

Game Development with Godot

Part Two

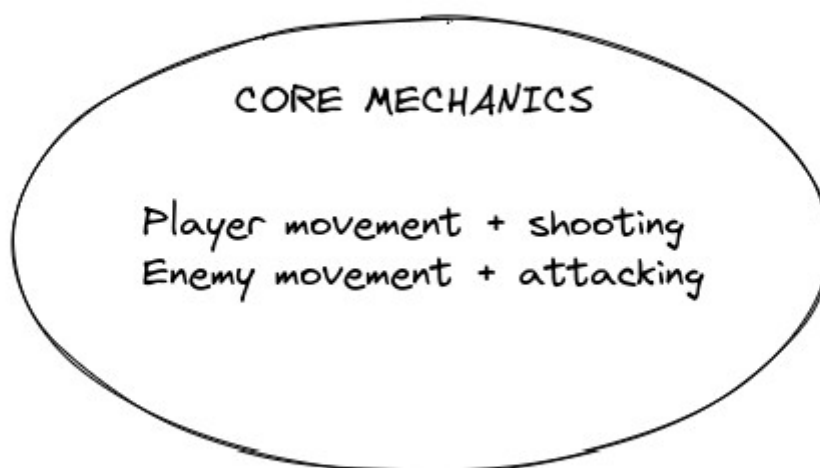
Preplanning

In this part of the tutorial we will start making the actual game, but before we do so we need to plan out a bit what we are actually doing. To get started we first need to decide what kind of game we are making and what we need to implement in the game.

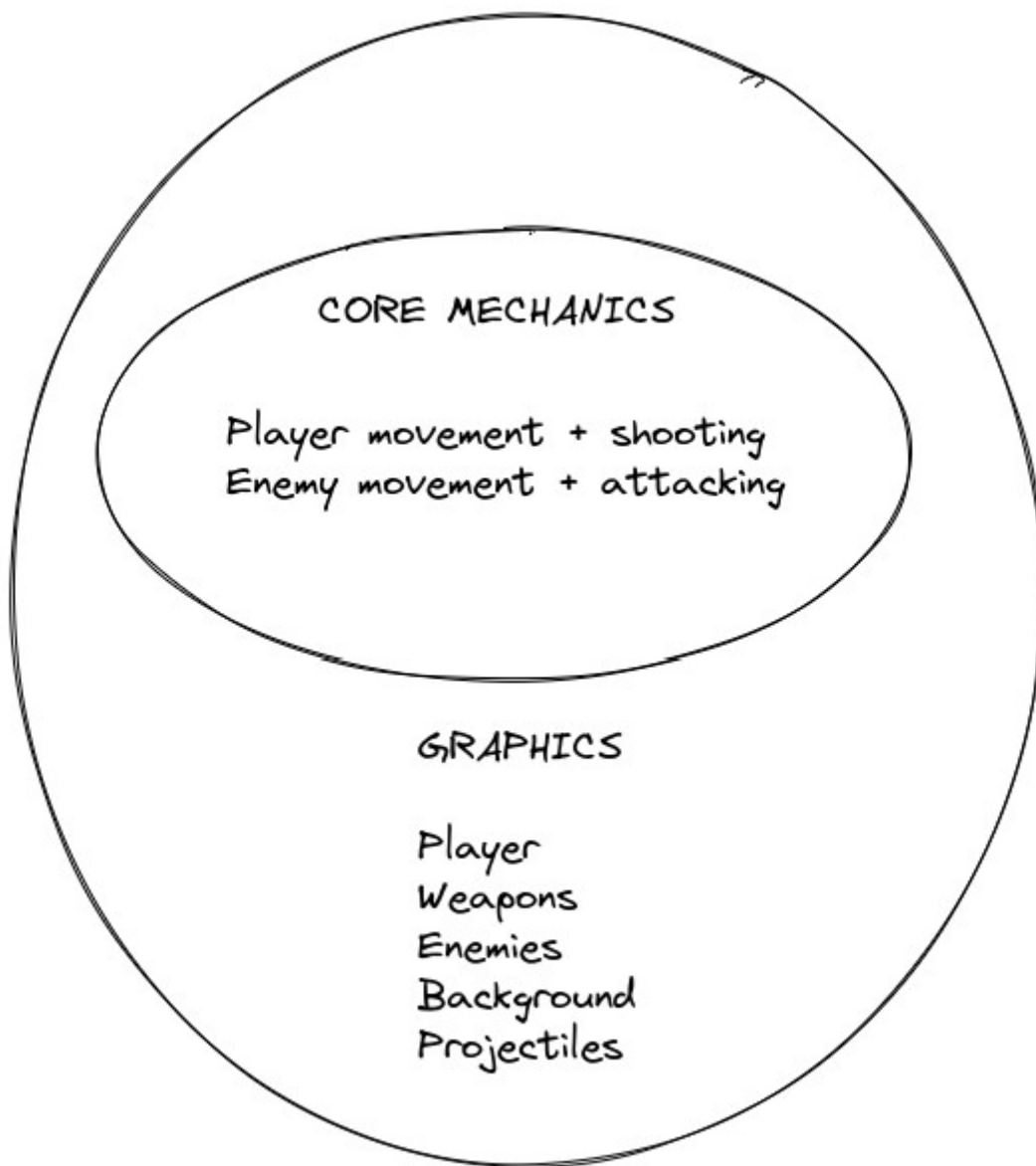
I want to make a simple top down shooter game, because that will give us a lot of flexibility on how to expand on the idea. I will call my project “Bug Zapper” and will make it about a robot shooting bugs on some alien planet.

The way I like to do this is to plan it out from the core mechanics outwards. I’m using Excalidraw in Obsidian to plan this out however you can do this in any application or even on a whiteboard or paper, whatever works, when you make your own projects.

The first things we need from core mechanics are player movement, shooting and enemies so I’ll make a circle with those things first.

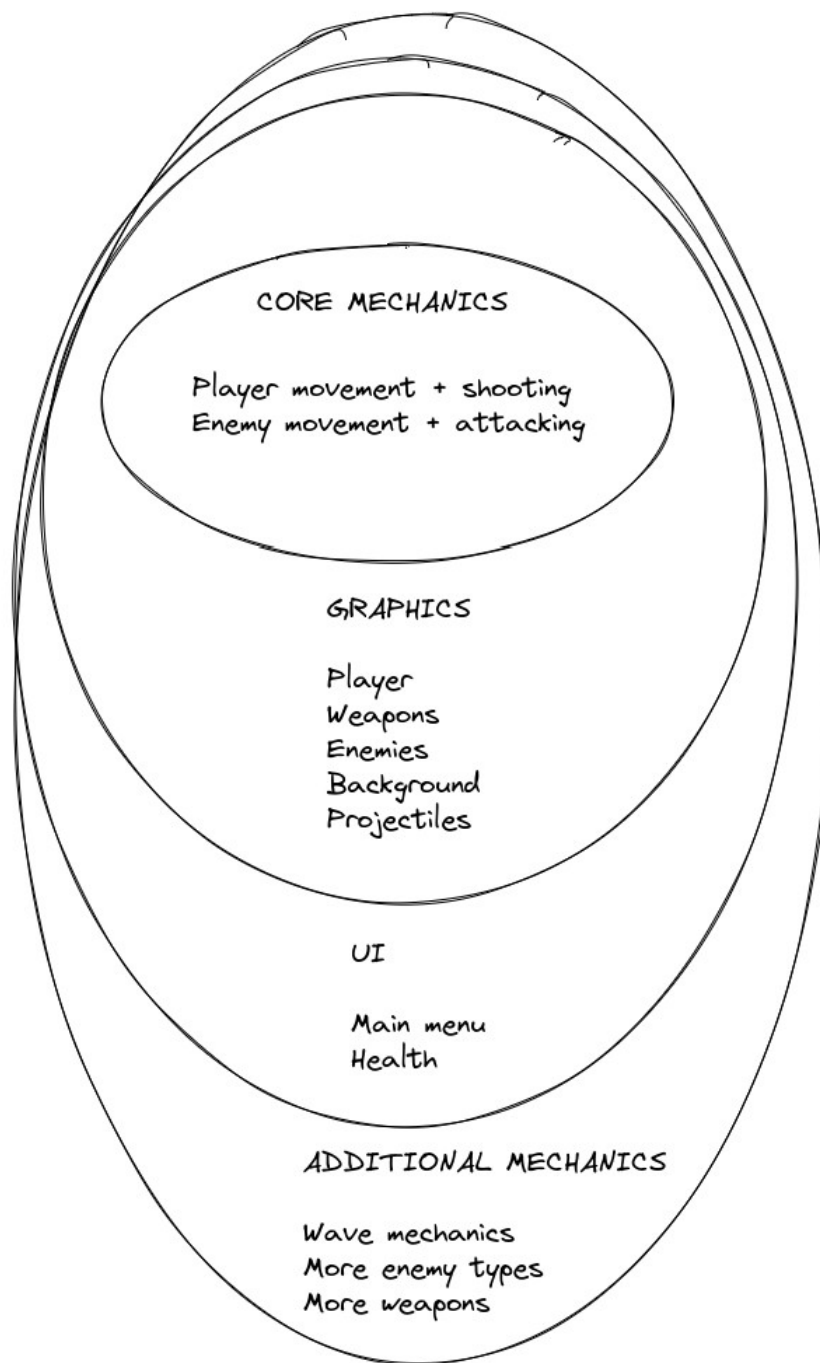


Next, to make the game actually look alright we're going to need some graphics, so I'm making another note with that and a circle around that:



This list will help us manage our priorities if, like me, you are prone to letting your ideas get out of hand.

Here is my "finished" plan:



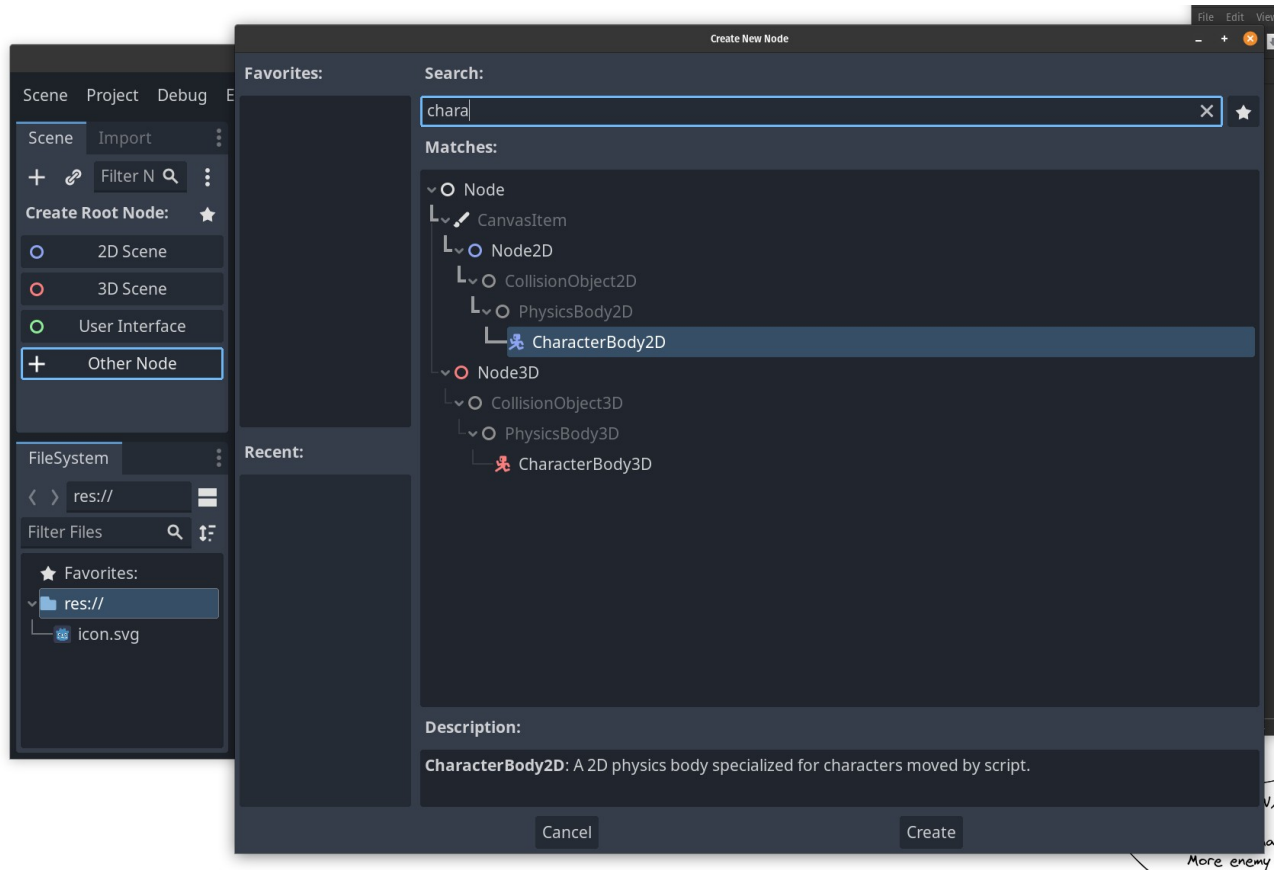
It's important to note this isn't carved into stone or anything. Things can change and evolve during development, this is just a starting point. I'd keep more detailed notes about how we want the weapons to work, how we want the enemies to work and so on as well, but we'll think about that later.

Drawing things out will also help us figure out the logic behind our code later on, but for now, let's actually get started on a new project.

Player Movement

So the first thing we need to do is start a new project. This was covered in the first tutorial, so it shouldn't be difficult for you at this point. Simply follow the steps from there and make a project with whatever name you want.

With a new, blank project we will set up our player first so we will create a new scene with a root node of CharacterBody2D, so click the Other Node option and search for CharacterBody2D:



First off, before anything else I want to rename the root node to Player, you can do this by right clicking the CharacterBody2D node in the scene tree and selecting rename. CharacterBody2D is a node that is commonly used for players and enemies in games, it has some features that are very useful if you want to have physics based movement while still being able to control the movement through code. You could also use a RigidBody2D but that becomes more complex and CharacterBody2D fits our needs better.

