CODE NINJAS®

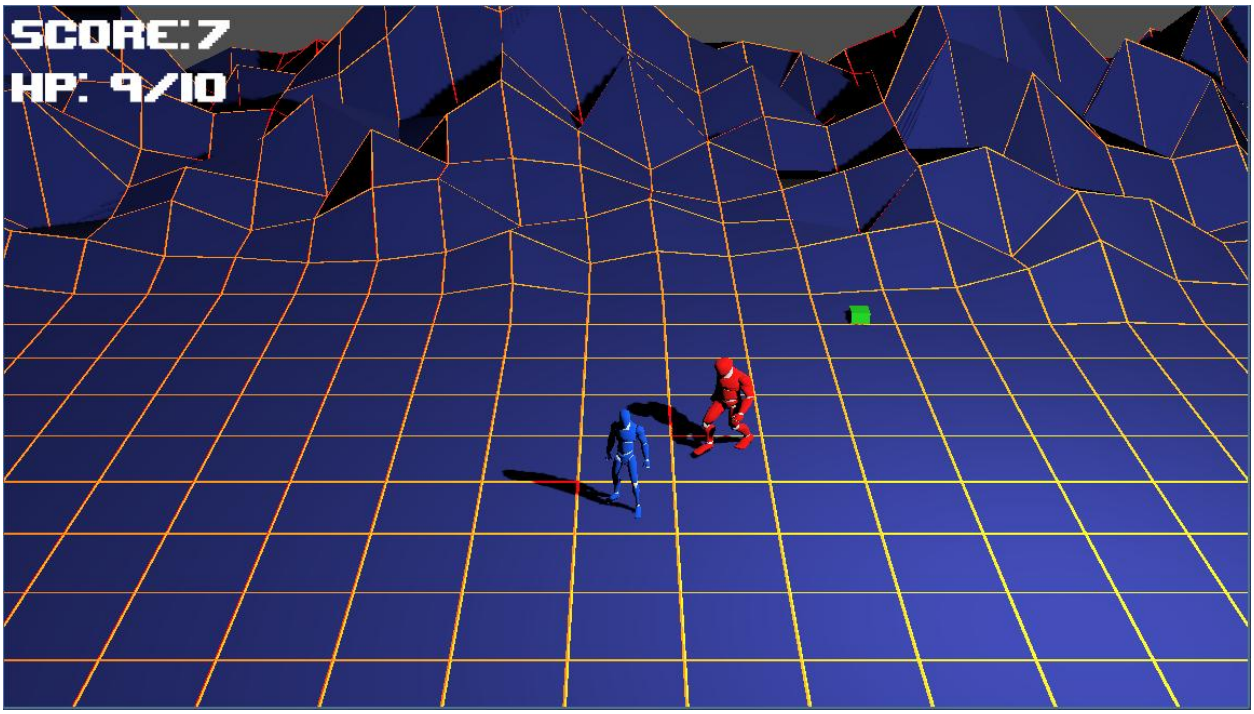
Silver Belt Ninja Guide

Activity 07: Cyber Fu Part 1

Activity 07: Cyber Fu Pt. 1

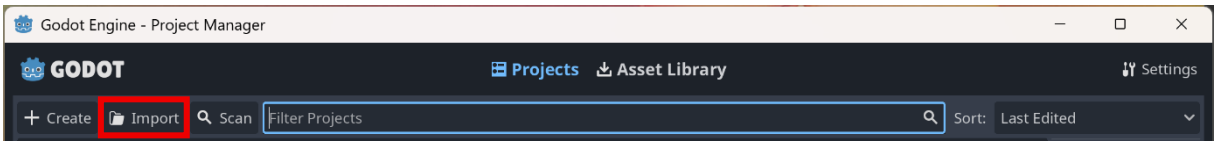
In this activity, you'll learn to train enemies to avoid taking damage. Survive as long as you can!

New mission: Code the enemy to follow and chase the player. The enemy will collide with the player, causing damage. When the player runs out of health, the game is over. Collect health packs to restore missing health.



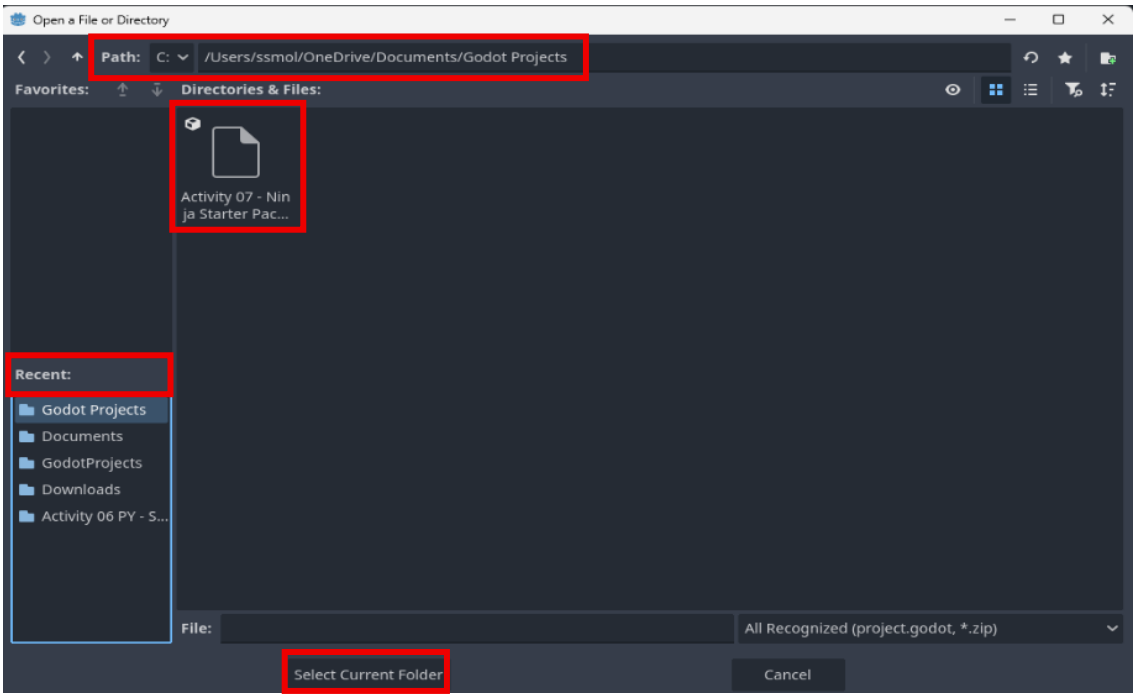
1 Begin the project by **importing** the starter code. This approach maintains the Input Map settings and other properties in the project file.

