Platformer Essentials COOKBOOK

Make platformer games by following step-by-step recipes

Henrique Campos

Platformer Essentials Cookbook

v.1.0.0 – 27.12.2023

A few words on copyrights

Hey! *Thank you so much for acquiring this book~*

I'm personally against the idea of copyrights, so straight to the point, this book is under public domain.

Besides that, note that all the money I get from my work, **I spend on further work**. So, your support is fundamental to keep things going.

It means I've made something valuable, and as such I can keep doing it instead of trying new things.

So, thank you so much if you got it from one of my official distribution channels: **ko-fi** and **itch.io**.

If you got it from another distribution channel, please consider supporting my work to get more quality content.

You can directly support me on Ko-fi and buy me a Frappuccino!

 ***https://ko-fi.com/pigdev***

On itch.io you will see more of my work. There you'll find my games, assets, courses, and books:

 ***https://pigdev.itch.io***

Follow me on the bird network to get frequent updates on my endeavors:

 ***https://twitter.com/pigdev***

Watch my YouTube content. There you'll find video tutorials, reviews, news, devlogs and more:

 ***https://youtube.com/pigdev***

Table of Contents

A few words on copyrights 3
Introduction 8
BasicMovingCharacter2D 12
- What is this recipe?....12
- When to use this recipe?....13
  - Pros & Cons:....13
- How to make this recipe?....15
  - Mise en Place:....15
  - Cooking it:....19
- Why does this recipe work?....22
  - Design-wise:....22
  - Engineering-wise:....22

PassThroughCharacter2D 24
- What is this recipe?....24
- When to use this recipe?....26
  - Pros & Cons:....26
- How to make this recipe?....27
  - Mise en Place:....27
  - Cooking it:....31
- Why does this recipe work?....35
  - Design-wise:....35
  - Engineering-wise:....35

MovingPlatform2D 37
- What is this recipe?....37
- When to use this recipe?....40
  - Pros & Cons:....40
- How to make this recipe?....42
  - Mise en Place:....42
  - Cooking it:....43
- Why does this recipe work?....45
  - Design-wise:....45
  - Engineering-wise:....45

PathFollowPlatform2D 47

How to make this recipe?.....47
Mise en Place:.....47
Cooking it:.....48
Hazard2D 52
What is this recipe?.....52
When to use this recipe?.....54
Pros & Cons:.....54
How to make this recipe?.....56
Mise en Place:.....56
Cooking it:.....66
Why does this recipe work?.....70
Design-wise:.....70
Engineering-wise:.....70
BumpingEnemy2D 71
What is this recipe?.....71
When to use this recipe?.....72
How to make this recipe?.....74
Why does this recipe work?.....78
Stomping & StompableObject2D.....81
How to make this recipe?.....81
Mise en Place.....81
StompableEnemy2D.....81
StompingObject2D.....83
Cooking it:.....87
PathFollowEnemy2D 91
What is this recipe?.....91
When to use this recipe?.....92
Pros & Cons:.....92
How to make this recipe?.....94
Mise en Place:.....94
Setup.....99
Wandering behavior.....101
Seeking behavior.....104
Returning behavior.....107
Seeking start and Seeking end.....110
Cooking it.....113

Why does this recipe work?....115
Design-wise:....115
Engineering-wise:....116
InteractiveArea2D 118
What is this recipe?....118
When to use this recipe?....121
Pros & Cons....121
How to make this recipe?....122
Mise en Place:....122
InteractionArea2D....127
Cooking it:....128
Design-wise:....133
Engineering-wise:....133
Portal2D 135
What is this recipe?....135
When to use this recipe?....137
Pros & Cons:....139
How to make this recipe?....140
Mise en Place:....140
TeleportData....140
Portal2D....142
Cooking it:....145
Why does this recipe work?....149
Design-wise:....149
Engineering-wise:....149
Checkpoint2D 151
What is this recipe?....151
When to use this recipe?....153
Pros & Cons:....157
How to make this recipe?....159
Mise en Place:....159
CheckpointData....159
Checkpoint2D....159
Cooking it:....167
Why does this recipe work?....171
Design-wise....171

Engineering-wise:.....171
Switch2D 173
What is this recipe?.....173
When to use this recipe?.....176
Pros & Cons.....177
How to make this recipe?.....178
Mise en Place:.....178
Switch2D.....178
Target Object.....181
Why does this recipe work?.....186
Design-wise:.....186
Engineering-wise:.....186
Conclusion 188

Introduction

Welcome to the **Platformer Essentials Cookbook**, an insightful journey into the world of game development, specifically focusing on making **platformer** games with **Godot Engine**. This book serves as a culinary guide for game developers, offering a collection of "recipes" - each a unique game development pattern tailored for platformer games. Let's explore what to expect from this book and draw parallels between these game development patterns and traditional cooking recipes.

Fundamental Recipes - The ***BasicMovingCharacter2D***: Just as every chef begins with basic recipes, this book starts with foundational game patterns like the *BasicMovingCharacter2D*, which represents the player's avatar in the game world. This recipe is essential, as players interact with the game world primarily through this avatar, akin to a chef's reliance on their foundational culinary skills.

Versatility and Adaptation: The recipes in this book are designed for broad applicability, similar to how basic cooking recipes can be modified to suit different tastes. The *BasicMovingCharacter2D*, for instance, is a versatile pattern used in many platformer games, adaptable to various game designs.

Ease of Management and Control: Each recipe in the book is crafted for easy management and offers complete control, much like how a well-designed cooking recipe allows chefs to manage ingredients and cooking processes effectively. The patterns

provide clear guidelines, ensuring minimal dependencies and full control over the game physics.

Step-by-Step Guidance - The *Mise en Place*: The book breaks down each pattern into manageable steps, mirroring the “mise en place” practice in cooking, which involves preparing and organizing ingredients. This approach simplifies complex game development processes, making them more approachable and structured.

Application in Context - *Cooking it*: Just as a chef brings together prepared ingredients to create a dish, the book guides you through applying these game patterns in real-world scenarios. This includes integrating input events with character actions, akin to how cooking involves combining ingredients in specific sequences to achieve desired flavors.

Design Considerations - *Why it Works*: The book not only provides the 'how' but also dives into the “why”. It explains the design rationale behind each pattern, similar to how culinary recipes often come with notes on why certain ingredients or techniques are used. For instance, it discusses the importance of maneuverability and collision handling in the *BasicMovingCharacter2D* recipe.

Technical Insights - The *Engineering Aspect*: Lastly, the book provides a deep dive into the engineering aspects of game patterns, similar to understanding the science behind cooking techniques. It explains how the Godot Engine's API contributes to the effective functioning of these patterns, ensuring a seamless gaming experience.

Introduction

The **Platformer Essentials Cookbook** is akin to a culinary guide for game developers. It combines the art of game design with the precision of engineering, offering a comprehensive collection of patterns that are as fundamental, versatile, and nuanced as classic recipes in the culinary world. The book offers valuable insights into creating engaging, well-designed platformer games using the Godot Engine.

Page intentionally left in blank

BasicMovingCharacter2D

What is this recipe?

A *BasicMovingCharacter2D* is the player's avatar in the game's world. It has gravity pointing downwards and snaps to the floor.

It has a smooth movement on slopes, jumps, and moves at constant speed on the horizontal axis.

It's like how characters usually move in Jump 'n' Shoot games like Cuphead.

This recipe is the most fundamental one. Players use this recipe to interact with the game world. So, this is the recipe they spend the most time with.

When to use this recipe?

Use a *BasicMovingCharacter2D* when you want players to use an avatar to interact and explore the game world.

Most platformer games use this recipe to some extent, or a variation of it. As some modern examples:

- Cuphead
- Flinthook
- Hollow Knight, and others

This recipe doesn't go well if you want physics-simulated movement. Especially if this movement tries to match the real world.

If you want physically accurate movement, I recommend using a custom implementation that uses a ***RigidBody2D*** instead.

Pros & Cons:

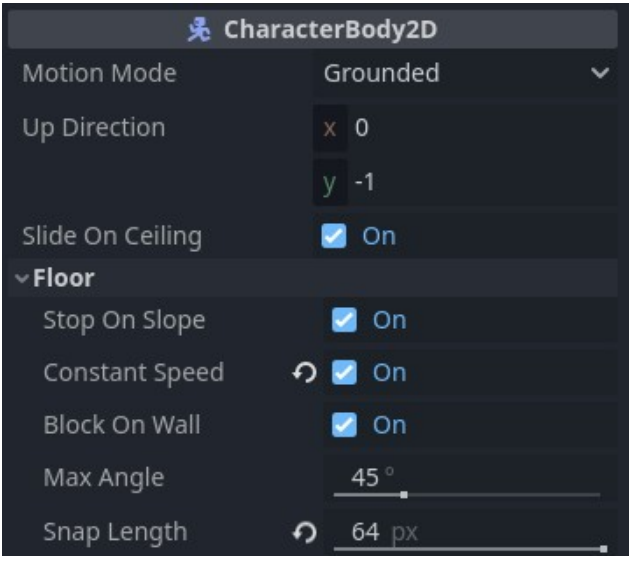
✔ Easy to manage, it's self contained meaning minimal to no dependency.
✔ Full control over physics, you get what you ask from the physics engine.
✔ Snappy movement, character stops as soon as player releases the key.
✔ Clear animation triggers, you know where to call each animation in the code.
✖ Lacks interesting physics behaviors like friction,

bounciness, and absorption.
✖ Lacks acceleration/deceleration curves to add personality to the character.

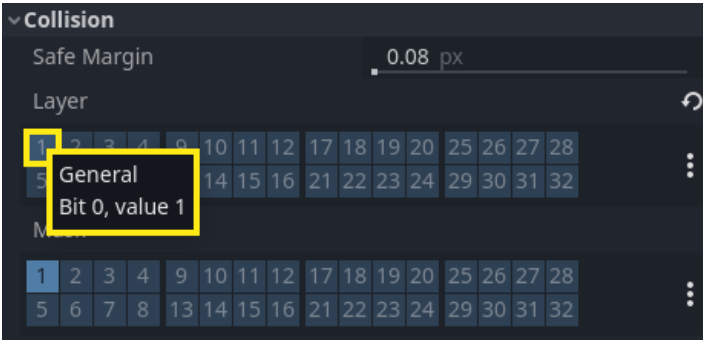
How to make this recipe?

Mise en Place:

1. Add a ***CharacterBody2D*** as the scene's root node
2. Rename the root node as *BasicMovingCharacter2D*
3. Set the *Floor Snap Length* long enough to reach the floor when the character is on slopes. In my case, 64.0 did the trick.
4. Toggle on the *Constant Speed* property



5. Toggle off the *Collision* > *Layer* that represents general collisions.



> **Tip:**
>
> We do that because, ideally, this class only detects other objects, but shouldn't be detected by them. So we remove it from the general collision layer that we use for floors, walls, ceilings, etc.

6. Attach a ***GDScript*** to the root node, then in the script:
   i. Add some variables to control the movement forces

```
@export var speed = 500.0
@export var gravity = 2000.0
@export var jump_strength = 800.0
```

   ii. After that, add a variable to control the look direction on the horizontal axis

```
var direction = 0
```

   iii. In the `_physics_process()` callback, apply gravity to the `velocity.y` axis. Then for the x axis, multiply the `speed` and the `direction`

```
func _physics_process(delta):
	velocity.y += gravity * delta
	velocity.x = direction.x * speed
```

iv. Then, call the `move_and_slide()` method to move the character
v. Create a `jump()` method. Check if the *BasicMovingCharacter2D* is on the floor. If so, set the `velocity.y` to be equal to the `jump_strength`

```
func jump():
	if is_on_floor():
		velocity.y = -jump_strength
```

vi. Create a `cancel_jump()` method. Check if the *BasicMovingCharacter2D* isn't on the floor and it's `velocity.y` is smaller than `0.0`. If so, set `velocity.y` to `0.0`

```
func cancel_jump():
	if not is_on_floor() and velocity.y < 0.0:
		velocity.y = 0.0
```

> **Tip:**
>
> We use that to allow players to control the jump height. It's common to say "the longer you hold the button the higher you jump". What happens is that if

> you release the button you cancel the jump before it reaches its maximum height. You aren't "charging" the jump.

With that we have a nice *BasicMovingCharacter2D* ready to use! At the end, the script should like that:

```
class_name BasicMovingCharacter2D
extends CharacterBody2D

@export var speed = 500.0
@export var gravity = 2000.0
@export var jump_strength = 800.0

var direction = 0

func _physics_process(delta):
	velocity.y += gravity * delta
	velocity.x = direction * speed
	move_and_slide()



func jump():
	if is_on_floor():
		velocity.y = -jump_strength
```

```
func cancel_jump():
     if not is_on_floor() and velocity.y < 0.0:
           velocity.y = 0.0
```

Cooking it:

Now, to actually use this recipe we can use it for our player. For that, we can create an inherited scene and extend the script.

Then, glue the input events to our character's actions. Let's do it.

1. Export some variable to hold the *Input Actions* that we are going to handle in the code.

```
@export var move_left_action = "move_left"
@export var move_right_action = "move_right"
@export var jump_action = "jump"
```

2. Then, in the `_unhandled_input()` callback, let's handle the horizontal movement using the `direction` variable.

```
func _unhandled_input(event):
     # Horizontal movement
     if event.is_action(move_left_action):
           if event.is_pressed():
                 direction = -1
           elif
Input.is_action_pressed(move_right_action):
```

```
                direction = 1
            else:
                direction = 0
        elif event.is_action(move_right_action):
            if event.is_pressed():
                direction = 1
            elif
Input.is_action_pressed(move_left_action):
                direction = -1
            else:
                direction = 0
```

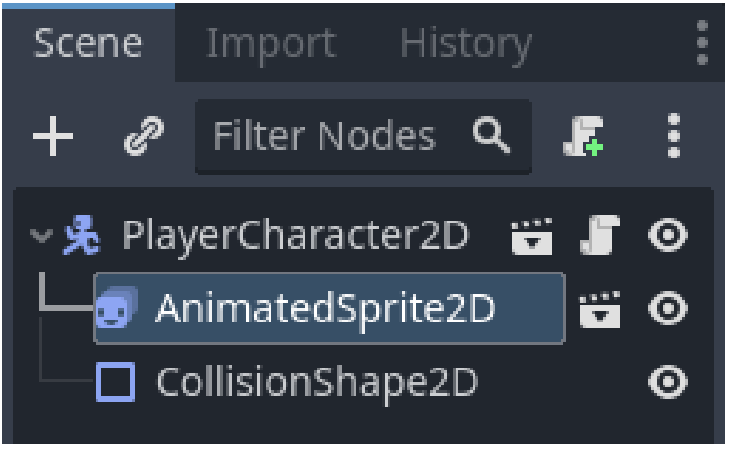
> **Tip:**
>
> We use this input approach due to a design decision:
>
> *The character should always move when the player is pressing a movement action. It only stops if the player isn't pressing any movement input action.*
>
> Some arithmetic approaches cause weird behaviors.
>
> For instance, when we sum the horizontal axis strength using `Input.get_axis_strength()`. This out the movement, making the character stop.
>
> Players can be confused with this, especially in action games.

3. And for the vertical movement, we can make use of the `jump()` and `cancel_jump()` methods

```
    # Vertical movement
    if event.is_action_pressed(jump_action):
        jump()
    elif event.is_action_released(jump_action):
        cancel_jump()
```

> **Tip:**
>
> We separate the movement logic from the input logic so we can use the *BasicMovingCharacter2D* for enemies as well.

4. Now, add a ***CollisionShape2D*** or a ***CollisionPolygon2D***
5. And, finally, add some graphics. You can use a Sprite2D, AnimatedSprite2D, or whatnot.



Why does this recipe work?

Design-wise:

In platformer games, the core feature is to use an avatar to explore the game world.

This avatar needs to be able to maneuver through the world's obstacles. In that sense, it needs to handle collisions and move around. Both vertically and horizontally.

We keep a constant horizontal speed to provide a snappy movement to players. Either the character is moving or stopped. Players don't have to do any acceleration or deceleration calculation on top of that.

This allows us to design levels with way more precision and fewer edge cases.

Engineering-wise:

Using Godot Engine's built-in physics simulation, we can simulate a movement.

The engine does the calculations and tests it this is a valid movement or not.

A valid movement is when there are no collisions between any *CollisionShape2Ds*

So, when there’s an invalid movement, Godot re-positions the character to a valid position.

Using a value above zero on the *Snap Length* property we ensure that the character will be "glued" to the floor. This is what allows it to smoothly climb slopes.

Without this snapping, the character hops when it goes from a slope to a flat floor.

The same would happen when the character goes down from a flat floor to a slope. It would look like it floated for a while.

This is because the sum of the vertical and horizontal velocity with the floor offsets it

Also, to maintain a constant horizontal speed, we toggle on the *Constant Speed* property. Otherwise, the resulting velocity when "climbing" a slope would be slower than on a flat floor.

The same would happen when moving down the slope, the character would move faster.

This is physically accurate, but not desired in this design.

PassThroughCharacter2D

What is this recipe?

A *PassThroughCharacter2D* is a ***BasicMovingCharacter2D*** that is capable of passing through specific areas of the game.

In platformer games is common to allow the player's avatar to jump down and jump through some objects. These areas can be platforms, floors, secret grounds, and other special objects.

This recipe extends the *BasicMovingCharacter2D* behavior so it can perform this type of movement.

As an example, look at the Floral Fury battle on Cuphead.

Cuphead has to be on a floating platform to be in the best position to shoot the boss. But when the boss decides to attack he has do jump down to the ground.

This allows him to also get into a defensive position that also brings a good positioning to score some hits on the boss.

When to use this recipe?

Use a *PassThroughCharacter2D* when you have areas of your game world that you want the player to actively decide to pass through them.

A good example of that is a multi-floored level where the player switches between floors to get the best results.

On top of Cuphead, there are other examples where this recipe is used:

- Ori and the Blind Forest
- Flinthook
- Hollow Knight
- And more

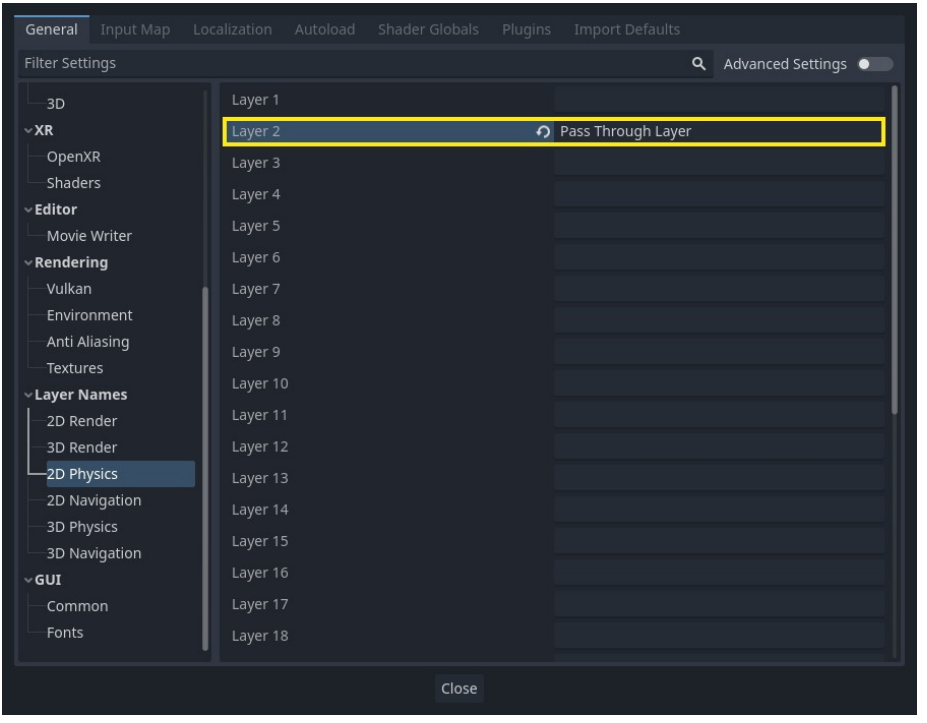
Pros & Cons:

✔ Adds another layer of interaction with the game world.
✔ Opens up a range of new possibilities for world design.
✔ Takes advantage of the vertical space of the game's world.
✖ Increases the number of edge cases for physics-related bugs.
✖ A new feature you must teach to players.
✖ One less physics layer is available for other features.

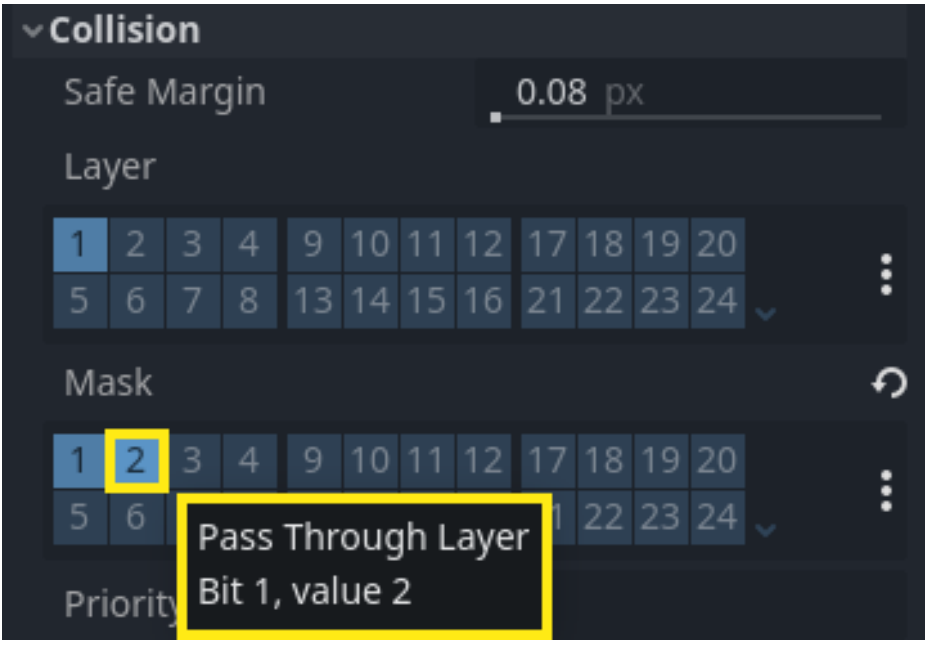
How to make this recipe?

