Welcome, future game developers! My name is Omar Zaki and I'm excited to be your instructor for this course. We will be exploring the robust Godot engine and creating a fun, survivor-style shooter game similar to survivor.io.

Course Objectives

Our journey through this course will be centered around three main objectives:

- Understanding and implementing object pooling
- Learning basic AI movement
- Grasping the use of basic collision layers

These learning outcomes are designed to provide you with a well-rounded understanding of game development in Godot.

Course Requirements

Before you dive into this course, there are a few prerequisites you should be aware of. You should have some basic knowledge of Godot, including how to navigate through the engine and access various settings. Familiarity with Godot will make your learning process smoother and more enjoyable.

Diving into Object Pooling

One of the major topics we will be covering in this course is object pooling. We will start by understanding what object pooling is and then delve into its application in Godot.

Object pooling is a crucial concept in game development that can be applied in various scenarios. For instance, we can use object pooling for creating bullets in our game. However, we won't stop at just one application. We will apply object pooling multiple times throughout the course to ensure you gain a thorough understanding of this concept.

About Zenva

Before we get started, let me introduce you to Zenva, our online learning platform. With over a million learners, Zenva offers a wide range of courses for beginners and those eager to learn something new. Our courses are versatile, offering video tutorials, lesson summaries, and project files for a comprehensive learning experience.

So, let's get started with our first lesson! I hope you find this course both fun and informative. Good luck, and I'll see you in the first lecture.

Course Updated to Godot 4.3

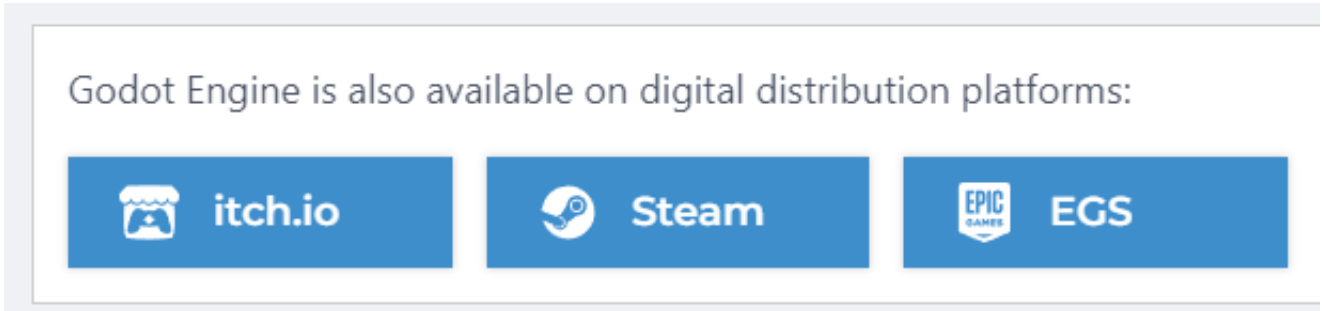
We've updated the project files to Godot version 4.3 for this course – the latest stable release.

How to Install Version 4.3

You can download the most recent version of Godot by heading to the Godot download page here: https://godotengine.org/download/windows/

Then, just click the option for **Godot 4.3.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application file – no further installation steps required!

If you are using another non-Windows operating system, such as MacOS or Linux, you can scroll down on the page to change the download link before hitting the button above.



Welcome everyone to this tutorial. In this lesson, we will be setting up our project and going over some project settings in Godot. This will not be difficult, so let's get started.

Setting up a New Project in Godot

Once you have opened Godot Project Manager, you might see a list of different projects. These could be your personal projects, and you might have more or less. But the first thing we need to do is to create a new project. If you are using Godot 4.2, find the new button. The new button used to be on the right, but it is now on the top left. If you don't see it on the top left, just look around for it. Clicking on the new button will allow us to create a new project.

Enter a name for your project and choose a location to save it. For this tutorial, we will name it "hell_bullets_project" and save it on the desktop. There are three different renders available, but we will choose "compatibility" as we are making a relatively low-key 2D game.

Now, your project should open, and you should see a blank canvas. Head over to the 2D scene, create a 2D scene, rename it to "world", and save it in your main folder. If you hit play, you will now see your game, even though there's nothing in it yet.

Adjusting Project Settings

Let's go into some project settings. In our project settings, there are two main parts that we will be looking at: the window and the filter. In the window, we will divide the viewport width and height by two. But before we do that, we need to copy these numbers into the override in advanced settings.

Scroll down to "mode stretch" and click this to "canvas items". This will ensure that when you stretch your canvas, it will properly stretch anything that is in your scene.

Texture Filtering

By default, the texture is inherited, which means it takes its settings from the project settings. This can result in blurry edges on your images. To achieve a pixelated look, we can change the default texture filter to "nearest".

Now, your image should be pixelated. Note that this setting will apply to all nodes in your project, so you don't need to change the filter for each node individually.

To conclude, in this tutorial, we created our project and covered some basic project settings in Godot. You are encouraged to explore more project settings and play around with different options, especially in the window tab. You might also want to experiment with the "icon", "name", and "description" fields in the config settings.

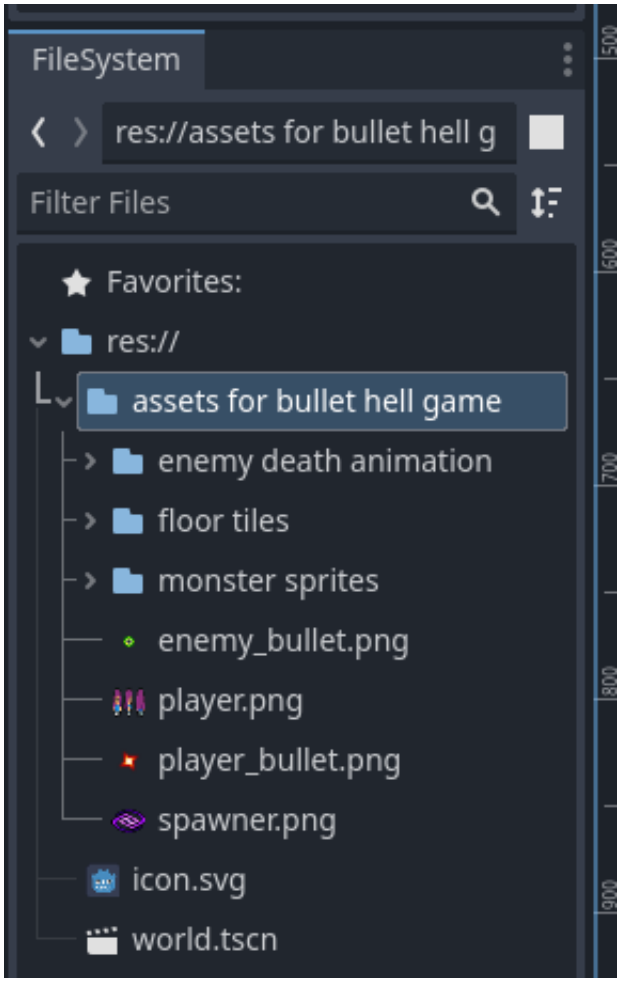
In the next lesson, we will be covering how to set up our tile map and compare tile map versus texture rectangle. See you there.

In the most recent version of Godot, the way we create tilemaps is a bit different. So in this lesson we will be going over that process. This means you do not need to view the next lesson, as the content here will catch you up with the most recent version of Godot.

Importing Assets

In order to design a tilemap, we need textures to represent our different tiles (grass, stone, sand, etc). In the **Course Files** tab, there is a download link to the assets we will be using in this course.

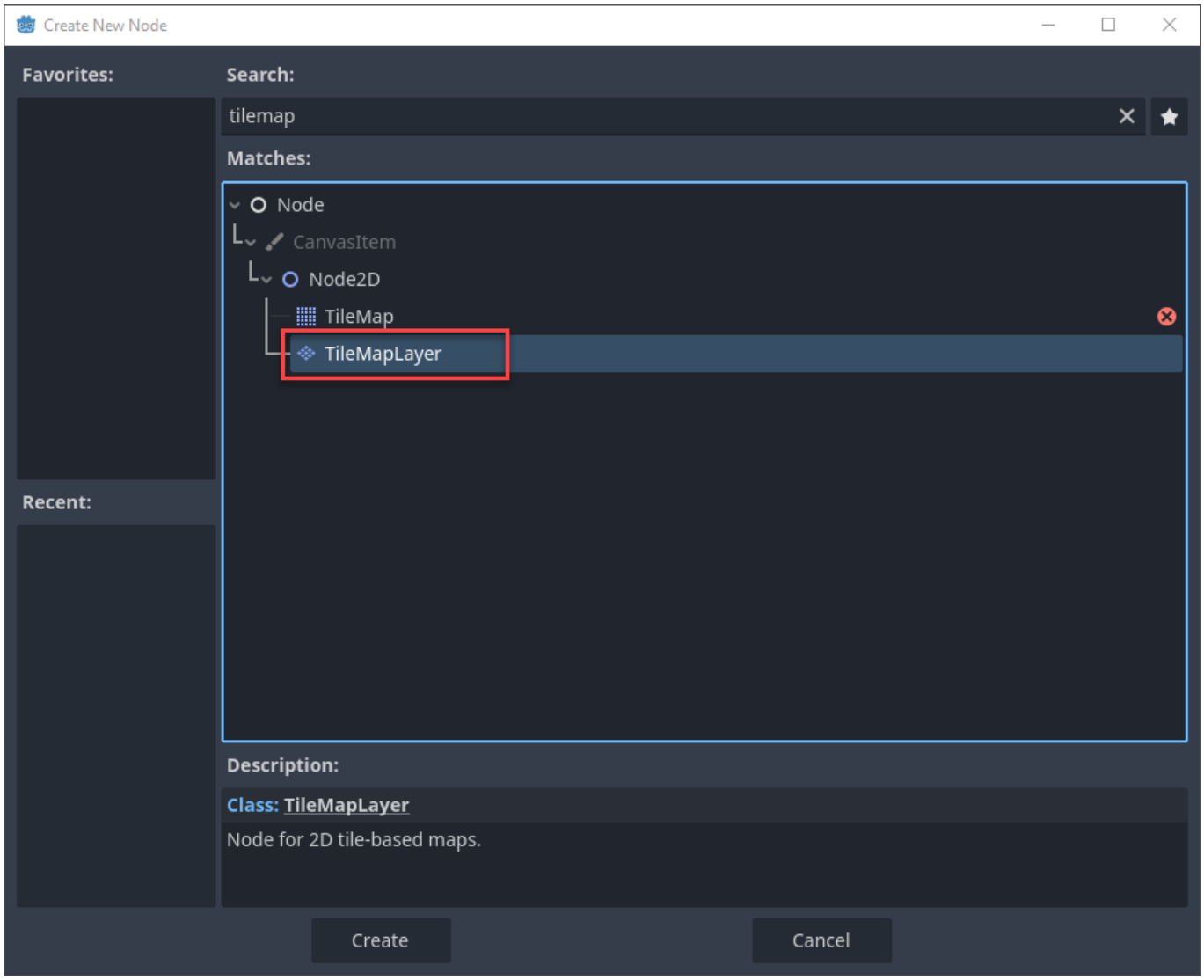
1. Download the assets.
2. Extract the folder.
3. Click and drag it into your project's File System.