Open Godot and click **Import.**



2 In the File Directory, navigate to the file path using the **Path** text field or by finding it under **Recent**.

Select **SB Activity 07 – Ninja Starter Pack.zip** and click **Open**.

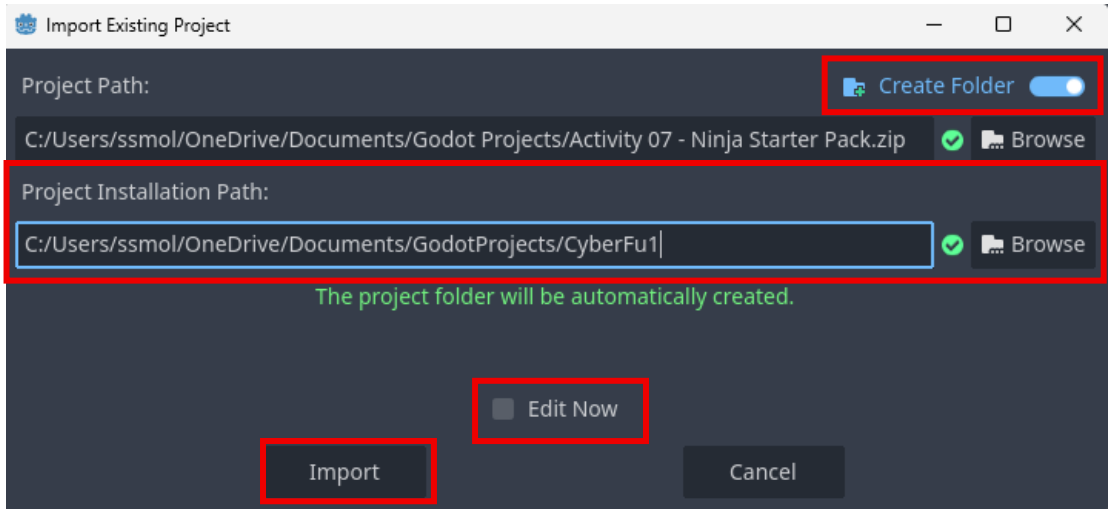


Pro Tip:

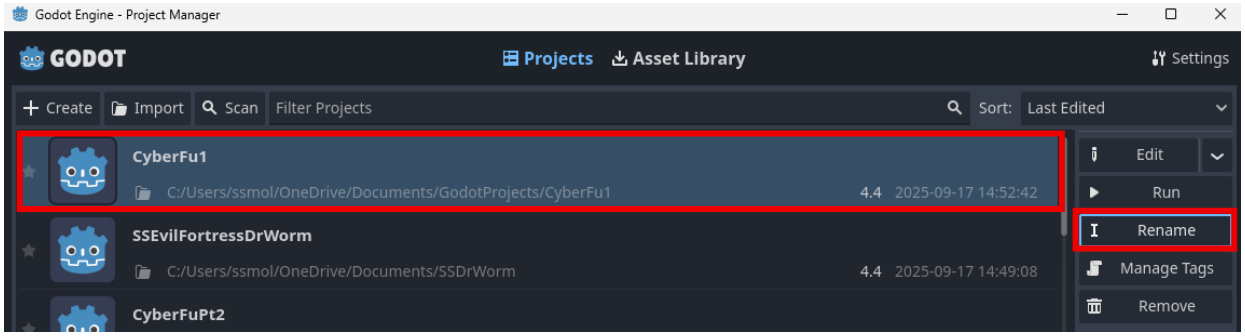
The folder should be **zipped** and must contain a **project.godot** file to be properly imported.

3 Update the **Project Installation Path** and make sure **Create Folder** is enabled.

Uncheck **Edit Now** and click Import.

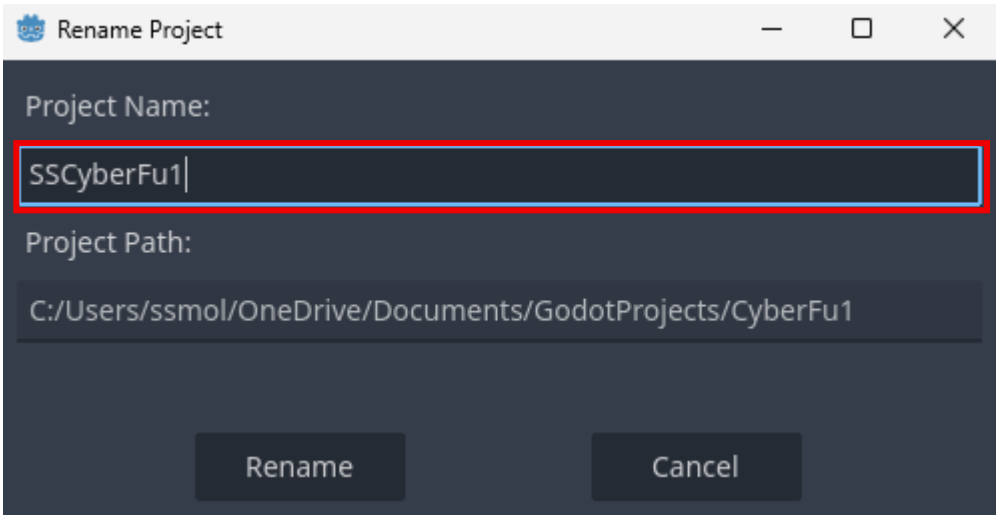


4 The project will appear at the top. Click on the project and select **Rename** on the right.



5 Update the Project Name to **[YourInitials]CyberFu1**.

Do not click Rename until after the Sensei stop!