Mise en Place:

1. Go to *Project -> Project Settings -> Layer Names -> 2D Physics*(optional)
2. Choose a layer and name it as *Pass Through Layer*(optional)



3. Create an Inherited Scene using *BasicMovingCharacter2D* as root node
4. Rename the root node as *PassThroughCharacter2D*
5. Toggle on the *Pass Through Layer* in the *Collision → Mask* property



6. Extend the root node's GDScript and save
7. In the script:
   i. Export a 2D physics layer variable to store the layers you want the *PassThroughCharacter2D* to pass through. By default, we set it to `2`, the layer we defined in the *Project Settings*.

```
@export_flags_2d_physics var pass_through_layer = 2
```

> **Tip:**
>
> We do that so we can select the masking layer using the Inspector later on. This also allow us to use other layers in this mechanic if we want to.

ii. Create two methods, one to enable the pass-through behavior and another to disable it.

```
func enable_pass_through():
     pass



func disable_pass_through():
     pass
```

iii. Inside each method, let's calculate the bitmask number and use the `set_collision_mask()` method to enable and disable the masking of the *Pass Through Layer*.

```
func enable_pass_through():
     var layer_number =
(log(pass_through_layer) / log(2)) + 1
     set_collision_mask_value(layer_number,
false)



func disable_pass_through():
```

```
    var layer_number =
(log(pass_through_layer) / log(2)) + 1
    set_collision_mask_value(layer_number, true)
```

> **Tip:**
>
> We use this formula to calculate the layer number based on its bit. You can check this ***Geeks for Geeks post*** to understand it better.

The final script should look like that:

```
class_name PassThroughCharacter2D
extends BasicMovingCharacter2D


@export_flags_2d_physics var pass_through_layer = 2


func enable_pass_through():
    var layer_number = (log(pass_through_layer) /
log(2)) + 1
    set_collision_mask_value(layer_number, false)



func disable_pass_through():
    var layer_number = (log(pass_through_layer) /
log(2)) + 1
    set_collision_mask_value(layer_number, true)
```

Cooking it:

Now, to actually use this recipe we just need to set what the triggers the enabling or disabling of the mechanic. We can do that with somekind of A.I. But let's see how we can do that in the player.

1. Create a new inherited scene using the *PassThroughCharacter2D* as root not
2. Add a *CollisionShape2D* or a *CollisionPolygon2D* as its child
3. Add some graphics using some *Sprite2Ds*, *AnimatedSprite2Ds*, or whatnot
4. Extend the script and then
    i. Use the same ***horizontal movement logic*** we used in the *BasicMovingPlayer2D*
    ii. Then, for the vertical movement, check if the jump action was pressed.
    iii. If so, check if the move down action is also pressed. If this is the case, call the `enable_pass_through()` method. Otherwise, just call the `jump()` method instead.
    iv. If the player released the jump action, we cancel the jump
    v. If the player released the move down action, we disable the pass through mechanic

```
    # Vertical movement
    if event.is_action_pressed(jump_action):
        # Pass through logic
```

```
            if
Input.is_action_pressed(move_down_action):
                  enable_pass_through()
            else:
                  jump()
      elif event.is_action_released(jump_action):
            cancel_jump()
      if
event.is_action_released(move_down_action):
            disable_pass_through()
```

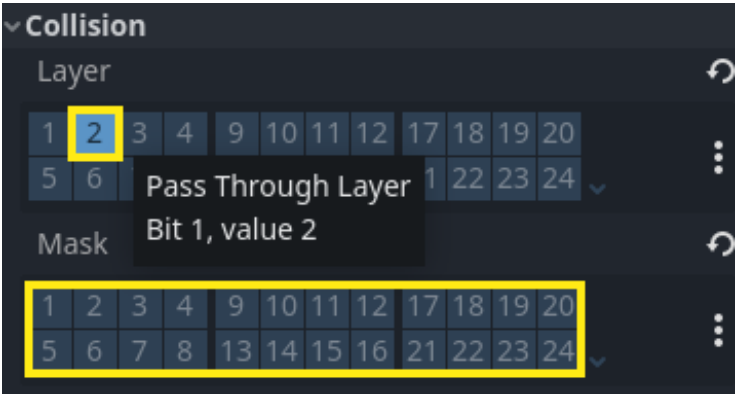
The complete script should look like this:

```
class_name PassThroughPlayer2D
extends PassThroughCharacter2D

@export var move_left_action = "move_left"
@export var move_right_action = "move_right"
@export var move_down_action = "move_down"
@export var jump_action = "jump"



func _unhandled_input(event):
      # Horizontal movement
      if event.is_action(move_left_action):
```

```
            if event.is_pressed():
                  direction = -1
            elif
Input.is_action_pressed(move_right_action):
                  direction = 1
            else:
                  direction = 0
      elif event.is_action(move_right_action):
            if event.is_pressed():
                  direction = 1
            elif
Input.is_action_pressed(move_left_action):
                  direction = -1
            else:
                  direction = 0
      # Vertical movement
      if event.is_action_pressed(jump_action):
            # Pass through logic
            if
Input.is_action_pressed(move_down_action):
                  enable_pass_through()
            else:
                  jump()
      elif event.is_action_released(jump_action):
            cancel_jump()
```

```
    if event.is_action_released(move_down_action):
        disable_pass_through()
```

5. Create a new scene using a ***CollisionObject2D***, it can be a ***StaticBody2D*** for instance.
6. In the *Collision → Layer* property, toggle off all bit flags and toggle on only the *Pass-Through Layer.*
7. In the *Collision → Mask* property, toggle off all bit flags



8. Save the scene as *PassableThroughBody2D*

Now, when using this object, you will add a *CollisionShape2D* or a *CollisionPolygon2D* as its child.

Make sure to toggle on the *One Way Collision* property as well when you do so.

Why does this recipe work?

Design-wise:

Since we are working with a 2D world, we are already constrained in the disposal of the environment elements.

We can clearly imagine a way to use the horizontal space. We can add some hills, slopes, make the player jump through some dead pits, add some hazards here and there... and that's it.

But then...what about the vertical dimension of your level? Why not use it as well?

Of course that are many solutions for that, flying mechanics, platforms, ladders... But what about moving downwards as well?

Imagine that in the middle of your level there's a special ground that if the player jumps down.

Let’s say that after jumping down three floors, they find a dungeon, a whole new special area that was hidden in the level.

That's what the *PassThroughCharacter2D* adds to your game design toolkit!

Engineering-wise:

In Godot Engine, collisions are handled from two perspectives:

- Collision Layer and
- Collision Mask

The Layer is where the object **is in**. The Mask is where it **looks for** when searching for collisions.

So two objects can be on the same *Collision Layer* but never interact with each other if their *Collision Mask* doesn't look for collision in that layer.

What we do in this recipe is that we toggle the *Collision Mask* that looks for collisions in the *Pass Through Layer*.

So we can play with that in many ways, in the first case we toggled this mask based on the player's inputs.

But imagine an enemy that is on the floor right above the player. Then when the player is within its sight, it jumps down to the player's floor and hits the player. Sounds cool, right?

If you want to learn more about how *Collision Layers* and *Collision Masks* work, check out these resources:

- ***Godot Engine - Collision Layer and Mask*** which is a video in my very channel
- ***Godot Engine Official Documentation - Collision Layers and Masks***

MovingPlatform2D

What is this recipe?

A *MovingPlatform2D* is a platform that moves in the game world. But not only that. It carries the player with it and may add its own velocity to the player's movement as well.

This recipe doesn't need a single line of code. But it needs a ***BasicMovingCharacter2D*** or, preferably, a ***PassThroughCharacter2D*** to work.

Combining some instances of this recipe, we can create platforming puzzles. This way we test the players' abilities to move through the game world.

Using this recipe we can fine tune the platform's movement to fit the level design. Each *MovingPlatform2D* can have its own set of movement and create a living world for our players.

As an example, take a look at the Mount Huru segment from Ori and the Blind Forest.

Here, Ori has to use a switch to toggle the movement of the highlighted platforms.

When they start moving, they block some lava streams and allow Ori to jump on them to reach the land again.

When to use this recipe?

Use this recipe when you want to add depth to your platformer game's core aspect: platforms.

With the *MovingPlatform2D* you add another layer of dynamics to your game. Players will have to deal not only with the game's controls and physics. They will also have to understand the timing of the *MovingPlatform2D*.

You can use this recipe to create moving objects such as elevators and carriers.

You can play with their rotation and turn them into a whole new mechanic. For instance, you can make them rotate 90 degrees in the middle of their movement to drop the player on a lava lake!

Pros & Cons:

✔ Add depth dynamic to the game world since platforms will be able to move instead of being always static.
✔ Opens up new possibilities for level design since we can create new tests for players' platforming skills.
✔ They are flexible and can fit any game and level theme.
✖ Can be a bit buggy due to some physics inconsistencies.
✖ Depends on the implementation of a *BasicMovingCharacter2D* due to snapping.
✖ You need to craft its movement by hand

> **Tip:**

> You can save the Animations on disk to create a set of pre-defined movements. This way you can load the movement when you want to use it again on another *MovingPlatform2D.*

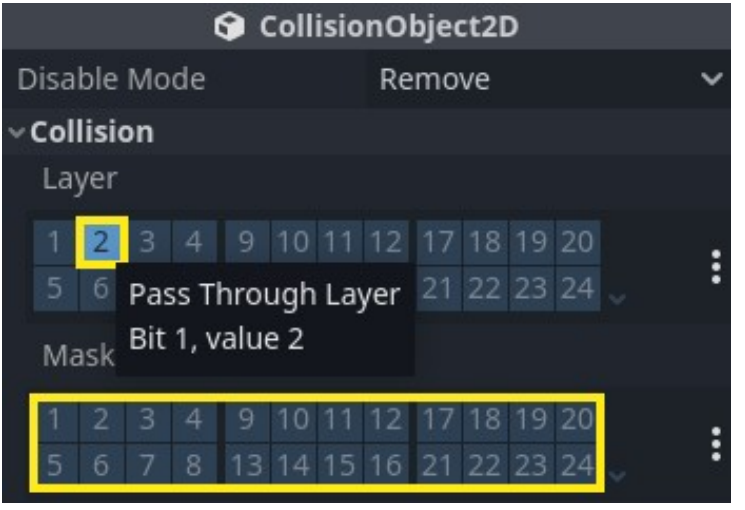
How to make this recipe?

Mise en Place:

1. Create a new scene with an ***AnimatableBody2D*** as root

> ⚠ **Warning:**
>
> Make sure the *Sync to Physics* property is toggled on

2. Rename it as *MovingPlatform2D*
3. Set its *Collision -> Layer* to be *Pass Through* only(optional)
4. Toggle off all its *Collision -> Mask* bits(optional)



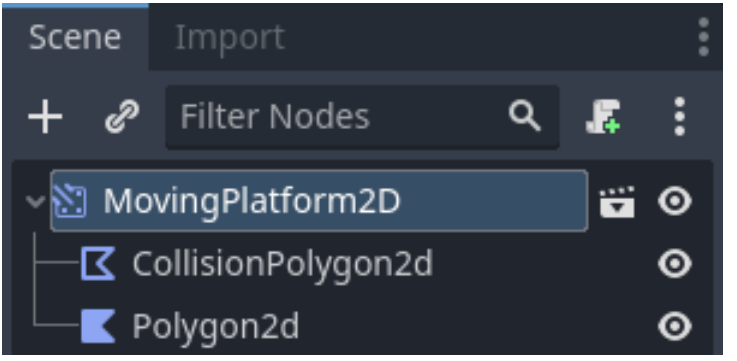
> **Tip:**
>
> Using this together with the ***PassThroughCharacter2D*** allows players to jump down the *MovingPlatform2D*.

> **Tip:**
>
> Set the *PassThroughCharacter2D → Moving Platform → On Leave* property to "Do Nothing" for snappy results.

5. Save the scene

Cooking it:

1. Create a new inherited scene using the *MovingPlatform2D* as the root node
2. Add a ***CollisionShape2D*** or a ***CollisionPolygon2D*** to it and draw its shape.
3. Add some graphics using ***Sprite2Ds***, ***AnimatedSprites2D***, or whatnot. In this case I used a ***Polygon2D***



4. Add an ***AnimationPlayer*** as its child
5. Set the *AnimationPlayer → Playback Options → Process Mode* to "Physics"
6. Create a new animation and animate the *Transform → Position* property according to your design

7. Open or create the scene you want to use the *MovingPlatform2D,* it can be your Level scene for instance
8. Add a new ***Node2D***
9. Add the *MovingPlatform2D* as its child

> **Tip**:
>
> Using a *Node2D* as parent for the *MovingPlatform2D* we ensure its movement is relative.
>
> This way we can reposition it across the level and the animation we've created will still work as intended.

Why does this recipe work?

Design-wise:

A core aspect of game design is to provide ever-harder challenges for players. This way we engage them and can test their skills.

A *MovingPlatform2D* adds a layer of difficulty to the game. Players will have to analyze the movement pattern to land on the platform.

This can escalate to many layers of combined difficulty as you stack *MovingPlatform2Ds*.

For instance, one *MovingPlatform2D* that carries the player from one cliff to another can become quite boring. But what if you add another one that moves at a different speed?

What about a lengthy travel where the player must jump from a *MovingPlatform2D* to another?

What if there are extra obstacles like floating pieces of land blocking the player's path.

With that, the player has to jump through them on top of following the *MovingPlatform2D*.

Engineering-wise:

Godot Engine has an amazing built-in physics simulation system. It also has one of the most feature-rich animation systems I know.

With these systems, Godot allows us to sync animations to the physics processing.

With the *AnimatableBody2D*, we can create animations that work on its position, rotation, scale, and other properties, such as its *Physics Material*.

Together with the *CharacterBody2D* ability to snap to other physic bodies, we can create an object that moves and carries the player's avatar.

And for that we just need, essentially, one line of code calling the `move_and_slide()` method on the *CharacterBody2D*.

Check the ***BasicMovingCharacter2D*** to understand this better.

With that, when the player's avatar jumps on a *MovingPlatform2D* Godot will calculate the *CharacterBody2D* final velocity taking into account the relative velocity of the *MovingPlatform2D*.

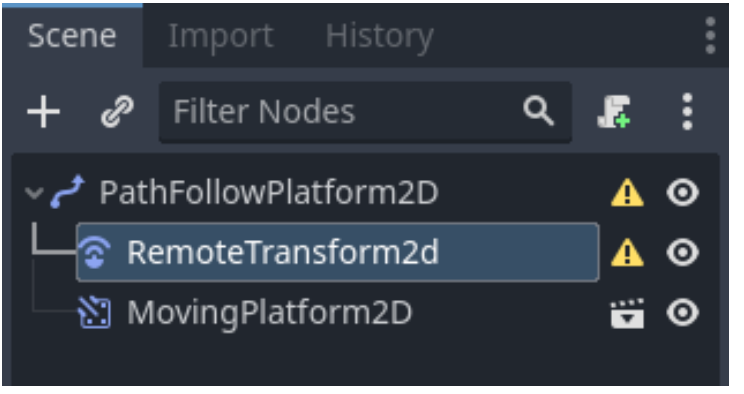
> **!! Secret Recipe found:**
>
> What's that? You just found the ***PathFollowPlatform2D*** recipe within this one!
>
> This is a special recipe that is essentially a variation of the *MovingPlatform2D* so we are going to skip directly to ***How to make this recipe?*** Section!

PathFollowPlatform2D

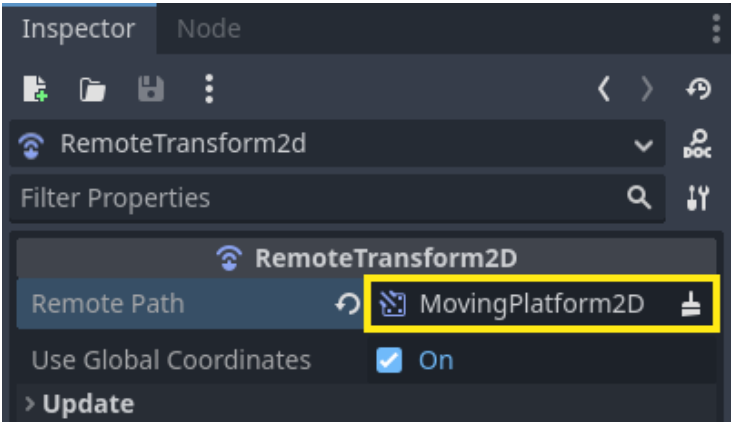
How to make this recipe?

Mise en Place:

1. Create a new scene using a ***PathFollow2D*** as the root node
2. Rename it as *PathFollowPlatform2D*
3. Toggle off the *PathFollow2D → Rotating* property (optional)
4. Toggle off the *PathFollow2D -> Loop* property (optional)
5. Add a ***MovingPlatform2D*** as a *PathFollowPlatform2D* child, here is better to use the concrete one you have after ***Cooking it***
6. Add a ***RemoteTransform2D*** as a *PathFollowPlatform2D* child



7. Set the *RemoteTransform2D → Remote Path* property to point to the *MovingPlatform2D*



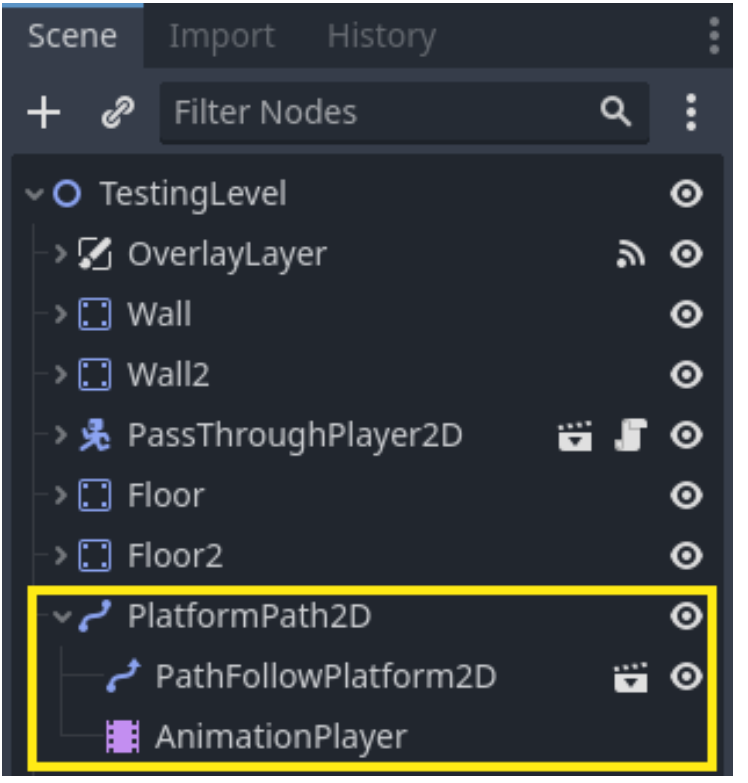
> ⚠ **Warning:**
>
> This is a workaround for a bug where the *AnimatableBody2D* doesn't update its position when its parent moves.
>
> You can follow the issue through the ***issue#54915*** report and the ***issue#58269*** as well. Both on the Godot Engine repository on GitHub.
>
> By the time you're reading, the developers may have fixed the issue already.

8. Save the scene

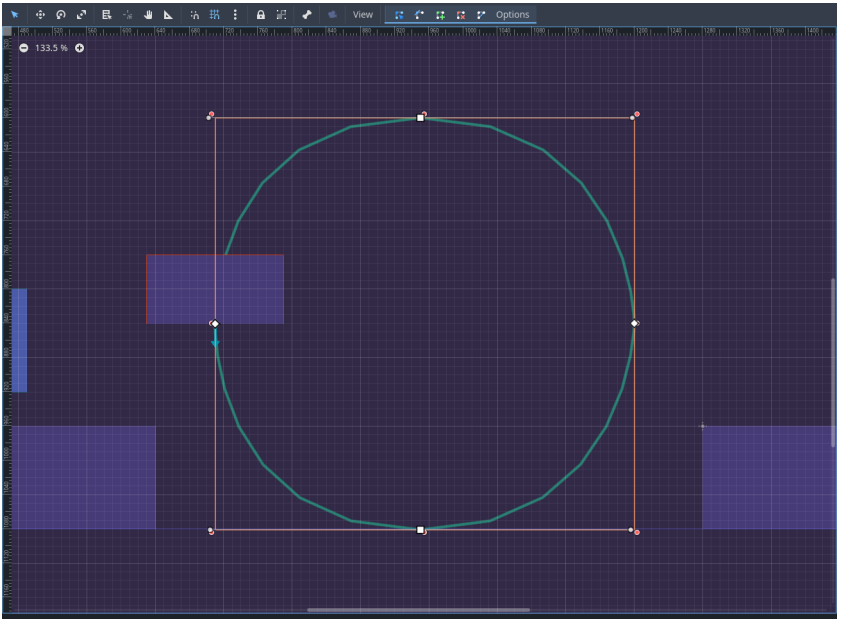
Cooking it:

1. Create a new scene, it can be the Level you're going to use the *PathFollowPlatform2D*
2. Add a ***Path2D*** node to the scene
3. Rename it as *PlatformPath2D* (optional)

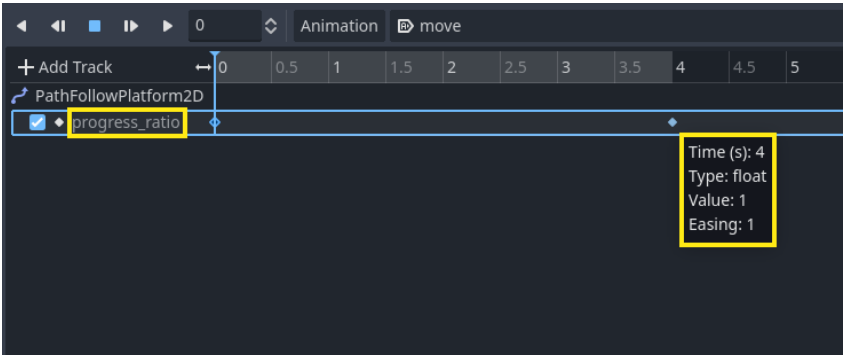
4. Add an instance of the *PathFollowPlatform2D* as a child of the *PlatformPath2D*
5. Add an ***AnimationPlayer*** as a child of the *PlatformPath2D*



6. Select the *PlatformPath2D* and draw the path you want the *PathFollowPlatform2D* to follow



7. Select the *AnimationPlayer* and create a new animation.
8. Animate the *PathFollow2D → Progress Ratio* from `0.0` to `1.0`

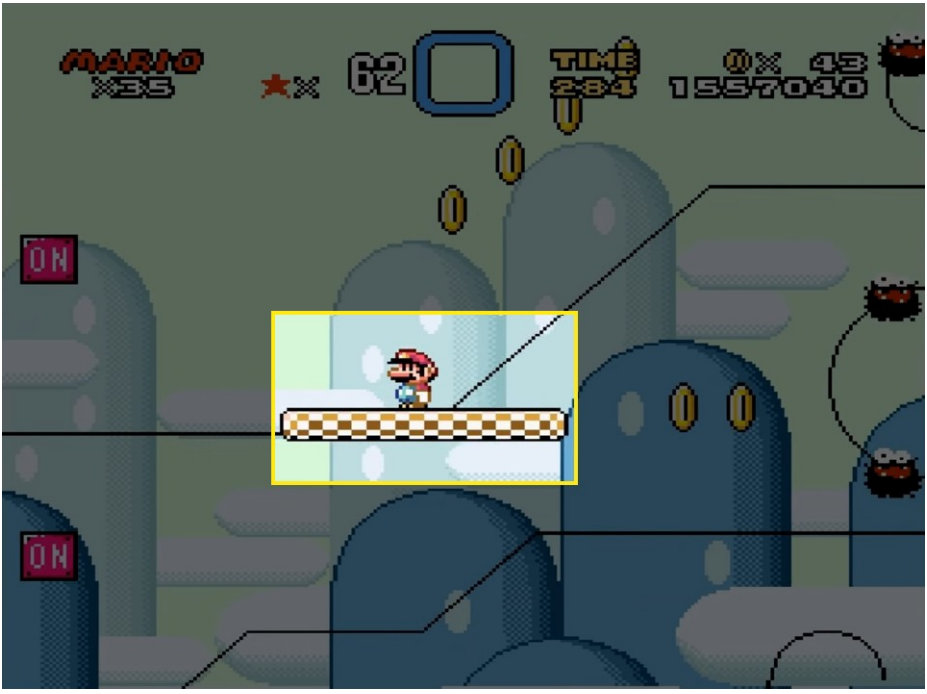


With that, you can easily design movement patterns for your *PathFollowPlatform2D*.