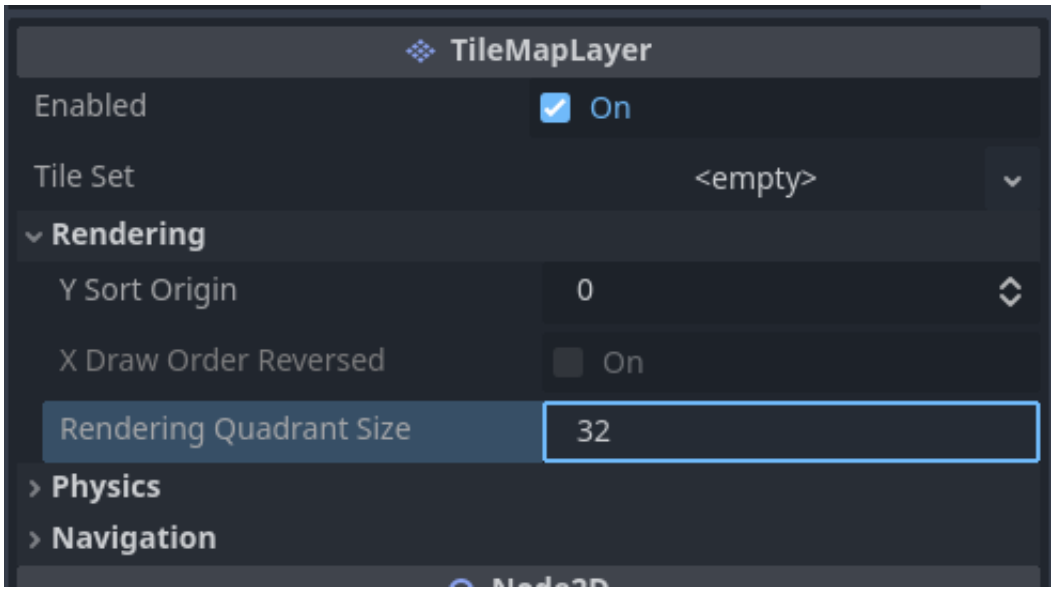
Inside of the **assets for bullet hell game > floor tiles** folder, we can find the images we'll use for our tilemap.

Creating the Tilemap

To create a tilemap, add a new node to our scene of type **TileMapLayer**.

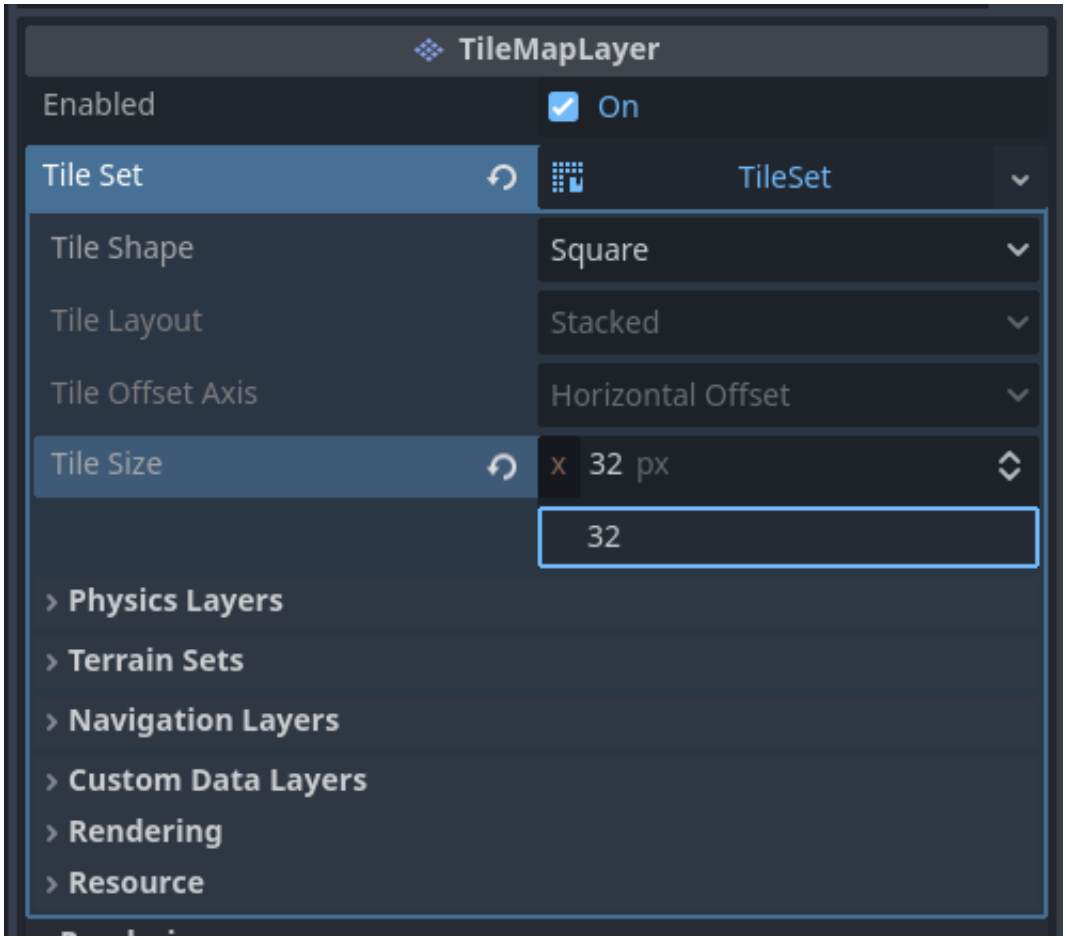


Once that node has been created, select it, and in the Inspector: set **Rendering Quadrant Size** to *32*, as our tiles are 32 x 32 pixels in size.

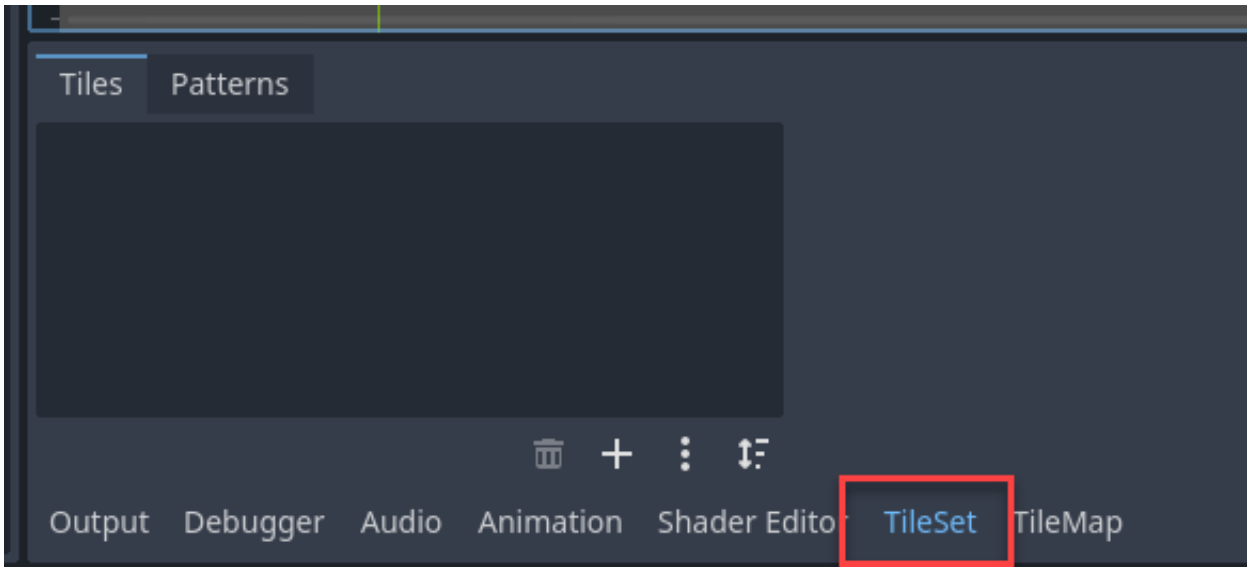


Next, click on the arrow next to **Tile Set** and create a new one. Opening it up, we can go down to **Tile Size** and set the X and Y to 32 pixels.

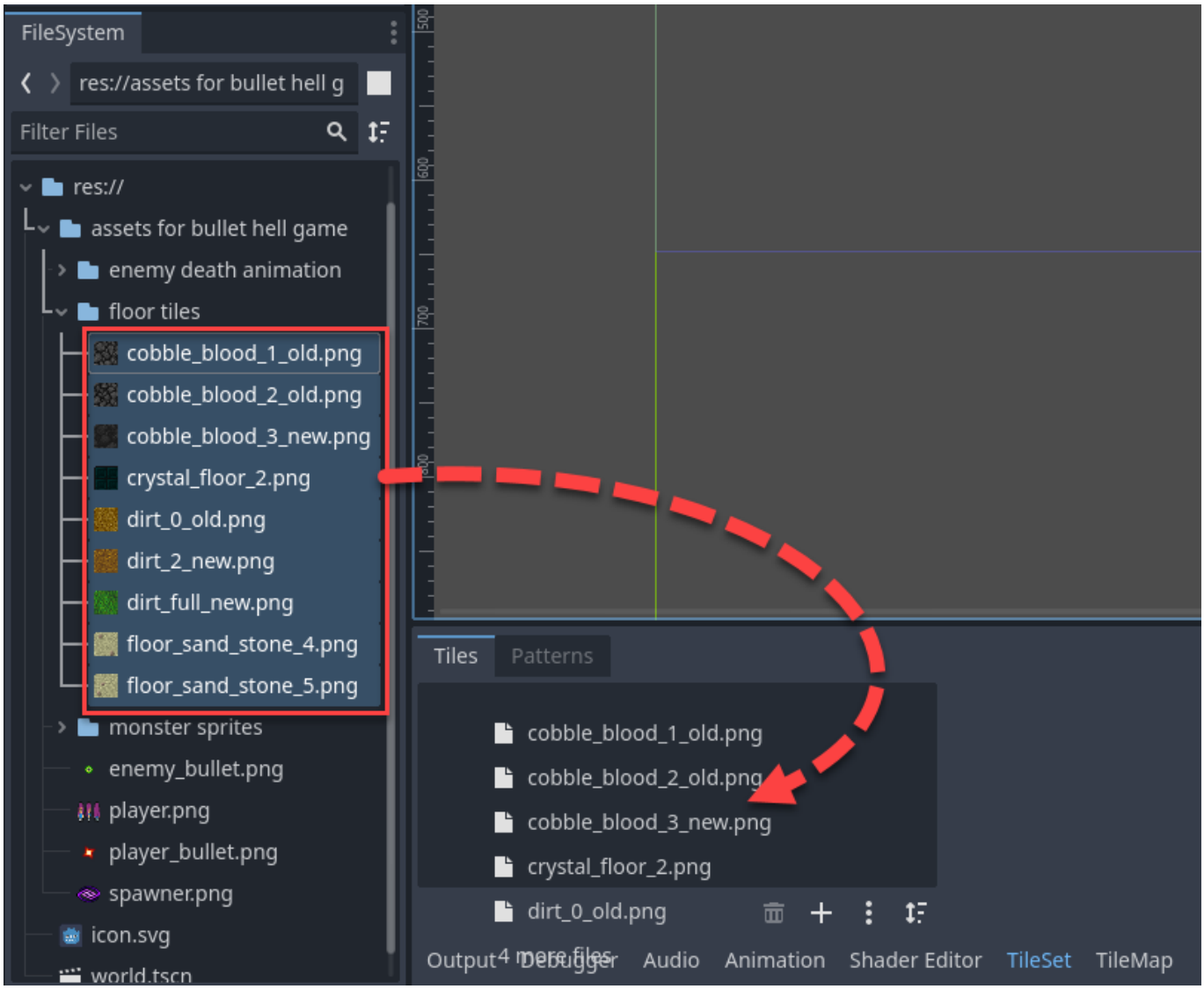
The tile set is where we define the images we want to use for tiles, setting up their individual properties, collisions, etc. The TileMapLayer node is where we then paint the tiles from the Tile Set collection.