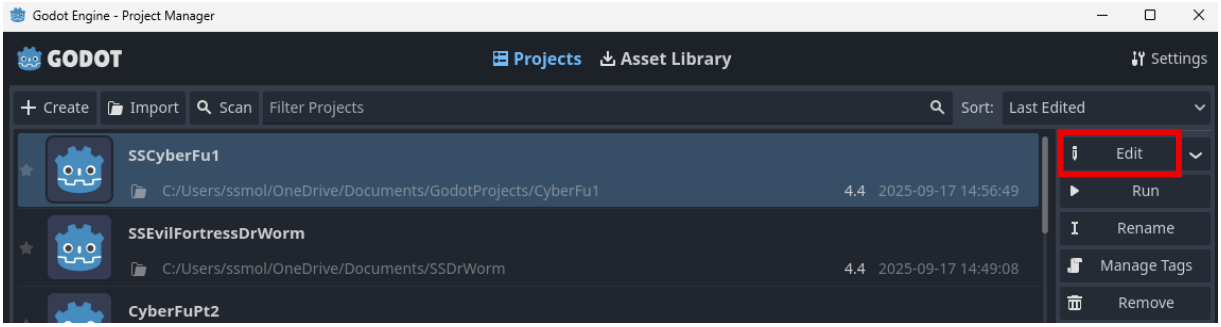
Pause for **Sensei Stop #1**!

Check in with a Code Sensei before moving on. Make sure the **project import path** is correct, and the project **has been renamed**.

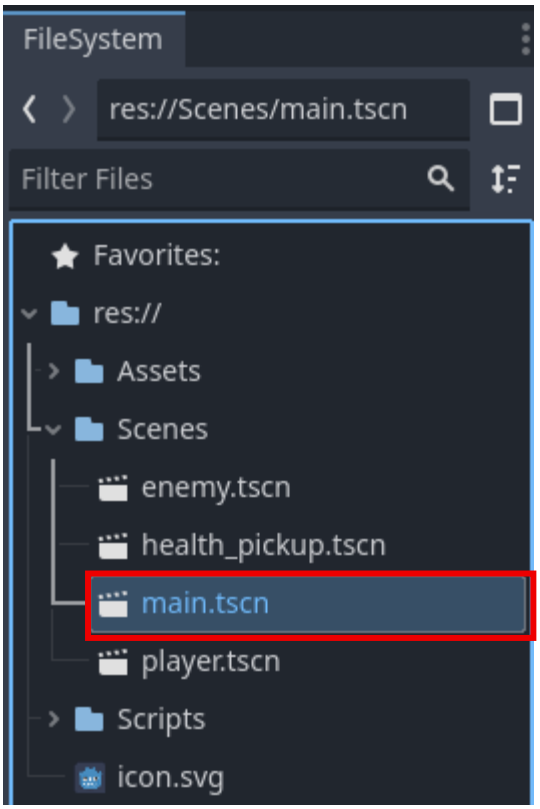
Click **Rename** when complete.

Reminder: Save your work!

6 Select the project and click **Edit** to open the starter code.



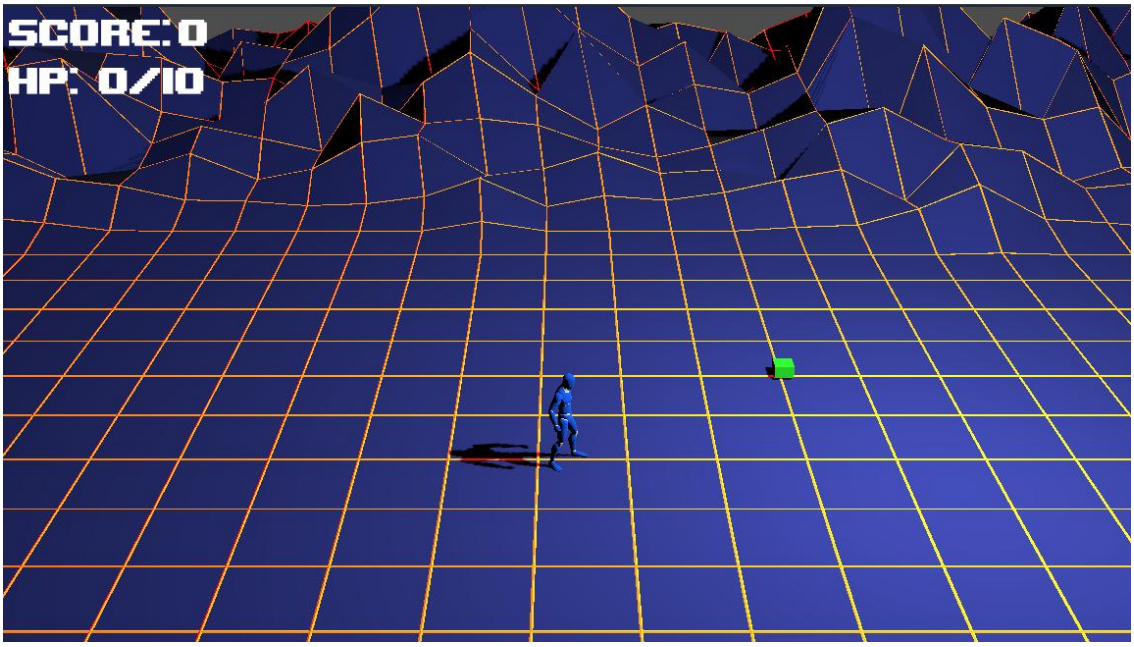
7 In **FileSystem**, navigate to **main.tscn** and double click to open it.



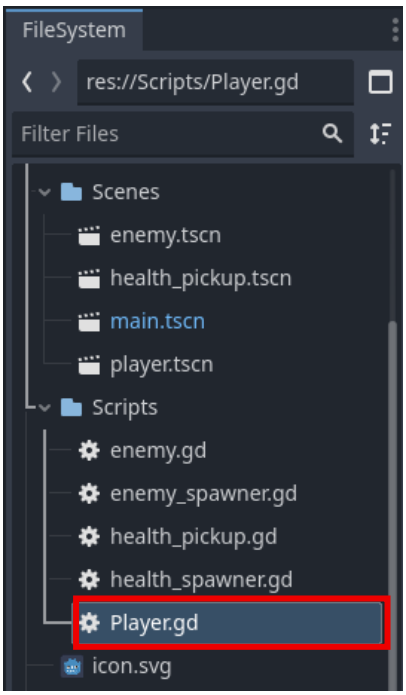
8 In the top right corner, click the **play** button to run the game.

Notice that the player is able to move using the WASD keys and interact with the health pickups. However, the UI is not updating and enemies are not spawning yet.

Close the playtest window.



9 In **FileSystem**, open the **Player.gd** script.



This script **extends** from **CharacterBody3D**, giving access to **CharacterBody3D** functions.