This tool helps you create unique designs for each game level.

For instance, what about some circular moving platforms? Or one that moves in waves? One that moves in a zigzag? You mean it!

In Super Mario World, they even designed a whole level using this recipe. The level is called Way Cool, you can check a whole playthrough ***in this video***.



Hazard2D

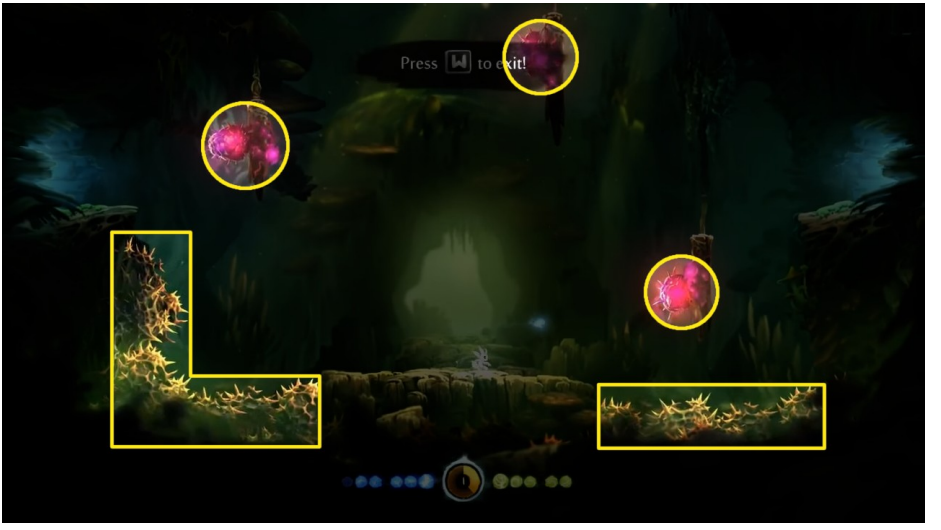
What is this recipe?

A *Hazard2D* is an object in the game world that inflicts damage, or kills, the player. Essentially, it's a hitbox with some kind of mechanic behavior.

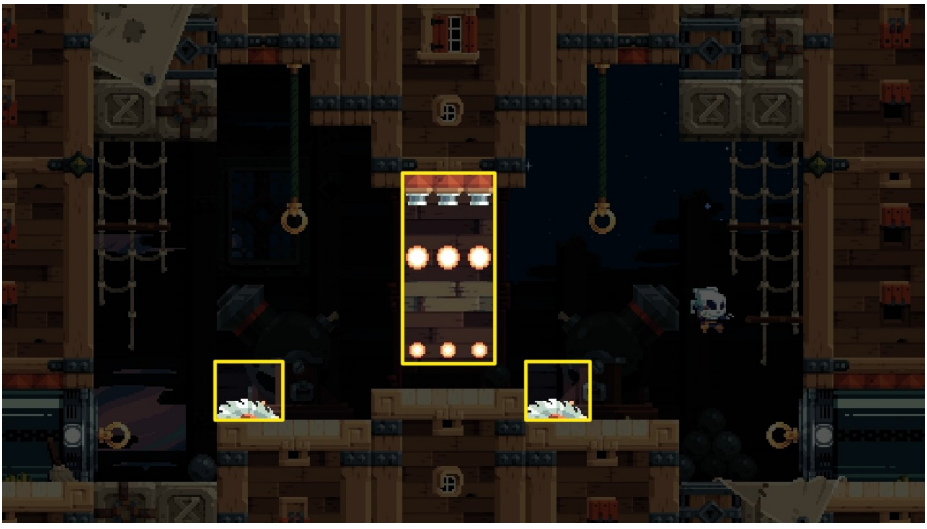
A *Hazard2D* doesn't have any kind of intelligence. Tho, we can add artificial intelligence to it, and it becomes an active enemy.

You can find this recipe everywhere, as some examples:

In Ori and the Blind Forest, a lot of the environment is made out of *Hazard2Ds.*



We can find this recipe on Flinthook as well, there are many mechanical gadgets around ships that are *Hazard2Ds.*



And, as a final example, in Hollow Knight, we can see them all over the world. Besides the game really focus on enemies and boss battles, *Hazard2Ds* play a major role in the world design as well.

When to use this recipe?

Obstacles are a fundamental aspect of a game. They are there to prevent the player from reaching the game's goal.

We use a *Hazard2D* as an obstacle that, if the player doesn't understand how it works, will punish the player.

Since games are systems of incentives, we must tell players when they are doing something good or bad.

When players do something that moves them in the direction of the game's goal, we reward them. When they do something that moves them away from the game's goal, we punish them.

So, whenever you want to add a sense of punishment for players' actions in your platformer game, a *Hazard2D* can be an option:

- A spike pit to test player's jump skills
- A sequence of pendulum blades to test player's slide skills
- Falling blocks to test player's timing skills

Pros & Cons:

✔ The recipe is flexible, we can use many "packages" for it, as mentioned above: spikes, blades, flames, flamethrowers, laser beams...
✔ Simple to implement.
✔ It helps improve the world-building of your game. We can create elements that are both interesting to the eyes, and that players can interact with.

❌ They need a fine-tuned level design to create interesting experiences.
❌ Relies on specific implementations of hit and hurt boxes.

How to make this recipe?

This is a two “dishes” recipe. In other words it’s a two-sided system.

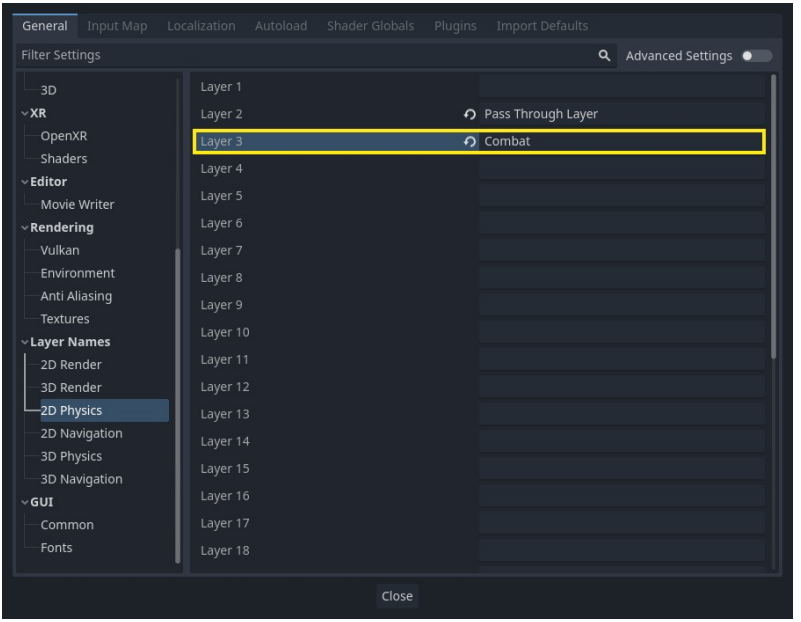
For it to work we need something that deals and something that “feels” damage.

So we are going to create both a *HitArea2D*, which will be our hit box. And we are also going to create a *HurtArea2D*, which is our hurt box.

Let’s start with the *HitArea2D*, which is the active side of the system.

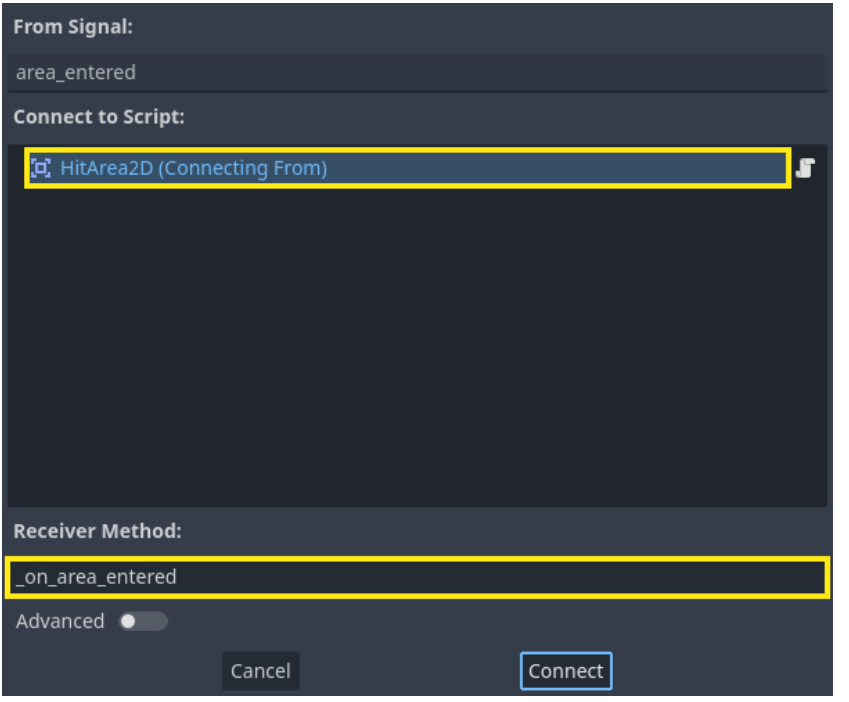
Mise en Place:

1. In the *Project -> Project Settings -> Layer Names -> 2D Physics* choose a layer and rename it as *Combat*. We are going to use it for every damage-dealing recipe moving on(optional)
2. Add an ***Area2D*** as the root node of a scene and rename it as *HitArea2D*



3. In the *Collision → Layer* property toggle off all layers
4. In the *Collision → Mask* property toggle off all layers, but the *Combat* one

5. Attach a *GDScript* to it
    i. Connect the `area_entered` signal to a signal callback method. In this case, I named it `_on_area_entered`
    ii. Create a new signal and name it `hit_landed`. It should pass an extra argument for the damage this *HitArea2D* deals



iii. Export a variable for the `damage`

iv. Export an enumerator variable with the “factions” or rather, teams, in your game. In this case I’ve used `Player` and `Not Player`

```
extends Area2D


signal hit_landed(damage)
```

```
@export  var  damage = 1

@export_enum ("Not Player", "Player")  var  team
 = 0
```

v. Create a method called `hit()`. It receives an argument to store the *HurtBox2D* it's interacting with
vi. Inside the `hit()` method, check if the *HurtArea2D* it's interacting with is **`not`** on the same team as it is.
vii. If so, emit the `hit_landed` signal. For the extra argument, we calculate the maximum value between `0` and the *HitArea2D*'s `damage` minus the *HurtArea2D*'s `defense`
viii. Finally, call the `get_hurt()` method on the *HurtArea2D* passing this *HitArea2D* as argument

```
func hit(hurt_area):

     if not hurt_area.team == team:

           hit_landed.emit(max(0, damage -
hurt_area.defense))

           hurt_area.get_hurt(self)
```

ix. Then, on the `_on_area_entered()` callback, call the `hit()` method passing the are as argument

```
func _on_area_entered(area2D):
	hit(area2D)
```

6. Save the scene

> **Spicing up:**
>
> You can add attributes to the *HitArea2D* and create a damage processing logic on the *HurtArea2D*.
>
> Damage types like pierce, crush, slash, and armor types like light, medium, and heavy can be a good start for that kind of mechanic.

This wraps up our *HitArea2D*, let's recap its whole script:

```
extends Area2D


signal hit_landed(damage)


@export var damage = 1
@export_enum("Not Player", "Player") var team = 0



func hit(hurt_area):
	if not hurt_area.team == team:
		hit_landed.emit(max(0, damage - hurt_area.defense))
		hurt_area.get_hurt(self)
```

```
func _on_area_entered(area2D):
     hit(area2D)
```

Ok, now let’s move on to the HurtArea2D. Which is quite a symmetric implementation. But this is the “passive” side of the system.

1. Add an *Area2D* as the root node of a scene and rename it as *HurtArea2D*
2. In the *Collision → Layer* property toggle off all layers, but the *Combat* one
3. In the *Collision → Mask* property toggle off all layers

4. Attach a new *GDScript* to it, then
   i. Create a signal called `hurt.` It should emit an extra argument for the damage dealt to it
   ii. Export a variable for its defense
   iii. Export an enumerator variable for its `team`. This should match the `team` enumerator on the *HitArea2D*

```
extends Area2D

signal hurt(damage)

@export var defense = 0
@export_enum("Not Player", "Player") var team = 0
```

iv. Create a method called `hurt()` that asks for a `hit_area` argument
v. In this method, check if the `hit_area` is `not` in the same `team` as this *HurtArea2D*
vi. If so, emit the hurt signal. As for the damage argument, calculate the maximum value between `0` and the difference between the *HitArea2D* `damage` and this *HurtArea2D* `defense`

```
func get_hurt(hit_area):
	if not hit_area.team == team:
		hurt.emit(max(0, hit_area.damage – defense))
```

And that wraps up our combat system. Let’s see the complete script, for reference:

```
extends Area2D

signal hurt(damage)

@export var defense = 0
@export_enum("Not Player", "Player") var team = 0



func get_hurt(hit_area):
```

```
    if not hit_area.team == team:
        hurt.emit(max(0, hit_area.damage - defense))
```

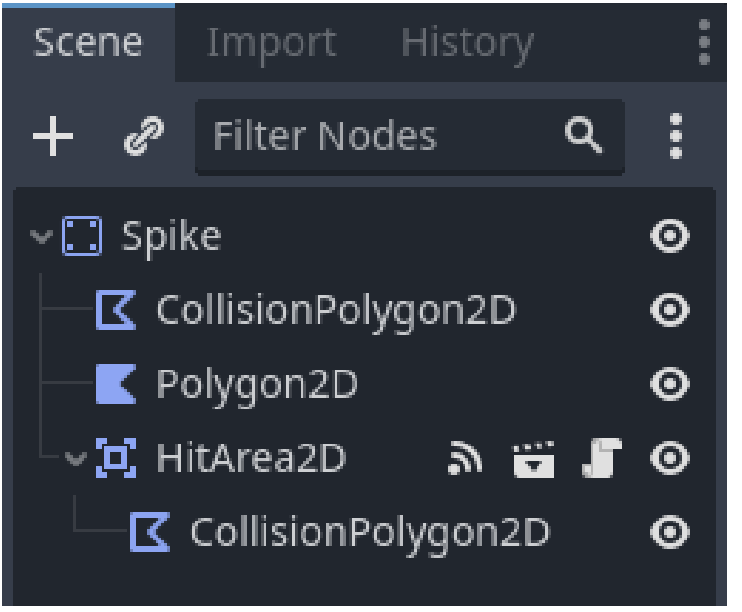
Now, to ***Cook it***, it's quite...abstract.

We can use this recipe in many ways. They all depend on the specific context of your game.

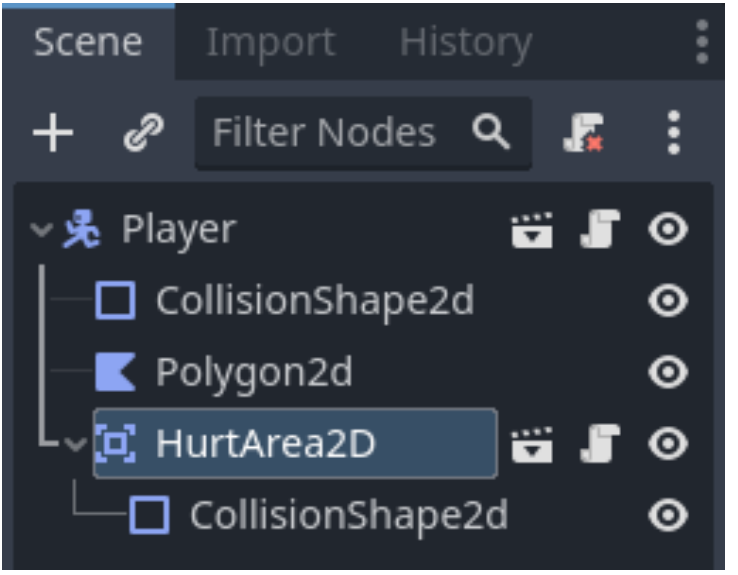
But the core idea is simple:

Attach a ***CollisionShape2D*** or a ***CollisionPolygon2D*** to the *HitArea2D* and the *HurtArea2D*. Make them be in opposite teams and make them overlap each other.

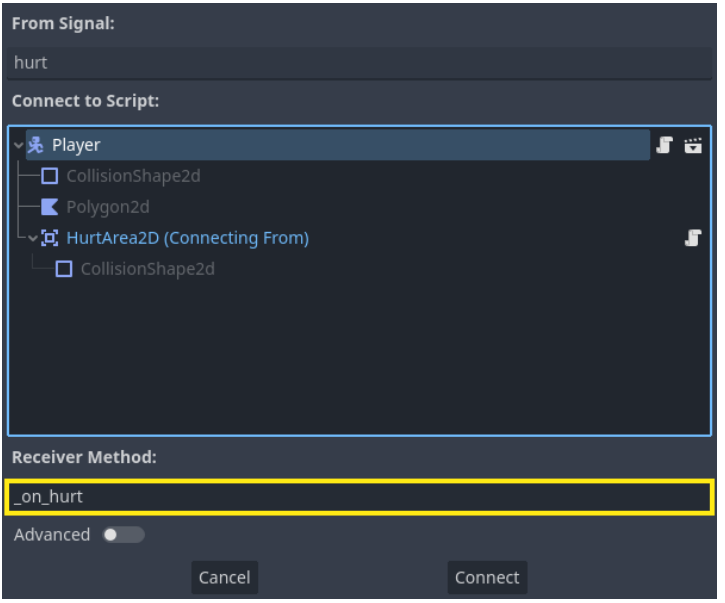
For instance, let's see how we can make some spikes to deal



7. Save the scene
8. Open your player scene. For instance, it can be a ***PassThroughCharacter2D***
9. Instance a *HurtArea2D* as its child
10. Set its `team` to “Player” and its `defense` to `0`
11. Add a *CollisionShape2D* or a *CollisionPolygon2D* as its child



12. Connect the *HurtArea2D*'s `hurt` signal to the *Player*'s script. You can name its callback `_on_hurt()`



13. Then, in the Player's script, setup the hurt logic. Here you can play animations, disable controls, decrease health… I'll straight up kill the player and restart the level

```
func _on_hurt(damage):
    get_tree().reload_current_scene()
```

Why does this recipe work?

Design-wise:

It's interesting to communicate to players that they did something wrong.

When we are talking about game design, we are talking about the incentives that players have to reach a goal.

Usually, game designers define these goals. For instance, to reach the end of the level.

One way to measure and communicate to players how well they are doing is through a health system.

The *Hazard2D* operates on that system by removing points from the player's health.

So, when they perform actions that move them away from the goal we apply damage. This makes players feel punished, so use it carefully.

Engineering-wise:

We take advantage of the Godot's ***Physics Layer*** system to create a separate dimension where only *HitArea2Ds* and *HurtArea2Ds* exist. This is why we don't have to make any duck typing or type checking.

Since only the *HitArea2D* checks for collisions using its *Collision Mask*, and only the *HurtBox2D* is in the actual *Collision Layer* we don't have to bother with *HitArea2Ds* “hitting” other *HitArea2Ds* and *HurtBox2Ds* "getting hurt" by other *HurtArea2Ds*.

BumpingEnemy2D

What is this recipe?

A ***BumpingEnemy2D*** it's a ***BasicMovingCharacter2D*** with a ***Hazard2D***. It core feature is that it has primitive intelligence.

It has a simple decision-making mechanism. It changes its movement direction when it bumps on a wall. This alone increases emergency in your platformer game.