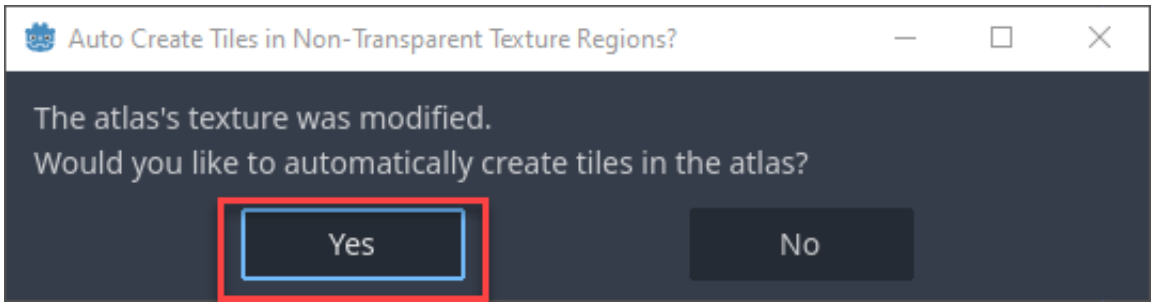
With the Tile Set selected, go down to the bottom of the editor and click on the **TileSet** button. This will open the Tile Set dock, where we can assign our tiles.



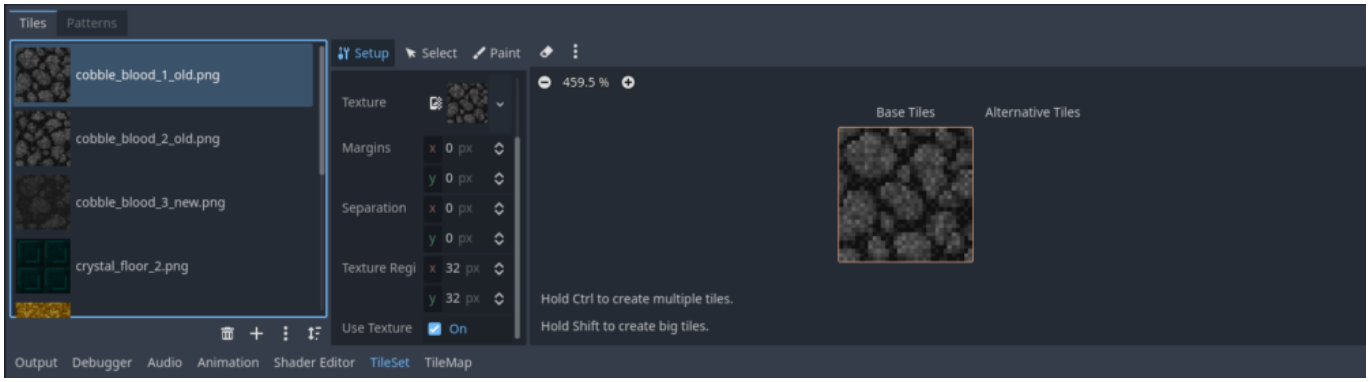
Now select all the images we want to turn into tiles and drag them into the window like this:



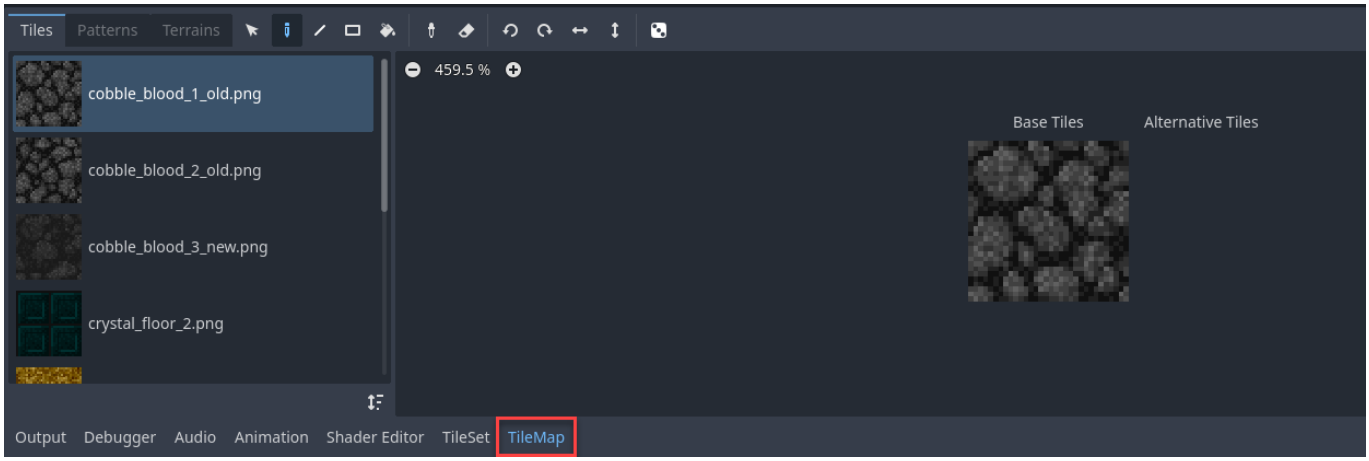
A pop up will appear. Click **Yes.**



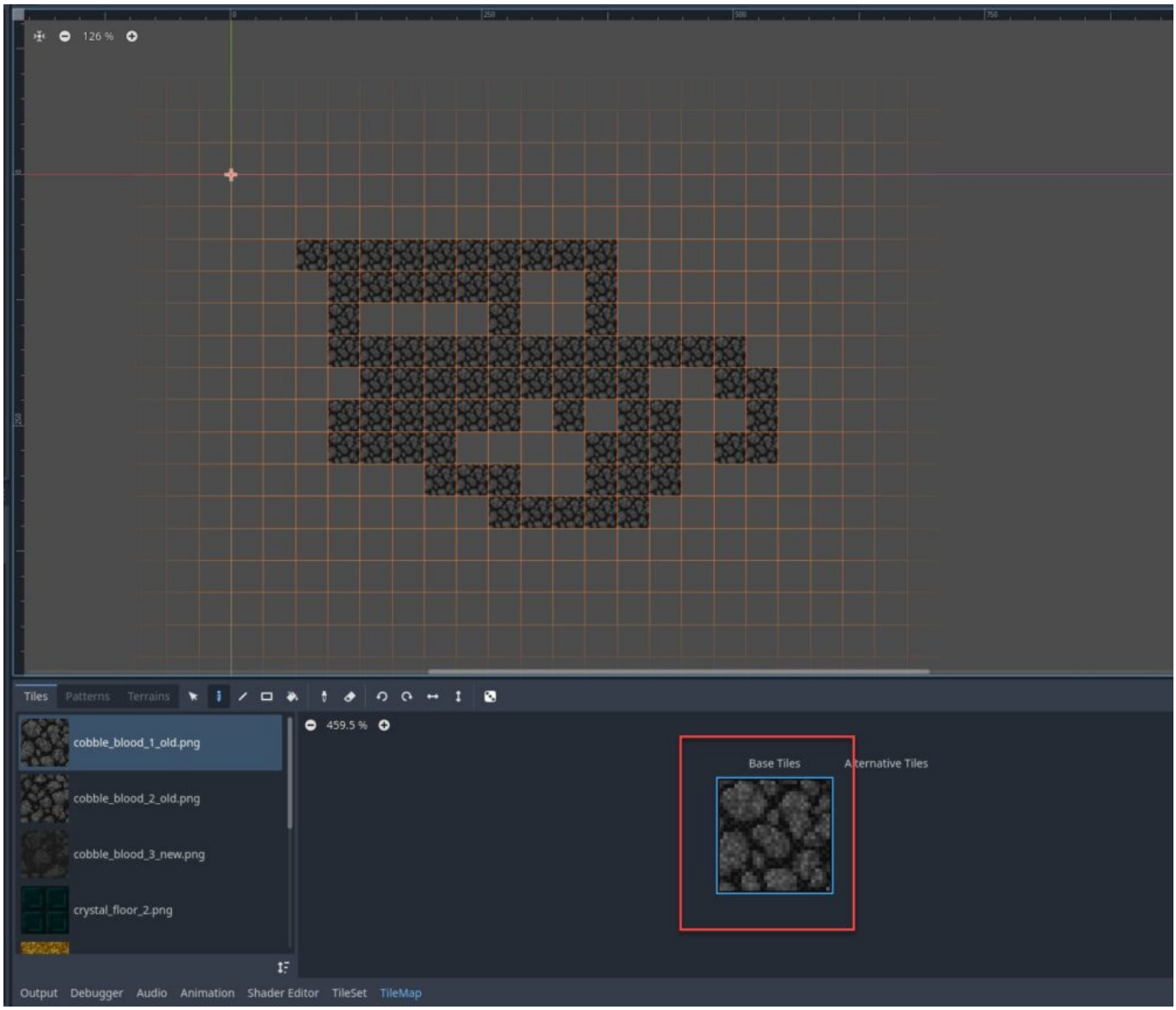
You should now see that we have all our tiles assigned into our Tile Set. You'll see that there are many settings to modify if you want to tweak the individual aspects of each tile, but we will not.



Now we can begin to paint these tiles onto our tile map. To do this, at the bottom of the editor, switch over to the **TileMap** dock. It will look similar, but this is where we can select tiles.

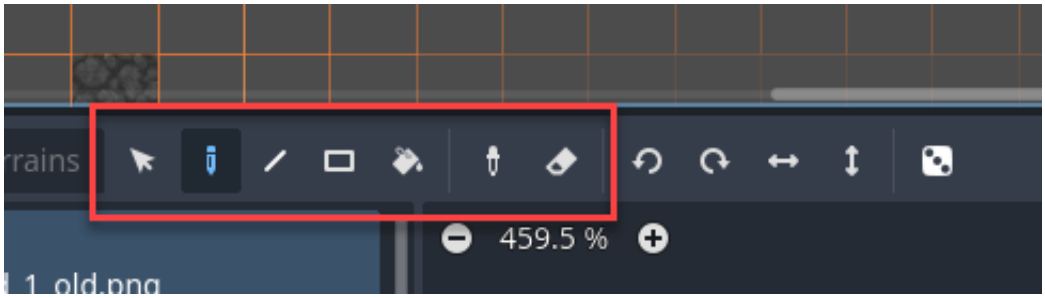


Once you have chosen a tile, select the **Base Tiles** image on the right, then begin to draw!