Review the `change_health()`, `update_health_ui()`, and `game_over_triggered()` functions already present in the script. What might they be used for?

```
31  func change_health(change: int):
32      #TODO 3
33      pass
34
35  func update_health_ui():
36      #TODO 4
37      pass
38
39  func game_over_triggered():
40      visible = false
41      move_speed = 0
42      await get_tree().create_timer(0.5).timeout
43      gameover_label.visible = true
44      restart_button.visible = true
```

10 Under **TODO 1** at the top of the script, declare an `@export` `max_health` variable of type `int`. Assign the variable an initial value of `10`.

This will be used to define the max health of the player.

```
3   @export var move_speed: float = 5.0
4   # TODO 1
5   @export var max_health: int = 10
```

11 Underneath, declare a local `health` variable of type `int`.

This variable will be used to store the current **health** of the player.

```
5   #TODO 1
6   @export var max_health: int = 10
7   var health: int
```

12 Find **TODO 2** inside the `_ready()` method. Inside, set `health` to `max_health`.

Each time the game runs, the starting health will be initialized to `max_health`.

```
18  func _ready() -> void:
19      #TODO 2
20      health = max_health
```

Reminder:

Remove the **pass** command inside the function!

13

Add code to detect when `health` has changed.

Find **TODO 3** inside the `change_health()` function. Inside, increase `health` by the value passed as the `change` parameter.

```
23  func change_health(change: int):
24      #TODO 3
25      health += change
```

14

Underneath, use an assignment operator `=` to update the value of `health`.

Use the `clamp()` method to ensure the value of `health` does not exceed the minimum or maximum health of the player.

> `clamp()`: a mathematical utility used to restrict a value within a specified minimum and maximum range. It ensures that a given value will never be less than the minimum or greater than the maximum.
>
> **Parameters:**
>
> - **Value (*Variant*):** The value to be clamped.
> - **Min (*Variant*):** The minimum allowed value.
> - **Max (*Variant*):** The maximum allowed value.
>
> **Returns (*Variant*)**: A value not less than min and not more than max.

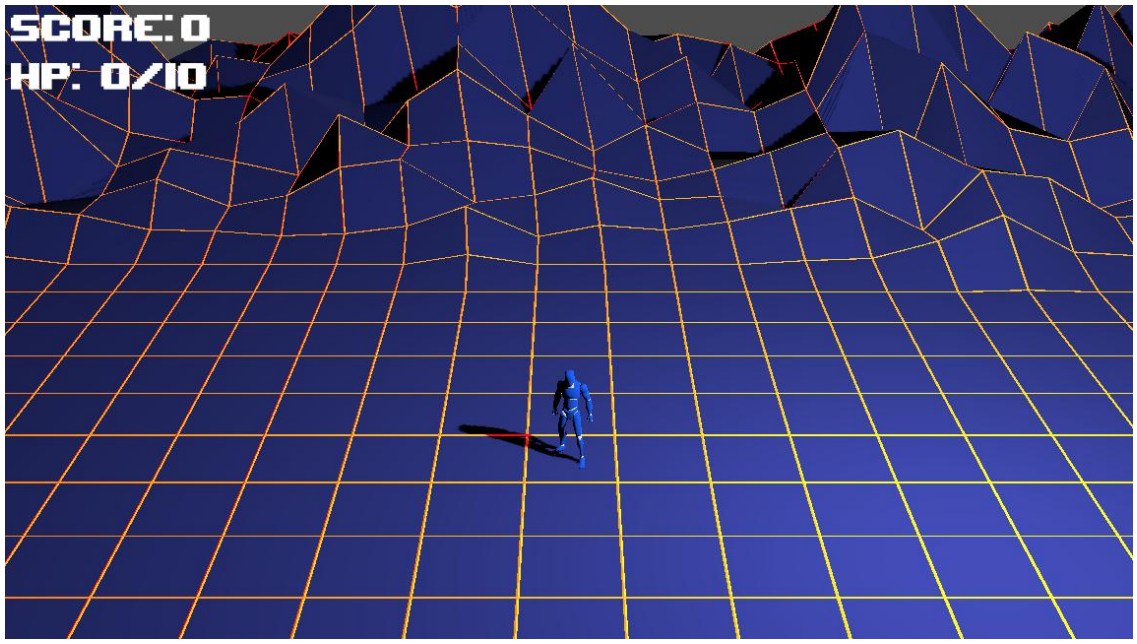
Pass `health`, `0`, and `max_health` as the three arguments.

```
33  func change_health(change: int):
34      #TODO 3
35      health += change
36      health = clamp(health, 0, max_health)
```

15 Play the game! Does the UI update when the player interacts with a health pickup?

As it is, changes to the player's health will not appear on the UI.

To fix this, the UI label needs to be updated.



Close the playtest window.

16

Find **TODO 4** inside the `update_health_ui()` function.

To access the UI label, use the `hp_label` variable, which references the **HPLabel** node in the **Main** scene.

Text is only displayed as a string, so when passing the health variables, they need to be converted into strings.

Use the **str** method and concatenation by setting the **HPLabel** text to `str(health) + “/” + str(max_health)`.

```
38 func update_health_ui():
39     #TODO 4
40     hp_label.text = str(health) + "/" + str(max_health)
```

17

Call the function to update the game’s UI.

Inside the `_ready()` method and the `change_health()` function, call the `update_health_ui()` function to display the current health status on screen.

```
12 func _ready() -> void:
13     ui = get_tree().get_first_node_in_group("ui")
14     #TODO 2
15     health = max_health
16     update_health_ui()
```

```
34 func change_health(change: int):
35     #TODO 3
36     health += change
37     health = clamp(health, 0, max_health)
38     update_health_ui()
```