Unlike *Hazard2Ds*, *BumpingEnemy2Ds* have different movements depending on the level's architecture.

It's an essential recipe to use in most platformer games due to its versatility.

When to use this recipe?

Unlike a hazard, an enemy has a personality and it may give the player a sense of antagonism in the game.

A hazard it's just there. It doesn't have will, drive, or goal.

An enemy may have some personality and goals. We can use other shenanigans like dialogues and sounds to express their intentions. They usually want to prevent players' progress.

So we often use a *BumpingEnemy2D* on levels with the purpose of a moving, sort of unpredictable, obstacle.

They add variation to hazards on a level. With the extra advantage of enhancing the perceived emergent gameplay from players' perspective.

Pros & Cons:

✔ Enhances emergent gameplay by creating a hazard with primitive intelligence
✔ Easy to put in place and extend
✔ Improves world design by adding antagonism to the story
✖ The primitive intelligence may hurt the player's perceived immersion

> **Tip**:
>
> We can mitigate that with more layers of intelligence. For instance, we can use a seeking behavior. You can take a look at the ***PathFollowEnemy2D*** recipe for reference.

✖ By default, it doesn't detect cliffs, which can cause weird behaviors.

> **Tip**:
>
> We can mitigate that with two ***RayCast2Ds*** that trigger a bump-like behavior.

✖ The movement pattern is very predictable

How to make this recipe?

Mise en Place:

1. Create a new inherited scene using the *BasicMovingCharacter2D* as the root node
2. Rename it as *BumpingEnemy2D*
3. Extend the script
   i. In the `_physics_process()` callback, make a super call passing the `delta`

```
class_name BumpingEnemy2D
extends BasicMovingCharacter2D



func _physics_process(delta):
	super(delta)
```

   ii. Check if the current movement collides with a wall using the `is_on_wall()` method. If it does, call the `bump()` method

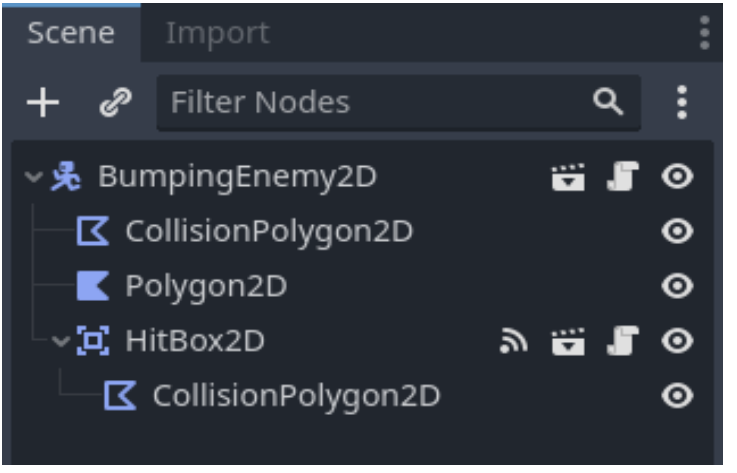
```
func _physics_process(delta):
	super(delta)
	if is_on_wall():
		bump()
```

iii. Create a new method called `bump()`. Inside this method, invert the current direction by multiplying it by `-1`

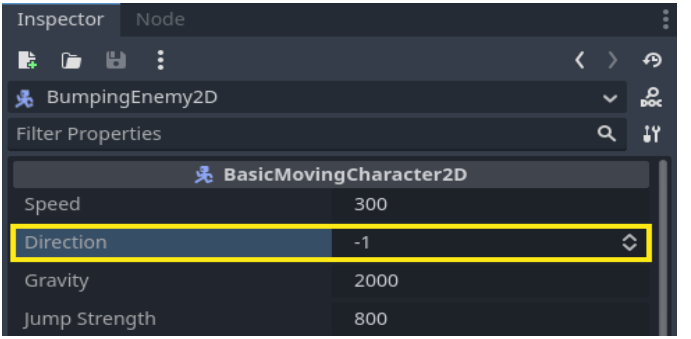
```
func bump():
    direction *= -1
```

Cooking it:

1. Create a new inherited scene using the ***BumpingEnemy2D*** you've created in the previous section
2. Add a ***CollisionShape2D*** or a ***CollisionPolygon2D*** and draw its shape
3. Add some graphics such as a ***Sprite2D*** or a ***Polygon2D***
4. Instance a ***HitBox2D*** as a *BumpingEnemy2D* child
5. Add or copy the *CollisionShape2D* or *CollisionPolygon2D* created in previous steps

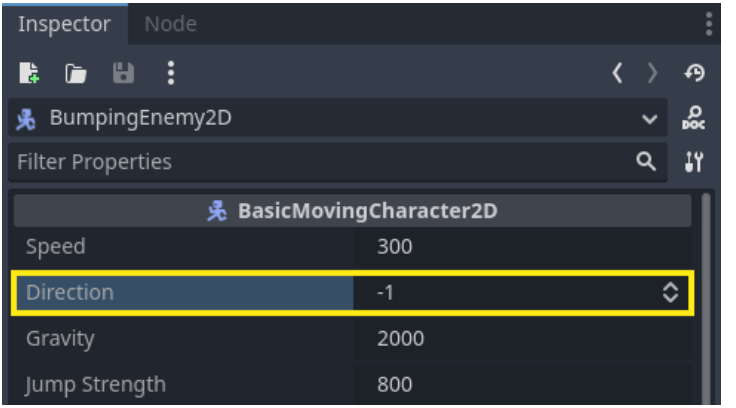


6. Add a ***VisibleOnScreenEnabler2D*** as a child of the



BumpingEnemy2D

7. Using the *Inspector* set the *Direction* property to `1` or `-1` depending on the direction you want the *BumpingEnemy2D* to start moving



8. Save the scene.

Congratulations!! You've created your first **concrete** *BumpingEnemy2D*!!

> **Spicing up!**
>
> Add two ***RayCast2Ds*** on each edge of *BumpingEnemy2D* pointing downwards. Then, in the `_physics_process()`, check if they are **not** colliding with the floor. If so, call the `bump()` method.

Why does this recipe work?

Design-wise:

When we only have hazards, objects without life, the player may feel like we are testing them like a robot.

The world design and storytelling can mitigate that feeling. But players can feel like that when obstacles don't act with purpose.

Using a *BumpingEnemy2D* we add that extra spice to levels.

Different from a ***Hazard2D***, *BumpingEnemy2Ds*' behavior changes depending on the level's architecture.

Using this ability, we can make the players feel like there are different *BumpingEnemy2Ds* on the same level.

For that, we can play with the distance between one wall and another and also with the enemies' aesthetics.

Engineering-wise:

Here we rely on Godot's built-in physics to handle collisions for us.

Godot detects horizontal collisions as wall collisions. It does that because we use a vector pointing upwards as the *CharacterBody2D → Up Direction*.

This collision detection has to do with the collision normal. Any collision that has a normal with a value of `1.0` or `-1.0` in the horizontal axis will be a wall collision.

To detect that we use the *CharacterBody2D.is_on_wall()* method.

Wait...what's that?

> **!! Secret Recipe found!**
>
> You found the ***Stomping & StompableObject2D*** recipe!
>
> This is a special recipe that relies on the ***BasicMovingCharacter2D*** and the ***BumpingEnemy2D*** so we are going to skip directly to the ***How to make this recipe?*** section!

Stomping & StompableObject2D

A *StompingObject2D* is an object, usually a ***BasicCharacter2D*** with the ability to stomp. It detects collisions when it falls on a floor and if it collides with a *StompableObject2D* it stomps it.

A *StompableObject2D* is an object that is inside the Stomping Objects physics layer.

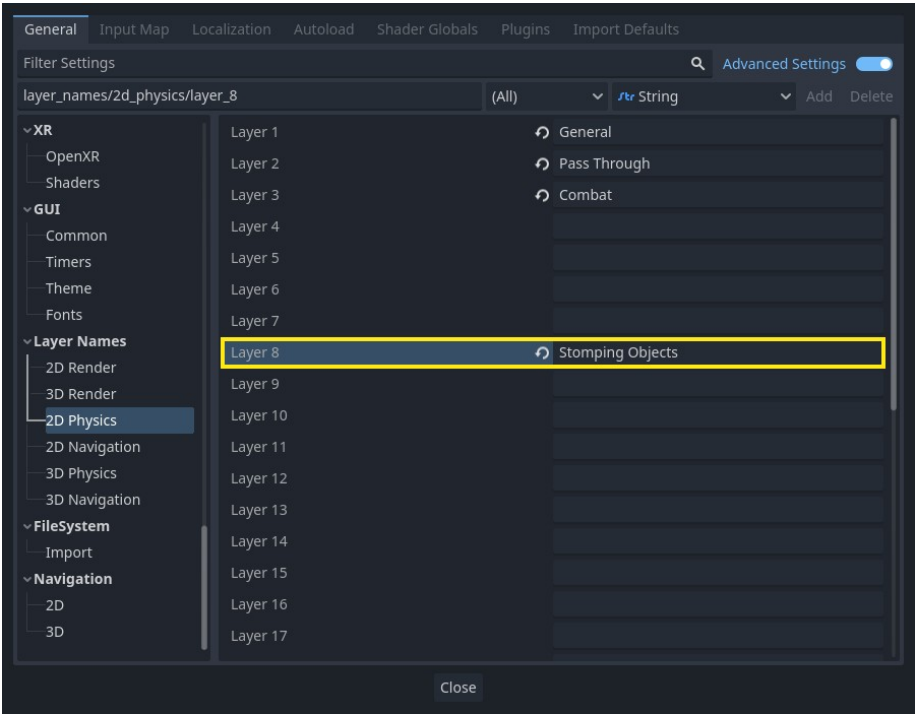
How to make this recipe?

Mise en Place

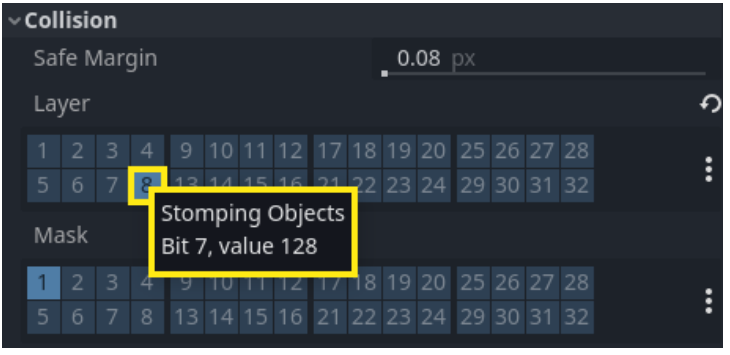
First, let's create a *StompableEnemy2D*.

StompableEnemy2D

1. Create a new inherited scene using the ***BumpingEnemy2D*** as root node.
2. Rename it as *StompableEnemy2D*
3. On *Project → Project Settings → Layer Names → 2D Physics* pick a layer and name it *Stomping Objects*



4. Toggle on the *Stomping Objects* layer bit on the *StompableEnemy2D > Collision Layer*



Now for the StompingObject2D, it's a bit more complex.

StompingObject2D

1. Create a new inherited scene using a ***BasicMovingCharacter2D*** or a ***PassThroughCharacter2D*** as root node.
2. Rename it as *StompingCharacter2D*
3. Toggle on the *Stomping Objects* layer on the *StompingObject2D → Collision Mask* so it can detect the *StompableEnemy2D*

4. Extend the *BasicMovingCharacter2D* script and open the new one
   i. Export a variable for the stomp impulse. This is similar to the jump, but we do that so we have more control over what's a jump and what's a stomp

```
class_name StompingCharacter2D
extends BasicMovingCharacter2D


@export  var  stomp_impulse = 1000.0
```

ii. Export a variable for the stomp threshold speed. We use that so that if the character is falling below this speed, it won't stomp the enemy. You can leave it as `0.0` if you rather consider any fall a stomp

```
@export  var stomp_threshold_speed = 500.0
```

iii. Export a variable to choose what's the physics layer you use for stomping objects, in our case we use the layer with value of `128`, the eight layer.

```
@export_flags_2d_physics var stomping_layer = 128
```

iv. Create a variable to store the falling speed. We store this information in a dedicated variable because the `velocity.y` is `0.0` after the *BasicMovingCharacter2D* hits the floor

```
var falling_speed = 0.0
```

v. In the `_physics_process()` callback, make a `super()` call passing the `delta` as argument. This will make sure the *BasicMovingCharacter2D* logic is still rolling in the background

```
func _physics_process(delta):
    super(delta)
```

vi. Check if the *StompingCharacter2D* is on floor and if it is check if the falling speed is greater or equal the

stomp threshold speed, if it does, call the `stomp()` method we are going to create soon

vii. If the *StompingCharacter2D* is not on the floor, set its falling speed to be the current vertical velocity. The `_physics_process()` should look like this at this point:

```
    if is_on_floor():
        if falling_speed >= stomp_threshold_speed:
            stomp()
        else:
            falling_speed = velocity.y
```

viii. Create the `stomp()` method

ix. Inside the `stomp()` method, create a variable to store the collider of the last slide collision. To understand that better, check out the ***KinematicCollision2D official documentation***.

```
func stomp():
    var collider = get_last_slide_collision().get_collider()
```

x. Create a variable to store the the layer number of our stomping layer. Currently we have a bit value. But to check if this layer is active in our collider, we need to convert it. You can check ***this article*** about how to convert bit values to base `2` values. We sum `1`

because we start counting collision layers from `1` instead of from `0`

```
var  stomping_layer_number =
 ( log (stomping_layer) /  log (2)) + 1
```

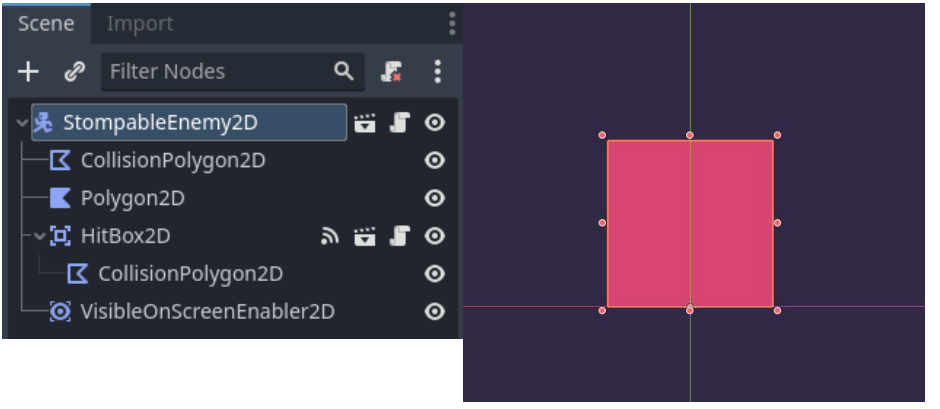
xi. Check if the stomping collision layer is active in the collider using the `get_collision_layer_value()` method
xii. If the stomping collision layer is active, call the `queue_free()` method, if you have a custom method to kill the enemy, you can call it instead using duck typing
xiii. Set the falling speed to `0.0` to reset it for further checks iterations
xiv. Set the `velocity.y` axis to be the negative of the stomping impulse so the character moves upwards when stomping. The complete `stomp()` method should look like this:

```
func stomp():

     var collider =
get_last_slide_collision().get_collider()

     var stomping_layer_number =
(log(stomping_layer) / log(2)) + 1

     if
collider.get_collision_layer_value(stomping_layer
_number):
```

```
            collider.queue_free()
            falling_speed = 0.0
            velocity.y = -stomp_impulse
```

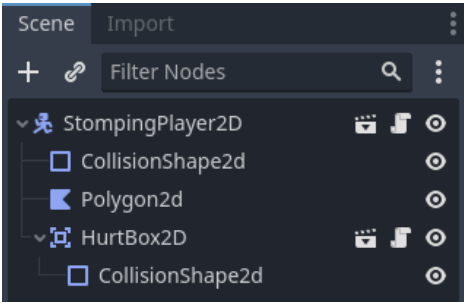
Cooking it:

1. Create a new inherited scene using the *StompableEnemy2D* as root
2. Add a ***CollisionShape2D*** or a ***CollisionPolygon2D*** as its child
3. Add some graphics, such as a ***Sprite2D*** or, like in this case, a ***Polygon2D***
4. Instance a ***HitBox2D*** and add a *CollisionShape2D* or a *CollisionPolygon2D* as its child
5. Add a ***VisibleOnScreenEnabler2D*** and enclose the *StompableEnemy2D* within its rect



6. Save the scene

7. Create a new inherited scene using the *StompingCharacter2D* as root, we are going to make our *StompingPlayer2D* based on it
8. Add a *CollisionShape2D* or a *CollisionPolygon2D* as its child
9. Add some graphics, such as a *Sprite2D* or, like in this case, a *Polygon2D*
10. Instance a ***HurtBox2D*** and add a *CollisionShape2D* or a *CollisionPolygon2D* as its child



11. Extend the script and add all the necessary input handling inside the `_unhandled_input()` callback

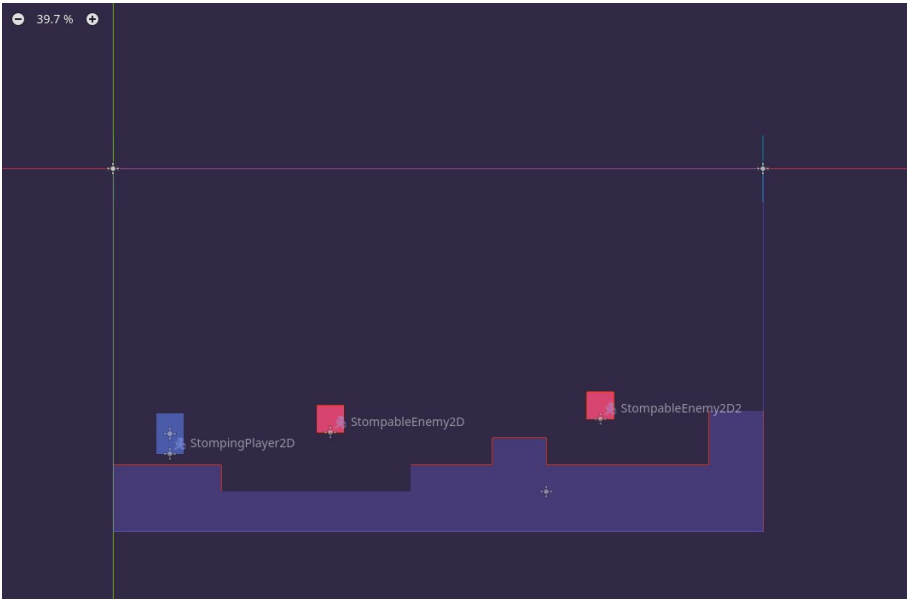
```
class_name StompingPlayer2D
extends StompingCharacter2D


@export var move_left_action = "move_left"
@export var move_right_action = "move_right"
@export var jump_action = "jump"
@export var stomp_impulse_penalty = 0.5
```

```
func _unhandled_input(event):
     # Horizontal movement
     if event.is_action(move_left_action):
           if event.is_pressed():
                 direction = -1
           elif
Input.is_action_pressed(move_right_action):
                 direction = 1
           else:
                 direction = 0
      elif event.is_action(move_right_action):
           if event.is_pressed():
                 direction = 1
           elif
Input.is_action_pressed(move_left_action):
                 direction = -1
           else:
                 direction = 0
     # Vertical movement
     if event.is_action_pressed(jump_action):
           jump()
     elif event.is_action_released(jump_action):
           cancel_jump()
```

12. Override the `stomp()` method to set the vertical velocity to a diminished version based on the above stomp impulse penalty if the player is not holding the jump action

```
func stomp():
	super()
	if not Input.is_action_pressed(jump_action):
		if velocity.y < 0:
			velocity.y = -(stomp_impulse * stomp_impulse_penalty)
```

13. Create a nice level and put them all together!



14. You can connect the player's *Hurtbox2D* `damaged` signal to the root node to reset the level when enemies hit the player

PathFollowEnemy2D

What is this recipe?

I usually distinguish between an object and an actor if the actor has some kind of intelligence.

In other words, an actor can make decisions based on the game world. In that sense, an enemy is an actor.

The *PathFollowEnemy2D* is an enemy that has adapts its behavior based on the following possible states:

- Wandering
- Seeking
- Returning

Most of the time, the *PathFollowEnemy2D* is wandering around, following its predefined movement path.

Once the player enters its sight, it seeks the player and deals damage once it hits the player. When the player moves away from the *PathFollowEnemy2D* sight, it returns back to its original position. After that, it starts wandering around again.

This is an essential type of enemy to have in platformer games to spice them up and increase their difficulty.

When to use this recipe?

We use a *PathFollowEnemy2D* to add antagonism to the game's world design. ***Hazard2Ds*** are obstacles, but they aren't characters.

The player rarely feel like the world's hazards are against them. But enemies have the solo purpose of preventing players from reaching their goals.

In that sense, a *PathFollowEnemy2D* is an enhanced version of a *Hazard2D*.

We can use a *PathFollowEnemy2D* to increase the perceived difficulty of a level when players learned how to deal with simple *Hazard2Ds*.

Using a *PathFollowEnemy2D* we guarantee some emergent gameplay. As they respond to player's behaviors, creating new and distinct outcomes from each interaction.

Pros & Cons:

✔ Enhances world design adding decision making to enemies

> **Tip:**
>
> We can leverage on this by adding animations, sound effects, war songs, onomatopoeia texts, etc...