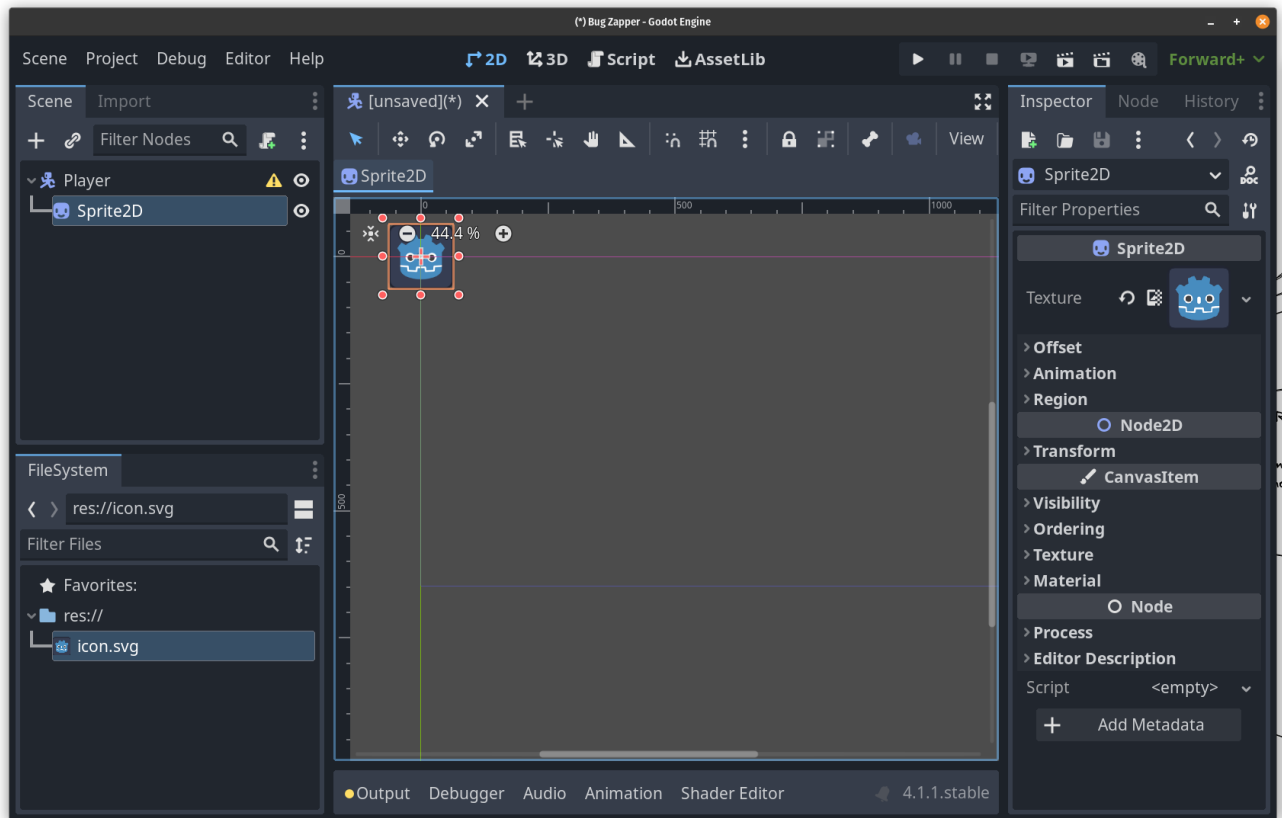
You can read up more on CharacterBody2D here:

https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html

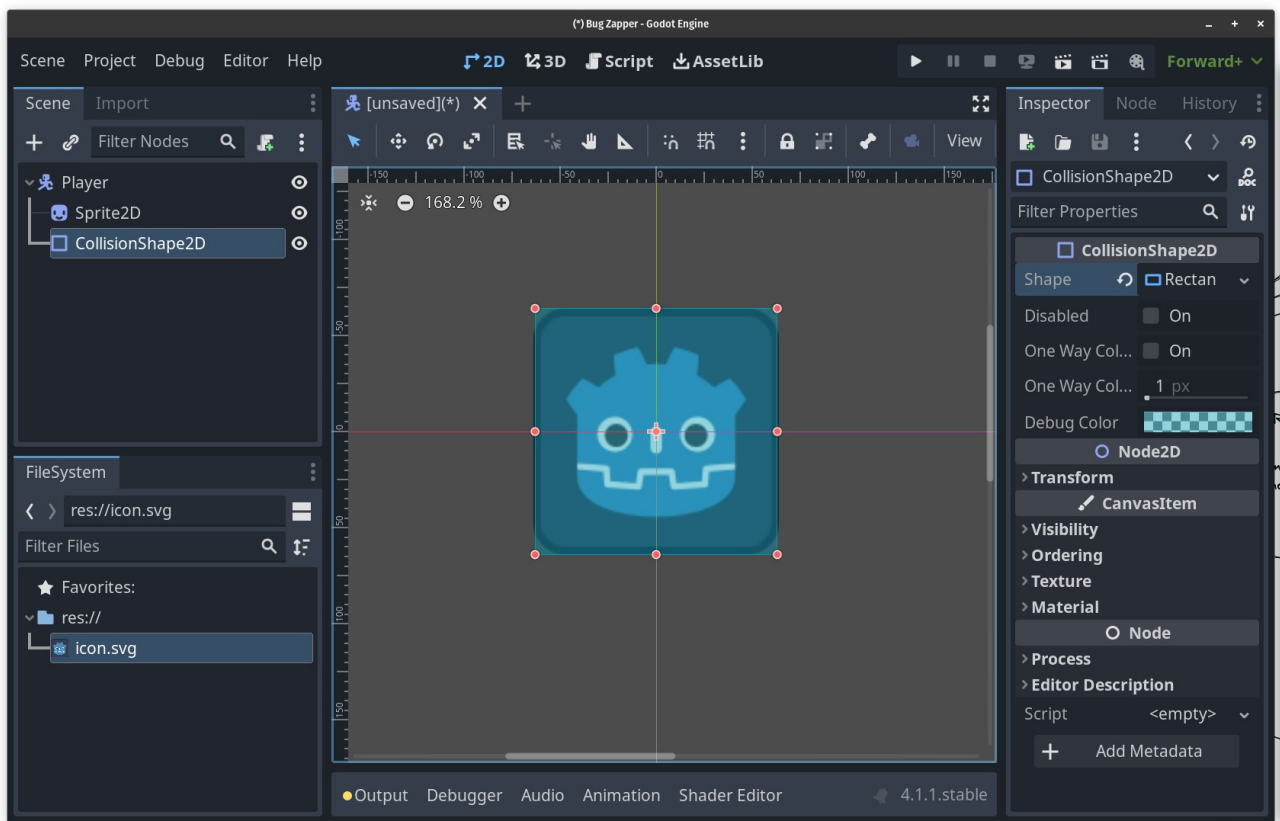
Next up, we have an issue where there is a warning triangle by the Player node, that's because we need a collision shape, but before we add one we can add some placeholder graphics. Handily godot comes with a graphics file we can use for this until we get our own graphics style going, so we need to add a node that can use that graphic.

So we will add a child node of the type Sprite2D under player and then in the Inspector where there

is a Texture field we can drag and drop the icon.svg from the FileSystem dock:

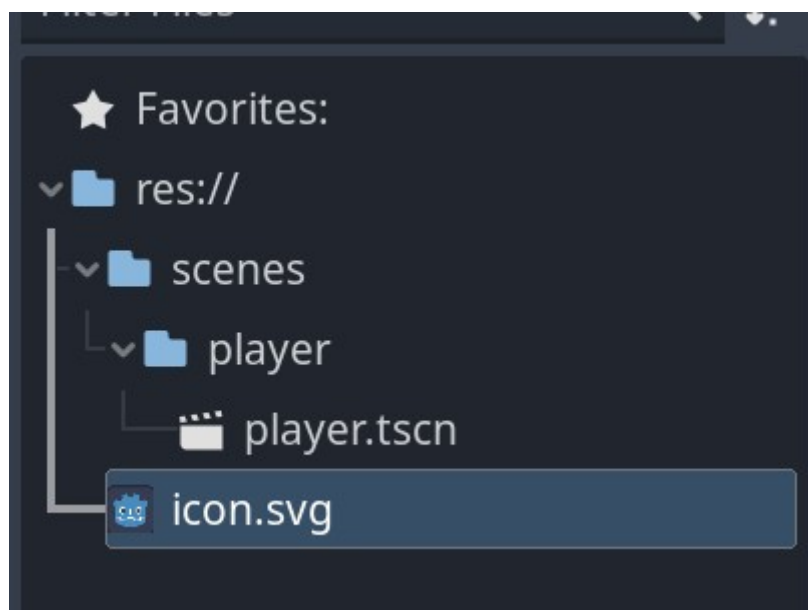


Now we have something we can see without needing to have the collision shape debugger option turned on. Next we need a collision shape though, so I'll make one of those as well:

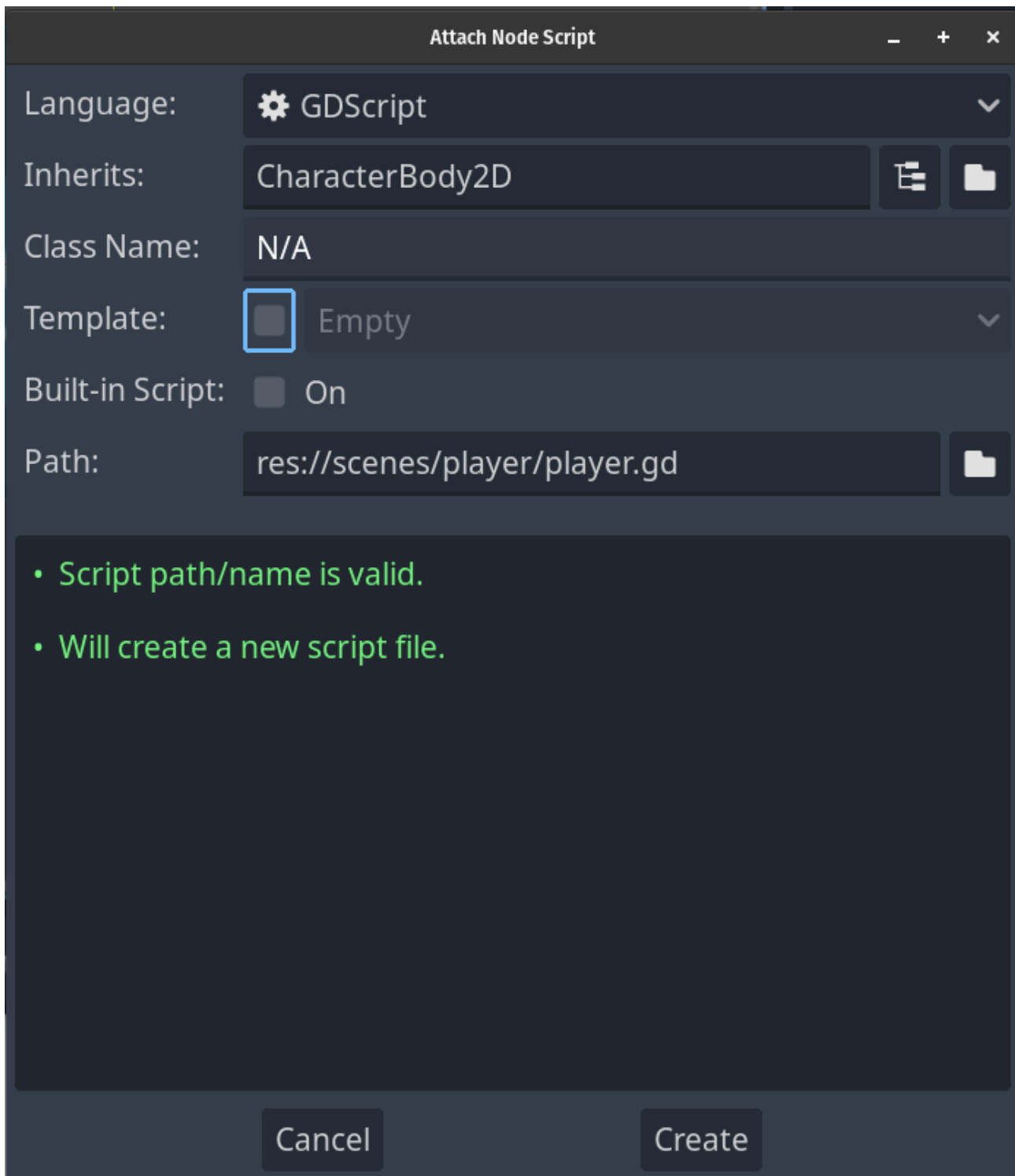


I made a CollisionShape2D child node and in the Shape property added a rectangle shape that I then expanded while holding the alt key so I could get it symmetric.

Now I want to save this scene, so I will make some folders and save it under scenes/player/player.tscn so my FileSystem now looks like this:

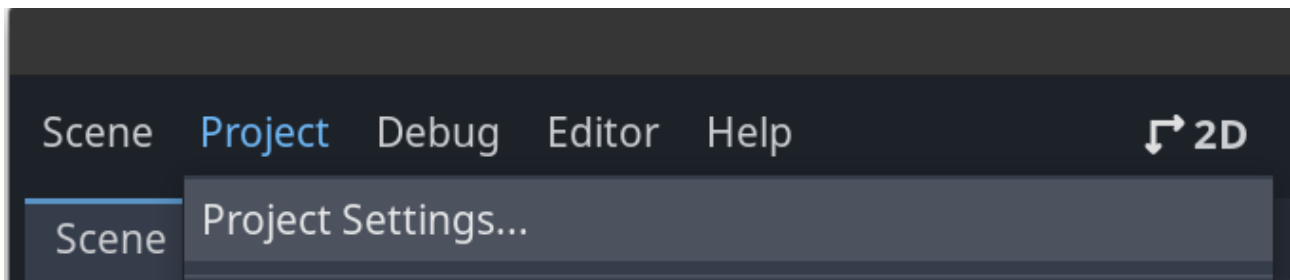


Next we need to attach a script to the Player, so right click the root node and click Attach Script. I'll save it in the same folder as the player scene and will not be using a template:

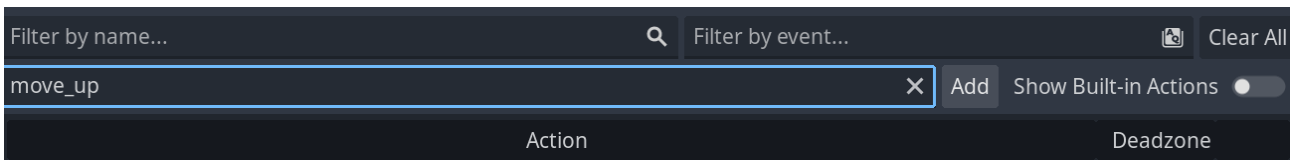


In this script we will handle a bunch of stuff, but for now we want to handle movement. Before we can make the character move however, we need to set up our inputs so the game knows what we want to press to make the character move.