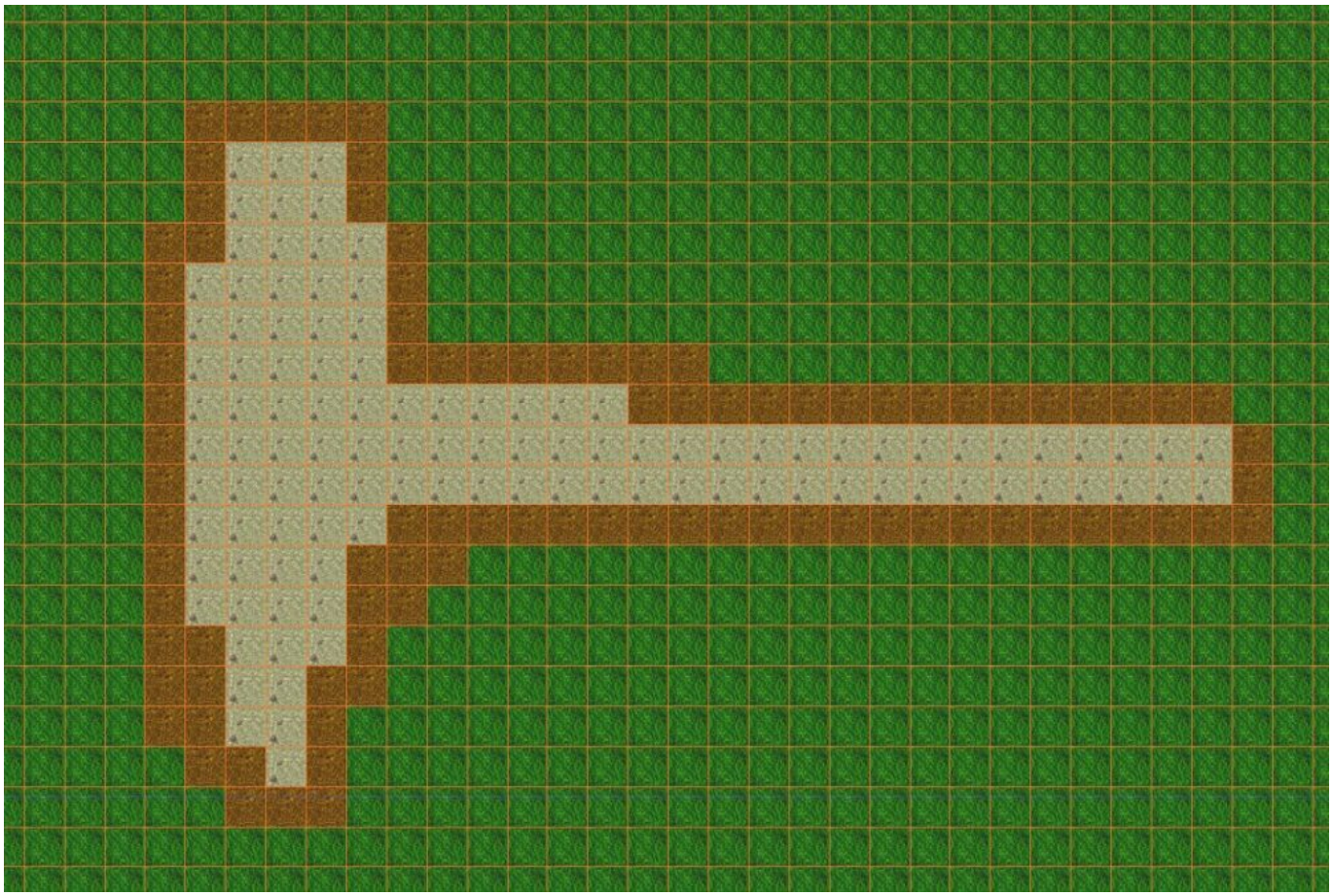


There are many tools that the tile map provides.

- **Selection** – Allows you to select a tile in the viewport and move it around.
- **Paint** – Click to draw tiles on the tile map.
- **Line** – Click and drag to create a straight line.
- **Rect** – Fill in large areas by drawing a rectangle.
- **Bucket** – Fills in empty voids.
- **Picker** – Selects the tile you click on the tile map.
- **Eraser** – Removes a tile.



From here, have a go at designing a map for the map.



Welcome to this tutorial on importing assets and using TextureRectangle and TileMap in Godot. In this tutorial, you will learn how to import new assets into your project, how to create and use a TextureRectangle, and how to create and use a TileMap. This tutorial assumes you have a basic understanding of Godot and have it installed on your computer.

Importing Assets

Importing assets into your Godot project is a straightforward process. All you need to do is to drag your asset files into the project window in Godot. Godot will automatically import the assets and make them available for use in your project. For better organization, you can create separate folders for different types of assets, such as scenes and scripts.

Using TextureRectangle

A TextureRectangle in Godot is a node that displays a texture. You can use it to display an image or a tile in your game. To create a TextureRectangle, follow these steps:

1. Add a new node to your scene and select “TextureRectangle” from the list of available nodes.
2. In the Inspector on the right side of the Godot window, find the “Texture” property and click on the “” button next to it.
3. Select the texture you want to display from your imported assets. The texture will be displayed in the TextureRectangle.

By default, the texture will be stretched to fit the TextureRectangle. If you want to display the texture as tiles instead, you can change the “Stretch Mode” property of the TextureRectangle to “Tile”. This will make the texture repeat itself within the TextureRectangle.

Using TileMap

A TileMap in Godot is a node that allows you to create a 2D grid of tiles. You can use it to create complex levels for your game using a set of tile images. To create a TileMap, follow these steps:

1. Add a new node to your scene and select “TileMap” from the list of available nodes.
2. In the Inspector, change the “Cell/Size” property to match the size of your tiles. If your tiles are 32×32 pixels, for example, set the “Cell/Size” property to “32”.
3. Next, create a new TileSet by clicking on the “” button next to the “Tile Set” property. A TileSet is a collection of tiles that you can use in your TileMap.
4. In the TileSet editor, add your tile images and set their properties as needed.
5. Finally, you can start drawing your TileMap in the scene editor. Select a tile from the TileSet and click on the grid in the scene editor to place the tile.

That’s it! You have now learned how to import assets, use TextureRectangle, and create a TileMap in Godot. In the next tutorial, we will cover player movement and interaction. See you there!

In this lesson, we will learn how to create a player with basic four-directional movement in the Godot game engine. We will create a new node, implement the required scripts, and animate the player. We will also cover how to get and use input in Godot.

Creating the Player Node

To begin with, we need to create a new node in our scene. This node will represent our player. In Godot, we can use the `CharacterBody2D` node for this purpose. This node used to be called `KinematicBody2D`, which is a more descriptive name because it indicates that this is a body that moves.

To add the `CharacterBody2D` node:

1. Go to the “Other Node” option in Godot.
2. Search for “CharacterBody2D”.
3. Select it and add it to the scene.

Once we have our node, we need to add a couple of things to it:

1. A collision shape, which will handle collisions with other objects in the game.
2. A placeholder for our actual player. This could be a sprite, a mesh, or any other visual representation of the player.

After adding these, we need to rename our `CharacterBody2D` node to “Player” so that we can easily identify it. We then save this into our scenes and create a new folder for the player.

Adding the Player Script

Next, we need to add a script to our player. This script will control the player’s movement. When adding the script, make sure to select the “Object (empty)” template. This is because the “CharacterBody2D (basic movement)” template is for platformers, which is not what we want for our game.

The first thing we’ll add to our script is a `speed` variable. This will control how fast our player can move. We’ll set it to 75 for now:

```
var speed: int = 75
```

Next, we’ll create a `physics_process()` function. This function is called every frame and is where we’ll put our movement code. For now, we’ll leave it empty:

```
func _physics_process(delta: float) -> void:
```

Getting and Using Input

Now, let’s talk about how to get input in Godot. There are several ways to do this, but the one we’ll use is the `Input.get_axis()` function. This function returns a value between -1 and 1, depending on the input. For example, if the player is pressing the right arrow key, `Input.get_axis(“UI_Left”, “UI_Right”)` will return 1. If they’re pressing the left arrow key, it will return -1. If they’re not pressing either, it will return 0.

We'll use this function to get both our horizontal and vertical input, and store them in a `Vector2`:

```
var inputDirection : Vector2 = Vector2(
  Input.get_axis("UI_Left", "UI_Right"),
  Input.get_axis("UI_Up", "UI_Down")
).normalized()
```

Note that we're calling `normalized()` on our vector. This function ensures that the length of the vector is 1, which is important for our movement code.

Implementing Four-Directional Movement

With our input vector, we can now implement our four-directional movement. We'll do this by creating a new `direction` variable and using a series of `if` statements to check which direction the player is moving in:

```
var direction : Vector2 = Vector2(0, 1)

if inputDirection.x > 0:
  direction = inputDirection
elif inputDirection.x < 0:
  direction = inputDirection
elif inputDirection.y > 0:
  direction = inputDirection
elif inputDirection.y < 0:
  direction = inputDirection
```

Finally, we'll use our `direction` variable to update our player's velocity, and call the `move_and_slide()` function to actually move the player:

```
velocity = direction * speed
move_and_slide()
```

And that's it! We now have a player that can move in four directions. In the next lesson, we will animate our player to make our game look more lively.

In this lesson, we will learn how to animate a player sprite in the Godot game engine. We will primarily use code for this, but before we dive into that, let's take a brief look at how to manually set up animation in a 2D scene in Godot.

Manual Animation Setup

Inside of our animation on the right side of the Godot editor, we'll see we have H frames, V frames, frames, and coordinates. We can either use frame or frame coordinates. In this lesson, we will use frame. However, there's one other problem, right? Because we have three frames, one for down, one for right, and one for the up. But what about the left? Well, we can actually do something clever here. We can set the frame to one, and then in our offset, we can actually see a flip H. If we set that, it will flip it to the left. This is how we would make our player look to the left.

Setting Up Collision Shape

Before we head into our player, let's actually go to our collision shape and we'll give our player a shape because it would probably make sense to have some sort of collision shape. So we'll give it a rectangle and adjust it to fit our sprite.

Animating with Code

Now let's head back into our code. We've actually already set up a good amount of stuff for the animation. All we need to do is we need to set the frames for each direction. We can do this by accessing our player node, which is the sprite, and setting the frame and flip H properties. Here is an example of how to do this:

```
# Access the player node and set the frame
get_node("player").frame = 1

# Access the player node again and set flip H to false
get_node("player").flip_h = false
```

We do this for each direction. The only time we actually want flip H to be true is when we're moving to the left. So in the left, you can just set this to true. Now for the frames, when moving to the right or left, we set the frame to 1. When moving down, we set the frame to 2, and when moving up, we set the frame to 0.

```
if inputDir.x > 0:
    get_node("Player").frame = 1
    get_node("Player").flip_h = false
elif inputDir.x < 0:
    get_node("Player").frame = 1
    get_node("Player").flip_h = true
elif inputDir.y < 0:
    get_node("Player").frame = 2
elif inputDir.y > 0:
    get_node("Player").frame = 0
```

And that's it! We have successfully animated our player sprite. In this lecture, we've just gone over

basic animation of a sprite in Godot. We've used some code to animate it, and we've also seen how to manually set up animation in a 2D scene. In the next lecture, we'll start learning about object pooling and how we can use it to optimize our game.

In this article, we're going to delve into the concept of object pooling, breaking down the term into its individual components and explaining how it functions in the context of game development. This is an important concept to understand, especially for those engaged in object-oriented programming.

Understanding Objects

Firstly, let's discuss what an object is. In the realm of object-oriented programming, which you may already be familiar with, an object is a fundamental component. However, in Godot, a popular open-source game engine, these objects are often referred to as nodes. So, for the purpose of this discussion, you can think of an object as a node that we can utilize in our game.

Understanding Pooling

Now, let's move on to the concept of pooling. In the simplest terms, a pool is a collection where we store objects. So, object pooling is essentially creating a pool filled with objects that we can use in our game.

Applying Object Pooling

Consider a scenario where you're developing a game with a player character who wields a gun. This gun can fire bullets. The idea behind object pooling is to create a predetermined number of bullets in our object pool as the game initiates. For instance, we might decide to create 10 bullets in our pool.