✔ Adds depth to the gameplay with dynamic behavior patterns instead of static patterns

✔ Opens a range of storytelling possibilities since we can theme each *PathFollowEnemy2D* according to each level's theme

❌ Their wandering behavior is poor and easily predictable
❌ May decrease perceived difficulty due to simplistic intelligence
❌ Players may understand them as a hazard capable of following the player

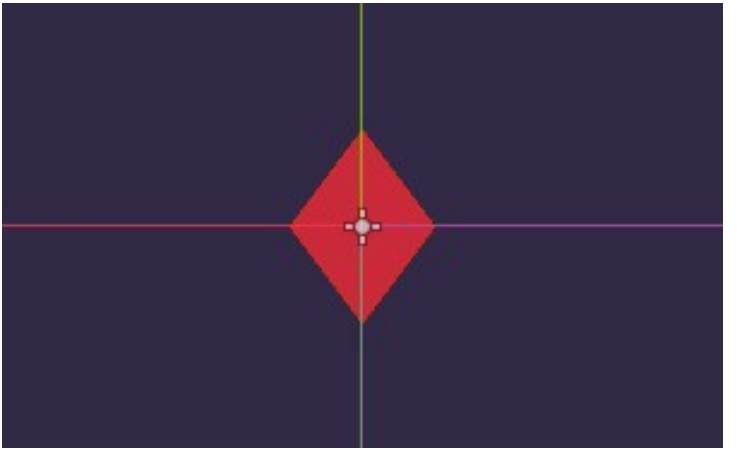
How to make this recipe?

This is quite a complex recipe. It involves some basic artificial intelligence that seeks the player.

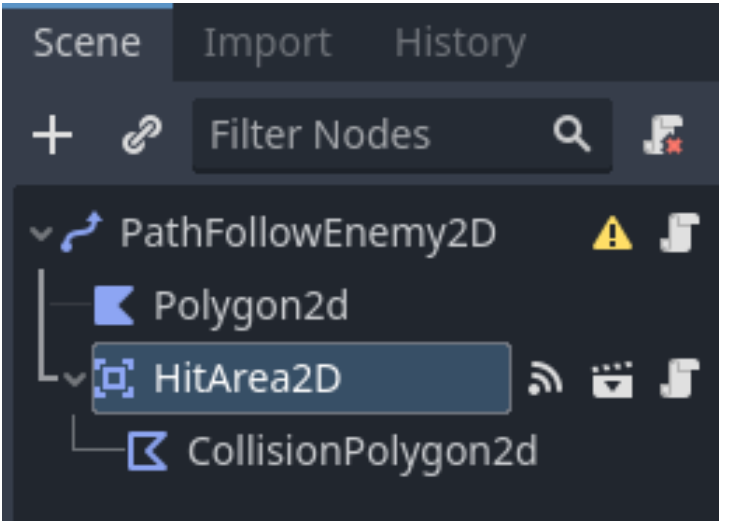
So let's start with the enemy itself. Then we'll finish it up with what the player's avatar needs to have in order for this recipe to work.

Mise en Place:

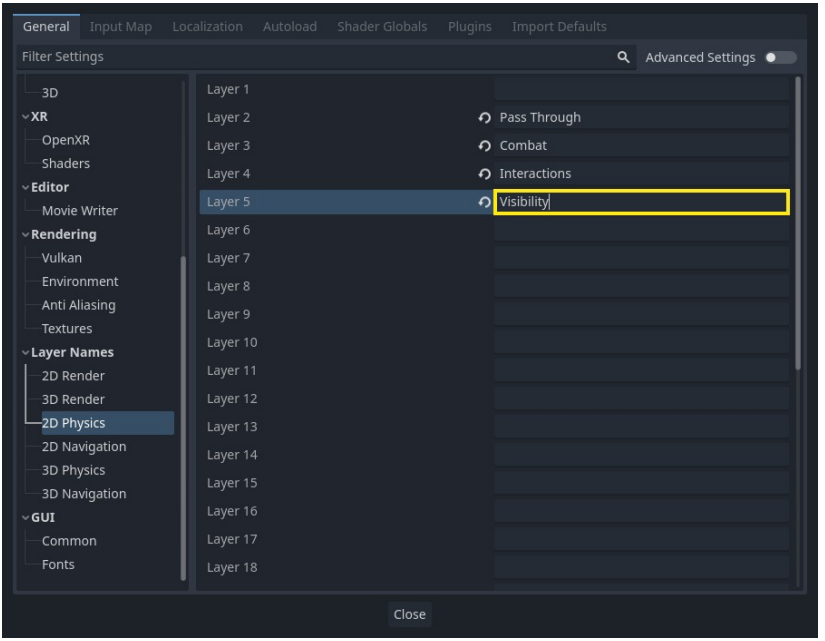
1. Create a new scene using a ***PathFollow2D*** as root node
2. Rename it as *PathFollowEnemy2D*
3. Add some graphics, in this case I used a ***Polygon2D*** to draw a diamond shape, but you can use ***Sprite2Ds*** or ***AnimatedSprite2Ds*** as well



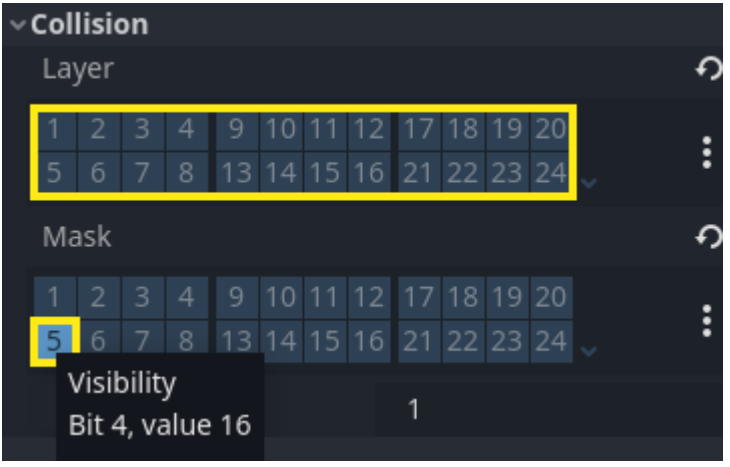
4. Add a ***HitArea2D*** as it child
5. Add a ***CollisionShape2D*** or a ***CollisionPolygon2D*** to represent the damaging area



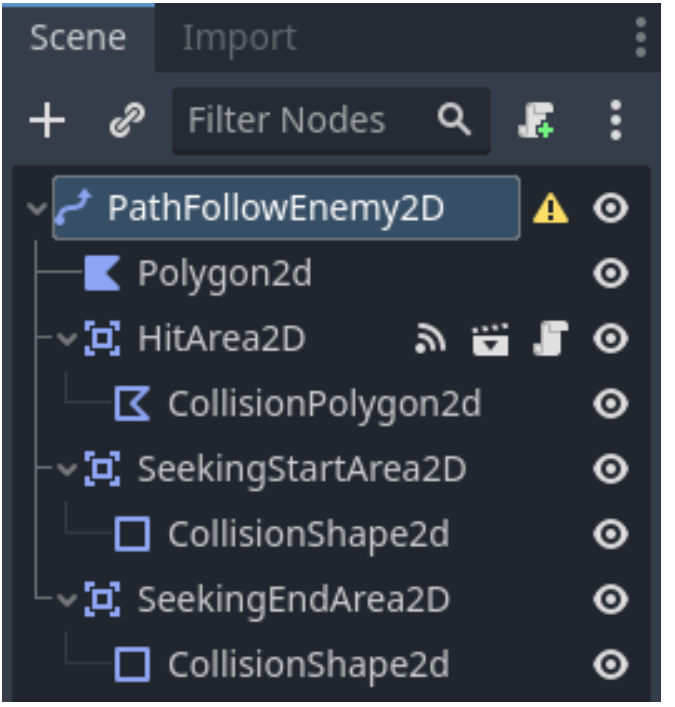
6. Add an ***Area2D*** as a *PathFollowEnemy2D* child and rename it as *SeekingStartArea2D*
7. Go to *Project → Project Settings → Layer Names → 2D Physics*, choose a new layer and rename it as *Visibility* (optional)

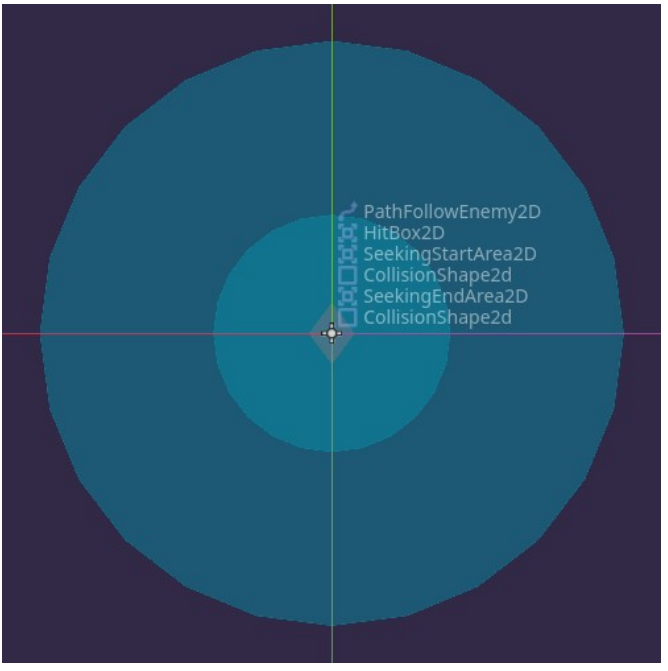


8. Toggle off all the *SeekingStartArea2D → Collision Layers*
9. Toggle on the *Visibility* in the *SeekingStartArea2D → Collision Masks* property

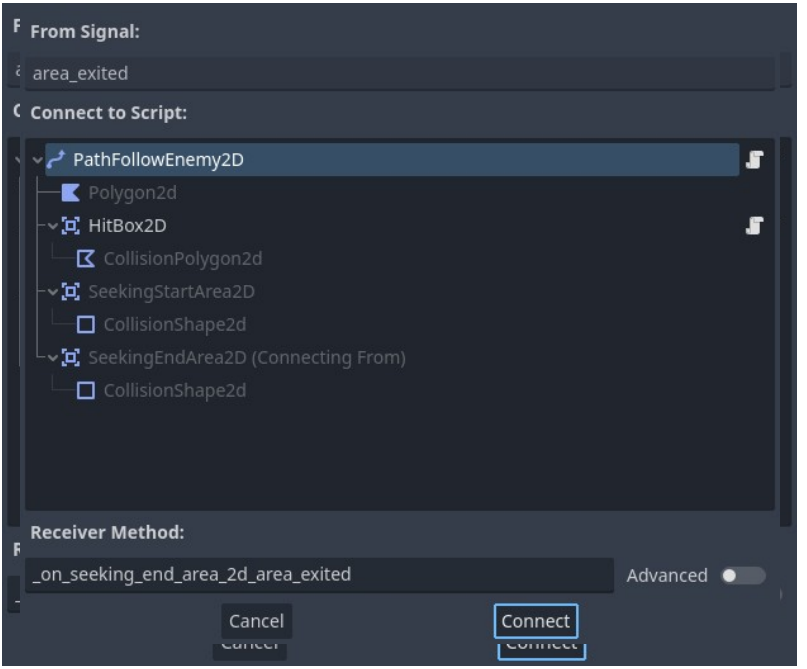


10. Add a *CollisionShape2D* as a *SeekingStartArea2D* child
11. Create a new *Shape2D* to represent the enemy's sight threshold, I recommend a ***CircleShape2D***
12. Add another *Area2D* as a *PathFollow2D* child and rename it as *SeekingEndArea2D*, set its *Collision Layers* and *Collision Masks* the same as the *SeekingStartArea2D*
13. Add a *CollisionShape2D* as a *SeekingEndArea2D* child
14. Create a new *Shape2D* to represent the enemy's sight limit, I recommend a *CircleShape2D*. The scene setup should look like this:





15. Attach a new *GDScript* to the *PathFollowEnemy2D*
16. Connect the *SeekingStartArea2D* → `area_entered` signal to the *PathFollowEnemy2D*
17. Connect the *SeekingEndArea2D* → `area_exited` signal to the *PathFollowEnemy2D*



Now, let's dive into the script to define its behavior. But first, let's set everything up so we can dig into each individual state.

Setup

1. Create a constant to store the distance threshold.

```
const DISTANCE_THRESHOLD = 5.0
```

> **Tip:**
>
> We use that to make the *PathFollowEnemy2D* stop moving when it reaches its original position.

2. Create an enumerator to store the three possible states the *PathFollowEnemy2D* can be in

```
enum STATES {WANDERING, SEEKING, RETURNING}
```

3. Export a variable for the movement speed

```
@export var speed = 300.0
```

4. Create a pseudo private variable to store the target object the *PathFollowEnemy2D* should seek, by default it should be `null`

```
var _target = null
```

5. Then, create a pseudo private variable to store the current state this *PathFollowEnemy2D* is in. By default, it should be in the wandering state

```
var _state = STATE.WANDERING
```

6. Create a pseudo private variable to store the original position the _PathFollowEnemy2D_ was before seeking the player

```
var _original_position = Vector2.ZERO
```

7. Finally, create some pseudo private variables for the movement logic. One thing to point out is that the `_path_direction` is used to make a *ping pong* for movement paths that aren't loopable

```
var _direction = Vector2.RIGHT
```

```
var _path_direction = 1
```

At this point the script should look like this:

```
extends PathFollow2D


const DISTANCE_THRESHOLD = 5.0
enum STATES {WANDERING, SEEKING, RETURNING}


@export var speed = 300.0


var _target = null
var _state = STATES.WANDERING
var _original_position = Vector2.ZERO
var _direction = Vector2.RIGHT
var _velocity = Vector2.ZERO
var _path_direction = 1
```

Now, let's move on to our first behavior, the *WANDERING*.

Wandering behavior

In the wandering state, the *PathFollowEnemy2D* basically moves along its parent's *Path2D's Curve*.

1. Override the `_physics_process` callback and set a match statement using the `_state` variable.

```
func _physics_process(delta):
	match _state:
```

2. When in the wandering state, we increment the *Progress* property using the movement speed and the path direction

```
	STATE.WANDERING:
		progress += (speed * _path_direction)
* delta
```

3. Then, if the *Loop* property is toggled off, we check if the *Progress Ratio* is above or equal to `1.0` and change the `_path_direction`. This means the *PathFollowEnemy2D* reached the end of the path, so it should move back.

```
		if not loop:
			if progress_ratio >= 1.0:
			_path_direction = -1
```

4. If the *Progress Ratio* is below or equal to `0.0`, meaning we reached the beginning of the path, we change the `_path_direction` back to `1`

```
		elif progress_ratio <= 0.0:
			_path_direction = 1
```

The wandering behavior is done, at this point the script looks like that:

PathFollowEnemy2D

```
extends PathFollow2D

const DISTANCE_THRESHOLD = 5.0
enum STATE {WANDERING, SEEKING, RETURNING}

@export var speed = 300.0

var _target = null
var _state = STATE.WANDERING
var _original_position = Vector2.ZERO
var _direction = Vector2.RIGHT
var _velocity = Vector2.ZERO
var _path_direction = 1



func _physics_process(delta):
     match _state:
           STATE.WANDERING:
                 progress += (speed * _path_direction)
* delta
                 if not loop:
                       if progress_ratio >= 1.0:
                             _path_direction = -1
                       elif progress_ratio <= 0.0:
```

```
                              _path_direction = 1



func _on_seeking_start_area_2d_area_entered(area):
     pass



func _on_seeking_end_area_2d_area_exited(area):
     pass
```

Now let's tell what happens when the *PathFollowEnemy2D* is seeking the player.

Seeking behavior

1. Still in the current match statement, when in the seeking state, check if there is a target. If so, set the `_direction` to be direction from the *PathFollowEnemy2D* towards the target global position

   ```
   STATE.SEEKING:
        if _target:
              _direction =
   global_position.direction_to(_target.global_posit
   ion)
   ```

2. Update the _velocity using the _direction and the speed values

```
     _velocity = _direction * speed
```

The seeking state behavior is ready! Let's take a look on how the script looks like at the moment:

```
extends PathFollow2D

const DISTANCE_THRESHOLD = 5.0
enum STATE {WANDERING, SEEKING, RETURNING}

@export var speed = 300.0

var _target = null
var _state = STATE.WANDERING
var _original_position = Vector2.ZERO
var _direction = Vector2.RIGHT
var _velocity = Vector2.ZERO
var _path_direction = 1



func _physics_process(delta):
     match _state:
          STATE.WANDERING:
               progress += (speed * _path_direction)
* delta
```

```
                if not loop:
                      if progress_ratio >= 1.0:
                            _path_direction = -1
                      elif progress_ratio <= 0.0:
                            _path_direction = 1
          STATE.SEEKING:
                if _target:
                      _direction =
global_position.direction_to(_target.global_position)
                      _velocity = _direction * speed
                      translate(_velocity * delta)


func _on_seeking_start_area_2d_area_entered(area):
     pass


func _on_seeking_end_area_2d_area_exited(area):
     pass
```

When the seeking ends the *PathFollowEnemy2D* should move back to its original position.

And when it's close enough to the original position, it starts wandering again. Let's create this returning logic now.

Returning behavior

1. Still in the current match statement, when in returning state, set the `_direction` to be the direction from the current global position towards the original position.

```
            STATE.RETURNING:
                  _direction =
global_position.direction_to(_original_position)
```

2. Update the `_velocity` to match the new direction the *PathFollowEnemy2D* should move towards

```
                  _velocity = _direction * speed
```

3. Check if the distance to the original position is equal to or below the distance threshold. If it is, change the current state back to wandering

```
                  if
global_position.distance_to(_original_position)
<= DISTANCE_THRESHOLD :
                        _state = STATE.WANDERING
```

And, it's done.

We can finally use the translate() method whenever the PathFollowEnemy2D is either seeking or returning.

```
          STATES.RETURNING, STATES.SEEKING:
                translate(_velocity * delta)
```

Now we have all the three behaviors ready! The script should look like this:

```
extends PathFollow2D

const DISTANCE_THRESHOLD = 5.0
enum STATES {WANDERING, SEEKING, RETURNING}

@export var speed = 300.0

var _target = null
var _state = STATES.WANDERING
var _original_position = Vector2.ZERO
var _direction = Vector2.RIGHT
var _velocity = Vector2.ZERO
var _path_direction = 1



func _physics_process(delta):
     match _state:
           STATES.WANDERING:
                 progress += (speed * _path_direction)
* delta
                 if not loop:
                       if progress_ratio >= 1.0:
```

```
                              _path_direction = -1
                        elif progress_ratio <= 0.0:
                              _path_direction = 1
            STATES.SEEKING:
                  if _target:
                        _direction =
global_position.direction_to(_target.global_position)
                        _velocity = _direction * speed
            STATES.RETURNING:
                  _direction =
global_position.direction_to(_original_position)
                  _velocity = _direction * speed
                  if
global_position.distance_to(_original_position) <=
DISTANCE_THRESHOLD :
                        _state = STATES.WANDERING
            STATES.RETURNING, STATES.SEEKING:
                  translate(_velocity * delta)



func _on_seeking_start_area_2d_area_entered(area):
     pass



func _on_seeking_end_area_2d_area_exited(area):
```

```
    pass
```

Notice that, we don’t have any way to transition to the seeking nor the returning states.

This is because we are going to use the *SeekingStartArea2D* and the *SeekingEndArea2D* signal callbacks for that.

Seeking start and Seeking end

Finally, let's tell what happens when the player enters and exits the seeking areas. This is what triggers the seeking behavior and the returning behavior.

1. Inside the `_on_seeking_start_area_2d_area_entered()` callback, check if the current state is the wandering state, then if it is, we set the `_original_position` to be the current global position

   ```
   func _on_seeking_start_area_2d_area_entered(area):
       if _state == STATE.WANDERING:
           _original_position = global_position
   ```

2. Outside this if statement, set the _target to become the area that just entered in the *SeekingStartArea2D*

   ```
   _target = area
   ```

3. Change the current state to be the seeking state

```
_state = STATE.SEEKING
```

> **Tip:**
>
> This is an excellent place to play an animation. You can make an ❗ appear on top of the enemy's head and play a sound effect before the *PathFollowEnemy2D* starts seeking the player.

4. Inside the _on_seeking_end_area_2d_area_exited() set the _target back to `null`

```
func _on_seeking_end_area_2d_area_exited(area):
	_target = null
```

5. Since there's no target, change the current state to be the returning state

```
	_state = STATE.RETURNING
```

The final version of the script should look like this:

```
extends PathFollow2D


const DISTANCE_THRESHOLD = 5.0
enum STATES {WANDERING, SEEKING, RETURNING}


@export var speed = 300.0
```

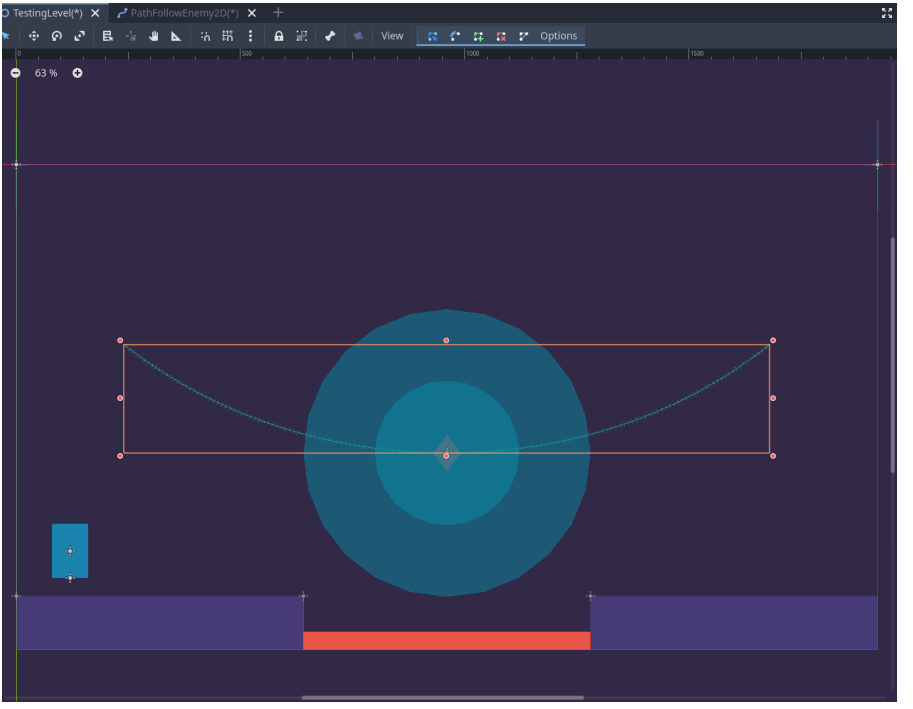
```
var _target = null
var _state = STATES.WANDERING
var _original_position = Vector2.ZERO
var _direction = Vector2.RIGHT
var _velocity = Vector2.ZERO
var _path_direction = 1



func _physics_process(delta):
     match _state:
          STATES.WANDERING:
               progress += (speed * _path_direction)
* delta
               if not loop:
                    if progress_ratio >= 1.0:
                         _path_direction = -1
                    elif progress_ratio <= 0.0:
                         _path_direction = 1
          STATES.SEEKING:
               if _target:
                    _direction =
global_position.direction_to(_target.global_position)
                    _velocity = _direction * speed
          STATES.RETURNING:
```

```
                _direction =
global_position.direction_to(_original_position)
                _velocity = _direction * speed
                if
global_position.distance_to(_original_position) <=
DISTANCE_THRESHOLD :
                      _state = STATES.WANDERING
          STATES.RETURNING, STATES.SEEKING:
                translate(_velocity * delta)



func _on_seeking_start_area_2d_area_entered(area):
     if _state == STATES.WANDERING:
          _original_position = global_position
     _target = area
     _state = STATES.SEEKING



func _on_seeking_end_area_2d_area_exited(area):
     _target = null
     _state = STATES.RETURNING
```

Now, we can finally cook it and see how we can use it in our games.