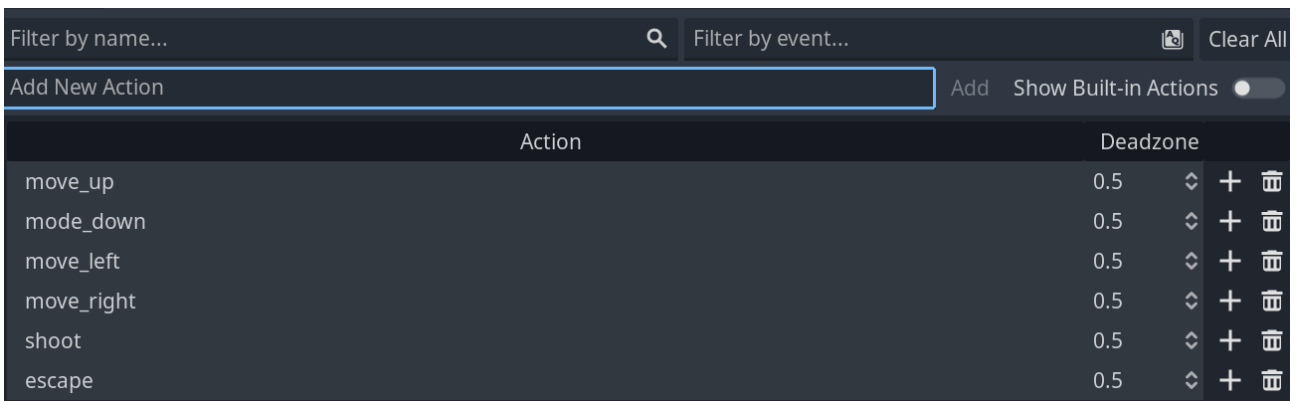
So under the Project menu we want to access Project Settings:



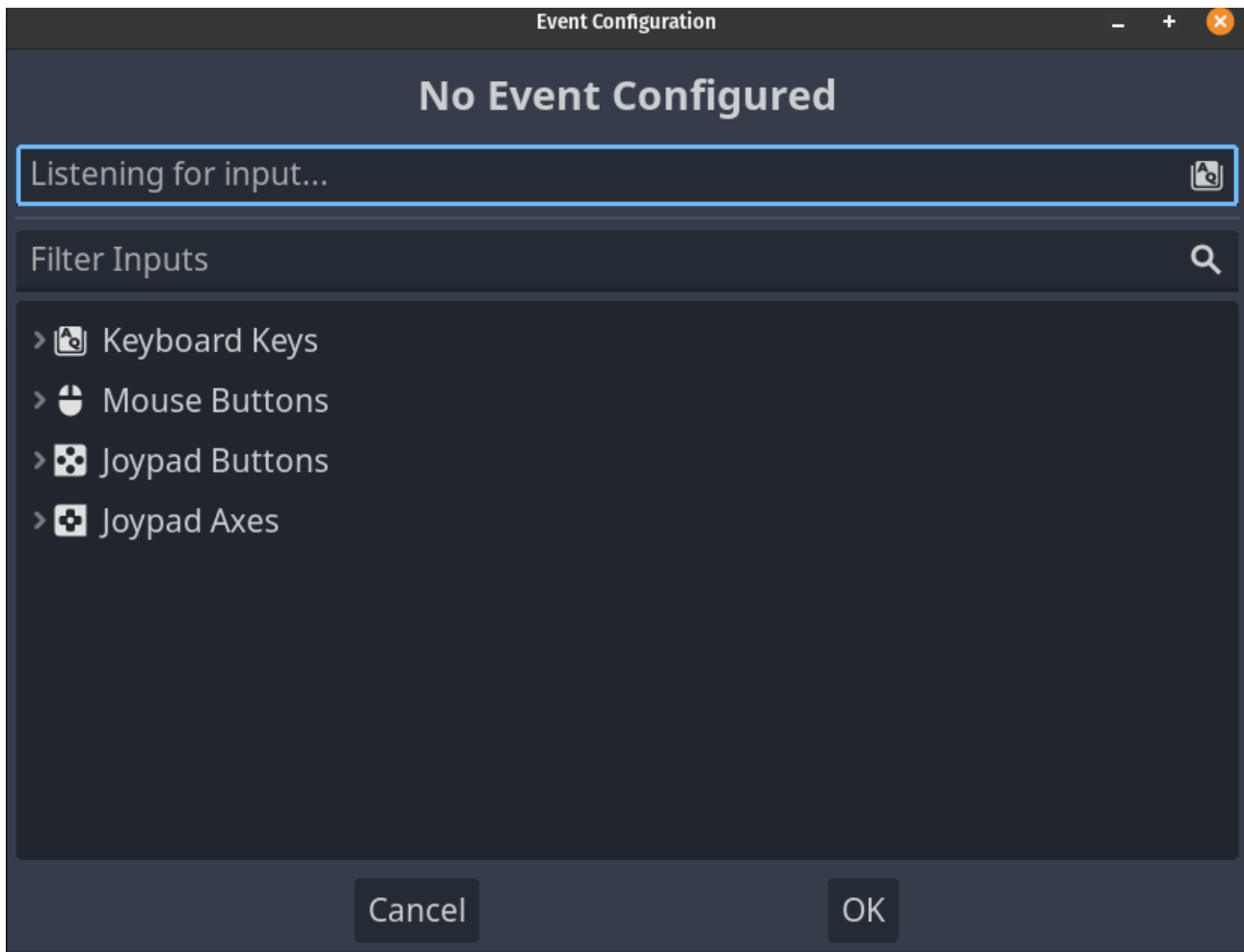
From there we want the input map tab, that's the second one. In here we can add new actions by writing the name in the "Add New Action" field and then clicking add like so:



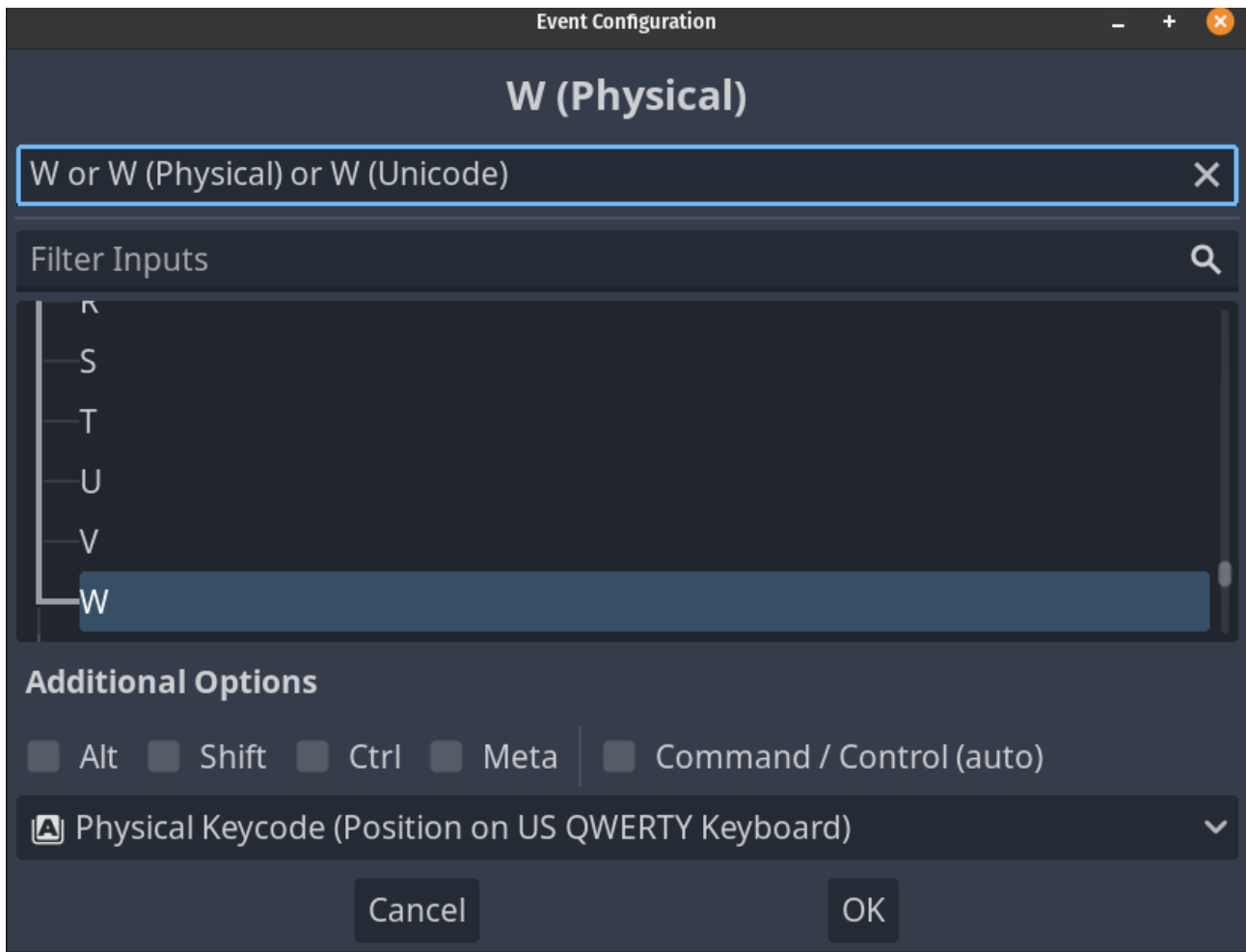
I want to add a bunch of actions so I'm going to do them all while I'm here:



Now we have a bunch of actions, but Godot still doesn't know what input corresponds to which action. To fix this we need to press the + by the action name and then we get a new window:



In here we can either search for the appropriate input or we can simply click in the “listening for input...” field and then press the input we want to hook up to the action. Since I clicked the `move_up` action I want to press the W key and it will find it automatically:



Then I can just press ok and then fill out the rest of the actions:



It's worth noting we can bind multiple inputs to an action as well, so we could set up the arrow keys to also move the player, but I won't do that just now. With this done we can close the project settings and go back to working on the script.

Here's the first pass of the script:

```

1  extends CharacterBody2D
2
3  func _physics_process(delta):
4      #Get direction from input
5      var direction = Input.get_vector("move_left", "move_right", "move_up", "move_down").normalized()
6
7      #If direction is getting input, multiply built in velocity with the direction and a speed
8      if direction:
9          velocity.x = direction.x * 700
10         velocity.y = direction.y * 700
11         #If there's nothing in direction, stop moving
12     else:
13         velocity.x = 0
14         velocity.y = 0
15         #Call the built in move and slide function
16     move_and_slide()
17

```

`_physics_process(delta)` is a built in function that runs at a set rate, by default it should be 60 times per second. It is recommended that movement is handled in this function.

Next we declare a variable called `direction`, which using another built in godot function we can use to store a `Vector2`. A `Vector2` is a variable type that contains 2 floats and we will be seeing a lot of it later on. The way this is set up, depending on what the player presses, `direction` will basically store two values from -1 to 1. So if the player presses the `move_left` action `direction` will be -1, 0. If the player presses `move_up` it will be 0, -1 because with godots coordinate system going up means reducing the value of `Y` which can be a bit confusing.

In the `if` statement we are basically checking that `direction` is anything except 0, in which case we set up `velocity` which is a built in value in `CharacterBody2D`. We do this by simply multiplying the corresponding axis with `direction` times a speed, in this case 700.

The `else:` statement is there to set the `velocity` to 0 if nothing is being pressed, otherwise we would keep moving the player.

Finally we call `move_and_slide()` which is a built-in function of this node.

Now if you press `F5` and select the player scene as the scene to run, `wasd` should move around the godot icon!

But there are still some improvements we could make. Let me write up the code and go through the changes.

```

1  extends CharacterBody2D
2
3  @export var speed: int = 700
4  @export var acceleration: float = 0.1
5  @export var deceleration: float = 0.5
6
7
8
9  func _physics_process(delta):
10     #Get direction from input
11     var direction = Input.get_vector("move_left", "move_right", "move_up", "move_down").normalized()
12
13     #If direction is getting input, multiply built in velocity with the direction and a speed
14     if direction:
15         velocity = velocity.lerp(direction * speed, acceleration)
16     #If there's nothing in direction, stop moving
17     else:
18         velocity = velocity.lerp(direction * speed, deceleration)
19     #Call the built in move and slide function
20     move_and_slide()
21

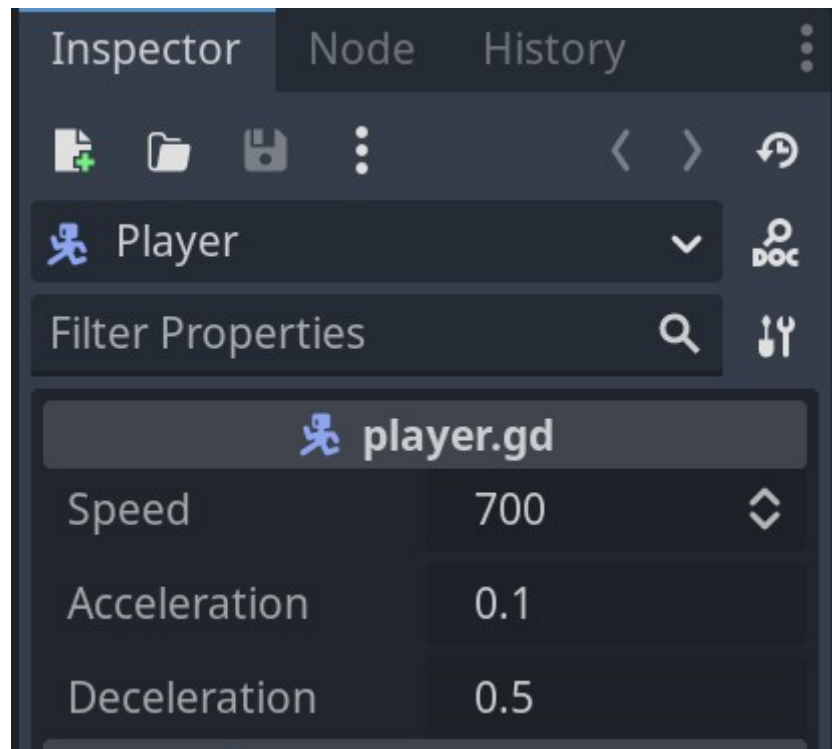
```

First I'm declaring some variables with the `@export` line, and I'm also declaring them by specifying what type they are. Our speed is an integer which means it's a whole number while we also have some new values namely acceleration and deceleration that are both floats which are numbers that can be fractions as well.

The `@export` option allows us to modify the values from the editor from the player node which is super handy when tuning the values.

The direction is still exactly as it was before, but under if direction we are now using a lerp or linear interpolation function. What this does is it will move velocity towards the end goal of going the full direction * speed value, using the acceleration value as a modifier. The same code is in the else: statement with the exception that we are using deceleration there.

