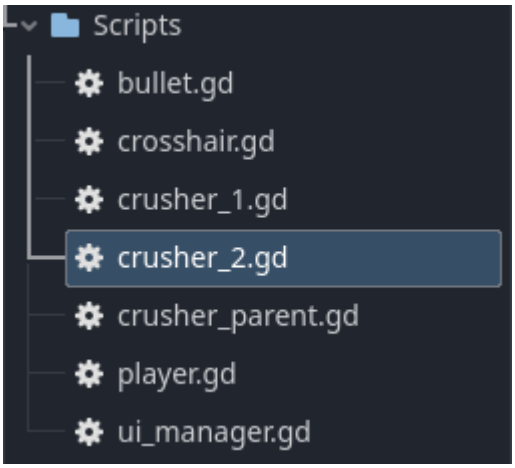
38 **Drag** the physics material onto the **Crusher2** node to apply it to the **RigidBody2D**.



39

Give **Crusher2** a bit of a push.

In **FileSystem**, open the **crusher_2** script in the **Scripts** folder.



40

Add code to the **Crusher2** script to add a bit of force to the right at the start of the game.

Add the `_ready()` method to the script.

Inside the method, use the code completion tool to add the `apply_central_impulse()` method. Pass the value of `Vector2.RIGHT` multiplied by `speed` as the parameter.

The `_ready()` method runs when the game starts. `apply_central_impulse()` adds momentum to the **Rigidbody** in the direction specified.

```
1  extends "res://Scripts/crusher_parent.gd"
2
3  func _ready() -> void:
4      apply_central_impulse(Vector2.RIGHT * speed)
5
```

`apply_central_impulse()`**:** part of the RigidBody2D/3D classes, applies an impulse to the body without affecting its rotation. Impulses are instantaneous changes in the momentum of a body.

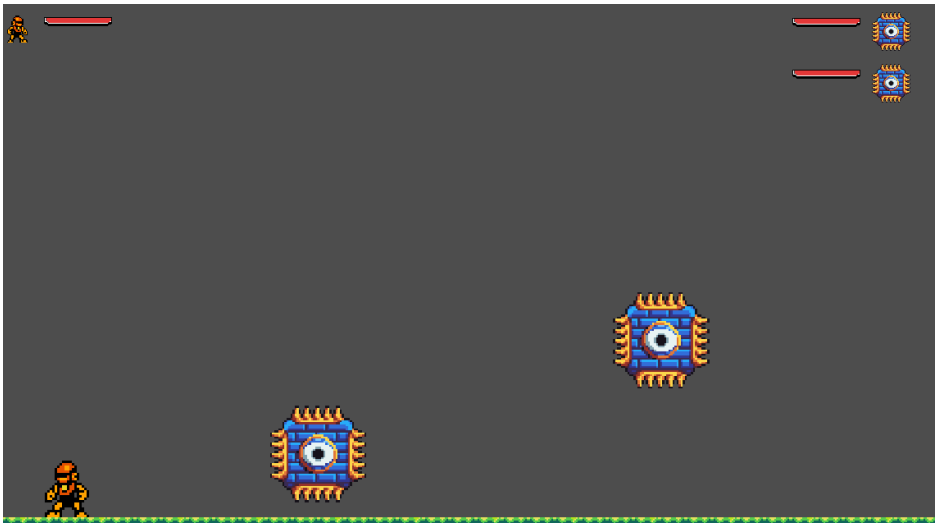
Parameters:

1. `impulse` ***(Vector2/3)*****:** the direction and magnitude of the impulse.

Returns *(void)*: this method is void, it returns nothing.

41 Playtest the game.

Crusher2 should be bouncing all around the screen!



Pause for **Sensei Stop #3**!

Congratulations on creating your first game with collision layers and physics bodies in Godot! Great job!

Before submitting, check in with a Code Sensei to check that **Crusher2** is bouncing around the screen then reflect on the following:

- What did you learn about collision layers and physics materials?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!