Cooking it

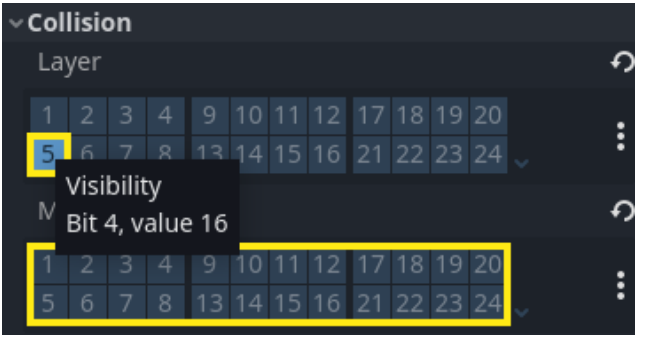
1. Create a new scene, it can be the level scene you are going to use a *PathFollowEnemy2D*
2. Add a ***Path2d*** to the scene
3. Rename it as *EnemyPath2d*(optional)
4. Use the path drawing interface to create a new movement path for your *PathFollowEnemy2D*
5. Instantiate a *PathFollowEnemy2D* as a child of the *EnemyPath2d*
6. In the *Path2d → Curve* property, create a new *Curve* and draw the *PathFollowEnemy2D's* movement path



Done, with that you have a working *PathFollowEnemy2D* to play with your player.

To make it react to players, you can use an *Area2D*. Toggle off all its *Collision Layers* and *Collision Masks* bits, and toggle on only the Visibility layer on the *Collision Layers* property.

You can call this *Area2D*, *VisibilityArea2D*.



Why does this recipe work?

Design-wise:

Working with a pre-defined path for the enemy's movement provides a sense of predictability.

Players love to recognize patterns, and as such we provide some for them.

Adding some level of artificial intelligence increases the player's perception of the game world. It's not just an object that performs a behavior. It has some kind of decision making process. It also has memory, as it moves back to where it stopped its wandering movement.

By using themed graphics for these enemies you create the idea that the enemies are characters, antagonists. Players can't blame a blade that swings brainlessly. If they fail, it's their fault.

But they can create a grunge against something that has a face and the clear goal of preventing the player's progress.

Engineering-wise:

We rely on Godot's *Area2d* and use it as sensors for our artificial intelligence. So it can detect when the player enters and leaves its sight.

With a *PathFollow2D* and a *Path2D* we can visually design movement patterns for the *PathFollowEnemy2D*.

> **Tip**:
>
> Note that since the *Path2D → Path* property is easily accessible through the Inspector we can animate it as well.
>
> So it is possible to create complex behaviors and movement patterns.
>
> For instance, you can change the movement path for a while when the enemy just finished the returning behavior.
>
> This can make it look like it is patrolling an area before moving back to its wandering path.

Using the builtin *Vector2* methods we calculate the directions we need for the *PathFollowEnemy2D* movement.

This way, we don't need to perform the calculations ourselves, this cut off a lot of time and prevents some bugs as well.

The *PathFollowEnemy2D* uses a pseudo state machine to control its states.

This is just a high level abstraction of its internal states. With that, we encapsulate each possible behavior on a match statement branch. With that, we have self-contained behaviors and prevents bugs since the behaviors logic don't overlap on each other.

The *PathFollowEnemy2D* only does what each of its states tells it to do.

InteractiveArea2D

What is this recipe?

An *InteractiveArea2D* is an area in the game world that players can interact when touching it. They can optionally use an input for the interaction.

Some examples of interactive objects are:

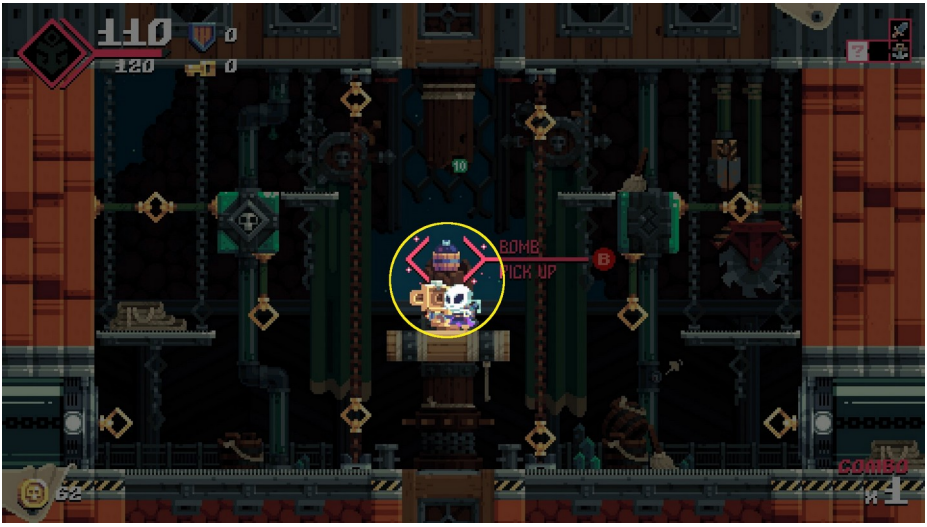
- Treasure chests
- Signs with warnings
- NPCs with dialogues
- Buttons, switches, and triggers

Using an *InteractiveArea2D* you enrich the game world. They allow players to explore and interact with the game world on a deeper level.

For instance, we can find them in *Hollow Knight* when players trigger a dialogue when they reach a given area.

In *Cuphead*, players can walk around in a world map to choose the level they are going to play. These are also *InteractiveArea2Ds.*

When you enter a room in Flinthook, you may find items to pick, NPCs to chat, shops to buy stuff, and also treasure chests to open. All of those are *InteractiveArea2Ds*.



When to use this recipe?

This is one of those recipes that goes well in almost every situation. Ultimately, is up to you to dictate when **not** to use it.

Anytime you want players to interact with your game's world, use it.

For instance, use it when players reach a cliff that leads to a lava lake. You can display an ‼ sign and allow players to press a button to interact with a wood sign. This can trigger a dialogue box explaining the story behind the lava lake. You can even give hints to players on how to overcome it.

Pros & Cons

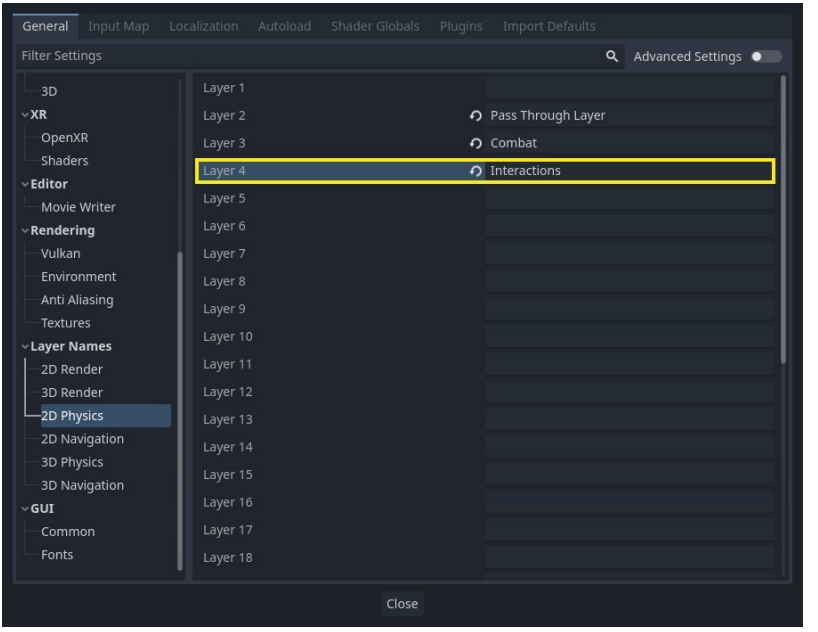
✔ Enriches the game world allowing players to actively explore it.
✔ Easy to setup, you can add it at any time on the production of the game.
✔ Expands the game design since anything can become an active object in the game world.
✖ Requires a dedicated Physics Layer.
✖ Works better with a dedicated Input Action that doesn't conflict with any other.
✖ Works better with other systems, like a dialogue system.

How to make this recipe?

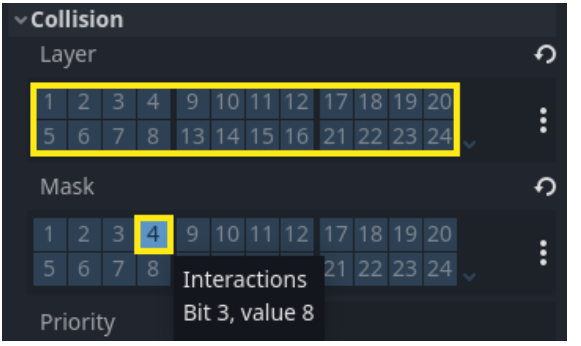
As a general purpose recipe, the *InteractiveArea2D* has a quite simple setup. Most of its value comes with what we do with it after that.

Mise en Place:

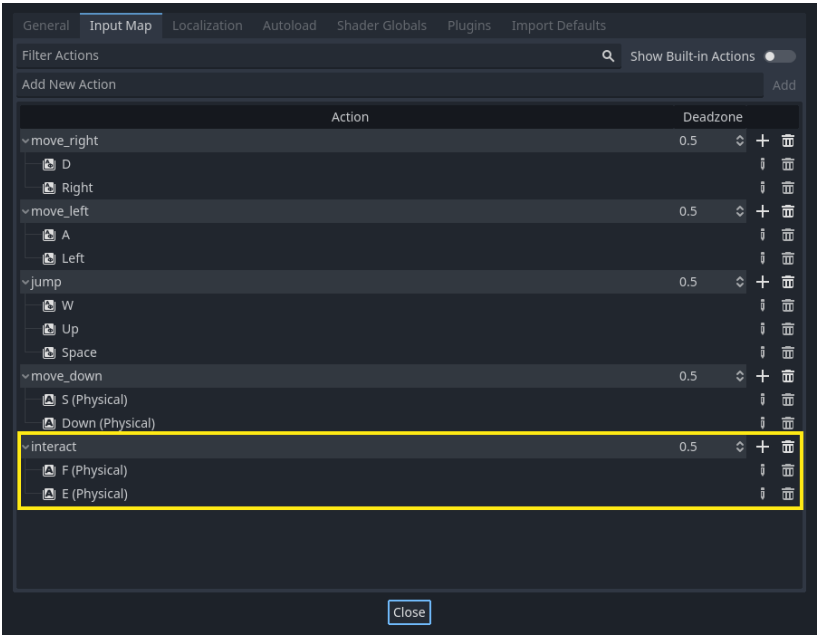
1. In *Project* → *Project Settings* → *Layer Names* → *2D Physics* choose a layer and rename it as *Interactions*(optional)



2. Add an ***Area2d*** as the scene's root node
3. Rename it as *InteractiveArea2D*
4. Disable all it's *Collision Layers* and enable the *Interactions* layer in the *Collision Mask*



5. In *Project → Project Settings → Input Map* create a new action to trigger interactions



6. Attach a *GDScript* to it
   i. Open the script and create some signals to communicate interaction events such as when the

player interacted and if they can or can't interact with this *InteractiveArea2D*

```
extends Area2D


signal interacted
signal interaction_available
signal interaction_unavailable
```

ii. Export a variable to store the name of the *InputAction* related to interactions

```
@export var interact_input_action = "interact"
```

iii. Disable the `_unhandled_input()` callback process in the `_ready()` callback

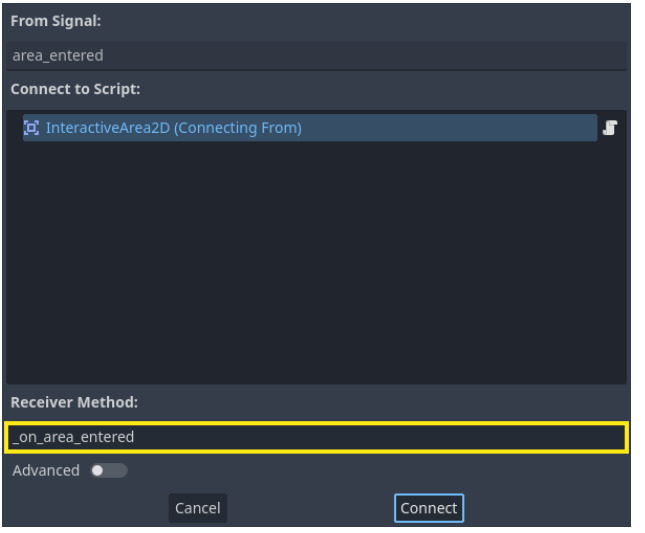
```
func _ready():
	set_process_unhandled_input(false)
```

> **Tip:**
>
> This prevents this object from checking player's inputs when this object isn't supposed to. We are going to enable it when an interaction is available.

iv. Using the `_unhandled_input()` callback, check if the `interact_input_action` was pressed, if so, emit the `interacted` signal and consume the *InputEvent*

```
func _unhandled_input(event):
    if event.is_action_pressed(interact_input_action):
        interacted.emit()
        get_viewport().set_input_as_handled()
```

v. Connect the `area_entered` and the `area_exited` signals and create their respective callbacks, in this case `_on_area_entered()` and `_on_area_exited()`



vi. In the `_on_area_entered()` callback, enable the `_unhandled_input` process and emit the `interaction_available` signal

```
func _on_area_entered(_area):
	set_process_unhandled_input(true)
	interaction_available.emit()
```

vii. In the `_on_area_exited()` callback, disable the `_unhandled_input` process and emit the `interaction_unavailable` signal

```
func _on_area_exited(_area):
	set_process_unhandled_input(false)
	interaction_unavailable.emit()
```

At the end, the script should look like this:

```
extends Area2D

signal interacted
signal interaction_available
signal interaction_unavailable

@export var interact_input_action = "interact"
```

```
func _ready():
     set_process_unhandled_input(false)



func _unhandled_input(event):
     if
event.is_action_pressed(interact_input_action):
           interacted.emit()
           get_viewport().set_input_as_handled()



func _on_area_entered(_area):
     set_process_unhandled_input(true)
     interaction_available.emit()



func _on_area_exited(_area):
     set_process_unhandled_input(false)
     interaction_unavailable.emit()
```

InteractionArea2D

Since this is a two-sided system you need to create an interaction area on player so they can trigger this behavior.

1. Create a new scene with an *Area2D* as root node

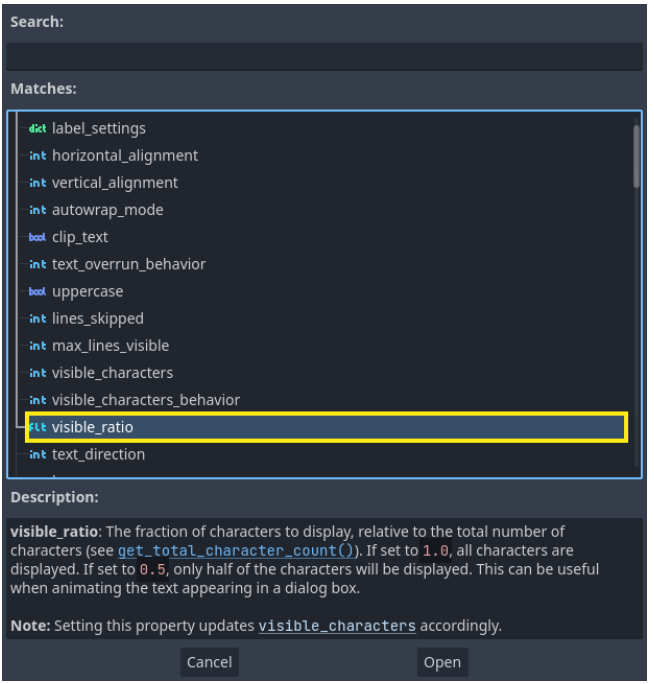
2. Rename as *InteractionArea2D*
3. Toggle off all its *Collision Layer* and *Collision Mask* bits
4. Toggle on only the *Interactions Collision Layer* bit

Done.

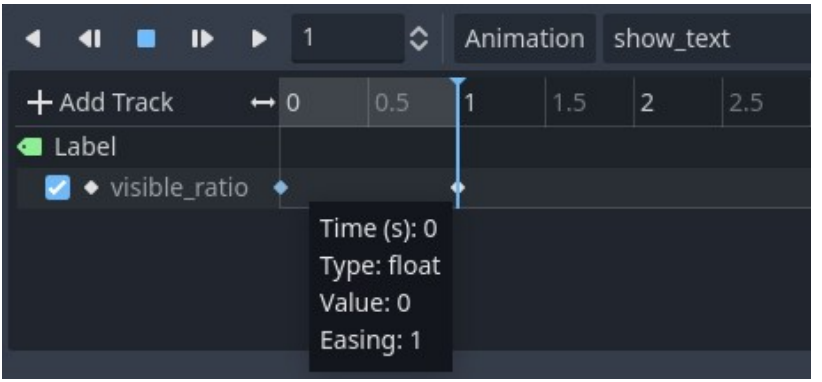
Cooking it:

Now, to cook this recipe you will need some creativity. Let's see how to use it to create an NPC that displays a simple dialogue.

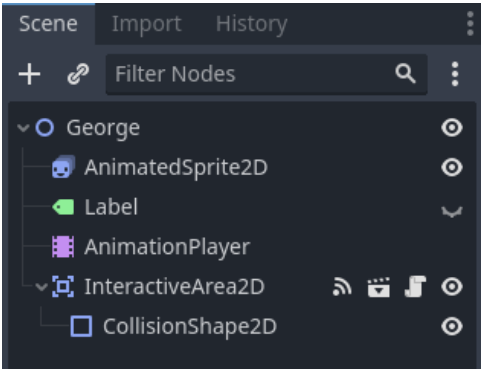
1. Create a new scene using a ***Node2D*** as root
2. Let's name it *George*
3. Add some graphics to George, you can use ***Sprite2Ds*** or, like in this case, ***AnimatedSprite2D***
4. Add a ***Label*** to George and in the Text property write down what he'll say, like "Hello, world!"
5. Then, add an ***AnimationPlayer*** and create a new animation called show_text
6. In the animation, create a new Property Track for the *Label → Visible Ratio*



7. Insert a new key and set its value to `0.0`
8. Then, at the end of the animation, insert a new key and set its value to `1.0`



9. Create a new track for the *Label* → *Visible* property and set its value to `true`
10. Toggle off the *Label* → *Visible* property. It should only show after the animation starts
11. Create an instance of the *InteractiveArea2D* as a child of George
12. Add a ***CollisionShape2D*** to it and set up its *Shape* property

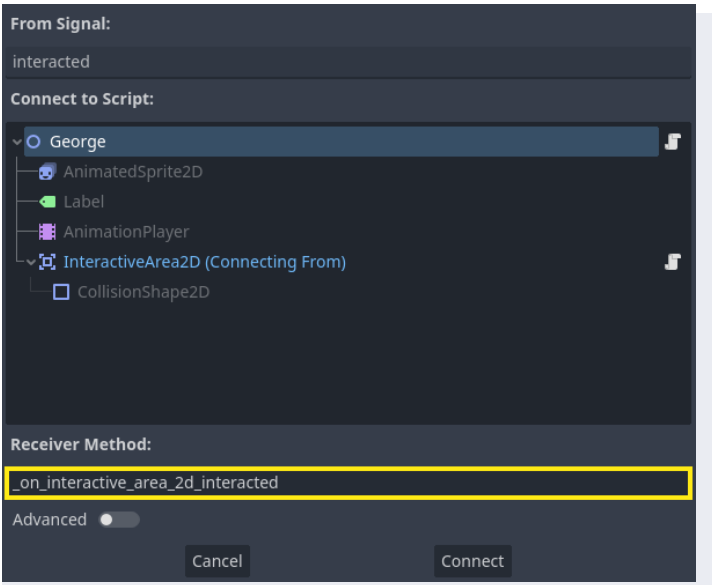


13. Attach a GDScript to George
    i. Create an `onready` reference for the *Label* and for the *AnimationPlayer*

```
extends Node2D


@onready var label = $Label

@onready var animation_player =
$AnimationPlayer
```

    ii. Connect the *InteractiveArea2d* `interacted` signal to a callback in *George*

iii. Inside this callback, play the `show_text` animation




```
func _on_interactive_area_2d_interacted():
    animation_player.play("show_text")
```

iv. Connect the *InteractiveArea2d* `interaction_unavailable` signal to a callback in *George*

v. Inside this callback, hide the *Label*

```
func _on_interactive_area_2d_interaction_unavailable():
    label.hide()
```

With that, the complete script should look like this:

```
extends Node2D


@onready var label = $Label
@onready var animation_player = $AnimationPlayer


func _on_interactive_area_2d_interacted():
	animation_player.play("show_text")



func _on_interactive_area_2d_interaction_unavailable():
	label.hide()
```

If you attach an InteractionArea2D to your player and test out *George*, he should pop up a “Hello world!” text

Design-wise:

To create a rich experience for the player to immerse, we need a world that they can explore at their own will.