This makes movement a bit smoother, and with the `@export` options we can now access all the values we need to play around straight from the player node in the inspector:



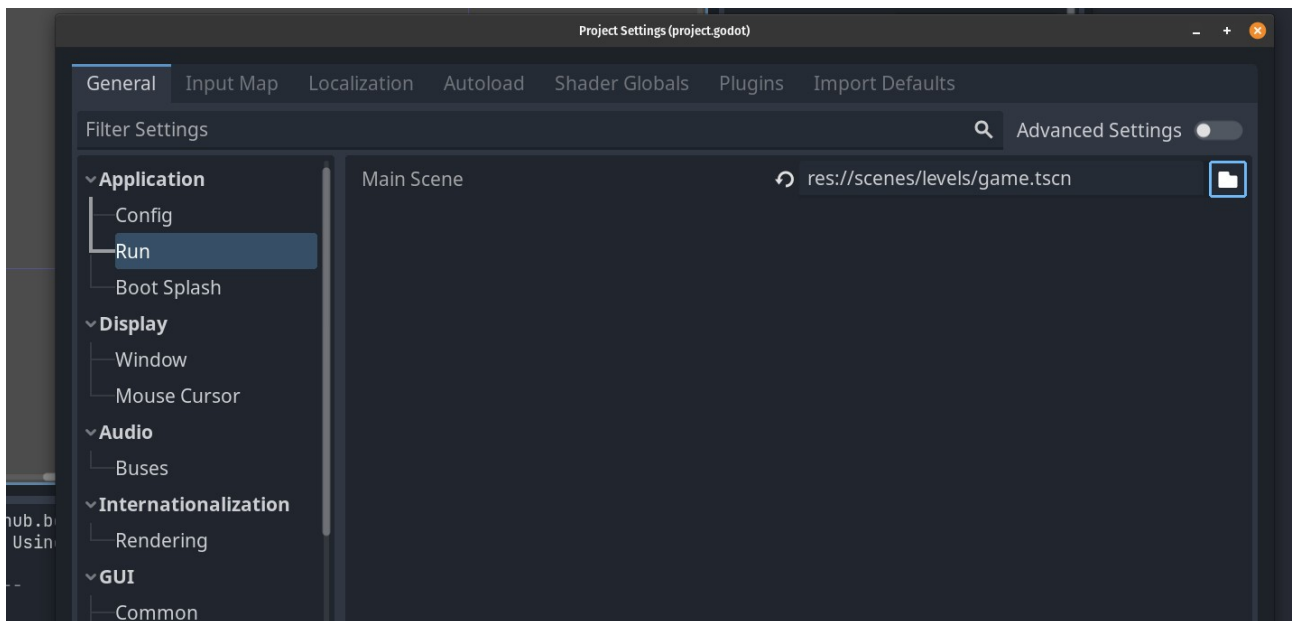
I recommend playing around with the values a bit, see how they affect movement.

That's our movement code done! Next up we will be working on shooting, so we'll set up some kind of placeholder weapon for that so the player can attack.

Player Shooting

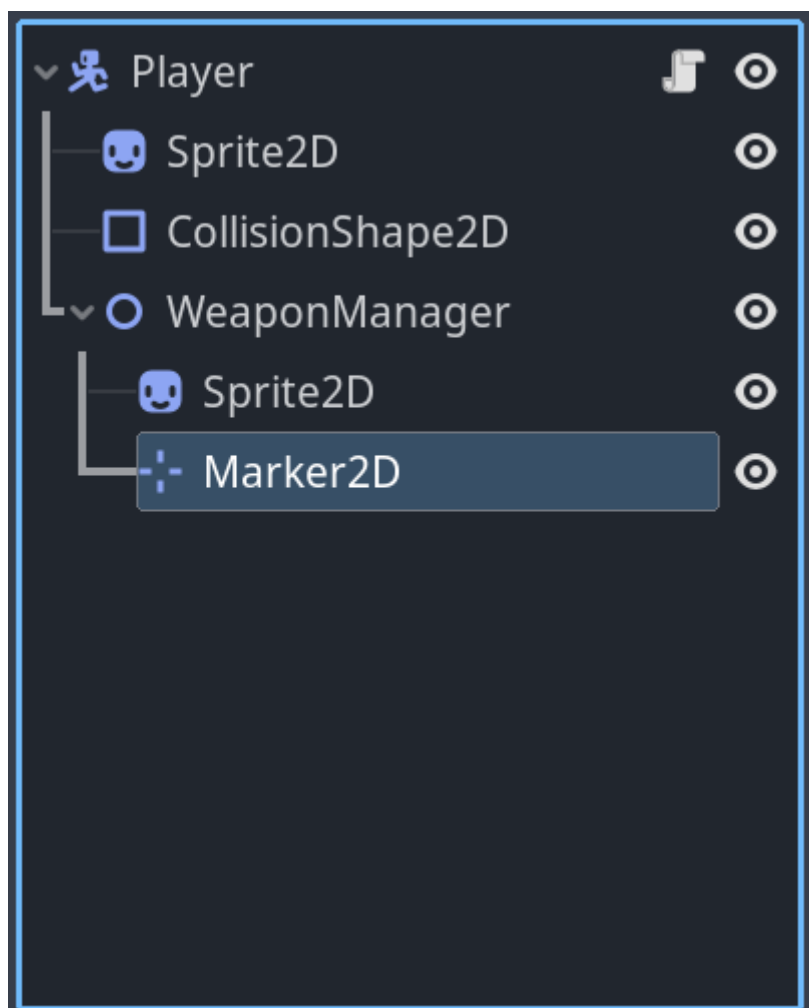
For the player to be able to shoot we need to first make a new scene that will act as the level for the game. So we will create a new scene by going to scene → new scene and choosing 2D scene as the root node. Let's call this scene game and add in the player scene. I'll save this scene under the scenes folder and make a new folder called levels. This is in case I later want to make more levels I have a place for them.

Next we will make this scene the main scene in the project, which we do by going to project → project settings and in the General tab under application → run we can set the main scene:



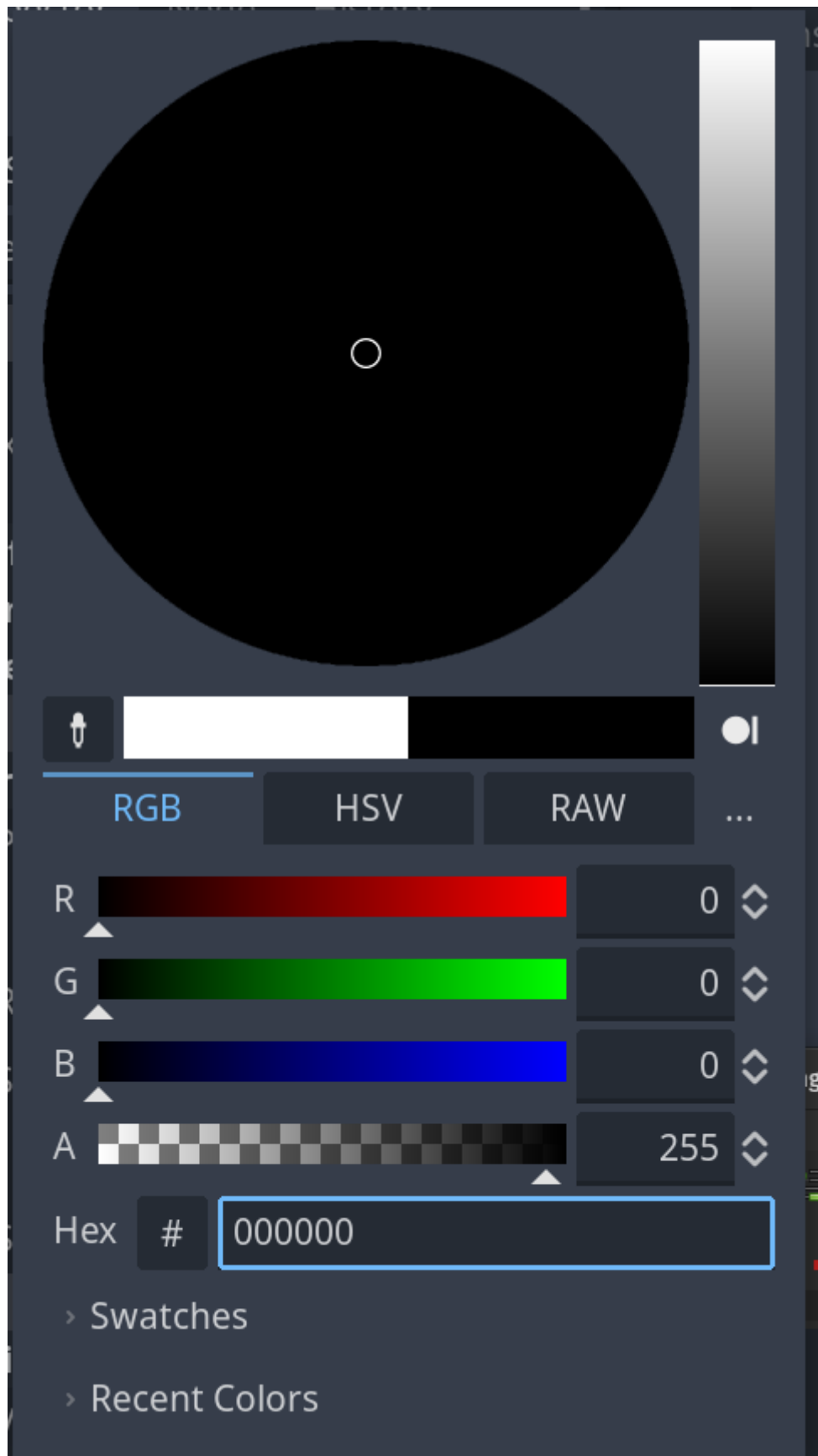
Now when we run the game with F5 or the play button we will be running the Game scene.

Next up we need to edit the player node a little bit to make it look like the player has a weapon, so I'm going to go back into the player scene and make a new node2d as a child of the player and call it WeaponManager. Under this node I will add a Sprite2D that will act as our weapon and a Marker2D that will be used to know where we will be shooting from.

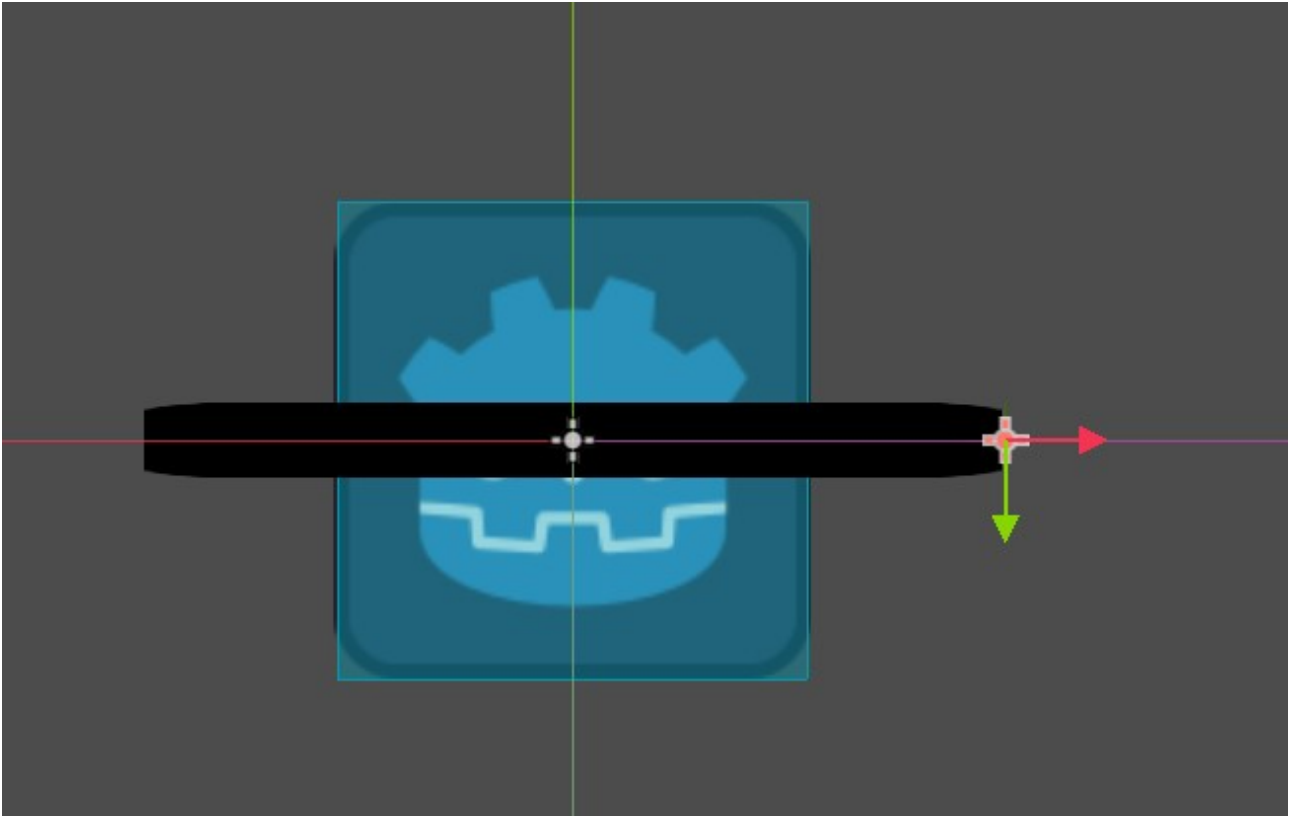


So now we need to add a sprite to the sprite2d node, I will just use the icon.svg again and change it up a bit by scaling it in the transform and then I'll change it's modulate settings under visibility.

I will scale the sprite2d by pressing on the scale tool and then change the modulation in the inspector, I'll simply make this all the way black.



Next I will drag the marker to the right, roughly to the end of the “weapon” using the move tool:



Next we need to make a projectile to shoot so I will make a new scene with a root node of Area2D and call it projectile. I'll save it under scenes/projectiles and set it up like so: First I'll add a Sprite2D that, once again, uses the icon.svg graphic. I'll modulate the visibility to be another color, and then I'll add a collision shape. All this should be familiar by now.