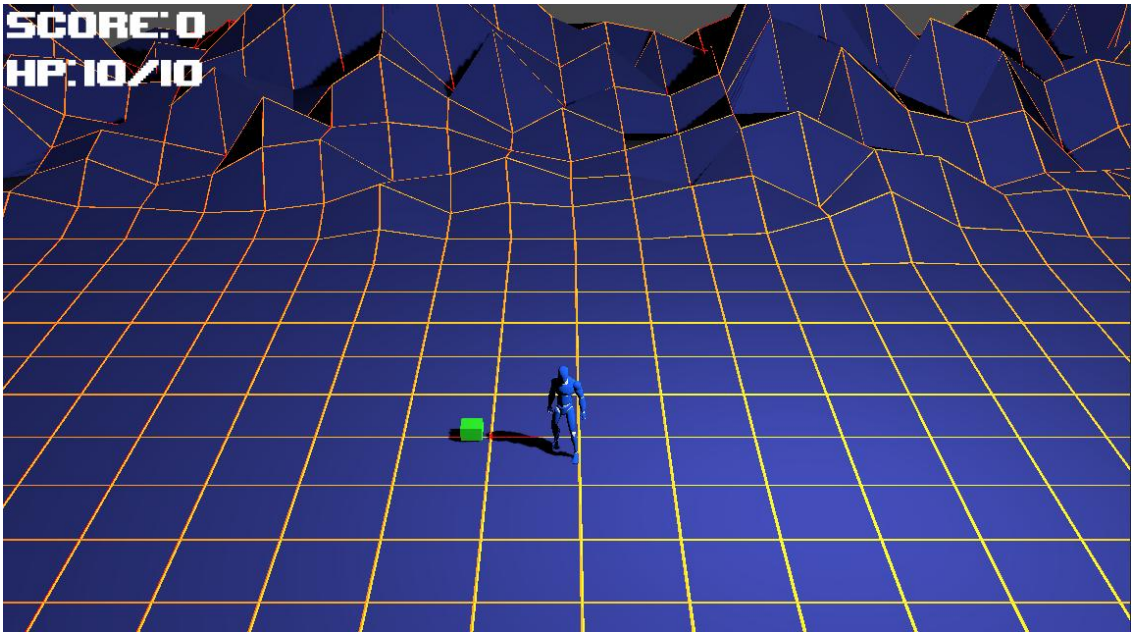
18

Playtest the game.

Notice that the UI is now set to **10/10** at the beginning of the game. However, the player is able to interact with the health pack, but still nothing happens. Why might this be?

The player has no way of taking damage! Add enemies to the game that will chase the player and explode on contact.

Close the playtest window.



Pause for **Sensei Stop #2**!

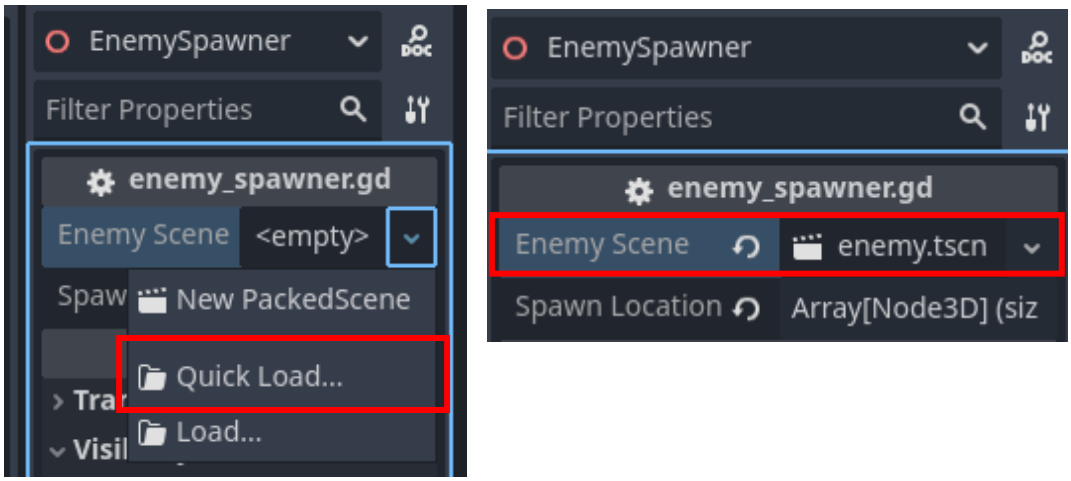
Check in with a Code Sensei before moving on. Make sure the **UI health** is set to 10 when the game starts.

Reminder: Save your work!

19 To spawn enemies, attach the enemy scene to the **EnemySpawner**.

In **Scene**, select **EnemySpawner**.

In **Inspector**, use **Quick Load** from the **Enemy Scene** drop-down menu and select **enemy.tscn**.



20 In **FileSystem**, open the **enemy.gd** script.

Notice that the script is fairly empty.

At the top of the enemy.gd script under **TODO 5**, declare an `export` `move_speed` variable of type `float`. Assign the variable an initial value of **3.0.**

```
3   # TODO 5
4   @export var move_speed: float = 3.0
```

21 Notice that there is already a `player` variable.

This variable is set to the first player found in the scene hierarchy. Note that the variable will be **null** if there is no player in the scene.

Inside of the `_process()` function, under **TODO 6**, write an `if` statement to check if `player` has a value.

```
7   func _process(_delta: float) -> void:
8       #TODO 6
9       if player:
```

22 Use the `look_at()` method to set the enemy to face in the direction of the player.

`look_at()`: part of the RayCast2D/3D classes. Used to orient the node so that its forward points towards a specified target position.

Parameters:

- **target (Vector3):** The position of the target to look at.
- **up (Vector3):** The direction of Up in relation to this node.
- **use_model_front (bool):** The direction of the z-axis being positive or negative. If true, the +Z axis is treated as forward.

Returns (*boolean*): whether the given RayCast is colliding with any objects on its masked collision layers.

The `look_at()` method will cause the enemy to rotate and look at the position set, which in this case is the player. This will cause enemies to always look at the player, which is the desired behavior.

Reminder:

Documentation can also be accessed by right clicking on the function and selecting **Lookup Symbol**.

23 Inside the `if` statement in the `_process()` method, replace `pass` with the `look_at()` method.

Pass `player.global_position` and `Vector3.UP` as the two parameters to the `look_at()` method. Don't worry about setting the third parameter yet.