Each time the player fires the gun, we take a bullet from this pool and use it in the game. Once a bullet is used, whether it hits a target or goes off-screen, we simply return it to the pool, making it available for reuse.

Extending the Object Pool

Object pooling also allows for flexibility. If all the bullets in the pool are used up, we can simply add more to the pool. This process is often referred to as 'appending', a term commonly used in the context of arrays. Arrays are a useful tool in implementing object pooling as they allow us to add or 'append' objects, which then remain in the array for the duration of the game.

Object pooling is a powerful technique that can greatly optimize your game's performance by reusing objects instead of constantly creating and destroying them. In the next section, we will look at how to implement object pooling in Godot.

In this lesson, we will learn how to make a game character shoot bullets in Godot. We will create a bullet scene, learn how to shoot it using player input, and understand the concept of object pooling. Let's get started.

Creating a Bullet Scene

The first step is to create a bullet scene. We will use a CharacterBody2D node as the base node for our bullet. This node will use velocity to move the bullet. Let's rename the node to "bullet" and save it in the "scenes" folder inside the "player" folder since it's part of the player.

Next, we'll add a sprite for the bullet. We'll also add a collision shape to the bullet. We'll use a circle for the collision shape and shrink it a little bit. The size of the collision shape doesn't matter much at this point because we'll add an AreaBody2D later, which needs to be larger than the collision area.

Adding a Script to the Bullet

Now, we'll add a script to the bullet. Inside the script, in the process function, we'll use the move_and_slide method to move the bullet. We can also add a line of code to rotate the bullet as it moves for a more dynamic effect:

```
self.rotation += 1
```

This line of code will rotate our bullet as it flies.

Shooting the Bullet

Next, we'll modify the player's script to allow shooting bullets. We'll add a new action called "shoot" in the project settings. This action can be triggered by a keyboard button or a mouse click. For this example, we'll use the spacebar.

Inside the player's script, we'll add an if statement to check if the "shoot" action has been pressed. If it has, we'll create a bullet. To do this, we'll first preload the bullet scene and then instantiate it:

```
onready var bullet = preload("path/to/bullet/scene")
var bulletTemp = bullet.instance()
```

Once the bullet is created, we'll add it as a child to the player. However, we need to set the velocity of the bullet, or it won't move. We'll set the bullet's velocity to the direction the player is facing multiplied by a speed factor:

```
bulletTemp.velocity = direction * 100
```

Fixing Bullet Collisions

At this point, if we run the game and shoot, we'll notice that the bullet collides with the player. To fix this, we'll go to the bullet's kinematic body and uncheck the collision layers and masks. Now, the

bullet won't collide with anything.

Improving the Bullet Shooting

Currently, when we shoot, the bullets move with the player. To fix this, we'll add a Node to the player and add the bullets as children to this node:

```
onready var bullets = get_node("Bullets")
bullets.add_child(bulletTemp)
```

However, this will cause the bullets to spawn at the wrong position. To fix this, we'll set the bullet's global position to the player's global position:

```
bulletTemp.global_position = self.global_position
```

Now, the bullets won't move with the player. To further improve the shooting, we can make the bullets spawn a little further away from the player. We'll add a Marker2D node called "spawn point" and set the bullet's global position to this spawn point:

```
bulletTemp.global_position = get_node("SpawnPoint").global_position
```

This concludes our lesson on making a game character shoot bullets in Godot. We've learned how to create a bullet scene, shoot bullets using player input, and handle bullet collisions. In the next lesson, we'll learn about object pooling and how to apply it in Godot.

Shooting Based on Mouse Direction

If you want to change it so we shoot based on the position of our mouse cursor, then we can modify the code for when we press the shoot button.

```
if Input.is_action_just_pressed("shoot"):
    direction = get_local_mouse_position().normalized()
    $SpawnPoint.position = direction * 5
    var bulletTemp: Bullet = $Bullets.get_bullet()
    bulletTemp.velocity = direction * 100
    bulletTemp.global_position = $SpawnPoint.global_position
```


Welcome back! In this lesson, we will be discussing how to apply and write a bullet pool script. This will allow us to use bullet pooling or object pooling for bullet shooting in our game. In the last lecture, we looked at instantiating bullets, but now we will create a bullet pool.

Creating the Bullet Pool Script

First, navigate to the player scripts in the bottom left of your screen. Right-click and select ‘Create New Script’. Name this script ‘Bullet Pool’ and open it. You can erase the pre-existing code so that we can write our own.

The first thing we want to do is load our bullet scene. We can do this by preloading our bullet scene like so:

```
bullet_scene : PackedScene = preload("res://Scenes/Bullets/bullet.tscn")
```

Next, we need to create two variables. The first one is ‘pool_size’, which will be an integer representing the number of bullets we have in our pool. The second one is ‘bullet_pool’, which is an array that will hold our bullets. We will initially set it to an empty array:

```
var pool_size : int = 20
var bullet_pool : Array = []
```

Filling the Bullet Pool

In the ready function, we want to fill our bullet pool, which is an array. We can do this using a for loop that will iterate over the range of our pool size and create a new bullet for each iteration:

```
func _ready() -> void:
  for i in range(pool_size):
    var bullet : Node = bullet_scene.instantiate()
    bullet.hide()
    bullet_pool.append(bullet)
    add_child(bullet)
```

Here, we instantiate a new bullet, hide it, append it to our bullet pool, and then add it as a child to our bullet pool node. We hide the bullet initially because we don’t want it to be visible until it’s in use.

Getting and Resetting Bullets

We also need two other functions: one to get a bullet and one to reset a bullet. The ‘getBullet’ function will return a bullet from our pool that’s not currently visible, and if there are no hidden bullets, it will create a new one. The ‘resetBullet’ function will hide the bullet and move it to a position that’s outside of our map:

```
func get_bullet() -> Node:
```



```
  for bullet in bullet_pool:
    if not bullet.visible:
      return bullet

  var new_bullet : Node = bullet_scene.instantiate()
  new_bullet.hide()
  bullet_pool.append(new_bullet)
  add_child(new_bullet)
  return new_bullet

func reset_bullet(bullet: Node) -> void:
  bullet.position = Vector2(-1000, -1000)
  bullet.hide()
```

Using the Bullet Pool in the Player Script

Now, we can use our bullet pool in our player script. Instead of preloading our bullet, we can get our bullet pool node. Then, in our shoot function, we can get a bullet from our bullet pool instead of creating a new one:

```
bulletPool : Node = get_node("BulletPool")

func shoot():
  var bullet_temp : Node = bulletPool.get_bullet()
  bullet_temp.velocity = direction * 140
  bullet_temp.global_position = get_node("Marker2D/SpawnPoint").global_position
  bullet_temp.show()
```

This will get a bullet from our pool, set its velocity and position, and then show it. By using object pooling, we can reuse bullets instead of creating a new one each time, making our game more efficient.

That’s the end of our object pooling lesson. We’ve gone over how object pooling can be coded in a script, how we can use it inside of our player, and we’ve created our bullet pooling with our node. In the next lecture, we’ll take a look at some basic AI stuff. See you in the next lecture!

Welcome to this introductory lecture on Artificial Intelligence (AI) and its general movement. AI might sound complex, but this article aims to simplify its concept, specifically in the context of game development. Here, we will focus on the basic idea of AI, player chasing and how to implement it.

Understanding the Basics

Imagine you, as a player in a game, and you're being chased by an AI character. Here, point A is where you're currently standing, and point B is the position of the AI character. The goal is for the AI to move from point B to point A. This is achieved by calculating the distance between these two points and then determining the direction. The AI character then moves in that direction.

Different Behaviors of AI

AI characters generally exhibit several behaviors:

Idle State
Chase or Run State
Attack State

In this article, we're combining the chase and attack states. The AI character or 'monster' will be shooting at the player at intervals while also chasing them. The idle state is not considered here.

After understanding this basic concept, you might want to challenge yourself by figuring out how to implement transitions from the idle state to the chase and attack states. However, this is beyond the scope of this introductory course.

Conclusion

In conclusion, this introductory lecture provided a simple understanding of AI, particularly in the context of game development. We discussed how to calculate the AI's movement from one point to another and briefly touched upon the different behaviors an AI character can exhibit. In the next lecture, we will delve into creating an AI scene and programming the AI character to chase the player.

In this tutorial, we will be setting up an Artificial Intelligence (AI) scene in Godot. We will create a simple sprite, and then we will program it to follow the player. The steps involved include creating a new scene, adding a character body, and writing a script to control the character's movement. Let's get started!

Creating a New Scene

First, we need to create a new scene in Godot. To do this, click on the 'Scene' menu and choose 'New Scene'. Then, select '2D Scene' as the root node. We need a character body for our AI, so let's create a 'CharacterBody2D' node. This node is a part of Godot's physics engine and it provides functionalities for 2D physics simulations.

Adding a Sprite and Collision Shape

Next, we need to add a sprite and a collision shape to our character body. The sprite is the visible representation of our character, and the collision shape defines the area in which collisions will be detected. In this tutorial, we will use a monster sprite and a rectangle collision shape.

After adding the sprite and collision shape, we can lock the sprite in place to prevent accidental movement when reshaping the collision shape. To do this, select the sprite and click on the lock button in the toolbar.

Writing the AI Script

Now, let's write a script to control the movement of our AI character. We will make the character follow the player. To do this, we need to calculate the direction from the character to the player, and then move the character in that direction.

```
# AI script
extends CharacterBody2D

var speed = 20  # Movement speed
var is_alive = true  # Whether the character is alive
var direction  # Direction to the player
var player  # Reference to the player node

func _ready():
    player = get_node("../../Player")  # Get reference to the player node

func _physics_process(delta):
    if is_alive:
        direction = (player.global_position - self.global_position).normalized()  # C
alculate direction to the player
        velocity = speed * direction  # Move in the direction of the player
        move_and_slide(velocity)
```

This script makes the character follow the player as long as the character is alive. The '_physics_process' function is called every physics frame, and it moves the character towards the player.

Flipping the Sprite

Finally, we can make our character look more dynamic by flipping its sprite based on its movement direction. To do this, we check the x-component of the direction vector. If it's less than zero, the player is on the left side of the character, so we flip the sprite horizontally.

```
# Flipping the sprite
if direction.x < 0:
    sprite.flip_h = true
else:
    sprite.flip_h = false
```

And that's it! Now we have a simple AI character that follows the player. In the next tutorial, we will add more functionalities to the character, such as the ability to shoot bullets and react to collisions with the player.

In this lesson, we'll be enhancing our bullet in Godot by making it interact with our mob (or enemy). We'll create an area in which the bullet can operate and check for collisions with the mob. If a collision is detected, we'll destroy the mob and reset the bullet. Let's dive in!

Adding an Area2D to the Bullet

First, we need to add an Area2D to our bullet scene. This will define the area in which the bullet operates and checks for collisions with other objects.

We also need to add a CollisionShape2D to the Area2D. This shape defines the area of the bullet that can collide with other objects. We'll make this a circle, slightly larger than the bullet itself:

Now, we need to connect a signal to the Area2D so that we know when a body (like our mob) enters it. We'll use the body_entered signal.

Checking for Collisions

When a body enters our Area2D, we need to check which body it is. It could be the player, a mob, or something else. To do this, we'll use the function connected to the body_entered signal:

```
func _on_Area2D_body_entered(body):
    if body.get("mob"):
        pass
```

This function checks if the body has a property named "mob". If it does, it means that the body is a mob and we can interact with it.