This makes us ready to dive into the code. Let's start by attaching a script to the WeaponManager node. I'll save the script in the player folder because it will be used by the player. Next we'll do some changes to the player script:

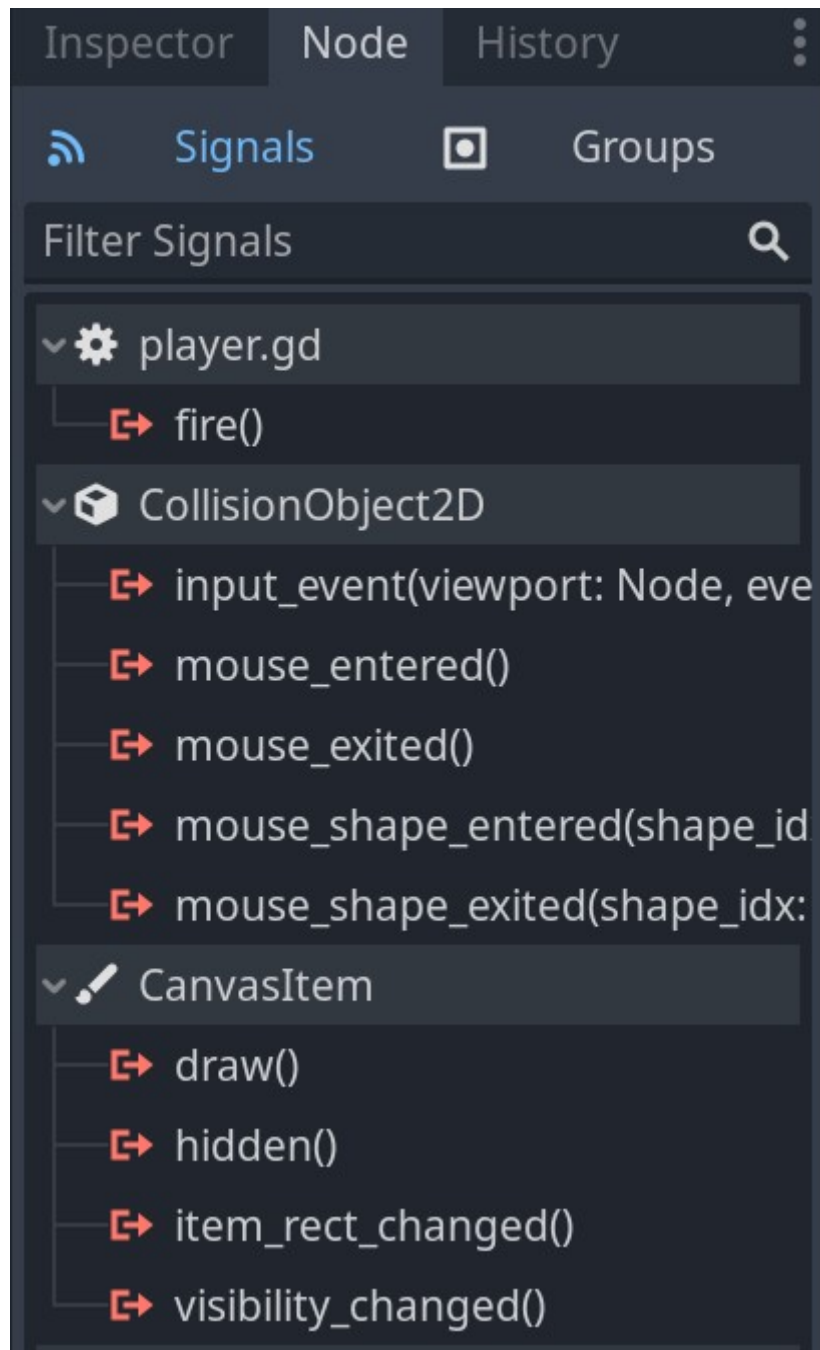

```

1  extends CharacterBody2D
2
3  @export var speed: int = 700
4  @export var acceleration: float = 0.1
5  @export var deceleration: float = 0.5
6
7  signal fire
8
9  func _physics_process(delta):
10     #Get direction from input
11     var direction = Input.get_vector("move_left", "move_right", "move_up", "move_down").normalized()
12
13     #If direction is getting input, multiply built in velocity with the direction and a speed
14     if direction:
15         velocity = velocity.lerp(direction * speed, acceleration)
16     #If there's nothing in direction, stop moving
17     else:
18         velocity = velocity.lerp(direction * speed, deceleration)
19     #Call the built in move and slide function
20     move_and_slide()
21
22     if Input.is_action_pressed("shoot"):
23         emit_signal("fire")
24

```

The new parts here are on line 7 and 22. We're declaring a new custom signal called fire, signals are something we haven't gone through yet but they are very useful for a variety of things. Here we will be using a signal to tell the weapon manager when we're shooting.

On line 22 we are adding a if statement to check if the player is holding down the shoot button and emitting the fire signal if so. If we wanted to only shoot once per every click we would use `is_action_just_pressed` instead. Next we need to connect the signal to the weapon manager, after saving this script we can go to the player nodes inspector, click the node tab and there we should see all signals:



If we double click the fire() signal we get this window:

Connect a Signal to a Method

From Signal:


fire()

Connect to Script:


Filter Nodes

Go to Source

▼  Player (Connecting From) 

 Sprite2D

 CollisionShape2D

▼  WeaponManager 

 Sprite2D

 Marker2D

Receiver Method:

_on_fire

 Pick

Advanced ☐

Cancel

Connect

We want to choose WeaponManager and then click connect.

Now we should have a new function in weapon managers script. I will make some more changes to the code and walk through them here:

```
1  extends Node2D
2
3  @onready var projectile = preload("res://scenes/projectiles/projectile.tscn")
4  @onready var marker_2d = $Marker2D
5
6  func _process(delta):
7      look_at(get_global_mouse_position())
8
9
10
11 func _on_player_fire():
12     var instance = projectile.instantiate()
13     add_child(instance)
14     instance.global_transform = marker_2d.global_transform
15
```

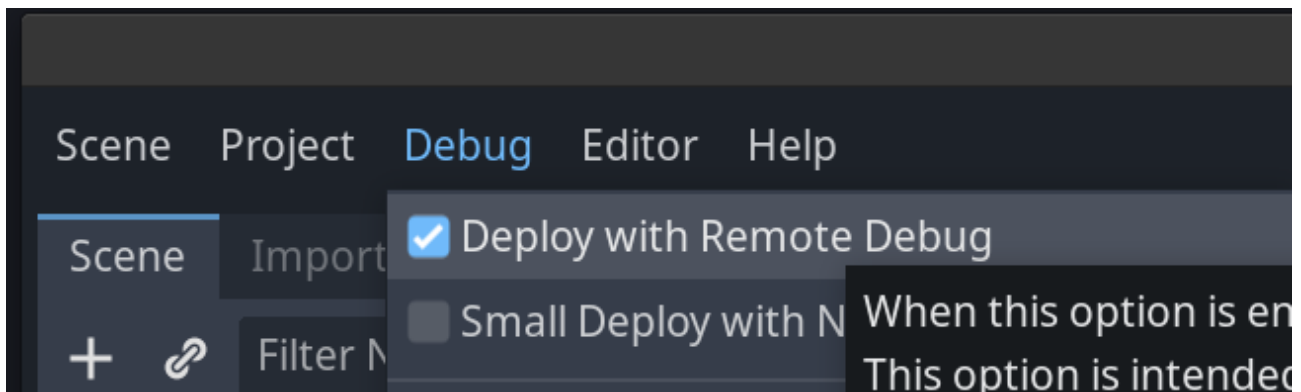
First we have two @onready declarations, one for the projectile where we're preloading in a scene. This lets us use the projectile scene later in the code, if you saved your projectile elsewhere you need to get the correct path for it. You can get the path by right clicking the projectile.tscn and then selecting "copy path", this lets you paste it in the preload.

For the next line we're simply getting a reference for the Marker2D, this is where we will spawn our projectiles.

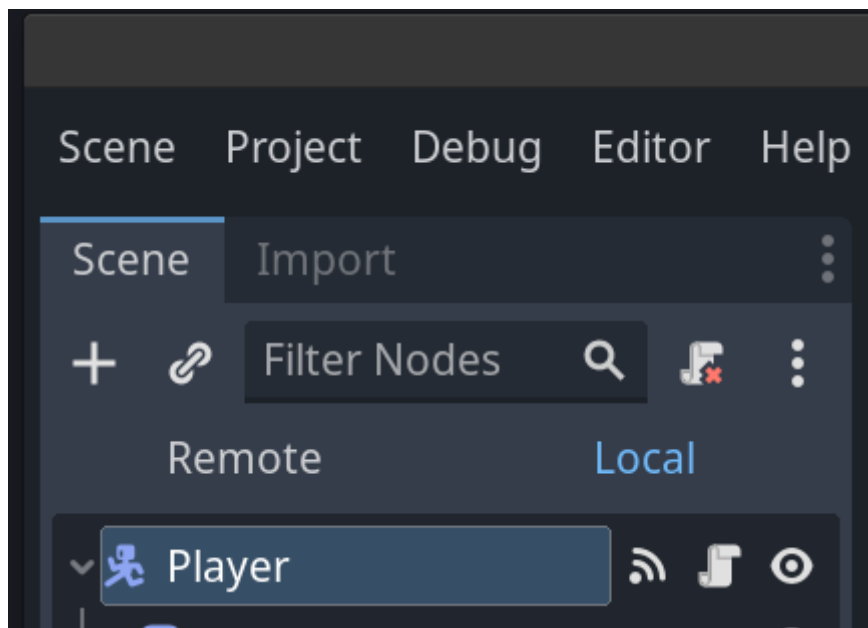
In _process we are using a function called look_at and in that we are getting the global mouse position. This will make our weaponmanager look at wherever the mouse is.

Then in _on_player_fire we are first declaring a new variable called instance and the instantiating a projectile under this new variable. Then we are adding instance as a child and setting the instances transform as the marker2d's transform. We are using transform here rather than position because transform also contains the rotation.

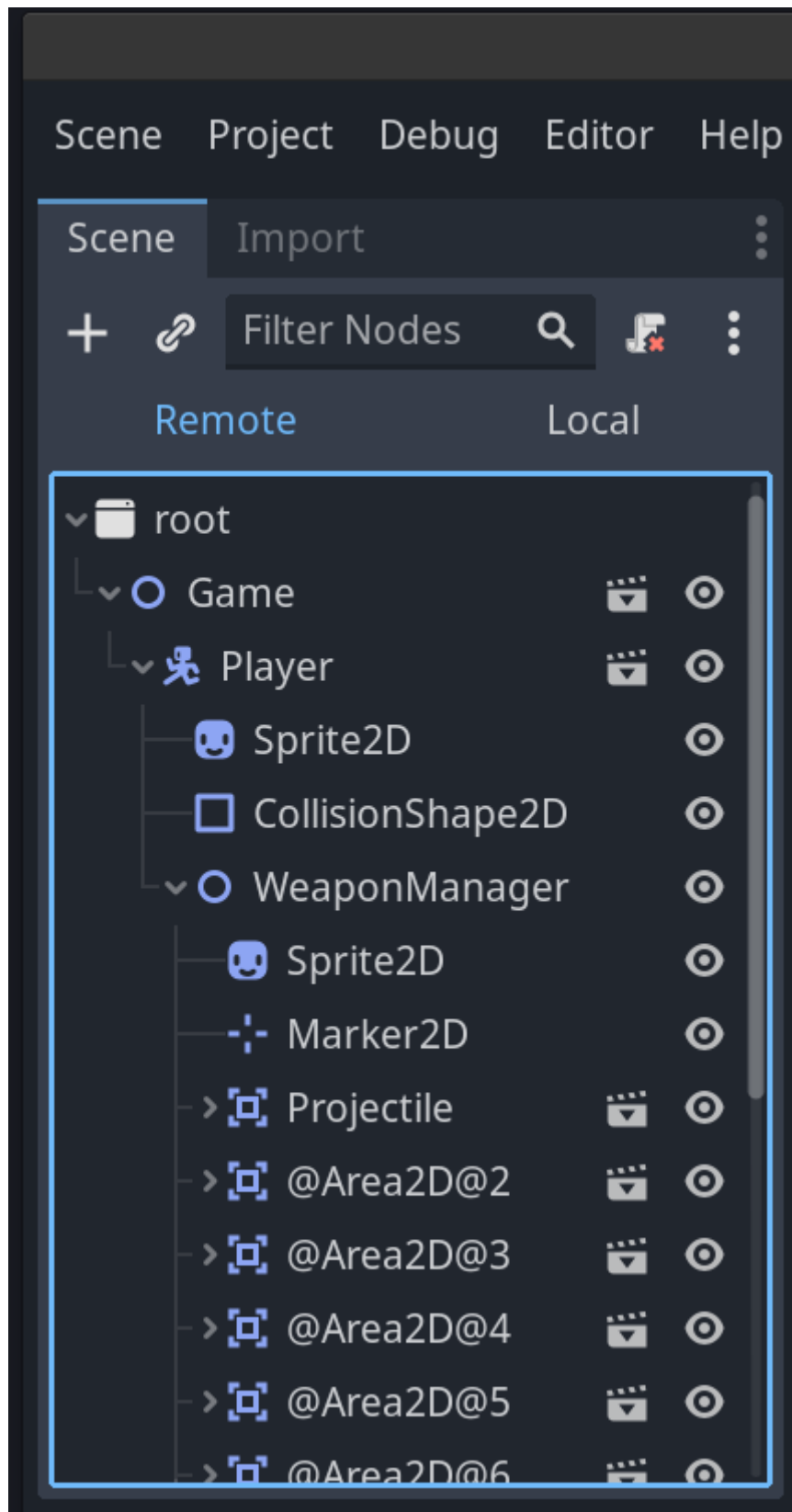
Now when we run the game and press fire we will spawn in the projectile but there are some issues with it as is. Currently it doesn't shoot away from the player and it also moves with the player, so lets work on fixing these issues. First I want the projectile to not stick to the player, to do this I need to somehow add it to the level instead of adding it as a child to the weaponmanager. We can check where these projectiles are spawning in the scene tree by enabling a debug option called "deploy with remote debug" under the Debug menu:



With this option enabled, when running the game if we go back to the scene tree we can switch between local and remote view

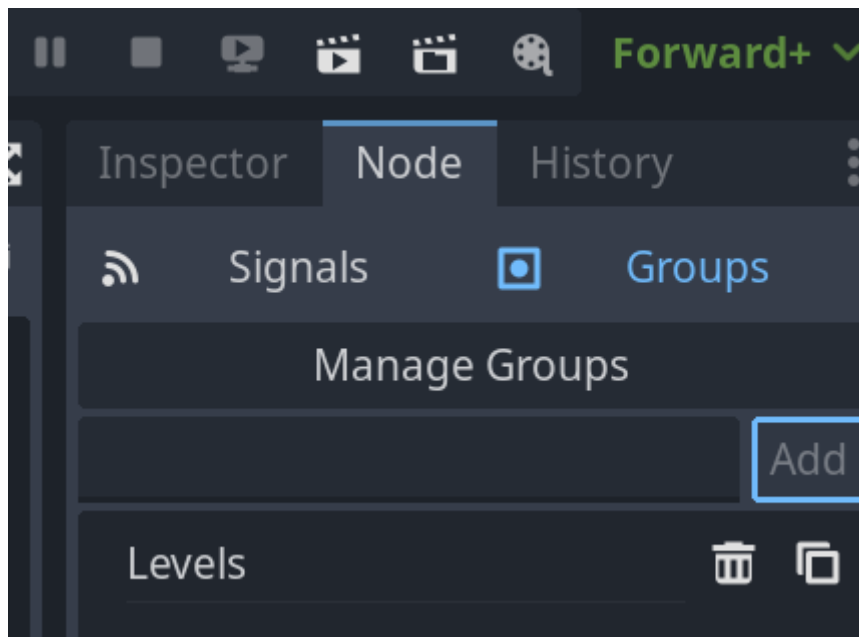


By selecting remote we can see all the nodes currently in the scene. If I shoot a bit we will see some projectiles there after expanding the trees:



So we know now that these are not spawning in the correct place, we want them to spawn under the game node.

To enable us to do this we will use a feature in Godot called groups. If we go to the game scene and choose the root node, then in the inspector, under node we can choose groups and add a new one to the root node. It's important to add this to the correct node. I'll add in a group called "Levels":



Just write in the group name and click add and that's it! Back to our weaponmanager script, we can now do this:

```
1  extends Node2D
2
3  @onready var projectile = preload("res://scenes/projectiles/projectile.tscn")
4  @onready var marker_2d = $Marker2D
5  var level
6
7  func _ready():
8      level = get_tree().get_first_node_in_group("Levels")
9
10 func _process(delta):
11     look_at(get_global_mouse_position())
12
13
14
15 func _on_player_fire():
16     var instance = projectile.instantiate()
17     level.add_child(instance)
18     instance.global_transform = marker_2d.global_transform
19
```

We're declaring a variable called level on line 5 and adding in a _ready function where we populate the level with the first node in the group "Levels". get_tree will search the whole scene tree we can see from the remote debug view, so we need to make sure there's only ever one node with "Levels" group loaded in but this should not be an issue for this project.