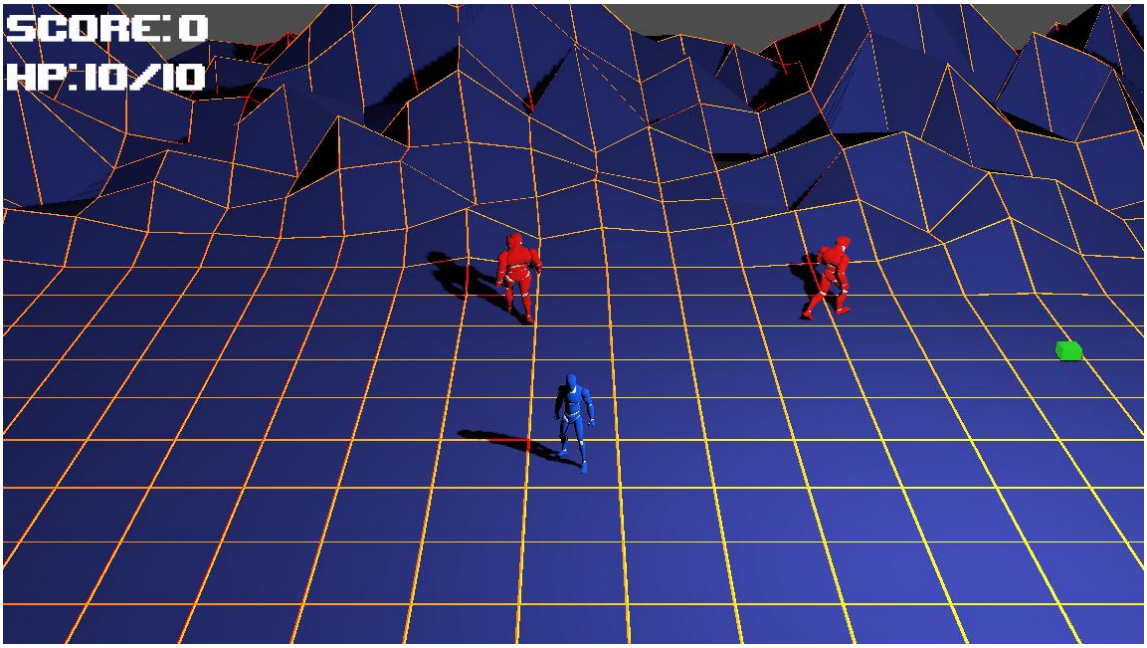
```
12  func _process(_delta: float) -> void:
13      #TODO 6
14      if player:
15          look_at(player.global_position, Vector3.UP)
```

24 Playtest the game.

Enemies should be spawning, but there are a few issues.

Notice how the enemies currently aren't moving. The enemies also seem to be facing the wrong way.

Close the playtest window.



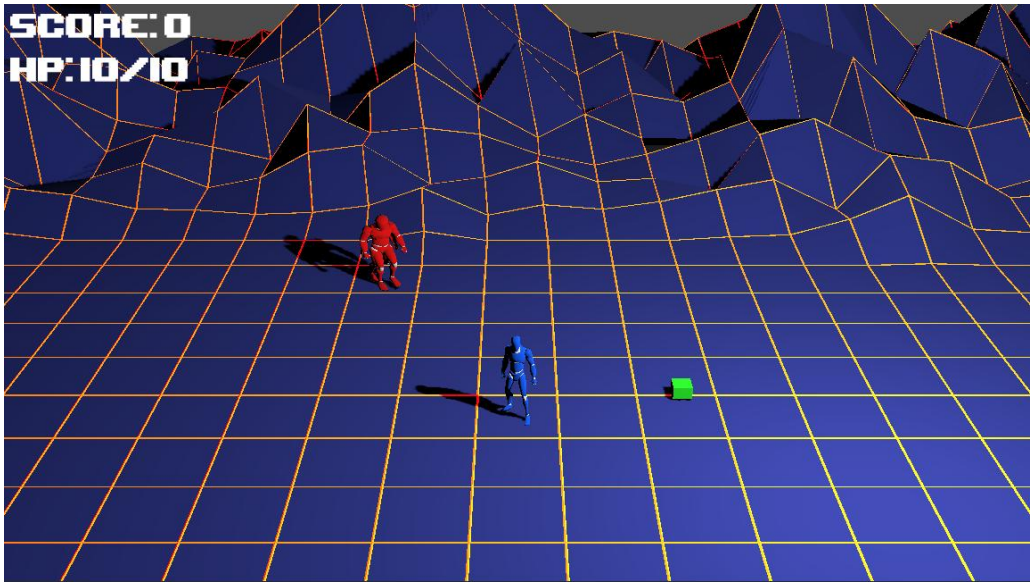
25 Navigate back to the **enemy.gd** script.

Inside of the `look_at()` function located in `_process()`, add `true` as a third parameter.

This will make the enemies face in the correct direction.

```
 8  func _process(_delta: float) -> void:
 9      # TODO 6
10      if player:
11          look_at(player.global_position, Vector3.UP, true)
```

26 Playtest the game again. Notice that enemies now look in the right direction but still aren't moving.



27 Code the enemy to move towards the player and explode on contact.

Inside of the `_physics_process()` function, underneath **TODO 7,** replace `pass` with a check for whether the `player` has value.

```
12  func _physics_process(_delta: float) -> void:
13      #TODO 7
14      if player:
15
16          #TODO 8
```

28 If the **player** is not **null**, calculate the **direction** to the **player**.

```
12 func _physics_process(_delta: float) -> void:
13     #TODO 7
14     if player:
15         var direction = (player.global_position - global_position).normalized()
```

Declare a `direction` variable inside the `if` statement, then assign its value to the difference of the **enemy's** `global_position` subtracted from the `player`**'s** `global_position`. Place the code inside parentheses.

Outside of the parentheses, call the `normalized()` **method** to properly calculate `velocity`.

29 Inside the `if` statement, set `velocity` to the calculated direction multiplied by the `move_speed` variable.

Underneath, call the `move_and_slide()` method. This will move an enemy based on the current velocity and position.