Adding the Mob Property

But wait, our mobs don't actually have a "mob" property yet. Let's add that to our mob scene:

```
var mob = true
```

This property will be checked by the bullet when a collision is detected. If the property exists and is true, the bullet will interact with the mob.

Interacting with the Mob

Now that we can detect collisions with mobs, we need to decide what to do when a collision occurs. For now, we'll simply remove the mob from the scene:

```
func _on_Area2D_body_entered(body):
    if body.get("mob"):
        body.queue_free()
```

Resetting the Bullet

After the bullet has hit a mob, we need to reset it so it can be used again. We have already created a function for this in our bullet pool script, so we can simply call this function:

```
func _on_Area2D_body_entered(body):
    if body.get("mob"):
        body.queue_free()
        get_parent().reset_bullet(self)
```

And that’s it! We have now created a bullet that can interact with mobs. When a bullet hits a mob, the mob is removed from the scene, and the bullet is reset and can be used again.

In the next lesson, we’ll look at creating a pool of mobs that can be reused, similar to our bullet pool. See you there!

Welcome back everyone. In this article, we will be diving into a particularly interesting topic – Object Pooling but specifically for monsters in the game. Object pooling is a software creational design pattern that uses a set of initialized objects kept ready to use, rather than allocating and deallocating them on the fly. If you are already familiar with Bullet pooling, you will find the process quite similar. Let's get started.

Setting up the main world:

In our main world, start by adding a node 2D that acts as our portal. Now proceed to add in a spawner as a child. Ensure that you reset the position of the spawner.

Adding a timer inside the portal:

The next step is to add a timer inside our portal. This is significant for our topic as we will be using object pooling to control our game's monsters.

Now, let's create the Object Pooling:

- Inside our portal, create a new script.
- Place this into our scripts, under AI, and give it a name such as 'mob pooling'.
- The structure of this script is similar to that of bullet pool, so you can copy everything except for the extend from the bullet pool.
- Remove 'extend' since this isn't a 2D instance.
- For the preload, replace the bullet with the snake (or your chosen monster).

```
func get_mob():
for mob in mob_pool:
    if mob.is_alive == false:
        return mob
var mob = preload("res://path_to_your_snake.tscn").instance()
mob_pool.append(mob)
self.add_child(mob)
return mob

func reset_mob(mob):
mob.is_alive = false
mob.get_node("collision_shape_2D").disabled = false
```

A good trick here is to use control R and replace all references to 'bullet' with 'mob'. This will ensure all the instances reflect the changes. In the reset_mob function, we also tell the system that the mob is no longer alive by setting 'is_alive = false'.

Disable the collision shape:

We need to disable the collision shape of our snake as well. This can be done with the mob.get_node and we set its disabled property to false.

Accessing the Timer:

Finally, we need to get access to our timer. We can do this using get_node and calling the 'Timer'. Remember, we won't be using the timer in this lecture, but it's crucial that we set this up now to further facilitate the addition of mobs to our game.

```
    var timer = get_node("Timer")
```

Now, how do we add mobs to our game? Go to your timer and try connecting the timer as a timeout to your pool, hinting at the next spot for a mob spawn. In the upcoming articles, we will delve deeper into a challenge on how to spawn them in intervals using the bullet pooling technique.

This wraps up the article. We hope it was helpful in gaining a better understanding of mob pooling. Practice the concepts, play around with them, and implement them into your game, or any game you are currently developing. See you in the next lecture!

In this lesson, we will walk through the solution to our challenge of creating spawn points for mobs (monsters) in our game. This involves using timers and signals in Godot to control the spawning of mobs at regular intervals.

Connecting a Signal to our Script

Firstly, we need to connect a signal to our script. This is done by navigating to our timer and connecting it to the portal in the timeout node. Here is how you do it:

Creating the Mob

Next, we need to create our mob. This can be done by declaring a variable ‘mobTemp’ and assigning it to ‘get mob’. Since ‘get mob’ is a node, we need to specify it as such:

```
var mobTemp = get mob # This is a node
```

Once the mob has been created, we need to add it to the game scene. We can do this by setting its global position to the global position of the self (the current object). After adding the mob to the scene, we also need to make it visible. This is because when we fetch the mob, we initially hide it.

```
mobTemp.global_position = self.global_position # Set the mob's global position
mobTemp.show() # Make the mob visible
```

Adjusting the Path of the Player

Due to the addition of mobs in the portal, the path of the player has changed. We need to adjust this in our code by going one more parent up.

Now, if you run the game, you will see that the monsters spawn every one second. However, they all appear at the same position. To make the game more interesting, we can randomize the spawn positions of the mobs.

Randomizing Mob Spawn Positions

To randomize the spawn positions of the mobs, we can create two variables, ‘randomX’ and ‘randomY’, which will hold random values within a specified range. We can then add these random values to the global position of the self:

```
var randomX = randi_range(-50, 50) # Random X position
var randomY = randi_range(-50, 50) # Random Y position
self.global_position += Vector2(randomX, randomY) # Add the random values to the glob
al position
```

This will spawn the mobs at random positions around the portal.

Adding a Camera Scene

Finally, to enhance the player's experience, we can add a camera scene to our player. We can zoom in a little bit and add some smoothness to the camera movement. We can also limit the camera to make sure it doesn't go above or away from our screen:

With these changes, our game now has mobs spawning at random positions at regular intervals. However, there is still room for improvement. For example, we could add health bars to the mobs and implement a system where the bullets reduce the mob's HP instead of killing them instantly. This will be the focus of our next lesson.

In this lesson, we will be discussing how to add a health bar to a player or a snake in Godot. We will also discuss how to fix a common mistake with global and local positions. Let's get started.

Fixing a Mistake in Reset Mob

First, we need to fix a small mistake in the reset mob function. The error lies in the fact that we used a local position instead of a global position. In Godot, global position refers to the position of the node in the game world, while local position refers to the node's position relative to its parent. To correct this, we need to ensure that we are using the global position.

Adding a Health Bar

Next, we will add a health bar to our snake. This involves two main steps:

1. Adding a progress bar to represent the health bar of our snake.
2. Creating a health variable and linking it to the progress bar.

Adding a Progress Bar

To add a progress bar, follow these steps:

1. Go to the snake and add a progress bar.
2. Choose to not show the percentage.
3. Go to the theme overrides styles.
4. Go to fill and select new style flat box for both of these.
5. Open both of them.
6. Make the bottom one green and the top one red.

Creating a Health Variable

Now that we have a progress bar, we need to create a health variable. We will set the max value of our progress bar to be equal to our health. We can do this by adding a ready function. However, we need to preload the progress bar to be able to use it. Once we've done that, we can set the max value of the progress bar to be equal to our health. The following code snippet illustrates this:

```
var health : int = 5
@onready var bar : ProgressBar = get_node("ProgressBar")
```

Now, the progress bar is ready. But we need to update it when the snake takes damage. We can do this by setting the value of the bar to be equal to health when the snake is alive.

```
func _ready() -> void:
  bar.max_value = health
```

Reducing Health

To reduce the health of the snake, we can add a condition in our reset mob function. If the health is greater than one, we reduce the health by one. Otherwise, the health is less than one, which means the snake is dead. When the snake dies, we can set the variable isAlive to false. We can also play an animation or add a score, which we will discuss in later lectures.


```
func reset_Mob(body: Node) -> void:  # Adjust the type if 'body' is a specific type
  if health > 1:
    health -= 1
  else:
    get_parent().reset_mob(body)
```

Testing the Health Bar

With the health bar and health variable set up, we can now test our game. The health bar should decrease as the snake takes damage.

In this lesson, we've learned how to add a working health bar to a snake in Godot. We've also learned how to fix a common mistake with global and local positions. In the next lesson, we will add a death animation to our snake.

In this lesson, we will learn how to animate the death of a player in a game using Godot. We'll be adding an animated sprite to our game, configuring the sprite frames, and scripting the death animation. Let's get started.

Adding an Animated Sprite

The first step is to add an animated sprite to our game. In Godot, you can do this by searching for "animated sprite" and adding it to your scene. Once you've added the sprite, you can configure the sprite frames in the top right of the Godot interface. Click on the "New SpriteFrames" button to create a new set of frames for your sprite.

Name your new sprite frame "death". This will be the animation that plays when the player dies. You can add individual sprites to the frame by clicking on the grid icon and selecting the sprites you want to include in the animation. If your sprites are all in one file, this grid will split them up into different sections. However, if your sprites are already split up, you can simply select all of them and add them to the frame.

Animating the Death of the Player

Now that we have our death animation set up, we can script it to play when the player dies. In the script for your player, you'll need to load the animation and add a function to play the death animation when the player's health reaches zero.

```
onReady var anim = getNode("Anim")

func _physics_process(delta: float) -> void:
  if health <= 0:
    anim.play("Death")
```

This code will play the death animation when the player's health reaches zero. However, there are a few more things we need to handle. We need to hide the player sprite and health bar when the player dies, and show the death animation. We can do this by adding the following lines:

```
if health <= 0:
  sprite.hide()
  bar.hide()
  anim.show()
  anim.play("Death")
```

Here, we hide the player sprite and health bar, show the death animation, and then play it. However, there's one more thing we need to handle – we need to make sure that the game waits until the death animation has finished before resetting the player. We can do this using the await keyword and the animation_finished signal:

```
if health <= 0:
  sprite.hide()
  bar.hide()
  anim.show()
  anim.play("Death")
```



```
await Anim.animation_finished
```

The await keyword will pause the execution of the code until the animation_finished signal is emitted, ensuring that the death animation plays fully before the game continues.

Looping and Disabling Looping

By default, Godot will loop an animation indefinitely. This is not what we want for our death animation – we want it to play once and then stop. To disable looping, go to the animation settings and turn off the loop option. This is a common issue in Godot, and it's something to keep in mind if you ever see animations looping when they shouldn't be.

Conclusion

And there you have it! You've just animated the death of a player in Godot. This is a basic introduction to using animated sprites and animations in Godot. In the next lesson, we'll look at how to detect the player with an area and hurt the player.

In this lesson, we are going to learn how to add collision detection between an enemy character (referred to as a mob or snake) and the player in a game using the Godot game engine. This process is relatively straightforward and involves the use of an Area2D node and a CollisionShape2D node.

Adding an Area2D and CollisionShape2D

The first step is to add an Area2D node to your enemy character. An Area2D node in Godot is used for detection and influence. This means it can detect when other objects enter, exit, or overlap with it, and it can also exert influence on those objects. In this case, we are using it to detect when the player's character comes into contact with the enemy character.

Once you've added the Area2D node, you need to attach a CollisionShape2D node to it. The CollisionShape2D node defines the shape of the area for the Area2D node. You can choose any shape you want for the CollisionShape2D, but in this case, we are using a circle. You can adjust the size of the circle to fit around your enemy character.

Setting Up Collision Detection

Next, we need to set up collision detection. This is done by connecting the "body_entered" signal from the Area2D node to a function in our script. The "body_entered" signal is emitted when a body enters the area, which in this case is our player's character.

In the connected function, we can check if the body that entered the area is the player. This can be done in several ways, but in this lesson, we are checking if the name of the body contains the string "player". It's important to note that this check is case sensitive, so make sure the case of the string matches the name of your player node.

```
func _on_area2D_body_entered(body: Node) -> void:
  if "Player" in body.name:
    pass  # Replace with your code
```

Reducing Player Health

If the player collides with the enemy character, we want to reduce the player's health. However, the player node itself doesn't have a health property, so we need to add it. This can be done in two ways: locally by adding the health property to the player's script or globally by creating a global script that holds the player's health.