Then in the _on_player_fire function we are adding "level." to the add_child line. Now when we shoot the projectile sticks around where it was spawned instead of moving with the player.

But it's still not moving. Luckily this is an easy fix, we simply need to add a script to the projectile

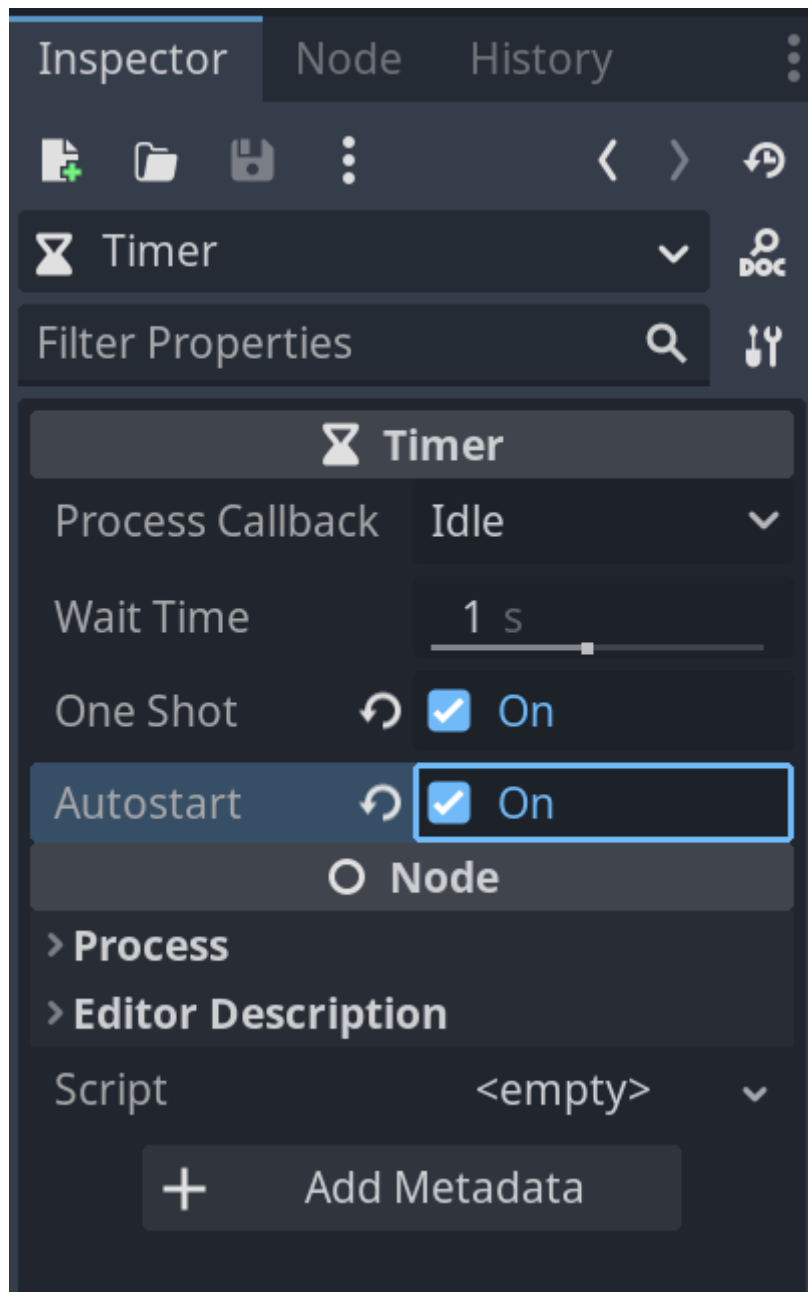
scene and then make it move which is easy enough. By this point you should know how to attach a script and save so I'm just going to show the code here:

```
1  extends Area2D
2
3  class_name Projectile
4
5
6  var speed: float = 25.0
7
8  func _physics_process(delta):
9      position += transform.x * speed
10
```

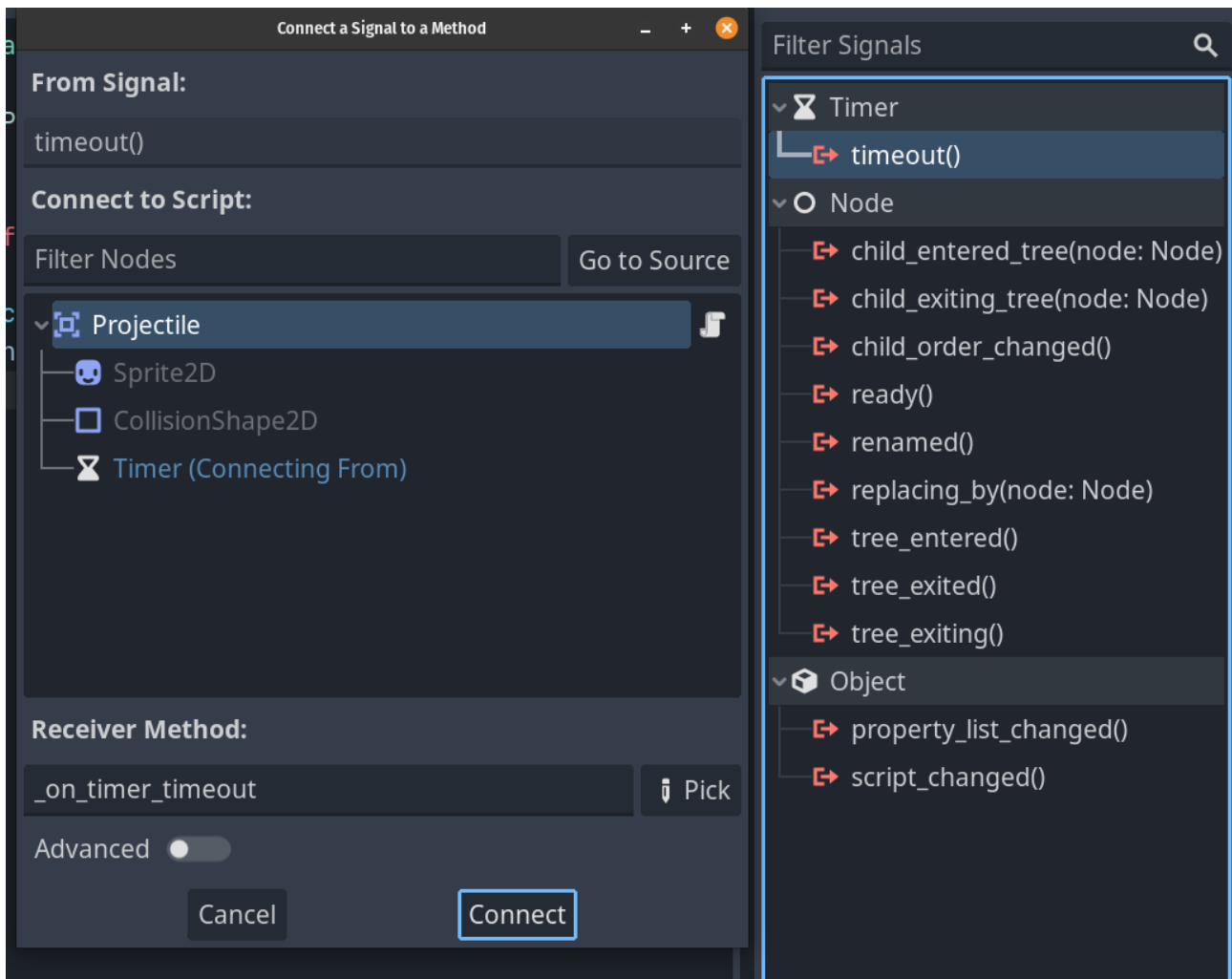
We saw something similar in the first part, we are simply adding speed to the position along the X axis. We are also declaring the projectile as a class which will become handy later when we talk about inheritance.

For now, we can run the project and shooting should work as expected! There are still a couple of problems though. For one, if you are using `Input.is_action_pressed` then we are shooting way too rapidly and also the projectiles will stick around forever which in the end will start causing issues.

Since we're already looking at the projectile script we can fix that issue first. We need to add a new node called a timer and set the properties "one shot" and "autostart" to true. The default time should be fine:



Next we need to connect a built in signal called timeout to the projectile script:



Now we have a new function called `_on_timer_timeout` that will be called when the timer runs out, so that is one second after the projectile enters the scene.

In the function we need one line:

```
11
12 func _on_timer_timeout():
13     queue_free()
14
```

`queue_free` will in essence delete this node and any nodes under it, so now we can shoot and projectiles will disappear after one second.

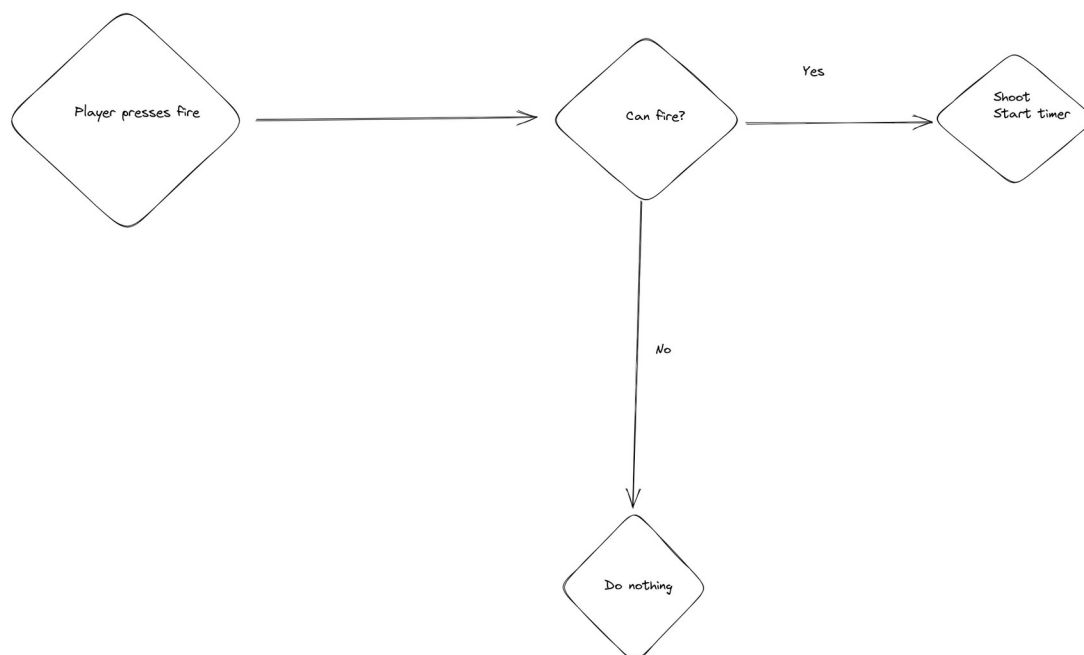
Next we will fix the fire rate issue. Let's take another look at how we can add timers, because we can also add them in code. So over to the `weaponmanager` script we will make some changes again:

```

1  extends Node2D
2
3  @onready var projectile = preload("res://scenes/projectiles/projectile.tscn")
4  @onready var marker_2d = $Marker2D
5  var level
6  var can_shoot = true
7
8  func _ready():
9      level = get_tree().get_first_node_in_group("Levels")
10
11  func _process(delta):
12      look_at(get_global_mouse_position())
13
14
15
16  func _on_player_fire():
17      if can_shoot:
18          can_shoot = false
19          var instance = projectile.instantiate()
20          level.add_child(instance)
21          instance.global_transform = marker_2d.global_transform
22          var timer: Timer = Timer.new()
23          add_child(timer)
24          timer.wait_time = 0.2
25          timer.one_shot = true
26          timer.timeout.connect(func(): can_shoot = true)
27          timer.start()
28

```

First we have a new variable called “can_shoot” and everything in _on_player_fire is in an if statement that checks if this is true. If true we immediately set it to false, do the normal shooting stuff and then declare a new variable called timer of type Timer and instantiate a new Timer, then we add this as a child, set the wait time to 0.2, one shot to true and then we do something interesting. On line 26 we’re connecting the timeout signal via code but we are declaring the function in the same line. This function basically sets the can_shoot variable to true again after a certain time. Finally we actually start the timer. So to sketch this whole thing out, it works like this:



So it's not that complex once drawn out. This code would also work to limit the fire rate of semi automatic weapons. I recommend playing around with the timer value a bit to get it feeling like it's "right". Though how things feel will change once we get the real graphics in.

Next we need something to actually shoot so on to the next section:

Enemies

We want enemies to do a couple of things which are, in order:

Detect player

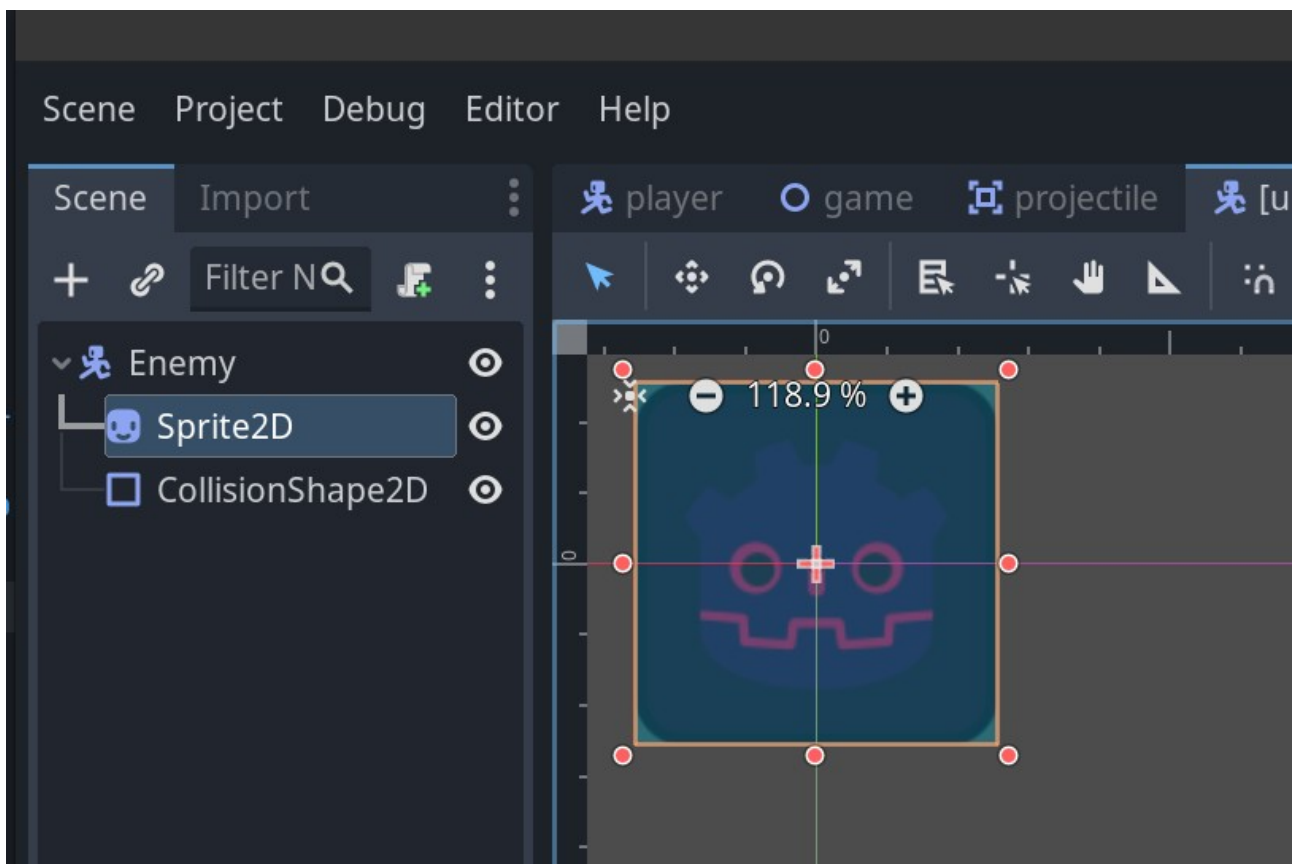
Move towards player

Attack player if close enough

Take damage and disappear when health is 0

We will just make an enemy that needs to get in melee range with the player to attack first. So let's make an enemy and make it move first of all.