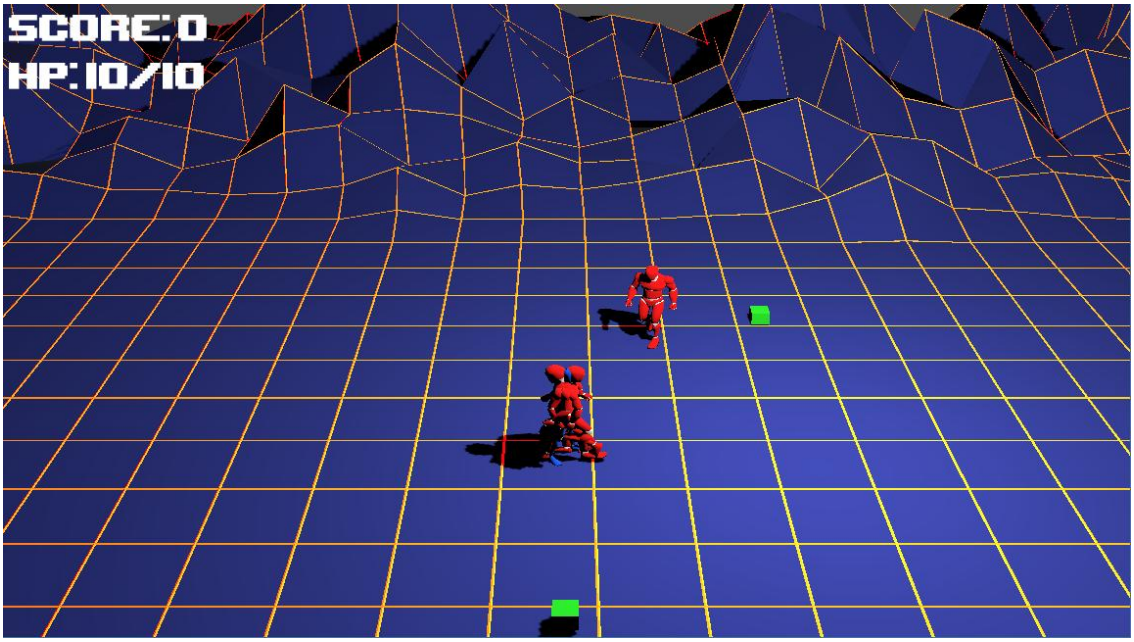
```
12 func _physics_process(_delta: float) -> void:
13     #TODO 7
14     if player:
15         var direction = (player.global_position - global_position).normalized()
16         velocity = direction * move_speed
17         move_and_slide()
18         #TODO 8
```

Customize the `move_speed` of the player and enemies. These parameters can be adjusted in the Inspectors in the main and enemy scenes.

30 Playtest the project.

Notice that enemies spawn and move towards the player.

What happens when the enemies reach the player?



31

Add code to make the enemies collide with the player upon contact.

Inside the `_physics_process()` function, under **TODO 8**, add an `if` statement. Access the player's `global_position`, then call the `distance_to()` method to calculate the difference between the enemy's `global_position`.

> `distance_to()`: A built-in method that calculates the distance between two Vector3.
>
> **Parameter (*Vector*):** The vector3 to compare to.
>
> **Returns (*float*)**: The distance between the two vectors.

Check if the **player's** global position is `0.2` or less away from the **enemy** global position.

```
15  func _physics_process(_delta: float) -> void:
16      #TODO 7
17      if player:
18          var direction = (player.global_position - global_position).normalized()
19          velocity = direction * move_speed
20          move_and_slide()
21          #TODO 8
22          if player.global_position.distance_to(global_position) < 0.2:
```

32 If the player and enemy are close enough, call the `change_health()` function to decrease the player's health by `-1`.

Underneath, call `queue_free()` to clear the enemy. This is helpful in making sure an enemy cannot explode more than once.

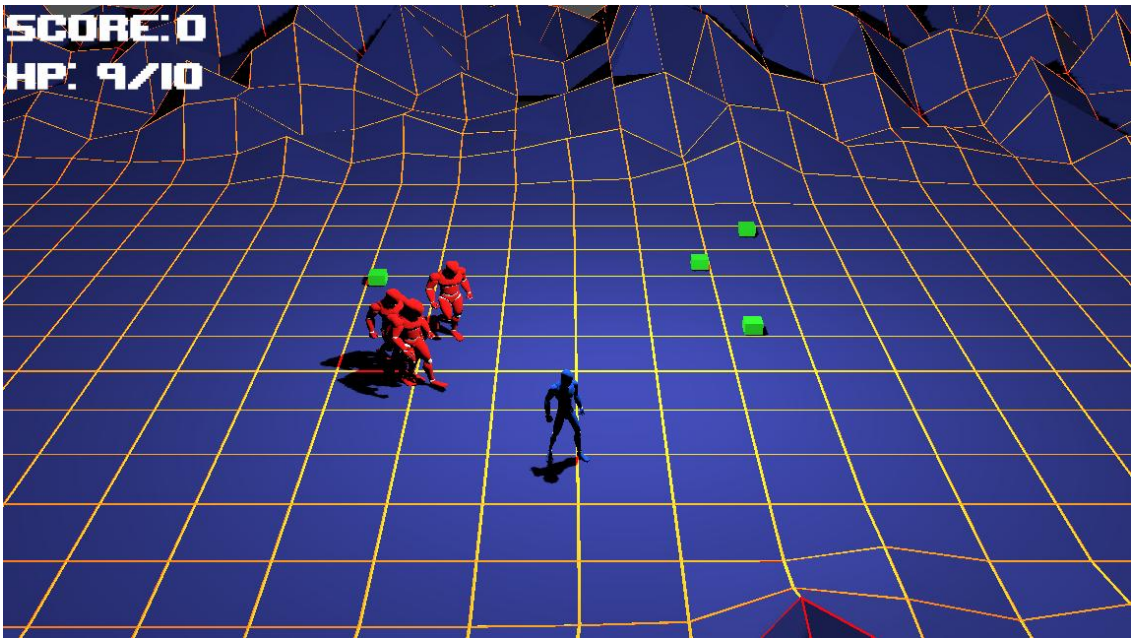
```
21      #TODO 8
22      if player.global_position.distance_to(global_position) < 0.2:
23          player.change_health(-1)
24          queue_free()
```

33 Playtest the game.

Notice that enemies will now collide with the player.

Health should decrease by 1 each time an enemy explodes. It should increase by 1 on each health pickup.

Notice that when enemies stack together, multiple enemies will explode at once, causing even more damage to the player.



Pause for **Sensei Stop #3**!

Check in with a Code Sensei before moving on. Make sure enemies are spawning and exploding on contact with the player.

Reminder: Save your work!

34

End the game once the player's health reaches 0.