In this lesson, we are creating a global script called "game.gd" and adding a "playerHP" property to it:

```
var playerHP: int = 10
```

Back in the enemy character's script, we can now reference this global script and reduce the player's health by 1 when a collision occurs:

```
if "Player" in body.name:
  Game.playerHP -= 1
```

However, this won't work until we set the "game.gd" script as a global script. This can be done by going to Project Settings, then to the Autoload tab, and adding the "game.gd" script there.

With this setup, the player's health will be reduced by 1 every time they collide with an enemy character. You can print out the player's health in the _physics_process() function to see it in action:

```
func _physics_process(delta: float) -> void:
  print(Game.playerHP)
```

In the next lesson, we will learn how to enable the enemy character to shoot bullets at the player. This will involve creating a bullet pool for the enemy character, which we will cover in detail.

That's it for this lesson! You've learned how to detect collisions between characters in Godot and how to reduce a character's health upon collision. You've also learned how to create a global script and use it to store a character's health. This is a fundamental concept in game development and will be useful in many different types of games.

In this lesson, we will learn how to enable an AI object to shoot bullets at a player object using the Godot game engine. We will be creating the bullet first, duplicating the bullet script, and modifying it to suit our AI object. We will then add a timer to control the shooting frequency of our AI.

Creating the Bullet

Let's start by creating the bullet for our AI object. Instead of creating it from scratch, we can duplicate the bullet used by our player object and modify it as needed. Here's how we can do this:

1. Go to the player object and find the bullet node.
2. Right-click on the bullet node and select 'Duplicate'.
3. Change the name of the duplicated bullet to 'AI bullet'.
4. Move the AI bullet into the AI section in the scene tree.

Duplicating the Bullet Script

Next, we need to duplicate the bullet script and modify it for our AI object. Here's how:

1. Go to the bullet script attached to the player bullet.
2. Right-click on the script and select 'Duplicate'.
3. Change the name of the duplicated script to 'AI bullet'.
4. Move the AI bullet script into the AI section in the file system.
5. Go back to the AI bullet node and replace the current script with the AI bullet script.

The AI bullet script will be similar to the player bullet script, but we will change the body enter check. Instead of checking for a mob, we will now check for the player.

Adding the Bullet Pool

To manage the bullets shot by our AI object, we need to create a bullet pool. We can do this by duplicating the bullet pool used by our player object and modifying it as needed. Here's how:

1. Go to the player script and find the bullet pool node.
2. Right-click on the bullet pool node and select 'Duplicate'.
3. Change the name of the duplicated bullet pool to 'AI bullet pool'.
4. Move the AI bullet pool into the AI section in the scene tree.
5. Go to the AI bullet pool script and replace the bullet scene with the AI bullet scene.

Adding the Shooting Mechanism

Now that we have our AI bullet and bullet pool ready, we can add the shooting mechanism to our AI object. We will do this by adding a timer to our AI object and connecting it to a function that will trigger the shooting. Here's how:

1. Add a Timer node to our AI object and name it 'ShootBullet'.
2. Enable the 'Auto Start' property of the timer.
3. Connect the 'timeout' signal of the timer to the AI object.
4. In the connected function, call a function named 'shootBullet'.

We will define the 'shootBullet' function in our AI script. This function will check if the AI object is visible, and if it is, it will call the 'get_bullet' function from the AI bullet pool. The direction and position of the bullet will be determined based on the position of the player object.

```
func shoot_bullet() -> void:
  if self.visible:
```



```
    var bulletTemp : Node = bulletPool.get_bullet()  # Adjust the type if 'bulletTemp
' is a specific type
    var direction : Vector2 = (player.global_position - get_node("Marker2D/SpawnPoint
").global_position).normalized()
    bulletTemp.velocity = direction * 140
    bulletTemp.global_position = get_node("Marker2D/SpawnPoint").global_position
    bulletTemp.show()
func _on_shoot_bullet_timeout() -> void:
  shoot_bullet()
```

Testing the AI Shooting Mechanism

Once we have set up everything, we can test our AI shooting mechanism. When we run our game, our AI object should start shooting bullets at our player object. If we notice any issues, such as the bullets not hitting the player or not spawning correctly, we can debug and fix them.

And that’s it! We have successfully added a shooting mechanism to our AI object. In the next lesson, we will look at how to add a user interface to our game, including a health bar and a global score display.

In this lesson, we will learn how to add a graphical user interface (GUI) to our game in the Godot game engine. Specifically, we will be adding a score and a health bar (HP) for our player. We will also learn how to use a Canvas Layer in Godot to display these elements on our game screen.

Setting up the GUI

The first step in creating a GUI is to add a Canvas Layer to our game. The Canvas Layer is an important component in the Godot engine as it allows us to draw elements (such as a score or health bar) on top of our game screen. This is done by painting these elements onto the Canvas Layer, which is then drawn onto our game screen.

To illustrate this, let's create a progress bar and a score label. We will add these elements to our Canvas Layer and position them at the top right corner of our screen.

Adding a Progress Bar

First, we will add a progress bar to our Canvas Layer. This progress bar will serve as our player's health bar. We will set its max value to the player's maximum HP, and its current value to the player's current HP. We will do this in a script attached to the progress bar.

```
extends ProgressBar

var game : Node = get_node("/root/Game")

func _ready() -> void:
  max_value = game.playerMaxHP

func _process(delta: float) -> void:
  value = game.playerHP
```

Adding a Score Label

Next, we will add a label to our Canvas Layer. This label will serve as our score display. We will set its text to the current score of the game. We will do this in a script attached to the label.

```
extends Label

var game : Node = get_node("/root/Game")

func _process(delta: float) -> void:
  text = "Score: " + str(game.score)
```

Updating the Score

Whenever a monster dies, we want to increase the score by one. We can do this in the script attached to our monster. Whenever the monster dies, we will increment the game's score by one.

```
func die() -> void:
  game.score += 1
  ...
```


Conclusion

In this lesson, we learned how to add a GUI to our game in Godot. We added a score and a health bar for our player, and we learned how to use a Canvas Layer to display these elements on our game screen. We also learned how to update the score whenever a monster dies. In the next lesson, we will learn how to add a game over screen to our game.

In this lesson, we will learn how to create a game over screen in the Godot game engine. We will also look at how to transition to this screen when the player's health reaches zero and how to restart the game upon user request. Let's get started.

Creating a Game Over Scene

First, we will create a new scene for our game over screen. In this scene, we will add a Node2D and rename it to "game over". We will also add two labels and a button. The labels will be used to display the game over message and the final score, while the button will be used to retry or restart the game.

Let's adjust the properties of the labels and the button. For the game over label, we will center its text and increase its font size for better visibility. For the score label, we will extend it to cover the entire screen width and also center its text. The button will be placed in the middle of the screen and labeled as "Retry".

Adding a Script to the Game Over Scene

Next, we will add a script to our game over scene. In this script, we will get access to the score label and set its text to display the final score when the scene is ready. We will also connect our retry button to the script so that it can respond to the press event.

```
var finalScore : Label = get_node("FinalScore")
finalScore.text = "Final Score: " + str(Game.score)
```

When the retry button is pressed, we want to go back to our world scene. To do this, we will use the changeSceneToFile() function of the getTree() object. However, before we do this, we need to reset the health and the score of the player. This is important to ensure that the game starts afresh when the player retries after a game over.

```
Game.playerHP = Game.playerMaxHP
Game.score = 0
get_tree().change_scene_to_file("res://Scenes/world.tscn")
```

Transitioning to the Game Over Scene

Now, we need a way to transition to the game over scene when the player's health reaches zero. We will do this in the physics process of our player script. We will add a condition to check if the player's health is less than or equal to zero. If it is, we will change the scene to our game over scene.

```
if Game.playerHP <= 0:
    get_tree().change_scene_to_file("res://Scenes/GameOver.tscn")
```

With these steps, we now have a functional game over screen that displays the final score and allows the player to retry the game. The game over scene is displayed whenever the player's health reaches zero, and the game restarts with the player's health and score reset when the retry button is

pressed.

Conclusion

In this lesson, we learned how to create a game over screen in Godot. We also learned how to transition to this screen when the player dies and how to restart the game when the player retries. This provides a complete game loop that enhances the gaming experience. In the next lesson, we will look at how to increase the difficulty of the game as the player survives for longer periods.

In this lesson, we will learn how to increase the difficulty of a game over time in the Godot game engine. This can be achieved by increasing the spawn rate of enemies as the player survives longer in the game. The two key components we need to implement this are a timer and a script attached to our world scene.

Setting Up the Timer

First, we need to add a timer to our world. This timer will keep track of how long the player has survived in the game. We also need to connect a timeout to our world.

Creating the Script

Next, we need to create a script for our world scene. This script will be responsible for increasing the game difficulty over time. We'll add this script to our scripts folder.

In our script, we create a variable called "timeSurvived" and initialize it to zero. This variable will keep track of the player's survival time. Every second, we increase the value of "timeSurvived" by one, effectively counting the number of seconds the player has survived.

```
var timeSurvived = 0
#...
timeSurvived += 1
```

This allows us to make the game more challenging by spawning more obstacles or enemies based on the player's survival time.

Adjusting the Spawn Rate

We can adjust the spawn rate of enemies in our game by modifying the timer of our enemy spawner. In our mob portal script, we add a process function where we adjust the timer accordingly. The idea is to make the timer spawn enemies more frequently as the player survives longer in the game.

We can do this by checking the value of "timeSurvived". If it's between one and nine, we adjust the spawn rate based on the survival time. This is done by subtracting the survival time from 10 and dividing the result by two. The result is then assigned to the wait time of the timer.

```
if (getParent().timeSurvived <= 9) and (getParent().timeSurvived >= 1):
    timer.wait_time = 10 - roundi(getParent().timeSurvived / 2)
```

For example, if the player has survived for two seconds, the enemies will now spawn every nine seconds. If the player has survived for ten seconds, the enemies will spawn every five seconds. The game thus becomes more challenging as the player survives longer.

Testing the Game

To see how the increased difficulty affects gameplay, we can run our game and observe the spawn rate of enemies. We should see that the spawn rate increases as we survive longer in the game. This makes the game more challenging and engaging for the player, as they need to adapt to the increasing difficulty over time.

In conclusion, by using a timer and adjusting the spawn rate of enemies based on the player’s survival time, we can make our game progressively more challenging. This can enhance the gameplay experience and keep players engaged for longer periods of time.

Congratulations on completing the game development course! This course was designed to be fun and interactive, and we're thrilled that you've managed to create your own game. In this course, we covered several key concepts, including object pooling, basic AI movements, and collision layers.

Object Pooling

Object pooling is a software design pattern that uses a set of initialized objects kept ready to use, rather than allocating and deallocating them on the fly. In this course, we applied object pooling several times. We also provided a detailed explanation of how object pooling works and how it can be implemented in Godot, one of the game engines we worked with.

AI Movement

We also covered basic AI movement for our game monsters. This is a crucial aspect of game development as it brings the game characters to life, making them seem more realistic to the player.

Collision Layers

Understanding collision layers is essential in game development. They help prevent unwanted interactions between game objects and can significantly improve the game's performance. We discussed how to set up and use collision layers in Godot to avoid strange game behavior.

What's Next?

With the knowledge you've gained from this course, you're encouraged to continue building upon the game we created together. You can add more monsters, create a different player character, or even change some of the gameplay mechanics. The possibilities are endless!