For the enemy we will use a root node of CharacterBody2D, then use a Sprite2D with the icon and a collision shape. So we end up with this:



I adjusted the modulation of the sprite a bit to differentiate it. Next I will save it under scenes/enemies and let's talk about movement.

We want the enemy to move towards the player, so the enemy needs to know the location of the player. There are multiple ways to do this, but I will use the group mechanic again, so I will add the player scenes root node to the group “Player”. Next I will add this script to the enemy:

```
1  extends CharacterBody2D
2
3  class_name Enemy
4
5  var health = 100.0
6  var speed = 200.0
7  var target
8
9  func _physics_process(delta):
10     if target == null:
11         target = get_tree().get_first_node_in_group("Player")
12     if target != null:
13         velocity = global_position.direction_to(target.global_position) * speed
14         move_and_slide()
15         look_at(target.global_position)
16
```

So first we need to get a target which will be the player so I have a variable simply called target. On line 10 we check if this target is null, in which case we use the `get_first_node_in_group` function which we’ve done before. This should always work because there should only ever be the one player present, if we had a multiplayer game with multiple players we would need to work some additional magic where we’d work out which player was the closest to the enemy.

If the target is not null which is what `!=` checks, we set the velocity to equal the result of a function called `direction_to` from our global position to the targets global position. We multiply this with the speed variable. Then we `move_and_slide` and use the `look_at` function to rotate towards the player.

Now if you add an enemy scene to the main scene and run the game, the enemy should chase the player around!

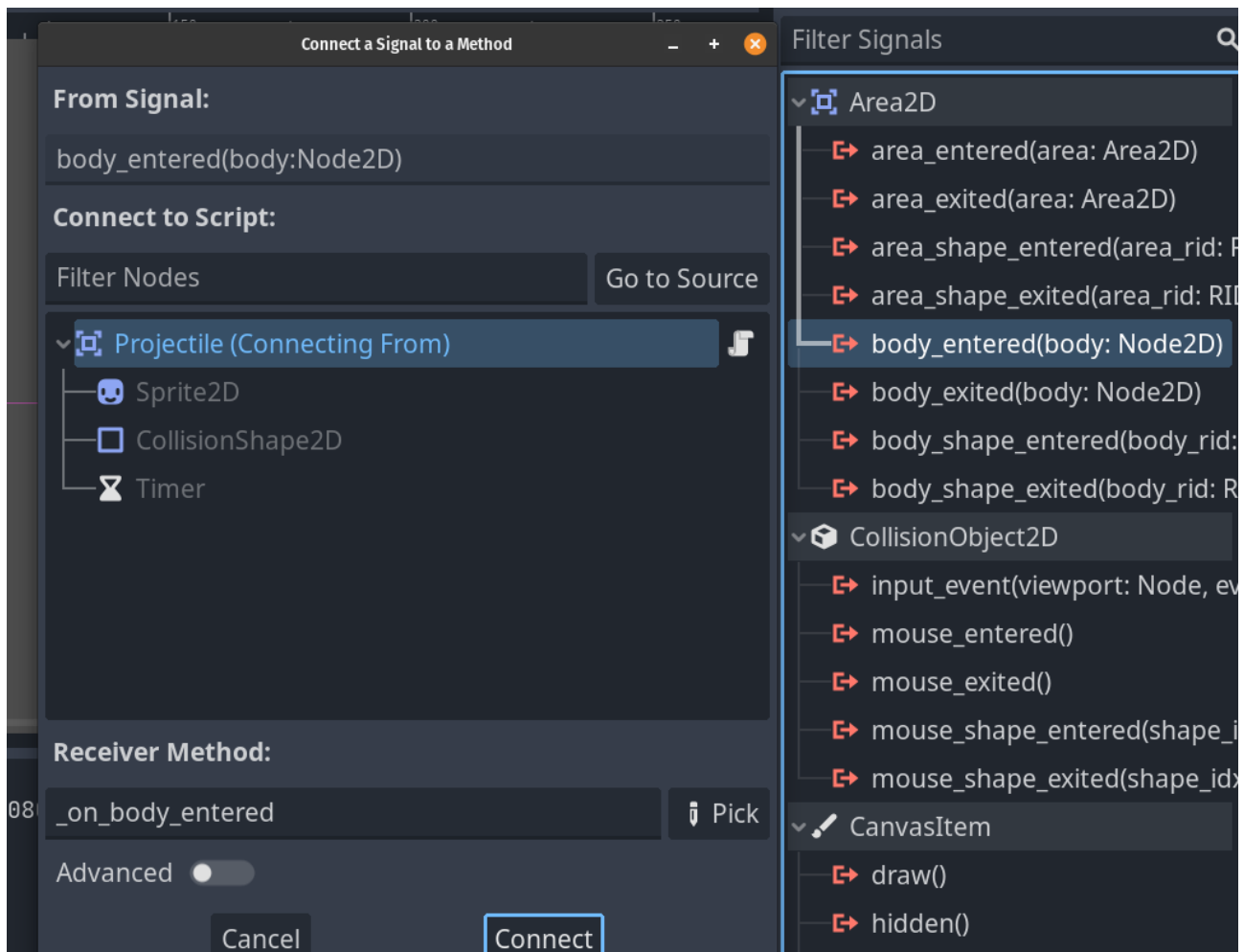
Next we need to make it so the enemy actually gets damaged. We can do this pretty quickly by simply modifying our enemy and projectile script. We need to first make a function in the enemy script:

```
func hit(dmg:float) -> void:
    print("Enemy hit")
```

I put this below the `_physics_process` function, I also noticed that I’ve been using a mix of static typing and dynamic typing during this tutorial so we will eventually go fix that, but for now this is a simple function called `hit` that takes an argument called `dmg` of type float and the `-> void` means this function does not return anything.

For now all we will do in this function is print that the enemy got hit.

In the projectile scene we need to hook up a signal to the script, namely the body entered signal:



The standard name will be fine, just connect it to the projectile script and we will make the function look like so:

```
func _on_body_entered(body):
    if body.has_method("hit"):
        body.hit(5)
```

In this code we check if the body we are hitting with the projectile has the method or function “hit”, and if it does we call that with the value 5 for damage. Using the `has_method` allows us to later make other things that can also be hit by projectiles by simply having the hit function. If you now shoot the enemy you should see some prints in your output:

```
Godot Engine v4.1.1.stable.flathub.bd6af8e0e - https://godotengine.org
Vulkan API 1.3.242 - Forward+ - Using Vulkan Device #0: NVIDIA - NVIDIA

Enemy hit
Enemy hit
Enemy hit
Enemy hit
Enemy hit
Enemy hit
--- Debugging process stopped ---
```

Of course this isn't quite enough but we are close to having an enemy that is defeatable. First I'll make an @export variable for the damage in projectile so I can more easily adjust it as needed, I'll also make the speed an @export variable so the complete code block looks like this:

```
extends Area2D

class_name Projectile

@export var damage: float = 25.0
@export var speed: float = 25.0

func _physics_process(delta):
    position += transform.x * speed

func _on_timer_timeout():
    queue_free()

func _on_body_entered(body):
    if body.has_method("hit"):
        body.hit(damage)
        queue_free()
```

Now we are passing 25.0 as the damage value but we can adjust it. I also added a queue_free() to the signal function so the projectiles disappear after hitting something with the hit function.

Now we need to look at the enemy script some more. Because we already have a variable for health we only need to modify the hit function like so:

```
func hit(dmg:float) -> void:
    health = clamp(health - dmg, 0, 100)
    if health == 0:
        queue_free()
```

Now we are simply subtracting the damage from the health, using a new function called clamp to ensure that the health doesn't go below 0 or above 100 (this could be useful if we added health bars to the enemy later) and checking if health is 0 to queue_free the enemy.

Ideally before we queue_free the enemy we would play some kind of animation on them but we don't have any graphics yet so this will do for now.

Next we need to make the enemy actually attack the player when the enemy is close enough. First the player needs a function for taking damage first, so we need a health variable I'll set to 100 and a hit function that will, for now, just print out the health because we don't want to queue_free the player. So the complete player script will look like this:

```
extends CharacterBody2D

@export var speed: int = 700
@export var acceleration: float = 0.1
@export var deceleration: float = 0.5
@export var health: float = 100.0

signal fire

func _physics_process(delta):
    #Get direction from input
    var direction = Input.get_vector("move_left", "move_right", "move_up",
    "move_down").normalized()
```

```

        #If direction is getting input, multiply built in velocity with the
direction and a speed
        if direction:
            velocity = velocity.lerp(direction * speed, acceleration)
        #If there's nothing in direction, stop moving
        else:
            velocity = velocity.lerp(direction * speed, deceleration)
        #Call the built in move and slide function
        move_and_slide()

        if Input.is_action_pressed("shoot"):
            emit_signal("fire")

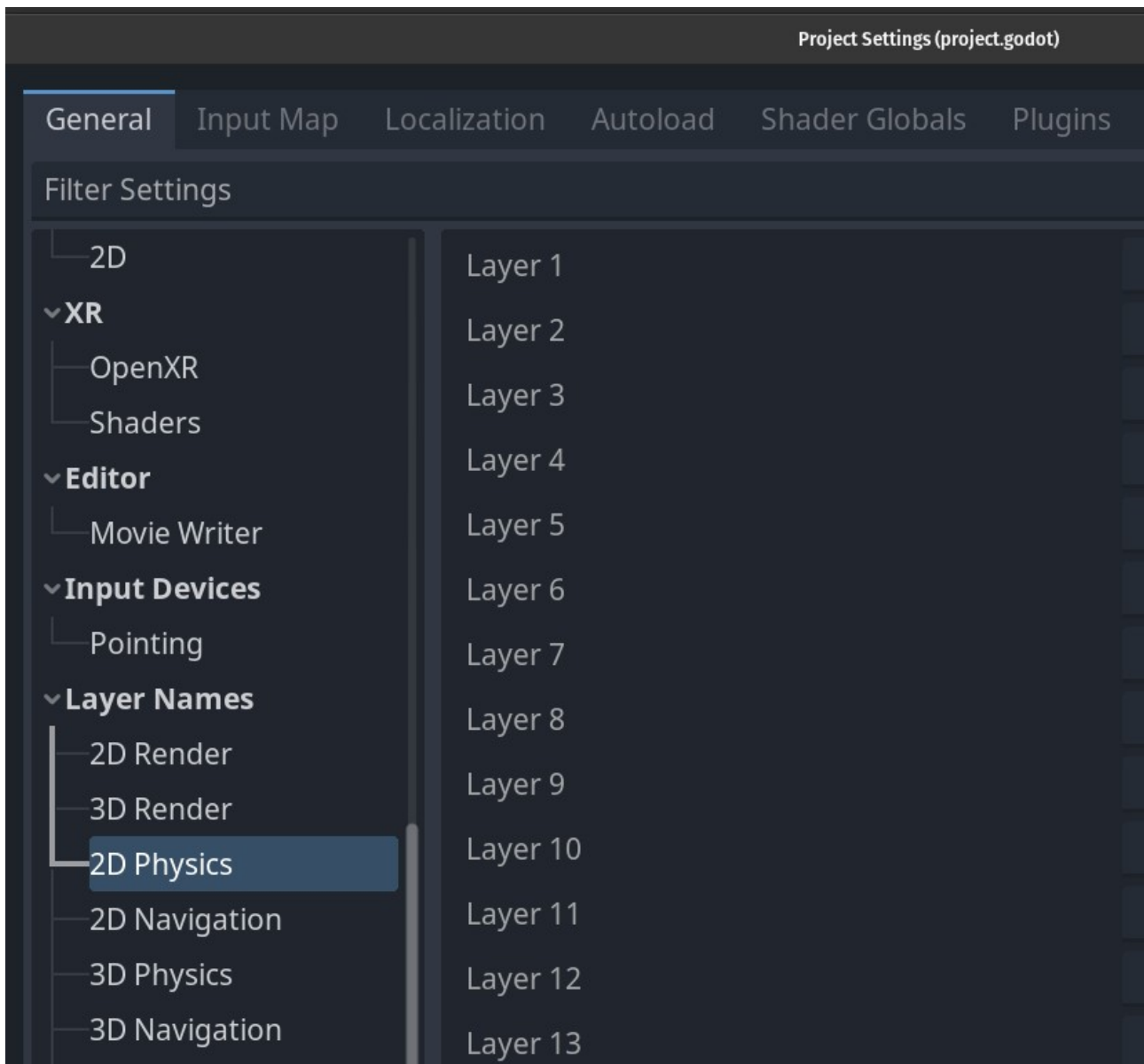
func hit(dmg:float) -> void:
    health = clamp(health - dmg, 0, 100)
    print(health)

```

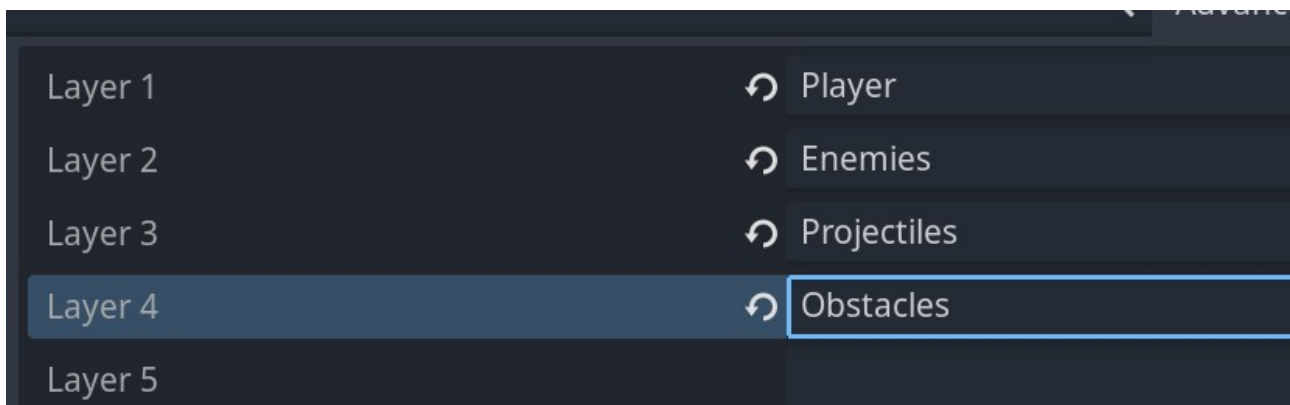
This is the complete player script now, but there may be a problem depending on your collisionshapes. You might be taking damage from your own projectiles! We need to fix this first of all, and there are some ways we could do it. We could just rename the function to something else, we could use something called collision layers or we could check in code what shot the projectile and not hit that.