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

Inside the `change_health()` function, below the `update_health_ui()` function call, check if `health` is less than or equal to `0`. If so, call the `game_over_triggered()` function.

```
24  func change_health(change: int):
25      #TODO 3
26      health += change
27      update_health_ui()
28      if health <= 0:
29          game_over_triggered()
```

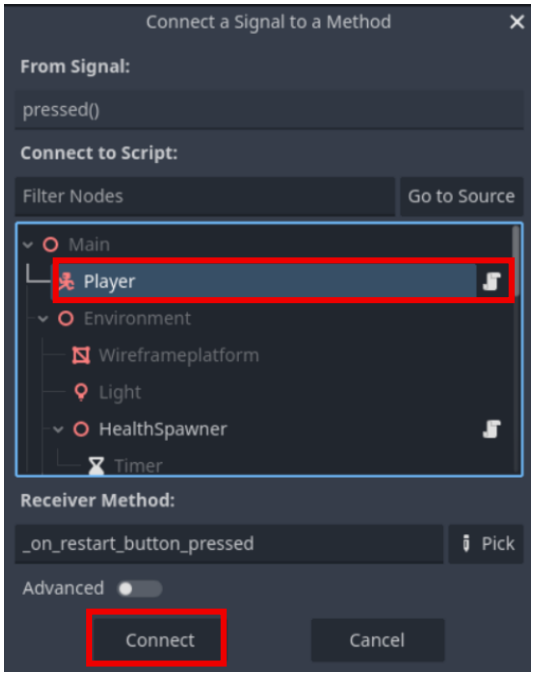
35 Playtest the game.

What happens when the player's health reaches 0?

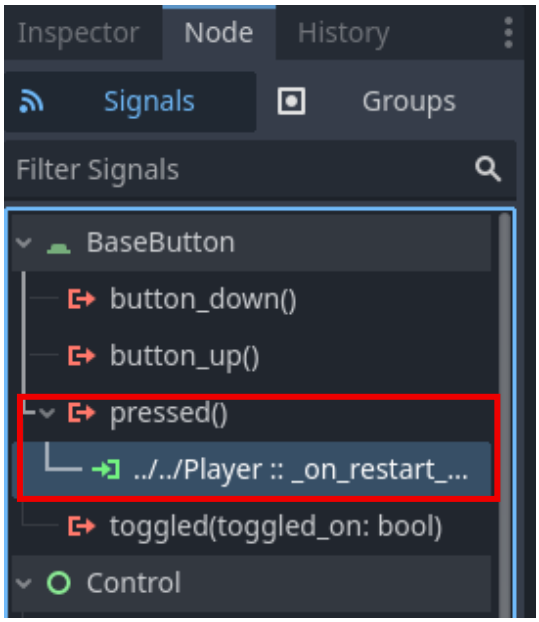
Notice anything wrong? The Restart button seems to be broken!

36 In the **Main Scene**, select the **RestartButton** node.

Next to the Inspector, click on **Node** then **Signals**. Connect its `pressed()` signal to the **Player** node.



Notice the green signal symbol appears once the signal has been successfully connected.



37 Playtest the game.

The game should now restart when the player clicks the Restart button!

What happens to the score value throughout the project?

38 Right now, the player score stays at 0. Add code to increase the score as time goes on.

In **player.gd**, scroll to the bottom to find **TODO 9 i**nside of the `_on_score_timer_timeout()` function.

Inside the function, replace `pass` with an `if` statement that checks if `health` is greater than `0`.

The score should only increase while the player is alive.

```
46  func _on_score_timer_timeout() -> void:
47      #TODO 9
48      if health > 0:
49
```

39 If the player still has health, increase the `score` by `1`.

Underneath, update the score label. Use the `score_label` variable to get the score label. Use `str` to update the text to the value of `score`.

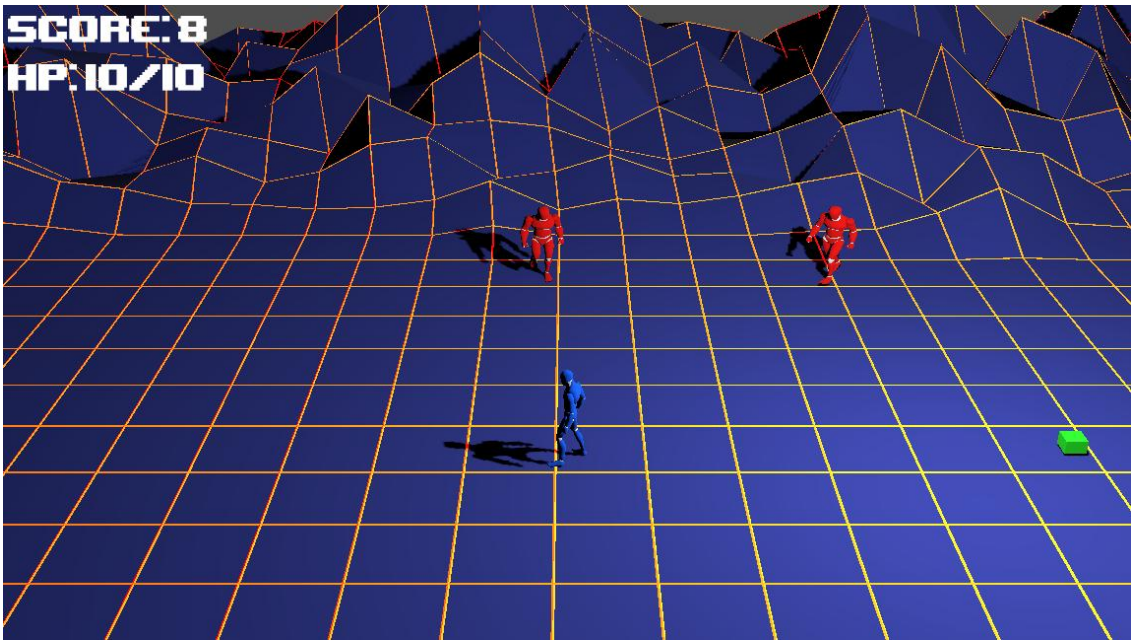
```
50  func _on_score_timer_timeout() -> void:
51      #TODO 9
52      if health > 0:
53          score += 1
54          score_label.text = str(score)
```

40 Playtest the game.

The score should increase every second that passes.

Enemies spawn and chase the player. When the player runs out of health, the game is over.

The player can collect health packs to restore health.



Pause for **Sensei Stop #4**!

Before submitting, check in with a Code Sensei to make sure enemies are following and exploding on contact with the player, and **Game Over** is reachable after the player runs out of health.

Reflect on the following:

- What did you learn about enemy movement?
- What did you enjoy most when creating this project?
- What was something you found difficult and why?

Reminder: Save your work!

Congratulations on completing **SB Activity 07: CyberFu Part 1** in Godot – **You Rock**! You are now ready to save this project and submit it.

Continue your exploration with Godot by opening the **SB Activity 08: World of Color** Ninja Guide.