About Zenva

Zenva is an online learning academy with over a million learners. We offer a wide range of courses for beginners and experienced learners alike. Our courses are flexible, allowing you to learn at your own pace and in the way that suits you best. Whether you prefer watching tutorial videos, reading lesson summaries, or following along with the instructor using the included project files, Zenva has got you covered.

Once again, congratulations on completing this course, and thank you for choosing Zenva. We hope to see you in another course soon!

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

AIBulletPool.gd

Found in the Project/Scripts/Bullets

This code creates a pool of preloaded bullet objects, initializes them as hidden, and allows the user to retrieve and reset bullet objects as needed.

```
extends Node

var bullet_scene : PackedScene = preload("res://Scenes/Bullets/Enemybullet.tscn")
var pool_size : int = 10
var bullet_pool : Array = []

func _ready() -> void:
  for i in range(pool_size):
    var bullet : Node = bullet_scene.instantiate()  # Adjust the type if your bullet
is a different type
    bullet.hide()  # Hide the bullet initially
    bullet_pool.append(bullet)
    add_child(bullet)

func get_bullet() -> Node:  # Adjust the return type if your bullet is a different ty
pe
  for bullet in bullet_pool:
    if not bullet.visible:
      return bullet

  # If no bullet is available, create a new one
  var new_bullet : Node = bullet_scene.instantiate()  # Adjust the type if your bulle
t is a different type
  new_bullet.hide()
  bullet_pool.append(new_bullet)
  add_child(new_bullet)
  return new_bullet

func reset_bullet(bullet: Node) -> void:  # Adjust the parameter type if your bullet
is a different type
  bullet.position = Vector2(1000, 10000)
  bullet.hide()
```

Bullet.gd

Found in the Project/Scripts/Bullets

This code defines a character body that extends the CharacterBody2D class. The object rotates slightly and moves using the move_and_slide() function in the _process() function. If the bullet enters

a 2D area and collides with a body that has a property “mob”, it checks if the body is alive and visible. If it is, it resets the mob and bullet in the game.

```
extends CharacterBody2D

var shooting : bool = false

func _process(delta: float) -> void:
  self.rotation += 0.1
  move_and_slide()

func _on_area_2d_body_entered(body: Node) -> void:  # Adjust 'Node' if 'body' is expe
cted to be a specific type
  if body.get("mob"):
    if body.isAlive and visible and body.visible:
      body.reset_Mob(body)
      get_parent().reset_bullet(self)
```

BulletPool.gd

Found in the Project/Scripts/Bullets

This code sets up a bullet pool for an enemy or player object. It creates a predefined number of bullet instances, hides them initially, and stores them in an array. It provides functions to retrieve and reset bullets from the pool as needed.

```
extends Node

var bullet_scene : PackedScene = preload("res://Scenes/Bullets/bullet.tscn")
var pool_size : int = 10
var bullet_pool : Array = []

func _ready() -> void:
  for i in range(pool_size):
    var bullet : Node = bullet_scene.instantiate()  # Adjust the type if your bullet
is a different type
    bullet.hide()  # Hide the bullet initially
    bullet_pool.append(bullet)
    add_child(bullet)

func get_bullet() -> Node:  # Adjust the return type if your bullet is a different ty
pe
  for bullet in bullet_pool:
    if not bullet.visible:
      return bullet

  # If no bullet is available, create a new one
  var new_bullet : Node = bullet_scene.instantiate()  # Adjust the type if your bulle
t is a different type
  new_bullet.hide()
  bullet_pool.append(new_bullet)
  add_child(new_bullet)
  return new_bullet
```

```
func reset_bullet(bullet: Node) -> void:  # Adjust the parameter type if your bullet
is a different type
  bullet.position = Vector2(1000, 10000)
  bullet.hide()
```

EnemyBullet.gd

Found in the Project/Scripts/Bullets

This code is a part of a character script that extends CharacterBody2D. It handles the character's movement using the move_and_slide() function and detects collisions with the player using the _on_area_2d_body_entered() function. If the bullet is visible and collides with the player, it decreases the player's health by 1 and calls a function to reset the character's position.

```
extends CharacterBody2D

var shooting : bool = false

func _process(delta: float) -> void:
  move_and_slide()

func _on_area_2d_body_entered(body: Node) -> void:  # Adjust 'Node' if 'body' is expe
cted to be a specific type
  if "Player" in body.name:
    if visible and body.visible:
      Game.playerHP -= 1
      get_parent().reset_bullet(self)
```

Game.gd

Found in the Project/Scripts/Global

This code extends the Node class and sets up variables for score, player maximum health points, and player current health points.

```
extends Node

var score : int = 0

var playerMaxHP : int = 10
var playerHP : int = 10
```

Player.gd

Found in the Project/Scripts/Player

This code extends the CharacterBody2D class and creates a character that can be controlled using arrow keys or WASD. The character can move in four directions and shoot bullets in the direction it is

facing. If the player's HP reaches 0, the game over scene is loaded.

```
extends CharacterBody2D

var speed : int = 75
@onready var bulletPool : Node = get_node("Marker2D/Bullets")  # Assuming bulletPool
is a Node type; adjust if different
var direction : Vector2 = Vector2(0, 1)

func _physics_process(delta: float) -> void:
  var inputDir : Vector2 = Vector2(
    Input.get_axis("MoveLeft", "MoveRight"),
    Input.get_axis("MoveUp", "MoveDown")
  ).normalized()

  if inputDir.x > 0:
    get_node("Player").frame = 1
    get_node("Player").flip_h = false
    direction = inputDir
  elif inputDir.x < 0:
    get_node("Player").frame = 1
    get_node("Player").flip_h = true
    direction = inputDir
  elif inputDir.y < 0:
    get_node("Player").frame = 2
    direction = inputDir
  elif inputDir.y > 0:
    get_node("Player").frame = 0
    direction = inputDir

  velocity = inputDir * speed
  get_node("Marker2D/SpawnPoint").position = direction

  if Input.is_action_just_pressed("Shoot"):
    var bulletTemp : Node = bulletPool.get_bullet()  # Assuming get_bullet() returns
a Node type; adjust if different
    bulletTemp.velocity = direction * 140
    bulletTemp.global_position = get_node("Marker2D/SpawnPoint").global_position
    bulletTemp.show()

  move_and_slide()

  if Game.playerHP <= 0:
    get_tree().change_scene_to_file("res://Scenes/GameOver.tscn")
```

AI.gd

Found in the Project/Scripts

This code is for a character object in a game. It handles the movement and behavior of the character, including tracking the player, shooting bullets, and managing health. The code also includes functions for resetting the character and handling collisions with the player.

```
extends CharacterBody2D

# onready variables
@onready var Anim : AnimatedSprite2D = get_node("Anim")
@onready var sprite : Sprite2D = get_node("CyclopsNew")
@onready var player : Node = get_node("../../Player")  # Adjust the type if 'player'
is a specific type
@onready var bar : ProgressBar = get_node("ProgressBar")
@onready var bulletPool : Node = get_node("Marker2D/Bullets") # Adjust the type if 'b
ulletPool' is a specific type
var mob : bool = true
var speed : float = 20.0
var isAlive : bool = true
var health : int = 5

func _ready() -> void:
  get_node("CollisionShape2D").disabled = true
  bar.max_value = health

func _physics_process(delta: float) -> void:
  if isAlive:
    bar.value = health
    get_node("CollisionShape2D").disabled = false
    var direction : Vector2 = (player.global_position - self.global_position).normali
zed()
    self.velocity = speed * direction
    if direction.x < 0:
      sprite.flip_h = true
    else:
      sprite.flip_h = false

    Anim.hide()
    sprite.show()
    move_and_slide()
  else:
    bar.hide()
    Anim.show()
    get_node("CollisionShape2D").disabled = true
    sprite.hide()

func reset_Mob(body: Node) -> void:  # Adjust the type if 'body' is a specific type
  if health > 1:
    health -= 1
  else:
    isAlive = false
    Anim.play("Death")
    Game.score += 1
    await Anim.animation_finished
    get_parent().reset_mob(body)

func _on_player_detection_body_entered(body: Node) -> void:  # Adjust the type if 'bo
dy' is a specific type
  if "Player" in body.name:
    if visible and body.visible:
      Game.playerHP -= 1
```

```
func shoot_bullet() -> void:
  if self.visible:
    var bulletTemp : Node = bulletPool.get_bullet()  # Adjust the type if 'bulletTemp
' is a specific type
    var direction : Vector2 = (player.global_position - get_node("Marker2D/SpawnPoint
").global_position).normalized()
    bulletTemp.velocity = direction * 140
    bulletTemp.global_position = get_node("Marker2D/SpawnPoint").global_position
    bulletTemp.show()

func _on_shoot_bullet_timeout() -> void:
  shoot_bullet()
```

GameOver.gd

Found in the Project/Scripts

This code extends the Node2D class and contains two functions. In the _ready function, it sets the text of the finalScore label node to the current score of the game. In the _on_retry_pressed function, it resets the player's HP and score, and changes the scene to "world.tscn" when the retry button is pressed.

```
extends Node2D

@onready var finalScore : Label = get_node("FinalScore")  # Assuming finalScore is a
Label node

func _ready() -> void:
  finalScore.text = "Final Score: " + str(Game.score)

func _on_retry_pressed() -> void:
  Game.playerHP = Game.playerMaxHP
  Game.score = 0
  get_tree().change_scene_to_file("res://Scenes/world.tscn")
```

Mob Pooling.gd

Found in the Project/Scripts

This code sets up a pool of mobs (enemies) and controls their spawning. It instantiates a specified number of mobs and adds them to the scene, hides them initially, and keeps track of them in an array. It also controls the timing of mob spawns based on the game's "timeSurvived" variable. When a mob needs to be spawned, it checks for an available mob from the pool and positions it randomly near the player's position before showing it.

```
extends Node2D

var mob_scene : PackedScene = preload("res://Scenes/cyclops_mob.tscn")
var pool_size : int = 10
var mob_pool : Array = []
```



```
@onready var timer : Timer = get_node("Timer")

func _ready() -> void:
  for i in range(pool_size):
    var mob : Node2D = mob_scene.instantiate() as Node2D  # Adjust the type if your m
ob is a different type
    mob.hide()  # Hide the mob initially
    mob_pool.append(mob)
    add_child(mob)

func _process(delta: float) -> void:
  if (get_parent().timeSurvived <= 9) and (get_parent().timeSurvived >= 1):
    timer.wait_time = 10 - roundi(get_parent().timeSurvived / 2)

func get_mob() -> Node2D:  # Adjust the return type if your mob is a different type
  for mob in mob_pool:
    if not mob.visible:
      return mob
  # If no mob is available, create a new one
  var new_mob : Node2D = mob_scene.instantiate() as Node2D  # Adjust the type if your
 mob is a different type
  new_mob.hide()
  mob_pool.append(new_mob)
  add_child(new_mob)
  return new_mob

func reset_mob(mob: Node2D) -> void:  # Adjust the parameter type if your mob is a di
fferent type
  mob.global_position = Vector2(-1000, -1000)
  mob.get_node("CollisionShape2D").disabled = false
  mob.isAlive = true
  mob.hide()

func _on_timer_timeout() -> void:
  # Spawn new mob
  var mobTemp : Node2D = get_mob() as Node2D  # Adjust the type if your mob is a diff
erent type
  var randX : int = randi_range(-50, 50)
  var randY : int = randi_range(-50, 50)
  mobTemp.global_position = self.global_position + Vector2(randX, randY)
  mobTemp.show()
```

world.gd

Found in the Project/Scripts

This code extends the Node2D class and is used to create a new custom class that can be added as a child node in Godot game engine.

```
extends Node2D
```