To do that, the world players are playing can't be just a bunch of graphics and collisions. It needs to be an adventure full of interactions that they can experiment with.

By using the _InteractiveArea2D_, we create an area that players can interact with.

This area can trigger a range of interactions:

- A chest that the player can open
- A flower that the player's character can comment about
- A sign that the character can read
- A person to talk with, an object to pick
- A door to enter

This way, the game goes beyond pixels and numbers. It turns into a living interactive media. One that we can use to tell a story and let players explore this story at their own pace.

Engineering-wise:

Since the *Area2d* node is capable of detecting other overlapping *Area2ds*. We use this ability to detect when the player's character is overlapping with the *InteractiveArea2D*. If so, we trigger events based on that overlapping.

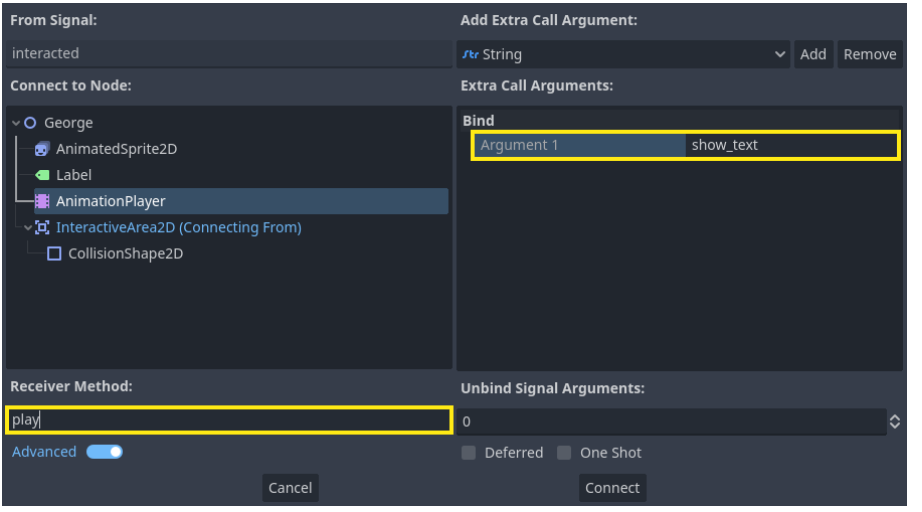
The `_unhandled_input()` logic is only available when the player is overlapping with the *InteractiveArea2D*. So we can make all the input handling logic self-contained within this node.

This also allows each instance of the *InteractiveArea2D* to handle the input on its own.

We've created a new layer for the `area_entered` and the `area_exited` signals by using the `interaction_available` and `interaction_unavailable`.

With that, the signals don't need extra arguments. This way, we can connect the signals directly to the Nodes methods.

For instance, George's script isn't necessary. We can play animations without using intermediary methods just using signals and extra call arguments.



Portal2D

What is this recipe?

A *Portal2D* is an ***InteractiveArea2D*** that teleports players to another place in the game world.

It doesn't need to be a literal portal, like a wormhole. It can be a door that teleports the player inside a house, the entrance to a cave, or a spot on the world map.

For instance, in *Flinthook* we have those pipes that players can hook.

When they do so, the character plays an animation and the game teleports the payer to the respective room.

Notice how smart the developers were.

Using this cookbooks recipes abstractions, we can analyze this mechanic like that:

We could add the ***InteractionArea2D*** to the grappling hook tip object.

When players shoot the hook, the game creates an instance of the hook's tip with an *InteractionArea2D*.

When this *InteractionArea2D* overlaps with the pipe's ***Portal2D***, it triggers the teleport behavior:

The player character plays the animation, the game loads the next scene.

When to use this recipe?

Usually, we use a *Portal2D* to create a sense of discrete progression. To explain it in other words:

We have continuous world progression and discrete world progression.

As an example, picture that:

The player reaches the edge of the screen.

If the game has a following camera, its canvas will show there's more to explore in that direction.

We can say that, in this situation, the game has continuous world progression.

Discrete continuity is different.

It means that two world areas, for instance, two levels, are only connected through a special type of transportation.

To move from one to another you need special transport.

Usually, there’s a teletransport to move between them. It’s like they were in two distinct universes.

Check out ***discrete math*** to better understand this abstraction. Also, other games and movies are great references too.

As a concrete example, in *Cuphead*, levels are discretely connected.

They are part of the same world. But to move from one to another you have to finish one. Then, change to a level selection screen. Then, you can select the next level.

The "transport" in this case is the level selection screen.

In contrast, most open-world games have continuous levels.

Besides being an action western RPG, in the *Elder Scrolls V: Skyrim* it’s a good example.

There, the game's main quest guides you through ever harder levels. Tho, the level areas are seamlessly connected and part of the same open world.

But when you open a door to a house, a *Portal3D* teleports you inside the house. The house interior is a separate area from the game world.

Pros & Cons:

✔ Enhances world design by allowing players to explore new locations of the world.
✔ Adds a layer of adventure as the game world itself becomes an open place for players to explore.
✔ Adds depth to levels by allowing players to go beyond the surface of what the screen shows. For instance, entering houses or hidden areas.
✖ May break the game's flow if players expect seamless world exploration, like in Metroidvanias.

How to make this recipe?

Let's start this recipe by creating a ***Singleton*** to hold some important data between one teleport and another.

Mise en Place:

TeleportData

1. Create a new scene using a ***Node*** as root
2. Rename it as *TeleportData*
3. Attach a script to it
   i. Export a variable to store if the player should teleport back to a portal or not

```
class_name TeleportData
extends Node


@export var teleport_back = false
```

   ii. Create a variable to store the current portal the player is interacting with

```
var current_portal = null
```

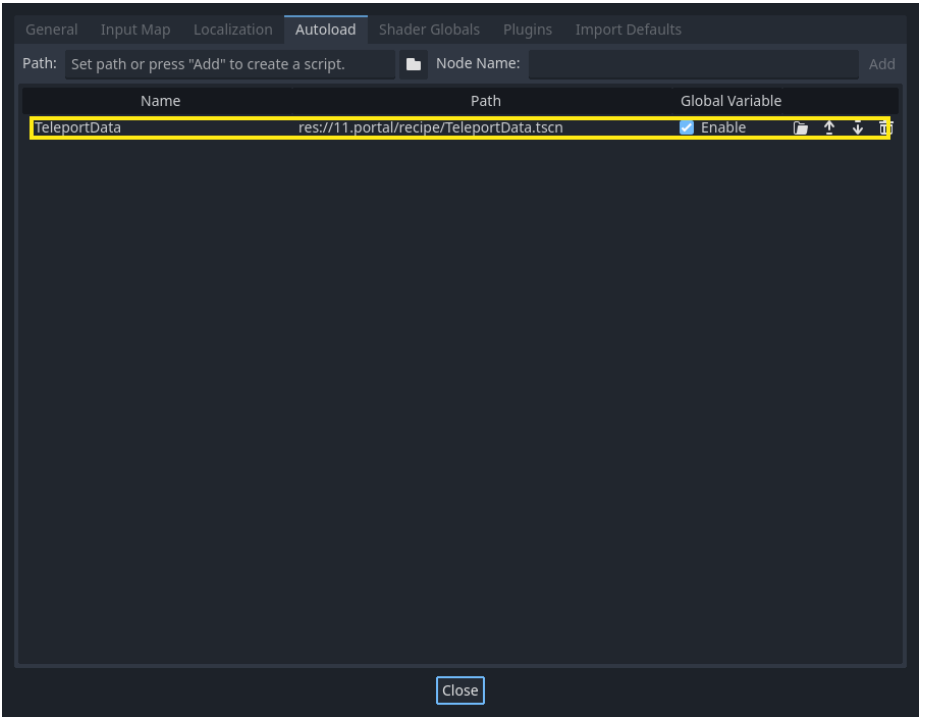
   iii. Create a variable to store the name of the target portal in the next scene.

```
var target_portal_name = "Portal2D"
```

The *TeleportData* script should look like this:

```
class_name TeleportData
extends Node

@export var teleport_back = false

var current_portal = null
var target_portal_name = "Portal2D"
```

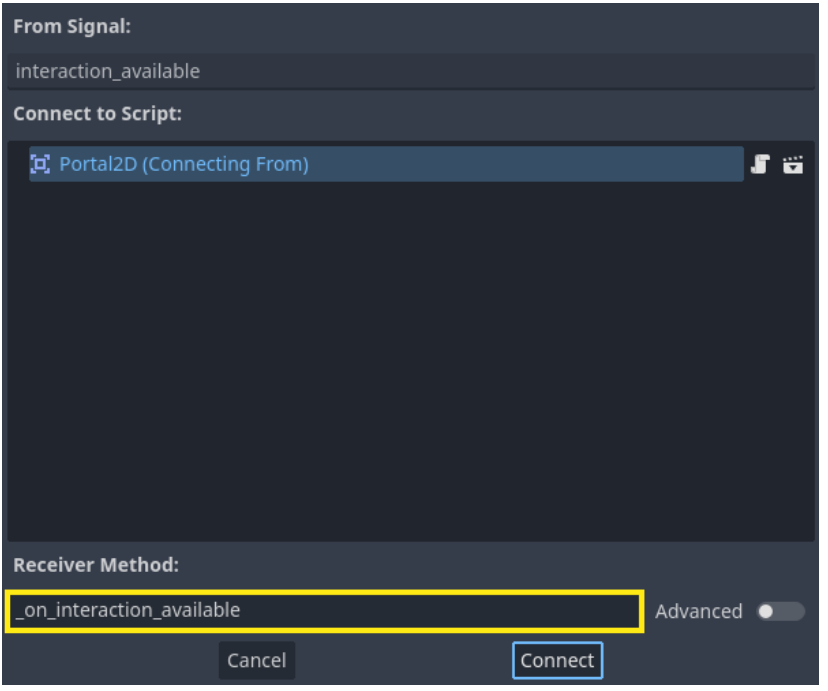
4. In the *Project → Project Setting → Autoload* add the *TeleportData* as a new *Autoload Singleton* and enable *Global Variable*



Now, let's create the actual *Portal2D* logic.

Portal2D

1. Create a new inherited scene using the ***InteractiveArea2D*** as base
2. Rename it as *Portal2D*
3. Extend the *InteractiveArea2D* script
4. Connect the `interaction_available` signal to a callback method, here we named it `_on_interaction_availabled()`



   i. Open the script and export a variable to store the path to the *PackedScene* file that this teleport will send the player to

```
class_name Portal2D
extends InteractiveArea2D


@export_file("*.tscn") var next_scene_path
```

ii. Export another variable to store the name of the target portal that the portal will teleport the player to in the next scene, which by default is `"Portal2D"`

```
@export var target_portal_name = "Portal2D"
```

iii. Create a method named `teleport_in()` that receives an object as argument

iv. Inside the `teleport_in()` method, set the `object` global position to the *Porta2D* global position

```
func teleport_in(object):
    object.global_position = global_position
```

v. Create a method named `teleport_out()` that receives a *PackedScene* as argument which by default is the *PackedScene* loaded using the `next_scene_path`

vi. Inside the `teleport_out()` method, set the *TeleportData* target portal to be this *Portal2D* target portal

```
func teleport_out(next_scene =
load(next_scene_path)):

     TeleportData.target_portal =
target_portal_name
```

vii. Set the *TeleportData* to enable teleporting back since the player entered a portal

```
     TeleportData.teleport_back = true
```

viii. Then change the scene to the next scene using the `get_tree().change_scene_to_packed()` method.

```
func teleport_out(next_scene =
load(next_scene_path)):

     TeleportData.target_portal =
target_portal_name

     TeleportData.teleport_back = true

 get_tree().change_scene_to_packed(next_scene)
```

ix. In the `_on_interaction_availabled()` method, set the *TeleportData* current portal to be this very *Portal2D*.

```
func _on_interaction_available():

     TeleportData.current_portal = self
```

The script should look like this at the end:

```
class_name Portal2D
```

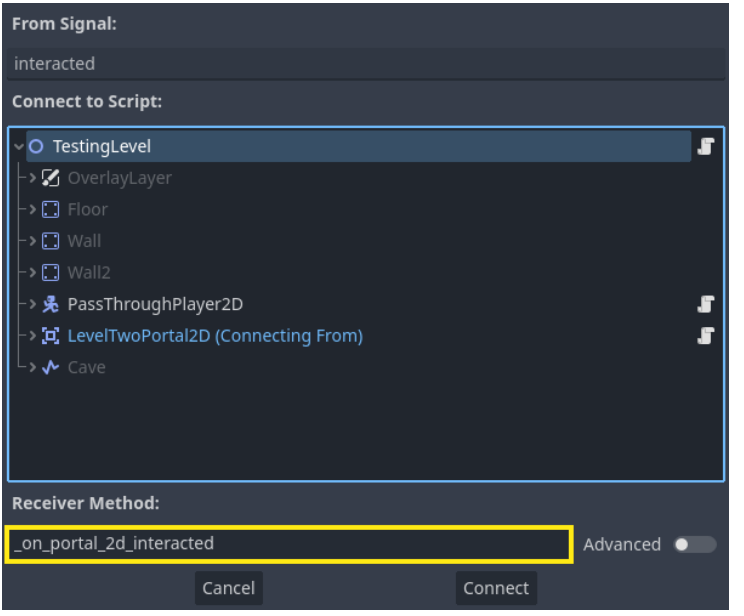
```
extends InteractiveArea2D

@export_file("*.tscn") var next_scene_path
@export var target_portal_name = "Portal2D"



func teleport_in(object):
     object.global_position = global_position



func teleport_out(next_scene = load(next_scene_path)):
     TeleportData.target_portal = target_portal_name
     TeleportData.teleport_back = true
     get_tree().change_scene_to_packed(next_scene)



func _on_interaction_available():
     TeleportData.current_portal = self
```

Cooking it:

To use this recipe, we need something actively mediating the *Portal2Ds* interactions. Nothing better than a *Level Node* for that. Something that is aware of both the *Player* and the *Portal2Ds*.

1. Create a new scene, it can be the level you want to use the *Portal2D*

2. Instantiate a new *Portal2D*
3. Add a ***CollisionShape2D*** as a child of the *Portal2D*
4. Add some graphics and a visual hint to indicate an interaction is available when the player avatar is close to the *Portal2D* (optional)
5. Add the player character that has an *InteractiveArea2D* as its child to allow it to interact with the *Portal2D*
6. Attach a new *GDScript* to the scene's root node and connect the *Portal2D* `interacted` signal to a callback on it. Here I called the callback `_on_portal_2d_interacted()`



From here, things get very subjective and depend on the architecture of your project.

The most fundamental part is to get the *TeleportData* target portal, you can use the `find_node()` for that.

Then call the *Portal2D* `teleport_in()` method passing the player avatar's node as argument when the *TeleportData* `teleport_back` variable is true.

In the `_on_portal_2d_interacted()` callback, use the *TeleportData* current portal and call the `teleport_out()` method on it.

This is how the script may look like in this example:

```
extends Node2D


@onready var player = $PassThroughPlayer2D

@onready var animation_player =
$OverlayLayer/AnimationPlayer




func _ready():

     if TeleportData.teleport_back:

           var portal =
find_child(TeleportData.target_portal_name)

           portal.teleport_in(player)
```

```
    animation_player.play("fade_in")
    await animation_player.animation_finished
    TeleportData.current_portal.teleport_out()
```

Why does this recipe work?

Design-wise:

In most games, especially the ones that focus on adventure and exploration, players have to leave and enter areas on the game world.

Designing everything to be seamlessly connect is not productive. We can rely on player's imagination to connect the dots between levels in the game world. This way they fill the gap of traveling from a Forest level to a Town map, or from a Desert to a Cave, etc.

In open-world games, these gaps are manually filled by environment and level designers. This can be incredibly well designed experiences, but it takes time and money.

Using a *Portal2D* we delegate this service to players' imagination and directly move them to what matters the most.

Engineering-wise:

Using ***Singletons*** we can glue and mediate the interactions between the users of this system: the level and sibling *Portal2Ds*.

The level node, or any equivalent, needs to know which *Portal2D* the player is currently interacting with so it can call the `teleport_out()` on the correct portal and move the player to the desired scene.

Since all the data is lost when a ***PackedScene*** changes, we need to know where the player must arrive in the next scene, for that we rely on the `Node.find_node()` method to figure out where is the target portal in the next scene's architecture.

Beware that due to this approach, instances of the *Portal2D* should ideally have unique names.

Checkpoint2D

What is this recipe?

A *Checkpoint2D* is a set of ***InteractiveArea2Ds*** that serves as a player avatar's spawning position when they touch or interact with it.

You can use as many *InteractiveArea2Ds* as your level needs. The *Checkpoint2D* will automatically manage them.

The *Checkpoint2D* is not the same as a save point. It can optionally store progress on the player's machine, but the only status that matters is the player's position. Unlike a save point, the *Checkpoint2D* does not store any other information.

We can find examples of Checkpoint2Ds throughout many platformer games. But one nice example of this mechanic is found in Ori and the Blind Forest.

Instead of being a passive area in the game that players may or may not visually see, in Ori and the Blind Forest it is a mechanic.

You can spend *Energy* to create what they call a *Soul Link.* This creates blue fire at *Ori's* current position that allows players to access the *Ability Tree* and save the game, including their current position.

In this specific case, developers merged many features in a single mechanic, including a *Checkpoint2D*. Nonetheless, it's interesting how they used this recipe.

When to use this recipe?

A *Checkpoint2D i*s especially useful during moments of high pressure or critical decision-making. It allows the player to feel a sense of accomplishment, no matter how small. Even if they need to restart a section of the level.

On top of that, the checkpoint system provides an extra layer of motivation to encourage players to reach the end of the level. It's good because players know they will be able to start from the same spot if they need to.

In essence, a checkpoint system allows players to experience a sense of progressive success.

Some examples of good places to use *Checkpoint2Ds* are

- Before a path forking, like in this Sunken Glades situation where the player used the Soul Link mechanic to save before making the decision of which path they would gonna take

- Before and after a boss battle, like in Mighty No.9 when players are about to reach a boss pit there’s always a checkpoint

This way, players can maintain their progress throughout the level and focus on understanding the boss’ movement patterns.

If the boss defeats the player, they can always come back to a point where their only concern is the boss himself.

- After or during a platforming puzzle. Let’s use Mighty No.9 again. Right before a platforming test, the developers put a checkpoint.

This is arguably the hardest section of this level. There are many ***Hazards*** in this section, including a fireball rain, unstable platforms that fall when anything, including the player, touches them, giant steam pipes falling from the background on the player's head…

So, it’s kind to allow players to take this test knowing that they won’t have to restart the whole level. This way, they can focus on understanding the section’s challenge, decreasing the level’s cognitive load

The *Checkpoint2D* is also useful on levels that require a play section above the average. This way, players can stop the game, quit, do their other stuff, and come back maintaining their progress.

Pros & Cons:

✔ Decreases players' frustration when they fail
✔ Increases level design extents as players can maintain their progress throughout play sections
✔ Improves the game's perceived quality and polish
✖ Needs access to write data on player's machine

✖ Prone to player hacking since the checkpoint data file is human readable.

> **Tip:**
>
> This can be mitigated using encryption to store and load the file from player's machine. Check the ***Crypto*** class documentation for more information.

How to make this recipe?

Mise en Place:

This recipe uses two classes to work. One is a ***Resource*** that bundles and stores the current checkpoint, we call it *CheckpointData*. The other is the actual *Checkpoint2D*.

Let's start with the *CheckpointData*.

CheckpointData

1. Create a new *Resource* and name it *CheckpointData*
2. Attach a new *GDScript* to the *CheckpointData* a open it
    i. Export a variable to store the current checkpoint index

```
extends Resource


@export var current_checkpoint_index = 0
```

That's it. Yeah, that was easy. Now, let's move on to the *Checkpoint2D*.