In the interest of learning, we will use the collision layers. They can be very useful for multiple different game types, so we will set them up.

In project settings we can scroll down on the side window until we find 2D Physics, there we can set up the layer names:



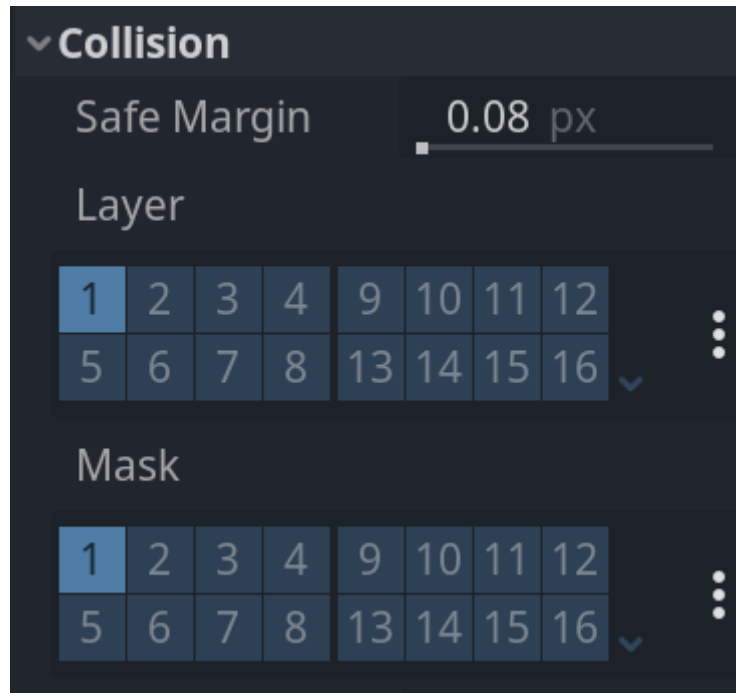
I will set up some layers called Player, Enemies, Projectiles and Obstacles. I'm not sure if we will use Obstacles yet but might as well add it while I'm here.



This now helps us keep the layers straight since they are named.

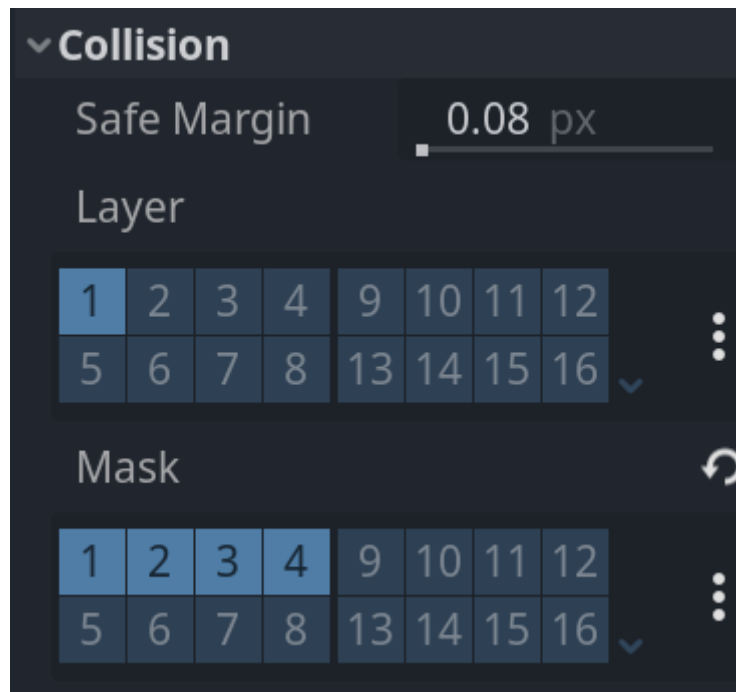
Next we need to actually edit the nodes that will use the collision layer information, so the Player

root will be first. In the inspector there is a section called collision with the following view:



The layer is which layer this node is on. The node can be on multiple on layers if needed. The Mask, meanwhile, is which other layers this node can interact with.

So for the player I have it set up like this for now:

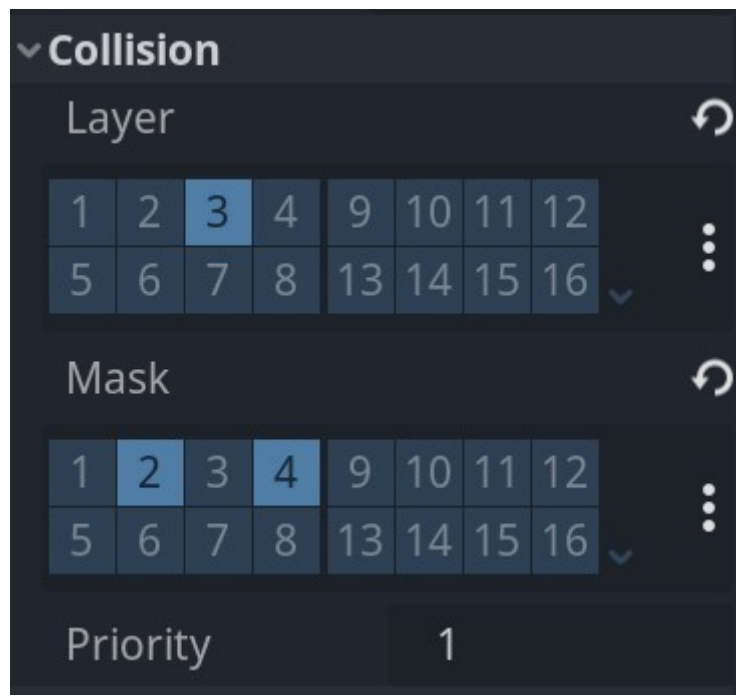


You can see the names of the layers by hovering the mouse over them. You can also set the values by clicking on the three dots on the side which will show the names instead of the numbers.

So the player is on layer 1 and can collide with all other layers. The player wouldn't really need to collide with itself but in case we added a player 2 for multiplayer (which we are not doing in this

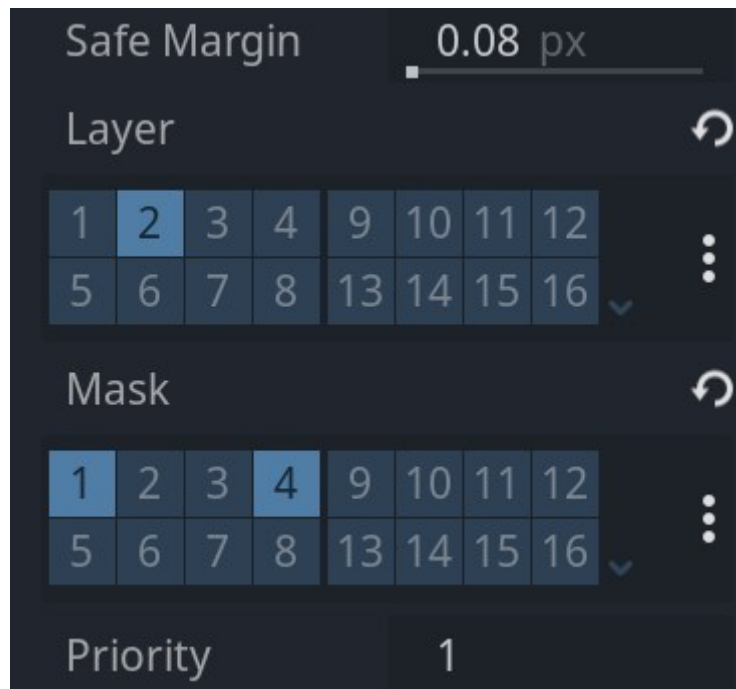
tutorial but just for future proofing) we want the players to collide with each other.

Next our projectile looks like this:



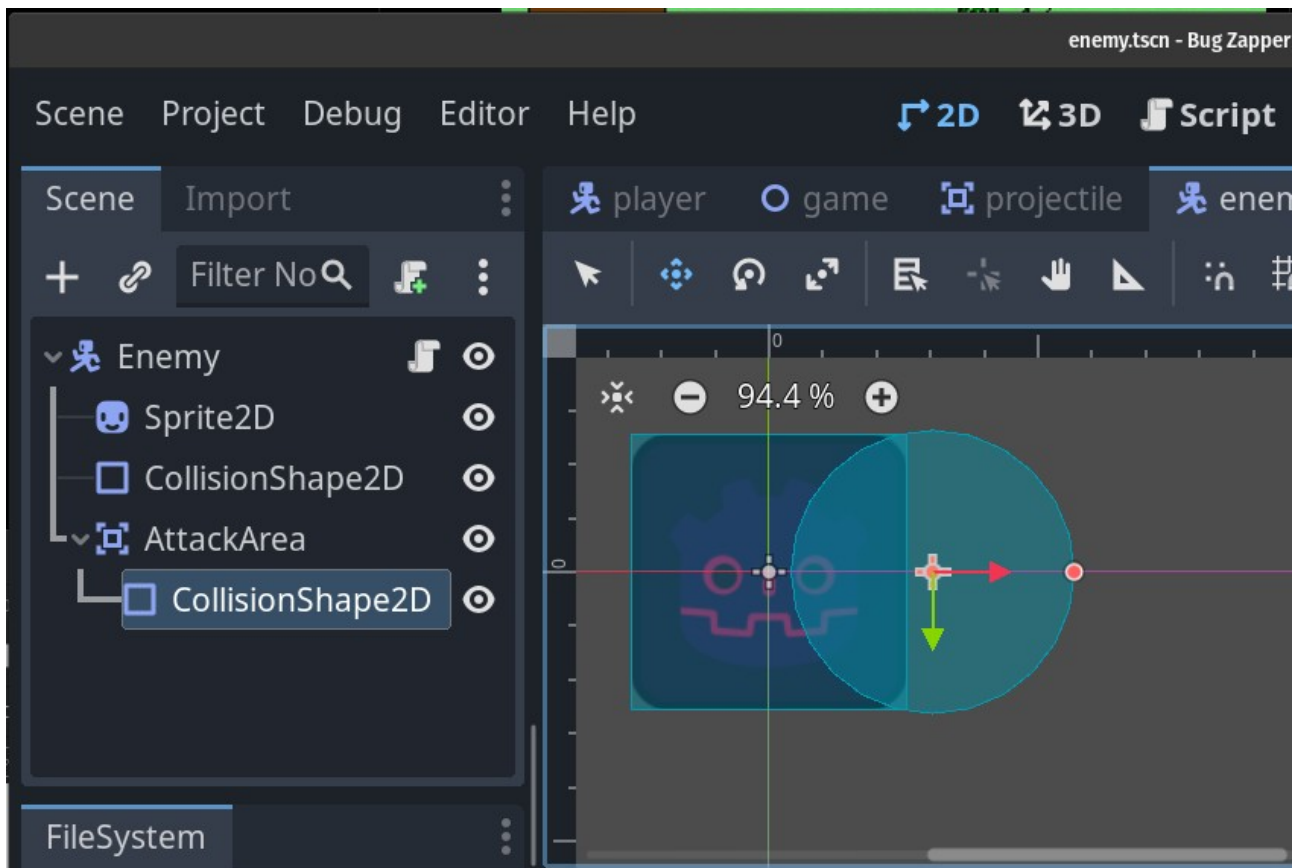
If we had an enemy projectile the mask would be 1 and 4 instead of 2 and 4.

Finally the enemy:



So now we should be all set up for collision layers.

Next up, we need to add the attack to the enemy. First I want to add an area2d to the enemy I'll call AttackArea and give it a collisionshape. This is the area of the enemy attack so I will move it to the front of the enemy which is to the right:



Something like that will be fine. Next I will set the AttackArea collision layers to be correct, so the layer will be on the enemy layer and the mask will be the player.

Finally I will connect the body entered signal of the AttackArea to the Enemy script and make a simple test in it:

```
func _on_attack_area_body_entered(body):
    print(body)
```

This will simply print the info of the body the AttackArea touches for now, just so we can test it actually works.

Next we need to work out how we want the attack to work. Here's what I'm thinking:

When the player is in the attack area, the enemy should stop moving, have a small charge up window for the attack and then attack. There will be some timers involved here, as we don't want the enemy to be constantly attacking.

We also need to connect the attack areas body exited signal so we can keep track of if the player is in the attack area or not.

So we're giving the enemy script a bit of a rewrite again, here's the complete script, I'm also fixing some static typing in this script while I'm at it:

```
extends CharacterBody2D

class_name Enemy

@export var health:float = 100.0
@export var speed:float = 200.0
var target:Node
```

```

@export var attackTimer: float = 0.5
@export var dmg:float = 15
var attacking:bool = false
var playerInArea = false

func _physics_process(delta) -> void:
    if target == null:
        target = get_tree().get_first_node_in_group("Player")
    if target != null && !attacking: #only move when not attacking
        velocity =
global_position.direction_to(target.global_position) * speed
        move_and_slide()
        look_at(target.global_position)
    if target != null && playerInArea && !attacking: #Only start attack if
player in area and not already attacking
        attacking = true
        await get_tree().create_timer(attackTimer).timeout
        if playerInArea: #only damage player if player is still in
area!
            target.hit(dmg)
            attacking = false

func hit(dmg:float) -> void:
    health = clamp(health - dmg, 0, 100)
    if health == 0:
        queue_free()

func _on_attack_area_body_entered(body) -> void:
    playerInArea = true

func _on_attack_area_body_exited(body) -> void:
    attacking = false
    playerInArea = false

```

So we are doing a few new things here. We have variables tracking how much damage the enemies do, how long their attacks take, if they're currently attacking and if a player is in the attack area.

This allows us to stop moving when attacking by modifying the if statement in the physics process and adding another if statement to start the attacks if the player is in the area and we are not currently attacking. In that case we use a new feature of Gdscript called Await that lets the enemy wait until a timer times out before triggering the hit function in the player, which we only do if the player is still in the area.

The signals we used simply set playerInArea to true or false and if the player leaves the area the enemy stops attacking. Now when we run the game the enemy can damage the player!

That's it for this section of the tutorial. In the next section we will start planning and making the graphics.

If you want to ensure you start with the exact same project I am, feel free to download the bug zapper files from this repository.