Checkpoint2D

1. Create a new scene using a ***Node2D*** as root
2. Rename it as *Checkpoint2D*
3. Attach a new *GDScript* to it and open it
    i. Export a new variable to store the checkpoint data *Resource*

```
@export var checkpoint_data: Resource
```

ii. Export a new variable to represent an option to tell if this checkpoint is incremental or not. An incremental checkpoint don't overwrite player's progress. In other words, only checkpoints with higher index are stored.

```
@export var incremental = false
```

iii. Export a variable to represent an option to tell if this checkpoint should be persistent or not. A persistent checkpoint saves and loads checkpoint data from the player's machine.

```
@export var persistent = true
```

iv. Export a variable to store the path to load and save the checkpoint data on the player's machine. Ideally, each level has its own checkpoint data, so they should use different paths. The script should look like this at this point:

> **Tip:**
>
> Check the User path documentation to know more about storing data locally on the player's machine

```
extends Node2D
```

```
@export var checkpoint_data: Resource

@export var incremental = false

@export var persistent = true

@export var user_file_path =
"user://testing-level/checkpoint_data.tres"
```

v. Create a method that "teleports" an object to the current checkpoint global position. It can optionally receive a point, but by default, it should teleport to the current checkpoint index in the checkpoint data

```
func move_to_checkpoint(object, point =
checkpoint_data.current_checkpoint_index):

  object.global_position =
get_child(point).global_position
```

vi. Create a method to load the checkpoint data if the *Checkpoint2D* is persistent. If there isn't any data to load, we create the directory and save the default checkpoint data *Resource* there

```
func load_checkpoint_data(data_path =
user_file_path):

  if FileAccess.file_exists(data_path):

    checkpoint_data =
ResourceLoader.load(data_path)
```

```
  else:
DirAccess.make_dir_recursive_absolute(data_
path.get_base_dir())

    save_checkpoint_data()
```

vii. Create the method to save the checkpoint data. It's the same method we called above

```
func save_checkpoint_data(data_path =
user_file_path):

        ResourceSaver.save(checkpoint_data,
user_file_path)
```

viii. Create a method to connect all children *InteractiveArea2D's* `interaction_available` signal to a callback in the *Checkpoint2D*, it should use an extra binding with the child index of the *InteractiveArea2D*. This is how we sort which checkpoint is closer to the level's goal

> **Tip:**
>
> You can optionally use the interacted signal instead. This means the player will have to actively interact with the *InteractiveArea2D* in order to save it as a checkpoint.

```
func connect_children_signals():
     for child in get_children():

     child.interaction_available.connect(
```

```
self._on_interaction_available.bind(child.get_index())
)
```

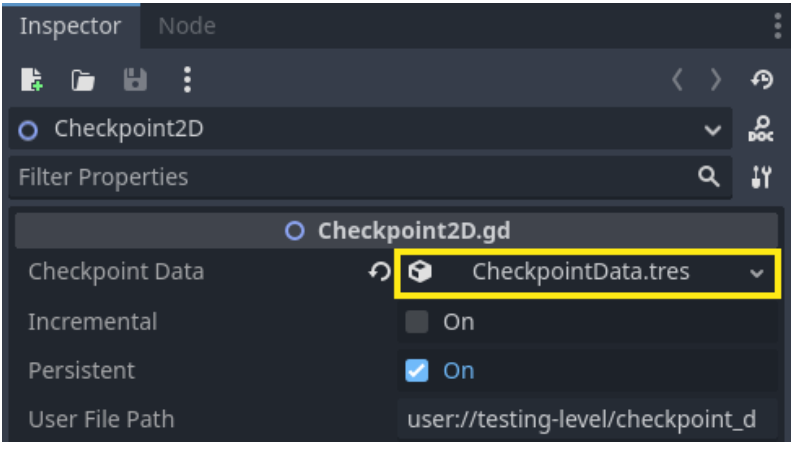
ix. Create the above callback method. When the player interacts with a checkpoint *InteractiveArea2D* we check if the checkpoint is incremental, if so, we check if the current checkpoint index is greater than the current checkpoint index. If it is, we overwrite the current checkpoint index.
x. Overwrite the current checkpoint index if the checkpoint is not incremental
xi. Save the checkpoint data

```
func _on_interaction_available(check_point_index):
        if incremental:
                if check_point_index > checkpoint_data.current_checkpoint_index:

checkpoint_data.current_checkpoint_index = check_point_index
        else:

checkpoint_data.current_checkpoint_index = check_point_index
```

xii. In the `_ready()` callback, connect the children signals and, if the checkpoint is incremental, load the checkpoint data

```
func _ready():
        connect_children_signals()
        if persistent:
                load_checkpoint_data()
```

4. Attach the *CheckpointData* to the *Checkpoint2D → Checkpoint Data* property



Now, let's see how to use this recipe in practice! Before we dive into the cooking part, let's see how the script should look like after all these steps:

```
extends Node2D
```

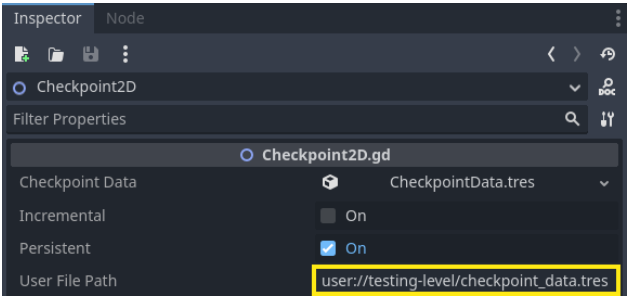
```
@export var checkpoint_data: Resource

@export var incremental = false

@export var persistent = true

@export var user_file_path =
"user://testing-level/checkpoint_data.tres"



func _ready():

     connect_children_signals()

     if persistent:

           load_checkpoint_data()



func move_to_checkpoint(object, point =
checkpoint_data.current_checkpoint_index):

     object.global_position =
get_child(point).global_position



func load_checkpoint_data(data_path = user_file_path):

     if FileAccess.file_exists(data_path):

                checkpoint_data =
ResourceLoader.load(data_path)

     else:
```

```
DirAccess.make_dir_recursive_absolute(data_path.get_ba
se_dir())
            save_checkpoint_data()



func save_checkpoint_data(data_path = user_file_path):
     ResourceSaver.save(checkpoint_data,
user_file_path)



func connect_children_signals():
     for child in get_children():
            child.interaction_available.connect(

self._on_interaction_available.bind(child.get_index())
            )



func _on_interaction_available(check_point_index):
     if incremental:
            if check_point_index >
checkpoint_data.current_checkpoint_index:
```

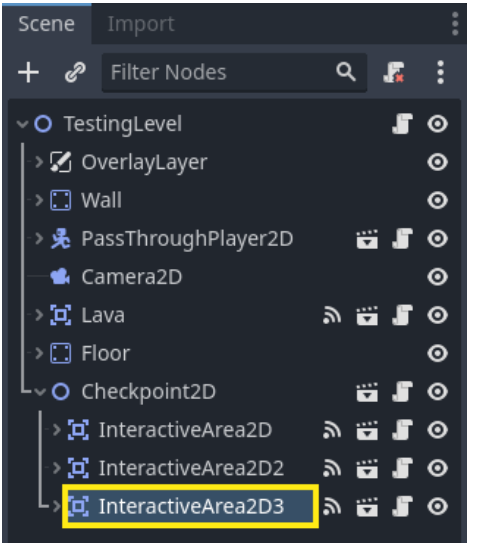
```
    else:
        checkpoint_data.current_checkpoint_index = check_point_index
    save_checkpoint_data()
```

Cooking it:

1. Create a new scene, it can be the level you want to have a *Checkpoint2D*
2. Add an instance of a *Checkpoint2D*
3. Change the *User File Path* property to match the folder and file which you want to save the this level's *CheckpointData*

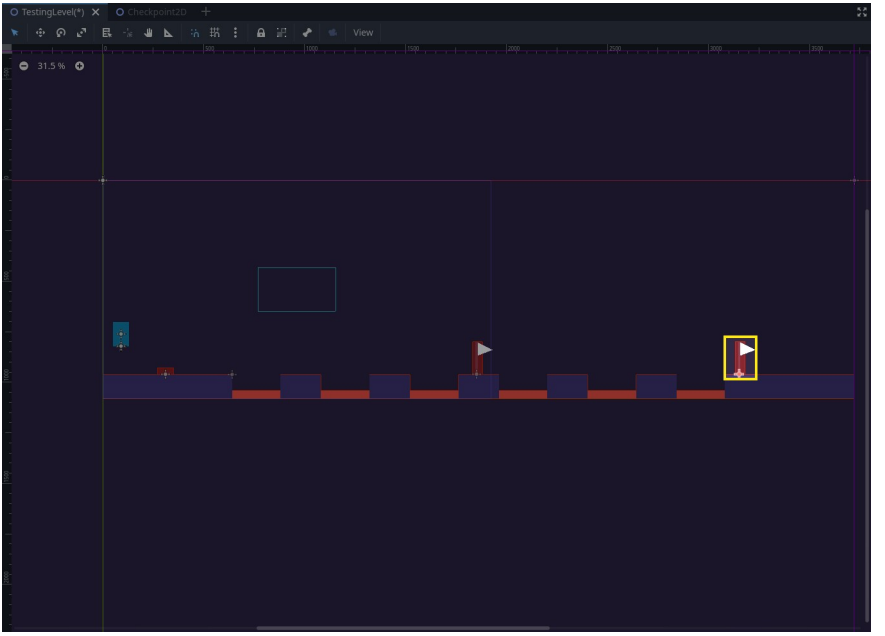


4. Add at least one instance of an *InteractiveArea2D* as a child of the *Checkpoint2D*.
5. Add a *CollisionShape2D* or a *CollisionPolygon2D* as a child of the *InteractiveArea2D*



6. Place the *InteractiveArea2Ds* where you want to have actual checkpoints in your level, remember they are indexed, so the closer to the goal, the down they should be in the *Checkpoint2D* children hierarchy
7. Add some graphics to the *InteractiveArea2D* to visually represent checkpoints (optional)

To use this recipe, we can call the `Checkpoint2D.move_to_checkpoint()` method from the topmost node, the *TestingLevel*, passing the player node, in this case the *PassThroughPlayer2D*.



In my case, I've also connected the player's `HurtBox2D.damaged` signal to trigger the lose condition, reloading the level scene. The *TestingLevel* script ended up like this:

```
extends Node2D

@onready var player = $PassThroughPlayer2D
@onready var checkpoint = $Checkpoint2D

func _ready():
        checkpoint.move_to_checkpoint(player)
```

Checkpoint2D

```
func _on_player_damaged(damage):
        get_tree().reload_current_scene()
```

Why does this recipe work?

Design-wise

Failure can feel really frustrating if players don't feel like their failure is progressing them toward a goal.

Saving players' progress throughout a level is an excellent way to ease and decrease players' frustration when they fail.

With checkpoints, we can decrease the players' cognitive load so they can focus on one section of the level at a time and understand where they are failing on this section.

Checkpoints can also work as exit gates for a play session, allowing the players to take some time to rest and come back to the game refreshed knowing their progress wasn't lost.

Engineering-wise:

We use Godot Engine *Scene Tree* architecture to create an indexing system for our checkpoints.

Then, using the built-in *Resource* loading and saving systems we store this data persistently in the player's machine so we can maintain their progress throughout play sessions.

The ***ResourceLoader*** and ***ResourceSaver*** singletons are amazing because they parse the human-readable data as *Resource* objects. With that, we don't have to parse it ourselves like we would if we used a JSON file.

And since *Resources* aren't freed from memory as long as there is still something using them, we can also maintain progress

throughout reloads of the same scene. This allows us to have a clean slate scene every time the player dies.

We can optionally just create a new instance of the player's node every time they die and pass the new instance as an argument to the `Checkpoint2D.move_to_checkpoint()` method. This way, if the player defeated some enemies and collected some items, this progress is also kept in this play session.

Switch2D

What is this recipe?

A *Switch2D* is an object used to toggle another object's state, triggering a new behavior. Sounds a bit abstract right? But think about a button that opens a door, for instance.

It's essentially an ***InteractiveArea2D*** that points to another object and executes a predefined method. We can use duck typing to check if the target object has a predefined method, such as `switch()` or `toggle()`.

> **Tip:**
>
> Duck typing is excellent for two-sided systems. It takes rid of strict typing problems and focuses on the object's interface. Take a look at the **Engineering** section of this recipe to understand how this works.

We can find a good example of a switch mechanism in this *Ori and the Blind Forest Mount Horu's* moving platform puzzle.

Here, the designers combined two *Switch2Ds* with two ***MovingPlatform2Ds*** to create a platforming puzzle where players should avoid some ***Hazard2Ds***.

The lava streams kill Ori immediately, so the player must find a way to reach the top-left portion of the area. For that, they need

to interact with these levers that toggle on the movement of the moving platform. In this case, the levers are the *Switch2Ds*.

When to use this recipe?

We usually use the *Switch2D* in puzzle sections of our game, especially in Lock & Key mechanisms.

Lock & Key mechanisms are a well unknown game design pattern where players can't progress unless they meet a given requirement.

This requirement can be straight up a key to unlock a door, but it can also be a skill to unlock a behavior. As an example, a double jump to reach the edge of a cliff.

In this case, we can put a *Switch2D* on top of this cliff. This *Switch2D* can be a statue that the character's pray to and it opens a *Portal2D* to the next level.

You can also use multiple instances of a *Switch2D* to increment progress on a given target object.

For instance, imagine a giant boss that only wakes up when players light up four torches.

Each torch can be a *Switch2D.* We can spread them around the level.

Then, the boss can be a target object that when its `switch()` method is called, it increments a `torches_lit` counter:

```
# Boss.gd
```

```
func switch():
  torches_lit += 1
  if torches_lit >= 4:
    wake_up()
```

Pros & Cons

✔ Enhances game design by making easy to implement Lock & Key mechanisms
✔ Improves world design by making objects in the world interact between themselves
✔ Easy to set up and can be implemented at any stage of production
✖ Two sided system where most of the behavior is a specific to the other object
✖ Hard to debug due to duck typing

How to make this recipe?

This recipe is a two-sided system where we have the *Switch2D* and the target object. Let's start with the *Switch2D* itself.

Mise en Place:

Switch2D

In the *Scene -> New Inherited Scene* menu create a new inherited scene using the *InteractiveArea2D*

1. Rename the root node as *Switch2D*
2. Extend the *GDScript*
    i. Export a *NodePath* variable pointing to the target object

```
@export var target_object_path: NodePath
```

    ii. Export a variable to store the name of the method this *Switch2D* should call on its target

```
@export var target_method = "switch"
```

    iii. Create a variable to store the actual target object

```
var target_object
```

> **Tip:**
>
> We can use this variable to inject target objects in the *Switch2D.* For instance, a *Level* node, which could be the *Switch2D* parent, can use

> this variable inside its `_ready()` callback to point to a *Bridge* node as the target object.

iv. In the `_ready()` callback, get the node using the `target_object_path`

```
func _ready():
     if target_object_path:
          target_object =
get_node(target_object_path)
```

> **Tip:**
>
> We use this check to prevent calling the `get_node()` method when the `target_object_path` isn't set, in other words, when `target_object_path == null`

v. Create a `toggle()` method that by default uses the `target_object` as its internal `_target_object`.

```
func toggle(_target_object =
target_object):
```

> **Tip:**
>
> This allows us to pass a different `target_object` as an argument if we need to.

vi. Inside the `toggle()` method, check if the `_target_object` has the method we want to call, and if it has, call it:

```
func toggle(_target_object = target_object):
     if _target_object.has_method(target_method):
 _target_object.call(target_method)
```

Alright, this concludes our **Switch2D.** Let's take a look at the complete script before moving on to the target object.

```
class_name Switch2D
extends InteractiveArea2D


@export var target_object_path: NodePath
@export var target_method = "switch"


var target_object


func _ready():
     if target_object_path:
                target_object = get_node(target_object_path)
```

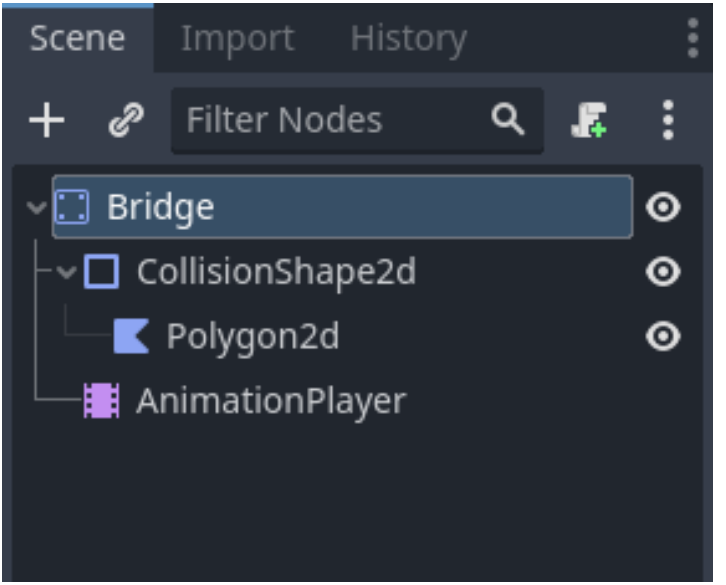
```
if _target_object.has_method(target_method):
    _target_object.call(target_method)
```

Target Object

Well, the target object is pretty simple and it's very abstract. It can be anything in your game. The only thing absolutely mandatory is that it has a method that matches the **Switch2D**'s `target_method`.

As an example, let's make a *Bridge* that extends and retracts based on the state of a *Lever*, that is a *Switch2D*.

1. Create a new scene using a ***StaticBody2D*** as root and name it *Bridge*
2. Add a ***CollisionShape2D*** as its child and set it up properly
3. Add some graphics such as a ***Sprite2D***, or like in this case, a ***Polygon2D*** as children of the *CollisionShape2D*
4. Add an ***AnimationPlayer*** as a child of the *Bridge*



5. Create a new script for the Bridge, then
    i. Export a variable to store the current state of the bridge, let's call it `extended` and by default it's `false`

```
extends StaticBody2D


@export var extended = false
```

    ii. Store a reference to the ***AnimationPlayer*** in an `onready` variable

```
@onready var animation_player =
$AnimationPlayer
```

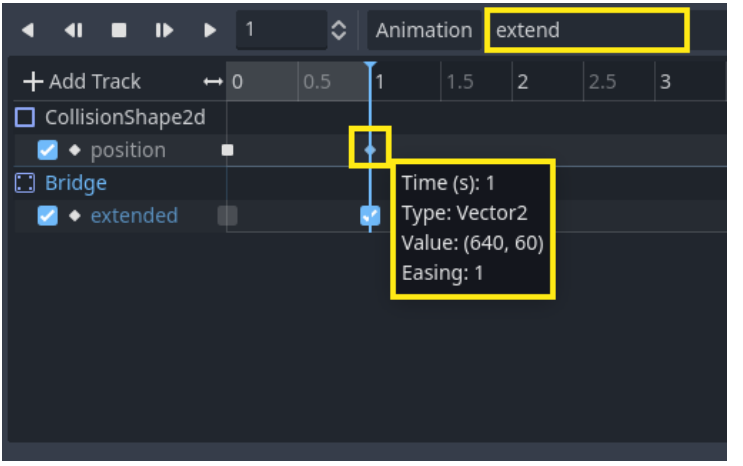
iii. Create a method that matches the *Switch2D* `target_method`

```
func switch():
```

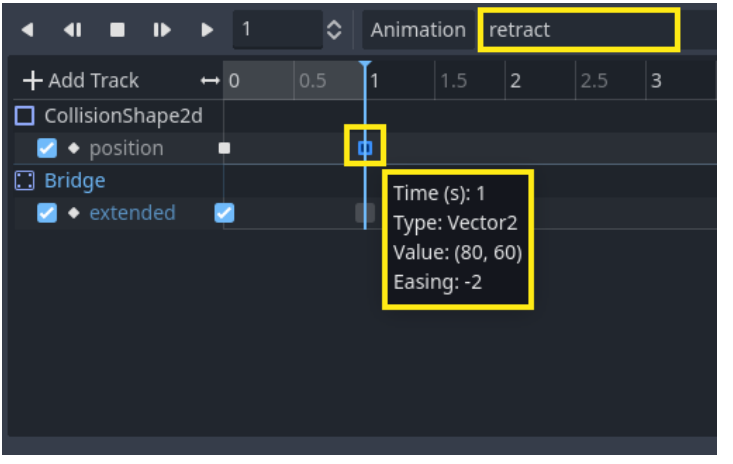
iv. Inside this method, check the current state of the Bridge and play the animation that corresponds to the opposite state, so it transitions to this other state at the end of the animation

```
func switch():
    if extended:
        animation_player.play("retract")
    else:
    animation_player.play("extend")
```

6. In the *AnimationPlayer*, create a new animation and name it “extend”
    i. Create a track to animate the ***CollisionShape2D*** → ***Position*** property
    ii. Create a track to animate the *Bridge* → *Extended* property
    iii. Animate the *CollisionShape2D* → *Position* to simulate an extension movement
    iv. At the end of the animation add a key to the *Bridge* → *Extended* track setting it to `true`
7. Create another animation called `retract` and do the same process but with opposite values. For instance, starting the movement where the `extend` animation



ended it. And most importantly, starting with the *Bridge → Extended* as `true` and finishing it as `false`



From here, you can create your lever using the ***Switch2D*** as root node.

You can add some graphics to animate the lever to visually represent its “pushed” state.

For that, you can extend the *Switch2D* script and add the necessary conditions. In my case, the script ended up like this:

```
extends Switch2D

@export var pushed = false

@onready var animation_player = $AnimationPlayer



func _ready():
	target_object = get_node(target_object_path)



func _on_interacted():
	if pushed:
		animation_player.play("pull")
	else:
		animation_player.play("push")
	pushed = not pushed
```

Note that the `_on_interacted` method is a callback for the `interacted` signal.

Why does this recipe work?

Design-wise:

The *Switch2D* enables a type of mechanism in games that adds a layer of indirection in players actions. This increases player's perceived awareness of the game world since they will understand that interacting with an object may have consequences on other objects.

With this new layer of indirection we can create all sorts of puzzling mechanics for our players. As an example, imagine a level where there are three *Switch2Ds*. Each of them points to a gate. Players must open the gates in a specific order. You can position each *Switch2D* in such a way that incentive players to explore the level.

Engineering-wise:

Taking advantage of the *InteractiveArea2D* we extended its behavior such that it now affects another object, making this a three points interaction:

- ***Player*** with ***Switch***
- ***Switch*** with ***Target***

We use duck typing such that as long as we have a `target_object` the *Switch2D* should either work or do nothing. With that, we shouldn't get any errors.

Duck typing allows us to be type agnostic. In this approach, **any** object in our game can become a target object as long as it has the `target_method` we want.

Of course, this implementation has the limitation that the `target_method` shouldn't ask for arguments. But we can fix that extending the *Switch2D*.

Conclusion

As we reach the conclusion of the **Platformer Essentials Cookbook,** let's reflect on the journey we've undertaken, a journey through the intricate landscape of platformer game development with the Godot Engine. This book has not only served as a guide but also as a mentor, teaching you the fine art of crafting game mechanics with precision and creativity.

Throughout the book, you've been exposed to a variety of game development recipes, each tailored to meet specific needs in platformer game creation. From the fundamental ***BasicMovingCharacter2D***, a cornerstone in player-avatar interaction, to more complex patterns that challenge and enhance your understanding of physics, animation, and player engagement. You've learned not just how to implement these patterns, but also why they work – understanding the underlying principles that make for a successful and engaging platformer game.

This book is much more than a one-time read. It's a resource meant to be revisited, a reference manual that you should keep at your side as you embark on your game development projects. Each recipe within this book has been meticulously crafted, offering insights into both the design and engineering aspects of game development with the Godot Engine. Whether you're tweaking physics for more realistic character movement, or ensuring seamless interactions within your game world, these recipes provide a foundation upon which you can build and innovate.

Remember, game development is an iterative process, one that benefits from continuous learning and adaptation. The **Platformer Essentials Cookbook** is designed to be a part of this process, a tool to be used over and over again. As you work on your platformer game projects, refer back to these recipes. Experiment with them, adapt them, and see how they can be transformed to fit the unique requirements of your game.

In essence, this book is your companion in the journey of game development. It's a testament to your commitment to excellence and a reflection of your passion for creating engaging, dynamic platformer games. Keep it close, and let it guide you as you continue to push the boundaries of what's possible in the realm of game development with the Godot Engine.

That's it, thank you so much fore reading.

Keep developing and until the next time!